JavaGI
A Language with Generalized Interfaces

Stefan Wehr

2010

Dissertation zur Erlangung des Doktorgrades der Technischen Fakultät der
Albert-Ludwigs-Universität Freiburg im Breisgau

Dekan der Technischen Fakultät: Prof. Dr. Hans Zappe

1. Gutachter: Prof. Dr. Peter Thiemann, Albert-Ludwigs-Universität Freiburg
2. Gutachter: Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau

Tag der Disputation: 9. Juli 2010

Abstract

Component-based software development in statically typed, object-oriented programming languages has proven successful in reducing development costs and raising software quality. However, this form of software development still poses many challenges and thus requires better support on the programming language level.

The language JavaGI, a conservative extension of Java 1.5, offers *generalized interfaces* as an effective improvement. Generalized interfaces subsume retroactive and type-conditional interface implementations, binary methods, symmetric multiple dispatch, interfaces over families of types, and static interface methods. These features allow non-invasive and in-place object adaptation, thus enabling solutions to several software extension, adaptation, and integration problems with components in binary form. Further, they make certain coding patterns redundant and increase the expressiveness of the type system. The generalized interface mechanism offers a unifying conceptual view on these seemingly disparate concerns, for which previously unrelated extensions have been suggested.

This dissertation introduces the language JavaGI by explaining its features and motivating its design. Technical contributions of the dissertation are the formalization of a core calculus for JavaGI and a proof of type soundness, determinacy of evaluation, and decidability of subtyping and typechecking. The formalization also includes a type- and behavior-preserving translation from a significant subset of the core calculus to a slightly extended version of Featherweight Java. Moreover, the dissertation explores two extensions of the type system, which both have undecidable subtyping relations but for which several decidable fragments exist. The undecidability result for one of the extensions sheds light on the decidability of subtyping in Scala and of subtyping with Java wildcards.

On the practical side, the dissertation presents the implementation of a JavaGI compiler and an accompanying run-time system. The compiler is based on an industrial-strength Java compiler and offers mostly modular typechecking but fully modular code generation. It defers certain well-formedness checks until load time to allow for greater flexibility and to enable full support for dynamic loading. Benchmarks show that the code generated by the compiler offers good performance. Several case studies demonstrate the practical utility of the language and its implementation. The implementation also includes a JavaGI plugin for the Eclipse IDE.

Zusammenfassung

Komponentenbasierte Softwareentwicklung in objektorientierten, statisch getypten Programmiersprachen hat sich als erfolgreich erwiesen, um Entwicklungskosten zu senken und die Qualität von Software zu erhöhen. Dennoch ergeben sich bei dieser Art der Softwareentwicklung noch immer viele Herausforderung, so dass eine bessere Unterstützung auf der Programmiersprachenebene gewünscht ist.

Die Sprache JavaGI, eine konservative Erweiterung von Java 1.5, bietet generalisierte Interfaces als effektive Verbesserung an. Generalisierte Interfaces umfassen retroaktive und typbedingte Interface–Implementierungen, binäre Methoden, symmetrischen Mehrfachdispatch, Interfaces über Typfamilien und statische Interface–Methoden. Diese Eigenschaften erlauben nichtinvasive und direkte Objektanpassung und ermöglichen damit Problemlösungen im Bereich der Erweiterung, Anpassung und Integration von Software mit Komponenten in binärer Form. Außerdem subsummieren die Eigenschaften verschiedene Programmiermuster und erhöhen die Ausdrucksfähigkeit des Typsystems. Der Generalisierungsmechanismus für Interfaces bietet einen einheitlichen Rahmen für diese scheinbar ungleichen Belange, welche in der Vergangenheit mit verschiedenen, nicht miteinander in Beziehung stehenden Erweiterungen angegangen wurden.

Die vorliegende Dissertation präsentiert die Sprache JavaGI, erklärt ihre Eigenschaften und motiviert das Sprachdesign. Technische Beiträge der Arbeit sind ein Kernkalkül für JavaGI und ein Beweis der Typkorrektheit, Eindeutigkeit der Auswertung und Entscheidbarkeit der Subtyprelation sowie der Typüberprüfung. Die Formalisierung beinhaltet auch eine typ- und verhaltenserhaltende Übersetzung einer signifikanten Teilmenge des Kalküls in eine leicht erweiterte Fassung von Featherweight Java. Desweiteren werden zwei Erweiterungen des Typsystems untersucht. Die Subtyprelation ist für beide Erweiterungen unentscheidbar, allerdings existieren mehrere entscheidbare Fragmente. Das Unentscheidbarkeitsresultat für eine der Erweiterungen wirft neues Licht auf die Frage der Entscheidbarkeit der Subtyprelationen von Scala und von Java mit Wildcards.

Auf der praktischen Seite präsentiert die Dissertation die Implementierung eines Compilers und eines entsprechenden Laufzeitsystems für JavaGI. Der Compiler basiert auf einem industriell eingesetzten Java Compiler und unterstützt eine größtenteils modulare Typüberprüfung sowie vollständig modulare Codeerzeugung. Bestimmte Wohlgeformtheitsüberprüfungen werden bis zur Linkzeit aufgeschoben, um größere Flexibilität und volle Unterstützung für dynamisches Laden bieten zu können. Benchmarks zeigen, dass der Compiler Code mit guter Performanz erzeugt. Mehrere Fallstudien demonstrieren die praktische Anwendbarkeit der Sprache und ihrer Implementierung. Die Implementierung beinhaltet auch ein JavaGI Plugin für die Entwicklungsumgebung Eclipse.

Acknowledgments

Peter Thiemann sparked my interest in programming languages and their underlying theory. I have learned a lot from him and many of the contributions in this dissertation benefited greatly from numerous discussions with him. Not only did he give me the freedom to work on my own ideas but he also provided careful guidance to bring these ideas into a polished and qualified form. Thank you, Peter!

Ralf Lämmel contributed valuable ideas to the initial design of JavaGI and raised questions that I addressed in later versions of the language. I would like to thank him for fruitful discussions and for co-reviewing my dissertation.

Matthias Neubauer, Markus Degen, Phillip Heidegger, Annette Bieniusa, and Konrad Anton (in order of their appearance) were great colleagues during my time at the University of Freiburg. Matthias, Phillip, and Annette provided useful feedback on previous versions of this dissertation, and Konrad's diploma thesis explored the design of Waitomo, a predecessor of JavaGI. I am also grateful to Alina Swiderska for developing the Eclipse plugin for JavaGI. David Leuschner gave helpful feedback on an earlier draft of this dissertation and confronted me with reality by asking the "what is this good for in practice" question.

Last not least, I would like to thank all my friends and my whole family for their support and for showing me that computer science is not the most important thing in life.

Stefan Wehr
December, 2009

Contents

List of Figures

1 Introduction

Developing and maintaining large and complex software systems is expensive, both in terms of time and money [56, 158]. Furthermore, software defects are not only the source of frequent annoyance but may also inflict serious damage [123, 160, 106]. Thus, it is highly beneficial to devise methods and techniques for controlling the inherent complexity of software, for reducing the number of defects in software, and for lowering the costs of developing and maintaining software.

In fact, industry and academia proposed numerous such methods and techniques. The proposals include (but are not limited to) processes and methodologies for organizing the development cycle of software [191, 231, 19, 11], various approaches to testing software [155, 17], formal verification of software [88, 190, 28], and new programming paradigms and languages [92, 142, 222].

A proposal by McIlroy [159], brought up at the famous 1968 NATO conference on software engineering, envisions the idea of *software components* [220]. The concept behind software components is simple: developers should not write applications from scratch but assemble them from pre-packaged, largely independent components. This approach reduces the complexity of software systems because each component can be analyzed, programmed, and tested in isolation. Moreover, it saves development costs if the same component is reused in different projects [9]. Last not least, components can increase software quality because defect fixes for components accumulate through reuse, and the fixes must be applied in only one place [124].

Static type systems [176], having their roots in mathematical logic of the early 1900s, are a lightweight formal method to reject certain potentially erroneous programs statically; that is, at compile time and not when the program is run. Thus, static type systems prevent a whole class of software defects right from the start. Further, type declarations may serve as lightweight documentation, which makes software easier to understand and maintain. Static type systems blend well with the idea of software components because types enable an abstract description of the functionality offered and required by a component.

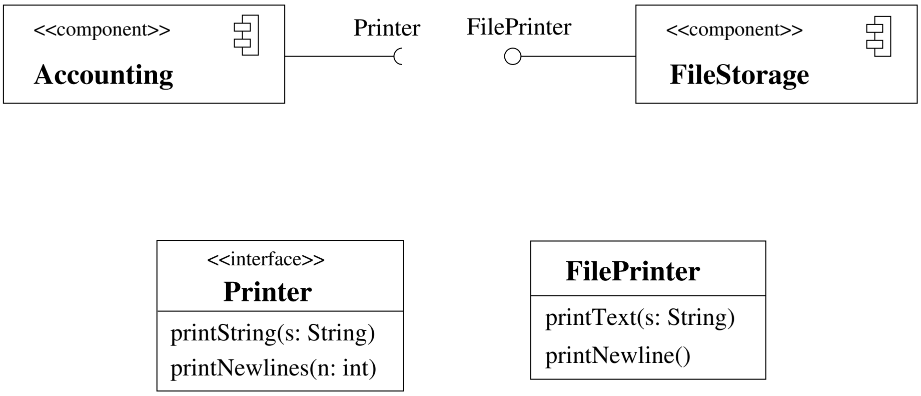
Nowadays, the *object-oriented programming* paradigm is a popular choice for software development in industry and academia. Introduced with the programming language Simula 67 [54], this paradigm features code reuse through inheritance and information hiding through encapsulation [142, 5]. As already explained, code reuse may lead to reduced development costs and enhanced software quality. The same holds for the principle of information hiding because it enables developers to write one part of a software system with only little knowledge about the internals of the other parts, and it allows changing the implementation of one part without affecting the rest of the system. Furthermore, information hiding is important for component-based systems to minimize dependencies between components.

Object-oriented programming languages with static type systems often serve as implementation languages for software components and component-based systems. In fact, according to Szyperski, "object technology, if harnessed carefully, is probably one of the best ways to realize component technology" [220, page 15]. Industry seems to agree with this statement, as demonstrated through several component standards such as the Common Object Request Broker Architecture (CORBA [164]), Sun's Java Beans [216] and Enterprise Java Beans (EJB [213]) technologies, and Microsoft's Component Object Model (COM [145]). Moreover, one factor of success of languages such as Java [82] or C# [64], two prominent object-oriented languages with static type systems, is the large number of libraries available for these languages. Libraries may also be regarded as components [99, 195].

Despite these success stories, there are still many unsolved problems in the realm of component-based software development. For instance, components written in Java or C# typically abstract over their required services by means of *interfaces*. (Interfaces are a built-in language mechanism that specifies a set of method signatures without committing to a particular implementation.) Other components fulfill these requirements by providing implementations of the corresponding interfaces. However, this approach has several disadvantages. First, it leads to difficulties in fulfilling the requirements of a component $\mathcal{C}_1$ by an independently developed component $\mathcal{C}_2$ because $\mathcal{C}_2$ is typically not aware of the exact interfaces required by $\mathcal{C}_1$. Second, the approach creates hardwired dependencies between components, thus impeding further reuse because components fulfilling certain dependencies cannot be replaced by other components easily.

Figure 1.1 depicts an example. The component `Accounting` requires a printer service, which has to implement the interface `Printer`. The independently developed component `FileStorage` offers such a service by the class `FilePrinter`. Although the methods of `Printer` and `FilePrinter` are slightly incompatible, it is straightforward to implement the `Printer` interface by using the methods of class `FilePrinter`. However, `FilePrinter` does not implement `Printer` formally. Now suppose a developer wants to use the `FileStorage` component to satisfy the printer service required by the `Accounting` component. Unfortunately, neither Java nor C# offer the possibility to implement the `Printer` interface *retroactively* for class `FilePrinter`. Thus, assuming that the source code of the two components is not accessible (the default with component-based software), the developer must circumvent the problem by using the Adapter pattern [73] to make `FilePrinter` compatible with `Printer`. That is, the developer needs to create an adapter class `PrinterAdapter` that implements `Printer` by delegating method calls to an instance of `FilePrinter`. Further, the developer has to insert extra code at

Figure 1.1 Incompatibility between two components.
The diagram uses UML 2 [165] syntax. Provided services are represented by circles, required services by half-circles.



the right places to convert between `FilePrinter` and `PrinterAdapter` objects. Obviously, this pattern is tedious to implement. Moreover, it often behaves fragile in practice [89, 198, 93].

This example and numerous proposals in the research literature [86, 115, 235, 143, 227, 168, 50, 237] substantiate the claim that the features of standard object-oriented languages such as Java or C# do not suffice for solving various extension, adaptation, and integration problems in the context of component-based software. (See Chapter 8 for a detailed discussion on related work.) Furthermore, there are many situations in which a Java or C# developer reaches the limits of the type system and has to resort to tedious coding patterns, unsafe cast operations, run-time exceptions, or code duplication, all of which may easily lead to an increase in development time and potentially more software defects. This introductory chapter refrains from discussing these examples in more detail; for further information see Chapter 2. Instead, it continues by establishing the goals and summarizing the contributions of this dissertation.

1.1 Goals and Contributions

Lämmel and Ostermann [119] demonstrated that type classes [107, 236, 104, 85], a structuring mechanism related to object-oriented–style interfaces but introduced by the functional programming language Haskell [173], provide clean solutions to a number of software extension, adaptation and integration problems. Their findings raise the question whether object-oriented–style interfaces could give rise to similar solutions if extended and generalized in the direction of type classes. A related question is whether such an extension could raise the expressiveness of the type system to prevent the programming problems described earlier. After all, many examples demonstrate that Haskell's type system provides powerful abstractions and strong static guarantees through type classes [117, 103, 174, 118].

The main goal of this dissertation is to answer these questions by designing, formalizing, and implementing the programming language JavaGI. This new language conservatively extends Java[1] with *generalized interfaces*, a mechanism extending and generalizing object-oriented–style interfaces with features from Haskell type classes. The generalization of interfaces is the unifying notion of JavaGI's design: it subsumes different concerns under a single concept. More specifically, JavaGI generalizes Java's interfaces in the following dimensions:

Retroactive Interface Implementations. The implementation of an interface may be retroactive; that is, separate from the definition of the interface and of the implementing class.

Explicit Implementing Types. An interface may explicitly reference its implementing type, thus allowing the specification of binary methods.

Multi-Headed Interfaces. An interface may be multi-headed; that is, it may span multiple types to specify mutual dependencies.

Symmetric Multiple Dispatch. Interface methods depending on implementing types in argument positions (binary methods and certain methods in multi-headed interfaces) are subject to symmetric multiple dispatch.

Implementation Constraints. An interface may not only be used as a type but also in a constraint to restrict a type or a family of types.

Type Conditionals. Methods and retroactive interface implementations may depend on type constraints, thus enabling type-conditional methods and interface implementations.

Static Interface Methods. An interface may contain static methods.

These features make certain coding patterns redundant and increase the expressiveness of the language to avoid unsafe cast operations, run-time exceptions, and code duplication. Moreover, the features allow solutions to extension, adaptation, and integration problems with components in binary form for which unrelated extensions had been suggested before. Compared with other work, retroactive interface implementations allow non-invasive and in-place object adaption [237], supersede the Adapter and Visitor patterns [73], and enable a solution to (a restricted version of) the expression problem [235, 227]; explicit implementing types are related to work on `MyType` and `ThisType` [32, 30] and supersede certain instances of F-bounded polymorphism [39]; multi-headed interfaces provide a restricted form of family polymorphism [68]; symmetric multiple dispatch supersedes the double dispatch pattern [98]; implementation constraints avoid certain cast operations; type conditionals avoid code duplication or run-time errors [91]; and static interface methods supersede uses of the Factory pattern [73].

[1] Throughout this dissertation, the term "Java" always refers to version 1.5 of the Java programming language [82].

JavaGI is unique in that it avoids a patchwork of unrelated features but offers a unifying conceptual view on these seemingly disparate concerns. We believe that the resulting design is coherent, elegant, and does not impose undue burden on the programmer.

Contributions

This dissertation makes the following contributions:

- It introduces the features of JavaGI and highlights the underlying design principles.
- It formalizes a core calculus of JavaGI in the style of Featherweight Generic Java [96] and proves type soundness, determinacy of evaluation, and decidability of subtyping and typechecking.
- It defines a translation from a significant subset of the core calculus to a slightly extended version of Featherweight Java [96] and proves that the translation preserves the static and the dynamic semantics of the source language.
- It explores two extensions of JavaGI's type system, proves that both extensions render subtyping undecidable, and identifies decidable fragments of the extensions. The undecidability result for one of the extensions also sheds light on the decidability of subtyping in Scala [166] and of subtyping for Java wildcards [229, 37].
- It reports on an implementation of a compiler for JavaGI and an accompanying run-time system. The implementation is based on the Eclipse Compiler for Java [62] and supports mostly modular typechecking, fully modular compilation, and dynamic loading of retroactive interface implementations. Besides the compiler and the run-time system, the implementation also provides a plugin for the Eclipse [60] IDE to facilitate the development of JavaGI applications.
- It summarizes the outcome of a number of case studies and describes the results of several performance benchmarks to demonstrate the practical utility of JavaGI and its implementation.
- It puts JavaGI in perspective by providing a comprehensive survey and discussion of related work.

The homepage of the JavaGI project [239] makes the source code of the compiler, the run-time system, the Eclipse plugin, the case studies, and the benchmarks available under the terms of the Eclipse Public License [61].

1.2 Road Map

The dissertation is organized as follows:

A Tour of JavaGI. Chapter 2 introduces the features of JavaGI through a series of examples, which also demonstrate how JavaGI solves the aforementioned programming problems. The chapter further explains the design principles of JavaGI and informally investigates the JavaGI-specific extensions of Java's type system and execution model.

Formalization of CoreGI. Chapter 3 formalizes CoreGI, a core calculus of JavaGI in the spirit of Featherweight Generic Java. The chapter proves that CoreGI's type system is sound and that its evaluation relation is deterministic. Further, it presents a typechecking algorithm for CoreGI and proves that the algorithm is equivalent to the original type system.

Translation. Chapter 4 specifies a translation from a language with generalized interfaces into a language without. The chapter first introduces the source language CoreGI$^\flat$, a simplified version of CoreGI. Then it defines the target language iFJ as an extension of Featherweight Java. Next, it presents a type-directed translation from CoreGI$^\flat$ to iFJ and proves that the translation preserves the static and the dynamic semantics of CoreGI$^\flat$. Finally, the chapter verifies that CoreGI$^\flat$ is a subset of CoreGI.

Extensions. Chapter 5 tests the boundaries of the design space for JavaGI by defining two extensions of JavaGI's type system and proving that the subtyping relations of both extensions are undecidable. The chapter also presents several decidable fragments of the extensions.

Implementation. Chapter 6 describes the implementation of a compiler and an accompanying run-time system for JavaGI. The chapter also explains how to extend the formalization given in Chapter 3, the translation defined in Chapter 4, and a decidable fragment of one of the extensions from Chapter 5 to the full JavaGI language.

Practical Experience. Chapter 7 reports on practical experience with JavaGI. It presents three case studies conducted with the JavaGI implementation and evaluates the performance of the implementation through various benchmarks.

Related Work. Chapter 8 reviews a broad range of research related to JavaGI.

Conclusion. Chapter 9 summarizes the dissertation and outlines possible directions for future work.

Part A of the appendix defines the syntax of JavaGI, expressed as an extension to the syntax of Java as defined in the first 17 chapters of *The Java Language Specification* [82]. Parts B, C, and D of the appendix contain the formal details of Chapters 3, 4, and 5, respectively, including the proofs of all theorems postulated in these chapters. The dissertation ends with a bibliography and an index of important terms, symbols, and notations.

Some of the material presented in the next chapters is based on previous publications by the author of this dissertation and others:

- A paper in the proceedings of ECOOP 2007 (joint work with Ralf Lämmel and Peter Thiemann [240]) proposed the initial design of JavaGI. (Section 8.11 contains a more detailed comparison with the ECOOP paper.)

- A paper in the proceedings of GPCE 2009 (joint work with Peter Thiemann [242]) reported on JavaGI's implementation and on practical experience through benchmarks and case studies (see Chapter 7).

- A paper in the proceedings of APLAS 2009 (joint work with Peter Thiemann [243]) established the undecidability results for two extensions of JavaGI's type system (see Chapter 5). An earlier version of the APLAS paper was presented at the FTfJP 2008 workshop [241].

2

A Tour of JavaGI

JavaGI is a new programming language that conservatively extends Java with generalized interfaces. This chapter provides a gentle introduction to JavaGI.

Chapter Outline. The chapter contains three sections.

- Section 2.1 presents and motivates the features of JavaGI through a series of examples, which also demonstrate how JavaGI solves the programming problems put forward in Chapter 1. The section closes by comparing the solutions in JavaGI with corresponding solutions in plain Java.
- Section 2.2 takes a step back and explains the design principles behind JavaGI.
- Section 2.3 informally investigates the JavaGI-specific extensions of Java's type system and execution model.

2.1 Features

The examples used to introduce the features of JavaGI are all based on the simple expression hierarchy shown in Figure 2.1. We assume that it is not possible to modify the source code of the expression hierarchy. As JavaGI is an extension of Java, JavaGI code (and Java code where appropriate) refers to common classes and interfaces from the Java API [212].[1]

2.1.1 Retroactive Interface Implementations

The expression hierarchy in Figure 2.1 supports only evaluation of expressions. Now suppose that we also want to produce nicely formatted string output from expression instances. To implement this functionality, we would like to use a library such as *The*

[1] The code uses classes and interfaces from the packages `java.lang`, `java.util`, and `java.io` without further qualification.

Figure 2.1 Expression hierarchy.

```
abstract class Expr {
  abstract int eval();
}
class IntLit extends Expr {
  int value;
  IntLit(int value) {
    this.value = value;
  }
  int eval() {
    return this.value;
  }
}
class PlusExpr extends Expr {
  Expr left;
  Expr right;
  PlusExpr(Expr left, Expr right) {
    this.left = left;
    this.right = right;
  }
  int eval() {
    return this.left.eval() + this.right.eval();
  }
}
```

Java Pretty Printer Library [78]. This library provides an interface that classes with pretty-printing support must implement:[2]

```
interface PrettyPrintable {
  String prettyPrint();
}
```

A Java programmer cannot add an implementation for the `PrettyPrintable` interface to the classes of the expression hierarchy because we assumed earlier that the source code of these classes is unmodifiable. Instead, a Java programmer would presumably use the Adapter pattern [73] and create a parallel hierarchy of expression adapters complying to the `PrettyPrintable` interface (see Section 2.1.8).

In JavaGI, we do not need the Adapter pattern because JavaGI supports *retroactive interface implementations* where the implementation of an interface may be separate from the implementing class. Here are three *implementation definitions* for the `PrettyPrintable` interface with the classes `Expr`, `IntLit`, and `PlusExpr` acting as the *implementing types* (enclosed in square brackets ‘`[...]`’):

```
implementation PrettyPrintable [Expr] {
  abstract String prettyPrint();
}
```

[2] We slightly modified the interface for the purpose of presentation.

```
implementation PrettyPrintable [IntLit] {
  String prettyPrint() { return String.valueOf(this.value); }
}
implementation PrettyPrintable [PlusExpr] {
  String prettyPrint() {
    return "(" + this.left.prettyPrint() + " + "
               + this.right.prettyPrint() + ")";
  }
}
```

The `prettyPrint` method for the abstract base class `Expr` remains abstract because there is no sensible default implementation. JavaGI guarantees that the implementation of `prettyPrint` is nevertheless *complete:* there exists a non-abstract definition of `prettyPrint` for each concrete subclass of `Expr`.

In the body of the two other `prettyPrint` methods, the static type of **this** is the implementing type of the surrounding implementation definition. That is, in the implementation for `IntLit`, **this** has static type `IntLit`, so the field access **this**`.value` is type correct. Similarly, in the implementation for `PlusExpr`, **this** has type `PlusExpr`, so the fields accesses **this**`.left` and **this**`.right` are valid. We can invoke `prettyPrint` recursively on these fields because there is an implementation of `PrettyPrintable` for `Expr`.

Methods of retroactive interface implementations are subject to dynamic dispatch, just as ordinary interface and class methods.[3] For instance, the recursive invocation **this**`.left.prettyPrint()` in the implementation for `PlusExpr` selects the method to execute based on the dynamic type of the receiver **this**`.left`. Hence, the call

```
new PlusExpr(new IntLit(1), new IntLit(2)).prettyPrint()
```

correctly returns `"(1 + 2)"`.

The implementations of `PrettyPrintable` for `Expr`, `IntLit`, and `PlusExpr` not only add the `prettyPrint` method to these classes but also make them compatible with the interface type `PrettyPrintable`. For example, we may pass an object of type `PlusExpr` to a method expecting an object of type `PrettyPrintable`:

```
class SomePrinter {
  void print(PrettyPrintable pp) {
    String s = pp.prettyPrint();
    System.out.println(s);
  }
  void usePrint() {
    PlusExpr expr = new PlusExpr(new IntLit(1), new IntLit(2));
    // use a "PlusExpr" instance at type "PrettyPrintable"
    print(expr);
  }
}
```

Retroactive implementation definitions can be placed in arbitrary compilation units. For example, it is possible to place the three implementations shown earlier in three

[3] In contrast, extension methods in C# 3.0 [64] are subject to static dispatch.

different compilation units, all of which may be different from the compilation units of the expression hierarchy and the PrettyPrintable interface.

This flexibility together with dynamic dispatch on retroactively implemented methods implies extensibility in the operation dimension and thus eliminates the need for the Visitor pattern [73]: to add a new operation, simply define an interface for the operation and provide suitable implementation definitions. Extensibility in the data dimension is also straightforward: add a new subclass of Expr and provide interface implementations for existing operations, unless the default for the base class suffices. Hence, JavaGI allows for a simple and elegant solution to (a restricted version of) the expression problem [235, 227] (see Section 8.4).

JavaGI does not require explicit import statements for retroactive implementation definitions. Instead, all retroactive implementations presented to the JavaGI compiler are automatically in scope. Imposing stricter visibility rules at compile time is not necessary because JavaGI's run-time system puts all implementation definitions into a global pool anyway (see Section 6.3).

2.1.2 Explicit Implementing Types

A *binary method* [29] is a method requiring the receiver type and some of the argument types to coincide. According to Bracha [24], the definition of a binary method in Java requires F-bounded polymorphism [39] and possibly wildcards [229] (see also Section 2.1.8). In contrast, JavaGI directly supports binary methods in interfaces through *explicit implementing types*. The following interface defines an equality operation that allows only objects with compatible types to be compared for equality.

```
interface EQ {
  boolean eq(This that);
}
```

The argument type of eq is the type variable **This**, which is implicitly bound by the interface and which denotes the type implementing the interface. Hence, eq qualifies as a binary method. The next example uses eq to define a generic function that searches for a specific element in a list.

```
class Lists {
  static <X implements EQ> X find(X x, List<X> list) {
    for (X y : list) {
      if (x.eq(y)) return y;
    }
    return null;
  }
}
```

We specify that X has to implement the EQ interface through the *implementation constraint* X **implements** EQ. This requirement on X is stronger than a regular Java bound X **extends** EQ because binary methods such as eq are only applicable to values of type X if the constraint X **implements** EQ holds (see Section 2.3.1).

When typechecking an implementation of EQ, the JavaGI compiler replaces the type variable **This** with the concrete implementing type. Here are EQ implementations for the classes of the expression hierarchy from Figure 2.1:

```
implementation EQ [Expr] {
  boolean eq(Expr that) {
    return false;
  }
}
implementation EQ [IntLit] {
  boolean eq(IntLit that) {
    return this.value == that.value;
  }
}
implementation EQ [PlusExpr] {
  boolean eq(PlusExpr that) {
    return this.left.eq(that.left) && this.right.eq(that.right);
  }
}
```

Given variables `le`, `e`, `li`, and `i` with static types `List<Expr>`, `Expr`, `List<IntLit>`, and `IntLit`, respectively, the following invocations of `Lists.find` now typecheck successfully:

```
    Lists.find(e, le);
    Lists.find(i, le);
    Lists.find(i, li);
```

The run-time behavior of methods mentioning explicit implementing types in their signatures is similar to that of multimethods [43]: JavaGI selects the most specific implementation dynamically, thereby extending dynamic dispatch to all parameters declared as implementing types (symmetric multiple dispatch, discussed in Section 8.5). Hence, invocations of `eq` dispatch on both the receiver and the first argument of the call.

Let us explain this behavior by considering the following variable declarations:

```
    Expr plus1  = new PlusExpr(new IntLit(1), new IntLit(2));
    Expr plus2  = new PlusExpr(new IntLit(1), new IntLit(2));
    Expr intLit = new IntLit(42);
```

All three variables have static type `Expr`. Nevertheless, the call `plus1.eq(plus2)` invokes the `eq` method of the implementation for `PlusExpr` because both the receiver `plus1` and the argument `plus2` have dynamic type `PlusExpr`. On the other hand, the call `plus1.eq(intLit)` invokes the `eq` method as implemented for the base class `Expr` because dynamic dispatch on the argument `intLit` rules out `eq` for `PlusExpr` and dynamic dispatch on the receiver `plus1` rules out `eq` for `IntLit`.

2.1.3 Type Conditionals

If the elements of two lists are comparable, then the lists should be comparable, too. JavaGI can express this implication with a *type-conditional interface implementation* [91, 66, 111, 131].

```
implementation<X> EQ [List<X>] where X implements EQ {
  boolean eq(List<X> that) {
    Iterator<X> thisIt = this.iterator();
    Iterator<X> thatIt = that.iterator();
    while (thisIt.hasNext() && thatIt.hasNext()) {
      X thisX = thisIt.next();
      X thatX = thatIt.next();
      if (!thisX.eq(thatX)) return false;
    }
    return !(thisIt.hasNext() || thatIt.hasNext());
  }
}
```

The implementation of EQ for List<X> is parameterized over X, the type of list elements. The constraint X **implements** EQ makes the eq operation available on objects of type X and ensures that only lists with comparable elements implement EQ. For example, if l1 and l2 have type List<Expr> and l3 has type List<List<Expr>>, then both calls l1.eq(l2) and Lists.find(l1, l3) are valid.

The notation **where** ..., reminiscent of .NET generics [112, 245], is not only available for constraints on interface implementations, but also for constraints on ordinary classes and interfaces. It may even be used to constrain type parameters of a class or interface on the basis of individual methods, as the next example shows.

```
class Box<X> {
  X x;
  boolean containedBy(List<X> list) where X implements EQ {
    return Lists.find(this.x, list) != null;
  }
}
```

The class Box itself places no constraint on its type parameter X. Thus, it may be instantiated with arbitrary types. However, method containedBy is only available if the actual type argument implements EQ; in other words, containedBy is a *type-conditional method*. For instance, an invocation of containedBy on a value of type Box<Expr> is valid, whereas an invocation on a value of type Box<String> is rejected by the compiler (unless we add an implementation of EQ for String).

2.1.4 Static Interface Methods

We not only want to evaluate and print expressions, but we also want to parse them from a string representation. Obviously, there are other situations (e.g., XML deserialization, parsing of XPath expressions, etc.) where we need to create an object from an external string representation. Ideally, we would like to abstract over these different situations.

As an example, consider a generic line processor: a method that loops over the lines of a given input stream, parses them, and then passes the result to some consumer. To reuse the code of looping over the input stream, we need to abstract over the parser and the consumer. Abstracting over the consumer is easily done using a plain Java interface:

```
// Consumes values of type X
interface Consumer<X> {
  void consume(X x);
}
```

However, a similar solution does not work for parsing because a parser acts like an additional class constructor: it creates an object from a string representation, so the `parse` method cannot be an instance method of the object being parsed. In this situation, Java programmers routinely use the Factory pattern [73] (see Section 2.1.8). In JavaGI, however, programmers may abstract over "constructor-like" methods through static interface methods:

```
// Parses a string and returns a value of the implementing type
interface Parseable {
  static This parse(String s);
}
```

(Again, the result type **This** refers to the implementing type.) Now it is easy to implement the line processor:

```
class LineProcessor {
  static <X> void process(InputStream in, Consumer<X> c)
             throws IOException where X implements Parseable {
    BufferedReader br = new BufferedReader(new InputStreamReader(in));
    String line;
    while ((line = br.readLine()) != null) {
      X x = Parseable[X].parse(line);  // parse the line ...
      c.consume(x);                    // ... and consume it
    }
  }
}
```

The expression `Parseable[X].parse(s)` invokes the `parse` method of `Parseable` with `X` as the implementing type. The invocation is well-typed because we require the constraint `X` **`implements`** `Parseable` (see Section 2.3.1). It returns an object of type `X` which we pass to the `consume` method.

Given an implementation of `Parseable` for `Expr`

```
implementation Parseable [Expr] {
  static Expr parse(String s) { ... }
}
```

we now can use the line processor the implement a simple Read-Evaluate-Print-Loop:

```
class REPL {
  public static void main(String... args) throws IOException {
    LineProcessor.process(System.in, new Consumer<Expr>() {
      public void consume(Expr e) {
        System.out.println(e.prettyPrint() + " => " + e.eval());
      }
    });
  }
}
```

2.1.5 Implementation Inheritance

Suppose we would like to have a richer set of operations available for the expression hierarchy, as expressed by the following interface:

```
interface RichExpr {
  int depth();               // Computes the depth of the expression
  int size();                // Computes the size of the expression
  List<RichExpr> subExprs(); // Returns all direct sub–expressions
}
```

Providing direct implementations of `depth` and `size` for `Expr` and its subclasses would duplicate work because both can be implemented in terms of the `subExprs` method. A Java programmer has to avoid this sort of code duplication proactively: he or she would write an abstract class, say `AbstractRichExpr`, that implements `RichExpr` partially by only providing the methods `depth` and `size`. Then, `Expr` would become a subclass of `AbstractRichExpr` and would only need to provide an implementation for `subExprs` to comply to the `RichExpr` interface. However, inserting such an abstract class restricts the inheritance hierarchy by ruling out other superclasses of `Expr`. Moreover, the source code of `Expr` is needed.

JavaGI's retroactive interface implementations offer a more flexible way for writing (partial) default implementations: simply provide an abstract implementation of `RichExpr` with `RichExpr` as the implementing type. This reflects the intention of implementing some methods of `RichExpr` in terms of other methods of `RichExpr`. Here is the code for the partial default implementation of `RichExpr`:[4]

```
abstract implementation RichExpr [RichExpr] {
  int depth() {
    int i = 0;
    for (RichExpr e : subExprs()) { i = Math.max(i, e.depth()); }
    return i+1;
  }
  int size() {
    int i = 1;
    for (RichExpr e : subExprs()) { i += e.size(); }
    return i;
  }
}
```

Other implementations of `RichExpr` may then inherit from this abstract implementation:

```
implementation RichExpr [Expr] extends RichExpr [RichExpr] {
  List<RichExpr> subExprs() {
    return new LinkedList<RichExpr>();
  }
}
```

We use the syntax "**extends** `RichExpr [RichExpr]`" to specify the super implementation. The effect of the **extends** clause is that the `RichExpr [Expr]` inherits the defi-

[4] Abstract implementation definitions and implementation definitions with abstract methods (which are not necessarily abstract as a whole) are two different things. The former do not introduce a new subtyping relationship between the implementing type and the interface, whereas the latter do. Hence, JavaGI's type system treats abstract implementations more liberal and imposes fewer restrictions on them (see Section 2.3.4).

nitions of depth and size from RichExpr [RichExpr].[5]

To complete the example, we also need an implementation for PlusExpr:

```
implementation RichExpr [PlusExpr] extends RichExpr [Expr] {
                               // extends RichExpr [RichExpr] possible too
  List<RichExpr> subExprs() {
    List<RichExpr> list = new LinkedList<RichExpr>();
    list.add(this.left);
    list.add(this.right);
    return list;
  }
}
```

In the examples just shown, we referred to a super implementation by explicitly stating the interface and the implementing type. Alternatively, we may provide explicit names for implementations and then use these names in the **extends** clause. In this case, the three implementations of RichExpr would look as follows:

```
abstract implementation RichExpr [RichExpr] as DefaultImpl {...}
implementation RichExpr [Expr] as ExprImpl extends DefaultImpl {...}
implementation RichExpr [PlusExpr] extends ExprImpl {...}
```

2.1.6 Dynamic Loading of Retroactive Interface Implementations

JavaGI's retroactive interface implementations integrate nicely with the dynamic loading capabilities of Java. Here is code that loads an (imaginary) subclass MultExpr of Expr together with its retroactive implementation of the PrettyPrintable interface. The code then constructs a new instance of MultExpr (we expect the class to have a constructor taking two Expr arguments) and invokes the prettyPrint method on the new instance.

```
  Class<?> clazz = javagi.runtime.RT.classForName("MultExpr",
                                                  PrettyPrintable.class);
  Expr e = (Expr) clazz.getDeclaredConstructor(Expr.class, Expr.class)
                    .newInstance(new IntLit(2), new IntLit(21));
  String s = e.prettyPrint();
  System.out.println(s);
```

The method classForName(String name, Class<?>... ifaces), provided by the run-time system of JavaGI, simultaneously loads a class and its implementations of all interfaces given. In the example just shown, it is not possible to load MultExpr first and the PrettyPrintable implementation at some later point. This approach would allow to invoke the prettyPrint method on a MultExpr object without loading the PrettyPrintable implementation at all. Such an invocation would lead to a run-time error because the only applicable prettyPrintable method would be the abstract version in the implementation of PrettyPrintable for Expr. Consequently, JavaGI's completeness check for abstract methods would prevent MultExpr from being loaded in the first place. Loading MultExpr and its PrettyPrintable implementation simultaneously avoids the problem.

[5] The notation "I [T]" denotes the retroactive implementation of interface I for type T.

2.1.7 Multi-Headed Interfaces

So far, we only considered interfaces with exactly one implementing type. However, we can easily generalize the interface concept to include *multi-headed interfaces*. Such interfaces relate multiple implementing types and their methods and thus can place mutual requirements on the methods of all participating types. For instance, here is a multi-headed interface for the well-known Observer pattern [73]:[6]

```
interface ObserverPattern [Subject, Observer] {
  receiver Subject {
    void register(Observer o);
    void notifyObservers();
  }
  receiver Observer {
    void update(Subject s);
  }
}
```

A multi-headed interface names the implementing types (Subject and Observer in this case) explicitly through type variables enclosed in square brackets '[...]'. Moreover, it groups methods by receiver type. In the example, the ObserverPattern interface demands that the Subject part provides the methods register and notifyObservers, whereas the Observer part has to provide an update method.

Implementations of multi-headed interfaces are defined analogously to implementations of single-headed interfaces.[7] Assume that there are classes ExprPool, which maintains a pool of expressions scheduled for evaluation, and ResultDisplay, which displays the result of evaluating an expression on the screen.

```
class ExprPool {
  ...
  void register(ResultDisplay d) { ... }
  void notifyObservers() { ... }
}
class ResultDisplay { ... }
```

Class ResultDisplay is an observer for ExprPool: whenever ExprPool evaluates an expression, it notifies ResultDisplay to update the screen. We can make this relationship explicit by providing an implementation of the ObserverPattern interface:

```
implementation ObserverPattern [ExprPool, ResultDisplay] {
  /* No need to specify methods for receiver ExprPool because
     this class already contains the required methods. */
  receiver ResultDisplay {
    void update(ExprPool m) { ... }
  }
}
```

[6] Two parties participate in the Observer pattern: subject and observer. Every observer registers itself with one or more subjects. Whenever a subject changes its state, it notifies its observers by sending itself for scrutiny.

[7] Single-headed interfaces are interfaces with exactly one implementing type. In general, we use the term "n-headed interface" to refer to an interface with n implementing types.

In conjunction with multi-headed interfaces, JavaGI's constraint notation is particularly useful because it allows to constrain multiple types. The following example uses this mechanism to demand that the type variables S and O together implement the ObserverPattern interface:[8]

```
<S,O> void genericUpdate(S sub, O obs) where S*O implements ObserverPattern {
  obs.update(sub);
}
```

Because ExprPool and ResultDisplay implement the ObserverPattern interface, the invocation genericUpdate(new ExprPool(), new ResultDisplay()) is type correct.

Methods of multi-headed interfaces also preserve dynamic dispatch. As with binary methods, JavaGI takes an approach similar to multimethods and dispatches on the receiver as well as on all parameters declared as implementing types (symmetric multiple dispatch). Section 8.5 demonstrates this behavior by encoding a classic examples for multimethods [49] in JavaGI.

We end the discussion of multi-headed interfaces by remarking that the notation for single-headed interfaces used so far is just syntactic sugar. Internally, a single-headed interface is represented in the same way as a multi-headed interface. For example, the EQ interface from Section 2.1.2 is fully spelled out as:

```
interface EQ [This] {
  receiver This { boolean eq(This that); }
}
```

2.1.8 Comparison with Java

The preceding sections introduced the main features of JavaGI and demonstrated how these features solve several important programming problems. In the following, we compare the JavaGI solutions with corresponding solutions in plain Java.

Retroactive Interface Implementations

As already noted in Section 2.1.1, Java does not offer the possibility of implementing interfaces such as PrettyPrintable without changing the classes of the expression hierarchy in Figure 2.1. As a workaround, Java programmers often use the Adapter pattern [73, 93]. Applying this design pattern to the problem in Section 2.1.1 requires adapter classes for each concrete subclass of Expr and a factory class that adapts expressions according to their run-time type. See Figure 2.2 for the corresponding Java code.

Assessment. The Adapter pattern has several disadvantages with respect to JavaGI's retroactive implementations:

- It requires explicit conversion between the original and the adapted object, as demonstrated by the explicit adapter invocations PPFactory.adapt(...) in the body of prettyPrint in class PPPlusExpr (see Figure 2.2).

[8] The first version of JavaGI [240] used the notation [S,O] **implements** ObserverPattern instead of S*O **implements** ObserverPattern.

Figure 2.2 Adapter classes for pretty printing in plain Java.

```java
// Java
class PPIntLit implements PrettyPrintable {
  IntLit adaptee;
  PPIntLit(IntLit expr) { this.adaptee = expr; }
  public String prettyPrint() { return String.valueOf(this.adaptee.value); }
}
class PPPlusExpr implements PrettyPrintable {
  PlusExpr adaptee;
  PPPlusExpr(PlusExpr expr) { this.adaptee = expr; }
  public String prettyPrint() {
    return "(" + PPFactory.adapt(this.adaptee.left).prettyPrint() +
           " + " + PPFactory.adapt(this.adaptee.right).prettyPrint() + ")";
  }
}
class PPFactory {
  static PrettyPrintable adapt(Expr expr) {
    if (expr instanceof IntLit) return new PPIntLit((IntLit) expr);
    else if (expr instanceof PlusExpr) return new PPPlusExpr((PlusExpr) expr);
    else throw new RuntimeException("Unexpected expression form");
  }
}
```

- It causes object schizophrenia [198, 89]. For example, a plus-expression `e` and its adapted form **`new`** `PPPlusExpr(e)` are no longer identical (i.e., the comparison `e ==` **`new`** `PPPlusExpr(e)` evaluates to `false`).

- It hides the original interface of the object being adapted. Gamma and coworkers [73] suggest *two-way adapters* as a potential solution to this problem.

- It requires a factory class (e.g., `PPFactory` in Figure 2.2) for constructing adapter objects. Adding new expression forms requires changes to this factory class.

- It has the tendency to “infect” large areas of a program. For example, treating a list of expressions as a list of pretty-printable objects requires an adapter for the list [89]. (The list adapter adapts the individual elements whenever they are retrieved from the list.)

Explicit Implementing Types

Section 2.1.2 demonstrated that JavaGI specifies signatures for binary methods through explicit implementing types. The section also argued that the specification of a binary method signature in Java requires F-bounded polymorphism and possibly wildcards. Figure 2.3 re-implements the example from Section 2.1.2 in Java to substantiate this claim. Bracha [24] gives a different example for the same purpose.

Figure 2.3 Binary methods in plain Java.
The code avoids the problem of implementing EQ retroactively for Expr and its subclasses by defining a variant of the expression hierarchy from Figure 2.1 that directly implements Java's version of EQ.

```
// Java
interface EQ<X> {
  boolean eq(X that);
}
class Lists {
  static <X extends EQ<X>> X find(X x, List<X> list) {
    for (X y : list) {
      if (x.eq(y)) return y;
    }
    return null;
  }
}
abstract class EQExpr implements EQ<EQExpr> {
  // eval removed for simplicity
  public boolean eq(EQExpr that) { return false; }
}
class EQIntLit extends EQExpr {
  int value;
  EQIntLit(int value) { this.value = value; }
  public boolean eq(EQExpr that) {
    // simulate multiple dispatch
    if (that instanceof EQIntLit) return this.value == ((EQIntLit) that).value;
    else return super.eq(that);
  }
}
class EQPlusExpr extends EQExpr { /* code omitted for brevity */ }
```

Given variables `le`, `e`, and `i` with static types `List<EQExpr>`, `EQExpr`, and `EQIntLit`, respectively, the two invocations `Lists.find(e, le)` and `Lists.find(i, le)` typecheck. However, in contrast to the JavaGI solution in Section 2.1.2, the invocation `Lists.find(i, li)` does not typecheck for a variable `li` with static type `List<EQIntLit>`, because it causes the type parameter `X` to be instantiated with `EQIntLit` but `EQIntLit` is not a subtype of `EQ<EQIntLit>` (but of `EQ<EQExpr>`).

Allowing for this kind of flexibility in Java requires an improved version of `find`'s signature with wildcards:

```
// Java
static <X extends EQ<? super X>> X betterFind(X x, List<X> l) { /* as before */ }
```

The bound `EQ<? super X>` states that `X` does not need to be a subtype of `EQ<X>`; instead, it only has to be a subtype of `EQ<T>` where `T` is some arbitrary supertype of `X`. With the improved version of `find`, the invocation `betterFind(i, li)` typechecks successfully because `EQIntLit` is a subtype of `EQ<EQExpr>` and `EQExpr` is a supertype of `EQIntLit`. (The invocations `betterFind(e, le)` and `betterFind(i, le)` typecheck too).

Assessment. Comparing the JavaGI version with its Java counterpart reveals that explicit implementing types are syntactically much simpler than F-bounds and wildcards. Moreover, JavaGI provides symmetric multiple dispatch on explicit implementing types, something that the Java approach has to simulate by hand (e.g., by **instanceof** tests as in Figure 2.3, class `EQIntLit`, method `eq`).

On the other hand, the solution in JavaGI only works in combination with interfaces whereas Java's solution also works in a setting without interfaces. Further, Java's approach is somewhat more flexible; for example, a class `C` may implement `EQ<T>` for some arbitrary type `T`, which may be totally unrelated to `C`. However, it is unclear whether this greater flexibility is really needed in practice.

Type Conditionals

Java neither supports type-conditional interface implementations nor type conditions on methods restricting type parameters other than that of the method itself. A common approach to simulate these features is checking the type conditions not statically but dynamically through run-time casts. A different approach omits the type-conditional parts from the base class but creates a new subclass which then places the type conditions on its generic arguments.

Both approaches have disadvantages compared with the JavaGI solution presented in Section 2.1.3: the first approach may lead to unexpected run-time errors, whereas the second approach requires boilerplate code to be written and does not offer much flexibility because the type-conditional parts are not available for the base class even if its type parameters meet the type conditions. Even worse, the boilerplate code grows exponentially in the number of independent type conditions because each combination of type conditions demands a new subclass.

Static Interface Methods

In JavaGI, programmers abstract over constructor-like methods through static interface methods. Java programmers use the Factory pattern [73] instead. Implementing the line processor from Section 2.1.4 with the Factory pattern requires an interface

```
interface Parser<X> {
  X parse(String s);
}
```

and the following modified signature of method `process` in class `LineProcessor`:

```
  static <X> void process(InputStream in, Consumer<X> c, Parser<X> p)
             throws IOException
```

The additional parameter `p` simulates the constraint `X` **implements** `Parseable` of the corresponding JavaGI signature in Section 2.1.4. However, JavaGI implicitly passes evidence for this constraint, whereas a Java programmer has to supply the extra parameter explicitly. For the tiny example from Section 2.1.4, the extra parameter does not make a big difference, but explicitly maintaining it over a long sequence of method calls quickly becomes a burden.

Multi-Headed Interfaces

JavaGI's multi-headed interfaces specify mutual dependencies between several types. In the literature, this phenomenon is known as *family polymorphism* [68]. It is well known [68] that object-oriented languages such as Java do not support family polymorphism in a statically safe and flexible way. JavaGI, however, provides a type-safe and sufficiently expressive form of family polymorphism, as demonstrated by the example in Section 2.1.7. (Section 8.3 evaluates support for family polymorphism in JavaGI according to the criteria established by Ernst.) In addition to family polymorphism, JavaGI's multi-headed interfaces in combination with explicit implementing types also support symmetric multiple dispatch, a feature not present in Java either.

2.2 Design Principles

The design of JavaGI rests on six principles.

Conservativeness. JavaGI is a conservative extension of Java. That is, a program that works in Java works the same way in JavaGI. The JavaGI compiler translates all input programs to standard Java byte code [125], retaining the semantics and the performance characteristics of Java programs even in the presence of retroactive implementations. Conservativeness enables easy migration from Java to JavaGI and ensures full compatibility with existing Java APIs.

Extensibility. JavaGI imposes no restrictions on the placement of retroactive interface implementations. That is, implementation definitions can be placed in arbitrary compilation units and arbitrary libraries. Extensibility maximizes flexibility and allows for a high degree of interworking between Java and JavaGI code.

Dynamicity. JavaGI fully supports dynamic loading. That is, not only classes and interfaces but also retroactive implementation definitions can be loaded dynamically at any time. Dynamicity ensures compatibility with existing Java libraries and frameworks. For example, dynamic loading is required to run JavaGI programs inside a servlet container [215].

Type Safety. JavaGI favors static type safety over unsafe dynamic checks. That is, the language provides an expressive type system and checks as many properties as possible at compile time. It resorts to dynamic checks only if required to support extensibility or dynamicity. Static type safety prevents a whole class of software defects right from the start.

Modularity. JavaGI features fully modular compilation and mostly modular typechecking. That is, compilation and typechecking of a compilation unit does not need access to internals of other compilation units, and code generation processes each compilation unit in isolation. To allow for extensibility, dynamicity, and type safety at the same time, the JavaGI compiler abandons completely modular typechecking and performs certain global checks on the set of types and implementation definitions available. However, the compiler never assumes that it knows all implementation definitions (open-world assumption), so new implementations can be added at any time provided they do not conflict with existing ones. Modularity is important for building large software projects. Further, the open-world assumption facilitates the extension of JavaGI libraries with new implementations without recompiling the libraries.

Transparency. JavaGI provides retroactive interface implementations in a transparent way. That is, the run-time behavior of a retroactive implementation cannot be distinguished from that of a Java-style interface implementation. Furthermore, the compile-time characteristics of a retroactive and a Java-style implementation are very similar. Transparency enables programmers to reason about retroactive implementations in almost the same way as they reason about Java-style implementations.

2.3 An Informal Account of Typechecking and Execution

This section informally investigates the JavaGI-specific extensions of Java's type system and execution model. It explains constraint entailment, subtyping, and method typing. Further, it defines global well-formedness criteria for programs and describes dynamic method lookup.

2.3.1 Constraint Entailment

Constraint entailment is a notion not present in Java's type system. It establishes the validity of constraints. JavaGI distinguishes two kinds of constraints, *subtype constraints* and *implementation constraints*.

- Subtype constraints generalize Java's type parameter bounds. A subtype constraint has the form `T` **`extends`** `U`, where `T` and `U` are both types.[9] Such a constraint is *valid* if `T` is a subtype of `U` (see Section 2.3.2).

- Implementation constraints have the form $\texttt{T}_1\texttt{*...*T}_n$ **`implements`** `K` where $\texttt{T}_1, \ldots, \texttt{T}_n$ are types and `K` is a n-headed interface. For simplicity, this informal discussion only considers the case where $n = 1$. Such a constraint `T` **`implements`** `K` is valid in any of the following cases (see Section 3.3 for the complete list).

  1. `T` implements interface `K` in the Java sense: `T` is a class and `T` itself or a superclass of `T` has an explicit **`implements`** clause for `K`.
  2. `T` is a type variable declared to implement `K` or some of its subinterfaces.

[9] Constraint declarations are restricted to the form `X` **`extends`** `U`, where `X` is a type variable. A Java type parameter bound `X` **`extends`** $\texttt{T}_1\texttt{\&...\&T}_n$ is represented by multiple constraints `X` **`extends`** $\texttt{T}_1$, ..., `X` **`extends`** $\texttt{T}_n$.

3. A non-abstract retroactive implementation matches K and T (or some supertype of T unless K contains methods with the implementing type in result position). If the implementation is type conditional (see Section 2.1.3), then the constraints of the implementation must also be satisfied.

Suppose a program contains the EQ implementations for Expr and List from Sections 2.1.2 and 2.1.3. The constraint LinkedList<Expr> **implements** EQ is valid by the third case:

- The implementing type of EQ does not appear in result position, so it is possible to lift LinkedList<Expr> to the supertype List<Expr>.
- There exists an implementation EQ [List<X>] (parameterized over X) that matches EQ and List<Expr> by instantiating X to Expr.
- The implementation's constraint after instantiation is Expr **implements** EQ, which is valid because of the implementation EQ [Expr].

In contrast, LinkedList<String> **implements** EQ cannot by derived from the set of implementations defined in Sections 2.1.2 and 2.1.3 because String **implements** EQ does not hold.

An implementation constraint is stronger than an subtype constraint: validity of T **implements** K implies validity of T **extends** K, but the reverse implication is not always true. To demonstrate this fact, continue the example code from Section 2.1.2 and Section 2.1.3 as follows:

```
EQ e1 = new IntLit(42);          // ok
EQ e2 = new LinkedList<Expr>();  // ok
if (e1.eq(e2)) ...               // type error
```

While e1 and e2 can both be subsumed to the interface type EQ (see Section 2.3.2) and EQ **extends** EQ is clearly valid, the binary method call with e1 and e2 does not make sense as it would compare an integer with a list. For this reason, JavaGI requires EQ **implements** EQ to typecheck the call e1.eq(e2). But EQ **implements** EQ does not hold, so the JavaGI compiler correctly rejects the call.

Besides being stronger, implementation constraints may be used to constrain a group of types with a multi-headed interface, as demonstrated in Section 2.1.7 by the constraint S*O **implements** ObserverPattern. In contrast, a subtype constraint relates exactly two types. Furthermore, each invocation of a retroactively implemented or static interface method must eventually be sanctioned by a corresponding implementation constraint to ensure type soundness.

2.3.2 Subtyping

The subtyping relation, written T <: U for types T and U, indicates that an object of type T can also be used with type U. JavaGI's subtyping relation extends Java's: it considers more types to be subtypes of each other than Java.

To test whether T <: U holds, JavaGI first checks whether T <: U already holds in Java. Otherwise, T <: U can only hold if U is an interface type and T implements U.

That is, there must be a supertype `V` of `T` (possibly `T` itself) such that the constraint `V implements U` holds.

2.3.3 Method Typing

JavaGI's algorithm for typechecking method invocations extends the corresponding algorithm employed by Java. If the rules of Java are sufficient to typecheck an invocation, then it also typechecks in JavaGI and the invocation is marked as a "Java call-site". Otherwise, JavaGI's constraint entailment tries to prove a suitable implementation constraint for the invocation.

In particular, assume that the method invocation not typeable according to Java's rule has the form $\mathtt{e}_0\mathtt{.m(e}_1\mathtt{,...,e}_n\mathtt{)}$ for expressions $\mathtt{e}_0, \mathtt{e}_1, \ldots, \mathtt{e}_n$ with static types $\mathtt{T}_0, \mathtt{T}_1, \ldots, \mathtt{T}_n$. To typecheck the invocation, the JavaGI compiler first searches all interfaces accessible from the current compilation unit under their unqualified name for methods matching name `m`, receiver type $\mathtt{T}_0$ and argument types $\mathtt{T}_1, \ldots, \mathtt{T}_n$. This process is very similar to the method typing algorithm described in sections 15.12.2 and 15.12.3 of *The Java Language Specification* [82]. It includes inference of type arguments and it instantiates the implementing types of the current interface according to the signature of the method being examined and according to the types $\mathtt{T}_0, \ldots, \mathtt{T}_n$. If the compiler does not find any matching methods, typechecking fails.

Next, the compiler shrinks the resulting set of candidate methods by removing methods that are less specific than other candidate methods. If this process results in one candidate, typechecking succeeds and the compiler marks the invocation as a "JavaGI call-site". Otherwise, it rejects the method invocation as ambiguous.

There is a mechanism for resolving ambiguities by explicitly specifying which interface to search for candidate methods. For example, suppose that interface `PrettyPrintable` from Section 2.1.1 and another interface `J` are in scope. Assume further that `J` defines a method `prettyPrint()` and that `Expr` implements `J`. Then the call `e.prettyPrint()`, where `e` is a variable with static type `Expr`, is ambiguous. But JavaGI also provides the syntax `e.prettyPrint::PrettyPrintable()` to invoke the `prettyPrint` method of interface `PrettyPrintable` explicitly.

A static interface method invocation is always explicit. It includes the interface name and all implementing types to avoid potential ambiguities from the start.

2.3.4 Well-Formedness Criteria for Programs

JavaGI's type system imposes certain global well-formedness criteria on the set of implementation definitions to guarantee that run-time lookup of retroactively implemented methods always finds a unique and most specific implementation definition that contains a non-abstract version of the method in question. Moreover, the criteria ensure that dynamic method lookup need not perform constraint entailment when searching for the most specific implementation. Constraint entailment at run time is not feasible because JavaGI inherits its type-erasure semantics from Java [26], so type arguments are not available when actually executing a program. Last but not least, the criteria establish decidability of constraint entailment and subtyping, and they enable efficient method lookup.

Criterion: No Overlap

Any two non-abstract implementations of the same interface must not overlap; that is, the erasures of the implementing types must not be equal. Overlapping implementation definitions lead to ambiguity in dynamic method lookup.

For example, a program must not contain the `PrettyPrintable` implementation for `IntLit` from Section 2.1.1 along with some other `PrettyPrintable` implementation for `IntLit`. Otherwise, both implementations would be candidates for an invocation like **`new`** `IntLit(42).prettyPrint()`, but neither implementation is more specific than the other. The "no overlap" criterion rejects such a program.

Criterion: Unique Interface Instantiation and Non-Dispatch Types

Any two non-abstract implementations of the same interface and with subtype compatible implementing types must have identical interface type arguments and identical non-dispatch types. Thereby, the implementing types $\mathtt{T}_1, \ldots, \mathtt{T}_n$ and $\mathtt{U}_1, \ldots, \mathtt{U}_n$ of two retroactive implementations are *subtype compatible* if, and only if, for all $i \in \{1, \ldots, n\}$ either $\mathtt{T}_i$ `<:` $\mathtt{U}_i$ or $\mathtt{U}_i$ `<:` $\mathtt{T}_i$ holds. Furthermore, an implementing type `X` of some interface is a *non-dispatch type* if the interface itself or some of its superinterfaces contains at least one non-static method such that `X` is neither the receiver type of the method nor does it appear among its argument types. Otherwise, `X` is a *dispatch type*.

The restriction on identical interface type arguments is necessary to avoid ambiguity in dynamic method lookup because JavaGI's type erasure semantics maps different instantiations of an interface to the same run-time representation. Moreover, Java disallows multiple instantiation inheritance for interfaces [82, § 8.1.5].

A program containing two implementations of the same interface and with subtype-compatible implementing types but different non-dispatch types may also exhibit ambiguous method lookup at run time. For example, suppose that a program contains the `ObserverPattern` implementation for `ExprPool` and `ResultDisplay` from Section 2.1.7, as well as an `ObserverPattern` implementation for `ExprPool` and some class `MyObserver`. Then the call **`new`** `ExprPool().notify()` cannot be resolved unambiguously at run time because the two implementations differ only in the second implementing type (`ResultDisplay` and `MyObserver`), but it is not possible to determine this implementing type from the call **`new`** `ExprPool().notify()`. However, the second implementing type of `ObserverPattern` is a non-dispatch type (it is neither the receiver nor an argument of `notify`), so the two `ObserverPattern` implementations considered violate the "unique non-dispatch types" criterion.

Criterion: Downward Closed

Any two non-abstract implementations of the same interface `I` must be downward closed. That is, if $\mathtt{T}_1, \ldots, \mathtt{T}_n$ and $\mathtt{U}_1, \ldots, \mathtt{U}_n$ are the implementing types of the two implementations given, and $\mathtt{V}_1, \ldots, \mathtt{V}_n$ is a vector of types such that each $\mathtt{V}_i$ is a maximal element of the set of lower bounds of T_i and U_i, then an implementation of interface I with implementing types $V_1, \ldots, V_n$ must exist.

This criterion rules out ambiguity of dynamic method lookup in cases like the following, where the `chooseIntLit` method is to return the `IntLit` instance among its arguments:

```
interface ChooseIntLit [Expr1, Expr2] {
  receiver Expr1 {
    IntLit chooseIntLit(Expr2 that);
  }
}
implementation ChooseIntLit [Expr, IntLit] {
  receiver Expr {
    IntLit chooseIntLit(IntLit that) { return that; }
  }
}
implementation ChooseIntLit [IntLit, Expr] {
  receiver IntLit {
    IntLit chooseIntLit(Expr that) { return this; }
  }
}
```

The call `new IntLit(42).chooseIntLit(new IntLit(3))` is ambiguous with these definitions because both implementations are applicable but none is more specific than the other. JavaGI rules out such programs because the two implementations are not downward closed. To make the program well-formed requires a third implementation that is more specific than the two implementations of `ChooseIntLit` already shown:

```
implementation ChooseIntLit [IntLit, IntLit] {
  receiver IntLit {
    IntLit chooseIntLit(IntLit i) { return this; }
  }
}
```

Another situation that exhibits ambiguous method lookup is the following:

```
interface J { ... }
interface K { ... }
class C implements J, K { ... }
implementation PrettyPrintable [J] {
  String prettyPrint() { return "J"; }
}
implementation PrettyPrintable [K] {
  String prettyPrint() { return "K"; }
}
```

The call `new C().prettyPrint()` may return either `"J"` or `"K"` because the implementations for `J` and `K` both match but none is more specific than the other. However, the two implementations are not downward closed, so JavaGI rejects the program. To successfully compile the program requires an implementation of `PrettyPrintable` for class `C`.

Criterion: Consistent Type Conditions

Constraints on non-abstract implementations must be consistent with subtyping: if the implementing types of a non-abstract implementation $\mathcal{I}_1$ are pairwise subtypes of the implementing types of another non-abstract implementation $\mathcal{I}_2$, then the constraints of $\mathcal{I}_2$ must imply the constraints of $\mathcal{I}_1$.

Without this criterion, JavaGI would need run-time constraint entailment to rule out certain implementations when performing dynamic method lookup. For example, consider the following extension of code from Section 2.1.3:

```
// repeated for clarity
implementation<X> EQ [List<X>]  where X implements EQ { ... }
// new implementation
implementation<X> EQ [LinkedList<X>] where X extends Number { ... }
```

Now consider the call `list1.eq(list2)`, where both `list1` and `list2` have (dynamic) type `LinkedList<Expr>`. The implementation for `List<X>` may be used to resolve this call but the one for `LinkedList<X>` may not because the constraint `Expr` **`extends`** `Number` does not hold. However, JavaGI's run-time system is unable to detect this mismatch because it cannot perform constraint entailment at run time (in particular, the type argument `Expr` is not available because of type erasure [26]).

Thus, JavaGI rejects the program statically because `LinkedList<X>` is a subtype of `List<X>` but the constraint `X` **`implements`** `EQ` of the `List<X>` implementation does not imply the constraint `X` **`extends`** `Number` of the `LinkedList<X>` implementation.

Criterion: No Implementation Chains

Retroactive implementations must not form a chain by using the interface of a non-abstract implementation as the implementing type of some (other) non-abstract implementation. For example, Section 2.1.2 implements the `EQ` interface retroactively, so it is not possible to use `EQ` as an implementing type of any non-abstract implementation.

Disallowing implementation chains ensures decidability of constraint entailment and subtyping (see Section 5.1 for details). Moreover, it allows for efficient run-time lookup of retroactively implemented methods.

Criterion: Completeness

The implementation of an interface method must be complete, even if there exist retroactive implementations with abstract definitions for the method. That is, if a retroactive implementation of interface `I` contains an abstract definition of method `m` with $\mathtt{T}_1, \ldots, \mathtt{T}_n$ being the dispatch-relevant argument types (i.e., the receiver type and those argument types declared as implementing types in `I`), then the following must hold: for each sequence of non-abstract types $\mathtt{U}_1, \ldots, \mathtt{U}_n$ with $\mathtt{U}_i$ `<:` $\mathtt{T}_i$ for all $i \in \{1, \ldots, n\}$, there exists a retroactive implementation of `I` containing a non-abstract definition of `m` with $\mathtt{V}_1, \ldots, \mathtt{V}_n$ being the dispatch-relevant argument types such that $\mathtt{U}_i$ `<:` $\mathtt{V}_i$ and $\mathtt{V}_i$ `<:` $\mathtt{T}_i$ for all $i \in \{1, \ldots, n\}$. The completeness criterion ensures that dynamic method lookup never encounters an abstract definition of some interface method.

For example, consider the following extension of the code from Section 2.1.1:

```
class MultExpr extends Expr { ... }
```

Dynamic dispatch for an invocation **new** MultExpr(...).prettyPrint() would find the abstract definition of prettyPrint in the PrettyPrintable implementation for Expr; consequently, a “message not understood” error would occur at run time. Fortunately, the completeness criterion prevents the definition of MultExpr without an additional implementation of PrettyPrintable for MultExpr.

Checking the Criteria

The JavaGI compiler checks the well-formedness criteria just described on all accessible types and implementations. At run time, however, a different set of types and implementations may be available because of subsequent edits or dynamic loading. Hence, JavaGI’s run-time system re-checks the well-formedness criteria every time it loads a new type or a new set of implementations. Nevertheless, the compiler can guarantee one important property: if a program meets the well-formedness criteria at compile time and the same set of types and implementations is available at run time, then the run-time checks never fail.

2.3.5 Dynamic Method Lookup

At program start, JavaGI’s run-time system loads all accessible implementations, checks the well-formedness criteria just explained, and installs the implementations loaded as the current pool of implementations. A dynamically loaded implementation extends this pool after checking that the well-formedness criteria still hold.

For Java call-sites (see Section 2.3.3), dynamic method lookup is the same as for plain Java. For JavaGI call-sites, which the compiler also marks with the interface defining the method and the argument positions of the implementing types, dynamic method lookup searches the pool of implementations for one that matches

1. the interface in which the method is defined,
2. the dynamic receiver type, and
3. the dynamic types of those arguments declared as implementing types in the interface method signature.

Static typing and the well-formedness criteria guarantee that this search always returns a unique most specific implementation.

The static distinction between Java call-sites and JavaGI call-sites requires that methods in retroactive implementations do not override methods defined in classes. However, the conservativeness principle postulated in Section 2.2 prevents such retroactive method overrides anyway: allowing them means that the behavior of an existing Java program could be modified by adding an appropriate implementation that overrides an internal method of some class.

3 Formalization of CoreGI

This chapter takes a more formal route than the preceding one: it distills the core features of JavaGI into a small calculus called CoreGI and provides a rigorous formalization of it. The definition of CoreGI is based on that of Featherweight Generic Java (FGJ [96]).

To keep the formalization within reasonable size and complexity limits, CoreGI omits many details of the full language. It includes, however, the essential aspects of JavaGI's generalized interface concept and allows to express the common programming idioms of JavaGI. One exception of this rule is the lack of support for interfaces as implementing types of retroactive implementations. CoreGI does not deal with this aspect of JavaGI and defers it until Chapter 5.

Chapter Outline. The chapter consists of seven sections.

- Section 3.1 introduces some basic notations.
- Section 3.2 defines the syntax of CoreGI.
- Section 3.3 formalizes constraint entailment and subtyping for CoreGI.
- Section 3.4 specifies CoreGI's dynamic semantics (i.e., its run-time behavior).
- Section 3.5 presents CoreGI's static semantics (i.e., its type system).
- Section 3.6 proves that the type system of CoreGI is sound and that its evaluation relation is deterministic.
- Section 3.7 defines algorithms for deciding constraint entailment, subtyping, expression typing, and program typing in CoreGI.

3.1 Basic Notations

This section introduces some basic notations used throughout the rest of the dissertation. In the following, ξ denotes some arbitrary syntactic construct.

Figure 3.1 Syntax.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{\mathit{def}}\ e \\
\mathit{def} & ::= & \mathit{cdef} \mid \mathit{idef} \mid \mathit{impl} \\
\mathit{cdef} & ::= & \textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\,\overline{T\,f}\ \ \overline{m : \mathit{mdef}}\,\} \\
\mathit{idef} & ::= & \textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\,\overline{Y}\,\textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{P}\,\{\,\overline{m : \textbf{static}\,\mathit{msig}}\ \ \overline{\mathit{rcsig}}\,\} \\
\mathit{impl} & ::= & \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ K\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \{\,\overline{\textbf{static}\,\mathit{mdef}}\ \ \overline{\mathit{rcdef}}\,\} \\
\mathit{rcsig} & ::= & \textbf{receiver}\,\{\overline{m : \mathit{msig}}\} \\
\mathit{rcdef} & ::= & \textbf{receiver}\,\{\overline{\mathit{mdef}}\} \\
\mathit{msig} & ::= & \texttt{<}\overline{X}\texttt{>}\,\overline{T\,x} \to T\ \textbf{where}\ \overline{P} \\
\mathit{mdef} & ::= & \mathit{msig}\,\{e\} \\
M, N & ::= & C\texttt{<}\overline{T}\texttt{>} \mid \mathit{Object} \\
G, H & ::= & X \mid N \\
K, L & ::= & I\texttt{<}\overline{T}\texttt{>} \\
T, U, V, W & ::= & G \mid K \\
R, S & ::= & \overline{G}\,\textbf{implements}\,K \\
\mathcal{R}, \mathcal{S} & ::= & \overline{T}\,\textbf{implements}\,K \\
P, Q & ::= & R \mid X\,\textbf{extends}\,T \\
\mathcal{P}, \mathcal{Q} & ::= & \mathcal{R} \mid T\,\textbf{extends}\,T \\
d, e & ::= & x \mid e.f \mid e.m\texttt{<}\overline{T}\texttt{>}(\overline{e}) \mid K[\overline{T}].m\texttt{<}\overline{T}\texttt{>}(\overline{e}) \mid \textbf{new}\,N(\overline{e}) \mid (T)\,e
\end{array}
$$

$$
\begin{array}{lll}
X, Y, Z \in \mathit{TvarName} & C, D \in \mathit{ClassName} & I, J \in \mathit{IfaceName} \\
m \in \mathit{MethodName} & f, g \in \mathit{FieldName} & x, y, z \in \mathit{VarName}
\end{array}
$$

Definition 3.1. Overbar notation $\overline{\xi}^n$ (or $\overline{\xi}$ for short) denotes the sequence $\xi_1 \ldots \xi_n$ where in some places commas separate the sequence items. The symbol $\bullet$ denotes the empty sequence. Using index variables i, j, k to subscript items from a sequence assumes that the index variables range over the length of the sequence. Furthermore, if the same index variable subscripts items from different sequences, then all sequences involved are assumed to be of the same length. An index variable under an overbar marks the parts that vary from sequence item to sequence item; for example, $\overline{\xi'\,\xi_i}$ abbreviates $\xi'\,\xi_1 \ldots \xi'\,\xi_n$. At some points, the sequence $\overline{\xi}$ stands for the set $\{\xi_1, \ldots, \xi_n\}$.

Definition 3.2. The notation $\xi^?$ denotes an optional construct; that is, $\xi^?$ is either a regular ξ or the special symbol nil.

Definition 3.3. The notation $[n]$ denotes the set $\{1, \ldots, n\}$ for some $n \in \mathbb{N}$. If $n = 0$ then $[n] = \emptyset$.

3.2 Syntax

Figure 3.1 defines the abstract syntax of CoreGI. The various kinds of identifiers are drawn from pairwise disjoint and countably infinite sets of type variables (ranged over by X, Y, Z), class names (ranged over by C, D), interface names (ranged over by I, J), method names (ranged over by m), field names (ranged over by f, g), and expression variables (ranged over by x, y, z).

A CoreGI program *prog* consists of a sequence of definitions *def* followed by a "main" expression e. A definition is either a class, interface, or implementation definition.

The type parameters $\overline{X}$ of classes, interfaces, implementations, and methods do not carry explicit bounds; instead, CoreGI exclusively uses constraint clauses of the form "**where** $\overline{P}$". For readability, code fragments omit empty type parameter lists "< • >" and empty constraint clauses "**where** •".

Each class C has an explicit superclass N, where N is a class type (either an instantiated class or *Object*). If the superclass is *Object*, we sometimes omit the **extends** clause completely. The predefined class *Object* does not have a superclass and it does not define any fields or methods. The body of an ordinary class contains a sequence of field definitions $T\,f$, where T is a type and f the name of the field, followed by a sequence of method definitions $m : \mathit{mdef}$, where m is the method name and *mdef* specifies the signature *msig* and the body expression e of the method. The signature of a method consists of type parameters $\overline{X}$, value parameters $\overline{x}$ together with their types $\overline{T}$, a result type T, and constraints $\overline{P}$.

An interface I is not only parameterized over regular type parameters $\overline{X}$ but also over type parameters $\overline{Y}$, standing for the interface's implementing types. The implementation constraints $\overline{R}$ (explained shortly) attached to the implementing type parameters specify the superinterfaces of I. These superinterface constraints naturally generalize Java's **extends** clause for interfaces, which are not expressive enough in the presence of multi-headed interfaces.

The body of an interface contains method signatures $m : \mathit{msig}$ for static methods and receiver signatures *rcsig* holding the signatures of non-static methods. Unlike in full JavaGI, receivers are matched by position, not by name; that is, the ith receiver corresponds to the ith implementing type. Furthermore, CoreGI does not support interface methods to be implemented directly in classes. With respect to naming of interface methods, the following conventions apply:

Convention 3.4 (Disjoint namespaces for class and interface methods). The namespaces for class and interface methods are disjoint. At some points, m^{c} or m^{i} explicitly denotes the name of a class or interface method, respectively.

Convention 3.5 (Globally unique names of interface methods). The names of interface methods are globally unique; that is, if some interface defines a method m then no other interface defines a method with the same name m.

An implementation definition specifies a retroactive implementation of interface K for implementing types $\overline{N}$, where $\overline{N}$ is a sequence of class types. (Full JavaGI also allows single-headed interfaces to be implemented by an interface type, see Section 6.1.6.) The body of an implementation contains static methods and receiver definitions. Static methods are anonymous because they are matched by position against the static methods of the interface being implemented. Similar to interfaces, receiver definitions are matched by position, so the ith receiver definition corresponds to the ith implementing type. Moreover, methods inside receiver definitions are anonymous because they are matched by position against the methods in the corresponding receiver signature of the interface being implemented. For example, in an implementation of interface I, the jth method of the ith receiver definition corresponds to the jth method of the ith receiver signature of I.

Metavariables M, N range over class types, whereas G, H denote either a type variable or a class type N. Metavariables K, L range over interface types. Full types (denoted by T, U, V, W) are either G-types or interface types. By convention, code fragments omit empty type argument lists "$<\bullet>$".

Constraints come in four forms:

- R, S denote *implementation constraints* that constrain only G-types;
- P, Q denote either *subtype constraints* on type variables or R-constraints;
- $\mathcal{R}, \mathcal{S}$ denote unrestricted implementation constraints that may constrain arbitrary types;
- $\mathcal{P}, \mathcal{Q}$ denote unrestricted P-constraints.

With single-headed interfaces, R-constraints on class types (i.e., constraints of the form N **implements** K) are merely obfuscated syntax for trivial constraints that are unconditionally true or false. With multi-headed interfaces, however, they allow the specification of dependencies between class types and type variables. The constraint forms $\mathcal{R}$ and $\mathcal{P}$ do not occur in source programs but only as the result of applying a type substitution to some R- or P-constraint.

Expressions d, e include variables, field accesses, method calls, object allocations, and casts. A method call of the form $e.m\texttt{<}\overline{T}\texttt{>}(\overline{e})$ invokes method m on receiver e with type arguments $\overline{T}$ and expression arguments $\overline{e}$. (Full JavaGI supports inference of type arguments much as Java does.) Calling a static interface method takes the form $K[\overline{T}].m\texttt{<}\overline{U}\texttt{>}(\overline{e})$, where K is the interface defining method m, $\overline{T}$ are the relevant implementing types, and $\overline{U}$ and $\overline{e}$ are the type and expression arguments, respectively.

Convention 3.6. Syntactic constructs that differ only in the names of bound type and expression variables are interchangeable in all contexts [176].

3.3 Constraint Entailment and Subtyping

Constraint entailment (entailment for short) and subtyping play important roles in both the dynamic and the static semantics of CoreGI: in the dynamic semantics, method dispatch and evaluation of cast operations rely on subtyping; in the static semantics, expression typing and many other definitions depend on entailment and subtyping. This section presents a declarative specification of constraint entailment and subtyping; we defer an algorithmic formulation until Section 3.7.

The auxiliary predicate $\mathsf{non\text{-}static}(I)$, defined in Figure 3.2, asserts that neither interface I nor any of its superinterfaces defines a static method. The *polarity* of the ith implementing type of interface I is positive (or negative) in I, written $i \in \mathsf{pol}^{+}(I)$ (or $i \in \mathsf{pol}^{-}(I)$), if it does not occur in contravariant (or covariant) positions. We let π range over $+$ and $-$. The notation $\mathsf{ftv}(\xi)$ denotes the set of type variables free in ξ.

Figure 3.2 Restrictions on interfaces and implementing types.

$\boxed{\mathsf{non\text{-}static}(I)}$

NON-STATIC-IFACE

$$\frac{\begin{array}{c}\mathbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\,\overline{Y}\ \mathbf{where}\ \overline{R}\,]\ \mathbf{where}\ \overline{P}\,\{\,\overline{m : \mathbf{static}\ msig}^{\,n}\ \ldots\,\}\\ n = 0 \qquad (\forall i)\ \text{if}\ R_i = \overline{Z}\ \mathbf{implements}\ J\texttt{<}\overline{T}\texttt{>}\ \text{then}\ \mathsf{non\text{-}static}(J)\end{array}}{\mathsf{non\text{-}static}(I)}$$

$\boxed{j \in \mathsf{pol}^{\pi}(I) \qquad X \in \mathsf{pol}^{\pi}(rcsig) \qquad X \in \mathsf{pol}^{\pi}(P) \qquad X \in \mathsf{pol}^{\pi}(msig)}$

POL-IFACE

$$\frac{\begin{array}{c}\mathbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\,\overline{Y}\ \mathbf{where}\ \overline{R}\,]\ \mathbf{where}\ \overline{P}\,\{\,\overline{m : \mathbf{static}\ msig}\ \overline{rcsig}\,\}\\ (\forall i)\ Y_j \in \mathsf{pol}^{\pi}(msig_i) \qquad (\forall i)\ Y_j \in \mathsf{pol}^{\pi}(rcsig_i) \qquad (\forall i)\ Y_j \in \mathsf{pol}^{\pi}(R_i) \qquad Y_j \notin \mathsf{ftv}(\overline{P})\end{array}}{j \in \mathsf{pol}^{\pi}(I)}$$

POL-RECV

$$\frac{(\forall i)\ X \in \mathsf{pol}^{\pi}(msig_i)}{X \in \mathsf{pol}^{\pi}(\mathbf{receiver}\ \{\overline{m : msig}\})}$$

POL-CONSTR

$$\frac{(\forall i)\ \text{if}\ X = G_i\ \text{then}\ i \in \mathsf{pol}^{\pi}(I)}{X \in \mathsf{pol}^{\pi}(\overline{G}\ \mathbf{implements}\ I\texttt{<}\overline{U}\texttt{>})}$$

POL-MSIG-PLUS

$$\frac{Y \notin \mathsf{ftv}(\overline{T}) \setminus \overline{X}}{Y \in \mathsf{pol}^{+}(\texttt{<}\overline{X}\texttt{>}\,\overline{T\,x} \to U\ \mathbf{where}\ \overline{P})}$$

POL-MSIG-MINUS

$$\frac{Y \notin \mathsf{ftv}(U) \setminus \overline{X}}{Y \in \mathsf{pol}^{-}(\texttt{<}\overline{X}\texttt{>}\,\overline{T\,x} \to U\ \mathbf{where}\ \overline{P})}$$

The definition of $j \in \mathsf{pol}^{\pi}(I)$ by rule POL-IFACE in Figure 3.2 relies on the polarity of an implementing type variable X in receiver signatures ($X \in \mathsf{pol}^{\pi}(rcsig)$), constraints ($X \in \mathsf{pol}^{\pi}(P)$), and method signatures ($X \in \mathsf{pol}^{\pi}(msig)$). The definition of the latter by rules POL-MSIG-PLUS and POL-MSIG-MINUS depends on a restriction stating that an implementing type variable may appear in a method signature only at the top level of the result type and at the top level of the argument types. Section 3.5.3 formalizes this restriction as well-formedness criterion WF-IFACE-3.

Definition 3.7 (Type environment). A *type environment* Δ is a finite set of type variables X and constraints P. The domain of a type environment Δ, written $\mathsf{dom}(\Delta)$, is the set of type variables contained in Δ. The notation Δ, P abbreviates $\Delta \cup \{P\}$ and Δ, X stands for $\Delta \cup \{X\}$ assuming $X \notin \mathsf{dom}(\Delta)$.

Constraint entailment, written $\Delta \Vdash \mathcal{P}$, asserts that constraint $\mathcal{P}$ holds under type environment Δ. The notation $\Delta \Vdash \overline{\mathcal{P}}$ abbreviates $(\forall i)\ \Delta \Vdash \mathcal{P}_i$. The definition of constraint entailment is interweaved with the definition of the subtyping relation $\Delta \vdash T \leq U$, which holds if, and only if, T is a subtype of U under type environment Δ. At some points, $\Delta \vdash \overline{T} \leq \overline{U}$ abbreviates $(\forall i)\ \Delta \vdash T_i \leq U_i$. Figure 3.3 defines entailment and subtyping.

Rule ENT-EXTENDS solves subtype constraints by invoking the subtyping relation, and rule ENT-ENV specifies that a constraint from the type environment is always considered

Figure 3.3 Constraint entailment and subtyping.

$\boxed{\Delta \Vdash \mathcal{P}}$

ENT-EXTENDS

$$\frac{\Delta \vdash T \leq U}{\Delta \Vdash T\,\textbf{extends}\,U}$$

ENT-ENV

$$\frac{P \in \Delta}{\Delta \Vdash P}$$

ENT-SUPER

$$\frac{\textbf{interface}\ I{<}\overline{X}{>}\,[\overline{Y}\ \textbf{where}\,\overline{R}]\ \ldots \qquad \Delta \Vdash \overline{U}\,\textbf{implements}\,I{<}\overline{T}{>}}{\Delta \Vdash [\overline{T/X}, \overline{U/Y}]R_i}$$

ENT-IMPL

$$\frac{\textbf{implementation}{<}\overline{X}{>}\ I{<}\overline{T}{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots \qquad \Delta \Vdash [\overline{U/X}]\overline{P}}{\Delta \Vdash [\overline{U/X}](\overline{N}\,\textbf{implements}\,I{<}\overline{T}{>})}$$

ENT-UP

$$\frac{\Delta \vdash U \leq U' \quad \Delta \Vdash \overline{T}\,U'\,\overline{V}\,\textbf{implements}\,I{<}\overline{W}{>} \quad n \in \mathsf{pol}^{-}(I)}{\Delta \Vdash \overline{T}^{n-1}\,U\,\overline{V}\,\textbf{implements}\,I{<}\overline{W}{>}}$$

ENT-IFACE

$$\frac{1 \in \mathsf{pol}^{+}(I) \qquad \mathsf{non\text{-}static}(I)}{\Delta \Vdash I{<}\overline{T}{>}\,\textbf{implements}\,I{<}\overline{T}{>}}$$

$\boxed{\Delta \vdash T \leq U}$

SUB-REFL

$$\Delta \vdash T \leq T$$

SUB-OBJECT

$$\Delta \vdash T \leq \mathit{Object}$$

SUB-TRANS

$$\frac{\Delta \vdash T \leq U \qquad \Delta \vdash U \leq V}{\Delta \vdash T \leq V}$$

SUB-VAR

$$\frac{X\,\textbf{extends}\,T \in \Delta}{\Delta \vdash X \leq T}$$

SUB-CLASS

$$\frac{\textbf{class}\ C{<}\overline{X}{>}\ \textbf{extends}\ N\ldots}{\Delta \vdash C{<}\overline{T}{>} \leq [\overline{T/X}]N}$$

SUB-IFACE

$$\frac{\textbf{interface}\ I{<}\overline{X}{>}\,[Y\ \textbf{where}\,\overline{R}]\ \ldots \qquad R_i = Y\,\textbf{implements}\,K}{\Delta \vdash I{<}\overline{T}{>} \leq [\overline{T/X}]K}$$

SUB-IMPL

$$\frac{\Delta \Vdash T\,\textbf{implements}\,K}{\Delta \vdash T \leq K}$$

valid. Rule ENT-SUPER states that a constraint implies all superinterface constraints of its corresponding interface. The notation $[\overline{T/X}]$ denotes the capture-avoiding *type substitution* that replaces type variables X_i with types T_i. Metavariables φ and ψ range over type substitutions.

Rule ENT-IMPL defines how an implementation definition establishes validity of a constraint. Rule ENT-UP allows to promote a type on the left-hand side of an implementation constraint to a supertype, provided the corresponding implementing type does not occur in covariant positions of the interface (premise $n \in \mathsf{pol}^-(I)$). Rule ENT-IFACE is a kind of reflexivity rule. However, the rule only fires for interfaces without binary methods (premise $1 \in \mathsf{pol}^+(I)$) to ensure type soundness.

The subtyping relation is reflexive and transitive, and it allows *Object* as a supertype of every other type. A type variable X is a subtype of T if the type environment contains the constraint $X\,\mathbf{extends}\,T$. Moreover, a class type is a subtype of its direct superclass. Rule SUB-IFACE formulates subtyping on interface types in terms of superinterface constraints. The rule is only applicable to single-headed interfaces because only these interfaces may serve as types. Finally, rule SUB-IMPL integrates constraint entailment into the subtyping relation by deriving $\Delta \vdash T \leq K$ from $\Delta \Vdash T\,\mathbf{implements}\,K$.

3.4 Dynamic Semantics

This section presents a structural operational semantics [179] defining the run-time behavior of CoreGI programs.

3.4.1 Method Lookup

Figure 3.5 formalizes dynamic method lookup, relying on auxiliaries defined in Figure 3.4. The relation $\mathsf{getmdef}^{\mathsf{c}}(m, N)$ performs dynamic lookup of class method m on a receiver with run-time type N. If possible, it returns the definition of m directly contained in N (rule DYN-MDEF-CLASS-BASE). Otherwise, it continues the search in N's superclass (rule DYN-MDEF-CLASS-SUPER). The search stops when it reaches *Object* because there is no matching rule.

For non-static interface methods, $\mathsf{getmdef}^{\mathsf{i}}(m, N, \overline{N})$ performs lookup of a retroactively implemented method m on receiver type N and actual parameter types $\overline{N}$. For static interface methods, $\mathsf{getsmdef}(m, K, \overline{U})$ searches for method m in an implementation definition matching interface K and implementing types $\overline{U}$. The definitions of $\mathsf{getmdef}^{\mathsf{i}}$ and $\mathsf{getsmdef}$ require several auxiliaries from Figure 3.4:

- $N_1 \sqcup N_2 = M$ computes the least upper bound M of class types N_1 and N_2.
- $\bigsqcup \mathscr{N} = N$ computes the least upper bound N of a set $\mathscr{N}$ of class types. If $\mathscr{N}$ is not empty, then the least upper bound is unique and always exists.
- $\mathsf{resolve}_X(\overline{T}, \overline{N}) = N^?$ resolves implementing type X with respect to formal parameter types $\overline{T}$ and run-time parameter types $\overline{N}$ as the optional class type $N^?$.

  The definition of $\mathsf{resolve}$ constructs a set $\mathscr{N}$ containing those run-time parameter types N_i such that the ith formal parameter dispatches on X (i.e., $T_i = X$). If

Figure 3.4 Auxiliaries for dynamic method lookup.

$$\boxed{\mathsf{least\text{-}impl}\{\overline{(\varphi, \mathit{impl})}\} = (\varphi, \mathit{impl}) \qquad \mathsf{resolve}_X(\overline{T}, \overline{N}) = M^?}$$

LEAST-IMPL

$$\frac{\begin{array}{c}\mathit{impl}_i = \textbf{implementation}\texttt{<}\overline{X_i}\texttt{>}\ I\texttt{<}\overline{V_i}\texttt{>}\ [\,\overline{N_i}^{\,l}\,]\ \ldots \\ n \geq 1 \qquad (\forall i \in [n])\ \emptyset \vdash \varphi_k \overline{N_k} \leq \varphi_i \overline{N_i}\end{array}}{\mathsf{least\text{-}impl}\{(\varphi_1, \mathit{impl}_1), \ldots, (\varphi_n, \mathit{impl}_n)\} = (\varphi_k, \mathit{impl}_k)}$$

RESOLVE-NON-EMPTY

$$\frac{\mathscr{N} = \{N_i \mid i \in [n], T_i = X\} \neq \emptyset \qquad \bigsqcup \mathscr{N} = M}{\mathsf{resolve}_X(\overline{T}^n, \overline{N}^n) = M}$$

RESOLVE-EMPTY

$$\frac{\{N_i \mid i \in [n], T_i = X\} = \emptyset}{\mathsf{resolve}_X(\overline{T}^n, \overline{N}^n) = \mathsf{nil}}$$

$$\boxed{N_1 \sqcup N_2 = M \qquad \bigsqcup \mathscr{N} = N}$$

LUB-RIGHT

$$\frac{\emptyset \vdash N \leq M}{N \sqcup M = M}$$

LUB-LEFT

$$\frac{\emptyset \vdash M \leq N}{N \sqcup M = N}$$

LUB-SUPER

$$\frac{\begin{array}{c}\text{not } \emptyset \vdash C\texttt{<}\overline{T}\texttt{>} \leq N \qquad \text{not } \emptyset \vdash N \leq C\texttt{<}\overline{T}\texttt{>} \\ \textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } N' \ \ldots \qquad [\overline{T/X}]N' \sqcup N = M\end{array}}{C\texttt{<}\overline{T}\texttt{>} \sqcup N = M}$$

LUB-SET-SINGLE

$$\bigsqcup \{N\} = N$$

LUB-SET-MULTI

$$\frac{\mathscr{N} \neq \emptyset \qquad \bigsqcup \mathscr{N} = M' \qquad M' \sqcup N = M}{\bigsqcup (\mathscr{N} \mathbin{\dot\cup} \{N\}) = M}$$

the set $\mathscr{N}$ is not empty (rule RESOLVE-NON-EMPTY), the resolution of X is the least upper bound $\bigsqcup \mathscr{N}$. Otherwise (rule RESOLVE-EMPTY), X does not occur in the formal parameter types $\overline{T}$, so $\mathsf{resolve}$ returns nil. There is a restriction ensuring that the implementing type X does not occur nested inside one of the formal parameter types T_i (see well-formedness criterion WF-IFACE-3 in Section 3.5.3).

- $\mathsf{least\text{-}impl}\mathscr{M}$ computes the least element of a set $\mathscr{M}$ containing pairs of substitutions and implementations. The pair (φ, impl) is considered smaller than the pair $(\varphi', \mathit{impl}')$ if, and only if, the implementing types of impl under substitution φ are pointwise subtypes of the implementing types of impl' under substitution φ'.

  There are several well-formedness criteria ensuring that $\mathsf{least\text{-}impl}$ always finds a unique solution when invoked by $\mathsf{getmdef}^{\mathsf{i}}$ or $\mathsf{getsmdef}$. Section 2.3.4 already discussed these criteria (“no overlap”, “unique interface instantiation and non-dispatch types”, “downward closed”) informally; Section 3.5.3 defines them formally as well-formedness criteria WF-PROG-1, WF-PROG-2, and WF-PROG-3.

With these auxiliaries in place, rule DYN-MDEF-IFACE in Figure 3.5 defines the relation $\mathsf{getmdef}^{\mathsf{i}}(m, N, \overline{N})$ as follows:

Figure 3.5 Dynamic method lookup.

$$\boxed{\mathsf{getmdef}^{\mathsf{c}}(m, N) = \texttt{<}\overline{X}\texttt{>}\,\overline{T\,x} \rightarrow T \ \textbf{where}\ \overline{\mathcal{P}}\,\{e\}}$$

DYN-MDEF-CLASS-BASE

$$\frac{\textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\,\overline{T\,f}\ \overline{m : \mathit{mdef}}\,\}}{\mathsf{getmdef}^{\mathsf{c}}(m_j,\, C\texttt{<}\overline{U}\texttt{>}) = [\overline{U/X}]\mathit{mdef}_j}$$

DYN-MDEF-CLASS-SUPER

$$\frac{\begin{array}{c}\textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\,\overline{T\,f}\ \overline{m : \mathit{mdef}}\,\}\\ m \notin \overline{m} \qquad \mathsf{getmdef}^{\mathsf{c}}(m,\, [\overline{U/X}]N) = \texttt{<}\overline{X}\texttt{>}\,\overline{V\,x} \rightarrow V\ \textbf{where}\ \overline{\mathcal{P}}\,\{e\}\end{array}}{\mathsf{getmdef}^{\mathsf{c}}(m,\, C\texttt{<}\overline{U}\texttt{>}) = \texttt{<}\overline{X}\texttt{>}\,\overline{V\,x} \rightarrow V\ \textbf{where}\ \overline{\mathcal{P}}\,\{e\}}$$

$$\boxed{\mathsf{getmdef}^{\mathsf{i}}(m, N, \overline{N}) = \texttt{<}\overline{X}\texttt{>}\,\overline{T\,x} \rightarrow T \ \textbf{where}\ \overline{\mathcal{P}}\,\{e\}}$$

DYN-MDEF-IFACE

$$\frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}^{l}\,\textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{P}\,\{\ldots\ \overline{\mathit{rcsig}}\,\}\\ \mathit{rcsig}_j = \textbf{receiver}\,\{\overline{m : \mathit{msig}}\} \qquad \mathit{msig}_k = \texttt{<}\overline{Y}\texttt{>}\,\overline{T\,x} \rightarrow T\ \textbf{where}\ \overline{Q}\\ (\forall i \in [l], i \neq j)\ \mathsf{resolve}_{Z_i}(\overline{T}, \overline{N}) = M_i^{?} \qquad \mathsf{resolve}_{Z_j}(Z_j\overline{T}, N\overline{N}) = M_j^{?}\\ \mathsf{least\text{-}impl}\{([\overline{V/X}], \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{U}\texttt{>}\ [\,\overline{M'}\,]\ \ldots)\\ \mid (\forall i)\ M_i^{?} = \mathsf{nil}\ \text{or}\ \emptyset \vdash M_i^{?} \leq [\overline{V/X}]M_i'\}\\ = (\varphi, \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{U}\texttt{>}\ [\,\overline{M'}\,]\ \textbf{where}\ \overline{P'}\ \{\ldots\ \overline{\mathit{rcdef}}\,\})\\ \mathit{rcdef}_j = \textbf{receiver}\,\{\overline{\mathit{mdef}}\}\end{array}}{\mathsf{getmdef}^{\mathsf{i}}(m_k, N, \overline{N}^{n}) = \varphi\,\mathit{mdef}_k}$$

$$\boxed{\mathsf{getsmdef}(m, K, \overline{U}) = \texttt{<}\overline{X}\texttt{>}\,\overline{T\,x} \rightarrow T \ \textbf{where}\ \overline{\mathcal{P}}\,\{e\}}$$

DYN-MDEF-STATIC

$$\frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}\,\textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{Q}\,\{\,\overline{m : \textbf{static}\,\mathit{msig}}\ \ldots\}\\ \mathsf{least\text{-}impl}\{([\overline{V/X}], \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{W}\texttt{>}\ [\,\overline{N}^{l}\,]\ \ldots)\\ \mid (\forall i \in [l])\ \emptyset \vdash U_i \leq [\overline{V/X}]N_i\}\\ = (\varphi, \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{W}\texttt{>}\ [\,\overline{N}^{l}\,]\ \textbf{where}\ \overline{P}\ \{\,\overline{\textbf{static}\,\mathit{mdef}}\ldots\})\end{array}}{\mathsf{getsmdef}(m_k, I\texttt{<}\overline{T}\texttt{>}, \overline{U}^{l}) = \varphi\,\mathit{mdef}_k}$$

- First, $\mathsf{getmdef}^{\mathsf{i}}$ retrieves the interface I and the receiver $rcsig_j$ defining method m.
- Then, it uses resolve to compute, for each implementing type variable Z_i, an optional least upper bound $M_i^?$ of all argument types contributing to the resolution of the ith implementing type.
- Next, it collects all implementations of I whose implementing types are pointwise supertypes of the $M_i^?$s. (If $M_i^?$ is nil, then every type is considered a supertype of $M_i^?$ because the ith implementing type does not occur in m's signature.)
- Finally, $\mathsf{getmdef}^{\mathsf{i}}$ selects among all these implementations the one with least implementing types.

The definition of $\mathsf{getsmdef}(m, K, \overline{U})$ in rule DYN-MDEF-STATIC is similar to that of $\mathsf{getmdef}^{\mathsf{i}}$ but simpler: getsmdef does not need to resolve the implementing types but gets them explicitly through the types $\overline{U}$. Thus, getsmdef just uses least-impl to choose the least implementation among all implementation definitions matching K and $\overline{U}$.

3.4.2 Evaluation

The definition of CoreGI's dynamic semantics is now straightforward and given in Figure 3.6. Values (ranged over by v, w) and call-by-value evaluation contexts (denoted by $\mathcal{E}$) are defined in the obvious way. Unlike FGJ, CoreGI uses a call-by-value evaluation order to ensure deterministic evaluation. The notation $\mathcal{E}[e]$ denotes the replacement of $\mathcal{E}$'s hole $\Box$ with expression e.

The *top-level evaluation* relation $e \longmapsto e'$ reduces an expression e at the top level to e'. Rule DYN-FIELD deals with field accesses $\textbf{new}\, N(\overline{v}).f_i$. The auxiliary relation $\mathsf{fields}(N) = \overline{T\, f}$, also defined in Figure 3.6, returns the fields declared by the superclasses of N and N itself. CoreGI assumes that the ith constructor argument v_i corresponds to the field $T_i\, f_i$, so $\textbf{new}\, N(\overline{v}).f_i$ reduces to v_i. Rules DYN-INVOKE-CLASS, DYN-INVOKE-IFACE, and DYN-INVOKE-STATIC handle invocations of class methods, non-static interface methods, and static interface methods, respectively. The notation $[\overline{e/x}]$ denotes the capture-avoiding expression substitution that replaces expression variables x_i with expressions e_i. Among the rules DYN-INVOKE-CLASS and DYN-INVOKE-IFACE, at most one is applicable because the namespaces for class and interface methods are disjoint (see Convention 3.4). Finally, rule DYN-CAST allows casts from $\textbf{new}\, N(\overline{v})$ to type T if N is a subtype of T.

The *proper evaluation* relation $e \longrightarrow e'$ reduces an expression e to e' by using a suitable evaluation context $\mathcal{E}$ together with the top-level evaluation relation $\longmapsto$.

Remark. Several places in the definition of the dynamic semantics rely on CoreGI's subtyping relation. Except for the premise of rule DYN-CAST in Figure 3.6, all uses of the subtyping relation have the form $\emptyset \vdash T \leq N$; that is, the type environment is empty and only class types appear as possible supertypes. In these cases, the full subtyping relation is not needed; instead, plain inheritance between classes and an additional rule covering the case $N = \mathit{Object}$ suffices.[1]

[1] The definition of inheritance between classes is standard. See Figure 3.16 on page 52 for a formal definition.

Figure 3.6 Dynamic semantics.

Values and evaluation contexts

$$
\begin{aligned}
v, w &::= \mathbf{new}\, N(\overline{v}) \\
\mathcal{E} &::= \square \mid \mathcal{E}.f \mid \mathcal{E}.m\texttt{<}\overline{T}\texttt{>}(\overline{e}) \mid v.m\texttt{<}\overline{T}\texttt{>}(\overline{v}, \mathcal{E}, \overline{e}) \\
&\quad \mid K[\overline{T}].m\texttt{<}\overline{T}\texttt{>}(\overline{v}, \mathcal{E}, \overline{e}) \mid \mathbf{new}\, N(\overline{v}, \mathcal{E}, \overline{e}) \mid (T)\, \mathcal{E}
\end{aligned}
$$

Top-level evaluation: $e \longmapsto e'$

$$
\textsc{dyn-field}\quad \frac{\mathsf{fields}(N) = \overline{T\, f}}{\mathbf{new}\, N(\overline{v}).f_i \longmapsto v_i}
\qquad
\textsc{dyn-invoke-class}\quad \frac{v = \mathbf{new}\, N(\overline{w}) \qquad \mathsf{getmdef}^{\mathrm{c}}(m^{\mathrm{c}}, N) = \texttt{<}\overline{X}\texttt{>}\, \overline{T\, x} \to T \;\mathbf{where}\; \overline{\mathcal{P}}\, \{e\}}{v.m^{\mathrm{c}}\texttt{<}\overline{U}\texttt{>}(\overline{v}) \longmapsto [v/\mathit{this}, \overline{v/x}][\overline{U/X}]e}
$$

$$
\textsc{dyn-invoke-iface}\quad \frac{(\forall i \in \{0, \ldots, n\})\ v_i = \mathbf{new}\, N_i(\overline{w_i}) \qquad \mathsf{getmdef}^{\mathrm{i}}(m^{\mathrm{i}}, N_0, \overline{N}) = \texttt{<}\overline{X}\texttt{>}\, \overline{T\, x} \to T \;\mathbf{where}\; \overline{\mathcal{P}}\, \{e\}}{v_0.m^{\mathrm{i}}\texttt{<}\overline{U}\texttt{>}(\overline{v}^n) \longmapsto [v_0/\mathit{this}, \overline{v/x}][\overline{U/X}]e}
$$

$$
\textsc{dyn-invoke-static}\quad \frac{\mathsf{getsmdef}(m, K, \overline{U}) = \texttt{<}\overline{X}\texttt{>}\, \overline{T\, x} \to T \;\mathbf{where}\; \overline{\mathcal{P}}\, \{e\}}{K[\overline{U}].m\texttt{<}\overline{V}\texttt{>}(\overline{v}) \longmapsto [\overline{v/x}][\overline{V/X}]e}
\qquad
\textsc{dyn-cast}\quad \frac{\emptyset \vdash N \leq T}{(T)\, \mathbf{new}\, N(\overline{v}) \longmapsto \mathbf{new}\, N(\overline{v})}
$$

Proper evaluation: $e \longrightarrow e'$

$$
\textsc{dyn-context}\quad \frac{e \longmapsto e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}
$$

$\mathsf{fields}(N) = \overline{T\, f}$

$$
\textsc{fields-object}\quad \mathsf{fields}(\mathit{Object}) = \bullet
$$

$$
\textsc{fields-class}\quad \frac{\mathbf{class}\; C\texttt{<}\overline{X}\texttt{>}\; \mathbf{extends}\; N \;\mathbf{where}\; \overline{P}\, \{\, \overline{T\, f} \ldots \} \qquad \mathsf{fields}([\overline{U/X}]N) = \overline{T'\, f'}}{\mathsf{fields}(C\texttt{<}\overline{U}\texttt{>}) = \overline{T'\, f'}, [\overline{U/X}]\overline{T\, f}}
$$

Figure 3.7 Well-formedness of types and constraints.

$$\boxed{\Delta \vdash T\ \mathsf{ok}}$$

$$\textsc{ok-tvar}\quad \frac{X \in \mathsf{dom}(\Delta)}{\Delta \vdash X\ \mathsf{ok}} \qquad\qquad \textsc{ok-object}\quad \Delta \vdash \mathit{Object}\ \mathsf{ok}$$

$$\textsc{ok-class}\quad \frac{\textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\ \ldots \qquad \Delta \vdash \overline{T}\ \mathsf{ok} \qquad \Delta \Vdash [\overline{T/X}]\overline{P}}{\Delta \vdash C\texttt{<}\overline{T}\texttt{>}\ \mathsf{ok}}$$

$$\textsc{ok-iface}\quad \frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\ [Y\ \textbf{where}\ \overline{R}]\ \textbf{where}\ \overline{P}\ \ldots \\ \Delta \vdash \overline{T}\ \mathsf{ok} \qquad Y \notin \mathsf{ftv}(\overline{T}, \Delta) \qquad \Delta, Y\ \textbf{implements}\ I\texttt{<}\overline{T}\texttt{>} \Vdash [\overline{T/X}](\overline{R}, \overline{P})\end{array}}{\Delta \vdash I\texttt{<}\overline{T}\texttt{>}\ \mathsf{ok}}$$

$$\boxed{\Delta \vdash \mathcal{P}\ \mathsf{ok}}$$

$$\textsc{ok-impl-constr}\quad \frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\ [\overline{Y}\ \textbf{where}\ \overline{R}]\ \textbf{where}\ \overline{P}\ \ldots \\ \Delta \vdash \overline{T}, \overline{U}\ \mathsf{ok} \qquad \Delta \Vdash [\overline{U/X}, \overline{T/Y}](\overline{R}, \overline{P})\end{array}}{\Delta \vdash \overline{T}\ \textbf{implements}\ I\texttt{<}\overline{U}\texttt{>}\ \mathsf{ok}} \qquad \textsc{ok-ext-constr}\quad \frac{\Delta \vdash T, U\ \mathsf{ok}}{\Delta \vdash T\ \textbf{extends}\ U\ \mathsf{ok}}$$

3.5 Static Semantics

This section presents a declarative specification of CoreGI's type system. We defer the definition of a typechecking algorithm until Section 3.7.

All types and constraints occurring in a type-correct CoreGI program must be well-formed. Formally, a type T or constraint $\mathcal{P}$ is well-formed under type environment Δ if, and only if, $\Delta \vdash T\ \mathsf{ok}$ or $\Delta \vdash \mathcal{P}\ \mathsf{ok}$, respectively, holds (see Figure 3.7). Often $\Delta \vdash \overline{T}\ \mathsf{ok}$ and $\Delta \vdash \overline{\mathcal{P}}\ \mathsf{ok}$ abbreviate $(\forall i)\ \Delta \vdash T_i\ \mathsf{ok}$ and $(\forall i)\ \Delta \vdash \mathcal{P}_i\ \mathsf{ok}$, respectively. Rule OK-IFACE in Figure 3.7 ensures that only single-headed interfaces form interface types. Well-formedness of a constraint $\overline{T}\ \textbf{implements}\ I\texttt{<}\overline{U}\texttt{>}$ (rule OK-IMPL-CONSTR) not only demands that $\overline{T}, \overline{U}$ are well-formed but also that the constraints of the interface I are fulfilled.

The relation $\mathsf{mtype}_\Delta(m, T)$, defined in Figure 3.8, looks up the signature of method m for receiver type T. Rule MTYPE-CLASS handles class methods m^{c}. Unlike the corresponding rule for FGJ, lookup of class methods does not ascend the inheritance hierarchy of classes because CoreGI's typing rules (explained shortly) allow subsumption on the receiver. Rule MTYPE-IFACE handles interface methods m^{i} by searching the interface and the receiver defining the method and asserting validity of the corresponding implementation constraint, possibly "guessing" the types $\overline{V}$ and some of the types $\overline{T}$. Figure 3.8 also

Figure 3.8 Method typing.

$$\boxed{\mathsf{mtype}_\Delta(m, T) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \mathcal{P}}$$

MTYPE-CLASS

$$\frac{\textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\ldots\ \overline{m : \mathit{msig}\,\{e\}}\,\}}{\mathsf{mtype}_\Delta(m^{\mathrm{c}}_j, C\texttt{<}\overline{T}\texttt{>}) = [\overline{T/X}]\mathit{msig}_j}$$

MTYPE-IFACE

$$\frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y}\,\textbf{where}\,\overline{R}]\ \textbf{where}\ \overline{P}\,\{\ldots\ \overline{\mathit{rcsig}}\,\}\\ \mathit{rcsig}_j = \textbf{receiver}\,\{\overline{m : \mathit{msig}}\} \qquad \Delta \Vdash \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\end{array}}{\mathsf{mtype}_\Delta(m^{\mathrm{i}}_k, T_j) = [\overline{V/X}, \overline{T/Y}]\mathit{msig}_k}$$

$$\boxed{\mathsf{smtype}_\Delta(m, K[\overline{T}]) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \mathcal{P}}$$

MTYPE-STATIC

$$\frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y}\,\textbf{where}\,\overline{R}]\ \textbf{where}\ \overline{P}\,\{\,\overline{m : \textbf{static}\,\mathit{msig}}\ \ldots\}\\ \Delta \Vdash \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{U}\texttt{>}\end{array}}{\mathsf{smtype}_\Delta(m^{\mathrm{i}}_k, I\texttt{<}\overline{U}\texttt{>}[\overline{T}]) = [\overline{U/X}, \overline{T/Y}]\mathit{msig}_k}$$

defines the relation $\mathsf{smtype}_\Delta(m, I\texttt{<}\overline{U}\texttt{>}[\overline{T}])$, which looks up the signature of static method m defined in interface I under type parameters $\overline{U}$ and implementing types $\overline{T}$.

3.5.1 Expression Typing

Expression typing, written $\Delta;\Gamma \vdash e : T$, states that under type environment Δ and variable environment Γ, expression e has type T. Variable environments Γ are defined as follows:

Definition 3.8 (Variable environment). A *variable environment* Γ is a finite mapping from variables x to types T. The notation $\Gamma, x : T$ extends Γ with a mapping from x to T assuming x is not already bound in Γ. The notation $\Gamma(x)$ denotes the type T such that Γ maps x to T. It assumes that Γ contains such a binding for x.

Figure 3.9 defines the expression typing judgment. Typechecking a field access $e.f_j$ looks up the type of field f_j in the fields declared by C (rule EXP-FIELD). There is no need to search the superclasses of C for a definition of f_j because rule EXP-SUBSUME allows lifting the type of e to some supertype. Thanks to mtype and smtype from Figure 3.8, typechecking method invocations is straightforward (rules EXP-INVOKE and EXP-INVOKE-STATIC).

Rule EXP-NEW handles an object allocation $\textbf{new}\,N(\overline{e})$ by asserting that N is well-formed and by checking that the ith argument e_i is type correct with respect to the type of the ith field declaration returned by $\mathsf{fields}(N)$. The auxiliary $\mathsf{fields}(N) = \overline{T\,f}$, already defined in Figure 3.6, computes the fields declared by the superclasses of N and

Figure 3.9 Expression typing.

$$\boxed{\Delta; \Gamma \vdash e : T}$$

EXP-VAR

$$\Delta; \Gamma \vdash x : \Gamma(x)$$

EXP-FIELD

$$\frac{\Delta; \Gamma \vdash e : C{<}\overline{T}{>} \qquad \textbf{class } C{<}\overline{X}{>} \textbf{ extends } N \textbf{ where } \overline{P}\,\{\,\overline{U\,f}\ldots\}}{\Delta; \Gamma \vdash e.f_j : [\overline{T/X}]U_j}$$

EXP-INVOKE

$$\frac{\Delta; \Gamma \vdash e : T \qquad \mathsf{mtype}_{\Delta}(m, T) = {<}\overline{X}{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}} \\ (\forall i)\ \Delta; \Gamma \vdash e_i : [\overline{V/X}]U_i \qquad \Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}} \qquad \Delta \vdash \overline{V}\ \mathsf{ok}}{\Delta; \Gamma \vdash e.m{<}\overline{V}{>}(\overline{e}) : [\overline{V/X}]U}$$

EXP-INVOKE-STATIC

$$\frac{\mathsf{smtype}_{\Delta}(m, I{<}\overline{W}{>}[\overline{T}]) = {<}\overline{X}{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}} \\ (\forall i)\ \Delta; \Gamma \vdash e_i : [\overline{V/X}]U_i \qquad \Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}} \qquad \Delta \vdash \overline{T}, \overline{V}\ \mathsf{ok}}{\Delta; \Gamma \vdash I{<}\overline{W}{>}[\overline{T}].m{<}\overline{V}{>}(\overline{e}) : [\overline{V/X}]U}$$

EXP-NEW

$$\frac{\Delta \vdash N\ \mathsf{ok} \qquad \mathsf{fields}(N) = \overline{T\,f} \qquad (\forall i)\ \Delta; \Gamma \vdash e_i : T_i}{\Delta; \Gamma \vdash \textbf{new } N(\overline{e}) : N}$$

EXP-CAST

$$\frac{\Delta \vdash T\ \mathsf{ok} \qquad \Delta; \Gamma \vdash e : U}{\Delta; \Gamma \vdash (T)\, e : T}$$

EXP-SUBSUME

$$\frac{\Delta; \Gamma \vdash e : U \qquad \Delta \vdash U \leq T}{\Delta; \Gamma \vdash e : T}$$

N itself. Unlike FGJ, which has three rules for cast expressions to differ between upcasts, downcasts, and stupid casts, CoreGI uses a single rule for casts because they are not in the focus of the formalization.

3.5.2 Program Typing

Figures 3.10 and 3.11 specify the well-formedness rules for definitions and programs, including several auxiliary relations.

- The relation $\Delta \vdash \mathit{msig} \leq \mathit{msig}'$ extends subtyping to method signatures by treating argument types invariantly and return types covariantly (Figure 3.10).
- The relation $\mathsf{override\text{-}ok}_{\Delta}(m : \mathit{msig}, N)$ asserts that class type N correctly overrides method m with signature msig (Figure 3.10).
- The relations $\Delta \vdash \mathit{msig}\ \mathsf{ok}$, $\Delta \vdash \mathit{mdef}\ \mathsf{ok}$, and $\Delta \vdash \mathit{rcsig}\ \mathsf{ok}$ assert well-formedness of method signatures, method definitions, and receiver signatures, respectively (Figure 3.10).

Figure 3.10 Auxiliaries for well-formedness of definitions.

$$\boxed{\Delta \vdash \mathit{msig} \le \mathit{msig}' \qquad \mathsf{override\text{-}ok}_{\Delta}(m : \mathit{msig}, N)}$$

SUB-MSIG

$$\frac{\Delta, \overline{P} \vdash T \le T'}{\Delta \vdash {<}\overline{X}{>}\,\overline{T\,x} \to T\ \textbf{where}\ \overline{P} \le {<}\overline{X}{>}\,\overline{T\,x} \to T'\ \textbf{where}\ \overline{P}}$$

OK-OVERRIDE

$$\frac{(\forall N')\ \text{if}\ \Delta \vdash N \le N'\ \text{and}\ \mathsf{mtype}_{\Delta}(m, N') = \mathit{msig}'\ \text{then}\ \Delta \vdash \mathit{msig} \le \mathit{msig}'}{\mathsf{override\text{-}ok}_{\Delta}(m : \mathit{msig}, N)}$$

$$\boxed{\Delta \vdash \mathit{msig}\ \mathsf{ok} \qquad \Delta; \Gamma \vdash \mathit{mdef}\ \mathsf{ok} \qquad \Delta \vdash \mathit{rcsig}\ \mathsf{ok} \qquad \Delta \vdash m : \mathit{mdef}\ \mathsf{ok\,in}\ N}$$

OK-MSIG

$$\frac{\Delta, \overline{P}, \overline{X} \vdash \overline{T}, U, \overline{P}\ \mathsf{ok}}{\Delta \vdash {<}\overline{X}{>}\,\overline{T\,x} \to U\ \textbf{where}\ \overline{P}\ \mathsf{ok}}$$

OK-MDEF

$$\frac{\Delta \vdash {<}\overline{X}{>}\,\overline{T\,x} \to U\ \textbf{where}\ \overline{P}\ \mathsf{ok} \qquad \Delta, \overline{P}, \overline{X}; \Gamma, \overline{x : T} \vdash e : U}{\Delta; \Gamma \vdash {<}\overline{X}{>}\,\overline{T\,x} \to U\ \textbf{where}\ \overline{P}\,\{e\}\ \mathsf{ok}}$$

OK-RCSIG

$$\frac{(\forall i)\ \Delta \vdash \mathit{msig}_i\ \mathsf{ok}}{\Delta \vdash \textbf{receiver}\,\{\overline{m : \mathit{msig}}\}\ \mathsf{ok}}$$

OK-MDEF-IN-CLASS

$$\frac{\Delta; \mathit{this} : N \vdash \mathit{msig}\,\{e\}\ \mathsf{ok} \qquad \mathsf{override\text{-}ok}_{\Delta}(m : \mathit{msig}, N)}{\Delta \vdash m : \mathit{msig}\,\{e\}\ \mathsf{ok\,in}\ N}$$

$$\boxed{\Delta \vdash \mathit{mdef}\ \mathsf{implements}\ \mathit{msig} \qquad \Delta \vdash \mathit{rcdef}\ \mathsf{implements}\ \mathit{rcsig}}$$

IMPL-METH

$$\frac{\Delta; \Gamma \vdash \mathit{msig}\,\{e\}\ \mathsf{ok} \qquad \Delta \vdash \mathit{msig} \le \mathit{msig}'}{\Delta; \Gamma \vdash \mathit{msig}\,\{e\}\ \mathsf{implements}\ \mathit{msig}'}$$

IMPL-RECV

$$\frac{(\forall i)\ \Delta; \Gamma \vdash \mathit{mdef}_i\ \mathsf{implements}\ \mathit{msig}_i}{\Delta; \Gamma \vdash \textbf{receiver}\,\{\overline{\mathit{mdef}}\}\ \mathsf{implements}\ \textbf{receiver}\,\{\overline{m : \mathit{msig}}\}}$$

Figure 3.11 Well-formedness of definitions and programs.

$$\boxed{\vdash \mathit{cdef}\ \mathsf{ok} \qquad \vdash \mathit{idef}\ \mathsf{ok} \qquad \vdash \mathit{impl}\ \mathsf{ok}}$$

OK-CDEF

$$\frac{\overline{P},\overline{X} \vdash N, \overline{P}, \overline{T}\ \mathsf{ok} \qquad (\forall i)\ \overline{P},\overline{X} \vdash m_i : \mathit{mdef}_i\ \mathsf{ok\,in}\ C\texttt{<}\overline{X}\texttt{>}}{\vdash \mathbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \mathbf{extends}\ N\ \mathbf{where}\ \overline{P}\,\{\,\overline{T\ f}\ \overline{m : \mathit{mdef}}\,\}\ \mathsf{ok}}$$

OK-IDEF

$$\frac{\overline{R}, \overline{P}, \overline{X}, \overline{Y} \vdash \overline{R}, \overline{P}, \overline{\mathit{msig}}, \overline{\mathit{rcsig}}\ \mathsf{ok}}{\vdash \mathbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\,\overline{Y}\ \mathbf{where}\ \overline{R}\,]\ \mathbf{where}\ \overline{P}\,\{\,\overline{m : \mathbf{static}\ \mathit{msig}}\ \overline{\mathit{rcsig}}\,\}\ \mathsf{ok}}$$

OK-IMPL

$$\frac{\begin{array}{c} \overline{P}, \overline{X} \vdash \overline{N}\ \mathbf{implements}\ I\texttt{<}\overline{T}\texttt{>}, \overline{P}\ \mathsf{ok} \\ \mathbf{interface}\ I\texttt{<}\overline{Y}\texttt{>}\,[\,\overline{Z}\ \mathbf{where}\ \overline{R}\,]\ \mathbf{where}\ \overline{Q}\,\{\,\overline{m : \mathbf{static}\ \mathit{msig}}\ \overline{\mathit{rcsig}}\,\} \\ (\forall i)\ \overline{P}, \overline{X}; \emptyset \vdash \mathit{mdef}_i\ \mathsf{implements}\ [\overline{T/Y}, \overline{N/Z}]\mathit{msig}_i \\ (\forall i)\ \overline{P}, \overline{X}; \mathit{this} : N_i \vdash \mathit{rcdef}_i\ \mathsf{implements}\ [\overline{T/Y}, \overline{N/Z}]\mathit{rcsig}_i \end{array}}{\vdash \mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \{\,\overline{\mathbf{static}\ \mathit{mdef}}\ \overline{\mathit{rcdef}}\,\}\ \mathsf{ok}}$$

$$\boxed{\vdash \mathit{prog}\ \mathsf{ok}}$$

OK-PROG

$$\frac{\begin{array}{c} \vdash \overline{\mathit{def}}\ \mathsf{ok} \qquad \emptyset; \emptyset \vdash e : T \\ \text{additional well-formedness criteria from Section 3.5.3 hold} \end{array}}{\vdash \overline{\mathit{def}}\ e\ \mathsf{ok}}$$

- The relation $\Delta \vdash m : \mathit{mdef}\ \mathsf{ok\,in}\ N$ asserts that the definition *mdef* of method m in class N is well-formed (Figure 3.10).

- The relation $\Delta \vdash \mathit{mdef}\ \mathsf{implements}\ \mathit{msig}$ asserts that method definition *mdef* is a valid implementation of signature *msig* (Figure 3.10).

- The relation $\Delta \vdash \mathit{rcdef}\ \mathsf{implements}\ \mathit{rcsig}$ asserts that receiver definition *rcdef* properly implements all methods from receiver signature *rcsig* (Figure 3.10). As already discussed in Section 3.2, methods in receiver definitions are matched by position against methods in receiver signatures.

- The relations $\vdash \mathit{cdef}\ \mathsf{ok}$, $\vdash \mathit{idef}\ \mathsf{ok}$, and $\vdash \mathit{impl}\ \mathsf{ok}$ assert well-formedness of class, interface, and implementation definitions, respectively (Figure 3.11).

- The relation $\vdash \mathit{prog}\ \mathsf{ok}$ asserts well-formedness of programs (Figure 3.11). Well-formedness of programs requires several additional well-formedness criteria. For the full JavaGI language, Section 2.3.4 already discussed the most important of them informally. The next section gives the complete list of additional well-formedness criteria for CoreGI.

3.5.3 Additional Well-Formedness Criteria

The additional well-formedness criteria for CoreGI are divided into criteria that apply to classes, interfaces, implementations, whole programs, and type environments.

Criteria for Classes

For each class

$$\textbf{class}\ C{<}\overline{X}{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\,\overline{T\,f}^{n}\ \overline{m : mdef}^{l}\,\}$$

the following well-formedness criteria must hold:

WF-CLASS-1 The field names, including names of inherited fields, are pairwise disjoint. That is, $i \neq j \in [n]$ implies $f_i \neq f_j$ and $\mathsf{fields}(N) = \overline{U\,g}$ implies $\overline{f} \cap \overline{g} = \emptyset$.

WF-CLASS-2 The method names $\overline{m}$ are pairwise disjoint. That is, $i \neq j \in [l]$ implies $m_i \neq m_j$.

Criterion WF-CLASS-1 states that CoreGI does not support field shadowing, whereas WF-CLASS-2 rules out method overloading (together with rule OK-OVERRIDE from Figure 3.11). Both restrictions are not present in the full JavaGI language.

Criteria for Interfaces

The predicate $\mathsf{at\text{-}top}(\overline{X}, T)$ ensures that each of the type variables $\overline{X}$ occur only at the top level of type T.

Definition 3.9. $\mathsf{at\text{-}top}(\overline{X}, T)$ holds if, and only if, $\overline{X} \cap \mathsf{ftv}(T) = \emptyset$ or $T \in \overline{X}$.

The well-formedness criteria for interfaces then require that for each interface

$$\textbf{interface}\ I{<}\overline{X}{>}\,[\,\overline{Y}\,\textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{P}\,\{\,\overline{m : \textbf{static}\,msig}\ \ \overline{rcsig}\,\}$$

the following conditions must hold:

WF-IFACE-1 The names $\overline{m}$ of static methods are pairwise disjoint.

WF-IFACE-2 In all superinterface constraints $\overline{G}\,\textbf{implements}\,K \in \overline{R}$, the implementing types $\overline{Y}$ do not occur in K and the types $\overline{G}$ are pairwise distinct type variables from $\overline{Y}$; that is, $\overline{Y} \cap \mathsf{ftv}(K) = \emptyset$ and $\overline{G} \subseteq \overline{Y}$ and $G_i \neq G_j$ for $i \neq j$.

WF-IFACE-3 In all method signatures ${<}\overline{Z}{>}\,\overline{T\,x} \to U$ **where** $\overline{Q}$ contained in $\overline{rcsig}$, the implementing types $\overline{Y}$ may occur only at the top level of $\overline{T}$ and U, and they do not appear in $\overline{Q}$; that is, $\mathsf{at\text{-}top}(\overline{Y}, T_i)$ for all i and $\mathsf{at\text{-}top}(\overline{Y}, U)$ and $\mathsf{ftv}(\overline{Q}) \cap \overline{Y} = \emptyset$.

Criterion WF-IFACE-1 prevents overloading of static interface methods. (It is not necessary to include inherited method in this check because invocations of static interface methods are always qualified with the interface defining the method.) The full JavaGI language does not have this restriction. Criterion WF-IFACE-2 restricts the form of superinterface constraints to simplify the superinterface relation.

Figure 3.12 Illegal CoreGI program (implementing type nested in result position).

```
class C<X> {}
class A {}
class B extends A {}
interface I [X] {
  receiver {m : • → C<X>} // illegal
}
implementation I [A] {
  receiver {
    • → C<A> {new C<A>()}
  }
}
implementation I [B] {
  receiver {
    • → C<B> {new C<B>()}
  }
}
new B().m() // has either type C<B> or C<A>
```

The last criterion WF-IFACE-3 limits implementing types in method signatures to appear only at the top level of the result and argument types. Allowing implementing types to occur nested inside argument types would make it impossible to implement method dispatch under Java's type erasure semantics [26]. Nested occurrences of implementing types in result positions would cause loss of minimal types, as shown by the program in Figure 3.12. The expression **new** $B().m()$ would have either type C<B> (when typing **new** $B()$ as B) or type C<A> (when typing **new** $B()$ as A), but subtyping does not relate these two types. Last not least, implementing types in constraints of method signatures would cause unsoundness. Consider the program in Figure 3.13. It is type correct (apart from the constraint X **implements** J on method m_I of interface I) but gets stuck at run time:

- **new** $B().m_I()$ reduces to **new** $B().m_J().\mathit{break}()$ because $\mathsf{getmdef}^{\mathsf{i}}(B, m_I, \bullet)$ selects the definition of m_I from **implementation** $I\,[B]$.

- **new** $B().m_J().\mathit{break}()$ reduces to **new** $A().\mathit{break}()$ but class A does not provide method *break*. Hence, evaluation gets stuck.

Criteria for Implementations

The specification of the well-formedness criteria for implementation definitions requires the introduction of an alternative formulation of constraint entailment and subtyping. This alternative formulation is called *quasi algorithmic* because it constitutes a first step towards an algorithm for checking constraint entailment and subtyping.

Figure 3.13 Illegal CoreGI program (implementing type in method constraint).

```
class A {}
class B extends A {
  break : • → Object {new Object()}
}
interface J [X] {
  receiver {m_J : • → X}
}
implementation J [A] {
  receiver {
    • → A {new A()}
  }
}
interface I [X] {
  receiver {
    m_I : • → Object where X implements J // illegal
  }
}
implementation I [A] {
  receiver {
    • → Object where A implements J {new Object()}
  }
}
implementation I [B] {
  receiver {
    • → Object where B implements J {
      // with local constraint B implements J, this.m_J() has type B
      this.m_J().break()
    }
  }
}
new B().m_I() // typechecks by assigning type A to the expression new B()
```

Figure 3.14 Illegal CoreGI program (misses an implementation of I for C).

```
class C extends Object {}
interface I [X] {
  receiver {m : • → Object}
}
interface J [X where X implements I] {}
implementation J [C] {}
new C().m()
```

Quasi-algorithmic constraint entailment is needed to ensure that an implementation of some interface comes with appropriate implementations for all its superinterfaces. As an example, consider the program in Figure 3.14. It fails at run time because there is no implementation of interface I for class C that could provide the code for m, so the expression **new** $C().m()$ is stuck. However, the typing rules for expressions (Figure 3.9) accept the expression **new** $C().m()$ because the constraint C **implements** I holds by rules ENT-SUPER and ENT-IMPL from Figure 3.3. The root of the problem is that there exists an implementation of interface J for class C without a suitable implementation of J's superinterface I.

A failed attempt to deal with the problem for the program in Figure 3.14 is to require the following condition:

> WF-IMPL-1-INFORMAL-WRONG
>
> "For every **implementation**<$\overline{X}$> J [N] **where** $\overline{P}$ ... the corresponding superinterface constraint N **implements** I must hold under type environment $\overline{P}$."

But $\overline{P} \Vdash N$ **implements** I *always* holds by rule ENT-SUPER because rules ENT-IMPL and ENT-ENV allow us to derive $\overline{P} \Vdash N$ **implements** J.

A similar problem arises with Haskell type classes when checking that suitable instance definitions for all superclasses of a given type class exist [238].[2] In the context of Haskell, Sulzmann [209] suggests a restricted form of constraint entailment to check for superclass instances.

We follow Sulzmann's approach and use quasi-algorithmic constraint entailment to check for appropriate implementations of superinterfaces. It is an open question whether it is possible to use the declarative form of constraint entailment instead. Figure 3.15 and Figure 3.16 define quasi-algorithmic constraint entailment and subtyping, respectively, together with several auxiliary relations.

- Quasi-algorithmic constraint entailment, written $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$, asserts validity of constraint $\mathcal{P}$ under type environment Δ. Section 3.6.1 shows that the quasi-algorithmic and the declarative version of constraint entailment are equivalent.

  The idea of quasi-algorithmic entailment is to restrict derivations of declarative entailment (Figure 3.3) such that consecutive applications of rule ENT-UP are merged into an application of a single rule, and that rule ENT-SUPER is applied only to constraints originally established by rule ENT-ENV or rule ENT-IFACE. In Figure 3.15, rule ENT-Q-ALG-UP mimics consecutive applications of rule ENT-UP: it establishes validity of a constraint $\overline{T}$ **implements** I<$\overline{V}$> by first lifting all T_j pointwise to supertypes U_j, thereby respecting j's polarity in I, and then solving the resulting constraint $\overline{U}$ **implements** I<$\overline{V}$>.

- The *kernel of quasi-algorithmic entailment*, written $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{P}$, is a subset of the quasi-algorithmic entailment relation. Rule ENT-Q-ALG-ENV simulates an application of

[2] Haskell's type classes and instance definitions are the analogon to JavaGI's generalized interfaces and implementation definitions, respectively (see Section 8.1).

Figure 3.15 Quasi-algorithmic constraint entailment.

$\boxed{\Delta \Vdash_{\mathrm{q}} \mathcal{P}}$

ENT-Q-ALG-EXTENDS

$$\frac{\Delta \vdash_{\mathrm{q}} T \le U}{\Delta \Vdash_{\mathrm{q}} T\,\textbf{extends}\,U}$$

ENT-Q-ALG-UP

$$\frac{(\forall i)\ \Delta \vdash_{\mathrm{q}}' T_i \le U_i \qquad (\forall i)\ \text{if } T_i \ne U_i \text{ then } i \in \mathsf{pol}^-(I) \qquad \Delta \Vdash_{\mathrm{q}}' \overline{U}\,\textbf{implements}\,I{<}\overline{V}{>}}{\Delta \Vdash_{\mathrm{q}} \overline{T}\,\textbf{implements}\,I{<}\overline{V}{>}}$$

$\boxed{\Delta \Vdash_{\mathrm{q}}' \mathcal{R}}$

ENT-Q-ALG-ENV

$$\frac{S \in \Delta \qquad R \in \mathsf{sup}(S)}{\Delta \Vdash_{\mathrm{q}}' R}$$

ENT-Q-ALG-IMPL

$$\frac{\textbf{implementation}{<}\overline{X}{>}\ I{<}\overline{T}{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots \qquad \Delta \Vdash_{\mathrm{q}} [\overline{U/X}]\overline{P}}{\Delta \Vdash_{\mathrm{q}}' [\overline{U/X}](\overline{N}\,\textbf{implements}\,I{<}\overline{T}{>})}$$

ENT-Q-ALG-IFACE

$$\frac{1 \in \mathsf{pol}^+(I) \qquad I{<}\overline{V}{>} \trianglelefteq_{\mathrm{i}} K \qquad \mathsf{non\text{-}static}(I)}{\Delta \Vdash_{\mathrm{q}}' I{<}\overline{V}{>}\,\textbf{implements}\,K}$$

$\boxed{\mathcal{R} \in \mathsf{sup}(\mathcal{S})}$

SUP-REFL

$$\mathcal{R} \in \mathsf{sup}(\mathcal{R})$$

SUP-STEP

$$\frac{\textbf{interface}\ I{<}\overline{X}{>}\ [\overline{Y}\ \textbf{where}\ \overline{S}]\ \ldots \qquad \overline{U}\,\textbf{implements}\,I{<}\overline{V}{>} \in \mathsf{sup}(\mathcal{R})}{[\overline{V/X},\overline{U/Y}]S_j \in \mathsf{sup}(\mathcal{R})}$$

Figure 3.16 Inheritance and quasi-algorithmic subtyping.

$\boxed{N \trianglelefteq_{\mathrm{c}} M}$

INH-CLASS-REFL

$$N \trianglelefteq_{\mathrm{c}} N$$

INH-CLASS-SUPER

$$\frac{\textbf{class } C{<}\overline{X}{>} \textbf{ extends } M \ldots \qquad [\overline{T/X}]M \trianglelefteq_{\mathrm{c}} N}{C{<}\overline{T}{>} \trianglelefteq_{\mathrm{c}} N}$$

$\boxed{K \trianglelefteq_{\mathrm{i}} L}$

INH-IFACE-REFL

$$K \trianglelefteq_{\mathrm{i}} K$$

INH-IFACE-SUPER

$$\frac{\textbf{interface } I{<}\overline{X}{>}\, [Y \textbf{ where } \overline{R}] \ \ldots \quad R_i = Y \textbf{ implements } K \quad [\overline{T/X}]K \trianglelefteq_{\mathrm{i}} L}{I{<}\overline{T}{>} \trianglelefteq_{\mathrm{i}} L}$$

$\boxed{\Delta \vdash_{\mathrm{q}} T \leq U}$

SUB-Q-ALG-KERNEL

$$\frac{\Delta \vdash_{\mathrm{q}}' T \leq U}{\Delta \vdash_{\mathrm{q}} T \leq U}$$

SUB-Q-ALG-IMPL

$$\frac{\Delta \vdash_{\mathrm{q}}' T \leq U \qquad \Delta \Vdash_{\mathrm{q}}' U \textbf{ implements } K}{\Delta \vdash_{\mathrm{q}} T \leq K}$$

$\boxed{\Delta \vdash_{\mathrm{q}}' T \leq U}$

SUB-Q-ALG-VAR-REFL

$$\Delta \vdash_{\mathrm{q}}' X \leq X$$

SUB-Q-ALG-OBJ

$$\Delta \vdash_{\mathrm{q}}' T \leq \mathit{Object}$$

SUB-Q-ALG-VAR

$$\frac{\begin{array}{c} X \textbf{ extends } T \in \Delta \\ U \neq X, U \neq \mathit{Object} \qquad \Delta \vdash_{\mathrm{q}}' T \leq U \end{array}}{\Delta \vdash_{\mathrm{q}}' X \leq U}$$

SUB-Q-ALG-CLASS

$$\frac{N \trianglelefteq_{\mathrm{c}} N' \qquad N' \neq \mathit{Object}}{\Delta \vdash_{\mathrm{q}}' N \leq N'}$$

SUB-Q-ALG-IFACE

$$\frac{K \trianglelefteq_{\mathrm{i}} K'}{\Delta \vdash_{\mathrm{q}}' K \leq K'}$$

rule ent-env followed by zero or more applications of rule ent-super. Similarly, rule ent-q-alg-iface imitates an application of rule ent-iface followed by zero or more applications of rule ent-super.

- The auxiliary relation $\mathcal{R} \in \mathsf{sup}(\mathcal{S})$ states that $\mathcal{R}$ is a *super constraint* of $\mathcal{S}$. Super constraints arise either through reflexivity (rule sup-refl) or through superinterface constraints (rule sup-step).

- Quasi-algorithmic subtyping, written $\Delta \vdash_{\mathrm{q}} T \leq U$, states that T is a subtype of U under type environment Δ. Section 3.6.1 proves that quasi-algorithmic and declarative subtyping coincide.

  Quasi-algorithmic subtyping distinguishes two cases: Rule sub-q-alg-kernel states that quasi-algorithmic subtyping includes its kernel variant (explained next), and rule sub-q-alg-impl establishes a subtyping relationship between type T and interface type K by lifting T to U and then solving the constraint U **implements** K. Different to rule ent-q-alg-up, there is no polarity check.

- The *kernel of quasi-algorithmic subtyping*, written $\Delta \vdash_{\mathrm{q}}{}' T \leq U$, is a subset of the quasi-algorithmic subtyping relation that does not include subtyping implied by constraint entailment. Essentially, the kernel of quasi-algorithmic subtyping corresponds to FGJ's subtyping relation extended with interface inheritance. The side conditions "$U \neq X, U \neq \mathit{Object}$ in rule sub-q-alg-var and "$N' \neq \mathit{Object}$" in rule sub-q-alg-class ensure that the kernel of quasi-algorithmic subtyping is syntax-directed; that is, given a derivation $\mathcal{D}$ of $\Delta \vdash_{\mathrm{q}}{}' T \leq U$, the two types T and U uniquely determine the last rule of $\mathcal{D}$.

- The relation $N \trianglelefteq_{\mathsf{c}} M$ denotes *class inheritance* between class types N and M, whereas $K \trianglelefteq_{\mathsf{i}} L$ denotes *interface inheritance* between interface types K and L. Rule inh-iface-super expresses non-trivial inheritance between interface types through superinterface constraints. The rule is only applicable to single-headed interfaces because multi-headed interfaces do not form valid types. The notation $\overline{N} \trianglelefteq_{\mathsf{c}} \overline{M}$ abbreviates $(\forall i)\ N_i \trianglelefteq_{\mathsf{c}} M_i$.

With quasi-algorithmic constraint entailment, the condition to ensure that all superinterfaces are properly implemented for the program in Figure 3.14 now reads as follows (cf. condition wf-impl-1-informal-wrong, page 50):

wf-impl-1-informal

> "For every **implementation**<$\overline{X}$> J [N] **where** $\overline{P}$ ... the corresponding superinterface constraint N **implements** I must hold under type environment $\overline{P}$ *with respect to quasi-algorithmic constraint entailment.*"

Indeed, unlike wf-impl-1-informal-wrong, this criterion detects that the program in Figure 3.14 misses an implementation of I for C: there exists no derivation for $\emptyset \Vdash_{\mathrm{q}} C$ **implements** I.

Before defining the well-formedness criteria for implementation definitions, Figure 3.17 introduces the notion of *dispatch types*. The jth implementing type of interface I is a

Figure 3.17 Dispatch types and positions.

$$\boxed{j \in \mathsf{disp}(I) \qquad Y \in \mathsf{disp}(\mathit{rcsig}) \qquad Y \in \mathsf{disp}(P) \qquad Y \in \mathsf{disp}(\mathit{msig})}$$

DISP-IFACE

$$\frac{\begin{array}{c}\mathbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\,\overline{Y}^{n}\ \mathbf{where}\ \overline{R}^{m}\,]\ \mathbf{where}\ \overline{P}\,\{\ldots\ \overline{\mathit{rcsig}}^{n}\,\}\\ (\forall i \in [n], i \neq j)\ Y_j \in \mathsf{disp}(\mathit{rcsig}_i) \qquad (\forall i \in [m])\ Y_j \in \mathsf{disp}(R_i)\end{array}}{j \in \mathsf{disp}(I)}$$

DISP-RCSIG

$$\frac{(\forall i)\ Y \in \mathsf{disp}(\mathit{msig}_i)}{Y \in \mathsf{disp}(\mathbf{receiver}\ \{\overline{\mathit{msig}}\})}$$

DISP-CONSTR

$$\frac{(\forall i)\ \text{if } G_i = Y \text{ then } i \in \mathsf{disp}(I)}{Y \in \mathsf{disp}(\overline{G}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>})}$$

DISP-MSIG

$$\frac{Y \notin \overline{X} \qquad Y \in \overline{T}}{Y \in \mathsf{disp}(\texttt{<}\overline{X}\texttt{>}\,\overline{T\,x} \to T\ \mathbf{where}\ \overline{P})}$$

dispatch type, written $j \in \mathsf{disp}(I)$, if it appears in every non-static method signature of I or one of its superinterfaces as the receiver or at the top level of some argument type. In other words: if m is a non-static method of I or any of its superinterfaces, then $j \in \mathsf{disp}(I)$ guarantees that every invocation of m resolves the jth implementing type of I. The auxiliary relations $Y \in \mathsf{disp}(\mathit{rcsig})$, $Y \in \mathsf{disp}(P)$, and $Y \in \mathsf{disp}(\mathit{msig})$ assert that implementing type variable Y is a dispatch type with respect to a receiver rcsig, a constraint P, and a method signature msig, respectively.

The well-formedness criteria for implementations now require that for each implementation

$$\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\,[\overline{N}]\ \mathbf{where}\ \overline{P}\ \ldots$$

the following conditions must hold:

WF-IMPL-1 There exist suitable implementations for all superinterfaces of I; that is, if $\mathcal{Q} \in \mathsf{sup}(\overline{N}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>})$ then $\overline{P} \Vdash_{\mathrm{q}} \mathcal{Q}$.

WF-IMPL-2 The dispatch types among $\overline{N}$ fully determine the type variables $\overline{X}$; that is $\overline{X} \subseteq \mathsf{ftv}(\{N_i \mid i \in \mathsf{disp}(I)\})$.

WF-IMPL-3 In all constraints $\overline{G}\ \mathbf{implements}\ K \in \overline{P}$, the types $\overline{G}$ are type variables from $\overline{X}$; that is, $\overline{G} \subseteq \overline{X}$.

As already discussed, criterion WF-IMPL-1 ensures that suitable implementations for all relevant superinterfaces exist. The two other criteria contribute to decidability of constraint entailment. Criterion WF-IMPL-2, in combination with WF-PROG-4 as defined shortly, bears some resemblance to the *coverage condition* given by Sulzmann and coworkers [210] for Haskell type classes. For criterion WF-IMPL-3, there exists a corresponding restriction in the Haskell 98 report [173]. Sulzmann and coworkers' *bound-variable condition* [210] is also similar to it.

Figure 3.18 Greatest lower bound.

$$\boxed{\Delta \vdash G_1 \sqcap G_2 = H}$$

$$\textsc{glb-left}\quad \frac{\Delta \vdash G_1 \le G_2}{\Delta \vdash G_1 \sqcap G_2 = G_1} \qquad\qquad \textsc{glb-right}\quad \frac{\Delta \vdash G_2 \le G_1}{\Delta \vdash G_1 \sqcap G_2 = G_2}$$

Criteria for Programs

The notation $\Delta \vdash G_1 \sqcap G_2 = H$ denotes that H is the *greatest lower bound* of G_1 and G_2 with respect to Δ. Figure 3.18 defines this relation formally. The notation $\Delta \vdash \overline{G} \sqcap \overline{G'} = \overline{H}$ abbreviates $(\forall i)\ \Delta \vdash G_i \sqcap G'_i = H_i$.

The CoreGI program under consideration must fulfill the following well-formedness criteria:

WF-PROG-1 A program does not contain two different implementations for the same interface with unifiable implementing types. That is, for each pair of disjoint implementation definitions

$$\textbf{implementation}<\overline{X}>\ I<\overline{T}>\ [\,\overline{M}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

$$\textbf{implementation}<\overline{Y}>\ I<\overline{U}>\ [\,\overline{N}\,]\ \textbf{where}\ \overline{Q}\ \ldots$$

it holds that, for all substitutions $[\overline{V/X}]$ and $[\overline{W/Y}]$, $[\overline{V/X}]\overline{M} \neq [\overline{W/Y}]\overline{N}$.

WF-PROG-2 A program does not contain two implementations of different instantiations of the same interface or for different non-dispatch types, provided the dispatch types of the implementations are subtype compatible. That is, for each pair of implementation definitions

$$\textbf{implementation}<\overline{X}>\ I<\overline{T}>\ [\,\overline{M}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

$$\textbf{implementation}<\overline{Y}>\ I<\overline{U}>\ [\,\overline{N}\,]\ \textbf{where}\ \overline{Q}\ \ldots$$

and for all substitutions $[\overline{V/X}]$ and $[\overline{W/Y}]$ such that $\emptyset \vdash [\overline{V/X}]M_i \sqcap [\overline{W/Y}]N_i$ exists for all $i \in \mathsf{disp}(I)$, it holds that $[\overline{V/X}]\overline{T} = [\overline{W/Y}]\overline{U}$ and that $[\overline{V/X}]M_j = [\overline{W/Y}]N_j$ for all $j \notin \mathsf{disp}(I)$.

WF-PROG-3 Implementation definitions are downward closed. That is, for each pair of implementation definitions

$$\textbf{implementation}<\overline{X}>\ I<\overline{T}>\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

$$\textbf{implementation}<\overline{X'}>\ I<\overline{T'}>\ [\,\overline{N'}\,]\ \textbf{where}\ \overline{P'}\ \ldots$$

Figure 3.19 Illegal CoreGI program (violates well-formedness criterion WF-PROG-7).

```
interface I [X] {
  receiver {m : • → X}
}
interface J₁ [X where X implements I] {receiver {}}
interface J₂ [X where X implements I] {receiver {}}
interface J [X where X implements J₁, X implements J₂] {// illegal
  receiver {m′ : X → Object}
}
```

and for all substitutions $[\overline{V/X}]$ and $[\overline{V'/X'}]$ with $\emptyset \vdash [\overline{V/X}]\overline{N} \sqcap [\overline{V'/X'}]\overline{N'} = \overline{M}$ there exists an implementation definition

$$\textbf{implementation}<\overline{Y}>\ I<\overline{U}>\ [\,\overline{M'}\,]\ \textbf{where}\ \overline{Q}\ \ldots$$

and a substitution $[\overline{W/Y}]$ such that $\overline{M} = [\overline{W/Y}]\overline{M'}$.

WF-PROG-4 Constraints on implementation definitions are consistent with constraints on implementation definitions for subclasses. That is, for each pair of implementation definitions

$$\textbf{implementation}<\overline{X}>\ I<\overline{T}>\ [\,\overline{M}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

$$\textbf{implementation}<\overline{Y}>\ I<\overline{U}>\ [\,\overline{N}\,]\ \textbf{where}\ \overline{Q}\ \ldots$$

and for all substitutions $[\overline{V/X}]$ and $[\overline{W/Y}]$ with $[\overline{V/X}]\overline{M} \trianglelefteq_{\mathrm{c}} [\overline{W/Y}]\overline{N}$ and $\emptyset \Vdash [\overline{W/Y}]\overline{Q}$, it holds that $\emptyset \Vdash [\overline{V/X}]\overline{P}$.

WF-PROG-5 The class and interface graphs of the program are acyclic. (Each class definition $\textbf{class}\ C<\overline{X}>\ \textbf{extends}\ D<\overline{T}>\ \ldots$ contributes an edge $C \to D$ to the class graph, and each interface definition $\textbf{interface}\ I<\overline{X}>\ [\overline{Y}\ \textbf{where}\ \overline{R}]\ \ldots$ and each constraint $\overline{G}\ \textbf{implements}\ J<\overline{V}> \in \overline{R}$ contribute an edge $I \to J$ to the interface graph.)

WF-PROG-6 Multiple instantiation inheritance for interfaces is not allowed. That is, if $K \trianglelefteq_{\mathrm{i}} I<\overline{T}>$ and $K \trianglelefteq_{\mathrm{i}} I<\overline{U}>$ then $\overline{T} = \overline{U}$.

WF-PROG-7 Multiple inheritance for single-headed interfaces with neither positive nor negative polarity is not allowed. That is, if $1 \notin \mathsf{pol}^{+}(I)$, $1 \notin \mathsf{pol}^{-}(I)$, $I<\overline{T}> \trianglelefteq_{\mathrm{i}} K_1$, and $I<\overline{T}> \trianglelefteq_{\mathrm{i}} K_2$, then either $K_1 \trianglelefteq_{\mathrm{i}} K_2$ or $K_2 \trianglelefteq_{\mathrm{i}} K_1$.

We already discussed criteria WF-PROG-1 to WF-PROG-4 in Section 2.3.4. Criteria WF-PROG-5 and WF-PROG-6 are standard for Java-like languages [82, § 8.1.4, § 8.1.5, § 9.1.3].

The last criterion WF-PROG-7 is required to ensure that minimal types exist. Consider the program in Figure 3.19, which violates the criterion because $1 \notin \mathsf{pol}^{-}(J)$, $1 \notin \mathsf{pol}^{+}(J)$,

Figure 3.20 Closure of a set of types.

$$\boxed{T \in \mathsf{closure}_{\Delta}(\mathscr{T})}$$

CLOSURE-ELEM
$$\frac{T \in \mathscr{T}}{T \in \mathsf{closure}_{\Delta}(\mathscr{T})}$$

CLOSURE-UP
$$\frac{T \in \mathsf{closure}_{\Delta}(\mathscr{T}) \qquad \Delta \vdash_{\mathrm{q}}{}' \, T \leq N}{N \in \mathsf{closure}_{\Delta}(\mathscr{T})}$$

CLOSURE-DECOMP-CLASS
$$\frac{C\texttt{<}\overline{T}\texttt{>} \in \mathsf{closure}_{\Delta}(\mathscr{T})}{T_i \in \mathsf{closure}_{\Delta}(\mathscr{T})}$$

CLOSURE-DECOMP-IFACE
$$\frac{I\texttt{<}\overline{T}\texttt{>} \in \mathsf{closure}_{\Delta}(\mathscr{T})}{T_i \in \mathsf{closure}_{\Delta}(\mathscr{T})}$$

$J \trianglelefteq_{\mathrm{i}} J_1$, $J \trianglelefteq_{\mathrm{i}} J_2$, but neither $J_1 \trianglelefteq_{\mathrm{i}} J_2$ nor $J_2 \trianglelefteq_{\mathrm{i}} J_1$ holds. We have $1 \in \mathsf{pol}^{+}(J_i)$, so $\emptyset \Vdash J_i \, \textbf{implements} \, I$ for $i = 1, 2$ by rules ENT-IFACE and ENT-SUPER from Figure 3.3. Thus, $\emptyset; x : J \vdash x.m() : J_i$ for $i = 1, 2$ by subsuming x to either J_1 or J_2. However, $1 \notin \mathsf{pol}^{+}(J)$, so $\emptyset \Vdash J \, \textbf{implements} \, I$ is not derivable. Consequently, $\emptyset; x : J \vdash x.m() : J$ is not derivable. Because J_1 and J_2 are not related by subtyping, we conclude that $x.m()$ does not have a minimal type under the variable environment $x : J$.

Criteria for Type Environments

The following definition is due to Trifonov and Smith [230].

Definition 3.10 (Contractive type environments). A type environment Δ is *contractive* if, and only if, there exist no type variables $X_1, \ldots, X_n$ such that $X_1 = X_n$ and $X_i \, \textbf{extends} \, X_{i+1} \in \Delta$ for each $i \in \{1, \ldots, n-1\}$.

The notation $\mathsf{closure}_{\Delta}(\mathscr{T})$ denotes the *closure* of a set of types $\mathscr{T}$ with respect to a type environment Δ. (Metavariables $\mathscr{T}$, $\mathscr{U}$, and $\mathscr{V}$ range over sets of types.) Figure 3.20 defines $\mathsf{closure}_{\Delta}(\mathscr{T})$ as the least superset of $\mathscr{T}$ closed under the kernel of quasi-algorithmic subtyping and under decomposition of generic class and interface types.

The well-formedness criteria on type environments now require that every type environment Δ must fulfill the following conditions.

WF-TENV-1 The type environment Δ is contractive.

WF-TENV-2 If $\mathscr{T}$ is a finite set of types, then the closure of $\mathscr{T}$ with respect to Δ is finite.

WF-TENV-3 A type variable does not have several unrelated G-types among its bounds. That is, if $X \, \textbf{extends} \, G_1 \in \Delta$ and $X \, \textbf{extends} \, G_2 \in \Delta$ then $\Delta \vdash G_1 \leq G_2$ or $\Delta \vdash G_2 \leq G_1$.

WF-TENV-4 A type variable is not a subtype of different instantiations of the same interface. That is, if $\Delta \vdash_{\mathrm{q}}{}' \, X \leq I\texttt{<}\overline{T}\texttt{>}$ and $\Delta \vdash_{\mathrm{q}}{}' \, X \leq I\texttt{<}\overline{U}\texttt{>}$ then $\overline{T} = \overline{U}$.

WF-TENV-5 A type variable has only negative interfaces among its bounds. That is, if $X \, \textbf{extends} \, I\texttt{<}\overline{T}\texttt{>} \in \Delta$ then $1 \in \mathsf{pol}^{-}(I)$.

WF-TENV-6 The type environment Δ does not contain two implementation constraints for different instantiations of the same interface or for different non-dispatch types in covariant position, provided the dispatch types of the implementation constraints are subtype compatible. The same holds for one implementation constraint in combination with an implementation definition. That is:

1. For each pair of constraints

$$\overline{G}\,\textbf{implements}\,I\texttt{<}\overline{T}\texttt{>} \in \mathsf{sup}(\Delta)$$

$$\overline{H}\,\textbf{implements}\,I\texttt{<}\overline{W}\texttt{>} \in \mathsf{sup}(\Delta)$$

such that $\Delta \vdash G_i \sqcap H_i$ exists for all $i \in \mathsf{disp}(I)$, it holds that $\overline{T} = \overline{W}$ and $G_j = H_j$ for all $j \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$.

2. For each constraint and each implementation definition

$$\overline{G}\,\textbf{implements}\,I\texttt{<}\overline{T}\texttt{>} \in \mathsf{sup}(\Delta)$$

$$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{W}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

such that $\Delta \vdash G_i \sqcap [\overline{U/X}]N_i$ exists for all $i \in \mathsf{disp}(I)$ and some $\overline{U}$, it holds that $\overline{T} = [\overline{U/X}]\overline{W}$ and $G_j = [\overline{U/X}]N_j$ for all $j \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$.

Criterion WF-TENV-1 and WF-TENV-2 are required to establish decidability of constraint entailment and subtyping. Strictly speaking, criterion WF-TENV-2 is not compatible with JavaGI being a conservative extension of Java 1.5 because Java allows programs to have an infinitary closure of types. However, neither the authors nor other researchers are aware of any such programs with practical value [233, 113]. Moreover, neither the Scala language [166, § 5.1.5] nor the Common Language Infrastructure of the .NET framework [65, Partition II, § 9.2] allows programs to have an infinitary closure of types.

Without well-formedness criterion WF-TENV-3, minimal types do not exist. For example, consider the interface

$$\textbf{interface}\ I\ [X]\ \{\ \textbf{receiver}\ \{\ m : X \to X\ \}\ \}$$

together with the type environment

$$\Delta = \{X\,\textbf{extends}\,Y_1, X\,\textbf{extends}\,Y_2, Y_1\,\textbf{implements}\,I, Y_2\,\textbf{implements}\,I\}$$

which violates WF-TENV-3. Then we have $\Delta; \Gamma \vdash x_1.m(x_2) : Y_i$ for $i = 1, 2$ and $\Gamma = x_1 : X, x_2 : X$. However, Y_1 and Y_2 are not related by subtyping. Moreover, $\Delta; \Gamma \vdash x_1.m(x_2) : X$ is not derivable because $1 \notin \mathsf{pol}^-(I)$ prevents $\Delta \Vdash X\,\textbf{implements}\,I$ from being valid. Hence, the expression $x_1.m(x_2)$ does not have a minimal type under Δ and Γ.

Criterion WF-TENV-4 is common for Java-like languages [82, § 4.4]. Moreover, the criterion is necessary to ensure minimal types. Assume two distinct classes C_1 and C_2, an interface

$$\textbf{interface}\ I\texttt{<}X\texttt{>}\ [Y]\ \{\ \textbf{receiver}\ \{\ m : \bullet \to X\ \}\ \}$$

Figure 3.21 CoreGI program demonstrating necessity of criterion WF-TENV-5.

```
interface I [X] {
  receiver {m : • → X}
}
interface J [X where X implements I] {receiver {}}
class C {}
implementation I [C] {
  receiver {
    • → C{new C()}
  }
}
```

Figure 3.22 CoreGI program demonstrating necessity of criterion WF-TENV-6(1).

```
interface I [X, Y] {
  receiver {m : • → Y}
  receiver {}
}
class A {}
class B extends A {}
class C1 {}
class C2 {}
```

and a type environment

$$\Delta = \{X\,\textbf{extends}\,I\text{<}C_1\text{>}, X\,\textbf{extends}\,I\text{<}C_2\text{>}\}$$

violating WF-TENV-4. Then $\Delta; x : X \vdash x.m() : C_i$ for $i = 1, 2$ but C_1 and C_2 are not related by subtyping, and $\Delta; x : X \vdash x.m() : T$ is not derivable for any common subtype T of C_1 and C_2.

Criterion WF-TENV-5 is also required to ensure the existence of minimal types. Consider the program in Figure 3.21 together with the type environment

$$\Delta = \{X\,\textbf{extends}\,C, X\,\textbf{extends}\,J\}$$

violating WF-TENV-5 (because $1 \notin \mathsf{pol}^{-}(J)$). The constraints $C\,\textbf{implements}\,I$ and $J\,\textbf{implements}\,I$ hold under Δ, so $\Delta; x : X \vdash x.m() : C$ and $\Delta; x : X \vdash x.m() : J$ but C and J are not related by subtyping. Moreover, $X\,\textbf{implements}\,I$ does not hold under Δ, so $\Delta; x : X \vdash x.m() : X$ is not derivable. Hence, $x.m()$ has no minimal type under Δ and $x : X$.

The last well-formedness criterion WF-TENV-6, which is somewhat related to WF-PROG-2, once again helps to guarantee the existence of minimal types. The example in Figure 3.22 shows why part (1) of the criterion is needed; a similar example shows why part (2) is needed. Consider the type environment

$$\Delta = \{A\,C_1\,\mathbf{implements}\,I,\;\; B\,C_2\,\mathbf{implements}\,I\}$$

which violates WF-PROG-2(1) because $\Delta \vdash A \sqcap B = B$, $2 \notin \mathsf{disp}(I)$, $2 \notin \mathsf{pol}^-(I)$, but $C_1 \neq C_2$. Then $\Delta; x : B \vdash x.m() : C_i$ for $i = 1, 2$ but C_1 and C_2 do not have a common subtype. Hence, minimal types do not exist.

3.6 Meta-Theoretical Properties

Having completed the definition of the static semantics, this sections proves that CoreGI enjoys type soundness and that its evaluation relation is deterministic. Moreover, the section shows that the declarative and the quasi-algorithmic formulations of constraint entailment and subtyping are equivalent. All theorems presented in this section make the implicit assumption that the underlying CoreGI program is well-formed.

3.6.1 Type Soundness

The type soundness proof relies on the equivalence of declarative and quasi-algorithmic constraint entailment and subtyping.

Theorem 3.11. *Quasi-algorithmic constraint entailment and subtyping are sound with respect to declarative constraint entailment and subtyping.*

(*i*) *If* $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{R}$ *then* $\Delta \Vdash \mathcal{R}$.

(*ii*) *If* $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ *then* $\Delta \Vdash \mathcal{P}$.

(*iii*) *If* $\Delta \vdash_{\mathrm{q}}{}' T \leq U$ *then* $\Delta \vdash T \leq U$.

(*iv*) *If* $\Delta \vdash_{\mathrm{q}} T \leq U$ *then* $\Delta \vdash T \leq U$.

Proof. The proof is by induction on the combined height of the derivations of $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{R}$, $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$, $\Delta \vdash_{\mathrm{q}}{}' T \leq U$, and $\Delta \vdash_{\mathrm{q}} T \leq U$. See Section B.1.1 for details. □

Theorem 3.12. *Quasi-algorithmic constraint entailment and subtyping are complete with respect to declarative constraint entailment and subtyping.*

(*i*) *If* $\Delta \Vdash \mathcal{P}$ *then* $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$.

(*ii*) *If* $\Delta \vdash T \leq U$ *then* $\Delta \vdash_{\mathrm{q}} T \leq U$.

Proof. The proof is by induction on the combined height of the derivations of $\Delta \Vdash \mathcal{P}$ and $\Delta \vdash T \leq U$. See Section B.1.2 for details. □

The type soundness proof of CoreGI follows the syntactic approach pioneered by Wright and Felleisen [244]. The progress theorem states that a well-typed expression is either a value or reduces to some other expression or is stuck on a bad cast.

Definition 3.13 (Stuck on a bad cast). An expression e is *stuck on a bad cast* if, and only if, there exists an evaluation context $\mathcal{E}$, a type T, and a value $v = \textbf{new}\, N(\overline{w})$ such that $e = \mathcal{E}[(T)\, v]$ and not $\emptyset \vdash N \leq T$.

Theorem 3.14 (Progress). *If $\emptyset; \emptyset \vdash e : T$ then either $e = v$ for some value v or $e \longrightarrow e'$ for some expression e' or e is stuck on a bad cast.*

Proof. The proof is by induction on the derivation of $\emptyset; \emptyset \vdash e : T$. See Section B.2.1 for details. □

The preservation theorems for the evaluation relations $\longmapsto$ and $\longrightarrow$ show that evaluation of expressions preserves types.

Theorem 3.15 (Preservation for top-level evaluation). *If $\emptyset; \emptyset \vdash e : T$ and $e \longmapsto e'$ then $\emptyset; \emptyset \vdash e' : T$.*

Proof. The proof is by induction on the derivation of $\emptyset; \emptyset \vdash e : T$. See Section B.2.2 for details. □

Theorem 3.16 (Preservation for proper evaluation). *If $\emptyset; \emptyset \vdash e : T$ and $e \longrightarrow e'$ then $\emptyset; \emptyset \vdash e' : T$.*

Proof. The derivation of $e \longrightarrow e'$ must end with rule DYN-CONTEXT, so there exists an evaluation context $\mathcal{E}$ and expressions e_0, e_0' such that $e = \mathcal{E}[e_0]$ and $e_0 \longmapsto e_0'$ and $\mathcal{E}[e_0'] = e'$. The claim $\emptyset; \emptyset \vdash \mathcal{E}[e'] : T$ now follows by induction on the structure of $\mathcal{E}$, using Theorem 3.15 for the base case. See Section B.2.3 for details. □

In the following, $\longrightarrow^*$ denotes the reflexive, transitive closure of the evaluation relation $\longrightarrow$. The type soundness theorem for CoreGI is very similar to that for FGJ.

Theorem 3.17 (Type soundness). *If $\emptyset; \emptyset \vdash e : T$ then either e diverges, or $e \longrightarrow^* v$ for some value v such that $\emptyset; \emptyset \vdash v : T$, or $e \longrightarrow^* e'$ for some expression e' such that e' is stuck on a bad cast.*

Proof. Assume that $e \longrightarrow^* e'$ for some normal form e'. Theorem 3.16 and an induction on the length of the evaluation sequence yields $\emptyset; \emptyset \vdash e' : T$. The claim now follows by Theorem 3.14. □

A stronger type soundness theorem holds for programs not containing any cast expressions.

Definition 3.18 (Cast-free). An expression e is *cast-free* if, and only if, neither e nor the underlying program contains a cast $(T)\, e'$ for some type T and some expression e'.

Theorem 3.19 (Type soundness for programs without casts). *If $\emptyset; \emptyset \vdash e : T$ and e is cast-free then either e diverges or $e \longrightarrow^* v$ for some value v such that $\emptyset; \emptyset \vdash v : T$.*

Proof. Obviously, if $e \longrightarrow^* e'$ and e is cast-free then so is e'. Moreover, a cast-free expression cannot be stuck on a bad cast. The claim now follows with Theorem 3.17. □

Figure 3.23 Program exhibiting nontermination of quasi-algorithmic entailment.

$\mathbf{interface}\ I\,[X]\ \{\mathbf{receiver}\ \{\}\}$
$\mathbf{class}\ C\texttt{<}X\texttt{>}\ \mathbf{extends}\ \mathit{Object}\ \{\}$
$\mathbf{class}\ D\ \mathbf{extends}\ C\texttt{<}D\texttt{>}\ \{\}$
$\mathbf{implementation}\texttt{<}X\texttt{>}\ I\,[C\texttt{<}X\texttt{>}]\ \mathbf{where}\ X\ \mathbf{implements}\ I\ \{\mathbf{receiver}\ \{\}\}$

Figure 3.24 Failed attempt to construct a derivation of $\emptyset \Vdash_{\mathrm{q}} D\,\mathbf{implements}\,I$.
Variables $\mathfrak{r}_1$ and $\mathfrak{r}_2$ stand for rule names ENT-Q-ALG-UP and ENT-Q-ALG-IMPL, respectively.

$$\dfrac{\dfrac{\text{(holds obviously)}}{\emptyset \vdash_{\mathrm{q}}' D \leq C\texttt{<}D\texttt{>}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{(holds obviously)}}{\emptyset \vdash_{\mathrm{q}}' D \leq C\texttt{<}D\texttt{>}} \quad \dfrac{\begin{array}{c}\dfrac{\vdots}{\emptyset \Vdash_{\mathrm{q}} D\,\mathbf{implements}\,I} \\ \mathbf{implementation}\texttt{<}X\texttt{>}\ I\,[C\texttt{<}X\texttt{>}] \\ \mathbf{where}\ X\,\mathbf{implements}\,I\ \ldots\end{array}}{\emptyset \Vdash_{\mathrm{q}}' C\texttt{<}D\texttt{>}\,\mathbf{implements}\,I}\,\mathfrak{r}_2}{\emptyset \Vdash_{\mathrm{q}} D\,\mathbf{implements}\,I}\,\mathfrak{r}_1 \\ \mathbf{implementation}\texttt{<}X\texttt{>}\ I\,[C\texttt{<}X\texttt{>}]\ \mathbf{where}\ X\,\mathbf{implements}\,I\ \ldots}{\emptyset \Vdash_{\mathrm{q}}' C\texttt{<}D\texttt{>}\,\mathbf{implements}\,I}\,\mathfrak{r}_2}{\emptyset \Vdash_{\mathrm{q}} D\,\mathbf{implements}\,I}\,\mathfrak{r}_1$$

3.6.2 Determinacy of Evaluation

CoreGI also enjoys a deterministic evaluation relation. This property is important because CoreGI's method lookup may involve more than one dispatch type, which could easily lead to ambiguities.

Theorem 3.20 (Determinacy of evaluation). *If $e \longrightarrow e'$ and $e \longrightarrow e''$ then $e' = e''$.*

Proof. See Section B.3. □

3.7 Typechecking Algorithm

The development of a typechecking algorithm for CoreGI proceeds in three steps: Section 3.7.1 shows how to decide constraint entailment and subtyping, Section 3.7.2 shows how to decide expression typing, and Section 3.7.3 shows how to decide program typing.

3.7.1 Deciding Constraint Entailment and Subtyping

The declarative specification of constraint entailment and subtyping in Section 3.3 is not immediately suitable for implementation: the conclusions of several rules overlap and the premises of rules ENT-SUPER, ENT-UP, and SUB-TRANS involve types not mentioned in the conclusions.

Figure 3.25 Algorithmic constraint entailment and subtyping.

$$\boxed{\Delta \Vdash_{\mathrm{a}} \mathcal{P} \qquad \Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \mathcal{P}}$$

$$\textsc{ent-alg-main}\quad \frac{\Delta; \emptyset; \texttt{false} \Vdash_{\mathrm{a}} \mathcal{P}}{\Delta \Vdash_{\mathrm{a}} \mathcal{P}} \qquad\qquad \textsc{ent-alg-extends}\quad \frac{\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq U}{\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} T \,\mathbf{extends}\, U}$$

$$\textsc{ent-alg-env}\quad \frac{R \in \Delta \qquad \overline{G}\,\mathbf{implements}\, I\texttt{<}\overline{V}\texttt{>} \in \mathsf{sup}(R) \qquad \Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{G}}{\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \overline{T}\,\mathbf{implements}\, I\texttt{<}\overline{V}\texttt{>}}$$

$$\textsc{ent-alg-iface}_1\quad \frac{\begin{array}{c}\Delta; \beta; I \vdash_{\mathrm{a}} T \uparrow I\texttt{<}\overline{V}\texttt{>} \\ 1 \in \mathsf{pol}^{+}(I) \qquad \mathsf{non\text{-}static}(I)\end{array}}{\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} T\,\mathbf{implements}\, I\texttt{<}\overline{V}\texttt{>}} \qquad \textsc{ent-alg-iface}_2\quad \frac{1 \in \mathsf{pol}^{+}(I) \qquad I\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathrm{i}} K \qquad \mathsf{non\text{-}static}(I)}{\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} I\texttt{<}\overline{V}\texttt{>}\,\mathbf{implements}\, K}$$

$$\textsc{ent-alg-impl}\quad \frac{\begin{array}{c}\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots \\ \Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow [\overline{U}/\overline{X}]\overline{N} \qquad \overline{V} = [\overline{U}/\overline{X}]\overline{V'} \qquad [\overline{U}/\overline{X}]\overline{N}\,\mathbf{implements}\, I\texttt{<}\overline{V}\texttt{>} \notin \mathscr{G} \\ \Delta; \mathscr{G} \cup \{[\overline{U}/\overline{X}]\overline{N}\,\mathbf{implements}\, I\texttt{<}\overline{V}\texttt{>}\}; \texttt{false} \Vdash_{\mathrm{a}} [\overline{U}/\overline{X}]\overline{P}\end{array}}{\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \overline{T}\,\mathbf{implements}\, I\texttt{<}\overline{V}\texttt{>}}$$

$$\boxed{\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}}$$

$$\textsc{ent-alg-lift}\quad \frac{(\forall i)\ \Delta \vdash_{\mathrm{q}}{}' T_i \leq U_i \qquad \beta \text{ or } \big((\forall i) \text{ if } T_i \neq U_i \text{ then } i \in \mathsf{pol}^{-}(I)\big)}{\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T}^{n} \uparrow \overline{U}^{n}}$$

$$\boxed{\Delta \vdash_{\mathrm{a}} T \leq U \qquad \Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq U}$$

$$\textsc{sub-alg-main}\quad \frac{\Delta; \emptyset \vdash_{\mathrm{a}} T \leq U}{\Delta \vdash_{\mathrm{a}} T \leq U} \qquad \textsc{sub-alg-kernel}\quad \frac{\Delta \vdash_{\mathrm{q}}{}' T \leq U}{\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq U} \qquad \textsc{sub-alg-impl}\quad \frac{\Delta; \mathscr{G}; \texttt{true} \Vdash_{\mathrm{a}} T\,\mathbf{implements}\, K}{\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq K}$$

Section 3.5.3 introduced an equivalent, quasi-algorithmic formulation of entailment and subtyping. However, this formulation does not lead directly to an implementation either: the conclusions of several rules overlap, the premises of rules ENT-Q-ALG-UP and SUB-Q-ALG-IMPL involve types not present in the conclusions, and the recursive invocation of constraint entailment in rule ENT-Q-ALG-IMPL may lead to nontermination. To illustrate the danger of nontermination, consider the program in Figure 3.23. Searching for a derivation of $\emptyset \Vdash_{\mathrm{q}} D\,\mathbf{implements}\, I$ quickly leads to infinite regress as demonstrated by the failed attempt in Figure 3.24.

Figure 3.25 shows an *algorithmic* formulation of constraint entailment and subtyping. It is straightforward to derive an implementation from this formulation (see Figures B.3 and B.4 in the appendix).

- Algorithmic constraint entailment, written $\Delta \Vdash_{\mathrm{a}} \mathcal{P}$, asserts validity of constraint $\mathcal{P}$ with respect to type environment Δ. The declarative specification of constraint entailment is equivalent to the algorithmic formulation (to be proved shortly).

- The auxiliary relation $\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \mathcal{P}$ for algorithmic constraint entailment establishes validity of constraint $\mathcal{P}$ with respect to type environment Δ, goal cache $\mathscr{G}$, and boolean flag β. The goal cache $\mathscr{G}$ maintains the set of implementation constraints encountered while searching for a derivation. Rule ENT-ALG-IMPL avoids nontermination by performing recursive invocations only on constraints not contained in $\mathscr{G}$. The boolean flag β specifies whether type T_j of some constraint $\overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ may be lifted to a supertype without checking that the polarity of the jth implementing type of I is negative.

- The auxiliary relation $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$ lifts the types $\overline{T}$ of an implementation constraint $\overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ to supertypes $\overline{U}$ under type environment Δ. The job of β is the same as before.

- Algorithmic subtyping, written $\Delta \vdash_{\mathrm{a}} T \leq U$, states that T is a subtype of U under type environment Δ. The declarative specification of subtyping is equivalent to the algorithmic formulation (to be proved shortly).

- The auxiliary relation $\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq U$ states that T is a subtype of U under type environment Δ and goal cache $\mathscr{G}$. Rule SUB-ALG-KERNEL falls back to the kernel variant of quasi-algorithmic subtyping because the corresponding rules are already syntax-directed and easily implementable (see Figure 3.16).

Following the rules in Figure 3.25 and the rules for quasi-algorithmic kernel subtyping in Figure 3.16, the implementation of a entailment and subtype checker becomes straightforward (see Figures B.3 and B.4 in the appendix). Only two details need further explanation:

- Rules ENT-ALG-ENV, ENT-ALG-IFACE$_1$, ENT-ALG-IFACE$_2$, and ENT-ALG-IMPL overlap. The implementation simply tries the rules in order of their appearance until one succeeds or all fail.

- Rule ENT-ALG-IMPL lifts types $\overline{T}$ to class types $[\overline{U/X}]\overline{N}$, which requires finding a suitable substitution $[\overline{U/X}]$. In other words, $[\overline{U/X}]$ must solve the matching problem modulo kernel subtyping $(\Delta, \overline{X}, \{T_1 \leq^? N_1, \ldots, T_n \leq^? N_n\})$.

Matching modulo kernel subtyping is a special case of unification modulo kernel subtyping, which the forthcoming Section 3.7.3 needs anyway. In the following, the notation $\mathsf{ftv}(\Delta)$ denotes the set $\bigcup\{\mathsf{ftv}(P) \mid P \in \Delta\}$ for some type environment Δ.

Figure 3.26 Transformation of unification modulo kernel subtyping problems.

$$\boxed{\{\overline{T_i \le^? U_i}\} \Longrightarrow_\Delta \{\overline{T_i' \le^? U_i}\}}$$

UNIFY-CLASS

$$\frac{C \neq D \qquad \textbf{class}\ C\texttt{<}\overline{Y}\texttt{>}\ \textbf{extends}\ M\ \ldots}{\{C\texttt{<}\overline{T}\texttt{>} \le^? D\texttt{<}\overline{U}\texttt{>}\} \,\dot\cup\, \mathscr{S} \Longrightarrow_\Delta \{[\overline{T/Y}]M \le^? D\texttt{<}\overline{U}\texttt{>}\} \cup \mathscr{S}}$$

UNIFY-IFACE-UP

$$\frac{\begin{array}{c} I \neq J \qquad \textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\ [Y\ \textbf{where}\ \overline{R}]\ \ldots \\ R_i = Y\ \textbf{implements}\ K \end{array}}{\{I\texttt{<}\overline{T}\texttt{>} \le^? J\texttt{<}\overline{U}\texttt{>}\} \,\dot\cup\, \mathscr{S} \Longrightarrow_\Delta \{[\overline{T/X}]K \le^? J\texttt{<}\overline{U}\texttt{>}\} \cup \mathscr{S}}$$

UNIFY-IFACE-OBJECT

$$\frac{}{\{K \le^? G\} \,\dot\cup\, \mathscr{S} \Longrightarrow_\Delta \{\mathit{Object} \le^? G\} \cup \mathscr{S}}$$

UNIFY-VAR-ENV

$$\frac{X\ \textbf{extends}\ T \in \Delta}{\{X \le^? U\} \,\dot\cup\, \mathscr{S} \Longrightarrow_\Delta \{T \le^? G\} \cup \mathscr{S}}$$

UNIFY-VAR-OBJECT

$$\frac{X\ \textbf{extends}\ T \notin \Delta \text{ for all } T}{\{X \le^? U\} \,\dot\cup\, \mathscr{S} \Longrightarrow_\Delta \{\mathit{Object} \le^? U\} \cup \mathscr{S}}$$

Definition 3.21 (Unification modulo kernel subtyping). A *unification problem modulo kernel subtyping* is a triple $\mathbb{U} = \big(\Delta, \overline{X}, \{T_1 \le^? U_1, \ldots T_n \le^? U_n\}\big)$ such that $\mathsf{ftv}(\Delta) \cap \overline{X} = \emptyset$ and $T_i = Y$ (or $U_i = Y$) implies $Y \notin \overline{X}$ for all $i \in [n]$. A *solution* of $\mathbb{U}$ is a substitution $\varphi = [\overline{V/X}]$ such that $\Delta \vdash_{\mathrm{q}}' \varphi T_i \le \varphi U_i$ for all $i = 1, \ldots, n$. A *most-general solution* of $\mathbb{U}$ is a solution φ that is more general than any other solution φ' of $\mathbb{U}$; that is, there exists a substitution ψ such that $\varphi' = \psi\varphi$ (where $\psi\varphi$ denotes the composition of ψ and φ).

The relation $\{\overline{T_i \le^? U_i}\} \Longrightarrow_\Delta \{\overline{T_i' \le^? U_i}\}$, defined in Figure 3.26, transforms a set of inequations $\{\overline{T_i \le^? U_i}\}$ into $\{\overline{T_i' \le^? U_i}\}$ by lifting one of the types T_i to a direct supertype T_i' under type environment Δ. The notation $\mathscr{M}_1 \,\dot\cup\, \mathscr{M}_2$ denotes the disjoint union of $\mathscr{M}_1$ and $\mathscr{M}_2$; that is, $\mathscr{M}_1 \dot\cup \mathscr{M}_2$ is the same as $\mathscr{M}_1 \cup \mathscr{M}_2$ but additionally asserts $\mathscr{M}_1 \cap \mathscr{M}_2 = \emptyset$. The metavariable $\mathscr{S}$ ranges over subtyping inequations $\{T_1 \le^? U_1, \ldots T_n \le^? U_n\}$.

Definition 3.22 (Algorithm for unification modulo kernel subtyping). The procedure $\mathtt{unify}_{\le}(\mathbb{U})$ solves a unification problem modulo kernel subtyping $\mathbb{U} = (\Delta, \overline{X}, \mathscr{S})$ by first reducing $\mathscr{S}$ to all its normal forms with respect to $\Longrightarrow_\Delta$. If syntactic unification [8] succeeds for any of these normal forms and returns a solution φ, $\mathtt{unify}_{\le}(\mathbb{U})$ also returns φ. Otherwise, it fails.

Theorem 3.23 (Soundness and completeness of $\mathtt{unify}_{\le}$). *Let $\mathbb{U}$ be a unification problem modulo kernel subtyping. If $\mathtt{unify}_{\le}(\mathbb{U})$ returns a substitution φ then φ is an idempotent, most general solution of $\mathbb{U}$ (soundness). Moreover, if $\mathbb{U}$ has a solution, then $\mathtt{unify}_{\le}(\mathbb{U})$ does not fail (completeness).*

Proof. If $\mathbb{U} = (\Delta, \overline{X}, \mathscr{S})$ and $\mathscr{S} \Longrightarrow_\Delta \mathscr{S}'$ then $(\Delta, \overline{X}, \mathscr{S}')$ is a unification problem modulo kernel subtyping with the same solution set as $\mathbb{U}$. The claim now follows because syntactic unification is sound and complete. □

Theorem 3.24 (Termination of $\texttt{unify}_\leq$). *Let $\mathbb{U}$ be a unification problem modulo kernel subtyping. Then $\texttt{unify}_\leq(\mathbb{U})$ terminates.*

Proof. Holds because syntactic unification terminates and the reduction relation $\Longrightarrow$ is terminating. See Section B.4.1 for details. □

Equivalence of algorithmic and quasi-algorithmic entailment and subtyping follows with the next two theorems.

Theorem 3.25. *Algorithmic constraint entailment and subtyping are sound with respect to quasi-algorithmic constraint entailment and subtyping.*

(*i*) *If $\Delta \Vdash_{\mathrm{a}} \mathcal{P}$ then $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$.*

(*ii*) *If $\Delta \vdash_{\mathrm{a}} T \leq U$ then $\Delta \vdash_{\mathrm{q}} T \leq U$.*

Proof. See Section B.4.2. □

Theorem 3.26. *Algorithmic constraint entailment and subtyping are complete with respect to quasi-algorithmic constraint entailment and subtyping.*

(*i*) *If $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ then $\Delta \Vdash_{\mathrm{a}} \mathcal{P}$*

(*ii*) *If $\Delta \vdash_{\mathrm{q}} T \leq U$ then $\Delta \vdash_{\mathrm{a}} T \leq U$.*

Proof. See Section B.4.3. □

Equivalence between the algorithmic and the declarative formulations of constraint entailment and subtyping then follows with Theorems 3.11 and 3.12. Algorithmic constraint entailment and subtyping also terminates:

Theorem 3.27 (Termination of algorithmic entailment and subtyping). *The entailment and subtyping algorithms induced by the rules in Figure 3.25 and by the rules for quasi-algorithmic kernel subtyping in Figure 3.16 terminate.*

Proof. The proof relies on well-formedness criterion WF-TENV-2 to show that the goal cache $\mathscr{G}$ does not grow indefinitely. Section B.4.4 gives all the details of the proof, including a precise definition of the entailment and subtyping algorithms. □

3.7.2 Deciding Expression Typing

The declarative specification of the typing relation for expressions from Section 3.5.1 is not well-suited for implementing a typechecking algorithm. The main culprit is the explicit subsumption rule EXP-SUBSUME that allows lifting the type of an expression to some arbitrary supertype. This section presents a syntax-directed variant of expression typing that is suitable for implementation and that computes minimal types.

Algorithmic Method Typing

Algorithmic method typing compensates for the lack of an explicit subsumption rule in the syntax-directed variant of expression typing (to be defined shortly) by integrating subsumption into method typing. Furthermore, it infers those types which the declarative specification of method typing must guess. Consider rule MTYPE-IFACE from Figure 3.8 on page 43. An application of this rule must guess all types T_i for $i \neq j$ and all types $\overline{V}$. Even if mtype also had access to the types of the actual parameters of a method invocation, this would, in general, not be enough to determine all $\overline{T}$ and all $\overline{V}$.

Fortunately, well-formedness criteria WF-PROG-2 and WF-TENV-6 make it possible to define an algorithmic variant of mtype that infers those $\overline{T}$ and $\overline{V}$ that are needed to compute the type (i.e., signature) of a method. Figure 3.27 defines the first part of the inference machinery by extending algorithmic constraint entailment to *entailment for constraints with optional types.*

A *constraint with optional types* has the form $\overline{T^?}\,\mathbf{implements}\,I\texttt{<}\overline{U^?}\texttt{>}$, where each $T_i^?$ and each $U_i^?$ is optional (i.e., either nil or a regular type). Entailment for such constraints has the form $\Delta \Vdash_{\mathrm{a}}^{?} \overline{T^?}\,\mathbf{implements}\,I\texttt{<}\overline{U^?}\texttt{>} \rightarrowtriangle \overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>}$. It takes a constraint $\overline{T^?}\,\mathbf{implements}\,I\texttt{<}\overline{U^?}\texttt{>}$ and completes it to $\overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>}$ by inferring types for those $T_i^?$ and $U_i^?$ that are nil. Moreover, it ensures that the completed constraint $\overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>}$ holds under type environment Δ. The definition of entailment for constraints with optional types relies on several auxiliaries:

- The auxiliary $\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}}^{?} \overline{T^?}\,\mathbf{implements}\,I\texttt{<}\overline{U^?}\texttt{>} \rightarrowtriangle \overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>}$ is the analogon to $\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \overline{T}\,\mathbf{implements}\,\overline{U}$ from Figure 3.25.
- The auxiliary $\Delta; \beta; I \vdash_{\mathrm{a}}^{?} \overline{T^?} \uparrow \overline{U} \rightarrowtriangle \overline{T}$ is the analogon to $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$ from Figure 3.25: it lifts those $T_i^? \neq \mathsf{nil}$ to a supertype U_i and completes those $T_i^? = \mathsf{nil}$ to U_i.
- The auxiliary $T^? \sim T$ matches an optional type $T^?$ with a regular type T.

Theorem 3.28. *Entailment for constraints with optional types is sound with respect to algorithmic entailment: if* $\Delta \Vdash_{\mathrm{a}}^{?} \overline{T^?}\,\mathbf{implements}\,I\texttt{<}\overline{W^?}\texttt{>} \rightarrowtriangle \mathcal{R}$ *then* $\Delta \Vdash_{\mathrm{a}} \mathcal{R}$.

Proof. The proof is by induction on the derivation given. See Section B.5.1 for details. □

Theorem 3.29. *Entailment for constraints with optional types is complete with respect to algorithmic entailment: if* $\Delta \Vdash_{\mathrm{a}} \overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ *and* $\overline{T^?}\,\overline{V^?} \sim \overline{T}\,\overline{V}$ *and* $T_i^? \neq \mathsf{nil}$ *for* $i \in \mathsf{disp}(I)$, *then* $\Delta \Vdash_{\mathrm{a}}^{?} \overline{T^?}\,\mathbf{implements}\,I\texttt{<}\overline{V^?}\texttt{>} \rightarrowtriangle \overline{U}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ *such that* $\Delta \vdash_{\mathrm{q}}' T_i \leq U_i$ *for all* i *and* $U_i = T_i$ *for those* i *with* $T_i^? \neq \mathsf{nil}$ *or* $i \notin \mathsf{pol}^{-}(I)$.

Proof. The claim follows with a case distinction on the last rule of the derivation given. See Section B.5.2 for details. □

Figure 3.29 formalizes algorithmic method typing, relying on the auxiliaries of Figure 3.28. The relation $\mathsf{a\text{-}mtype}_{\Delta}(m, T, \overline{T})$ determines the signature of non-static method m when invoked on receiver and arguments with static types T and $\overline{T}$, respectively. The

Figure 3.27 Entailment for constraints with optional types.

$$\boxed{\Delta \Vdash^?_{\mathrm{a}} \overline{T^?}\ \textbf{implements}\ I{<}\overline{U^?}{>} \rightarrowtriangle \mathcal{R} \qquad \Delta;\mathscr{G};\beta \Vdash^?_{\mathrm{a}} \overline{T^?}\ \textbf{implements}\ I{<}\overline{U^?}{>} \rightarrowtriangle \mathcal{R}}$$

ENT-NIL-ALG-MAIN

$$\frac{\Delta;\emptyset;\texttt{false} \Vdash^?_{\mathrm{a}} \overline{T^?}\ \textbf{implements}\ I{<}\overline{U^?}{>} \rightarrowtriangle \mathcal{R}}{\Delta \Vdash^?_{\mathrm{a}} \overline{T^?}\ \textbf{implements}\ I{<}\overline{U^?}{>} \rightarrowtriangle \mathcal{R}}$$

ENT-NIL-ALG-ENV

$$\frac{R \in \Delta \qquad \overline{G}\ \textbf{implements}\ I{<}\overline{V}{>} \in \mathsf{sup}(R) \qquad \Delta;\beta;I \vdash^?_{\mathrm{a}} \overline{T^?} \uparrow \overline{G} \rightarrowtriangle \overline{T} \qquad (\forall i)\ V_i^? \sim V_i}{\Delta;\mathscr{G};\beta \Vdash^?_{\mathrm{a}} \overline{T^?}\ \textbf{implements}\ I{<}\overline{V^?}{>} \rightarrowtriangle \overline{T}\ \textbf{implements}\ I{<}\overline{V}{>}}$$

ENT-NIL-ALG-IFACE$_1$

$$\frac{\Delta;\beta;I \vdash_{\mathrm{a}} T \uparrow I{<}\overline{V}{>} \qquad 1 \in \mathsf{pol}^+(I) \qquad \mathsf{non\text{-}static}(I) \qquad (\forall i)\ V_i^? \sim V_i}{\Delta;\mathscr{G};\beta \Vdash^?_{\mathrm{a}} T\ \textbf{implements}\ I{<}\overline{V^?}{>} \rightarrowtriangle T\ \textbf{implements}\ I{<}\overline{V}{>}}$$

ENT-NIL-ALG-IFACE$_2$

$$\frac{1 \in \mathsf{pol}^+(I) \qquad \mathsf{non\text{-}static}(I) \qquad I{<}\overline{V}{>} \trianglelefteq_{\mathrm{i}} J{<}\overline{U}{>} \qquad (\forall i)\ U_i^? \sim U_i}{\Delta;\mathscr{G};\beta \Vdash^?_{\mathrm{a}} I{<}\overline{V}{>}\ \textbf{implements}\ J{<}\overline{U^?}{>} \rightarrowtriangle I{<}\overline{V}{>}\ \textbf{implements}\ J{<}\overline{U}{>}}$$

ENT-NIL-ALG-IMPL

$$\frac{\begin{array}{c}\textbf{implementation}{<}\overline{X}{>}\ I{<}\overline{V}{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots \qquad \Delta;\beta;I \vdash^?_{\mathrm{a}} \overline{T^?} \uparrow [\overline{U/X}]\overline{N} \rightarrowtriangle \overline{T}\\ (\forall i)\ V_i^? \sim [\overline{U/X}]V_i \qquad [\overline{U/X}]\overline{N}\ \textbf{implements}\ I{<}[\overline{U/X}]\overline{V}{>} \notin \mathscr{G}\\ \Delta;\mathscr{G} \cup \{[\overline{U/X}]\overline{N}\ \textbf{implements}\ I{<}[\overline{U/X}]\overline{V}{>}\};\texttt{false} \Vdash_{\mathrm{a}} [\overline{U/X}]\overline{P}\end{array}}{\Delta;\mathscr{G};\beta \Vdash^?_{\mathrm{a}} \overline{T^?}\ \textbf{implements}\ I{<}\overline{V^?}{>} \rightarrowtriangle \overline{T}\ \textbf{implements}\ I{<}[\overline{U/X}]\overline{V}{>}}$$

$$\boxed{\Delta;\beta;I \vdash^?_{\mathrm{a}} \overline{T^?} \uparrow \overline{U} \rightarrowtriangle \overline{V} \qquad T^? \sim T}$$

ENT-NIL-ALG-LIFT

$$\frac{\begin{array}{c}(\forall i)\ T_i^? = \mathsf{nil} \text{ or } \Delta \vdash_{\mathrm{q}}{}' T_i^? \le U_i\\ \beta \text{ or } \Big((\forall i) \text{ if } T_i^? \ne U_i \text{ and } T_i^? \ne \mathsf{nil} \text{ then } i \in \mathsf{pol}^-(I)\Big)\\ (\forall i)\ \text{ if } T_i^? = \mathsf{nil} \text{ then } V_i = U_i \text{ else } V_i = T_i^?\end{array}}{\Delta;\beta;I \vdash^?_{\mathrm{a}} \overline{T^?}^n \uparrow \overline{U}^n \rightarrowtriangle \overline{V}^n}$$

MATCHES-NIL

$$\mathsf{nil} \sim T$$

MATCHES-EQUAL

$$T \sim T$$

Figure 3.28 Auxiliaries for algorithmic method typing.

$$\boxed{\mathsf{bound}_{\Delta}(T) = N}$$

$$\textsc{bound}\quad \frac{\Delta \vdash_{\mathrm{q}}{}' \, T \leq N \qquad \text{if } \Delta \vdash_{\mathrm{q}}{}' \, T \leq N' \text{ then } N \trianglelefteq_{\mathsf{c}} N'}{\mathsf{bound}_{\Delta}(T) = N}$$

$$\boxed{\mathsf{pick\text{-}constr}^{k^?}_{\Delta} \mathscr{R} = \mathcal{R}}$$

$$\textsc{pick-constr-nil}\quad \frac{n \geq 1 \qquad i \in [n]}{\mathsf{pick\text{-}constr}^{\mathsf{nil}}_{\Delta} \{\overline{\mathcal{R}}^{n}\} = \mathcal{R}_i}$$

$$\textsc{pick-constr-non-nil}\quad \frac{n \geq 1 \qquad (\forall i \in [n])\ \Delta \vdash_{\mathrm{q}}{}' \, T_{jk} \leq T_{ik}}{\mathsf{pick\text{-}constr}^{k}_{\Delta} \{\overline{T_1}\ \mathbf{implements}\ K_1, \ldots, \overline{T_n}\ \mathbf{implements}\ K_n\} = \overline{T_j}\ \mathbf{implements}\ K_j}$$

$$\boxed{\mathsf{sresolve}_{\Delta;X}(\overline{T}, \overline{T}) = \mathscr{T}}$$

$$\textsc{sresolve-non-empty}\quad \frac{\mathscr{C} = \{T_i \mid i \in [n], U_i = X\} \qquad \mathscr{C} \neq \emptyset \qquad \mathscr{T} = \mathsf{mub}_{\Delta}(\mathscr{C})}{\mathsf{sresolve}_{\Delta;X}(\overline{U}^{n}, \overline{T}^{n}) = \mathscr{T}}$$

$$\textsc{sresolve-empty}\quad \frac{\{T_i \mid i \in [n], U_i = X\} = \emptyset}{\mathsf{sresolve}_{\Delta;X}(\overline{U}^{n}, \overline{T}^{n}) = \emptyset}$$

$$\boxed{\mathsf{mub}_{\Delta}(\mathscr{T}) = \mathscr{T}}$$

$$\textsc{mub}\quad \frac{\begin{array}{c} \mathscr{V} = \{V \mid (\forall T \in \mathscr{T}), \Delta \vdash_{\mathrm{q}}{}' \, T \leq V\} \\ \mathscr{U} = \{V \in \mathscr{V} \mid (\forall V' \in \mathscr{V} \setminus \{V\}) \text{ not } \Delta \vdash_{\mathrm{q}}{}' \, V' \leq V\} \end{array}}{\mathsf{mub}_{\Delta}(\mathscr{T}) = \mathscr{U}}$$

Figure 3.29 Algorithmic method typing.

$$\boxed{\textsf{a-mtype}_{\Delta}(m, T, \overline{T}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}}}$$

ALG-MTYPE-CLASS

$$\frac{\textsf{bound}_{\Delta}(T) = N \qquad \textsf{a-mtype}^{\mathrm{c}}(m^{\mathrm{c}}, N) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}}}{\textsf{a-mtype}_{\Delta}(m^{\mathrm{c}}, T, \overline{T}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}}}$$

ALG-MTYPE-IFACE

$$\frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}^{l}\ \textbf{where}\ \overline{R}\,]\ \textbf{where}\ \overline{P}\,\{\,\ldots\ \overline{rcsig}\,\}\\ rcsig_{j} = \textbf{receiver}\ \{\overline{m : msig}\} \qquad m^{\mathrm{i}} = m_{k} \qquad msig_{k} = \texttt{<}\overline{Y}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{Q}\\ (\forall i \in [l], i \neq j)\ \textsf{sresolve}_{\Delta;Z_i}(\overline{U}, \overline{T}) = \mathscr{V}_{i} \qquad \textsf{sresolve}_{\Delta;Z_j}(Z_j\,\overline{U}, T\,\overline{T}) = \mathscr{V}_{j}\\ p^{?} = (\text{if } U = Z_i \text{ for some } i \in [l] \text{ then } i \text{ else } \textsf{nil})\\ \overline{W}\ \textbf{implements}\ I\texttt{<}\overline{W'}\texttt{>} =\\ \textsf{pick-constr}^{p^{?}}_{\Delta}\,\{\overline{V}\ \textbf{implements}\ I\texttt{<}\overline{V''}\texttt{>}\\ \mid (\forall i \in [l])\ \text{if } \mathscr{V}_{i} = \emptyset \text{ then } V^{?}_{i} = \textsf{nil}\\ \text{else define } V^{?}_{i} \text{ such that}\\ \Delta \vdash_{\mathrm{q}}{}'\ V'_{i} \leq V^{?}_{i} \text{ for some } V'_{i} \in \mathscr{V}_{i},\\ \Delta \Vdash^{?}_{\mathrm{a}}\ \overline{V^{?}}\ \textbf{implements}\ I\texttt{<}\overline{\textsf{nil}}\texttt{>} \twoheadrightarrow \overline{V}\ \textbf{implements}\ I\texttt{<}\overline{V''}\texttt{>}\}\end{array}}{\textsf{a-mtype}_{\Delta}(m^{\mathrm{i}}, T, \overline{T}) = [\overline{W/Z}, \overline{W'/Z'}]\,msig_{k}}$$

$$\boxed{\textsf{a-smtype}_{\Delta}(m, K[\overline{T}]) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}}}$$

ALG-MTYPE-STATIC

$$\frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\,\overline{Y}\ \textbf{where}\ \overline{R}\,]\ \textbf{where}\ \overline{P}\,\{\,\overline{m : \textbf{static}\ msig}\ \ldots\}\\ \Delta \Vdash_{\mathrm{a}} \overline{T}\ \textbf{implements}\ I\texttt{<}\overline{U}\texttt{>}\end{array}}{\textsf{a-smtype}_{\Delta}(m^{\mathrm{i}}_{k}, I\texttt{<}\overline{U}\texttt{>}[\overline{T}]) = [\overline{U/X}, \overline{T/Y}]\,msig_{k}}$$

$$\boxed{\textsf{a-mtype}^{\mathrm{c}}(m, N) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}}}$$

ALG-MTYPE-CLASS-BASE

$$\frac{\textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\,\overline{T\,f}\ \overline{m : mdef}\,\} \qquad mdef_{i} = msig\,\{e\}}{\textsf{a-mtype}^{\mathrm{c}}(m_{i}, C\texttt{<}\overline{T}\texttt{>}) = [\overline{T/X}]\,msig}$$

ALG-MTYPE-CLASS-SUPER

$$\frac{\begin{array}{c}\textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\,\overline{T\,f}\ \overline{m : mdef}\,\}\\ m \notin \overline{m} \qquad \textsf{a-mtype}^{\mathrm{c}}(m, [\overline{T/X}]N) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}}\end{array}}{\textsf{a-mtype}^{\mathrm{c}}(m, C\texttt{<}\overline{T}\texttt{>}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}}}$$

relation $\textsf{a-smtype}_\Delta(m, I\texttt{<}\overline{U}\texttt{>}[\overline{T}])$ determines the signature of a static interface method m for interface $I\texttt{<}\overline{U}\texttt{>}$ and implementing types $\overline{T}$. The definition of $\textsf{a-smtype}$ is straightforward, the one for $\textsf{a-mtype}$ requires several auxiliaries from Figure 3.28:

- $\textsf{a-mtype}^{\textsf{c}}(m, N) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U$ **where** $\overline{\mathcal{P}}$ determines the signature of a class method m by ascending the inheritance hierarchy starting at class N. The $\textsf{a-mtype}^{\textsf{c}}$ relation is very similar to the method typing relation of Featherweight Generic Java [96].
- $\textsf{bound}_\Delta(T) = N$ computes the bound N of a type T with respect to a type environment Δ.
- $\textsf{pick-constr}^{k^?}_\Delta \mathscr{R} = \mathcal{R}$ takes a set $\mathscr{R}$ of $\mathcal{R}$-constraints, a type environment Δ, and an optional index $k^?$. If $k = \textsf{nil}$ and $\mathscr{R} \neq \emptyset$, $\textsf{pick-constr}$ returns an arbitrary constraint $\mathcal{R} \in \mathscr{R}$. If $k \in \mathbb{N}$ and $\mathscr{R} \neq \emptyset$, it returns a constraint $\mathcal{R} \in \mathscr{R}$ such that the kth implementing type of $\mathcal{R}$ is minimal with respect to the kth implementing types of all other constraints in $\mathscr{R}$.
- $\textsf{sresolve}_{\Delta;X}(\overline{U}, \overline{T}) = \mathscr{T}$ is the static analogon of $\textsf{resolve}$ from Figure 3.5 on page 39. It resolves implementing type X with respect to formal parameter types $\overline{U}$, the static types $\overline{T}$ of the actual parameters, and type environment Δ. Whereas $\textsf{resolve}$ returns an optional type (the least upper bound, if existing, of a set of class types), $\textsf{sresolve}$ returns a set of types (the minimal elements of the upper bounds of all static parameter types contributing to the resolution of X).
- $\textsf{mub}_\Delta(\mathscr{T}) = \mathscr{U}$ takes a set of types $\mathscr{T}$ and returns a set of types $\mathscr{U}$ containing the minimal elements of the upper bounds of all types in $\mathscr{T}$.

The definition of $\textsf{a-mtype}$ for class methods relies on $\textsf{a-mtype}^{\textsf{c}}$ to find the signature of the method in question. The definition of $\textsf{a-mtype}$ for interface methods is more involved:

- First, $\textsf{a-mtype}$ retrieves interface I and receiver $rcsig_j$ defining method m.
- Then, it uses $\textsf{sresolve}$ to compute, for each implementing type variable Z_i, a set $\mathscr{V}_i$. This set contains the minimal elements of the upper bounds of all static argument types that contribute to the resolution of the ith implementing type.
- Next, it collects all implementation constraints for I that are entailed by Δ and that match the $\mathscr{V}_i$ pointwise. This step also infers unknown types.
- Finally, $\textsf{a-mtype}$ uses $\textsf{pick-constr}^{p^?}_\Delta$ to pick an element from the collected constraints. To minimize the result type of the signature computed by $\textsf{a-mtype}$, $p^? \neq \textsf{nil}$ if, and only if, the signature declared in the interface uses the pth implementing type as its result type. (Criterion WF-IFACE-3 ensures that implementing types do not occur nested inside the result type.)

Definition 3.30. A type environment Δ is well-formed, written $\vdash \Delta\ \textsf{ok}$ if, and only if, $\Delta \vdash P\ \textsf{ok}$ for all $P \in \Delta$.

Figure 3.30 Algorithmic expression typing.

$$\boxed{\Delta;\Gamma \vdash_{\mathrm{a}} e : T}$$

EXP-ALG-VAR

$$\Delta;\Gamma \vdash_{\mathrm{a}} x : \Gamma(x)$$

EXP-ALG-FIELD

$$\frac{\Delta;\Gamma \vdash_{\mathrm{a}} e : T \qquad \mathsf{bound}_{\Delta}(T) = N \qquad \mathsf{fields}(N) = \overline{U\,f}}{\Delta;\Gamma \vdash_{\mathrm{a}} e.f_j : U_j}$$

EXP-ALG-INVOKE

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash_{\mathrm{a}} e : T \qquad (\forall i)\ \Delta;\Gamma \vdash_{\mathrm{a}} e_i : T_i \qquad \mathsf{a\text{-}mtype}_{\Delta}(m, T, \overline{T}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \ \mathbf{where}\ \overline{\mathcal{P}} \\ (\forall i)\ \Delta \vdash_{\mathrm{a}} T_i \leq [\overline{V/X}]U_i \qquad \Delta \Vdash_{\mathrm{a}} [\overline{V/X}]\overline{\mathcal{P}} \qquad \Delta \vdash_{\mathrm{a}} \overline{V}\ \mathsf{ok}\end{array}}{\Delta;\Gamma \vdash_{\mathrm{a}} e.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V/X}]U}$$

EXP-ALG-INVOKE-STATIC

$$\frac{\begin{array}{c}\mathsf{a\text{-}smtype}_{\Delta}(m, I\texttt{<}\overline{W}\texttt{>}[\overline{T}]) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \mathbf{where}\ \overline{\mathcal{P}} \\ (\forall i)\ \Delta;\Gamma \vdash_{\mathrm{a}} e_i : U_i' \qquad (\forall i)\ \Delta \vdash_{\mathrm{a}} U_i' \leq [\overline{V/X}]U_i \qquad \Delta \Vdash_{\mathrm{a}} [\overline{V/X}]\overline{\mathcal{P}} \qquad \Delta \vdash_{\mathrm{a}} \overline{T}, \overline{V}\ \mathsf{ok}\end{array}}{\Delta;\Gamma \vdash_{\mathrm{a}} I\texttt{<}\overline{W}\texttt{>}[\overline{T}].m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V/X}]U}$$

EXP-ALG-NEW

$$\frac{(\forall i)\ \Delta;\Gamma \vdash_{\mathrm{a}} e_i : T_i \qquad \Delta \vdash_{\mathrm{a}} N\ \mathsf{ok} \qquad \mathsf{fields}(N) = \overline{U\,f} \qquad (\forall i)\ \Delta \vdash_{\mathrm{a}} T_i \leq U_i}{\Delta;\Gamma \vdash_{\mathrm{a}} \mathbf{new}\ N(\overline{e}) : N}$$

EXP-ALG-CAST

$$\frac{\Delta \vdash_{\mathrm{a}} T\ \mathsf{ok} \qquad \Delta;\Gamma \vdash_{\mathrm{a}} e : U}{\Delta;\Gamma \vdash_{\mathrm{a}} (T)\,e : T}$$

Theorem 3.31 (Soundness of algorithmic method typing). *Assume that* $\vdash \Delta$ ok *and* $\Delta \vdash T, \overline{T}$ ok. *If* $\mathsf{a\text{-}mtype}_{\Delta}(m, T, \overline{T}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \mathbf{where}\ \overline{\mathcal{P}}$ *then there exists a type* T' *such that* $\Delta \vdash T \leq T'$ *and* $\mathsf{mtype}_{\Delta}(m, T') = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \mathbf{where}\ \overline{\mathcal{P}}$.

Proof. See Section B.5.3. □

Theorem 3.32 (Completeness of algorithmic method typing). *Assume* $\mathsf{mtype}_{\Delta}(m, T) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x}^{n} \to U\ \mathbf{where}\ \overline{\mathcal{P}}$ *and let* φ *be a substitution* $[\overline{V/X}]$. *Furthermore, suppose* $\vdash \Delta$ ok *and* $\Delta \vdash T'$ ok. *If* $\Delta \vdash T' \leq T$ *and* $\Delta \vdash T_i \leq \varphi U_i$ *for all* $i \in [n]$ *and* $\Delta \Vdash \varphi\overline{\mathcal{P}}$, *then* $\mathsf{a\text{-}mtype}_{\Delta}(m, T', \overline{T}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U'\,x}^{n} \to U'\ \mathbf{where}\ \overline{\mathcal{P}}$ *such that* $\Delta \vdash T_i \leq \varphi U_i'$ *for all* $i \in [n]$ *and* $\Delta \vdash \varphi U' \leq \varphi U$.

Proof. See Section B.5.4. □

Algorithmic Expression Typing

With algorithmic method typing in hand, the definition of an algorithm for typechecking expressions is straightforward and follows closely the approach taken by Featherweight

Generic Java [96]. Figure 3.30 presents the relation $\Delta;\Gamma \vdash_{\mathrm{a}} e : T$ that assigns type T to expression e under type environment Δ and variable environment Γ. The rules defining the relation are syntax-directed and easy to implement. They rely on algorithmic formulations of the well-formedness judgments from Figure 3.7 on 42:

Definition 3.33. The relations $\Delta \vdash_{\mathrm{a}} T\ \mathsf{ok}$ and $\Delta \vdash_{\mathrm{a}} \mathcal{P}\ \mathsf{ok}$ are defined analogously to the relations $\Delta \vdash T\ \mathsf{ok}$ and $\Delta \vdash \mathcal{P}\ \mathsf{ok}$, respectively, replacing $\vdash$ with $\vdash_{\mathrm{a}}$ and $\Vdash$ with $\Vdash_{\mathrm{a}}$.

Algorithmic expression typing is equivalent to the declarative specification of expression typing in Figure 3.9.

Definition 3.34. A variable environment Γ is well-formed under type environment Δ, written $\Delta \vdash \Gamma\ \mathsf{ok}$, if, and only if, $\Delta \vdash T :\ \mathsf{ok}$ for all $x : T$ occurring in Γ.

Theorem 3.35 (Soundness of algorithmic expression typing). *Suppose* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash \Gamma\ \mathsf{ok}$*. If* $\Delta;\Gamma \vdash_{\mathrm{a}} e : T$ *then* $\Delta;\Gamma \vdash e : T$*.*

Proof. The proof is by induction on the derivation of $\Delta;\Gamma \vdash_{\mathrm{a}} e : T$. See Section B.5.5 for details. □

Theorem 3.36 (Completeness of algorithmic expression typing). *Assume* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash \Gamma\ \mathsf{ok}$*. If* $\Delta;\Gamma \vdash e : T$ *then* $\Delta;\Gamma \vdash_{\mathrm{a}} e : U$ *such that* $\Delta \vdash U \leq T$*.*

Proof. The proof is by induction on the derivation of $\Delta;\Gamma \vdash e : T$. See Section B.5.6 for details. □

Algorithmic expression typing also terminates.

Theorem 3.37. *The algorithm induced by the rules in Figures 3.27, 3.28, 3.29, and 3.30 terminates.*

Proof. See Section B.5.7. □

3.7.3 Deciding Program Typing

Given the algorithms for constraint entailment, subtyping, and expression typing, implementing a typechecker for CoreGI programs is almost straightforward, only the implementation of well-formedness criteria WF-PROG-2, WF-PROG-3, WF-PROG-4, WF-TENV-2, and WF-TENV-6(2) poses a challenge.

Checking WF-PROG-2, WF-PROG-3, WF-PROG-4, WF-TENV-6(2)

A direct implementation of these criteria is not possible because their definition involves universal quantification over substitutions subject to subtype or greatest lower bound conditions.

Definition 3.38 (Unification modulo greatest lower bounds). A *unification problem modulo greatest lower bounds* is a triple $\mathbb{L} = \big(\Delta, \overline{X}, \{G_1 \sqcap^? H_1, \ldots G_n \sqcap^? H_n\}\big)$ such that $\mathsf{ftv}(\Delta) \cap \overline{X} = \emptyset$ and $G_i = Y$ (or $H_i = Y$) implies $Y \notin \overline{X}$ for all $i \in [n]$. A *solution* of $\mathbb{L}$ is a substitution $\varphi = [\overline{V/X}]$ such that $\Delta \vdash \varphi T_i \sqcap \varphi U_i$ exists for all $i = 1, \ldots, n$. A *most-general solution* of $\mathbb{L}$ is a solution that is more general than any other solution of $\mathbb{L}$ (see Definition 3.21).

Obviously, a solution of $(\Delta, \overline{X}, \{G_{11} \sqcap^? G_{12}, \dots, G_{n1} \sqcap^? G_{n2}\})$ also solves the unification problem modulo kernel subtyping $(\Delta, \overline{X}, \{G_{1i_1} \leq^? G_{1j_1}, \dots, G_{ni_n} \leq^? G_{nj_n}\})$ for some set of pairs $\{(i_1, j_1), \dots, (i_n, j_n)\}$ where $(i_k, j_k) \in \{(1,2),(2,1)\}$ for all $k \in [n]$. Thus, a naive algorithm for solving unification modulo greatest lower bounds simply enumerates all of these unification problems modulo kernel subtyping and checks whether any of them has a solution φ. If so, it returns φ and fails otherwise. Call this naive algorithm $\texttt{unify}_\sqcap$.

Theorem 3.39 (Soundness, completeness, and termination of $\texttt{unify}_\sqcap$). *Let $\mathbb{L}$ be a unification problem modulo greatest lower bounds. If $\mathbb{L}$ has a solution then $\texttt{unify}_\sqcap(\mathbb{L})$ returns an idempotent, most general solution of $\mathbb{L}$. If $\mathbb{L}$ does not have a solution, $\texttt{unify}_\sqcap(\mathbb{L})$ terminates with a failure.*

Proof. See Section B.6.1. □

The following alternative formulations of WF-PROG-2, WF-PROG-3, WF-PROG-4, and WF-TENV-6(2) are straightforward to implement.

WF-PROG-2′ For each pair of disjoint implementation definitions

$$\textbf{implementation}{<}\overline{X}{>}\ I{<}\overline{T}{>}\ [\,\overline{M}\,]\ \textbf{where}\ \overline{P}\ \dots$$

$$\textbf{implementation}{<}\overline{Y}{>}\ I{<}\overline{U}{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{Q}\ \dots$$

with $\overline{X} \cap \overline{Y} = \emptyset$ and $\texttt{unify}_\sqcap(\emptyset, \overline{X}\,\overline{Y}, \{M_i \sqcap^? N_i \mid i \in \mathsf{disp}(I)\}) = \varphi$, it holds that $\varphi\overline{T} = \varphi\overline{U}$ and that $\varphi M_j = \varphi N_j$ for all $j \notin \mathsf{disp}(I)$.

WF-PROG-3′ For each pair of disjoint implementation definitions

$$\textbf{implementation}{<}\overline{X}{>}\ I{<}\overline{T}{>}\ [\,\overline{N}^n\,]\ \textbf{where}\ \overline{P}\ \dots$$

$$\textbf{implementation}{<}\overline{X'}{>}\ I{<}\overline{T'}{>}\ [\,\overline{N'}^n\,]\ \textbf{where}\ \overline{P'}\ \dots$$

with $\overline{X} \cap \overline{X'} = \emptyset$ and $\texttt{unify}_\sqcap(\emptyset, \overline{X}\,\overline{X'}, \{N_i \sqcap^? N'_i \mid i \in [n]\}) = \varphi$, there exists an implementation definition

$$\textbf{implementation}{<}\overline{Y}{>}\ I{<}\overline{U}{>}\ [\,\overline{M}\,]\ \textbf{where}\ \overline{Q}\ \dots$$

and a substitution $[\overline{W/Y}]$ such that $\emptyset \vdash \varphi\overline{N} \sqcap \varphi\overline{N'} = [\overline{W/Y}]\overline{M}$.

WF-PROG-4′ For each pair of disjoint implementation definitions

$$\textbf{implementation}{<}\overline{X}{>}\ I{<}\overline{T}{>}\ [\,\overline{M}\,]\ \textbf{where}\ \overline{P}\ \dots$$

$$\textbf{implementation}{<}\overline{Y}{>}\ I{<}\overline{U}{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{Q}\ \dots$$

with $\overline{X} \cap \overline{X'} = \emptyset$ and $\texttt{unify}_\leq(\emptyset, \overline{X}\,\overline{Y}, \{M_i \leq^? N_i \mid i \in [n]\}) = \varphi$, it holds that for all $\mathcal{P} \in \varphi\overline{P}$ either $\{Q \in \varphi\overline{Q}\} \Vdash \mathcal{P}$ or $\mathcal{P} \in \varphi\overline{Q} \cup \mathsf{sup}(\varphi\overline{Q}) \cup \{T\,\textbf{extends}\,U \mid T\,\textbf{extends}\,U' \in \varphi\overline{Q}, \{Q \in \varphi\overline{Q}\} \vdash_{\mathrm{q}}{}' U' \leq U\}$.

WF-TENV-6′

1. Unchanged from criterion WF-TENV-6.
2. For each constraint and each implementation definition

$$\overline{G}\,\textbf{implements}\; I\texttt{<}\overline{T}\texttt{>} \in \mathsf{sup}(\Delta)$$

$$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\; I\texttt{<}\overline{W}\texttt{>}\; [\,\overline{N}\,]\;\textbf{where}\;\overline{P}\;\ldots$$

   with $\overline{X} \cap (\bigcup\{\mathsf{ftv}(\mathcal{S}) \mid \mathcal{R} \in \Delta, \mathcal{S} \in \mathsf{sup}(\mathcal{R})\}) = \emptyset$ and $\mathtt{unify}_{\sqcap}(\Delta, \overline{X}, \{G_i \sqcap^? N_i \mid i \in \mathsf{disp}(I)\}) = \varphi$, it holds that $\overline{T} = \varphi\overline{W}$ and $G_j = \varphi N_j$ for all $j \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$.

Theorem 3.40. *Criteria* WF-PROG-2′*,* WF-PROG-3′*, and* WF-TENV-6′ *are equivalent to their counterparts from Section 3.5.3. Criterion* WF-PROG-4′ *is sound with respect to* WF-PROG-4 *(i.e.,* WF-PROG-4′ *implies* WF-PROG-4*).*

Proof. See Section B.6.2. □

It is an open question whether there exists a complete algorithm for checking well-formedness criterion WF-PROG-4.

Checking WF-TENV-2

This criterion requires the closure of a finite set of types to be finite. Thanks to Viroli [232], there is an equivalent but syntactic characterization of this property. Roughly speaking, Viroli's approach defines a dependency graph between the formal type parameters of all classes such that finitary closure of a finite set of types is equivalent to the absence of certain cycles in the dependency graph. Section B.7 recasts Viroli's approach and shows that it leads to an equivalent and implementable formulation of well-formedness criterion WF-TENV-2.

Concluding Remarks

This chapter formalized CoreGI, a small calculus capturing most aspects of the generalized interface mechanism of JavaGI. The formalization included the definition of CoreGI's dynamic semantics and a declarative specification of its static semantics. Two important properties hold for a well-typed CoreGI program: evaluation is deterministic and evaluation cannot get stuck if all cast operations succeed.

Besides proving these properties, the chapter also demonstrated how to typecheck CoreGI programs. To this end, algorithmic formulations of constraint entailment, subtyping, method typing, expression typing, and program typing were presented.

4 Translation

The preceding chapter formalized the static and the dynamic semantics of CoreGI, a small calculus capturing most aspects of the full JavaGI language. Such a formalization is important to gain assurance that JavaGI programs per se do not behave in unexpected ways. However, JavaGI programs are not executed by some custom interpreter but compiled to standard Java byte code and executed on the Java Virtual Machine [125]. Thus, it is also important to verify that the compilation step does not change the behavior of JavaGI programs. To this end, the present chapter formalizes a translation from a significant subset of CoreGI to a slightly extended version of Featherweight Java [96]. It suffices to consider such a source-to-source translation because the main challenge in the implementation of a compiler for JavaGI is the mapping from JavaGI to plain Java constructs. The actual generation of byte code is standard.

Chapter Outline. The chapter is divided into five sections:

- Section 4.1 introduces CoreGI$^\flat$, the source language of the translation. The section defines the syntax and the dynamic semantics of CoreGI$^\flat$ but defers the definition of its static semantics until Section 4.3.

- Section 4.2 formalizes iFJ, the target language of the translation. The formalization includes syntax, dynamic semantics, static semantics, and a proof of type soundness.

- Section 4.3 defines a type-directed translation from CoreGI$^\flat$ to iFJ, which also serves as the definition of the static semantics of CoreGI$^\flat$.

- Section 4.4 proves that the translation from CoreGI$^\flat$ to iFJ preserves the static and the dynamic semantics of CoreGI$^\flat$.

- Section 4.5 shows that CoreGI$^\flat$ is indeed a subset of CoreGI, a fact that implies type soundness and determinacy of evaluation for CoreGI$^\flat$.

Figure 4.1 Syntax of CoreGI$^\flat$.

$$
\begin{aligned}
\mathit{prog} &::= \overline{\mathit{def}}\ e \\
\mathit{def} &::= \mathit{cdef} \mid \mathit{idef} \mid \mathit{impl} \\
\mathit{cdef} &::= \textbf{class}\ C\ \textbf{extends}\ N\,\{\,\overline{T\,f}\ \overline{m : \mathit{mdef}}\,\} \\
\mathit{idef} &::= \textbf{interface}\ I\ \textbf{extends}\ \overline{I}\,\{\,\overline{m : \mathit{msig}}\,\} \\
\mathit{impl} &::= \textbf{implementation}\ I\ [\,N\,]\,\{\,\overline{\mathit{mdef}}\,\} \\
\mathit{msig} &::= \overline{T\,x} \to T \\
\mathit{mdef} &::= \mathit{msig}\,\{e\} \\
M, N &::= C \mid \mathit{Object} \\
T, U, V, W &::= N \mid I \\
d, e &::= x \mid e.f \mid e.m(\overline{e}) \mid \textbf{new}\ N(\overline{e}) \mid (T)\,e
\end{aligned}
$$

$$
C, D \in \mathit{ClassName} \quad I, J \in \mathit{IfaceName}
$$

$$
m \in \mathit{MethodName} \quad f, g \in \mathit{FieldName} \quad x, y, z \in \mathit{VarName}
$$

4.1 Source Language: CoreGI$^\flat$

To keep the formal setup within reasonable size and complexity limits, the translation presented in this chapter considers only a simplified version of CoreGI as its source language. The source language, dubbed CoreGI$^\flat$, does not support type variables, constraints, explicit implementing types, multi-headed interfaces, static interface methods, and covariant return types because these features do not pose significant challenges to the full translation from JavaGI to Java. However, CoreGI$^\flat$ supports retroactive interface implementations, which are the most difficult part of the full translation.

The definition of CoreGI$^\flat$ in this section comprises only the syntax (Section 4.1.1) and the dynamic semantics (Section 4.1.2). Section 4.3 completes the definition by specifying a static semantics, which is interweaved with the translation from CoreGI$^\flat$ to iFJ.

4.1.1 Syntax

Figure 4.1 defines the abstract syntax of CoreGI$^\flat$. As in Chapter 3, overbar notation denotes sequencing (see Definition 3.1) and the various kinds of identifiers are drawn from pairwise disjoint and countably infinite sets of class names (ranged over by C, D), interface names (ranged over by I, J), method names (ranged over by m), field names (ranged over by f, g), and variable names (ranged over by x, y, z). CoreGI$^\flat$ shares the identifier sets for class, interface, method, field, and variable names with CoreGI.

A CoreGI$^\flat$ program *prog* consists of a sequence of definitions *def* followed by a “main” expression e. A definition is either a class, interface, or implementation definition.

Each class C has an explicit superclass N, where N is a class type (either an instantiated class or *Object*). If the superclass is *Object*, we sometimes omit the **extends** clause altogether. The predefined class *Object* does not have a superclass and it does not define any fields or methods. The body of a class contains a sequence of field definitions $T\,f$, where T is a type and f the name of the field, followed by a sequence of method defini-

Figure 4.2 Class and interface inheritance for CoreGI$^\flat$.

$$\boxed{N \trianglelefteq^\flat_{\mathbf{c}} M}$$

$$\textsc{inh-class-refl}^\flat \quad N \trianglelefteq^\flat_{\mathbf{c}} N \qquad\qquad \textsc{inh-class-super}^\flat \quad \frac{\textbf{class } C \textbf{ extends } M \ldots \qquad M \trianglelefteq^\flat_{\mathbf{c}} N}{C \trianglelefteq^\flat_{\mathbf{c}} N}$$

$$\boxed{I \trianglelefteq^\flat_{\mathbf{i}} J}$$

$$\textsc{inh-iface-refl}^\flat \quad I \trianglelefteq^\flat_{\mathbf{i}} I \qquad\qquad \textsc{inh-iface-super}^\flat \quad \frac{\textbf{interface } I \textbf{ extends } \overline{J} \ldots \qquad J_i \trianglelefteq^\flat_{\mathbf{i}} I'}{I \trianglelefteq^\flat_{\mathbf{i}} I'}$$

tions $m : \mathit{mdef}$, where m is the method name and mdef specifies the signature msig and the body expression e of the method. The signature of a method consists of arguments $\overline{x}$ together with their types $\overline{T}$ and the result type T.

The definition of an interface I specifies its superinterfaces $\overline{J}$ through an **extends** clause, which we omit if $\overline{J} = \bullet$. An interface definition also lists the names and the signatures of the methods supported by the interface. For the names of interface methods the following conventions apply:

Convention 4.1 (Disjoint namespaces for class and interface methods). The namespaces for class and interface methods are disjoint. At some points, m^{c} or m^{i} explicitly denotes the name of a class or interface method, respectively.

Convention 4.2 (Globally unique names of interface methods). The names of interface methods are globally unique; that is, if some interface defines a method m then no other interface defines a method with the same name m.

A retroactive implementation definition impl specifies an implementation of interface I for implementing type N. The body of an implementation contains definitions for the methods of I. These definitions are anonymous because they are matched by position against the methods declared in I.

Metavariables M, N range over class types, whereas full types (ranged over by T, U, V, W) also include interface types. Expressions d, e include variables, field accesses, method calls, object allocations, and casts. By convention, syntactic constructs that differ only in the names of bound expression variables are interchangeable in all contexts [176].

4.1.2 Dynamic Semantics

Dynamic method lookup in CoreGI$^\flat$ depends on the class inheritance relation $N \trianglelefteq^\flat_{\mathbf{c}} M$, which expresses that class type N is a subclass of type M. Figure 4.2 defines this relation together with the inheritance relation on interfaces $I \trianglelefteq^\flat_{\mathbf{i}} J$, which expresses that interface

Figure 4.3 Dynamic method lookup for $\mathsf{CoreGI}^\flat$.

$$\boxed{\mathsf{getmdef}^\flat(m, N) = \mathit{mdef}}$$

$$\textsc{dyn-mdef-class-base}^\flat\quad \frac{\textbf{class } C \textbf{ extends } N \,\{\, \overline{T\,f}\ \overline{m : \mathit{mdef}} \,\}}{\mathsf{getmdef}^\flat(m_j, C) = \mathit{mdef}_j}$$

$$\textsc{dyn-mdef-class-super}^\flat\quad \frac{\begin{array}{c}\textbf{class } C \textbf{ extends } N \,\{\, \overline{T\,f}\ \overline{m : \mathit{mdef}} \,\} \\ m^{\mathsf{c}} \notin \overline{m} \qquad \mathsf{getmdef}^\flat(m^{\mathsf{c}}, N) = \mathit{mdef}\end{array}}{\mathsf{getmdef}^\flat(m^{\mathsf{c}}, C) = \mathit{mdef}}$$

$$\textsc{dyn-mdef-iface}^\flat\quad \frac{\begin{array}{c}\textbf{interface } I \textbf{ extends } \overline{I} \,\{\, \overline{m : \mathit{msig}} \,\} \\ \mathsf{least\text{-}impl}^\flat\{\textbf{implementation } I\ [\,M\,]\ \ldots \mid N \trianglelefteq^\flat_{\mathsf{c}} M\} \\ = \textbf{implementation } I\ [\,M\,]\,\{\, \overline{m : \mathit{mdef}} \,\}\end{array}}{\mathsf{getmdef}^\flat(m_k, N) = \mathit{mdef}_k}$$

$$\boxed{\mathsf{least\text{-}impl}^\flat\{\overline{\mathit{impl}}\} = \mathit{impl}}$$

$$\textsc{least-impl}^\flat\quad \frac{\mathit{impl}_i = \textbf{implementation } I\ [\,N_i\,]\ \ldots \qquad n \geq 1 \qquad (\forall i \in [n])\ N_k \trianglelefteq^\flat_{\mathsf{c}} N_i}{\mathsf{least\text{-}impl}^\flat\{\mathit{impl}_1, \ldots, \mathit{impl}_n\} = \mathit{impl}_k}$$

type I is a subinterface of J. The definition of $\trianglelefteq^\flat_{\mathsf{c}}$ and $\trianglelefteq^\flat_{\mathsf{i}}$ is straightforward and similar to that of the corresponding relations $\trianglelefteq_{\mathsf{c}}$ and $\trianglelefteq_{\mathsf{i}}$ for CoreGI as defined in Figure 3.16.

Figure 4.3 defines dynamic method lookup for $\mathsf{CoreGI}^\flat$. The $\mathsf{getmdef}^\flat(m, N)$ relation searches for a definition of method m for receiver of run-time type N. If m is a class method, $\mathsf{getmdef}^\flat$ first retrieves the definition of m directly from N (rule $\textsc{dyn-mdef-class-base}^\flat$). If this fails, $\mathsf{getmdef}^\flat$ continues searching in N's superclass (rule $\textsc{dyn-mdef-class-super}^\flat$). The search stops when it reaches *Object* because there is no matching rule. Rule $\textsc{dyn-mdef-iface}^\flat$ handles the case where m is not a class but an interface method. The rule first collects all implementations whose implementing types are superclasses of N. Among these implementations, the rule then chooses the minimal one by using the $\mathsf{least\text{-}impl}^\flat$ auxiliary, which Figure 4.3 defines as well.

To properly support run-time casts, $\mathsf{CoreGI}^\flat$'s dynamic semantics makes use of the subtyping relation defined in Figure 4.4. As in Chapter 3, the figure uses the notation $\xi^?$ to denote an optional construct: $\xi^?$ is either a regular ξ or the special symbol nil. The relation $\vdash^{\flat'} T \leq U$ is the *kernel* of $\mathsf{CoreGI}^\flat$ subtyping. The full subtyping relation $\vdash^\flat T \leq U \rightsquigarrow I^?$ establishes a subtyping relationship between types T and U. The "$\rightsquigarrow I^?$" part specifies whether this relationship depends on a retroactive interface implementation; it is only relevant for the translation given in Section 4.3. Other places simply omit this part and use $\vdash^\flat T \leq U$ to abbreviate $\vdash^\flat T \leq U \rightsquigarrow I^?$ for some fresh metavariable I.

Figure 4.5 specifies the dynamic semantics of $\mathsf{CoreGI}^\flat$. The definition of values (ranged over by v, w) and of call-by-value evaluation contexts (denoted by $\mathcal{E}$) is standard. The

Figure 4.4 Subtyping for CoreGI$^\flat$.

$$\boxed{\vdash^{\flat'} T \leq U}$$

$$\textsc{sub-object}^\flat \quad \vdash^{\flat'} T \leq \mathit{Object} \qquad \textsc{sub-class}^\flat \quad \frac{C \trianglelefteq^\flat_{\mathrm{c}} C'}{\vdash^{\flat'} C \leq C'} \qquad \textsc{sub-iface}^\flat \quad \frac{I \trianglelefteq^\flat_{\mathrm{i}} I'}{\vdash^{\flat'} I \leq I'}$$

$$\boxed{\vdash^\flat T \leq U \rightsquigarrow I^?}$$

$$\textsc{sub-kernel}^\flat \quad \frac{\vdash^{\flat'} T \leq U}{\vdash^\flat T \leq U \rightsquigarrow \mathsf{nil}} \qquad \textsc{sub-impl}^\flat \quad \frac{\vdash^{\flat'} T \leq N \qquad \textbf{implementation}\ I\ [\,N\,]\ \ldots}{\vdash^\flat T \leq I \rightsquigarrow I}$$

Figure 4.5 Dynamic semantics of CoreGI$^\flat$.

$$\boxed{\text{Values and evaluation contexts}}$$

$$\begin{aligned} v, w &::= \textbf{new}\ N(\overline{v}) \\ \mathcal{E} &::= \square \mid \mathcal{E}.f \mid \mathcal{E}.m(\overline{e}) \mid v.m(\overline{v}, \mathcal{E}, \overline{e}) \mid \textbf{new}\ N(\overline{v}, \mathcal{E}, \overline{e}) \mid (T)\,\mathcal{E} \end{aligned}$$

$$\boxed{\text{Top-level evaluation: } e \longmapsto^\flat e'}$$

$$\textsc{dyn-field}^\flat \quad \frac{\mathsf{fields}^\flat(N) = \overline{T\,f}}{\textbf{new}\ N(\overline{v}).f_i \longmapsto^\flat v_i} \qquad \textsc{dyn-invoke}^\flat \quad \frac{\begin{array}{c} v = \textbf{new}\ N(\overline{w}) \\ \mathsf{getmdef}^\flat(m, N) = \overline{T\,x} \to T\,\{e\} \end{array}}{v.m(\overline{v}) \longmapsto^\flat [v/\mathit{this}, \overline{v/x}]e} \qquad \textsc{dyn-cast}^\flat \quad \frac{\begin{array}{c} v = \textbf{new}\ N(\overline{v}) \\ \vdash^\flat N \leq T \end{array}}{(T)\,v \longmapsto^\flat v}$$

$$\boxed{\text{Proper evaluation: } e \longrightarrow^\flat e'}$$

$$\textsc{dyn-context}^\flat \quad \frac{e \longmapsto^\flat e'}{\mathcal{E}[e] \longrightarrow^\flat \mathcal{E}[e']}$$

$$\boxed{\mathsf{fields}^\flat(N) = \overline{T\,f}}$$

$$\textsc{fields-object}^\flat \quad \mathsf{fields}^\flat(\mathit{Object}) = \bullet \qquad \textsc{fields-class}^\flat \quad \frac{\textbf{class}\ C\ \textbf{extends}\ N\,\{\,\overline{T\,f}\ldots\} \qquad \mathsf{fields}^\flat(N) = \overline{T'\,f'}}{\mathsf{fields}^\flat(C) = \overline{T'\,f'}, \overline{T\,f}}$$

Figure 4.6 Syntax of iFJ.

$$
\begin{aligned}
\mathit{prog} &::= \overline{\mathit{def}}\ e \\
\mathit{def} &::= \mathit{cdef} \mid \mathit{idef} \\
\mathit{cdef} &::= \mathsf{class}\ C\ \mathsf{extends}\ N\ \mathsf{implements}\ \overline{J}\,\{\,\overline{T\,f}\ \ \overline{m : \mathit{mdef}}\}\\
\mathit{idef} &::= \mathsf{interface}\ I\ \mathsf{extends}\ \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\} \\
\mathit{msig} &::= \overline{T\,x} \to T \\
\mathit{mdef} &::= \mathit{msig}\,\{\,e\,\} \\
M, N &::= C \mid \mathit{Object} \\
T, U, V, W &::= N \mid I \\
d, e &::= x \mid e.f \mid e.m(\overline{e}) \mid \mathsf{new}\,N(\overline{e}) \mid \mathsf{cast}(T, e) \\
&\quad\ \mid \mathsf{getdict}(I, e) \mid \mathsf{let}\,T\,x = e\,\mathsf{in}\,e
\end{aligned}
$$

$$C, D \in \mathit{ClassName}_{\mathsf{iFJ}} \quad I, J \in \mathit{IfaceName}_{\mathsf{iFJ}}$$
$$m \in \mathit{MethodName}_{\mathsf{iFJ}} \quad f, g \in \mathit{FieldName}_{\mathsf{iFJ}} \quad x, y, z \in \mathit{VarName}_{\mathsf{iFJ}}$$

top-level evaluation relation $e \longmapsto^{\flat} e'$ reduces an expression e at the top level to e'. Rule DYN-FIELD$^{\flat}$ deals with field accesses **new** $N(\overline{v}).f_i$. The auxiliary relation $\mathsf{fields}^{\flat}(N) = \overline{T\,f}$, also defined in Figure 4.5, returns the fields declared by the superclasses of N and N itself. CoreGI$^{\flat}$ assumes that the ith constructor argument v_i corresponds to the field $T_i\,f_i$, so **new** $N(\overline{v}).f_i$ reduces to v_i. Rule DYN-INVOKE$^{\flat}$ handles method invocations, using the notation $[\overline{e/x}]$ to denote the capture-avoiding expression substitution that replaces variables x_i with expressions e_i. Finally, rule DYN-CAST$^{\flat}$ allows casts from **new** $N(\overline{v})$ to type T if N is a subtype of T.

The proper evaluation relation $e \longrightarrow e'$ reduces an expression e to e' by using a suitable evaluation context $\mathcal{E}$ together with the top-level evaluation relation $\longmapsto^{\flat}$.

Definition 4.3. The notation $\longrightarrow^{\flat+}$ denotes the transitive closure of $\longrightarrow^{\flat}$, whereas $\longrightarrow^{\flat*}$ denotes the reflexive and transitive closure of $\longrightarrow^{\flat}$.

4.2 Target Language: iFJ

The target language of the translation, dubbed iFJ, extends Featherweight Java (FJ [96]) with interfaces, let-expressions, and a primitive operation simulating CoreGI$^{\flat}$'s lookup of retroactive implementation definitions.[1] The following subsections define iFJ's syntax (Section 4.2.1), its dynamic semantics (Section 4.2.2), and its static semantics (Section 4.2.3). Furthermore, Section 4.2.4 proves type soundness for iFJ.

4.2.1 Syntax

Figure 4.6 defines the abstract syntax of iFJ. To facilitate the distinction between iFJ and CoreGI$^{\flat}$ constructs, the syntax of iFJ uses a **sans-serif** font to typeset keywords. Again, overbar notation denotes sequencing (see Definition 3.1) and the various kinds

[1] For full JavaGI, the run-time system provides this primitive operation.

of identifiers are drawn from pairwise disjoint and countable infinite sets of class names (ranged over by C, D), interface names (ranged over by I, J), method names (ranged over by m), field names (ranged over by f, g), and variable names (ranged over by x, y, z).

The translation from $\mathsf{CoreGI}^\flat$ to iFJ requires several designated class, interface, and field names. The definition of iFJ provides these names as follows:

- For each $\mathsf{CoreGI}^\flat$ interface name $I \in \mathit{IfaceName}$ and each $\mathsf{CoreGI}^\flat$ class type N, there exists an iFJ class name $\mathit{Dict}^{I,N} \in \mathit{ClassName}_{\mathsf{iFJ}}$ denoting the name of a *dictionary class* for I and N.[2] Dictionary classes result as the translation of retroactive implementation definitions.

- For each $\mathsf{CoreGI}^\flat$ interface name $I \in \mathit{IfaceName}$, there exists an iFJ interface name $\mathit{Dict}^{I} \in \mathit{IfaceName}_{\mathsf{iFJ}}$ denoting the name of a *dictionary interface* for I. Each dictionary class $\mathit{Dict}^{I,N}$ implements the corresponding dictionary interface Dict^{I}.

- For each $\mathsf{CoreGI}^\flat$ interface name $I \in \mathit{IfaceName}$, there exists an iFJ class name $\mathit{Wrap}^{I} \in \mathit{ClassName}_{\mathsf{iFJ}}$ denoting the name of a *wrapper class* for I. The translation from $\mathsf{CoreGI}^\flat$ to iFJ uses wrapper classes as adapters for classes that implement the corresponding interface retroactively in $\mathsf{CoreGI}^\flat$.

- There exists a field name $\mathit{wrapped} \in \mathit{FieldName}_{\mathsf{iFJ}}$.

The designated names just introduced are subject to the following convention:

Convention 4.4. The namespaces for regular classes, dictionary classes, and wrapper classes are pairwise disjoint. Similarly, the namespaces for regular interfaces and dictionary interfaces are disjoint. Furthermore, dictionary classes, dictionary interfaces, and wrapper classes do not appear in stand-alone iFJ programs; they only occur as the result of the translation from $\mathsf{CoreGI}^\flat$ to iFJ. Similarly, the field name *wrapped* is reserved for the translation and appears only in wrapper classes.

An iFJ program *prog* consists of a sequence of definitions *def* and a “main expression” e. A definition is either a class definition *cdef* or an interface definition *idef*. Class definitions are similar to those in FJ, except that iFJ classes also support an **implements** clause specifying the interfaces implemented by the class. The predefined class *Object* does not have a superclass and contains no fields and methods. The definition of an interface I specifies its superinterfaces $\overline{J}$ through an **extends** clause. Moreover, it also lists the names and the signatures of the methods supported by the interface. We often omit an **extends** clause of a class whose superclass is *Object*. Moreover, we also omit **implements** clauses if the sequence of superinterfaces is empty.

A method signature *msig* specifies that a method accepts parameters $\overline{x}$ of types $\overline{T}$ and produces a result of type T. A method definition *mdef* pairs a method signature *msig* with a body expression e.

Metavariables M, N range over class types, full types (ranged over by T, U, V, W) also comprise interface types. Expressions d, e include variables, field access, method calls, object allocations, and casts, just as FJ does. However, the syntax of casts is

[2] The term “dictionary” goes back to early work on type classes in Haskell [236] and is well-established in the Haskell community.

Figure 4.7 Subtyping for iFJ.

$$\boxed{\vdash_{\mathsf{iFJ}} T \le U}$$

$$\begin{array}{c}\textsc{sub-refl-ifj}\\ \vdash_{\mathsf{iFJ}} T \le T\end{array} \qquad \begin{array}{c}\textsc{sub-object-ifj}\\ \vdash_{\mathsf{iFJ}} T \le \mathit{Object}\end{array} \qquad \begin{array}{c}\textsc{sub-trans-ifj}\\ \dfrac{\vdash_{\mathsf{iFJ}} T \le U \qquad \vdash_{\mathsf{iFJ}} U \le V}{\vdash_{\mathsf{iFJ}} T \le V}\end{array}$$

$$\begin{array}{c}\textsc{sub-class-ifj}\\ \dfrac{\textbf{class } C \textbf{ extends } N \ldots}{\vdash_{\mathsf{iFJ}} C \le N}\end{array} \qquad \begin{array}{c}\textsc{sub-class-iface-ifj}\\ \dfrac{\textbf{class } C \textbf{ extends } N \textbf{ implements } \overline{J} \ldots}{\vdash_{\mathsf{iFJ}} C \le J_i}\end{array}$$

$$\begin{array}{c}\textsc{sub-iface-ifj}\\ \dfrac{\textbf{interface } I \textbf{ extends } \overline{J} \ldots}{\vdash_{\mathsf{iFJ}} I \le J_i}\end{array}$$

different than in FJ to emphasize that their dynamic behavior differs from that in FJ (see Section 4.2.2). In addition to the expression forms of FJ, the iFJ calculus supports a **getdict**(I, e) construct that simulates CoreGI$^\flat$'s lookup of retroactive implementation definitions. Moreover, the expression form $\textbf{let}\, T\, x = e_1 \,\textbf{in}\, e_2$ binds the result of e_1 to x within e_2. The type T prescribes to type of e_1 and x.

As for CoreGI$^\flat$, syntactic constructs that differ only in the names of bound expression variables are interchangeable in all contexts [176]. However, Conventions 4.1 and 4.2 do *not* apply to iFJ programs; that is, the namespaces of class and interface methods may overlap, and names of interface methods do not need to be globally unique.

4.2.2 Dynamic Semantics

The dynamic semantics of iFJ depends on the subtyping relation defined in Figure 4.7. The judgment $\vdash_{\mathsf{iFJ}} T \le U$ asserts that T is a subtype of U with respect to the semantics of iFJ, as indicated by the subscript "iFJ". The subtyping rules extend those of FJ with support for interfaces (rules SUB-CLASS-IFACE-IFJ and SUB-IFACE-IFJ) and a rule SUB-OBJECT-IFJ stating that *Object* is a supertype of any other type, including interface types. Subtyping in iFJ is reflexive and transitive, as usual.

Figure 4.8 defines several auxiliary relations:

- $\mathsf{fields}_{\mathsf{iFJ}}(N)$ returns the fields declared by the superclasses of N and N itself.
- $\mathsf{getmdef}_{\mathsf{iFJ}}(m, C)$ returns the definition of method m as defined by class C or one of its superclasses.
- $\mathsf{mindict}_{\mathsf{iFJ}} \mathscr{M}$ selects a minimal class from a set $\mathscr{M}$ of dictionary classes. If there exists no minimal class then $\mathsf{mindict}_{\mathsf{iFJ}} \mathscr{M}$ is undefined.
- $\mathsf{unwrap}(v)$ removes all wrappers at the top level of v. The metavariables v and w range over values as defined in Figure 4.9.

Figure 4.8 Auxiliaries for iFJ's dynamic semantics.

$$\boxed{\mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{T\,f}}$$

FIELDS-OBJECT-IFJ

$$\mathsf{fields}_{\mathsf{iFJ}}(\mathit{Object}) = \bullet$$

FIELDS-CLASS-IFJ

$$\frac{\begin{array}{c}\textbf{class}\ C\ \textbf{extends}\ N\ \textbf{implements}\ \overline{J}\,\{\,\overline{T\,f}\ \ldots\}\\ \mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{U\,g}\end{array}}{\mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\,g}, \overline{T\,f}}$$

$$\boxed{\mathsf{getmdef}_{\mathsf{iFJ}}(m, C) = \mathit{msig}}$$

DYN-MDEF-CLASS-BASE-IFJ

$$\frac{\textbf{class}\ C\ \textbf{extends}\ N\ \textbf{implements}\ \overline{J}\,\{\ldots\ \overline{m : \mathit{mdef}}\}}{\mathsf{getmdef}_{\mathsf{iFJ}}(m_k, C) = \mathit{mdef}_k}$$

DYN-MDEF-CLASS-SUPER-IFJ

$$\frac{\begin{array}{c}\textbf{class}\ C\ \textbf{extends}\ N\ \textbf{implements}\ \overline{J}\,\{\ldots\ \overline{m : \mathit{mdef}}\}\\ m \notin \overline{m} \qquad \mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = \mathit{mdef}\end{array}}{\mathsf{getmdef}_{\mathsf{iFJ}}(m, C) = \mathit{mdef}}$$

$$\boxed{\mathsf{mindict}_{\mathsf{iFJ}}\{\overline{\textbf{class}\ \mathit{Dict}^{I,N}\ \ldots}\} = N}$$

MINDICT-IFJ

$$\frac{(\forall i \in [n])\ \ \vdash_{\mathsf{iFJ}} N_k \leq N_i}{\mathsf{mindict}_{\mathsf{iFJ}}\{\textbf{class}\ \mathit{Dict}^{I,N_1}\ \ldots, \ldots, \textbf{class}\ \mathit{Dict}^{I,N_n}\ \ldots\} = \mathit{Dict}^{I,N_k}}$$

$$\boxed{\mathsf{unwrap}(v) = v}$$

UNWRAP-BASE-IFJ

$$\frac{N \neq \mathit{Wrap}^I \text{ for any } I}{\mathsf{unwrap}(\textbf{new}\ N(\overline{v})) = \textbf{new}\ N(\overline{v})}$$

UNWRAP-STEP-IFJ

$$\frac{\mathsf{unwrap}(v) = w}{\mathsf{unwrap}(\textbf{new}\ \mathit{Wrap}^I(v)) = w}$$

Figure 4.9 Dynamic semantics of iFJ.

Values and evaluation contexts

$$
\begin{aligned}
v, w &::= \textbf{new}\, N(\overline{v}) \\
\mathcal{E} &::= \Box \mid \mathcal{E}.f \mid \mathcal{E}.m(\overline{e}) \mid v.m(\overline{v}, \mathcal{E}, \overline{e}) \mid \textbf{new}\, N(\overline{v}, \mathcal{E}, \overline{e}) \\
&\quad \mid \textbf{cast}(T, \mathcal{E}) \mid \textbf{getdict}(I, \mathcal{E}) \mid \textbf{let}\, T\, x = \mathcal{E}\, \textbf{in}\, e
\end{aligned}
$$

Top-level evaluation: $e \longmapsto_{\mathsf{iFJ}} e'$

DYN-FIELD-IFJ

$$\frac{\mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{U\, f}}{\textbf{new}\, N(\overline{v}).f_i \longmapsto_{\mathsf{iFJ}} v_i}$$

DYN-INVOKE-IFJ

$$\frac{v = \textbf{new}\, N(\overline{w}) \qquad \mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = \overline{T\, x} \to T\, \{e\}}{v.m(\overline{v}) \longmapsto_{\mathsf{iFJ}} [v/\mathit{this}, \overline{v/x}]e}$$

DYN-CAST-IFJ

$$\frac{\mathsf{unwrap}(v) = \textbf{new}\, N(\overline{v}) \qquad \vdash_{\mathsf{iFJ}} N \leq T}{\textbf{cast}(T, v) \longmapsto_{\mathsf{iFJ}} \textbf{new}\, N(\overline{v})}$$

DYN-CAST-WRAP-IFJ

$$\frac{\mathsf{unwrap}(v) = \textbf{new}\, N(\overline{v}) \qquad \text{not } \vdash_{\mathsf{iFJ}} N \leq I \qquad \textbf{class}\ \mathit{Dict}^{I,M} \ldots \qquad \vdash_{\mathsf{iFJ}} N \leq M}{\textbf{cast}(I, v) \longmapsto_{\mathsf{iFJ}} \textbf{new}\ \mathit{Wrap}^{I}(\textbf{new}\, N(\overline{v}))}$$

DYN-GETDICT-IFJ

$$\frac{\mathsf{unwrap}(v) = \textbf{new}\, N(\overline{v}) \qquad \mathsf{mindict}_{\mathsf{iFJ}}\{\textbf{class}\ \mathit{Dict}^{I,N'} \ldots \mid \vdash_{\mathsf{iFJ}} N \leq N'\} = M}{\textbf{getdict}(I, v) \longmapsto_{\mathsf{iFJ}} \textbf{new}\, M()}$$

DYN-LET-IFJ

$$\textbf{let}\, T\, x = v\, \textbf{in}\, e \longmapsto_{\mathsf{iFJ}} [v/x]e$$

Proper evaluation: $e \longrightarrow_{\mathsf{iFJ}} e'$

DYN-CONTEXT-IFJ

$$\frac{e \longmapsto_{\mathsf{iFJ}} e'}{\mathcal{E}[e] \longrightarrow_{\mathsf{iFJ}} \mathcal{E}[e']}$$

Besides the syntax of values, Figure 4.9 also defines call-by-value evaluation contexts (denoted by $\mathcal{E}$), the top-level evaluation relation (written $e \longmapsto_{\mathsf{iFJ}} e'$), and the proper evaluation relation (written $e \longrightarrow_{\mathsf{iFJ}} e'$). The definition of the latter is simple because it just selects an appropriate evaluation context and delegates the rest of the work to the top-level evaluation relation.

At the top level of an expression, the $\longmapsto_{\mathsf{iFJ}}$ relation reduces field accesses, method invocations, and let-expressions in the obvious way. (As before, the notation $[\overline{e/x}]$ denotes the capture-avoiding expression substitution that replaces variables x_i with expressions e_i.) The rules for expressions of the form $\textbf{cast}(T, v)$ and $\textbf{getdict}(I, v)$ are slightly more involved. All three rules (DYN-CAST-IFJ, DYN-CAST-WRAP-IFJ, DYN-GETDICT-IFJ) first remove the wrappers at the top level of v to access the true run-time type N of v. Rule DYN-GETDICT-IFJ then uses $\mathsf{mindict}_{\mathsf{iFJ}}$ to reduce $\textbf{getdict}(I, v)$ to the minimal dictionary class $\mathit{Dict}^{I,N'}$ with $\vdash_{\mathsf{iFJ}} N \leq N'$. There are two rules for casts of the form $\textbf{cast}(T, v)$. The

Figure 4.10 Method types for iFJ.

$$\boxed{\mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \mathit{msig}}$$

MTYPE-CLASS-BASE-IFJ

$$\frac{\textbf{class } C \textbf{ extends } N \textbf{ implements } \overline{J}\,\{\ldots\ \overline{m : \mathit{msig}\,\{e\}}\,\}}{\mathsf{mtype}_{\mathsf{iFJ}}(m_k, C) = \mathit{msig}_k}$$

MTYPE-CLASS-SUPER-IFJ

$$\frac{\begin{array}{c}\textbf{class } C \textbf{ extends } N \textbf{ implements } \overline{J}\,\{\ldots\ \overline{m : \mathit{mdef}}\}\\ m \notin \overline{m} \qquad \mathsf{mtype}_{\mathsf{iFJ}}(m, N) = \mathit{msig}\end{array}}{\mathsf{mtype}_{\mathsf{iFJ}}(m, C) = \mathit{msig}}$$

MTYPE-IFACE-BASE-IFJ

$$\frac{\textbf{interface } I \textbf{ extends } \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\}}{\mathsf{mtype}_{\mathsf{iFJ}}(m_k, I) = \mathit{msig}_k}$$

MTYPE-IFACE-SUPER-IFJ

$$\frac{\begin{array}{c}\textbf{interface } I \textbf{ extends } \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\}\\ m \notin \overline{m} \qquad \mathsf{mtype}_{\mathsf{iFJ}}(m, J_i) = \mathit{msig}\end{array}}{\mathsf{mtype}_{\mathsf{iFJ}}(m, I) = \mathit{msig}}$$

first, rule DYN-CAST-IFJ, handles the case where N is indeed a subtype of T. The second, rule DYN-CAST-WRAP-IFJ, is only relevant to iFJ programs in the image of the translation from CoreGI$^\flat$ to iFJ because it assumes the existence of dictionary and wrapper classes (see Convention 4.4). The rule applies if T is an interface type I such that N is not a subtype of I according to iFJ's subtyping rules, but where a retroactive interface implementation established a subtyping relationship between N and I in the original CoreGI$^\flat$ program. Such a retroactive interface implementation translates to a dictionary class $\mathit{Dict}^{I,M}$ with $\vdash_{\mathsf{iFJ}} N \leq M$, as reflected in the premise of the rule. The result of the cast carries a fresh wrapper for I to compensate for the missing iFJ-subtyping relationship between N and I.

Definition 4.5. The notation $\longrightarrow^{+}_{\mathsf{iFJ}}$ denotes the transitive closure of $\longrightarrow_{\mathsf{iFJ}}$, whereas $\longrightarrow^{*}_{\mathsf{iFJ}}$ denotes the reflexive and transitive closure of $\longrightarrow_{\mathsf{iFJ}}$.

4.2.3 Static Semantics

The relation $\mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \mathit{msig}$, defined in Figure 4.10, looks up the signature of method m for receiver type T. It extends FJ's *mtype* relation with support for interfaces in the obvious way. The choice of superinterface J_i in the premise of rule MTYPE-IFACE-SUPER-IFJ is deterministic because the typing rules for programs, to be defined shortly, ensure that two distinct superinterfaces do not define methods with identical names.

As in Chapter 3, a variable environment Γ is a finite mapping from variables to types. The notation $\Gamma, x : T$ extends Γ with a mapping from x to T, assuming that x is not already bound in Γ. The notation $\Gamma(x)$ denotes the type T such that Γ maps x to T. It assumes that Γ contains such a binding for x.

Figure 4.11 Expression typing for iFJ.

$$\boxed{\Gamma \vdash_{\mathsf{iFJ}} e : T}$$

$$\textsc{exp-var-ifj}\quad \Gamma \vdash_{\mathsf{iFJ}} x : \Gamma(x) \qquad\qquad \textsc{exp-field-ifj}\quad \frac{\Gamma \vdash_{\mathsf{iFJ}} e : C \qquad \mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\,f}}{\Gamma \vdash_{\mathsf{iFJ}} e.f_j : U_j}$$

$$\textsc{exp-invoke-ifj}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{iFJ}} e : T \qquad \mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \overline{U\,x} \to U \\ (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_i : T_i \qquad (\forall i)\ \ \vdash_{\mathsf{iFJ}} T_i \leq U_i\end{array}}{\Gamma \vdash_{\mathsf{iFJ}} e.m(\overline{e}) : U} \qquad\qquad \textsc{exp-new-ifj}\quad \frac{\begin{array}{c}(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_i : T_i \\ \mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{U\,f} \\ (\forall i)\ \ \vdash_{\mathsf{iFJ}} T_i \leq U_i\end{array}}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{e}) : N}$$

$$\textsc{exp-cast-ifj}\quad \frac{\Gamma \vdash_{\mathsf{iFJ}} e : U}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{cast}(T, e) : T} \qquad\qquad \textsc{exp-getdict-ifj}\quad \frac{\Gamma \vdash_{\mathsf{iFJ}} e : T}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{getdict}(I, e) : \mathit{Dict}^{I}}$$

$$\textsc{exp-let-ifj}\quad \frac{\Gamma \vdash_{\mathsf{iFJ}} e_1 : T' \qquad \vdash_{\mathsf{iFJ}} T' \leq T \qquad \Gamma, x : T \vdash_{\mathsf{iFJ}} e_2 : U}{\textbf{let}\, T\, x = e_1\, \textbf{in}\, e_2 : U}$$

Figure 4.11 defines the expression typing judgment $\Gamma \vdash_{\mathsf{iFJ}} e : T$, which states that under variable environment Γ the expression e has type T. The rules for variables, fields, method calls, and object allocations are identical to the corresponding rules for FJ. Unlike in FJ, there is only one rule for casts because FJ's distinction between upcasts, downcasts, and stupid casts is not relevant to iFJ. The typing rules for dictionary lookup and let-expressions are straightforward.

Figure 4.12 specifies the typing rules for iFJ programs, including several auxiliary relations.

- The relation $\mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m : \mathit{msig}, C)$ asserts that class C correctly overrides method $m : \mathit{msig}$ (see rule OK-OVERRIDE-IFJ). Method overriding requires invariant return types as in FJ.

- The relation $\vdash_{\mathsf{iFJ}} m : \mathit{mdef}\ \mathsf{ok\,in}\, C$ asserts that the definition mdef of method m in class C is well-formed (see rule OK-MDEF-IN-CLASS-IFJ).

- The relation $\vdash_{\mathsf{iFJ}} C\ \mathsf{implements}\ I$ asserts that class C correctly implements all methods required by interface I (see rule IMPL-IFACE-IFJ).

- The relations $\vdash_{\mathsf{iFJ}} \mathit{cdef}\ \mathsf{ok}$ and $\vdash_{\mathsf{iFJ}} \mathit{idef}\ \mathsf{ok}$ assert well-formedness of class and interface definitions, respectively (see rules OK-CDEF-IFJ and OK-IDEF-IFJ, respectively). To keep things simple, well-formedness for interfaces requires that an interface does not override any method defined in one of its superinterfaces and that an interface

Figure 4.12 Program typing for iFJ.

$$\boxed{\mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m : \mathit{msig}, C)}$$

OK-OVERRIDE-IFJ

$$\frac{\textbf{class}\ C\ \textbf{extends}\ N\ \ldots \qquad \text{if } \mathsf{mtype}_{\mathsf{iFJ}}(m, N) = \mathit{msig'} \text{ then } \mathit{msig} = \mathit{msig'}}{\mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m : \mathit{msig}, C)}$$

$$\boxed{\vdash_{\mathsf{iFJ}} m : \mathit{mdef}\ \mathsf{ok\ in}\ C}$$

OK-MDEF-IN-CLASS-IFJ

$$\frac{\mathit{this} : C, \overline{x : T} \vdash_{\mathsf{iFJ}} e : U' \qquad \vdash_{\mathsf{iFJ}} U' \leq U \qquad \mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m : \overline{T\,x} \to U, C)}{\vdash_{\mathsf{iFJ}} m : \overline{T\,x} \to U\,\{e\}\ \mathsf{ok\ in}\ C}$$

$$\boxed{\vdash_{\mathsf{iFJ}} C\ \mathsf{implements}\ I}$$

IMPL-IFACE-IFJ

$$\frac{\begin{array}{c}\textbf{interface}\ I\ \textbf{extends}\ \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\} \\ (\forall i)\ \ \vdash_{\mathsf{iFJ}} C\ \mathsf{implements}\ J_i \qquad (\forall i)\ \ \mathsf{mtype}_{\mathsf{iFJ}}(m_i, C) = \mathit{msig}\end{array}}{\vdash_{\mathsf{iFJ}} C\ \mathsf{implements}\ I}$$

$$\boxed{\vdash_{\mathsf{iFJ}} \mathit{cdef}\ \mathsf{ok} \qquad \vdash_{\mathsf{iFJ}} \mathit{idef}\ \mathsf{ok}}$$

OK-CDEF-IFJ

$$\frac{(\forall i)\ \ \vdash_{\mathsf{iFJ}} m_i : \mathit{mdef}_i\ \mathsf{ok\ in}\ C \qquad (\forall i)\ \ \vdash_{\mathsf{iFJ}} C\ \mathsf{implements}\ J_i}{\vdash_{\mathsf{iFJ}} \textbf{class}\ C\ \textbf{extends}\ N\ \textbf{implements}\ \overline{J}\,\{\,\overline{T\,f}\ \ \overline{m : \mathit{mdef}}\,\}\ \mathsf{ok}}$$

OK-IDEF-IFJ

$$\frac{\begin{array}{c}(\forall i, j)\ \mathsf{mtype}_{\mathsf{iFJ}}(J_i, m_j) \text{ undefined} \\ (\forall i) \text{ if } \mathsf{mtype}_{\mathsf{iFJ}}(J_i, m) = \mathit{msig} \text{ then } \mathsf{mtype}_{\mathsf{iFJ}}(J_j, m) \text{ undefined for all } j \neq i\end{array}}{\vdash_{\mathsf{iFJ}} \textbf{interface}\ I\ \textbf{extends}\ \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\}}$$

$$\boxed{\vdash_{\mathsf{iFJ}} \mathit{prog}\ \mathsf{ok}}$$

OK-PROG-IFJ

$$\frac{\begin{array}{c}\text{well-formedness criteria defined in Figure 4.13 hold} \\ (\forall i)\ \ \vdash_{\mathsf{iFJ}} \mathit{def}_i\ \mathsf{ok} \qquad \emptyset \vdash_{\mathsf{iFJ}} e : T\end{array}}{\vdash_{\mathsf{iFJ}} \overline{\mathit{def}}\ e\ \mathsf{ok}}$$

Figure 4.13 Additional well-formedness criteria for iFJ.

WF-IFJ-1 If a class or interface name appears anywhere in a program, then the program also contains a definition for that class or interface.

WF-IFJ-2 The class and interface hierarchies are acyclic.

WF-IFJ-3 The names of the fields defined in a class and any of its superclasses are pairwise disjoint. (That is, iFJ does not support field shadowing.)

WF-IFJ-4 The names of the methods defined in a class or an interface are pairwise disjoint. (That is, iFJ does not support method overloading.)

WF-IFJ-5 For all dictionary classes $\mathit{Dict}^{I,N}$, it holds that $\mathsf{fields}_{\mathsf{iFJ}}(\mathit{Dict}^{I,N}) = \bullet$ and that $\mathit{Dict}^{I,N}$ implements interface Dict^{I}.

WF-IFJ-6 Wrapper classes Wrap^{I} have the form

$$\textbf{class } \mathit{Wrap}^{I} \textbf{ extends } \mathit{Object} \textbf{ implements } I \,\{\, \mathit{Object}\ \mathit{wrapped}\ \overline{m : \mathit{mdef}}\}$$

for some sequence $\overline{m : \mathit{mdef}}$.

does not have two distinct superinterfaces both defining a method with the same name.

- The relation $\vdash_{\mathsf{iFJ}} \mathit{prog}\ \mathsf{ok}$ asserts well-formedness of programs (see rule OK-PROG). Well-formedness of programs relies on the additional well-formedness criteria defined in Figure 4.13.

4.2.4 Type Soundness

The type soundness proof for iFJ follows the syntactic approach developed by Wright and Felleisen [244] and the type soundness proof for FJ. The theorems of this section implicitly assume that the underlying iFJ program is well-formed.

The preservation theorem states that an evaluation step preserves the type of an expression.

Theorem 4.6 (Preservation for proper evaluation of iFJ). *If* $\emptyset \vdash_{\mathsf{iFJ}} e : T$ *and* $e \longrightarrow_{\mathsf{iFJ}} e'$ *then* $\emptyset \vdash_{\mathsf{iFJ}} e' : T'$ *for some* T' *with* $\vdash_{\mathsf{iFJ}} T' \leq T$.

Proof. It suffices to show that $\emptyset \vdash_{\mathsf{iFJ}} \mathcal{E}[e] : T$ and $e \longmapsto_{\mathsf{iFJ}} e'$ imply $\emptyset \vdash_{\mathsf{iFJ}} \mathcal{E}[e'] : T'$ with $\vdash_{\mathsf{iFJ}} T' \leq T$. This proof is by induction on the structure of $\mathcal{E}$. See Section C.1.1 for details. □

In FJ, an expression may get stuck on a bad cast. The same may happen in iFJ.

Figure 4.14 Well-formedness of $\mathsf{CoreGI}^\flat$ types.

$$\boxed{\vdash^\flat T\ \mathsf{ok}}$$

$$\textsc{ok-object}^\flat \quad \vdash^\flat \mathit{Object}\ \mathsf{ok} \qquad\qquad \textsc{ok-class}^\flat \quad \frac{\mathbf{class}\ C\ldots}{\vdash^\flat C\ \mathsf{ok}} \qquad\qquad \textsc{ok-iface}^\flat \quad \frac{\mathbf{interface}\ I\ldots}{\vdash^\flat I\ \mathsf{ok}}$$

Definition 4.7 (Stuck on a bad cast for iFJ). An iFJ expression e is *stuck on a bad cast* if, and only if, there exists an evaluation context $\mathcal{E}$, a type T, and a value v such that $e = \mathcal{E}[\mathbf{cast}(T, v)]$, $\mathsf{unwrap}(v) = \mathbf{new}\ N(\overline{v})$, and neither $\vdash_{\mathsf{iFJ}} N \leq T$ nor $\vdash_{\mathsf{iFJ}} N \leq M$ for some dictionary class $\mathit{Dict}^{I,M}$ with $T = I$ holds.

Additionally, an iFJ expression may also get stuck on a bad dictionary lookup.

Definition 4.8 (Stuck on a bad dictionary lookup). An iFJ expression e is *stuck on a bad dictionary lookup* if, and only if, there exists an evaluation context $\mathcal{E}$, an interface type I, and a value v such that $e = \mathcal{E}[\mathbf{getdict}(I, v)]$, $\mathsf{unwrap}(v) = \mathbf{new}\ N(\overline{v})$, and $\mathsf{mindict}_{\mathsf{iFJ}}\mathscr{M}$ is undefined for $\mathscr{M} = \{\mathbf{class}\ \mathit{Dict}^{I,N'}\ \ldots \mid \vdash_{\mathsf{iFJ}} N \leq N'\}$.

The progress theorem states that a well-typed expression is either a value, or reducible, or stuck for one of the two reasons just defined.

Theorem 4.9 (Progress for iFJ). *If $\emptyset \vdash_{\mathsf{iFJ}} e : T$ then either $e = v$ for some value v, or $e \longrightarrow_{\mathsf{iFJ}} e'$ for some expression e', or e is stuck on a bad cast or a bad dictionary lookup.*

Proof. By induction on the derivation of $\emptyset \vdash_{\mathsf{iFJ}} e : T$. See Section C.1.2 for details. □

Theorem 4.10 (Type soundness for iFJ). *If $\emptyset \vdash_{\mathsf{iFJ}} e : T$ then either e diverges, or $e \longrightarrow^*_{\mathsf{iFJ}} v$ for some value v such that $\emptyset \vdash_{\mathsf{iFJ}} v : T'$ with $\vdash_{\mathsf{iFJ}} T' \leq T$, or $e \longrightarrow^*_{\mathsf{iFJ}} e'$ for some expression e' that is stuck on a bad cast or a bad dictionary lookup.*

Proof. Assume that $e \longrightarrow^*_{\mathsf{iFJ}} e'$ for some normal form e'. Using Theorem 4.6, transitivity of subtyping, and an induction on the length of the evaluation sequence yields $\emptyset \vdash_{\mathsf{iFJ}} e' : T'$ with $\vdash_{\mathsf{iFJ}} T' \leq T$. The claim now follows with Theorem 4.9. □

4.3 From CoreGI$^\flat$ to iFJ

Having defined the source and target languages, it is now time to formalize the translation from $\mathsf{CoreGI}^\flat$ to iFJ. The translation is not a purely syntactic one but may depend on the type of the construct being translated. Thus, we interweave the translation with the definition of a static semantics for $\mathsf{CoreGI}^\flat$.

Figure 4.14 defines the relation $\vdash^\flat T\ \mathsf{ok}$, which states that the $\mathsf{CoreGI}^\flat$ type T is well-formed. The relation $\mathsf{mtype}^\flat(m, T) = \mathit{msig} \rightsquigarrow I^?$, defined in Figure 4.15, looks up the signature of method m for static receiver type T. The optional interface name $I^?$ is

Figure 4.15 Method types for $\mathsf{CoreGI}^\flat$.

$$\boxed{\mathsf{mtype}^\flat(m, T) = \mathit{msig} \rightsquigarrow I^?}$$

$$\frac{\textbf{class}\ C\ \textbf{extends}\ N\,\{\ldots\ \overline{m : \mathit{msig}\,\{e\}}\,\} \qquad m^{\mathrm{c}} = m_k}{\mathsf{mtype}^\flat(m^{\mathrm{c}}, C) = \mathit{msig}_k \rightsquigarrow \mathsf{nil}}\ \textsc{mtype-class-base}^\flat$$

$$\frac{\textbf{class}\ C\ \textbf{extends}\ N\,\{\ldots\ \overline{m : \mathit{mdef}}\,\} \qquad m^{\mathrm{c}} \notin \overline{m} \qquad \mathsf{mtype}^\flat(m^{\mathrm{c}}, N) = \mathit{msig} \rightsquigarrow \mathsf{nil}}{\mathsf{mtype}^\flat(m^{\mathrm{c}}, C) = \mathit{msig} \rightsquigarrow \mathsf{nil}}\ \textsc{mtype-class-super}^\flat$$

$$\frac{\textbf{interface}\ I\ \textbf{extends}\ \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\} \qquad \vdash^\flat T \leq I \rightsquigarrow J^?}{\mathsf{mtype}^\flat(m^{\mathrm{i}}_k, T) = \mathit{msig}_k \rightsquigarrow J^?}\ \textsc{mtype-iface}^\flat$$

different from nil if, and only if, m is a method of interface $I^?$ and T implements $I^?$ only retroactively. As before, we omit the part "$\rightsquigarrow I^?$" if this information is irrelevant; that is, $\mathsf{mtype}^\flat(m, T) = \mathit{msig}$ abbreviates $\mathsf{mtype}^\flat(m, T) = \mathit{msig} \rightsquigarrow I^?$ for some fresh I.

Figure 4.16 defines the typing and translation rules for $\mathsf{CoreGI}^\flat$ expressions. The judgment $\Gamma \vdash^\flat e : T \rightsquigarrow e'$ denotes that under variable environment Γ the $\mathsf{CoreGI}^\flat$ expression e has type T and translates to the iFJ expression e'. If the translation part "$\rightsquigarrow e'$" is irrelevant, we simply omit it, so $\Gamma \vdash^\flat e : T$ means that there exists some iFJ expression e' with $\Gamma \vdash^\flat e : T \rightsquigarrow e'$.

To lighten the notation, we do not make the translation of identifiers explicit. Instead, we simply use $\mathsf{CoreGI}^\flat$ identifiers as if they were iFJ identifiers and assume an implicit translation of identifiers. It is always clear from the context whether an identifier acts as a $\mathsf{CoreGI}^\flat$ or as an iFJ identifier.

The translation of variables (rule $\textsc{exp-var}^\flat$), field accesses (rule $\textsc{exp-field}^\flat$), and cast operations (rule $\textsc{exp-cast}^\flat$) is straightforward. The translation of method invocations (rule $\textsc{exp-invoke}^\flat$) and object allocations (rule $\textsc{exp-new}^\flat$) is more involved because it needs to compensate the lack of retroactive interface implementations in the target language iFJ by using wrappers [10]. The general scheme is as follows: if a $\mathsf{CoreGI}^\flat$ expression e has type T but the context of the expression uses e at interface type I such that the subtyping relationship between T and I in $\mathsf{CoreGI}^\flat$ depends on a retroactive implementation (see Figure 4.4), then the translation wraps e with a wrapper of class Wrap^I. Omitting the wrapper would produce an ill-typed iFJ expression because iFJ does not support retroactive interface implementations, so $\vdash_{\mathsf{iFJ}} T \leq I$ does *not* hold. The auxiliary function wrap, also defined in Figure 4.16, performs the wrapping just described.

We next consider typing and translation of $\mathsf{CoreGI}^\flat$ classes, interfaces, implementation definitions, and programs. Before presenting the formal rules, it helps to look at some concrete examples. Figure 4.17 shows several $\mathsf{CoreGI}^\flat$ constructs and their translations

Figure 4.16 Typing and translating $\mathsf{CoreGI}^\flat$ expressions.

$\boxed{\Gamma \vdash^\flat e : T \leadsto e'}$

$$\textsc{exp-var}^\flat \quad \Gamma \vdash^\flat x : \Gamma(x) \leadsto x$$

$$\textsc{exp-field}^\flat \quad \frac{\Gamma \vdash^\flat e : C \leadsto e' \qquad \mathsf{fields}^\flat(C) = \overline{U\,f}}{\Gamma \vdash^\flat e.f_j : U_j \leadsto e'.f_j}$$

$$\textsc{exp-invoke}^\flat \quad \frac{\begin{array}{c}\Gamma \vdash^\flat e : T \leadsto e' \qquad \mathsf{mtype}^\flat(m, T) = \overline{U\,x} \to U \leadsto I^? \qquad (\forall i)\ \Gamma \vdash^\flat e_i : T_i \leadsto e'_i \\ (\forall i)\ \vdash^\flat T_i \le U_i \leadsto J^?_i \qquad e'' = \mathsf{wrap}(I^?, e') \qquad (\forall i)\ e''_i = \mathsf{wrap}(J^?_i, e'_i)\end{array}}{\Gamma \vdash^\flat e.m(\overline{e}) : U \leadsto e''.m(\overline{e''})}$$

$$\textsc{exp-new}^\flat \quad \frac{\begin{array}{c}(\forall i)\ \Gamma \vdash^\flat e_i : T_i \leadsto e'_i \\ \vdash^\flat N\ \mathsf{ok} \qquad \mathsf{fields}^\flat(N) = \overline{U\,f} \qquad (\forall i)\ \vdash^\flat T_i \le U_i \leadsto J^?_i \qquad (\forall i)\ e''_i = \mathsf{wrap}(J^?_i, e'_i)\end{array}}{\Gamma \vdash^\flat \mathbf{new}\ N(\overline{e}) : N \leadsto \mathbf{new}\ N(\overline{e''})}$$

$$\textsc{exp-cast}^\flat \quad \frac{\vdash^\flat T\ \mathsf{ok} \qquad \Gamma \vdash^\flat e : U \leadsto e'}{\Gamma \vdash^\flat (T)\,e : T \leadsto \mathbf{cast}(T, e')}$$

$\boxed{\mathsf{wrap}(I^?, e) = e'}$

$$\mathsf{wrap}(I^?, e) = \begin{cases} e & \text{if } I^? = \mathsf{nil} \\ \mathbf{new}\ \mathit{Wrap}^I(e) & \text{if } I^? = I \end{cases}$$

to iFJ. The $\mathsf{CoreGI}^\flat$ interface I translates into an identical iFJ interface I, a dictionary interface Dict^I, and a wrapper class Wrap^I. The dictionary interface serves as the common interface for all dictionaries that the translation generates for I's retroactive implementations. The method *foo* of Dict^I has the same signature as the *foo* method of I but extended with an additional parameter y of type *Object* to abstract over the implementing type of potential retroactive implementations of I. The wrapper class Wrap^I adapts objects of classes that implement I only retroactively in $\mathsf{CoreGI}^\flat$. It implements the *foo* method of I as follows: first retrieve the dictionary for I and the wrapped object to get a value of type Dict^I; then invoke the *foo* method on this value and pass the wrapped object as the additional parameter.

The translation of the $\mathsf{CoreGI}^\flat$ classes D and C is straightforward. The translation of the retroactive implementation of I with implementing type C is more interesting. It produces a dictionary class $\mathit{Dict}^{I,C}$ that implements the dictionary interface Dict^I. Method *foo* of this class is the translation of the method that remains anonymous in

Figure 4.17 Sample translation.
The left-hand side shows CoreGI$^\flat$ constructs, the right-hand side shows their translations to iFJ.

interface I {
$\quad$ $foo : \bullet \to D$
}
	interface I {
	$\quad$ $foo : \bullet \to D$
	}
	interface $Dict^I$ {
	$\quad$ $foo : Object\ y \to D$
	}
	class $Wrap^I$ **implements** I {
	$\quad$ $Object\ wrapped$
	$\quad$ $foo : \bullet \to D$ {
	$\qquad$ **getdict**$(I, this.wrapped).foo(this.wrapped)$
	$\quad$ }
	}

class D {
$\quad$ $bar : I\ x \to D\{x.foo()\}$
}
	class D {
	$\quad$ $bar : I\ x \to D\{x.foo()\}$
	}

class C {
$\quad$ $D\ f$
}
	class C {
	$\quad$ $D\ f$
	}

implementation $I\ [C]$ {
$\quad$ $\bullet \to D\ \{this.f\}$
}
	class $Dict^{I,C}$ **implements** $Dict^I$ {
	$\quad$ $foo : Object\ y \to D$ {
	$\qquad$ **let** $C\ z =$ **cast**(C, y) **in** $z.f$
	$\quad$ }
	}

new $C($**new** $D()).foo()$	**new** $Wrap^I($**new** $C($**new** $D())).foo()$
new $D().bar($**new** $C($**new** $D()))$	**new** $D().bar($**new** $Wrap^I($**new** $C($**new** $D())))$

the retroactive implementation. The additional parameter y of *foo* abstracts over the implementing type C. Its type is *Object* as demanded by Dict^I, so the body of *foo* first casts y to C and then accesses the field f.

Figure 4.17 also shows two expressions and their translations. The translation of the first expression has to wrap the receiver of the call because the receiver implements the target method *foo* only retroactively. In the second expression, the argument of the call requires wrapping because the method being invoked expects an object of type I, which the argument class C implements only retroactively.

Let us turn to the formal typing and translation rules for CoreGI$^\flat$ classes, interfaces, implementation definitions, and programs. Figure 4.18 defines several auxiliaries:

- $\mathsf{override\text{-}ok}^\flat(m : \mathit{msig}, C)$ is the usual check to verify that a CoreGI$^\flat$ method m with signature *msig* correctly overrides method m of C's direct superclass.

- $\vdash^\flat \mathit{msig}\ \mathsf{ok}$ establishes well-formedness of a CoreGI$^\flat$ method signature *msig*.

- $\Gamma \vdash^\flat \mathit{mdef}\ \mathsf{ok} \rightsquigarrow e$ checks well-formedness of a CoreGI$^\flat$ method definition *mdef* under variable environment Γ and translates the body of the method definition to the iFJ expression e.

- $\vdash^\flat m : \mathit{mdef}\ \mathsf{ok\,in}\, C \rightsquigarrow \mathit{mdef}'$ asserts that $m : \mathit{mdef}$ is well-formed in class C and translates the CoreGI$^\flat$ method definition *mdef* to the iFJ method definition mdef'.

- $\Gamma \vdash^\flat \mathit{mdef}\ \mathsf{implements}\ \mathit{msig} \rightsquigarrow \mathit{mdef}'$ ensures that *mdef*, a CoreGI$^\flat$ method definition from a retroactive implementation, properly implements the CoreGI$^\flat$ method signature *msig* under variable environment Γ. Moreover, it translates *mdef* into an iFJ method definition mdef' such that mdef' may be used inside the dictionary class serving as the translation of *mdef*'s implementation definition.

- $\mathsf{wrapper\text{-}methods}(I) = \overline{m : \mathit{mdef}}$ computes all iFJ methods $\overline{m : \mathit{mdef}}$ that should be contained in the wrapper class for I.

- $\mathsf{dict\text{-}methods}(I) = \overline{m : \mathit{mdef}}$ computes all iFJ methods $\overline{m : \mathit{mdef}}$ that are needed by a dictionary class to implement the methods of the dictionary interface Dict^I. The translation of a retroactive implementation of interface J invokes dict-methods for all direct superinterfaces of J.

With these preparations, the definition of the typing and translation rules for CoreGI$^\flat$ programs is straightforward (Figure 4.19).

- The judgment $\vdash^\flat \mathit{cdef}\ \mathsf{ok} \rightsquigarrow \mathit{cdef}'$ asserts well-formedness of the CoreGI$^\flat$ class *cdef* and translates it into an iFJ class cdef'.

- The judgment $\vdash^\flat \mathit{idef}\ \mathsf{ok} \rightsquigarrow \overline{\mathit{def}}$ asserts well-formedness of the CoreGI$^\flat$ interface *idef* and translates it into a sequence of iFJ definitions $\overline{\mathit{def}}$. These definitions consist of the iFJ version of the interface, the corresponding dictionary interface, and an appropriate wrapper class.

Figure 4.18 Auxiliaries for typing and translating $\mathsf{CoreGI}^\flat$ programs.

$$\boxed{\mathsf{override\text{-}ok}^\flat(m : \mathit{msig}, C)}$$

OK-OVERRIDE$^\flat$
$$\frac{\textbf{class } C \textbf{ extends } N \ldots \qquad \text{if } \mathsf{mtype}^\flat(m, N) = \mathit{msig}' \leadsto \mathsf{nil} \text{ then } \mathit{msig} = \mathit{msig}'}{\mathsf{override\text{-}ok}^\flat(m : \mathit{msig}, C)}$$

$$\boxed{\vdash^\flat \mathit{msig}\ \mathsf{ok} \quad \Gamma \vdash^\flat \mathit{mdef}\ \mathsf{ok} \leadsto e \quad \vdash^\flat m : \mathit{mdef}\ \mathsf{ok\,in}\, C \leadsto \mathit{mdef}'}$$

OK-MSIG$^\flat$
$$\frac{\vdash^\flat \overline{T}, U\ \mathsf{ok}}{\vdash^\flat \overline{T\,x} \to U\ \mathsf{ok}}$$

OK-MDEF$^\flat$
$$\frac{\vdash^\flat \overline{T\,x} \to U\ \mathsf{ok} \qquad \Gamma, \overline{x : T} \vdash^\flat e : U' \leadsto e' \qquad \vdash^\flat U' \le U \leadsto I^?}{\Gamma \vdash^\flat \overline{T\,x} \to U\,\{e\}\ \mathsf{ok} \leadsto \mathsf{wrap}(I^?, e')}$$

OK-MDEF-IN-CLASS$^\flat$
$$\frac{\mathit{this} : C \vdash^\flat \mathit{msig}\,\{e\}\ \mathsf{ok} \leadsto e' \qquad \mathsf{override\text{-}ok}^\flat(m : \mathit{msig}, C)}{\vdash^\flat m : \mathit{msig}\,\{e\}\ \mathsf{ok\,in}\, C \leadsto \mathit{msig}\,\{e'\}}$$

$$\boxed{\Gamma \vdash^\flat \mathit{mdef}\ \mathsf{implements}\ \mathit{msig} \leadsto \mathit{mdef}'}$$

IMPL-METH$^\flat$
$$\frac{\Gamma \vdash^\flat \overline{T\,x} \to U\,\{e\}\ \mathsf{ok} \leadsto e' \qquad \overline{T\,x} \to U = \mathit{msig} \qquad \Gamma(\mathit{this}) = N \qquad y, z \text{ fresh}}{\begin{array}{c}\Gamma \vdash^\flat \overline{T\,x} \to U\,\{e\}\ \mathsf{implements}\ \mathit{msig} \\ \leadsto \mathit{Object}\, y, \overline{T\,x} \to\ U\{\mathbf{let}\, N\, z = \mathbf{cast}(N, y)\ \mathbf{in}\ [z/\mathit{this}]e'\}\end{array}}$$

$$\boxed{\mathsf{wrapper\text{-}methods}(I) = \overline{m : \mathit{mdef}} \quad \mathsf{dict\text{-}methods}(I) = \overline{m : \mathit{mdef}}}$$

WRAPPER-METHODS$^\flat$
$$\frac{\begin{array}{l}\qquad\qquad\textbf{interface } I \textbf{ extends } \overline{J}^n\,\{\,\overline{m : \mathit{msig}}\,\} \\ (\forall i)\ \ \mathit{msig}_i = \overline{T\,x} \to U \text{ and} \\ \qquad \mathit{mdef}_i = \overline{T\,x} \to\ U\{\,\mathbf{getdict}(I, \mathit{this.wrapped}).m_i(\mathit{this.wrapped}, \overline{x})\}\end{array}}{\mathsf{wrapper\text{-}methods}(I) = \overline{m : \mathit{mdef}}\ \ \mathsf{wrapper\text{-}methods}(J_1) \ldots \mathsf{wrapper\text{-}methods}(J_n)}$$

DICT-METHODS$^\flat$
$$\frac{\begin{array}{c}\textbf{interface } I \textbf{ extends } \overline{J}^n\,\{\,\overline{m : \mathit{msig}}\,\} \\ (\forall i)\ \ \mathit{msig}_i = \overline{T\,x} \to U \text{ and } \mathit{mdef}_i = \mathit{Object}\ y, \overline{T\,x} \to\ U\{\,\mathbf{getdict}(I, y).m_i(y, \overline{x})\}\end{array}}{\mathsf{dict\text{-}methods}(I) = \overline{m : \mathit{mdef}}\ \ \mathsf{dict\text{-}methods}(J_1) \ldots \mathsf{dict\text{-}methods}(J_n)}$$

Figure 4.19 Typing and translating CoreGI$^\flat$ programs.

$$\boxed{\vdash^\flat \mathit{cdef}\ \mathsf{ok} \leadsto \mathit{cdef}' \qquad \vdash^\flat \mathit{idef}\ \mathsf{ok} \leadsto \overline{\mathit{def}} \qquad \vdash^\flat \mathit{impl}\ \mathsf{ok} \leadsto \mathit{cdef}}$$

$$\textsc{ok-cdef}^\flat\quad \frac{\vdash^\flat N, \overline{T}\ \mathsf{ok} \qquad (\forall i)\ \vdash^\flat m_i : \mathit{mdef}_i\ \mathsf{ok\ in}\ C \leadsto \mathit{mdef}_i'}{\begin{array}{l}\quad\vdash^\flat \mathbf{class}\ C\ \mathbf{extends}\ N\,\{\,\overline{T\ f}\ \overline{m : \mathit{mdef}}\,\}\ \mathsf{ok}\\ \leadsto \textsf{\textbf{class}}\ C\ \textsf{\textbf{extends}}\ N\ \textsf{\textbf{implements}}\ \bullet\ \{\,\overline{T f}\ \overline{m : \mathit{mdef}'}\,\}\end{array}}$$

$$\textsc{ok-idef}^\flat\quad \frac{\vdash^\flat \overline{J}, \overline{\mathit{msig}}\ \mathsf{ok}}{\begin{array}{l}\qquad\qquad\vdash^\flat \mathbf{interface}\ I\ \mathbf{extends}\ \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\}\\ \leadsto \textsf{\textbf{interface}}\ I\ \textsf{\textbf{extends}}\ \overline{J}\{\,\overline{m : \mathit{msig}}\}\\ \quad \textsf{\textbf{interface}}\ \mathit{Dict}^I\ \textsf{\textbf{extends}}\ \overline{\mathit{Dict}^J}\,\{\,\overline{m :\ \mathit{Object}\ y, \mathit{msig}}\}\\ \quad \textsf{\textbf{class}}\ \mathit{Wrap}^I\ \textsf{\textbf{extends}}\ \mathit{Object}\ \textsf{\textbf{implements}}\ I\{\ \mathit{Object}\ \mathit{wrapped}\ \ \textsf{wrapper-methods}(I)\,\}\end{array}}$$

$$\textsc{ok-impl}^\flat\quad \frac{\begin{array}{l}\vdash^\flat N, I\ \mathsf{ok} \qquad \mathbf{interface}\ I\ \mathbf{extends}\ \overline{J}^{\,n}\,\{\,\overline{m : \mathit{msig}}\,\}\\ \quad (\forall i)\ \mathit{this} : N \vdash^\flat \mathit{mdef}_i\ \mathsf{implements}\ \mathit{msig}_i \leadsto \mathit{mdef}_i'\end{array}}{\begin{array}{l}\quad\vdash^\flat \mathbf{implementation}\ I\ [\,N\,]\,\{\,\overline{m : \mathit{mdef}}\,\}\ \mathsf{ok}\\ \leadsto \textsf{\textbf{class}}\ \mathit{Dict}^{I,N}\ \textsf{\textbf{extends}}\ \mathit{Object}\ \textsf{\textbf{implements}}\ \mathit{Dict}^I\,\{\\ \qquad\quad \overline{m : \mathit{mdef}'}\\ \qquad\quad \textsf{dict-methods}(J_1)\ldots\textsf{dict-methods}(J_n)\\ \qquad \}\end{array}}$$

$$\boxed{\vdash^\flat \mathit{prog}\ \mathsf{ok} \leadsto \mathit{prog}'}$$

$$\textsc{ok-prog}^\flat\quad \frac{\begin{array}{c}\text{well-formedness criteria defined in Figure 4.20 hold}\\ (\forall i)\ \vdash^\flat \mathit{def}_i\ \mathsf{ok} \leadsto \overline{\mathit{def}_i'} \qquad \emptyset \vdash^\flat e : T \leadsto e'\end{array}}{\vdash^\flat \overline{\mathit{def}}^{\,n}\ e\ \mathsf{ok} \leadsto \overline{\mathit{def}_1'}\ \ldots\ \overline{\mathit{def}_n'}\ e'}$$

Figure 4.20 Additional well-formedness criteria for CoreGI$^\flat$.

- For each class definition **class** C **extends** $N\,\{\,\overline{T\,f}^{\,n}\ \overline{m : \mathit{mdef}}^{\,l}\,\}$ the following well-formedness criteria must hold:

  WF$^\flat$-CLASS-1 The field names, including names of inherited fields, are pairwise disjoint. That is, $i \neq j \in [n]$ implies $f_i \neq f_j$ and $\mathsf{fields}(N) = \overline{U\,g}$ implies $\overline{f} \cap \overline{g} = \emptyset$.

  WF$^\flat$-CLASS-2 The method names $\overline{m}$ are pairwise disjoint.

- For each implementation definition **implementation** $I\,[N]\,\ldots$ the following well-formedness criterion must hold:

  WF$^\flat$-IMPL-1 There exist suitable implementations for all superinterfaces of I. Suppose J is a direct superinterface of I. Then there exists a definition **implementation** $J\ [\,M\,]\ \ldots$ such that $\vdash^\flat N \leq M$.

  (This criterion corresponds to WF-IMPL-1 in Chapter 3.)

- The CoreGI$^\flat$ program under consideration must fulfill the following well-formedness criteria:

  WF$^\flat$-PROG-1 A program does not contain two different implementations for the same interface with identical implementation types. That is, for each pair of disjoint implementation definitions **implementation** $I\ [\,N\,]\ \ldots$ and **implementation** $I\ [\,M\,]\ \ldots$ it holds that $N \neq M$.

  (This criterion corresponds to WF-PROG-1 in Chapter 3.)

  WF$^\flat$-PROG-2 The class and interface hierarchies of the program are acyclic.

  (This criterion corresponds to WF-PROG-5 in Chapter 3.)

- The judgment $\vdash^\flat \mathit{impl}\ \mathsf{ok} \leadsto \mathit{cdef}$ asserts well-formedness of the CoreGI$^\flat$ implementation definition *impl* and translates it into a dictionary class *cdef*.

- The judgment $\vdash^\flat \mathit{prog}\ \mathsf{ok} \leadsto \mathit{prog}'$ asserts well-formedness of the CoreGI$^\flat$ program *prog* and translates it into an iFJ program prog'. The judgment depends on the additional well-formedness criteria defined in Figure 4.20.

4.4 Meta-Theoretical Properties

The translation from CoreGI$^\flat$ to iFJ has two important meta-theoretical properties: it preserves the static semantics and the dynamic semantics of CoreGI$^\flat$. The next two subsections prove these properties formally.

Figure 4.21 Potentially commuting diagram.

$$\begin{tikzcd}
e \arrow[r, "{}^{\flat}" near end] \arrow[d, squiggly, "\Gamma \vdash^{\flat} e : T \rightsquigarrow d"'] & e' \arrow[d, squiggly, "\Gamma \vdash^{\flat} e' : T' \rightsquigarrow d'"] \\
d \arrow[r, "{}^{+}_{\mathsf{iFJ}}" near end] & d'
\end{tikzcd}$$

4.4.1 Translation Preserves Static Semantics

The following theorem shows that a CoreGI$^\flat$ expression e of type T translates into an iFJ expression e' of the same type T. (It is possible to use the same type T for both calculi because the translation of identifiers happens implicitly.)

Theorem 4.11 (Translation preserves types of expressions). *Suppose that the underlying iFJ program is the translation of the underlying CoreGI$^\flat$ program. If $\Gamma \vdash^{\flat} e : T \rightsquigarrow e'$ then $\Gamma \vdash_{\mathsf{iFJ}} e' : T$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash^{\flat} e : T \rightsquigarrow e'$. See Section C.2.1 for details. □

Moreover, well-formedness of a CoreGI$^\flat$ program implies well-formedness of its iFJ counterpart.

Theorem 4.12 (Translation preserves well-formedness of programs). *If $\vdash^{\flat}$ prog* ok $\rightsquigarrow$ *prog′ then $\vdash_{\mathsf{iFJ}}$ prog′* ok.

Proof. See Section C.2.2. □

4.4.2 Translation Preserves Dynamic Semantics

The probably easiest way to show that the translation from CoreGI$^\flat$ to iFJ preserves the dynamic semantics would be to prove that translation commutes with evaluation. Commutativity of translation and evaluation is often depicted as a commuting diagram (Figure 4.21): it does not matter whether we first evaluate e to e' in CoreGI$^\flat$ and then translate e' to d', or first translate e to d and then evaluate d to d' in iFJ.

Unfortunately, the translation from CoreGI$^\flat$ to iFJ does *not* commute with evaluation because the expression d in Figure 4.21 does not necessarily reduce to d'. Instead, it may reduce to some other iFJ expression d'' such that d' and d'' differ modulo wrappers. In the following, two examples demonstrate in what ways d' and d'' possibly differ. These examples then motivate the definition of a type-directed equivalence relation on iFJ expressions that formalizes what we mean with "modulo wrappers". It turns out that this equivalence relation is sound with respect to contextual equivalence [153, 177] and that translation and evaluation commute modulo wrappers.

Figure 4.22 CoreGI$^\flat$ definitions used to illustrate non-commutativity.

```
interface I {}
class D {}
implementation I [D] {}
class E {Object obj}
class C {
  Object bar(I x){x}
      E foo(I x){new E(x)}
}
```

Examples

Consider the CoreGI$^\flat$ definitions in Figure 4.22.

1. It holds that

$$\Gamma \vdash^\flat \overbrace{\textbf{new}\; C().\mathit{bar}(\textbf{new}\; D())}^{=:e_1} : \mathit{Object} \rightsquigarrow \overbrace{\mathsf{new}\; C().\mathit{bar}(\mathsf{new}\; \mathit{Wrap}^I(\mathsf{new}\; D()))}^{=:d_1}$$

for any variable environment Γ, so we arrive at the following diagram:

$$\begin{tikzcd}
e_1 \arrow[r, "\flat"] \arrow[d, squiggly, "\Gamma \vdash^\flat e_1 : \mathit{Object} \rightsquigarrow d_1"'] & \textbf{new}\; D() \arrow[d, squiggly] \\
d_1 \arrow[r, "+", "\mathsf{iFJ}"'] & ???
\end{tikzcd}$$

However, evaluating d_1 and translating **new** $D()$ yield different results:

$$d_1 \longrightarrow_{\mathsf{iFJ}} \mathsf{new}\; \mathit{Wrap}^I(\mathsf{new}\; D())$$
$$\Gamma \vdash^\flat \textbf{new}\; D() : D \rightsquigarrow \mathsf{new}\; D()$$

2. It holds that

$$\Gamma \vdash^\flat \overbrace{\textbf{new}\; C().\mathit{foo}(\textbf{new}\; D())}^{=:e_2} : E \rightsquigarrow \overbrace{\mathsf{new}\; C().\mathit{foo}(\mathsf{new}\; \mathit{Wrap}^I(\mathsf{new}\; D()))}^{=:d_2}$$

for any variable environment Γ, so we arrive at the following diagram:

$$\begin{tikzcd}
e_2 \arrow[r, "\flat"] \arrow[d, squiggly, "\Gamma \vdash^\flat e_2 : E \rightsquigarrow d_2"'] & \textbf{new}\; E(\textbf{new}\; D()) \arrow[d, squiggly] \\
d_2 \arrow[r, "+", "\mathsf{iFJ}"'] & ???
\end{tikzcd}$$

However, evaluating d_2 and translating **new** $E($**new** $D())$ yield different results:

$$d_2 \longrightarrow_{\mathsf{iFJ}} \mathsf{new}\; E(\mathsf{new}\; \mathit{Wrap}^I(\mathsf{new}\; D()))$$
$$\Gamma \vdash^\flat \textbf{new}\; E(\textbf{new}\; D()) : E \rightsquigarrow \mathsf{new}\; E(\mathsf{new}\; D())$$

Figure 4.23 Auxiliaries for type-directed equivalence modulo wrappers.

$\boxed{\textsf{defines-field}(C, f)}$

$$\textsc{defines-field}\quad \frac{\textbf{class } C \textbf{ extends } N \textbf{ implements } \overline{J}\,\{\,\overline{U\ f}\ \ \ldots\}}{\textsf{defines-field}(C, f_i)}$$

$\boxed{\textsf{topmost}(T, m)}$

$$\textsc{topmost-class}\quad \frac{\begin{array}{c}\textbf{class } C \textbf{ extends } N \textbf{ implements } \overline{J}\,\{\ldots\ \overline{m : mdef}\}\\ \textsf{mtype}(m_i, N) \text{ undefined} \qquad (\forall j)\ \textsf{mtype}(m_i, J_j) \text{ undefined}\end{array}}{\textsf{topmost}(C, m_i)}$$

$$\textsc{topmost-iface}\quad \frac{\textbf{interface } I \textbf{ extends } \overline{J}\,\{\,\overline{m : msig}\,\}}{\textsf{topmost}(I, m_i)}$$

Type-Directed Equivalence Modulo Wrappers

The examples just shown suggest that two iFJ expressions should be considered equivalent if they are syntactically identical modulo removal of wrapper constructors. Further, the equivalence is type-directed in the sense that it allows the removal of wrapper constructors only at positions of certain types.

The definition of such a type-directed equivalence relation relies on two auxiliaries defined in Figure 4.23:

- $\textsf{defines-field}(C, f)$ asserts that class C defines a field of name f.
- $\textsf{topmost}(T, m)$ asserts that type T defines method m such that no supertype of T contains another definition of m.

Figure 4.24 formalizes *type-directed equivalence modulo wrappers*, written $\Gamma \vdash_{\textsf{iFJ}} e \equiv d : T$. This judgment states that under type environment Γ the iFJ expressions e and d are equivalent at type T. The rules EQUIV-VAR, EQUIV-FIELD, EQUIV-INVOKE, EQUIV-NEW-CLASS, EQUIV-CAST, EQUIV-GETDICT, and EQUIV-LET are similar to the corresponding iFJ typing rules for expressions (Figure 4.11); they simply assert that two expressions with the same top-level form are equivalent if their subexpressions are equivalent. The premises $\textsf{defines-field}(C, f_j)$ and $\textsf{topmost}(V, m)$ in rules EQUIV-FIELD and EQUIV-INVOKE, respectively, are required to show transitivity of the $\equiv$-relation. Rule EQUIV-FIELD-WRAPPED states that accesses of the *wrapped* field on two wrapper objects are equivalent if the objects being wrapped are equivalent. Rule EQUIV-NEW-WRAP defines equivalence between wrapper objects used at an interface type. Finally, the rules EQUIV-NEW-OBJECT-LEFT

Figure 4.24 Type-directed equivalence modulo wrappers.

$$\boxed{\Gamma \vdash_{\mathsf{iFJ}} e \equiv e' : T}$$

EQUIV-VAR

$$\frac{\vdash_{\mathsf{iFJ}} \Gamma(x) \leq T}{\Gamma \vdash_{\mathsf{iFJ}} x \equiv x : T}$$

EQUIV-FIELD

$$\frac{\Gamma \vdash_{\mathsf{iFJ}} e \equiv e' : C \qquad \mathsf{defines\text{-}field}(C, f_j) \qquad \mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\, f} \qquad \vdash_{\mathsf{iFJ}} U_j \leq T}{\Gamma \vdash_{\mathsf{iFJ}} e.f_j \equiv e'.f_j : T}$$

EQUIV-FIELD-WRAPPED

$$\frac{\Gamma \vdash_{\mathsf{iFJ}} e \equiv e' : \mathit{Object}}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\ \mathit{Wrap}^{I}(e).\mathit{wrapped} \equiv \textbf{new}\ \mathit{Wrap}^{J}(e').\mathit{wrapped} : \mathit{Object}}$$

EQUIV-INVOKE

$$\frac{\Gamma \vdash_{\mathsf{iFJ}} e \equiv e' : V \qquad \mathsf{topmost}(V, m) \qquad \mathsf{mtype}_{\mathsf{iFJ}}(m, V) = \overline{U\, x} \to U \qquad (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_i \equiv e'_i : U_i \qquad \vdash_{\mathsf{iFJ}} U \leq T}{\Gamma \vdash_{\mathsf{iFJ}} e.m(\overline{e}) \equiv e'.m(\overline{e'}) : T}$$

EQUIV-NEW-CLASS

$$\frac{\vdash_{\mathsf{iFJ}} N \leq T \qquad \mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{U\, f} \qquad (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_i \equiv e'_i : U_i}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{e}) \equiv \textbf{new}\, N(\overline{e'}) : T}$$

EQUIV-NEW-WRAP

$$\frac{\vdash_{\mathsf{iFJ}} I \leq J \qquad \vdash_{\mathsf{iFJ}} I' \leq J \qquad \Gamma \vdash_{\mathsf{iFJ}} e \equiv e' : \mathit{Object}}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\ \mathit{Wrap}^{I}(e) \equiv \textbf{new}\ \mathit{Wrap}^{I'}(e') : J}$$

EQUIV-NEW-OBJECT-LEFT

$$\frac{\Gamma \vdash_{\mathsf{iFJ}} e \equiv e' : \mathit{Object}}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\ \mathit{Wrap}^{I}(e) \equiv e' : \mathit{Object}}$$

EQUIV-NEW-OBJECT-RIGHT

$$\frac{\Gamma \vdash_{\mathsf{iFJ}} e \equiv e' : \mathit{Object}}{\Gamma \vdash_{\mathsf{iFJ}} e \equiv \textbf{new}\ \mathit{Wrap}^{I}(e') : \mathit{Object}}$$

EQUIV-CAST

$$\frac{\Gamma \vdash e \equiv e' : \mathit{Object} \qquad \vdash_{\mathsf{iFJ}} U \leq T}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{cast}(U, e) \equiv \textbf{cast}(U, e') : T}$$

EQUIV-GETDICT

$$\frac{\Gamma \vdash_{\mathsf{iFJ}} e \equiv e' : \mathit{Object} \qquad \vdash_{\mathsf{iFJ}} \mathit{Dict}^{I} \leq T}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{getdict}(I, e) \equiv \textbf{getdict}(I, e') : T}$$

EQUIV-LET

$$\frac{\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e'_1 : T \qquad \Gamma, x : T \vdash_{\mathsf{iFJ}} e_2 \equiv e'_2 : U}{\Gamma \vdash_{\mathsf{iFJ}} \textbf{let}\, T\, x = e_1\, \textbf{in}\, e_2 \equiv \textbf{let}\, T\, x = e'_1\, \textbf{in}\, e'_2 : U}$$

Figure 4.25 Visualization of Theorem 4.16.

$$\begin{tikzcd}
e \arrow[r, "_{\mathsf{iFJ}}"' near end] \arrow[d, equal, "\Gamma \vdash_{\mathsf{iFJ}} e \equiv d : T"'] & e' \arrow[d, equal, "\Gamma \vdash_{\mathsf{iFJ}} e' \equiv d' : T"] \\
d \arrow[r, "_{\mathsf{iFJ}}"' near end] & d'
\end{tikzcd}$$

and EQUIV-NEW-OBJECT-RIGHT allows the removal of a wrapper constructor when the two expressions involved are used at type *Object*.

Definition 4.13. Let Γ be a variable environment and T be a type. The set $\mathscr{E}_{\Gamma,T}$ is defined as the set containing all iFJ expressions e such that $\Gamma \vdash_{\mathsf{iFJ}} e : T'$ for some type T' with $\vdash_{\mathsf{iFJ}} T' \leq T$.

Theorem 4.14 ($\equiv$ is an equivalence relation). *Suppose that the* iFJ *program under consideration is well-formed and in the image of the translation from* $\mathsf{CoreGI}^\flat$ *to* iFJ*. Moreover, let* Γ *be a variable environment and* T *be a type. Then the relation* $\Gamma \vdash_{\mathsf{iFJ}} \cdot \equiv \cdot : T$ *is an equivalence relation over* $\mathscr{E}_{\Gamma,T}$*.*

Proof. See Section C.3.1. The proofs of reflexivity and symmetry do not rely on the assumption that the iFJ program under consideration is in the image of the translation from $\mathsf{CoreGI}^\flat$ to iFJ. □

The $\equiv$-relation is stable under substitution and evaluation.

Theorem 4.15 (Substitution preserves $\equiv$). *Suppose that the* iFJ *program under consideration is well-formed. If* $\Gamma, x : U \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$ *and* $\Gamma \vdash_{\mathsf{iFJ}} d_1 \equiv d_2 : U$ *then* $\Gamma \vdash_{\mathsf{iFJ}} [d_1/x]e_1 \equiv [d_2/x]e_2 : T$*.*

Proof. See Section C.3.2. □

Theorem 4.16 (Evaluation preserves $\equiv$). *Suppose that the* iFJ *program under consideration is well-formed. If* $\Gamma \vdash_{\mathsf{iFJ}} e \equiv d : T$ *and* $e \longrightarrow_{\mathsf{iFJ}} e'$ *then* $d \longrightarrow_{\mathsf{iFJ}} d'$ *such that* $\Gamma \vdash_{\mathsf{iFJ}} e' \equiv d' : T$*. In other words, the diagram in Figure 4.25 commutes.*

Proof. See Section C.3.3. □

Equivalence modulo wrappers relates only iFJ expressions that are contextually equivalent [153]. Informally, two expressions e_1 and e_2 of the same type T are contextually equivalent if no context is able to distinguish them. That is, if $d[e_1]$ is a well-typed expressions containing instances of e_1 and $d[e_2]$ is the expression obtained by replacing those instances by e_2, then $d[e_1]$ and $d[e_2]$ give exactly the same observable results when evaluated [177, Definition 7.3.2]. It is common to consider only termination and non-termination as observable results.

Expressions in iFJ do not provide binding constructs, so it is possible to build $d[e_1]$ and $d[e_2]$ from an expression d by substituting e_1 and e_2, respectively, for a designated

Figure 4.26 Visualization of Theorem 4.19.

$$\begin{tikzcd}
e_1 \arrow[rrr, tail, "\flat" near end] \arrow[ddd, squiggly, "\Gamma \vdash^\flat e_1 : T \leadsto e_1'"'] & & & e_2 \arrow[d, squiggly, "\Gamma \vdash^\flat e_2 : T' \leadsto e_2''"] \\
& & & e_2'' \arrow[d, dotted, "\vdash^\flat T' \leq T \leadsto I^?"] \\
& & & \mathsf{wrap}(I^?, e_2'') \arrow[d, equal, "\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e_2'') \equiv e_2' : T"] \\
e_1' \arrow[rrr, tail, "+"' near end, "\mathsf{iFJ}" near end] & & & e_2'
\end{tikzcd}$$

variable $\chi \in \mathit{VarName}$; that is, $d[e_1] := [e_1/\chi]d$ and $d[e_2] := [e_2/\chi]d$. This leads to a formal definition of contextual equivalence in iFJ.

Definition 4.17 (Contextual equivalence in iFJ). Assume that e_1 and e_2 are two iFJ expressions such that $\Gamma \vdash_{\mathsf{iFJ}} e_1 : T_1$ and $\Gamma \vdash_{\mathsf{iFJ}} e_2 : T_2$ with $\vdash_{\mathsf{iFJ}} T_1 \leq T$ and $\vdash_{\mathsf{iFJ}} T_2 \leq T$ for some type T. Then e_1 and e_2 are *contextually equivalent* at value environment Γ and type T, written $\Gamma \vdash_{\mathsf{iFJ}} e_1 =_{\mathrm{ctx}} e_2 : T$, if, and only if, for any expression d with $\Gamma, \chi : T \vdash_{\mathsf{iFJ}} d : U$ for some type U, it holds that either both $[e_1/\chi]d$ and $[e_2/\chi]d$ diverge or both $[e_1/\chi]d$ and $[e_2/\chi]d$ terminate.

The following theorem verifies the claim that equivalence modulo wrappers relates only iFJ expressions that are contextually equivalent.

Theorem 4.18 ($\equiv$ is sound with respect to contextual equivalence). *Suppose that the underlying iFJ program is well-formed. If* $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$ *then* $\Gamma \vdash_{\mathsf{iFJ}} e_1 =_{\mathrm{ctx}} e_2 : T$.

Proof. Follows with Theorems 4.15 and 4.16. See Section C.3.4 for details. □

Equivalence modulo wrappers is not complete with respect to contextual equivalence. For example, given the class definition

$$\begin{array}{l} \textbf{class}\ C\ \{ \\ \quad C\ m()\{\mathit{this}\} \\ \} \end{array}$$

it obviously holds that $\emptyset \vdash_{\mathsf{iFJ}} \textbf{new}\, C() =_{\mathrm{ctx}} \textbf{new}\, C().m() : C$ but the equivalence $\emptyset \vdash_{\mathsf{iFJ}} \textbf{new}\, C() \equiv \textbf{new}\, C().m() : C$ is not derivable.

Translation and Evaluation Commute Modulo Wrappers

The following theorem states that the translation from CoreGI$^\flat$ to iFJ commutes modulo wrappers with single-step evaluation in CoreGI$^\flat$ and multi-step evaluation in iFJ.

Theorem 4.19. *Suppose that the underlying CoreGI$^\flat$ program prog is well-formed and that the underlying iFJ program is the translation of prog. If* $\Gamma \vdash^\flat e_1 : T \leadsto e_1'$ *and*

Figure 4.27 Visualization of Theorem 4.20.

$$\begin{tikzcd}
e_0 \arrow[r, "\flat *"] \arrow[ddd, squiggly, "\Gamma \vdash^{\flat} e_0 : T \rightsquigarrow e'_0"'] & e_n \arrow[d, squiggly, "\Gamma \vdash^{\flat} e_n : T' \rightsquigarrow e'"] \\
& e' \arrow[d, dotted, "\vdash^{\flat} T' \leq T \rightsquigarrow I^?"] \\
& \mathsf{wrap}(I^?, e') \arrow[d, equal, "\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e') \equiv e : T"] \\
e'_0 \arrow[r, "*"', "\mathsf{iFJ}"] & e
\end{tikzcd}$$

$e_1 \longrightarrow^{\flat} e_2$, *then* $e'_1 \longrightarrow^{+}_{\mathsf{iFJ}} e'_2$ *such that* $\Gamma \vdash^{\flat} e_2 : T' \rightsquigarrow e''_2$ *and* $\vdash^{\flat} T' \leq T \rightsquigarrow I^?$ *and* $\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e''_2) \equiv e'_2 : T$. *In other words, the diagram in Figure 4.26 commutes.*

Proof. It suffices to prove the following claim:

> *If* $\Gamma \vdash^{\flat} \mathcal{E}[d_1] : T \rightsquigarrow e_1$ *and* $d_1 \longmapsto^{\flat} d_2$, *then* $e_1 \longrightarrow^{+}_{\mathsf{iFJ}} e_2$ *such that* $\Gamma \vdash^{\flat} \mathcal{E}[d_2] : T' \rightsquigarrow e'_2$ *and* $\vdash^{\flat} T' \leq T \rightsquigarrow I^?$ *and* $\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e'_2) \equiv e_2 : T$.

The proof of this claim is by induction on the structure of $\mathcal{E}$. See Section C.3.5 for details. □

A generalization of Theorem 4.19 considers multi-step evaluation in $\mathsf{CoreGI}^{\flat}$ instead of single-step evaluation.

Theorem 4.20 (Translation and evaluation commute modulo wrappers). *Suppose that the underlying $\mathsf{CoreGI}^{\flat}$ program prog is well-formed and that the underlying iFJ program is the translation of prog. If* $\Gamma \vdash^{\flat} e_0 : T \rightsquigarrow e'_0$ *and* $e_0 \longrightarrow^{\flat *} e_n$, *then* $e'_0 \longrightarrow^{*}_{\mathsf{iFJ}} e$ *such that* $\Gamma \vdash^{\flat} e_n : T' \rightsquigarrow e'$ *and* $\vdash^{\flat} T' \leq T \rightsquigarrow I^?$ *and* $\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e') \equiv e : T$. *In other words, the diagram in Figure 4.27 commutes.*

Proof. The proof is by induction on the length n of the evaluation sequence $e_0 \longrightarrow^{\flat *} e_n$. For the case $n > 0$, Figure 4.28 sketches the proof idea: commutativity of (a) follows from Theorem 4.19; an application of the induction hypothesis yields commutativity of (b); Commutativity of (c) holds trivially; commutativity of (d) follows with Theorem 4.16. Section C.3.6 gives all details of the proof. □

Figure 4.28 Proof sketch for Theorem 4.20.

$$\begin{tikzcd}
e_0 \arrow[r, "\flat"] \arrow[ddddd, rightsquigarrow] \arrow[dddddr, phantom, "\text{(a)}"] & e_1 \arrow[r, "\flat *"] \arrow[ddd, rightsquigarrow] \arrow[ddr, phantom, "\text{(b)}"] & e_n \arrow[d, rightsquigarrow] \\
 & & e' \arrow[d, dotted] \\
 & & \mathsf{wrap}(J'^{?}, e') \arrow[d, phantom, sloped, "\equiv"] \\
 & e''_1 \arrow[r, "*_{\mathsf{iFJ}}"] \arrow[d, dotted] \arrow[dr, phantom, "\text{(c)}"] & d \arrow[d, dotted] \\
 & \mathsf{wrap}(J^{?}, e''_1) \arrow[r, "*_{\mathsf{iFJ}}"] \arrow[d, phantom, sloped, "\equiv"] \arrow[dr, phantom, "\text{(d)}"] & \mathsf{wrap}(J^{?}, d) \arrow[d, phantom, sloped, "\equiv"] \\
e'_0 \arrow[r, "+_{\mathsf{iFJ}}"] & e'_1 \arrow[r, "*_{\mathsf{iFJ}}"] & e
\end{tikzcd}$$

4.5 Relating CoreGI$^\flat$ and CoreGI

Chapter 3 introduced the calculus CoreGI and Section 4.1 defined CoreGI$^\flat$as a simplified version of CoreGI. This section formally proves that CoreGI$^\flat$ is a subset of CoreGI. As a consequence, meta-theoretical properties of CoreGI—for example, type soundness and deterministic evaluation—automatically hold for CoreGI$^\flat$ too.

Figure 4.29 defines a restricted variant of CoreGI's syntax. The figure highlights differences with respect to the definition of CoreGI's original syntax in Figure 3.1 on page 32. Obviously, each syntactic phrase that is valid according to the syntax in Figure 4.29 is also valid according to the syntax in Figure 3.1.

Definition 4.21. A syntactic phrase of CoreGI is said to be *restricted* if, and only if, it is valid according to the syntax in Figure 4.29.

Figure 4.30 defines a family of functions mapping syntactic phrases of CoreGI$^\flat$ to restricted syntactic phrases of CoreGI. More specifically, function $\mathcal{B}_{\mathrm{p}}$ maps CoreGI$^\flat$ programs to restricted CoreGI programs, $\mathcal{B}_{\mathrm{d}}$ maps CoreGI$^\flat$ definitions to restricted CoreGI definitions, $\mathcal{B}_{\mathrm{ms}}$ maps CoreGI$^\flat$ method signatures to restricted CoreGI method signatures, $\mathcal{B}_{\mathrm{md}}$ maps CoreGI$^\flat$ method definitions to restricted CoreGI method definitions, $\mathcal{B}_{\mathrm{t}}$ maps CoreGI$^\flat$ types to restricted CoreGI types, and $\mathcal{B}_{\mathrm{e}}$ maps CoreGI$^\flat$ expressions to restricted CoreGI expressions. The working of most of these functions is straightforward. Function $\mathcal{B}_{\mathrm{d}}$ maps the **extends** clause of a CoreGI$^\flat$ interface to superinterface constraints of CoreGI.

All functions defined in Figure 4.30 are invertible because they are bijective, as stated in the following theorem:

Figure 4.29 Restricted syntax of CoreGI.

$$
\begin{array}{rcll}
\mathit{prog} & ::= & \overline{\mathit{def}}\ e & \\
\mathit{def} & ::= & \mathit{cdef} \mid \mathit{idef} \mid \mathit{impl} & \\
\mathit{cdef} & ::= & \textbf{class}\ C\texttt{<}\bullet\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \bullet\ \{\,\overline{T\ f}\ \overline{m : \mathit{mdef}}\,\} & \\
\mathit{idef} & ::= & \textbf{interface}\ I\texttt{<}\bullet\texttt{>}\,[\ \mathit{This}\ \textbf{where}\ \overline{\mathit{This}\ \textbf{implements}\ I\texttt{<}\bullet\texttt{>}}\ ]\ \textbf{where}\ \bullet & \\
 & & \quad \{\ \bullet\ \textbf{receiver}\ \{\overline{m : \mathit{msig}}\}\ \} & \\
\mathit{impl} & ::= & \textbf{implementation}\texttt{<}\bullet\texttt{>}\ K\ [\ N\ ]\ \textbf{where}\ \bullet\ \{\ \bullet\ \textbf{receiver}\ \{\overline{\mathit{mdef}}\}\ \} & \\
\mathit{msig} & ::= & \texttt{<}\bullet\texttt{>}\,\overline{T\,x} \to T\ \textbf{where}\ \bullet & \\
\mathit{mdef} & ::= & \mathit{msig}\,\{e\} & \\
M, N & ::= & C\texttt{<}\bullet\texttt{>} \mid \mathit{Object} & \\
G, H & ::= & N & \text{no type variables} \\
K, L & ::= & I\texttt{<}\bullet\texttt{>} & \\
T, U, V, W & ::= & G \mid K & \\
d, e & ::= & x \mid e.f \mid e.m\texttt{<}\bullet\texttt{>}(\overline{e}) \mid \textbf{new}\ N(\overline{e}) \mid (T)\,e & \text{no calls of static interface methods}
\end{array}
$$

$$\mathit{This} \in \mathit{TvarName}\ \text{(fixed)}$$

Theorem 4.22. *The functions $\mathcal{B}_\mathrm{p}$, $\mathcal{B}_\mathrm{d}$, $\mathcal{B}_\mathrm{ms}$, $\mathcal{B}_\mathrm{md}$, $\mathcal{B}_\mathrm{t}$, and $\mathcal{B}_\mathrm{e}$ are bijections between CoreGI$^\flat$ and restricted CoreGI programs, definitions, method signatures, method definitions, types, and expressions, respectively.*

Proof. Obviously, $\mathcal{B}_\mathrm{t}$ is injective. A straightforward induction on the structure of CoreGI$^\flat$ expressions then shows that $\mathcal{B}_\mathrm{e}$ is injective. It is then easy to show that $\mathcal{B}_\mathrm{ms}$, $\mathcal{B}_\mathrm{md}$, $\mathcal{B}_\mathrm{d}$, and $\mathcal{B}_\mathrm{p}$ are injections as well.

Similarly, $\mathcal{B}_\mathrm{t}$ is clearly surjective on the set of restricted CoreGI types. An easy induction on the structure of restricted CoreGI expressions then shows that $\mathcal{B}_\mathrm{e}$ is also surjective. Now it is straightforward to verify that $\mathcal{B}_\mathrm{ms}$, $\mathcal{B}_\mathrm{md}$, $\mathcal{B}_\mathrm{d}$, and $\mathcal{B}_\mathrm{p}$ are surjective as well. □

Besides removing certain syntactic constructs from CoreGI, it is also necessary to remove CoreGI's support for covariant return types because CoreGI$^\flat$ requires invariant return types.

Definition 4.23. A restricted CoreGI program has *invariant return types* if, and only if, the following two conditions hold:

1. Assume

$$\textbf{class}\ C\texttt{<}\bullet\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \bullet\ \{\dots\ \overline{m : \mathit{mdef}}\,\}$$

$$\textbf{class}\ D\texttt{<}\bullet\texttt{>}\ \textbf{extends}\ N'\ \textbf{where}\ \bullet\ \{\dots\ \overline{m' : \mathit{mdef}'}\,\}$$

such that $D\texttt{<}\bullet\texttt{>} \trianglelefteq_\mathrm{c} C\texttt{<}\bullet\texttt{>}$ and $m_i = m'_j$. If $\mathit{mdef}_i = \texttt{<}\bullet\texttt{>}\,\overline{T\,x} \to T\ \textbf{where}\ \bullet\ \{e\}$ and $\mathit{mdef}'_j = \texttt{<}\bullet\texttt{>}\,\overline{U\,y} \to U\ \textbf{where}\ \bullet\ \{d\}$ then $T = U$.

Figure 4.30 Bijections between $\mathsf{CoreGI}^\flat$ and the restricted variant of CoreGI.

$$
\begin{aligned}
\mathcal{B}_{\mathrm{p}}[\![\,\overline{def}\ e\,]\!] &= \overline{\mathcal{B}_{\mathrm{d}}[\![def_i]\!]}\ \mathcal{B}_{\mathrm{e}}[\![e]\!] \\
\mathcal{B}_{\mathrm{d}}\left[\!\!\left[\begin{array}{l}\textbf{class } C \textbf{ extends } N \\ \quad\{\,\overline{T\,f}\ \overline{m : mdef}\,\}\end{array}\right]\!\!\right] &= \begin{array}[t]{l}\textbf{class } C\texttt{<}\bullet\texttt{>} \textbf{ extends } \mathcal{B}_{\mathrm{t}}[\![N]\!] \textbf{ where } \bullet \\ \quad\{\,\overline{\mathcal{B}_{\mathrm{t}}[\![T_i]\!]\ f_i}\ \overline{m_i : \mathcal{B}_{\mathrm{md}}[\![mdef_i]\!]}\,\}\end{array} \\
\mathcal{B}_{\mathrm{d}}\left[\!\!\left[\begin{array}{l}\textbf{interface } I \textbf{ extends } \overline{I} \\ \quad\{\,\overline{m : msig}\,\}\end{array}\right]\!\!\right] &= \begin{array}[t]{l}\textbf{interface } I\texttt{<}\bullet\texttt{>} \\ \quad[\,\mathit{This} \textbf{ where } \overline{\mathit{This} \textbf{ implements } I_i\texttt{<}\bullet\texttt{>}}\,] \\ \quad\textbf{where } \bullet\ \{\,\bullet \textbf{ receiver } \{\,\overline{m_i : \mathcal{B}_{\mathrm{ms}}[\![msig_i]\!]}\,\}\,\}\end{array} \\
\mathcal{B}_{\mathrm{d}}\left[\!\!\left[\begin{array}{l}\textbf{implementation } I\ [N] \\ \quad\{\,\overline{mdef}\,\}\end{array}\right]\!\!\right] &= \begin{array}[t]{l}\textbf{implementation}\texttt{<}\bullet\texttt{>}\ \mathcal{B}_{\mathrm{t}}[\![I]\!]\ \ [\,\mathcal{B}_{\mathrm{t}}[\![N]\!]\,] \\ \quad\textbf{where } \bullet\ \{\,\bullet \textbf{ receiver } \{\,\overline{\mathcal{B}_{\mathrm{md}}[\![mdef_i]\!]}\,\}\,\}\end{array} \\
\mathcal{B}_{\mathrm{ms}}[\![\,\overline{T\,x} \to T\,]\!] &= \texttt{<}\bullet\texttt{>}\,\overline{\mathcal{B}_{\mathrm{t}}[\![T_i]\!]\ x_i} \to \mathcal{B}_{\mathrm{t}}[\![T]\!] \textbf{ where } \bullet \\
\mathcal{B}_{\mathrm{md}}[\![msig\ \{e\}]\!] &= \mathcal{B}_{\mathrm{ms}}[\![msig]\!]\ \{\mathcal{B}_{\mathrm{e}}[\![e]\!]\} \\
\mathcal{B}_{\mathrm{t}}[\![\mathit{Object}]\!] &= \mathit{Object} \\
\mathcal{B}_{\mathrm{t}}[\![C]\!] &= C\texttt{<}\bullet\texttt{>} \\
\mathcal{B}_{\mathrm{t}}[\![I]\!] &= I\texttt{<}\bullet\texttt{>} \\
\mathcal{B}_{\mathrm{e}}[\![x]\!] &= x \\
\mathcal{B}_{\mathrm{e}}[\![e.f]\!] &= \mathcal{B}_{\mathrm{e}}[\![e]\!].f \\
\mathcal{B}_{\mathrm{e}}[\![e.m(\overline{e})]\!] &= \mathcal{B}_{\mathrm{e}}[\![e]\!].m\texttt{<}\bullet\texttt{>}(\overline{\mathcal{B}_{\mathrm{e}}[\![e_i]\!]}) \\
\mathcal{B}_{\mathrm{e}}[\![\textbf{new } N(\overline{e})]\!] &= \textbf{new } N(\overline{\mathcal{B}_{\mathrm{e}}[\![e_i]\!]}) \\
\mathcal{B}_{\mathrm{e}}[\![(T)\,e]\!] &= (\mathcal{B}_{\mathrm{t}}[\![T]\!])\,\mathcal{B}_{\mathrm{e}}[\![e]\!]
\end{aligned}
$$

2. Assume

$$
\begin{array}{l}
\textbf{interface } I\texttt{<}\bullet\texttt{>}[\,\mathit{This} \textbf{ where } \overline{\mathit{This} \textbf{ implements } I_i}\,] \textbf{ where } \bullet \\
\quad\{\,\bullet \textbf{ receiver } \{\,\overline{m : msig}\,\}\,\}
\end{array}
$$

$$
\textbf{implementation}\texttt{<}\bullet\texttt{>}\ I\texttt{<}\bullet\texttt{>}\ [\,N\,] \textbf{ where } \bullet\ \ \{\,\bullet \textbf{ receiver } \{\,\overline{mdef}\,\}\,\}
$$

If $msig_i = \texttt{<}\bullet\texttt{>}\overline{T\,x} \to T \textbf{ where } \bullet$ and $mdef_i = \texttt{<}\bullet\texttt{>}\overline{U\,y} \to U \textbf{ where } \bullet\ \{e\}$ then $T = U$.

The following theorems now show that the dynamic and the static semantics of $\mathsf{CoreGI}^\flat$ (as specified in Sections 4.1 and 4.3) are equivalent to the dynamic and the static semantics of CoreGI (as defined in Chapter 3), provided all CoreGI constructs involved are restricted and the CoreGI program under consideration has invariant return types. The rest of this section implicitly assumes that all CoreGI constructs mentioned are restricted and that the underlying CoreGI program is the image according to $\mathcal{B}_{\mathrm{p}}$ of the underlying $\mathsf{CoreGI}^\flat$ program. Further, the notation $\mathcal{B}^{-1}$ denotes the *inverse* of some function $\mathcal{B}$.

Theorem 4.24 (Equivalence of subtyping). *If* $\vdash^\flat T \leq U$ *then* $\Delta \vdash \mathcal{B}_\mathrm{t}[\![T]\!] \leq \mathcal{B}_\mathrm{t}[\![U]\!]$ *for any type environment* Δ*. Furthermore, if* $\emptyset \vdash V \leq W$ *then* $\vdash^\flat \mathcal{B}_\mathrm{t}^{-1}[\![V]\!] \leq \mathcal{B}_\mathrm{t}^{-1}[\![W]\!]$*.*

Proof. See Section C.4.1. □

Theorem 4.25 (Equivalence of dynamic semantics).

(*i*) *If* $e \longmapsto^\flat e'$ *then* $\mathcal{B}_\mathrm{e}[\![e]\!] \longmapsto \mathcal{B}_\mathrm{e}[\![e']\!]$*.*

(*ii*) *If* $e \longmapsto e'$ *then* $\mathcal{B}_\mathrm{e}^{-1}[\![e]\!] \longmapsto^\flat \mathcal{B}_\mathrm{e}^{-1}[\![e']\!]$*.*

(*iii*) *If* $e \longrightarrow^\flat e'$ *then* $\mathcal{B}_\mathrm{e}[\![e]\!] \longrightarrow \mathcal{B}_\mathrm{e}[\![e']\!]$*.*

(*iv*) *If* $e \longrightarrow e'$ *then* $\mathcal{B}_\mathrm{e}^{-1}[\![e]\!] \longrightarrow^\flat \mathcal{B}_\mathrm{e}^{-1}[\![e']\!]$*.*

Proof. See Section C.4.2. □

The next theorem extends the definition of $\mathcal{B}_\mathrm{t}$ to value environments Γ by applying $\mathcal{B}_\mathrm{t}$ pointwise to all types in Γ.

Theorem 4.26 (Equivalence of expression typing).

(*i*) *Assume* $\vdash^\flat U$ ok *for all* $x : U \in \Gamma$*. If* $\Gamma \vdash^\flat e : T$ *then* $\Delta; \mathcal{B}_\mathrm{t}[\![\Gamma]\!] \vdash \mathcal{B}_\mathrm{e}[\![e]\!] : \mathcal{B}_\mathrm{t}[\![T]\!]$ *for any type environment* Δ*.*

(*ii*) *Assume* $\emptyset \vdash U$ ok *for all* $x : U \in \Gamma$*. If* $\emptyset; \Gamma \vdash e : T$ *then* $\mathcal{B}_\mathrm{t}^{-1}[\![\Gamma]\!] \vdash^\flat \mathcal{B}_\mathrm{e}^{-1}[\![e]\!] : U$ *for some* U *with* $\vdash^\flat U \leq \mathcal{B}_\mathrm{t}^{-1}[\![T]\!]$*.*

Proof. See Section C.4.3. □

Theorem 4.27 (Equivalence of program typing).

(*i*) *If prog is a CoreGI*$^\flat$ *program such that* $\vdash^\flat$ *prog* ok*, then* $\vdash \mathcal{B}_\mathrm{p}[\![prog]\!]$ ok *and* $\mathcal{B}_\mathrm{p}[\![prog]\!]$ *has invariant return types.*

(*ii*) *If prog is a restricted CoreGI program with invariant return types and* $\vdash$ *prog* ok*, then* $\vdash^\flat \mathcal{B}_\mathrm{p}^{-1}[\![prog]\!]$ ok*.*

Proof. See Section C.4.4. □

Now it is straightforward to prove type soundness and deterministic evaluation for CoreGI$^\flat$.

Definition 4.28 (Stuck on a bad cast for CoreGI$^\flat$). A CoreGI$^\flat$ expression e is *stuck on a bad cast* if, and only if, there exists an evaluation context $\mathcal{E}$, a type T, and a value $v = \mathbf{new}\, N(\overline{w})$ such that $e = \mathcal{E}[(T)\, v]$ and not $\vdash^\flat N \leq T$.

Theorem 4.29 (Type soundness for CoreGI$^\flat$). *Assume that the underlying CoreGI*$^\flat$ *program is well-formed. If* $\emptyset \vdash^\flat e : T$ *then either* e *diverges, or* $e \longrightarrow^{\flat *} v$ *for some value* v *such that* $\emptyset \vdash^\flat v : T'$ *for some* T' *with* $\vdash^\flat T' \leq T$*, or* $e \longrightarrow^{\flat *} e'$ *for some expression* e' *such that* e' *is stuck on a bad cast.*

Proof. Follows easily using Theorem 3.17, Theorem 4.22, Theorem 4.24, Theorem 4.25 Theorem 4.26, and Theorem 4.27. □

Theorem 4.30 (Determinacy of evaluation for $\mathsf{CoreGI}^\flat$). *Assume that the underlying $\mathsf{CoreGI}^\flat$ program is well-formed. If $e \longrightarrow^\flat e'$ and $e \longrightarrow^\flat e''$ then $e' = e''$.*

Proof. Follows from Theorem 3.20, Theorem 4.25, and Theorem 4.27. □

5
Extensions

Developing a new programming language involves exploring the boundaries of the design space. Whereas the two preceding chapters formalized only features that are present in the full JavaGI language, the chapter at hand defines two extensions of JavaGI's subtyping relation that both lead to undecidability of subtyping and are thus not included in the full language, at least not without further restrictions.

Chapter Outline. The chapter consists of two sections:

- Section 5.1 defines the calculus IIT, which increases the flexibility of retroactive interface implementations by supporting interfaces as implementing types. (The calculus CoreGI from Chapter 3 supports only classes as implementing types.) The section proves that subtyping in IIT is undecidable and presents several restrictions that ensure decidability. The full JavaGI language features interfaces as implementing types under one of these restrictions.

- Section 5.2 studies the calculus EXuplo, which supports bounded existential types [40] with lower and upper bounds. Subtyping in EXuplo is shown to be undecidable, but there exist two decidable fragments. The full JavaGI language does not provide bounded existential types because both decidable fragments are too weak to be of practical value. The results in Section 5.2 are not only relevant to JavaGI's full type system, but also to Scala [166] and formal systems for modeling Java wildcards [228, 38, 37].

5.1 Interfaces as Implementing Types

In Java, only classes may implement interfaces. Consequently, the calculus CoreGI from Chapter 3 supports only classes as implementing types of retroactive interface implementations. However, it is sometimes desirable to implement the methods of an interface in terms of the methods declared by some other interface. For example, the interface `EQ` from Section 2.1.2 defines a single method `boolean eq(This that)` that compares

Figure 5.1 Syntax and subtyping for IIT.

$\boxed{\text{Syntax}}$

$$\begin{aligned} T, U, V, W &::= X \mid I\texttt{<}\overline{T}\texttt{>} \\ \mathit{def} &::= \textbf{interface}\ I\texttt{<}\overline{X}\texttt{>} \mid \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\,[I\texttt{<}\overline{T}\texttt{>}] \\ & X, Y, Z \in \mathit{TvarName}_{\mathsf{IIT}} \qquad I, J \in \mathit{IfaceName}_{\mathsf{IIT}} \end{aligned}$$

$\boxed{\vdash_{\mathrm{i}} T \leq U}$

$$\textsc{iit-refl}\quad \vdash_{\mathrm{i}} T \leq T \qquad \textsc{iit-trans}\quad \frac{\vdash_{\mathrm{i}} T \leq U \qquad \vdash_{\mathrm{i}} U \leq V}{\vdash_{\mathrm{i}} T \leq V} \qquad \textsc{iit-impl}\quad \frac{\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\,[J\texttt{<}\overline{U}\texttt{>}]}{\vdash_{\mathrm{i}} [\overline{V/X}]J\texttt{<}\overline{U}\texttt{>} \leq [\overline{V/X}]I\texttt{<}\overline{T}\texttt{>}}$$

two objects for equality. Implementing `eq` for lists simply requires to iterate over the two lists involved and to compare the individual elements for equality. Iterating over the elements of a list is readily available through method `Iterator<X> iterator()` of interface `List<X>`, so Section 2.1.3 provides a retroactive implementation of `EQ` with the interface type `List<X>` acting as the implementing type.[1]

As already mentioned, CoreGI supports only classes as implementing types, so it is unclear whether the decidability result for subtyping in CoreGI (see Section 3.7.1) also holds if interfaces are allowed as implementing types. To answer this question, Section 5.1.1 defines the calculus IIT, which models the essential aspects of subtyping in the presence of retroactive implementations with interfaces as implementing types. Section 5.1.2 shows that subtyping in IIT is undecidable by reduction from Post's Correspondence Problem [182]. Finally, Section 5.1.3 presents several decidable fragments of IIT, one of which serves as the basis for the "no implementation chains" restriction imposed in Section 2.3.4 on the full JavaGI language.

5.1.1 The Calculus IIT

Figure 5.1 defines the syntax along with the subtyping relation of IIT. As usual, overbar notation denotes sequencing (see Definition 3.1). A type T is either a type variable X or an interface type $I\texttt{<}\overline{T}\texttt{>}$. For simplicity, there are no class types. A definition *def* is either an interface or an (retroactive) implementation definition. Definitions do not contain methods and there is no support for interface inheritance because these aspects are irrelevant to the decidability issues discussed here. A definition $\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\,[J\texttt{<}\overline{U}\texttt{>}]$ implicitly assumes that $\overline{X} = \mathsf{ftv}(J\texttt{<}\overline{U}\texttt{>})$, where $\mathsf{ftv}(\xi)$ denotes the set of type variables free in ξ. Further, each occurrence of a type $I\texttt{<}\overline{T}^{n}\texttt{>}$ implicitly assumes the existence of a definition of interface I with n type parameters.

[1] The interfaces `List<X>` and `Iterator<X>` are part of the package `java.util` of the standard Java 1.5 API [212].

The judgment $\vdash_{\mathrm{i}} T \leq U$, also defined in Figure 5.1, states that T is a subtype of U in IIT. The subtyping relation is reflexive and transitive as usual and incorporates retroactive interface implementations through rule IIT-IMPL. The notation $[\overline{T/X}]$ denotes the capture-avoiding type substitution replacing each X_i with T_i.

5.1.2 Undecidability of Subtyping in IIT

The undecidability of subtyping in IIT follows by reduction from Post's Correspondence Problem (PCP). In the following, Σ ranges over finite alphabets, Σ^* denotes the set of words over Σ, η and ζ range over elements from Σ^*, and ε denotes the empty word.

Definition 5.1 (PCP). Let $\{(\eta_1, \zeta_1) \ldots, (\eta_n, \zeta_n)\}$ be a set of pairs of non-empty words over some finite alphabet Σ with at least two elements. A solution of PCP is a sequence of indices $i_1 \ldots i_r$ such that $\eta_{i_1} \ldots \eta_{i_r} = \zeta_{i_1} \ldots \zeta_{i_r}$. The decision problem asks whether such a solution exists.

Fact 5.2. The decision problem for PCP is undecidable [182, 90].

Theorem 5.3. *Subtyping in IIT is undecidable.*

Proof. Let $\mathcal{P} = \{(\eta_1, \zeta_1), \ldots, (\eta_n, \zeta_n)\}$ be a particular instance of PCP over the alphabet Σ. The encoding of $\mathcal{P}$ as an equivalent subtyping problem in IIT looks as follows. First, words over Σ must be represented as types in IIT.

$$\begin{array}{ll} \textbf{interface } \mathbb{E} & \text{(empty word } \varepsilon) \\ \textbf{interface } L\texttt{<}X\texttt{>} & \text{(letter, for every } L \in \Sigma) \end{array}$$

Words $\eta \in \Sigma^*$ are formed with these interfaces through nested interface types. For example, the word AB is represented by $A\texttt{<}B\texttt{<}\mathbb{E}\texttt{>}\texttt{>}$ Formally, the representation of a word u is $[\![\eta]\!] := \eta \mathbin{\#} \mathbb{E}$, where $\eta \mathbin{\#} T$ is the concatenation of η with a type T:

$$\begin{aligned} \varepsilon \mathbin{\#} T &:= T \\ L\eta \mathbin{\#} T &:= L\texttt{<}\eta \mathbin{\#} T\texttt{>} \qquad \textit{for every } L \in \Sigma \end{aligned}$$

Two interfaces are required to model the search for a solution of PCP:

$$\begin{array}{ll} \textbf{interface } \mathbb{S}\texttt{<}X, Y\texttt{>} & \text{(search state)} \\ \textbf{interface } \mathbb{G} & \text{(search goal)} \end{array}$$

The type $\mathbb{S}\texttt{<}[\![\eta]\!], [\![\zeta]\!]\texttt{>}$ represents a particular search state with accumulated indices $i_1, \ldots, i_k$ such that $\eta = \eta_{i_1} \ldots \eta_{i_k}$ and $\zeta = \zeta_{i_1} \ldots \zeta_{i_k}$. To model valid transitions between search states requires retroactive implementations of $\mathbb{S}$ for all $i \in \{1, \ldots, n\}$:

$$\textbf{implementation}\texttt{<}X, Y\texttt{>}\ \mathbb{S}\texttt{<}\eta_i \mathbin{\#} X, \zeta_i \mathbin{\#} Y\texttt{>}\ [\mathbb{S}\texttt{<}X, Y\texttt{>}] \tag{5.1}$$

The type $\mathbb{G}$ represents the goal of a search, as expressed by the following implementation:

$$\textbf{implementation}\texttt{<}X\texttt{>}\ \mathbb{G}\ [\mathbb{S}\texttt{<}X, X\texttt{>}] \tag{5.2}$$

It now holds that $\mathcal{P}$ has a solution if, and only if, there exists some $i \in \{1, \ldots, n\}$ such that $\vdash_{\mathrm{i}} \mathbb{S}\texttt{<}[\![\eta_i]\!], [\![\zeta_i]\!]\texttt{>} \leq \mathbb{G}$ is derivable. See Section D.1.1 for details. $\square$

Example. Suppose the PCP instance $\mathcal{P} = \{(\eta_1, \zeta_1), (\eta_2, \zeta_2)\}$ with $\eta_1 = A$, $\eta_2 = ABA$, $\zeta_1 = AA$, and $\zeta_2 = B$ is given. The instance has the solution $1, 2, 1$ because $\eta_1\eta_2\eta_1 = \zeta_1\zeta_2\zeta_1 = AABAA$. The IIT encoding of this problem looks like this:

$$\begin{array}{lll} \textbf{interface } \mathbb{E} & \textbf{interface } A\texttt{<}X\texttt{>} & \textbf{interface } B\texttt{<}X\texttt{>} \\ \textbf{interface } \mathbb{S}\texttt{<}X, Y\texttt{>} & \textbf{interface } \mathbb{G} & \end{array}$$

$$\textbf{implementation}\texttt{<}X, Y\texttt{>}\ \mathbb{S}\texttt{<}A\texttt{<}X\texttt{>}, A\texttt{<}A\texttt{<}Y\texttt{>>>}\ [\mathbb{S}\texttt{<}X, Y\texttt{>}] \tag{5.3}$$

$$\textbf{implementation}\texttt{<}X, Y\texttt{>}\ \mathbb{S}\texttt{<}A\texttt{<}B\texttt{<}A\texttt{<}X\texttt{>>>}, B\texttt{<}Y\texttt{>>}\ [\mathbb{S}\texttt{<}X, Y\texttt{>}] \tag{5.4}$$

$$\textbf{implementation}\texttt{<}X\texttt{>}\ \mathbb{G}\ [\mathbb{S}\texttt{<}X, X\texttt{>}] \tag{5.5}$$

Define

$$\begin{aligned} T_1 &= \mathbb{S}\texttt{<}[\![\eta_1]\!], [\![\zeta_1]\!]\texttt{>} = \mathbb{S}\texttt{<}[\![A]\!], [\![AA]\!]\texttt{>} \\ T_2 &= \mathbb{S}\texttt{<}[\![ABAA]\!], [\![BAA]\!]\texttt{>} \\ T_3 &= \mathbb{S}\texttt{<}[\![AABAA]\!], [\![AABAA]\!]\texttt{>} \end{aligned}$$

Applications of rule IIT-IMPL with implementations (5.4), (5.3), and (5.5) yield $\vdash_{\mathrm{i}} T_1 \le T_2$, $\vdash_{\mathrm{i}} T_2 \le T_3$, and $\vdash_{\mathrm{i}} T_3 \le \mathbb{G}$, respectively. Combining these three derivations through rule IIT-TRANS then yields $\vdash_{\mathrm{i}} T_1 \le \mathbb{G}$ as required.

5.1.3 Decidable Fragments

The undecidability proof of subtyping in IIT relies on two main ingredients:

Cyclic Interface Subtyping. Implementation definitions in IIT allow the introduction of cycles in the subtyping graph of interfaces. Consider one of the implementations defined by equation (5.1): it states that $\mathbb{S}\texttt{<}\eta_i \# X, \zeta_i \# Y\texttt{>}$ is a supertype of $\mathbb{S}\texttt{<}X, Y\texttt{>}$. In the reduction from PCP, such cycles are used to encode the individual steps in the search for a solution.

Multiple Instantiation Subtyping. Implementation definitions in IIT allow to introduce two different instantiations of the same interface as supertypes of some other interface. Consider again the implementations defined by equation (5.1): for $\eta_i \neq \eta_j$ or $\zeta_i \neq \zeta_j$, the implementations state that $\mathbb{S}\texttt{<}\eta_i \# X, \zeta_i \# Y\texttt{>} \neq \mathbb{S}\texttt{<}\eta_j \# X, \zeta_j \# Y\texttt{>}$ are both supertypes of $\mathbb{S}\texttt{<}X, Y\texttt{>}$. In the reduction from PCP, multiple instantiation subtyping encodes the choice between different pairs (η_i, ζ_i) and (η_j, ζ_j).

An obvious way to obtain decidable subtyping for IIT is to require that each type T has only finitely many supertypes.

Definition 5.4. The *set of T-supertypes*, written $\mathscr{S}_T$, denotes the set of all supertypes of T; that is, $\mathscr{S}_T := \{U \mid \vdash_{\mathrm{i}} T \le U\}$.

Restriction 5.5. The set $\mathscr{S}_T$ must be finite for all types T.

Theorem 5.6. *Under Restriction 5.5, subtyping in IIT is decidable.*

Figure 5.2 Algorithmic subtyping for IIT.

$\boxed{\mathscr{G} \vdash_{\mathrm{ia}} T \leq U}$

$$\textsc{iit-alg-refl}\quad \mathscr{G} \vdash_{\mathrm{ia}} T \leq T$$

$$\textsc{iit-alg-impl}\quad \frac{\begin{array}{cc} [\overline{V/X}]J<\overline{U}> \neq T & \textbf{implementation}<\overline{X}>\ I<\overline{T}>\ [J<\overline{U}>] \\ [\overline{V/X}]I<\overline{T}> \notin \mathscr{G} & \mathscr{G} \cup \{[\overline{V/X}]I<\overline{T}>\} \vdash_{\mathrm{ia}} [\overline{V/X}]I<\overline{T}> \leq T \end{array}}{\mathscr{G} \vdash_{\mathrm{ia}} [\overline{V/X}]J<\overline{U}> \leq T}$$

$\boxed{\vdash_{\mathrm{ia}} T \leq U}$

$$\textsc{iit-alg-sub}\quad \frac{\{T\} \vdash_{\mathrm{ia}} T \leq U}{\vdash_{\mathrm{ia}} T \leq U}$$

Proof. Figure 5.2 defines an algorithmic subtyping relation $\vdash_{\mathrm{ia}} T \leq U$ for IIT. The auxiliary relation $\mathscr{G} \vdash_{\mathrm{ia}} T \leq U$ uses a set of types $\mathscr{G}$ to prevent recursive invocations on a goal that was visited before. Section D.1.2 proves that $\vdash_{\mathrm{i}} T \leq U$ if, and only if, $\vdash_{\mathrm{ia}} T \leq U$. Moreover, it proves that the algorithm induced by the rules in Figure 5.2 terminates. □

Here is a restriction that eliminates cyclic interface subtyping.

Restriction 5.7. The underlying program must not contain a sequence $\mathit{def}_1, \ldots, \mathit{def}_n$ such that

$$(\forall i \in \{1, \ldots, n\})\ \mathit{def}_i = \textbf{implementation}<\overline{X_i}>\ J_i<\overline{U_i}>\ [I_i<\overline{T_i}>]$$

and $J_i = I_{i+1}$ for all $i = 1, \ldots, n-1$ and $J_n = I_1$.

Theorem 5.8. *Restriction 5.7 implies Restriction 5.5.*

Proof. See Section D.1.3. □

Remark. Restriction 5.5 does not imply Restriction 5.7. A program containing only one implementation, namely **implementation** $I\ [I]$, obviously meets Restriction 5.5 but violates Restriction 5.7.

The next restriction is strictly stronger than Restriction 5.7.

Restriction 5.9. For all implementation definitions

$$\mathit{def}_1 = \textbf{implementation}<\overline{X}>\ J_1<\overline{U}>\ [I_1<\overline{T}>]$$
$$\mathit{def}_2 = \textbf{implementation}<\overline{Y}>\ J_2<\overline{W}>\ [I_2<\overline{V}>]$$

of the underlying IIT program, it must hold that $J_1 \neq I_2$.

The full JavaGI language supports retroactive implementations with interfaces as implementing types under this restriction (see the “no implementation chains” criterion in Section 2.3.4). Section 6.1 explains this design decision and discusses decidability of subtyping in full JavaGI.

Theorem 5.10. *Under Restriction 5.9, subtyping in* IIT *is decidable.*

Proof. Obviously, Restriction 5.9 implies Restriction 5.7, so the claim follows with Theorem 5.8 and Theorem 5.6. □

The last restriction considered eliminates multiple instantiation subtyping.

Restriction 5.11. If $\vdash_{\mathrm{i}} I{<}\overline{T}{>} \leq J{<}\overline{U}{>}$ and $\vdash_{\mathrm{i}} I{<}\overline{T}{>} \leq J{<}\overline{V}{>}$ then $\overline{U} = \overline{V}$.

Theorem 5.12. *Restriction 5.11 implies Restriction 5.5.*

Proof. Assume that Restriction 5.11 holds but Restriction 5.5 does not. Thus, there exists a type $I{<}\overline{T}{>}$ such that $\mathscr{S}_{I{<}\overline{T}{>}}$ is infinite. Because types are formed from only finitely many interface names, there must exist an interface name J and infinitely many, pairwise disjoint sequences of types $\overline{U_1}, \overline{U_2}, \overline{U_3}, \ldots$ such that $J{<}\overline{U_i}{>} \in \mathscr{S}_{I{<}\overline{T}{>}}$ for all $i \in \mathbb{N}$. This contradicts Restriction 5.11. □

Remark. Neither Restriction 5.5 nor Restriction 5.7 implies Restriction 5.11: a program consisting of

$$\begin{gathered}\textbf{interface } I\\ \textbf{interface } J{<}X{>}\\ \textbf{implementation } J{<}A{>}\,[I]\\ \textbf{implementation } J{<}B{>}\,[I]\end{gathered}$$

meets both Restriction 5.5 and Restriction 5.7 but Restriction 5.11 does not hold. Moreover, Restriction 5.11 does not imply Restriction 5.7: a program consisting of

$$\begin{gathered}\textbf{interface } I\\ \textbf{interface } J\\ \textbf{implementation } I\,[J]\\ \textbf{implementation } J\,[I]\end{gathered}$$

meets Restriction 5.11 but violates Restriction 5.7.

5.2 Bounded Existential Types with Lower and Upper Bounds

A preliminary design of JavaGI [240] included bounded existential types [40] with lower and upper bounds. Additionally, bounded existential types (*existentials* for short) also supported implementation constraints. The main motivation for the inclusion of existentials was to subsume different features under a single concept. In the following discussion, the notation $\exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,T$ denotes a bounded existential type with quantified type variables $\overline{X}$, bounds $\overline{P}$, and body type T. A bound is either a lower bound $X\,\textbf{super}\,T$, an upper bound $X\,\textbf{extends}\,T$, or an implementation constraint $\overline{U}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$, where $\overline{U}$ are types and $I\texttt{<}\overline{V}\texttt{>}$ is an interface I with type arguments $\overline{V}$.

Existentials of this fashion subsume the following features:

- They properly generalize interface types. After all, an interface type $I\texttt{<}\overline{T}\texttt{>}$ simply represents an unknown type implementing interface $I\texttt{<}\overline{T}\texttt{>}$. Thus, $I\texttt{<}\overline{T}\texttt{>}$ is equivalent to $\exists X\,\textbf{where}\,X\,\textbf{implements}\,I\texttt{<}\overline{T}\texttt{>}\,.\,X$.

- They allow the general composition of interface types. For example, the type $\exists X\,\textbf{where}\,X\,\textbf{implements}\,I\texttt{<}\overline{T}\texttt{>}, X\,\textbf{implements}\,J\texttt{<}\overline{U}\texttt{>}\,.\,X$ denotes the intersection of types that implements both interface $I\texttt{<}\overline{T}\texttt{>}$ and $J\texttt{<}\overline{U}\texttt{>}$.[2]

- They allow meaningful types in the presence of multi-headed interfaces. Consider the observer pattern example from Section 2.1.7, which introduced a two-headed interface `ObserverPattern` and an implementation of `ObserverPattern` for classes `ExprPool` and `ResultDisplay`. In this context, the type $\exists X\,\textbf{where}\,\texttt{ExprPool} * X\,\textbf{implements}\,\texttt{ObserverPattern}\,.\,X$ comprises all objects that act as an observer for class `ExprPool`.

- They encompass Java wildcards [229, 37]. For example, consider the wildcard type `List<? extends Number>`, which stands for a list with elements of some subtype of `Number`. Its existential encoding is $\exists X\,\textbf{where}\,X\,\textbf{extends}\,\texttt{Number}\,.\,\texttt{List<}X\texttt{>}$. Java also supports wildcards with lower bounds as in `Comparator<? super String>`, which denotes a comparator for some unknown supertype of `String`. The existential encoding of this wildcard type is $\exists X\,\textbf{where}\,X\,\textbf{super}\,\texttt{String}\,.\,\texttt{Comparator<}X\texttt{>}$.

This section investigates decidability of subtyping for bounded existential types with lower and upper bounds. It ignores implementation constraints for existentials because lower and upper bounds are enough to render subtyping undecidable. Starting point of the investigation is the calculus EXuplo to be defined in Section 5.2.1. Next, Section 5.2.2 proves undecidability of subtyping in EXuplo by reduction from subtyping in $F^D_\leq$ [175], a restricted form of the polymorphic λ-calculus extended with subtyping [40]. Finally, Section 5.2.3 present two decidable fragments of EXuplo.

The results in this section are not only relevant to JavaGI's full type system. First, it may shed new light on the question whether or not subtyping for Java wildcards is decidable. Second, the programming language Scala [166] also supports bounded existential types with lower and upper bounds. The subtyping rules for Scala's existentials [166, Sections 3.2.10 and 3.5.2] are similar to that in EXuplo, so it is likely that subtyping in Scala is also undecidable. Section 8.10 discusses these matters in more detail.

5.2.1 The Calculus EXuplo

The calculus EXuplo supports bounded existential types with lower and upper bounds. Other researchers [228, 38, 37] use formal systems similar to EXuplo for modeling Java

[2] Java 1.5 can denote such types only in the bound of generic type variables.

Figure 5.3 Syntax, constraint entailment, and subtyping for EXuplo.

$\boxed{\text{Syntax}}$

$$\begin{array}{rcl} N, M &::=& C\texttt{<}\overline{X}\texttt{>} \mid \mathit{Object} \\ T,U,V,W &::=& X \mid N \mid \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N \\ P,Q &::=& X\,\mathbf{extends}\,T \mid X\,\mathbf{super}\,T \end{array}$$

$$X,Y,Z \in \mathit{TvarName}_{\mathsf{EXuplo}} \qquad C,D \in \mathit{ClassName}_{\mathsf{EXuplo}}$$

$\boxed{\Delta \Vdash_{\mathrm{ex}} T\,\mathbf{extends}\,U \qquad \Delta \Vdash_{\mathrm{ex}} T\,\mathbf{super}\,U}$

EXUPLO-EXTENDS
$$\frac{\Delta \vdash_{\mathrm{ex}} T \leq U}{\Delta \Vdash_{\mathrm{ex}} T\,\mathbf{extends}\,U}$$

EXUPLO-SUPER
$$\frac{\Delta \vdash_{\mathrm{ex}} U \leq T}{\Delta \Vdash_{\mathrm{ex}} T\,\mathbf{super}\,U}$$

$\boxed{\Delta \vdash_{\mathrm{ex}} T \leq U}$

EXUPLO-REFL
$$\Delta \vdash_{\mathrm{ex}} T \leq T$$

EXUPLO-TRANS
$$\frac{\Delta \vdash_{\mathrm{ex}} T \leq U \qquad \Delta \vdash_{\mathrm{ex}} U \leq V}{\Delta \vdash_{\mathrm{ex}} T \leq V}$$

EXUPLO-OBJECT
$$\Delta \vdash_{\mathrm{ex}} T \leq \mathit{Object}$$

EXUPLO-EXTENDS
$$\frac{X\,\mathbf{extends}\,T \in \Delta}{\Delta \vdash_{\mathrm{ex}} X \leq T}$$

EXUPLO-SUPER
$$\frac{X\,\mathbf{super}\,T \in \Delta}{\Delta \vdash_{\mathrm{ex}} T \leq X}$$

EXUPLO-OPEN
$$\frac{\Delta, \overline{P} \vdash_{\mathrm{ex}} N \leq T \qquad \overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset}{\Delta \vdash_{\mathrm{ex}} \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N \leq T}$$

EXUPLO-ABSTRACT
$$\frac{T = [\overline{U/X}]N \qquad (\forall i)\ \Delta \Vdash_{\mathrm{ex}} [\overline{U/X}]P_i}{\Delta \vdash_{\mathrm{ex}} T \leq \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N}$$

wildcards. It is not the intention of EXuplo to provide another formalization of wildcards, but rather to expose the essential ingredients that make subtyping undecidable in a calculus as simple as possible.

Figure 5.3 defines the syntax and the constraint entailment and subtyping relations of EXuplo. As usual, overbar notation denotes sequencing (see Definition 3.1). A class type N is either *Object* or an instantiated generic class $C\texttt{<}\overline{X}\texttt{>}$, where the type arguments must be type variables. A type T is a type variable, a class type, or an existential. In EXuplo, the body of an existential must be a class type. Existentials that differ only in the names of bound type variables are considered equal. A constraint P places either an upper bound ($X\,\mathbf{extends}\,T$) or a lower bound ($X\,\mathbf{super}\,T$) on a type variable X. Type environments Δ are finite set of constraints P with Δ, P standing for $\Delta \cup \{P\}$.

Class definitions and inheritance are omitted from EXuplo. The only assumption is that every class name C comes with a fixed arity that is respected when applying C to type arguments. There are some further (implicit) restrictions:

Restriction 5.13. An existential must abstract over at least one type variable and all its bounded type variables must appear in the body type. That is, if $T = \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N$ then $\overline{X} \neq \bullet$ and $\overline{X} \subseteq \mathsf{ftv}(N)$.

Restriction 5.14. An existential may only constrain bounded type variables. That is, if $T = \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N$ and $P \in \overline{P}$, then $P = Y\,\mathbf{extends}\,T$ or $P = Y\,\mathbf{super}\,T$ with $Y \in \overline{X}$.

Restriction 5.15. A type variable must not have both upper and lower bounds.[3]

Constraint entailment $\Delta \Vdash_{\text{ex}} P$ establishes validity of constraint P under type environment Δ. The subtyping relation $\Delta \vdash_{\text{ex}} T \leq U$ states that T is a subtype of U under type environment Δ. It is reflexive and transitive as usual, has *Object* as a supertype of all other types, and incorporates lower and upper bounds of type variables via rules EXUPLO-SUPER and EXUPLO-EXTENDS, respectively. Rule EXUPLO-OPEN opens an existential on the left-hand side of the subtyping relation by moving its constraints into the type environment. The premise $\overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset$ ensures that the existentially quantified type variables are sufficiently fresh and do not escape their scope. Rule EXUPLO-ABSTRACT deals with existentials on the right-hand side of the subtyping relation. It states that a type is a subtype of some existential if the constraints of the existential hold under an appropriate substitution. As before, $[\overline{T/X}]$ denotes the capture-avoiding type substitution replacing each X_i with T_i.

5.2.2 Undecidability of Subtyping in EXuplo

To get a feeling how subtyping derivations in EXuplo may lead to infinite regress, assume that $\mathbb{D}$ and $\mathbb{D}'$ are two unary classes and consider the goal

$$\Delta \vdash_{\text{ex}} X \leq \neg\,\mathbb{D}'{<}X{>}$$

where $\Delta := \{X\,\mathbf{extends}\,\neg\,U\}$, $U := \exists X\,\mathbf{where}\,X\,\mathbf{extends}\,\neg\,\mathbb{D}'{<}X{>}\,.\,\mathbb{D}'{<}X{>}$ and, for any type T, the notation $\neg\,T$ abbreviates $\exists X\,\mathbf{where}\,X\,\mathbf{super}\,T\,.\,\mathbb{D}{<}X{>}$ for some fresh X. Searching for a derivation of this goal quickly leads to a subgoal of the form $\Delta' \vdash_{\text{ex}} X \leq \neg\,\mathbb{D}'{<}X{>}$ such that $\Delta' := \Delta, Z\,\mathbf{super}\,U$ where Z is a fresh type variable introduced by rule EXUPLO-OPEN:

$$\dfrac{\textsc{exuplo-extends}\;\dfrac{X\,\mathbf{extends}\,\neg\,U \in \Delta}{\Delta \vdash_{\text{ex}} X \leq \neg\,U} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\textsc{exuplo-abstract}\;\dfrac{\textsc{exuplo-extends}\;\dfrac{\dfrac{\vdots}{\Delta' \vdash_{\text{ex}} X \leq \neg\,\mathbb{D}'{<}X{>}}}{\Delta' \Vdash_{\text{ex}} X\,\mathbf{extends}\,\neg\,\mathbb{D}'{<}X{>}}}{\Delta' \vdash_{\text{ex}} \mathbb{D}'{<}X{>} \leq U} \quad \dfrac{Z\,\mathbf{super}\,U \in \Delta'}{\Delta' \vdash_{\text{ex}} U \leq Z}\;\textsc{exuplo-super}}{\Delta' \vdash_{\text{ex}} \mathbb{D}'{<}X{>} \leq Z}\;\textsc{exuplo-trans}}{\Delta' \Vdash_{\text{ex}} Z\,\mathbf{super}\,\mathbb{D}'{<}X{>}}\;\textsc{exuplo-super}}{\Delta' \vdash_{\text{ex}} \mathbb{D}{<}Z{>} \leq \neg\,\mathbb{D}'{<}X{>}}\;\textsc{exuplo-abstract}}{\Delta \vdash_{\text{ex}} \neg\,U \leq \neg\,\mathbb{D}'{<}X{>}}\;\textsc{exuplo-open}}{\Delta \vdash_{\text{ex}} X \leq \neg\,\mathbb{D}'{<}X{>}}\;\textsc{exuplo-trans}$$

[3] Modeling Java wildcards requires upper and lower bounds for the same type variable in certain situations.

Figure 5.4 Syntax and subtyping for $F^D_{\leq}$.

$\boxed{\text{Syntax}}$

$$\begin{aligned}\tau^+ &::= \mathsf{Top} \mid \forall\alpha_0{\leq}\tau_0^- \ldots \alpha_n{\leq}\tau_n^- \,.\, \neg\,\tau^- \\ \tau^- &::= \alpha \mid \forall\alpha_0 \ldots \alpha_n \,.\, \neg\,\tau^+ \\ \Omega^- &::= \emptyset \mid \Omega^-, \alpha{\leq}\tau^-\end{aligned}$$

$$\alpha, \gamma \in \mathit{TvarName}_D$$

$\boxed{\Omega^- \vdash_D \sigma^- \leq \tau^+}$

$$\textsc{d-top}\quad \Omega \vdash_D \tau \leq \mathsf{Top} \qquad\qquad \textsc{d-var}\quad \frac{\tau \neq \mathsf{Top} \qquad \Omega \vdash_D \Omega(\alpha) \leq \tau}{\Omega \vdash_D \alpha \leq \tau}$$

$$\textsc{d-all-neg}\quad \frac{\Omega, \alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n \vdash_D \tau \leq \sigma}{\Omega \vdash_D \forall\alpha_0 \ldots \alpha_n \,.\, \neg\,\sigma \leq \forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n \,.\, \neg\,\tau}$$

The formal undecidability proof of subtyping in EXuplo is by reduction from $F^D_{\leq}$ [175], a restricted version of $F_{\leq}$ [40]. Pierce defines $F^D_{\leq}$ for his undecidability proof of $F_{\leq}$ subtyping [175]. Figure 5.4 recapitulates the syntax and the subtyping relation of $F^D_{\leq}$. Let n be a fixed natural number. A type τ is either an n-positive type, τ^+, or an n-negative type, τ^-, where n stands for the number of type variables (minus one) bound at the top level of the type. (The symbol "$\neg$" used in the syntax of types is not an abbreviation as before but merely serves as a syntactic marker.) An n-negative type environment Ω^- associates type variables α with upper bounds τ^-. The polarity ($+$ or $-$) characterizes at which positions of a subtyping judgment a type or type environment may appear. For readability, the polarity is often omitted and n is left implicit.

An *n-ary subtyping judgment* in $F^D_{\leq}$ has the form $\Omega^- \vdash_D \sigma^- \leq \tau^+$, where Ω^- is an n-negative type environment, σ^- is an n-negative type, and τ^+ is an n-positive type. Only n-negative types appear to the left and only n-positive types appear to the right of the $\leq$ symbol. The subtyping rule $\textsc{d-all-neg}$ compares two quantified types $\sigma = \forall\alpha_0 \ldots \alpha_n \,.\, \neg\,\sigma'$ and $\tau = \forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\alpha_n \,.\, \neg\,\tau'$ by swapping the left- and right-hand sides of the subtyping judgment and checking $\tau' \leq \sigma'$ under the extended environment $\Omega, \alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n$. The rule is correct with respect to $F_{\leq}$ because we may interpret every $F^D_{\leq}$ type as an $F_{\leq}$ type:

$$\begin{aligned}\forall\alpha_0 \ldots \alpha_n \,.\, \neg\,\sigma' &\equiv \forall\alpha_0{\leq}\mathsf{Top} \ldots \forall\alpha_n{\leq}\mathsf{Top}.\forall\gamma{\leq}\sigma' \,.\, \gamma && (\gamma \text{ fresh}) \\ \forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\alpha_n \,.\, \neg\,\tau' &\equiv \forall\alpha_0{\leq}\tau_0 \ldots \forall\alpha_n{\leq}\tau_n.\forall\gamma{\leq}\tau' \,.\, \gamma && (\gamma \text{ fresh})\end{aligned}$$

Using these abbreviations, every $F^D_{\leq}$ subtyping judgment can be read as an $F_{\leq}$ subtyping judgment. The subtyping relations in $F^D_{\leq}$ and $F_{\leq}$ coincide for judgments in their common domain [175].

Figure 5.5 Reduction from $F_{\leq}^{D}$ to EXuplo.

$$\begin{aligned}
\llbracket \mathsf{Top} \rrbracket^{+} &= \mathit{Object} \\
\llbracket \forall \alpha_0 {\leq} \tau_0^{-} \ldots \alpha_n {\leq} \tau_n^{-} \,.\, \neg\, \tau^{-} \rrbracket^{+} &= \neg\, \exists Y, \overline{X^{\alpha_i}}\, \textbf{where}\, X^{\alpha_0}\, \textbf{extends}\, \llbracket \tau_0 \rrbracket^{-} \ldots \\
&\qquad\qquad X^{\alpha_n}\, \textbf{extends}\, \llbracket \tau_n \rrbracket^{-}, Y\, \textbf{extends}\, \llbracket \tau \rrbracket^{-} \\
&\qquad .\, \mathbb{C}\texttt{<}Y, \overline{X^{\alpha_i}}\texttt{>} \\
\llbracket \alpha \rrbracket^{-} &= X^{\alpha} \\
\llbracket \forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \tau^{+} \rrbracket^{-} &= \neg\, \exists Y, \overline{X^{\alpha_i}}\, \textbf{where}\, Y\, \textbf{extends}\, \llbracket \tau \rrbracket^{+} \,.\, \mathbb{C}\texttt{<}Y, \overline{X^{\alpha_i}}\texttt{>} \\
\llbracket \emptyset \rrbracket^{-} &= \emptyset \\
\llbracket \Omega, \alpha {\leq} \tau^{-} \rrbracket^{-} &= \llbracket \Omega \rrbracket^{-}, X^{\alpha}\, \textbf{extends}\, \llbracket \tau \rrbracket^{-} \\
\llbracket \Omega^{-} \vdash_D \tau^{-} \leq \sigma^{+} \rrbracket &= \llbracket \Omega \rrbracket^{-} \vdash_{\mathrm{ex}} \llbracket \tau \rrbracket^{-} \leq \llbracket \sigma \rrbracket^{+}
\end{aligned}$$

$$\neg\, T \equiv \exists X\, \textbf{where}\, X\, \textbf{super}\, T \,.\, \mathbb{D}\texttt{<}X\texttt{>} \qquad (X \text{ sufficiently fresh})$$

It is sufficient to consider only *closed judgments*. Define the *domain* of a $F_{\leq}^{D}$ type environment as $\mathsf{dom}(\alpha_1{\leq}\tau_1, \ldots, \alpha_n{\leq}\tau_n) := \{\alpha_1, \ldots, \alpha_n\}$. A type τ is *closed* under Ω if $\mathsf{ftv}(\tau) \subseteq \mathsf{dom}(\Omega)$ and, if $\tau = \forall \alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n \,.\, \neg\, \sigma$, then no α_i appears free in any τ_j. A type environment Ω is closed if $\Omega = \emptyset$ or $\Omega = \Omega', \alpha{\leq}\tau$ with Ω' closed and τ closed under Ω'. A judgment $\Omega \vdash_D \tau \leq \sigma$ is closed if Ω is closed and τ, σ are closed under Ω.

Fact 5.16. Subtyping in $F_{\leq}^{D}$ is undecidable [175].

We now state the central theorem of this section and sketch its proof.

Theorem 5.17. *Subtyping in EXuplo is undecidable.*

Proof. The proof is by reduction from $F_{\leq}^{D}$. Figure 5.5 defines a translation from $F_{\leq}^{D}$ types, type environments, and subtyping judgments to their corresponding EXuplo forms. The translation of an n-ary subtyping judgment assumes the existence of two EXuplo classes: $\mathbb{C}$ accepts $n{+}2$ type arguments, and $\mathbb{D}$ takes one type argument. The superscripts in $\llbracket\cdot\rrbracket^{+}$ and $\llbracket\cdot\rrbracket^{-}$ indicate whether the translation acts on positive or negative entities.

As in the example at the beginning of this subsection, a negated type, written $\neg\, T$, is an abbreviation for an existential with a single **super** constraint (see Figure 5.5). The **super** constraint simulates the behavior of the $F_{\leq}^{D}$ subtyping rule D-ALL-NEG, which swaps the left- and right-hand sides of subtyping judgments.

An n-positive type $\forall \alpha_0{\leq}\tau_0^{-} \ldots \alpha_n{\leq}\tau_n^{-} \,.\, \neg\, \tau^{-}$ is translated into a negated existential. The existentially quantified type variables $\overline{X^{\alpha_i}}$ (abbreviating $X^{\alpha_0}, \ldots, X^{\alpha_n}$) correspond to the universally quantified type variables $\alpha_0, \ldots, \alpha_n$. The bound $\llbracket \tau \rrbracket^{-}$ of the fresh type variable Y represents the body $\neg\, \tau^{-}$ of the original type. It is not possible to use $\llbracket \tau \rrbracket^{-}$ directly as the body because existentials in EXuplo have only class types as their bodies. The translation for n-negative types is similar to the one for n-positive types. It is easy to see that the EXuplo types in the image of the translation meet Restrictions 5.13, 5.14, and 5.15. Type environments and subtyping judgments are translated in the obvious way.

Figure 5.6 Subtyping for EXuplo without transitivity rule.

$$\boxed{\Delta \Vdash_{\text{ex}}{}' \; T\,\mathbf{extends}\,U \qquad \Delta \Vdash_{\text{ex}}{}' \; T\,\mathbf{super}\,U}$$

$$\textsc{exuplo-extends'}\;\frac{\Delta \vdash_{\text{ex}}{}' \; T \le U}{\Delta \Vdash_{\text{ex}}{}' \; T\,\mathbf{extends}\,U} \qquad \textsc{exuplo-super'}\;\frac{\Delta \vdash_{\text{ex}}{}' \; U \le T}{\Delta \Vdash_{\text{ex}}{}' \; T\,\mathbf{super}\,U}$$

$$\boxed{\Delta \vdash_{\text{ex}}{}' \; T \le U}$$

$$\textsc{exuplo-refl'}\;\frac{T = X \text{ or } T = N}{\Delta \vdash_{\text{ex}}{}' \; T \le T} \qquad \textsc{exuplo-object'}\;\Delta \vdash_{\text{ex}}{}' \; T \le \mathit{Object} \qquad \textsc{exuplo-extends'}\;\frac{X\,\mathbf{extends}\,T' \in \Delta \qquad \Delta \vdash_{\text{ex}}{}' \; T' \le T}{\Delta \vdash_{\text{ex}}{}' \; X \le T}$$

$$\textsc{exuplo-super'}\;\frac{X\,\mathbf{super}\,T' \in \Delta \qquad \Delta \vdash_{\text{ex}}{}' \; T \le T'}{\Delta \vdash_{\text{ex}}{}' \; T \le X} \qquad \textsc{exuplo-open'}\;\frac{\Delta, \overline{P} \vdash_{\text{ex}}{}' \; N \le T \qquad \overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset}{\Delta \vdash_{\text{ex}}{}' \; \exists \overline{X}\,\mathbf{where}\,\overline{P}\,.\,N \le T}$$

$$\textsc{exuplo-abstract'}\;\frac{N = [\overline{Y/X}]M \qquad (\forall i)\; \Delta \Vdash_{\text{ex}}{}' \; [\overline{Y/X}]P_i}{\Delta \vdash_{\text{ex}}{}' \; N \le \exists \overline{X}\,\mathbf{where}\,\overline{P}\,.\,M}$$

It remains to verify that $\Omega \vdash_D \tau \le \sigma$ is derivable in $F_{\le}^{D}$ if, and only if, $[\![\Omega \vdash_D \tau \le \sigma]\!]$ is derivable in EXuplo. The “$\Rightarrow$” direction is an easy induction on the derivation of $\Omega \vdash_D \tau \le \sigma$. The “$\Leftarrow$” direction requires more work because the transitivity rule EXUPLO-TRANS (Figure 5.3) involves an intermediate type which is not necessarily in the image of the translation. Hence, a direct proof by induction on the derivation of $[\![\Omega \vdash_D \tau \le \sigma]\!]$ fails. To solve this problem, Figure 5.6 defines an alternative subtyping relation $\Delta \vdash_{\text{ex}}{}' \; T \le U$ for EXuplo that does not have a built-in transitivity rule. It is then possible to prove that $\Delta \vdash_{\text{ex}}{}' \; T \le U$ if, and only if, $\Delta \vdash_{\text{ex}} T \le U$ and that $[\![\Omega]\!]^{-} \vdash_{\text{ex}}{}' \; [\![\tau]\!]^{-} \le [\![\sigma]\!]^{+}$ implies $\Omega \vdash_D \tau \le \sigma$. Section D.2.1 provides all the details and the full proofs. □

5.2.3 Decidable Fragments

This section presents two decidable fragments of EXuplo. Definition 3.10 on page 57 already introduced the notion of contractive type environments in the context of CoreGI. The following definition restates the definition for EXuplo:

Definition 5.18 (Contractive type environments for EXuplo). A type environment Δ is *contractive* if, and only if, there exist no type variables $X_1, \ldots, X_n$ such that $X_1 = X_n$ and either $X_i\,\mathbf{extends}\,X_{i+1} \in \Delta$ for all $i \in \{1, \ldots, n-1\}$ or $X_i\,\mathbf{super}\,X_{i+1} \in \Delta$ for all $i \in \{1, \ldots, n-1\}$.

Theorem 5.19. *If all type environments involved are contractive and support for lower bounds is dropped, then subtyping in EXuplo becomes decidable.*

Proof. The relation $\Delta \vdash_{\text{ex}}{}' T \leq U$ defined in Figure 5.6 is equivalent to EXuplo's subtyping relation. Moreover, the algorithm induced by the rules in Figure 5.6 terminates. See Section D.2.2 for details. □

Definition 5.20. A bounded existential type $\exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N$ is *variable-bounded* if all constraints in $\overline{P}$ have only type variables as their bounds; that is, for all $P \in \overline{P}$ either $P = Y\,\mathbf{extends}\,Z$ or $P = Y\,\mathbf{super}\,Z$.

Theorem 5.21. *If all type environments involved are contractive, support for upper bounds is dropped, and all existentials are variable-bounded, then subtyping in EXuplo becomes decidable.*

Proof. Similar to the proof of Theorem 5.19, see Section D.2.3. □

6 Implementation

A new programming language with a convincing design and a rigorous formalization is not very useful without a proper implementation in form of a compiler or interpreter. The current chapter addresses this problem and presents the implementation of a compiler and a run-time system for JavaGI.

The JavaGI compiler is an extension of the Eclipse Compiler for Java [62] and generates byte code that runs on a standard Java Virtual Machine (JVM [125]). It supports the full Java 1.5 language and all JavaGI-specific features presented in this dissertation. The run-time system assists the compiler by maintaining the pool of available retroactive implementations, by checking the well-formedness criteria defined in Section 2.3.4, and by providing certain run-time operations.

Besides the compiler and the run-time system, there also exists a JavaGI plugin for Eclipse [60], a widely used integrated development environment (IDE). The homepage of the JavaGI project [239] makes the source code of the compiler, the run-time system, and the Eclipse plugin available under the terms of the Eclipse Public License [61].

Chapter Outline. The chapter contains four sections.

- Section 6.1 sketches how to extend CoreGI to the full JavaGI language.
- Section 6.2 shows how to translate JavaGI constructs to Java byte code.
- Section 6.3 discusses JavaGI's run-time system.
- Section 6.4 describes the JavaGI plugin for Eclipse.

6.1 Extending CoreGI to JavaGI

The CoreGI calculus from Chapter 3 lacks several features present in the full JavaGI language. Section 2.3.3 already sketched how to typecheck method invocations without CoreGI's restrictions that namespaces for class and interface methods are disjoint and that names of interface methods are globally unique. Other features missing in CoreGI include imperative features, visibility modifiers, type erasure [26], wildcards [229], inference of type arguments for method invocations [82, § 15.12.2.7][204], and interfaces as implementing types. The following discussion explains how the compiler for the full language handles these features. Other features of JavaGI not included in CoreGI are straightforward to implement.

6.1.1 Imperative Features

JavaGI does not introduce any new imperative features (with respect to Java) and most of Java's imperative features are orthogonal to the JavaGI-specific extensions. Thus, we conjecture that type soundness of CoreGI also holds in a setting with Java's imperative features. A minor problem arises when JavaGI's dynamic-dispatch algorithm for method invocations encounters **`null`** as one of the dispatch arguments. The implementation handles this case by throwing a `NullPointerException`, analogously to the case in Java where **`null`** appears as the receiver of a method invocation.

6.1.2 Visibility Modifiers

JavaGI fully respects Java's encapsulation properties. Inside retroactive implementations, regular Java visibility rules apply; for example, private fields and methods of the implementing types are not accessible. JavaGI regards all implementations as **`public`**.

6.1.3 Type Erasure

CoreGI's dynamic semantics is a type-passing semantics; that is, type arguments are available at run time. In contrast, Java and the full JavaGI language perform type erasure during compilation, so type arguments are not available at run time.

The definition of CoreGI carefully avoids relying too much on run-time type arguments. For example, well-formedness criterion WF-IFACE-3 prevents implementing types from appearing nested inside argument types of method signatures and criterion WF-PROG-4 requires constraints of implementation definitions to be consistent with respect to subtyping among implementing types. Both criteria ensure that dynamic dispatch does not require run-time type arguments.

At other places, the definition of CoreGI requires minor adjustments to work under a type erasure semantics. For example, CoreGI's well-formedness criterion WF-PROG-1, which prevents overlapping implementation definitions, needs to be adapted for the full language (see Section 2.3.4, criterion "no overlap").

6.1.4 Wildcards

Proving type soundness for Java wildcards [229] is known to be a tricky business [37]. Nevertheless, we believe that type soundness holds for the full JavaGI language including wildcards because JavaGI generalizes CoreGI's well-formedness criteria WF-IFACE-2 and WF-IFACE-3 to prevent implementing type variables such as **`This`** from appearing nested inside generic types at all. Thus, implementing type variables, which behave covariantly, never form upper or lower bounds of wildcards, the latter of which behave contravariantly.

Wildcards do not only challenge type soundness but also decidability of subtyping. In general, it is still an open question whether subtyping for Java wildcards is decidable (see Section 8.10). However, the inclusion of wildcards in JavaGI is a concession to ensure backwards compatibility with Java 1.5. An embedding of generalized interfaces in other languages such as C# could easily drop support for wildcards. Thus, the decidability question for wildcards is not intrinsic to the decidability of subtyping in JavaGI.

6.1.5 Inference of Type Arguments

The JavaGI compiler supports inference of type arguments for method invocations by simply reusing Java's inference algorithm [82, § 15.12.2.7]. Consequently, JavaGI-specific constraints in method signatures do not contribute to the improvement of type arguments. In general, this is not a problem because Java's inference algorithm is incomplete anyway [204]. If inference fails, then the programmer may still invoke the method in question by specifying the type arguments explicitly.

JavaGI-specific features run no risk of introducing soundness-holes into the type inference process because the JavaGI compiler verifies correctness of inference during typechecking. This verification step is also needed for plain Java because Java's inference algorithm is unsound [204].

6.1.6 Interfaces as Implementing Types

CoreGI supports only classes as implementing types of retroactive interface implementations. The full language, however, also supports interfaces as implementing types (see for example the implementation of `EQ` for interface `List<X>` in Section 2.1.3). Section 5.1 proved that interfaces as implementing types renders subtyping—and hence typechecking—undecidable. That section also defined four different restrictions that still allow interfaces as implementing types but keep subtyping decidable.

The full JavaGI language supports interfaces as implementing types under one of these restrictions (Restriction 5.9 in Section 5.1, mentioned as well-formedness criterion "no implementation chains" in Section 2.3.4). It prefers Restriction 5.9 over Restriction 5.7 because the former is easier to check and simplifies the detection of ambiguities arising through conflicting implementation definitions. Further, Restriction 5.9 gives raise to an efficient implementation of dynamic method lookup because it allows the use of Java's subtype check instead of JavaGI's when searching for an implementation definition matching certain run-time types. It is unclear how to check the two other restrictions from Section 5.1 (Restriction 5.5 and Restriction 5.11) in practice.

6.2 Translating JavaGI to Java Byte Code

The compilation scheme employed by the JavaGI compiler is based on the formal translation from CoreGI$^\flat$ to iFJ defined in Chapter 4. It never modifies existing source or byte code, so existing clients are not affected and retroactive interface implementations can

Figure 6.1 Translation of interface EQ and class Lists from Section 2.1.2.

```
// Java 1.4
import javagi.runtime.RT;
import javagi.runtime.Wrapper;
import java.util.*;
// Translation of the EQ interface
interface EQ { boolean eq(Object that); }
public interface EQ$Dict {
  public static final int[] eq$DispatchVector = new int[]{0,0,0,1};
  public boolean eq(Object this$, Object that);
}
public class EQ$Wrapper extends Wrapper implements EQ {
  public static boolean eq$Dispatcher(Object this$, Object that) {
    Object dict = RT.getDict(EQ$Dict.class, EQ$Dict.eq$DispatchVector,
                             new Object[]{this$, that});
    return ((EQ$Dict) dict).eq(this$, that);
  }
  public EQ$Wrapper(Object obj) {
    super(obj);  // The superclass constructors stores obj in field this.wrapped
  }
  public boolean eq(Object that) {
    // JavaGI compiler guarantees that this method is never called
    throw new Error("Binary method invoked on wrapper object");
  }
  // Superclass delegates hashCode, equals, and toString to this.wrapped
}
// Translation of class Lists
class Lists {
  static Object find(Object x, List list) {
    Iterator iter = list.iterator();
    while (iter.hasNext()) {
      Object y = iter.next();
      if (EQ$Wrapper.eq$Dispatcher(x, y)) return y;
    }
    return null;
  }
}
```

be defined for arbitrary classes and interfaces, even if they are part of Java's standard library. Nevertheless, the compilation scheme allows for in-place object adaption; that is, new operations are available even for existing objects.

To demonstrate how the compilation scheme works, Figure 6.1 contains the translation of the interface EQ and the class Lists from Section 2.1.2. Moreover Figure 6.2 contains the translation of the retroactive implementations defined in Sections 2.1.2 and 2.1.3. For readability, the figures show Java 1.4 source code instead of the byte code generated by the JavaGI compiler.

6.2.1 Translating Interfaces

The `JavaGI` compiler generates for each interface `J` a *dictionary interface* `J$Dict`. For single-headed interfaces, it also generates a *wrapper class* `J$Wrapper` and a Java 1.4 interface `J` using Java's erasure translation [96, 82]. For example, the type variable **`This`** of interface `EQ` becomes `Object` in the code in Figure 6.1.

The dictionary interface contains the same methods as the original interface but makes the receiver of all non-static methods explicit by introducing a fresh argument of type `Object` (the `this$` argument of `eq` in `EQ$Dict`). Furthermore, the dictionary interface contains a *dispatch vector* of name `m$DispatchVector` for each non-static method `m` of the original interface. The dispatch vector connects the interface's implementing types with the method's receiver and argument types. `JavaGI`'s run-time system relies on the dispatch vector to perform multiple dispatch. For an n-headed interface, the dispatch vector is an `int` array of length $2n$ where, for $i \in \{0, \ldots, n-1\}$, the value at index $2i$ denotes the zero-based position of the implementing type corresponding to the receiver or argument whose position is stored at index $2i + 1$.[1] (Positions of argument types start at one, the receiver type has position zero.) For example, the receiver and the first argument of `eq` both refer to the implementing type **`This`** of `EQ`, so the dispatch vector in `EQ$Dict` is `{0,0,0,1}`.

The wrapper class serves as an adapter when a class is used at an interface type that it implements only retroactively. Most aspects of wrapper classes are standard (see Section 4.3 and the work by Baumgartner and coworkers [10]), but there are some `JavaGI`-specific issues. First, the `eq` method of `EQ$Wrapper` always throws an exception because `JavaGI`'s type system ensures that such a binary method is never called on a wrapper object. (Section 2.3.1 explains why such a call would be unsound.)

Second, the wrapper class provides a static *dispatcher method* `m$Dispatcher` for every method `m` of the original interface. These dispatcher methods simplify the translation of retroactive method invocations. The dispatcher method for `eq` (named `eq$Dispatcher`) calls `getDict` from class `javagi.runtime.RT`,[2] passing the class object for `EQ`'s dictionary, the dispatch vector for `eq`, and an array containing the actual arguments. Based on this information, the run-time system returns a dictionary object corresponding to some retroactive implementation of `EQ`, through which the dispatcher invokes the `eq` method. For a non-binary method, the dispatcher would first try to invoke the method directly on `this$`, provided `this$` implemented the method's declaring interface non-retroactively.

6.2.2 Translating Invocations of Retroactively Implemented Methods

The translation of an invocation of a retroactively implemented method just invokes the corresponding dispatcher method of the wrapper class of the method's defining interface. For example, to compare two expressions for equality, the `find` method of class `Lists` calls `eq$Dispatcher` defined in `EQ$Wrapper` (see Figure 6.1).

[1] It would be more natural to encode the dispatch vector as an n-element array of pairs of `int`s. However, Java does not support a primitive type for pairs, so we choose the alternative, flat representation.

[2] The `getDict` method is the analogon to iFJ's **getdict** primitive.

6.2.3 Translating Retroactive Interface Implementations

Figure 6.2 presents the translation of the retroactive implementation definitions from Sections 2.1.2 and 2.1.3, again displaying Java 1.4 source code instead of byte code. The translation of a retroactive implementation definition results in a *dictionary class* that implements the dictionary interface corresponding to the implementation's interface. For example, the dictionary class `EQ$Dict$IntLit` corresponds to the implementation `EQ [IntLit]` and implements the dictionary interface `EQ$Expr`.

To implement the methods of the dictionary interface, the methods of the original implementation need to be adapted: they have an extra parameter `this$` to make the receiver explicit and the types of those arguments declared as implementing types are lifted to match the corresponding argument types in the dictionary interface. For example, the argument `that` of the `eq` method in the implementation `EQ [IntLit]` has type `IntLit`, but the corresponding argument in the original `EQ` interface is declared with implementing type **`This`**. Hence, the JavaGI compiler lifts the type of `that` to `Object`, as required by the `eq` method of the `EQ$Dict` interface.

To recover from this loss of type information, the compiler performs appropriate downcasts on these arguments. For example, the `eq` method of class `EQ$Dict$IntLit` casts the arguments `this$` and `that` from `Object` to `IntLit`, assigns the results to fresh local variables `i1` and `i2`, respectively, and uses these local variables instead of `this$` and `that` in the rest of the method body.

Besides the dictionary interface, each dictionary class also implements the interface `javagi.runtime.Dictionary` provided by JavaGI's runtime system. This interface requires a method `_$JavaGI$implementationInfo` used to reify information about the implementation. More specifically, the `ImplementationInfo` object returned by the method contains information about the type parameters, the interface, the implementing types, the constraints, and the abstract methods of the implementation. Further, it also specifies which of the implementing types are dispatch types.[3]

Figure 6.2 also contains the translation of the parameterized implementation of `EQ` for `List<X>` from Section 2.1.3. The resulting code demonstrates that the translation mechanism generalizes seamlessly to parameterized and type conditional implementations. The translation of inheritance between implementation definitions (not shown in Figure 6.2) is also straightforward because it simply boils down to inheritance between the corresponding dictionary classes.

6.3 Run-Time System

JavaGI's run-time system maintains the available implementation definitions, checks their well-formedness according to the criteria in Section 2.3.4, loads new implementation definitions at run time, and performs dynamic dispatch on retroactively implemented methods. Moreover, it carries out certain cast operations, **`instanceof`** tests, and identity comparisons (`==`), for which the regular JVM instructions are not sufficient in the presence of wrappers (see also Section 4.3). For example, to execute a JavaGI cast `(J)obj`, where

[3] Section 2.3.4 and Figure 3.17 defined the notion of dispatch types.

Figure 6.2 Translation of retroactive implementations from Sections 2.1.2 and 2.1.3.

```
// Java 1.4
import javagi.runtime.Dictionary;
import javagi.runtime.ImplementationInfo;
import java.util.*;
// Translations of EQ [Expr]
public class EQ$Dict$Expr implements EQ$Dict, Dictionary {
  public boolean eq(Object this$, Object that) {
    // load-time checks ensure that this method is never called
    throw new Error("abstract method");
  }
  public ImplementationInfo _$JavaGI$implementationInfo() { ... }
}
// Translation of  EQ [IntLit]
public class EQ$Dict$IntLit implements EQ$Dict, Dictionary {
  public boolean eq(Object this$, Object that) {
    IntLit i1 = (IntLit) this$;  IntLit i2 = (IntLit) that;
    return i1.value == i2.value;
  }
  public ImplementationInfo _$JavaGI$implementationInfo() { ... }
}
// Translation of EQ [PlusExpr]
public class EQ$Dict$PlusExpr implements EQ$Dict, Dictionary {
  public boolean eq(Object this$, Object that) {
    PlusExpr e1 = (PlusExpr) this$;  PlusExpr e2 = (PlusExpr) that;
    return EQ$Wrapper.eq$Dispatcher(e1.left, e2.left) &&
           EQ$Wrapper.eq$Dispatcher(e1.right, e2.right);
  }
  public ImplementationInfo _$JavaGI$implementationInfo() { ... }
}
// Translation of EQ [List<X>]
public class EQ$Dict$List implements EQ$Dict, Dictionary {
  public boolean eq(Object this$, Object that) {
    List l1 = (List) this$;  List l2 = (List) that;
    Iterator thisIt = l1.iterator(); Iterator thatIt = l2.iterator();
    while (thisIt.hasNext() && thatIt.hasNext()) {
      Object thisX = thisIt.next();  Object thatX = thatIt.next();
      if (! EQ$Wrapper.eq$Dispatcher(thisX, thatX)) return false;
    }
    return !(thisIt.hasNext() || thatIt.hasNext());
  }
  public ImplementationInfo _$JavaGI$implementationInfo() { ... }
}
```

J is an interface, the run-time system performs the following steps:

1. Remove a potential wrapper around obj to arrive at object obj'.

2. Check whether the run-time type T of obj' implements J.

3a. If T implements J retroactively then the result of the cast is obj' wrapped by a J-wrapper.

3b. If T implements J non-retroactively then the result of the cast is simply obj'.

3c. If T does not implement J then the cast throws a ClassCastException.

The JavaGI-specific version of **instanceof** works similarly but evaluates to true in cases 3a and 3b and to false in case 3c. Performing an identity comparison x == y on two non-primitive values x and y requires to remove potential wrappers around x and y (unless their static types are class types different from Object) before performing the corresponding JVM instruction.

Initialization of the run-time system happens lazily through a static initializer. The initializer code first searches all available implementation definitions by reading the names of dictionary classes from extra files generated by the compiler. It then loads the dictionary classes and performs the well-formedness checks described in Section 2.3.4. Finally, it groups the implementation definitions according to the interface they implement. If several implementations for the same interface exist, the run-time system orders them by specificity to ensure correct and efficient method lookup.

Optionally, JavaGI's run-time system installs a custom class loader, which assists in checking the "downward closed" and the "completeness" criterion described in Section 2.3.4. Without the custom class loader, the run-time system has to resort to conservative approximations of these criteria. The custom class loader could also automatically load the retroactive implementations whenever java.lang.Class.forName(String name) is invoked, thus eliminating the need for the custom class loading method provided by JavaGI's runtime system.[4]

6.4 JavaGI Eclipse Plugin

The JavaGI Eclipse Plugin (JEP) allows the development of JavaGI applications using the familiar Eclipse IDE [60]. The aim of JEP is to provide a drop-in replacement for Eclipse's Java Development Toolkit (JDT). JEP's functionality includes syntax highlighting, support for compiling and executing JavaGI programs, interoperability between JavaGI and Java projects, most of JDT's refactorings, and Java-specific content assist.[5] At the moment, JEP does not support the debugging of JavaGI programs and content assist for JavaGI-specific constructs. Implementing these features, however, is straightforward and does not pose significant challenges.

[4] The current implementation does not support this feature, though.

[5] Content assist is an Eclipse feature that enables completion of code fragments.

7 Practical Experience

The preceding chapter described the implementation of a compiler and a run-time system for JavaGI. This chapter reports on practical experience with JavaGI and its implementation. First, it describes three real-world case studies that go far beyond the toy examples from Chapter 2. The case studies once again demonstrate the benefits of generalized interfaces and they show that the JavaGI compiler and the run-time system are stable and mature. Second, the chapter presents benchmark data indicating that the JavaGI compiler generates code with good performance: plain Java code compiled with the JavaGI compiler runs as fast as the same code compiled with a regular Java compiler, but there is a performance penalty for JavaGI-specific features. The source code of the case studies and the benchmarks is available online [239].

Chapter Outline. Section 7.1 describes three real-world case studies and contrasts the solutions in JavaGI with solutions in other languages. Section 7.2 presents benchmarks and compares the performance of JavaGI with that of plain Java.

7.1 Case Studies

We performed three case studies using the JavaGI implementation described in Chapter 6: a framework for evaluating XPath [47] expressions (Section 7.1.1), a web application framework (Section 7.1.2), and a refactoring of the Java Collection Framework [211] (Section 7.1.3).

7.1.1 XPath Evaluation

For this case study, we implemented a framework for evaluating XPath[1] expressions. The framework is not bound to a specific XML [27] implementation but can be used with and adapted to many different object models, including object models unrelated to XML. For

[1] XPath is a language for addressing parts of an XML [27] document [47].

Figure 7.1 Jaxen's `Navigator` interface (excerpt).

```
// Java
package org.jaxen;
import java.util.Iterator;
public interface Navigator {
  // Returns an Iterator matching the child XPath axis.
  Iterator getChildAxisIterator(Object node) throws UnsupportedAxisException;
  // Returns the local name of the given element node.
  String getElementName(Object element);
  // Returns the qualified name of the given attribute node.
  String getAttributeQName(Object attr);
  // Loads a document from the given URI.
  Object getDocument(String uri) throws FunctionCallException;
  // Returns a parsed form of the given XPath string.
  XPath parseXPath(String xpath) throws SAXPathException;
  // omitted 36 methods
}
```

plain Java, Jaxen [102] already provides such a framework. The goal of the case study was to compare the JavaGI solution with the one provided by Jaxen.

The Jaxen Approach

Jaxen specifies an interface `Navigator`, which contains all methods required by its internal XPath evaluation engine. The interface has methods for accessing the names of element nodes and attribute nodes, for retrieving the values of attribute and text nodes, for constructing iterators over the various XPath axis, and so on.[2] To stay generic, the `Navigator` interface uniformly uses `Object` as type for the different node kinds. Figure 7.1 shows an excerpt from this interface.

Using Jaxen requires to implement the `Navigator` interface for the object model under consideration. To simplify this task, Jaxen comes with an abstract class `DefaultNavigator` that implements the `Navigator` interface and contains default implementations for roughly half of the interface's methods. Jaxen also provides concrete `Navigator` implementations for various XML libraries such as dom4j [57] and JDOM [94]. Figure 7.2 shows an excerpt of Jaxen's implementation of the `Navigator` interface for dom4j.

The JavaGI Approach

The JavaGI XPath evaluation framework specifies a model of the XPath node hierarchy based on interfaces rooted at interface `XNode`. These interfaces provide the methods required by the evaluation engine. (Internally, the JavaGI framework relies on Jaxen to perform the actual evaluation.) Figure 7.3 shows those parts of the node hierarchy that correspond to the excerpt of the `Navigator` interface in Figure 7.1.

[2] The following discussion ignores comment, namespace, and processing instruction nodes. It is straightforward to include these additional kinds of nodes.

Figure 7.2 Jaxen's implementation of the `Navigator` interface for dom4j (excerpt).

```
// Java
package org.jaxen.dom4j;
import java.util.Iterator;
import org.jaxen.DefaultNavigator;
import org.jaxen.XPath;
import org.jaxen.JaxenConstants;
import org.jaxen.FunctionCallException;
import org.jaxen.saxpath.SAXPathException;
import org.dom4j.Attribute;
import org.dom4j.Branch;
import org.dom4j.Element;
import org.dom4j.Document;
public class Dom4jNavigator extends DefaultNavigator {
  public Iterator getChildAxisIterator(Object node) {
    if (node instanceof Branch) return ((Branch)node).nodeIterator();
    else return JaxenConstants.EMPTY_ITERATOR;
  }
  public String getElementName(Object obj) {
    return ((Element)obj).getName();
  }
  public String getAttributeQName(Object obj) {
    return ((Attribute)obj).getQualifiedName();
  }
  public Object getDocument(String uri) throws FunctionCallException {
    try { return getSAXReader().read(uri); }
    catch (Exception e) {
      throw new FunctionCallException("Failed to parse document");
    }
  }
  public XPath parseXPath(String xpath) throws SAXPathException {
    return new Dom4jXPath(xpath);
  }
  // many methods omitted
}
// some auxiliary classes omitted
```

Figure 7.3 XPath node hierarchy (excerpt).

```
package javagi.casestudies.xpath;
import org.jaxen.UnsupportedAxisException;
import org.jaxen.FunctionCallException;
import org.jaxen.XPath;
import org.jaxen.saxpath.SAXPathException;
public interface XNode {
  Iterator<XNode> getChildAxisIterator() throws UnsupportedAxisException;
  // omitted 25 methods
}
public interface XElement extends XNode {
  String getName();
  // omitted 2 methods
}
public interface XAttribute extends XNode {
  String getQName();
  // omitted 2 methods
}
public interface XDocument extends XNode {
  static This getDocument(String uri) throws FunctionCallException;
  static XPath parseXPath(String xpath) throws SAXPathException;
}
// omitted interfaces XNamespace and XProcessingInstruction with
// 3 methods in total
```

A JavaGI programmer adapts existing object models to the XPath node hierarchy through retroactive interface implementations. Similar to Jaxen's `DefaultNavigator` class, the JavaGI version provides an abstract implementation of the `XNode` interface, which contains default implementations for 23 out of 26 methods. The rest of the section shows how we adapted the XML libraries dom4j [57] and JDOM [94] to the XPath node hierarchy.

dom4j. The dom4j library comes with its own node hierarchy rooted in the interface `org.dom4j.Node`. Figure 7.4 shows a diagram illustrating the adaptation of the dom4j API to the XPath node hierarchy.[3] To avoid code duplication, we made use of implementation inheritance, as shown in the diagram in Figure 7.5.[4] The implementation `XNode [XNode]` at the top of the diagram is the default implementation of the `XNode` interface mentioned before. For concreteness, Figure 7.6 shows some sample code from the dom4j adaptation. The sample code corresponds to the Java code in Figure 7.2.

[3] Diagrams use standard UML notation [165] to display packages, classes, interfaces, and inheritance. Dotted lines (a non-standard notation) represent non-abstract retroactive interface implementations, where the arrow points to the interface being implemented.

[4] Boxes with the stereotype «implementation» (or «abstract implementation») denote (abstract) implementation definitions. Arrows between implementation definitions denote inheritance links, the arrow pointing to the super implementation.

Figure 7.4 Adaptation of the dom4j API to the XPath node hierarchy.

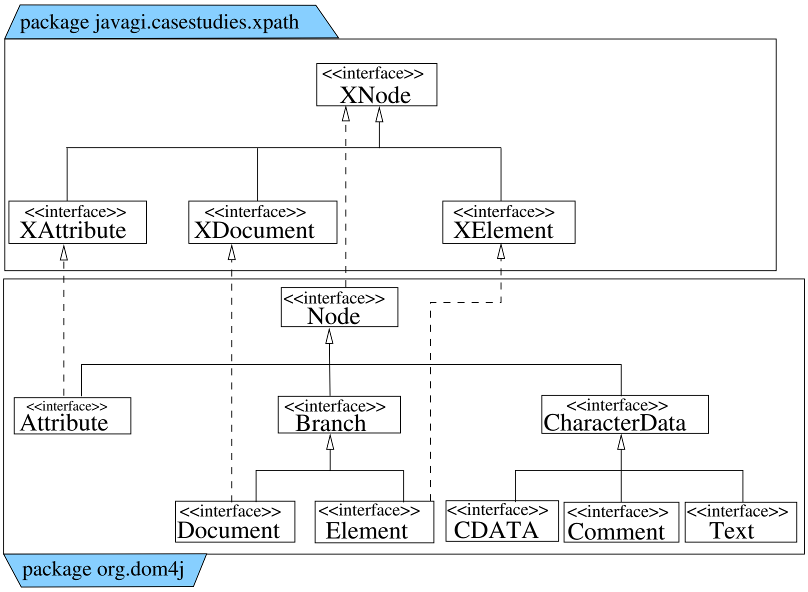


Figure 7.5 Uses of implementation inheritance in the adaptation for dom4j.

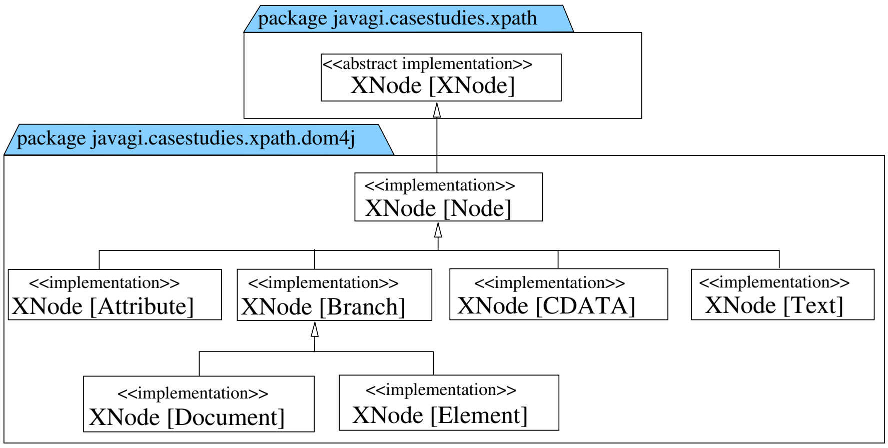


Figure 7.6 Sample code from the dom4j adaptation.

```
package javagi.casestudies.xpath.dom4j;
import java.util.Iterator;
import org.dom4j.Attribute;
import org.dom4j.Branch;
import org.dom4j.Document;
import org.dom4j.Element;
import org.dom4j.Node;
import org.jaxen.JaxenConstants;
import org.jaxen.XPath;
import org.jaxen.FunctionCallException;
import org.jaxen.saxpath.SAXPathException;
import javagi.casestudies.xpath.dom4j.XAttribute;
import javagi.casestudies.xpath.dom4j.XDocument
import javagi.casestudies.xpath.dom4j.XElement;
import javagi.casestudies.xpath.dom4j.XNode;
implementation XNode [Node] extends XNode [XNode] {
  Iterator<XNode> getChildAxisIterator() {
    return JaxenConstants.EMPTY_ITERATOR;
  }  // several methods omitted
}
implementation XNode [Branch] extends XNode [Node] {
  Iterator<XNode> getChildAxisIterator() {
    return this.nodeIterator();
  }  // omitted 1 method
}
implementation XElement [Element] {
  String getName() { return this.getName(); } // omitted 2 methods
}
implementation XAttribute [Attribute] {
  String getQName() { return this.getQualifiedName(); }
  // omitted 2 methods
}
implementation XDocument [Document] {
  static Document getDocument(String s) throws FunctionCallException
    { return DocumentLoader.load(s); }
  static XPath parseXPath(String xpath) throws SAXPathException {
    return new GIDom4jXPath(xpath);
  }
}
// omitted 9 implementation definitions and some auxiliary classes
```

Figure 7.7 Adaptation of the JDOM API to the XPath node hierarchy.

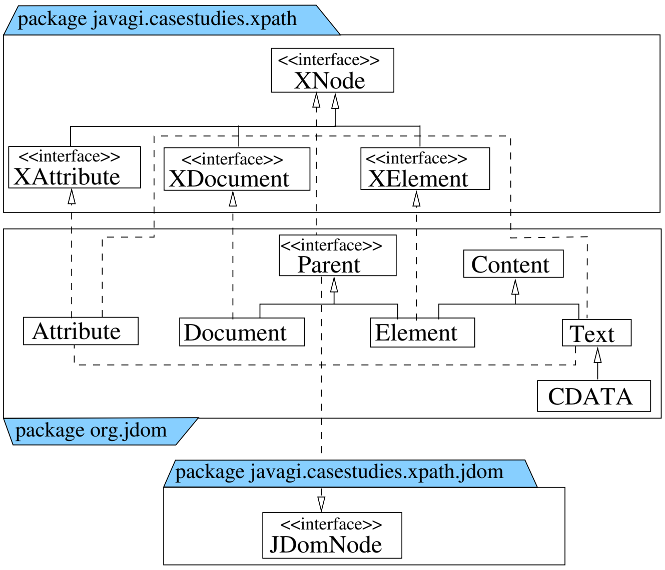
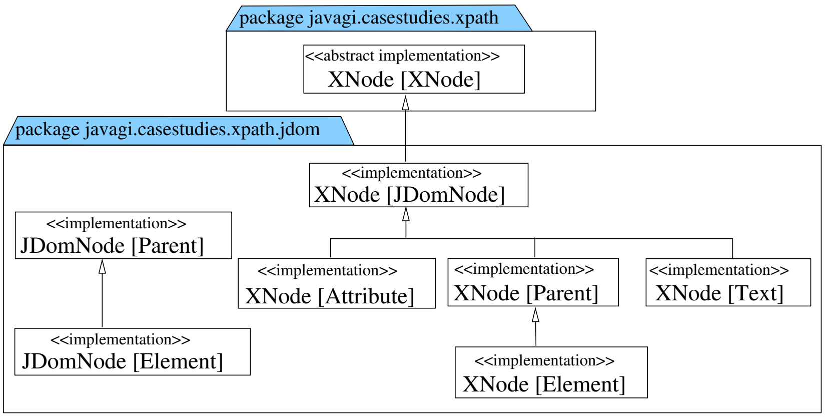


Figure 7.8 Uses of implementation inheritance in the adaptation for JDOM.



JDOM. Figure 7.7 shows the adaptation of JDOM's API to the XPath node hierarchy. JDOM uses its own set of classes and interfaces to represent the various XML node kinds. Unlike dom4j, the classes and interfaces do not form a true hierarchy because they do not have a designated root class (except `Object`). This non-hierarchic API is problematic because it offers no place for putting implementations of methods shared by several node kinds. (In the dom4j example, we simply placed such methods in the implementation `XNode [org.dom4j.Node]`. This approach allowed, for example, the reuse of several methods between `org.dom4j.Attribute`, `org.dom4j.CDATA`, and `org.dom4j.Text`.)

Despite the non-hierarchic JDOM API, we managed to get by without code duplication by introducing an interface `JDomNode`, which serves as the (artificial) root of the JDOM

API. Figure 7.7 shows JDomNode and the corresponding implementations at the bottom. Thanks to the newly introduced root interface, code duplication could be avoided by implementation inheritance (see Figure 7.8).

Assessment

The JavaGI-based XPath evaluation framework has several advantages over the plain Java solution. The main advantage is that the JavaGI-based approach requires significantly fewer cast operations than the solution using Jaxen. Jaxen's implementation of the Navigator interface for dom4j requires 28 casts, the one for JDOM even 47 casts. Most of these casts are caused by the use of Object as the type of nodes in the Navigator interface (see Figure 7.2). In contrast, the JavaGI solution requires *no casts at all* to adapt both dom4j and JDOM to the node hierarchy for XPath evaluation.

An approach to lower the number of casts required by the Jaxen solution would be to parameterize the Navigator interface by the different node types and use these type parameters in method signatures. While such a parameterization would lower the number of casts significantly, it would also limit expressiveness. For instance, in dom4j both interfaces org.dom4j.CDATA and org.dom4j.Text may serve as text nodes, however, their least upper bound org.dom4j.CharacterData may not. Thus, there exists no sensible instantiation for the text node type. Hence, a generic version of the Navigator interface is not an option.

Another advantage of the JavaGI approach is that it offers a simple and clear specification of the requirements an object model has to fulfill to support XPath-based navigation. The JavaGI solution specifies six interfaces for the different node kinds. The interfaces have at most three methods, except for the XNode interface, which has 26 methods. Using different interfaces for different node kinds results in a clear separation of concerns. In contrast, the Jaxen solution requires clients to implement the 41 methods of the monolithic Navigator interface.

7.1.2 JavaGI for the Web

As a second case study, we developed a web application framework in JavaGI. The framework uses the Java servlet technology [215] and borrows ideas from the Haskell [173] framework WASH [224]. We applied the framework to implement an application handling workshop registrations. The goal of the case study was to evaluate whether JavaGI can provide the same static guarantees as WASH and how JavaGI behaves in a servlet environment where dynamic loading is the default.

WASH is a domain specific language for server-side Web scripting embedded in Haskell. It supports the generation of HTML [234], guaranteeing well-formedness and adherence to a Document Type Definition (DTD [27]) . Furthermore, there are operators for defining typed input widgets and ways to extract the user inputs from them without being exposed to the underlying string-based protocol. A WASH program automatically redisplays a form until the user has entered syntactically correct values in all input widgets.

The implementation of WASH relies heavily on Haskell's type classes. It enforces quasi-validity of HTML documents by providing type classes specifying the allowed parent-

Figure 7.9 Modeling HTML elements and attributes.

```
package javagi.casestudies.servlet;
class UL extends Element implements ChildOfBODY, ChildOfLI /* rest omitted */ {
  public String getName() { return "ul"; }
  public UL add(ChildOfUL... children) {
    super.add(children);
    return this;
  }
}
interface ChildOfUL extends Node {}
interface ChildOfLI extends Node {}
class AttrCLASS extends Attribute
                implements ChildOfUL, ChildOfLI /* rest omitted */ {
  public String getName() { return "class"; }
  public AttrCLASS(String v) { super (v); }
}
class GenHTML {
  public static UL ul(ChildOfUL... cs) { return new UL().add(cs); }
  public static AttrCLASS attrCLASS(String v) { return new AttrCLASS(v);  }
  // remaining factory methods omitted
}
```

child relationships among elements, attributes, and other kinds of HTML nodes. These type classes are generated from a HTML DTD. Also, the type of an input widget is parameterized by the expected type of the value. Again, a type class provides type-specific parsers and error messages.

Much of the core functionality of WASH can be implemented in JavaGI. Briefly put, plain Java interfaces are sufficient to support generation of quasi-valid HTML documents, retroactive implementation is useful in many places, the implementation of typed input widgets relies on static interface methods, and dynamic loading of implementations is essential for working in a servlet environment.

To generate HTML documents, the JavaGI framework defines a type hierarchy with a `Node` interface on top, abstract classes `Element` and `Attribute`, and a class `Text`, all implementing `Node`. In addition, there are element- and attribute-specific subclasses and interfaces: for each kind of attribute, there is a subclass of `Attribute`; for each kind of element, there is a subclass of `Element` and a subinterface of `Node` that characterizes potential child nodes of this kind of element. For convenience, there is a class `GenHTML` with static factory methods for creating all kinds of nodes. Figure 7.9 contains excerpts from these classes.

The implementation of typed input fields relies on the `Parseable` interface already explained in Section 2.1.4. An input field for a value of type `X` is represented by an object of class `Field<X>`. The method

```
  public <X> Field<X> defineField(String name, String type, X init)
    where X implements Parseable;
```

is retroactively attached to javax.servlet.ServletRequest, which contains the internal data of an HTML-form submission to a servlet. The defineField method parses the submitted string, detects errors, and creates a Field<X> instance. The latter has methods INPUT getInput(), which constructs a HTML input element, and X getValue(), which returns the field's value.

Figure 7.10 shows parts of a workshop registration application that we implemented with the JavaGI web framework.[5] The Register class inherits from JavaGIServlet, which extends javax.servlet.http.HttpServlet to perform dynamic loading of implementation definitions. The doPost method first creates input fields using defineField. Next, the code applies method fieldsOK() to the ServletRequest object to check whether all required user entries are present and syntactically correct. If so, the servlet proceeds to processing the user's entry. Otherwise, the servlet creates an object structure representing the HTML page. This structure includes the input elements extracted from the fields created in the first step. In case of a syntactically invalid input, the elements contain suitable error notifications. Finally, the code serializes the HTML structure to the servlet response and terminates. The screenshot in Figure 7.11 shows the registration page after the user entered an incorrect date string.

Assessment

The JavaGI solution yields the same static guarantees as the WASH system with respect to well-formedness and validity of the generated HTML and with respect to automatic form validation. Further, the case study demonstrates that JavaGI integrates seamlessly into a servlet environment where all application code is loaded dynamically.

WASH also provides a typed submit facility, where submit buttons (e.g. the input element with type "submit" in Figure 7.10) are created implicitly. In WASH, the constructor for a submit button accepts a list of typed fields and a callback function that accepts an argument list typed according to the fields. On submission of the page, the submit button invokes the callback function, provided the values of all fields validate correctly. This facility is not incorporated in the JavaGI version because it seems to require higher-kind generics [152]. We were, however, able to implement a less flexible approach that requires programmers to prepare designated classes for storing the submitted data.

A Java implementation of WASH's core functionality is possible but requires more work than the solution with JavaGI. Creating class instances from parsed and validated input data would have to be performed using the Factory pattern [73], thus requiring an extra parameter for many methods. Moreover, retroactive interface implementations would have to be emulated either through the Adapter pattern [73] or with static helper methods.

7.1.3 Java Collection Framework

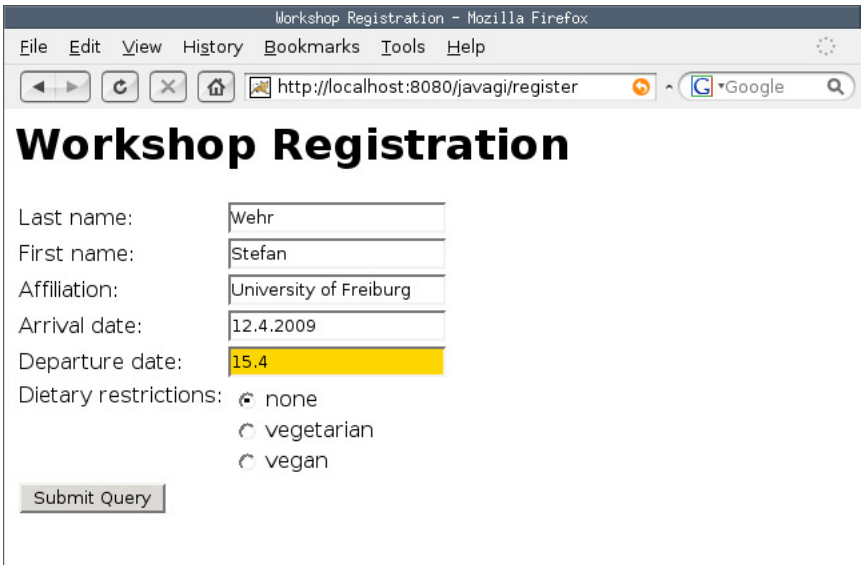
The Java Collection Framework (JCF [211]) provides interfaces for common data structures such as Collection, Set, List, and Map as well as various implementations of these data structures. By default, all collections are modifiable but programmers can

[5] Some familiarity with servlet programming is assumed.

Figure 7.10 Sample code from the workshop registration application.

```
package javagi.casestudies.servlet;
import java.io.IOException;
import java.util.Date;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import static javagi.casestudies.servlet.GenHTML.*;
enum Diet { NONE, VEGETARIAN, VEGAN }
public class Register extends JavaGIServlet {
  protected void doPost(HttpServletRequest req, HttpServletResponse res) {
    Field<String> ln = req.<String>defineField("ln", "text", "");
    Field<Date> ad = req.<Date>defineField("ad", "text", null);
    // code for remaining input fields fn, af, dd, and diet omitted
    if (req.fieldsOK()) {
      processRegistration(res, ln.getValue(), fn.getValue(),
                          af.getValue(), ad.getValue(),
                          dd.getValue(), diet.getValue());
    } else {
      TABLE ptable = table();
      TABLE diettable = table();
      FORM pform = form(attrMETHOD("post"), attrACTION(""), ptable);
      HTML page = html(head(title("Workshop Registration")),
                       body(h1("Workshop Registration"), pform));
      ptable.addRow(text("Last name: "), ln.getInput());
      ptable.addRow(text("Arrival date: "), ad.getInput());
      // code for remaining input fields omitted
      ptable.addRow(input(attrTYPE("submit")));
      try {
        res.setContentType("text/html; charset=UTF-8");
        page.out(res.getWriter());  res.flushBuffer();
      } catch (IOException e) {}
    }
  }
  public void processRegistration(HttpServletResponse res, String ln,
                                  String fn, String af, Date ad,
                                  Date dd, Diet diet) {
    // code omitted for brevity
  }
}
```

Figure 7.11 Sample page of the workshop registration application.


Workshop Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/javagi/register
Google
Submit Query

Workshop Registration

Last name: Wehr

First name: Stefan

Affiliation: University of Freiburg

Arrival date: 12.4.2009

Departure date: 15.4

Dietary restrictions: ● none ○ vegetarian ○ vegan

explicitly mark a collection as unmodifiable. However, unmodifiable collections have the same interface as modifiable ones, so programmers may call modifying operations on an unmodifiable collection, resulting in a run-time error.

Huang and colleagues [91] demonstrated how to turn such run-time errors into compile-time errors using type conditionals as provided by their Java extension cJ. The basic idea is to parameterize a collection not only over the element type but also over a "mode" type that specifies further attributes of a collection. Type conditionals then ensure that operations modifying a collection are only available if the mode parameter indicates that the collection is indeed modifiable. In a case study, Huang and collaborators refactored the whole JCF using this idea.

While JavaGI's type conditionals are slightly less powerful than cJ's, all features needed for refactoring the JCF are available. Hence, porting the refactored JCF to JavaGI was straightforward.

As an example, Figure 7.12 shows JavaGI's version of the `java.util.List` interface [212] with type conditionals. As in Java, the type parameter `E` is the type of the list elements. The second type parameter `M`, not present in the original Java version of the interface, denotes the mode of the collection, where the mode is one of the classes shown at the bottom of the figure: `Modifiable` specifies that individual list elements can be changed, but no elements can be added or removed; `Shrinkable` specifies that elements can be removed; `Resizable` specifies that elements can be added and removed. Mode `Object` indicates that the list cannot be modified at all.

For example, the `set` method of interface `List` may only be called if `M` is at least `Modifiable`, whereas `clear` requires that `M` is (a subtype of) `Shrinkable`. For instance, assume that `list` has static type `List<String,Modifiable>`. Then the call `list.clear()` fails at compile time because `Modifiable` is not a subtype of `Shrinkable`.

Figure 7.12 Refactoring of the Java Collection Framework.

```
package cj.util;
public interface List<E,M> extends Collection<E,M> {
  E set(int index, E element) where M extends Modifiable;

  void add(int index, E element) where M extends Resizable;
  boolean add(E o) where M extends Resizable;
  boolean addAll(Collection<? extends E,?> c) where M extends Resizable;

  E remove(int index) where M extends Shrinkable;
  boolean remove(Object o) where M extends Shrinkable;
  boolean removeAll(Collection<?,?> c) where M extends Shrinkable;
  boolean retainAll(Collection<?,?> c) where M extends Shrinkable;
  void clear() where M extends Shrinkable;

  // omitted 16 read-only operations such as size(), isEmpty()
}
// Mode types (besides Object):
public class Modifiable {}
public class Shrinkable extends Modifiable {}
public class Resizable extends Shrinkable {}
```

In contrast, the corresponding Java code would compile successfully but result in a runtime exception.

Assessment

The main difference (besides syntactic ones) between the JavaGI and the cJ versions of the JCF refactoring is that cJ offers a grouping mechanism for type conditionals. This grouping mechanism allows programmers to specify a type conditional for a whole group of methods. JavaGI requires restating the conditional for each method.

Furthermore, in cJ superclasses and fields are also subject to type conditionals. However, these features were not needed for the JCF case study, and the original cJ paper [91] does not contain realistic examples using them. Hence, we conjecture that most applications of type conditionals do not need this additional level of expressivity.

7.2 Benchmarks

Several benchmarks were used to compare the performance of JavaGI programs with their Java counterparts. The results show that the JavaGI compiler generates code with good performance. Plain Java code compiled with the JavaGI compiler runs as fast as the same code compiled with a regular Java compiler, but there is a performance penalty for JavaGI-specific features.

All benchmarks were executed on a Thinkpad x60s with an Intel Core Duo L2400 1.66 GHz CPU and 4GB of RAM, running Linux 2.6.24. The Java Virtual Machine

Figure 7.13 Micro benchmarks for different kinds of method call instructions.

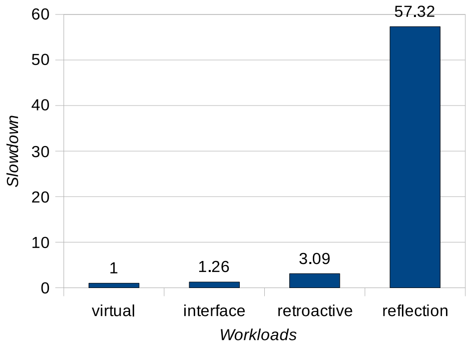


Figure 7.14 Micro benchmarks for casts, **`instanceof`** tests, and identity comparisons.

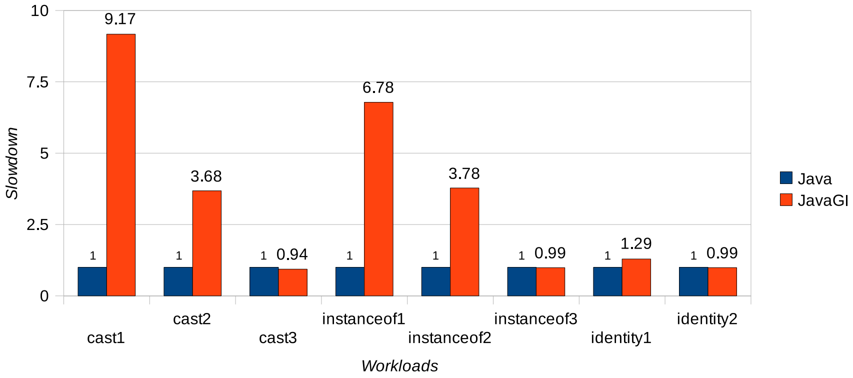
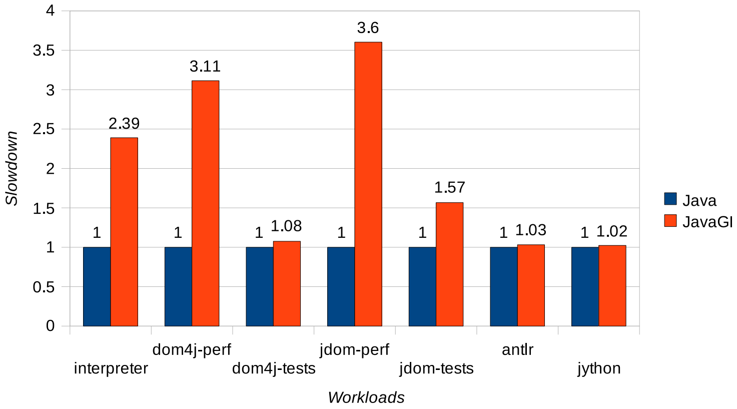


Figure 7.15 Performance of JavaGI with respect to Java.



(JVM [125]) used was the server virtual machine of Sun's Java SE (version 1.6.0_06 [218]). The Java code was compiled with the Eclipse Compiler for Java (version 0.883_R34x [62]), the baseline of the JavaGI compiler. The JavaGI code was compiled with the JavaGI compiler presented in Chapter 6.

The individual workloads were repeatedly executed, until performance stabilized. The mean of the last three or five repetitions (depending on the total number of repetitions) then represents the performance index for a workload. The raw benchmark data is available online [239].

Figure 7.13 shows performance results of micro benchmarks demonstrating that calls of retroactively implemented methods are 3.09 times slower than method calls using the `invokevirtual` instruction of the JVM and 2.46 times slower than calls using the `invokeinterface` instruction. This slowdown is not surprising because the machinery needed to perform dynamic lookup of retroactively implemented methods is more involved than that required for ordinary class or interface methods (see Section 6.2.2). For reference, Figure 7.13 also includes the slowdown of method calls via reflection.

Figure 7.14 compares cast operations, **`instanceof`** tests, and identity comparisons (==) in Java and JavaGI. The workload *cast1* casts objects to an interface that these objects implement directly in the Java version but retroactively in the JavaGI version. In general, casts are complex operations in JavaGI (see Section 6.3), so the JavaGI version is 9.17 times slower than the Java version.[6] The workload *cast2* casts objects with static type `Object` to a class type. The JavaGI version is 3.68 slower than its Java counterpart because such casts require unpacking of potential wrappers. The workload *cast3* casts objects whose static types are class types other than `Object` to some other class type. In such situations, the JavaGI compiler generates a regular `checkcast` JVM instruction, so there is no significant difference between the Java and the JavaGI versions. The workloads *instanceof1*, *instanceof2*, and *instanceof3* are similar to cast1, cast2, and cast3, respectively, but perform **`instanceof`** tests instead of cast operations. The workload *identity1* checks whether two objects with static type `Object` are identical using the == operator. The JavaGI version is slower because it involves unpacking of potential wrappers. The workload *identity2* is similar but compares objects whose static types are class types different from `Object`. In this case, no unpacking is required, so there is no significant performance difference.

Figure 7.15 compares the performance of JavaGI with that of Java using seven real-world workloads. The *interpreter* workload is an interpreter for a language with arithmetic expressions, variables, conditionals, and function calls, implemented once in plain Java and once in JavaGI. The Java interpreter uses ordinary virtual methods to perform expression evaluation, whereas the JavaGI interpreter uses retroactively implemented methods for this purpose. The JavaGI version is 2.39 times slower than the Java version. A large number of calls to retroactively implemented methods in the JavaGI version lead to this slowdown.

[6] Under a different workload, casts in JavaGI were up to 830 times slower than in Java. However, this different workload is unrealistic because it performs repeated cast operations always on the *same* object. In this scenario, a caching mechanism of the JVM apparently leads to very fast execution times. In contrast, workload *cast1* is more realistic because it uses *different* objects to measure the performance of cast operations.

The workloads *dom4j-perf*, *dom4j-tests*, *jdom-perf*, and *jdom-tests* were taken from the Jaxen [102] distribution (see Section 7.1.1). Dom4j-perf and jdom-perf are performance tests for the adaptation of dom4j [57] and JDOM [94] to jaxen's XPath evaluation framework, dom4j-tests and jdom-tests are the corresponding unit-test suites. The Java versions of these workloads directly use the code from the jaxen distribution, whereas the JavaGI versions replace jaxen's XPath evaluation framework with the framework presented in Section 7.1.1.

The JavaGI versions of the dom4j-tests and jdom-tests workloads are 1.08 and 1.57, respectively, times slower than the Java versions. The domj4-perf and jdom-perf workloads for JavaGI are 3.11 and 3.6, respectively, times slower than their Java counterparts.[7] Numerous invocations of retroactively implemented methods, the construction of many wrapper objects, and a large number of cast operations are the main reason for this slowdown. (Many of the casts are inserted automatically by type erasure, the remaining ones are part of the internal adaptation layer between the public API of the JavaGI framework and the evaluation engine provided by jaxen, on which the JavaGI framework builds.)

The workloads *antlr* and *jython* in Figure 7.15 are from the DaCapo benchmark suite (version 2006-10-MR2 [18]). The JavaGI and Java versions of these two workloads use the same source code, once compiled with the JavaGI and once with the Java compiler. The results show no significant difference between JavaGI and Java.

[7] A slight variation of the workloads dom4j-perf and jdom-perf increases the performance of the Java versions. In this setting, the workloads for JavaGI are 3.59 and 5.73, respectively, times slower than their Java counterparts. The variation, however, is quite unrealistic because it evaluates an XPath expression repeatedly on the *same* object graph. In contrast, the original domj4-perf and jdom-perf workloads are more realistic because they use *different* object graphs to evaluate XPath expressions.

8
Related Work

This dissertation builds on results from different research areas. The present chapter summarizes these results and compares them with the contributions of JavaGI.

Chapter Outline. The chapter consists of eleven sections.

- Section 8.1 compares type classes in Haskell with generalized interfaces in JavaGI.
- Section 8.2 reviews work on generic programming.
- Section 8.3 discusses different approaches to family polymorphism.
- Section 8.4 presents solutions to software extension, adaptation, and integration problems.
- Section 8.5 analyzes systems supporting external methods and multiple dispatch.
- Section 8.6 discusses ways to typecheck binary methods.
- Section 8.7 presents work related to type conditionals.
- Section 8.8 considers traits.
- Section 8.9 summarizes work on advanced subtyping mechanisms.
- Section 8.10 reviews work related to the undecidability results of Chapter 5.
- Section 8.11 compares the current design of JavaGI with an earlier version.

8.1 Type Classes in Haskell

Type classes [236, 107, 104] in the functional programming language Haskell [173] are closely related to the work of this dissertation. Like a generalized interface, a type class

Figure 8.1 Type classes in Haskell.
The type `[a]` is the type of lists with elements of type `a`. The pattern `[]` matches the empty list, whereas `y:ys` matches any non-empty list binding `y` to the head and `ys` to the tail of the list. The type `Maybe a` denotes an optional value of type `a`, where `Nothing` signals absence of a value and `Just x` signals presence of value `x`. (In JavaGI, every value of a reference type is optional in the sense that **`null`** signals absence of a value.)

```
— Haskell
class EQ a where — Type variable ”a” denotes the implementing type
  eq :: a → a → Bool

— Optional type signature with constraint ”EQ a” making the
— ”eq” operation available on values of type ”a”
find :: EQ a => a → [a] → Maybe a
find _ []     = Nothing
find x (y:ys) = if eq x y then Just x else find x ys

instance EQ Int where
  eq i j = ...
— Parametric and constrained (type—conditional) instance definition
instance EQ a => EQ [a] where
  eq []     []     = True
  eq (x:xs) (y:ys) = eq x y && eq xs ys
  eq _      _      = False
```

declares the signatures of its member functions depending on one or more specified implementing type. (The Haskell 98 standard [173] supports only one implementing type, but multi-parameter type classes [174] lift this restriction.) Unlike in JavaGI, however, member functions of type classes are not attached to some receiver object but denote top-level functions that may be overloaded for different types. Thus, methods of Haskell type classes are similar to static interface methods in JavaGI. One difference is that Haskell infers, at least in most cases, the instance from which a method should be taken, whereas this information has to be specified explicitly in JavaGI. Haskell's type classes support multiple inheritance, just as interfaces in JavaGI do. Further, both languages provide constraint mechanisms to limit possible instantiations of universally quantified type variables. In contrast to JavaGI, Haskell infers constraints and types automatically. Like JavaGI's retroactive interface implementations, Haskell's instance definitions specify that one or several types are members of some type class, thereby providing overloaded versions of the member functions of the class. As in JavaGI, instances are defined in separation from types, and they can be parametric and subject to constraints. To illustrate the correspondence between type classes and generalized interfaces, Figure 8.1 recasts some of the examples from Section 2.1.2 and Section 2.1.3 in Haskell.

Functional dependencies [105], a well-known extension of Haskell type classes, express dependencies between implementing types. For example, given the declaration **`class`** `C a b | a → b ...` of a two-parameter type class `C`, the functional dependency `a → b` specifies that in all instances of `C` the first implementing type uniquely determines the second. Such dependencies are to some degree expressible in JavaGI because its type system (as well as Java's) requires that a program does not define two implementations for different instantiations of the same interface (see criterion "unique interface instantiation and non-dispatch types" in Section 2.3.4 and criterion WF-PROG-2 in Section 3.5.3). For example, the JavaGI interface `interface I<b>[a] ...` specifies the same dependency between the type variables `a` and `b` as the type class `C` just presented. More complex functional dependencies such as `a` → `b`, `b` → `a` are not expressible in JavaGI. Associated types [156, 42, 41] present an alternative to functional dependencies (see Section 8.2).

Haskell also allows constructor classes [103] (type classes whose implementing types are in fact type constructors). JavaGI only supports first-order parametric polymorphism (as Java does). We conjecture that lifting this restriction [152] would allow a mechanism similar to constructor classes for JavaGI. Definitions of type classes in Haskell may provide default implementations for the methods of the type class. JavaGI can encode such default implementations with abstract implementations and implementation inheritance (see Section 2.1.5).

In comparison with object-oriented languages, Haskell has neither subtyping nor dynamic dispatch. Thus, Haskell can construct evidence for type-class instances needed in a function body statically or from the evidence present at the call sites of the function. This approach is too limiting for JavaGI because it either prevents dynamic dispatch or severely restricts the choice of compilation units into which retroactive implementations can be placed. Hence, one major contribution of JavaGI with respect to Haskell is the type-safe integration of subtyping and dynamic dispatch. Another difference between the two languages is that type classes only constrain types but never appear as types on their own. (There exists, however, an extension [225] that provides exactly this feature.) In contrast, JavaGI's single-headed interfaces can be used in constraints and as types.

The OOHaskell project [116] shows how Haskell 98 with common extensions supports many object-oriented programming idioms such as encapsulation, mutable state, inheritance, and overriding. Essentially, OOHaskell builds on extensible polymorphic records from the HList library [117] and on a semi-explicit subsumption operation. The approaches of OOHaskell and JavaGI are different: OOHaskell emulates object-oriented programming in Haskell, whereas JavaGI extends an object-oriented programming language with features influenced by Haskell.

8.2 Generic Programming

Concepts for C++ borrow ideas from Haskell type classes to specify requirements on template parameters [156, 139, 100, 184, 84, 16]. The main motivation behind concepts is to improve error messages caused by malformed template instantiations and to enable separate compilation for templates. Like type classes and generalized interfaces, concepts can span multiple types, they support some form of inheritance, and they can appear in constraints. In addition, concepts can also contain type definitions, leading to the notion of associated types [156]. There are two choices for implementing a concept with existing types [84]: either programmers provide explicit concept maps (similar to retroactive interface implementations in JavaGI and instance definitions in Haskell) or the compiler derives an implicit implementation based on the types and operations in scope. Like retroactive implementations, concept maps can be parametric and subject to type conditions. Siek and Lumsdaine [200] provided a formalization of concepts as an extension to System F [80, 187], which the first author extended [201] to a realistic programming language $\mathcal{G}$ that allows prototype implementations of the Standard Template Library [206] and the Boost Graph Library [202]. The main difference between concepts and JavaGI's generalized interfaces is that concepts are resolved at compile time: the compiler instantiates a template parameter based on the most specific implementations of the concepts imposed on the parameter. In contrast, JavaGI resolves methods of retroactive implementations dynamically through multiple dispatch. Another difference is that concept maps are lexically scoped whereas retroactive implementations share a global scope. Further, the concept mechanism as presented by Gregor and colleagues [84] supports concept-based overloading, same-type and negative constraints, and constraint propagation [101]. The idea of negative constraints conflicts with JavaGI's open-world assumption for retroactive implementations. Concept-based overloading is not available in JavaGI because neither static nor dynamic resolution of overloading based purely on concepts (i.e. implementation constraints) is possible due to JavaGI's open-world assumption and its type-erasure semantics, respectively.

For the purpose of illustration, Figure 8.2 shows a concept-based encoding of some of the examples from Section 2.1.2 and Section 2.1.3. (Figure 8.1 shows the Haskell version of these examples.)

A comparative study [75, 74] identified eight features that are important to properly support generic programming. Apart from associated types and the closely related feature of type aliases, JavaGI supports all of them, including two properties (“multi-type concepts” and “retroactive modeling”) not supported by Java. Other researchers proposed associated types as extensions of Haskell type classes [42, 41] and C# [101], so we conjecture that their addition to JavaGI does not pose significant challenges.

8.3 Family Polymorphism

Traditional polymorphism fails to express collaborations between families of types in a way that is both type safe (mixing objects from different families is rejected at compile time) and generic (abstraction over the family per se is possible). Ernst [68] suggested family polymorphism as a solution to the problem. His running example demonstrates how virtual classes (or, more precisely, virtual patterns) in gbeta [67] allow a type-safe and generic abstraction over graphs. (A graph can be seen as a collaboration of two family members “node” and “edge”). Before comparing Ernst's example to an encoding in JavaGI, we first explain the general idea behind virtual classes.

Virtual classes [132], originally introduced in the language Beta [133], are class-valued attributes of objects; that is, virtual classes are accessed relative to an object instance by using late binding, quite similar to virtual methods. (Virtual classes differ from Java's inner classes [95] because the latter are not subject to late binding.) With virtual classes, types may depend on values, or, more specifically, on paths formed from immutable vari-

Figure 8.2 Concepts in C++.
The code uses the syntax as implemented in ConceptGCC [83].

```
// C++
concept EQ<typename T> {
  bool eq(const T& x, const T& y);
}
template<typename T> requires EQ<T> const T* find(const T& x, const list<T>& l) {
  typename list<T>::const_iterator first = l.begin();
  typename list<T>::const_iterator end = l.end();
  for (; first != end; ++first) {
    if (EQ<T>::eq(x, *first)) return &*first;
  }
  return NULL;
}
concept_map EQ<int> {
  bool eq(const int& x, const int& y) { return x == y; }
}
template<typename T> requires EQ<T> concept_map EQ<list<T>> {
  bool eq(const list<T>& l1, const list<T>& l2) {
    typename list<T>::const_iterator first1 = l1.begin();
    typename list<T>::const_iterator first2 = l2.begin();
    typename list<T>::const_iterator end1 = l1.end();
    typename list<T>::const_iterator end2 = l2.end();
    for (; first1 != end1 && first2 != end2; ++first1, ++first2) {
      if (! EQ<T>::eq(*first1, *first2)) return false;
    }
    return (first1 == end1 && first2 == end2);
  }
}
```

ables and fields. There exists an extension of Java with a variation of virtual classes [226]. The extension, however, relies on dynamic type checks to ensure soundness. Two formalization [70, 48] demonstrate that such dynamic checks are not necessarily needed to support virtual classes in a type sound manner. A generalization of virtual classes [76] expresses similar semantics by parameterization rather than by nesting. Virtual classes also enable solutions to several software extension and adaptation problems, an aspect that we discuss in Section 8.4.

We now come back to Ernst's graph example used to motivate family polymorphism [68]. Figure 8.3 shows an encoding of this example with JavaGI's multi-headed interfaces. As in the original example, the encoding expresses the relation between the nodes and edges of a graph in a type-safe way that nevertheless allows for reusability. However, JavaGI represents families at the type level, which has several disadvantages compared with Ernst's value-level representation: only a fixed number of distinct families can be defined; and only classes not related by subclassing can form distinct families (e.g., if classes $\mathsf{C}_1, \ldots, \mathsf{C}_n$ belong to some family then $\mathsf{C}'_1, \ldots, \mathsf{C}'_n$ usually belong to the same family in JavaGI if each C'_i is a subclass of C_i). A drawback of the value-level representation is that it complicates the type system a lot. Ernst's solution allows the construction of heterogeneous data structures over families. In general, such data structures are possible in JavaGI but their encoding is quite complex and hardly usable in practice (it relies on the well-known trick to simulate existential types through continuations and rank-2 polymorphism).

Other approaches to family polymorphism include Scala's abstract type members with self-type annotations [168], OCaml's object system [122, 192, 185, 186], variant path types [97], lightweight family polymorphism in the context of Java [194], type parameter members [108], lightweight dependent classes [109], Helm and coworkers' contracts [87], and a generalization of MyType [33, 34] to mutually recursive types [31]. The last approach bears close resemblance to JavaGI's multi-headed interfaces but relies on exact types to prevent unsoundness in the presence of binary methods, whereas JavaGI uses multiple dispatch instead. (Section 8.6 discusses MyType and exact types in more detail.)

8.4 Software Extension, Adaptation, and Integration

A lot of research projects address better support for software extension, adaptation, and integration. This section discusses work most relevant to JavaGI.

The Expression Problem

The expression problem, going back to Reynolds [188, 189] and Cook [52] but popularized under its name by Wadler [235], highlights a key problem in the area of software extensibility: how to extend a given data structure modularly in the dimensions of data *and* operation. Torgersen [227] defined a solution to the expression problem as a "combination of a programming language, an implementation of a Composite structure in that language, and a discipline for extension which allows both new data types and operations to be subsequently added any number of times, without modification of existing source code, without replication of non-trivial code, without risk of unhandled combinations of data and operations." JavaGI's approach to the expression problem, as outlined in Section 2.1.1, fulfills these requirements. Torgersen also evaluated solutions to the expression problem according to their degree of extensibility: "code-level extensibility" requires that existing code can be extended without recompilation, and "object-level extensibility" requires that objects created before introducing an extension remain valid and compatible afterwards. JavaGI provides both kinds of extensibility. An additional requirement [167] is that a solution to the expression problem must typecheck statically and that it must be possible to combine independently developed extensions. JavaGI fulfills both of these requirements (assuming that the independently developed extensions are sufficiently disjoint), although typechecking in JavaGI is not fully modular.

Solutions with Type Classes in Haskell

Lämmel and Ostermann [119] showed how Haskell type classes solve several extensibility, adaptability, and integration problems that have been used to illustrate limitations of object-oriented languages. Their Haskell solutions to the adapter problem [73],

Figure 8.3 Ernst's graph example encoded in JavaGI.

```
// A multi–headed interface for modeling graphs
interface Graph [Node,Edge] {
  receiver Node { boolean touches(Edge e); }
  receiver Edge { void setSource(Node n);  void setTarget(Node n); }
}
// An abstract default implementation of Graph
abstract class AbstractNode {}
abstract class AbstractEdge { AbstractNode source;  AbstractNode target; }
abstract implementation Graph [AbstractNode,AbstractEdge] {
  receiver AbstractNode {
    public boolean touches(AbstractEdge e) {
      return e.source == this || e.target == this;
    }}
  receiver AbstractEdge {
    public void setSource(AbstractNode n) { this.source = n; }
    public void setTarget(AbstractNode n) { this.target = n; }
  }
}
// First implementation of Graph
class Node extends AbstractNode {}
class Edge extends AbstractEdge {}
implementation Graph [Node,Edge] extends Graph[AbstractNode,AbstractEdge]{}
// Second implementation of Graph
class OnOffNode extends AbstractNode {}
class OnOffEdge extends AbstractEdge { boolean enabled = false; }
implementation Graph [OnOffNode,OnOffEdge]
       extends Graph [AbstractNode,AbstractEdge] {
  receiver OnOffNode {
    boolean touches(OnOffEdge e) { return e.enabled && super.touches(e); }
  }
}
// A test class
public class GraphTest {
  static <N,E> void build(N n, E e, boolean b) where N*E implements Graph {
    e.setSource(n);  e.setTarget(n);
    if (b == n.touches(e)) System.out.println("OK");
  }
  public static void main(String[] args) {
    build(new Node(), new Edge(), true);
    build(new OnOffNode(), new OnOffEdge(), false);
    // Fails because "OnOffNode*Edge implements Graph" does not hold
    // build(new OnOffNode(), new Edge(), true)
  }
}
```

the tyranny of the dominant decomposition problem [86, 169], the expression problem [235, 227], and the framework integration problem [136, 144] can be ported to JavaGI easily. Further, their graph example used to demonstrate Haskell's approach to family polymorphism is expressible in JavaGI as well but leads to a different encoding compared with the one presented in Section 8.3. As outlined in Section 8.1, translating their three-parameter type class `Graph g n e` with the functional dependency `g` → `n e` results in a single-headed interface `Graph<n,e>`. The JavaGI encoding in Section 8.3 uses a two-headed interface with explicit implementing types for nodes and edges instead. This approach leads to more flexibility because implementing types behave covariantly with respect to subtyping, whereas type parameters are invariant. On the other hand, the interface `Graph<n,e>` provides an explicit representation of the graph itself, whereas the encoding in Section 8.3 leaves the graph implicit. Lämmel and Ostermann's approach to multiple dispatch differs from that in JavaGI because Haskell does not support dynamic dispatch as already discussed in Section 8.1. (See Section 8.5 for an encoding of multiple dispatch in JavaGI.)

Virtual and Nested Classes

Section 8.3 already discussed virtual classes [132] in general and in the context of family polymorphism [68]. But virtual classes also enable solutions to a number of extensibility problems. Higher-order hierarchies [69] allow programmers to extend, combine, and modify existing class hierarchies. The main features enabling this kind of extensibility are furtherbinding (virtual classes are not overridden but enhanced in subclasses) and virtual superclasses (superclass declarations are subject to late binding). JavaGI's retroactive interface implementations also allow the extension of existing class hierarchies with new functionality. Although changing existing hierarchies is not possible in JavaGI, retroactive interface implementations allow to introduce new superinterfaces for existing classes and interfaces. The combination of extensions is implicit in JavaGI because retroactive interface implementations perform in-place object adaptation, whereas higher-order hierarchies create new copies of existing hierarchies and thus need an explicit combine operator. This copy-based approach prevents extensions from being available for existing class instances, a limitation not shared by JavaGI. Further, adding functionality to existing classes in the style of higher-order hierarchies seems to require a default implementation for the root of the class hierarchy, whereas JavaGI avoids the need for such default implementations by allowing abstract methods in retroactive implementations. Completeness checking for abstract methods requires load-time checks, though. Higher-order hierarchies support the addition of state (i.e., instance variables) to existing classes but JavaGI does not.

Nested inheritance [161] also supports the extension of class hierarchies through nesting and furtherbinding of classes. Unlike virtual classes, nested inheritance treats a nested class as an attribute of its enclosing class. Nested intersection [162] generalizes nested inheritance and enables the composition of class hierarchies by some form of multiple inheritance. As higher-order hierarchies, nested inheritance and nested intersection both follow a copy-based approach and make extensions not available for instances of existing classes. Class sharing [183] adds support for in-place object adaptation to nested intersection: a sharing relation between classes implies that shared classes have the same set of object instances. Each shared class is a distinct view of such an instance, and an explicit operation may change that view. JavaGI does not require an explicit operation to combine different extensions. The extension mechanisms of JavaGI and nested inheritance are quite different: the former uses retroactive interface implementations, the latter inheritance. None of these mechanisms is superior to the other. From a programmers point of view, the additional complexity introduced by JavaGI seems to be lower than that of nested inheritance and its successors: JavaGI's additional features are all driven by a generalization of interfaces, whereas nested inheritance/intersection and class sharing confronts the programmer not only with a generalization of inheritance but also with a complex type language making use of exact types, view-dependent types, prefix types, mask types, and sharing constraints [183].

Collaboration interfaces [143] allow the declaration of types for components as a set of mutually recursive types by treating nested interface as virtual. Moreover, collaboration interfaces provide support for expressing not only the provided but also the required services of a component. While JavaGI addresses to problem of specifying mutually recursive types through multi-headed interfaces, there is no support for expressing required services. Conversely, collaboration interfaces do not take retroactive implementation into account, so it might be worthwhile to investigate how a combination of collaboration interfaces and JavaGI's generalized interfaces would look like. The work on collaboration interfaces also suggested wrapper recycling to avoid object schizophrenia [198, 89] caused by wrappers. Essentially, wrapper recycling ensures that there exists at most one wrapper for each interface/object combination, thus avoiding object schizophrenia between two wrapped objects but not between a wrapped and an unwrapped object. JavaGI deals with object schizophrenia by using special cast operations, **instanceof** tests, and identity comparisons (==). This approach avoids object schizophrenia also between wrapped and unwrapped objects but does not work as soon as a wrapped objects is passed to a method that has not been compiled with the JavaGI compiler.

Virtual classes express dependencies between classes and objects through nesting. Hence, a class may depend at most on one object. Dependent classes [76] replace nesting by parametrization and so allow dependencies between a class and multiple objects. Further, the type parameters of a class are subject to dynamic dispatch, so dependent classes complement multimethods (see Section 8.5) by providing multi-dispatched abstractions.

Advanced Separation of Concerns

Subject-oriented programming [86, 169] and work on multi-dimensional separation of concerns [223] deals with the tyranny of the dominant decomposition problem. This problem arises because most languages support only one fixed decomposition of a system, even though other decompositions might be meaningful and appropriate. Hyper/J [170] provides multi-dimensional separation of concerns for Java. The tool allows an existing application to be decomposed into hyperslices, and it offers the possibility to define new hyperslices from scratch. Hyperslices represent different decompositions of a system and allow developers to view a system from different perspectives. A flexible composition mechanism then creates a full Java class from several hyperslices. As mentioned before, Lämmel and Ostermann use Haskell type classes to emulate some of the functionality of hyperslices [119, Section 2.3]. Their solution also works in JavaGI, so Lämmel and Ostermann's comparison with Hyper/J remains valid in the context of the JavaGI language.

Aspect-oriented programming [115] is another technique for improving separation of concerns. It allows programmers to express crosscutting concerns (called aspects) in an explicit and modular manner. AspectJ [114, 7, 6], an aspect-oriented extension of Java, provides two kinds of crosscutting concerns: dynamic crosscutting supports the definition of additional code to run at certain points in the execution of a program; and static crosscutting affects the static type signature of a program. Inter-type member declarations and the `declare parents` form, two examples for static crosscutting, offer functionality similar to JavaGI's retroactive interface implementations: the former enable the addition of new members to existing classes, whereas the latter allows changes to the inheritance structure of a program by inserting new superinterfaces. There is no notion of dynamic crosscutting in JavaGI. The current implementation of AspectJ relies on compile-time weaving to support inter-type member declarations and the `declare parents` form [6, Chapter 5]. That is, the AspectJ compiler rewrites the byte code of the relevant classes, so it is not possible to modify classes that are not under the control of the compiler (e.g., classes from Java's standard library). In contrast, JavaGI never changes the byte code of existing classes, so it is possible to update arbitrary classes. Moreover, AspectJ's invasive compilation strategy causes changes to be visible to all clients of a class, whereas JavaGI guarantees that modifications do not change the behavior of existing clients.

Module Systems for Java

Keris [246] adds a module system to Java that allows for type-safe addition, refinement, replacement, and specialization of modules without pre-planning. The resulting language provides composition of modules through nesting and infers module dependencies automatically. As in JavaGI, Keris' extensibility mechanism does not require source-code access and preserves the original version of a module being extended. The main difference between Keris and JavaGI is that the former introduces a new language construct (modules) whereas the latter makes an existing construct (interfaces) more powerful.

Inspired by MzScheme's [157] units [72], Jiazzi [138] also enhances Java with module-like constructs to provide better support for component-based development. Unlike JavaGI and Keris, however, Jiazzi does not directly extend the Java language but introduces an external language for specifying package signatures and for defining and linking units. The system supports separate compilation, cyclic linking, and mixins [25], and it allows the modular addition of new features to existing classes. In contrast to JavaGI, Jiazzi requires all extensions of a class to be integrated into one module. Further, Jiazzi does not support dynamic loading of extensions.

Other work on module systems for Java include JavaMod [2], JAM [208, 214], and Component NextGen [197].

Statically Scoped Extension Mechanisms

Classboxes [14, 12, 13] offer scoped refinement of classes. Refining a class either adds a new feature (i.e., method, field, superinterface, constructor, inner class) or redefines an existing one, thereby creating a new version of the class. A scoping mechanism ensures that refinements are only locally visible so that potentially conflicting refinements can coexist inside the same program. In contrast, JavaGI's retroactive interface implementations can only add new methods and superinterfaces to classes, additions of other features and redefinitions are not possible. Further, retroactive interface implementations in JavaGI share a global scope so two implementations of the same interface for the same class lead to a conflict. In the other hand, the compilation strategy for classboxes in Java [13] is not modular because the compiler weaves all refinements of a class into the declaration of the class. The JavaGI compiler, however, supports non-invasive and modular code generation. Furthermore, classboxes do not provide multiple dispatch and advanced typing constructs such as explicit implementing types, multi-headed interfaces, and type conditionals.

Expanders [237] are quite similar to classboxes. They offer statically scoped, retroactive extension of classes with new fields, methods, and superinterfaces. The work on expanders also emphasizes modularity: a class may have multiple, independent extensions at the same time, but in each scope only explicitly opened extensions are visible. Unlike classboxes, however, expanders offer modular typechecking and compilation. JavaGI only offers mostly modular typechecking and fully modular compilation. Different from JavaGI, expanders impose some restrictions on the placement of extension code. For example, consider a class hierarchy contained in compilation unit $\mathcal{U}_1$, an extension of the class hierarchy (either through expanders or through retroactive interface implementations) in $\mathcal{U}_2$, and a refinement of the class hierarchy by standard subclassing in $\mathcal{U}_3$. Now suppose that the extension in $\mathcal{U}_2$ should be augmented to take the refinement in $\mathcal{U}_3$ into account. With expanders their are two options, neither of which is satisfactory: either edit the code in $\mathcal{U}_2$ to make the augmentation globally effective or provide a locally overriding expander in some compilation unit $\mathcal{U}_4$ to make the augmentation only visible from within $\mathcal{U}_4$. In contrast, JavaGI's retroactive implementation definitions enable a globally effective augmentation without touching the code in $\mathcal{U}_2$. Moreover, expanders do not support abstract methods, which may result in unwanted run-time exceptions because a reasonable default implementation of an operation does not always exist [148]. Last not last, expanders do not provide multiple dispatch and JavaGI's advanced typing constructs.

Miscellaneous

Hölzle [89] argued that minor incompatibilities between independently developed components are unavoidable. Further, he discussed several mechanism for dealing with such incompatibilities. JavaGI's retroactive interface implementations is an realization of his type adaptation proposal. Binary component adaptation [110] supports the adaptation and evolution of components in binary form by rewriting component binaries at load-time. In contrast, JavaGI never changes the byte code of existing classes.

Scala [166] supports implicit parameters and methods, which can be used to define implicit conversions called views. A view from type `T` to interface `I` may simulate a retroactive implementation of `I` for `T`. However, unlike JavaGI's multiple dynamic dispatch, view selection in Scala is based on a single static type. Further, the implementation of a view often uses explicit wrappers, which suffer from object schizophrenia [198, 89].

Partial classes in C# 2.0 [63] provide a primitive, code-level modularization tool. The different partial slices of a class (comprising superinterfaces, fields, methods, and other members) are merged by a preprocessing phase of the compiler. Extension methods in C# 3.0 [64] support full separate compilation, but the added methods cannot be virtual, and members other than methods cannot be added.

Smalltalk [81] and Objective-C [4] support the extension of existing classes with new methods. Smalltalk also supports redefinitions of methods. In contrast to JavaGI, Smalltalk is a dynamically typed language and the type language of Objective-C is much weaker than that of JavaGI.

8.5 External Methods and Multiple Dispatch

This section complements the preceding one by discussing work on external methods in combination with multiple dispatch. External methods allow extensions of existing classes by defining methods outside of class definitions. Their common motivation is to supersede the Visitor and the Adapter design patterns [73] and to solve the expression problem [235, 227]. Multiple dispatch denotes the ability to perform dynamic dispatch not only on the receiver but also on the arguments of a method call. This generalization of the traditional object-oriented dispatch mechanism solves the binary-methods problem [29] and improves code modularity and readability by avoiding double dispatch [98] and cascades of **`instanceof`** tests. (Section 8.6 presents alternative solutions to the problem of statically typechecking binary methods.)

The combination of external methods and multiple dispatch is found in languages such as Common Lisp [205], Dylan [199], Cecil [44, 43, 45], as well as in the Java extension MultiJava [49, 50] and its successor Relaxed MultiJava [148]. JavaGI supports multiple dispatch through multi-headed interfaces and explicit implementing types. A standard example [49] for multiple dispatch is to provide an operation for computing the intersection of different kinds of geometric shapes such that the "best" intersection algorithm is automatically chosen based on the run-time type of the two shapes being intersected. Figure 8.4 shows a JavaGI encoding of this example. The two-headed interface `Intersect` defines a multimethod (i.e., a method subject to multiple dispatch) of name `intersect` that dispatches on the receiver `Shape1` and on its first argument `Shape2`. Retroactive implementations of `Intersect` then provide the intersection algorithms for different combinations of shapes. Declaring the signature of a multimethod in an interfaces fixes the dispatch positions for all implementations of the method in advance. The language Tuple [121] shares this restriction with JavaGI, whereas Common Lisp, Dylan, Cecil, and (Relaxed) MultiJava allow different dispatch positions for different implementations.

The main problem in fitting multiple dispatch to a typed object-oriented language is modular typechecking without imposing too many restrictions. Common Lisp and Dy-

Figure 8.4 Multiple dispatch in JavaGI.

```
// A simple hierarchy of geometric shapes
abstract class Shape { ... }
class Rectangle extends Shape { ... }
class Circle extends Shape { ... }
// Declaration of the intersection operation
interface Intersect [Shape1, Shape2] {
  receiver Shape1 { boolean intersect(Shape2 that); }
}
// Implementations of different intersection algorithms
implementation Intersect [Shape, Shape] {
  receiver Shape {
    boolean intersect(Shape that) { /* standard algorithm */ }
  }
}
implementation Intersect [Rectangle, Rectangle] {
  receiver Rectangle {
    boolean intersect(Rectangle that) { /* algorithm for rectangles */ }
  }
}
// more implementations omitted
```

lan are both dynamically typed, so the problem does not occur in these languages. Cecil requires the whole program to perform typechecking. The core language Dubious [149] investigates what restrictions are necessary to support modular typechecking. The outcome of this investigation are three different systems: System M supports fully modular typechecking at the price of losing expressiveness; System E maximizes expressiveness but requires some regional typechecking and a simple link-time check; System ME lets programmers decide on a case-by-case basis whether to use System M or System E. All three systems are type sound; that is, neither "message-not-understood" nor "message-ambiguous" errors can occur at run time. The core calculus of JavaGI defined in Chapter 3 also enjoys type soundness in this sense.

MultiJava's design [49, 50] is based on System M. Hence, it supports fully modular typechecking at the price of several restrictions. JavaGI's initial design [240] (see also Section 8.11) followed MultiJava and reformulated the restrictions as follows: (i) retroactively defined methods must not be abstract; and (ii) if an implementation of interface `I` in compilation unit $\mathcal{U}$ retroactively adds a method to class `C`, then $\mathcal{U}$ must contain either `C`'s definition or any implementation of `I` for a superclass of `C`. These two restrictions allow modular typechecking but also severely limit expressiveness. Thus, JavaGI takes the same approach as Relaxed MultiJava [148] and defers some checks to link time. These link-time checks allow JavaGI to drop the two restrictions just mentioned. Relaxed MultiJava and JavaGI support fully modular code generation.

Dylan, Cecil, (Relaxed) MultiJava, and JavaGI all rely on symmetric multiple dispatch; that is, they treat all dispatch arguments identically. Only few approaches (e.g., Common Lisp) use asymmetric dispatch, which avoids ambiguities by preferring certain dispatch arguments when searching for a method implementation.

Half & Half [10] is a Java extension supporting asymmetric multiple dispatch but no external methods. Instead, it offers the ability to add new superinterfaces to existing classes, thereby relying on structural conformance of the existing class with the new superinterface. JavaGI's retroactive interface implementations are more powerful because they allow to compensate for structural non-conformance by providing missing methods externally.

Nice [21] is a Java-like language supporting external methods and symmetric multiple dispatch. It has its roots in ML$_{\leq}$ [23], an explicitly typed extension of ML [150] with subtyping and higher-order multimethods. Nice also provides some form of retroactive interface implementation. Different from JavaGI, these retroactive implementations are not available for ordinary interfaces but only for so-called abstract interfaces. Unlike ordinary interfaces, abstract interfaces are not types but can be used to constrain type parameters [20], in quite similar ways as JavaGI's implementation constraints.

Pirkelbauer and colleagues [178] study external methods and multiple dispatch in the context of C++. Their extension deals with the additional ambiguities arising through multiple inheritance by employing link-time checks. Allen and coworkers [1] consider a formalization of external methods and multiple dispatch in the context of Fortress [217]. Their formalization includes multiple inheritance and defines modular restrictions that rule out ambiguous or undefined method calls.

An empirical study [154] analyzed the use of multiple dispatch in practice and suggested that "Java programs would have scope to use more multiple dispatch were it supported in the language." Predicate dispatch [71, 146, 147] is more expressive than multiple dispatch because each method may specify a predicate that defines the conditions under which the method should be invoked. JavaGI does not support predicate dispatch.

8.6 Binary Methods

A binary method [29] is a method requiring the receiver type and some of the argument types to coincide. Static typechecking of binary methods is challenging because subtyping treats methods arguments contravariantly, whereas binary methods require arguments to vary covariantly.

PolyTOIL [34] is a statically typed object-oriented languages supporting a keyword `MyType`, which represents the type of **`this`**. Using `MyType` as the type of certain method arguments provides faithful signatures for binary methods. To avoid the aforementioned tension between contra- and covariance, PolyTOIL separates subtyping from subclassing. Instead of subtyping, subclassing only induces a matching relation between types. Matching, written `<#`, is weaker than subtyping (i.e., relates more types) and can be used to constrain type parameters of classes and methods, leading to the notion of match-bounded polymorphism.

Although matching and subtyping are different relations, they are still quite similar. To avoid confusion between them, the successor $\mathcal{LOOM}$ [32] of PolyTOIL completely eliminates subtyping in favor of matching. To address some loss of expressiveness, $\mathcal{LOOM}$ introduces hash types. A hash type `#T` denotes the set of all types matching `T`; that is,

`#T` can be seen as an abbreviation for the match-bounded existential type ∃`X<#T.X`.

The language LOOJ [30] integrates the ideas of `MyType` into Java. It introduces `ThisClass` to capture the class type of **`this`** and `ThisType` to denote the public interface type of the definition where `ThisType` occurs. LOOJ ensures type safety in the presence of `ThisClass` and `ThisType` through exact types that essentially prohibit subtype polymorphism. Self-type constructors [193] integrates the `ThisClass` construct of LOOJ with generics, so that `ThisClass` inside a generic class no longer denotes a specific instantiation of the class but takes type parameters on its own.

JavaGI provides explicit implementing types to express the signatures of binary methods in interfaces. To regain type soundness, JavaGI prevents invocations of binary methods on receivers whose static types are interface types and uses multiple dispatch to select the most specific implementation of a binary method dynamically. JavaGI also supports retroactive and constrained interface implementations, as well as static interface methods; these features have no correspondence in PolyTOIL, $\mathcal{LOOM}$, or LOOJ.

Eiffel's **`like`** `Current` construct [140] also allows to express signatures of binary methods. Unfortunately, the construct renders the type system unsound [51]. Attempts to recover type soundness include a global system validity check [140] and a complex condition preventing "polymorphic catcalls" [141].

Scala [166] supports singleton types of the form **`this`**`.type`, which are similar to (covariant uses of) `MyType` [168]. Moreover, Scala's self-type annotations allow programmers to state the type of **`this`** explicitly.

8.7 Type Conditionals

JavaGI's facility to provide methods and retroactive implementations of interfaces depending on the validity of type conditions is related to cJ [91], a Java extension that provides type-conditional declarations of fields, methods, superclasses, and superinterfaces. JavaGI does not support type-conditional fields and superclasses. A type condition in cJ is any subtype constraint on generic parameters, whereas JavaGI additionally allows implementation constraints. The language cJ concentrates on type conditionals: it does not support JavaGI's retroactive implementations, multiple dispatch, explicit implementing types, multi-headed interfaces, and static interface methods.

Constraint-based polymorphism [131, 130] for Cecil [45] offers the possibility to define classes, subtype relationships, methods, and fields depending on certain type constraints. These constraints, expressed in `where`-clauses as in JavaGI, come in two forms: `isa`-constraints specify nominal subtype conditions, whereas `method`-constraints express structural subtype conditions. The system also supports external methods and multiple dispatch but does not provide an interface concept in the sense of JavaGI. The type system for constraint-based polymorphism is sound and there exists a sound and terminating but incomplete typechecking algorithm. In contrast, JavaGI's typechecking algorithm is sound, terminating, and complete, albeit for a weaker constraint system. Further, JavaGI is a conservative extension of the class-based language Java, whereas Cecil is an object-based language.

An extension of C# with type-equation constraints enables cast-free programs for object-oriented encodings of generalized algebraic datatypes [111]. Further, it allows to specify generic methods that only apply to certain instantiations of the enclosing class. While JavaGI does not support type equations in their general form, it is nevertheless possible to encode several of the examples written with type equations (e.g., the "typed expressions in typed environments" and the "list flatten" examples [111]) using JavaGI's type conditionals. There exist a generalization of type-equation constraints to arbitrary subtype constraints that also considers variance for generic types [66].

As discussed in Section 8.4, Scala's views [166] can emulate some functionality of retroactive interface implementations. This functionality includes type-conditional interface implementations because views may place type conditions on their arguments.

Constrained types [163] in X10 [196] are a form of dependent types [177, Chapter 2] that allow to enforce conditions on the immutable state of a class. This sort of condition is quite different from JavaGI's type conditions, which express subtype and implementation constraints on type variables.

The idea of separate `where`-clauses to specify type conditionals goes back to the programming language CLU [127, 129]. CLU and its successor Theta [128, 55] support structural constraints on type parameters, even if the parameters are defined in an enclosing scope.

8.8 Traits

Originally, a trait is a stateless collection of methods implementing a particular concern, but separate from a class [59, 58]. Traits can be composed in various ways and have to be included in a class to attach their methods to objects of that class. Recent work also addresses stateful traits [15] and integrates traits into statically typed languages [203, 126, 22]. The main difference between traits and generalized interfaces in JavaGI is the intention behind these two concepts: traits are meant as units of reuse whereas interfaces describe signatures of objects.

Scala [166] combines these two intentions. As interface in Java, traits specify signatures of objects but they may also contain fields and default implementations of certain methods. Modular mixin composition [168] integrates traits into classes. Unlike JavaGI, however, Scala does not support retroactive implementations of traits. Traits in Fortress [217] are like Java interfaces but they may contain code, properties, and allow parameterization over values.

Mohnen [151] suggested interfaces with default implementations for Java. JavaGI can encode such default implementations with abstract implementations and implementation inheritance (see Section 2.1.5).

8.9 Advanced Subtyping Mechanisms

This section discusses some advanced subtyping mechanisms related to JavaGI.

Most object-oriented languages (e.g., Java, C#, Scala, and also JavaGI) rely on nominal subtyping; that is, explicit declarations establish the subtyping relation. In contrast, structural subtyping considers type `T` a subtype of another type `U` if `T` matches `U` structurally; that is, `T` supports at least the features provided by `U`. Structural subtyping enables retroactive interface implementation if the names and signatures of the methods of a class happen to match the requirements of an interface. In practice, however, situations where a class does not implement an interface nominally but nevertheless provides all the interface's methods with exactly matching signatures seem to be quite rare. More common appear scenarios where a class provides the methods of an interface with slight mismatches with respect to method names or argument ordering [89]. Structural subtyping alone does not help in such situations but JavaGI's retroactive interface implementations do. Nevertheless, structural subtyping provides benefits to other problem areas [135]. Ostermann [171] provided a detailed comparison between nominal and structural subtyping.

Compound types [35] integrate a form of intersection types [176, Section 15.7] into Java. They are subject to structural subtyping, but other constructs of the language still rely on nominal subtyping. Läufer and coworkers considered structural conformance to interface types in the context of Java [120]. In their work, a type is a subtype of some interface if it matches the interface structurally. The authors also discussed a renaming mechanism for methods to make structural conformance more widely applicable. Whiteoak [79] is an extension of Java that introduces designated `struct` types. These types are subject to structural subtyping and support flexible composition operations. Unity [134] is a language design that integrates nominal and structural subtyping, and also provides external methods.

The programming language Sather [221, 207] is based on nominal subtyping but allows for some of the flexibility of structural subtyping by supporting not only declarations of sub- but also of supertypes. Further, Sather decouples inheritance from subtyping [53]. The calculus $\mathrm{FJ}_{<:}$ [171] combines Sather's subtyping mechanism with compound types [35] to arrive at a non-transitive subtyping relation. Pedersen [172] proposed specialization (i.e., the possibility to introduce new superclasses) as a technique to improve reusability of classes.

8.10 Subtyping and Decidability

Chapter 5 discussed two extensions of JavaGI's type system that both render subtyping undecidable. This section reviews work related to this topic.

Kennedy and Pierce [113] investigated undecidability of subtyping under multiple instantiation inheritance and declaration-site variance. They proved that the general case is undecidable and presented three decidable fragments. The undecidability proof for subtyping in IIT given in Section 5.1 is similar to theirs, although undecidability has different causes: Kennedy and Pierce's system is undecidable because of contravariant generic types, expansive class tables, and multiple instantiation inheritance, whereas undecidability of the system in Section 5.1 is due to implementation definitions for interface types, which cause cyclic interface and multiple instantiation subtyping.

Pierce [175] proved undecidability of subtyping in $F_{\leq}$ [40] by a chain of reductions from the halting problem for two-counter Turing machines. An intermediate link in this chain is the subtyping relation of $F^D_\leq$, which is also undecidable. The undecidability proof for subtyping in EXuplo from Section 5.2 works by reduction from $F^D_\leq$ and is inspired by a reduction given by Ghelli and Pierce [77], who studied bounded existential types in the context of $F_\leq$ and showed undecidability of subtyping. Crucial to the undecidability proof of $F^D_\leq$ is rule D-ALL-NEG (Figure 5.4 on page 120): it extends the typing context and essentially swaps the sides of a subtyping judgment. In EXuplo, rule EXUPLO-OPEN and rule EXUPLO-ABSTRACT (Figure 5.3 on page 118) together with lower bounds on type variables play a similar role. In an object-oriented setting, it is possible to define a restricted variant of $F_\leq$ by separating subtyping and subclassing such that quantified type variables are subject to subclassing bounds only [46]. The resulting subtyping and expression typing relations are decidable.

WildFJ [228] is a model for Java wildcards based on bounded existential types. There exists no type soundness proof for WildFJ. The calculus ∃J [38] is similar to WildFJ but comes with a proof of type soundness. However, ∃J is not a full model for Java wildcards because it does not support lower bounds for type variables. TameFJ [37] is a type-sound calculus supporting all essential features of Java wildcards. WildFJ's and TameFJ's subtyping rules are similar to the ones of EXuplo defined in Section 5.2, so we conjecture that subtyping in WildFJ and TameFJ is also undecidable. The rule XS-ENV of TameFJ is roughly equivalent to the rules EXUPLO-OPEN and EXUPLO-ABSTRACT (Figure 5.3 on page 118) of EXuplo. Other calculi [36] use existential types to yield a unified model of subtyping in Java.

Decidability of subtyping for Java wildcards is still an open question [137]. One step in the right direction might be the work of Plümicke, who solved the problem of finding a substitution φ such that $\varphi T \leq \varphi U$ for Java types T, U with wildcards [180, 181]. The undecidability result for EXuplo does not imply undecidability for Java subtyping with wildcards. The proof of this claim would require a translation from subtyping derivations in EXuplo to subtyping derivations in Java with wildcards, which is not addressed in this dissertation. In general, existentials in EXuplo are strictly more powerful than Java wildcards. For example, the existential $\exists X.C\texttt{<}X, X\texttt{>}$ cannot be encoded as the wildcard type C<?, ?> because the two occurrences of ? may denote two distinct types.

Scala [166] supports bounded existential types to provide better interoperability with Java libraries using wildcards and to address the avoidance problem [177, Chapter 8]. The subtyping rules for Scala's existentials (Section 3.2.10 and Section 3.5.2 of the specification [166]) are very similar to the ones for EXuplo. This raises the question whether Scala's subtyping relation with existentials is decidable.

8.11 JavaGI's Initial Design

An article [240] at ECOOP 2007 presented the initial design of JavaGI. The language introduced in that article included full-blown bounded existential types and omitted many restrictions, thus rendering subtyping and typechecking undecidable. The undecidability results were first established in two papers [241, 243] at FTfJP 2008 and APLAS 2009; Chapter 5 of this dissertation builds on and slightly extends the APLAS paper. Apart from decidability issues, the ECOOP paper did not define a dynamic semantics, so there was no implementation and the type soundness proof was not worked out. Furthermore, the translation scheme sketched in the ECOOP paper was too limiting because it did not support dynamic loading of implementation definitions and required severe restrictions on the locations of retroactive implementation definitions (see Section 8.5). In contrast, JavaGI as presented in this dissertation is fully implemented and integrated with Java, it supports dynamic loading and implementation inheritance, and it places no restrictions on the locations of implementation definitions. It comes with a formalization that enjoys type soundness, decidable subtyping and typechecking, as well as deterministic evaluation. Further, there exists a type- and behavior-preserving translation that demonstrates how to translate the JavaGI constructs to plain Java.

9 Conclusion

JavaGI is a conservative extension of Java based on the notion of generalized interfaces. It offers a flexible approach to adapting, extending, and integrating existing software components, even in binary form. Further, JavaGI supersedes tedious applications of design patterns and offers save and convenient alternatives to unsafe cast operations, run-time exceptions, and code duplication. The generalization of interfaces serves as the unifying notion that leads to a coherent and elegant language design. Thus, JavaGI smoothly integrates features only loosely connected in other language proposals.

Chapter Outline. The last chapter of the dissertation summarizes the content of the preceding chapters (Section 9.1) and outlines directions for future work (Section 9.2).

9.1 Summary

The introduction of the dissertation set the scene by motivating why component-based software development in statically-typed, object-oriented programming languages is beneficial to reducing development costs and raising software quality. It also pointed out a particular problem with software components in object-oriented languages: how to implement the interfaces required by one component with classes provided by another, independently developed component?

The introduction also established the main goal of this dissertation: the design, formalization, and implementation of a programming language that enables clean solutions to software extension, adaptation and integration problems, and that raises the expressiveness of the type system to prevent developers from resorting to tedious coding patterns, unsafe cast operations, run-time exceptions, and code duplication. The new language should be a conservative extension of Java to reuse as much infrastructure (libraries, tools, knowledge of developers, etc.) as possible. Further, the design of the language should be based on a generalization of Java's interfaces to subsume different concerns under a single concept.

Design

Chapter 2 provided a gentle introduction to the design of this new language JavaGI. It first explained the concept of retroactive interface implementations. This feature enables developers to provide implementations of interfaces that are not attached to class definitions. Thus, retroactive interface implementations solve the aforementioned problem of connecting two independently developed components.

Chapter 2 continued by stepwise unfolding more features of JavaGI. The examples used to introduce the features demonstrated how

- retroactive interface implementations enable non-invasive and in-place object adaptation and thus eliminate the need for the Adapter pattern [73] (Sections 2.1.1 and 2.1.8);
- retroactive interface implementations enable extensibility in the data and operation dimension and thus supersede the Visitor pattern [73] and solve a restricted version of the expression problem [235, 227] (Sections 2.1.1 and 2.1.8, but see also Section 8.4);
- explicit implementing types allow the specification of signatures for binary methods without resorting to awkward uses of F-bounded polymorphism and Java wildcards (Sections 2.1.2 and 2.1.8);
- explicit implementing types enable multiple dispatch and thus avoid clumsy coding patterns (Sections 2.1.2, 2.1.7, and 2.1.8, but see also Section 8.5);
- type conditionals prevent code duplication and unsafe run-time casts (Sections 2.1.3 and 2.1.8);
- static interface methods abstract over class constructors and thus supersede the Factory pattern [73] (Sections 2.1.4 and 2.1.8);
- inheritance for retroactive interface implementations allows to provide (partial) default implementations of interfaces and thus avoids code duplication without restricting the inheritance hierarchy (Section 2.1.5);
- multi-headed interfaces allow to express relationships between multiple types in a static way and thus eliminate certain run-time casts (Sections 2.1.7 and 2.1.8, but see also Section 8.3);
- dynamic loading of retroactive implementation definitions provides seamless integration with Java's approach of loading all classes and interfaces dynamically (Section 2.1.6).

Furthermore, Chapter 2 presented JavaGI's design principles of conservativeness, extensibility, dynamicity, type safety, modularity, and transparency. It also gave a high-level overview on the principles of typechecking and executing JavaGI programs. The overview included the specification of well-formedness criteria that ensure successful and unambiguous dynamic method lookup without depending on run-time type arguments.

The JavaGI compiler checks these criteria globally, and JavaGI's run-time system repeats the checks whenever a new class or implementation is loaded. Thus, JavaGI gives up fully modular typechecking in favor of flexibility: checking the criteria modularly and at compile time only would require severe restrictions on the placement of retroactive implementations, and it would make support for dynamic loading of implementation definitions very hard to achieve (see also Section 8.11).

Formalization

The formalization of JavaGI ranged over three chapters. Chapter 3 introduced CoreGI, a calculus in the spirit of Featherweight Generic Java [96]. CoreGI includes the essential aspects of generalized interfaces in the full JavaGI language, with the exception that interfaces cannot be used as implementing types of retroactive implementations.

The formalization of CoreGI in Chapter 3 started with the definition of a dynamic semantics and a declarative specification of CoreGI's type system. The type system also includes the global well-formedness criteria mentioned at the end of the preceding section, except for the "no implementation chains" and the "completeness" criteria (Section 2.3.4), which are only relevant if interfaces are allowed as implementing types and if methods of retroactive implementations may be static, respectively. Next, Chapter 3 verified that CoreGI's type system is sound and that its evaluation relation is deterministic. Finally, the chapter presented constraint entailment, subtyping, and typechecking algorithms for CoreGI and proved them equivalent to their declarative specification.

Chapter 4 formalized the compilation from JavaGI into standard Java constructs. The source language of the formal translation is CoreGI$^\flat$, a subset of CoreGI without support for generics and some other, minor features. The target language is iFJ, an extension of Featherweight Java [96] with interfaces, let-expressions, and a primitive operation for dictionary lookup. The chapter defined a type-directed translation from CoreGI$^\flat$ to iFJ and verified that the translation preserves the static and the dynamic semantics of CoreGI$^\flat$. It also proved that the type systems of iFJ and CoreGI$^\flat$ are both sound, and that the evaluation relation of CoreGI$^\flat$ is deterministic.

Chapter 5 investigated two extensions of JavaGI's subtyping relation. The first extension provides support for interfaces as implementing types of retroactive implementations. In its most general form, subtyping is undecidable in this setting. However, there exist several restrictions that ensure decidability. The full JavaGI language uses one of these restrictions (Restriction 5.9) as well-formedness criterion "no implementation chains" (Section 2.3.4) to support interfaces as implementing types without rendering the subtyping relation undecidable.

The second extension looked at bounded existential types with lower and upper bounds. Existential types of this form are attractive because they subsume and generalize several other features of JavaGI. Unfortunately, subtyping is undecidable for existentials with lower and upper bounds. Although there exist two decidable fragments, JavaGI does not support existentials because both fragments are not powerful enough to be of practical value. The undecidability result for bounded existential types with lower and upper bounds also sheds light on the decidability of subtyping in Scala [166] and of subtyping for Java wildcards [229, 37] (see Section 8.10).

Implementation

Chapter 6 discussed the implementation of a compiler and a run-time system for JavaGI. The implementation demonstrates that the typechecking algorithm for CoreGI and the translation from CoreGI$^\flat$ to iFJ scales to the full language without major problems. The JavaGI compiler is based on the Eclipse Compiler for Java [62], so it supports the full Java 1.5 language and all JavaGI-specific features presented in this dissertation. The typechecking strategy of the compiler can be described as "mostly modular": although the compiler typechecks each compilation unit in isolation, it needs one global pass at the end to verify the well-formedness criteria mentioned earlier. Code generation, however, works in a modular way. JavaGI's run-time system has the following responsibilities: it maintains the pool of available implementation definitions, it re-checks the well-formedness criteria if necessary, it loads new implementation definitions at run time, it performs dynamic dispatch on retroactively implemented methods, and it carries out certain cast operations, **instanceof** tests, and identity comparisons.

Chapter 7 reported on practical experience with JavaGI and its implementation. It started by describing three real world case studies:

- The first case study implemented a framework for evaluating XPath [47] expressions and adapted two existing XML libraries written in Java to the framework. It demonstrated that retroactive interface implementations allow for a straightforward and elegant adaptation of the XML libraries. Compared with an existing adaptation of the same libraries to a corresponding framework in plain Java, the JavaGI solution requires no cast operations at all, whereas the Java solution contains 75 casts.

- The second case study implemented a framework for developing web applications and used this framework to provide a workshop registration application. The framework is based on the Java servlet technology [215] and on ideas from the WASH framework [224]. The case study demonstrated the usefulness of retroactive interface implementations and static interface methods. Moreover, it showed that JavaGI's support for dynamic loading of implementation definitions is essential when working within a servlet container such as Tomcat [3].

- The third case study refactored the Java Collection Framework so that invocations of destructive operations on unmodifiable collections lead to compile-time instead of run-time errors. Inspired by work on the Java extension cJ [91], the case study implemented this functionality using JavaGI's form of type conditionals.

The chapter continued by presenting benchmark data suggesting that the JavaGI compiler generates code with good performance. Plain Java code compiled with the JavaGI compiler runs as fast as the same code compiled with a regular Java compiler, but there is a performance penalty for JavaGI-specific features.

Related Work

Chapter 8 discussed research related to JavaGI. The discussion covered a broad range: it compared JavaGI's generalized interface concept with Haskell's type class mechanism;

it looked at various approaches to generic programming and family polymorphism; it evaluated JavaGI according to criteria established for solutions to the expression problem; it considered different solutions to software extension, adaptation, and integration problems; it reviewed proposals providing external methods in combination with multiple dispatch; it discussed work on binary methods, type conditionals, traits, and advanced subtyping mechanisms; it studied subtyping and decidability issues related to the undecidability results of Chapter 5; and it compared the current design of JavaGI with an earlier version.

9.2 Future Work

Future work addresses support for associated types and proper reflection facilities. Moreover, the following directions may be worthwhile to pursue.

Implementation Families

Currently, all retroactive implementation definitions share a global scope. This approach may lead to problems composing separately developed parts of an application because it impedes independent extensibility [219]. For example, different parts of an application may need to provide different implementations of the same interface with identical implementing types. Unfortunately, JavaGI prevents such overlapping implementations to rule out ambiguities in dynamic method lookup. *Implementation families* are a possible solution to this problem. The idea is to partition the set of implementation definitions into disjoint families such that JavaGI's global well-formedness criteria must hold only within each family and not for all implementation definitions. To avoid run-time ambiguities, an invocation of a retroactively implemented method must specify, either explicitly or implicitly, the family from which to resolve the implementation.

Better Support for Interfaces as Implementing Types

Currently, JavaGI prevents retroactive implementations of interfaces that are used as implementing types in other implementations (criterion "no implementation chains" in Section 2.3.4, Restriction 5.9 in Section 5.1.3). Again, this restriction endangers independent extensibility. It is an open question how to lift the restriction in a satisfactory manner. On the theoretical side, the restriction is important to ensure decidability of constraint entailment and subtyping (see Section 5.1). On the practical side, the restriction allows for efficient run-time lookup of retroactive implementations.

Retroactive Interface Implementations for the Java Virtual Machine

Currently, the JavaGI compiler generates code that is executable on an unmodified Java Virtual Machine (JVM [125]). It would be worthwhile to explore what modifications to the JVM are necessary to support retroactive interface implementations directly. Possible benefits of such an extension include better performance and improved compatibility with libraries compiled by an ordinary Java compiler. (Libraries compiled by an ordinary Java compiler are not aware of wrappers and thus may exhibit unexpected behavior under the current compilation approach.)

Generalized Interfaces for C#

Currently, generalized interfaces are only available as an extension of the language Java. What about generalized interfaces for other object-oriented languages such as C#? Although Java and C# are quite similar, there are still enough differences that would make such an undertaking interesting. For example, Java has a type-erasure semantics; that is, type arguments of generic classes are not available at run time. In contrast, C# provides run-time types. As discussed in Section 6.1.3, Java's type-erasure semantics influenced the design of JavaGI at several places, so the availability of run-time types may change some of these design decisions.

Appendix

A

Syntax of JavaGI

Figures A.1 and A.2 define the syntax of JavaGI, expressed as an extension to the syntax of Java as defined in the first 17 chapters of *The Java Language Specification* (JLS) [82]. The syntax definition shows nonterminal symbols in *italic* font and terminal symbols in `fixed width` font. The subscript "*opt*" indicates an optional item. Alternative productions for the same nonterminal are separated by the symbol "|". A nonterminal already defined in the JLS carries a superscript annotation specifying the JLS section of its original definition. A JLS section annotation in parenthesis indicates that the syntax of JavaGI redefines the annotated nonterminal. An ellipsis "..." represents unmodified JLS productions. The figure highlights changes to other productions from the JLS.

There are three new keywords in JavaGI: `implementation`, `receiver`, and `where`. The nonterminal `as` in the production for *ImplName* in Figure A.1 is not parsed as a keyword but as a regular identifier.

Figure A.1 Syntax of JavaGI (1/2).

Implementations

TypeDeclaration$^{(\S\,7.6)}$: . . . | *ImplDeclaration*

ImplDeclaration : *ImplModifier*$_{opt}$ `implementation` *TypeParameters*$^{\S\,8.1.2}_{opt}$
InterfaceType$^{\S\,4.3}$ `[` *ClassOrInterfaceTypeList* `]`
ImplName$_{opt}$ *ExtendsImpl*$_{opt}$ *ConstraintClause*$_{opt}$
`{` *ImplBodyDeclarations*$_{opt}$ `}`

ImplModifier : one of `final` `abstract`

ClassOrInterfaceTypeList : non-empty list of *ClassOrInterfaceType*$^{\S\,4.3}$ separated by `,`

ImplName : `as` *Identifier*$^{\S\,3.8}$

ExtendsImpl : `extends` *ImplReference*

ImplReference : *InterfaceType*$^{\S\,4.3}$ `[` *ClassOrInterfaceTypeList* `]`
| *TypeName*$^{\S\,4.3}$

ConstraintClause : `where` *ConstraintList*

ConstraintList : non-empty list of *Constraint* separated by `,`

Constraint : *ReferenceType*$^{\S\,4.3}$ *TypeBound*$^{(\S\,4.4)}$
| *ImplTypeList* `implements` *InterfaceType*$^{\S\,4.3}$

ImplTypeList : non-empty list of *ReferenceType*$^{\S\,4.3}$ separated by `*`

ImplBodyDeclarations : possibly empty list of *ImplBodyDeclaration*

ImplBodyDeclaration : *MethodDeclaration*$^{\S\,8.4}$ | *ReceiverImpl*

ReceiverImpl : `receiver` *ClassOrInterfaceType*$^{\S\,4.3}$
`{` *MethodDeclarations*$_{opt}$ `}`

MethodDeclarations : possibly empty list of *MethodDeclaration*$^{\S\,8.4}$

Interfaces

NormalInterfaceDeclaration$^{(\S\,9.1)}$: *InterfaceModifiers*$^{\S\,9.1.1}_{opt}$ `interface` *Identifier*$^{\S\,3.8}$
TypeParameters$^{\S\,8.1.2}_{opt}$ *ImplParameters*$_{opt}$
ExtendsInterfaces$^{\S\,9.1.3}_{opt}$ *ConstraintClause*$_{opt}$
InterfaceBody$^{\S\,9.1.4}$

ImplParameters : `[` *IdentifierList* *ConstraintClause*$_{opt}$ `]`

IdentifierList : non-empty list of *Identifier*$^{\S\,3.8}$ separated by `,`

InterfaceMemberDeclaration$^{(\S\,9.1.4)}$: . . . | *ReceiverDeclaration*

ReceiverDeclaration : `receiver` *Identifier*$^{\S\,3.8}$ `{` *AbstractMethodDeclarations*$^{\S\,9.4}_{opt}$ `}`

Classes

NormalClassDeclaration$^{(\S\,8.1)}$: *ClassModifiers*$^{\S\,8.1.1}_{opt}$ `class` *Identifier*$^{\S\,3.8}$
TypeParameters$^{\S\,8.1.2}_{opt}$ *Super*$^{\S\,8.1.4}_{opt}$ *Interfaces*$^{\S\,8.1.5}_{opt}$
ConstraintClause$_{opt}$ *ClassBody*$^{\S\,8.1.6}$

Figure A.2 Syntax of JavaGI (2/2).

Methods		
$\mathit{MethodHeader}^{(\S\,8.4)}$	:	$\mathit{MethodModifiers}^{\S\,8.4.3}_{opt}$ $\mathit{TypeParameters}^{\S\,8.1.2}_{opt}$ $\mathit{ResultType}^{\S\,8.4}$ $\mathit{MethodDeclarator}^{\S\,8.4}$ $\mathit{Throws}^{\S\,8.4.6}$ $\mathit{ConstraintClause}_{opt}$
$\mathit{AbstractMethodDeclaration}^{(\S\,9.4)}$	:	$\mathit{AbstractMethodModifiers}^{\S\,9.4}_{opt}$ $\mathit{TypeParameters}^{\S\,8.1.2}_{opt}$ $\mathit{ResultType}^{\S\,8.4}$ $\mathit{MethodDeclarator}^{\S\,8.4}$ $\mathit{Throws}^{\S\,8.4.6}_{opt}$ $\mathit{ConstraintClause}_{opt}$
$\mathit{AbstractMethodModifier}^{(\S\,9.4)}$	:	one of $\mathit{Annotation}^{\S\,9.7}$ `public` `abstract` `static`
Type bounds		
$\mathit{TypeBound}^{(\S\,4.4)}$	:	. . . | `implements` $\mathit{InterfaceType}^{\S\,4.3}$ $\mathit{AdditionalBoundList}^{\S\,4.4}_{opt}$
$\mathit{WildcardBounds}^{(\S\,4.5.1)}$	:	. . . | `implements` $\mathit{InterfaceType}^{\S\,4.3}$
Expressions		
$\mathit{MethodInvocation}^{(\S\,15.12)}$	:	. . .
	|	$\mathit{MethodName}^{\S\,6.5}$ $\mathit{InterfaceSpecifier}$ `(` $\mathit{ArgumentList}^{\S\,15.9}_{opt}$ `)`
	|	$\mathit{Primary}^{\S\,15.8}$ `.` $\mathit{NonWildTypeArguments}^{\S\,8.8.7.1}_{opt}$ $\mathit{Identifier}^{\S\,3.8}$ $\mathit{InterfaceSpecifier}$ `(` $\mathit{ArgumentList}^{\S\,15.9}_{opt}$ `)`
	|	$\mathit{InterfaceType}^{\S\,4.3}$ `[` $\mathit{ClassOrInterfaceTypeList}$ `].` $\mathit{NonWildTypeArguments}^{\S\,8.8.7.1}_{opt}$ $\mathit{Identifier}^{\S\,3.8}$ `(` $\mathit{ArgumentList}^{\S\,15.9}_{opt}$ `)`
$\mathit{InterfaceSpecifier}$	:	`::` $\mathit{TypeName}^{\S\,6.5}$

B

Formal Details of Chapter 3

B.1 Equivalence of Declarative and Quasi-Algorithmic Entailment and Subtyping

This section proves Theorems 3.11 and 3.12, which state soundness and completeness, respectively, between the declarative and the quasi-algorithmic formulation of constraint entailment and subtyping. We make the global assumption that the underlying program *prog* is well-formed; that is, $\vdash \mathit{prog}\ \mathsf{ok}$.

B.1.1 Proof of Theorem 3.11

Theorem 3.11 states that quasi-algorithmic constraint entailment and subtyping is sound with respect to the declarative formulation.

Lemma B.1.1 (Transitivity of sup). *If* $\mathcal{R}_3 \in \mathsf{sup}(\mathcal{R}_2)$ *and* $\mathcal{R}_2 \in \mathsf{sup}(\mathcal{R}_1)$ *then* $\mathcal{R}_3 \in \mathsf{sup}(\mathcal{R}_1)$.

Proof. Straightforward induction on the height of the derivation of $\mathcal{R}_3 \in \mathsf{sup}(\mathcal{R}_2)$. □

Lemma B.1.2. *If* $I{<}\overline{T}{>} \trianglelefteq_{\mathtt{i}} K$*, then* $U\ \textbf{implements}\ K \in \mathsf{sup}(U\ \textbf{implements}\ I{<}\overline{T}{>})$ *for any* U*.*

Proof. By induction on the derivation of $I{<}\overline{T}{>} \trianglelefteq_{\mathtt{i}} K$. If the derivation ends with INH-IFACE-REFL, then $I{<}\overline{T}{>} = K$ and the claim follows trivially.

Now suppose the derivation ends with INH-IFACE-SUPER:

$$\frac{\textbf{interface}\ I{<}\overline{X}{>}\ [Y\ \textbf{where}\ \overline{R}]\ \ldots \qquad R_i = \overline{G}\ \textbf{implements}\ L \qquad [\overline{T/X}]L \trianglelefteq_{\mathtt{i}} K}{I{<}\overline{T}{>} \trianglelefteq_{\mathtt{i}} K}$$

By applying the induction hypothesis (I.H.) to $[\overline{T/X}]L \trianglelefteq_{\mathtt{i}} K$, we get

$$U\ \textbf{implements}\ K \in \mathsf{sup}(U\ \textbf{implements}\ [\overline{T/X}]L)$$

for any type U.

By SUP-REFL, we have $U\,\textbf{implements}\,I<\overline{T}> \in \mathsf{sup}(U\,\textbf{implements}\,I<\overline{T}>)$. Thus, by SUP-STEP also $[U/Y, \overline{T/X}]R_i \in \mathsf{sup}(U\,\textbf{implements}\,I<\overline{T}>)$. With criterion WF-IFACE-2 we have $\overline{G} = Y$ and $Y \notin \mathsf{ftv}(L)$. Thus, $[U/Y, \overline{T/X}]R_i = U\,\textbf{implements}\,[\overline{T/X}]L$. Hence,

$$U\,\textbf{implements}\,[\overline{T/X}]L \in \mathsf{sup}(U\,\textbf{implements}\,I<\overline{T}>)$$

With Lemma B.1.1 we then get $U\,\textbf{implements}\,K \in \mathsf{sup}(U\,\textbf{implements}\,I<\overline{T}>)$ as required. □

Lemma B.1.3. *If $\Delta \Vdash \mathcal{R}$ and $\mathcal{S} \in \mathsf{sup}(\mathcal{R})$ then $\Delta \Vdash \mathcal{S}$.*

Proof. Straightforward induction on the derivation of $\mathcal{S} \in \mathsf{sup}(\mathcal{R})$. □

Proof of Theorem 3.11. The proof is by induction on the combined height of the derivations of $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{R}$, $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$, $\Delta \vdash_{\mathrm{q}}{}' T \leq U$, and $\Delta \vdash_{\mathrm{q}} T \leq U$, which we call $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$, and $\mathcal{D}_4$, respectively. (In general, $\mathcal{D}$ ranges over derivations.)

(i) *Case distinction* on the last rule used in $\mathcal{D}_1$.

- *Case* ENT-Q-ALG-ENV: We then have $S \in \Delta$ and $\mathcal{R} \in \mathsf{sup}(S)$ With ENT-ENV we get $\Delta \Vdash S$. Applying Lemma B.1.3 then yields $\Delta \Vdash \mathcal{R}$.
- *Case* ENT-Q-ALG-IMPL: By appeal to part (ii) of the I.H. and rule ENT-IMPL.
- *Case* ENT-Q-ALG-IFACE: We then have

$$\mathcal{R} = I<\overline{V}>\,\textbf{implements}\,K$$
$$1 \in \mathsf{pol}^{+}(I)$$
$$\mathsf{non\text{-}static}(I)$$
$$I<\overline{V}> \trianglelefteq_{\mathrm{i}} K$$

With Lemma B.1.2 we get

$$I<\overline{V}>\,\textbf{implements}\,K \in \mathsf{sup}(I<\overline{V}>\,\textbf{implements}\,I<\overline{V}>)$$

Because $1 \in \mathsf{pol}^{+}(I)$ and $\mathsf{non\text{-}static}(I)$ we have with ENT-IFACE

$$\Delta \Vdash I<\overline{V}>\,\textbf{implements}\,I<\overline{V}>$$

Then $\Delta \Vdash \mathcal{R}$ by Lemma B.1.3.

End case distinction on the last rule used in $\mathcal{D}_1$.

(ii) *Case distinction* on the last rule used in $\mathcal{D}_2$.

- *Case* ENT-Q-ALG-EXTENDS: Follows by part (iv) of the I.H. and an application of rule ENT-EXTENDS.
- *Case* ENT-Q-ALG-UP: We have $\mathcal{P} = \overline{T}^{n}\,\textbf{implements}\,I<\overline{V}>$ and

$$\frac{\begin{array}{c}(\forall i)\ \Delta \vdash_{\mathrm{q}}{}' T_i \leq U_i \\ (\forall i)\ \text{if } T_i \neq U_i \text{ then } i \in \mathsf{pol}^{-}(I) \qquad \Delta \Vdash_{\mathrm{q}}{}' \overline{U}\,\textbf{implements}\,I<\overline{V}>\end{array}}{\Delta \Vdash_{\mathrm{q}} \overline{T}^{n}\,\textbf{implements}\,I<\overline{V}>} \tag{B.1.1}$$

By part (iii) and (i), we get

$$(\forall i)\ \Delta \vdash T_i \leq U_i \tag{B.1.2}$$
$$\Delta \Vdash \overline{U}^{n}\,\textbf{implements}\,I<\overline{V}>$$

We now show $\Delta \Vdash \overline{T}^{n}\,\textbf{implements}\,I<\overline{V}>$ by an inner induction on the number m of indices i with $T_i \neq U_i$.

- If $m = 0$ then $\overline{T} = \overline{U}$ and the claim follows trivially.
- Assume $m > 0$. Without loss of generality (w.l.o.g.), suppose $T_n \neq U_n$. We get by the inner I.H.

$$\Delta \Vdash \overline{T}^{n-1}\, U_n \,\textbf{implements}\, I{<}\overline{V}{>} \tag{B.1.3}$$

Because $T_n \neq U_n$ we have $n \in \mathsf{pol}^-(I)$ by (B.1.1). With (B.1.2), (B.1.3), and ENT-UP we then get $\Delta \Vdash \overline{T}^n \,\textbf{implements}\, I{<}\overline{V}{>}$ as required.

End case distinction on the last rule used in $\mathcal{D}_2$.

(iii) *Case distinction* on the last rule used in $\mathcal{D}_3$.

- *Case* SUB-Q-ALG-OBJ: Follows with SUB-OBJECT.
- *Case* SUB-Q-ALG-VAR-REFL: Follows with SUB-REFL.
- *Case* SUB-Q-ALG-VAR: Follows by appeal to part (iii) of the I.H. and by applications of rules SUB-VAR and SUB-TRANS.
- *Case* SUB-Q-ALG-CLASS: Follows by combining (possibly repeated) applications of rule SUB-CLASS with rule SUB-TRANS.
- *Case* SUB-Q-ALG-IFACE: Follows by combining (possibly repeated) applications of rule SUB-IFACE with rule SUB-TRANS.

End case distinction on the last rule used in $\mathcal{D}_3$.

(iv) *Case distinction* on the last rule used in $\mathcal{D}_4$.

- *Case* SUB-Q-ALG-KERNEL: Follows from part (iii) of the I.H.
- *Case* SUB-Q-ALG-IMPL: We have

$$\frac{\Delta \vdash_{\mathrm{q}}{}' \; T \leq T' \qquad \Delta \Vdash_{\mathrm{q}}{}' \; T' \,\textbf{implements}\, K}{\Delta \vdash_{\mathrm{q}} T \leq \underbrace{K}_{=U}}$$

By parts (iii) and (i) we get

$$\Delta \vdash T \leq T'$$
$$\Delta \Vdash T' \,\textbf{implements}\, K$$

With SUB-IMPL we then have $\Delta \vdash T' \leq K$, so SUB-TRANS yields the desired result.

End case distinction on the last rule used in $\mathcal{D}_4$. □

B.1.2 Proof of Theorem 3.12

Theorem 3.12 states that quasi-algorithmic constraint entailment and subtyping is complete with respect to the declarative formulations.

Lemma B.1.4 (Transitivity of class and interface inheritance). *If $N_1 \trianglelefteq_{\mathrm{c}} N_2$ and $N_2 \trianglelefteq_{\mathrm{c}} N_3$ then $N_1 \trianglelefteq_{\mathrm{c}} N_3$. If $K_1 \trianglelefteq_{\mathrm{i}} K_2$ and $K_2 \trianglelefteq_{\mathrm{i}} K_3$ then $K_1 \trianglelefteq_{\mathrm{i}} K_3$.*

Proof. By straightforward inductions on the derivations of $N_1 \trianglelefteq_{\mathrm{c}} N_2$ and $K_1 \trianglelefteq_{\mathrm{i}} K_2$, respectively. □

Lemma B.1.5. *If $N \trianglelefteq_{\mathrm{c}} N'$ and $N' \neq Object$ then $N \neq Object$.*

Proof. Follows because programs do not define *Object* explicitly. □

Figure B.1 Transitive and reflexive-transitive containment in type environments.

$$\boxed{X\,\mathbf{extends}\,T \in^{+} \Delta}$$

$$\textsc{in-trans-base}\quad \frac{X\,\mathbf{extends}\,T \in \Delta}{X\,\mathbf{extends}\,T \in^{+} \Delta} \qquad\qquad \textsc{in-trans-step}\quad \frac{X\,\mathbf{extends}\,Y \in \Delta \qquad Y\,\mathbf{extends}\,T \in^{+} \Delta}{X\,\mathbf{extends}\,T \in^{+} \Delta}$$

$$\boxed{X\,\mathbf{extends}\,T \in^{*} \Delta}$$

$$\textsc{in-refl-trans-refl}\quad X\,\mathbf{extends}\,X \in^{*} \Delta \qquad\qquad \textsc{in-refl-trans-trans}\quad \frac{X\,\mathbf{extends}\,T \in^{+} \Delta}{X\,\mathbf{extends}\,T \in^{*} \Delta}$$

Lemma B.1.6 (Reflexivity of kernel of quasi-algorithmic subtyping). *For all types T, it holds that $\Delta \vdash_{\mathrm{q}}{}' T \leq T$.*

Proof. Straightforward. □

We let $\mathcal{J}$ range over judgments. (Remember that $\mathcal{D}$ ranges over derivations.) The notation $\mathcal{D} :: \mathcal{J}$ denotes that $\mathcal{D}$ is a derivation of judgment $\mathcal{J}$.

Lemma B.1.7 (Transitivity of kernel of quasi-algorithmic subtyping). *If $\mathcal{D}_1 :: \Delta \vdash_{\mathrm{q}}{}' T \leq U$ and $\mathcal{D}_2 :: \Delta \vdash_{\mathrm{q}}{}' U \leq V$ then $\Delta \vdash_{\mathrm{q}}{}' T \leq V$.*

Proof. By induction on $\mathcal{D}_1$.
Case distinction on the last rule used in $\mathcal{D}_1$.

- *Case* sub-q-alg-obj: Then $\mathcal{D}_2$ also ends with sub-q-alg-obj (it cannot end with rule sub-q-alg-class because of Lemma B.1.5). Hence, $V = \mathit{Object}$ and the claim follows with sub-q-alg-obj.
- *Case* sub-q-alg-var-refl: Trivial because $T = U$.
- *Case* sub-q-alg-var: By inverting the rule we get $T = X$, $X\,\mathbf{extends}\,T' \in \Delta$, and $\Delta \vdash_{\mathrm{q}}{}' T' \leq U$. If $V = X$ or $V = \mathit{Object}$, then the claim follows directly. Otherwise, we apply the I.H. to $\Delta \vdash_{\mathrm{q}}{}' T' \leq U$ and get $\Delta \vdash_{\mathrm{q}}{}' T' \leq V$. The claim now follows with sub-q-alg-var.
- *Case* sub-q-alg-class: If $V = \mathit{Object}$, then the claim follows with rule sub-q-alg-obj, otherwise by applying Lemma B.1.4.
- *Case* sub-q-alg-iface: Follows from Lemma B.1.4.

End case distinction on the last rule used in $\mathcal{D}_1$. □

Lemma B.1.8. *If $\Delta \Vdash_{\mathrm{q}}{}' K\,\mathbf{implements}\,L$ then $K \trianglelefteq_{\mathrm{i}} L$.*

Proof. The claim follows directly by inverting rule ent-q-alg-iface (other rules are not applicable). □

The notation $X\,\textbf{extends}\,T \in^{+} \Delta$ denotes that the constraint $X\,\textbf{extends}\,T$ is transitively contained in type environment Δ. Correspondingly, $X\,\textbf{extends}\,T \in^{*} \Delta$ denotes that either $X = T$ or that $X\,\textbf{extends}\,T \in^{+} \Delta$. See Figure B.1 for a formal definition of these two relations.

Convention B.1.9. The metavariable B ranges over both class types and interface types. The notation $B \trianglelefteq_{\mathbf{ci}} B'$ abbreviates that either $B = N$, $B' = N'$ for class types N and N' with $N \trianglelefteq_{\mathbf{c}} N'$, or that $B = K$, $B' = K'$ for interface types K, K' with $K \trianglelefteq_{\mathbf{i}} K'$.

Lemma B.1.10 (Inversion of kernel of quasi-algorithmic subtyping). *Suppose* $\Delta \vdash_{\mathrm{q}}{}' T \leq U$.

- *(i)* *If* $T = X$ *for some* X *then either* $U = Y$ *for some* Y *and* $X\,\textbf{extends}\,Y \in^{*} \Delta$, *or* $U = \mathit{Object}$, *or* $U = B$ *for some* $B \neq \mathit{Object}$ *and* $X\,\textbf{extends}\,B' \in^{+} \Delta$ *for some* B' *with* $B' \trianglelefteq_{\mathbf{ci}} B$.
- *(ii)* *If* $U = Y$ *for some* Y *then* $T = X$ *for some* X *and* $X\,\textbf{extends}\,Y \in^{*} \Delta$.
- *(iii)* *If* $T = N$ *for some* N *then* $U = N'$ *for some* N' *with* $N \trianglelefteq_{\mathbf{c}} N'$.
- *(iv)* *If* $T = K$ *for some* K *then either* $U = K'$ *for some* K' *with* $K \trianglelefteq_{\mathbf{i}} K'$ *or* $U = \mathit{Object}$.

Proof. Claims (iii) and (iv) follow by inspecting the rules defining the relation $\cdot \vdash_{\mathrm{q}}{}' \cdot \leq \cdot$. Claim (ii) follows by inspecting the rules defining the relation $\cdot \vdash_{\mathrm{q}}{}' \cdot \leq \cdot$ and by claim (i).

We now prove claim (i) by induction on the derivation of $\Delta \vdash_{\mathrm{q}}{}' T \leq U$. Thereby, we assume that $U \neq \mathit{Object}$ as the claim holds trivially in this case. Because $T = X$, the derivation either ends with SUB-Q-ALG-VAR-REFL or SUB-Q-ALG-VAR. The first case is trivial. For the second case we have

$$\frac{X\,\textbf{extends}\,U' \in \Delta \qquad U \neq \mathit{Object}, U \neq X \qquad \Delta \vdash_{\mathrm{q}}{}' U' \leq U}{\Delta \vdash_{\mathrm{q}}{}' X \leq U}\ \textsc{sub-q-alg-var}$$

Case distinction on the form of U'.

- *Case* $U' = Z$ for some Z: Applying the I.H. to $\Delta \vdash_{\mathrm{q}}{}' U' \leq U$ yields that either $U = Y$ for some Y and $Z\,\textbf{extends}\,Y \in^{*} \Delta$, or that $U = B$ for some B and $Z\,\textbf{extends}\,B' \in^{+} \Delta$ for some B' with $B' \trianglelefteq_{\mathbf{ci}} B$. It is easy to verify that claim (i) follows from these facts.
- *Case* $U' = B'$ for some B': Using claims (iii) and (iv) we get that $U = B$ for some $B \neq \mathit{Object}$ with $B' \trianglelefteq_{\mathbf{ci}} B$. The claim now follows trivially.

End case distinction on the form of U'. □

Lemma B.1.11. *If* $\Delta \vdash_{\mathrm{q}}{}' T \leq U$ *and* $\Delta \vdash_{\mathrm{q}} U \leq V$, *then* $\Delta \vdash_{\mathrm{q}} T \leq V$.

Proof. If the derivation of $\Delta \vdash_{\mathrm{q}} U \leq V$ ends with SUB-Q-ALG-KERNEL, then $\Delta \vdash_{\mathrm{q}}{}' U \leq V$ so the claim follows by Lemma B.1.7. Otherwise, we have $V = K$ and

$$\frac{\Delta \vdash_{\mathrm{q}}{}' U \leq U' \qquad \Delta \Vdash_{\mathrm{q}}{}' U'\,\textbf{implements}\,K}{\Delta \vdash_{\mathrm{q}} U \leq K}\ \textsc{sub-q-alg-impl}$$

With Lemma B.1.7 we have $\Delta \vdash_{\mathrm{q}}{}' T \leq U'$, so the claim follows with SUB-Q-ALG-IMPL. □

Lemma B.1.12 (Type substitution preserves inheritance). *If* $\vdash B \trianglelefteq_{\mathbf{ci}} B'$ *then* $\vdash \varphi B \trianglelefteq_{\mathbf{ci}} \varphi B'$.

Proof. We show the claim for $B = K$ and $B' = K'$ by induction on the derivation of $K \trianglelefteq_{\mathbf{i}} K'$; the proof for $B = N$ and $B' = N'$ is similar.
Case distinction on the last rule used in $K \trianglelefteq_{\mathbf{i}} K'$.

- *Case* INH-IFACE-REFL: Trivial because $K = K'$.

Figure B.2 Generalization of sup to subtype constraints.

SUP-EXT-REFL

$$T\,\mathbf{extends}\,U \in \mathsf{sup}(T\,\mathbf{extends}\,U)$$

SUP-EXT-INH

$$\frac{\vdash K \trianglelefteq_{\mathrm{i}} L}{T\,\mathbf{extends}\,L \in \mathsf{sup}(T\,\mathbf{extends}\,K)}$$

- *Case* INH-IFACE-SUPER: Then $K = I\texttt{<}\overline{T}\texttt{>}$ and

$$\frac{\mathbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y}\,\mathbf{where}\,\overline{R}]\ \ldots \qquad R_i = \overline{G}\,\mathbf{implements}\,L \qquad [\overline{T/X}]L \trianglelefteq_{\mathrm{i}} K'}{I\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathrm{i}} K'}$$

  Applying the I.H. to $[\overline{T/X}]L \trianglelefteq_{\mathrm{i}} K'$ yields $\varphi[\overline{T/X}]L \trianglelefteq_{\mathrm{i}} \varphi K'$. Because the definition of I does not contain free type variables (we globally assume that the underlying program is well-formed), we have $\varphi[\overline{T/X}]L = [\overline{\varphi T/X}]L$. Hence, $\varphi K \trianglelefteq_{\mathrm{i}} \varphi K'$ by INH-IFACE-SUPER.

End case distinction on the last rule used in $K \trianglelefteq_{\mathrm{i}} K'$. □

Lemma B.1.13 (Type substitution preserves sup). *If $\mathcal{R} \in \mathsf{sup}(\mathcal{S})$ then $\varphi\mathcal{R} \in \mathsf{sup}(\varphi\mathcal{S})$.*

Proof. The proof is by induction on the derivation of $\mathcal{R} \in \mathsf{sup}(\mathcal{S})$. The claim holds trivially if this derivation ends with rule SUP-REFL. Now suppose the last rule is SUP-STEP:

$$\frac{\mathbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y}\,\mathbf{where}\,\overline{R}]\ \ldots \qquad \overline{U}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \in \mathsf{sup}(\mathcal{S})}{\underbrace{[\overline{V/X}, \overline{U/Y}]R_k}_{=\mathcal{R}} \in \mathsf{sup}(\mathcal{S})}$$

By the I.H. we have

$$\varphi(\overline{U}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}) \in \mathsf{sup}(\varphi\mathcal{S})$$

. Thus, by rule SUP-STEP we get $[\overline{\varphi V/X}, \overline{\varphi U/Y}]R_k \in \mathsf{sup}(\varphi\mathcal{S})$. The definition of I does not contain free type variables, so $\mathsf{ftv}(R_k) \subseteq \{\overline{X}, \overline{Y}\}$. Hence

$$[\overline{\varphi V/X}, \overline{\varphi U/Y}]R_k = \varphi([\overline{V/X}, \overline{U/Y}]R_k) = \varphi\mathcal{R}$$

. □

Lemma B.1.14. *If $\Delta \vdash_{\mathrm{q}} T \leq U$ and $U \neq K$ for any K then $\Delta \vdash_{\mathrm{q}}{}' T \leq U$.*

Proof. Obvious. □

Convention B.1.15. The notation $\mathsf{sup}(\Delta)$ denotes the type environment $\{\mathcal{P} \mid \mathcal{P} \in \mathsf{sup}(\mathcal{Q}), \mathcal{Q} \in \Delta\}$, where Figure B.2 defines the generalization of sup to subtype constraints. Applying a type substitution φ to a type environment Δ, written $\varphi\Delta$, yields the type environment $\{\varphi\mathcal{P} \mid \mathcal{P} \in \Delta\}$. The notation $\Delta \Vdash \Delta'$ abbreviates $(\forall \mathcal{P} \in \Delta')\ \Delta \Vdash \mathcal{P}$, and $\Delta \Vdash_{\mathrm{q}} \Delta'$ is defined analogously.

Lemma B.1.16. *Suppose $\Delta \Vdash_{\mathrm{q}} \mathsf{sup}(\varphi\Delta')$.*

(*i*) *If $X\,\mathbf{extends}\,Y \in^{*} \Delta'$ then either $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi Y$ or $\varphi Y = K$ for some K such that $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$ for all K' with $K \trianglelefteq_{\mathrm{i}} K'$.*

(*ii*) *If $X\,\mathbf{extends}\,B \in^{+} \Delta'$ then either $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi B$ or $\varphi B = K$ for some K such that $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$ for all K' with $K \trianglelefteq_{\mathrm{i}} K'$.*

Proof. We prove both claims separately.

(i) The proof of claim (i) is by induction on the derivation of $X\,\mathbf{extends}\,Y \in^* \Delta'$. If $X = Y$ then the claim follows with Lemma B.1.6. Otherwise, we have

$$\frac{X\,\mathbf{extends}\,Y' \in \Delta' \qquad Y'\,\mathbf{extends}\,Y \in^* \Delta'}{X\,\mathbf{extends}\,Y \in^* \Delta'}$$

By the assumption we have

$$\Delta \vdash_{\mathrm{q}} \varphi X \leq \varphi Y' \tag{B.1.4}$$

and, if $\varphi Y' = L$ for some L, then

$$\Delta \vdash_{\mathrm{q}} \varphi X \leq L' \text{ for all } L' \text{ with } L \trianglelefteq_{\mathrm{i}} L' \tag{B.1.5}$$

Applying the I.H. to $Y'\,\mathbf{extends}\,Y \in^* \Delta'$ yields either

$$\Delta \vdash_{\mathrm{q}}{}' \varphi Y' \leq \varphi Y \tag{B.1.6}$$

or $\varphi Y = K$ for some K and

$$\Delta \vdash_{\mathrm{q}} \varphi Y' \leq K' \text{ for all } K' \text{ with } K \trianglelefteq_{\mathrm{i}} K' \tag{B.1.7}$$

Case distinction on the form of $\varphi Y'$ and on whether (B.1.6) or (B.1.7) holds.

- *Case* $\varphi Y' \neq L$ for any L and (B.1.6) holds: By Lemma B.1.14 and (B.1.4) we get

  $$\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi Y'$$

  With (B.1.6) and Lemma B.1.7 we get $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi Y$ as required.
- *Case* $\varphi Y' \neq L$ for any L and (B.1.7) holds: As in the preceding case, we have $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi Y'$. Using (B.1.7) and Lemma B.1.11 we get

  $$\Delta \vdash_{\mathrm{q}} \varphi X \leq K' \text{ for all } K' \text{ with } K \trianglelefteq_{\mathrm{i}} K'$$

  as required.
- *Case* $\varphi Y' = L$ for some L and (B.1.6) holds: With (B.1.6) and Lemma B.1.10 we get either that $\varphi Y = \mathit{Object}$ or that $\varphi Y = K$ for some K with $L \trianglelefteq_{\mathrm{i}} K$. If $\varphi Y = \mathit{Object}$ then $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi Y$ by SUB-Q-ALG-OBJ. Now assume $\varphi Y = K$. With (B.1.5) and $L \trianglelefteq_{\mathrm{i}} K$ we get $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$ for all K' with $K \trianglelefteq_{\mathrm{i}} K'$.
- *Case* $\varphi Y' = L$ for some L and (B.1.7) holds: Suppose $K \trianglelefteq_{\mathrm{i}} K'$ for some K'.
  - If the derivation of $\Delta \vdash_{\mathrm{q}} \varphi Y' \leq K'$ in (B.1.7) ends with SUB-Q-ALG-KERNEL, then we have $\Delta \vdash_{\mathrm{q}}{}' \varphi Y' \leq K'$. Hence, by Lemma B.1.10 $L \trianglelefteq_{\mathrm{i}} K'$. Using (B.1.5) we get $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$.
  - If the derivation of $\Delta \vdash_{\mathrm{q}} \varphi Y' \leq K'$ in (B.1.7) ends with SUB-Q-ALG-IMPL, we have

    $$\frac{\Delta \vdash_{\mathrm{q}}{}' \varphi Y' \leq T \qquad \Delta \Vdash_{\mathrm{q}}{}' T\,\mathbf{implements}\,K'}{\Delta \vdash_{\mathrm{q}} \varphi Y' \leq K'}$$

    With Lemma B.1.10 we need to consider two cases for the form of T:

    * $T = \mathit{Object}$. Then we have $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq T$, so $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$.

- ∗ $T = L'$ and $L \trianglelefteq_{\mathrm{i}} L'$. With Lemma B.1.8 we get $L' \trianglelefteq_{\mathrm{i}} K'$. Thus, $L \trianglelefteq_{\mathrm{i}} K'$. Equation (B.1.5) then gives us $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$.

We now have $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$ for all K' with $K \trianglelefteq_{\mathrm{i}} K'$ as required.

End case distinction on the form of $\varphi Y'$ and on whether (B.1.6) or (B.1.7) holds.

(ii) We prove claim (ii) by induction on the derivation of $X\,\mathbf{extends}\,B \in^{+} \Delta'$. We have

$$\frac{X\,\mathbf{extends}\,Y \in^{*} \Delta' \qquad Y\,\mathbf{extends}\,B \in \Delta'}{X\,\mathbf{extends}\,B \in^{+} \Delta'}$$

By claim (i) we have that either

$$\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi Y \tag{B.1.8}$$

or that

$$\begin{gathered}\varphi Y = L \text{ for some } L \text{ and}\\ \Delta \vdash_{\mathrm{q}} \varphi X \leq L' \text{ for all } L' \text{ with } L \trianglelefteq_{\mathrm{i}} L'\end{gathered} \tag{B.1.9}$$

We have by the assumption

$$\Delta \vdash_{\mathrm{q}} \varphi Y \leq \varphi B \tag{B.1.10}$$

and, if $\varphi B = K$ for some K then

$$\Delta \vdash_{\mathrm{q}} \varphi Y \leq K' \text{ for all } K' \text{ with } K \trianglelefteq_{\mathrm{i}} K' \tag{B.1.11}$$

Case distinction on the form of φB and on whether (B.1.8) or (B.1.9) holds.

- *Case* $\varphi B = N$ for some N and (B.1.8) holds: Then by (B.1.10) and Lemma B.1.14 $\Delta \vdash_{\mathrm{q}}{}' \varphi Y \leq \varphi B$. With (B.1.8) and Lemma B.1.7 $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi B$ as required.
- *Case* $\varphi B = K$ for some K and (B.1.8) holds: Assume K' such that $K \trianglelefteq_{\mathrm{i}} K'$.
  - If the derivation of $\Delta \vdash_{\mathrm{q}} \varphi Y \leq K'$ in (B.1.11) ends with sub-q-alg-kernel, then $\Delta \vdash_{\mathrm{q}}{}' \varphi Y \leq K'$, so $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq K'$ by (B.1.8) and Lemma B.1.7. Hence, $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$
  - If the derivation of $\Delta \vdash_{\mathrm{q}} \varphi Y \leq K'$ in (B.1.11) ends with sub-q-alg-impl, then we have

    $$\frac{\Delta \vdash_{\mathrm{q}}{}' \varphi Y \leq T \qquad \Delta \Vdash_{\mathrm{q}}{}' T\,\mathbf{implements}\,K'}{\Delta \vdash_{\mathrm{q}} \varphi Y \leq K'}$$

    By (B.1.8) and Lemma B.1.7 we then have $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq T$, thus $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$.

  We now have $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$ for all K' with $K \trianglelefteq_{\mathrm{i}} K'$ as required.
- *Case* $\varphi B = N$ for some N and (B.1.9) holds: Then by (B.1.10) and Lemma B.1.14: $\Delta \vdash_{\mathrm{q}}{}' \varphi Y \leq \varphi B$. With (B.1.9) we know that $\varphi Y = L$ for some L. Hence, by Lemma B.1.10 $\varphi B = \mathit{Object}$. We then have $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi B$ by sub-q-alg-obj.
- *Case* $\varphi B = K$ for some K and (B.1.9) holds: By (B.1.9) we have $\varphi Y = L$ for some L. Assume K' such that $K \trianglelefteq_{\mathrm{i}} K'$.
  - If the derivation of $\Delta \vdash_{\mathrm{q}} \varphi Y \leq K'$ in (B.1.11) ends with sub-q-alg-kernel, then $\Delta \vdash_{\mathrm{q}}{}' \varphi Y \leq K'$. Hence, $L \trianglelefteq_{\mathrm{i}} K'$ by Lemma B.1.10. Using (B.1.9) we then have $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$.

- If the derivation of $\Delta \vdash_{\mathrm{q}} \varphi Y \leq K'$ in (B.1.11) ends with SUB-Q-ALG-IMPL, then we have

$$\frac{\Delta \vdash_{\mathrm{q}}{}' \varphi Y \leq T \qquad \Delta \Vdash_{\mathrm{q}}{}' T \,\mathbf{implements}\, K'}{\Delta \vdash_{\mathrm{q}} \varphi Y \leq K'}$$

With Lemma B.1.10 we need to consider two cases for the form of T:

  - $T = \mathit{Object}$. Then we have $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq T$, so $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$.
  - $T = L'$ and $L \trianglelefteq_{\mathrm{i}} L'$. With Lemma B.1.8 we get $L' \trianglelefteq_{\mathrm{i}} K'$. Thus, $L \trianglelefteq_{\mathrm{i}} K'$. Equation (B.1.9) then gives us $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$.

We now have $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$ for all K' with $K \trianglelefteq_{\mathrm{i}} K'$ as required.

End case distinction on the form of φB and on whether (B.1.8) or (B.1.9) holds. □

Lemma B.1.17. *If $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{R}$ then $\Delta \Vdash_{\mathrm{q}} \mathcal{R}$.*

Proof. Obvious with rule ENT-Q-ALG-UP. □

Lemma B.1.18 (Inheritance preserves polarity). *If $J\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{U}\texttt{>}$ and $\mathsf{pol}^{\pi}(J)$ then $1 \in \mathsf{pol}^{\pi}(I)$.*

Proof. Induction on the derivation of $J\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{U}\texttt{>}$. If the derivation ends with INH-IFACE-REFL, then $J\texttt{<}\overline{T}\texttt{>} = I\texttt{<}\overline{U}\texttt{>}$ and the claim holds trivially. Otherwise, assume

$$\frac{\begin{array}{c}\mathbf{interface}\ J\texttt{<}\overline{X}\texttt{>}\,[Y\,\mathbf{where}\,R]\ \ldots \\ R_i = \overline{G}^{n}\,\mathbf{implements}\,J'\texttt{<}\overline{V}\texttt{>} \qquad [\overline{T/X}]J'\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{U}\texttt{>}\end{array}}{J\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{U}\texttt{>}}\ \text{INH-IFACE-SUPER}$$

By criterion WF-IFACE-2 we have $n = 1$ and $G_1 = Y$. With $1 \in \mathsf{pol}^{\pi}(J)$ we have $Y \in \mathsf{pol}^{\pi}(R_i)$ by POL-IFACE, so $1 \in \mathsf{pol}^{\pi}(J')$ by POL-CONSTR. We can now apply the I.H. to $[\overline{T/X}]J'\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{U}\texttt{>}$ and get $1 \in \mathsf{pol}^{\pi}(I)$ as required. □

Lemma B.1.19 (Inheritance preserves non-static). *Assume $J\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{U}\texttt{>}$ and $\mathsf{non\text{-}static}(J)$. Then also $\mathsf{non\text{-}static}(I)$.*

Proof. Straightforward induction on the derivation of $J\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{U}\texttt{>}$. □

Lemma B.1.20. *If $\mathcal{D} :: \Delta \Vdash_{\mathrm{q}}{}' \overline{T}^{n}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>}$ and there exists $i \in [n]$ such that $T_i = K$ for some K, then $n = 1$, $1 \in \mathsf{pol}^{+}(I)$, and $\mathsf{non\text{-}static}(I)$.*

Proof. It is easy to see that $\mathcal{D}$ must end with rule ENT-Q-ALG-IFACE. We then have $n = 1$, $T_1 = J\texttt{<}\overline{V}\texttt{>}$ for some $J\texttt{<}\overline{V}\texttt{>}$, $1 \in \mathsf{pol}^{+}(J)$, $\mathsf{non\text{-}static}(J)$, and $J\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{U}\texttt{>}$. By Lemma B.1.18 $1 \in \mathsf{pol}^{+}(I)$ and by Lemma B.1.19 $\mathsf{non\text{-}static}(I)$. □

Lemma B.1.21. *Suppose $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ for all $\mathcal{P} \in \mathsf{sup}(\varphi\Delta')$.*

(i) *If $\mathcal{D}_1 :: \Delta' \vdash_{\mathrm{q}}{}' T \leq U$ then either $\Delta \vdash_{\mathrm{q}}{}' \varphi T \leq \varphi U$ or $\varphi U = K$ for some K and $\Delta \vdash_{\mathrm{q}} \varphi T \leq K'$ for all K' with $K \trianglelefteq_{\mathrm{i}} K'$.*

(ii) *If $\mathcal{D}_2 :: \Delta' \vdash_{\mathrm{q}} T \leq U$ then $\Delta \vdash_{\mathrm{q}} \varphi T \leq \varphi U$.*

(iii) *If $\mathcal{D}_2 :: \Delta' \Vdash_{\mathrm{q}}{}' \mathcal{R}$ then $\Delta \Vdash_{\mathrm{q}} \varphi\mathcal{R}$.*

(iv) *If $\mathcal{D}_4 :: \Delta' \Vdash_{\mathrm{q}} \mathcal{Q}$ then $\Delta \Vdash_{\mathrm{q}} \varphi\mathcal{Q}$.*

Proof. We proceed by induction on the combined height of $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4$.

(i) *Case distinction* on the last rule used in $\mathcal{D}_1$.

- *Case* SUB-Q-ALG-OBJ: Trivial.
- *Case* SUB-Q-ALG-VAR-REFL: Follows with Lemma B.1.6.
- *Case* SUB-Q-ALG-VAR: We have $T = X$. Thus, by Lemma B.1.10, we can distinguish three different cases:
  - $U = Y$ for some Y and $X\,\textbf{extends}\,Y \in^* \Delta'$. Then the claim follows with Lemma B.1.16.
  - $U = \mathit{Object}$. In this case, $\Delta \vdash_{\mathrm{q}}{}' \varphi T \leq \varphi U$ holds by SUB-Q-ALG-OBJ.
  - $U = B$ for some $B \neq \mathit{Object}$ and $X\,\textbf{extends}\,B' \in^+ \Delta'$ for some B' with $B' \trianglelefteq_{\mathbf{ci}} B$. Then $\varphi B' \trianglelefteq_{\mathbf{i}} \varphi B$ by Lemma B.1.12. By Lemma B.1.16, we either have $\Delta \vdash_{\mathrm{q}}{}' \varphi X \leq \varphi B'$ or $\varphi B' = L$ for some L and $\Delta \vdash_{\mathrm{q}} \varphi X \leq L'$ for all L' with $L \trianglelefteq_{\mathbf{i}} L'$.
    * For the first case, we note that $\varphi B' \trianglelefteq_{\mathbf{i}} \varphi B$ implies $\Delta \vdash_{\mathrm{q}}{}' \varphi B' \leq \varphi B$. The claim now follows with Lemma B.1.7.
    * For the second case, we have with $\varphi B' = L$ for some L that $\varphi B = K$ for some K such that $L \trianglelefteq_{\mathbf{i}} K$. If now $K \trianglelefteq_{\mathbf{i}} K'$ then $L \trianglelefteq_{\mathbf{i}} K'$ (by Lemma B.1.4), so $\Delta \vdash_{\mathrm{q}} \varphi X \leq K'$ as required.
- *Case* SUB-Q-ALG-CLASS: Follows with Lemma B.1.12.
- *Case* SUB-Q-ALG-IFACE: Follows with Lemma B.1.12.

End case distinction on the last rule used in $\mathcal{D}_1$.

(ii) *Case distinction* on the last rule used in $\mathcal{D}_2$.

- *Case* SUB-Q-ALG-KERNEL: We have

$$\frac{\Delta' \vdash_{\mathrm{q}}{}' T \leq U}{\Delta' \vdash_{\mathrm{q}} T \leq U}$$

By part (i) of the I.H., we have either $\Delta \vdash_{\mathrm{q}}{}' \varphi T \leq \varphi U$ (which implies $\Delta \vdash_{\mathrm{q}} \varphi T \leq \varphi U$) or $\Delta \vdash_{\mathrm{q}} \varphi T \leq \varphi U$, so the claim holds.

- *Case* SUB-Q-ALG-IMPL: We have $U = I\texttt{<}\overline{W}\texttt{>}$ for some $I\texttt{<}\overline{W}\texttt{>}$ and

$$\frac{\Delta' \vdash_{\mathrm{q}}{}' T \leq V \qquad \Delta' \Vdash_{\mathrm{q}}{}' V\,\textbf{implements}\,I\texttt{<}\overline{W}\texttt{>}}{\Delta' \vdash_{\mathrm{q}} T \leq I\texttt{<}\overline{W}\texttt{>}}$$

Applying parts (i) and (iii) of the I.H. yields

$$\frac{\begin{array}{c}\Delta \vdash_{\mathrm{q}}{}' \varphi V \leq V' \\ \text{if } \varphi V \neq V' \text{ then } 1 \in \mathsf{pol}^-(I) \\ \Delta \Vdash_{\mathrm{q}}{}' V'\,\textbf{implements}\,\varphi I\texttt{<}\overline{W}\texttt{>}\end{array}}{\Delta \Vdash_{\mathrm{q}} \varphi V\,\textbf{implements}\,\varphi I\texttt{<}\overline{W}\texttt{>}}\ \text{ENT-Q-ALG-UP} \tag{B.1.12}$$

and either

$$\Delta \vdash_{\mathrm{q}}{}' \varphi T \leq \varphi V \tag{B.1.13}$$

or

$$\begin{array}{c}\varphi V = L \text{ for some } L \text{ and} \\ \Delta \vdash_{\mathrm{q}} \varphi T \leq L' \text{ for all } L' \text{ with } L \trianglelefteq_{\mathbf{i}} L'\end{array} \tag{B.1.14}$$

- Suppose (B.1.13). Then we have by the first premise in (B.1.12), by (B.1.13), and by Lemma B.1.11 that $\Delta \vdash_{\mathrm{q}}' \varphi T \leq V'$. With the last premise in (B.1.12) and with rule SUB-Q-ALG-IMPL, we then get $\Delta \vdash_{\mathrm{q}} \varphi T \leq \varphi I{<}\overline{W}{>}$ as required.
- Suppose (B.1.14). Then we have by the first premise in (B.1.12), by the fact that $\varphi V = L$, and by Lemma B.1.10 that either $V' = \mathit{Object}$ or that $V' = L'$ for some L' with $L \trianglelefteq_{\mathrm{i}} L'$.
  - If $V' = \mathit{Object}$ then $\Delta \vdash_{\mathrm{q}}' \varphi T \leq V'$, so the claim follows with the last premise in (B.1.12) and with rule SUB-Q-ALG-IMPL.
  - Otherwise, $V' = L'$ and $L \trianglelefteq_{\mathrm{i}} L'$. From the last premise in (B.1.12), we have $\Delta \vdash_{\mathrm{q}}' L'\ \mathbf{implements}\ \varphi I{<}\overline{W}{>}$, so we get with Lemma B.1.8 that $L' \trianglelefteq_{\mathrm{i}} \varphi I{<}\overline{W}{>}$. Hence, $L \trianglelefteq_{\mathrm{i}} \varphi I{<}\overline{W}{>}$ by Lemma B.1.4. By (B.1.14) we then have $\Delta \vdash_{\mathrm{q}} \varphi T \leq \varphi I{<}\overline{W}{>}$ as required (note that $\varphi I{<}\overline{W}{>} = \varphi U$).

End case distinction on the last rule used in $\mathcal{D}_2$.

(iii) *Case distinction* on the last rule used in $\mathcal{D}_3$.

- *Case* ENT-Q-ALG-ENV: We have $S \in \Delta'$ and $R \in \mathsf{sup}(S)$ such that $R = \mathfrak{R}$. With Lemma B.1.13 we get $\varphi R \in \mathsf{sup}(\varphi S)$. Clearly, $\varphi S \in \varphi\Delta'$, so the assumption gives us $\Delta \Vdash_{\mathrm{q}} \varphi R$ as required.
- *Case* ENT-Q-ALG-IMPL: We have

$$\frac{\mathbf{implementation}{<}\overline{X}{>}\ I{<}\overline{T}{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots \qquad \Delta' \Vdash_{\mathrm{q}} [\overline{V/X}]\overline{P}}{\Delta' \Vdash_{\mathrm{q}}' \underbrace{[\overline{V/X}](\overline{N}\ \mathbf{implements}\ I{<}\overline{T}{>})}_{=\mathfrak{R}}}$$

Applying part (iv) of the I.H. yields

$$\Delta \Vdash_{\mathrm{q}} \varphi[\overline{V/X}]\overline{P}$$

Because implementation definitions do not contain free type variables, we have

$$\begin{gathered} \varphi[\overline{V/X}]\overline{P} = [\overline{\varphi V/X}]\overline{P} \\ \varphi[\overline{V/X}](\overline{N}\ \mathbf{implements}\ I{<}\overline{T}{>}) = [\overline{\varphi V/X}](\overline{N}\ \mathbf{implements}\ I{<}\overline{T}{>}) \end{gathered}$$

By ENT-Q-ALG-IMPL we then have $\Delta \Vdash_{\mathrm{q}}' \varphi[\overline{V/X}](\overline{N}\ \mathbf{implements}\ I{<}\overline{T}{>})$, thus $\Delta \Vdash_{\mathrm{q}} \varphi\mathfrak{R}$ by Lemma B.1.17.

- *Case* ENT-Q-ALG-IFACE: We have

$$\frac{1 \in \mathsf{pol}^{+}(I) \qquad \mathsf{non\text{-}static}(I) \qquad I{<}\overline{V}{>} \trianglelefteq_{\mathrm{i}} K}{\Delta' \Vdash_{\mathrm{q}}' \underbrace{I{<}\overline{V}{>}\ \mathbf{implements}\ K}_{=\mathfrak{R}}}$$

By Lemma B.1.12, we have $\varphi I{<}\overline{V}{>} \trianglelefteq_{\mathrm{i}} \varphi K$. Thus, with ENT-Q-ALG-IFACE, we get $\Delta \Vdash_{\mathrm{q}}' \varphi\mathfrak{R}$, so $\Delta \Vdash_{\mathrm{q}} \varphi\mathfrak{R}$ by Lemma B.1.17.

End case distinction on the last rule used in $\mathcal{D}_3$.

(iv) *Case distinction* on the last rule used in $\mathcal{D}_4$.

- *Case* ENT-Q-ALG-EXTENDS: Follows from part (ii) of the I.H.

- *Case* ENT-Q-ALG-UP: We have

$$\frac{(\forall i)\ \Delta' \vdash_{\mathrm{q}}{}'\ T_i \le U_i \quad \text{if } T_i \ne U_i \text{ then } i \in \mathsf{pol}^-(I) \qquad \Delta' \Vdash_{\mathrm{q}}{}'\ \overline{U}\ \textbf{implements}\ I{<}\overline{V}{>}}{\Delta' \Vdash_{\mathrm{q}} \underbrace{\overline{T}^n\ \textbf{implements}\ I{<}\overline{V}{>}}_{=\mathcal{Q}}} \tag{B.1.15}$$

We get by part (iii) of the I.H.:

$$\frac{\begin{array}{c}(\forall i)\ \Delta \vdash_{\mathrm{q}}{}'\ \varphi U_i \le U'_i \quad \text{if } \varphi U_i \ne U'_i \text{ then } i \in \mathsf{pol}^-(I) \\ \Delta \Vdash_{\mathrm{q}}{}'\ \overline{U'}\ \textbf{implements}\ I{<}\overline{\varphi V}{>}\end{array}}{\Delta \Vdash_{\mathrm{q}} \varphi\overline{U}\ \textbf{implements}\ I{<}\overline{\varphi V}{>}}\ \textsc{ent-q-alg-up} \tag{B.1.16}$$

Suppose $i \in [n]$. If $i \in \mathsf{pol}^-(I)$ does not hold, then we have $T_i = U_i$ and $\varphi U_i = U'_i$. Hence,

$$\varphi T_i = U'_i \text{ or } i \in \mathsf{pol}^-(I) \tag{B.1.17}$$

Moreover, by part (i) of the I.H. applied to the first premise in (B.1.15) we get that either

$$\Delta \vdash_{\mathrm{q}}{}'\ \varphi T_i \le \varphi U_i \tag{B.1.18}$$

or

$$\begin{array}{c}\varphi U_i = K_i \text{ for some } K_i \text{ and} \\ \Delta \vdash_{\mathrm{q}} \varphi T_i \le K'_i \text{ for all } K'_i \text{ with } K_i \trianglelefteq_{\mathrm{i}} K'_i\end{array} \tag{B.1.19}$$

We now partition $[n] = \mathscr{M}_1 \mathbin{\dot\cup} \mathscr{M}_2$ such that

$$\begin{aligned}\mathscr{M}_1 &= \{j \in [n] \mid \text{Equation (B.1.18) holds for } j\} \\ \mathscr{M}_2 &= \{l \in [n] \mid \text{Equation (B.1.19) holds for } l\}\end{aligned}$$

  - If $j \in \mathscr{M}_1$, then we have with (B.1.18), the first premise in (B.1.16), and Lemma B.1.7 that $\Delta \vdash_{\mathrm{q}}{}'\ \varphi T_j \le U'_j$.
  - If $l \in \mathscr{M}_2$, then $\varphi U_l = K_l$ for some K_l. By Lemma B.1.10 applied to the first premise in (B.1.16), we then have that either $U'_l = K'_l$ for some K'_l or $U'_l = \mathit{Object}$.

  Now we further partition $\mathscr{M}_2$ into $\mathscr{M}_{21} \mathbin{\dot\cup} \mathscr{M}_{22}$ such that

$$\begin{aligned}\mathscr{M}_{21} &= \{l \in \mathscr{M}_2 \mid U'_l = K'_l \text{ for some } K'_l\} \\ \mathscr{M}_{22} &= \{l \in \mathscr{M}_2 \mid U'_l = \mathit{Object}\}\end{aligned}$$

  Case distinction on whether or not $\mathscr{M}_{21} = \emptyset$.

  - *Case* $\mathscr{M}_{21} = \emptyset$: Then we have $[n] = \mathscr{M}_1 \mathbin{\dot\cup} \mathscr{M}_{22}$, so $\Delta \vdash_{\mathrm{q}}{}'\ \varphi T_i \le U'_i$ for all $i \in [n]$. Thus, with (B.1.17) and the last premise in (B.1.16) we can apply ENT-Q-ALG-UP and get $\Delta \Vdash_{\mathrm{q}} \varphi\overline{T}\ \textbf{implements}\ I{<}\overline{\varphi V}{>}$ as required.
  - *Case* $\mathscr{M}_{21} \ne \emptyset$: With Lemma B.1.20 applied to the last premise in (B.1.16), we get that $n = 1$ and that

$$1 \in \mathsf{pol}^+(I) \tag{B.1.20}$$

$$\mathsf{non\text{-}static}(I) \tag{B.1.21}$$

In the following, we may assume

$$1 \in \mathsf{pol}^-(I) \tag{B.1.22}$$

Otherwise, we have $\varphi T_1 = U_1'$ with (B.1.17) and the claim then follows with the last premise in (B.1.16) and ENT-Q-ALG-UP.

With $n = 1$ and $\mathscr{M}_{21} \neq \emptyset$, we have $1 \in \mathscr{M}_{21}$. Hence, $U_1' = K_1'$ for some K_1'. With the last premise in (B.1.16) and Lemma B.1.8 we then have $K_1' \trianglelefteq_{\mathsf{i}} I<\overline{\varphi V}>$. Because $1 \in \mathscr{M}_{21} \subseteq \mathscr{M}_2$, we have we have $\varphi U_1 = K_1$ for some K_1. The first premise in (B.1.16) and Lemma B.1.10 then gives us $K_1 \trianglelefteq_{\mathsf{i}} K_1'$. With Lemma B.1.4: $K_1 \trianglelefteq_{\mathsf{i}} I<\overline{\varphi V}>$. Equation (B.1.19) holds because $1 \in \mathscr{M}_2$, so

$$\Delta \vdash_{\mathsf{q}} \varphi T_1 \leq I<\overline{\varphi V}> \tag{B.1.23}$$

Case distinction on the last rule used in the derivation of (B.1.23).

- *Case* SUB-Q-ALG-KERNEL: Then $\Delta \vdash_{\mathsf{q}}{}' \varphi T_1 \leq I<\overline{\varphi V}>$. With (B.1.20), (B.1.21), and (B.1.22) we then have

$$\dfrac{\textsc{ent-q-alg-iface}\ \dfrac{\begin{array}{c}\Delta \vdash_{\mathsf{q}}{}' \varphi T_1 \leq I<\overline{\varphi V}> \qquad 1 \in \mathsf{pol}^-(I) \\ 1 \in \mathsf{pol}^+(I) \qquad \mathsf{non\text{-}static}(I) \qquad I<\overline{\varphi V}> \trianglelefteq_{\mathsf{i}} I<\overline{\varphi V}>\end{array}}{\Delta \Vdash_{\mathsf{q}}{}' I<\overline{\varphi V}>\,\mathbf{implements}\,I<\overline{\varphi V}>}}{\Delta \Vdash_{\mathsf{q}} \varphi T_1\,\mathbf{implements}\,I<\overline{\varphi V}>}\ \textsc{ent-q-alg-up}$$

- *Case* SUB-Q-ALG-IMPL: We then have

$$\dfrac{\Delta \vdash_{\mathsf{q}}{}' \varphi T_1 \leq W \qquad \Delta \Vdash_{\mathsf{q}}{}' W\,\mathbf{implements}\,I<\overline{\varphi V}>}{\Delta \vdash_{\mathsf{q}} \varphi T_1 \leq I<\overline{\varphi V}>}$$

  With (B.1.22) we get

$$\dfrac{\Delta \vdash_{\mathsf{q}}{}' \varphi T_1 \leq W \qquad 1 \in \mathsf{pol}^-(I) \qquad \Delta \Vdash_{\mathsf{q}}{}' W\,\mathbf{implements}\,I<\overline{\varphi V}>}{\Delta \Vdash_{\mathsf{q}} \varphi T_1\,\mathbf{implements}\,I<\overline{\varphi V}>}\ \textsc{ent-q-alg-up}$$

End case distinction on the last rule used in the derivation of (B.1.23).

End case distinction on whether or not $\mathscr{M}_{21} = \emptyset$.

This finishes the proof of $\Delta \Vdash_{\mathsf{q}} \varphi \mathcal{Q}$.

End case distinction on the last rule used in $\mathcal{D}_4$. □

Lemma B.1.22. *If* $\mathcal{R} \in \mathsf{sup}(T\,\mathbf{implements}\,L)$ *then* $\mathcal{R} = T\,\mathbf{implements}\,L'$ *with* $L \trianglelefteq_{\mathsf{i}} L'$.

Proof. We proceed by induction on the derivation of $\mathcal{R} \in \mathsf{sup}(T\,\mathbf{implements}\,L)$.
Case distinction on the last rule of the derivation of $\mathcal{R} \in \mathsf{sup}(T\,\mathbf{implements}\,L)$.

- *Case* rule SUP-REFL: Obvious.
- *Case* rule SUP-STEP: We have

$$\dfrac{\mathbf{interface}\ I<\overline{X}>\,[\overline{Y}\,\mathbf{where}\,\overline{S}]\ \ldots \qquad \overline{U}\,\mathbf{implements}\,I<\overline{V}> \in \mathsf{sup}(T\,\mathbf{implements}\,L)}{[\overline{V/X}, \overline{U/Y}]S_j \in \mathsf{sup}(T\,\mathbf{implements}\,L)}$$

with $\mathcal{R} = [\overline{V/X}, \overline{U/Y}]S_j$. Applying the I.H. yields

$$\begin{gathered}\overline{U}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} = T\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \\ L \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{V}\texttt{>}\end{gathered}$$

Hence,

$$\overline{Y} = Y$$

By criterion WF-IFACE-2 we have

$$\begin{gathered}S_j = Y\,\textbf{implements}\,K \\ Y \notin \mathsf{ftv}(K)\end{gathered}$$

Hence,

$$[\overline{V/X}, \overline{U/Y}]S_j = T\,\textbf{implements}\,[\overline{V/X}]K$$

Moreover,

$$I\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathrm{i}} [\overline{V/X}]K$$

Hence, with Lemma B.1.4

$$L \trianglelefteq_{\mathrm{i}} [\overline{V/X}]K$$

End case distinction on the last rule of the derivation of $\mathcal{R} \in \mathsf{sup}(T\,\textbf{implements}\,L)$. □

Lemma B.1.23. *If $\mathcal{S} \in \mathsf{sup}(R)$ then there exists an* **implements** *constraint S with $\mathcal{S} = S$.*

Proof. By induction on the derivation of $\mathcal{S} \in \mathsf{sup}(R)$. The case where the derivation ends with rule SUP-REFL is trivial because $\mathcal{S} = R$. Now suppose that the derivation ends with an application of rule SUP-STEP:

$$\frac{\textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\ [\overline{Y}\ \textbf{where}\ \overline{S}]\ \ldots \qquad \overline{U}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \in \mathsf{sup}(R)}{\underbrace{[\overline{V/X}, \overline{U/Y}]S_k}_{=\mathcal{S}} \in \mathsf{sup}(R)}$$

Suppose $S_k = \overline{G}\,\textbf{implements}\,K$. By using the I.H., we get that there exists $\overline{H}$ such that $\overline{U} = \overline{H}$. From criterion WF-IFACE-2 we then know that $\{[\overline{V/X}, \overline{U/Y}]\overline{G}\} \subseteq \{\overline{H}\}$. Thus, there exists $S = \mathcal{S}$. □

Lemma B.1.24. *If $\mathcal{R} \in \mathsf{sup}(\varphi\mathcal{S})$ then there exists a $\mathcal{R}' \in \mathsf{sup}(\mathcal{S})$ with $\varphi\mathcal{R}' = \mathcal{R}$.*

Proof. By induction on the derivation of $\mathcal{R} \in \mathsf{sup}(\varphi\mathcal{S})$. The case where the derivation ends with rule SUP-REFL is trivial because $\mathcal{R} = \varphi\mathcal{S}$. Now suppose that the derivation ends with an application of rule SUP-STEP:

$$\frac{\textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\ [\overline{Y}\ \textbf{where}\ \overline{R}]\ \ldots \qquad \overline{U}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \in \mathsf{sup}(\varphi\mathcal{S})}{\underbrace{[\overline{V/X}, \overline{U/Y}]R_k}_{=\mathcal{R}} \in \mathsf{sup}(\varphi\mathcal{S})}$$

From the I.H. we get the existence of $\overline{U'}$ and $\overline{V'}$ such that $\overline{U'}\,\mathbf{implements}\,I{<}\overline{V'}{>} \in \mathsf{sup}(\mathcal{S})$ and $\varphi\overline{U'} = \overline{U}$, $\varphi\overline{V'} = \overline{V}$. By rule SUP-STEP we then have

$$[\overline{V'/X}, \overline{U'/Y}]Q_k \in \mathsf{sup}(\mathcal{S})$$

Define $\mathcal{R}' = [\overline{V'/X}, \overline{U'/Y}]R_k$. We then get

$$\varphi\mathcal{R}' = \varphi[\overline{V'/X}, \overline{U'/Y}]R_k \overset{\mathsf{ftv}(R_k)\subseteq\{\overline{X},\overline{Y}\}}{=} [\overline{\varphi V'/X}, \overline{\varphi U'/Y}]R_k = [\overline{V/X}, \overline{U/Y}]R_k = \mathcal{R}$$

as required. □

Lemma B.1.25 (Inversion of quasi-algorithmic entailment). *Suppose* $\Delta \Vdash_{\mathrm{q}} \overline{T}^n\,\mathbf{implements}\,I{<}\overline{V}{>}$*. Then there exist* $\overline{U}^n$ *such that* $\Delta \Vdash_{\mathrm{q}}{}' \,\overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>}$*, and for all* $i \in [n]$*,* $\Delta \vdash_{\mathrm{q}}{}' \,T_i \leq U_i$ *and* $i \in \mathsf{pol}^-(I)$ *unless* $T_i = U_i$*.*

Proof. The derivation of $\Delta \Vdash_{\mathrm{q}} \overline{T}^n\,\mathbf{implements}\,I{<}\overline{V}{>}$ must end with ENT-Q-ALG-UP. The claim now follows from the premises of this rule. □

Lemma B.1.26. *Suppose* $\mathcal{D} :: \overline{V}\,\mathbf{implements}\,J{<}\overline{W}{>} \in \mathsf{sup}(\overline{T}\,\mathbf{implements}\,I{<}\overline{U}{>})$ *and* $\Delta \vdash_{\mathrm{q}}{}' \,T_i \leq T'_i$ *with* $T_i = T'_i$ *unless* $i \in \mathsf{pol}^-(I)$ *for all* i*. Then there exist* $\overline{V'}$ *such that* $\overline{V'}\,\mathbf{implements}\,J{<}\overline{W}{>} \in \mathsf{sup}(\overline{T'}\,\mathbf{implements}\,I{<}\overline{U}{>})$ *and* $\Delta \vdash_{\mathrm{q}}{}' \,V_i \leq V'_i$ *with* $V_i = V'_i$ *unless* $i \in \mathsf{pol}^-(J)$ *for all* i*.*

Proof. By induction on $\mathcal{D}$. If the last rule of this derivation is SUP-REFL, then choose $\overline{V'} = \overline{T'}$ and the claim holds trivially. Now suppose the last rule of the derivation is SUP-STEP:

$$\frac{\mathbf{interface}\ I'{<}\overline{X}{>}\ [\overline{Y}^n\ \mathbf{where}\ \overline{R}]\ \ldots \qquad \overline{T''}^n\,\mathbf{implements}\,I'{<}\overline{U'}{>} \in \mathsf{sup}(\overline{T}\,\mathbf{implements}\,I{<}\overline{U}{>})}{[\overline{U'/X}, \overline{T''/Y}]R_k \in \mathsf{sup}(\overline{T}\,\mathbf{implements}\,\overline{U})}$$

with

$$[\overline{U'/X}, \overline{T''/Y}]R_k = \overline{V}\,\mathbf{implements}\,J{<}\overline{W}{>}$$

$$R_k = \overline{G}^m\,\mathbf{implements}\,J{<}\overline{W'}{>} \tag{B.1.24}$$

Applying the I.H. to $\overline{T''}^n\,\mathbf{implements}\,I'{<}\overline{U'}{>} \in \mathsf{sup}(\overline{T}\,\mathbf{implements}\,I{<}\overline{U}{>})$ yields the existence of $\overline{T'''}^n$ such that

$$\overline{T'''}\,\mathbf{implements}\,I'{<}\overline{U'}{>} \in \mathsf{sup}(\overline{T'}\,\mathbf{implements}\,I{<}\overline{U}{>})$$

$$(\forall j \in [n])\ \Delta \vdash_{\mathrm{q}}{}' \,T''_j \leq T'''_j$$

$$(\forall j \in [n])\ T''_j = T'''_j \text{ or } j \in \mathsf{pol}^-(I')$$

We then have by SUP-STEP

$$[\overline{U'/X}, \overline{T'''/Y}]R_k \in \mathsf{sup}(\overline{T'}\,\mathbf{implements}\,I{<}\overline{U}{>}) \tag{B.1.25}$$

Suppose $j \in [n]$ such that $T'''_j \neq T''_j$. Then we have $j \in \mathsf{pol}^-(I')$. By examining the definition of pol^-, we get $Y_j \in \mathsf{pol}^-(R_k)$. The definition of pol^- now gives us

$$Y_j \notin \mathsf{ftv}(\overline{W'}) \tag{B.1.26}$$

$$(\forall i \in [m])\ (Y_j = G_i \text{ and } i \in \mathsf{pol}^-(J)) \text{ or } Y_j \notin \mathsf{ftv}(G_i) \tag{B.1.27}$$

Thus, we have with (B.1.26) that

$$[\overline{U'/X}, \overline{T'''/Y}]\overline{W'} = [\overline{U'/X}, \overline{T''/Y}]\overline{W'} = \overline{W} \tag{B.1.28}$$

Now define

$$\overline{V'}^m = [\overline{U'/X}, \overline{T'''/Y}]\overline{G}$$

Then we have with (B.1.24), (B.1.25), and (B.1.28) that

$$\overline{V'}\,\mathbf{implements}\,J\texttt{<}\overline{W}\texttt{>} \in \mathsf{sup}(\overline{T'}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>})$$

Suppose $i \in [m]$ and $V_i \neq V'_i$. Then there exists $j \in [n]$ such that $Y_j \in \mathsf{ftv}(G_i)$ and $T'''_j \neq T''_j$. By (B.1.27) we then have $Y_j = G_i$ and $i \in \mathsf{pol}^-(J)$. Hence, $V_i = T''_j$ and $V'_i = T'''_j$, so $\Delta \vdash_{\mathrm{q}}' V_i \leq V'_i$. □

Lemma B.1.27.

(*i*) *If* $\mathcal{D}_1 :: \Delta \Vdash_{\mathrm{q}} \mathcal{P}$ *and* $\mathcal{Q} \in \mathsf{sup}(\mathcal{P})$*, then* $\Delta \Vdash_{\mathrm{q}} \mathcal{Q}$*.*

(*ii*) *If* $\mathcal{D}_2 :: \Delta \Vdash_{\mathrm{q}}' \mathcal{R}$ *and* $\mathcal{S} \in \mathsf{sup}(\mathcal{R})$*, then* $\Delta \Vdash_{\mathrm{q}} \mathcal{S}$*.*

(*iii*) *If* $\mathcal{D}_3 :: \Delta \vdash_{\mathrm{q}} T \leq K$ *and* $K \trianglelefteq_{\mathrm{i}} L$*, then* $\Delta \vdash_{\mathrm{q}} T \leq L$*.*

Proof. We proceed by induction on the combined height of $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$.

(i) *Case distinction* on the last rule used in $\mathcal{D}_1$.

- *Case* ENT-Q-ALG-EXTENDS: Then

$$\frac{\Delta \vdash_{\mathrm{q}} T \leq U}{\Delta \Vdash_{\mathrm{q}} \underbrace{T\,\mathbf{extends}\,U}_{=\mathcal{P}}}$$

  If U is not an interface type, the $\mathcal{Q} = \mathcal{P}$ and the claim holds trivially. Otherwise $U = K$ for some K and $\mathcal{Q} = T\,\mathbf{extends}\,L$ for some L with $K \trianglelefteq_{\mathrm{i}} L$. By part (iii) of the I.H., we get $\Delta \vdash_{\mathrm{q}} T \leq L$. Hence, $\Delta \Vdash_{\mathrm{q}} \mathcal{Q}$ by ENT-Q-ALG-EXTENDS.

- *Case* ENT-Q-ALG-UP: Then we have

$$\frac{\begin{array}{c}(\forall i)\ \Delta \vdash_{\mathrm{q}}' T_i \leq T'_i \qquad \text{if } T_i \neq T'_i \text{ then } i \in \mathsf{pol}^-(I) \\ \Delta \Vdash_{\mathrm{q}}' \overline{T'}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>}\end{array}}{\Delta \Vdash_{\mathrm{q}} \underbrace{\overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>}}_{=\mathcal{P}}} \tag{B.1.29}$$

  Assume $\mathcal{Q} = \overline{V}\,\mathbf{implements}\,J\texttt{<}\overline{W}\texttt{>}$. With Lemma B.1.26 we get the existence of $\overline{V'}$ such that

$$\overline{V'}\,\mathbf{implements}\,J\texttt{<}\overline{W}\texttt{>} \in \mathsf{sup}(\overline{T'}\,\mathbf{implements}\,I\texttt{<}\overline{U}\texttt{>}) \tag{B.1.30}$$

$$(\forall i)\ \Delta \vdash_{\mathrm{q}}' V_i \leq V'_i \tag{B.1.31}$$

$$(\forall i)\ \text{if } V_i \neq V'_i \text{ then } i \in \mathsf{pol}^-(J) \tag{B.1.32}$$

  Applying part (ii) of the I.H. to (B.1.30) and the last premise in (B.1.29) yields

$$\Delta \Vdash_{\mathrm{q}} \overline{V'}\,\mathbf{implements}\,J\texttt{<}\overline{W}\texttt{>}$$

  Hence

$$\frac{\begin{array}{c}(\forall i)\ \Delta \vdash_{\mathrm{q}}' V'_i \leq V''_i \\ (\forall i)\ \text{if } V'_i \neq V''_i \text{ then } i \in \mathsf{pol}^-(J) \\ \Delta \Vdash_{\mathrm{q}}' \overline{V''}\,\mathbf{implements}\,J\texttt{<}\overline{W}\texttt{>}\end{array}}{\Delta \Vdash_{\mathrm{q}} \overline{V'}\,\mathbf{implements}\,J\texttt{<}\overline{W}\texttt{>}}\ \text{ENT-Q-ALG-UP}$$

With (B.1.31) and Lemma B.1.7 we get $\Delta \vdash_{\mathrm{q}}' V_i \leq V_i''$ for all i. Moreover, if $V_i \neq V_i''$ then either $V_i \neq V_i'$ or $V_i' \neq V_i''$. Hence, noting (B.1.32), we have $i \in \mathsf{pol}^-(J)$ for those i with $V_i \neq V_i''$. By rule ENT-Q-ALG-UP we then get $\Delta \Vdash_{\mathrm{q}} \overline{V}\,\mathbf{implements}\,J\texttt{<}\overline{W}\texttt{>}$ as required.

End case distinction on the last rule used in $\mathcal{D}_1$.

(ii) *Case distinction* on the last rule used in $\mathcal{D}_2$.

- *Case* ENT-Q-ALG-ENV: Then $\mathcal{R} = R$ for some R and $R' \in \Delta$ and $R \in \mathsf{sup}(R')$. With Lemma B.1.23 we know that there exists $S = \mathcal{S}$. Thus, we also have $S \in \mathsf{sup}(R)$. With Lemma B.1.1 we then get $S \in \mathsf{sup}(R')$. Hence, $\Delta \Vdash_{\mathrm{q}}' \mathcal{S}$.
- *Case* ENT-Q-ALG-IMPL: We have

$$\frac{\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{Q}\ \ldots \qquad \Delta \Vdash_{\mathrm{q}} \varphi\overline{Q} \qquad \mathsf{dom}(\varphi) = \overline{X}}{\Delta \Vdash_{\mathrm{q}} \underbrace{\varphi(\overline{N}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>})}_{=\mathcal{R}}} \tag{B.1.33}$$

From Lemma B.1.24 we know that there exists $\mathcal{S}' \in \mathsf{sup}(\overline{N}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>})$ such that $\varphi\mathcal{S}' = \mathcal{S}$. Let $\mathcal{S}' = \overline{T}\,\mathbf{implements}\,J\texttt{<}\overline{U}\texttt{>}$. By criterion WF-IMPL-1 we get that

$$\overline{Q} \Vdash_{\mathrm{q}} \mathcal{S}'$$

Applying part (i) of the I.H. to $\Delta \Vdash_{\mathrm{q}} \varphi\overline{Q}$ in (B.1.33) yields

$$\Delta \Vdash_{\mathrm{q}} \mathcal{Q}' \text{ for all } \mathcal{Q}' \in \mathsf{sup}(\varphi\overline{Q})$$

Using this equation together with Lemma B.1.21 yields

$$\Delta \Vdash_{\mathrm{q}} \varphi\mathcal{S}'$$

as required.

- *Case* ENT-Q-ALG-IFACE: We have

$$\frac{1 \in \mathsf{pol}^+(I) \qquad \mathsf{non\text{-}static}(I) \qquad I\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathrm{i}} K}{\Delta \Vdash_{\mathrm{q}}' \underbrace{I\texttt{<}\overline{V}\texttt{>}\,\mathbf{implements}\,K}_{=\mathcal{R}}}$$

With Lemma B.1.22 we get $\mathcal{S} = I\texttt{<}\overline{V}\texttt{>}\,\mathbf{implements}\,L$ with $K \trianglelefteq_{\mathrm{i}} L$. Lemma B.1.4 yields $I\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathrm{i}} L$. Hence, with ENT-Q-ALG-IFACE, we have $\Delta \Vdash_{\mathrm{q}}' \mathcal{S}$.

End case distinction on the last rule used in $\mathcal{D}_2$.

(iii) *Case distinction* on the last rule used in $\mathcal{D}_3$.

- *Case* SUB-Q-ALG-KERNEL: Then we have $\Delta \vdash_{\mathrm{q}}' T \leq K$ and from $K \trianglelefteq_{\mathrm{i}} L$ we get $\Delta \vdash_{\mathrm{q}}' K \leq L$. Using Lemma B.1.7 we get $\Delta \vdash_{\mathrm{q}}' T \leq L$, from which we get $\Delta \vdash_{\mathrm{q}} T \leq L$ by rule SUB-Q-ALG-KERNEL.
- *Case* SUB-Q-ALG-IMPL: We have

$$\frac{\Delta \vdash_{\mathrm{q}}' T \leq U \qquad \Delta \Vdash_{\mathrm{q}}' U\,\mathbf{implements}\,K}{\Delta \vdash_{\mathrm{q}} T \leq K}$$

With $K \trianglelefteq_{\mathrm{i}} L$ and Lemma B.1.2, we get

$$U\,\mathbf{implements}\,L \in \mathsf{sup}(U\,\mathbf{implements}\,K)$$

Thus, with part (ii) of the I.H. we get

$$\Delta \Vdash_{\mathrm{q}} U\, \mathbf{implements}\, L$$

By Lemma B.1.25 we get the existence of U' such that

$$\Delta \vdash_{\mathrm{q}}{}' \, U \leq U'$$
$$\Delta \Vdash_{\mathrm{q}}{}' \, U'\, \mathbf{implements}\, L$$

By Lemma B.1.7 we then get $\Delta \vdash_{\mathrm{q}}{}' \, T \leq U'$, so the claim follows by using rule ENT-Q-ALG-IMPL.

End case distinction on the last rule used in $\mathcal{D}_3$. □

Corollary B.1.28. *Suppose* $\Delta \Vdash_{\mathrm{q}} \varphi\Delta'$*.*

(*i*) *If* $\Delta' \vdash_{\mathrm{q}} T \leq U$ *then* $\Delta \vdash_{\mathrm{q}} \varphi T \leq \varphi U$*.*

(*ii*) *If* $\Delta' \Vdash_{\mathrm{q}} \mathcal{P}$ *then* $\Delta \Vdash_{\mathrm{q}} \varphi\mathcal{P}$*.*

Proof. Combine Lemma B.1.21 and Lemma B.1.27. □

Lemma B.1.29 (Transitivity of quasi-algorithmic subtyping). *If* $\mathcal{D}_1 :: \Delta \vdash_{\mathrm{q}} T \leq U$ *and* $\mathcal{D}_2 :: \Delta \vdash_{\mathrm{q}} U \leq V$ *then* $\Delta \vdash_{\mathrm{q}} T \leq V$*.*

Proof. Case distinction on the last rules used in the derivations $\mathcal{D}_1$ and $\mathcal{D}_2$.

- *Case* SUB-Q-ALG-KERNEL and SUB-Q-ALG-KERNEL: Then the claim follows with Lemma B.1.7.
- *Case* SUB-Q-ALG-KERNEL and SUB-Q-ALG-IMPL: Then the claim follows with Lemma B.1.11.
- *Case* SUB-Q-ALG-IMPL and SUB-Q-ALG-KERNEL: Then we have $U = K$ for some K. With Lemma B.1.10 we get that either $V = \mathit{Object}$ or $V = L$ for some L with $K \trianglelefteq_{\mathrm{i}} L$.

  If $V = \mathit{Object}$, then the claim follows with SUB-Q-ALG-OBJ and SUB-Q-ALG-KERNEL. Otherwise, $V = L$ for some L with $K \trianglelefteq_{\mathrm{i}} L$. The claim now follows with Lemma B.1.27.
- *Case* SUB-Q-ALG-IMPL and SUB-Q-ALG-IMPL: We then have $U = K$ for some K and $V = L$ for some L. Moreover,

  $$\frac{\Delta \vdash_{\mathrm{q}}{}' \, T \leq T' \qquad \Delta \Vdash_{\mathrm{q}}{}' \, T'\, \mathbf{implements}\, K}{\Delta \vdash_{\mathrm{q}} T \leq K}$$
  $$\frac{\Delta \vdash_{\mathrm{q}}{}' \, K \leq U' \qquad \Delta \Vdash_{\mathrm{q}}{}' \, U'\, \mathbf{implements}\, L}{\Delta \vdash_{\mathrm{q}} K \leq L}$$

  With $\Delta \vdash_{\mathrm{q}}{}' \, K \leq U'$ and Lemma B.1.10 we know that either $U' = \mathit{Object}$ or $U' = K'$ for some K' with $K \trianglelefteq_{\mathrm{i}} K'$. If $U' = \mathit{Object}$, then $\Delta \vdash_{\mathrm{q}}{}' \, T \leq U'$ by SUB-Q-ALG-OBJ, so $\Delta \vdash_{\mathrm{q}} T \leq L$ follows by SUB-Q-ALG-IMPL.

  Now suppose $U' = K'$ for some K' with $K \trianglelefteq_{\mathrm{i}} K'$. By Lemma B.1.8 and the fact that $\Delta \Vdash_{\mathrm{q}}{}' \, U'\, \mathbf{implements}\, L$ we have $K' \trianglelefteq_{\mathrm{i}} L$. Hence, with Lemma B.1.4 $K \trianglelefteq_{\mathrm{i}} L$. With $\Delta \vdash_{\mathrm{q}} T \leq K$ and Lemma B.1.27 we then get $\Delta \vdash_{\mathrm{q}} T \leq L$ as required.

End case distinction on the last rules used in the derivations $\mathcal{D}_1$ and $\mathcal{D}_2$. □

Lemma B.1.30. *If* $\Delta \vdash_{\mathrm{q}}{}' \, T \leq U$ *and* $\Delta \Vdash_{\mathrm{q}}{}' \, U\, \mathbf{implements}\, K$ *and* $K \trianglelefteq_{\mathrm{i}} I{<}\overline{V}{>}$ *and* $1 \in \mathsf{pol}^{-}(I)$*, then* $\Delta \Vdash_{\mathrm{q}} T\, \mathbf{implements}\, I{<}\overline{V}{>}$*.*

Proof. With $K \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{V}\texttt{>}$ and Lemma B.1.2 we have

$$U\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \in \mathsf{sup}(U\,\mathbf{implements}\,K)$$

Hence, with Lemma B.1.27 we have

$$\Delta \Vdash_{\mathrm{q}} U\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$$

By Lemma B.1.25 we then get the existence of U' with

$$\begin{gathered}\Delta \vdash_{\mathrm{q}}{}' \; U \leq U' \\ \Delta \Vdash_{\mathrm{q}}{}' \; U'\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\end{gathered}$$

By Lemma B.1.7 we have $\Delta \vdash_{\mathrm{q}}{}' \; T \leq U'$, so with $1 \in \mathsf{pol}^{-}(I)$ and rule ENT-Q-ALG-UP, we get $\Delta \Vdash_{\mathrm{q}} T\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$. □

Lemma B.1.31. *If* $\Delta \Vdash_{\mathrm{q}} \overline{T}^{n-1}\,U'\,\overline{V}\,\mathbf{implements}\,I\texttt{<}\overline{W}\texttt{>}$ *and* $n \in \mathsf{pol}^{-}(I)$ *and* $\Delta \vdash_{\mathrm{q}}{}' \; U \leq U'$*, then* $\Delta \Vdash_{\mathrm{q}} \overline{T}^{n-1}\,U\,\overline{V}\,\mathbf{implements}\,I\texttt{<}\overline{W}\texttt{>}$*.*

Proof. From $\Delta \Vdash_{\mathrm{q}} \overline{T}^{n-1}\,U'\,\overline{V}\,\mathbf{implements}\,I\texttt{<}\overline{W}\texttt{>}$ we get with Lemma B.1.25 the existence of $\overline{T'}^{n-1}\,U''\,\overline{V'}$ such that

$$\begin{gathered}(\forall i)\ \Delta \vdash_{\mathrm{q}}{}' \; T_i \leq T'_i \\ (\forall i)\ \text{if } T_i \neq T'_i \text{ then } i \in \mathsf{pol}^{-}(I) \\ (\forall i)\ \Delta \vdash_{\mathrm{q}}{}' \; V_i \leq V'_i \\ (\forall i)\ \text{if } V_i \neq V'_i \text{ then } n+i \in \mathsf{pol}^{-}(I) \\ \Delta \vdash_{\mathrm{q}}{}' \; U' \leq U'' \\ \Delta \Vdash_{\mathrm{q}}{}' \; \overline{T'}^{n-1}\,U''\,\overline{V'}\,\mathbf{implements}\,I\texttt{<}\overline{W}\texttt{>}\end{gathered}$$

With Lemma B.1.7 we then have

$$\Delta \vdash_{\mathrm{q}}{}' \; U \leq U''$$

Because $n \in \mathsf{pol}^{-}(I)$ we can apply rule ENT-Q-ALG-UP and get

$$\Delta \Vdash_{\mathrm{q}} \overline{T}^{n-1}\,U\,\overline{V}\,\mathbf{implements}\,I\texttt{<}\overline{W}\texttt{>}$$

as required. □

Lemma B.1.32. *Suppose* $\Delta \Vdash_{\mathrm{q}} \overline{T}^{n}\,\mathbf{implements}\,I\texttt{<}\overline{W}\texttt{>}$ *and* $[n] = \mathscr{N}_1 \mathbin{\dot\cup} \mathscr{N}_2$ *such that* $T_i = K_i$ *for all* $i \in \mathscr{N}_1$ *and* $T_i = G_i$ *for all* $i \in \mathscr{N}_2$*. Then one of the following holds:*

- $\Delta \Vdash_{\mathrm{q}} \overline{U}^{n}\,\mathbf{implements}\,I\texttt{<}\overline{W}\texttt{>}$ *for any* $\overline{U}$ *with* $U_i = G_i$ *for all* $i \in \mathscr{N}_2$*. Moreover,* $i \in \mathsf{pol}^{-}(I)$ *for all* $i \in \mathscr{N}_1$*.*
- $\mathscr{N}_1 = \{1\}$*,* $\mathscr{N}_2 = \emptyset$*,* $1 \in \mathsf{pol}^{+}(I)$*, and* $K_1 \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{W}\texttt{>}$*. Moreover, if* $1 \notin \mathsf{pol}^{-}(I)$ *then, if* $K_1 = J\texttt{<}\overline{W'}\texttt{>}$*,* $1 \in \mathsf{pol}^{+}(J)$

Proof. From $\Delta \Vdash_{\mathrm{q}} \overline{T}^n\ \textbf{implements}\ I{<}\overline{W}{>}$ we get with Lemma B.1.25 the existence of $\overline{T'}^n$ such that

$$(\forall i \in [n])\ \Delta \vdash_{\mathrm{q}}{}' T_i \leq T'_i$$

$$(\forall i \in [n])\ \text{if } T_i \neq T'_i \text{ then } i \in \mathsf{pol}^-(I) \tag{B.1.34}$$

$$\Delta \Vdash_{\mathrm{q}}{}' \overline{T'}^n\ \textbf{implements}\ I{<}\overline{W}{>} \tag{B.1.35}$$

With Lemma B.1.10 we know for all $i \in \mathcal{N}_1$ that either $T'_i = K'_i$ for some K'_i with $K_i \trianglelefteq_{\mathrm{i}} K'_i$ or $T'_i = \mathit{Object}$.

- Assume there exists some $i \in \mathcal{N}_1$ such that $T'_i = K'_i$ for some K'_i. Then the derivation of $\Delta \Vdash_{\mathrm{q}}{}' \overline{T'}^n\ \textbf{implements}\ I{<}\overline{W}{>}$ must end with rule ENT-Q-ALG-IFACE. Hence:

  $$[n] = \{1\}$$
  $$\mathcal{N}_1 = \{1\}$$
  $$\mathcal{N}_2 = \emptyset$$
  $$T'_1 = J{<}\overline{W'}{>}\ (= K'_1)$$
  $$J{<}\overline{W'}{>} \trianglelefteq_{\mathrm{i}} I{<}\overline{W}{>}$$
  $$1 \in \mathsf{pol}^+(J)$$

  With $K_1 \trianglelefteq_{\mathrm{i}} K'_1$ we then also have $K_1 \trianglelefteq_{\mathrm{i}} I{<}\overline{W}{>}$. With Lemma B.1.18 then $1 \in \mathsf{pol}^+(I)$.

- Assume $T'_i = \mathit{Object}$ for all $i \in \mathcal{N}_1$. Because $T_i = K_i \neq \mathit{Object}$ we have $i \in \mathsf{pol}^-(I)$ by (B.1.34). Let $\overline{U}^n$ be given with $U_i = G_i$ for all $i \in \mathcal{N}_2$. Then

  $$(\forall i \in [n])\ \Delta \vdash_{\mathrm{q}}{}' U_i \leq T'_i$$
  $$(\forall i \in [n])\ \text{if } U_i \neq T'_i \text{ then } i \in \mathsf{pol}^-(I)$$

  With (B.1.35) and rule ENT-Q-ALG-UP we then have $\Delta \Vdash_{\mathrm{q}} \overline{U}^n\ \textbf{implements}\ I{<}\overline{W}{>}$.

  Finally, suppose $1 \notin \mathsf{pol}^-(I)$. Then $T_1 = T'_1$ by (B.1.34), so $K_1 = K'_1 = J{<}\overline{W'}{>}$ and $1 \in \mathsf{pol}^+(J)$ as required. □

Proof of Theorem 3.12. We proceed by induction on the combined height of the derivations of $\Delta \Vdash \mathcal{P}$ and $\Delta \vdash T \leq U$.

(i) *Case distinction* on the last rule used in the derivation of $\Delta \Vdash \mathcal{P}$.

- *Case* ENT-EXTENDS: Follows with part (ii) of the I.H.
- *Case* ENT-ENV: With rules SUP-REFL and ENT-Q-ALG-ENV we have $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{P}$. The claim then follows from Lemma B.1.17.
- *Case* ENT-SUPER: Then we have

  $$\frac{\textbf{interface}\ I{<}\overline{X}{>}\ [\overline{Y}\ \textbf{where}\ \overline{R}]\ \ldots \qquad \Delta \Vdash \overline{U}\ \textbf{implements}\ I{<}\overline{T}{>}}{\Delta \Vdash \underbrace{[\overline{T/X}, \overline{U/Y}]R_i}_{=\mathcal{P}}}$$

  We get by part (i) of the I.H.

  $$\Delta \Vdash_{\mathrm{q}} \overline{U}\ \textbf{implements}\ I{<}\overline{T}{>}$$

  By looking at the rules defining sup, we also have

  $$\mathcal{P} \in \mathsf{sup}(\overline{U}\ \textbf{implements}\ I{<}\overline{T}{>})$$

  The claim $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ now follows from Lemma B.1.27.

- *Case* ENT-IMPL: We have

$$\frac{\textbf{implementation}<\overline{X}>\ I<\overline{T}>\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots \qquad \Delta \Vdash [\overline{U/X}]\overline{P}}{\Delta \Vdash \underbrace{[\overline{U/X}](\overline{N}\,\textbf{implements}\,I<\overline{T}>)}_{=\mathcal{P}}}$$

  By part (i) of the I.H. we get $\Delta \Vdash_{\mathrm{q}} [\overline{U/X}]\overline{P}$. With rule ENT-Q-ALG-IMPL we then have $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{P}$. The claim now follows with Lemma B.1.17.

- *Case* ENT-UP: We have

$$\frac{\Delta \vdash U \le U' \qquad \Delta \Vdash \overline{T}\,U'\,\overline{V}\,\textbf{implements}\,I<\overline{W}> \qquad n \in \mathsf{pol}^-(I)}{\Delta \Vdash \underbrace{\overline{T}^{n-1}\,U\,\overline{V}^{m}\,\textbf{implements}\,I<\overline{W}>}_{=\mathcal{P}}} \tag{B.1.36}$$

  Applying part (i) of the I.H. yields

$$\Delta \Vdash_{\mathrm{q}} \overline{T}\,U'\,\overline{V}\,\textbf{implements}\,I<\overline{W}> \tag{B.1.37}$$

  and part (ii) yields

$$\Delta \vdash_{\mathrm{q}} U \le U' \tag{B.1.38}$$

  Case distinction on the last rule used in the derivation of (B.1.38).

  - *Case* rule SUB-Q-ALG-KERNEL: Then $\Delta \vdash_{\mathrm{q}}{}' U \le U'$. Lemma B.1.31 now yields $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ as required.
  - *Case* rule SUB-Q-ALG-IMPL: Then we have $U' = K$ for some K such that

$$\frac{\Delta \vdash_{\mathrm{q}}{}' U \le U'' \qquad \Delta \Vdash_{\mathrm{q}}{}' U''\,\textbf{implements}\,K}{\Delta \vdash_{\mathrm{q}} U \le K}$$

    Applying Lemma B.1.32 to (B.1.37) with $U' = K$ yields that either $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ (we are done in this case) or that $n = 1$, $m = 0$, and $K \trianglelefteq_{\mathrm{i}} I<\overline{W}>$. With $\Delta \vdash_{\mathrm{q}}{}' U \le U''$, $\Delta \Vdash_{\mathrm{q}}{}' U''\,\textbf{implements}\,K$, $K \trianglelefteq_{\mathrm{i}} I<\overline{W}>$, $1 \in \mathsf{pol}^-(I)$ (follows from (B.1.36)), and Lemma B.1.30 we get

$$\Delta \Vdash_{\mathrm{q}} U\,\textbf{implements}\,I<\overline{W}>$$

    as required.

  End case distinction on the last rule used in the derivation of (B.1.38).

- *Case* ENT-IFACE: We then have $\mathcal{P} = I<\overline{T}>\textbf{implements}\,I<\overline{T}>$, $1 \in \mathsf{pol}^+(I)$, and $\mathsf{non\text{-}static}(I)$. Hence, the claim follows with rule ENT-Q-ALG-IFACE.

End case distinction on the last rule used in the derivation of $\Delta \Vdash \mathcal{P}$.

(ii) *Case distinction* on the last rule used in the derivation of $\Delta \vdash T \le U$.

- *Case* SUB-REFL: Follows with Lemma B.1.6 and rule SUB-Q-ALG-KERNEL.
- *Case* SUB-OBJECT: Follows with rules SUB-Q-ALG-OBJ and SUB-Q-ALG-KERNEL.
- *Case* SUB-TRANS: Follows with Lemma B.1.29.
- *Case* SUB-VAR: Then we have $T = X$ for some X and $X\,\textbf{extends}\,U \in \Delta$. If $U = X$ or $U = \mathit{Object}$, then the claim follows using rules SUB-Q-ALG-VAR-REFL or SUB-Q-ALG-OBJ, respectively, together with rule SUB-Q-ALG-KERNEL. Otherwise, rule SUB-Q-ALG-VAR is applicable (note Lemma B.1.6), so the claim follows with SUB-Q-ALG-KERNEL.

- *Case* SUB-CLASS: Follows with SUB-Q-ALG-CLASS or SUB-Q-ALG-OBJ.
- *Case* SUB-IFACE: Follows with SUB-Q-ALG-IFACE.
- *Case* SUB-IMPL: Then

$$\frac{\Delta \Vdash T\,\mathbf{implements}\,K}{\Delta \vdash T \leq \underbrace{K}_{=U}}$$

Applying the I.H. yields $\Delta \Vdash_{\mathrm{q}} T\,\mathbf{implements}\,K$. With Lemma B.1.25 we get the existence of T' such that

$$\Delta \vdash_{\mathrm{q}}{}' T \leq T'$$
$$\Delta \Vdash_{\mathrm{q}}{}' T'\,\mathbf{implements}\,K$$

Using rule SUB-Q-ALG-IMPL we now can derive $\Delta \vdash T \leq K$.

End case distinction on the last rule used in the derivation of $\Delta \vdash T \leq U$. □

B.2 Type Soundness for CoreGI

This section contains the proofs of Theorem 3.14 (progress), Theorem 3.15 (preservation for top-level evaluation), and Theorem 3.16 (preservation for proper evaluation), which are necessary to complete the type soundness proof for CoreGI (see Section 3.6.1). The section implicitly assumes that the underlying CoreGI program *prog* is well-formed; that is, $\vdash$ *prog* ok.

B.2.1 Proof of Theorem 3.14

Theorem 3.14 is the progress theorem for CoreGI.

Lemma B.2.1. *$N \trianglelefteq_{\mathbf{c}} M$ if, and only if, $\emptyset \vdash N \leq M$.*

Proof. The two implications are verified separately.

"⇒": The claim is obvious if $M = \mathit{Object}$. Otherwise, it follows using rule SUB-Q-ALG-CLASS, rule SUB-Q-ALG-KERNEL, and Theorem 3.11.

"⇐": By Theorem 3.12 $\Delta \vdash_{\mathrm{q}} N \leq M$, so $\Delta \vdash_{\mathrm{q}}{}' N \leq M$ by Lemma B.1.14. The claim now follows with Lemma B.1.10. □

From now on, we use Lemma B.2.1 implicitly.

Lemma B.2.2. *If $\emptyset \vdash T \leq N$ then either $N = \mathit{Object}$ or $N \neq \mathit{Object}$ and $T = N'$ for some N' with $N' \trianglelefteq_{\mathbf{c}} N$.*

Proof. If $N = \mathit{Object}$ then we are done. Thus, assume $N \neq \mathit{Object}$. With Theorem 3.12 we have $\emptyset \vdash_{\mathrm{q}} T \leq N$, so $\emptyset \vdash_{\mathrm{q}}{}' T \leq N$ with Lemma B.1.14. The claim now follows with Lemma B.1.10. □

Lemma B.2.3. *If $\mathsf{mtype}_{\emptyset}(m^{\mathrm{c}}, N) = {<}\overline{X}^{n}{>}\overline{U\,x}^{m} \to U$ **where** $\overline{\mathcal{P}}$ and $N' \trianglelefteq_{\mathbf{c}} N$ then it holds that $\mathsf{getmdef}^{\mathrm{c}}(m^{\mathrm{c}}, N') = {<}\overline{Y}^{n}{>}\overline{V\,y}^{m} \to V$ **where** $\overline{\mathcal{Q}}\,\{e\}$.*

Proof. We proceed by induction on the derivation of $N' \trianglelefteq_{\mathbf{c}} N$.
Case distinction on the last rule used in the derivation of $N' \trianglelefteq_{\mathbf{c}} N$.

- *Case* rule INH-CLASS-REFL: Then $N' = N$ and the claim follows with criterion WF-CLASS-2 and rule DYN-MDEF-CLASS-BASE.
- *Case* rule INH-CLASS-SUPER: Then

$$\frac{\textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } M \textbf{ where } \overline{P'}\,\{\ldots\,\overline{m : \mathit{mdef}}\,\} \qquad [\overline{T/X}]M \trianglelefteq_{\mathbf{c}} N}{C\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathbf{c}} N}\ \text{INH-CLASS-SUPER}$$

  with $N' = C\texttt{<}\overline{T}\texttt{>}$.
  - Assume $m^{\mathrm{c}} \notin \overline{m}$. We get by the I.H.

    $$\mathsf{getmdef}^{\mathrm{c}}(m^{\mathrm{c}}, [\overline{T/X}]M) = \texttt{<}\overline{Y}^{n}\texttt{>}\,\overline{V\,y}^{m} \to V \textbf{ where } \overline{\mathcal{Q}}\,\{e\}$$

    With $m^{\mathrm{c}} \notin \overline{m}$ we then have

    $$\mathsf{getmdef}^{\mathrm{c}}(m^{\mathrm{c}}, C\texttt{<}\overline{T}\texttt{>}) = \texttt{<}\overline{Y}^{n}\texttt{>}\,\overline{V\,y}^{m} \to V \textbf{ where } \overline{\mathcal{Q}}\,\{e\}$$

    by rule DYN-MDEF-CLASS-SUPER.
  - Assume $m^{\mathrm{c}} \in \overline{m}$. Then

    $$\mathsf{getmdef}^{\mathrm{c}}(m^{\mathrm{c}}, C\texttt{<}\overline{T}\texttt{>}) = \texttt{<}\overline{Y}^{n'}\texttt{>}\,\overline{V\,y}^{m'} \to V \textbf{ where } \overline{\mathcal{Q}}\,\{e\}$$

    and, by rule MTYPE-CLASS,

    $$\mathsf{mtype}_{\Delta}(m^{\mathrm{c}}, C\texttt{<}\overline{T}\texttt{>}) = \texttt{<}\overline{Y}^{n'}\texttt{>}\,\overline{V\,y}^{m'} \to V \textbf{ where } \overline{\mathcal{Q}}\,\{e\}$$

    Because the underlying program is well-typed, we know that method m^{c} of class C correctly overrides method m^{c} of class D, where $N = D\texttt{<}\overline{W}\texttt{>}$. But this implies $n = n'$ and $m = m'$ as required.

End case distinction on the last rule used in the derivation of $N' \trianglelefteq_{\mathbf{c}} N$. □

Lemma B.2.4. *If* $N \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{T}\texttt{>}$ *and* **class** $C\texttt{<}\overline{X}\texttt{>} \ldots \{\overline{U\,f}\ \ldots\}$ *and* $\mathsf{fields}(N) = \overline{V\,g}$*, then* $\overline{V\,g} = \overline{V'\,g'}\,([\overline{T/X}]\overline{U\,f})\,\overline{V''\,g''}$ *for some* V', g', V'', g''*.*

Proof. Straightforward induction on the derivation of $N \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{T}\texttt{>}$. □

Lemma B.2.5. *If* $\mathsf{fields}(N) = \overline{U\,f}^{n}$ *and* $i, j \in [n]$ *with* $i \neq j$*, then* $f_i \neq f_j$*.*

Proof. Follows by induction on the derivation of $\mathsf{fields}(N) = \overline{U\,f}$, using criterion WF-CLASS-1. □

Definition B.2.6. The *depth* of a type T, written $\mathsf{depth}(T)$, is defined as follows:

$$\begin{aligned}
\mathsf{depth}(\mathit{Object}) &= 0\\
\mathsf{depth}(C\texttt{<}\overline{T}\texttt{>}) &= 1 + \mathsf{depth}(N)\\
&\qquad \text{where } \textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } N \ldots\\
\mathsf{depth}(I\texttt{<}\overline{T}\texttt{>}) &= 1\\
&\qquad \text{where } \textbf{interface } I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y} \textbf{ where } \bullet] \ldots\\
\mathsf{depth}(I\texttt{<}\overline{T}\texttt{>}) &= 1 + \mathsf{max}(\{\mathsf{depth}(J\texttt{<}\overline{U}\texttt{>}) \mid \overline{G}\,\textbf{implements}\,J\texttt{<}\overline{U}\texttt{>} \in \overline{R}\})\\
&\qquad \text{where } \textbf{interface } I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y} \textbf{ where } \overline{R}] \ldots\\
\mathsf{depth}(X) &= 1
\end{aligned}$$

Criterion WF-PROG-5 ensures that this definition is proper (i.e., terminates).

Lemma B.2.7. *For all N, there exist $\overline{U}$ and $\overline{f}$ such that* $\mathsf{fields}(N) = \overline{U\,f}$.

Proof. The claim follows by induction on the depth of N. □

Lemma B.2.8. *If $\emptyset \Vdash \overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ then one of the following holds:*

- *There exists an implementation definition*

$$\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots$$

and a substitution $[\overline{U/X}]$ such that $\emptyset \Vdash [\overline{U/X}]\overline{P}$, $\overline{V} = [\overline{U/X}]\overline{V'}$, and $(\forall i)\ \emptyset \vdash T_i \leq [\overline{U/X}]N_i$ with $T_i \neq [\overline{U/X}]N_i$ implying $i \in \mathsf{pol}^-(I)$.

- $\overline{T} = T$ *such that* $\emptyset \vdash T \leq J\texttt{<}\overline{U}\texttt{>}$, $J\texttt{<}\overline{U}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{V}\texttt{>}$, $1 \in \mathsf{pol}^+(J)$, $\mathsf{non\text{-}static}(J)$, *and* $1 \in \mathsf{pol}^-(I)$ *unless* $T = J\texttt{<}\overline{U}\texttt{>}$.

Proof. From $\emptyset \Vdash \overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ we get $\emptyset \Vdash_{\mathrm{q}} \overline{T}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ by Theorem 3.11. By Lemma B.1.25 we then get the existence of $\overline{T'}$ such that

$$\emptyset \Vdash_{\mathrm{q}}{}' \ \overline{T'}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$$
$$(\forall i)\ \emptyset \vdash_{\mathrm{q}}{}' \ T_i \leq T'_i$$
$$(\forall i)\ i \in \mathsf{pol}^-(I)\ \text{unless}\ T_i = T'_i \tag{B.2.1}$$

By Theorem 3.12 and rule SUB-Q-ALG-KERNEL we have

$$(\forall i)\ \emptyset \vdash T_i \leq T'_i \tag{B.2.2}$$

Case distinction on the last rule of the derivation of $\emptyset \Vdash_{\mathrm{q}}{}' \ \overline{T'}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$.

- *Case* rule ENT-Q-ALG-ENV: Impossible.
- *Case* rule ENT-Q-ALG-IMPL: Then

$$\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots$$
$$\emptyset \Vdash_{\mathrm{q}} [\overline{U/X}]\overline{P}$$

with $\overline{V} = [\overline{U/X}]\overline{V'}$ and $\overline{T'} = [\overline{U/X}]\overline{N}$. By Theorem 3.12 we get $\emptyset \Vdash [\overline{U/X}]\overline{P}$. Thus, with (B.2.1) and (B.2.2), we conclude that the first proposition of the lemma holds.

- *Case* rule ENT-Q-ALG-IFACE: Then $\overline{T'} = J\texttt{<}\overline{U}\texttt{>}$, $1 \in \mathsf{pol}^+(J)$, $\mathsf{non\text{-}static}(J)$, and $J\texttt{<}\overline{U}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{V}\texttt{>}$. With (B.2.1) and (B.2.2), it is now easy to see that the second proposition of the lemma holds.

End case distinction on the last rule of the derivation of $\emptyset \Vdash_{\mathrm{q}}{}' \ \overline{T'}\,\mathbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$. □

Lemma B.2.9. *If $\emptyset \vdash N \leq I\texttt{<}\overline{V}\texttt{>}$ then $N \trianglelefteq_{\mathrm{c}} M$ for some M and there exists an*

$$\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,M'\,]\ \mathbf{where}\ \overline{P}\ \ldots$$

and a substitution $[\overline{U/X}]$ such that $\emptyset \Vdash [\overline{U/X}]\overline{P}$, $\overline{V} = [\overline{U/X}]\overline{V'}$, and $M = [\overline{U/X}]M'$.

Proof. From $\emptyset \vdash N \leq I\texttt{<}\overline{V}\texttt{>}$ we get $\emptyset \vdash_{\mathrm{q}} N \leq I\texttt{<}\overline{V}\texttt{>}$ by Theorem 3.12.
Case distinction on the last rule of the derivation of $\emptyset \vdash_{\mathrm{q}} N \leq I\texttt{<}\overline{V}\texttt{>}$.

- *Case* rule SUB-Q-ALG-KERNEL: Then $\emptyset \vdash_{\mathrm{q}}{}' \ N \leq I\texttt{<}\overline{V}\texttt{>}$, which contradicts Lemma B.1.10.

- *Case* rule SUB-Q-ALG-IMPL: Hence

  $$\emptyset \vdash_{\mathrm{q}}{}' N \leq T$$
  $$\emptyset \Vdash_{\mathrm{q}}{}' T\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$$

  By Lemma B.1.10 we have $T = M$ for some M with $N \trianglelefteq_{\mathrm{c}} M$. Moreover, the derivation of $\emptyset \Vdash_{\mathrm{q}}{}' M\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ must end with rule ENT-Q-ALG-IMPL. Inverting this rule and using Theorem 3.11 finishes this case.

End case distinction on the last rule of the derivation of $\emptyset \vdash_{\mathrm{q}} N \leq I\texttt{<}\overline{V}\texttt{>}$. □

Lemma B.2.10. *If $\emptyset \Vdash \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ and there exists j with $\emptyset \vdash M \leq T_j$ for some M, then there exists a definition*

$$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

and a substitution $[\overline{U/X}]$ such that

(i) $\emptyset \Vdash [\overline{U/X}]\overline{P}$;

(ii) $\overline{V} = [\overline{U/X}]\overline{V'}$;

(iii) $\emptyset \vdash M \leq [\overline{U/X}]N_j$;

(iv) *if $j \notin \mathsf{pol}^+(I)$ then $\emptyset \vdash T_j \leq [\overline{U/X}]N_j$ with $T_j \neq [\overline{U/X}]N_j$ implying $j \in \mathsf{pol}^-(I)$;*

(v) *if $j \in \mathsf{pol}^+(I)$ and $j \notin \mathsf{pol}^-(I)$ and $T_j \neq [\overline{U/X}]N_j$, then $\overline{T} = T_j = J\texttt{<}\overline{W}\texttt{>}$ with $J\texttt{<}\overline{W}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{V}\texttt{>}$ and $1 \in \mathsf{pol}^+(J)$;*

(vi) *$(\forall i \neq j)\ \emptyset \vdash T_i \leq [\overline{U/X}]N_i$ with $T_i \neq [\overline{U/X}]N_i$ implying $i \in \mathsf{pol}^-(I)$.*

Proof. By Lemma B.2.8, there are two possibilities. The first of these possibilities implies the existence of a definition

$$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

and a substitution $[\overline{U/X}]$ such that

- $\emptyset \Vdash [\overline{U/X}]\overline{P}$
- $\overline{V} = [\overline{U/X}]\overline{V'}$
- $(\forall i)\ \emptyset \vdash T_i \leq [\overline{U/X}]N_i$ with $T_i \neq [\overline{U/X}]N_i$ implying $i \in \mathsf{pol}^-(I)$.

With $\emptyset \vdash M \leq T_j$ we then also have $\emptyset \vdash M \leq [\overline{U/X}]N_j$ by transitivity of subtyping. Claim (v) also holds because it is impossible to have $j \notin \mathsf{pol}^-(I)$ and $T_j \neq [\overline{U/X}]N_j$ at the same time.

Now assume that the second possibility of Lemma B.2.8 holds. That is,

$$\overline{T} = T$$
$$\emptyset \vdash T \leq J\texttt{<}\overline{W}\texttt{>}$$
$$J\texttt{<}\overline{W}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{V}\texttt{>}$$
$$1 \in \mathsf{pol}^+(J)$$
$$1 \in \mathsf{pol}^-(I)\ \text{unless}\ T = J\texttt{<}\overline{W}\texttt{>}$$

This implies $j = 1$. By transitivity of subtyping, we have $\emptyset \vdash M \leq I\texttt{<}\overline{V}\texttt{>}$ Hence, with Lemma B.2.9, we know that there exists M' such that

$$
\begin{gathered}
M \trianglelefteq_{\mathsf{c}} M' \\
\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,N\,]\ \textbf{where}\ \overline{P}\ \ldots \\
\emptyset \Vdash \overline{[U/X]}\overline{P} \\
\overline{V} = \overline{[U/X]}\overline{V'} \\
M' = \overline{[U/X]}N
\end{gathered}
$$

We then have $\emptyset \vdash M \leq \overline{[U/X]}N$, so claim (iii) holds. Moreover, we get from $1 \in \mathsf{pol}^{+}(J)$ and Lemma B.1.18 that $1 \in \mathsf{pol}^{+}(I)$, so claim (iv) holds. Now assume $1 \notin \mathsf{pol}^{-}(I)$. Then $T = J\texttt{<}\overline{W}\texttt{>}$, so claim (v) holds. Claim (vi) holds trivially. Setting $\overline{N} = N$ finishes the proof. □

Lemma B.2.11. *If $\emptyset \Vdash \overline{T}$ **implements** $I\texttt{<}\overline{V}\texttt{>}$ and interface I contains at least one static method, then there exists a definition*

$$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

such that

- $\emptyset \Vdash \overline{[U/X]}\overline{P}$
- $\overline{V} = \overline{[U/X]}\overline{V'}$
- $(\forall i)\ \emptyset \vdash T_i \leq \overline{[U/X]}N_i$ *with* $T_i \neq \overline{[U/X]}N_i$ *implying* $i \in \mathsf{pol}^{-}(I)$

Proof. By Lemma B.2.8, there are two possibilities. The first of these possibilities directly implies the claim. Now assume that the second possibility holds. That is, $\overline{T} = T$, $\emptyset \vdash T \leq J\texttt{<}\overline{W}\texttt{>}$, $J\texttt{<}\overline{W}\texttt{>} \trianglelefteq_{\mathsf{i}} I\texttt{<}\overline{V}\texttt{>}$, and $\mathsf{non\text{-}static}(J)$. With Lemma B.1.19 we then get $\mathsf{non\text{-}static}(I)$. But this contradicts the assumption that I contains at least one static method. □

Lemma B.2.12. *If $N \trianglelefteq_{\mathsf{c}} N_1$ and $N \trianglelefteq_{\mathsf{c}} N_2$ then either $N_1 \trianglelefteq_{\mathsf{c}} N_2$ or $N_2 \trianglelefteq_{\mathsf{c}} N_1$.*

Proof. By straightforward induction on the combined height of the derivations of $N \trianglelefteq_{\mathsf{c}} N_1$ and $N \trianglelefteq_{\mathsf{c}} N_2$. □

Lemma B.2.13. *Let*

$$
\begin{aligned}
\mathscr{M} = \{(\varphi, &\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}^{l}\,]\ \ldots) \\
&\mid \mathsf{dom}(\varphi) = \overline{X}, (\forall i \in [l])\ M_i^{?} = \mathsf{nil}\ \textit{or}\ M_i^{?} \trianglelefteq_{\mathsf{c}} \varphi N_i\}
\end{aligned}
$$

If $\mathscr{M} \neq \emptyset$, $\mathscr{M}$ finite, and $i \in \mathsf{disp}(I)$ implies $M_i^{?} \neq \mathsf{nil}$ for all $i \in [l]$, then there exist $(\varphi, impl)$ such that $\mathsf{least\text{-}impl}\mathscr{M} = (\varphi, impl)$.

Proof. Assume

$$
\begin{gathered}
\mathscr{M} = \{(\varphi_1, impl_1), \ldots, (\varphi_n, impl_n)\} \\
(\forall i \in [n])\ impl_i = \textbf{implementation}\texttt{<}\overline{X_i}\texttt{>}\ I\texttt{<}\overline{V_i}\texttt{>}\ [\,\overline{N_i}^{l}\,]\ \ldots
\end{gathered}
$$

We then need to show that there exists some $k \in [n]$ such that

$$(\forall i \in [n])\ \varphi_k \overline{N_k}^{l} \trianglelefteq_{\mathsf{c}} \varphi_i \overline{N_i}^{l}$$

We proceed by induction on n.

- $n = 1$. Obvious because class inheritance is reflexive.
- $n > 1$. Assume

$$\mathscr{M}' = \{(\varphi_1, \mathit{impl}_1), \ldots, (\varphi_{n-1}, \mathit{impl}_{n-1})\}$$

such that that $\mathscr{M} = \mathscr{M}' \cup \{(\varphi_n, \mathit{impl}_n)\}$. By the I.H. we know that there exists $k' \in [n-1]$ such that

$$(\forall i \in [n-1])\ \varphi_{k'}\overline{N_{k'}} \trianglelefteq_{\mathbf{c}} \varphi_i\overline{N_i} \tag{B.2.3}$$

Now consider impl_n. We partition $[l]$ into $[l] = \mathscr{L}_1 \,\dot\cup\, \mathscr{L}_2 \,\dot\cup\, \mathscr{L}_3$ (where $\dot\cup$ denotes the *disjoint union* of two sets) such that

$$\begin{array}{ll} (\forall j \in \mathscr{L}_1) & \varphi_n N_{nj} \trianglelefteq_{\mathbf{c}} \varphi_{k'} N_{k'j} \\ (\forall j \in \mathscr{L}_2) & \varphi_n N_{nj} \ntrianglelefteq_{\mathbf{c}} \varphi_{k'} N_{k'j} \text{ but } \varphi_{k'} N_{k'j} \trianglelefteq_{\mathbf{c}} \varphi_n N_{nj} \\ (\forall j \in \mathscr{L}_3) & \varphi_n N_{nj} \ntrianglelefteq_{\mathbf{c}} \varphi_{k'} N_{k'j} \text{ and } \varphi_{k'} N_{k'j} \ntrianglelefteq_{\mathbf{c}} \varphi_n N_{nj} \end{array} \tag{B.2.4}$$

We first show that $j \in \mathscr{L}_3$ implies $j \notin \mathsf{disp}(I)$. For the sake of a contradiction, assume $j \in \mathscr{L}_3$ and $j \in \mathsf{disp}(I)$. Then $M_j^? \neq \mathsf{nil}$, so we have

$$\begin{aligned} M_j^? &\trianglelefteq_{\mathbf{c}} \varphi_n N_{nj} \\ M_j^? &\trianglelefteq_{\mathbf{c}} \varphi_{k'} N_{k'j} \end{aligned}$$

By Lemma B.2.12 we then have either $\varphi_n N_{nj} \trianglelefteq_{\mathbf{c}} \varphi_{k'} N_{k'j}$ or $\varphi_{k'} N_{k'j} \trianglelefteq_{\mathbf{c}} \varphi_N N_{nj}$. But this is a contradiction to the definition of $\mathscr{L}_3$. Thus, we have shown that

$$j \in \mathscr{L}_3 \text{ implies } j \notin \mathsf{disp}(I) \tag{B.2.5}$$

Next, we define for $j \in \mathscr{L}_1 \cup \mathscr{L}_2 \cup \mathscr{L}_3$:

$$M_j = \begin{cases} \varphi_n N_{nj} & \text{if } j \in \mathscr{L}_1 \\ \varphi_{k'} N_{k'j} & \text{if } j \in \mathscr{L}_2 \\ \varphi_n N_{nj} & \text{if } j \in \mathscr{L}_3 \end{cases} \tag{B.2.6}$$

We then have by definition of $\mathscr{L}_1$ and $\mathscr{L}_2$ that

$$(\forall j \in \mathscr{L}_1 \cup \mathscr{L}_2)\ \emptyset \vdash \varphi_n N_{nj} \sqcap \varphi_{k'} N_{k'j} = M_j$$

Moreover, from (B.2.5) we have that $j \in \mathsf{disp}(I)$ implies $j \notin \mathscr{L}_3$ which in turn implies $j \in \mathscr{L}_1 \cup \mathscr{L}_2$. Thus, criterion WF-PROG-2 yields $\varphi_n N_{nj} = \varphi_{k'} N_{k'j}$ for all $j \notin \mathsf{disp}(I)$, so we have with (B.2.5) that

$$(\forall j \in \mathscr{L}_3)\ \varphi_n N_{nj} = \varphi_{k'} N_{k'j} \tag{B.2.7}$$

Thus, we have

$$\emptyset \vdash \varphi_n \overline{N_n}^l \sqcap \varphi_{k'} \overline{N_{k'}}^l = \overline{M}^l$$

By criterion WF-PROG-3 we get the existence of a definition

$$\mathit{impl} = \textbf{implementation}\texttt{<}\overline{Y}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{M'}\,]\ \ldots$$

and a substitution ψ with $\mathsf{dom}(\psi) = \overline{Y}$ such that $\psi\overline{M'} = \overline{M}$. By construction of $\overline{M}$, we know that

$$(\psi, \mathit{impl}) \in \mathscr{M} \tag{B.2.8}$$

Moreover, we have for all $i \in [n-1]$, $j \in [l] = \mathscr{L}_1 \mathbin{\dot\cup} \mathscr{L}_2 \mathbin{\dot\cup} \mathscr{L}_3$ that

$$\psi M'_j = M_j \overset{(B.2.6)}{=} \begin{cases} \varphi_n N_{nj} \overset{(B.2.4)}{\trianglelefteq_{\mathbf{c}}} \varphi_{k'} N_{k'j} \overset{(B.2.3)}{\trianglelefteq_{\mathbf{c}}} \varphi_i N_{ij} & \text{if } j \in \mathscr{L}_1 \\ \varphi_{k'} N_{k'j} \overset{(B.2.3)}{\trianglelefteq_{\mathbf{c}}} \varphi_i N_{ij} & \text{if } j \in \mathscr{L}_2 \\ \varphi_n N_{nj} \overset{(B.2.7)}{=} \varphi_{k'} N_{k'j} \overset{(B.2.3)}{\trianglelefteq_{\mathbf{c}}} \varphi_i N_{ij} & \text{if } j \in \mathscr{L}_3 \end{cases}$$

But we also have for all $j \in [l]$ that

$$\psi M'_j = M_j \overset{(B.2.6)}{=} \begin{cases} \varphi_n N_{nj} & \text{if } j \in \mathscr{L}_1 \\ \varphi_{k'} N_{k'j} \overset{(B.2.4)}{\trianglelefteq_{\mathbf{c}}} \varphi_n N_{nj} & \text{if } j \in \mathscr{L}_2 \\ \varphi_n N_{nj} & \text{if } j \in \mathscr{L}_3 \end{cases}$$

Thus,

$$(\forall i \in [n], j \in [l])\ \psi M'_j \trianglelefteq_{\mathbf{c}} \varphi_i N_{ij}$$

Finally, with (B.2.8) and rule LEAST-IMPL, we get

$$\mathsf{least\text{-}impl}\,\mathscr{M} = (\psi, \mathit{impl})$$

□

Lemma B.2.14. *Let*

$$\begin{aligned}\mathscr{M} = \{(\varphi, \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}^{l}\,]\ \ldots) \\ \mid \mathsf{dom}(\varphi) = \overline{X}, (\forall i \in [l])\ N_i = \mathit{Object}\ \mathit{or}\ M_i \trianglelefteq_{\mathbf{c}} \varphi N_i\}\end{aligned}$$

If $\mathscr{M} \neq \emptyset$ and $\mathscr{M}$ finite, then there exist (φ, impl) such that $\mathsf{least\text{-}impl}\,\mathscr{M} = (\varphi, \mathit{impl})$.

Proof. Assume

$$\mathscr{M} = \{(\varphi_1, \mathit{impl}_1), \ldots, (\varphi_n, \mathit{impl}_n)\}$$
$$(\forall i \in [n])\ \mathit{impl}_i = \textbf{implementation}\texttt{<}\overline{X_i}\texttt{>}\ I\texttt{<}\overline{V_i}\texttt{>}\ [\,\overline{N_i}^{l}\,]\ \ldots$$

Then we have for all $i \in [n]$ and all $j \in [l]$ that

$$N_{ij} = \mathit{Object} \text{ or } M_j \trianglelefteq_{\mathbf{c}} \varphi_i N_{ij}$$

Now define

$$\mathscr{L}_1 := \{j \in [l] \mid \text{ there exists } i \in [n], M_j \trianglelefteq_{\mathbf{c}} \varphi_i N_{ij}\}$$
$$\mathscr{L}_2 := [l] \setminus \mathscr{L}_1 = \{j \in [l] \mid \text{ for all } i \in [n], N_{ij} = \mathit{Object}\}$$
$$(\forall j \in [l])\ M'_j = \begin{cases} M_j & \text{if } j \in \mathscr{L}_1 \\ \mathit{Object} & \text{if } j \in \mathscr{L}_2 \end{cases}$$

We now show for

$$\begin{aligned}\mathscr{M}' = \{(\varphi, \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}^{l}\,]\ \ldots) \\ \mid \mathsf{dom}(\varphi) = \overline{X}, (\forall i \in [l])\ M'_i \trianglelefteq_{\mathbf{c}} \varphi N_i\}\end{aligned}$$

that $\mathscr{M} = \mathscr{M}'$. The claim then follows with Lemma B.2.13.

- "$\mathscr{M} \subseteq \mathscr{M}'$". Assume $(\varphi, \mathit{impl}) \in \mathscr{M}$, that is, $(\varphi, \mathit{impl}) = (\varphi_i, \mathit{impl}_i)$ for some $i \in [n]$. Then

  $$(\forall j \in [l])\ M_j' \trianglelefteq_{\mathbf{c}} \varphi_i N_{ij}$$

  by construction of M_j'. Then $(\varphi, \mathit{impl}) \in \mathscr{M}'$.

- "$\mathscr{M} \supseteq \mathscr{M}'$". Assume $(\varphi, \mathit{impl}) \in \mathscr{M}'$ with

  $$\mathit{impl} = \textbf{implementation}<\overline{X}>\ I<\overline{V}>\ [\,\overline{N}^{l}\,]\ \ldots$$

  Then $(\forall i \in [l])\ M_i' \trianglelefteq_{\mathbf{c}} \varphi N_i$. Suppose $j \in [l]$. If $M_j' = \mathit{Object}$ then $N_j = \mathit{Object}$. Otherwise, $M_j' = M_j$, so $M_j \trianglelefteq_{\mathbf{c}} \varphi N_j$. Hence, $(\varphi, \mathit{impl}) \in \mathscr{M}$. □

Lemma B.2.15. *If $\Delta; \Gamma \vdash e : T$ then $\mathcal{D} :: \Delta; \Gamma \vdash e : T'$ with $\Delta \vdash T' \leq T$ such that derivation $\mathcal{D}$ does not end with an application of rule* EXP-SUBSUME.

Proof. Straightforward induction on the derivation of $\Delta; \Gamma \vdash e : T$. □

Lemma B.2.16. *If $\Delta; \Gamma \vdash \textbf{new}\, N(\overline{e}) : T$ then $\Delta \vdash N \leq T$ and $\Delta \vdash N$ ok.*

Proof. By Lemma B.2.15 we have $\mathcal{D} :: \Delta; \Gamma \vdash \textbf{new}\, N(\overline{e}) : T'$ such that $\Delta \vdash T' \leq T$ and $\mathcal{D}$ does not end with rule EXP-SUBSUME. Thus, $\mathcal{D}$ must end with rule EXP-NEW. Inverting the rule yields $T' = N$ and $\Delta \vdash N$ ok □

Lemma B.2.17. *If $M_1 \trianglelefteq_{\mathbf{c}} N$ and $M_2 \trianglelefteq_{\mathbf{c}} N$ then $M_1 \sqcup M_2 \trianglelefteq_{\mathbf{c}} N$.*

Proof. By induction on the derivation of $M_1 \trianglelefteq_{\mathbf{c}} N$.
Case distinction on the last rule of the derivation of $M_1 \trianglelefteq_{\mathbf{c}} N$.

- *Case* rule INH-CLASS-REFL: Then $M_1 = N$ and $M_1 \sqcup M_2 = M_1$ by rule LUB-LEFT, so the claim holds with rule INH-CLASS-REFL.

- *Case* rule INH-CLASS-SUPER: Then

  $$\frac{\textbf{class}\ C<\overline{X}>\ \textbf{extends}\ M_1'\ \ldots \qquad [\overline{T/X}]M_1' \trianglelefteq_{\mathbf{c}} N}{C<\overline{T}> \trianglelefteq_{\mathbf{c}} N}\ \textsc{inh-class-super}$$

  with $M_1 = C<\overline{T}>$. The claim holds obviously if $M_1 \trianglelefteq_{\mathbf{c}} M_2$ or $M_2 \trianglelefteq_{\mathbf{c}} M_1$. Otherwise, we have

  $$M_1 \sqcup M_2 = [\overline{T/X}]M_1' \sqcup M_2$$

  by rule LUB-SUPER. Applying the I.H. yields

  $$[\overline{T/X}]M_1' \sqcup M_2 \trianglelefteq_{\mathbf{c}} N$$

  Hence, the claim also holds.

End case distinction on the last rule of the derivation of $M_1 \trianglelefteq_{\mathbf{c}} N$. □

Lemma B.2.18. *If $M_i \trianglelefteq_{\mathbf{c}} N$ for all $i \in [n]$ with $n > 0$, then $\bigsqcup\{M_1, \ldots, M_n\} \trianglelefteq_{\mathbf{c}} N$.*

Proof. We proceed by induction on n.

- $n = 1$. Then $\bigsqcup\{M_1, \ldots, M_n\} = M_1$ and the claim is obvious.

- $n > 1$. By the I.H. we know that
  $$\bigsqcup\{M_1, \dots, M_{n-1}\} \trianglelefteq_{\mathbf{c}} N$$
  By inverting rule LUB-SET-MULTI we get
  $$\bigsqcup\{M_1, \dots, M_{n-1}\} \sqcup M_n = \bigsqcup\{M_1, \dots, M_n\}$$
  The claim now follows from the assumption $M_n \trianglelefteq_{\mathbf{c}} N$ and Lemma B.2.17. □

Proof of Theorem 3.14. The proof is by induction on the derivation of $\emptyset; \emptyset \vdash e : T$. *Case distinction* on the last rule of the derivation of $\emptyset; \emptyset \vdash e : T$.

- *Case* rule EXP-VAR: Impossible.
- *Case* rule EXP-FIELD: Then
  $$\frac{\emptyset; \emptyset \vdash e_0 : C\texttt{<}\overline{T}\texttt{>} \qquad \textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } N \textbf{ where } \overline{P}\,\{\,\overline{U\,f}\dots\}}{\emptyset; \emptyset \vdash e_0.f_j : [\overline{T/X}]U_j}\ \text{EXP-FIELD}$$
  with $T = [\overline{T/X}]U_j$. Applying the I.H. to $\emptyset; \emptyset \vdash e_0 : C\texttt{<}\overline{T}\texttt{>}$ leaves us with three cases:
  1. $e_0 = v$ for some v. Then $v = \textbf{new } D\texttt{<}\overline{V}\texttt{>}(\overline{v})$ and $\emptyset \vdash D\texttt{<}\overline{V}\texttt{>} \leq C\texttt{<}\overline{T}\texttt{>}$ by Lemma B.2.15. By Lemma B.2.2 then $D\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{T}\texttt{>}$. By Lemma B.2.7, there exists $\overline{W}$ and $\overline{g}$ such that $\mathsf{fields}(D\texttt{<}\overline{V}\texttt{>}) = \overline{W\,g}$. By Lemma B.2.4 and Lemma B.2.5 we know that there exists a unique i such that $W_i\,g_i = [\overline{T/X}]U_j\,f_j$. Hence, $v.f_j \longrightarrow v_i$ by rule DYN-FIELD and rule DYN-CONTEXT.
  2. $e_0 \longrightarrow e_0'$ for some e_0'. It is easy to see that in this case also $e_0.f_j \longrightarrow e_0'.f_j$.
  3. e_0 is stuck on a bad cast. Then $e_0.f_j$ is also stuck on a bad cast.
- *Case* rule EXP-INVOKE: Then
  $$\frac{\begin{array}{l}\emptyset; \emptyset \vdash e_0 : T_0 \qquad \mathsf{mtype}_{\emptyset}(m, T_0) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}} \\ (\forall i \in [n])\ \emptyset; \emptyset \vdash e_i : [\overline{V/X}]U_i \qquad \emptyset \Vdash [\overline{V/X}]\overline{\mathcal{P}} \qquad \emptyset \vdash \overline{V}\ \mathsf{ok}\end{array}}{\emptyset; \emptyset \vdash \underbrace{e_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e}^n)}_{=e} : \underbrace{[\overline{V/X}]U}_{=T}}\ \text{EXP-INVOKE} \tag{B.2.9}$$
  We now apply to I.H. to $\emptyset; \emptyset \vdash e_i : T_i$ (for $i = 0, \dots, n$). This leaves us with three possibilities:
  1. There exist $v_0, \dots, v_n$ such that $e_i = v_i$ for all $i = 0, \dots, n$. We deal with this case shortly.
  2. There exist some $m < n$ and some $v_0, \dots, v_m$ such that $e_i = v_i$ for all $i = 0, \dots, m$, and $e_{m+1} \longrightarrow e'_{m+1}$. It is easy to see that in this case e also makes an evaluation step.
  3. There exist some $m < n$ and some $v_0, \dots, v_m$ such that $e_i = v_i$ for all $i = 0, \dots, m$, and e_{m+1} is stuck on a bad cast. In this case, e is also stuck on a bad cast.

  We now deal with the case that there exist $v_0, \dots, v_n$ such that $e_i = v_i$ for all $i = 0, \dots, n$. Assume
  $$e_i = v_i = \textbf{new } N_i(\overline{w_i}) \quad \text{for } i = 0, \dots, n \tag{B.2.10}$$
  Define $\varphi_1 = [\overline{V/X}]$. By Lemma B.2.15 and (B.2.9) we get
  $$\begin{gathered}\emptyset \vdash N_0 \leq T_0 \\ (\forall i \in [n])\ \emptyset \vdash N_i \leq \varphi_1 U_i\end{gathered}$$
  Case distinction on the form of m.

- *Case* $m = m^{\mathrm{c}}$: From (B.2.9) we get by inverting rule MTYPE-CLASS that $T_0 = C\texttt{<}\overline{T}\texttt{>}$ with $C \neq \mathit{Object}$. By Lemma B.2.2 we have $N_0 \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{T}\texttt{>}$. Hence, with Lemma B.2.3

  $$\mathsf{getmdef}^{\mathrm{c}}(m, N_0) = \texttt{<}\overline{X'}\texttt{>}\,\overline{U'\,x'} \to U' \ \textbf{where}\ \overline{\mathcal{Q}}\,\{e''\}$$

  such that $\overline{X}$ and $\overline{X'}$ as well as $\overline{U\,x}$ and $\overline{U'\,x'}$ have the same length. But then by rule DYN-INVOKE-CLASS

  $$e_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e}^n) \longrightarrow [e_0/\mathit{this}, \overline{e/x'}][\overline{V/X'}]e''$$

- *Case* $m = m^{\mathrm{i}}$: Then we can invert rule MTYPE-IFACE and get

  $$\frac{\begin{array}{c}\textbf{interface}\ I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}^{l}\ \textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{P}\,\{\ldots\ \overline{\mathit{rcsig}}\,\} \\ \mathit{rcsig}_j = \textbf{receiver}\,\{\overline{m : \mathit{msig}}\} \\ \emptyset \Vdash \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{T''}\texttt{>} \qquad m_k = m \qquad T_j = T_0\end{array}}{\mathsf{mtype}_{\emptyset}(m, T_0) = \ \underbrace{[\overline{T/Z}, \overline{T''/Z'}]\mathit{msig}_k}_{=\texttt{<}\overline{X}^{p}\texttt{>}\,\overline{U\,x}^{n} \to U\ \ \textbf{where}\ \overline{\mathcal{P}}}}\ \textsc{mtype-iface} \tag{B.2.11}$$

  Define $\varphi_2 = [\overline{T/Z}, \overline{T''/Z'}]$. By Lemma B.2.10, we get

  $$\textbf{implementation}\texttt{<}\overline{Z''}\texttt{>}\ I\texttt{<}\overline{T'''}\texttt{>}\ [\,\overline{M}\,]\ \textbf{where}\ \overline{Q}\ \ldots \tag{B.2.12}$$
  $$\mathsf{dom}(\varphi_3) = \overline{Z''}$$
  $$\emptyset \Vdash \varphi_3\overline{Q}$$
  $$\overline{T''} = \varphi_3\overline{T'''}$$
  $$\emptyset \vdash N_0 \leq \varphi_3 M_j \tag{B.2.13}$$
  $$j \in \mathsf{pol}^{+}(I) \text{ or } \emptyset \vdash T_j \leq \varphi_3 M_j \tag{B.2.14}$$
  $$(\forall i \neq j)\ \emptyset \vdash T_i \leq \varphi_3 M_i \tag{B.2.15}$$

  Assume

  $$\mathit{msig}_k = \texttt{<}\overline{X}\texttt{>}\,\overline{U'\,x} \to U'\ \textbf{where}\ \overline{P} \tag{B.2.16}$$

  Suppose $i \in [l]$. Then define

  $$M_i^{?} = \begin{cases} \mathsf{resolve}_{Z_i}(\overline{U'}, \overline{N}) & \text{if } i \neq j \\ \mathsf{resolve}_{Z_j}(Z_j\overline{U'}, N_0\overline{N}) & \text{otherwise} \end{cases} \tag{B.2.17}$$

  Our goal is now to prove

  $$(\forall i \in [l])\ M_i^{?} = \mathsf{nil} \text{ or } M_i^{?} \trianglelefteq_{\mathbf{c}} \varphi_3 M_i \tag{B.2.18}$$

  Assume $i \in [l]$ and $M_i^{?} \neq \mathsf{nil}$. We then show $M_i^{?} \trianglelefteq_{\mathbf{c}} \varphi_3 M_i$. First, we define

  $$\mathscr{C}_i = \{N_p \mid p \in [n], U'_p = Z_i\}$$

  and show that $N_p \trianglelefteq_{\mathbf{c}} \varphi_3 M_i$ for all $N_p \in \mathscr{C}_i$. Assume $N_p \in \mathscr{C}_i$. Then $p \in [n]$ and $U'_p = Z_i$. Hence,

  $$U_p = \varphi_2 U'_p = \varphi_2 Z_i = T_i$$

From (B.2.9), we then have

$$\emptyset; \emptyset \vdash e_p : \varphi_1 T_i$$

W.l.o.g., $\overline{X} \cap \mathsf{ftv}(\overline{T}) = \emptyset$, so $\varphi_1 T_i = T_i$. From (B.2.10) we have $e_p = \textbf{new}\ N_p(\overline{w_p})$. Thus, with Lemma B.2.16 we get

$$\emptyset \vdash N_p \leq T_i$$

If $i = j$ then $U'_p = Z_j$, so $j \notin \mathsf{pol}^+(I)$. With (B.2.14) and (B.2.15) we thus have $\emptyset \vdash T_i \leq \varphi_3 M_i$. Hence, by transitivity of subtyping $\emptyset \vdash N_p \leq \varphi_3 M_i$, so

$$N_p \trianglelefteq_{\mathsf{c}} \varphi_3 M_i \text{ for all } N_p \in \mathscr{C}_i \tag{B.2.19}$$

Now we show $M_i^? \trianglelefteq_{\mathsf{c}} \varphi_3 M_i$ depending on whether or not $i = j$.

∗ If $i \neq j$, then, by (B.2.17) and the definition of $\mathsf{resolve}$

$$M_i^? = \bigsqcup \mathscr{C}_i$$

The claim follows from (B.2.19) and Lemma B.2.18.

∗ If $i = j$, then, by (B.2.17) and the definition of $\mathsf{resolve}$

$$M_i^? = \bigsqcup(\{N_0\} \cup \mathscr{C}_i)$$

The claim follows from (B.2.19), (B.2.13), and Lemma B.2.18.

This finishes the prove of (B.2.18)

We now define

$$\begin{aligned} &\mathscr{M} := \\ &\{(\varphi_4, \textbf{implementation}\texttt{<}\overline{Z'''}\texttt{>}\ I\texttt{<}\overline{W'}\texttt{>}\ [\,\overline{M'}\,]\ \textbf{where}\ \overline{Q'}\ \ldots) \\ &\quad \mid \mathsf{dom}(\varphi_4) = \overline{Z'''}, (\forall i \in [l])\ M_i^? = \mathsf{nil} \text{ or } M_i^? \trianglelefteq_{\mathsf{c}} \varphi_4 M'_i\} \end{aligned} \tag{B.2.20}$$

With (B.2.18) we have $(\varphi_3, \mathit{impl}) \in \mathscr{M}$ where impl is the implementation definition from (B.2.12). Clearly, $\mathscr{M}$ is also finite because a program has only finitely many implementation definitions. Moreover, suppose $i \in [l]$, $i \in \mathsf{disp}(I)$. Then either $i = j$ or there exists some argument type $U_{i'}$ with $U_{i'} = Z_i$. In any case, we have with (B.2.17) that $M_i^? \neq \mathsf{nil}$. With Lemma B.2.13 we then get that there exists $(\varphi, \mathit{impl}')$ such that

$$\mathsf{least\text{-}impl}\mathscr{M} = (\varphi, \mathit{impl}') \tag{B.2.21}$$

Assume $\mathit{impl}' = \textbf{implementation} \ldots \{\ldots \overline{\mathit{rcdef}}\}$ Because the underlying program is well-formed, it is easy to check that

$$\begin{gathered} \mathit{rcdef}_j = \textbf{receiver}\,\{\overline{\mathit{mdef}}\} \\ \mathit{mdef}_k = \texttt{<}\overline{X'}^p\texttt{>}\,\overline{U''\ x''}^n \to U''\ \textbf{where}\ \overline{P'}\ \{e''\} \end{gathered} \tag{B.2.22}$$

With (B.2.11), (B.2.16), (B.2.17), (B.2.20), (B.2.21), (B.2.22), and an application of rule DYN-MDEF-IFACE, we get

$$\mathsf{getmdef}^{\mathsf{i}}(m, N_0, \overline{N}) = \varphi\mathit{mdef}_k$$

Hence, with rule DYN-INVOKE-IFACE and DYN-CONTEXT

$$e_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e}^n) \longrightarrow [e_0/\mathit{this}, \overline{e/x''}][\overline{V/X'}]e''$$

End case distinction on the form of m.

- *Case* rule EXP-INVOKE-STATIC: Then

$$\dfrac{\begin{array}{c}\mathsf{smtype}_\emptyset(m, I{<}\overline{V}{>}[\overline{T}]) = {<}\overline{X}^p{>}\,\overline{U\,x}^n \to U\ \mathbf{where}\ \overline{\mathcal{P}} \\ (\forall i)\ \emptyset;\emptyset \vdash e_i : [\overline{W/X}]U_i \qquad \emptyset \Vdash [\overline{W/X}]\overline{\mathcal{P}} \qquad \emptyset \vdash \overline{T}, \overline{W}\ \mathsf{ok}\end{array}}{\emptyset;\emptyset \vdash \underbrace{I{<}\overline{V}{>}[\overline{T}^l].m{<}\overline{W}{>}(\overline{e})}_{=e} : \underbrace{[\overline{V/X}]U}_{=T}}\ \textsc{exp-invoke-static} \tag{B.2.23}$$

We now apply the I.H. to $\emptyset;\emptyset \vdash e_i : [\overline{W/X}]U_i$, for $i = 1, \ldots, n$. As in the case for rule EXP-INVOKE, the only interesting case is the one where

$$(\forall i)\ e_i = v_i = \mathbf{new}\ N_i(\overline{w_i})$$

Define $\varphi_1 = [\overline{W/X}]$. With Lemma B.2.16 we have

$$(\forall i)\ \emptyset \vdash N_i \leq \varphi_1 U_i$$

Inverting rule MTYPE-STATIC yields

$$\dfrac{\begin{array}{c}\mathbf{interface}\ I{<}\overline{Z'}^l{>}\,[\,\overline{Z}\,\mathbf{where}\,\overline{R}\,]\ \mathbf{where}\ \overline{Q}\,\{\,\overline{m : \mathbf{static}\,\mathit{msig}}\ \ldots\} \\ \emptyset \Vdash \overline{T}\,\mathbf{implements}\,I{<}\overline{V}{>} \qquad m = m_k\end{array}}{\mathsf{smtype}_\emptyset(m, I{<}\overline{V}{>}[\overline{T}]) = \underbrace{[\overline{V/Z'}, \overline{T/Z}]}_{=\varphi_2}\,\mathit{msig}_k}\ \textsc{mtype-static}$$

With Lemma B.2.11 we get

$$\begin{gathered}\mathit{impl} = \mathbf{implementation}{<}\overline{Y}{>}\ I{<}\overline{V'}{>}\ [\,\overline{N}^l\,]\ \mathbf{where}\ \overline{Q'}\ \ldots \\ \mathsf{dom}(\varphi_3) = \overline{Y} \\ \emptyset \Vdash \varphi_3\overline{Q'} \\ \overline{V} = \varphi_3\overline{V'} \\ (\forall i \in [l])\ \emptyset \vdash T_i \leq \varphi_3 N_i\end{gathered}$$

With Lemma B.2.2 we then get for all $i \in [l]$

$$N_i = \mathit{Object}\ \text{or}\ T_i = M_i\ \text{for some}\ M_i\ \text{with}\ M_i \trianglelefteq_{\mathsf{c}} \varphi_3 N_i$$

Now define

$$\begin{aligned}\mathscr{M} = \{(\varphi_4, \mathbf{implementation}{<}\overline{Y'}{>}\ I{<}\overline{V''}{>}\ [\,\overline{N'}^l\,]\ \mathbf{where}\ \overline{Q''}\ \ldots) \\ \mid \mathsf{dom}(\varphi_4) = \overline{Y'}, (\forall i \in [l])\ N'_i = \mathit{Object}\ \text{or}\ T_i \trianglelefteq_{\mathsf{c}} \varphi_4 N_i\}\end{aligned}$$

Clearly, $(\varphi_3, \mathit{impl}) \in \mathscr{M}$. Moreover, $\mathscr{M}$ is finite because programs contain only finitely many implementation definitions. Hence, by Lemma B.2.14 we know that there exists $(\varphi, \mathit{impl}')$ such that

$$\mathsf{least\text{-}impl}\,\mathscr{M} = (\varphi, \mathit{impl}')$$

Suppose that $\overline{\mathbf{static}\ \mathit{mdef}}$ are the static methods of impl'. Because the underlying program is well-typed, we know $\mathit{mdef}_k = {<}\overline{X'}^p{>}\,\overline{U'\,x'}^n \to U'\ \mathbf{where}\ \overline{P'}\ \{e''\}$. Hence, we have

$$\mathsf{getsmdef}(m, I{<}\overline{V}{>}[\overline{T}]) = \varphi\,\mathit{mdef}_k$$

by rule DYN-MDEF-STATIC and so

$$I{<}\overline{V}{>}[\overline{T}].m{<}\overline{W}{>}(\overline{e}) \longrightarrow [\overline{e/x'}][\overline{W/X'}]e''$$

by rule DYN-INVOKE-STATIC and rule DYN-CONTEXT.

- *Case* rule EXP-NEW: Then $e = \mathbf{new}\, N(\overline{e}^n)$ and $(\forall i)\ \emptyset; \emptyset \vdash e_i : T_i$. Applying the I.H. yields three possibilities:
  - All e_i are values. Then e is a value.
  - The first m expressions are values ($m < n$) and $e_{m+1} \longrightarrow e'_{m+1}$. Then $e \longrightarrow \mathbf{new}\, N(e_1, \ldots, e_m, e'_{m+1}, e_{m+2}, \ldots, e_n)$.
  - The first m expressions are values ($m < n$) and e_{m+1} is stuck on a bad cast. Then e is stuck on a bad cast as well.
- *Case* rule EXP-CAST: Then

  $$\frac{\emptyset \vdash U \ \mathsf{ok} \qquad \emptyset; \emptyset \vdash e_0 : T}{\emptyset; \emptyset \vdash (U)\, e_0 : U}\ \text{EXP-CAST}$$

  with $e = (U)\, e_0$. Applying the I.H. leaves us with three possibilities:
  - e_0 is a value. Then $e_0 = \mathbf{new}\, M(\overline{v})$. If $\emptyset \vdash M \leq U$ then $e \longrightarrow e_0$ by rules DYN-CAST and DYN-CONTEXT. Otherwise, $\emptyset \vdash U$ ok ensures that U is not a type variable, so e is stuck on a bad cast.
  - $e_0 \longrightarrow e'_0$. Then $e \longrightarrow (U)\, e'_0$ by rule DYN-CONTEXT.
  - e_0 is stuck on a bad cast. Then e is also stuck on a bad cast.
- *Case* rule EXP-SUBSUME: In this case, the claim follows directly from the I.H.

End case distinction on the last rule of the derivation of $\emptyset; \emptyset \vdash e : T$. □

B.2.2 Proof of Theorem 3.15

Theorem 3.15 states that CoreGI's top-level evaluation relation preserves the types of expressions.

Lemma B.2.19. *If $N_1 \sqcup N_2 = M$ then $N_i \trianglelefteq_{\mathsf{c}} M$ for $i = 1, 2$.*

Proof. Straightforward induction on the derivation of $N_1 \sqcup N_2 = M$. □

Lemma B.2.20. *If $N \in \mathscr{N}$ and $M = \bigsqcup \mathscr{N}$ then $N \trianglelefteq_{\mathsf{c}} M$.*

Proof. Straightforward induction on the derivation of $M = \bigsqcup \mathscr{N}$, making use of Lemma B.2.19. □

Lemma B.2.21 (Well-formedness for subterms).

(*i*) *If $\Delta \vdash [U/X]T$ ok and $X \in \mathsf{ftv}(T)$ then $\Delta \vdash U$ ok.*

(*ii*) *If $\Delta \vdash [U/X]\mathcal{P}$ ok and $X \in \mathsf{ftv}(\mathcal{P})$ then $\Delta \vdash U$ ok.*

Proof. We prove both parts by routine inductions on the derivations given. □

Lemma B.2.22 (Type substitution preserves entailment and subtyping). *Suppose $\Delta \Vdash \varphi\Delta'$.*

(*i*) *If $\Delta' \vdash T \leq U$ then $\Delta \vdash \varphi T \leq \varphi U$.*

(*ii*) *If $\Delta' \Vdash \mathcal{P}$ then $\Delta \Vdash \varphi\mathcal{P}$.*

Proof. Follows with Corollary B.1.28, Theorem 3.12, and Theorem 3.11. □

Lemma B.2.23 (Weakening). *Assume $\Delta \subseteq \Delta'$.*

(*i*) *If $\Delta \Vdash \mathcal{P}$ then $\Delta' \Vdash \mathcal{P}$.*

(*ii*) *If* $\Delta \vdash T \leq U$ *then* $\Delta' \vdash T \leq U$.

(*iii*) *If* $\Delta \vdash \mathcal{P}\ \mathsf{ok}$ *then* $\Delta' \vdash \mathcal{P}\ \mathsf{ok}$.

(*iv*) *If* $\Delta \vdash T\ \mathsf{ok}$ *then* $\Delta' \vdash T\ \mathsf{ok}$.

Proof. We prove the first two parts by induction on the combined height of the derivations of $\Delta \Vdash \mathcal{P}$ and $\Delta \vdash T \leq U$. Similarly, we prove the last two parts by induction on the combined height of the derivations of $\Delta \vdash \mathcal{P}\ \mathsf{ok}$ and $\Delta \vdash T\ \mathsf{ok}$. □

In the following, the notation $\mathsf{dom}([\overline{T/X}])$ denotes the *domain* of the type substitution $[\overline{T/X}]$ defined as the set $\{\overline{X}\}$.

Lemma B.2.24 (Type substitution preserves well-formedness). *Suppose* $\Delta \Vdash \varphi\Delta'$ *and* $\Delta \vdash \varphi X\ \mathsf{ok}$ *for all* $X \in \mathsf{dom}(\varphi)$ *and* $\mathsf{dom}(\Delta) \supseteq \mathsf{dom}(\Delta') \setminus \mathsf{dom}(\varphi)$.

(*i*) *If* $\Delta' \vdash T\ \mathsf{ok}$ *then* $\Delta \vdash \varphi T\ \mathsf{ok}$

(*ii*) *If* $\Delta' \vdash \mathcal{P}\ \mathsf{ok}$ *then* $\Delta \vdash \varphi\mathcal{P}\ \mathsf{ok}$

Proof. We proceed by induction on the combined height of the two derivations given.

(i) *Case distinction* on the last rule used in the derivation of $\Delta' \vdash T\ \mathsf{ok}$.

- *Case* rule OK-TVAR: Then $T = X$ and $X \in \mathsf{dom}(\Delta')$.
  - If $X \in \mathsf{dom}(\varphi)$ then $\Delta \vdash \varphi X\ \mathsf{ok}$ by assumption.
  - If $X \notin \mathsf{dom}(\varphi)$ then $X \in \mathsf{dom}(\Delta)$ by assumption. Hence, $\Delta \vdash \varphi X\ \mathsf{ok}$.
- *Case* rule OK-OBJECT: Trivial.
- *Case* rule OK-CLASS: Follows from the I.H., Lemma B.2.22, and the assumption that classes of the underlying program are closed.
- *Case* rule OK-IFACE: Then

$$\frac{\begin{array}{lll} & \mathbf{interface}\ I{<}\overline{X}{>}\ [Y\ \mathbf{where}\ \overline{R}]\ \mathbf{where}\ \overline{P}\ \ldots & \\ \Delta' \vdash \overline{T}\ \mathsf{ok} & Y \notin \mathsf{ftv}(\overline{T}, \Delta') & \Delta', Y\ \mathbf{implements}\ I{<}\overline{T}{>} \Vdash [\overline{T/X}]\overline{R}, \overline{P} \end{array}}{\Delta' \vdash I{<}\overline{T}{>}\ \mathsf{ok}}$$

with $T = I{<}\overline{T}{>}$. By the I.H. we have $\Delta \vdash \varphi\overline{T}\ \mathsf{ok}$. W.l.o.g., $Y \notin \mathsf{ftv}(\varphi\overline{T}, \Delta) \cup \mathsf{dom}(\varphi)$. We get with the assumption $\Delta \Vdash \varphi\Delta'$, an application of Lemma B.2.23, and rule ENT-ENV that

$$\Delta, Y\ \mathbf{implements}\ I{<}\varphi\overline{T}{>} \Vdash \varphi(\Delta', Y\ \mathbf{implements}\ I{<}\overline{T}{>})$$

Lemma B.2.22 now yields

$$\Delta, Y\ \mathbf{implements}\ I{<}\varphi\overline{T}{>} \Vdash \underbrace{\varphi[\overline{T/X}]\overline{R}, \overline{P}}_{=[\overline{\varphi T/X}]\overline{R}, \overline{P}}$$

Hence, by rule OK-IFACE, $\Delta \vdash \varphi I{<}\overline{T}{>}\ \mathsf{ok}$.

End case distinction on the last rule used in the derivation of $\Delta' \vdash T\ \mathsf{ok}$.

(ii) We proceed by case distinction on the last rule used in the derivation of $\Delta' \vdash \mathcal{P}\ \mathsf{ok}$. For rule OK-IMPL-CONSTR, the claim follows with Lemma B.2.22 and the I.H. For rule OK-EXT-CONSTR the claim follows directly from the I.H. □

Lemma B.2.25 (Class inheritance propagates well-formedness). *If* $N \trianglelefteq_{\mathsf{c}} M$ *and* $\Delta \vdash N$ ok *then* $\Delta \vdash M$ ok.

Proof. We proceed by induction on the derivation of $N \trianglelefteq_{\mathsf{c}} M$.
Case distinction on the last rule of the derivation of $N \trianglelefteq_{\mathsf{c}} M$.

- *Case* rule INH-CLASS-REFL: Obvious.
- *Case* rule INH-CLASS-SUPER: Then

$$\frac{\textbf{class } C{<}\overline{X}{>} \textbf{ extends } N' \textbf{ where } \overline{P}\ldots \qquad [\overline{V/X}]N' \trianglelefteq_{\mathsf{c}} M}{\Delta \vdash C{<}\overline{V}{>} \leq M}$$

with $N = C{<}\overline{V}{>}$. Because $\Delta \vdash N$ ok, we have $\Delta \Vdash [\overline{V/X}]\overline{P}$ and $\Delta \vdash \overline{V}$ ok. The underlying program is well-typed, so $\overline{P}, \overline{X} \vdash N'$ ok. With Lemma B.2.24 then $\Delta \vdash [\overline{V/X}]N'$ ok. Applying the I.H. now yields $\Delta \vdash M$ ok.

End case distinction on the last rule of the derivation of $N \trianglelefteq_{\mathsf{c}} M$. □

Lemma B.2.26. *If* **implementation**$\langle\overline{X}\rangle$ $I{<}\overline{V}{>}$ $[\overline{N}^{l}]$ ... *and* $M_i^? \neq \mathsf{nil}$ *for all* $i \in \mathsf{disp}(I)$ *and, for all* $i \in [l]$ *with* $M_i^? \neq \mathsf{nil}$, $\Delta \vdash M_i^?$ ok *and* $M_i^? \trianglelefteq_{\mathsf{c}} [\overline{U/X}]N_i$, *then* $\Delta \vdash \overline{U}$ ok.

Proof. Suppose $i \in [l]$ such that $M_i^? \neq \mathsf{nil}$. Then we get with Lemma B.2.25 that $\Delta \vdash [\overline{U/X}]N_i$ ok. By Lemma B.2.21 we know that $\Delta \vdash U_j$ ok for all j with $X_j \in \mathsf{ftv}(N_i)$. Moreover, by criterion WF-IMPL-2 we have that $\overline{X} \subseteq \mathsf{ftv}\{N_i \mid i \in \mathsf{disp}(I)\}$. Hence, $\Delta \vdash \overline{U}$ ok. □

Lemma B.2.27. *If* **implementation**$\langle\overline{X}\rangle$ $I{<}\overline{V}{>}$ $[\overline{N}^{l}]$ ... *and for all* $i \in [l]$ *either* $N_i = \mathit{Object}$ *or* $M_i \trianglelefteq_{\mathsf{c}} [\overline{U/X}]N_i$ *for some* M_i *with* $\emptyset \vdash M_i$ ok, *then* $\Delta \vdash \overline{U}$ ok.

Proof. The proof is similar to that of Lemma B.2.26. □

Lemma B.2.28. *If* $\bigsqcup \mathscr{N} = M$ *and* $\Delta \vdash N$ ok *for some* $N \in \mathscr{N}$, *then* $\Delta \vdash M$ ok.

Proof. From Lemma B.2.20, we have $N \trianglelefteq_{\mathsf{c}} M$. Because $\Delta \vdash N$ ok we then have $\Delta \vdash M$ ok by Lemma B.2.25. □

Lemma B.2.29 (Type substitution preserves method types). *If* $\mathsf{mtype}_{\Delta'}(m, T) = \mathit{msig}$ *and* $\Delta \Vdash \varphi\Delta'$ *then* $\mathsf{mtype}_{\Delta}(m, \varphi T) = \varphi\mathit{msig}$.

Proof. Follows by case distinction on the rule used to derive $\mathsf{mtype}_{\Delta'}(m, T) = \mathit{msig}$. The case where this rule is MTYPE-IFACE relies on Lemma B.2.22. Moreover, we use the assumption that classes and interfaces of the underlying program are closed. □

Lemma B.2.30 (Type substitution preserves static method types). *If* $\mathsf{smtype}_{\Delta'}(m, K[\overline{T}]) = \mathit{msig}$ *and* $\Delta \Vdash \varphi\Delta'$ *then* $\mathsf{smtype}_{\Delta}(m, \varphi K[\varphi\overline{T}]) = \varphi\mathit{msig}$.

Proof. Follows immediately from Lemma B.2.22 and the assumption that interfaces of the underlying program are closed. □

Lemma B.2.31 (Type substitution preserves fields). *If* $\mathsf{fields}(N) = \overline{T\,f}$ *then* $\mathsf{fields}(\varphi N) = \varphi\overline{T\,f}$.

Proof. Straightforward induction on the derivation of $\mathsf{fields}(N) = \overline{T\,f}$. □

Lemma B.2.32 (Type substitution preserves expression typing). *Assume that* $\Delta \Vdash \varphi\Delta'$ *and* $\Delta \vdash \varphi X$ ok *for all* $X \in \mathsf{dom}(\varphi)$ *and* $\mathsf{dom}(\Delta) \supseteq \mathsf{dom}(\Delta') \setminus \mathsf{dom}(\varphi)$. *If* $\Delta'; \Gamma \vdash e : T$ *then* $\Delta; \varphi\Gamma \vdash \varphi e : \varphi T$.

Proof. We proceed by induction on the derivation of $\Delta'; \Gamma \vdash e : T$.
Case distinction on the last rule of the derivation of $\Delta'; \Gamma \vdash e : T$.

- *Case* rule <span style="font-variant:small-caps">exp-var</span>: Obvious.
- *Case* rule EXP-FIELD: Then

$$\frac{\Delta'; \Gamma \vdash e' : C\texttt{<}\overline{T}\texttt{>} \qquad \textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\,\overline{U\,f}\ldots\}}{\Delta'; \Gamma \vdash e'.f_j : [\overline{T/X}]U_j}\ \textsc{exp-field}$$

with $e = e'.f_j$ and $T = [\overline{T/X}]U_j$. Applying the I.H. yields $\Delta; \varphi\Gamma \vdash \varphi e' : C\texttt{<}\varphi\overline{T}\texttt{>}$. With rule EXP-FIELD we then get $\Delta; \varphi\Gamma \vdash \varphi(e'.f_j) : [\overline{\varphi T/X}]U_j$. Because the underlying program is well-typed, we have $\mathsf{ftv}(U_j) \subseteq \overline{X}$. Hence, $[\overline{\varphi T/X}]U_j = \varphi[\overline{T/X}]U_j = \varphi T$ as required.

- *Case* rule EXP-INVOKE: Then

$$\frac{\begin{array}{c}\Delta'; \Gamma \vdash e' : T' \qquad \mathsf{mtype}_{\Delta'}(m, T') = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}} \\ (\forall i)\ \Delta'; \Gamma \vdash e_i : [\overline{V/X}]U_i \qquad \Delta' \Vdash [\overline{V/X}]\overline{\mathcal{P}} \qquad \Delta' \vdash \overline{V}\ \mathsf{ok}\end{array}}{\Delta'; \Gamma \vdash e'.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V/X}]U}\ \textsc{exp-invoke}$$

with $e = e'.m\texttt{<}\overline{V}\texttt{>}(\overline{e})$ and $T = [\overline{V/X}]U$. From the I.H. we get

$$\Delta; \varphi\Gamma \vdash \varphi e' : \varphi T'$$
$$(\forall i)\ \Delta; \varphi\Gamma \vdash \varphi e_i : \varphi[\overline{V/X}]U_i$$

By Lemma B.2.22 we get

$$\Delta \Vdash \varphi[\overline{V/X}]\overline{\mathcal{P}}$$

By Lemma B.2.24 we get

$$\Delta \vdash \varphi\overline{V}\ \mathsf{ok}$$

W.l.o.g., $\overline{X}$ fresh, so with Lemma B.2.29

$$\mathsf{mtype}_{\Delta}(m, \varphi T') = \texttt{<}\overline{X}\texttt{>}\,\overline{\varphi U\,x} \to \varphi U\ \textbf{where}\ \varphi\overline{\mathcal{P}}$$

With $\overline{X}$ fresh we have $\varphi[\overline{V/X}](\overline{U}, U, \overline{\mathcal{P}}) = [\overline{\varphi V/X}]\varphi(\overline{U}, U, \overline{\mathcal{P}})$, so applying rule EXP-INVOKE yields $\Delta; \varphi\Gamma \vdash \varphi e : [\overline{\varphi V/X}]\varphi U$. But $[\overline{\varphi V/X}]\varphi U = \varphi T$ as required.

- *Case* rule EXP-INVOKE-STATIC: Then

$$\frac{\begin{array}{c}\mathsf{smtype}_{\Delta'}(m, I\texttt{<}\overline{W}\texttt{>}[\overline{T}]) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}} \\ (\forall i)\ \Delta'; \Gamma \vdash e_i : [\overline{V/X}]U_i \qquad \Delta' \Vdash [\overline{V/X}]\overline{\mathcal{P}} \qquad \Delta' \vdash \overline{T}, \overline{V}\ \mathsf{ok}\end{array}}{\Delta'; \Gamma \vdash I\texttt{<}\overline{W}\texttt{>}[\overline{T}].m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V/X}]U}\ \textsc{exp-invoke-static}$$

with $e = I\texttt{<}\overline{W}\texttt{>}[\overline{T}].m\texttt{<}\overline{V}\texttt{>}(\overline{e})$ and $T = [\overline{V/X}]U$. W.l.o.g., $\overline{X}$ fresh. Hence, by Lemma B.2.30

$$\mathsf{smtype}_{\Delta}(m, \varphi I\texttt{<}\overline{W}\texttt{>}[\overline{T}]) = \texttt{<}\overline{X}\texttt{>}\,\overline{\varphi U\,x} \to \varphi U\ \textbf{where}\ \varphi\overline{\mathcal{P}}$$

Moreover, $\varphi[\overline{V/X}](\overline{U}, U, \overline{\mathcal{P}}) = [\overline{\varphi V/X}]\varphi(\overline{U}, U, \overline{\mathcal{P}})$. Applying the I.H. then yields

$$(\forall i)\ \Delta; \varphi\Gamma \vdash \varphi e_i : [\overline{\varphi V/X}]\varphi U_i$$

With Lemma B.2.22 we also have

$$\Delta \Vdash [\overline{\varphi V/X}]\varphi\mathcal{P}$$

Moreover, with Lemma B.2.24

$$\Delta \vdash \varphi(\overline{T}, \overline{V})\ \mathsf{ok}$$

We now get with rule exp-invoke-static that $\Delta; \varphi\Gamma \vdash \varphi e : [\overline{\varphi V/X}]\varphi U$. Noting that $[\overline{\varphi V/X}]\varphi U = \varphi T$ finishes this case.

- *Case* rule exp-new: Follows from the I.H., Lemma B.2.24, and Lemma B.2.31.
- *Case* rule exp-cast: Follows from the I.H. and Lemma B.2.24.
- *Case* rule exp-subsume: Follows from the I.H. and Lemma B.2.22.

End case distinction on the last rule of the derivation of $\Delta'; \Gamma \vdash e : T$. □

Lemma B.2.33. *If* $C\texttt{<}\overline{T}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U}\texttt{>}$ *then, for fresh and pairwise distinct type variables* $\overline{X}$, $C\texttt{<}\overline{X}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U'}\texttt{>}$ *with* $[\overline{T/X}]D\texttt{<}\overline{U'}\texttt{>} = D\texttt{<}\overline{U}\texttt{>}$.

Proof. By induction on the derivation of $C\texttt{<}\overline{T}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U}\texttt{>}$.
Case distinction on the last rule in the derivation of $C\texttt{<}\overline{T}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U}\texttt{>}$.

- *Case* inh-class-refl: Obvious with $\overline{U'} = \overline{X}$.
- *Case* inh-class-super: Then

$$\frac{\textbf{class}\ C\texttt{<}\overline{Y}\texttt{>}\ \textbf{extends}\ C'\texttt{<}\overline{V}\texttt{>}\ldots \qquad [\overline{T/Y}]C'\texttt{<}\overline{V}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U}\texttt{>}}{C\texttt{<}\overline{T}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U}\texttt{>}}$$

By the I.H. there exists $\overline{Z}, \overline{U''}$ with

$$C'\texttt{<}\overline{Z}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U''}\texttt{>}$$
$$[\overline{[\overline{T/Y}]V/Z}]D\texttt{<}\overline{U''}\texttt{>} = D\texttt{<}\overline{U}\texttt{>}$$

We also have for $\varphi = [\overline{X/Y}]$ that $C\texttt{<}\overline{X}\texttt{>} \unlhd_{\mathsf{c}} \varphi C'\texttt{<}\overline{V}\texttt{>}$. From $C'\texttt{<}\overline{Z}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U''}\texttt{>}$ we get with Lemma B.1.12 that $[\overline{\varphi V/Z}]C'\texttt{<}\overline{Z}\texttt{>} \unlhd_{\mathsf{c}} [\overline{\varphi V/Z}]D\texttt{<}\overline{U''}\texttt{>}$. With $[\overline{\varphi V/Z}]C'\texttt{<}\overline{Z}\texttt{>} = \varphi C'\texttt{<}\overline{V}\texttt{>}$ and Lemma B.1.4 we then have

$$C\texttt{<}\overline{X}\texttt{>} \unlhd_{\mathsf{c}} [\overline{\varphi V/Z}]D\texttt{<}\overline{U''}\texttt{>}$$

Moreover,

$$[\overline{T/X}][\overline{\varphi V/Z}]D\texttt{<}\overline{U''}\texttt{>} \overset{\overline{X}\ \text{fresh}}{=} [\overline{[\overline{T/Y}]V/Z}]D\texttt{<}\overline{U''}\texttt{>} = D\texttt{<}\overline{U}\texttt{>}$$

Define $\overline{U'} = [\overline{\varphi V/Z}]\overline{U''}$ to finish the proof.

End case distinction on the last rule in the derivation of $C\texttt{<}\overline{T}\texttt{>} \unlhd_{\mathsf{c}} D\texttt{<}\overline{U}\texttt{>}$. □

Lemma B.2.34.

(*i*) *If* $\Delta \vdash T\ \mathsf{ok}$ *then* $\mathsf{ftv}(T) \subseteq \mathsf{dom}(\Delta)$.

(*ii*) *If* $\Delta \vdash \mathcal{P}\ \mathsf{ok}$ *then* $\mathsf{ftv}(\mathcal{P}) \subseteq \mathsf{dom}(\Delta)$.

Proof. We prove the first claim by induction on the derivation of $\Delta \vdash T\ \mathsf{ok}$. The second claim follows from the first one by inverting the last rule in the derivation of $\Delta \vdash \mathcal{P}\ \mathsf{ok}$. □

Lemma B.2.35. *If* $\Delta; \Gamma, x : T \vdash e : U$ *and* $\Delta \vdash T' \leq T$ *then* $\Delta; \Gamma, x : T' \vdash e : U$*.*

Proof. Straightforward induction on the derivation of $\Delta; \Gamma, x : T \vdash e : U$. □

Lemma B.2.36. *Suppose*

$$\mathsf{mtype}_{\emptyset}(m^{\mathrm{c}}, N) = <\overline{X}>\overline{U\,x} \to U\ \mathbf{where}\ \overline{\mathcal{P}}$$
$$\mathsf{getmdef}^{\mathrm{c}}(m^{\mathrm{c}}, N') = <\overline{X'}>\overline{U'\,x'} \to U'\ \mathbf{where}\ \overline{\mathcal{P}'}\,\{e\}$$

Moreover, assume $\emptyset \vdash N'\ \mathsf{ok}$ *and* $N' \trianglelefteq_{\mathbf{c}} N$ *and* $\emptyset \Vdash \varphi\overline{\mathcal{P}}$ *for some substitution* φ *with* $\mathsf{dom}(\varphi) = \overline{X}$ *and* $\emptyset \vdash \varphi X\ \mathsf{ok}$ *for all* $X \in \mathsf{dom}(\varphi)$*. Then* $\overline{X} = \overline{X'}$*,* $\overline{x} = \overline{x'}$*, and* $\emptyset; \mathit{this} : N', \overline{x : \varphi U} \vdash \varphi e : \varphi U$*.*

Proof. In the following, we write simply m instead of m^{c}. The proof is by induction on the derivation of $\mathsf{getmdef}^{\mathrm{c}}(m, N') = <\overline{X'}>\overline{U'\,x'} \to U'\ \mathbf{where}\ \overline{\mathcal{P}'}\,\{e\}$.
Case distinction on the last rule used in the derivation of $\mathsf{getmdef}^{\mathrm{c}}(m, N')$.

- *Case* rule DYN-MDEF-CLASS-BASE: Then

  $$\frac{\mathbf{class}\ C<\overline{Z}>\ \mathbf{extends}\ M\ \mathbf{where}\ \overline{Q}\,\{\ldots\ \overline{m : \mathit{mdef}}\ \} \qquad m = m_k}{\mathsf{getmdef}^{\mathrm{c}}(m, \underbrace{C<\overline{T}>}_{=N'}) = \underbrace{[\overline{T/Z}]\mathit{mdef}_k}_{=<\overline{X'}>\,\overline{U'\,x'} \to U'\ \mathbf{where}\ \overline{\mathcal{P}'}\,\{e\}}}\ \text{DYN-MDEF-CLASS-BASE} \tag{B.2.24}$$

  Assume

  $$\mathit{mdef}_k = \underbrace{<\overline{X'}>\overline{U''\,x'} \to U''\ \mathbf{where}\ \overline{P''}}_{=\mathit{msig}}\,\{e'\} \tag{B.2.25}$$

  The underlying program is well-typed, so we have

  $$\overline{Q}, \overline{Z} \vdash m_k : \mathit{mdef}_k\ \mathsf{ok\,in}\ C<\overline{Z}>$$

  Hence,

  $$\underbrace{\overline{Q}, \overline{P''}, \overline{Z}, \overline{X'}}_{=\Delta}; \underbrace{\mathit{this} : C<\overline{X}>, \overline{x' : U''}}_{=\Gamma} \vdash e' : U'' \tag{B.2.26}$$
  $$\mathsf{override\text{-}ok}_{\overline{Q},\overline{Z}}(m_k : \mathit{msig}, C<\overline{Z}>) \tag{B.2.27}$$

  Assume $N = D<\overline{V}>$. From $C<\overline{T}> \trianglelefteq_{\mathbf{c}} D<\overline{V}>$ we get with Lemma B.2.33 that

  $$C<\overline{Z}> \trianglelefteq_{\mathbf{c}} D<\overline{W}>$$
  $$[\overline{T/Z}]D<\overline{W}> = D<\overline{V}> \tag{B.2.28}$$

  for some $\overline{W}$. From $\mathsf{mtype}_{\emptyset}(m, D<\overline{V}>) = <\overline{X}>\overline{U\,x} \to U\ \mathbf{where}\ \overline{\mathcal{P}}$ we get

  $$\mathbf{class}\ D<\overline{Z'}>\ \ldots\ \{\ldots\ \overline{m' : \mathit{msig}\{e''\}}\}$$
  $$m = m'_j$$
  $$\mathit{msig}_j = <\overline{X}>\overline{U'''\,x} \to U'''\ \mathbf{where}\ \overline{P'''} \tag{B.2.29}$$
  $$<\overline{X}>\overline{U\,x} \to U\ \mathbf{where}\ \overline{\mathcal{P}} = [\overline{V/Z'}]\mathit{msig}_j \tag{B.2.30}$$

Hence, with criterion WF-CLASS-2

$$\mathsf{mtype}_{\overline{Q},\overline{Z}}(m, D\texttt{<}\overline{W}\texttt{>}) = [\overline{W/Z'}]\mathit{msig}_j$$

From (B.2.24), (B.2.27), and rule OK-OVERRIDE

$$\overline{Q}, \overline{Z} \vdash \mathit{msig} \leq [\overline{W/Z'}]\mathit{msig}_j \tag{B.2.31}$$

Define

$$\begin{aligned} \varphi_1 &= [\overline{T/Z}] \\ \varphi_2 &= [\overline{V/Z'}] \\ \varphi_3 &= [\overline{W/Z'}] \end{aligned}$$

We then have from (B.2.25), (B.2.29) and (B.2.31) that

$$\overline{X} = \overline{X'}$$

$$\overline{x} = \overline{x'}$$

$$\overline{U''} = \varphi_3\overline{U'''} \tag{B.2.32}$$

$$\overline{P''} = \varphi_3\overline{P'''} \tag{B.2.33}$$

$$\Delta \vdash U'' \leq \varphi_3 U''' \tag{B.2.34}$$

From the assumption $\emptyset \vdash C\texttt{<}\overline{T}\texttt{>}\ \mathsf{ok}$ we get that $\emptyset \Vdash \varphi_1\overline{Q}$ (by inverting rule OK-CLASS) and that $\mathsf{ftv}(\overline{T}) = \emptyset$. (by Lemma B.2.34). The underlying program is well-typed, so $\mathsf{ftv}(\overline{Q}) \subseteq \overline{Z}$. Hence, $\varphi\varphi_1\overline{Q} = \varphi_1\overline{Q}$ by definition of φ_1.[1] Thus

$$\emptyset \Vdash \varphi\varphi_1\overline{Q} \tag{B.2.35}$$

We have $\emptyset \Vdash \varphi\overline{\mathcal{P}}$ by assumption. Moreover,

$$\varphi\overline{\mathcal{P}} \overset{(\text{B.2.29}),(\text{B.2.30})}{=} \varphi\varphi_2\overline{P'''} \overset{(\text{B.2.28})}{=} \varphi[\overline{\varphi_1 W/Z'}]\overline{P'''} \overset{\text{w.l.o.g.},\overline{Z}\cap\mathsf{ftv}(\overline{P'''})=\emptyset}{=} \varphi\varphi_1\varphi_3\overline{P'''} \overset{(\text{B.2.33})}{=} \varphi\varphi_1\overline{P''}$$

Hence,

$$\emptyset \Vdash \varphi\varphi_1\overline{P''} \tag{B.2.36}$$

Noting that $\mathsf{ftv}(\overline{T}) = \emptyset$, we see that $\varphi\varphi_1 = [\overline{\varphi X/X}, \overline{T/Z}]$. Thus, with $\overline{X} = \overline{X'}$

$$\mathsf{dom}(\Delta) \setminus \mathsf{dom}(\varphi\varphi_1) = \emptyset$$

Moreover, from $\emptyset \vdash C\texttt{<}\overline{T}\texttt{>}\ \mathsf{ok}$ we have $\emptyset \vdash \overline{T}\ \mathsf{ok}$, so with the assumptions we get

$$\emptyset \vdash \varphi\varphi_1 Y\ \mathsf{ok}\ \text{for all}\ Y \in \mathsf{dom}(\varphi\varphi_1)$$

Hence, we may apply Lemma B.2.32 to (B.2.26) and get

$$\emptyset; \varphi\varphi_1\Gamma \vdash \varphi\varphi_1 e' : \varphi\varphi_1 U'' \tag{B.2.37}$$

[1] For two type substitutions φ and ψ, the notation $\varphi\psi$ denotes the *composition* of φ and ψ where the application of $\varphi\psi$ to some ξ is defined as $\varphi\psi\xi := \varphi(\psi\xi)$.

With $\mathsf{ftv}(\overline{T}) = \emptyset$, we have $\varphi\varphi_1 N' = N'$. Moreover,

$$\varphi\varphi_1 U_i'' \overset{(\text{B.2.32})}{=} \varphi\varphi_1\varphi_3 U_i''' \overset{\text{w.l.o.g.},\overline{Z}\cap\mathsf{ftv}(\overline{U'''})=\emptyset}{=} \varphi[\overline{\varphi_1 W/Z'}]U_i''' \overset{(\text{B.2.28})}{=} \varphi\varphi_2 U_i''' \overset{(\text{B.2.30})}{=} \varphi U_i$$

Hence,

$$\varphi\varphi_1\Gamma = \mathit{this} : N', \overline{x : \varphi U} \tag{B.2.38}$$

We also have from (B.2.24) and (B.2.25)

$$\varphi\varphi_1 e' = \varphi e \tag{B.2.39}$$

With (B.2.34), (B.2.35), (B.2.36) and Lemma B.2.22 we get

$$\emptyset \vdash \varphi\varphi_1 U'' \leq \varphi\varphi_1\varphi_3 U'''$$

We also have

$$\varphi\varphi_1\varphi_3 U''' \overset{\text{w.l.o.g.},\overline{Z}\cap\mathsf{ftv}(U''')=\emptyset}{=} \varphi[\overline{\varphi_1 W/Z'}]U'' \overset{(\text{B.2.28})}{=} \varphi\varphi_2 U''' \overset{(\text{B.2.30})}{=} \varphi U$$

Hence,

$$\emptyset \vdash \varphi\varphi_1 U'' \leq \varphi U$$

With (B.2.37), (B.2.38), (B.2.39), and rule EXP-SUBSUME then

$$\emptyset; \mathit{this} : N', \overline{x : \varphi U} \vdash \varphi e : \varphi U$$

as required.

- *Case* rule DYN-MDEF-CLASS-SUPER: Then

$$\frac{\begin{array}{c}\textbf{class } C\texttt{<}\overline{Z}\texttt{>} \textbf{ extends } M \textbf{ where } \overline{Q}\,\{\ldots\ \overline{m : \mathit{mdef}}\ \} \\ m \notin \overline{m} \qquad \mathsf{getmdef}^{\mathrm{c}}(m, [\overline{T/Z}]M) = \texttt{<}\overline{X'}\texttt{>}\,\overline{U'\,x'} \to U' \textbf{ where } \overline{\mathcal{P}'}\,\{e\}\end{array}}{\mathsf{getmdef}^{\mathrm{c}}(m, C\texttt{<}\overline{T}\texttt{>}) = \texttt{<}\overline{X'}\texttt{>}\,\overline{U'\,x'} \to U' \textbf{ where } \overline{\mathcal{P}'}\,\{e\}}\ \text{DYN-MDEF-CLASS-SUPER}$$

  with $N' = C\texttt{<}\overline{T}\texttt{>}$. Assume $[\overline{T/Z}]M \not\trianglelefteq_{\mathsf{c}} N$. Then, because $N' \trianglelefteq_{\mathsf{c}} N$, we must have $N' = N$. But with $\mathsf{mtype}_{\emptyset}(m^{\mathrm{c}}, N) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}}$ we then have $m \in \overline{m}$, which is a contradiction.

  Thus, $[\overline{T/Z}]M \trianglelefteq_{\mathsf{c}} N$. Obviously also $N' \trianglelefteq_{\mathsf{c}} [\overline{T/Z}]M$, so with Lemma B.2.25 $\emptyset \vdash [\overline{T/Z}]M$ ok. Hence, we may apply the I.H. and get

$$\overline{X} = \overline{X'}$$
$$\overline{x} = \overline{x'}$$
$$\emptyset; \mathit{this} : [\overline{T/Z}]M, \overline{x : \varphi U} \vdash \varphi e : \varphi U$$

  An application of Lemma B.2.35 finishes this case.

End case distinction on the last rule used in the derivation of $\mathsf{getmdef}^{\mathrm{c}}(m, N')$. □

Lemma B.2.37. *If* $\mathsf{fields}(N) = \overline{T\,f}$ *and* $\mathsf{fields}(N) = \overline{U\,g}$ *then* $\overline{T} = \overline{U}$ *and* $\overline{f} = \overline{g}$.

Proof. Straightforward induction on the derivation of $\mathsf{fields}(N) = \overline{T\,f}$. □

Lemma B.2.38 (Expression substitution preserves expression typing). *If* $\Delta;\Gamma, x : T \vdash e : U$ *and* $\Delta;\Gamma : e' : T$ *then* $\Delta;\Gamma \vdash [e'/x]e : U$.

Proof. By induction on the derivation of $\Delta;\Gamma, x : T \vdash e : U$. Assume that the derivation ends with rule EXP-VAR. If $e = x$ then $T = U$ and $[e'/x]e = e'$, so the claim follows from the assumptions. Otherwise, $e = y$ for some $y \neq x$ with $(\Gamma, x : T)(y) = U$. Hence, $\Gamma(y) = U$, so the claim follows with rule EXP-VAR.

If the derivation ends with some other rule, the claim follows from the I.H. □

Proof of Theorem 3.15. The proof is by induction on the derivation of $\emptyset;\emptyset \vdash e : T$. *Case distinction* on the last rule of the derivation of $\emptyset;\emptyset \vdash e : T$.

- *Case* rule EXP-VAR: Impossible.
- *Case* rule EXP-FIELD: Then

  $$\frac{\emptyset;\emptyset \vdash e_0 : C\text{<}\overline{T}\text{>} \qquad \textbf{class}\ C\text{<}\overline{X}\text{>}\ \textbf{extends}\ M\ \textbf{where}\ \overline{P}\,\{\,\overline{U\,f}\dots\}}{\emptyset;\emptyset \vdash e_0.f_j : [\overline{T/X}]U_j}\ \textsc{exp-field}$$

  with $T = [\overline{T/X}]U_j$ and $e = e_0.f_j$. From $e \longmapsto e'$ we get

  $$\begin{aligned} e_0 &= \textbf{new}\ N(\overline{v}) \\ \mathsf{fields}(N) &= \overline{V\,f'} \\ e' &= v_i \\ f'_i &= f_j \end{aligned}$$

  We have by Lemma B.2.15, inspection of the expression typing rules, and Lemma B.2.37 that

  $$\frac{\emptyset;\emptyset \vdash N\ \mathsf{ok} \qquad \mathsf{fields}(N) = \overline{V\,f'} \qquad (\forall i)\ \emptyset;\emptyset \vdash v_i : V_i}{\emptyset;\emptyset \vdash \textbf{new}\ N(\overline{v}) : N}\ \textsc{exp-new}$$

  such that $N \trianglelefteq_{\mathbf{c}} C\text{<}\overline{T}\text{>}$. From Lemma B.2.4 we get $V_i = [\overline{T/X}]U_j$, so $\emptyset;\emptyset \vdash e' : T$ as required.
- *Case* rule EXP-INVOKE: Then

  $$\frac{\begin{array}{c}\emptyset;\emptyset \vdash v_0 : T_0 \qquad \mathsf{mtype}_{\emptyset}(m, T_0) = \text{<}\overline{X}\text{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}} \\ (\forall i \in [n])\ \emptyset;\emptyset \vdash v_i : [\overline{V/X}]U_i \qquad \emptyset \Vdash [\overline{V/X}]\overline{\mathcal{P}} \qquad \emptyset \vdash \overline{V}\ \mathsf{ok}\end{array}}{\emptyset;\emptyset \vdash \underbrace{v_0.m\text{<}\overline{V}\text{>}(\overline{v}^n)}_{=e} : \underbrace{[\overline{V/X}]U}_{=T}}\ \textsc{exp-invoke} \tag{B.2.40}$$

  Case distinction on the rule used to reduce e.

  - *Case* rule DYN-INVOKE-CLASS: Then

    $$\begin{aligned} v_0 &= \textbf{new}\ N(\overline{w}) \\ \mathsf{getmdef}^{\mathrm{c}}(m, N) &= \text{<}\overline{X'}\text{>}\,\overline{U'\,x'} \to U'\ \textbf{where}\ \overline{\mathcal{P}'}\,\{e''\} \\ e' &= [v_0/\mathit{this}, \overline{v/x}][\overline{V/X'}]e'' \\ m &= m^{\mathrm{c}} \end{aligned}$$

By definition of mtype, we know that $T_0 = N'$ for some N'. By Lemma B.2.16 we get

$$\begin{gathered} N \trianglelefteq_{\mathbf{c}} N' \\ \emptyset \vdash N \ \mathsf{ok} \end{gathered}$$

We now get with Lemma B.2.36 that

$$\begin{gathered} \overline{X} = \overline{X'} \\ \overline{x} = \overline{x'} \\ \emptyset; \mathit{this} : N, \overline{x : [\overline{V/X}]U} \vdash [\overline{V/X}]e : [\overline{V/X}]U \end{gathered}$$

Possibly repeated applications of Lemma B.2.38 yield

$$\emptyset; \emptyset \vdash e' : T$$

– *Case* rule DYN-INVOKE-IFACE: Then $m = m^{\mathrm{i}}$, $e' = [v_0/\mathit{this}, \overline{v/x}][\overline{V/X}]\varphi_1 e''$, and

$$\frac{\begin{gathered} \textbf{interface}\ I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}^{l}\ \textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{Q'}\,\{\ldots\ \overline{\mathit{rcsig}}\,\} \\ \mathit{rcsig}_j = \textbf{receiver}\,\{\overline{m : \mathit{msig}}\} \qquad m = m_k \qquad \mathit{msig}_k = \texttt{<}\overline{X''}\texttt{>}\,\overline{W\ x''} \to W\ \ \textbf{where}\ \overline{Q} \\ (\forall i \in [l], i \neq j)\ \mathsf{resolve}_{Z_i}(\overline{W}, \overline{N}) = M_i^{?} \qquad \mathsf{resolve}_{Z_j}(Z_j\overline{W}, N_0\overline{N}) = M_j^{?} \\ (\varphi_1, \textbf{implementation}\texttt{<}\overline{Z''}\texttt{>}\ I\texttt{<}\overline{W''}\texttt{>}\ [\,\overline{M'}\,]\ \textbf{where}\ \overline{Q''}\ \{\ldots\ \overline{\mathit{rcdef}}\ \}) \\ = \mathsf{least\text{-}impl}\,\mathscr{M} \\ \mathit{rcdef}_j = \textbf{receiver}\,\{\overline{\mathit{mdef}}\} \end{gathered}}{\mathsf{getmdef}^{\mathrm{i}}(m, N_0, \overline{N}) = \varphi_1\,\mathit{mdef}_k} \tag{B.2.41}$$

where

$$\begin{gathered} \mathit{mdef}_k = \texttt{<}\overline{X'}\texttt{>}\,\overline{U'\ x'} \to U'\ \ \textbf{where}\ \overline{P'}\,\{e''\} \\ (\forall i \in \{0, \ldots, n\})\ v_i = \textbf{new}\ N_i(\overline{w_i}) \\ \mathscr{M} = \{(\varphi, \textbf{implementation}\texttt{<}\overline{Z''}\texttt{>}\ I\texttt{<}\overline{W''}\texttt{>}\ [\,\overline{M'}\,]\ \ldots) \\ \mid \mathsf{dom}(\varphi) = \overline{Z''}, (\forall i \in [l])\ M_i^{?} = \mathsf{nil}\ \text{or}\ \emptyset \vdash M_i^{?} \leq \varphi M_i'\} \end{gathered} \tag{B.2.42}$$

By definition of mtype and Convention 3.5, we have from (B.2.40) that

$$\frac{\begin{gathered} \textbf{interface}\ I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}^{l}\ \textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{Q'}\,\{\ldots\ \overline{\mathit{rcsig}}\,\} \\ \mathit{rcsig}_j = \textbf{receiver}\,\{\overline{m : \mathit{msig}}\} \\ m = m_k \qquad \emptyset \Vdash \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{T''}\texttt{>} \qquad T_j = T_0 \end{gathered}}{\mathsf{mtype}_{\emptyset}(m, T_0) = \underbrace{[\overline{T''/Z'}, \overline{T/Z}]\mathit{msig}_k}_{=\texttt{<}\overline{X}\texttt{>}\,\overline{U\ x} \to U\ \ \textbf{where}\ \overline{\mathcal{P}}}}\ \text{MTYPE-IFACE} \tag{B.2.43}$$

With $\varphi_2 = [\overline{T''/Z'}, \overline{T/Z}]$ we then get

$$\overline{X} = \overline{X''} \tag{B.2.44}$$

$$\overline{x} = \overline{x''} \tag{B.2.45}$$

$$\varphi_2(\overline{W}, W, \overline{Q}) = \overline{U}, U, \overline{\mathcal{P}} \tag{B.2.46}$$

The underlying program is well-typed, so we have

$$\overline{Q''}, \overline{Z''}; \mathit{this} : M_j' \vdash \mathit{rcdef}_j\ \mathsf{implements}\ \underbrace{[\overline{W''/Z'}, \overline{M'/Z}]}_{=\varphi_3}\,\mathit{rcsig}_j$$

This especially implies

$$\overline{Q''}, \overline{Z''}; \mathit{this} : M'_j \vdash \mathit{mdef}_k \ \mathsf{implements}\ \varphi_3 \mathit{msig}_k$$

which in turn implies

$$\underbrace{\overline{Q''}, \overline{Z''}, \overline{P'}, \overline{X'}}_{=\Delta} \vdash \overline{U'}, U', \overline{P'}\ \mathsf{ok} \tag{B.2.47}$$

$$\Delta; \underbrace{\mathit{this} : M'_j, \overline{x' : U'}}_{=\Gamma} \vdash e'' : U' \tag{B.2.48}$$

$$\overline{X'} = \overline{X''} \tag{B.2.49}$$

$$\overline{x'} = \overline{x''} \tag{B.2.50}$$

$$\overline{U'} = \varphi_3 \overline{W} \tag{B.2.51}$$

$$\overline{P'} = \varphi_3 \overline{Q} \tag{B.2.52}$$

$$\Delta \vdash U' \leq \varphi_3 W \tag{B.2.53}$$

By (B.2.40) we get $\emptyset; \emptyset \vdash v_0 : T_0$, so with (B.2.42) and Lemma B.2.16

$$\emptyset \vdash N_0 \leq T_0 \tag{B.2.54}$$

Using (B.2.43) we get $\emptyset \Vdash \overline{T}\, \textbf{implements}\, I\texttt{<}\overline{T''}\texttt{>}$ with $T_j = T_0$. Lemma B.2.10 yields

$$\mathit{impl} = \textbf{implementation}\texttt{<}\overline{Z_3}\texttt{>}\ I\texttt{<}\overline{W_3}\texttt{>}\ [\,\overline{M_3}\,]\ \textbf{where}\ \overline{Q_3}\ \{ \ldots\ \overline{\mathit{rcdef}'}\,\}$$

$$\emptyset \Vdash \varphi_4 \overline{Q_3} \tag{B.2.55}$$

$$\mathsf{dom}(\varphi_4) = \overline{Z_3}$$

$$\overline{T''} = \varphi_4 \overline{W_3} \tag{B.2.56}$$

$$\emptyset \vdash N_0 \leq \varphi_4 M_{3j} \tag{B.2.57}$$

$$\begin{gathered}\text{if } j \notin \mathsf{pol}^+(I) \text{ then } \emptyset \vdash T_j \leq \varphi_4 M_{3j} \text{ with} \\ T_j \neq \varphi_4 M_{3j} \text{ implying } j \in \mathsf{pol}^-(I)\end{gathered} \tag{B.2.58}$$

$$(\forall i \neq j)\ \emptyset \vdash T_i \leq \varphi_4 M_{3i} \text{ with } T_i \neq \varphi_4 M_{3i} \text{ implying } i \in \mathsf{pol}^-(I) \tag{B.2.59}$$

$$\begin{gathered}\text{if } j \in \mathsf{pol}^+(I) \text{ and } j \notin \mathsf{pol}^-(I) \text{ and } T_j \neq \varphi_4 M_{3j} \text{ then} \\ \overline{T} = T_j = J\texttt{<}\overline{W_4}\texttt{>} \text{ and } J\texttt{<}\overline{W_4}\texttt{>} \trianglelefteq_{\mathbf{i}} I\texttt{<}\overline{W_3}\texttt{>} \text{ and } 1 \in \mathsf{pol}^+(J)\end{gathered} \tag{B.2.60}$$

We now show that $(\varphi_4, \mathit{impl}) \in \mathscr{M}$. To do so, we prove that $(\forall i \in [l]) M^?_i = \mathsf{nil}$ or $M^?_i \trianglelefteq_{\mathbf{c}} \varphi_4 M_{3i}$. Suppose $i \in [l]$ and assume $M^?_i \neq \mathsf{nil}$. By definition of $M^?_i$ in (B.2.41) and by Lemma B.2.18, it suffices to show that $N_p \trianglelefteq_{\mathbf{c}} \varphi_4 M_{3i}$ for all $p \in [n]$ with $W_p = Z_i$, and that $N_0 \trianglelefteq_{\mathbf{c}} \varphi_4 M_{3j}$. The latter follows directly from (B.2.57). Now assume $p \in [n]$ with $W_p = Z_i$. Then

$$\varphi_2 W_p = T_i$$

From (B.2.40) we have $\emptyset; \emptyset \vdash v_p : [\overline{V/X}] U_p$, so with (B.2.46)

$$\emptyset; \emptyset \vdash v_p : [\overline{V/X}] T_i$$

W.l.o.g., $\overline{X} \cap \mathsf{ftv}(T_i) = \emptyset$, so $[\overline{V/X}] T_i = T_i$. Thus, with (B.2.42) and Lemma B.2.16

$$\emptyset \vdash N_p \leq T_i$$

Because $W_p = Z_i$, we have $i \notin \mathsf{pol}^+(I)$. Hence, we get from (B.2.58) and (B.2.59) that $\emptyset \vdash T_i \leq \varphi_4 M_{3i}$. By transitivity of subtyping we then get $N_p \trianglelefteq_{\mathsf{c}} \varphi_4 M_{3i}$ as required. We now have established the fact that

$$(\varphi_4, \mathit{impl}) \in \mathscr{M}$$

From (B.2.41) and the definition of $\mathsf{least\text{-}impl}$, we get that

$$(\forall i \in [l])\ \varphi_1 M'_i \trianglelefteq_{\mathsf{c}} \varphi_4 M_{3i} \tag{B.2.61}$$

We then get from (B.2.55) and criterion WF-PROG-4 that

$$\emptyset \Vdash \varphi_1 \overline{Q''} \tag{B.2.62}$$

By criterion WF-PROG-2 we get $\varphi_1 \overline{W''} = \varphi_4 \overline{W_3}$, so with (B.2.56)

$$\varphi_1 \overline{W''} = \overline{T''} \tag{B.2.63}$$

By criterion WF-IFACE-3 we have $\overline{Z} \cap \mathsf{ftv}(\overline{Q}) = \emptyset$. Then $\mathsf{ftv}(\overline{Q}) \subseteq \overline{Z'}$ because the underlying program is well-typed. W.l.o.g., $\overline{Z''} \cap \mathsf{ftv}(\overline{Q}) = \emptyset$, so

$$\varphi_1\varphi_3\overline{Q} = \varphi_1[\overline{W''/Z'}]\overline{Q} = [\overline{\varphi_1 W''/Z'}]\overline{Q} = [\overline{T''/Z'}]\overline{Q} = \varphi_2\overline{Q}$$

From (B.2.40) and (B.2.46) we get $\emptyset \Vdash [\overline{V/X}]\varphi_2\overline{Q}$. Thus,

$$\emptyset \Vdash [\overline{V/X}]\varphi_1\varphi_3\overline{Q} \tag{B.2.64}$$

We have $v_0 = \mathbf{new}\ N_0(\overline{w_0})$ by (B.2.42). By (B.2.41), the definition of $\mathsf{resolve}$, and Lemma B.2.20 $N_0 \trianglelefteq_{\mathsf{c}} M^?_j$. Moreover, $M^?_j \trianglelefteq_{\mathsf{c}} \varphi_1 M'_j$ by definition of $\mathscr{M}$ and $\mathsf{least\text{-}impl}$. Then with EXP-SUBSUME $\emptyset; \emptyset \vdash v_0 : \varphi_1 M'_j$. We have $\emptyset \vdash \overline{V}\ \mathsf{ok}$ by (B.2.40) so $\emptyset; \emptyset \vdash [\overline{V/X}]v_0 : [\overline{V/X}]\varphi_1 M'_j$ by Lemma B.2.32. W.l.o.g., $\overline{X} \cap \mathsf{ftv}(v_0) = \emptyset$, so

$$\emptyset; \emptyset \vdash v_0 : [\overline{V/X}]\varphi_1 M'_j \tag{B.2.65}$$

Next, we prove that $\emptyset; \emptyset \vdash v_i : [\overline{V/X}]\varphi_1 U'_i$ for all $i \in [n]$. Assume $i \in [n]$. By criterion WF-IFACE-3 we have either $\overline{Z} \cap \mathsf{ftv}(W_i) = \emptyset$ or $W_i \in \overline{Z}$. Because the underlying program is well-typed, we have $\mathsf{ftv}(W_i) \subseteq \{\overline{Z}, \overline{Z'}\}$. W.l.o.g., $\overline{Z''} \cap \mathsf{ftv}(W_i) = \emptyset$.

- ∗ Assume $\overline{Z} \cap \mathsf{ftv}(W_i) = \emptyset$. Then

  $$\varphi_1\varphi_3 W_i = \varphi_1[\overline{W''/Z'}]W_i = [\overline{\varphi_1 W''/Z'}]W_i \overset{(B.2.63)}{=} [\overline{T''/Z'}]W_i = \varphi_2 W_i$$

  Hence,

  $$U_i \overset{(B.2.46)}{=} \varphi_2 W_i = \varphi_1\varphi_3 W_i \overset{(B.2.51)}{=} \varphi_1 U'_i$$

  From (B.2.40) we have $\emptyset; \emptyset \vdash v_i : [\overline{V/X}]U_i$. Thus, $\emptyset; \emptyset \vdash v_i : [\overline{V/X}]\varphi_1 U'_i$.

- ∗ Assume $W_i = Z_k$ for some $k \in [l]$. We have $v_i = \mathbf{new}\ N_i(\overline{w_i})$ by (B.2.42). By (B.2.41), the definition of $\mathsf{resolve}$, and Lemma B.2.20 $M^?_k \neq \mathsf{nil}$ and $N_i \trianglelefteq_{\mathsf{c}} M^?_k$. Moreover, $M^?_k \trianglelefteq_{\mathsf{c}} \varphi_1 M'_k$ by definition of $\mathscr{M}$. By rule EXP-NEW, Lemma B.1.4, and rule EXP-SUBSUME we then have $\emptyset; \emptyset \vdash v_i : \varphi_1 M'_k$. We also have

  $$\varphi_1 M'_k = \varphi_1\varphi_3 Z_k = \varphi_1\varphi_3 W_i \overset{(B.2.51)}{=} \varphi_1 U'_i$$

  We have $\emptyset \vdash \overline{V}\ \mathsf{ok}$ by (B.2.40) so $\emptyset; \emptyset \vdash [\overline{V/X}]v_i : [\overline{V/X}]\varphi_1 U'_i$ by Lemma B.2.32. W.l.o.g., $\overline{X} \cap \mathsf{ftv}(v_0) = \emptyset$, so $\emptyset; \emptyset \vdash v_i : [\overline{V/X}]\varphi_1 U'_i$.

This finishes the proof of

$$(\forall i \in [n])\ \emptyset; \emptyset \vdash v_i : [\overline{V/X}]\varphi_1 U'_i \tag{B.2.66}$$

Next, we prove $\emptyset \vdash \varphi_1\varphi_3 W \leq \varphi_2 W$. Note that $\mathsf{ftv}(W) \subseteq \{\overline{Z}, \overline{Z'}\}$ because the underlying program is well-typed. W.l.o.g., $\overline{Z''} \cap \mathsf{ftv}(W) = \emptyset$.

Case distinction on whether or not $\overline{Z} \cap \mathsf{ftv}(W) = \emptyset$.

- *Case* $\overline{Z} \cap \mathsf{ftv}(W) = \emptyset$: Then

  $$\varphi_1\varphi_3 W = \varphi_1[\overline{W''/Z'}]W = [\overline{\varphi_1 W''/Z'}]W \overset{(\text{B.2.63})}{=} [\overline{T''/Z'}]W = \varphi_2 W$$

  By reflexivity of subtyping then

  $$\emptyset \vdash \varphi_1\varphi_3 W \leq \varphi_2 W$$

- *Case* $\overline{Z} \cap \mathsf{ftv}(W) \neq \emptyset$: By criterion WF-IFACE-3 then $W = Z_k$ for some $k \in [l]$. Then

  $$k \notin \mathsf{pol}^-(I) \tag{B.2.67}$$

  We first concentrate on the case where $k \neq j$ or $j \notin \mathsf{pol}^+(I)$ or $\varphi_4 M_{3k} = T_k$. Then we have

  $$\begin{aligned}\varphi_1\varphi_3 W = \varphi_1\varphi_3 Z_k = \varphi_1 M'_k &\overset{(\text{B.2.61})}{\trianglelefteq_{\mathbf{c}}} \varphi_4 M_{3k} \\ &\overset{(\text{B.2.58}) \text{ or } (\text{B.2.59}) \text{ or assumption}}{=} T_k \overset{\text{definition of } \varphi_2}{=} \varphi_2 Z_k = \varphi_2 W\end{aligned}$$

  Thus, we get

  $$\emptyset \vdash \varphi_1\varphi_3 W \leq \varphi_2 W$$

  Now we consider the case $k = j$ and $j \in \mathsf{pol}^+(I)$ and $\varphi_4 M_{3k} \neq T_k$. From (B.2.60) we get

  $$j = k = l = 1 \tag{B.2.68}$$

  $$\overline{T} = T_j = J\texttt{<}\overline{W_4}\texttt{>} \tag{B.2.69}$$

  $$J\texttt{<}\overline{W_4}\texttt{>} \trianglelefteq_{\mathbf{i}} I\texttt{<}\overline{W_3}\texttt{>} \tag{B.2.70}$$

  $$1 \in \mathsf{pol}^+(J) \tag{B.2.71}$$

  With (B.2.54) and (B.2.43) we then get $\emptyset \vdash N_0 \leq J\texttt{<}\overline{W_4}\texttt{>}$. Lemma B.2.2 yields

  $$\mathbf{implementation}\texttt{<}\overline{Z_4}\texttt{>}\ J\texttt{<}\overline{W'_4}\texttt{>}\ [\,N'_0\,]\ \mathbf{where}\ \overline{Q_4}\ \ldots \tag{B.2.72}$$

  $$\mathsf{dom}(\psi) = \overline{Z_4}$$

  $$\emptyset \Vdash \psi\overline{Q_4} \tag{B.2.73}$$

  $$\psi\overline{W'_4} = \overline{W_4} \tag{B.2.74}$$

  $$N_0 \trianglelefteq_{\mathbf{c}} \psi N'_0 \tag{B.2.75}$$

  With (B.2.70) and Lemma B.1.2 we get

  $$\psi N'_0\ \mathbf{implements}\ I\texttt{<}\overline{W_3}\texttt{>} \in \mathsf{sup}(\psi N'_0\ \mathbf{implements}\ J\texttt{<}\overline{W_4}\texttt{>})$$

With Lemma B.1.24 and (B.2.74) we get the existence of N_0'' and $I\texttt{<}\overline{W_3'}\texttt{>}$ such that

$$\begin{gathered} N_0'' \,\textbf{implements}\, I\texttt{<}\overline{W_3'}\texttt{>} \in \mathsf{sup}(N_0' \,\textbf{implements}\, J\texttt{<}\overline{W_4'}\texttt{>}) \\ \psi N_0'' = \psi N_0' \\ \psi I\texttt{<}\overline{W_3'}\texttt{>} = I\texttt{<}\overline{W_3}\texttt{>} \end{gathered} \tag{B.2.76}$$

Now by criterion WF-IMPL-1, (B.2.67), and (B.2.68)

$$\begin{gathered} \mathit{impl}' = \textbf{implementation}\texttt{<}\overline{Z_5}\texttt{>}\ I\texttt{<}\overline{W_3''}\texttt{>}\ [\,N_0'''\,]\ \ldots \\ \mathsf{dom}(\psi') = \overline{Z_5} \\ N_0'' = \psi' N_0''' \\ I\texttt{<}\overline{W_3'}\texttt{>} = \psi' I\texttt{<}\overline{W_3''}\texttt{>} \end{gathered} \tag{B.2.77}$$

With (B.2.76) we then get $\psi N_0' = \psi\psi' N_0'''$. Hence, with (B.2.75)

$$N_0 \trianglelefteq_{\mathbf{c}} \psi\psi' N_0'''$$

From (B.2.71) it is easy to see that $Z_j \notin \mathsf{ftv}(\overline{W})$. Thus, from (B.2.41) and the definition of resolve, we have $M_j^? = N_0$. With (B.2.68) and the definition of $\mathscr{M}$ we then have

$$(\psi\psi', \mathit{impl}') \in \mathscr{M}$$

From (B.2.41) and the definition of least-impl we then have

$$\varphi_1 M_j' \trianglelefteq_{\mathbf{c}} \psi\psi' N_0'''$$

From (B.2.72), (B.2.73), (B.2.74), and rule ENT-IMPL, we have

$$\emptyset \Vdash \psi N_0' \,\textbf{implements}\, J\texttt{<}\overline{W_4}\texttt{>}$$

Hence, with rule SUB-IMPL then $\emptyset \vdash \psi N_0' \leq J\texttt{<}\overline{W_4}\texttt{>}$. With (B.2.76) and (B.2.77) $\psi N_0' = \psi\psi' N_0'''$, and with (B.2.69) $J\texttt{<}\overline{W_4}\texttt{>} = T_j$. With transitivity of subtyping, and (B.2.68) we then have

$$\emptyset \vdash \varphi_1 M_k' \leq T_k$$

Moreover, we have

$$\begin{gathered} \varphi_1\varphi_3 W = \varphi_1\varphi_3 Z_k = \varphi_1 M_k' \\ T_k \overset{\text{definition of } \varphi_2}{=} \varphi_2 Z_k = \varphi_2 W \end{gathered}$$

Thus, we get

$$\emptyset \vdash \varphi_1\varphi_3 W \leq \varphi_2 W$$

End case distinction on whether or not $\overline{Z} \cap \mathsf{ftv}(W) = \emptyset$.

We now have proved $\emptyset \vdash \varphi_1\varphi_3 W \leq \varphi_2 W$. Using Lemma B.2.22 we conclude

$$\emptyset \vdash [\overline{V/X}]\varphi_1\varphi_3 W \leq [\overline{V/X}]\varphi_2 W \tag{B.2.78}$$

W.l.o.g., $\mathsf{ftv}(\varphi_1\overline{Q''}) \cap \overline{X} = \emptyset$, so with (B.2.62) $\emptyset \Vdash [\overline{V/X}]\varphi_1\overline{Q''}$. From (B.2.64) and (B.2.52) we get $\emptyset \Vdash [\overline{V/X}]\varphi_1\overline{P'}$. Hence, with (B.2.47)

$$\emptyset \Vdash [\overline{V/X}]\varphi_1\Delta \tag{B.2.79}$$

Assume $\varphi_1 = [\overline{V'/Z''}]$. W.l.o.g., $\mathsf{ftv}(\overline{V'}) \cap \overline{X} = \emptyset$. With (B.2.44) and (B.2.49) then $[\overline{V/X}]\varphi_1 = [\overline{V/X'}, \overline{V'/Z''}]$. Hence, with (B.2.47)

$$\mathsf{dom}(\Delta) \setminus \mathsf{dom}([\overline{V/X}]\varphi_1) = \emptyset$$

From (B.2.40) we have $\emptyset \vdash \overline{V}$ ok. From Lemma B.2.16, (B.2.40), and (B.2.42) we get $\emptyset \vdash N_i$ ok for all $i = 0, \ldots, n$. By definition of $\mathsf{resolve}$ and Lemma B.2.28 we then get $\emptyset \vdash M_i^?$ ok unless $M_i^? = \mathsf{nil}$. Moreover, by definition of $\mathsf{resolve}$ and disp, we get $M_i^? \neq \mathsf{nil}$ for all $i \in \mathsf{disp}(I)$. Hence, with (B.2.41), the definition of $\mathscr{M}$, and Lemma B.2.26 we get $\emptyset \vdash \varphi_1 X$ ok for all $X \in \mathsf{dom}(\varphi_1)$. Thus,

$$\emptyset \vdash [\overline{V/X}]\varphi_1 Z \text{ for all } Z \in \mathsf{dom}([\overline{V/X}]\varphi_1)$$

We now get with (B.2.48) and Lemma B.2.32 that

$$\emptyset; [\overline{V/X}]\varphi_1\Gamma \vdash [\overline{V/X}]\varphi_1 e'' : [\overline{V/X}]\varphi_1 U'$$

We have with (B.2.45), (B.2.50), and (B.2.48) that $\Gamma = \mathit{this} : M_j', \overline{x : U'}$. Thus, with (B.2.65), (B.2.66), and repeated applications of Lemma B.2.38, we get

$$\emptyset; \emptyset \vdash \underbrace{[v_0/\mathit{this}, \overline{v/x}][\overline{V/X}]\varphi_1 e''}_{=e'} : [\overline{V/X}]\varphi_1 U'$$

To finish the case where e is reduced using rule <span style="font-variant:small-caps">dyn-invoke-iface</span>, we still need to show that $\emptyset \vdash [\overline{V/X}]\varphi_1 U' \leq T$. (The claim then follows with rule EXP-SUBSUME.) From (B.2.53) we get with (B.2.79) and Lemma B.2.22 that

$$\emptyset \vdash [\overline{V/X}]\varphi_1 U' \leq [\overline{V/X}]\varphi_1\varphi_3 W$$

Moreover, with (B.2.78) and transitivity of subtyping we then get

$$\emptyset \vdash [\overline{V/X}]\varphi_1 U' \leq [\overline{V/X}]\varphi_2 W$$

Ultimately, we have

$$[\overline{V/X}]\varphi_2 W \overset{(\text{B.2.46})}{=} [\overline{V/X}]U \overset{(\text{B.2.40})}{=} T$$

- *Case* other rules: Impossible.

End case distinction on the rule used to reduce e.

- *Case* rule EXP-INVOKE-STATIC: Then

$$\dfrac{\begin{array}{c}\mathsf{smtype}_{\emptyset}(m, I\texttt{<}\overline{W}\texttt{>}[\overline{T}]) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\;x} \to U \textbf{ where } \overline{\mathcal{P}} \\ (\forall i)\ \emptyset; \emptyset \vdash e_i : [\overline{V/X}]U_i \qquad \emptyset \Vdash [\overline{V/X}]\overline{\mathcal{P}} \qquad \emptyset \vdash \overline{T}, \overline{V}\ \mathsf{ok}\end{array}}{\emptyset; \emptyset \vdash \underbrace{I\texttt{<}\overline{W}\texttt{>}[\overline{T}].m\texttt{<}\overline{V}\texttt{>}(\overline{e})}_{=e} : \underbrace{[\overline{V/X}]U}_{=T}}\ \textsc{exp-invoke-static} \tag{B.2.80}$$

Expanding the definition of smtype yields:

$$\frac{\begin{array}{c}\textbf{interface}\ I<\overline{Y'}>[\,\overline{Y}\,\textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{Q'}\,\{\,\overline{m : \textbf{static}\,\mathit{msig}}\ \ldots\}\\ \emptyset \Vdash \overline{T}\,\textbf{implements}\,I<\overline{W}> \qquad m = m_k\end{array}}{\mathsf{smtype}_{\emptyset}(m, I<\overline{W}>[\overline{T}]) = \underbrace{[\overline{W/Y'}, \overline{T/Y}]\mathit{msig}_k}_{=<\overline{X}>\,\overline{U\,x}\rightarrow U\ \ \textbf{where}\ \overline{\mathcal{P}}}}\ \textsc{mtype-static} \tag{B.2.81}$$

Define $\varphi_2 = [\overline{W/Y'}, \overline{T/Y}]$ and assume

$$\mathit{msig}_k = <\overline{X''}>\,\overline{U''\,x''} \rightarrow U''\ \textbf{where}\ \overline{P}$$

Then

$$\overline{X''} = \overline{X} \tag{B.2.82}$$

$$\overline{x''} = \overline{x} \tag{B.2.83}$$

$$\varphi_2(\overline{U''}, U'', \overline{P}) = (\overline{U}, U, \overline{\mathcal{P}}) \tag{B.2.84}$$

By looking at the form of e, we see that $e \longmapsto e'$ must have been performed by rule DYN-INVOKE-STATIC. Thus,

$$\frac{\mathsf{getsmdef}(m, I<\overline{W}>, \overline{T}) = <\overline{X'}>\,\overline{U'\,x'} \rightarrow U'\ \textbf{where}\ \overline{\mathcal{P}'}\,\{e''\}}{I<\overline{W}>[\overline{T}].m<\overline{V}>(\overline{v}) \longmapsto \underbrace{[\overline{v/x}][\overline{V/X}]e''}_{=e'}}\ \textsc{dyn-invoke-static} \tag{B.2.85}$$

$$\overline{v} = \overline{e} \tag{B.2.86}$$

Expanding the definition of getsmdef (i.e. inverting rule DYN-MDEF-STATIC) yields together with criterion WF-IFACE-1 that

$$\frac{\begin{array}{c}\textbf{interface}\ I<\overline{Y'}>[\,\overline{Y}\,\textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{Q'}\,\{\,\overline{m : \textbf{static}\,\mathit{msig}}\ \ldots\}\\ m = m_k \qquad (\varphi_1, \textbf{implementation}<\overline{Z}>\ I<\overline{W'}>\ [\,\overline{N'}^{l}\,]\ \textbf{where}\ \overline{Q}\ \{\,\overline{\textbf{static}\,\mathit{mdef}}\ldots\})\\ = \mathsf{least\text{-}impl}\mathscr{M}\end{array}}{\mathsf{getsmdef}(m, I<\overline{W}>, \overline{T}^{l}) = \underbrace{\varphi_1\,\mathit{mdef}_k}_{=<\overline{X'}>\,\overline{U'\,x'}\rightarrow U'\ \ \textbf{where}\ \overline{\mathcal{P}'}\,\{e''\}}} \tag{B.2.87}$$

where

$$\begin{array}{l}\mathscr{M} = \{(\varphi, \textbf{implementation}<\overline{X}>\ I<\overline{U}>\ [\,\overline{N}^{l}\,]\ \ldots)\\ \qquad \mid \mathsf{dom}(\varphi) = \overline{X}, (\forall i \in [l])\ N_i = \mathit{Object}\ \text{or}\ T_i \trianglelefteq_{\mathbf{c}} \varphi N_i\}\end{array}$$

Assume

$$\mathit{mdef}_k = <\overline{X'}>\,\overline{U'''\,x'} \rightarrow U'''\ \textbf{where}\ \overline{P'}\,\{e'''\}$$

Then

$$\varphi_1(\overline{U'''}, U''', \overline{P'}, e''') = \overline{U'}, U', \overline{\mathcal{P}'}, e'' \tag{B.2.88}$$

Because the underlying program is well-typed, we have by inverting rule OK-IMPL and criterion WF-IFACE-1

$$\overline{Q}, \overline{Z}; \emptyset \vdash \mathit{mdef}_k\ \mathsf{implements}\ \underbrace{[\overline{W'/Y'}, \overline{Y/N'}]}_{=\varphi_3}\,\mathit{msig}_k$$

We then have

$$\underbrace{\overline{Q}, \overline{Z}, \overline{P'}, \overline{X'}}_{=\Delta} \vdash \overline{U'''}, U''', \overline{P'}\ \mathsf{ok} \tag{B.2.89}$$

$$\Delta; \underbrace{\overline{x' : U'''}}_{=\Gamma} \vdash e''' : U''' \tag{B.2.90}$$

$$\overline{X'} = \overline{X''} \tag{B.2.91}$$

$$\overline{U'''} = \varphi_3 \overline{U''} \tag{B.2.92}$$

$$\overline{x'} = \overline{x''} \tag{B.2.93}$$

$$\overline{P'} = \varphi_3 \overline{P} \tag{B.2.94}$$

$$\Delta \vdash U''' \leq \varphi_3 U'' \tag{B.2.95}$$

From (B.2.80) and (B.2.81) we have $\emptyset \Vdash \overline{T}\ \textbf{implements}\ I\texttt{<}\overline{W}\texttt{>}$. With Lemma B.2.11 we get

$$\mathit{impl} = \textbf{implementation}\texttt{<}\overline{Z'}\texttt{>}\ I\texttt{<}\overline{W''}\texttt{>}\ [\overline{N''}]\ \textbf{where}\ \overline{Q''}\ \ldots$$

$$\mathsf{dom}(\varphi_4) = \overline{Z'}$$

$$\emptyset \Vdash \varphi_4 \overline{Q''} \tag{B.2.96}$$

$$\overline{W} = \varphi_4 \overline{W''} \tag{B.2.97}$$

$$(\forall i)\ \emptyset \vdash T_i \leq \varphi_4 N''_i \text{ with } T_i \neq \varphi_4 N''_i \text{ implying } i \in \mathsf{pol}^-(I) \tag{B.2.98}$$

With Lemma B.2.2 and by looking at the definition of $\mathscr{M}$, we see that

$$(\varphi_4, \mathit{impl}) \in \mathscr{M} \tag{B.2.99}$$

Thus, with (B.2.87) and the definition of $\mathsf{least\text{-}impl}$

$$(\forall i)\ \varphi_1 N'_i \trianglelefteq_{\mathsf{c}} \varphi_4 N''_i \tag{B.2.100}$$

With (B.2.96) and criterion WF-PROG-4 we get $\emptyset \Vdash \varphi_1 \overline{Q}$. With Lemma B.2.22 then

$$\emptyset \Vdash [\overline{V/X}] \varphi_1 \overline{Q} \tag{B.2.101}$$

From (B.2.99), (B.2.87), (B.2.100), and criterion WF-PROG-2 we get $\varphi_4 \overline{W''} = \varphi_1 \overline{W'}$, so with (B.2.97)

$$\overline{W} = \varphi_1 \overline{W'} \tag{B.2.102}$$

We get from criterion WF-IFACE-3 that $\overline{Y} \cap \mathsf{ftv}(\overline{P}) = \emptyset$. W.l.o.g., $\mathsf{dom}(\varphi_1) = \overline{Z} \cap \mathsf{ftv}(\overline{P}) = \emptyset$. Hence,

$$\varphi_2 \overline{P} = [\overline{W/Y'}]\overline{P} \overset{\text{(B.2.102)}}{=} [\overline{\varphi_1 W'/Y'}]\overline{P} = \varphi_1 [\overline{W'/Y'}]\overline{P} = \varphi_1 \varphi_3 \overline{P}$$

From (B.2.80) we have $\emptyset \Vdash [\overline{V/X}]\overline{\mathcal{P}}$ and from (B.2.81) we have $[\overline{V/X}]\overline{\mathcal{P}} = [\overline{V/X}]\varphi_2 \overline{P}$. Thus, $\emptyset \Vdash [\overline{V/X}]\varphi_1 \varphi_3 \overline{P}$, so with (B.2.94) $\emptyset \Vdash [\overline{V/X}]\varphi_1 \overline{P'}$. With (B.2.101) and (B.2.89) then

$$\emptyset \Vdash [\overline{V/X}] \varphi_1 \Delta \tag{B.2.103}$$

Next, we show that $(\forall i)\ \emptyset; \emptyset \vdash v_i : [\overline{V/X}]\varphi_1 U'''_i$. Fix some i. W.l.o.g., $\mathsf{dom}(\varphi_1) = \overline{Z} \cap \mathsf{ftv}(U''_i) = \emptyset$.

Case distinction on whether or not $\overline{Y} \cap \mathsf{ftv}(U''_i) = \emptyset$.

- *Case* $\overline{Y} \cap \mathsf{ftv}(U''_i) = \emptyset$: Then

  $$U_i \overset{(\text{B.2.84})}{=} \varphi_2 U''_i = [\overline{W/Y'}]U''_i \overset{(\text{B.2.102})}{=} [\overline{\varphi_1 W'/Y'}]U''_i = \varphi_1[\overline{W'/Y'}]U''_i = \varphi_1\varphi_3 U''_i \overset{(\text{B.2.92})}{=} \varphi_1 U'''_i$$

  Using reflexivity of subtyping, we get

  $$\emptyset \vdash U_i \leq \varphi_1 U'''_i$$

- *Case* $\overline{Y} \cap \mathsf{ftv}(U''_i) \neq \emptyset$: By criterion WF-IFACE-3 we than have $U''_i = Y_j$ for some $j \in [l]$. Then

  $$U_i \overset{(\text{B.2.84})}{=} \varphi_2 U''_i = \varphi_2 Y_j = T_j$$

  We also have

  $$\varphi_1 N'_j \overset{\text{definition of } \varphi_3}{=} \varphi_1\varphi_3 Y_j = \varphi_1\varphi_3 U''_i \overset{(\text{B.2.92})}{=} \varphi_1 U'''_i$$

  By definition of $\mathscr{M}$ we have that either $\varphi_1 N'_j = \mathit{Object}$ or $T_j \trianglelefteq_{\mathsf{c}} \varphi_1 N'_j$. In both cases we get

  $$\emptyset \vdash U_i \leq \varphi_1 U'''_i$$

End case distinction on whether or not $\overline{Y} \cap \mathsf{ftv}(U''_i) = \emptyset$.

We now have established that $\emptyset \vdash U_i \leq \varphi_1 U'''_i$. With Lemma B.2.22 we get $\emptyset \vdash [\overline{V/X}]U_i \leq [\overline{V/X}]\varphi_1 U'''_i$. From (B.2.80) and (B.2.86) we have $(\forall i)\ \emptyset; \emptyset \vdash v_i : [\overline{V/X}]U_i$, so we get with rule EXP-SUBSUME that

$$(\forall i)\ \emptyset; \emptyset \vdash v_i : [\overline{V/X}]\varphi_1 U'''_i \tag{B.2.104}$$

Our next goal is to show that $\emptyset \vdash [\overline{V/X}]\varphi_1 U''' \leq [\overline{V/X}]U$. W.l.o.g., $\mathsf{dom}(\varphi_1) = \overline{Z} \cap \mathsf{ftv}(U'') = \emptyset$.

Case distinction on whether or not $\overline{Y} \cap \mathsf{ftv}(U'') = \emptyset$.

- *Case* $\overline{Y} \cap \mathsf{ftv}(U'') = \emptyset$: Then

  $$U \overset{(\text{B.2.84})}{=} \varphi_2 U'' = [\overline{W/Y'}]U'' \overset{(\text{B.2.102})}{=} [\overline{\varphi_1 W'/Y'}]U'' = \varphi_1[\overline{W'/Y'}]U'' = \varphi_1\varphi_3 U''$$

  Hence,

  $$\emptyset \vdash \varphi_1\varphi_3 U'' \leq U$$

- *Case* $\overline{Y} \cap \mathsf{ftv}(U'') \neq \emptyset$: By criterion WF-IFACE-3 we than have $U'' = Y_j$ for some $j \in [l]$. Moreover, $j \notin \mathsf{pol}^-(I)$. Then

  $$\varphi_1\varphi_3 U'' = \varphi_1\varphi_3 Y_j \overset{\text{definition of } \varphi_3}{=} \varphi_1 N'_j \overset{(\text{B.2.99}),\text{definition of } \mathsf{least\text{-}impl}}{\trianglelefteq_{\mathsf{c}}} \varphi_4 N''_j \overset{(\text{B.2.98})}{=} T_i = \varphi_2 Y_j = \varphi_2 U'' \overset{(\text{B.2.84})}{=} U$$

  We then get

  $$\emptyset \vdash \varphi_1\varphi_3 U'' \leq U$$

End case distinction on whether or not $\overline{Y} \cap \mathsf{ftv}(U'') = \emptyset$.

In both cases, we have shown $\emptyset \vdash \varphi_1\varphi_3 U'' \leq U$ so with Lemma B.2.22

$$\emptyset \vdash [\overline{V/X}]\varphi_1\varphi_3 U'' \leq [\overline{V/X}]U$$

From (B.2.95), (B.2.103), and Lemma B.2.22 we have

$$\emptyset \vdash [\overline{V/X}]\varphi_1 U''' \leq [\overline{V/X}]\varphi_1\varphi_3 U''$$

With transitivity of subtyping, we then get

$$\emptyset \vdash [\overline{V/X}]\varphi_1 U''' \leq [\overline{V/X}]U \tag{B.2.105}$$

Now we combine the various results. Assume $\varphi_1 = [\overline{V'/Z}]$. W.l.o.g., $\mathsf{ftv}(\overline{V'}) \cap \mathsf{ftv}(\overline{X}) = \emptyset$. Thus, with (B.2.82) and (B.2.91) we have $[\overline{V/X}]\varphi_1 = [\overline{V/X}, \overline{V'/Z}]$. With (B.2.89) then

$$\mathsf{dom}(\Delta) \setminus \mathsf{dom}([\overline{V/X}]\varphi_1) = \emptyset$$

From (B.2.80) we get $\emptyset \vdash \overline{T}, \overline{V}\ \mathsf{ok}$. With Lemma B.2.27 and the definition of $\mathscr{M}$ we then get $\emptyset \vdash \varphi_1 X\ \mathsf{ok}$ for all $X \in \mathsf{dom}(\varphi_1)$. Thus,

$$\emptyset \vdash [\overline{V/X}]\varphi_1 Z \text{ for all } Z \in \mathsf{dom}([\overline{V/X}]\varphi_1)$$

With (B.2.103), (B.2.90), and Lemma B.2.32 we now get

$$\emptyset; [\overline{V/X}]\varphi_1\Gamma \vdash [\overline{V/X}]\varphi_1 e''' : [\overline{V/X}]\varphi_1 U'''$$

With (B.2.104), the definition of Γ, and possibly repeated applications of Lemma B.2.38 we then get

$$\emptyset; \emptyset \vdash [\overline{v/x}][\overline{V/X}]\varphi_1 e''' : [\overline{V/X}]\varphi_1 U'''$$

With (B.2.85) and (B.2.88) we get $[\overline{v/x}][\overline{V/X}]\varphi_1 e''' = e'$. Thus, with (B.2.80), (B.2.105), and rule EXP-SUBSUME we get

$$\emptyset; \emptyset \vdash e' : T$$

as required.

- *Case* rule EXP-NEW: Then $e = \mathbf{new}\, N(\overline{e})$. But this is a contradiction to $e \longmapsto e'$.
- *Case* rule EXP-CAST: Then

  $$\frac{\emptyset \vdash T\ \mathsf{ok} \qquad \emptyset; \emptyset \vdash e_0 : T'}{\emptyset; \emptyset \vdash (T)\, e_0 : T}\ \textsc{exp-cast}$$

  with $e = (T)\, e_0$. The reduction step $e \longmapsto e'$ must have been performed through rule DYN-CAST. Thus,

  $$e' = e_0$$
  $$e_0 = \mathbf{new}\, M(\overline{w})$$
  $$\emptyset \vdash M \leq T$$

  By Lemma B.2.15 and a case analysis on the form of e_0, we know that

  $$\emptyset; \emptyset \vdash e_0 : M$$

  Hence, the claim $\emptyset; \emptyset \vdash e' : T$ follows with rule EXP-SUBSUME.
- *Case* rule EXP-SUBSUME: In this case, the claim follows directly from the I.H. and rule EXP-SUBSUME.

End case distinction on the last rule of the derivation of $\emptyset; \emptyset \vdash e : T$. □

B.2.3 Proof of Theorem 3.16

Theorem 3.16 states that CoreGI's proper evaluation relation preserves the types of expressions.

Proof of Theorem 3.16. By inverting rule DYN-CONTEXT we know that there exists an evaluation context $\mathcal{E}$ and expressions e_0, e_0' such that $e = \mathcal{E}[e_0]$ and $e_0 \longmapsto e_0'$ and $\mathcal{E}[e_0'] = e'$. Hence, it suffices to show the following claim:

$$\textit{If } \emptyset; \emptyset \vdash \mathcal{E}[e] : T \textit{ and } e \longmapsto e' \textit{ then } \emptyset; \emptyset \vdash \mathcal{E}[e'] : T.$$

The proof of this claim is by induction on $\mathcal{E}$. If $\mathcal{E} = \square$, then the claim holds by Theorem 3.15. In all other cases, we first use Lemma B.2.15 to obtain a derivation $\mathcal{D}$ for $\emptyset; \emptyset \vdash \mathcal{E}[e] : T'$ such that $\emptyset \vdash T' \leq T$ and $\mathcal{D}$ does not end with rule EXP-SUBSUME. Then the form of $\mathcal{E}$ uniquely determines the last rule $\mathfrak{r}$ used in $\mathcal{D}$. In each case, the claim then follows by the I.H. and applications of rules $\mathfrak{r}$ and EXP-SUBSUME. □

B.3 Determinacy of Evaluation for CoreGI

This section shows that CoreGI's evaluation relation is deterministic.

Lemma B.3.1. *If* $\mathsf{least\text{-}impl}\mathscr{M} = (\varphi_1, \mathit{impl}_1)$ *and* $\mathsf{least\text{-}impl}\mathscr{M} = (\varphi_2, \mathit{impl}_2)$ *then* $\varphi_1 = \varphi_2$ *and* $\mathit{impl}_1 = \mathit{impl}_2$*.*

Proof. Assume

$$\begin{aligned} \mathit{impl}_1 &= \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ [\,\overline{M}\,]\ \ldots \\ \mathit{impl}_2 &= \textbf{implementation}\texttt{<}\overline{Y}\texttt{>}\ I\texttt{<}\overline{U}\texttt{>}\ [\,\overline{N}\,]\ \ldots \end{aligned}$$

Then $\mathsf{dom}(\varphi_1) = \overline{X}$, $\mathsf{dom}(\varphi_2) = \overline{Y}$, and, by definition of $\mathsf{least\text{-}impl}$, $\varphi_1\overline{M} \trianglelefteq_{\mathsf{c}} \varphi_2\overline{N}$ and $\varphi_2\overline{N} \trianglelefteq_{\mathsf{c}} \varphi_1\overline{M}$. The class graph is acyclic by criterion WF-PROG-5, so $\varphi_1\overline{M} = \varphi_2\overline{N}$. Criterion WF-PROG-1 then yields $\mathit{impl}_1 = \mathit{impl}_2$. Hence, $\overline{X} = \overline{Y}$ and $\overline{M} = \overline{N}$. We have $\overline{X} \subseteq \mathsf{ftv}(\overline{M})$ by criterion WF-IMPL-2, so with $\varphi_1\overline{M} = \varphi_2\overline{N}$ also $\varphi_1 = \varphi_2$. □

Lemma B.3.2 (Determinacy of method lookup).

(i) *If* $\mathsf{getmdef}^{\mathsf{c}}(m, N) = \mathit{mdef}$ *and* $\mathsf{getmdef}^{\mathsf{c}}(m, N) = \mathit{mdef}'$ *then* $\mathit{mdef} = \mathit{mdef}'$*.*

(ii) *If* $\mathsf{getmdef}^{\mathsf{i}}(m, N, \overline{N}) = \mathit{mdef}$ *and* $\mathsf{getmdef}^{\mathsf{i}}(m, N, \overline{N}) = \mathit{mdef}'$ *then* $\mathit{mdef} = \mathit{mdef}'$*.*

(iii) *If* $\mathsf{getsmdef}(m, K, \overline{N}) = \mathit{mdef}$ *and* $\mathsf{getsmdef}(m, K, \overline{N}) = \mathit{mdef}'$ *then* $\mathit{mdef} = \mathit{mdef}'$*.*

Proof. We prove the three claims separately.

(i) It is easy to see that both derivations must end with the same rule. The claim now follows with a routine rule induction.

(ii) We first prove that $N_1 \sqcup N_2 = M$ and $N_1 \sqcup N_2 = M'$ imply $M = M'$. This proof is by induction on the derivations of $N_1 \sqcup N_2 = M$ and $N_1 \sqcup N_2 = M'$. If both derivations end with the same rule then the claim follows directly (rules LUB-RIGHT and LUB-LEFT) or via the I.H. (rule LUB-SUPER). Otherwise, one derivation ends with rule LUB-RIGHT and the other with rule LUB-LEFT. Then $N_1 \trianglelefteq_{\mathsf{c}} N_2$ and $N_2 \trianglelefteq_{\mathsf{c}} N_1$, so $M = N_2 = N_1 = M'$ as the class graph is acyclic by criterion WF-PROG-5.

We then get that $\bigsqcup \mathscr{N} = M$ and $\bigsqcup \mathscr{N} = M'$ imply $M = M'$. From this we have that $\mathsf{resolve}_X(\overline{T}, \overline{N}) = M$ and $\mathsf{resolve}_X(\overline{T}, \overline{N}) = M'$ imply $M = M'$.

The claim now follows with Lemma B.3.1.

(iii) Follows with Lemma B.3.1. □

Lemma B.3.3 (Determinacy of top-level evaluation). *If $e \longmapsto e'$ and $e \longmapsto e''$ then $e' = e''$.*

Proof. Case distinction on the form of e.

- *Case* $e = x$: Impossible.
- *Case* $e = e_0.f$: Then both reductions are due to rule DYN-FIELD. Hence, $e_0 = \textbf{new}\, N(\overline{v})$, $\mathsf{fields}(N) = \overline{U\, f}$, $f = f_j$, and $e' = v_j$. By Lemma B.2.37, fields is deterministic. Moreover, field shadowing is not allowed (criterion WF-CLASS-1), so f occurs exactly once in $\overline{f}$. Thus, $e'' = v_j = e'$.
- *Case* $e = e_0.m\texttt{<}\overline{T}\texttt{>}(\overline{e})$: Identifier sets for class and interface methods are disjoint (see Convention 3.4), so the two reductions are either both due to rule DYN-INVOKE-CLASS or both due to rule DYN-INVOKE-IFACE. In any case, the claim follows with Lemma B.3.2.
- *Case* $e = K[\overline{T}].m\texttt{<}\overline{U}\texttt{>}(\overline{e})$: The claim follows from Lemma B.3.2.
- *Case* $e = \textbf{new}\, N(\overline{e})$: Impossible.
- *Case* $e = (T)\, e_0$: Obvious.

End case distinction on the form of e. □

Lemma B.3.4. *Assume $\mathcal{E}_1[e_1] = \mathcal{E}_2[e_2]$. If $e_1 \longmapsto e_1'$ and $e_2 \longmapsto e_2'$ then $\mathcal{E}_1 = \mathcal{E}_2$.*

Proof. We prove the claim by induction on the combined size of $\mathcal{E}_1$ and $\mathcal{E}_2$. A case distinction on the form of $\mathcal{E}_1[e_1]$ reveals that either $\mathcal{E}_1 = \square = \mathcal{E}_2$ or that $\mathcal{E}_1$ and $\mathcal{E}_2$ are identical up to sub-contexts $\mathcal{E}_1'$ and $\mathcal{E}_2'$ with $\mathcal{E}_1'[e_1] = \mathcal{E}_2'[e_2]$. In the first case, the claim is immediate. In the second case, we get by the I.H. that $\mathcal{E}_1' = \mathcal{E}_2'$. But then also $\mathcal{E}_1 = \mathcal{E}_2$. □

Proof of Theorem 3.20. By rule DYN-CONTEXT, we have that $e = \mathcal{E}[\tilde{e}]$, $\tilde{e} \longmapsto \tilde{e}'$, $e' = \mathcal{E}[\tilde{e}']$, and that $e = \mathcal{E}'[\hat{e}]$, $\hat{e} \longmapsto \hat{e}'$, $e'' = \mathcal{E}'[\hat{e}']$. By Lemma B.3.4 we get $\mathcal{E} = \mathcal{E}'$, so we have $\tilde{e} = \hat{e}$. By Lemma B.3.3 we then get $\tilde{e}' = \hat{e}'$. Hence, $e' = e''$. □

B.4 Deciding Constraint Entailment and Subtyping

This section proves Theorem 3.24 (termination of $\texttt{unify}_{\leq}$), Theorem 3.25 (soundness of algorithmic entailment and subtyping with respect to their quasi-algorithmic variants), Theorem 3.26 (completeness of algorithmic entailment and subtyping with respect to their quasi-algorithmic variants), and Theorem 3.27 (termination of entailment and subtyping).

B.4.1 Proof of Theorem 3.24

Theorem 3.24 states that $\texttt{unify}_{\leq}$ terminates.

Definition B.4.1. The *weight of a type T with respect to a type environment* Δ, written $\mathsf{weight}_\Delta(T)$, is defined as follows:

$$\begin{aligned}
&\mathsf{weight}_\Delta(X) = 1 + \mathsf{max}(\{\mathsf{weight}_\Delta(T) \mid X\, \textbf{extends}\, T \in \Delta\})\\
&\mathsf{weight}_\Delta(N) = 1\\
&\mathsf{weight}_\Delta(K) = 1
\end{aligned}$$

By convention, $\mathsf{max}(\emptyset) = 0$. The definition of weight is proper (i.e., terminates) because Δ is contractive by criterion WF-TENV-1.

Proof of Theorem 3.24. Because syntactic unification is known to terminate [8], we only need to show that the rewrite rules in Figure 3.26 terminate. We define the following measure for a set of equations $\{T_1 \le^? U_1, \dots, T_n \le^? U_n\}$:

$$(\sum_{i=1}^{n} \mathsf{weight}_\Delta(T_i), \sum_{i=1}^{n} \mathsf{depth}(T_i)) \in \mathbb{N} \times \mathbb{N}$$

It is easy to see that each transformation rule from Figure 3.26 decreases this measure with respect to the usual lexicographic ordering on $\mathbb{N} \times \mathbb{N}$. □

B.4.2 Proof of Theorem 3.25

Theorem 3.25 states that algorithmic entailment and subtyping are sound with respect to quasi-algorithmic entailment and subtyping.

Lemma B.4.2. *If* $\Delta \Vdash_\mathrm{q}{}' \overline{U}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ *and* $\Delta; \texttt{false}; I \vdash_\mathrm{a} \overline{T} \uparrow \overline{U}$ *then it holds that* $\Delta \Vdash_\mathrm{q} \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$*.*

Proof. From the assumption $\Delta; \texttt{false}; I \vdash_\mathrm{a} \overline{T} \uparrow \overline{U}$ we get

$$(\forall i)\ \Delta \vdash_\mathrm{q}{}' T_i \le U_i$$
$$(\forall i)\ \text{if } T_i \ne U_i \text{ then } i \in \mathsf{pol}^-(I)$$

The claim now follows with rule ENT-Q-ALG-UP. □

Lemma B.4.3.

(*i*) *If* $\mathcal{D}_1 :: \Delta; \mathscr{G}; \beta \Vdash_\mathrm{a} \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ *then* $\Delta \Vdash_\mathrm{q}{}' \overline{U}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ *for some* $\overline{U}$ *with* $\Delta; \beta; I \vdash_\mathrm{a} \overline{T} \uparrow \overline{U}$*.*

(*ii*) *If* $\mathcal{D}_2 :: \Delta; \mathscr{G} \vdash_\mathrm{a} T \le U$ *then* $\Delta \vdash_\mathrm{q} T \le U$*.*

Proof. We proceed by induction on the combined height of $\mathcal{D}_1$ and $\mathcal{D}_2$.

(i) *Case distinction* on the last rule used in $\mathcal{D}_1$.

- *Case* ENT-ALG-EXTENDS: Impossible.
- *Case* ENT-ALG-ENV: Inverting the rule yields

$$R \in \Delta$$
$$\overline{G}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \in \mathsf{sup}(R)$$
$$\Delta; \beta; I \vdash_\mathrm{a} \overline{T} \uparrow \overline{G}$$

  With rule ENT-Q-ALG-ENV we have $\Delta \Vdash_\mathrm{q}{}' \overline{G}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$. Defining $\overline{U} = \overline{G}$ finishes this case.

- *Case* ENT-ALG-IMPL: Inverting the rule yields

$$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \dots \tag{B.4.1}$$
$$\Delta; \beta; I \vdash_\mathrm{a} \overline{T} \uparrow [\overline{W/X}]\overline{N} \tag{B.4.2}$$
$$\overline{V} = [\overline{W/X}]\overline{V'}$$
$$\Delta; \mathscr{G} \cup \{[\overline{W/X}]\overline{N}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\}; \texttt{false} \Vdash_\mathrm{a} [\overline{W/X}]\overline{P} \tag{B.4.3}$$

  Case distinction on the form of $[\overline{W/X}]P_i$.

- *Case* $[\overline{W/X}]P_i = \overline{T'}\,\textbf{implements}\,J\texttt{<}\overline{U'}\texttt{>}$: Assume Applying part (i) of the I.H. to (B.4.3) gives us the existence of $\overline{T''}$ such that

$$\Delta \Vdash_{\mathrm{q}}{}' \overline{T''}\,\textbf{implements}\,J\texttt{<}\overline{U'}\texttt{>}$$
$$\Delta; \texttt{false}; J \vdash_{\mathrm{a}} \overline{T'} \uparrow \overline{T''}$$

With Lemma B.4.2 we then have

$$\Delta \Vdash_{\mathrm{q}} \overline{T'}\,\textbf{implements}\,J\texttt{<}\overline{U'}\texttt{>}$$

- *Case* $[\overline{W/X}]P_i = T'\,\textbf{extends}\,U'$: Inverting the derivation in (B.4.3) yields

$$\Delta; \mathscr{G} \cup \{[\overline{W/X}]\overline{N}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\} \vdash_{\mathrm{a}} T' \leq U'$$

Applying part (ii) of the I.H. yields $\Delta \vdash_{\mathrm{q}} T' \leq U'$. Thus

$$\Delta \Vdash_{\mathrm{q}} T'\,\textbf{extends}\,U'$$

with rule ENT-Q-ALG-EXTENDS.

End case distinction on the form of $[\overline{W/X}]P_i$. Thus, we have

$$\Delta \Vdash_{\mathrm{q}} [\overline{W/X}]\overline{P} \tag{B.4.4}$$

With (B.4.1), (B.4.4), and rule ENT-Q-ALG-IMPL we get

$$\Delta \Vdash_{\mathrm{q}}{}' [\overline{W/X}]\overline{N}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$$

Define $\overline{U} = [\overline{W/X}]\overline{N}$; then (B.4.2) finishes this case.

- *Case* ENT-ALG-IFACE₁: We then have $\overline{T} = T$ for some T. Inverting the rule yields

$$\Delta; \beta; I \vdash_{\mathrm{a}} T \uparrow I\texttt{<}\overline{V}\texttt{>}$$
$$1 \in \mathsf{pol}^{+}(I)$$
$$\mathsf{non\text{-}static}(I)$$

By rule ENT-Q-ALG-IFACE, we have $\Delta \Vdash_{\mathrm{q}}{}'\ I\texttt{<}\overline{V}\texttt{>}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$. Defining $\overline{U} = I\texttt{<}\overline{V}\texttt{>}$ finishes this case.

- *Case* ENT-ALG-IFACE₂: Then $\overline{T} = J\texttt{<}\overline{W}\texttt{>}$ for some $J\texttt{<}\overline{W}\texttt{>}$. Inverting the rule yields

$$1 \in \mathsf{pol}^{+}(J)$$
$$\mathsf{non\text{-}static}(J)$$
$$J\texttt{<}\overline{W}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{V}\texttt{>}$$

The claim now follows with rule ENT-Q-ALG-IFACE.

End case distinction on the last rule used in $\mathcal{D}_1$.

(ii) *Case distinction* on the last rule used in $\mathcal{D}_2$.

- *Case* SUB-ALG-KERNEL: Inverting the rule yields $\Delta \vdash_{\mathrm{q}}{}'\ T \leq U$, so the claim follows with rule SUB-Q-ALG-KERNEL.

- *Case* SUB-ALG-IMPL: Then $U = I\texttt{<}\overline{V}\texttt{>}$ for some $I\texttt{<}\overline{V}\texttt{>}$. Inverting the rule yields

  $$\Delta; \mathscr{G}; \texttt{true} \Vdash_{\mathrm{a}} T\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$$

  Applying part (i) of the I.H. gives us the existence of T' such that

  $$\Delta \Vdash_{\mathrm{q}}{}' \; T'\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$$
  $$\Delta; \texttt{true}; I \vdash_{\mathrm{a}} T \uparrow T'$$

  Inverting the derivation of $\Delta; \texttt{true}; I \vdash_{\mathrm{a}} T \uparrow T'$ yields $\Delta \vdash_{\mathrm{q}}{}' \; T \leq T'$. An application of rule SUB-Q-ALG-IMPL now proves the claim.

End case distinction on the last rule used in $\mathcal{D}_2$. □

Proof of Theorem 3.25. We prove both claims separately.

(i) The derivation of $\Delta \Vdash_{\mathrm{a}} \mathcal{P}$ ends with rule ENT-ALG-MAIN. Inverting the rule yields

$$\mathcal{D} :: \Delta; \emptyset; \texttt{false} \Vdash_{\mathrm{a}} \mathcal{P}$$

Case distinction on the form of $\mathcal{P}$.

- *Case* $\mathcal{P} = T\,\textbf{extends}\,U$: Then $\mathcal{D}$ ends with rule ENT-ALG-EXTENDS. Inverting the rule yields $\Delta; \emptyset \vdash_{\mathrm{a}} T \leq U$. By Lemma B.4.3 we get $\Delta \vdash_{\mathrm{q}} T \leq U$, thus $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ by rule ENT-Q-ALG-EXTENDS,
- *Case* $\mathcal{P} = \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$: Applying Lemma B.4.3 to $\mathcal{D}$ yields the existence of $\overline{U}$ such that

  $$\Delta \Vdash_{\mathrm{q}}{}' \; \overline{U}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$$
  $$\Delta; \texttt{false}; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$$

  We then get $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ by Lemma B.4.2.

End case distinction on the form of $\mathcal{P}$.

(ii) The derivation of $\Delta \vdash_{\mathrm{a}} T \leq U$ ends with rule SUB-ALG-MAIN. Inverting the rule yields $\Delta; \emptyset \vdash_{\mathrm{a}} T \leq U$. The claim now follows with Lemma B.4.3. □

B.4.3 Proof of Theorem 3.26

Theorem 3.26 states that algorithmic entailment and subtyping are complete with respect to quasi-algorithmic entailment and subtyping.

The algorithmic formulation of entailment and subtyping restricts derivations to certain forms through the use of a goal cache $\mathscr{G}$. Thus, the section starts by proving various properties of derivations in general before turning to derivations that are specific to algorithmic entailment and subtyping.

Definition B.4.4 (Small derivations). A derivation $\mathcal{D}$ is *small* if, and only if, its direct subderivations are small and all its proper subderivations end with a conclusion other then the conclusion of $\mathcal{D}$.

Remember that $\mathcal{D} :: \mathcal{J}$ denotes that $\mathcal{D}$ is a derivation of judgment $\mathcal{J}$. Moreover, we write $\mathcal{D}; \mathfrak{r} :: \mathcal{J}$ if $\mathcal{D} :: \mathcal{J}$ and $\mathcal{D}$ ends with an application of rule $\mathfrak{r}$. The notation $\mathsf{height}(\mathcal{D})$ denotes the height of a derivation $\mathcal{D}$.

Lemma B.4.5. *Let $\mathcal{J}$ be a judgment such that the inference rules defining $\mathcal{J}$ do not put restrictions on properties of derivations. Now suppose $\mathcal{D} :: \mathcal{J}$. Then there exists $\widehat{\mathcal{D}} :: \mathcal{J}$ such that $\widehat{\mathcal{D}}$ is small and* $\mathsf{height}(\widehat{\mathcal{D}}) \leq \mathsf{height}(\mathcal{D})$.

Proof. By induction on the height of $\mathcal{D}$. If $\mathcal{D}$ is already small then we are done. In the following, $\mathfrak{r}$ ranges over rule names. Assume $\mathcal{D}$ is not small. Hence

$$\frac{\mathcal{D}_1 :: \mathcal{J}_1 \qquad \dots \qquad \mathcal{D}_n :: \mathcal{J}_n}{\mathcal{D} :: \mathcal{J}}\;\mathfrak{r}$$

By applying the I.H. we get $\mathcal{D}'_i :: \mathcal{J}_i$ for all $i \in [n]$ whereby $\mathcal{D}'_i$ is small and $\mathsf{height}(\mathcal{D}'_i) \leq \mathsf{height}(\mathcal{D}_i)$. An application of rule $\mathfrak{r}$ now yields $\mathcal{D}' :: \mathcal{J}$ such that $\mathsf{height}(\mathcal{D}') \leq \mathsf{height}(\mathcal{D})$. If $\mathcal{D}'$ is small then we are done. Otherwise, we have the following situation:

$$\frac{\begin{array}{c}\mathcal{D}'' :: \mathcal{J} \\ \vdots \end{array}}{\mathcal{D} :: \mathcal{J}}\;\mathfrak{r}$$

with $\mathsf{height}(\mathcal{D}'') < \mathsf{height}(\mathcal{D})$. We now apply the I.H. to $\mathcal{D}'' :: \mathcal{J}$ and get $\mathcal{D}''' :: \mathcal{J}$ such that $\mathcal{D}'''$ is small and $\mathsf{height}(\mathcal{D}''') \leq \mathsf{height}(\mathcal{D}'') < \mathsf{height}(\mathcal{D})$. □

Lemma B.4.6. *If $\mathcal{D}'$ is a subderivation of a small derivation $\mathcal{D}$, then $\mathcal{D}'$ is also small.*

Proof. By induction on the height of $\mathcal{D}$. If $\mathcal{D}' = \mathcal{D}$ then the claim is immediate. Otherwise, there exist a direct subderivation $\mathcal{D}''$ of $\mathcal{D}$ such that $\mathcal{D}'$ is a subderivation of $\mathcal{D}''$. By Definition B.4.4, we know that $\mathcal{D}''$ is small. Applying the I.H. proves that $\mathcal{D}'$ is small. □

Definition B.4.7 (Entailment goals). Let $\mathcal{D}$ be a derivation. The set of *entailment goals* occurring in $\mathcal{D}$ is defined as follows:

$$\mathsf{goals}(\mathcal{D}) = \{R \mid \mathcal{D} \text{ contains a subderivation } \mathcal{D}'; \textsc{ent-q-alg-impl} :: \Delta \Vdash_{\mathrm{q}} R\}$$

Lemma B.4.8. *If $\mathcal{D}'$ is a subderivation of $\mathcal{D}$ then $\mathsf{goals}(\mathcal{D}') \subseteq \mathsf{goals}(\mathcal{D})$.*

Proof. Obvious. □

Lemma B.4.9. *Suppose $\mathcal{D}; \textsc{ent-q-alg-impl} :: \Delta \Vdash_{\mathrm{q}} R$. If $\mathcal{D}$ is small and $\mathcal{D}'$ is a proper subderivation of $\mathcal{D}$, then $R \notin \mathsf{goals}(\mathcal{D}')$.*

Proof. Assume $R \in \mathsf{goals}(\mathcal{D}')$. Hence, $\mathcal{D}'$ has a subderivation

$$\mathcal{D}''; \textsc{ent-q-alg-impl} :: \Delta \Vdash_{\mathrm{q}} R$$

But this is a contradiction to $\mathcal{D}$ being small because $\mathcal{D}''$ is a proper subderivation of $\mathcal{D}$. □

Lemma B.4.10.

(*i*) *If $\mathcal{D}_1 :: \Delta \Vdash_{\mathrm{q}} \mathcal{P}$ and $\mathcal{D}_1$ is small, then $\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \mathcal{P}$ for all β and all $\mathscr{G}$ with $\mathsf{goals}(\mathcal{D}_1) \cap \mathscr{G} = \emptyset$.*

(*ii*) *If $\mathcal{D}_2 :: \Delta \Vdash_{\mathrm{q}}' \overline{U}$ **implements** $I{<}\overline{V}{>}$ and $\mathcal{D}_2$ is small and $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$, then $\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}}$ $\overline{T}$ **implements** $I{<}\overline{V}{>}$ for all $\mathscr{G}$ with $\mathsf{goals}(\mathcal{D}_2) \cap \mathscr{G} = \emptyset$.*

(*iii*) *If $\mathcal{D}_3 :: \Delta \vdash_{\mathrm{q}} T \leq U$ and $\mathcal{D}_3$ is small, then $\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq U$ for all $\mathscr{G}$ with $\mathsf{goals}(\mathcal{D}_3) \cap \mathscr{G} = \emptyset$.*

Proof. We proceed by induction on the combined height of $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$.

(i) Suppose $\mathscr{G}$ is a set of entailment goals such that $\mathsf{goals}(\mathcal{D}_1) \cap \mathscr{G} = \emptyset$ and let $\beta \in \{\mathtt{false}, \mathtt{true}\}$.

Case distinction on the last rule used in $\mathcal{D}_1$.

- *Case* rule ENT-Q-ALG-EXTENDS: We then have $\mathcal{P} = T\,\mathbf{extends}\,U$. By inverting the rule, we get $\mathcal{D}'_1 :: \Delta \vdash_{\mathrm{q}} T \le U$ such that $\mathcal{D}'_1$ is a subderivation of $\mathcal{D}_1$. From Lemma B.4.6 we know that $\mathcal{D}'_1$ is small and Lemma B.4.8 gives us $\mathsf{goals}(\mathcal{D}'_1) \cap \mathscr{G} = \emptyset$. Applying part (iii) of the I.H. yields $\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \le U$, so the claim follows with rule ENT-ALG-EXTENDS.
- *Case* rule ENT-Q-ALG-UP: We then have

$$\frac{(\forall i)\ \text{if } T_i \neq U_i \text{ then } i \in \mathsf{pol}^-(I) \qquad \begin{array}{c}(\forall i)\ \Delta \vdash_{\mathrm{q}}{}' T_i \le U_i \\ \mathcal{D}'_1 :: \Delta \Vdash_{\mathrm{q}}{}' \overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>}\end{array}}{\mathcal{D}_1 :: \Delta \Vdash_{\mathrm{q}} \underbrace{\overline{T}\,\mathbf{implements}\,I{<}\overline{V}{>}}_{=\mathcal{P}}}$$

  Thus, we have

$$\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$$

  by rule ENT-ALG-LIFT. Moreover, $\mathcal{D}_1$ is small so $\mathcal{D}'_1$ is small by Lemma B.4.6. Furthermore,

$$\mathsf{goals}(\mathcal{D}'_1) \cap \mathscr{G} = \emptyset$$

  with Lemma B.4.8 and $\mathsf{goals}(\mathcal{D}_1) \cap \mathscr{G} = \emptyset$. Applying part (ii) of the I.H. now yields $\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \mathcal{P}$.

End case distinction on the last rule used in $\mathcal{D}_1$.

(ii) *Case distinction* on the last rule used in $\mathcal{D}_2$.

- *Case* rule ENT-Q-ALG-ENV: We have

$$\overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>} = \overline{G}\,\mathbf{implements}\,I{<}\overline{V}{>}$$

  Inverting the rule yields $R \in \Delta$ and $\overline{G}\,\mathbf{implements}\,I{<}\overline{V}{>} \in \mathsf{sup}(R)$. The claim now follows with the assumption $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{G}$ by rule ENT-ALG-ENV.
- *Case* rule ENT-Q-ALG-IMPL: We have

$$\frac{\mathbf{implementation}{<}\overline{X}{>}\ I{<}\overline{V'}{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots \qquad \Delta \Vdash_{\mathrm{q}} [\overline{W/X}]\overline{P}}{\mathcal{D}_2 :: \Delta \Vdash_{\mathrm{q}}{}' \underbrace{[\overline{W/X}](\overline{N}\,\mathbf{implements}\,I{<}\overline{V'}{>})}_{=\overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>}}} \tag{B.4.5}$$

  Suppose $\mathcal{D}'_i :: \Delta \Vdash_{\mathrm{q}} [\overline{W/X}]P_i$, let $\mathscr{G}$ be a set of entailment goals such that $\mathsf{goals}(\mathcal{D}_2) \cap \mathscr{G} = \emptyset$, and assume $\beta \in \{\mathtt{false}, \mathtt{true}\}$.

  $\mathcal{D}_2$ is small by assumption, so $\mathcal{D}'_i$ is small with Lemma B.4.6. Using Lemma B.4.9 we get $\overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>} \notin \mathsf{goals}(\mathcal{D}'_i)$. Moreover, $\mathsf{goals}(\mathcal{D}'_i) \subseteq \mathsf{goals}(\mathcal{D}_2)$. Because $\mathsf{goals}(\mathcal{D}_2) \cap \mathscr{G} = \emptyset$ we then have

$$\mathsf{goals}(\mathcal{D}'_i) \cap (\mathscr{G} \cup \{\overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>}\}) = \emptyset$$

  By part (i) of the I.H. we now get

$$\Delta; \mathscr{G} \cup \{\overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>}\}; \mathtt{false} \vdash_{\mathrm{a}} [\overline{W/X}]P_i \tag{B.4.6}$$

  Moreover, $\overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>} \in \mathsf{goals}(\mathcal{D}_2)$ by Definition B.4.7 and $\mathsf{goals}(\mathcal{D}_2) \cap \mathscr{G} = \emptyset$ by the assumption, so

$$\overline{U}\,\mathbf{implements}\,I{<}\overline{V}{>} \notin \mathscr{G} \tag{B.4.7}$$

Furthermore, $\overline{U} = [\overline{W/X}]\overline{N}$ from (B.4.5) and $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$ by the assumption; hence

$$\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow [\overline{W/X}]\overline{N} \tag{B.4.8}$$

We conclude by using rule ENT-ALG-IMPL

$$\dfrac{\begin{array}{c} [\overline{W/X}]\overline{N}\,\mathbf{implements}\,I{<}\overline{V}{>} \notin \mathscr{G} \quad \text{from (B.4.7) and (B.4.5)} \\ \mathbf{implementation}{<}\overline{X}{>}\ I{<}\overline{V'}{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots \quad \text{from (B.4.5)} \\ \Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow [\overline{W/X}]\overline{N} \quad \text{from (B.4.8)} \\ \overline{V} = [\overline{W/X}]\overline{V'} \quad \text{from (B.4.5)} \\ \Delta; \mathscr{G} \cup \{[\overline{W/X}]\overline{N}\,\mathbf{implements}\,I{<}\overline{V}{>}\}; \mathtt{false} \Vdash_{\mathrm{a}} [\overline{W/X}]\overline{P} \quad \text{from (B.4.6)} \end{array}}{\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \overline{T}\,\mathbf{implements}\,I{<}\overline{V}{>}}$$

- *Case* rule ENT-Q-ALG-IFACE: We then have $\overline{U} = J{<}\overline{W}{>}$ such that

$$1 \in \mathsf{pol}^{+}(J) \tag{B.4.9}$$
$$\mathsf{non\text{-}static}(J) \tag{B.4.10}$$
$$J{<}\overline{W}{>} \trianglelefteq_{\mathrm{i}} I{<}\overline{V}{>} \tag{B.4.11}$$

With Lemma B.1.18 and Lemma B.1.19 we get

$$1 \in \mathsf{pol}^{+}(I) \tag{B.4.12}$$
$$\mathsf{non\text{-}static}(I) \tag{B.4.13}$$

With the assumption $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$ we get $\overline{T} = T$ for some T and

$$\Delta \vdash_{\mathrm{q}}{}' \, T \leq J{<}\overline{W}{>}$$
$$\beta \text{ or } T = J{<}\overline{W}{>} \text{ or } 1 \in \mathsf{pol}^{-}(I) \tag{B.4.14}$$

With (B.4.11), rule SUB-Q-ALG-IFACE, and Lemma B.1.7 we get

$$\Delta \vdash_{\mathrm{q}}{}' \, T \leq I{<}\overline{V}{>} \tag{B.4.15}$$

Case distinction on the form of T.

  - *Case* $T \neq J{<}\overline{W}{>}$: With (B.4.14) we get β or $1 \in \mathsf{pol}^{-}(I)$. With (B.4.15) and rule ENT-ALG-LIFT we get $\Delta; \beta; I \vdash_{\mathrm{a}} T \uparrow I{<}\overline{V}{>}$. With (B.4.12), (B.4.13), and rule ENT-ALG-IFACE₁ we get

$$\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \overline{T}\,\mathbf{implements}\,I{<}\overline{V}{>}$$

  - *Case* $T = J{<}\overline{W}{>}$: The claim then follows with (B.4.9), (B.4.10), (B.4.11), and rule ENT-ALG-IFACE₂.

End case distinction on the form of T.

End case distinction on the last rule used in $\mathcal{D}_2$.

(iii) *Case distinction* on the last rule used in $\mathcal{D}_3$.

- *Case* rule SUB-Q-ALG-KERNEL: By inverting the rule, we get $\Delta \vdash_{\mathrm{q}}{}' \, T \leq U$, so $\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq U$ by SUB-ALG-KERNEL.

- *Case* rule SUB-Q-ALG-IMPL: We have $U = I\texttt{<}\overline{V}\texttt{>}$ for some $I\texttt{<}\overline{V}\texttt{>}$ such that

$$\frac{\Delta \vdash_{\mathrm{q}}{}' T \leq T' \qquad \mathcal{D}_3' :: \Delta \Vdash_{\mathrm{q}}{}' T' \,\textbf{implements}\, I\texttt{<}\overline{V}\texttt{>}}{\mathcal{D}_3 :: \Delta \vdash_{\mathrm{q}} T \leq I\texttt{<}\overline{V}\texttt{>}}$$

By rule ENT-ALG-LIFT

$$\Delta; \texttt{true}; I \vdash_{\mathrm{a}} T \uparrow T'$$

Because $\mathcal{D}_3$ is small, we get with Lemma B.4.6 that $\mathcal{D}_3'$ is small. Moreover, by Lemma B.4.8 $\mathsf{goals}(\mathcal{D}_3') \subseteq \mathsf{goals}(\mathcal{D}_3)$, so with the assumption $\mathsf{goals}(\mathcal{D}_3) \cap \mathscr{G} = \emptyset$ we have

$$\mathsf{goals}(\mathcal{D}_3') \cap \mathscr{G} = \emptyset$$

Applying part (ii) of the I.H. now yields

$$\Delta; \mathscr{G}; \texttt{true} \Vdash_{\mathrm{a}} T \,\textbf{implements}\, I\texttt{<}\overline{V}\texttt{>}$$

so we get $\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq I\texttt{<}\overline{V}\texttt{>}$ by rule SUB-ALG-IMPL.

End case distinction on the last rule used in $\mathcal{D}_3$. □

Proof of Theorem 3.26. By Lemma B.4.5 we may safely assume that the two derivations given are small. Then the two claims follow from Lemma B.4.10 and applications of rules ENT-ALG-MAIN and SUB-ALG-MAIN. □

B.4.4 Proof of Theorem 3.27

Theorem 3.27 states that the entailment and subtyping algorithms induced by the rules in Figure 3.25 and by the rules for quasi-algorithmic kernel subtyping in Figure 3.16 terminate. Figure B.3 and Figure B.4 define these algorithms in pseudo code.

Lemma B.4.11. *The algorithms in Figure B.3 and Figure B.4 are equivalent to the algorithmic entailment and subtyping rules defined in Figure 3.25 and the rules for quasi-algorithmic subtyping defined in Figure 3.16.*

- $\Delta \Vdash_{\mathrm{a}} \mathcal{P}$ *if, and only if,* $\texttt{entails}(\Delta, \mathcal{P})$ *returns* $\texttt{true}$*.*
- $\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \mathcal{P}$ *if, and only if,* $\texttt{entailsAux}(\Delta, \mathscr{G}, \beta, \mathcal{P})$ *returns* $\texttt{true}$*.*
- $\Delta \vdash_{\mathrm{a}} T \leq U$ *if, and only if,* $\texttt{sub}(\Delta, T, U)$ *returns* $\texttt{true}$*.*
- $\Delta; \mathscr{G} \vdash_{\mathrm{a}} T \leq U$ *if, and only if,* $\texttt{subAux}(\Delta, \mathscr{G}, T, U)$ *returns* $\texttt{true}$*.*
- $\Delta \vdash_{\mathrm{q}}{}' T \leq U$ *if, and only if,* $\texttt{sub'}(\Delta, T, U)$ *returns* $\texttt{true}$*.*
- $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$ *if, and only if,* $\texttt{lift}(\Delta, \beta, I, \overline{T}, \overline{U})$ *returns* $\texttt{true}$*.*

Proof. Completeness (⇒) follows by straightforward rule induction. Soundness (⇐) follows by induction on the depth of the recursion. □

The termination proof requires that the goal cache $\mathscr{G}$ in an invocation of either `entailsAux` or `subAux` has a finite upper bound (Lemmas B.4.19 and B.4.21). The set of *entailment candidates* of a constraint $\mathcal{P}$ with respect to a type environment Δ, written $\mathsf{cand}_{\Delta}(\mathcal{P})$, plays a crucial role in the definition of that upper bound. Figure B.5 defines $\mathsf{cand}_{\Delta}(\mathcal{P})$ formally.

Figure B.3 Constraint entailment algorithm.

```
entails($\Delta, \mathcal{P}$) { return entailsAux($\Delta, \emptyset$, false, $\mathcal{P}$); }
entailsAux($\Delta, \mathscr{G}, \beta, \mathcal{P}$) {
  switch ($\mathcal{P}$) {
    case $T$ extends $U$: return subAux($\Delta, \mathscr{G}, T, U$);
    case $\overline{T}$ implements $I$<$\overline{V}$>:
      // rule ENT-ALG-ENV
      for ($R \in \Delta$, $\overline{G}$ implements $I$<$\overline{V}$> $\in$ sup($R$)) {
        if (lift($\Delta, \beta, I, \overline{T}, \overline{G}$)) return true;
      }
      switch ($\overline{T}$) {
        // rule ENT-ALG-IFACE$_1$
        case $T$:
          if (lift($\Delta, \beta, I, T, I$<$\overline{V}$>) && $1 \in$ pol$^+(I)$ && non-static($I$))
            return true;
        // rule ENT-ALG-IFACE$_2$
        case $J$<$\overline{W}$>:
          if ($1 \in$ pol$^+(J)$ && $J$<$\overline{W}$> $\trianglelefteq_{\mathrm{i}}$ $I$<$\overline{V}$> && non-static($J$))
            return true;
      }
      // rule ENT-ALG-IMPL
      for implementation<$\overline{X}$> $I$<$\overline{W}$> [$\overline{N}$] where $\overline{P}^n$ ... {
        if (unify$_\leq$($\Delta, \overline{X}, \{\overline{T_i \leq^? N_i}\}$) == $\varphi$ && lift($\Delta, \beta, I, \overline{T}, \varphi\overline{N}$)
            && $\overline{V}$==$\varphi\overline{W}$ && ($\varphi\overline{N}$) implements $I$<$\overline{V}$> $\notin \mathscr{G}$) {
          $\mathscr{G}_0$ = $\mathscr{G} \cup \{\varphi\overline{N}$ implements $I$<$\overline{V}$>$\}$;
          if ($\forall i \in [n]$, entailsAux($\Delta, \mathscr{G}_0$, false, $\varphi P_i$)) return true;
        }
      }
      return false;      // no rule applicable
  }
}

lift($\Delta, \beta, I, \overline{T}^n, \overline{U}^m$) {
  return ($n$==$m$ && $\forall i \in [n]$,(sub'($\Delta, T_i, U_i$) &&
                              ($\beta$ || $T_i$==$U_i$ || $i$ $\in$ pol$^-(I)$)));
}
```

Figure B.4 Subtyping algorithm.

```
sub($\Delta$,$T$,$U$) { return subAux($\Delta$,$\emptyset$,$T$,$U$); }
subAux($\Delta$,$\mathscr{G}$,$T$,$U$) {
  if (sub'($\Delta$,$T$,$U$)) return true;
  switch (U) {
    case $K$: return entailsAux($\Delta$,$\mathscr{G}$,true,$T\,\mathbf{implements}\,K$);
  }
  return false;
}

sub'($\Delta$,$T$,$U$) {
  switch ($T$,$U$) {
    case (_,$\mathit{Object}$): return true;
    case ($X$,$X$): return true;
    case ($X$,_):
      for $X\,\mathbf{extends}\,V \in \Delta$ { if (sub'($\Delta$,$V$,$U$)) return true; }
      return false;
    case ($N_1$,$N_2$): return $N_1 \trianglelefteq_{\mathrm{c}} N_2$;
    case ($K_1$,$K_2$): return $K_1 \trianglelefteq_{\mathrm{i}} K_2$;
  }
  return false;
}
```

Figure B.5 Entailment candidates.

$\boxed{\mathcal{P} \in \mathsf{cand}_\Delta(\mathcal{Q})}$

CAND-CLOSURE

$$\frac{\overline{U} \subseteq \mathsf{closure}_\Delta(\overline{T})}{\overline{U}\,\mathbf{implements}\,K \in \mathsf{cand}_\Delta(\overline{T}\,\mathbf{implements}\,K)}$$

CAND-IMPL$_1$

$$\frac{\begin{array}{c}\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots\\ \overline{U} \subseteq \mathsf{closure}_\Delta(\overline{T}) \qquad \overline{U'} \subseteq \mathsf{closure}_\Delta(\overline{T}) \qquad P_i = \overline{W}\,\mathbf{implements}\,L\end{array}}{\overline{U}\,\mathbf{implements}\,[\overline{U'/X}]L \in \mathsf{cand}_\Delta(\overline{T}\,\mathbf{implements}\,K)}$$

CAND-IMPL$_2$

$$\frac{\begin{array}{c}\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots\\ U \in \mathsf{closure}_\Delta(\overline{T}) \qquad \overline{U'} \subseteq \mathsf{closure}_\Delta(\overline{T}) \qquad P_i = W\,\mathbf{extends}\,W'\end{array}}{U\,\mathbf{extends}\,[\overline{U'/X}]W' \in \mathsf{cand}_\Delta(\overline{T}\,\mathbf{implements}\,K)}$$

CAND-EXTENDS

$$\frac{\mathcal{P} \in \mathsf{cand}_\Delta(T\,\mathbf{implements}\,K)}{\mathcal{P} \in \mathsf{cand}_\Delta(T\,\mathbf{extends}\,K)}$$

Definition B.4.12. For a constraint $\mathcal{P}$, we define $\mathsf{left}(\mathcal{P})$ as follows:

$$\begin{aligned} \mathsf{left}(\overline{T}\,\textbf{implements}\,K) &= \overline{T} \\ \mathsf{left}(T\,\textbf{extends}\,U) &= U \end{aligned}$$

Lemma B.4.13. *If* $\mathcal{P} \in \mathsf{cand}_{\Delta}(\mathcal{Q})$ *then* $\mathsf{left}(\mathcal{P}) \subseteq \mathsf{closure}_{\Delta}(\mathsf{left}(\mathcal{Q}))$.

Proof. Straightforward case distinction on the last rule used in the derivation of $\mathcal{P} \in \mathsf{cand}_{\Delta}(\mathcal{Q})$. □

Lemma B.4.14. *If* $\mathscr{T}_3 \subseteq \mathsf{closure}_{\Delta}(\mathscr{T}_2)$ *and* $\mathscr{T}_2 \subseteq \mathsf{closure}_{\Delta}(\mathscr{T}_1)$ *then* $\mathscr{T}_3 \subseteq \mathsf{closure}_{\Delta}(\mathscr{T}_1)$.

Proof. It suffices to show that $T \in \mathsf{closure}_{\Delta}(\mathscr{T}_2)$ implies $T \in \mathsf{closure}_{\Delta}(\mathscr{T}_1)$ for all T. The proof is a straightforward induction on the derivation of $T \in \mathsf{closure}_{\Delta}(\mathscr{T}_2)$. □

Lemma B.4.15. *If* $\mathcal{P} \in \mathsf{cand}_{\Delta}(\mathcal{Q})$ *then* $\mathsf{cand}_{\Delta}(\mathcal{P}) \subseteq \mathsf{cand}_{\Delta}(\mathcal{Q})$.

Proof. We show that $\mathcal{P}' \in \mathsf{cand}_{\Delta}(\mathcal{P})$ implies $\mathcal{P}' \in \mathsf{cand}_{\Delta}(\mathcal{Q})$ for all $\mathcal{P}'$.
Case distinction on the last rule used in the derivation of $\mathcal{P}' \in \mathsf{cand}_{\Delta}(\mathcal{P})$.

- *Case* CAND-CLOSURE: We then have

  $$\begin{aligned} \mathcal{P}' &= \overline{U}\,\textbf{implements}\,K \\ \mathcal{P} &= \overline{T}\,\textbf{implements}\,K \\ \overline{U} &\subseteq \mathsf{closure}_{\Delta}(\overline{T}) \end{aligned}$$

  By Lemma B.4.13 we have $\overline{T} \subseteq \mathsf{closure}_{\Delta}(\mathsf{left}(\mathcal{Q}))$, so with Lemma B.4.14

  $$\overline{U} \subseteq \mathsf{closure}_{\Delta}(\mathsf{left}(\mathcal{Q})) \tag{B.4.16}$$

  Case distinction on the last rule in the derivation of $\mathcal{P} \in \mathsf{cand}_{\Delta}(\mathcal{Q})$.

  - *Case* CAND-CLOSURE: Then $\mathcal{Q} = \overline{V}\,\textbf{implements}\,K$. With (B.4.16) we have $\overline{U} \subseteq \mathsf{closure}_{\Delta}(\overline{V})$, so $\mathcal{P}' \in \mathsf{cand}_{\Delta}(\mathcal{Q})$ by rule CAND-CLOSURE.
  - *Case* CAND-IMPL$_1$: Then

    $$\frac{\begin{array}{c}\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots \\ \overline{T} \subseteq \mathsf{closure}_{\Delta}(\overline{V}) \qquad \overline{T'} \subseteq \mathsf{closure}_{\Delta}(\overline{V}) \qquad P_i = \overline{W}\,\textbf{implements}\,K'\end{array}}{\overline{T}\,\textbf{implements}\ \underbrace{[\overline{T'/X}]K'}_{=K} \in \mathsf{cand}_{\Delta}(\underbrace{\overline{V}\,\textbf{implements}\,L}_{=\mathcal{Q}})}$$

    With (B.4.16) we have $\overline{U} \subseteq \mathsf{closure}_{\Delta}(\overline{V})$, so $\mathcal{P}' \in \mathsf{cand}_{\Delta}(\mathcal{Q})$ by rule CAND-IMPL$_1$.
  - *Case* CAND-IMPL$_2$: Impossible because $\mathcal{P}$ is not an **extends**-constraint.
  - *Case* CAND-EXTENDS: Then $\mathcal{Q} = V\,\textbf{extends}\,L$ and

    $$\mathcal{P} \in \mathsf{closure}_{\Delta}(V\,\textbf{implements}\,L)$$

    Because this derivation cannot end with rule CAND-EXTENDS, the claim follows with the same argumentation as in one of the three preceding cases.

  End case distinction on the last rule in the derivation of $\mathcal{P} \in \mathsf{cand}_{\Delta}(\mathcal{Q})$.

- *Case* CAND-IMPL$_1$: We then have

$$\frac{\begin{array}{c}\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots\\ \overline{U}\subseteq \mathsf{closure}_\Delta(\overline{T})\qquad \overline{U'}\subseteq\mathsf{closure}_\Delta(\overline{T})\qquad P_i=\overline{W}\,\textbf{implements}\,L\end{array}}{\underbrace{\overline{U}\,\textbf{implements}\,[\overline{U'/X}]L}_{=\mathcal{P}'}\in\mathsf{cand}_\Delta(\underbrace{\overline{T}\,\textbf{implements}\,K}_{=\mathcal{P}})}$$

  By Lemma B.4.13 we have $\overline{T}\subseteq\mathsf{closure}_\Delta(\mathsf{left}(\mathcal{Q}))$, so with Lemma B.4.14

$$\overline{U}\subseteq\mathsf{closure}_\Delta(\mathsf{left}(\mathcal{Q})) \tag{B.4.17}$$
$$\overline{U'}\subseteq\mathsf{closure}_\Delta(\mathsf{left}(\mathcal{Q})) \tag{B.4.18}$$

  If now $\mathcal{Q}=\overline{W'}\,\textbf{implements}\,L'$ for some $\overline{W'}$ and L', then the claim follows with rule CAND-IMPL$_1$. Otherwise, $\mathcal{Q}=W'\,\textbf{extends}\,W''$. Because $\mathcal{P}\in\mathsf{cand}_\Delta(\mathcal{Q})$, we must have that $W''=L'$ for some L'. With rule CAND-IMPL$_1$, we have $\mathcal{P}'\in\mathsf{cand}_\Delta(W'\,\textbf{implements}\,L')$, so the claim follows with rule CAND-EXTENDS.

- *Case* CAND-IMPL$_2$: The claim follows analogously to the preceding case, replacing CAND-IMPL$_1$ with CAND-IMPL$_2$.

- *Case* CAND-EXTENDS: Then $\mathcal{P}=T\,\textbf{extends}\,K$ and

$$\mathcal{P}'\in\mathsf{cand}_\Delta(T\,\textbf{implements}\,K)$$

  Because this derivation cannot end with rule CAND-EXTENDS, the claim follows with the same argumentation as in one of the three preceding cases.

End case distinction on the last rule used in the derivation of $\mathcal{P}'\in\mathsf{cand}_\Delta(\mathcal{P})$. □

Lemma B.4.16. *Assume* $\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots$ *and* $\overline{U}\subseteq\mathsf{closure}_\Delta(\overline{T})$. *Then* $[\overline{U/X}]P_i\in\mathsf{cand}_\Delta(\overline{T}\,\textbf{implements}\,K)$ *for all* i.

Proof. Case distinction on the form of P_i.

- *Case* $P_i=\overline{T'}\,\textbf{implements}\,K'$ for some $\overline{T'}$ and K': By criterion WF-IMPL-3 we have $\overline{T'}\subseteq\overline{X}$. Hence, $[\overline{U/X}]\overline{T'}\subseteq\overline{U}\subseteq\mathsf{closure}_\Delta(\overline{T}\,\textbf{implements}\,K)$. Thus

$$\frac{\begin{array}{c}\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots\\ [\overline{U/X}]\overline{T'}\subseteq\mathsf{closure}_\Delta(\overline{T})\qquad \overline{U}\subseteq\mathsf{closure}_\Delta(\overline{T})\qquad P_i=\overline{T'}\,\textbf{implements}\,K'\end{array}}{[\overline{U/X}]P_i\in\mathsf{cand}_\Delta(\overline{T}\,\textbf{implements}\,K)}\ \textsc{cand-impl}_1$$

- *Case* $P_i=T'\,\textbf{extends}\,T''$: By criterion WF-IMPL-3 we have $T'\in\overline{X}$. The claim now follows analogously to the preceding case, replacing rule CAND-IMPL$_1$ with CAND-IMPL$_2$.

End case distinction on the form of P_i. □

Definition B.4.17. The *call tree* of $\texttt{entailsAux}(\Delta,\mathscr{G},\beta,\mathcal{P})$ consists of a root node with label $\texttt{entailsAux}(\Delta,\mathscr{G},\beta,\mathcal{P})$ such that its subtrees are the call trees of all the direct recursive calls of `entailsAux` and `subAux`. The call tree of $\texttt{subAux}(\Delta,\mathscr{G},T,U)$ is defined analogously.

Definition B.4.18. Assume $\mathfrak{n}$ is a node in the call tree of `entailsAux` or `subAux`. The notation $\mathsf{cache}(\mathfrak{n})$ denote the set of goals cached at node $\mathfrak{n}$:

$$\mathsf{cache}(\texttt{entailsAux}(\Delta,\mathscr{G},\beta,\mathcal{P}))=\mathscr{G}$$
$$\mathsf{cache}(\texttt{subAux}(\Delta,\mathscr{G},T,U))=\mathscr{G}$$

The notation $\mathsf{cand}_\Delta(\mathfrak{n})$ denotes the entailment candidates at node $\mathfrak{n}$:

$$\begin{aligned}\mathsf{cand}_\Delta(\texttt{entailsAux}(\Delta', \mathscr{G}, \beta, \mathcal{P})) &= \mathsf{cand}_\Delta(\mathcal{P})\\ \mathsf{cand}_\Delta(\texttt{subAux}(\Delta', \mathscr{G}, T, U)) &= \mathsf{cand}_\Delta(T\,\textbf{extends}\,U)\end{aligned}$$

Lemma B.4.19. *If $\mathfrak{n}$ is a node in the call tree of $\texttt{entailsAux}(\Delta, \mathscr{G}, \beta, \mathcal{P})$ then $\mathsf{cache}(\mathfrak{n}) \subseteq \mathscr{G} \cup \mathsf{cand}_\Delta(\mathcal{P})$. Similarly, if $\mathfrak{n}$ is a node in the call tree of $\texttt{subAux}(\Delta, \mathscr{G}, T, U))$ then $\mathsf{cache}(\mathfrak{n}) \subseteq \mathscr{G} \cup \mathsf{cand}_\Delta(T\,\textbf{extends}\,U)$.*

Proof. We prove the following, stronger claim:

> *If $\mathfrak{n}$ is a node in the call tree of $\texttt{entailsAux}(\Delta, \mathscr{G}, \beta, \mathcal{P})$ define $\mathscr{M}$ as $\mathsf{cand}_\Delta(\mathcal{P})$. If $\mathfrak{n}$ is a node in the call tree of $\texttt{subAux}(\Delta, \mathscr{G}, T, U)$ define $\mathscr{M}$ as $\mathsf{cand}_\Delta(T\,\textbf{extends}\,U)$. In both cases, it holds that $\mathsf{cache}(\mathfrak{n}) \subseteq \mathscr{G} \cup \mathscr{M}$ and $\mathsf{cand}_\Delta(\mathfrak{n}) \subseteq \mathscr{M}$.*

The proof is by induction on the depth of $\mathfrak{n}$. If $\mathfrak{n}$ is the root node, then the claim is immediate. Otherwise, $\mathfrak{n}$ is the child of some node $\mathfrak{n}'$. Assume that the claim already holds for $\mathfrak{n}'$; that is,

$$\mathsf{cache}(\mathfrak{n}') \subseteq \mathscr{G} \cup \mathscr{M} \tag{B.4.19}$$

$$\mathsf{cand}_\Delta(\mathfrak{n}') \subseteq \mathscr{M} \tag{B.4.20}$$

Case distinction on the form of $\mathfrak{n}'$.

- *Case* $\mathfrak{n}' = \texttt{entailsAux}(\Delta', \mathscr{G}', \beta', \mathcal{P}')$: It is obvious that the type environment Δ remains constant throughout the whole call tree; hence, we may safely assume that $\Delta' = \Delta$.

  Case distinction on the line number of the call site corresponding to $\mathfrak{n}$.

  - *Case* line 4: Then $\mathsf{cache}(\mathfrak{n}) = \mathsf{cache}(\mathfrak{n}')$ and $\mathsf{cand}_\Delta(\mathfrak{n}) = \mathsf{cand}_\Delta(\mathfrak{n}')$, so the claim is immediate.
  - *Case* line 25: We have

    $$\begin{gathered}\mathcal{P}' = \overline{T}^m\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\\ \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}^n\ \ldots\\ \texttt{lift}(\Delta, \beta', I, \overline{T}, \overline{[U/X]}\overline{N})\\ \overline{V} = \overline{[U/X]}\overline{V'}\\ (\overline{[U/X]}\overline{N})\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \notin \mathscr{G}'\\ \mathscr{G}_0 = \mathscr{G}' \cup \{\overline{[U/X]}\overline{N}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\}\end{gathered}$$

    and

    $$\mathfrak{n} = \texttt{entailsAux}(\Delta, \mathscr{G}_0, \texttt{false}, \overline{[U/X]}P_i)$$

    for some $i \in [n]$.

    From $\texttt{lift}(\Delta, \beta', I, \overline{T}, \overline{[U/X]}\overline{N})$ we get with Lemma B.4.11 that $\Delta \vdash_{\mathrm{q}}{}' T_j \leq \overline{[U/X]}N_j$ for all $j \in [m]$, hence

    $$\overline{[U/X]}N_j \in \mathsf{closure}_\Delta(\overline{T}) \tag{B.4.21}$$

    for all $j \in [m]$ by rule CLOSURE-UP. With (B.4.21) and rule CAND-CLOSURE we get

    $$(\overline{[U/X]}\overline{N})\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \in \mathsf{cand}_\Delta(\overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>})$$

By (B.4.20) we have $\mathsf{cand}_\Delta(\overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}) \subseteq \mathscr{M}$, so we get

$$([\overline{U/X}]\overline{N})\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \in \mathscr{M}$$

Hence

$$\begin{aligned}\mathsf{cache}(\mathfrak{n}) &= \mathscr{G}' \cup \{([\overline{U/X}]\overline{N})\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\}\\ &= \mathsf{cache}(\mathfrak{n}') \cup \{([\overline{U/X}]\overline{N})\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\}\\ &\overset{(B.4.19)}{\subseteq} \mathscr{G} \cup \mathscr{M} \cup \{([\overline{U/X}]\overline{N})\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\}\\ &= \mathscr{G} \cup \mathscr{M}\end{aligned}$$

We still need to show $\mathsf{cand}_\Delta(\mathfrak{n}) \subseteq \mathscr{M}$. By criterion WF-IMPL-2, we have $\overline{X} \subseteq \mathsf{ftv}(\overline{N})$, so for each X_k there exists some N_j such that $X_k \in \mathsf{ftv}(N_j)$. Thus, U_k is a subterm of $[\overline{U/X}]N_j$. With (B.4.21) and possibly repeated applications of rules CLOSURE-DECOMP-CLASS and CLOSURE-DECOMP-IFACE, we get $U_k \in \mathsf{closure}_\Delta(\overline{T})$. Thus

$$\overline{U} \subseteq \mathsf{closure}_\Delta(\overline{T})$$

With Lemma B.4.16

$$[\overline{U/X}]P_i \in \mathsf{cand}_\Delta(\overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>})$$

Lemma B.4.15 now yields

$$\mathsf{cand}_\Delta([\overline{U/X}]P_i) \subseteq \mathsf{cand}_\Delta(\overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>})$$

From (B.4.20) we have $\mathsf{cand}_\Delta(\overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}) \subseteq \mathscr{M}$. Moreover, $\mathsf{cand}_\Delta(\mathfrak{n}) = \mathsf{cand}_\Delta([\overline{U/X}]P_i)$, so $\mathsf{cand}_\Delta([\overline{U/X}]P_i) \subseteq \mathscr{M}$.

End case distinction on the line number of the call site corresponding to $\mathfrak{n}$.

- *Case* $\mathfrak{n}' = \texttt{subAux}(\Delta', \mathscr{G}', T', U')$: Again, we may safely assume $\Delta = \Delta'$. The call site corresponding to $\mathfrak{n}$ must be in line 5. We then have

$$\begin{gathered}\mathscr{G}' = \mathscr{G}\\ U' = K \text{ for some } K\\ \mathfrak{n} = \texttt{entailsAux}(\Delta, \mathscr{G}, \texttt{true}, T'\,\textbf{implements}\,K)\end{gathered}$$

We get

$$\mathsf{cache}(\mathfrak{n}) = \mathscr{G} = \mathsf{cache}(\mathfrak{n}') \overset{(B.4.19)}{\subseteq} \mathscr{G} \cup \mathscr{M}$$

and

$$\mathsf{cand}_\Delta(\mathfrak{n}) = \mathsf{cand}_\Delta(T'\,\textbf{implements}\,K) \overset{\text{by rule CAND-EXTENDS}}{=} \mathsf{cand}_\Delta(T'\,\textbf{extends}\,K) = \mathsf{cand}_\Delta(\mathfrak{n}') \overset{(B.4.20)}{\subseteq} \mathscr{M}$$

End case distinction on the form of $\mathfrak{n}'$. □

Definition B.4.20. The *size* of a type T, written $\mathsf{size}(T) \in \mathbb{N}^+$, or constraint $\mathcal{P}$, written $\mathsf{size}(\mathcal{P}) \in \mathbb{N}^+$, is defined as follows:

$$\begin{aligned}
\mathsf{size}(X) &= 1 \\
\mathsf{size}(C\texttt{<}\overline{T}\texttt{>}) &= 1 + \mathsf{size}(\overline{T}) \\
\mathsf{size}(I\texttt{<}\overline{T}\texttt{>}) &= 1 + \mathsf{size}(\overline{T}) \\
\mathsf{size}(\overline{T}\,\textbf{implements}\,K) &= 1 + \mathsf{size}(K) + \mathsf{size}(\overline{T}) \\
\mathsf{size}(T\,\textbf{extends}\,U) &= 1 + \mathsf{size}(T) + \mathsf{size}(U)
\end{aligned}$$

Thereby, the size of a sequence of types $\overline{T}$ is defined as $\mathsf{size}(\overline{T}) = \sum_i \mathsf{size}(T_i)$.

Lemma B.4.21. *Suppose* $\mathsf{closure}_\Delta(\mathscr{T})$ *is finite for every finite* $\mathscr{T}$. *Then* $\mathsf{cand}_\Delta(\mathcal{P})$ *is finite for all* $\mathcal{P}$.

Proof. We show that for all $\mathcal{P}$ there exists a $\delta(\mathcal{P}) \in \mathbb{N}^+$ such that $\mathsf{size}(\mathcal{Q}) \leq \delta(\mathcal{P})$ for all $\mathcal{Q} \in \mathsf{cand}_\Delta(\mathcal{P})$. The original claim then follows immediately because the set of types and constraints of a certain size is finite.

Let $\rho \in \mathbb{N}^+$ be a bound on the size of the constraints in the set $\mathscr{P}$ where

$$\mathscr{P} = \{P_i \mid \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}^{\,n}\ \ldots, i \in [n]\}$$

Let $\vartheta(\mathcal{P}) \in \mathbb{N}^+$ be a bound on the size of the types in $\mathsf{closure}_\Delta(\mathsf{left}(\mathcal{P}))$. Note that $\vartheta(\mathcal{P})$ exists because $\mathsf{closure}_\Delta(\mathsf{left}(\mathcal{P}))$ is finite by the assumption. Define

$$\delta(\mathcal{P}) = \rho \cdot \vartheta(\mathcal{P}) \cdot \mathsf{size}(\mathcal{P})$$

Now suppose $\mathcal{Q} \in \mathsf{cand}_\Delta(\mathcal{P})$.
Case distinction on the last rule in the derivation of $\mathcal{Q} \in \mathsf{cand}_\Delta(\mathcal{P})$.

- *Case* CAND-CLOSURE: Then $\mathcal{P} = \overline{T}\,\textbf{implements}\,K$ and $\mathcal{Q} = \overline{U}\,\textbf{implements}\,K$ with $\overline{U} \subseteq \mathsf{closure}_\Delta(\overline{T})$. Hence, $\mathsf{size}(U_j) \leq \vartheta(\mathcal{P})$ for all j and the following inequality holds:

$$\begin{aligned}
\mathsf{size}(\mathcal{Q}) &= 1 + \mathsf{size}(\overline{U}) + \mathsf{size}(K) \\
&\leq \vartheta(\mathcal{P}) + \mathsf{size}(\overline{T}) \cdot \vartheta(\mathcal{P}) + \mathsf{size}(K) \cdot \vartheta(K) \\
&= \vartheta(\mathcal{P}) \cdot \mathsf{size}(\mathcal{P}) \\
&\leq \vartheta(\mathcal{P}) \cdot \mathsf{size}(\mathcal{P}) \cdot \rho = \delta(\mathcal{P})
\end{aligned}$$

- *Case* CAND-IMPL$_1$: Then

$$\frac{\begin{array}{c}\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots \\ \overline{U} \subseteq \mathsf{closure}_\Delta(\overline{T}) \qquad \overline{U'} \subseteq \mathsf{closure}_\Delta(\overline{T}) \qquad P_i = \overline{W}\,\textbf{implements}\,L\end{array}}{\underbrace{\overline{U}\,\textbf{implements}\,[\overline{U'/X}]L}_{=\mathcal{Q}} \in \mathsf{cand}_\Delta(\underbrace{\overline{T}\,\textbf{implements}\,K}_{=\mathcal{P}})}$$

  We have $\mathsf{size}(U_j) \leq \vartheta(\mathcal{P})$ and $\mathsf{size}(U'_k) \leq \vartheta(\mathcal{P})$ for all j, k. Moreover, $\mathsf{size}(P_i) \leq \rho$. Then the following inequality holds:

$$\begin{aligned}
\mathsf{size}(\mathcal{Q}) &= 1 + \mathsf{size}(\overline{U}) + \mathsf{size}([\overline{U'/X}]L) \\
&\leq \vartheta(\mathcal{P}) + \mathsf{size}(\overline{W}) \cdot \vartheta(\mathcal{P}) + \mathsf{size}(L) \cdot \vartheta(\mathcal{P}) \\
&= \vartheta(\mathcal{P}) \cdot \mathsf{size}(P_i) \\
&\leq \vartheta(\mathcal{P}) \cdot \rho \cdot \mathsf{size}(\mathcal{P}) = \delta(\mathcal{P})
\end{aligned}$$

- *Case* cand-impl$_2$: Analogously to the preceding case.
- *Case* cand-extends: Then $\mathcal{P} = T\,\mathbf{extends}\,K$ and

  $$\mathcal{Q} \in \mathsf{closure}_\Delta(T\,\mathbf{implements}\,K)$$

  Because this derivation cannot end with rule cand-extends, the claim follows with the same argumentation as in one of the three preceding cases.

End case distinction on the last rule in the derivation of $\mathcal{Q} \in \mathsf{cand}_\Delta(\mathcal{P})$. □

Proof of Theorem 3.27. We show for all Δ, $\mathcal{P}$, T, and U that $\mathtt{entails}(\Delta, \mathcal{P})$ and $\mathtt{sub}(\Delta, T, U)$ and $\mathtt{sub'}(\Delta, T, U)$ terminate. By Definition 3.7 and the criteria wf-tenv-1 and wf-tenv-2, we know that Δ is finite and contractive and that $\mathsf{closure}_\Delta(\mathscr{T})$ is finite for every finite $\mathscr{T}$.

$\mathtt{sub'}$ terminates. The weight function from Definition B.4.1 is extended to recursive calls of $\mathtt{sub'}$ in the obvious way:

$$\mathsf{weight}(\mathtt{sub'}(\Delta, T, U)) = \mathsf{weight}_\Delta(T) + \mathsf{weight}_\Delta(U)$$

It is straightforward to verify that for each recursive call of $\mathtt{sub'}$, the weight of the recursive call is strictly smaller than the weight of the original call. Moreover, the algorithms for checking class ($\trianglelefteq_\mathbf{c}$) and interface ($\trianglelefteq_\mathbf{i}$) inheritance terminate because the class and interface hierarchy is acyclic by criterion wf-prog-5. Thus, $\mathtt{sub'}$ terminates.

$\mathtt{entails}$ terminates. To prove that $\mathtt{entails}(\Delta, \mathcal{P})$ terminates, we show for finite $\mathscr{G}$ that both $\mathtt{entailsAux}(\Delta, \mathscr{G}, \beta, \mathcal{P})$ and $\mathtt{subAux}(\Delta, \mathscr{G}, T, U)$ terminate. The claim then follows because $\mathtt{entails}(\Delta, \mathcal{P})$ invokes $\mathtt{entailsAux}$ only with $\mathscr{G} = \emptyset$.

To obtain a contradiction, assume that an invocation of either $\mathtt{entailsAux}(\Delta, \mathscr{G}, \beta, \mathcal{P})$ or $\mathtt{subAux}(\Delta, \mathscr{G}, T, U)$ diverges. It is easy to see that infinitely many calls of $\mathtt{entailsAux}$ or $\mathtt{subAux}$ must cause divergence:

- There are only finitely many choices for R in line 8 because Δ is finite.
- The algorithms for checking the relations $\mathcal{R} \in \mathsf{sup}(\mathcal{R})$, $i \in \mathsf{pol}^+(I)$, $i \in \mathsf{pol}^-(I)$ and $K \trianglelefteq_\mathbf{i} K$ terminate because the interface graph is acyclic (criterion wf-prog-5).
- The function $\mathtt{lift}$ terminates because $\mathtt{sub'}$ terminates as shown in the preceding case.
- The function $\mathtt{unify}_\leq$ terminates by Theorem 3.24.

Hence, there exists a call tree $\mathfrak{t}$ of infinite size. We lead this to a contradiction by defining a measure μ from call tree nodes into $\mathbb{N} \times \mathbb{N}$ that strictly decreases (with respect to the usual lexicographic ordering on pairs) when moving from a node to any of its children.

Suppose the root node of $\mathfrak{t}$ is $\mathtt{entailsAux}(\Delta, \mathscr{G}, \beta, \mathcal{P})$ (or $\mathtt{subAux}(\Delta, \mathscr{G}, T, U)$) and define $\mathscr{M} = \mathsf{cand}_\Delta(\mathcal{P})$ (or $\mathscr{M} = \mathsf{cand}_\Delta(T\,\mathbf{extends}\,U)$). We have the assumption that $\mathsf{closure}_\Delta(\mathscr{T})$ is finite for every finite $\mathscr{T}$, so $\mathscr{M}$ is finite by Lemma B.4.21. Because $\mathscr{G}$ is also finite, we now may define

$$\delta = |\mathscr{G}| + |\mathscr{M}| \in \mathbb{N}$$

($|\cdot|$ denotes set *cardinality*.) We have by Lemma B.4.19 that $\mathsf{cache}(\mathfrak{n}) \subseteq \mathscr{G} \cup \mathscr{M}$ for all nodes $\mathfrak{n}$ in $\mathfrak{t}$. Hence, $(\delta - |\mathsf{cache}(\mathfrak{n})|, i) \in \mathbb{N} \times \mathbb{N}$ for all $i \in \mathbb{N}$ and all nodes $\mathfrak{n}$ in $\mathfrak{t}$. We now define the measure μ on nodes in $\mathfrak{t}$ as follows:

$$\begin{aligned}
\mu(\mathtt{entailsAux}(\Delta', \mathscr{G}', \beta', \overline{T}\,\mathbf{implements}\,K)) &= (\delta - |\mathscr{G}'|, 0) && \in \mathbb{N} \times \mathbb{N} \\
\mu(\mathtt{entailsAux}(\Delta', \mathscr{G}', \beta', T\,\mathbf{extends}\,K)) &= (\delta - |\mathscr{G}'|, 2) && \in \mathbb{N} \times \mathbb{N} \\
\mu(\mathtt{subAux}(\Delta', \mathscr{G}', T, U)) &= (\delta - |\mathscr{G}'|, 1) && \in \mathbb{N} \times \mathbb{N}
\end{aligned}$$

We now show that this measure strictly decreases when moving from a node to its children. Assume $\mathfrak{n}$ is a node in $\mathfrak{t}$ with children $\mathfrak{n}_1, \ldots, \mathfrak{n}_n$ and suppose $i \in [n]$.

Case distinction on the line number of the call site corresponding the $\mathfrak{n}_i$.

- *Case* line 4: We have $\mathfrak{n} = \texttt{entailsAux}(\Delta', \mathscr{G}', \beta', T' \, \textbf{extends} \, U')$, $n = 1$, and $\mathfrak{n}_1 = \texttt{subAux}(\Delta', \mathscr{G}', T', U')$. Hence,

$$\mu(\mathfrak{n}_1) = (\delta - |\mathscr{G}'|, 1) < (\delta - |\mathscr{G}'|, 2) = \mu(\mathfrak{n})$$

- *Case* line 25: We have

$$\begin{gathered}\mathfrak{n} = \texttt{entailsAux}(\Delta', \mathscr{G}', \beta', \overline{T} \, \textbf{implements} \, I\texttt{<}\overline{V}\texttt{>}) \\ \mathscr{G}_0 = \mathscr{G}' \cup \{(\varphi \overline{N}) \, \textbf{implements} \, I\texttt{<}\overline{V}\texttt{>}\} \\ (\varphi \overline{N}) \, \textbf{implements} \, I\texttt{<}\overline{V}\texttt{>} \notin \mathscr{G}' \\ \mathfrak{n}_i = \texttt{entailsAux}(\Delta', \mathscr{G}_0, \texttt{false}, \varphi P_i)\end{gathered}$$

  Thus, $|\mathscr{G}_0| = |\mathscr{G}'| + 1$. Hence,

$$\mu(\mathfrak{n}_i) = (\delta - |\mathscr{G}_0|, j) = (\delta - |\mathscr{G}'| - 1, j) < (\delta - |\mathscr{G}'|, 0) = \mu(\mathfrak{n})$$

  for some $j \in \{0, 1, 2\}$.

- *Case* line 5: We have $\mathfrak{n} = \texttt{subAux}(\Delta', \mathscr{G}', T', K)$, $n = 1$, and

$$\mathfrak{n}_1 = \texttt{entailsAux}(\Delta', \mathscr{G}', \texttt{true}, T' \, \textbf{implements} \, K)$$

  Thus

$$\mu(\mathfrak{n}_1) = (\delta - |\mathscr{G}'|, 0) < (\delta - |\mathscr{G}'|, 1) = \mu(\mathfrak{n})$$

End case distinction on the line number of the call site corresponding the $\mathfrak{n}_i$.

sub terminates. In the preceding case, we showed that $\texttt{entailsAux}(\Delta, \mathscr{G}, \beta, \mathcal{P})$ terminates and that $\texttt{subAux}(\Delta, \mathscr{G}, T, U)$ terminates (for finite $\mathscr{G}$). The claim follows immediately because $\texttt{sub}(\Delta, T, U)$ invokes subAux only with $\mathscr{G} = \emptyset$. □

B.5 Deciding Expression Typing

This section proves Theorem 3.28 (soundness of entailment for constraints with optional types), Theorem 3.29 (completeness of entailment for constraints with optional types), Theorem 3.31 (soundness of algorithmic method typing), Theorem 3.32 (completeness of algorithmic method typing), Theorem 3.35 (soundness of algorithmic expression typing), Theorem 3.36 (completeness of algorithmic expression typing), and Theorem 3.37 (termination of algorithmic expression typing).

B.5.1 Proof of Theorem 3.28

Theorem 3.28 states that entailment for constraints with optional types is sound with respect to algorithmic entailment for ordinary constraints.

Proof of Theorem 3.28. We first show that

$$\Delta; \mathscr{G}; \beta \vdash^{?}_{\mathrm{a}} \overline{T^{?}}^{n} \uparrow \overline{U}^{n} \mathrel{-\!\triangleright} \overline{V}^{n} \;\; \textit{implies} \;\; \Delta; \mathscr{G}; \beta \vdash_{\mathrm{a}} \overline{V}^{n} \uparrow \overline{U}^{n} \tag{B.5.1}$$

From $\Delta; \mathscr{G}; \beta \vdash^{?}_{\mathrm{a}} \overline{T^{?}}^{n} \uparrow \overline{U}^{n} \mathrel{-\!\triangleright} \overline{V}^{n}$ we get

$$(\forall i)\ T^{?}_{i} = \mathsf{nil} \text{ or } \Delta \vdash_{\mathrm{q}}{}' T^{?}_{i} \leq U_{i}$$
$$\beta \text{ or } \big((\forall i) \text{ if } T^{?}_{i} \neq U_{i} \text{ and } T^{?}_{i} \neq \mathsf{nil} \text{ then } i \in \mathsf{pol}^{-}(I)\big)$$
$$(\forall i)\ \text{ if } T^{?}_{i} = \mathsf{nil} \text{ then } V_{i} = U_{i} \text{ else } V_{i} = T^{?}_{i}$$

Hence, $(\forall i)\ \Delta \vdash_{\mathrm{q}}{}' V_{i} \leq U_{i}$ and $(\beta$ or (if $V_{i} \neq U_{i}$ then $i \in \mathsf{pol}^{-}(I)))$. We then have by rule ENT-ALG-LIFT that $\Delta; \mathscr{G}; \beta \vdash_{\mathrm{a}} \overline{V}^{n} \uparrow \overline{U}^{n}$.

We now prove that $\mathcal{D} :: \Delta; \mathscr{G}; \beta \Vdash^{?}_{\mathrm{a}} \overline{T^{?}}\, \textbf{implements}\, I\texttt{<}\overline{W^{?}}\texttt{>} \mathrel{-\!\triangleright} \mathcal{R}$ implies $\Delta; \mathscr{G}; \beta \Vdash_{\mathrm{a}} \mathcal{R}$ by induction on $\mathcal{D}$. The claim then follows with rule ENT-ALG-MAIN.
Case distinction on the last rule used in $\mathcal{D}$.

- *Case* rule ENT-NIL-ALG-ENV: Then

  $$R \in \Delta$$
  $$\overline{G}\, \textbf{implements}\, I\texttt{<}\overline{W}\texttt{>} \in \mathsf{sup}(R)$$
  $$\Delta; \beta; I \vdash^{?}_{\mathrm{a}} \overline{T^{?}} \uparrow \overline{G} \mathrel{-\!\triangleright} \overline{T}$$
  $$(\forall i)\ W^{?}_{i} \sim W_{i}$$

  with $\mathcal{R} = \overline{T}\, \textbf{implements}\, I\texttt{<}\overline{W}\texttt{>}$. We then have by (B.5.1) that $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{G}$. The claim now follows with rule ENT-ALG-ENV.

- *Case* rule ENT-NIL-ALG-IFACE$_1$: Then

  $$\Delta; \beta; I \vdash_{\mathrm{a}} T \uparrow I\texttt{<}\overline{W}\texttt{>}$$
  $$1 \in \mathsf{pol}^{+}(I)$$
  $$\mathsf{non\text{-}static}(I)$$
  $$(\forall i)\ W^{?}_{i} \sim W_{i}$$

  with $\overline{T^{?}} = T$ and $\mathcal{R} = T\, \textbf{implements}\, I\texttt{<}\overline{W}\texttt{>}$. The claim follows from rule ENT-ALG-IFACE$_1$.

- *Case* rule ENT-NIL-ALG-IFACE$_2$: Then

  $$1 \in \mathsf{pol}^{+}(J)$$
  $$\mathsf{non\text{-}static}(J)$$
  $$J\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{W}\texttt{>}$$
  $$(\forall i)\ W^{?}_{i} \sim W_{i}$$

  with $\overline{T^{?}} = J\texttt{<}\overline{V}\texttt{>}$ and $\mathcal{R} = J\texttt{<}\overline{V}\texttt{>}\, \textbf{implements}\, I\texttt{<}\overline{W}\texttt{>}$. The claim follows by applying rule ENT-ALG-IFACE$_2$.

- *Case* rule ENT-NIL-ALG-IMPL: Then

  $$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots$$
  $$\Delta; \beta; I \vdash^{?}_{\mathrm{a}} \overline{T^{?}} \uparrow [\overline{U/X}]\overline{N} \mathrel{-\!\triangleright} \overline{T}$$
  $$(\forall i)\ W^{?}_{i} \sim [\overline{U/X}]V_{i}$$
  $$[\overline{U/X}]\overline{N}\, \textbf{implements}\, I\texttt{<}[\overline{U/X}]\overline{V}\texttt{>} \notin \mathscr{G}$$
  $$\Delta; \mathscr{G} \cup \{[\overline{U/X}]\overline{N}\, \textbf{implements}\, I\texttt{<}[\overline{U/X}]\overline{V}\texttt{>}\}; \texttt{false} \Vdash_{\mathrm{a}} [\overline{U/X}]\overline{P}$$

with $\mathcal{R} = \overline{T}\,\textbf{implements}\,I{<}\overline{[U/X]}\overline{V}{>}$. From $\Delta; \beta; I \vdash^{?}_{\mathrm{a}} \overline{T^?} \uparrow \overline{[U/X]}\overline{N} \rightarrowtriangle \overline{T}$ we get with (B.5.1) that $\Delta; \beta; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{[U/X]}\overline{N}$. The claim now follows with rule ENT-ALG-IMPL.

End case distinction on the last rule used in $\mathcal{D}$. □

B.5.2 Proof of Theorem 3.29

Theorem 3.29 states that entailment for constraints with optional types is complete with respect to algorithmic entailment for ordinary constraints.

Lemma B.5.1. *If I is a single-headed interface, then $1 \in \mathsf{disp}(I)$.*

Proof. The proof is by induction on the depth of I (see Definition B.2.6). □

Proof of Theorem 3.29. We first show:

$$\begin{gathered} \textit{If } \overline{T^?}^n \sim \overline{T}^n \textit{ and } \Delta; \texttt{false}; I \vdash_{\mathrm{a}} \overline{T}^n \uparrow \overline{U}^n \textit{ then } \Delta; \texttt{false}; I \vdash_{\mathrm{a}} \overline{T^?}^n \uparrow \overline{U}^n \rightarrowtriangle \overline{V}^n \\ \textit{such that } \Delta \vdash_{\mathrm{q}}{}' T_i \le V_i \textit{ for all } i \textit{ and} \\ V_i = T_i \textit{ for those } i \textit{ with } T^?_i \ne \mathsf{nil} \textit{ or } i \notin \mathsf{pol}^-(I). \end{gathered} \tag{B.5.2}$$

Assume $\Delta; \mathscr{G}; \texttt{false} \vdash_{\mathrm{a}} \overline{T}^n \uparrow \overline{U}^n$ and $\overline{T^?}^n \sim \overline{T}^n$. By inverting rule ENT-ALG-LIFT, we get $\Delta \vdash_{\mathrm{q}}{}'$ $T_i \le U_i$ for all i and $T_i = U_i$ for $i \notin \mathsf{pol}^-(I)$. By rule ENT-NIL-ALG-LIFT, we have $\Delta; \mathscr{G}; \texttt{false} \vdash_{\mathrm{a}}$ $\overline{T^?}^n \uparrow \overline{U}^n \rightarrowtriangle \overline{V}^n$ for some $\overline{V}$. Now let $i \in [n]$.

- If $T^?_i = \mathsf{nil}$ then $V_i = U_i$. Hence, $\Delta \vdash_{\mathrm{a}} T_i \le V_i$. If additionally $i \notin \mathsf{pol}^-(I)$, then $V_i = U_i = T_i$.
- If $T^?_i \ne \mathsf{nil}$ then $V_i = T^?_i$. With $T^?_i \sim T_i$ then $V_i = T_i$.

This finishes the proof of (B.5.2).

We now show that $\mathcal{D} :: \Delta; \mathscr{G}; \texttt{false} \Vdash_{\mathrm{a}} \overline{T}\,\textbf{implements}\,I{<}\overline{V}{>}$ and $\overline{T^?}\,\overline{V^?} \sim \overline{T}\,\overline{V}$ and $T^?_i \ne \mathsf{nil}$ for $i \in \mathsf{disp}(i)$ imply

$$\Delta; \mathscr{G}; \texttt{false} \Vdash^{?}_{\mathrm{a}} \overline{T^?}\,\textbf{implements}\,I{<}\overline{V^?}{>} \rightarrowtriangle \overline{U}\,\textbf{implements}\,I{<}\overline{V}{>}$$

such that $\Delta \vdash_{\mathrm{q}}{}' T_i \le U_i$ for all i and $U_i = T_i$ for those i with $T^?_i \ne \mathsf{nil}$ or $i \notin \mathsf{pol}^-(I)$. The original claim then follows with rule ENT-NIL-ALG-MAIN.
Case distinction on the last rule used in $\mathcal{D}$.

- *Case* rule ENT-ALG-ENV: Then

  $$\begin{gathered} R \in \Delta \\ \overline{G}\,\textbf{implements}\,I{<}\overline{V}{>} \in \mathsf{sup}(R) \\ \Delta; \texttt{false}; I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{G} \end{gathered}$$

  By (B.5.2) we have $\Delta; \mathscr{G}; \beta \vdash^{?}_{\mathrm{a}} \overline{T^?} \uparrow \overline{G} \rightarrowtriangle \overline{U}$ such that $\overline{U}$ has the desired properties. The claim now follows by rule ENT-NIL-ALG-ENV.

- *Case* rule ENT-ALG-IFACE$_1$: Then

  $$\begin{gathered} \Delta; \texttt{false}; I \vdash_{\mathrm{a}} T \uparrow I{<}\overline{V}{>} \\ 1 \in \mathsf{pol}^+(I) \\ \mathsf{non\text{-}static}(I) \end{gathered}$$

  with $\overline{T} = T$. By Lemma B.5.1, $1 \in \mathsf{disp}(I)$. Hence, $T^?_1 = T_1 = T$. We get with (B.5.2) that $\Delta; \mathscr{G}; \beta \vdash^{?}_{\mathrm{a}} T^? \uparrow I{<}\overline{V}{>} \rightarrowtriangle T$. The claim now follows by rule ENT-NIL-ALG-IFACE$_1$.

- *Case* rule ENT-ALG-IFACE$_2$: Then

  $$\begin{gathered} 1 \in \mathsf{pol}^{+}(J) \\ \mathsf{non\text{-}static}(J) \\ J\texttt{<}\overline{W}\texttt{>} \trianglelefteq_{\mathsf{i}} I\texttt{<}\overline{V}\texttt{>} \end{gathered}$$

  with $\overline{T} = J\texttt{<}\overline{W}\texttt{>}$. By Lemma B.5.1, $1 \in \mathsf{disp}(I)$. Hence, $T_1^{?} = T_1 = J\texttt{<}\overline{W}\texttt{>}$. The claim now follows by rule ENT-NIL-ALG-IFACE$_2$.

- *Case* rule ENT-ALG-IMPL: Then

  $$\begin{gathered} \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots \\ \Delta;\beta;I \vdash_{\mathrm{a}} \overline{T} \uparrow [\overline{W/X}]\overline{N} \\ \overline{V} = [\overline{W/X}]\overline{V'} \\ [\overline{W/X}]\overline{N}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>} \notin \mathscr{G} \\ \Delta;\mathscr{G} \cup \{[\overline{W/X}]\overline{N}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}\};\texttt{false} \Vdash_{\mathrm{a}} [\overline{W/X}]\overline{P} \end{gathered}$$

  By (B.5.2), we have $\Delta;\beta;I \vdash_{\mathrm{a}}^{?} \overline{T^{?}} \uparrow [\overline{W/X}]\overline{N} \mathrel{-\!\triangleright} \overline{U}$ such that $\overline{U}$ has the desired properties. The claim now follows by rule ENT-NIL-ALG-IMPL.

End case distinction on the last rule used in $\mathcal{D}$. □

B.5.3 Proof of Theorem 3.31

Theorem 3.31 states that algorithmic method typing in Figure 3.29 is sound with respect to its declarative specification in Figure 3.8. All proofs in this section apply the equivalences and implications of the following corollary implicitly.

Corollary B.5.2.

$$\begin{array}{rcll}
\Delta \vdash T \leq U & \textit{iff} & \Delta \vdash_{\mathrm{q}} T \leq U & \textit{(Theorem 3.12, Theorem 3.11)} \\
\Delta \Vdash \mathcal{P} & \textit{iff} & \Delta \Vdash_{\mathrm{q}} \mathcal{P} & \textit{(Theorem 3.12, Theorem 3.11)} \\
\Delta \vdash_{\mathrm{q}} T \leq U & \textit{iff} & \Delta \vdash_{\mathrm{a}} T \leq U & \textit{(Theorem 3.26, Theorem 3.25)} \\
\Delta \Vdash_{\mathrm{q}} \mathcal{P} & \textit{iff} & \Delta \Vdash_{\mathrm{a}} \mathcal{P} & \textit{(Theorem 3.26, Theorem 3.25)} \\
\Delta \vdash_{\mathrm{q}} T \leq G & \textit{implies} & \Delta \vdash_{\mathrm{q}}' T \leq G & \textit{(Lemma B.1.14)} \\
\Delta \vdash_{\mathrm{q}}' T \leq U & \textit{implies} & \Delta \vdash_{\mathrm{q}} T \leq U & \textit{(Rule } \textsc{sub-q-alg-kernel}\textit{)} \\
\Delta \Vdash_{\mathrm{q}}' \mathcal{P} & \textit{implies} & \Delta \Vdash_{\mathrm{q}} \mathcal{P} & \textit{(Lemma B.1.17)} \\
N \trianglelefteq_{\mathsf{c}} M & \textit{iff} & \Delta \vdash_{\mathrm{q}}' N \leq M & \textit{(rule } \textsc{sub-q-alg-class} \textit{ and Lemma B.1.10)} \\
K \trianglelefteq_{\mathsf{i}} L & \textit{iff} & \Delta \vdash_{\mathrm{q}}' K \leq L & \textit{(rule } \textsc{sub-q-alg-iface} \textit{ and Lemma B.1.10)}
\end{array}$$

Lemma B.5.3. *If* $\mathsf{bound}_{\Delta}(T) = N$ *then* $\Delta \vdash T \leq N$.

Proof. Obvious by inspecting rule BOUND. □

Lemma B.5.4. *If* $\Delta \Vdash_{\mathrm{a}}^{?} \overline{T^{?}}\,\textbf{implements}\,I\texttt{<}\overline{U^{?}}\texttt{>} \mathrel{-\!\triangleright} \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{U}\texttt{>}$ *and* $T_i^{?} \neq \mathsf{nil}$ *then* $T_i^{?} = T_i$.

Proof. Follows by inspecting the rules in Figure 3.27. □

Proof of Theorem 3.31. Case distinction on the form of m.

- *Case* $m = m^{\mathrm{c}}$: Then

  $$\begin{gathered}\mathsf{bound}_{\Delta}(T) = N \\ \mathcal{D} :: \mathsf{a\text{-}mtype}^{\mathrm{c}}(m, N) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \ \textbf{where}\ \overline{\mathcal{P}}\end{gathered}$$

  A straightforward induction on the derivation $\mathcal{D}$ shows that there exists N' such that

  $$\begin{gathered}\mathsf{mtype}(m, N') = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \ \textbf{where}\ \overline{\mathcal{P}} \\ N \trianglelefteq_{\mathsf{c}} N'\end{gathered}$$

  With $\mathsf{bound}_{\Delta}(T) = N$ and Lemma B.5.3 we have $\Delta \vdash T \leq N$. Thus, by transitivity of subtyping,

  $$\Delta \vdash T \leq N'$$

  We finish this case by setting $T' = N'$.

- *Case* $m = m^{\mathrm{i}}$: Then

  $$\begin{gathered}\textbf{interface}\ I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}^{l}\ \textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{P}\,\{\ldots\ \overline{rcsig}\,\} \\ rcsig_j = \textbf{receiver}\,\{\overline{m : msig}\} \\ msig_k = \texttt{<}\overline{X}\texttt{>}\,\overline{U'\,x} \to U'\ \textbf{where}\ \overline{Q} \\ (\forall i \in [l], i \neq j)\ \mathsf{sresolve}_{\Delta;Z_i}(\overline{U'}, \overline{T}) = \mathscr{V}_i \\ \mathsf{sresolve}_{\Delta;Z_j}(Z_j\,\overline{U'}, T\,\overline{T}) = \mathscr{V}_j \\ p^{?} = (\text{if } U' = Z_i \text{ for some } i \in [l] \text{ then } i \text{ else } \mathsf{nil}) \\ \overline{W}\,\textbf{implements}\,I\texttt{<}\overline{W'}\texttt{>} = \mathsf{pick\text{-}constr}_{\Delta}^{p^{?}}\,\mathscr{M}\end{gathered}$$

  $$\begin{aligned}\mathscr{M} = \{\overline{V}\,\textbf{implements}\,I\texttt{<}\overline{V''}\texttt{>} \mid (\forall i \in [l])\ & \text{if } \mathscr{V}_i = \emptyset \text{ then } V_i^{?} = \mathsf{nil} \\ & \text{else define } V_i^{?} \text{ such that} \\ & \quad \Delta \vdash_{\mathrm{q}}{}' V_i' \leq V_i^{?} \text{ for } V_i' \in \mathscr{V}_i, \\ \Delta \Vdash_{\mathrm{a}}^{?} \overline{V^{?}}\,\textbf{implements}\,I\texttt{<}\mathsf{nil}\texttt{>} & \rightarrowtriangle \overline{V}\,\textbf{implements}\,I\texttt{<}\overline{V''}\texttt{>}\}\end{aligned}$$

  and

  $$\texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}} = [\overline{W/Z}, \overline{W'/Z'}](\texttt{<}\overline{X}\texttt{>}\,\overline{U'\,x} \to U'\ \textbf{where}\ \overline{Q})$$

  Obviously, $\mathscr{V}_j \neq \emptyset$. With Lemma B.5.16 and the definition of $\mathsf{sresolve}$ we get

  $$\Delta \vdash_{\mathrm{q}}{}' T \leq V_j' \text{ for all } V_j' \in \mathscr{V}_j$$

  With Lemma B.5.4, we know that for all $\overline{V}\,\textbf{implements}\,I\texttt{<}\overline{V''}\texttt{>} \in \mathscr{M}$ there exists some $V_j' \in \mathscr{V}_j$ such that

  $$\Delta \vdash_{\mathrm{q}}{}' V_j' \leq V_j$$

  With rule SUB-TRANS we thus have

  $$\Delta \vdash T \leq W_j$$

  By Theorem 3.28 we get

  $$\Delta \Vdash \overline{W}\,\textbf{implements}\,I\texttt{<}\overline{W'}\texttt{>}$$

  By rule MTYPE-IFACE we now have

  $$\mathsf{mtype}_{\Delta}(m, W_j) = [\overline{W/Z}, \overline{W'/Z'}]msig_k = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U\ \textbf{where}\ \overline{\mathcal{P}}$$

  Define $T' = W_j$ to finish this case.

End case distinction on the form of m. □

B.5.4 Proof of Theorem 3.32

Theorem 3.32 states that algorithmic method typing in Figure 3.29 is complete with respect to its declarative specification in Figure 3.8. All proofs in this section apply the equivalences and implications of Corollary B.5.2 implicitly.

Lemma B.5.5 (Transitivity of $\in^+$ and $\in^*$).

(*i*) *If* $X\,\mathbf{extends}\,Y \in^+ \Delta$ *and* $Y\,\mathbf{extends}\,T \in^+ \Delta$ *then* $X\,\mathbf{extends}\,T \in^+ \Delta$.

(*ii*) *If* $X\,\mathbf{extends}\,Y \in^* \Delta$ *and* $Y\,\mathbf{extends}\,T \in^* \Delta$ *then* $X\,\mathbf{extends}\,T \in^* \Delta$.

Proof. Claim (i) is proved by induction on the derivation of $X\,\mathbf{extends}\,Y \in^+ \Delta$. Claim (ii) follows by claim (i) and a case distinction on the last rule used in the derivation of $X\,\mathbf{extends}\,Y \in^* \Delta$. □

Lemma B.5.6. *If* $X\,\mathbf{extends}\,T \in^+ \Delta$ *or* $X\,\mathbf{extends}\,T \in^* \Delta$ *then* $\Delta \vdash_{\mathrm{q}}{}' X \leq T$.

Proof. If $X\,\mathbf{extends}\,T \in^+ \Delta$ then the claim follows by a straightforward induction on the derivation given. The other case is now trivial. Note that we use Lemma B.1.6. □

Lemma B.5.7. *If* $\Delta \vdash T \leq G_1$ *and* $\Delta \vdash T \leq G_2$ *then* $\Delta \vdash G_1 \leq G_2$ *or* $\Delta \vdash G_2 \leq G_1$.

Proof. We first note that $\Delta \vdash T \leq G_i$ implies $\Delta \vdash_{\mathrm{q}}{}' T \leq G_i$ by Corollary B.5.2. If $G_1 = \mathit{Object}$ or $G_2 = \mathit{Object}$, then the claim is obvious. Thus, assume $G_1 \neq \mathit{Object}$ and $G_2 \neq \mathit{Object}$.
Case distinction on the form of T.

- *Case* $T = X$ for some X: If $G_1 = X$ or $G_2 = X$ then the claim is obvious. Now assume $G_1 \neq X$ and $G_2 \neq X$. By Lemma B.1.10 we have that

$$X\,\mathbf{extends}\,G_i \in^+ \Delta \quad (i = 1, 2) \tag{B.5.3}$$

  Define $\mathsf{level} : \mathit{TvarName} \to \mathbb{N}$ as follows. Let $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ be a directed graph with

$$\begin{aligned}\mathscr{V} &= \{X \in \mathit{TvarName} \mid X\,\mathbf{extends}\,T \in \Delta \text{ or } Y\,\mathbf{extends}\,X \in \Delta\}\\ \mathscr{E} &= \{(X, Y) \mid Y\,\mathbf{extends}\,X \in \Delta\}\end{aligned}$$

  Δ is contractive by criterion WF-TENV-1, so $\mathscr{G}$ is acyclic. Hence, there exists a topological ordering $X_0, X_1, \ldots, X_n$ on $\mathscr{V}$ such that $(X_i, X_j) \in \mathscr{E}$ implies $i < j$. Then

$$\mathsf{level}(X) = \begin{cases} i & \text{if } X \in \mathscr{V} \text{ and } X = X_i \\ 0 & \text{if } X \notin \mathscr{V} \end{cases}$$

  We have that

$$X\,\mathbf{extends}\,Y \in \Delta \text{ implies } \mathsf{level}(X) > \mathsf{level}(Y)$$

  We now show that $X\,\mathbf{extends}\,G_i \in^+ \Delta$ for $i = 1, 2$ implies $\Delta \vdash_{\mathrm{q}}{}' G_1 \leq G_2$ or $\Delta \vdash_{\mathrm{q}}{}' G_2 \leq G_1$ by induction on $\mathsf{level}(X)$. Together with (B.5.3), this finishes the case "$T = X$".

  - $\mathsf{level}(X) = 0$. Assume $X\,\mathbf{extends}\,Y \in \Delta$. Then $0 = \mathsf{level}(X) > \mathsf{level}(Y)$ which is impossible because $\mathsf{level}(Y) \in \mathbb{N}$.

    Hence, $G_i = N_i$ for some N_i and $X\,\mathbf{extends}\,G_i \in \Delta$ (for $i = 1, 2$). The claim now follows with criterion WF-TENV-3.

  - $\mathsf{level}(X) = n > 0$ and the claim holds for $n' < n$. We proceed by case distinction on the pair of last rules in the derivations of $X\,\mathbf{extends}\,G_i \in^+ \Delta$

    Case distinction on the pair of last rules.

- *Case* IN-TRANS-BASE / IN-TRANS-BASE: The claim follows with well-formedness criterion WF-TENV-3.
- *Case* IN-TRANS-STEP / IN-TRANS-BASE: Then

$$\begin{gathered} X\,\mathbf{extends}\,Y \in \Delta \\ Y\,\mathbf{extends}\,G_1 \in^{+} \Delta \qquad \text{(B.5.4)} \\ X\,\mathbf{extends}\,G_2 \in \Delta \end{gathered}$$

By criterion WF-TENV-3 either $\Delta \vdash Y \leq G_2$ or $\Delta \vdash G_2 \leq Y$. By Corollary B.5.2 either $\Delta \vdash_{\mathrm{q}}{}' \; Y \leq G_2$ or $\Delta \vdash_{\mathrm{q}}{}' \; G_2 \leq Y$.

  - Suppose $\Delta \vdash_{\mathrm{q}}{}' \; Y \leq G_2$. If $Y = G_2$ then $\Delta \vdash G_2 \leq G_1$ by (B.5.4) and Lemma B.5.6. If $Y \neq G_2$ then $Y\,\mathbf{extends}\,G_2 \in^{+} \Delta$ by Lemma B.1.10. Because $\mathsf{level}(Y) < \mathsf{level}(X)$ we can use the I.H. on (B.5.4) and get the desired result.
  - Suppose $\Delta \vdash_{\mathrm{q}}{}' \; G_2 \leq Y$. By Lemma B.1.10, $G_2 = Z$ for some Z with either $Y = Z$ or $Z\,\mathbf{extends}\,Y \in^{+} \Delta$. If $Y = Z = G_2$ then $\Delta \vdash G_2 \leq G_1$ by (B.5.4) and Lemma B.5.6. Otherwise, $Z\,\mathbf{extends}\,G_1 \in^{+} \Delta$ by (B.5.4) and Lemma B.5.5, so $\Delta \vdash G_2 \leq G_1$ by Lemma B.5.6.

- *Case* IN-TRANS-BASE / IN-TRANS-STEP: Analogously to the preceding case.
- *Case* IN-TRANS-STEP / IN-TRANS-STEP: Then

$$\begin{gathered} X\,\mathbf{extends}\,Y_1 \in \Delta \\ Y_1\,\mathbf{extends}\,G_1 \in^{+} \Delta \qquad \text{(B.5.5)} \\ X\,\mathbf{extends}\,Y_2 \in \Delta \\ Y_2\,\mathbf{extends}\,G_2 \in^{+} \Delta \qquad \text{(B.5.6)} \end{gathered}$$

By criterion WF-TENV-3 either $\Delta \vdash Y_1 \leq Y_2$ or $\Delta \vdash Y_2 \leq Y_1$. We now consider the case $\Delta \vdash Y_1 \leq Y_2$, the proof for the other case is very similar. From $\Delta \vdash Y_1 \leq Y_2$ we get $\Delta \vdash_{\mathrm{q}}{}' \; Y_1 \leq Y_2$ by Corollary B.5.2. With Lemma B.1.10 either $Y_1 = Y_2$ or $Y_1\,\mathbf{extends}\,Y_2 \in^{+} \Delta$. In the following, note that $\mathsf{level}(Y_i) < \mathsf{level}(X)$ for $i = 1, 2$.

  - If $Y_1 = Y_2$ then the claim follows by applying the I.H. to (B.5.5) and (B.5.6).
  - If $Y_1\,\mathbf{extends}\,Y_2 \in^{+} \Delta$, then we get by (B.5.5) and the I.H. that either $\Delta \vdash Y_2 \leq G_1$ or $\Delta \vdash G_1 \leq Y_2$. In the latter case, we have with $Y_2\,\mathbf{extends}\,G_2 \in^{+} \Delta$, Lemma B.5.6, and transitivity that $\Delta \vdash G_1 \leq G_2$. If $\Delta \vdash Y_2 \leq G_1$ then $\Delta \vdash_{\mathrm{q}}{}' \; Y_2 \leq G_1$ by Corollary B.5.2. With Lemma B.1.10 either $Y_2 = G_1$ or $Y_2\,\mathbf{extends}\,G_1 \in^{+} \Delta$. In the former case, we get with (B.5.6) and Lemma B.5.6 that $\Delta \vdash G_1 \leq G_2$. In the latter case, the claim follows by applying the I.H. to $Y_2\,\mathbf{extends}\,G_1 \in^{+} \Delta$ and (B.5.6).

End case distinction on the pair of last rules.

- *Case* $T = N$ for some N or $T = K$ for some K: Because $\Delta \vdash_{\mathrm{q}}{}' \; T \leq G_i$ and $G_i \neq \mathit{Object}$ we have with Lemma B.1.10 that $T = N$ and $G_i = N_i$ ($i = 1, 2$). Hence, $N \trianglelefteq_{\mathbf{c}} N_1$ and $N \trianglelefteq_{\mathbf{c}} N_2$. The claim now follows by Lemma B.2.12.

End case distinction on the form of T. □

Lemma B.5.8 (Existence of $\sqcap$). *If $\Delta \vdash T \leq G_i$ for $i = 1, 2$ then there exists H with $\Delta \vdash G_1 \sqcap G_2 = H$.*

Proof. With Lemma B.5.7 we have either $\Delta \vdash G_1 \leq G_2$ or $\Delta \vdash G_2 \leq G_1$. With rule GLB-LEFT or GLB-RIGHT, respectively, we then have $\Delta \vdash G_1 \sqcap G_2 = G_1$ or $\Delta \vdash G_1 \sqcap G_2 = G_2$. □

Lemma B.5.9. *If $\Delta \vdash N_1 \sqcap N_2 = H$ then $\Delta' \vdash N_1 \sqcap N_2 = H$ for any Δ'.*

Proof. From $\Delta \vdash N_1 \sqcap N_2 = H$ we have w.l.o.g. $N_1 \trianglelefteq_{\mathsf{c}} N_2$. Hence, $\Delta' \vdash N_1 \leq N_2$, so the claim holds. □

Lemma B.5.10. *If $\Delta \Vdash \overline{T}\,\mathbf{implements}\,I{<}\overline{U}{>}$ and $\Delta \Vdash \overline{V}\,\mathbf{implements}\,I{<}\overline{W}{>}$ such that for all $i \in \mathsf{disp}(I)$ there exists T_i' with $\Delta \vdash_{\mathsf{q}}{}' T_i' \leq T_i$ and $\Delta \vdash_{\mathsf{q}}{}' T_i' \leq V_i$, then $\overline{U} = \overline{W}$ and $T_j = V_j$ for all $j \notin \mathsf{disp}(I) \cup \mathsf{pol}^{-}(I)$.*

Proof. Define $\mathcal{P} = \Delta \Vdash \overline{T}\,\mathbf{implements}\,I{<}\overline{U}{>}$ and $\mathcal{Q} = \Delta \Vdash \overline{V}\,\mathbf{implements}\,I{<}\overline{W}{>}$. We first prove the following auxiliary lemma:

$$\begin{gathered}\textit{If } \Delta \Vdash_{\mathsf{q}}{}' \mathcal{P} \textit{ and } \Delta \Vdash_{\mathsf{q}}{}' \mathcal{Q} \textit{ and for all } i \in \mathsf{disp}(I) \textit{ there exists } T_i' \textit{ with} \\ \Delta \vdash_{\mathsf{q}}{}' T_i' \leq T_i \textit{ and } \Delta \vdash_{\mathsf{q}}{}' T_i' \leq V_i\textit{, then } \overline{U} = \overline{W} \textit{ and } T_j = V_j \\ \textit{for all } j \notin \mathsf{disp}(I) \cup \mathsf{pol}^{-}(I). \end{gathered} \tag{B.5.7}$$

The proof is by induction in the combined height of the derivations of $\Delta \Vdash_{\mathsf{q}}{}' \mathcal{P}$ and $\Delta \Vdash_{\mathsf{q}}{}' \mathcal{Q}$. We proceed by case analysis on the last rules of these derivations. The following table lists all possible cases; cases marked with ↯ can never occur because they put conflicting requirements on the form of $\mathcal{P}$ and $\mathcal{Q}$. The remaining cases are dealt with shortly.

		$\Delta \Vdash_{\mathsf{q}}{}' \mathcal{Q}$		
		ENT-Q-ALG-ENV	ENT-Q-ALG-IMPL	ENT-Q-ALG-IFACE
$\Delta \Vdash_{\mathsf{q}}{}' \mathcal{P}$	ENT-Q-ALG-ENV	(1)	(2)	↯
	ENT-Q-ALG-IMPL	(2)	(3)	↯
	ENT-Q-ALG-IFACE	↯	↯	(4)

For (1), (2), and (3) we have $\overline{T} = \overline{G}$ and $\overline{V} = \overline{G'}$ for some $\overline{G}$ and $\overline{G'}$. Hence, by Lemma B.5.8

$$\text{for all } i \in \mathsf{disp}(I) \text{ exists } H_i \text{ with } \Delta \vdash G_i \sqcap G_i' = H_i \tag{B.5.8}$$

1. Then $\mathcal{P} \in \mathsf{sup}(\Delta)$ and $\mathcal{Q} \in \mathsf{sup}(\Delta)$. The claim now follows with WF-TENV-6.
2. Then, w.l.o.g., $\mathcal{P} \in \mathsf{sup}(\Delta)$ and $\mathcal{Q} = [\overline{U'/X}](\overline{N}\,\mathbf{implements}\,I{<}\overline{W'}{>})$ for some

   $$\mathbf{implementation}{<}\overline{X}{>}\ I{<}\overline{W'}{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots$$

   As in the preceding case, the claim follows with WF-TENV-6.
3. Then

   $$\begin{gathered}\mathbf{implementation}{<}\overline{X}{>}\ I{<}\overline{U'}{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots \\ \mathbf{implementation}{<}\overline{Y}{>}\ I{<}\overline{W'}{>}\ [\,\overline{M}\,]\ \mathbf{where}\ \overline{Q}\ \ldots\end{gathered}$$

   such that

   $$\mathcal{P} = \varphi(\overline{N}\,\mathbf{implements}\,I{<}\overline{U'}{>})$$

   with $\mathsf{dom}(\varphi) = \overline{X}$ and

   $$\mathcal{Q} = \psi(\overline{M}\,\mathbf{implements}\,I{<}\overline{W'}{>}$$

   with $\mathsf{dom}(\psi) = \overline{Y}$. We have by (B.5.8) and Lemma B.5.9 that

   $$\text{for all } i \in \mathsf{disp}(I) \text{ exists } H_i \text{ with } \emptyset \vdash \varphi N_i \sqcap \psi M_i = H_i$$

   The claim now follows with criterion WF-PROG-2.

4. Then $\overline{T} = J\text{<}\overline{U'}\text{>}$, $1 \in \mathsf{pol}^+(J)$, $J\text{<}\overline{U'}\text{>} \trianglelefteq_{\mathsf{i}} I\text{<}\overline{U}\text{>}$, and $\overline{V} = J'\text{<}\overline{W'}\text{>}$, $1 \in \mathsf{pol}^+(J')$, $J'\text{<}\overline{W'}\text{>} \trianglelefteq_{\mathsf{i}} I\text{<}\overline{W}\text{>}$. Because I is a single-headed interface, $1 \in \mathsf{disp}(I)$ by Lemma B.5.1. Hence,

$$\Delta \vdash_{\mathrm{q}}{}' T_1' \leq J\text{<}\overline{U'}\text{>}$$
$$\Delta \vdash_{\mathrm{q}}{}' T_1' \leq J'\text{<}\overline{W'}\text{>}$$

By Lemma B.1.10 one of the following holds:

- $T_1' = X$ and $X\,\mathbf{extends}\,K \in^+ \Delta$ with $K \trianglelefteq_{\mathsf{i}} J\text{<}\overline{U'}\text{>}$ and $X\,\mathbf{extends}\,K' \in^+ \Delta$ with $K' \trianglelefteq_{\mathsf{i}} J'\text{<}\overline{W'}\text{>}$. With Lemma B.5.6 and Lemma B.1.7 then $\Delta \vdash_{\mathrm{q}}{}' X \leq I\text{<}\overline{U}\text{>}$ and $\Delta \vdash_{\mathrm{q}}{}' X \leq I\text{<}\overline{W}\text{>}$. Criterion WF-TENV-4 now yields $\overline{U} = \overline{W}$ as required.
- $T_1' = L$ with $L \trianglelefteq_{\mathsf{i}} J\text{<}\overline{U'}\text{>}$ and $L \trianglelefteq_{\mathsf{i}} J'\text{<}\overline{W'}\text{>}$. With Lemma B.1.4 then $L \trianglelefteq_{\mathsf{i}} I\text{<}\overline{U}\text{>}$ and $L \trianglelefteq_{\mathsf{i}} I\text{<}\overline{W}\text{>}$. Hence, $\overline{U} = \overline{W}$ by criterion WF-PROG-6.

This finishes the proof of (B.5.7).

From $\Delta \Vdash \mathcal{P}$ and $\Delta \Vdash \mathcal{Q}$ we have $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ and $\Delta \Vdash_{\mathrm{q}} \mathcal{Q}$. By Lemma B.1.25 there exists $\overline{T''}$ and $\overline{V'}$ such that for all i

$$\Delta \vdash_{\mathrm{q}}{}' T_i \leq T_i''$$
$$T_i = T_i'' \text{ if } i \notin \mathsf{pol}^-(I)$$
$$\Delta \Vdash_{\mathrm{q}}{}' \overline{T''}\,\mathbf{implements}\,I\text{<}\overline{U}\text{>}$$
$$\Delta \vdash_{\mathrm{q}}{}' V_i \leq V_i'$$
$$V_i = V_i' \text{ if } i \notin \mathsf{pol}^-(I)$$
$$\Delta \Vdash_{\mathrm{q}}{}' \overline{V'}\,\mathbf{implements}\,I\text{<}\overline{W}\text{>}$$

With Lemma B.1.7 then $\Delta \vdash_{\mathrm{q}}{}' T_i' \leq T_i''$ and $\Delta \vdash_{\mathrm{q}}{}' T_i' \leq V_i'$ for all $i \in \mathsf{disp}(I)$. With (B.5.7) now $\overline{U} = \overline{W}$ and $T_i'' = V_i'$ if $i \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$. Assume $i \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$. Then $i \notin \mathsf{pol}^-(I)$, so $T_i = T_i''$ and $V_i = V_i'$. Hence, $T_i = V_i$ for $i \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$. □

Lemma B.5.11 (Antisymmetry of kernel subtyping). *If $\Delta \vdash_{\mathrm{q}}{}' T \leq U$ and $\Delta \vdash_{\mathrm{q}}{}' U \leq T$ then $T = U$.*

Proof. We proceed by case distinction on the last rules of the two derivations. The only combinations possible are:

SUB-Q-ALG-OBJ / SUB-Q-ALG-OBJ: Then $T = \mathit{Object} = U$.

SUB-Q-ALG-OBJ / SUB-Q-ALG-CLASS or SUB-Q-ALG-CLASS / SUB-Q-ALG-OBJ: Impossible because programs cannot define *Object*.

SUB-Q-ALG-VAR-REFL / SUB-Q-ALG-VAR-REFL: Then $T = X = U$ for some X.

SUB-Q-ALG-VAR / SUB-Q-ALG-VAR: Then $T = X$, $X\,\mathbf{extends}\,T' \in \Delta$, and $U = Y$, $Y\,\mathbf{extends}\,U' \in \Delta$, and $\Delta \vdash_{\mathrm{q}}{}' T' \leq Y$, $\Delta \vdash_{\mathrm{q}}{}' U' \leq X$. By Lemma B.1.10 then $T' = Y'$, $Y'\,\mathbf{extends}\,Y \in^* \Delta$, and $U' = X'$, $X'\,\mathbf{extends}\,X \in^* \Delta$. Hence, we have $X\,\mathbf{extends}\,Y' \in \Delta$, $Y'\,\mathbf{extends}\,Y \in^* \Delta$, $Y\,\mathbf{extends}\,X' \in \Delta$, and $X'\,\mathbf{extends}\,X \in^* \Delta$. This is a contradiction because Δ is contractive by criterion WF-TENV-1.

SUB-Q-ALG-CLASS / SUB-Q-ALG-CLASS: Then $T = N_1$, $U = N_2$ with $N_1 \trianglelefteq_{\mathsf{c}} N_2$ and $N_2 \trianglelefteq_{\mathsf{c}} N_1$. Because the class graph is acyclic by criterion WF-PROG-5, we have $N_1 = N_2$.

SUB-Q-ALG-IFACE / SUB-Q-ALG-IFACE: Then $T = K_1$, $U = K_2$ with $K_1 \trianglelefteq_{\mathsf{i}} K_2$ and $K_2 \trianglelefteq_{\mathsf{i}} K_1$. Because the interface graph is acyclic by criterion WF-PROG-5, we have $K_1 = K_2$. □

Lemma B.5.12. *If* $\Delta \vdash_{\mathrm{q}}{}' \mathit{Object} \leq T$ *then* $T = \mathit{Object}$.

Proof. With rule SUB-Q-ALG-OBJ, we have $\Delta \vdash_{\mathrm{q}}{}' T \leq \mathit{Object}$. The claim now follows with Lemma B.5.11. □

Lemma B.5.13. *The set* $\{U \mid \Delta \vdash_{\mathrm{q}}{}' T \leq U\}$ *is finite for any* T *and* Δ.

Proof. We prove that there exists a bound on the size of all types $U \in \{U \mid \Delta \vdash_{\mathrm{q}}{}' T \leq U\}$. Then, because the set of types of a certain size is finite, $\{U \mid \Delta \vdash_{\mathrm{q}}{}' T \leq U\}$ must be finite.

Let $\delta \in \mathbb{N}$ be a bound on the size of Δ and the program's superclasses and superinterfaces. That is,

- if $P \in \Delta$ then $\mathsf{size}(P) \leq \delta$,
- if $\textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } N \textbf{ where } \overline{P} \ldots$ then $\mathsf{size}(N) \leq \delta$,
- if $\textbf{interface } I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y} \textbf{ where } \overline{R}] \ldots$ then $\mathsf{size}(\overline{R}) \leq \delta$.

Differing from Definition B.4.1, the proof of this lemma defines the weight of a type as follows:

$$\begin{aligned}
\mathsf{weight}'(X) &= \mathsf{max}\{\mathsf{weight}'(T) \mid X \textbf{ extends } T \in \Delta\}\\
\mathsf{weight}'(N) &= \mathsf{size}(N)\\
\mathsf{weight}'(K) &= \mathsf{size}(K)
\end{aligned}$$

Here, by convention $\mathsf{max}\emptyset = 1$. The definition of weight' is well-formed (i.e. terminating) because Δ is contractive by criterion WF-TENV-1. Moreover, $\mathsf{weight}'(T) \in \mathbb{N}^+$ and $\mathsf{weight}'(T) \geq \mathsf{size}(T)$ for all types T.

Define the level of a type as follows:

$$\begin{aligned}
\mathsf{level}'(\mathit{Object}) &= 1\\
\mathsf{level}'(C\texttt{<}\overline{T}\texttt{>}) &= n+1 && \text{if } \textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } N \ldots \text{ and } \mathsf{level}'([\overline{T/X}]N) = n\\
\mathsf{level}'(I\texttt{<}\overline{T}\texttt{>}) &= 1 && \text{if } \textbf{interface } I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y}] \ldots\\
\mathsf{level}'(I\texttt{<}\overline{T}\texttt{>}) &= n+1 && \text{if } \textbf{interface } I\texttt{<}\overline{X}\texttt{>}\,[\overline{Y} \textbf{ where } \overline{R}] \ldots,\\
&&& \quad R_i = \overline{V_i} \textbf{ implements } K_i, \text{ and}\\
&&& \quad n = \mathsf{max}_i(\mathsf{level}'([\overline{T/X}]K_i))\\
\mathsf{level}'(X) &= \mathsf{max}\{\mathsf{level}'(T) \mid X \textbf{ extends } T \in \Delta\}
\end{aligned}$$

The definition of level' is well-formed (i.e., terminating) because the class and interface graph is acyclic by criterion WF-PROG-5. Moreover, $\mathsf{level}'(T) \in \mathbb{N}^+$ for all types T. We now show that

$$\Delta \vdash_{\mathrm{q}}{}' T \leq U \text{ implies } \mathsf{weight}'(U) \leq \delta^{\mathsf{level}'(T)} \cdot \mathsf{weight}'(T) \tag{B.5.9}$$

The proof of (B.5.9) is by induction on the derivation of $\Delta \vdash_{\mathrm{q}}{}' T \leq U$.
Case distinction on the last rule used in the derivation of $\Delta \vdash_{\mathrm{q}}{}' T \leq U$.

- *Case* SUB-Q-ALG-OBJ: Obvious.
- *Case* SUB-Q-ALG-VAR-REFL: Obvious.
- *Case* SUB-Q-ALG-VAR: Then $T = X$ and

$$\frac{X \textbf{ extends } T' \in \Delta \qquad \Delta \vdash_{\mathrm{q}}{}' T' \leq U}{\Delta \vdash_{\mathrm{q}}{}' X \leq U}$$

By the I.H. $\mathsf{weight}'(U) \leq \delta^{\mathsf{level}'(T')} \cdot \mathsf{weight}'(T') \leq \delta^{\mathsf{level}'(X)} \cdot \mathsf{weight}'(X)$.

- *Case* SUB-Q-ALG-CLASS: Then $T = N$, $U = N'$, and $N \trianglelefteq_{\mathtt{c}} N'$. We now show that

  $$N \trianglelefteq_{\mathtt{c}} N' \text{ implies } \mathsf{size}(N') \leq \delta^{\mathsf{level}'(N)} \cdot \mathsf{size}(N) \tag{B.5.10}$$

  We then have $\mathsf{weight}'(N') = \mathsf{size}(N') \leq \delta^{\mathsf{level}'(N)} \cdot \mathsf{size}(N) = \delta^{\mathsf{level}'(N)} \cdot \mathsf{weight}'(N)$ as required. The proof of (B.5.10) is by induction on the derivation of $N \trianglelefteq_{\mathtt{c}} N'$.

  Case distinction on the last rule used in the derivation of $N \trianglelefteq_{\mathtt{c}} N'$.

  - *Case* INH-CLASS-REFL: Obvious.
  - *Case* INH-CLASS-SUPER: Then $N = C\texttt{<}\overline{T}\texttt{>}$ and

    $$\frac{\textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } M \ldots \qquad [\overline{T/X}]M \trianglelefteq_{\mathtt{c}} N'}{C\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathtt{c}} N'}$$

    We have

    $$\begin{aligned} \mathsf{size}([\overline{T/X}]M) &\leq \mathsf{size}(M) + \mathsf{max}_i(\mathsf{size}(T_i)) \cdot (\mathsf{size}(M) - 1) \\ &\leq \mathsf{size}(M) + (\mathsf{size}(N) - 1) \cdot (\mathsf{size}(M) - 1) \\ &= \mathsf{size}(N) \cdot \mathsf{size}(M) - \mathsf{size}(N) + 1 \\ &\leq \delta \cdot \mathsf{size}(N) \\ \mathsf{level}'(N) &= \mathsf{level}'([\overline{T/X}]M) + 1 \end{aligned}$$

    Hence,

    $$\begin{aligned} \mathsf{size}(N') \overset{\text{I.H.}}{\leq} \delta^{\mathsf{level}'([\overline{T/X}]M)} \cdot \mathsf{size}([\overline{T/X}]M) \\ \leq \delta^{\mathsf{level}'([\overline{T/X}]M)} \cdot \delta \cdot \mathsf{size}(N) = \delta^{\mathsf{level}'(N)} \cdot \mathsf{size}(N) \end{aligned}$$

  End case distinction on the last rule used in the derivation of $N \trianglelefteq_{\mathtt{c}} N'$.

- *Case* SUB-Q-ALG-IFACE: Hence, $T = K$, $U = K'$, and $K \trianglelefteq_{\mathtt{i}} K'$. Similar to the preceding case, we show that $K \trianglelefteq_{\mathtt{i}} K'$ implies $\mathsf{size}(K') \leq \delta^{\mathsf{level}'(K)} \cdot \mathsf{size}(K)$ by induction on the derivation of $K \trianglelefteq_{\mathtt{i}} K'$. The claim also follows analogously to the preceding case.

End case distinction on the last rule used in the derivation of $\Delta \vdash_{\mathrm{q}}' T \leq U$. □

Lemma B.5.14. *Let $\mathscr{T}$ be a non-empty set of types. Suppose $\Delta \vdash_{\mathrm{q}}' T \leq V$ for all $T \in \mathscr{T}$. Then there exists a $V' \in \mathsf{mub}_\Delta(\mathscr{T})$ such that $\Delta \vdash_{\mathrm{q}}' V' \leq V$.*

Proof. We argue by contradiction. To do so, we construct an infinite chain $U_0, U_1, \ldots$ such that $U_i \neq U_j$ for all $i \neq j$ and $\Delta \vdash_{\mathrm{q}}' T \leq U_i$ for all $T \in \mathscr{T}$ and all i. Hence, because $\mathscr{T} \neq \emptyset$, there exists some $T \in \mathscr{T}$ such that the set $\{U \mid \Delta \vdash_{\mathrm{q}}' T \leq U\}$ is infinite. This is then a contradiction to Lemma B.5.13.

Here is how we construct the infinite chain $U_0, U_1, U_2, \ldots$:

- Assume $V = U_0 \notin \mathsf{mub}_\Delta(\mathscr{T})$. (Otherwise, choose $V' = U_0$ and we are done.) Hence, there exists $U_1 \neq U_0$ with $\Delta \vdash_{\mathrm{q}}' T \leq U_1$ for all $T \in \mathscr{T}$ and $\Delta \vdash_{\mathrm{q}}' U_1 \leq U_0$.
- Assume $U_1 \notin \mathsf{mub}_\Delta(\mathscr{T})$. (Otherwise, choose $V' = U_1$ and we are done.) Hence, there exists $U_2 \neq U_1$ with $\Delta \vdash_{\mathrm{q}}' T \leq U_2$ for all $T \in \mathscr{T}$ and $\Delta \vdash_{\mathrm{q}}' U_2 \leq U_1$.
- ...
- Assume $U_i \notin \mathsf{mub}_\Delta(\mathscr{T})$. (Otherwise, choose $V' = U_i$ and we are done.) Hence, there exists $U_{i+1} \neq U_i$ with $\Delta \vdash_{\mathrm{q}}' T \leq U_{i+1}$ for all $T \in \mathscr{T}$ and $\Delta \vdash_{\mathrm{q}}' U_{i+1} \leq U_i$.

- …

From this construction we have:

$$\begin{gathered}\Delta \vdash_{\mathrm{q}}' T \leq U_i \quad \text{for all } i \in \mathbb{N}, T \in \mathscr{T} \\ U_i \neq U_{i+1} \quad \text{for all } i \in \mathbb{N} \\ \Delta \vdash_{\mathrm{q}}' U_{i+1} \leq U_i \quad \text{for all } i \in \mathbb{N}\end{gathered}$$

We still have to verify that $U_i \neq U_j$ if $i \neq j$. Suppose $i < j$ with $U_i = U_j$. Because subtyping is transitive we have $\Delta \vdash_{\mathrm{q}}' U_j \leq U_{i+1}$. Hence, $\Delta \vdash_{\mathrm{q}}' U_i \leq U_{i+1}$. But we also have $\Delta \vdash_{\mathrm{q}}' U_{i+1} \leq U_i$. With Lemma B.5.11 now $U_i = U_{i+1}$ which is a contradiction. □

If we choose $V = \mathit{Object}$ in Lemma B.5.14, we get the following corollary:

Corollary B.5.15. *For any set of types $\mathscr{T} \neq \emptyset$, $\mathsf{mub}_\Delta(\mathscr{T}) \neq \emptyset$.*

Lemma B.5.16. *If $T \in \mathsf{mub}_\Delta(\mathscr{U})$ then $\Delta \vdash_{\mathrm{q}}' U \leq T$ for all $U \in \mathscr{U}$.*

Proof. Obvious. □

Lemma B.5.17. *Let $\mathscr{T}$ be a non-empty set of types. If $G_1 \in \mathsf{mub}_\Delta(\mathscr{T})$ and $G_2 \in \mathsf{mub}_\Delta(\mathscr{T})$ then $G_1 = G_2$.*

Proof. Because $\mathscr{T} \neq \emptyset$, there exists $T \in \mathscr{T}$ such that $\Delta \vdash_{\mathrm{q}}' T \leq G_i$ for $i = 1, 2$. By Lemma B.5.7 either $\Delta \vdash_{\mathrm{q}}' G_1 \leq G_2$ or $\Delta \vdash_{\mathrm{q}}' G_2 \leq G_1$. W.l.o.g. assume $\Delta \vdash_{\mathrm{q}}' G_1 \leq G_2$. But because $G_2 \in \mathsf{mub}_\Delta(\mathscr{T})$ we must have that $G_1 = G_2$. □

Lemma B.5.18. *If $\Delta \vdash_{\mathrm{q}}' T \leq N$ then $\mathsf{bound}_\Delta(T) = M$ with $M \trianglelefteq_{\mathsf{c}} N$.*

Proof. Obvious. □

Lemma B.5.19.

(*i*) *If $N \trianglelefteq_{\mathsf{c}} N'$ then $\mathsf{ftv}(N') \subseteq \mathsf{ftv}(N)$.*

(*ii*) *If $K \trianglelefteq_{\mathsf{i}} K'$ then $\mathsf{ftv}(K') \subseteq \mathsf{ftv}(K)$.*

(*iii*) *If $\Delta \vdash_{\mathrm{q}}' T \leq U$ then $\mathsf{ftv}(U) \subseteq \mathsf{ftv}(\Delta, T)$.*

Proof. We prove all three parts by straightforward inductions on the given derivations. □

Lemma B.5.20 (Strengthening). *Let $\Delta' = \Delta, X$ **implements** K and $\Delta'' = \Delta, X$.*

(*i*) *If $\Delta' \vdash T$ ok and $X \notin \mathsf{ftv}(\Delta, K, T)$ then $\Delta \vdash T$ ok.*

(*ii*) *If $\Delta' \vdash \mathcal{P}$ ok and $X \notin \mathsf{ftv}(\Delta, K, \mathcal{P})$ then $\Delta \vdash \mathcal{P}$ ok.*

(*iii*) *If $\Delta'' \vdash T$ ok and $X \notin \mathsf{ftv}(\Delta, T)$ then $\Delta \vdash T$ ok.*

(*iv*) *If $\Delta'' \vdash \mathcal{P}$ ok and $X \notin \mathsf{ftv}(\Delta, \mathcal{P})$ then $\Delta \vdash \mathcal{P}$ ok.*

Proof. We first prove:

(*a*) *If $\mathcal{D}_1 :: \Delta' \vdash_{\mathrm{q}}' V \leq U$ then $\Delta \vdash_{\mathrm{q}}' V \leq U$.*

(*b*) *If $\mathcal{D}_2 :: \Delta' \Vdash_{\mathrm{q}}' \mathcal{P}$ and $X \notin \mathsf{ftv}(\Delta, K, \mathcal{P})$ then $\Delta \Vdash_{\mathrm{q}}' \mathcal{P}$.*

(*c*) *If $\mathcal{D}_3 :: \Delta' \vdash_{\mathrm{q}} V \leq U$ and $X \notin \mathsf{ftv}(\Delta, V, U)$ then $\Delta \vdash_{\mathrm{q}} V \leq U$.*

(*d*) *If $\mathcal{D}_4 :: \Delta' \Vdash_{\mathrm{q}} \mathcal{P}$ and $X \notin \mathsf{ftv}(\Delta, K, \mathcal{P})$ then $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$.*

The proof of (a) is straightforward because kernel subtyping does not use implementation constraints. The proof of (b), (c), and (d) is by induction on the combined height of $\mathcal{D}_2$, $\mathcal{D}_3$, and $\mathcal{D}_4$.

(b) *Case distinction* on the last rule of the derivation of $\Delta' \Vdash_{\mathrm{q}}{}' \mathcal{P}$.

- *Case* rule ENT-Q-ALG-ENV: Then $R \in \Delta'$ and $\mathcal{P} \in \mathsf{sup}(R)$. If $R = X\,\mathbf{implements}\,K$ then by Lemma B.1.22 $\mathcal{P} = X\,\mathbf{implements}\,K'$. But this is a contradiction to the assumption $X \notin \mathsf{ftv}(\mathcal{P})$. Hence, $R \neq X\,\mathbf{implements}\,K$, so $R \in \Delta$ and the claim follows with ENT-Q-ALG-ENV.
- *Case* rule ENT-Q-ALG-IMPL: Then

$$\frac{\mathbf{implementation}\texttt{<}\overline{Y}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots \qquad \Delta' \Vdash_{\mathrm{q}} [\overline{U/Y}]\overline{P}}{\Delta' \Vdash_{\mathrm{q}}{}' \underbrace{[\overline{U/Y}](\overline{N}\,\mathbf{implements}\,I\texttt{<}\overline{T}\texttt{>})}_{=\mathcal{P}}}$$

With criterion WF-IMPL-2 we have $\overline{X} \subseteq \mathsf{ftv}(\overline{N})$. With $X \notin \mathsf{ftv}(\mathcal{P})$ we then have $X \notin \mathsf{ftv}(\overline{U})$. Hence, $X \notin \mathsf{ftv}([\overline{U/X}]\overline{P})$. Applying part (d) of the I.H. yields $\Delta \Vdash_{\mathrm{q}} [\overline{U/X}]\overline{P}$, so the claim follows with ENT-Q-ALG-IMPL.
- *Case* rule ENT-Q-ALG-IFACE: Obvious.

End case distinction on the last rule of the derivation of $\Delta' \Vdash_{\mathrm{q}}{}' \mathcal{P}$.

(c) If the last rule of $\mathcal{D}_3$ is SUB-Q-ALG-KERNEL, then the claim follows by (a). Otherwise, we have

$$\Delta' \vdash_{\mathrm{q}}{}' V \leq W$$
$$\Delta' \Vdash_{\mathrm{q}}{}' W\,\mathbf{implements}\,L$$

with $U = L$. By (a) then $\Delta \vdash_{\mathrm{q}}{}' V \leq W$. With Lemma B.5.19 we have $\mathsf{ftv}(W) \subseteq \mathsf{ftv}(V, \Delta)$. Hence, $X \notin \mathsf{ftv}(W)$. With part (b) of the I.H. we then have $\Delta \Vdash_{\mathrm{q}}{}' W\,\mathbf{implements}\,L$. The claim now follows with rule SUB-Q-ALG-IMPL.

(d) Follows trivially from (a) and parts (b), (c) of the I.H.

Constraint entailment does not use the type variable component of Δ'' at all, so the following claim is trivial to prove:

$$\text{If } \Delta'' \Vdash_{\mathrm{q}} \mathcal{P} \text{ then } \Delta \Vdash_{\mathrm{q}} \mathcal{P} \tag{B.5.11}$$

Using (d) and (B.5.11), we easily show the original claim by an induction on the given derivations. □

Lemma B.5.21 (Interface inheritance propagates well-formedness). *If $K \trianglelefteq_{\mathrm{i}} L$ and $\Delta \vdash K$ ok then $\Delta \vdash L$ ok*

Proof. We proceed by induction on the derivation of $K \trianglelefteq_{\mathrm{i}} L$
Case distinction on the last rule of the derivation of $K \trianglelefteq_{\mathrm{i}} L$.

- *Case* rule INH-IFACE-REFL: Obvious.
- *Case* rule INH-IFACE-SUPER: Then

$$\frac{\begin{array}{l}\mathbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\ [Y\ \mathbf{where}\ \overline{R}]\ \mathbf{where}\ \overline{P}\ \ldots\\ R_i = Y\,\mathbf{implements}\,K' \qquad [\overline{V/X}]K' \trianglelefteq_{\mathrm{i}} L\end{array}}{\Delta \vdash I\texttt{<}\overline{V}\texttt{>} \leq L}$$

with $K = I{<}\overline{V}{>}$. We now prove that $\Delta \vdash [\overline{V/X}]K' \ \mathsf{ok}$. The original claim then follows by the I.H.

Because $\Delta \vdash K \ \mathsf{ok}$, we have

$$\Delta, Y\ \mathbf{implements}\ I{<}\overline{V}{>}, Y \Vdash [\overline{V/X}]\overline{R}, \overline{P}$$

$$\Delta \vdash \overline{V}\ \mathsf{ok}$$

with

$$Y \notin \mathsf{ftv}(\overline{V}, \Delta) \tag{B.5.12}$$

Lemma B.2.23 gives us $\Delta, Y\ \mathbf{implements}\ I{<}\overline{V}{>}, Y \vdash \overline{V}\ \mathsf{ok}$. The underlying program is well-typed, so $\overline{R}, \overline{P}, \overline{X}, Y \vdash R_i\ \mathsf{ok}$. Hence, with Lemma B.2.24,

$$\Delta, Y\ \mathbf{implements}\ I{<}\overline{V}{>}, Y \vdash [\overline{V/X}]R_i\ \mathsf{ok}$$

Then $\Delta, Y\ \mathbf{implements}\ I{<}\overline{V}{>}, Y \vdash [\overline{V/X}]K'\ \mathsf{ok}$. By criterion WF-IFACE-2, $Y \notin \mathsf{ftv}(K')$. With (B.5.12) and two applications of Lemma B.5.20, we get $\Delta \vdash [\overline{V/X}]K'\ \mathsf{ok}$ as required.

End case distinction on the last rule of the derivation of $K \trianglelefteq_{\mathsf{i}} L$. □

Lemma B.5.22 (Kernel subtyping propagates well-formedness). *If* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash T\ \mathsf{ok}$ *and* $\Delta \vdash_{\mathsf{q}}{}' T \leq U$ *then* $\Delta \vdash U\ \mathsf{ok}$.

Proof. Straightforward induction on the derivation of $\Delta \vdash_{\mathsf{q}}{}' T \leq U$, making use of Lemma B.2.25 and Lemma B.5.21. □

Lemma B.5.23. *If* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash T\ \mathsf{ok}$ *and* $\mathsf{bound}_\Delta(T) = N$*, then* $\Delta \vdash N\ \mathsf{ok}$.

Proof. Follows by Lemma B.5.22. □

Lemma B.5.24. *If* $\Delta \vdash_{\mathsf{q}}{}' X \leq I{<}\overline{T}{>}$ *then* $1 \in \mathsf{pol}^-(I)$.

Proof. We proceed by induction on the derivation of $\Delta \vdash_{\mathsf{q}}{}' X \leq I{<}\overline{T}{>}$. The derivation must end with an application of rule SUB-Q-ALG-VAR. Hence, $X\ \mathbf{extends}\ T \in \Delta$ and $\Delta \vdash_{\mathsf{q}}{}' T \leq I{<}\overline{T}{>}$.
Case distinction on the form of T.

- *Case* $T = Y$: The claim then follows from the I.H.
- *Case* $T = N$: Impossible by Lemma B.1.10.
- *Case* $T = J{<}\overline{U}{>}$: Then $J{<}\overline{U}{>} \trianglelefteq_{\mathsf{i}} I{<}\overline{T}{>}$ by Lemma B.1.10 and $1 \in \mathsf{pol}^-(J)$ by criterion WF-TENV-5. The claim now follows with Lemma B.1.18.

End case distinction on the form of T. □

Lemma B.5.25. *Assume* $\mathsf{mtype}_\Delta(m^{\mathrm{c}}, C{<}\overline{W}{>}) = {<}\overline{X}{>}\overline{U\,x}^n \to U\ \mathbf{where}\ \overline{\mathcal{P}}$ *and let* φ *be a substitution with* $\mathsf{dom}(\varphi) = \overline{X}$*. Suppose* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash N\ \mathsf{ok}$*. If* $N \trianglelefteq_{\mathsf{c}} C{<}\overline{W}{>}$ *and* $\Delta \Vdash \varphi\overline{\mathcal{P}}$*, then* $\mathsf{a\text{-}mtype}^{\mathrm{c}}(m, N) = {<}\overline{X}{>}\overline{U\,x}^n \to U'\ \mathbf{where}\ \overline{\mathcal{P}}$ *such that* $\Delta \vdash \varphi U' \leq \varphi U$.

Proof. From $\mathsf{mtype}_\Delta(m^{\mathrm{c}}, C{<}\overline{W}{>}) = {<}\overline{X}{>}\overline{U\,x}^n \to U\ \mathbf{where}\ \overline{\mathcal{P}}$ we get

$$\mathbf{class}\ C{<}\overline{Y}{>}\ \mathbf{extends}\ M\ \mathbf{where}\ \overline{Q}\,\{\ldots\ \overline{m : \mathit{msig}\,\{e\}}\,\}$$

$$m_j = m^{\mathrm{c}}$$

$${<}\overline{X}{>}\overline{U\,x}^n \to U\ \mathbf{where}\ \overline{\mathcal{P}} = [\overline{W/Y}]\mathit{msig}_j \tag{B.5.13}$$

Case distinction on the last rule in the derivation of $N \trianglelefteq_{\mathsf{c}} C{<}\overline{W}{>}$.

- *Case* inh-class-refl: Then $N = C\texttt{<}\overline{W}\texttt{>}$, so the claim follows with an application of rule alg-mtype-class-base and reflexivity of subtyping.
- *Case* inh-class-super: Then $N = D\texttt{<}\overline{V}\texttt{>}$ and

$$\frac{\textbf{class } D\texttt{<}\overline{Z}\texttt{>} \textbf{ extends } M' \textbf{ where } \overline{Q'}\,\{\ldots\ \overline{m' : msig'\ \{e'\}}\,\} \qquad [\overline{V/Z}]M' \trianglelefteq_{\mathsf{c}} C\texttt{<}\overline{W}\texttt{>}}{D\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathsf{c}} C\texttt{<}\overline{W}\texttt{>}}$$

Clearly, $D\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathsf{c}} [\overline{V/Z}]M'$, so we get with $\Delta \vdash N\ \mathsf{ok}$ and Lemma B.2.25 that $\Delta \vdash [\overline{V/Z}]M'\ \mathsf{ok}$.

Case distinction on whether or not $m \in \overline{m'}$.

- *Case* $m \notin \overline{m'}$: The claim then follows from the I.H. and an application of rule alg-mtype-class-super.
- *Case* $m \in \overline{m'}$: Assume $m = m'_i$. Because the underlying program is well-typed, we have

$$\overline{Q'}, \overline{Z} \vdash m'_i : msig'_i\ \{e'_i\}\ \mathsf{ok\ in}\ D\texttt{<}\overline{Z}\texttt{>}$$

Hence,

$$\mathsf{override\text{-}ok}_{\overline{Q'},\overline{Z}}(m'_i : msig'_i, D\texttt{<}\overline{Z}\texttt{>})$$

With $D\texttt{<}\overline{V}\texttt{>} \trianglelefteq_{\mathsf{c}} C\texttt{<}\overline{W}\texttt{>}$ and Lemma B.2.33 there exists $\overline{W'}$ such that

$$\begin{gathered} D\texttt{<}\overline{Z}\texttt{>} \trianglelefteq_{\mathsf{c}} C\texttt{<}\overline{W'}\texttt{>} \\ [\overline{V/Z}]\overline{W'} = \overline{W} \end{gathered} \tag{B.5.14}$$

By inverting rule ok-override

$$\overline{Q'}, \overline{Z} \vdash msig'_i \leq [\overline{W'/Y}]msig_j$$

Assume

$$\begin{aligned} msig'_i &= \texttt{<}\overline{X'''}\texttt{>}\,\overline{U'''\ x'''} \to U''' \textbf{ where } \overline{P'''} \\ msig_j &= \texttt{<}\overline{X''}\texttt{>}\,\overline{U''\ x''} \to U'' \textbf{ where } \overline{P''} \end{aligned}$$

Then by rule sub-msig

$$\overline{X'''} = \overline{X''}$$

$$\overline{U'''} = [\overline{W'/Y}]\overline{U''}$$

$$\overline{x'''} = \overline{x''}$$

$$\overline{P'''} = [\overline{W'/Y}]\overline{P''} \tag{B.5.15}$$

$$\overline{Q'}, \overline{Z}, \overline{P'''}, \overline{X'''} \vdash U''' \leq [\overline{W'/Y}]U'' \tag{B.5.16}$$

From (B.5.13)

$$\overline{X''} = \overline{X}$$

$$[\overline{W/Y}]\overline{U''} = \overline{U}$$

$$\overline{x''} = \overline{x}$$

$$[\overline{W/Y}]U'' = U$$

$$[\overline{W/Y}]\overline{P''} = \overline{\mathcal{P}}$$

Moreover, we have with (B.5.14) and the fact that $\overline{Z} \cap \mathsf{ftv}(\overline{U''}, U'', \overline{P''}) = \emptyset$

$$\begin{aligned}[\overline{V/Z}][\overline{W'/Y}](\overline{U''}, U'', \overline{P''}) &= \\ [\overline{W/Y}](\overline{U''}, U'', \overline{P''}) &= \\ (\overline{U}, U, \overline{\mathcal{P}})\end{aligned} \tag{B.5.17}$$

Hence, we have with rule ALG-MTYPE-CLASS-BASE

$$\begin{aligned}\mathsf{a\text{-}mtype}^{\mathrm{c}}(m, D\texttt{<}\overline{V}\texttt{>}) &= [\overline{V/Z}]\mathit{msig}'_i \\ &= [\overline{V/Z}](\texttt{<}\overline{X'''}\texttt{>}\,\overline{U'''\,x'''} \to U''' \textbf{ where } \overline{P'''}) \\ &= [\overline{V/Z}](\texttt{<}\overline{X}\texttt{>}\,\overline{[\overline{W'/Y}]U''\,x} \to U''' \textbf{ where } \overline{[\overline{W'/Y}]P''}) \\ &= \texttt{<}\overline{X}\texttt{>}\,\overline{[\overline{W/Y}]U''\,x} \to [\overline{V/Z}]\overline{U'''} \textbf{ where } [\overline{W/Y}]\overline{P''} \\ &= \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to [\overline{V/Z}]\overline{U'''} \textbf{ where } \overline{\mathcal{P}}\end{aligned}$$

To finish this case, we still need to show that for $U' = [\overline{V/Z}]\overline{U'''}$ we have $\Delta \vdash \varphi U' \leq \varphi U$.

From the assumption $\Delta \vdash D\texttt{<}\overline{V}\texttt{>}\ \mathsf{ok}$ we get $\Delta \Vdash [\overline{V/Z}]\overline{Q'}$. W.l.o.g. $\overline{X} \cap \mathsf{ftv}([\overline{V/Z}]\overline{Q'}) = \emptyset$. Hence, $\Delta \Vdash \varphi[\overline{V/Z}]\overline{Q'}$. From (B.5.15) and (B.5.17) and the assumption $\Delta \Vdash \varphi\overline{\mathcal{P}}$ we get $\Delta \Vdash \varphi[\overline{V/Z}]\overline{P'''}$. Thus, with (B.5.16) and Corollary B.1.28

$$\Delta \vdash \varphi[\overline{V/Z}]U''' \leq \varphi[\overline{V/Z}][\overline{W'/Y}]U''$$

But with (B.5.17) we have $\varphi[\overline{V/Z}][\overline{W'/Y}]U'' = \varphi U$.

End case distinction on whether or not $m \in \overline{m'}$.

End case distinction on the last rule in the derivation of $N \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{W}\texttt{>}$. □

Proof of Theorem 3.32. Case distinction on the form of m.

- *Case* $m = m^{\mathrm{c}}$: Then $T = C\texttt{<}\overline{W}\texttt{>}$. We have by Lemma B.1.14 that $\Delta \vdash_{\mathrm{q}}' T' \leq C\texttt{<}\overline{W}\texttt{>}$. By Lemma B.5.18 we have

$$\mathsf{bound}_\Delta(T') = N$$
$$N \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{W}\texttt{>}$$

  With Lemma B.5.23 we get $\Delta \vdash N\ \mathsf{ok}$. The claim now follows with an application of Lemma B.5.25.

- *Case* $m = m^{\mathrm{i}}$: From $\mathsf{mtype}_\Delta(m, T) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x}^{n} \to U \textbf{ where } \overline{\mathcal{P}}$ we get

$$\textbf{interface } I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}^{l} \textbf{ where } \overline{R}\,] \textbf{ where } \overline{P}\,\{\ldots\ \overline{\mathit{rcsig}}\,\}$$
$$\mathit{rcsig}_j = \textbf{receiver}\,\{\overline{m : \mathit{msig}}\}$$
$$m = m_k$$
$$\mathit{msig}_k = \texttt{<}\overline{X}\texttt{>}\,\overline{U''\,x} \to U'' \textbf{ where } \overline{P''}$$
$$\Delta \Vdash \overline{T'} \textbf{ implements } I\texttt{<}\overline{W}\texttt{>} \tag{B.5.18}$$
$$T'_j = T$$
$$(\overline{U}, U, \overline{\mathcal{P}}) = [\overline{T'/Z}, \overline{W/Z'}](\overline{U''}, U'', \overline{P''}) \tag{B.5.19}$$

  By Lemma B.1.32, there are two possibilities.

  Case distinction on the possibilities left by Lemma B.1.32.

- *Case* first possibility:

$$
\begin{aligned}
&[l] = \mathscr{N}_1 \mathbin{\dot\cup} \mathscr{N}_2 \\
&T'_i = K_i \text{ for all } i \in \mathscr{N}_1 \\
&i \in \mathsf{pol}^-(I) \text{ for all } i \in \mathscr{N}_1 && \text{(B.5.20)} \\
&T'_i = G_i \text{ for all } i \in \mathscr{N}_2 && \text{(B.5.21)} \\
&\Delta \Vdash \overline{T''}\ \textbf{implements}\ I\texttt{<}\overline{W}\texttt{>} \\
&\text{for all } \overline{T''} \text{ with } T''_i = G_i \text{ for all } i \in \mathscr{N}_2 && \text{(B.5.22)}
\end{aligned}
$$

Define for all $i \in [l]$:

$$
\mathscr{V}_i = \begin{cases} \mathsf{sresolve}_{\Delta;Z_i}(\overline{U''}, \overline{T}) & \text{if } i \neq j \\ \mathsf{sresolve}_{\Delta;Z_i}(Z_j\,\overline{U''}, T'\,\overline{T}) & \text{if } i = j \end{cases} \qquad \text{(B.5.23)}
$$

$$
V^?_i = \begin{cases} \mathsf{nil} & \text{if } \mathscr{V}_i = \emptyset \\ T'_i & \text{if } \mathscr{V}_i \neq \emptyset \text{ and } i \in \mathscr{N}_2 \\ \mathit{Object} & \text{if } \mathscr{V}_i \neq \emptyset \text{ and } i \in \mathscr{N}_1 \end{cases} \qquad \text{(B.5.24)}
$$

We now prove

$$
\begin{gathered}
\textit{for all } i \in [l]\textit{, either } V^?_i = \mathsf{nil} \\
\textit{or } V^?_i \neq \mathsf{nil} \textit{ and } \Delta \vdash_{\mathrm{q}}{}' V'_i \leq V_i \textit{ for some } V'_i \in \mathscr{V}_i
\end{gathered} \qquad \text{(B.5.25)}
$$

Assume $i \in [l]$.
Case distinction on whether or not $\mathscr{V}_i = \emptyset$.

- *Case* $\mathscr{V}_i = \emptyset$: Then $V_i = \mathsf{nil}$. Thus, (B.5.25) holds for this specific i.
- *Case* $\mathscr{V}_i \neq \emptyset$: Define

$$
\mathscr{T}_i = \{T_q \mid q \in [n], U''_q = Z_i\} \cup (\text{if } i = j \text{ then } \{T'\} \text{ else } \emptyset)
$$

Then

$$
\mathscr{V}_i = \mathsf{mub}_\Delta \mathscr{T}_i \neq \emptyset \qquad \text{(B.5.26)}
$$

by definition of $\mathsf{sresolve}$. With Corollary B.5.15 we get $\mathscr{V}_i \neq \emptyset$. If $i \in \mathscr{N}_1$ then $V^?_i = \mathit{Object}$, so (B.5.25) holds for this specific i. Now suppose $i \in \mathscr{N}_2$. Then $T'_i = G_i$ by (B.5.21). From the assumptions we get

$$
(\forall q \in [n])\ \Delta \vdash T_q \leq \varphi U_q
$$

Let $q \in [n]$ such that $U''_q = Z_i$. W.l.o.g., $\overline{X} \cap \mathsf{ftv}(\overline{T'}) = \emptyset$. Hence, with (B.5.19)

$$
\varphi U_q = \varphi T'_i = T'_i = G_i
$$

Thus, with Lemma B.1.14

$$
\Delta \vdash_{\mathrm{q}}{}' T_q \leq T'_i
$$

If $i = j$ then we also have $T'_i = T'_j = T$, so by the assumption $\Delta \vdash T' \leq T$

$$
\Delta \vdash T' \leq T'_i
$$

Then again with Lemma B.1.14

$$\Delta \vdash_{\mathrm{q}}{}' T' \leq T_i'$$

Hence,

$$\Delta \vdash_{\mathrm{q}}{}' \tilde{T} \leq T_i' \text{ for all } \tilde{T} \in \mathscr{T}_i$$

By (B.5.26) and Lemma B.5.14, there exists $V_i' \in \mathscr{V}_i$ such that

$$\Delta \vdash_{\mathrm{q}}{}' V_i' \leq T_i'$$

But $V_i^? = T_i'$ because $i \in \mathscr{N}_2$.

End case distinction on whether or not $\mathscr{V}_i = \emptyset$.

This finishes the proof of (B.5.25).

Now define

$$\begin{aligned} \mathscr{M} = \{\overline{V}\,\mathbf{implements}\,I\text{<}\overline{V''}\text{>} \mid (\forall i \in [l])\ &\text{if } \mathscr{V}_i = \emptyset \text{ then } V_i^? = \mathsf{nil} \\ &\text{else define } V_i^? \text{ such that} \\ &\quad \Delta \vdash_{\mathrm{q}}{}' V_i' \leq V_i^? \text{ for } V_i' \in \mathscr{V}_i, \\ \Delta \Vdash_{\mathrm{a}}^? \overline{V^?}\,\mathbf{implements}\,I\text{<}\overline{\mathsf{nil}}\text{>} \mathrel{-\!\!\triangleright} \overline{V}\,&\mathbf{implements}\,I\text{<}\overline{V''}\text{>}\} \end{aligned} \tag{B.5.27}$$

We now show that $\mathscr{M} \neq \emptyset$. Define for all $i \in [l]$

$$T_i''' = \begin{cases} T_i' & \text{if } V_i^? = \mathsf{nil} \\ V_i^? & \text{otherwise} \end{cases} \tag{B.5.28}$$

With (B.5.22) and the definition of $V_i^?$:

$$\Delta \Vdash \overline{T'''}\,\mathbf{implements}\,I\text{<}\overline{W}\text{>} \tag{B.5.29}$$

Clearly, $\overline{V^?}\,\overline{\mathsf{nil}} \sim \overline{T'''}\,\overline{W}$ and $V_i^? \neq \mathsf{nil}$ if $i \in \mathsf{disp}(I)$. Hence, by Theorem 3.29

$$\Delta \Vdash_{\mathrm{a}}^? \overline{V^?}\,\mathbf{implements}\,I\text{<}\overline{\mathsf{nil}}\text{>} \mathrel{-\!\!\triangleright} \overline{W'}\,\mathbf{implements}\,I\text{<}\overline{W}\text{>}$$

for $\overline{W'}$ such that

$$T_i''' = W_i' \text{ if } V_i^? \neq \mathsf{nil} \text{ or } i \notin \mathsf{pol}^-(I) \tag{B.5.30}$$

With (B.5.25) we thus have

$$\overline{W'}\,\mathbf{implements}\,I\text{<}\overline{W}\text{>} \in \mathscr{M} \tag{B.5.31}$$

so

$$\mathscr{M} \neq \emptyset \tag{B.5.32}$$

Moreover, for all $\overline{V}\,\mathbf{implements}\,I\text{<}\overline{V''}\text{>} \in \mathscr{M}$ the following holds:

$$\Delta \vdash_{\mathrm{q}}{}' T_q \leq V_i \text{ for all } i \in [l], q \in [n] \text{ with } U_q'' = Z_i \tag{B.5.33}$$

$$\Delta \vdash_{\mathrm{q}}{}' T' \leq V_j \tag{B.5.34}$$

$$\overline{V''} = \overline{W} \tag{B.5.35}$$

$$V_i = T_i' \text{ for all } i \in [l], i \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I) \tag{B.5.36}$$

- ∗ Equations (B.5.33) and (B.5.34) follow from (B.5.27) and (B.5.23) and with Lemma B.5.4.
- ∗ To prove equations (B.5.35) and (B.5.36), proceed as follows: We have by Theorem 3.28 and (B.5.27) that

  $$\Delta \Vdash \overline{V}\ \textbf{implements}\ I\texttt{<}\overline{V''}\texttt{>}$$

  With (B.5.29) we get

  $$\Delta \Vdash \overline{T'''}\ \textbf{implements}\ I\texttt{<}\overline{W}\texttt{>}$$

  Suppose $i' \in \mathsf{disp}(I)$. Clearly, $\mathscr{V}_{i'} \neq \emptyset$. Thus, using Lemma B.5.4, (B.5.31), (B.5.30), and (B.5.27) there exists $V'_{i'}, V''_{i'} \in \mathscr{V}_{i'}$ such that

  $$\Delta \vdash_{\mathrm{q}}{}' V'_{i'} \leq V_{i'}$$
  $$\Delta \vdash_{\mathrm{q}}{}' V''_{i'} \leq T'''_{i'}$$

  Define $T'' = T'$ if $i' = j$ and $T'' = T_q$ for some $q \in [n]$ with $U''_q = Z_{i'}$ otherwise. By (B.5.23), the definition of $\mathsf{sresolve}$, and Lemma B.5.16 we have

  $$\Delta \vdash_{\mathrm{q}}{}' T'' \leq V'_{i'}$$
  $$\Delta \vdash_{\mathrm{q}}{}' T'' \leq V''_{i'}$$

  Hence,

  $$\Delta \vdash_{\mathrm{q}}{}' T'' \leq V_{i'}$$
  $$\Delta \vdash_{\mathrm{q}}{}' T'' \leq T'''_{i'}$$

  With Lemma B.5.10 we then get

  $$\overline{V''} = \overline{W}$$
  $$V_i = T'''_i \text{ for all } i \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$$

  This proves (B.5.35). Now assume $i \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$. Then $i \in \mathscr{N}_2$ by (B.5.20). By (B.5.28) and (B.5.24) we have $T'''_i = T'_i$. This proves (B.5.36).

Define

$$p^? = \begin{cases} i & \text{if } U'' = Z_i \\ \mathsf{nil} & \text{otherwise} \end{cases} \tag{B.5.37}$$

Now assume

$$\mathsf{pick\text{-}constr}^{p^?}_{\Delta}\,\mathscr{M} = \overline{V}\ \textbf{implements}\ I\texttt{<}\overline{V''}\texttt{>} \tag{B.5.38}$$

for some $\overline{V}\ \textbf{implements}\ I\texttt{<}\overline{V''}\texttt{>}$. (We will prove (B.5.38) shortly.)
We then can use rule ALG-MTYPE-IFACE to derive

$$\begin{aligned} \mathsf{a\text{-}mtype}_{\Delta}(m, T', \overline{T}) &= [\overline{V/Z}, \overline{V''/Z'}]\mathit{msig}_k \\ &= [\overline{V/Z}, \overline{V''/Z'}](\texttt{<}\overline{X}\texttt{>}\ \overline{U''\,x} \rightarrow U''\ \textbf{where}\ \overline{P''}) \end{aligned}$$

From criterion WF-IFACE-3 we have $\overline{Z} \cap \mathsf{ftv}(\overline{P'''}) = \emptyset$. With (B.5.19) and (B.5.35) we thus get

$$[\overline{V/Z}, \overline{V''/Z'}]\overline{P''} = \overline{\mathcal{P}}$$

Now suppose $i \in [n]$. Define $U'_i = [\overline{V/Z}, \overline{V''/Z'}]U''_i$.

- If $\overline{Z} \cap \mathsf{ftv}(U''_i) = \emptyset$ then with (B.5.19) and (B.5.35)

  $$\varphi U'_i = \varphi[\overline{V/Z}, \overline{V''/Z'}]U''_i = \varphi[\overline{V''/Z'}]U''_i = \varphi U_i$$

  We now get

  $$\Delta \vdash T_i \le \varphi U'_i$$

  by the assumption $\Delta \vdash T_i \le \varphi U_i$.
- If $\overline{Z} \cap \mathsf{ftv}(U''_i) \neq \emptyset$ then by criterion WF-IFACE-3 $U''_i = Z_{i'}$ for some $i' \in [l]$. W.l.o.g., $\overline{X} \cap \mathsf{ftv}(\overline{V}) = \emptyset$. Hence,

  $$\varphi U'_i = \varphi[\overline{V/Z}, \overline{V''/Z'}]U''_i = \varphi V_{i'} = V_{i'}$$

  With (B.5.33) we have

  $$\Delta \vdash_{\mathrm{q}}{}' T_i \le V_{i'}$$

  Hence,

  $$\Delta \vdash T_i \le \varphi U'_i$$

Thus, $\Delta \vdash T_i \le \varphi U'_i$ for all $i \in [n]$.
Define

$$U' = [\overline{V/Z}, \overline{V''/Z'}]U'' \tag{B.5.39}$$

We still need to prove $\Delta \vdash \varphi U' \le \varphi U$ and (B.5.38).
Case distinction on whether or not $U'' \in \overline{Z}$.

- *Case* $U'' \notin \overline{Z}$: Then $\overline{Z} \cap \mathsf{ftv}(U'') = \emptyset$ by criterion WF-IFACE-3. By (B.5.37) we have $p^? = \mathsf{nil}$. Then (B.5.38) holds trivially by rule PICK-CONSTR-NIL. Moreover, we have with (B.5.19) and (B.5.35) that

  $$\varphi U' = \varphi[\overline{V/Z}, \overline{V''/Z'}]U'' = \varphi[\overline{V''/Z'}]U'' = \varphi U$$

- *Case* $U'' \in \overline{Z}$: Then $U'' = Z_i$ for some $i \in [l]$ by criterion WF-IFACE-3. By (B.5.37) we have $p^? = i$. Moreover,

  $$i \notin \mathsf{pol}^-(I) \tag{B.5.40}$$

  In the following, we use the notation $\mathsf{impl}(\mathcal{R}, q)$ to denote the qth implementing type of $\mathcal{R}$; that is, $\mathsf{impl}(\overline{T}\, \textbf{implements}\, K, q) := T_q$.

  Case distinction on whether or not $V^?_i = \mathsf{nil}$.

  - *Case* $V^?_i = \mathsf{nil}$: By (B.5.24) $\mathscr{V}_i = \emptyset$, so we get by (B.5.23) and the definition of $\mathsf{sresolve}$ that $Z_i \notin \overline{U''}$ and $i \neq j$. Thus, it is easy to verify that $i \notin \mathsf{disp}(I)$. With (B.5.40) then $i \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$. Hence, for all $\mathcal{R} \in \mathscr{M}$, $\mathsf{impl}(\mathcal{R}, i) = T'_i$ by (B.5.36). By rule PICK-CONSTR-NON-NIL we get (B.5.38). Obviously, $\overline{V}\, \textbf{implements}\, I$<$\overline{V''}$> $\in \mathscr{M}$, so $V_i = T'_i$. With (B.5.19), (B.5.39), and the fact $U'' = Z_i$ then

    $$\varphi U' = \varphi[\overline{V/Z}, \overline{V''/Z'}]U'' = \varphi V_i = \varphi T'_i = \varphi U$$

- *Case* $V_i^? \neq \mathsf{nil}$: Because of (B.5.40) we have by (B.5.20) and (B.5.24)

$$i \in \mathscr{N}_2 \tag{B.5.41}$$
$$V_i^? = T_i'$$
$$\mathscr{V}_i \neq \emptyset \tag{B.5.42}$$

Suppose $\mathcal{R} \in \mathscr{M}$. By (B.5.27) and (B.5.35)

$$\mathcal{R} = \ldots\ \textbf{implements}\ I{<}\overline{W}{>}$$

With (B.5.27), Theorem 3.28, and Lemma B.5.4

$$\Delta \Vdash \mathcal{R}$$
$$\Delta \vdash_{\mathrm{q}}{}'\ V_{i,\mathcal{R}} \leq \mathsf{impl}(\mathcal{R}, i) \text{ for some } V_{i,\mathcal{R}} \in \mathscr{V}_i \tag{B.5.43}$$

Next, we show that

$$\mathsf{impl}(\mathcal{R}, i) = G_{i,\mathcal{R}} \tag{B.5.44}$$

for some $G_{i,\mathcal{R}}$. Assume that this is not the case; that is, $\mathsf{impl}(\mathcal{R}, i)$ is an interface type. Because of (B.5.40) we get by Lemma B.1.32

$$[l] = \{1\}$$
$$\mathcal{R} = J{<}\overline{W''}{>}\ \textbf{implements}\ I{<}\overline{W}{>} \tag{B.5.45}$$
$$J{<}\overline{W''}{>} \trianglelefteq_{\mathrm{i}} I{<}\overline{W}{>} \tag{B.5.46}$$
$$1 \in \mathsf{pol}^{+}(I)$$
$$1 \in \mathsf{pol}^{+}(J)$$

Hence, $i = j = 1$. Because $1 \in \mathsf{pol}^{+}(I)$ we have $Z_i \notin \mathsf{ftv}(\overline{U''})$. With (B.5.41) and (B.5.21) $T' = G$ for some G, so we have with (B.5.23) and the definition of $\mathsf{sresolve}$ that $\mathscr{V}_i = \{G\}$. With (B.5.43) and (B.5.45) then

$$\Delta \vdash_{\mathrm{q}}{}'\ G \leq J{<}\overline{W''}{>}$$

By Lemma B.1.10 we then have $G = X$ for some X. Thus, by Lemma B.5.24

$$1 \in \mathsf{pol}^{-}(J)$$

With (B.5.46) and Lemma B.1.18 then also $1 \in \mathsf{pol}^{-}(I)$, which is a contradiction to (B.5.40). This finishes the proof of (B.5.44).

Our next goal is to prove that there exists some $\mathcal{R}' \in \mathscr{M}$ such that

$$\Delta \vdash_{\mathrm{q}}{}'\ \mathsf{impl}(\mathcal{R}', i) \leq \mathsf{impl}(\mathcal{R}, i) \tag{B.5.47}$$

for all $\mathcal{R} \in \mathscr{M}$.

Together with (B.5.32), we then use rule PICK-CONSTR-NON-NIL to derive (B.5.38), yielding

$$\mathsf{pick\text{-}constr}^{p^?}_{\Delta}\, \mathscr{M} = \overline{V}\ \textbf{implements}\ I{<}\overline{V''}{>} = \mathcal{R}' \tag{B.5.48}$$

W.l.o.g., assume that $\mathsf{impl}(\mathcal{R}, i) \neq \mathit{Object}$ for all $\mathcal{R} \in \mathscr{M}$. (If $\mathsf{impl}(\mathcal{R}, i) = \mathit{Object}$ then (B.5.47) holds trivially for this $\mathcal{R}$.) Hence, we have with (B.5.44)

$$\mathsf{impl}(\mathcal{R}, i) = G_{i,\mathcal{R}} \neq \mathit{Object}$$

With (B.5.43), Lemma B.1.10, and Lemma B.5.12 we then get

$$\mathscr{V}_i \ni V_{i,\mathcal{R}} = H_{i,\mathcal{R}} \neq \mathit{Object} \tag{B.5.49}$$

By (B.5.23) and the definition of $\mathsf{sresolve}$

$$\mathscr{V}_i = \mathsf{mub}_\Delta \underbrace{\left(\{T_q \mid q \in [n], U''_q = Z_i\} \cup (\text{if } i = j \text{ then } \{T'\} \text{ else } \emptyset)\right)}_{=:\mathscr{T}}$$

Hence, because $V_{i,\mathcal{R}} \in \mathscr{V}_i$, we have with Lemma B.5.16

$$\Delta \vdash_{\mathrm{q}}' T_q \leq V_{i,\mathcal{R}} \text{ for all } q \in [n], U''_q = Z_i$$
$$\Delta \vdash_{\mathrm{q}}' T' \leq V_{i,\mathcal{R}} \text{ if } i = j$$

By (B.5.23), (B.5.42), and the definition of $\mathsf{sresolve}$, we get $\mathscr{T} \neq \emptyset$. By (B.5.43), (B.5.49), and Lemma B.5.17, we get that there exists $V_i \in \mathscr{V}_i$ such that $V_i = V_{i,\mathcal{R}}$ for all $\mathcal{R} \in \mathscr{M}$. Hence, with (B.5.43)

$$\Delta \vdash_{\mathrm{q}}' V_i \leq \mathsf{impl}(\mathcal{R}, i) \tag{B.5.50}$$

for all $\mathcal{R} \in \mathscr{M}$. Now suppose $\mathcal{R}_1, \mathcal{R}_2 \in \mathscr{M}$. We then have $\Delta \vdash_{\mathrm{q}}' V_i \leq \mathsf{impl}(\mathcal{R}_1, i)$ and $\Delta \vdash_{\mathrm{q}}' V_i \leq \mathsf{impl}(\mathcal{R}_2, i)$, so with Lemma B.5.7 and (B.5.44)

$$\Delta \vdash_{\mathrm{q}}' \mathsf{impl}(\mathcal{R}_1, i) \leq \mathsf{impl}(\mathcal{R}_2, i) \text{ or } \Delta \vdash_{\mathrm{q}}' \mathsf{impl}(\mathcal{R}_2, i) \leq \mathsf{impl}(\mathcal{R}_1, i)$$

But with (B.5.50) and Lemma B.5.13, we know that the set $\{\mathsf{impl}(\mathcal{R}, i) \mid \mathcal{R} \in \mathscr{M}\}$ is finite. Thus, there exists some $\mathcal{R}' \in \mathscr{M}$ such that $\Delta \vdash_{\mathrm{q}}' \mathsf{impl}(\mathcal{R}', i) \leq \mathsf{impl}(\mathcal{R}, i)$. This finishes the proof of (B.5.47) and thus the proof of (B.5.38).

Finally, we prove $\Delta \vdash \varphi U' \leq \varphi U$. With (B.5.31) we have some $\mathcal{R}'' \in \mathscr{M}$ such that

$$\mathsf{impl}(\mathcal{R}'', i) = W'_i \overset{\text{(B.5.30),(B.5.40)}}{=} T'''_i \overset{\text{(B.5.28)}}{=} V^?_i \overset{\text{(B.5.41),(B.5.24)}}{=} T'_i$$

By (B.5.47) then

$$\Delta \vdash_{\mathrm{q}}' \mathsf{impl}(\mathcal{R}', i) \leq T'_i \tag{B.5.51}$$

We also have (note $U'' = Z_i$)

$$U' \overset{\text{(B.5.39)}}{=} [\overline{V/Z}, \overline{V''/Z'}]U'' = [\overline{V/Z}, \overline{V''/Z'}]Z_i = V_i$$
$$\overset{\text{(B.5.48)}}{=} \mathsf{impl}(\mathcal{R}', i)U \overset{\text{(B.5.19)}}{=} T'_i$$

W.l.o.g., $\overline{X} \cap \mathsf{ftv}(\overline{V}) = \emptyset = \overline{X} \cap \mathsf{ftv}(\overline{T'})$. Thus, with (B.5.51), $\Delta \vdash \varphi U' \leq \varphi U$, as required.

End case distinction on whether or not $V^?_i = \mathsf{nil}$.

End case distinction on whether or not $U'' \in \overline{Z}$.

– *Case* second possibility left by Lemma B.1.32:

$$
\begin{aligned}
[l] &= \{1\} \\
1 &\in \mathsf{pol}^{+}(I) \\
T = T_1' &= K && \text{(B.5.52)} \\
K &\trianglelefteq_{\mathrm{i}} I{<}\overline{W}{>} && \text{(B.5.53)}
\end{aligned}
$$

(By abuse of notation, we identify $\mathsf{pol}(K)$ with $\mathsf{pol}(J)$ for $K = J{<}\overline{T}{>}$.) Because $1 \in \mathsf{pol}^{+}(I)$ we have

$$Z_1 \notin \mathsf{ftv}(\overline{U''}) \tag{B.5.54}$$

Define

$$
\begin{aligned}
\mathscr{V}_1 &= \mathsf{sresolve}_{\Delta;Z_1}(Z_1\,\overline{U''}, T'\,\overline{T}) = \mathsf{mub}_{\Delta}\{T'\} = \{T'\} \\
p^{?} &= (\text{if } U'' = Z_1 \text{ then } 1 \text{ else } \mathsf{nil}) \\
\mathscr{M} &= \{V\,\mathbf{implements}\,I{<}\overline{V''}{>} \mid V' \in \mathscr{V}_1, \Delta \vdash_{\mathrm{q}}{}' V' \leq V, && \text{(B.5.55)} \\
&\qquad\qquad \Delta \Vdash_{\mathrm{a}}^{?} V\,\mathbf{implements}\,I{<}\overline{\mathsf{nil}}{>} \rightarrowtriangle V\,\mathbf{implements}\,I{<}\overline{V''}{>}\} \\
&\quad\ \{V\,\mathbf{implements}\,I{<}\overline{V''}{>} \mid \Delta \vdash_{\mathrm{q}}{}' T' \leq V, \\
&\qquad\qquad \Delta \Vdash_{\mathrm{a}}^{?} V\,\mathbf{implements}\,I{<}\overline{\mathsf{nil}}{>} \rightarrowtriangle V\,\mathbf{implements}\,I{<}\overline{V''}{>}\}
\end{aligned}
$$

We now prove that there exists some T'' such that

$$
\begin{aligned}
T''\,\mathbf{implements}\,I{<}\overline{W}{>} &\in \mathscr{M} && \text{(B.5.56)} \\
\Delta \vdash T'' &\leq K && \text{(B.5.57)}
\end{aligned}
$$

From the assumption $\Delta \vdash T' \leq T$ and $T = K$ we get $\Delta \vdash_{\mathrm{a}} T' \leq K$.

Case distinction on whether or not $\Delta \vdash_{\mathrm{q}}{}' T' \leq K$.

- *Case* $\Delta \vdash_{\mathrm{q}}{}' T' \leq K$: From (B.5.18) we get

  $$\Delta \Vdash_{\mathrm{a}} K\,\mathbf{implements}\,I{<}\overline{W}{>}$$

  so with Theorem 3.29 we get that $K\,\mathbf{implements}\,I{<}\overline{W}{>} \in \mathscr{M}$. The claims (B.5.56) and (B.5.57) then follow for $T'' = K$.

- *Case* not $\Delta \vdash_{\mathrm{q}}{}' T' \leq K$: Hence, by inverting rule SUB-Q-ALG-IMPL,

  $$
  \begin{aligned}
  \Delta \vdash_{\mathrm{q}}{}' T' &\leq T'' \\
  \Delta \Vdash_{\mathrm{q}}{}' T''\,&\mathbf{implements}\,K && \text{(B.5.58)}
  \end{aligned}
  $$

  By rule SUB-IMPL then $\Delta \vdash T'' \leq K$. This proves (B.5.57). With (B.5.53), Lemma B.1.2, and Lemma B.1.27, we get

  $$\Delta \Vdash_{\mathrm{a}} T''\,\mathbf{implements}\,I{<}\overline{W}{>}$$

  With Theorem 3.29 and (B.5.55) then

  $$T''\,\mathbf{implements}\,I{<}\overline{W}{>} \in \mathscr{M}$$

  This proves (B.5.56).

End case distinction on whether or not $\Delta \vdash_{\mathrm{q}}{}' T' \leq K$.
This finishes the proof of (B.5.56) and (B.5.57).
Let $\mathcal{R} \in \mathscr{M}$. By (B.5.55) and Theorem 3.28:

$$\Delta \Vdash_{\mathrm{a}} \mathcal{R} \tag{B.5.59}$$

$$\Delta \vdash_{\mathrm{q}}{}' T' \leq \mathsf{impl}(\mathcal{R}, 1) \tag{B.5.60}$$

Moreover, we have with Lemma B.5.10, (B.5.56), and (B.5.55) that

$$\mathcal{R} = V_{\mathcal{R}}\ \mathbf{implements}\ I\text{<}\overline{W}\text{>} \tag{B.5.61}$$

for some $V_{\mathcal{R}}$.
Case distinction on the form of $p^?$.

- ∗ *Case* $p^? = \mathsf{nil}$: Then $U'' \neq Z_1$. By criterion WF-IFACE-3

  $$\overline{Z} \cap \mathsf{ftv}(U'') = \emptyset$$
  $$\overline{Z} \cap \mathsf{ftv}(\overline{P''}) = \emptyset$$

  Moreover, by (B.5.56) we know that $\mathscr{M} \neq \emptyset$, so with (B.5.61)

  $$\mathsf{pick\text{-}constr}^{p^?}_{\Delta}\,\mathscr{M} = V\ \mathbf{implements}\ I\text{<}\overline{W}\text{>}$$

  for some $V\ \mathbf{implements}\ I\text{<}\overline{W}\text{>} \in \mathscr{M}$. We have by rule ALG-MTYPE-IFACE and (B.5.54)

  $$\begin{aligned}\mathsf{a\text{-}mtype}_{\Delta}(m, T', \overline{T}) &= [\overline{W/Z'}]\mathit{msig}_k \\ &= [\overline{T'/Z}, \overline{W/Z'}]\mathit{msig}_k \\ &\overset{\text{(B.5.19)}}{=} \text{<}\overline{X}\text{>}\,\overline{U\;x}^{n} \to U\ \mathbf{where}\ \mathcal{P}\end{aligned}$$

  as required.
- ∗ *Case* $p^? \neq \mathsf{nil}$: Then $p^? = 1$ and $U'' = Z_1$. Hence

  $$1 \notin \mathsf{pol}^{-}(I) \tag{B.5.62}$$

  We now prove that there exists some $\mathcal{R}' \in \mathscr{M}$ such that

  $$\Delta \vdash_{\mathrm{q}}{}' \mathsf{impl}(\mathcal{R}', 1) \leq \mathsf{impl}(\mathcal{R}, 1) \text{ for all } \mathcal{R} \in \mathscr{M} \tag{B.5.63}$$

  In the following, we assume w.l.o.g. that $\mathsf{impl}(\mathcal{R}, 1) \neq \mathit{Object}$ for all $\mathcal{R} \in \mathscr{M}$. (If $\mathsf{impl}(\mathcal{R}, 1) = \mathit{Object}$ then (B.5.63) holds trivially for this $\mathcal{R}$.) We proceed by case distinction on the existence of L and $\mathcal{R}' \in \mathscr{M}$ with $\mathsf{impl}(\mathcal{R}', 1) = L$

  Case distinction on the existence of L and $\mathcal{R}' \in \mathscr{M}$.

  - · *Case* there exists $\mathcal{R}' \in \mathscr{M}$ with $\mathsf{impl}(\mathcal{R}', 1) = L$ for some L: Then we have $\Delta \Vdash_{\mathrm{q}} L\ \mathbf{implements}\ I\text{<}\overline{W}\text{>}$ by (B.5.59). Hence, Lemma B.1.32 and (B.5.62) give us that $L \trianglelefteq_{\mathrm{i}} I\text{<}\overline{W}\text{>}$, so with (B.5.60) and Lemma B.1.7 we have

    $$\Delta \vdash_{\mathrm{q}}{}' T' \leq I\text{<}\overline{W}\text{>}$$

    If $T' = X$ then, by Lemma B.5.24, $1 \in \mathsf{pol}^{-}(I)$, which is a contradiction to (B.5.62). If $T' = N$ then, by Lemma B.5.24, $1 \in \mathsf{pol}^{-}(I)$, which is a contradiction to (B.5.62). Finally, we consider the case where $T' = K'$. Because $\mathsf{impl}(\mathcal{R}, 1) \neq \mathit{Object}$ for all $\mathcal{R} \in \mathscr{M}$, we have with (B.5.60) and Lemma B.1.10 that for all $\mathcal{R} \in \mathscr{M}$:

$$\begin{aligned} \mathsf{impl}(\mathcal{R}, 1) &= L_{\mathcal{R}} \text{ for some } L_{\mathcal{R}} \\ K' &\trianglelefteq_{\mathrm{i}} L_{\mathcal{R}} \end{aligned} \tag{B.5.64}$$

With (B.5.61), (B.5.59), (B.5.62), and Lemma B.1.32 we get

$$\begin{gathered} L_{\mathcal{R}} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{W}\texttt{>} \\ 1 \in \mathsf{pol}^{+}(L_{\mathcal{R}}) \end{gathered}$$

With Lemma B.1.4 then

$$K' \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{W}\texttt{>}$$

Now assume $1 \in \mathsf{pol}^{+}(K')$. Then

$$\Delta \Vdash_{\mathrm{q}} K' \,\mathbf{implements}\, I\texttt{<}\overline{W}\texttt{>}$$

by rule ENT-Q-ALG-ENV and Lemma B.1.17. Hence, with (B.5.55) and Theorem 3.29

$$K' \,\mathbf{implements}\, I\texttt{<}\overline{W}\texttt{>} \in \mathscr{M}$$

With (B.5.64), we have $\Delta \vdash_{\mathrm{q}}' K' \leq L_{\mathcal{R}}$ for all $\mathcal{R} \in \mathscr{M}$, so (B.5.63) holds.

On the other hand, assume $1 \notin \mathsf{pol}^{+}(K')$. Because of (B.5.62), we get with Lemma B.1.18 that $1 \notin \mathsf{pol}^{-}(K')$. With (B.5.64) and criterion WF-PROG-7 we then have for all $\mathcal{R}_1, \mathcal{R}_2 \in \mathscr{M}$:

$$L_{\mathcal{R}_1} \trianglelefteq_{\mathrm{i}} L_{\mathcal{R}_2} \text{ or } L_{\mathcal{R}_2} \trianglelefteq_{\mathrm{i}} L_{\mathcal{R}_1}$$

With (B.5.60) and Lemma B.5.13, we know that the set $\{\mathsf{impl}(\mathcal{R}, 1) \mid \mathcal{R} \in \mathscr{M}\}$ is finite. Thus, (B.5.63) holds.

- *Case* there does not exist $\mathcal{R}' \in \mathscr{M}$ with $\mathsf{impl}(\mathcal{R}', 1) = L$ for some L: With (B.5.60) and Lemma B.5.7 we have for all $\mathcal{R}_1, \mathcal{R}_2 \in \mathscr{M}$:

$$L_{\mathcal{R}_1} \trianglelefteq_{\mathrm{i}} L_{\mathcal{R}_2} \text{ or } L_{\mathcal{R}_2} \trianglelefteq_{\mathrm{i}} L_{\mathcal{R}_1}$$

Thus, (B.5.63) holds.

End case distinction on the existence of L and $\mathcal{R}' \in \mathscr{M}$.

This finishes the proof of (B.5.63).

We now use rule PICK-CONSTR-NON-NIL to derive

$$\mathsf{pick\text{-}constr}_{\Delta}^{p^?} \mathscr{M} = \mathcal{R}'$$

such that $\Delta \vdash_{\mathrm{q}}' \mathsf{impl}(\mathcal{R}', 1) \leq \mathsf{impl}(\mathcal{R}, 1)$ for all $\mathcal{R} \in \mathscr{M}$. We now have by ALG-MTYPE-IFACE (note that $U'' = Z_1$ and, by criterion WF-IFACE-3, $\overline{Z} \cap \mathsf{ftv}(\overline{P''}) = \emptyset$)

$$\begin{aligned} &\mathsf{a\text{-}mtype}_{\Delta}(m, T', \overline{T}) \\ &= [\mathsf{impl}(\mathcal{R}', 1)/Z_1, \overline{W/Z'}](\texttt{<}\overline{X}\texttt{>}\,\overline{U''\,x} \to U'' \;\mathbf{where}\; \overline{P''}) \\ &\overset{(\text{B.5.19}),(\text{B.5.54})}{=} \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to \mathsf{impl}(\mathcal{R}', 1) \;\mathbf{where}\; \overline{P} \end{aligned}$$

Define $U' = \mathsf{impl}(\mathcal{R}', 1)$. With (B.5.56), (B.5.57), and (B.5.63) we have

$$\Delta \vdash_{\mathsf{a}} \mathsf{impl}(\mathcal{R}', 1) \leq K$$

By (B.5.19) and (B.5.52) we have

$$U = T_1' = T = K$$

Hence,

$$\Delta \vdash U' \leq U$$

W.l.o.g., $\overline{X} \cap \mathsf{ftv}(\overline{T'}, \mathcal{R}') = \emptyset$. Hence

$$\Delta \vdash \varphi U' \leq \varphi U$$

as required.

End case distinction on the form of $p^?$.

End case distinction on the possibilities left by Lemma B.1.32.

End case distinction on the form of m. □

B.5.5 Proof of Theorem 3.35

Theorem 3.35 states that algorithmic expression typing is sound with respect to its declarative specification in Figure 3.9. All proofs in this section apply the equivalences and implications of Corollary B.5.2 implicitly.

Lemma B.5.26.

(i) $\Delta \vdash T\ \mathsf{ok}$ *if, and only if,* $\Delta \vdash_{\mathsf{a}} T\ \mathsf{ok}$.

(ii) $\Delta \vdash \mathcal{P}\ \mathsf{ok}$ *if, and only if,* $\Delta \vdash_{\mathsf{a}} \mathcal{P}\ \mathsf{ok}$.

Proof. Follows by straightforward induction on the combined size of the given derivations. □

From now on, we use Lemma B.5.26 implicitly.

Lemma B.5.27. *If* $\Delta \vdash \mathcal{R}\ \mathsf{ok}$ *and* $\mathcal{S} \in \mathsf{sup}(\mathcal{R})$ *then* $\Delta \vdash \mathcal{S}\ \mathsf{ok}$.

Proof. We proceed by induction on the derivation of $\mathcal{S} \in \mathsf{sup}(\mathcal{R})$. If this derivation ends with rule SUP-REFL, then the claim holds trivially. Otherwise, we have

$$\frac{\textbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\ [\overline{Y}\ \textbf{where}\ \overline{S}]\ \textbf{where}\ \overline{P}\ \ldots \qquad \overline{U}\ \textbf{implements}\ I\texttt{<}\overline{V}\texttt{>} \in \mathsf{sup}(\mathcal{R})}{\underbrace{[\overline{V/X}, \overline{U/Y}]S_j}_{=\mathcal{S}} \in \mathsf{sup}(\mathcal{R})}$$

By the I.H., we have $\Delta \vdash \overline{U}\ \textbf{implements}\ I\texttt{<}\overline{V}\texttt{>}\ \mathsf{ok}$. This derivation must end with OK-IMPL-CONSTR. Inverting the rule then yields

$$\Delta \Vdash [\overline{V/X}, \overline{U/Y}]\overline{S}, \overline{P}$$
$$\Delta \vdash \overline{U}, \overline{V}\ \mathsf{ok}$$

The underlying program is well-typed, so we have $\overline{S}, \overline{P}, \overline{X}, \overline{Y} \vdash S_j\ \mathsf{ok}$. With Lemma B.2.24 then $\Delta \vdash \mathcal{S}\ \mathsf{ok}$. □

Lemma B.5.28. *Assume* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash \overline{T}\ \mathsf{ok}$. *If* $\Delta \Vdash_{\mathrm{q}} \overline{T}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}$ *then* $\Delta \vdash \overline{T}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}\ \mathsf{ok}$.

Proof. We have

$$\textsc{ent-q-alg-up}\ \frac{\begin{array}{c}(\forall i)\ \Delta \vdash_{\mathrm{q}}{}'\ T_i \leq U_i \\ (\forall i)\ \text{if } T_i \neq U_i \text{ then } i \in \mathsf{pol}^-(I) \qquad \Delta \Vdash_{\mathrm{q}}{}'\ \overline{U}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}\end{array}}{\Delta \Vdash_{\mathrm{q}} \overline{T}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}}$$

By Lemma B.5.22

$$\Delta \vdash \overline{U}\ \mathsf{ok} \tag{B.5.65}$$

Case distinction on the last rule of the derivation of $\Delta \Vdash_{\mathrm{q}}{}'\ \overline{U}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}$.

- *Case* rule ENT-Q-ALG-ENV: Then $R \in \Delta$ and $\overline{U}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>} \in \mathsf{sup}(R)$. With $\vdash \Delta\ \mathsf{ok}$ we have $\Delta \vdash R\ \mathsf{ok}$. By Lemma B.5.27 we have

$$\Delta \vdash \overline{U}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}\ \mathsf{ok}$$

- *Case* rule ENT-Q-ALG-IMPL: Then

$$\frac{\mathbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V'}\texttt{>}\ [\,\overline{N}\,]\ \mathbf{where}\ \overline{P}\ \ldots \qquad \Delta \Vdash_{\mathrm{q}} [\overline{W/X}]\overline{P}}{\Delta \Vdash_{\mathrm{q}} \underbrace{[\overline{W/X}](\overline{N}\ \mathbf{implements}\ I\texttt{<}\overline{V'}\texttt{>})}_{=\overline{U}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}}}$$

Because the underlying program is well-typed, we have

$$\overline{P}, \overline{X} \vdash \overline{N}\ \mathbf{implements}\ I\texttt{<}\overline{V'}\texttt{>}\ \mathsf{ok}$$

Moreover, with (B.5.65) $\Delta \vdash [\overline{W/X}]\overline{N}\ \mathsf{ok}$ and by criterion WF-IMPL-2 $\overline{X} \subseteq \mathsf{ftv}(\overline{N})$. Hence, with Lemma B.2.21, $\Delta \vdash \overline{W}\ \mathsf{ok}$. Thus, with Lemma B.2.24

$$\Delta \vdash [\overline{W/X}](\overline{N}\ \mathbf{implements}\ I\texttt{<}\overline{V'}\texttt{>})\ \mathsf{ok}$$

- *Case* rule ENT-Q-ALG-IFACE: Then

$$\frac{1 \in \mathsf{pol}^+(J) \qquad \mathsf{non\text{-}static}(J) \qquad J\texttt{<}\overline{W}\texttt{>} \trianglelefteq_{\mathrm{i}} I\texttt{<}\overline{V}\texttt{>}}{\Delta \Vdash_{\mathrm{q}}{}'\ \underbrace{J\texttt{<}\overline{W}\texttt{>}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}}_{=\overline{U}\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}}}$$

From $\Delta \vdash J\texttt{<}\overline{W}\texttt{>}\ \mathsf{ok}$ and Lemma B.5.21 we have

$$\Delta \vdash I\texttt{<}\overline{V}\texttt{>}\ \mathsf{ok} \tag{B.5.66}$$

Assume

$$\mathbf{interface}\ I\texttt{<}\overline{X}\texttt{>}\ [Y\ \mathbf{where}\ \overline{R}]\ \mathbf{where}\ \overline{P}\ldots \tag{B.5.67}$$

From (B.5.66) then

$$\Delta, Y\ \mathbf{implements}\ I\texttt{<}\overline{V}\texttt{>}, Y \Vdash [\overline{V/X}]\overline{R}, \overline{P} \tag{B.5.68}$$

$$Y \notin \mathsf{ftv}(\Delta, \overline{V})$$

With $\Delta \Vdash_{\mathrm{q}}{}' \; J{<}\overline{W}{>}\,\textbf{implements}\,I{<}\overline{V}{>}$ we have $\Delta \Vdash J{<}\overline{W}{>}\,\textbf{implements}\,I{<}\overline{V}{>}$ by Corollary B.5.2. Hence,

$$\Delta \Vdash [J{<}\overline{W}{>}/Y](\Delta, Y\,\textbf{implements}\,I{<}\overline{V}{>}, Y)$$

Thus, with Corollary B.1.28 applied to (B.5.68)

$$\Delta \Vdash \underbrace{[J{<}\overline{W}{>}/Y][\overline{V/X}]\overline{R}, \overline{P}}_{=[J{<}\overline{W}{>}/Y,\overline{V/X}]\overline{R},\overline{P}} \tag{B.5.69}$$

We then have with $\Delta \vdash J{<}\overline{W}{>}\ \mathsf{ok}$, (B.5.66), (B.5.67), (B.5.69), and an application of rule OK-IMPL-CONSTR that

$$\Delta \vdash J{<}\overline{W}{>}\,\textbf{implements}\,I{<}\overline{V}{>}\ \mathsf{ok}$$

End case distinction on the last rule of the derivation of $\Delta \Vdash_{\mathrm{q}}{}' \; \overline{U}\,\textbf{implements}\,I{<}\overline{V}{>}$. Let

$$\textbf{interface}\ I{<}\overline{X}{>}\,[\overline{Y}\,\textbf{where}\,\overline{R}]\,\textbf{where}\,\overline{P}\ldots$$

Because we just proved that $\Delta \vdash \overline{U}\,\textbf{implements}\,I{<}\overline{V}{>}\ \mathsf{ok}$, we have

$$\Delta \vdash \overline{V}\ \mathsf{ok}$$
$$\Delta \Vdash [\overline{V/X}, \overline{U/Y}]\overline{R}, \overline{P}$$

We now prove by induction on the number of indices i with $T_i \neq U_i$ that $\Delta \Vdash [\overline{V/X}, \overline{T/Y}]\overline{R}, \overline{P}$. The original claim then follows with rule OK-IMPL-CONSTR.

- Assume there are no indices i with $T_i \neq U_i$. Then $\Delta \Vdash [\overline{V/X}, \overline{T/Y}]\overline{R}, \overline{P}$ holds trivially.
- Assume i such that $T_i \neq U_i$. The I.H. then gives us that $\Delta \Vdash [\overline{V/X}, \overline{T'/Y}]\overline{R}, \overline{P}$ where

$$T'_j = \begin{cases} T_j & \text{if } i \neq j \\ U_j & \text{if } i = j \end{cases}$$

From $T_i \neq U_i$ we have $i \in \mathsf{pol}^-(I)$. Hence $Y_i \notin \mathsf{ftv}(\overline{P})$. Thus,

$$\Delta \Vdash [\overline{W/X}, \overline{T/Y}]\overline{P}$$

Now suppose $Y_i \in \mathsf{ftv}(\overline{G}\,\textbf{implements}\,J'{<}\overline{W'}{>})$ for some $\overline{G}\,\textbf{implements}\,J'{<}\overline{W'}{>} \in \overline{R}$. Then we have with $i \in \mathsf{pol}^-(I)$ and well-formedness criteria WF-IFACE-2 that $Y_i \notin \mathsf{ftv}(\overline{W'})$ and that $Y_i \in \mathsf{ftv}(G_j)$ implies $Y_i = G_j$ and $j \in \mathsf{pol}^-(J')$. Hence, with $\Delta \vdash_{\mathrm{q}}{}' \; T_i \leq U_i$ and (possibly) some applications of rule ENT-UP, we also get $\Delta \Vdash [\overline{W/X}, \overline{T/Y}]\overline{R}$, as required. □

Lemma B.5.29. *Assume*

$$\textbf{interface}\ I{<}\overline{Z}{>}\,[\,\overline{Y}\,\textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{Q}\,\{\,\overline{m : \textbf{static}\,\mathit{msig}}\ \overline{\mathit{rcsig}}\,\}$$
$$\mathit{msig} = {<}\overline{X}{>}\overline{U\,x} \to U\ \textbf{where}\ \overline{P}$$
$$\Delta \Vdash \overline{T}\,\textbf{implements}\,I{<}\overline{W}{>}$$
$$\Delta \Vdash [\overline{V/X}][\overline{T/Y}, \overline{W/Z}]\overline{P}$$
$$\Delta \vdash \overline{T}, \overline{V}\ \mathsf{ok}$$

such that either $\mathit{msig} \in \overline{\mathit{msig}}$ *or that there exists* **receiver** $\{\overline{m' : \mathit{msig'}}\} \in \overline{\mathit{rcsig}}$ *with* $\mathit{msig} \in \overline{\mathit{msig'}}$. *Then* $\Delta \vdash [\overline{V/X}][\overline{T/Y}, \overline{W/Z}]U\ \mathsf{ok}$.

Proof. We get with Lemma B.5.28 and the assumptions $\Delta \Vdash \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{W}\texttt{>}$ and $\Delta \vdash \overline{T}\ \mathsf{ok}$ that

$$\Delta \vdash \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{W}\texttt{>}\ \mathsf{ok}$$

Hence,

$$\begin{gathered}\Delta \vdash \overline{W}\ \mathsf{ok}\\ \Delta \Vdash [\overline{T/Y}, \overline{W/Z}]\overline{R}, \overline{Q} \qquad \text{(B.5.70)}\end{gathered}$$

Because the underlying program is well-typed, we have

$$\overline{R}, \overline{Q}, \overline{Y}, \overline{Z}, \overline{P}, \overline{X} \vdash U\ \mathsf{ok} \tag{B.5.71}$$

W.l.o.g., $\overline{X} \cap \mathsf{ftv}(\overline{R}, \overline{Q}, \overline{T}, \overline{W}) = \emptyset$. Hence,

$$[\overline{T/Y}, \overline{W/Z}]\overline{R}, \overline{Q} = [\overline{V/X}, \overline{T/Y}, \overline{W/Z}]\overline{R}, \overline{Q} \tag{B.5.72}$$

$$[\overline{V/X}][\overline{T/Y}, \overline{W/Z}]\overline{P} = [\overline{V/X}, \overline{T/Y}, \overline{W/Z}]\overline{P} \tag{B.5.73}$$

$$[\overline{V/X}][\overline{T/Y}, \overline{W/Z}]U = [\overline{V/X}, \overline{T/Y}, \overline{W/Z}]U \tag{B.5.74}$$

With (B.5.72) and (B.5.70) we then have

$$\Delta \Vdash [\overline{V/X}, \overline{T/Y}, \overline{W/Z}]\overline{R}, \overline{Q}$$

With (B.5.73) and the assumption $\Delta \Vdash [\overline{V/X}][\overline{T/Y}, \overline{W/Z}]\overline{P}$ we have

$$\Delta \Vdash [\overline{V/X}, \overline{T/Y}, \overline{W/Z}]\overline{P}$$

With Lemma B.2.24 and (B.5.71) we then have

$$\Delta \vdash [\overline{V/X}, \overline{T/Y}, \overline{W/Z}]U\ \mathsf{ok}$$

so the claim follows with (B.5.74). □

Lemma B.5.30. *Suppose* $\vdash \Delta\ \mathsf{ok}$ *and and* $\Delta \vdash T_j\ \mathsf{ok}$ *for all* $j \in \mathsf{disp}(I)$. *If*

$$\Delta \Vdash^{?}_{\mathrm{a}} \overline{T^{?}}\,\textbf{implements}\,I\texttt{<}\overline{V^{?}}\texttt{>} \rightarrowtriangle \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$$

and $T^{?}_{i} = \mathsf{nil}$ *then* $\Delta \vdash T_i\ \mathsf{ok}$.

Proof. We first note that

$$\Delta; \beta; J \vdash^{?}_{\mathrm{a}} \overline{T^{?}} \uparrow \overline{U} \rightarrowtriangle \overline{V} \text{ and } T_j = \mathsf{nil} \text{ imply } V_j = U_j. \tag{B.5.75}$$

$$\Delta; \beta; J \vdash^{?}_{\mathrm{a}} \overline{T^{?}} \uparrow \overline{U} \rightarrowtriangle \overline{V} \text{ implies } \Delta \vdash_{\mathrm{q}}{}' V_i \leq U_i \text{ for all } i. \tag{B.5.76}$$

Now we show that

> *If* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash T_j\ \mathsf{ok}$ *for all* $j \in \mathsf{disp}(I)$ *and* $\mathcal{D}{::}\Delta; \mathscr{G}; \beta \Vdash^{?}_{\mathrm{a}} \overline{T^{?}}^{n}\,\textbf{implements}\,I\texttt{<}\overline{V^{?}}\texttt{>} \rightarrowtriangle \overline{T}\,\textbf{implements}\,I\texttt{<}\overline{V}\texttt{>}$ *and* $T^{?}_{i} = \mathsf{nil}$ *then* $\Delta \vdash T_i\ \mathsf{ok}$.

Assume $T^{?}_{i} = \mathsf{nil}$. W.l.o.g., $i \notin \mathsf{disp}(I)$.
Case distinction on the last rule of $\mathcal{D}$.

- *Case* rule ENT-NIL-ALG-ENV: Then

$$\begin{gathered} R \in \Delta \\ \overline{G}\,\textbf{implements}\,I{<}\overline{V}{>} \in \mathsf{sup}(R) \\ \Delta; \beta; I \vdash^{?}_{\mathsf{a}} \overline{T^?} \uparrow \overline{G} \rightarrowtriangle \overline{T} \end{gathered}$$

  From the assumption $\vdash \Delta$ ok and Lemma B.5.27 we get

$$\Delta \vdash \overline{G}\,\textbf{implements}\,I{<}\overline{V}{>}\ \mathsf{ok}$$

  so $\Delta \vdash G_i$ ok. But with (B.5.75) we have $T_i = G_i$.

- *Case* rule ENT-NIL-ALG-IFACE$_1$: Impossible because $n = 1$ and $T_1 \neq \mathsf{nil}$ in this rule.
- *Case* rule ENT-NIL-ALG-IFACE$_2$: Impossible because $n = 1$ and $T_1 \neq \mathsf{nil}$ in this rule.
- *Case* rule ENT-NIL-ALG-IMPL: Then

$$\textbf{implementation}{<}\overline{X}{>}\ I{<}\overline{V'}{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

$$\Delta; \beta; I \vdash^{?}_{\mathsf{a}} \overline{T^?} \uparrow [\overline{U/X}]\overline{N} \rightarrowtriangle \overline{T} \tag{B.5.77}$$

$$\Delta; \mathscr{G} \cup \{[\overline{U/X}]\overline{N}\,\textbf{implements}\,I{<}[\overline{U/X}]\overline{V'}{>}\}; \texttt{false} \Vdash_{\mathsf{a}} [\overline{U/X}]\overline{P} \tag{B.5.78}$$

  With (B.5.77) and (B.5.76) we get

$$(\forall i)\ \Delta \vdash T_i \leq [\overline{U/X}]N_i$$

  Thus, if $j \in \mathsf{disp}(I)$ then $\Delta \vdash T_j$ ok by assumption, so with Lemma B.5.22

$$\Delta \vdash_{\mathsf{q}}{}' \ [\overline{U/X}]N_j\ \mathsf{ok}$$

  From criterion WF-IMPL-2 we get $\overline{X} \subseteq \mathsf{ftv}(\{N_j \mid j \in \mathsf{disp}(I)\})$. Thus, withLemma B.2.21,

$$\Delta \vdash \overline{U}\ \mathsf{ok}$$

  With Lemma B.4.3, (B.5.78), and rule ENT-Q-ALG-UP, we get

$$\Delta \Vdash_{\mathsf{q}} [\overline{U/X}]\overline{P}$$

  Because the underlying program is well-typed we have

$$\overline{P}, \overline{X} \vdash \overline{N}\,\textbf{implements}\,I{<}\overline{V'}{>}\ \mathsf{ok}$$

  Now Lemma B.2.24 yields

$$\Delta \vdash [\overline{U/X}](\overline{N}\,\textbf{implements}\,I{<}\overline{V'}{>})\ \mathsf{ok}$$

  Thus,

$$\Delta \vdash [\overline{U/X}]\overline{N}\ \mathsf{ok}$$

  But with (B.5.75) and (B.5.77) we have $T_i = [\overline{U/X}]N_i$.

End case distinction on the last rule of $\mathcal{D}$. □

Lemma B.5.31. *Suppose* $\vdash \Delta$ ok *and* $\Delta \vdash N, \overline{V}$ ok *and* $\Delta \Vdash [\overline{V/X}]\mathcal{P}$*. Then* $\mathsf{a\text{-}mtype}^{\mathsf{c}}(m, N) = {<}\overline{X}{>}\overline{U\,x} \to U$ **where** $\overline{\mathcal{P}}$ *implies* $\Delta \vdash [\overline{V/X}]U$ ok.

Proof. By induction on the derivation of $\textsf{a-mtype}^{\mathrm{c}}(m, N)$.
Case distinction on the last rule of the derivation of $\textsf{a-mtype}^{\mathrm{c}}(m, N)$.

- *Case* rule ALG-MTYPE-CLASS-BASE: Then

  $$\begin{gathered} N = C\texttt{<}\overline{T}\texttt{>} \\ \textbf{class } C\texttt{<}\overline{Y}\texttt{>} \textbf{ extends } M \textbf{ where } \overline{Q}\,\{\ldots\ \overline{m : \mathit{msig}\,\{e\}}\,\} \\ m = m_j \\ \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}} = [\overline{T/Y}]\mathit{msig}_j \end{gathered}$$

  Assume

  $$\mathit{msig}_j = \texttt{<}\overline{X}\texttt{>}\,\overline{U'\,x} \to U' \textbf{ where } \overline{P}$$

  Because the underlying program is well-typed, we have

  $$\overline{Q}, \overline{Y} \vdash m_j : \mathit{msig}_j\,\{e_j\}\ \textsf{ok in}\ C\texttt{<}\overline{Y}\texttt{>}$$

  Hence,

  $$\overline{Q}, \overline{Y}, \overline{P}, \overline{X} \vdash U'\ \textsf{ok} \tag{B.5.79}$$

  From $\Delta \vdash N\ \textsf{ok}$ we get $\Delta \Vdash [\overline{T/Y}]\overline{Q}$ and $\Delta \vdash \overline{T}\ \textsf{ok}$. W.l.o.g., $\overline{X} \cap \textsf{ftv}(\overline{T}, \overline{Q}) = \emptyset$. Hence, $[\overline{T/Y}]\overline{Q} = [\overline{V/X}, \overline{T/Y}]\overline{Q}$, so we have

  $$\Delta \Vdash [\overline{V/X}, \overline{T/Y}]\overline{Q}$$

  Moreover, the assumption $\Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}}$ can be written as

  $$\Delta \Vdash [\overline{V/X}, \overline{T/Y}]\overline{P}$$

  Using Lemma B.2.24 on (B.5.79) yields

  $$\Delta \vdash \underbrace{[\overline{V/X}, \overline{T/Y}]U'}_{=[\overline{V/X}]U}\ \textsf{ok}$$

  as required.

- *Case* rule ALG-MTYPE-CLASS-SUPER: Then

  $$\begin{gathered} \textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } M\ \ldots \\ \textsf{a-mtype}^{\mathrm{c}}(m, [\overline{T/X}]M) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}} \\ N = C\texttt{<}\overline{T}\texttt{>} \end{gathered}$$

  Then $N \trianglelefteq_{\mathsf{c}} [\overline{T/X}]M$, so we get with $\Delta \vdash N\ \textsf{ok}$ and Lemma B.2.25 that $\Delta \vdash [\overline{T/X}]M\ \textsf{ok}$. The claim now follows from the I.H.

End case distinction on the last rule of the derivation of $\textsf{a-mtype}^{\mathrm{c}}(m, N)$. □

Lemma B.5.32. *Suppose* $\vdash \Delta\ \textsf{ok}$ *and* $\Delta \vdash T, \overline{T}, \overline{V}\ \textsf{ok}$ *and* $\Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}}$. *If* $\textsf{a-mtype}_{\Delta}(m, T, \overline{T}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}}$ *then* $\Delta \vdash [\overline{V/X}]U\ \textsf{ok}$.

Proof. *Case distinction* on the rule used to derive $\textsf{a-mtype}_{\Delta}(m, T, \overline{T})$.

- *Case* rule ALG-MTYPE-CLASS: Then $\mathsf{bound}_\Delta(T) = N$. Moreover,

  $$\mathsf{a\text{-}mtype}^{\mathsf{c}}(m, N) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \rightarrow U\ \textbf{where}\ \overline{\mathcal{P}}$$

  With Lemma B.5.23 we have $\Delta \vdash N\ \mathsf{ok}$. The claim now follows with Lemma B.5.31.

- *Case* rule ALG-MTYPE-IFACE: Then

  $$\begin{gathered}
  \textbf{interface}\ I\texttt{<}\overline{Z'}\texttt{>}\,[\,\overline{Z}^{l}\ \textbf{where}\,\overline{R}\,]\ \textbf{where}\ \overline{P}\,\{\ldots\ \overline{rcsig}\,\} \\
  rcsig_j = \textbf{receiver}\ \{\overline{m : msig}\} \\
  msig_k = \texttt{<}\overline{X}\texttt{>}\,\overline{U'\,x} \rightarrow U'\ \textbf{where}\ \overline{Q} \\
  (\forall i \in [l], i \neq j)\ \mathsf{sresolve}_{\Delta;Z_i}(\overline{U}, \overline{T}) = \mathcal{V}_i \\
  \mathsf{sresolve}_{\Delta;Z_j}(Z_j\,\overline{U}, T\,\overline{T}) = \mathcal{V}_j \\
  p^{?} = (\text{if } U = Z_i \text{ for some } i \in [l] \text{ then } i \text{ else } \mathsf{nil}) \\
  \overline{W}\,\textbf{implements}\,I\texttt{<}\overline{W'}\texttt{>} = \\
  \begin{aligned}
  \mathsf{pick\text{-}constr}^{p^?}_{\Delta}\{\overline{V''}\,\textbf{implements}\,I\texttt{<}\overline{V'''}\texttt{>} \mid (\forall i \in [l])\ & \text{if } \mathcal{V}_i = \emptyset \text{ then } V_i^? = \mathsf{nil} \\
  & \text{else define } V_i^? \text{ such that} \\
  & \Delta \vdash_{\mathrm{q}}{}'\ V_i' \leq V_i^? \text{ for } V_i' \in \mathcal{V}_i, \\
  \Delta \Vdash^{?}_{\mathrm{a}}\ \overline{V^?}\,\textbf{implements}\,I\texttt{<}\overline{\mathsf{nil}}\texttt{>} \rightharpoonup\ & \overline{V''}\,\textbf{implements}\,I\texttt{<}\overline{V'''}\texttt{>}\}
  \end{aligned}
  \end{gathered}$$

  and

  $$\begin{gathered}
  m = m_k \\
  \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \rightarrow U\ \textbf{where}\ \overline{\mathcal{P}} = [\overline{W/Z}, \overline{W'/Z'}]msig_k
  \end{gathered}$$

  With Lemma B.5.22 and the assumption $\Delta \vdash T, \overline{T}\ \mathsf{ok}$ we easily verify that $\Delta \vdash V_i'\ \mathsf{ok}$ for all $V_i' \in \mathcal{V}_i$. Hence, we have with Lemma B.5.22 for the $V_i^?$ in the argument to $\mathsf{pick\text{-}constr}^{p^?}_{\Delta}$ that

  $$V_i^? \neq \mathsf{nil} \text{ implies } \Delta \vdash V_i^?\ \mathsf{ok}$$

  Then, by Lemma B.5.4, we have for the V_i'' in the argument to $\mathsf{pick\text{-}constr}^{p^?}_{\Delta}$

  $$V_i^? \neq \mathsf{nil} \text{ implies } \Delta \vdash V_i''\ \mathsf{ok}$$

  Clearly, $\mathcal{V}_i \neq \mathsf{nil}$ for all $i \in \mathsf{disp}(I)$, so $V_i^? \neq \mathsf{nil}$ for all $i \in \mathsf{disp}(I)$. Hence, with Lemma B.5.30

  $$V_i^? = \mathsf{nil} \text{ implies } \Delta \vdash V_i''\ \mathsf{ok}$$

  Hence,

  $$\Delta \vdash \overline{W}\ \mathsf{ok}$$

  With Theorem 3.28

  $$\Delta \Vdash \overline{W}\,\textbf{implements}\,I\texttt{<}\overline{W'}\texttt{>}$$

  We have $[\overline{V/X}]\overline{\mathcal{P}} = [\overline{V/X}][\overline{W/Z}, \overline{W'/Z'}]\overline{Q}$, so with the assumption $\Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}}$ and Lemma B.5.29

  $$\Delta \vdash [\overline{V/X}]\,\underbrace{[\overline{W/Z}, \overline{W'/Z'}]U'}_{=U}\ \mathsf{ok}$$

  as required.

End case distinction on the rule used to derive $\mathsf{a\text{-}mtype}_\Delta(m, T, \overline{T})$. □

Lemma B.5.33. *If* $\Delta \vdash N$ ok *and* $\mathsf{fields}(N) = \overline{U\,f}^n$, *then* $\Delta \vdash U_i$ ok *for all* $i \in [n]$.

Proof. We proceed by induction on the derivation of $\mathsf{fields}(N) = \overline{U\,f}^n$.
Case distinction on the last rule in the derivation of $\mathsf{fields}(N) = \overline{U\,f}^n$.

- *Case* rule FIELDS-OBJECT: Then $n = 0$ and the claim holds trivially.
- *Case* rule FIELDS-CLASS: Then $N = C\texttt{<}\overline{V}\texttt{>}$ and

$$\textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } M \textbf{ where } \overline{P}\,\{\,\overline{T\,f}\ldots\}$$
$$\mathsf{fields}([\overline{V/X}]M) = \overline{T'\,f'}$$
$$\overline{U\,f}^n = \overline{T'\,f'}, [\overline{V/X}]\overline{T\,f}$$

Clearly, $N \unlhd_{\mathsf{c}} [\overline{V/X}]M$, so $\Delta \vdash [\overline{V/X}]M$ ok by Lemma B.2.25. Hence, we have by the I.H. that

$$\Delta \vdash \overline{T'}\ \mathsf{ok}$$

The underlying program is well-typed, so we have $\overline{P}, \overline{X} \vdash \overline{T}$ ok. From $\Delta \vdash C\texttt{<}\overline{V}\texttt{>}$ ok we get $\Delta \Vdash [\overline{V/X}]\overline{P}$ and $\Delta \vdash \overline{V}$ ok. Hence, with Lemma B.2.24,

$$\Delta \vdash [\overline{V/X}]\overline{T}\ \mathsf{ok}$$

End case distinction on the last rule in the derivation of $\mathsf{fields}(N) = \overline{U\,f}^n$. □

Lemma B.5.34 (Expression typing ensures well-formedness). *Suppose that* $\vdash \Delta$ ok *and* $\Delta \vdash \Gamma$ ok. *If* $\Delta; \Gamma \vdash_{\mathrm{a}} e : T$ *then* $\Delta \vdash T$ ok.

Proof. We proceed by induction on the derivation of $\Delta; \Gamma \vdash_{\mathrm{a}} e : T$.
Case distinction on the last rule used in the derivation of $\Delta; \Gamma \vdash_{\mathrm{a}} e : T$.

- *Case* rule EXP-ALG-VAR: Follows with the assumption $\Delta \vdash \Gamma$ ok.
- *Case* rule EXP-ALG-FIELD: Then

$$\Delta; \Gamma \vdash_{\mathrm{a}} e' : T'$$
$$\mathsf{bound}_\Delta(T') = N$$
$$\mathsf{fields}(N) = \overline{U\,f}$$
$$e = e'.f_j$$
$$T = U_j$$

We get from the I.H. that $\Delta \vdash T'$ ok. With Lemma B.5.23 then $\Delta \vdash N$ ok. Then we get with Lemma B.5.33 that $\Delta \vdash U_j$ ok.

- *Case* rule EXP-ALG-INVOKE: Then

$$e = e'.m\texttt{<}\overline{V}\texttt{>}(\overline{e})$$
$$T = [\overline{V/X}]U$$
$$\Delta; \Gamma \vdash_{\mathrm{a}} e' : T'$$
$$(\forall i)\ \Delta; \Gamma \vdash_{\mathrm{a}} e_i : T_i$$
$$\mathsf{a\text{-}mtype}_\Delta(m, T', \overline{T}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}}$$
$$\Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}}$$
$$\Delta \vdash \overline{V}\ \mathsf{ok}$$

Applying the I.H. yields $\Delta \vdash T', \overline{T}$ ok, so we can apply Lemma B.5.32 and get $\Delta \vdash [\overline{V/X}]U$ ok, as required.

- *Case* rule EXP-ALG-INVOKE-STATIC: Then

$$\begin{gathered}
e = I\texttt{<}\overline{W}\texttt{>}[\overline{T}].m\texttt{<}\overline{V}\texttt{>}(\overline{e}) \\
T = [\overline{V/X}]U \\
\Delta \vdash \overline{T}, \overline{V}\ \mathsf{ok} \\
\Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}} \\
\mathsf{a\text{-}smtype}_\Delta(m, I\texttt{<}\overline{W}\texttt{>}[\overline{T}]) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U \textbf{ where } \overline{\mathcal{P}}
\end{gathered}$$

  Applying Lemma B.5.32 yields $\Delta \vdash [\overline{V/X}]U$ ok, as required.

- *Case* rule EXP-ALG-NEW: Then $\Delta \vdash T$ ok from the premise of this rule.

- *Case* rule EXP-ALG-CAST: Then $\Delta \vdash T$ ok from the premise of this rule.

End case distinction on the last rule used in the derivation of $\Delta; \Gamma \vdash_{\mathrm{a}} e : T$. □

Lemma B.5.35. *If* $\mathsf{fields}(N) = \overline{T\,f}^n$ *and* $i \in [n]$*, then there exists*

$$\textbf{class } C\texttt{<}\overline{X}\texttt{>} \ldots \{\overline{V\,g}\ \ldots\}$$

such that $N \trianglelefteq_{\mathrm{c}} C\texttt{<}\overline{U}\texttt{>}$ *and* $T_i\,f_i = [\overline{U/X}]V_j\,g_j$ *for some* j*.*

Proof. We proceed by induction on the derivation of $\mathsf{fields}(N) = \overline{T\,f}^n$. The derivation cannot end with rule FIELDS-OBJECT because this would contradict $i \in [n]$. Hence, the last rule must be FIELDS-CLASS. We get

$$\begin{gathered}
N = D\texttt{<}\overline{W}\texttt{>} \\
\textbf{class } D\texttt{<}\overline{X}\texttt{>} \textbf{ extends } M \textbf{ where } \overline{P}\,\{\,\overline{T'\,f'}\ldots\} \\
\mathsf{fields}([\overline{W/X}]M) = \overline{T''\,f''} \\
\overline{T\,f}^n = \overline{T''\,f''}^m, [\overline{W/X}]\overline{T'\,f'}
\end{gathered}$$

If $i > m$ set $C\texttt{<}\overline{U}\texttt{>} = D\texttt{<}\overline{W}\texttt{>}$. Otherwise, the claim follows with the I.H., the fact that $D\texttt{<}\overline{W}\texttt{>} \trianglelefteq_{\mathrm{c}} [\overline{W/X}]M$, and Lemma B.1.4. □

Proof of Theorem 3.35. We proceed by induction on the derivation of $\Delta; \Gamma \vdash_{\mathrm{a}} e : T$.
Case distinction on the last rule of the derivation of $\Delta; \Gamma \vdash_{\mathrm{a}} e : T$.

- *Case* rule EXP-ALG-VAR: Obvious.

- *Case* rule EXP-ALG-FIELD: Inverting the rule yields

$$\begin{gathered}
e = e'.f_j \\
\Delta; \Gamma \vdash_{\mathrm{a}} e' : T' \\
\mathsf{bound}_\Delta(T') = N \\
\mathsf{fields}(N) = \overline{U\,f} \\
T = U_j
\end{gathered}$$

With Lemma B.5.35 there exists a class C such that

$$\begin{gathered}\textbf{class } C\texttt{<}\overline{X}\texttt{>} \ldots \{\overline{V\, g} \ldots\} \\ N \trianglelefteq_{\mathsf{c}} C\texttt{<}\overline{W}\texttt{>} \\ U_j\, f_j = [\overline{W/X}]V_i\, g_i \end{gathered} \tag{B.5.80}$$

By Lemma B.5.3 we have $\Delta \vdash T' \leq N$, so $\Delta \vdash T' \leq C\texttt{<}\overline{W}\texttt{>}$. We get by the I.H. that $\Delta; \Gamma \vdash e' : T'$, so with rule EXP-SUBSUME, $\Delta; \Gamma \vdash e' : C\texttt{<}\overline{W}\texttt{>}$. The claim now follows with rule EXP-FIELD and (B.5.80).

- *Case* rule EXP-ALG-INVOKE: We get from the premises of the rule

$$\begin{gathered} e = e'.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) \\ T = [\overline{V/X}]U \\ \Delta; \Gamma \vdash_{\mathsf{a}} e' : T' \\ (\forall i)\ \Delta; \Gamma \vdash_{\mathsf{a}} e_i : T_i \\ \mathsf{a\text{-}mtype}_{\Delta}(m, T', \overline{T}) = \texttt{<}\overline{X}\texttt{>}\, \overline{U\, x} \to U \textbf{ where } \overline{\mathcal{P}} \\ (\forall i)\ \Delta \vdash_{\mathsf{a}} T_i \leq [\overline{V/X}]U_i \\ \Delta \Vdash_{\mathsf{a}} [\overline{V/X}]\overline{\mathcal{P}} \\ \Delta \vdash_{\mathsf{a}} \overline{V}\ \mathsf{ok} \end{gathered}$$

By the I.H.

$$\begin{gathered} \Delta; \Gamma \vdash e' : T' \\ (\forall i)\ \Delta; \Gamma \vdash e_i : T_i \end{gathered}$$

With Lemma B.5.34

$$\Delta \vdash T', \overline{T}\ \mathsf{ok}$$

With Theorem 3.31, we get the existence of T'' such that

$$\begin{gathered} \Delta \vdash T' \leq T'' \\ \mathsf{mtype}_{\Delta}(m, T'') = \texttt{<}\overline{X}\texttt{>}\, \overline{U\, x} \to U \textbf{ where } \overline{\mathcal{P}} \end{gathered}$$

We have by rule EXP-SUBSUME

$$\begin{gathered} \Delta; \Gamma \vdash e' : T'' \\ (\forall i)\ \Delta; \Gamma \vdash e_i : [\overline{V/X}]U_i \end{gathered}$$

so the claim follows with rule EXP-INVOKE.

- *Case* rule EXP-ALG-INVOKE-STATIC: We use the I.H. and rule EXP-SUBSUME to derive the correct types for the arguments of the call. With Corollary B.5.2, we get that smtype and a-smtype are equivalent. The claim then follows with rule EXP-INVOKE-STATIC.
- *Case* rule EXP-ALG-NEW: We use the I.H. and rule EXP-SUBSUME to derive the correct types for the arguments of the constructor call. The claim then follows with rule EXP-NEW.
- *Case* rule EXP-ALG-CAST: Follows from the I.H.

End case distinction on the last rule of the derivation of $\Delta; \Gamma \vdash_{\mathsf{a}} e : T$. □

B.5.6 Proof of Theorem 3.36

Theorem 3.36 states that algorithmic expression typing is complete with respect to its declarative specification in Figure 3.9. All proofs in this section apply the equivalences and implications of Corollary B.5.2 implicitly.

Lemma B.5.36. *If* **class** $C\texttt{<}\overline{X}\texttt{>} \ldots \{\overline{U\,f}\ \ldots\}$ *and* $N \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{T}\texttt{>}$ *then* $\mathsf{fields}(N) = \ldots \overline{U'\,f} \ldots$ *such that* $[\overline{T/X}]\overline{U} = \overline{U'}$.

Proof. Follows by a routine induction on the derivation of $N \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{T}\texttt{>}$. □

Proof of Theorem 3.36. We proceed by induction on the derivation of $\Delta;\Gamma \vdash e : T$.
Case distinction on the last rule used in the derivation of $\Delta;\Gamma \vdash e : T$.

- *Case* rule EXP-VAR: Obvious.
- *Case* rule EXP-FIELD: By inverting the rule, we get

$$\Delta;\Gamma \vdash e' : C\texttt{<}\overline{T}\texttt{>}$$
$$\textbf{class}\ C\texttt{<}\overline{X}\texttt{>}\ \textbf{extends}\ N\ \textbf{where}\ \overline{P}\,\{\,\overline{U\,f}\ldots\}$$
$$e = e'.f_j$$
$$T = [\overline{T/X}]U_j$$

  We get from the I.H.

$$\Delta;\Gamma \vdash_{\mathrm{a}} e' : T'$$
$$\Delta \vdash T' \leq C\texttt{<}\overline{T}\texttt{>}$$

  Hence, with Corollary B.5.2,

$$\Delta \vdash_{\mathrm{q}}{}' T' \leq C\texttt{<}\overline{T}\texttt{>}$$

  By Lemma B.5.18

$$\mathsf{bound}_\Delta(T') = N$$
$$N \trianglelefteq_{\mathbf{c}} C\texttt{<}\overline{T}\texttt{>}$$

  By Lemma B.5.36

$$\mathsf{fields}(N) = \ldots \overline{U'\,f} \ldots$$
$$[\overline{T/X}]\overline{U} = \overline{U'}$$

  The claim now follows with rule EXP-ALG-FIELD.

- *Case* rule EXP-INVOKE: Inverting the rule yields

$$e = e'.m\texttt{<}\overline{V}\texttt{>}(\overline{e})$$
$$T = [\overline{V/X}]U'$$
$$\Delta;\Gamma \vdash e' : T'$$
$$(\forall i)\ \Delta;\Gamma \vdash e_i : [\overline{V/X}]U_i$$
$$\mathsf{mtype}_\Delta(m, T') = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U'\ \textbf{where}\ \overline{\mathcal{P}}$$
$$\Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}}$$
$$\Delta \vdash \overline{V}\ \mathsf{ok}$$

By the I.H.

$$\begin{gathered}\Delta;\Gamma \vdash_{\mathrm{a}} e' : T'' \\ \Delta \vdash T'' \leq T' \\ (\forall i)\ \Delta;\Gamma \vdash_{\mathrm{a}} e_i : W_i \\ (\forall i)\ \Delta \vdash W_i \leq [\overline{V/X}]U_i\end{gathered}$$

Now with Lemma B.5.34

$$\Delta \vdash T'' \ \mathsf{ok}$$

By Theorem 3.32

$$\begin{gathered}\mathsf{a\text{-}mtype}_{\Delta}(m, T'', \overline{W}) = \texttt{<}\overline{X}\texttt{>}\,\overline{U'\,x} \to U'' \textbf{ where } \overline{\mathcal{P}} \\ (\forall i)\ \Delta \vdash W_i \leq [\overline{V/X}]U'_i \\ \Delta \vdash [\overline{V/X}]U'' \leq [\overline{V/X}]U'\end{gathered}$$

We now get with rule EXP-ALG-INVOKE

$$\Delta;\Gamma \vdash_{\mathrm{a}} e'.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V/X}]U''$$

- *Case* rule EXP-INVOKE-STATIC: Inverting the rule yields

$$\begin{gathered}e = I\texttt{<}\overline{W}\texttt{>}[\overline{T}].m\texttt{<}\overline{V}\texttt{>}(\overline{e}) \\ T = [\overline{V/X}]U' \\ \mathsf{smtype}_{\Delta}(m, I\texttt{<}\overline{W}\texttt{>}[\overline{T}]) = \texttt{<}\overline{X}\texttt{>}\,\overline{U\,x} \to U' \textbf{ where } \overline{\mathcal{P}} \\ (\forall i)\ \Delta;\Gamma \vdash e_i : [\overline{V/X}]U_i \\ \Delta \Vdash [\overline{V/X}]\overline{\mathcal{P}} \\ \Delta \vdash \overline{T}, \overline{V} \ \mathsf{ok}\end{gathered}$$

By the I.H.

$$\begin{gathered}(\forall i)\ \Delta;\Gamma \vdash_{\mathrm{a}} e_i : W_i \\ \Delta \vdash W_i \leq [\overline{V/X}]U_i\end{gathered}$$

With Corollary B.5.2, we get that smtype and a-smtype are equivalent. We then have by rule EXP-ALG-INVOKE-STATIC

$$\Delta;\Gamma \vdash_{\mathrm{a}} I\texttt{<}\overline{W}\texttt{>}[\overline{T}].m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V/X}]U'$$

- *Case* rule EXP-NEW: The claim follows from the I.H. and rule EXP-ALG-NEW.
- *Case* rule EXP-CAST: The claim follows from the I.H. and rule EXP-ALG-CAST.
- *Case* rule EXP-SUBSUME: From the premise of the rule, we get $\Delta;\Gamma \vdash e : U'$ and $\Delta \vdash U' \leq T$. The I.H. yields $\Delta;\Gamma \vdash_{\mathrm{a}} e : U$ and $\Delta \vdash U \leq U'$. We then have $\Delta \vdash U \leq T$ by rule SUB-TRANS.

End case distinction on the last rule used in the derivation of $\Delta;\Gamma \vdash e : T$. □

B.5.7 Proof of Theorem 3.37

Theorem 3.37 states that the expression typing algorithm induced by the rules in Figures 3.27, 3.29 and 3.30 terminates. All proofs in this section apply the equivalences and implications of Corollary B.5.2 implicitly.

Lemma B.5.37. *If* $T^?_i \neq \mathsf{nil}$ *for all* $i \in \mathsf{disp}(I)$*, then the set*

$$\mathscr{R} = \{\mathcal{R} \mid \Delta \Vdash^?_{\mathrm{a}} \overline{T^?}\,\textbf{implements}\,I\texttt{<}\overline{V^?}\texttt{>} \mathrel{-\!\!\triangleright} \mathcal{R}\}$$

is finite.

Proof. We generalize the claim and prove that

$$\mathscr{R} = \{\mathcal{R} \mid \Delta; \mathscr{G}; \beta \Vdash^?_{\mathrm{a}} \overline{T^?}\,\textbf{implements}\,I\texttt{<}\overline{V^?}\texttt{>} \mathrel{-\!\!\triangleright} \mathcal{R}\}$$

is finite. Assume $\mathscr{R} = \{\mathcal{R}_1, \mathcal{R}_2, \dots\}$ is infinite. W.l.o.g., assume for all $i \in \mathbb{N}$

$$\mathcal{D}_i :: \Delta; \mathscr{G}; \beta \Vdash^?_{\mathrm{a}} \overline{T^?}\,\textbf{implements}\,I\texttt{<}\overline{V^?}\texttt{>} \mathrel{-\!\!\triangleright} \mathcal{R}_i$$
$$i \neq j \text{ implies } \mathcal{R}_i \neq \mathcal{R}_j$$

such that all $\mathcal{D}_i$ end with the same rule.
Case distinction on the last rule in all $\mathcal{D}_i$.

- *Case* rule ENT-NIL-ALG-ENV: Impossible because Δ is finite and, obviously, $\mathsf{sup}(\mathcal{S})$ is finite for all $\mathcal{S}$.
- *Case* rule ENT-NIL-ALG-IFACE$_1$: Impossible because the set $\{I\texttt{<}\overline{V}\texttt{>} \mid \Delta; \beta; I \vdash_{\mathrm{a}} T_1 \uparrow I\texttt{<}\overline{V}\texttt{>}\}$ is finite by Lemma B.5.13.
- *Case* rule ENT-NIL-ALG-IFACE$_2$: Impossible because the set $\{J\texttt{<}\overline{V}\texttt{>} \mid J'\texttt{<}\overline{W}\texttt{>} \trianglelefteq_{\mathrm{i}} J\texttt{<}\overline{V}\texttt{>}\}$ is finite by Lemma B.5.13.
- *Case* rule ENT-NIL-ALG-IMPL: W.l.o.g., assume that the same implementation definition

  $$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{V}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{P}\ \dots$$

  appears in the premise of the last rule of every $\mathcal{D}_i$. (There are only finitely many implementation definitions in a program, so infinitely many derivations must share the same implementation definition.) We then have

  $$\mathcal{R}_i = \overline{T}\,\textbf{implements}\,I\texttt{<}[\overline{U_i/X}]\overline{V}\texttt{>}$$
  $$\Delta; \beta; I \vdash^?_{\mathrm{a}} \overline{T^?} \uparrow [\overline{U_i/X}]\overline{N} \mathrel{-\!\!\triangleright} \overline{T}$$

  Clearly, for $j \in \mathsf{disp}(I)$, we have

  $$\Delta \vdash_{\mathrm{q}}{}' T^?_j \leq [\overline{U_i/X}]N_j$$

  With criterion WF-IMPL-2 we have $\overline{X} \subseteq \mathsf{ftv}(\{N_i \mid i \in \mathsf{disp}(I)\}$, so with Lemma B.5.13 we know that the set $\{U_i \mid i \in \mathbb{N}\}$ is finite. Hence, the set

  $$\{[\overline{U_i/X}]\overline{N} \mid i \in \mathbb{N}\} \cup \{[\overline{U_i/X}]\overline{V} \mid i \in \mathbb{N}\}$$

  is finite. But if $T^?_j = \mathsf{nil}$ then $T_j = [\overline{U_i/X}]N_j$. Hence, the set $\mathscr{R}$ cannot be infinite, which contradicts our assumption.

End case distinction on the last rule in all $\mathcal{D}_i$. □

Lemma B.5.38. *Let*

$$\begin{aligned}\mathscr{M} = \{\overline{V}\,\textbf{implements}\,I\texttt{<}\overline{V''}\texttt{>} \mid (\forall i \in [l])\ &\text{if } \mathscr{V}_i = \emptyset \text{ then } V_i^? = \mathsf{nil} \\ &\text{else define } V_i^? \text{ such that} \\ &\qquad \Delta \vdash_{\mathrm{q}}' V_i' \leq V_i^? \text{ for } V_i' \in \mathscr{V}_i, \\ \Delta \Vdash_{\mathrm{a}}^{?} \overline{V^?}\,\textbf{implements}\,I\texttt{<}\overline{\mathsf{nil}}\texttt{>} &\rightharpoonup \overline{V}\,\textbf{implements}\,I\texttt{<}\overline{V''}\texttt{>}\}\end{aligned}$$

If $\mathscr{V}_i \neq \emptyset$ for all $i \in \mathsf{disp}(I)$ and all $\mathscr{V}_i$ are finite, then $\mathscr{M}$ is finite.

Proof. With Lemma B.5.13 we know that only finitely many choices for the $V_i^?$s in the definition of $\mathscr{M}$ exist. Moreover, $V_i^? \neq \mathsf{nil}$ for all $i \in \mathsf{disp}(I)$. The claim now follows with Lemma B.5.37. □

Proof of Theorem 3.37. Let $\mathtt{type}(\Delta, \Gamma, e)$ be the algorithm induced by the rules in Figures 3.27, 3.29, and 3.30. Clearly, the third argument of a recursive call of $\mathtt{type}$ is always a subexpression of the original expression argument; hence, there are only finitely many recursive calls of $\mathtt{type}$. Similarly, the function checking the relations $\Delta \vdash_{\mathrm{a}} T\ \mathsf{ok}$ and $\Delta \vdash_{\mathrm{a}} \mathcal{P}\ \mathsf{ok}$ calls itself only on strictly smaller arguments. Moreover, checking entailment and subtyping terminates by Theorem 3.27.

The only possible sources of non-termination left are the auxiliaries $\mathsf{a\text{-}mtype}$, $\mathsf{a\text{-}smtype}$, bound, and fields. Thereof, $\mathsf{a\text{-}smtype}$ and fields obviously terminate. For $\mathsf{bound}_\Delta(T)$, we get with an application of Lemma B.5.13 that the set $\{\Delta \vdash_{\mathrm{q}}' T \leq N\}$ is finite, so such a call also terminates.

We now consider a call $\mathsf{a\text{-}mtype}(m, T, \overline{T})$. If $m = m^{\mathrm{c}}$, then the call obviously terminates. Otherwise, we check that all premises of rule ALG-MTYPE-IFACE terminate. With Lemma B.5.13 we easily verify that all $\mathscr{V}_i$ in the premise are finite and that $\mathscr{V}_i \neq \emptyset$ for all $i \in \mathsf{disp}(I)$. By Lemma B.5.38 we then have that the argument of $\mathsf{pick\text{-}constr}$ is finite, so the premise involving $\mathsf{pick\text{-}constr}$ terminates. The remaining premises terminate trivially. □

B.6 Deciding Program Typing

This section proves Theorem 3.39 (soundness, completeness, and termination of $\mathtt{unify}_\sqcap$) and Theorem 3.40 (equivalence/soundness of the well-formedness criteria defined in Section 3.7.3 with respect to the criteria given in Section 3.5.3),

B.6.1 Proof of Theorem 3.39

Theorem 3.39 states that $\mathtt{unify}_\sqcap$ is sound, complete, and terminating. All proofs in this section apply the equivalences and implications of Corollary B.5.2 implicitly.

Definition B.6.1. The notation $\mathsf{sol}(\mathbb{L})$ denotes the set of solutions of a unification problem modulo greatest lower bounds $\mathbb{L}$.

Lemma B.6.2. *Assume that $\mathbb{L} = (\Delta, \overline{X}, \{G_{11} \sqcap^? G_{12}, \ldots, G_{n1} \sqcap^? G_{n2}\})$ is a unification problem modulo greatest lower bounds. Choose $(i_k, j_k) \in \{(1,2),(2,1)\}$ for all $k \in [n]$ and define $\mathbb{L}' = (\Delta, \overline{X}, \{G_{1i_1} \leq^? G_{1j_1}, \ldots, G_{ni_n} \leq^? G_{nj_n}\})$. Then $\mathsf{sol}(\mathbb{L}') = \emptyset$ or $\mathsf{sol}(\mathbb{L}) = \mathsf{sol}(\mathbb{L}')$.*

Proof. If $\mathsf{sol}(\mathbb{L}') = \emptyset$, then nothing is to prove. Thus, assume $\mathsf{sol}(\mathbb{L}') \neq \emptyset$.

- "$\mathsf{sol}(\mathbb{L}) \subseteq \mathsf{sol}(\mathbb{L}')$". Assume $\varphi \in \mathsf{sol}(\mathbb{L})$. Then, by the rules in Figure 3.18, there exists

$$((i_1', j_1'), \ldots, (i_n', j_n')) \in \prod_{i=1}^{n} \{(1,2),(2,1)\}$$

such that for all $k \in [n]$

$$\Delta \vdash \varphi G_{ki'_k} \leq \varphi G_{kj'_k} \tag{B.6.1}$$

From $\mathsf{sol}(\mathbb{L}') \neq \emptyset$ we get the existence of a substitution ψ such that for all $k \in [n]$

$$\Delta \vdash \psi G_{ki_k} \leq \varphi G_{kj_k}$$

It is easy to see that $\mathbb{L}'$ is a well-defined unification problem modulo greatest lower bounds. Hence,

$$\mathsf{dom}(\varphi) \subseteq \overline{X} \tag{B.6.2}$$

$$\mathsf{dom}(\psi) \subseteq \overline{X} \tag{B.6.3}$$

We now show $\Delta \vdash \varphi G_{ki_k} \leq \varphi G_{kj_k}$ for all $k \in [n]$. This implies $\varphi \in \mathsf{sol}(\mathbb{L}')$.
Assume $k \in [n]$. We have $(i_k, j_k) = (1, 2)$ or $(i_k, j_k) = (2, 1)$, and $(i'_k, j'_k) = (1, 2)$ or $(i'_k, j'_k) = (2, 1)$. If $(i_k, j_k) = (i'_k, j'_k)$ then with (B.6.1) $\Delta \vdash \varphi G_{ki_k} \leq \varphi G_{kj_k}$. Thus, assume $(i_k, j_k) \neq (i'_k, j'_k)$. W.l.o.g., $(i_k, j_k) = (1, 2)$ and $(i'_k, j'_k) = (2, 1)$. Hence, $\Delta \vdash \varphi G_{k2} \leq \varphi G_{k1}$ and $\Delta \vdash \psi G_{k1} \leq \psi G_{k2}$. With (B.6.2), (B.6.3), and because $\mathbb{L}$ is a unification problem modulo greatest lower bounds, we know that φG_{k2}, φG_{k1}, ψG_{k2}, and ψG_{k1} are all G-types. Thus, with Theorem 3.12 and Lemma B.1.14:

$$\Delta \vdash_{\mathrm{q}}{}' \varphi G_{k2} \leq \varphi G_{k1}$$
$$\Delta \vdash_{\mathrm{q}}{}' \psi G_{k1} \leq \psi G_{k2}$$

Case distinction on the form of G_{k2}.

- *Case* $G_{k2} = Y$ for some Y: $\mathbb{L}$ is a unification problem modulo greatest lower bounds, so $Y \notin \overline{X}$. Hence, with (B.6.2) and (B.6.3), $\varphi G_{k2} = Y = \psi G_{k2}$. With Lemma B.1.10 then $\psi G_{k1} = Y$, so $G_{k1} = Y$. Thus, $\Delta \vdash \varphi G_{ki_k} \leq \varphi G_{kj_k}$.
- *Case* $G_{k2} = C\texttt{<}\overline{T}\texttt{>}$ for some $C\texttt{<}\overline{T}\texttt{>}$: With Lemma B.1.10 then $\varphi G_{k1} = \varphi D\texttt{<}\overline{U}\texttt{>}$. By inverting rule SUB-Q-ALG-CLASS, we get

  $$\varphi C\texttt{<}\overline{T}\texttt{>} \trianglelefteq_{\mathsf{c}} \varphi D\texttt{<}\overline{U}\texttt{>}$$
  $$\psi D\texttt{<}\overline{U}\texttt{>} \trianglelefteq_{\mathsf{c}} \psi C\texttt{<}\overline{T}\texttt{>}$$

  The class graph is acyclic (criterion WF-PROG-5), so

  $$C = D$$
  $$\varphi\overline{T} = \varphi\overline{U}$$

  Thus, $\Delta \vdash \varphi G_{k1} \leq \varphi G_{k2}$, so $\Delta \vdash \varphi G_{ki_k} \leq \varphi G_{kj_k}$.

End case distinction on the form of G_{k2}.

- "$\mathsf{sol}(\mathbb{L}') \subseteq \mathsf{sol}(\mathbb{L})$". If $\varphi \in \mathsf{sol}(\mathbb{L}')$ then obviously also $\varphi \in \mathsf{sol}(\mathbb{L})$. □

Proof of Theorem 3.39. Termination of $\texttt{unify}_{\sqcap}$ follows with Theorem 3.24.

Next, assume the unification problem modulo greater lower bounds $\mathbb{L}$ does not have a solution. Thus, none of the unification problems modulo kernel subtyping constructed by $\texttt{unify}_{\sqcap}$ has a solution. The claim now follows from Theorem 3.23.

Finally, assume that $\mathbb{L}$ has a solution. Thus, some of the unification problems modulo kernel subtyping constructed by $\texttt{unify}_{\sqcap}$ have solutions. Assume that $\mathbb{L}'$ is the first of these problems. According to Lemma B.6.2, we then have $\mathsf{sol}(\mathbb{L}) = \mathsf{sol}(\mathbb{L}')$. The claim now follows with Theorem 3.23. □

B.6.2 Proof of Theorem 3.40

Theorem 3.40 states equivalence/soundness of the well-formedness criteria defined in Section 3.7.3 with respect to the criteria given in Section 3.5.3. All proofs in this section apply the equivalences and implications of Corollary B.5.2 implicitly.

Lemma B.6.3. *If a unification problem modulo kernel subtyping (or modulo greatest lower bounds) has a solution, than it also has a most general solution.*

Proof. Follows from Theorems 3.23, 3.24, and 3.39. □

Proof of Theorem 3.40. The equivalence proofs for WF-PROG-2′, WF-PROG-3′, and WF-TENV-6′ are easy, using Lemma B.6.3 for proving that the formulations in Section 3.7.3 imply the original formulations in Section 3.5.3.

To prove that criterion WF-PROG-4′ implies criterion WF-PROG-4 requires slightly more work. Assume

$$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ [\,\overline{M}\,]\ \textbf{where}\ \overline{P}\ \ldots$$

$$\textbf{implementation}\texttt{<}\overline{Y}\texttt{>}\ I\texttt{<}\overline{U}\texttt{>}\ [\,\overline{N}\,]\ \textbf{where}\ \overline{Q}\ \ldots$$

with $[\overline{V/X}]\overline{M} \trianglelefteq_{\mathbf{c}} [\overline{W/Y}]\overline{N}$ and $\emptyset \Vdash [\overline{W/Y}]\overline{Q}$.

W.l.o.g., $\overline{X} \cap \overline{Y} = \emptyset$ and the two implementation definitions given are disjoint. From $[\overline{V/X}]\overline{M} \trianglelefteq_{\mathbf{c}} [\overline{W/Y}]\overline{N}$ and Lemma B.6.3 we get the existence of a substitution φ such that $\varphi\overline{M} \trianglelefteq_{\mathbf{c}} \varphi\overline{N}$ and φ is more general than $[\overline{V/X}]$ and $[\overline{W/Y}]$; that is, $[\overline{V/X}] = \varphi'\varphi$ and $[\overline{W/Y}] = \varphi'\varphi$ for some substitution φ'.

Now assume $\mathcal{P} \in [\overline{V/X}]\overline{P}$. That is, there exists some i such that $\mathcal{P} = [\overline{V/X}]P_i$. From criterion WF-PROG-4′ we then get that either $\{Q \in \varphi\overline{Q}\} \Vdash \varphi P_i$ or $\varphi P_i \in \mathsf{sup}(\varphi\overline{Q}) \cup \{T\,\textbf{extends}\,U \mid T\,\textbf{extends}\,U' \in \varphi\overline{Q}, \varphi\overline{Q} \vdash_{\mathrm{q}}{}' U' \leq U\}$.

- Case $\{Q \in \varphi\overline{Q}\} \Vdash \varphi P_i$. We have $\emptyset \Vdash \varphi'\{Q \in \varphi\overline{Q}\}$, so $\emptyset \Vdash [\overline{V/X}]P_i$ by Lemma B.2.22.
- Case $\varphi P_i \in \mathsf{sup}(\varphi\overline{Q}) \cup \{T\,\textbf{extends}\,U \mid T\,\textbf{extends}\,U' \in \varphi\overline{Q}, \varphi\overline{Q} \vdash_{\mathrm{q}}{}' U' \leq U\}$.

  If $\varphi P_i \in \mathsf{sup}(\varphi\overline{Q})$, then $[\overline{V/X}]P_i \in \mathsf{sup}([\overline{W/Y}]\overline{Q})$ by Lemma B.1.13. We then get with $\emptyset \Vdash [\overline{W/Y}]\overline{Q}$, Theorem 3.12, Lemma B.1.27, and Theorem 3.11. that $\emptyset \Vdash [\overline{V/X}]P_i$.

  Suppose $\varphi P_i \in \{T\,\textbf{extends}\,U \mid T\,\textbf{extends}\,U' \in \varphi\overline{Q}, \varphi\overline{Q} \vdash_{\mathrm{q}}{}' U' \leq U\}$ and assume $\varphi P_i = T\,\textbf{extends}\,U$ with $T\,\textbf{extends}\,U' \in \varphi\overline{Q}$ and $\varphi\overline{Q} \vdash_{\mathrm{q}}{}' U' \leq U$. With $\emptyset \Vdash [\overline{W/Y}]\overline{Q}$ then

  $$\emptyset \vdash \varphi' T \leq \varphi' U'$$

  and with rule SUB-Q-ALG-KERNEL, Theorem 3.11, and Lemma B.2.22

  $$\emptyset \vdash \varphi' U' \leq \varphi' U$$

  By transitivity of subtyping and rule ENT-EXTENDS then $\emptyset \Vdash [\overline{V/X}]P_i$.

This proves $\emptyset \Vdash [\overline{V/X}]\overline{P}$. □

B.7 Syntactic Characterization of Finitary Closure

This section defines a syntactic but equivalent formulation of well-formedness criterion WF-TENV-2. Most definitions, lemmas, and proofs in this section are heavily based on work by Viroli [232] and by Kennedy and Pierce [113]. All proofs in this section apply the equivalences and implications of Corollary B.5.2 implicitly.

Definition B.7.1 (Type parameter dependency graph). The *type parameter dependency graph* $\mathscr{D}$ is a labeled graph $\mathscr{D} = (\mathscr{V}, \mathscr{E})$. The set of vertices $\mathscr{V}$ consists of all the formal type parameters to classes in the program:

$$\mathscr{V} = \{C\#i \mid \textbf{class } C\texttt{<}\overline{X}^n\texttt{>} \textbf{ extends } N \ldots, i \in [n]\}$$

The set of labeled edges $\mathscr{E} = \mathscr{E}_0 \cup \mathscr{E}_1$, where the labels are drawn from the set $\{0, 1\}$, represent uses of formal type parameters. Edges labeled with 0 are called non-expansive edges:

$$\mathscr{E}_0 = \{C\#i \xrightarrow{0} D\#j \mid \textbf{class } C\texttt{<}\overline{X}^n\texttt{>} \textbf{ extends } N \ldots, D\texttt{<}\overline{T}\texttt{>} \text{ subterm of } N, X_i = T_j\}$$

Edges labeled with 1 are called expansive edges:

$$\begin{aligned}\mathscr{E}_1 = \{C\#i \xrightarrow{1} D\#j \mid \textbf{class } C\texttt{<}\overline{X}^n\texttt{>} \textbf{ extends } N \ldots, D\texttt{<}\overline{T}\texttt{>} \text{ subterm of } N,\\ X_i \text{ proper subterm of } T_j\}\end{aligned}$$

The type parameter dependency graph is said to be *expansive* if, and only if, it contains a cycle with at least one expansive edge. Otherwise, the type parameter dependency graph is said to be *non-expansive.*

At some points, we use the name of the formal type parameter X_i instead of $C\#i$, assuming the names of all formal type parameters are (α-converted to be) distinct. If labels of edges are irrelevant, we simply omit them.

Definition B.7.2 (Levels in the type parameter dependency graph). Let $\mathscr{D} = (\mathscr{V}, \mathscr{E})$ be a type parameter dependency graph. The *level* of a vertex $X \in \mathscr{V}$, written $\mathsf{vlevel}(X)$, is a natural number such that for $X, Y \in \mathscr{V}$ the following property holds:

$$\text{if } X \to Y \text{ and } Y \to^+ X \text{ then } \mathsf{vlevel}(X) = \mathsf{vlevel}(Y)$$
$$\text{if } X \to Y \text{ and not } Y \to^+ X \text{ then } \mathsf{vlevel}(X) > \mathsf{vlevel}(Y)$$

Definition B.7.3 (Paths). A *path* ι is a sequence of formal type parameters, where ϵ denotes the empty path and $X \circ \iota$ is the path consisting of formal type parameter X prepended to path ι. By interpreting a path ι as a partial function from terms to subterms, we may use ι to identify a particular subterm in a type:

$$\epsilon(T) = T \qquad\qquad \frac{\iota(T_i) = U}{(C\#i \circ \iota)(C\texttt{<}\overline{T}\texttt{>}) = U}$$

We say that ι *is a path in T* if $\iota(T)$ is defined.

In the following $|\overline{\xi}|$ denotes the length of a sequence $\overline{\xi}$.

Definition B.7.4. Let $L, \delta \in \mathbb{N}$. The predicate $\phi_{L,\delta}(\iota)$ holds for a path ι if, and only if, ι can be divided into a sequence of (possibly empty) sequences of type parameters whose levels are bounded by $0, \ldots, L-1$ and whose lengths are bounded by δ. That is, $\phi_{L,\delta}(\iota)$ means that ι has the form $\overline{X_0}\,\overline{X_1} \ldots \overline{X_{L-1}}$, such that, for all $l \in \{0, \ldots, L-1\}$, $\mathsf{vlevel}(X) \leq l$ for all $X \in \overline{X_l}$ and $|\overline{X_l}| \leq \delta$.

The predicate $\phi_{L,\delta}$ is extended to types by defining that $\phi_{L,\delta}(T)$ holds for a type T if, and only if, $\phi_{L,\delta}(\iota)$ holds for every path ι in T.

Definition B.7.5. The *height* of a type T, written $\mathsf{height}(T)$, is defined as follows:

$$\begin{aligned}\mathsf{height}(X) &= 1\\ \mathsf{height}(C\texttt{<}\overline{T}\texttt{>}) &= 1 + \mathsf{max}_i(\mathsf{height}(T_i))\\ \mathsf{height}(I\texttt{<}\overline{T}\texttt{>}) &= 1 + \mathsf{max}_i(\mathsf{height}(T_i))\end{aligned}$$

Lemma B.7.6. *If* $\phi_{L,\delta}(T)$ *then* $\mathsf{height}(T) \leq \delta L$.

Proof. Easy. □

Let $\mathscr{D} = (\mathscr{V}, \mathscr{E})$ be the type parameter dependency graph of the underlying program. We define $L \in \mathbb{N}$ as the number of levels in $\mathscr{D}$ (that is, $0 \leq \mathsf{vlevel}(X) < L$ for any formal type parameter X). Moreover, we define $\delta \in \mathbb{N}$ as a bound on the height of the superclasses of the underlying program. That is, **class** $C\texttt{<}\overline{X}\texttt{>}$ **extends** $N \ldots$ implies $\mathsf{height}(N) \leq \delta$. In the following, we write ϕ instead of $\phi_{L,\delta}$.

Lemma B.7.7. *If* $\mathsf{height}(T) \leq \delta$ *then* $\phi(T)$.

Proof. Easy. □

Lemma B.7.8. *Suppose the type parameter dependency graph of the underlying program is non-expansive. If* $N \trianglelefteq_{\mathsf{c}} M$ *and* $\phi(N)$ *then* $\phi(M)$.

Proof. We proceed by induction on the derivation of $N \trianglelefteq_{\mathsf{c}} M$. If the last rule in this derivation is INH-CLASS-REFL, then the claim holds trivially. Otherwise, we have

$$\textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } N' \ldots$$

$$[\overline{T/X}]N' \trianglelefteq_{\mathsf{c}} M$$

$$N = C\texttt{<}\overline{T}\texttt{>}$$

We now show $\phi([\overline{T/X}]N')$, the claim then follows by the I.H. Note that $\mathsf{height}(N') \leq \delta$ by definition of δ.

Consider a path ι in $[\overline{T/X}]N'$. There are two possibilities. First, ι could be simply a path in N' that maps to a non-variable type. In this case, we know $|\iota| \leq \delta$, so we have $\phi(\iota)$ immediately.

Otherwise $\iota = \iota' \circ \iota''$ for paths ι' and ι'' such that ι' is non-empty, $\iota'(N') = X_i$ and ι'' is a path in T_i. Hence, $C\#i \circ \iota''$ is a path in $C\texttt{<}\overline{T}\texttt{>}$, and so from $\phi(C\texttt{<}\overline{T}\texttt{>})$, we can deduce $\phi(C\#i \circ \iota'')$, or written another way, $\phi(X_i \circ \iota'')$. Now if $\mathsf{vlevel}(X_i) = k$ then $\iota'' = \overline{Y_k}\,\overline{Y_{k+1}} \ldots \overline{Y_{L-1}}$, with $\mathsf{vlevel}(Y_{li}) \leq l$ for all i and $k \leq l < L$ and with $|\overline{Y_k}| < \delta$ and $|\overline{Y_l}| \leq \delta$ for $k < l < L$. Suppose $\iota' = \overline{Z} \circ Z$. By definition of the type parameter dependency graph, we know that $X_i \overset{1}{\to} Z_j$ for each j and that $X_i \overset{0}{\to} Z$. The type parameter dependency graph is non-expansive, so there is no j such that $Z_j \to^{+} X_i$. Hence, $\mathsf{vlevel}(Z_j) < \mathsf{vlevel}(X_i) = k$ for each j. Finally, because $|\overline{Z}| < \delta$ and $\mathsf{vlevel}(Z) \leq k$ and $|\overline{Y_k}| < \delta$, we see that $\iota = \overline{Z}\,(Z \circ \overline{Y_k})\,\overline{Y_{k+1}} \ldots \overline{Y_{L-1}}$ satisfies ϕ, as required. □

Lemma B.7.9. *Suppose the type parameter dependency graph of the underlying program is non-expansive. Moreover, assume that* δ *is not only a bound on the height of the superclasses of the underlying program, but also a bound on the height of the types in* Δ. *If* $\Delta \vdash_{\mathrm{q}}{}' U \leq N$ *and* $\phi(U)$, *then* $\phi(N)$.

Proof. We proceed by induction on the derivation of $\Delta \vdash_{\mathrm{q}}{}' U \leq N$.
Case distinction on the last rule in the derivation of $\Delta \vdash_{\mathrm{q}}{}' U \leq N$.

- *Case* rule SUB-Q-ALG-OBJ: Trivial.
- *Case* rule SUB-Q-ALG-VAR-REFL: Impossible.
- *Case* rule SUB-Q-ALG-VAR: Then $X = U$, $X\,\textbf{extends}\,U' \in \Delta$, and $\Delta \vdash_{\mathrm{q}}{}' U' \leq N$. Hence, $\mathsf{height}(U') \leq \delta$, so $\phi(U')$ by Lemma B.7.7. The claim now follows from the I.H.
- *Case* rule SUB-Q-ALG-CLASS: Follows by Lemma B.7.8.

- *Case* rule SUB-Q-ALG-IFACE: Impossible.

End case distinction on the last rule in the derivation of $\Delta \vdash_{\mathrm{q}}{}' U \leq N$. □

Lemma B.7.10. *Suppose Δ is finite and assume that the type parameter dependency graph of the underlying program is non-expansive. Then $\mathsf{closure}_\Delta(\mathscr{T})$ is finite for every finite $\mathscr{T}$.*

Proof. Let $\mathscr{T}$ be a finite set of types. We can safely assume that δ is not only a bound on height of the superclasses of the underlying program, but also a bound on the height of the types in $\mathscr{T}$ and Δ. We now prove that the height of types in $\mathsf{closure}_\Delta(\mathscr{T})$ is bounded by δL; then, because the set of types of a certain height is finite, it follows that $\mathsf{closure}_\Delta(\mathscr{T})$ is finite.

By Lemma B.7.6, it suffices to show that ϕ holds for all types in $\mathsf{closure}_\Delta(\mathscr{T})$. Assume $T \in \mathsf{closure}_\Delta(\mathscr{T})$. We proceed by induction on the derivation of $T \in \mathsf{closure}_\Delta(\mathscr{T})$.
Case distinction on the last rule of the derivation of $T \in \mathsf{closure}_\Delta(\mathscr{T})$.

- *Case* rule CLOSURE-ELEM: Then T in $\mathscr{T}$, so $\mathsf{height}(T) \leq \delta$. Then $\phi(T)$ with Lemma B.7.7.
- *Case* rule CLOSURE-UP: Then we have $U \in \mathsf{closure}_\Delta(\mathscr{T})$ and $\Delta \vdash_{\mathrm{q}}{}' U \leq N$ and $T = N$. From the I.H. we get $\phi(U)$. Moreover, with Lemma B.4.11 we have $\Delta \vdash_{\mathrm{q}}{}' U \leq N$. The claim now follows with Lemma B.7.9.
- *Case* rule CLOSURE-DECOMP-CLASS: Then $C\texttt{<}\overline{U}\texttt{>} \in \mathsf{closure}_\Delta(\mathscr{T})$ and $T = U_i$. From the I.H. we know $\phi(C\texttt{<}\overline{U}\texttt{>})$, so $\phi(U_i)$ also holds.
- *Case* rule CLOSURE-DECOMP-IFACE: Analogously to the preceding case.

End case distinction on the last rule of the derivation of $T \in \mathsf{closure}_\Delta(\mathscr{T})$. □

Lemma B.7.11. *Suppose $C\texttt{<}\overline{T}\texttt{>} \in \mathsf{closure}_\Delta(\mathscr{T})$.*

(i) If $C\#i \xrightarrow{0} D\#j$ then $D\texttt{<}\overline{U}\texttt{>} \in \mathsf{closure}_\Delta(\mathscr{T})$ for some $\overline{U}$ with $U_j = T_i$.

(ii) If $C\#i \xrightarrow{1} D\#j$ then $D\texttt{<}\overline{U}\texttt{>} \in \mathsf{closure}_\Delta(\mathscr{T})$ for some $\overline{U}$ such that T_i is a proper subterm of U_j.

Proof. We only proof the first claim. The proof of the second claim is similar.

From the definition of the type parameter dependency graph, we get

$$\begin{gathered}\textbf{class } C\texttt{<}\overline{X}\texttt{>} \textbf{ extends } N \ldots \\ D\texttt{<}\overline{V}\texttt{>} \text{ subterm of } N \\ V_j = X_i\end{gathered}$$

Obviously, $\Delta \vdash_{\mathrm{q}}{}' C\texttt{<}\overline{T}\texttt{>} \leq [\overline{T/X}]N$, so we have with rule CLOSURE-UP that

$$[\overline{T/X}]N \in \mathsf{closure}_\Delta(\mathscr{T})$$

Possibly repeated applications of rule CLOSURE-DECOMP-CLASS yield $[\overline{T/X}]D\texttt{<}\overline{V}\texttt{>} \in \mathsf{closure}_\Delta(\mathscr{T})$, from which the claim follows immediately. □

Lemma B.7.12. *Assume $\mathsf{closure}_\Delta(\mathscr{T})$ is finite for every finite $\mathscr{T}$. Then the type parameter dependency graph is non-expansive.*

Proof. We prove the contraposition; that is, we assume that the type parameter dependency graph is expansive and show that there exists a finite set $\mathscr{T}$ such that $\mathsf{closure}_\Delta(\mathscr{T})$ infinite.

Suppose the type parameter dependency graph is expansive; that is, there is a cycle such that at least one of the edges of the cycle (say the first) is expansive. Thus, either $C\#i \xrightarrow{1} C\#i$ or $C\#i \xrightarrow{1} D\#j \rightarrow^{+} C\#i$. Now consider $\mathscr{C} = \mathsf{closure}_\Delta(\{C\texttt{<}\overline{Object}\texttt{>}\})$.

- By possibly repeated applications of Lemma B.7.11 we see that also $C\text{<}\overline{U_1}\text{>} \in \mathscr{C}$ for types $\overline{U_1}$ such that *Object* is a proper subterm of U_{1i}.
- By possibly repeated applications of Lemma B.7.11 we see that also $C\text{<}\overline{U_2}\text{>} \in \mathscr{C}$ for types $\overline{U_2}$ such that U_{1i} is a proper subterm of U_{2i}.
- By possibly repeated applications of Lemma B.7.11 we see that also $C\text{<}\overline{U_3}\text{>} \in \mathscr{C}$ for types $\overline{U_3}$ such that U_{2i} is a proper subterm of U_{3i}.
- ...

Hence, there is a chain of types $C\text{<}\overline{\mathit{Object}}\text{>} = C\text{<}\overline{U_0}\text{>}, C\text{<}\overline{U_1}\text{>}, C\text{<}\overline{U_2}\text{>}, \ldots$ such that $C\text{<}\overline{U_i}\text{>} \in \mathscr{C}$ and $C\text{<}\overline{U_{i+1}}\text{>}$ is strictly larger than $C\text{<}\overline{U_i}\text{>}$ for all $i \in \mathbb{N}$. Thus, $\mathscr{C}$ is infinite. □

We are now ready to give an equivalent and implementable formulation of criterion WF-TENV-2.

WF-TENV-2′ The type parameter dependency graph of the underlying program is non-expansive.

Theorem B.1. *Criterion* WF-TENV-2 *and criterion* WF-TENV-2′ *are equivalent.*

Proof. Follows from Lemma B.7.10, Lemma B.7.12, and the fact that type environments are finite. □

C

Formal Details of Chapter 4

C.1 Type Soundness for iFJ

This section contains the proofs of Theorem 4.6 (preservation) and Theorem 4.9 (progress), which are necessary to complete the type soundness proof for iFJ (see Section 4.2.4). The section implicitly assumes that the underlying iFJ program *prog* is well-formed; that is, $\vdash_{\mathsf{iFJ}}$ *prog* ok.

C.1.1 Proof of Theorem 4.6

Theorem 4.6 is the preservation theorem for iFJ. To reason about subtyping in iFJ, Figure C.1 introduces the relation $\vdash_{\mathsf{iFJ\text{-}a}} T \leq U$, which serves as an algorithmic variant of iFJ's subtyping relation.

Lemma C.1.1 (Reflexivity of algorithmic iFJ-subtyping). *For all types T, $\vdash_{\mathsf{iFJ\text{-}a}} T \leq T$.*

Proof. Obvious. □

Lemma C.1.2 (Transitivity of algorithmic iFJ-subtyping). *If $\vdash_{\mathsf{iFJ\text{-}a}} T \leq U$ and $\vdash_{\mathsf{iFJ\text{-}a}} U \leq V$ then $\vdash_{\mathsf{iFJ\text{-}a}} T \leq V$.*

Proof. We proceed by induction on the height of the derivation of $\vdash_{\mathsf{iFJ\text{-}a}} T \leq U$. The following table lists all possible combinations for the last rules of the derivations of $\vdash_{\mathsf{iFJ\text{-}a}} T \leq U$ and $\vdash_{\mathsf{iFJ\text{-}a}} U \leq V$. (We omit the prefix "SUB-ALG-" and the suffix "-IFJ" from the rule names.)

		$\vdash_{\mathsf{iFJ\text{-}a}} U \leq V$				
		REFL	OBJECT	CLASS	CLASS-IFACE	IFACE
$\vdash_{\mathsf{iFJ\text{-}a}} T \leq U$	REFL	✓	✓	✓	✓	✓
	OBJECT	↯	✓	↯	↯	↯
	CLASS	✓	✓	I.H.	I.H.	↯
	CLASS-IFACE	✓	✓	↯	↯	I.H.
	IFACE	✓	✓	↯	↯	I.H.

For the combinations marked with "I.H.", the claim follows directly from the induction hypothesis. Combinations marked with "↯" can never occur, because they put conflicting constraints on the form of U. Combinations marked with "✓" hold obviously. □

Figure C.1 Algorithmic subtyping for iFJ.

$$\boxed{\vdash_{\mathsf{iFJ\text{-}a}} T \leq U}$$

$$\textsc{sub-alg-refl-ifj}\quad \frac{T \neq \mathit{Object}}{\vdash_{\mathsf{iFJ\text{-}a}} T \leq T} \qquad \textsc{sub-alg-object-ifj}\quad \vdash_{\mathsf{iFJ\text{-}a}} T \leq \mathit{Object} \qquad \textsc{sub-alg-class-ifj}\quad \frac{\textbf{class } C \textbf{ extends } N \;\ldots \qquad \vdash_{\mathsf{iFJ\text{-}a}} N \leq D}{\vdash_{\mathsf{iFJ\text{-}a}} C \leq D}$$

$$\textsc{sub-alg-class-iface-ifj}\quad \frac{\vdash_{\mathsf{iFJ\text{-}a}} C \leq D \qquad \textbf{class } D \textbf{ extends } N \textbf{ implements } \overline{J} \;\ldots \qquad \vdash_{\mathsf{iFJ\text{-}a}} J_i \leq I}{\vdash_{\mathsf{iFJ\text{-}a}} C \leq I}$$

$$\textsc{sub-alg-iface-ifj}\quad \frac{\textbf{interface } I \textbf{ extends } \overline{J} \;\ldots \qquad \vdash_{\mathsf{iFJ\text{-}a}} J_i \leq J}{\vdash_{\mathsf{iFJ\text{-}a}} I \leq J}$$

Lemma C.1.3 (Equivalence of declarative and algorithmic subtyping for iFJ). *For all types T and U, it holds that $\vdash_{\mathsf{iFJ}} T \leq U$ if, and only if, $\vdash_{\mathsf{iFJ\text{-}a}} T \leq U$.*

Proof. The claim that $\vdash_{\mathsf{iFJ\text{-}a}} T \leq U$ implies $\vdash_{\mathsf{iFJ}} T \leq U$ follows by a straightforward induction on the derivation of $\vdash_{\mathsf{iFJ\text{-}a}} T \leq U$. The proof of the other implication is straightforward, using Lemma C.1.1 and Lemma C.1.2. □

Lemma C.1.4. *If $\vdash_{\mathsf{iFJ}} T \leq C$ then $T = D$ for some D.*

Proof. By Lemma C.1.3, we may assume $\vdash_{\mathsf{iFJ\text{-}a}} T \leq C$. Then the claim holds obviously. □

Lemma C.1.5. *If $\mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{T\,f}$ and $\vdash_{\mathsf{iFJ}} M \leq N$ then $\mathsf{fields}_{\mathsf{iFJ}}(M) = \overline{T\,f}, \overline{U\,g}$ and $\overline{f}, \overline{g}$ are pairwise disjoint.*

Proof. From $\vdash_{\mathsf{iFJ}} M \leq N$ we get $\vdash_{\mathsf{iFJ\text{-}a}} M \leq N$ by Lemma C.1.3. The claim now follows by induction on the derivation of $\vdash_{\mathsf{iFJ\text{-}a}} M \leq N$, using well-formedness criterion WF-IFJ-3 to show that the field names are disjoint. □

Lemma C.1.6. *If $\mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \mathit{msig}$ and $\vdash_{\mathsf{iFJ}} T' \leq T$ then $\mathsf{mtype}_{\mathsf{iFJ}}(m, T') = \mathit{msig}$.*

Proof. From $\vdash_{\mathsf{iFJ}} T' \leq T$ we get $\vdash_{\mathsf{iFJ\text{-}a}} T' \leq T$ by Lemma C.1.3. If $T = C$ for some C, then $T' = D$ for some D by Lemma C.1.4. In this case, the claim follows by a straightforward induction on the derivation of $\vdash_{\mathsf{iFJ\text{-}a}} D \leq C$, using the premise of rule OK-OVERRIDE-IFJ to ensure that overriding method m preserves its signature.

The case $T = \mathit{Object}$ is impossible because *Object* does not define any methods. Now assume $T = I$ for some I.

- If $T' = I'$ for some I', then the claim follows by a straightforward induction on the derivation of $\vdash_{\mathsf{iFJ\text{-}a}} I' \leq I$, using the premise of rule OK-IDEF-IFJ to ensure that interfaces do not override methods of superinterfaces and that the names of all methods defined in the superinterfaces of some interface are pairwise disjoint.

- If $T' = N$ for some N then we know from $\vdash_{\mathsf{iFJ\text{-}a}} N \leq I$ by inverting SUB-ALG-CLASS-IFACE-IFJ that

$$\vdash_{\mathsf{iFJ\text{-}a}} N \leq C$$
$$\textbf{class } C \textbf{ extends } M \textbf{ implements } \overline{J} \ldots$$
$$\vdash_{\mathsf{iFJ\text{-}a}} J_i \leq I$$

Because the underlying program is well-typed we know $\vdash_{\mathsf{iFJ}} C$ $\mathsf{implements}$ J_i. A straightforward induction shows $\vdash_{\mathsf{iFJ}} C$ $\mathsf{implements}$ I, so $\mathsf{mtype}_{\mathsf{iFJ}}(m, C) = \mathit{msig}$ by inverting rule IMPL-IFACE-IFJ. But we already showed that $\vdash_{\mathsf{iFJ\text{-}a}} N \leq C$ and $\mathsf{mtype}_{\mathsf{iFJ}}(m, C) = \mathit{msig}$ imply $\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = \mathit{msig}$. □

Lemma C.1.7. *If* $\mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \mathit{msig}$ *and* $\mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = \mathit{msig}'\,\{e\}$ *and* $\vdash_{\mathsf{iFJ}} N \leq T$ *then* $\mathit{msig} = \mathit{msig}'$*.*

Proof. An induction on the derivation of $\mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = \mathit{msig}'\,\{e\}$ shows $\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = \mathit{msig}'$. Then $\mathit{msig} = \mathit{msig}'$ follows by Lemma C.1.6. □

Lemma C.1.8. *If* $\mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = \overline{T\,x} \to T\,\{e\}$ *then* $\mathit{this} : N', \overline{x : T} \vdash_{\mathsf{iFJ}} e : T'$ *such that* $\vdash_{\mathsf{iFJ}} N \leq N'$ *and* $\vdash_{\mathsf{iFJ}} T' \leq T$*.*

Proof. We proceed by induction on the derivation of $\mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = \overline{T\,x} \to T\,\{e\}$. If the last rule of this derivation is DYN-MDEF-CLASS-SUPER-IFJ, then the claim follows from the I.H. Otherwise, the last rule is DYN-MDEF-CLASS-BASE-IFJ, so N is a class that defines the method in question. The claim now follows by rules OK-CDEF-IFJ and OK-MDEF-IN-CLASS-IFJ. □

Lemma C.1.9 (Substitution lemma for iFJ)**.** *If* $\Gamma, x : T \vdash_{\mathsf{iFJ}} e : U$ *and* $\Gamma \vdash_{\mathsf{iFJ}} d : T'$ *with* $\vdash_{\mathsf{iFJ}} T' \leq T$ *then* $\Gamma \vdash_{\mathsf{iFJ}} [d/x]e : U'$ *for some* U' *with* $\vdash_{\mathsf{iFJ}} U' \leq U$*.*

Proof. In the following, define $\Gamma' := \Gamma, x : T$. We proceed by induction on the derivation of $\Gamma' \vdash_{\mathsf{iFJ}} e : U$.
Case distinction on the last rule in the derivation of $\Gamma' \vdash_{\mathsf{iFJ}} e : U$.

- *Case* rule EXP-VAR-IFJ: Easy.
- *Case* rule EXP-FIELD-IFJ: Then

$$e = e_0.f_i$$
$$\Gamma' \vdash_{\mathsf{iFJ}} e_0 : C$$
$$\mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\,f}$$
$$U = U_i$$

Applying the I.H., together with Lemma C.1.4, yields

$$\Gamma \vdash_{\mathsf{iFJ}} [d/x]e_0 : C'$$
$$\vdash_{\mathsf{iFJ}} C' \leq C$$

By Lemma C.1.5

$$\mathsf{fields}_{\mathsf{iFJ}}(C') = \overline{U\,f}, \overline{U'\,f'}$$

Applying rule EXP-FIELD-IFJ yields

$$\Gamma \vdash_{\mathsf{iFJ}} [d/x]e_0.f_i : U$$

- *Case* rule EXP-INVOKE-IFJ: Then

$$\begin{gathered} e = e_0.m(\overline{e}) \\ \Gamma' \vdash_{\mathsf{iFJ}} e_0 : T_0 \\ \mathsf{mtype}_{\mathsf{iFJ}}(m, T_0) = \overline{U\,x} \to U \\ (\forall i)\ \Gamma' \vdash_{\mathsf{iFJ}} e_i : T_i \\ (\forall i)\ \vdash_{\mathsf{iFJ}} T_i \leq U_i \end{gathered}$$

  Applying the I.H. yields

$$\begin{gathered} \Gamma \vdash_{\mathsf{iFJ}} [d/x]e_0 : T_0' \\ \vdash_{\mathsf{iFJ}} T_0' \leq T_0 \\ (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} [d/x]e_i : T_i' \\ (\forall i)\ \vdash_{\mathsf{iFJ}} T_i' \leq T_i \end{gathered}$$

  By Lemma C.1.6

$$\mathsf{mtype}_{\mathsf{iFJ}}(m, T_0') = \overline{U\,x} \to U$$

  Moreover, we have $(\forall i)\ \vdash_{\mathsf{iFJ}} T_i' \leq U_i$ by transitivity of subtyping. The claim now follows with rule EXP-INVOKE-IFJ.

- *Case* rule EXP-NEW-IFJ: Then

$$\begin{gathered} e = \textbf{new}\, N(\overline{e}) \\ \mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{U\,f} \\ (\forall i)\ \Gamma' \vdash_{\mathsf{iFJ}} e_i : T_i \\ (\forall i)\ \vdash_{\mathsf{iFJ}} T_i \leq U_i \end{gathered}$$

  Applying the I.H. yields

$$\begin{gathered} (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} [d/x]e_i : T_i' \\ (\forall i)\ \vdash_{\mathsf{iFJ}} T_i' \leq T_i \end{gathered}$$

  By transitivity of subtyping then $(\forall i)\ \vdash_{\mathsf{iFJ}} T_i' \leq U_i$, so the claims follows by rule EXP-NEW-IFJ.

- *Case* rule EXP-CAST-IFJ: Then $e = \textbf{cast}(U, e')$ and $\Gamma' \vdash_{\mathsf{iFJ}} e' : V$. Applying the I.H. yields $\Gamma \vdash_{\mathsf{iFJ}} [d/x]e' : V'$, so the claim follows with rule EXP-CAST-IFJ.

- *Case* rule EXP-GETDICT-IFJ: Then $e = \textbf{getdict}(I, e')$, $U = \mathit{Dict}^I$, and $\Gamma' \vdash_{\mathsf{iFJ}} e' : V$. Applying the I.H. yields $\Gamma \vdash_{\mathsf{iFJ}} [d/x]e' : V'$, so the claim follows with rule EXP-GETDICT-IFJ.

- *Case* rule EXP-LET-IFJ: Then

$$\begin{gathered} e = (\textbf{let}\, V\, y = e_1\, \textbf{in}\, e_2) \\ \Gamma' \vdash_{\mathsf{iFJ}} e_1 : V' \\ \vdash_{\mathsf{iFJ}} V' \leq V \\ \Gamma', y : V \vdash_{\mathsf{iFJ}} e_2 : U \end{gathered}$$

  W.l.o.g., $y \neq x$. Applying the I.H. yields

$$\begin{gathered} \Gamma \vdash_{\mathsf{iFJ}} [d/x]e_1 : V'' \\ \vdash_{\mathsf{iFJ}} V'' \leq V' \\ \Gamma, y : V \vdash_{\mathsf{iFJ}} [d/x]e_2 : U' \\ \vdash_{\mathsf{iFJ}} U' \leq U \end{gathered}$$

  By transitivity of subtyping $\vdash_{\mathsf{iFJ}} V'' \leq V$, so the claim follows with rule EXP-LET-IFJ.

End case distinction on the last rule in the derivation of $\Gamma' \vdash_{\mathsf{iFJ}} e : U$. □

Lemma C.1.10. *If* $\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{e}^n) : T$ *then* $T = N$*. Moreover, if* $\mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{U\, f}^m$ *then* $n = m$ *and* $\Gamma \vdash_{\mathsf{iFJ}} e_i : U_i'$ *with* $\vdash_{\mathsf{iFJ}} U_i' \leq U_i$ *for all* $i \in [n]$*.*

Proof. Follows from inverting rule EXP-NEW-IFJ. □

Lemma C.1.11. *If* $\Gamma \vdash_{\mathsf{iFJ}} v : T$ *and* $\mathsf{unwrap}(v) = \textbf{new}\, N(\overline{w})$ *then* $\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{w}) : N$*.*

Proof. We proceed by induction on the derivation of $\mathsf{unwrap}(v) = \textbf{new}\, N(\overline{w})$. If the last rule in this derivation is UNWRAP-BASE-IFJ, then $v = \mathsf{unwrap}(v)$ and the claim follows with Lemma C.1.10. Otherwise, the last rule is UNWRAP-STEP-IFJ. Hence,

$$\begin{gathered} v = \textbf{new}\ \mathit{Wrap}^I(v') \\ \mathsf{unwrap}(v) = \mathsf{unwrap}(v') \end{gathered}$$

The claim now follows from the I.H. □

Lemma C.1.12 (Preservation for top-level reduction of iFJ)**.** *If* $\emptyset \vdash_{\mathsf{iFJ}} e : T$ *and* $e \longmapsto_{\mathsf{iFJ}} e'$ *then* $\emptyset \vdash_{\mathsf{iFJ}} e' : T'$ *for some* T' *with* $\vdash_{\mathsf{iFJ}} T' \leq T$*.*

Proof. Case distinction on the last rule in the derivation of $\emptyset \vdash_{\mathsf{iFJ}} e : T$.

- *Case* rule EXP-VAR-IFJ: Impossible because there is no reduction rule for variables.
- *Case* rule EXP-FIELD-IFJ: Then

  $$\begin{gathered} e = e_0.f_j \\ \emptyset \vdash_{\mathsf{iFJ}} e_0 : C \\ \mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\, f} \\ T = U_j \end{gathered}$$

  The reduction $e \longmapsto_{\mathsf{iFJ}} e'$ must have been performed through rule DYN-FIELD-IFJ. With Lemma C.1.10 we thus have

  $$\begin{gathered} e_0 = \textbf{new}\, C(\overline{v}) \\ e' = v_j \\ \emptyset \vdash_{\mathsf{iFJ}} v_j : U_j' \\ \vdash_{\mathsf{iFJ}} U_j' \leq U_j \end{gathered}$$

- *Case* rule EXP-INVOKE-IFJ: Then

  $$\begin{gathered} e = e_0.m(\overline{e}) \\ \emptyset \vdash_{\mathsf{iFJ}} e_0 : T_0 \\ \mathsf{mtype}_{\mathsf{iFJ}}(m, T_0) = \overline{U\, x} \to T \\ (\forall i)\ \emptyset \vdash_{\mathsf{iFJ}} e_i : T_i \\ (\forall i)\ \vdash_{\mathsf{iFJ}} T_i \leq U_i \end{gathered}$$

The reduction $e \longmapsto_{\mathsf{iFJ}} e'$ must have been performed through rule DYN-INVOKE-IFJ. With Lemma C.1.10 and Lemma C.1.7 we thus have

$$\begin{aligned} e_0 = v &= \textbf{new}\, N(\overline{w}) \\ T_0 &= N \\ \overline{e} &= \overline{v} \\ \mathsf{getmdef}_{\mathsf{iFJ}}(m, N) &= \overline{U\, x} \to T\,\{d\} \\ e' &= [v/\mathit{this}, \overline{v/x}]d \end{aligned}$$

An application of Lemma C.1.8 yields

$$\begin{aligned} \mathit{this} : N', \overline{x : U} \vdash_{\mathsf{iFJ}} d : T'' \\ \vdash_{\mathsf{iFJ}} T'' \leq T \\ \vdash_{\mathsf{iFJ}} N \leq N' \end{aligned}$$

Repeated applications of Lemma C.1.9 and transitivity of subtyping then yield

$$\begin{aligned} \emptyset \vdash_{\mathsf{iFJ}} [v/\mathit{this}, \overline{v/x}]d : T' \\ \vdash_{\mathsf{iFJ}} T' \leq T \end{aligned}$$

as required.

- *Case* rule EXP-NEW-IFJ: Impossible because there is no matching reduction rule.
- *Case* rule EXP-CAST-IFJ: Then

$$\begin{aligned} e = \textbf{cast}(T, e_0) \\ \emptyset \vdash_{\mathsf{iFJ}} e_0 : U \end{aligned}$$

Case distinction on the rule used to perform the reduction $e \longmapsto_{\mathsf{iFJ}} e'$.

  - *Case* rule DYN-CAST-IFJ: Then

$$\begin{aligned} e_0 &= v \\ \mathsf{unwrap}(v) &= \textbf{new}\, N(\overline{w}) \\ \vdash_{\mathsf{iFJ}} N &\leq T \\ e' &= \textbf{new}\, N(\overline{w}) \end{aligned}$$

  We now get with Lemma C.1.11

$$\emptyset \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{w}) : N$$

  as required.

  - *Case* rule DYN-CAST-WRAP-IFJ: Then

$$\begin{aligned} e_0 &= v \\ T &= I \\ \mathsf{unwrap}(v) &= \textbf{new}\, N(\overline{w}) \\ e' &= \textbf{new}\, \mathit{Wrap}^{I}(\textbf{new}\, N(\overline{w})) \end{aligned}$$

We get with Lemma C.1.11 that

$$\emptyset \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{w}) : N$$

Well-formedness criterion WF-IFJ-6 yields

$$\begin{gathered} \vdash_{\mathsf{iFJ}} \mathit{Wrap}^{I} \leq I \\ \mathsf{fields}_{\mathsf{iFJ}}(\mathit{Wrap}^{I}) = \mathit{Object}\, \mathit{wrapped} \end{gathered}$$

With rule EXP-NEW-IFJ then

$$\emptyset \vdash_{\mathsf{iFJ}} \textbf{new}\, \mathit{Wrap}^{I}(\textbf{new}\, N(\overline{w})) : \mathit{Wrap}^{I}$$

as required.

- *Case* any other rule: Impossible.

End case distinction on the rule used to perform the reduction $e \longmapsto_{\mathsf{iFJ}} e'$.

- *Case* rule EXP-GETDICT-IFJ: Then

$$\begin{gathered} e = \textbf{getdict}(I, e_0) \\ T = \mathit{Dict}^{I} \\ \emptyset \vdash_{\mathsf{iFJ}} e_0 : U \end{gathered}$$

The reduction $e \longmapsto_{\mathsf{iFJ}} e'$ must have been performed through rule DYN-GETDICT-IFJ. Hence

$$\begin{gathered} e_0 = v \\ \mathsf{unwrap}(v) = \textbf{new}\, N(\overline{w}) \\ \mathsf{mindict}_{\mathsf{iFJ}}\{\textbf{class}\ \mathit{Dict}^{I,N'} \ldots \mid \vdash_{\mathsf{iFJ}} N \leq N'\} = M \\ e' = \textbf{new}\, M() \end{gathered}$$

By definition of $\mathsf{mindict}_{\mathsf{iFJ}}$, we know that $M = \mathit{Dict}^{I,N'}$ for some $\mathit{Dict}^{I,N'}$. With well-formedness criterion WF-IFJ-5 we then have

$$\begin{gathered} \mathsf{fields}_{\mathsf{iFJ}}(M) = \bullet \\ \vdash_{\mathsf{iFJ}} M \leq \mathit{Dict}^{I} \end{gathered}$$

Rule EXP-NEW-IFJ yields $\emptyset \vdash_{\mathsf{iFJ}} \textbf{new}\, M() : M$ as required.

- *Case* rule EXP-LET-IFJ: Then

$$\begin{gathered} e = (\textbf{let}\, U\, x = e_1\, \textbf{in}\, e_2) \\ \emptyset \vdash_{\mathsf{iFJ}} e_1 : U' \\ \vdash_{\mathsf{iFJ}} U' \leq U \\ x : U \vdash_{\mathsf{iFJ}} e_2 : T \end{gathered}$$

The reduction $e \longmapsto_{\mathsf{iFJ}} e'$ must have been performed through rule DYN-LET-IFJ. Thus

$$\begin{gathered} e_1 = v \\ e' = [v/x]e_2 \end{gathered}$$

Lemma C.1.9 now yields $\emptyset \vdash_{\mathsf{iFJ}} [v/x]e_2 : T'$ with $\vdash_{\mathsf{iFJ}} T' \leq T$ as required.

End case distinction on the last rule in the derivation of $\emptyset \vdash_{\mathsf{iFJ}} e : T$. □

Proof of Theorem 4.6. From $e \longrightarrow_{\mathsf{iFJ}} e'$ we get by inverting rule DYN-CONTEXT the existence of an evaluation context $\mathcal{E}$ and expressions d, d' such that $e = \mathcal{E}[d]$, $e' = \mathcal{E}[d']$, and $d \longmapsto_{\mathsf{iFJ}} d'$. Thus, it suffices to show the following claim:

If $\emptyset \vdash_{\mathsf{iFJ}} \mathcal{E}[e] : T$ *and* $e \longmapsto_{\mathsf{iFJ}} e'$ *then* $\emptyset \vdash_{\mathsf{iFJ}} \mathcal{E}[e'] : T'$ *with* $\vdash_{\mathsf{iFJ}} T' \leq T$.

The proof of this claim is by induction on the structure of $\mathcal{E}$. If $\mathcal{E} = \square$ then the claim follows by Lemma C.1.12. In all other cases, the form of $\mathcal{E}$ uniquely determines the rule used to derive $\emptyset \vdash_{\mathsf{iFJ}} \mathcal{E}[e] : T$. Using the I.H. and applying the rule in question then proves the claim. If $\mathcal{E} = \mathcal{E}'.f$ or $\mathcal{E} = \mathcal{E}.m(\overline{e})$ then we additionally need Lemma C.1.5 and Lemma C.1.6, respectively. □

C.1.2 Proof of Theorem 4.9

Theorem 4.9 is the progress theorem for iFJ.

Proof of Theorem 4.9. We proceed by induction on the derivation of $\emptyset \vdash_{\mathsf{iFJ}} e : T$.
Case distinction on the last rule in the derivation of $\emptyset \vdash_{\mathsf{iFJ}} e : T$.

- *Case* rule EXP-VAR-IFJ: Impossible.
- *Case* rule EXP-FIELD-IFJ: Then

$$\begin{gathered} e = e_0.f_i \\ \emptyset \vdash_{\mathsf{iFJ}} e_0 : C \\ \mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\, f}^n \\ T = U_i \end{gathered}$$

  - If e_0 is value, we get by Lemma C.1.10

$$e_0 = \textbf{new}\, C(\overline{v}^n)$$

    Thus, $e \longmapsto_{\mathsf{iFJ}} v_i$ by rule DYN-FIELD-IFJ, so $e \longrightarrow_{\mathsf{iFJ}} e'$ by rule DYN-CONTEXT for $e' := v_i$.
  - If e_0 is not a value then we get from the I.H. that either $e_0 \longrightarrow_{\mathsf{iFJ}} e_0'$ or that e_0 is stuck on a bad cast or a bad dictionary lookup. The claim now follows easily by constructing an appropriate evaluation context.
- *Case* rule EXP-INVOKE-IFJ: Then

$$\begin{gathered} e = e_0.m(\overline{e}) \\ \emptyset \vdash_{\mathsf{iFJ}} e_0 : T_0 \\ \mathsf{mtype}_{\mathsf{iFJ}}(m, T_0) = \overline{U\, x} \to T \\ (\forall i)\ \emptyset \vdash_{\mathsf{iFJ}} e_i : T_i \\ (\forall i)\ \ \vdash_{\mathsf{iFJ}} T_i \leq U_i \end{gathered}$$

  - If e_0 and all e_i are values then we get with Lemma C.1.10 and the fact that $\mathsf{mtype}_{\mathsf{iFJ}}$ is undefined for *Object*

$$\begin{gathered} e_0 = v_0 = \textbf{new}\, C_0(\overline{w_0}) \\ T_0 = C_0 \\ \overline{e} = \overline{v} \end{gathered}$$

By Lemma C.1.7 then

$$\mathsf{getmdef}_{\mathsf{iFJ}}(m, C_0) = \overline{U\,x} \to T\,\{d\}$$

The claim now follows by setting $\mathcal{E} := \square$ and using rules DYN-INVOKE-IFJ in combination with DYN-CONTEXT to derive $v_0.m(\overline{v}) \longrightarrow_{\mathsf{iFJ}} [v_0/\mathit{this}, \overline{v/x}]d$.

- If e_0 or one of the e_i is not a value then the claim follows from the I.H. by constructing an appropriate evaluation context.

- *Case* rule EXP-NEW-IFJ: Then $e = \textbf{new}\,N(\overline{e})$. If all e_i are values then e is a value. Otherwise, the claim follows from the I.H. by constructing an appropriate evaluation context.
- *Case* rule EXP-CAST-IFJ: Then

$$e = \textbf{cast}(T, e_0)$$
$$\emptyset \vdash_{\mathsf{iFJ}} e_0 : U$$

- Assume $e_0 = v$ for some value v. Obviously, $\mathsf{unwrap}(v) = \textbf{new}\,N(\overline{w})$ for some N and some $\overline{w}$.
  - ∗ If $\vdash_{\mathsf{iFJ}} N \leq T$ then $e \longrightarrow_{\mathsf{iFJ}} \textbf{new}\,N(\overline{w})$ by rules DYN-CAST-IFJ and DYN-CONTEXT-IFJ.
  - ∗ If not $\vdash_{\mathsf{iFJ}} N \leq T$ but $T = I$ and there exists a dictionary class $\mathit{Dict}^{I,M}$ such that $\vdash_{\mathsf{iFJ}} N \leq M$, then $e \longrightarrow_{\mathsf{iFJ}} \textbf{new}\,\mathit{Wrap}^{I}(\textbf{new}\,N(\overline{w}))$ by rules DYN-CAST-WRAP-IFJ and DYN-CONTEXT-IFJ.
  - ∗ Otherwise, e is stuck on a bad cast by Definition 4.7 for $\mathcal{E} = \square$.
- If e_0 is not a value, then the claim follows from the I.H. by constructing an appropriate evaluation context.

- *Case* rule EXP-GETDICT-IFJ: Then

$$e = \textbf{getdict}(I, e_0)$$
$$\emptyset \vdash_{\mathsf{iFJ}} e_0 : U$$

- Assume $e_0 = v$ for some value v. Obviously, $\mathsf{unwrap}(v) = \textbf{new}\,N(\overline{w})$ for some N and some $\overline{w}$. Define

$$\mathscr{M} := \{\textbf{class}\ \mathit{Dict}^{I,N'}\ \ldots \mid \vdash_{\mathsf{iFJ}} N \leq N'\}$$

If $\mathsf{mindict}_{\mathsf{iFJ}}\mathscr{M}$ is undefined, then e is stuck on a bad dictionary lookup by Definition 4.8 with $\mathcal{E} = \square$. Otherwise, $\mathsf{mindict}_{\mathsf{iFJ}}\mathscr{M} = M$ for some M, so $e \longrightarrow_{\mathsf{iFJ}} \textbf{new}\,M()$ by rules DYN-GETDICT-IFJ and DYN-CONTEXT-IFJ.

- If e_0 is not a value, then the claim follows from the I.H. by constructing an appropriate evaluation context.

- *Case* rule EXP-LET-IFJ: Then

$$e = (\textbf{let}\,U\,x = e_1\,\textbf{in}\,e_2)$$
$$\emptyset \vdash_{\mathsf{iFJ}} e_1 : U'$$

- If e_1 is a value, then $e \longrightarrow_{\mathsf{iFJ}} [e_1/x]e_2$ follows by rules DYN-LET-IFJ and DYN-CONTEXT-IFJ.
- If e_1 is not a value, then the claim follows from the I.H. by constructing an appropriate evaluation context.

End case distinction on the last rule in the derivation of $\emptyset \vdash_{\mathsf{iFJ}} e : T$. □

C.2 Translation Preserves Static Semantics

This section shows that the translation from $\mathsf{CoreGI}^\flat$ to iFJ preserves the static semantics. It includes the proofs for Theorem 4.12 (translation preserves types of expressions) and Theorem 4.12 (translation preserves well-formedness of programs). Each lemma in this section mentioning both $\mathsf{CoreGI}^\flat$ and iFJ constructs makes the implicit assumption that the underlying iFJ program is the translation of the underlying $\mathsf{CoreGI}^\flat$ program.

C.2.1 Proof of Theorem 4.11

Theorem 4.11 states that the translation from $\mathsf{CoreGI}^\flat$ to iFJ preserves the types of expressions.

Lemma C.2.1. *If* $\vdash^{\flat'} T \leq U$ *then* $\vdash_{\mathsf{iFJ}} T \leq U$*.*

Proof. Straightforward rule inductions show that $C \trianglelefteq^\flat_{\mathsf{c}} D$ and $I \trianglelefteq^\flat_{\mathsf{i}} J$ imply $\vdash_{\mathsf{iFJ}} C \leq D$ and $\vdash_{\mathsf{iFJ}} I \leq J$, respectively. The original claim then follows by case distinction on the last rule in the derivation of $\vdash^{\flat'} T \leq U$. □

Lemma C.2.2. *If* $\vdash^\flat T \leq U \leadsto \mathsf{nil}$ *then* $\vdash_{\mathsf{iFJ}} T \leq U$*.*

Proof. The last rule in the derivation of $\vdash^\flat T \leq U \leadsto \mathsf{nil}$ must be SUB-KERNEL$^\flat$. Inverting the rule yields $\vdash^{\flat'} T \leq U$. The claim now follows with Lemma C.2.1. □

Lemma C.2.3. *If* $\vdash^\flat T \leq U \leadsto I$ *then* $U = I$*.*

Proof. Obvious. □

Lemma C.2.4. *If* $\Gamma \vdash_{\mathsf{iFJ}} e : T$ *and* $\vdash^\flat T \leq U \leadsto I^?$ *then* $\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e) : U'$ *for some* U' *with* $\vdash_{\mathsf{iFJ}} U' \leq U$*.*

Proof. Case distinction on the form of $I^?$.

- *Case* $I^? = \mathsf{nil}$: Then, by Lemma C.2.2, $\vdash_{\mathsf{iFJ}} T \leq U$. Moreover, $\mathsf{wrap}(I^?, e) = e$. Defining $U' := T$ finishes this case.
- *Case* $I^? = I$: By Lemma C.2.3 we have $U = I$. Moreover, $\mathsf{wrap}(I^?, e) = \textbf{new}\ \mathit{Wrap}^I(e)$. By Convention 4.4, examining rule OK-IDEF$^\flat$, and applying rule EXP-NEW-IFJ, we now get

$$\Gamma \vdash \mathsf{wrap}(I^?, e) : \mathit{Wrap}^I$$
$$\vdash_{\mathsf{iFJ}} \mathit{Wrap}^I \leq I$$

  Defining $U' := \mathit{Wrap}^I$ finishes this case.

End case distinction on the form of $I^?$. □

Lemma C.2.5. *If* $\vdash^\flat T \leq I \leadsto \mathsf{nil}$ *then* $T = J$ *for some* J *with* $J \trianglelefteq^\flat_{\mathsf{i}} I$*.*

Proof. The derivation of $\vdash^\flat T \leq I \leadsto \mathsf{nil}$ must end with rule SUB-KERNEL$^\flat$. Thus, $\vdash^{\flat'} T \leq I$. The last rule in this derivation must be SUB-IFACE$^\flat$, so the claim holds by inverting this rule. □

Lemma C.2.6. *If* $\mathsf{mtype}^\flat(m, T) = \mathit{msig} \leadsto \mathsf{nil}$ *then* $\mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \mathit{msig}$*.*

Proof. Case distinction on the form of m.

- *Case* $m = m^{\mathrm{c}}$: We proceed by induction on the derivation of $\mathsf{mtype}^{\flat}(m, T) = \mathit{msig} \leadsto \mathsf{nil}$. The last rule of this derivation cannot be MTYPE-IFACE$^{\flat}$ because this rule requires $m = m^{\mathrm{i}}$. If the last is MTYPE-CLASS-SUPER$^{\flat}$, then the claim follows from the I.H. and rule MTYPE-CLASS-SUPER-IFJ. If the last rule is MTYPE-CLASS-BASE$^{\flat}$, then the claim follows by rule MTYPE-CLASS-BASE-IFJ because the translation from $\mathsf{CoreGI}^{\flat}$ to iFJ leaves signatures of class methods unchanged.

- *Case* $m = m^{\mathrm{i}}$: Thus, the derivation of $\mathsf{mtype}^{\flat}(m, T) = \mathit{msig} \leadsto \mathsf{nil}$ ends with rule MTYPE-IFACE$^{\flat}$. Inverting the rules yields

  $$\textbf{interface } I \textbf{ extends } \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\}$$
  $$\vdash^{\flat} T \le I \leadsto \mathsf{nil}$$
  $$m = m_k$$
  $$\mathit{msig} = \mathit{msig}_k$$

  By Lemma C.2.5, we get $T = I'$ for some I' with $I' \trianglelefteq^{\flat}_{\mathrm{i}} I$. Convention 4.2 ensures that I is the only interface defining m. Moreover, the translation from $\mathsf{CoreGI}^{\flat}$ to iFJ leaves signatures of interface methods unchanged. An easy induction on the derivation of $I' \trianglelefteq^{\flat}_{\mathrm{i}} I$ then shows that $\mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \mathit{msig}$.

End case distinction on the form of m. □

Lemma C.2.7. *If* $\mathsf{fields}^{\flat}(N) = \overline{T\,f}$ *then* $\mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{T\,f}$*.*

Proof. Straightforward induction on the derivation of $\mathsf{fields}^{\flat}(N) = \overline{T\,f}$, using the fact that the translation from $\mathsf{CoreGI}^{\flat}$ to iFJ neither changes the types of fields nor the superclass of some class. □

Lemma C.2.8. *If* $\mathsf{mtype}^{\flat}(m, T) = \mathit{msig} \leadsto I$ *then* $\vdash^{\flat} T \le I \leadsto I$*,* $\mathsf{mtype}_{\mathsf{iFJ}}(m, I) = \mathit{msig}$*, and interface* I *contains a definition of method* m*.*

Proof. The derivation of $\mathsf{mtype}^{\flat}(m, T) = \mathit{msig} \leadsto I$ ends with rule MTYPE-IFACE$^{\flat}$. Inverting the rule, together with Lemma C.2.3, yields

$$\textbf{interface } I \textbf{ extends } \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\}$$
$$\vdash^{\flat} T \le I \leadsto I$$
$$m = m_k$$
$$\mathit{msig} = \mathit{msig}_k$$

Looking at rule OK-IDEF$^{\flat}$, it is easy to verify that $\mathsf{mtype}_{\mathsf{iFJ}}(m, I) = \mathit{msig}$. □

Proof of Theorem 4.11. We perform induction on the derivation of $\Gamma \vdash^{\flat} e : T \leadsto e'$.
Case distinction on the last rule in the derivation of $\Gamma \vdash^{\flat} e : T \leadsto e'$.

- *Case* rule EXP-VAR$^{\flat}$: Obvious.

- *Case* rule EXP-FIELD$^{\flat}$: Follows from the I.H., Lemma C.2.7, and an application of rule EXP-FIELD-IFJ.

- *Case* rule EXP-INVOKE$^\flat$: Then

$$\begin{gathered}
e = e_0.m(\overline{e}) \\
\Gamma \vdash^\flat e_0 : T_0 \rightsquigarrow e'_0 \\
\mathsf{mtype}^\flat(m, T_0) = \overline{U\,x} \to T \rightsquigarrow I^? \\
(\forall i)\ \Gamma \vdash^\flat e_i : T_i \rightsquigarrow e'_i \\
(\forall i)\ \vdash^\flat T_i \leq U_i \rightsquigarrow J^?_i \\
e''_0 = \mathsf{wrap}(I^?, e'_0) \\
(\forall i)\ e''_i = \mathsf{wrap}(J^?_i, e'_i) \\
e' = e''_0.m(\overline{e''})
\end{gathered} \tag{C.2.1}$$

Applying the I.H. yields

$$\begin{gathered}
\Gamma \vdash_{\mathsf{iFJ}} e'_0 : T_0 \\
(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e'_i : T_i
\end{gathered} \tag{C.2.2}$$

With Lemma C.2.4 we then get for some $\overline{T'}$ that

$$\begin{gathered}
(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e''_i : T'_i \\
(\forall i)\ \vdash_{\mathsf{iFJ}} T'_i \leq U_i
\end{gathered}$$

Case distinction on the form of $I^?$.

  - *Case* $I^? = \mathsf{nil}$: Then $e''_0 = e'_0$. Moreover, by Lemma C.2.6

$$\mathsf{mtype}_{\mathsf{iFJ}}(m, T_0) = \overline{U\,x} \to T$$

    The claim now follows with rule EXP-INVOKE-IFJ.

  - *Case* $I^? \neq \mathsf{nil}$: Then $I^? = I$ for some I. With (C.2.2), an examination of rule OK-IDEF$^\flat$, and rule EXP-NEW-IFJ then

$$\begin{gathered}
\Gamma \vdash_{\mathsf{iFJ}} e''_0 : \mathit{Wrap}^I \\
\vdash_{\mathsf{iFJ}} \mathit{Wrap}^I \leq I
\end{gathered}$$

    We have with (C.2.1) and Lemma C.2.8 that $\mathsf{mtype}_{\mathsf{iFJ}}(m, I) = \overline{U\,x} \to T$. An application of Lemma C.1.6 yields

$$\mathsf{mtype}_{\mathsf{iFJ}}(m, \mathit{Wrap}^I) = \overline{U\,x} \to T$$

    The claim now follows with rule EXP-INVOKE-IFJ.

End case distinction on the form of $I^?$.

- *Case* rule EXP-NEW$^\flat$: Then

$$\begin{gathered}
e = \mathbf{new}\ N(\overline{e}) \\
T = N \\
\mathsf{fields}^\flat(N) = \overline{U\,f} \\
(\forall i)\ \Gamma \vdash^\flat e_i : T_i \rightsquigarrow e'_i \\
(\forall i)\ \vdash^\flat T_i \leq U_i \rightsquigarrow J^?_i \\
(\forall i)\ e''_i = \mathsf{wrap}(J^?_i, e'_i) \\
e' = \mathbf{new}\ N(\overline{e''})
\end{gathered}$$

Applying the I.H. yields $(\forall i)\ \Gamma \vdash^{\flat} e'_i : T_i$, so with Lemma C.2.4

$$
\begin{gathered}
(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e''_i : U'_i \\
(\forall i)\ \ \vdash_{\mathsf{iFJ}} U''_i \leq U_i
\end{gathered}
$$

With Lemma C.2.7

$$\mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{U\, f}$$

The claim now follows with rule EXP-NEW-IFJ.

- *Case* rule EXP-CAST$^{\flat}$: Follows from the I.H. and rule EXP-CAST-IFJ.

End case distinction on the last rule in the derivation of $\Gamma \vdash^{\flat} e : T \rightsquigarrow e'$. □

C.2.2 Proof of Theorem 4.12

Theorem 4.12 postulates that the translation from CoreGI$^{\flat}$ to iFJ preserves well-formedness of programs.

Lemma C.2.9. *If* $\vdash^{\flat}$ *prog* ok $\rightsquigarrow$ *prog′ then prog′ fulfills all well-formedness criteria for iFJ programs from Figure 4.13.*

Proof. Easy. □

Lemma C.2.10. *If* N *is an iFJ class that results as the translation of a CoreGI$^{\flat}$ class, then* $\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = msig$ *implies* $\mathsf{mtype}^{\flat}(m, N) = msig \rightsquigarrow \mathsf{nil}$.

Proof. The translation from CoreGI$^{\flat}$ to iFJ neither changes the superclass nor the method names of a class. An induction on the derivation of $\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = msig$ then shows $\mathsf{mtype}^{\flat}(m, N) = msig' \rightsquigarrow \mathsf{nil}$ for some $msig'$. With Lemma C.2.6 and Lemma C.1.6 then $msig' = msig$. □

Lemma C.2.11. *If* $\mathsf{override\text{-}ok}^{\flat}(m : msig, C)$ *then* $\mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m : msig, C)$.

Proof. Assume

$$
\begin{gathered}
\textbf{class } C \textbf{ extends } N \ldots \\
\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = msig'
\end{gathered}
$$

Because *Object* does not define any methods, $N \neq Object$. If N is the translation of a CoreGI$^{\flat}$ class, then we have with Lemma C.2.10 that $\mathsf{mtype}^{\flat}(m, N) = msig' \rightsquigarrow \mathsf{nil}$. The premise of rule OK-OVERRIDE$^{\flat}$ then yields $msig = msig'$. The claim now follows with rule OK-OVERRIDE-IFJ.

If N is not the translation of a CoreGI$^{\flat}$ class, then N is either a wrapper class of the form $Wrap^{I}$ or a dictionary class of the form $Dict^{I,M}$. But the translation from CoreGI$^{\flat}$ to iFJ never uses such classes as superclasses of other classes, so we obtain a contradiction. □

Lemma C.2.12. *If* $\vdash^{\flat} m : mdef$ ok in $C \rightsquigarrow mdef'$ *then* $\vdash_{\mathsf{iFJ}} mdef'$ ok in C.

Proof. Assume

$$
\begin{aligned}
mdef &= msig\,\{e\} \\
msig &= \overline{T\,x} \to T
\end{aligned}
$$

Figure C.2 Interface implementation through methods.

$$\boxed{\vdash_{\mathsf{iFJ}} \overline{m : \mathit{mdef}} \ \mathsf{implements}\ I}$$

IMPL-IFACE-METHODS-IFJ

$$\frac{\begin{array}{c}\textbf{interface}\ I\ \textbf{extends}\ \overline{J}^{l}\ \{\ \overline{m' : \mathit{msig}'}^{k}\ \} \qquad (\forall i \in [l])\ \overline{m : \mathit{msig}\,\{\,e\,\}}^{n}\ \mathsf{implements}\ J_i \\ (\forall i \in [k] \exists j \in [n])\ m'_i = m_j \text{ and } \mathit{msig}'_i = \mathit{msig}_j\end{array}}{\vdash_{\mathsf{iFJ}} \overline{m : \mathit{msig}\,\{\,e\,\}}^{n}\ \mathsf{implements}\ I}$$

Inverting rule OK-MDEF-IN-CLASS$^\flat$ and OK-MDEF$^\flat$ yields

$$\begin{gathered}\mathit{mdef}' = \mathit{msig}\,\{e'\} \\ \mathsf{override\text{-}ok}^\flat(m : \mathit{msig}, C) \\ \underbrace{\mathit{this} : C, \overline{x : T}}_{=:\Gamma} \vdash^\flat e : T' \rightsquigarrow e'' \\ \vdash^\flat T' \le T \rightsquigarrow I^? \\ e' = \mathsf{wrap}(I^?, e'')\end{gathered}$$

By Theorem 4.11 and Lemma C.2.4

$$\begin{gathered}\Gamma \vdash_{\mathsf{iFJ}} e' : T'' \\ \vdash_{\mathsf{iFJ}} T'' \le T\end{gathered}$$

With Lemma C.2.11 also $\mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m : \mathit{msig}, C)$. The claim now follows by applying rule OK-MDEF-IN-CLASS-IFJ. □

Lemma C.2.13. *If* $\vdash^\flat \mathit{cdef}\ \mathsf{ok} \rightsquigarrow \mathit{cdef}'$ *then* $\vdash_{\mathsf{iFJ}} \mathit{cdef}'\ \mathsf{ok}$.

Proof. Follows easily with Lemma C.2.12. □

Figure C.2 defines the auxiliary relation $\vdash_{\mathsf{iFJ}} \overline{m : \mathit{mdef}}\ \mathsf{implements}\ I$, which asserts that all methods of I are implemented by some method in $\overline{m : \mathit{mdef}}$.

Lemma C.2.14. *If* $\vdash_{\mathsf{iFJ}} \overline{m : \mathit{mdef}}\ \mathsf{implements}\ I$ *and a class* C *defines all methods in* $\overline{m : \mathit{mdef}}$*, then* $\vdash_{\mathsf{iFJ}} C\ \mathsf{implements}\ I$.

Proof. Easy induction on the derivation of $\vdash_{\mathsf{iFJ}} \overline{m : \mathit{mdef}}\ \mathsf{implements}\ I$. □

Lemma C.2.15. *If* $\mathsf{wrapper\text{-}methods}(I) = \overline{m : \mathit{mdef}}$ *then* $\vdash_{\mathsf{iFJ}} \overline{m : \mathit{mdef}}\ \mathsf{implements}\ I$.

Proof. Easy induction on the derivation of $\mathsf{wrapper\text{-}methods}(I) = \overline{m : \mathit{mdef}}$. □

Lemma C.2.16. *If a* $\mathsf{CoreGI}^\flat$ *interface* I *defines a method* m *with signature* $\overline{T\,x} \to T$ *then* $\mathsf{mtype}_{\mathsf{iFJ}}(m, \mathit{Dict}^I) = \mathit{Object}\,y, \overline{T\,x} \to T$.

Proof. Obvious by inverting rule OK-IDEF$^\flat$. □

Lemma C.2.17. *If* $\mathsf{wrapper\text{-}methods}(I) = \overline{m : \mathit{mdef}}^{n}$ *and* $i \in [n]$ *then* $\vdash_{\mathsf{iFJ}} m_i : \mathit{mdef}_i\ \mathsf{ok\,in}\,C$ *for all classes* C *that have* Object *as their superclass and* $\mathsf{fields}_{\mathsf{iFJ}}(C) = \mathit{Object}\,\mathit{wrapped}$.

Proof. We proceed by induction on the derivation of $\mathsf{wrapper\text{-}methods}(I) = \overline{m : \mathit{mdef}}$. Inverting rule WRAPPER-METHODS$^\flat$ yields

$$\begin{gathered}
\textbf{interface}\ I\ \textbf{extends}\ \overline{J}^{l}\ \{\, \overline{m' : \mathit{msig}}^{k}\ \} \\
(\forall j \in [k])\ \mathit{msig}_j = \overline{T\,x} \to U \\
(\forall j \in [k])\ \mathit{mdef}'_j = \overline{T\,x} \to\ U\{\, \textbf{getdict}(I, \mathit{this.wrapped}).m'_j(\mathit{this.wrapped}, \overline{x})\} \\
\overline{m : \mathit{mdef}} = \overline{m' : \mathit{mdef}'}^{k}\ \mathsf{wrapper\text{-}methods}(J_1)\ldots\mathsf{wrapper\text{-}methods}(J_l)
\end{gathered}$$

We need to consider two cases:

- If $i > k$ then $m_i : \mathit{mdef}_i \in \mathsf{wrapper\text{-}methods}(J_p)$ for some $p \in [l]$. In this case, applying the I.H. yields the desired result.
- If $i \leq k$ then $m_i : \mathit{mdef}_i = m'_i : \mathit{mdef}'_i$. Assume

  $$m_i : \mathit{mdef}_i = m : \mathit{mdef} = m : \mathit{msig}\,\{e\} = m : \overline{T\,x} \to U\,\{e\}$$

  and suppose C is a class with *Object* as its superclass and $\mathsf{fields}_{\mathsf{iFJ}}(C) = \mathit{Object\ wrapped}$. Obviously,

  $$\mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m : \mathit{msig}, C)$$

  by rule OK-OVERRIDE-IFJ. Moreover, we have for $\Gamma := \mathit{this} : C, \overline{x : T}$ that

  $$\Gamma \vdash_{\mathsf{iFJ}} \mathit{this.wrapped} : \mathit{Object}$$

  by rules EXP-VAR-IFJ and EXP-FIELD-IFJ. Hence, with rule EXP-GETDICT-IFJ then

  $$\Gamma \vdash_{\mathsf{iFJ}} \textbf{getdict}(I, \mathit{this.wrapped}) : \mathit{Dict}^I$$

  By Lemma C.2.16

  $$\mathsf{mtype}_{\mathsf{iFJ}}(m, \mathit{Dict}^I) = \mathit{Object}\,y, \overline{T\,x} \to U$$

  By rule EXP-VAR-IFJ also $\Gamma \vdash_{\mathsf{iFJ}} x_i : T_i$ for all suitable i. Thus, with rule EXP-INVOKE-IFJ

  $$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{\textbf{getdict}(I, \mathit{this.wrapped}).m(\mathit{this.wrapped}, \overline{x})}_{=e} : U$$

  With reflexivity of subtyping we now get

  $$\vdash_{\mathsf{iFJ}} m_i : \mathit{mdef}_i\ \mathsf{ok\,in}\ C$$

  as required. □

Lemma C.2.18. *If* $\vdash^\flat \mathit{idef}\ \mathsf{ok} \leadsto \overline{\mathit{def}}^{n}$ *then* $\vdash_{\mathsf{iFJ}} \mathit{def}_i\ \mathsf{ok}$ *for all* $i \in [n]$.

Proof. Assume

$$\mathit{idef} = \textbf{interface}\ I\ \textbf{extends}\ \overline{J}\,\{\, \overline{m : \mathit{msig}}\,\}$$

Then $\overline{\mathit{def}} = \mathit{idef}_1, \mathit{idef}_2, \mathit{cdef}$ where

$$\begin{aligned}
\mathit{idef}_1 &= \textbf{interface}\ I\ \textbf{extends}\ \overline{J}\{\, \overline{m : \mathit{msig}}\,\} \\
\mathit{idef}_2 &= \textbf{interface}\ \mathit{Dict}^I\ \textbf{extends}\ \overline{\mathit{Dict}^J}\,\{\, \overline{m : \mathit{Object}\ y, \mathit{msig}}\,\} \\
\mathit{cdef} &= \textbf{class}\ \mathit{Wrap}^I\ \textbf{extends}\ \mathit{Object}\ \textbf{implements}\ I\{\, \mathit{Object\ wrapped} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{wrapper\text{-}methods}(I)\,\}
\end{aligned}$$

With Convention 4.2 we immediately get that $\vdash_{\mathsf{iFJ}} \mathit{idef}_1\ \mathsf{ok}$ and $\vdash_{\mathsf{iFJ}} \mathit{idef}_2\ \mathsf{ok}$. With Lemma C.2.14 and Lemma C.2.15 we get

$$\vdash_{\mathsf{iFJ}} \mathit{Wrap}^I\ \mathsf{implements}\ I$$

Obviously, Wrap^I fulfills the condition required by Lemma C.2.17. Thus, we have

$$\vdash_{\mathsf{iFJ}} m : \mathit{mdef}\ \mathsf{ok\,in}\ \mathit{Wrap}^I$$

for all methods $m : \mathit{mdef}$ of Wrap^I. Now we get $\vdash_{\mathsf{iFJ}} \mathit{cdef}\ \mathsf{ok}$ by rule OK-CDEF-IFJ. □

Lemma C.2.19. *If* $\mathsf{mtype}_{\mathsf{iFJ}}(m, I) = \mathit{msig}$ *then* $\mathsf{mtype}_{\mathsf{iFJ}}(m, \mathit{Dict}^I) = \mathit{Object}\, y, \mathit{msig}$.

Proof. By Convention 4.4, implementation interfaces such as Dict^I are not part of standalone iFJ programs, so the underlying iFJ program must be in the image of the translation from CoreGI$^\flat$. Moreover, implementation interfaces are only generated for interfaces originally contained in the CoreGI$^\flat$ program. Thus, the iFJ interface I is the translation of a CoreGI$^\flat$ interface.

We proceed by induction on the derivation of $\mathsf{mtype}_{\mathsf{iFJ}}(m, I) = \mathit{msig}$. If the last rule in the derivation is MTYPE-IFACE-BASE-IFJ then the claim follows immediate by rule OK-IDEF-IFJ. Otherwise, the last rule in the derivation is MTYPE-IFACE-SUPER-IFJ. Hence, I does not define method m and $\mathsf{mtype}_{\mathsf{iFJ}}(m, J) = \mathit{msig}$ for some direct superinterface J of I. Applying the I.H. yields $\mathsf{mtype}_{\mathsf{iFJ}}(m, \mathit{Dict}^J) = \mathit{Object}\, y, \mathit{msig}$. Because I is the translation of a CoreGI$^\flat$ interface, we know with Convention 4.2 that J is unique; that is, no other superinterface of I contains a definition of m. Because I also does not define m, we get by examining rule OK-IDEF$^\flat$ that Dict^I does not define m. Hence, applying rule MTYPE-IFACE-SUPER-IFJ yields the desired result. □

Lemma C.2.20. *If* $\mathsf{dict\text{-}methods}(I) = \overline{m : \mathit{mdef}}^n$ *and* $i \in [n]$ *and* C *is a class with* Object *as its superclass, then* $\vdash_{\mathsf{iFJ}} m_i : \mathit{mdef}_i\ \mathsf{ok\,in}\ C$.

Proof. We proceed by induction on the derivation of $\mathsf{dict\text{-}methods}(I) = \overline{m : \mathit{mdef}}^n$. We have

$$\textbf{interface}\ I\ \textbf{extends}\ \overline{J}^l\ \{\overline{m' : \mathit{msig}'}^k\}$$
$$(\forall i \in [k])\ \mathit{msig}'_i = \overline{T\,x} \to U \text{ and } \mathit{mdef}'_i = \mathit{Object}\ y, \overline{T\,x} \to U\{\,\textbf{getdict}(I, y).m'_i(y, \overline{x})\}$$
$$\overline{m : \mathit{mdef}}^n = \overline{m' : \mathit{mdef}'}^k\ \mathsf{dict\text{-}methods}(J_1) \ldots \mathsf{dict\text{-}methods}(J_l)$$

If $i > k$ then the claim follows by the I.H. Thus, assume $i \leq k$ and suppose

$$m_i : \mathit{mdef}_i = m'_i : \mathit{mdef}'_i = m'_i : \mathit{Object}\ y, \overline{T\,x} \to U\,\{\,\textbf{getdict}(I, y).m'_i(y, \overline{x})\}$$

Define $\Gamma := \mathit{this} : C, y : \mathit{Object}, \overline{x : T}$. Then $\Gamma \vdash_{\mathsf{iFJ}} \textbf{getdict}(I, y) : \mathit{Dict}^I$ by rules EXP-GETDICT-IFJ and EXP-VAR-IFJ. Obviously, $\mathsf{mtype}_{\mathsf{iFJ}}(m_i, I) = \overline{T\,x} \to U$, so with Lemma C.2.19

$$\mathsf{mtype}_{\mathsf{iFJ}}(m_i, \mathit{Dict}^I) = \mathit{Object}\ y, \overline{T\,x} \to U$$

Using rule EXP-VAR-IFJ, reflexivity of subtyping, and rule EXP-INVOKE-IFJ, we then get

$$\Gamma \vdash_{\mathsf{iFJ}} \textbf{getdict}(I, y).m'_i(y, \overline{x}) : U$$

Because Object is the superclass of C, we also have $\mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m_i : \mathit{Object}\ y, \overline{T\,x} \to U)$. Thus, with reflexivity of subtyping and rule OK-MDEF-IN-CLASS-IFJ

$$\vdash_{\mathsf{iFJ}} m_i : \mathit{mdef}_i\ \mathsf{ok\,in}\ C$$

as required. □

The notation $\Gamma \subseteq \Gamma'$ asserts that $x : T \in \Gamma$ implies $x : T \in \Gamma'$.

Lemma C.2.21 (Weakening for iFJ). *If* $\Gamma \vdash_{\mathsf{iFJ}} e : T$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma' \vdash_{\mathsf{iFJ}} e : T$.

Proof. Straightforward induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e : T$ □

Lemma C.2.22. *If* $\mathit{this} : N \vdash^{\flat} \mathit{mdef}$ $\mathsf{implements}$ $\mathit{msig} \rightsquigarrow \mathit{mdef}'$ *then* $\vdash_{\mathsf{iFJ}} m : \mathit{mdef}'$ $\mathsf{ok\,in}\,C$ *for any* m *and any class* C *with* Object *as its superclass.*

Proof. Define

$$\Gamma := \mathit{this} : N, \overline{x : T}$$

and choose $\overline{T}$, $\overline{x}$, U, and e such that

$$\mathit{mdef} = \overline{T\,x} \to U\,\{e\}$$

Then we get by inverting rules IMPL-METH$^\flat$ and OK-MDEF$^\flat$ that

$$\Gamma \vdash^{\flat} e : U' \rightsquigarrow e'$$
$$\vdash^{\flat} U' \leq U \rightsquigarrow I^?$$
$$d := \mathsf{wrap}(I^?, e')$$
$$\mathit{mdef}' = \mathit{Object}\,y, \overline{T\,x} \to U\,\{\mathbf{let}\,N\,z = \mathbf{cast}(N, y)\,\mathbf{in}\,[z/\mathit{this}]d\}$$
$$y, z \text{ fresh}$$

By Theorem 4.11

$$\Gamma \vdash_{\mathsf{iFJ}} e' : U'$$

so with Lemma C.2.4 and Lemma C.2.21

$$\Gamma, z : N \vdash_{\mathsf{iFJ}} d : U''$$
$$\vdash_{\mathsf{iFJ}} U'' \leq U$$

By Lemma C.1.9 then

$$\overline{x : T}, z : N \vdash_{\mathsf{iFJ}} [z/\mathit{this}]d : U'''$$
$$\vdash_{\mathsf{iFJ}} U''' \leq U''$$

With Lemma C.2.21 then for any class C

$$\underbrace{\mathit{this} : C, y : \mathit{Object}, \overline{x : T}}_{=:\Gamma'}, z : N \vdash_{\mathsf{iFJ}} [z/\mathit{this}]d : U'''$$

Moreover, with rules EXP-VAR-IFJ and EXP-CAST-IFJ

$$\Gamma' \vdash_{\mathsf{iFJ}} \mathbf{cast}(N, y) : N$$

Thus, with reflexivity of subtyping and rule EXP-LET-IFJ

$$\Gamma' \vdash_{\mathsf{iFJ}} \mathbf{let}\,N\,z = \mathbf{cast}(N, y)\,\mathbf{in}\,[z/\mathit{this}]d : U'''$$

By transitivity of subtyping

$$\vdash_{\mathsf{iFJ}} U''' \leq U$$

If we additionally assume that C's superclass is Object, then by rule OK-OVERRIDE-IFJ

$$\mathsf{override\text{-}ok}_{\mathsf{iFJ}}(m : \mathit{Object}\,y, \overline{T\,x} \to U, C)$$

The claim now follows with rule OK-MDEF-IN-CLASS-IFJ. □

Lemma C.2.23. *If* $\mathsf{dict\text{-}methods}(I) = \overline{m : \mathit{mdef}}$ *then* $\vdash_{\mathsf{iFJ}} \overline{m : \mathit{mdef}}$ $\mathsf{implements}$ Dict^{I}.

Proof. Straightforward induction on the derivation of $\mathsf{dict\text{-}methods}(I) = \overline{m : \mathit{mdef}}$. □

Lemma C.2.24. *If* $\vdash^{\flat} \mathit{impl}$ $\mathsf{ok} \rightsquigarrow \mathit{cdef}$ *then* $\vdash_{\mathsf{iFJ}} \mathit{cdef}$ ok.

Proof. Assume

$$\mathit{impl} = \textbf{implementation}\ I\ [\,N\,]\,\{\,\overline{m : \mathit{mdef}}\,\}$$

Then

$$\begin{array}{rl} & \textbf{interface}\ I\ \textbf{extends}\ \overline{J}^{n}\ \{\,\overline{m : \mathit{msig}}\,\} \\ & (\forall i)\ \mathit{this} : N \vdash^{\flat} \mathit{mdef}_i\ \mathsf{implements}\ \mathit{msig}_i \rightsquigarrow \mathit{mdef}_i' \qquad (C.2.3) \\ \mathit{cdef} = & \mathsf{\textbf{class}}\ \mathit{Dict}^{I,N}\ \mathsf{\textbf{extends}}\ \mathit{Object}\ \mathsf{\textbf{implements}}\ \mathit{Dict}^{I}\ \{ \\ & \quad \overline{m : \mathit{mdef}'} \\ & \quad \mathsf{dict\text{-}methods}(J_1) \ldots \mathsf{dict\text{-}methods}(J_n) \\ & \} \end{array}$$

With Lemma C.2.20 and Lemma C.2.22 we get that

$$\vdash_{\mathsf{iFJ}} m : \mathit{mdef}\ \mathsf{ok\,in}\ \mathit{Dict}^{I,N} \tag{C.2.4}$$

for all methods $m : \mathit{mdef}$ of $\mathit{Dict}^{I,N}$. By Lemma C.2.23 we get

$$\mathsf{dict\text{-}methods}(J_i)\ \mathsf{implements}\ \mathit{Dict}^{J_i}$$

for all $i \in [n]$. With (C.2.3) we get by examining rule OK-MDEF-IN-CLASS$^{\flat}$ that mdef_i and mdef_i' have the same method signature. Looking at how rule OK-IDEF$^{\flat}$ generates the dictionary interface Dict^{I} thus yields

$$\vdash_{\mathsf{iFJ}} \overline{m : \mathit{mdef}'}\ \overline{\mathsf{dict\text{-}methods}.J_i}\ \mathsf{implements}\ \mathit{Dict}^{I}$$

by rule IMPL-IFACE-METHODS-IFJ. With Lemma C.2.14 then

$$\vdash_{\mathsf{iFJ}} \mathit{Dict}^{I,N}\ \mathsf{implements}\ \mathit{Dict}^{I} \tag{C.2.5}$$

Using (C.2.4) and (C.2.5) with rule OK-CDEF-IFJ then yields $\vdash_{\mathsf{iFJ}} \mathit{cdef}$ ok. □

Proof of Theorem 4.12. The claim that the translation from CoreGI$^{\flat}$ to iFJ preserves well-formedness of programs follows with Lemma C.2.13, Lemma C.2.18, Lemma C.2.24, Theorem 4.11, and Lemma C.2.9. □

C.3 Translation Preserves Dynamic Semantics

This section contains detailed proofs for Theorem 4.14 ($\equiv$ is an equivalence relation), Theorem 4.15 (substitution preserves $\equiv$), Theorem 4.16 (evaluation preserves $\equiv$), Theorem 4.18 ($\equiv$ is sound with respect to contextual equivalence), Theorem 4.19 (translation and single-step evaluation commute modulo wrappers), and Theorem 4.20 (translation and multi-step evaluation commute modulo wrappers). The lemmas in this section implicitly assume that the underlying CoreGI$^{\flat}$ and iFJ programs are well-formed. Moreover, each lemma mentioning both CoreGI$^{\flat}$ and iFJ constructs makes the implicit assumption that the underlying iFJ program is the translation of the underlying CoreGI$^{\flat}$ program. In this case, well-formedness of the CoreGI$^{\flat}$ program already guarantees well-formedness of the iFJ program by Theorem 4.12.

C.3.1 Proof of Theorem 4.14

Theorem 4.14 states that $\equiv$ is an equivalence relation.

Lemma C.3.1. *Assume* $\mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\,f}^{n}$ *and* $i \in [n]$*. Then there exists* C' *such that* $\vdash_{\mathsf{iFJ}} C \leq C'$*,* $\mathsf{defines\text{-}field}(C', f_i)$*, and* $\mathsf{fields}_{\mathsf{iFJ}}(C') = \overline{U\,f}^{m}$ *with* $m \geq i$*.*

Proof. Straightforward induction on the derivation of $\mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\,f}$. (Note that iFJ does not support field shadowing by well-formedness criterion WF-IFJ-3.) □

Lemma C.3.2. *If* $\mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \mathit{msig}$ *then there exists* T' *with* $\vdash_{\mathsf{iFJ}} T \leq T'$*,* $\mathsf{topmost}(T', m)$ *and* $\mathsf{mtype}_{\mathsf{iFJ}}(m, T') = \mathit{msig}$*.*

Proof. If $T = I$ for some I, then the claim follows from a straightforward induction on the derivation of $\mathsf{mtype}_{\mathsf{iFJ}}(m, T) = \mathit{msig}$.

Otherwise, suppose $T = N$ for some class type N. We then proceed by induction on the depth of N in the inheritance hierarchy. (The depth of a class type N in the inheritance hierarchy is 0 if $N = \mathit{Object}$, otherwise it is $1 + \delta$, where δ is the depth of N's superclass.)

Suppose that N has depth δ. If $\delta = 0$, we obtain a contradiction because *Object* does not have any methods. Thus, $\delta > 0$.

Case distinction on the rule used to derive $\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = \mathit{msig}$.

- *Case* MTYPE-CLASS-BASE-IFJ: Thus,

$$N = C$$
$$\textbf{class } C \textbf{ extends } M \textbf{ implements } \overline{J}\,\{\ldots\ \overline{m : \mathit{msig}\,\{\,e\,\}}\,\}$$
$$\mathit{msig} = \mathit{msig}_i$$
$$m = m_i$$

  - If $\mathsf{mtype}_{\mathsf{iFJ}}(m, M)$ is undefined and $\mathsf{mtype}_{\mathsf{iFJ}}(m, J_j)$ are undefined for all j, then rule TOPMOST-CLASS yields $\mathsf{topmost}(N, m)$, so the claim holds.
  - If there exists j such that $\mathsf{mtype}_{\mathsf{iFJ}}(m, J_j) = \mathit{msig}'$, then we have already shown at the beginning of this proof that $\mathsf{mtype}_{\mathsf{iFJ}}(m, T') = \mathit{msig}'$ for some T' such that $\vdash_{\mathsf{iFJ}} J_j \leq T'$ and $\mathsf{topmost}(T', m)$. By transitivity of subtyping $\vdash_{\mathsf{iFJ}} N \leq T'$. We then get $\mathit{msig} = \mathit{msig}'$ by Lemma C.1.6.
  - Otherwise, $\mathsf{mtype}_{\mathsf{iFJ}}(m, M) = \mathit{msig}'$. By rule OK-OVERRIDE-IFJ, we get $\mathit{msig} = \mathit{msig}'$. The claim now follows by the I.H. and transitivity of subtyping.

- *Case* MTYPE-CLASS-SUPER-IFJ: In this case, the claim follows by the I.H. and transitivity of subtyping.

End case distinction on the rule used to derive $\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = \mathit{msig}$. □

Lemma C.3.3 (Reflexivity of $\equiv$). *If* $\Gamma \vdash_{\mathsf{iFJ}} e : T'$ *and* $\vdash_{\mathsf{iFJ}} T' \leq T$ *then* $\Gamma \vdash_{\mathsf{iFJ}} e \equiv e : T$*.*

Proof. The proof is by induction on the structure of e.
Case distinction on the form of e.

- *Case* $e = x$: Obvious.

- *Case* $e = e'.f$: By inverting the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e'.f : T'$, we get

$$\begin{gathered}\Gamma \vdash_{\mathsf{iFJ}} e' : C \\ \mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\,f} \\ f = f_j \\ T' = U_j\end{gathered}$$

From Lemma C.3.1 we get that there exists C' with $\vdash_{\mathsf{iFJ}} C \leq C'$ such that $\mathsf{defines\text{-}field}(C', f)$, $\mathsf{fields}(C') = \overline{V\,g}$, $f = f_j = g_j$, and $T' = U_j = V_j$. Using the I.H. we also get $\Gamma \vdash_{\mathsf{iFJ}} e' \equiv e' : C'$. The claim now follows with rule EQUIV-FIELD.

- *Case* $e = e_0.m(\overline{e})$: By inverting the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e_0.m(\overline{e}) : T'$, we get

$$\begin{gathered}\Gamma \vdash_{\mathsf{iFJ}} e_0 : U \\ \mathsf{mtype}_{\mathsf{iFJ}}(m, U) = \overline{T\,x} \to T' \\ (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_i : T'_i \\ (\forall i)\ \vdash_{\mathsf{iFJ}} T'_i \leq T_i\end{gathered}$$

By Lemma C.3.2 we get the existence of U' such that

$$\begin{gathered}\vdash_{\mathsf{iFJ}} U \leq U' \\ \mathsf{topmost}(U', m) \\ \mathsf{mtype}_{\mathsf{iFJ}}(m, U') = \overline{T\,x} \to T'\end{gathered}$$

Applying the I.H. yields

$$\begin{gathered}\Gamma \vdash_{\mathsf{iFJ}} e_0 \equiv e_0 : U' \\ (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_i \equiv e_i : T_i\end{gathered}$$

The claim now follows by rule EQUIV-INVOKE.

- *Case* $e = \textbf{new}\,N(\overline{e})$: By inverting the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\,N(\overline{e}) : T'$, we get

$$\begin{gathered}T' = N \\ \mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{T\,f} \\ (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_i : T'_i \\ (\forall i)\ \vdash_{\mathsf{iFJ}} T'_i \leq T_i\end{gathered}$$

We then get from the I.H.

$$(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_i \equiv e_i : T_i$$

The claim now follows by EQUIV-NEW-CLASS.

- *Case* $e = \textbf{cast}(U, e')$: By inverting the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \textbf{cast}(U, e') : T'$, we get

$$\begin{gathered}U = T' \\ \Gamma \vdash_{\mathsf{iFJ}} e' : U'\end{gathered}$$

Obviously, $\vdash_{\mathsf{iFJ}} U' \leq \mathit{Object}$, so $\Gamma \vdash_{\mathsf{iFJ}} e' \equiv e' : \mathit{Object}$ follows from the I.H. The claim now follows with rule EQUIV-CAST.

- *Case* $e = \textbf{getdict}(I, e')$: By inverting the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \textbf{getdict}(I, e') : T'$, we get

$$T' = \mathit{Dict}^I$$
$$\Gamma \vdash_{\mathsf{iFJ}} e' : U$$

As in the preceding case, $\Gamma \vdash_{\mathsf{iFJ}} e' \equiv e' : \mathit{Object}$, so the claim follows by rule EQUIV-GETDICT.

- *Case* $e = \textbf{let}\, U\, x = e_1 \,\textbf{in}\, e_2$: By inverting the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \textbf{let}\, U\, x = e_1 \,\textbf{in}\, e_2 : T'$, we get

$$\Gamma \vdash_{\mathsf{iFJ}} e_1 : U'$$
$$\vdash_{\mathsf{iFJ}} U' \leq U$$
$$\Gamma, x : U \vdash_{\mathsf{iFJ}} e_2 : T'$$

Applying the I.H. yields

$$\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_1 : U$$
$$\Gamma, x : U \vdash_{\mathsf{iFJ}} e_2 \equiv e_2 : T$$

The claim now follows with rule EQUIV-LET.

End case distinction on the form of e. □

Lemma C.3.4 (Symmetry of $\equiv$). *If* $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$ *then* $\Gamma \vdash_{\mathsf{iFJ}} e_2 \equiv e_1 : T$.

Proof. Straightforward induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$. □

Lemma C.3.5. *If* $\mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{T\, f}$ *and* $\mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\, g}$ *then* $\overline{T\, f} = \overline{U\, g}$.

Proof. The claim holds because iFJ does not support field shadowing (see well-formedness criterion WF-IFJ-3). □

Lemma C.3.6. *If* $\Gamma \vdash_{\mathsf{iFJ}} e : T$ *and* $\Gamma \vdash_{\mathsf{iFJ}} e : U$ *then* $T = U$.

Proof. We proceed by induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e : T$. It is obvious that the derivations of $\Gamma \vdash_{\mathsf{iFJ}} e : T$ and $\Gamma \vdash_{\mathsf{iFJ}} e : U$ end with the same rule. If this rule is EXP-VAR-IFJ, EXP-NEW-IFJ, EXP-CAST-IFJ, or EXP-GETDICT-IFJ, then the claim holds trivially. If the last rule of the two derivations is EXP-FIELD-IFJ, then the claim follows with the I.H. and Lemma C.3.5. If the last rule is EXP-INVOKE-IFJ, then the claim follows with the I.H. and Lemma C.1.6. Finally, if the last rule is EXP-LET-IFJ, then the claim follows from the I.H. □

Lemma C.3.7. *If* $\vdash_{\mathsf{iFJ}} C \leq D_1$ *and* $\vdash_{\mathsf{iFJ}} C \leq D_2$ *then either* $\vdash_{\mathsf{iFJ}} D_1 \leq D_2$ *or* $\vdash_{\mathsf{iFJ}} D_2 \leq D_1$.

Proof. By Lemma C.1.3, we may assume $\vdash_{\mathsf{iFJ\text{-}a}} C \leq D_1$ and $\vdash_{\mathsf{iFJ\text{-}a}} C \leq D_2$. By induction on these two derivations and with Lemma C.1.4, we get that either $\vdash_{\mathsf{iFJ\text{-}a}} D_1 \leq D_2$ or $\vdash_{\mathsf{iFJ\text{-}a}} D_2 \leq D_1$. An application of Lemma C.1.3 then finishes the proof. □

Lemma C.3.8. *Suppose that the iFJ program under consideration is in the image of the translation from CoreGI*$^\flat$ *to iFJ. If* $\mathsf{topmost}(T, m)$ *and* $\mathsf{topmost}(U, m)$ *and there exists a type* V *such that* $\vdash_{\mathsf{iFJ}} V \leq T$ *and* $\vdash_{\mathsf{iFJ}} V \leq U$, *then* $T = U$.

Proof. *Case distinction* on the forms of T and U.

- *Case* $T = I$ and $U = J$: From $\mathsf{topmost}(I, m)$ and $\mathsf{topmost}(J, m)$ we know that both interfaces I and J define a method of name m. The only places where the translation from $\mathsf{CoreGI}^\flat$ to iFJ generates interfaces is rule OK-IDEF$^\flat$. Also, we know that distinct interfaces in $\mathsf{CoreGI}^\flat$ define methods with disjoint names (Convention 4.2).

  Thus, unless $I = J$, w.l.o.g. $J = \mathit{Dict}^I$. Because the namespaces for regular interfaces such as I and dictionary interfaces such as Dict^I are disjoint (Convention 4.4), it is straightforward to verify that no type V exists with both $\vdash_{\mathsf{iFJ}} V \leq I$ and $\vdash_{\mathsf{iFJ}} V \leq \mathit{Dict}^I$. Hence $I = J$ as required.

- *Case* $T = N$ and $U = M$: Class *Object* does not define any methods, so with $\mathsf{topmost}(N, m)$ and $\mathsf{topmost}(M, m)$ we know that $N = C$ and $M = D$. With Lemma C.1.4 we get that $V = C'$ for some C', so with Lemma C.3.7 either $\vdash_{\mathsf{iFJ}} C \leq D$ or $\vdash_{\mathsf{iFJ}} D \leq C$. In both cases, it is easy to see that $\mathsf{topmost}(C, m)$ and $\mathsf{topmost}(D, m)$ imply $C = D$ as required.

- *Case* $T = N$ and $U = I$: With both $\mathsf{topmost}(N, m)$ and $\mathsf{topmost}(I, m)$, it is straightforward to verify that $N = C$ for some C and that $\vdash_{\mathsf{iFJ}} C \leq I$ does not hold. Moreover, with $\vdash_{\mathsf{iFJ}} V \leq C$ and Lemma C.1.4, we know that $V = D$ for some D. With $\vdash_{\mathsf{iFJ}} D \leq I$ we also know $C \neq D$.

  In $\mathsf{CoreGI}^\flat$, the namespaces of class and interface methods is disjoint and names of interface methods are unique (Convention 4.1 and Convention 4.2). However, $\mathsf{topmost}(C, m)$ and $\mathsf{topmost}(I, m)$ imply that both C and I define a method of name m, so the only way how the translation can insert m into class C is via rule OK-IDEF$^\flat$ or via rule OK-IMPL$^\flat$.

  In the first case, we have $C = \mathit{Wrap}^I$, and in the second case, we have $C = \mathit{Dict}^{I,N'}$ for some N'. However, the translation never uses classes such as Wrap^I or $\mathit{Dict}^{I,N'}$ as superclasses of other classes. (Note that the namespaces for regular classes, for wrapper classes, and for dictionary classes are disjoint by Convention 4.4.) Thus, no class $D \neq C$ can exist with $\vdash_{\mathsf{iFJ}} D \leq C$. But this is a contradiction.

- *Case* $T = I$ and $U = N$: Analogously to the preceding case.

End case distinction on the forms of T and U. □

Lemma C.3.9. *If* $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$ *then* $\Gamma \vdash_{\mathsf{iFJ}} e_1 : U_1$ *and* $\Gamma \vdash_{\mathsf{iFJ}} e_2 : U_2$ *such that* $\vdash_{\mathsf{iFJ}} U_1 \leq T$ *and* $\vdash_{\mathsf{iFJ}} U_2 \leq T$.

Proof. By Lemma C.3.4, it suffices to prove the claim for e_1. We proceed by induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$. If the last rule of this derivation is EQUIV-VAR, then the claim holds obviously. If the last rule is EQUIV-FIELD-WRAPPED, then we have

$$e_1 = \textbf{new}\ \mathit{Wrap}^I(e_1').\mathit{wrapped}$$
$$e_2 = \textbf{new}\ \mathit{Wrap}^J(e_2').\mathit{wrapped}$$
$$T = \mathit{Object}$$
$$\Gamma \vdash_{\mathsf{iFJ}} e_1' \equiv e_2' : \mathit{Object}$$

Applying the I.H. yields

$$\Gamma \vdash_{\mathsf{iFJ}} e_1' : \mathit{Object}$$

With well-formedness criterion WF-IFJ-6 and rule EXP-NEW-IFJ then

$$\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\ \mathit{Wrap}^I(e_1') : \mathit{Wrap}^I$$

Rule EXP-FIELD-IFJ and well-formedness criterion WF-IFJ-6 then yield $\Gamma \vdash_{\mathsf{iFJ}} e_1 : T$ as required.

In all other cases, the claims follows from the I.H., using Lemma C.1.5 if the last rule is EQUIV-FIELD, Lemma C.1.6 if the last rule is EQUIV-INVOKE, and well-formedness criterion WF-IFJ-6 if the last rule is either rule EQUIV-NEW-WRAP or rule EQUIV-NEW-OBJECT-LEFT. □

Lemma C.3.10. *If* $\mathsf{defines\text{-}field}(C, f)$ *and* $\mathsf{defines\text{-}field}(D, f)$ *and either* $\vdash_{\mathsf{iFJ}} C \leq D$ *or* $\vdash_{\mathsf{iFJ}} D \leq C$*, then* $C = D$*.*

Proof. W.l.o.g., assume $\vdash_{\mathsf{iFJ}} C \leq D$. Using Lemma C.1.3, we then have $\vdash_{\mathsf{iFJ\text{-}a}} C \leq D$; that is, D is a superclass of C. Well-formedness criterion WF-IFJ-3 then implies $C = D$. □

The notation $\mathcal{D} :: \mathcal{J}$ names the derivation of judgment $\mathcal{J}$ as $\mathcal{D}$.

Lemma C.3.11 (Transitivity of $\equiv$). *Suppose that the iFJ program under consideration is in the image of the translation from CoreGI$^\flat$ to iFJ. If now* $\mathcal{D}_1 :: \Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$ *and* $\mathcal{D}_2 :: \Gamma \vdash_{\mathsf{iFJ}} e_2 \equiv e_3 : T$ *then* $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_3 : T$*.*

Proof. We proceed by induction on the combined height of $\mathcal{D}_1$ and $\mathcal{D}_2$.
Case distinction on the form of e_2.

- *Case* $e_2 = x$: Then $\mathcal{D}_1$ and $\mathcal{D}_2$ both end with EQUIV-VAR and the claim holds trivially.
- *Case* $e_2 = e_2'.f$:

  Case distinction on the last rules of $\mathcal{D}_1$ and $\mathcal{D}_2$.

  – *Case* EQUIV-FIELD / EQUIV-FIELD: Then

$$\begin{gathered} e_1 = e_1'.f \\ \Gamma \vdash_{\mathsf{iFJ}} e_1' \equiv e_2' : C \\ \mathsf{defines\text{-}field}(C, f) \\ \mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\, f} \\ f = f_i \\ \vdash_{\mathsf{iFJ}} U_i \leq T \end{gathered}$$

and also

$$\begin{gathered} e_3 = e_3'.f \\ \Gamma \vdash_{\mathsf{iFJ}} e_2' \equiv e_3' : C' \\ \mathsf{defines\text{-}field}(C', f) \\ \mathsf{fields}_{\mathsf{iFJ}}(C') = \overline{U'\, f'} \\ f = f_i' \\ \vdash_{\mathsf{iFJ}} U_i' \leq T \end{gathered}$$

By Lemma C.3.9 we get

$$\begin{gathered} \Gamma \vdash_{\mathsf{iFJ}} e_2' : C_2 \\ \vdash_{\mathsf{iFJ}} C_2 \leq C \\ \Gamma \vdash_{\mathsf{iFJ}} e_2' : C_2' \\ \vdash_{\mathsf{iFJ}} C_2' \leq C' \end{gathered}$$

By Lemma C.3.6 then $C_2 = C_2'$, so with Lemma C.3.7 either $\vdash_{\mathsf{iFJ}} C \leq C'$ or $\vdash_{\mathsf{iFJ}} C' \leq C$. With Lemma C.3.10 we then get $C = C'$. Applying the I.H. then yields $\Gamma \vdash_{\mathsf{iFJ}} e_1' \equiv e_3' : C$, so the claim follows with rule EQUIV-FIELD.

- *Case* EQUIV-FIELD-WRAPPED / EQUIV-FIELD-WRAPPED: Then

$$\begin{gathered} e_1 = \textsf{\textbf{new}}\ \mathit{Wrap}^{I_1}(e_1').\mathit{wrapped} \\ e_2 = \textsf{\textbf{new}}\ \mathit{Wrap}^{I_2}(e_2').\mathit{wrapped} \\ e_3 = \textsf{\textbf{new}}\ \mathit{Wrap}^{I_3}(e_3').\mathit{wrapped} \\ \Gamma \vdash_{\textsf{iFJ}} e_1' \equiv e_2' : \mathit{Object} \\ \Gamma \vdash_{\textsf{iFJ}} e_2' \equiv e_3' : \mathit{Object} \end{gathered}$$

  Applying the I.H. yields $\Gamma \vdash_{\textsf{iFJ}} e_1' \equiv e_3' : \mathit{Object}$, so the claim follows with rule EQUIV-FIELD-WRAPPED.

- *Case* EQUIV-FIELD-WRAPPED / EQUIV-FIELD: Then

$$\begin{gathered} e_1 = \textsf{\textbf{new}}\ \mathit{Wrap}^{I_1}(e_1').\mathit{wrapped} \\ e_2 = \textsf{\textbf{new}}\ \mathit{Wrap}^{I_2}(e_2').\mathit{wrapped} \\ e_3 = e_3'.\mathit{wrapped} \\ \Gamma \vdash_{\textsf{iFJ}} e_1' \equiv e_2' : \mathit{Object} \\ \Gamma \vdash_{\textsf{iFJ}} \textsf{\textbf{new}}\ \mathit{Wrap}^{I_2}(e_2') \equiv e_3' : C \qquad \text{(C.3.1)} \end{gathered}$$

  Obviously, the derivation of (C.3.1) ends with rule EQUIV-NEW-CLASS. Inverting the rule yields, together with WF-IFJ-6,

$$\begin{gathered} e_3' = \textsf{\textbf{new}}\ \mathit{Wrap}^{I_2}(e_3'') \\ \Gamma \vdash_{\textsf{iFJ}} e_2' \equiv e_3'' : \mathit{Object} \end{gathered}$$

  Applying the I.H. yields $\Gamma \vdash_{\textsf{iFJ}} e_1' \equiv e_3'' : \mathit{Object}$, so the claim follows by rule EQUIV-FIELD-WRAPPED.

- *Case* EQUIV-FIELD / EQUIV-FIELD-WRAPPED: Analogously to the preceding case.

End case distinction on the last rules of $\mathcal{D}_1$ and $\mathcal{D}_2$.

- *Case* $e_2 = e_{20}.m(\overline{e_2})$: Then $\mathcal{D}_1$ and $\mathcal{D}_2$ both end with EQUIV-INVOKE, so we have

$$\begin{gathered} e_1 = e_{10}.m(\overline{e_1}) \\ \Gamma \vdash_{\textsf{iFJ}} e_{10} \equiv e_{20} : V \\ \textsf{topmost}(V, m) \\ (\forall i)\ \Gamma \vdash_{\textsf{iFJ}} e_{1i} \equiv e_{2i} : U_i \\ \textsf{mtype}_{\textsf{iFJ}}(m, V) = \overline{U\,x} \to U \\ \vdash_{\textsf{iFJ}} U \leq T \end{gathered}$$

and also

$$\begin{gathered} e_3 = e_{30}.m(\overline{e_3}) \\ \Gamma \vdash_{\textsf{iFJ}} e_{20} \equiv e_{30} : V' \\ \textsf{topmost}(V', m) \\ (\forall i)\ \Gamma \vdash_{\textsf{iFJ}} e_{2i} \equiv e_{3i} : U_i' \\ \textsf{mtype}_{\textsf{iFJ}}(m, V') = \overline{U'\,x'} \to U' \\ \vdash_{\textsf{iFJ}} U' \leq T \end{gathered}$$

By Lemma C.3.6 and Lemma C.3.9 we have

$$\begin{gathered}\Gamma \vdash_{\mathsf{iFJ}} e_{20} : W \\ \vdash_{\mathsf{iFJ}} W \leq V \\ \vdash_{\mathsf{iFJ}} W \leq V'\end{gathered}$$

Because the program under consideration is in the image of the translation from $\mathsf{CoreGI}^\flat$ to iFJ, we get with Lemma C.3.8 that $V = V'$. Thus, we also have $\overline{U} = \overline{U'}$ by Lemma C.1.6. Moreover, the I.H. yields

$$\begin{gathered}\Gamma \vdash_{\mathsf{iFJ}} e_{10} \equiv e_{30} : V \\ (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} e_{1i} \equiv e_{3i} : U_i\end{gathered}$$

Thus, the claim follows from rule EQUIV-INVOKE.

- *Case* $e_2 = \textbf{new}\, N(\overline{e_2})$: The following table lists all possible combinations for the last rules of $\mathcal{D}_1$ and $\mathcal{D}_2$ (we omit the prefix "EQUIV-NEW-" from the rule names):

	CLASS	WRAP	OBJECT-LEFT	OBJECT-RIGHT
CLASS	I.H.	⋆	⋆	I.H.
WRAP	⋆	I.H.	↯	↯
OBJECT-LEFT	I.H.	↯	I.H.	I.H.
OBJECT-RIGHT	I.H.	↯	I.H.	I.H.

  For the combinations marked with "I.H.", the claim follows directly from the induction hypothesis. Combinations marked with "⋆" require the I.H. and well-formedness criterion WF-IFJ-6. Combinations marked with "↯" can never occur because they put conflicting constraints on the form of T.

- *Case* $e_2 = \textbf{cast}(T_2, e_2')$: Hence, both $\mathcal{D}_1$ and $\mathcal{D}_2$ end with rule EQUIV-CAST. The claim then follows directly from the I.H.

- *Case* $e_2 = \textbf{getdict}(I_2, e_2')$: Hence, both $\mathcal{D}_1$ and $\mathcal{D}_2$ end with rule EQUIV-GETDICT. The claim then follows directly from the I.H.

- *Case* $e_2 = \textbf{let}\, U\, x = e_{21}\, \textbf{in}\, e_{22}$: In this case, both $\mathcal{D}_1$ and $\mathcal{D}_2$ end with rule EQUIV-LET. Thus, the claim follows directly from the I.H.

End case distinction on the form of e_2. □

Proof of Theorem 4.14. Follows from Lemmas C.3.3, C.3.4, and C.3.11. □

C.3.2 Proof of Theorem 4.15

Theorem 4.15 states that substitution preserves equivalence modulo wrappers.

Lemma C.3.12. *If* $\vdash_{\mathsf{iFJ}} \mathit{Object} \leq T$ *then* $T = \mathit{Object}$.

Proof. With $\vdash_{\mathsf{iFJ}} \mathit{Object} \leq T$ we have $\vdash_{\mathsf{iFJ\text{-}a}} \mathit{Object} \leq T$ by Lemma C.1.3. The claim now follows because the derivation of $\vdash_{\mathsf{iFJ\text{-}a}} \mathit{Object} \leq T$ must end with rule SUB-ALG-OBJECT-IFJ. □

Lemma C.3.13. *If* $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$ *and* $\vdash_{\mathsf{iFJ}} T \leq U$ *then* $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : U$.

Proof. We proceed by induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$. If the last rule of this derivation is EQUIV-LET, then the claim follows from the I.H. If the last rule is EQUIV-FIELD-WRAPPED, EQUIV-NEW-OBJECT-LEFT, or EQUIV-NEW-OBJECT-RIGHT, then $U = \mathit{Object}$ by Lemma C.3.12, so the claim obviously holds. In any other case, the premise of the last rule in the derivation allows us to lift T to U using transitivity of subtyping. □

Proof of Theorem 4.15. We proceed by induction on the derivation of $\Gamma, x : U \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$. If the last rule in the derivation is not EQUIV-VAR, the claim follows from the I.H. If the last rule in the derivation is EQUIV-VAR then $e_1 = e_2 = y$ and $\vdash_{\mathsf{iFJ}} (\Gamma, x : U)(y) \leq T$. If $y \neq x$ then the claim holds obviously. Otherwise, we have $[d_1/x]e_1 = d_1$, $[d_2/x]e_2 = d_2$, and $\vdash_{\mathsf{iFJ}} U \leq T$. The claim then follows from the assumption $\Gamma \vdash_{\mathsf{iFJ}} d_1 \equiv d_2 : U$ and Lemma C.3.13. □

C.3.3 Proof of Theorem 4.16

Theorem 4.16 states that evaluation in iFJ preserves equivalence modulo wrappers.

Lemma C.3.14. *If* $\vdash_{\mathsf{iFJ}} J_1 \leq I$ *and* $\vdash_{\mathsf{iFJ}} J_2 \leq I$ *and* $\mathsf{topmost}(I, m)$ *then* $\mathsf{getmdef}_{\mathsf{iFJ}}(m, \mathit{Wrap}^{J_1}) = \mathsf{getmdef}_{\mathsf{iFJ}}(m, \mathit{Wrap}^{J_2})$.

Proof. By Convention 4.4, wrapper classes are not part of standalone iFJ programs, so the underlying iFJ program must be in the image of the translation from CoreGI$^\flat$. Moreover, wrapper classes are only generated for interfaces originally contained in the CoreGI$^\flat$ program. Thus, the iFJ interfaces J_1 and J_2 are translation of CoreGI$^\flat$ interfaces. Because the translation from CoreGI$^\flat$ to iFJ leaves the superinterface hierarchy of such interfaces unchanged, the iFJ interface I must also be the translation of a CoreGI$^\flat$ interface.

With $\mathsf{topmost}(I, m)$ we know that interface I contains a definition of m. Because J_1, J_2, and I are translations of CoreGI$^\flat$ interfaces, we get by Convention 4.2 that I is the only superinterface of J_1 and J_2 that contains a definition of m. By rule WRAPPER-METHODS$^\flat$ we then have that $\mathsf{wrapper\text{-}methods}(J_1)$ and $\mathsf{wrapper\text{-}methods}(J_2)$ each contain exactly one definition of m and that this definition is identical. Examining rule OK-IDEF-IFJ and the definition of $\mathsf{getmdef}_{\mathsf{iFJ}}$ yields the desired result. □

Lemma C.3.15. *If* $\mathsf{topmost}(I, m)$ *then* $\mathsf{topmost}(\mathit{Dict}^I, m)$.

Proof. By $\mathsf{topmost}(I, m)$ we know that I defines method m. Examining rule OK-IDEF$^\flat$ yields that interface Dict^I also contains a definition of m. Thus, $\mathsf{topmost}(\mathit{Dict}^I, m)$. □

Lemma C.3.16. *If* $\Gamma \vdash_{\mathsf{iFJ}} v \equiv w : \mathit{Object}$ *and* $\mathsf{unwrap}(v) = \textbf{new}\, N(\overline{v})$ *then* $\mathsf{unwrap}(w) = \textbf{new}\, N(\overline{w})$ *and* $\Gamma \vdash \textbf{new}\, N(\overline{v}) \equiv \textbf{new}\, N(\overline{w}) : N$.

Proof. We proceed by induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} v \equiv w : \mathit{Object}$.
Case distinction on the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} v \equiv w : \mathit{Object}$.

- *Case* rule EQUIV-NEW-CLASS: Then

$$v = \textbf{new}\, M(\overline{v'})$$
$$w = \textbf{new}\, M(\overline{w'})$$
$$\mathsf{fields}_{\mathsf{iFJ}}(M) = \overline{U\, f}$$
$$(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} v'_i \equiv w'_i : U_i$$

If M is not a wrapper class, then the claim obviously holds by EQUIV-NEW-CLASS. Otherwise, $M = \mathit{Wrap}^I$ and, together with well-formedness criterion WF-IFJ-6 and the definition of unwrap,

$$\begin{aligned} \overline{v'} &= v'_1 \\ \overline{w'} &= w'_1 \\ \overline{U\,f} &= \mathit{Object}\,f_1 \\ \mathsf{unwrap}(v) &= \mathsf{unwrap}(v'_1) \\ \mathsf{unwrap}(w) &= \mathsf{unwrap}(w'_1) \end{aligned}$$

Thus, $\Gamma \vdash v'_1 \equiv w'_1 : \mathit{Object}$, so applying the I.H. yields the desired result.

- *Case* rule EQUIV-NEW-WRAP: Impossible because $\mathit{Object} \neq I$ for any interface I.
- *Case* rule EQUIV-NEW-OBJECT-LEFT: Follows from the I.H. and the definition of unwrap.
- *Case* rule EQUIV-NEW-OBJECT-RIGHT: Follows from the I.H. and the definition of unwrap.
- *Case* any other rule: Impossible.

End case distinction on the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} v \equiv w : \mathit{Object}$. □

Lemma C.3.17. *If $\Gamma \vdash_{\mathsf{iFJ}} e \equiv d : T$ and e is a value, then d is also a value.*

Proof. Straightforward induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e \equiv d : T$. □

Lemma C.3.18. *For all iFJ evaluation contexts $\mathcal{E}_1$ and $\mathcal{E}_2$, there exists an iFJ evaluation context $\mathcal{E}_3$ such that for all expressions e it holds that $\mathcal{E}_1[\mathcal{E}_2[e]] = \mathcal{E}_3[e]$.*

Proof. Straightforward induction on the structure of $\mathcal{E}_1$. □

Lemma C.3.19. *Assume $e \longrightarrow_{\mathsf{iFJ}} e'$. Then $\mathcal{E}[e] \longrightarrow_{\mathsf{iFJ}} \mathcal{E}[e']$ for any evaluation context $\mathcal{E}$.*

Proof. We get from $e \longrightarrow_{\mathsf{iFJ}} e'$ by inverting rule DYN-CONTEXT-IFJ that there exist $\mathcal{E}', d, d'$ such that

$$\begin{aligned} e &= \mathcal{E}'[d] \\ e' &= \mathcal{E}'[d'] \\ d &\longmapsto_{\mathsf{iFJ}} d' \end{aligned}$$

By Lemma C.3.18 we get the existence of $\mathcal{E}''$ such that

$$\begin{aligned} \underbrace{\mathcal{E}[\mathcal{E}'[d]]}_{=\mathcal{E}[e]} &= \mathcal{E}''[d] \\ \underbrace{\mathcal{E}[\mathcal{E}'[d']]}_{=\mathcal{E}[e']} &= \mathcal{E}''[d'] \end{aligned}$$

Hence, rule DYN-CONTEXT-IFJ yields $\mathcal{E}[e] \longrightarrow_{\mathsf{iFJ}} \mathcal{E}[e']$. □

Lemma C.3.20 (Weakening lemma for type-directed equivalence modulo wrappers). *If $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$.*

Proof. Straightforward induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$. □

Lemma C.3.21 (Top-level evaluation preserves $\equiv$). *If* $\Gamma \vdash_{\mathsf{iFJ}} e \equiv d : T$ *and* $e \longmapsto_{\mathsf{iFJ}} e'$ *then* $d \longrightarrow_{\mathsf{iFJ}} d'$ *such that* $\Gamma \vdash_{\mathsf{iFJ}} e' \equiv d' : T$.

Proof. Induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e \equiv d : T$.
Case distinction on the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e \equiv d : T$.

- *Case* rule EQUIV-VAR: Impossible because there is no reduction rule for variables.
- *Case* rule EQUIV-FIELD: We then have

$$\begin{gathered} e = e''.f_j \\ d = d''.f_j \\ \Gamma \vdash_{\mathsf{iFJ}} e'' \equiv d'' : C \\ \mathsf{defines\text{-}field}(C, f_j) \\ \mathsf{fields}_{\mathsf{iFJ}}(C) = \overline{U\, f} \\ \vdash_{\mathsf{iFJ}} U_j \leq T \end{gathered}$$

Moreover, the reduction $e \longmapsto_{\mathsf{iFJ}} e'$ must have been performed through rule DYN-FIELD-IFJ. Thus

$$\begin{gathered} e'' = \textbf{new}\, N(\overline{v}) \\ \mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{V\, g} \\ f_j = g_k \\ e' = v_k \end{gathered}$$

By Lemma C.3.9 and inverting rule EXP-NEW-IFJ we know

$$\begin{gathered} \Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{v}) : N \\ \vdash_{\mathsf{iFJ}} N \leq C \end{gathered}$$

By Lemma C.1.5 we have that

$$\begin{gathered} \overline{V\, g} = \overline{U\, f}, \overline{V'\, g'} \\ k = j \end{gathered}$$

A case analysis on the last rule of the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{v}) \equiv d'' : C$ reveals that this derivation must end with rule EQUIV-NEW-CLASS. Thus, together with Lemma C.3.17

$$\begin{gathered} d'' = \textbf{new}\, N(\overline{w}) \\ (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} v_i \equiv w_i : V_i \end{gathered}$$

We then have by rules DYN-FIELD-IFJ and DYN-CONTEXT-IFJ

$$\underbrace{\textbf{new}\, N(\overline{w}).f_j}_{=d} \longrightarrow_{\mathsf{iFJ}} \underbrace{w_k}_{=:d'}$$

and because $j = k$ we get $\Gamma \vdash_{\mathsf{iFJ}} v_k \equiv w_k : U_j$. With $\vdash_{\mathsf{iFJ}} U_j \leq T$ and Lemma C.3.13 we finally get

$$\Gamma \vdash_{\mathsf{iFJ}} e' \equiv d' : T$$

- *Case* rule EQUIV-FIELD-WRAPPED: We have

$$
\begin{aligned}
e &= \textbf{new}\ \mathit{Wrap}^I(e_0).\mathit{wrapped}\\
d &= \textbf{new}\ \mathit{Wrap}^J(d_0).\mathit{wrapped}\\
\Gamma \vdash_{\mathsf{iFJ}} e_0 &\equiv d_0 : \mathit{Object} \qquad\qquad (C.3.2)\\
T &= \mathit{Object}
\end{aligned}
$$

  Obviously, the reduction $e \longmapsto_{\mathsf{iFJ}} e'$ has been performed through DYN-FIELD-IFJ. Inverting the rule yields, together with well-formedness criterion WF-IFJ-6,

$$e' = e_0$$

  Also by rule DYN-FIELD-IFJ and well-formedness criterion WF-IFJ-6,

$$d \longmapsto_{\mathsf{iFJ}} d_0$$

  The claim now follows with (C.3.2) and rule DYN-CONTEXT-IFJ.

- *Case* rule EQUIV-INVOKE: By inverting the rule and because the reduction $e \longmapsto_{\mathsf{iFJ}} e'$ must have been performed through rule DYN-INVOKE-IFJ, we get

$$
\begin{aligned}
e &= v_0.m(\overline{v})\\
d &= d_0.m(\overline{d})\\
\Gamma \vdash_{\mathsf{iFJ}} v_0 &\equiv d_0 : V && (C.3.3)\\
&\mathsf{topmost}(V, m) && (C.3.4)\\
(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} v_i &\equiv d_i : U_i && (C.3.5)\\
\mathsf{mtype}_{\mathsf{iFJ}}(m, V) &= \overline{U\,x} \to U && (C.3.6)\\
\vdash_{\mathsf{iFJ}} U &\leq T && (C.3.7)\\
v_0 &= \textbf{new}\ N(\overline{w}) && (C.3.8)\\
\mathsf{getmdef}_{\mathsf{iFJ}}(m, N) &= \overline{U'\,x'} \to U'\ \{e''\} && (C.3.9)\\
e' &= [v_0/\mathit{this}, \overline{v/x'}]e''
\end{aligned}
$$

  By Lemma C.3.9 and inverting rule EXP-NEW-IFJ we know

$$
\begin{aligned}
\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\ N(\overline{w}) &: N\\
\vdash_{\mathsf{iFJ}} N &\leq V
\end{aligned}
$$

  Thus, by Lemma C.1.7

$$\overline{U\,x} \to U = \overline{U'\,x'} \to U' \qquad (C.3.10)$$

  Case distinction on the form of V.

  - *Case* $V = \mathit{Object}$: Contradiction to $\mathsf{topmost}(V, m)$.
  - *Case* $V = C$: Then the derivation of (C.3.3) ends with rule EQUIV-NEW-CLASS. Thus, (C.3.8) together with Lemma C.3.17 yields

$$
\begin{aligned}
\Gamma \vdash_{\mathsf{iFJ}} v_0 &\equiv d_0 : N \qquad\qquad (C.3.11)\\
d_0 &= \textbf{new}\ N(\overline{w'})\\
\vdash_{\mathsf{iFJ}} N &\leq C
\end{aligned}
$$

From (C.3.5) and Lemma C.3.17

$$\overline{d} = \overline{v'}$$

Thus, by rules DYN-INVOKE-IFJ and DYN-CONTEXT-IFJ

$$d \longrightarrow_{\mathsf{iFJ}} \underbrace{[d_0/\mathit{this}, \overline{d/x}]e''}_{=:d'}$$

We have by (C.3.9), (C.3.10), and Lemma C.1.8 that

$$\mathit{this} : N', \overline{x : U} \vdash_{\mathsf{iFJ}} e'' : U'' \tag{C.3.12}$$

$$\vdash_{\mathsf{iFJ}} U'' \leq U \tag{C.3.13}$$

$$\vdash_{\mathsf{iFJ}} N \leq N'$$

By (C.3.11) and Lemma C.3.13 we have

$$\Gamma \vdash_{\mathsf{iFJ}} v_0 \equiv d_0 : N'$$

We get from (C.3.12), (C.3.13), (C.3.7), transitivity of subtyping, and Lemma C.3.3 that

$$\mathit{this} : N', \overline{x : U} \vdash_{\mathsf{iFJ}} e'' \equiv e'' : T$$

Hence, with (C.3.5) and possibly repeated applications of Theorem 4.15

$$\emptyset \vdash_{\mathsf{iFJ}} \underbrace{[v_0/\mathit{this}, \overline{v/x}]e''}_{=e'} \equiv \underbrace{[d_0/\mathit{this}, \overline{d/x}]e''}_{=d'} : T$$

An application of Lemma C.3.20 then finishes this case.

– *Case* $V = I$: Then the derivation of (C.3.3) must end with rule EQUIV-NEW-WRAP, so we have

$$v_0 = \textbf{new } \overbrace{\mathit{Wrap}^J}^{=N}(w_1)$$

$$d_0 = \textbf{new } \mathit{Wrap}^{J'}(w_1')$$

$$\vdash_{\mathsf{iFJ}} J' \leq I$$

$$\vdash_{\mathsf{iFJ}} J \leq I$$

$$\Gamma \vdash_{\mathsf{iFJ}} w_1 \equiv w_1' : \mathit{Object} \tag{C.3.14}$$

With (C.3.5) and Lemma C.3.17

$$\overline{d} = \overline{v'}$$

By Lemma C.3.14, (C.3.4), (C.3.9), and (C.3.10) we then get

$$\mathsf{getmdef}_{\mathsf{iFJ}}(m, \mathit{Wrap}^{J'}) = \overline{U\, x} \rightarrow U\, \{e''\}$$

Hence, by rules DYN-INVOKE-IFJ and DYN-CONTEXT-IFJ

$$\underbrace{\textbf{new } \mathit{Wrap}^{J'}(w_1').m(\overline{d})}_{=d} \longrightarrow_{\mathsf{iFJ}} \underbrace{[d_0/\mathit{this}, \overline{d/x}]e''}_{=:d'}$$

Because wrapper classes are generated only by the translation from CoreGI$^\flat$ to iFJ, we know by inverting rules OK-IDEF$^\flat$ and WRAPPER-METHODS$^\flat$ that

$$e'' = \textbf{getdict}(I, \mathit{this}.\mathit{wrapped}).m(\mathit{this}.\mathit{wrapped}, \overline{x})$$

By rule EQUIV-FIELD-WRAPPED and (C.3.14)

$$\Gamma \vdash_{\mathsf{iFJ}} v_0.\mathit{wrapped} \equiv d_0.\mathit{wrapped} : \mathit{Object}$$

Hence, by rule EQUIV-GETDICT

$$\begin{aligned}\Gamma \vdash_{\mathsf{iFJ}} & [v_0/\mathit{this}, \overline{v/x}]\textbf{getdict}(I, \mathit{this}.\mathit{wrapped}) \equiv \\ & [d_0/\mathit{this}, \overline{d/x}]\textbf{getdict}(I, \mathit{this}.\mathit{wrapped}) : \mathit{Dict}^I\end{aligned}$$

By Lemma C.3.15 applied to (C.3.4), and $V = I$ we have

$$\mathsf{topmost}(\mathit{Dict}^I, m)$$

With Lemma C.2.19, (C.3.6), and $V = I$ we get

$$\mathsf{mtype}_{\mathsf{iFJ}}(m, \mathit{Dict}^I) = \mathit{Object}\, y, \overline{U\, x} \to U$$

Thus, with (C.3.5) and (C.3.7) we get by rule EQUIV-INVOKE

$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{[v_0/\mathit{this}, \overline{v/x}]e''}_{=e'} \equiv \underbrace{[d_0/\mathit{this}, \overline{d/x}]e''}_{=d'} : T$$

as required.

End case distinction on the form of V.

- *Case* rule EQUIV-NEW-CLASS: Impossible because e would then have the form **new** $N(\overline{e})$, but such expressions are not reducible via $\longmapsto_{\mathsf{iFJ}}$.
- *Case* rule EQUIV-NEW-WRAP: Impossible, for the same reason as in the preceding case.
- *Case* rule EQUIV-NEW-OBJECT-LEFT: Impossible, for the same reason as in the case for rule EQUIV-NEW-CLASS.
- *Case* rule EQUIV-NEW-OBJECT-RIGHT: We then have from the premise of this rule

$$\begin{gathered} d = \textbf{new}\ \mathit{Wrap}^I(d'') \\ T = \mathit{Object} \\ \Gamma \vdash_{\mathsf{iFJ}} e \equiv d'' : \mathit{Object} \end{gathered}$$

Applying the I.H. yields

$$\begin{gathered} d'' \longrightarrow_{\mathsf{iFJ}} d''' \\ \Gamma \vdash_{\mathsf{iFJ}} e' \equiv d''' : \mathit{Object} \end{gathered}$$

By Lemma C.3.19

$$d \longrightarrow_{\mathsf{iFJ}} \underbrace{\textbf{new}\ \mathit{Wrap}^I(d''')}_{=:d'}$$

and by rule EQUIV-NEW-OBJECT-RIGHT

$$\Gamma \vdash_{\mathsf{iFJ}} e' \equiv d' : \mathit{Object}$$

- *Case* rule EQUIV-CAST: Then we have

$$\begin{aligned} e &= \textsf{\textbf{cast}}(U, e'') \\ d &= \textsf{\textbf{cast}}(U, d'') \\ \Gamma \vdash_{\mathsf{iFJ}} e'' &\equiv d'' : \mathit{Object} \\ \vdash_{\mathsf{iFJ}} U &\leq T \end{aligned} \tag{C.3.15}$$

It is obvious that $e \longmapsto_{\mathsf{iFJ}} e'$ must have been derived either through rule DYN-CAST-IFJ or rule DYN-CAST-WRAP-IFJ. In both cases, we have

$$\begin{aligned} e'' &= v \\ \mathsf{unwrap}(v) &= \textsf{\textbf{new}}\, N(\overline{v}) \end{aligned}$$

We then get by Lemma C.3.17 for some value w that

$$d'' = w$$

By Lemma C.3.16

$$\begin{aligned} \mathsf{unwrap}(w) &= \textsf{\textbf{new}}\, N(\overline{w}) \\ \Gamma \vdash_{\mathsf{iFJ}} \textsf{\textbf{new}}\, N(\overline{v}) &\equiv \textsf{\textbf{new}}\, N(\overline{w}) : N \end{aligned} \tag{C.3.16}$$

Case distinction on the rule used to derive $e \longmapsto_{\mathsf{iFJ}} e'$.

  - *Case* rule DYN-CAST-IFJ: Then

$$\begin{aligned} \vdash_{\mathsf{iFJ}} N &\leq U \\ e' &= \textsf{\textbf{new}}\, N(\overline{v}) \end{aligned} \tag{C.3.17}$$

Thus, by rule DYN-CAST-IFJ

$$\underbrace{d}_{=\textsf{\textbf{cast}}(U,w)} \longmapsto_{\mathsf{iFJ}} \underbrace{\textsf{\textbf{new}}\, N(\overline{w})}_{:=d'}$$

Finally, we get with (C.3.15), (C.3.17), (C.3.16), transitivity of subtyping and an application of Lemma C.3.13 that

$$\Gamma \vdash_{\mathsf{iFJ}} e' \equiv d' : T$$

  - *Case* rule DYN-CAST-WRAP-IFJ: Then

$$\begin{aligned} U &= I \\ \text{not } \vdash_{\mathsf{iFJ}} N &\leq U \\ \textsf{\textbf{class}}\ & \mathit{Dict}^{I,M} \ldots \\ \vdash_{\mathsf{iFJ}} N &\leq M \\ e' &= \textsf{\textbf{new}}\, \mathit{Wrap}^{I}(\textsf{\textbf{new}}\, N(\overline{v})) \end{aligned} \tag{C.3.18}$$

By rule DYN-CAST-WRAP-IFJ then

$$d \longmapsto_{\mathsf{iFJ}} \underbrace{\textsf{\textbf{new}}\, \mathit{Wrap}^{I}(\textsf{\textbf{new}}\, N(\overline{w}))}_{=:d'}$$

With (C.3.16) and Lemma C.3.13 we get

$$\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\, N(\overline{v}) \equiv \textbf{new}\, N(\overline{w}) : \mathit{Object}$$

With (C.3.18), the definition of d', rule EQUIV-NEW-WRAP, (C.3.15), and Lemma C.3.13 then

$$\Gamma \vdash_{\mathsf{iFJ}} e' \equiv d' : T$$

End case distinction on the rule used to derive $e \longmapsto_{\mathsf{iFJ}} e'$.

- *Case* rule EQUIV-GETDICT: Then we have

$$\begin{aligned} e &= \textbf{getdict}(I, e'') \\ d &= \textbf{getdict}(I, d'') \\ \Gamma \vdash_{\mathsf{iFJ}} e'' &\equiv d'' : \mathit{Object} \\ \vdash_{\mathsf{iFJ}} \mathit{Dict}^I &\leq T \end{aligned} \tag{C.3.19}$$

From $e \longmapsto_{\mathsf{iFJ}} e'$ we get

$$\begin{gathered} e'' = v \\ \mathsf{unwrap}(e'') = \textbf{new}\, N(\overline{v}) \\ \mathsf{mindict}_{\mathsf{iFJ}}\{\textbf{class}\ \mathit{Dict}^{I,N'} \ldots \mid \vdash_{\mathsf{iFJ}} N \leq N'\} = M \\ e' = \textbf{new}\, M() \end{gathered}$$

We then get by Lemma C.3.17 for some value w that

$$d'' = w$$

By Lemma C.3.16

$$\mathsf{unwrap}(w) = \textbf{new}\, N(\overline{w})$$

Thus, by rule DYN-GETDICT-IFJ

$$d \longmapsto_{\mathsf{iFJ}} \underbrace{\textbf{new}\, M()}_{=:d'}$$

By well-formedness criterion WF-IFJ-5 and rule MINDICT-IFJ we get

$$\vdash_{\mathsf{iFJ}} M \leq \mathit{Dict}^I$$

Thus, with rule EQUIV-NEW-CLASS, (C.3.19), Lemma C.3.3, and transitivity of subtyping

$$\Gamma \vdash_{\mathsf{iFJ}} e' \equiv d' : T$$

- *Case* rule EQUIV-LET: Thus, together with $e \longmapsto_{\mathsf{iFJ}} e'$

$$\begin{gathered} e = (\textbf{let}\, U\, x = v\, \textbf{in}\, e_2) \\ e' = [v/x]e_2 \\ d = (\textbf{let}\, U\, x = d_1\, \textbf{in}\, d_2) \\ \Gamma \vdash_{\mathsf{iFJ}} v \equiv d_1 : U \\ \Gamma, x : U \vdash_{\mathsf{iFJ}} e_2 \equiv d_2 : T \end{gathered}$$

From Lemma C.3.17 we get for some value w that $d_1 = w$. By rule DYN-LET-IFJ then

$$d \longmapsto_{\mathsf{iFJ}} \underbrace{[w/x]d_2}_{=:d'}$$

Moreover, by Theorem 4.15

$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{[v/x]e_2}_{=e'} \equiv d' : T$$

End case distinction on the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e \equiv d : T$. □

Lemma C.3.22. *If* $\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_1] \equiv d : T$ *and* $e_1 \longmapsto_{\mathsf{iFJ}} e_2$ *then there exist* $\mathcal{E}'$, d_1, *and* d_2 *such that* $d = \mathcal{E}'[d_1]$ *and* $d_1 \longrightarrow_{\mathsf{iFJ}} d_2$ *and* $\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_2] \equiv \mathcal{E}'[d_2] : T$.

Proof. The proof of this claim is by induction on the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_1] \equiv d : T$.
Case distinction on the form of $\mathcal{E}$.

- *Case* $\mathcal{E} = \square$: Follows with Lemma C.3.21 for $\mathcal{E}' = \square$.
- *Case* $\mathcal{E} = \mathcal{E}''.f$: If the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_1] \equiv d : T$ is EQUIV-FIELD, then the claim follows by inverting the rule and from the I.H. Otherwise, the derivation ends with rule EQUIV-FIELD-WRAPPED. Then

$$\begin{gathered} f = \mathit{wrapped} \\ T = \mathit{Object} \\ \mathcal{E}''[e_1] = \textsf{new}\ \mathit{Wrap}^I(e_1') \\ d = \textsf{new}\ \mathit{Wrap}^J(d').\mathit{wrapped} \\ \Gamma \vdash_{\mathsf{iFJ}} e_1' \equiv d' : \mathit{Object} \end{gathered}$$

  Expressions of the form $\textsf{new}\ \mathit{Wrap}^I(e_1')$ are not reducible via $\longmapsto_{\mathsf{iFJ}}$, so $\mathcal{E}'' \neq \square$. Thus

$$\begin{gathered} \mathcal{E}'' = \textsf{new}\ \mathit{Wrap}^I(\mathcal{E}''') \\ e_1' = \mathcal{E}'''[e_1] \end{gathered}$$

  Applying the I.H. yields existence of $\mathcal{E}_4, d_1, d_2$ with

$$\begin{gathered} d' = \mathcal{E}_4[d_1] \\ d_1 \longrightarrow_{\mathsf{iFJ}} d_2 \\ \Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}'''[e_2] \equiv \mathcal{E}_4[d_2] : \mathit{Object} \end{gathered}$$

  Define $\mathcal{E}' := \textsf{new}\ \mathit{Wrap}^I(\mathcal{E}_4).\mathit{wrapped}$. Then $\mathcal{E}'[d_1] = d$. Moreover, an application of rule EQUIV-FIELD-WRAPPED yields

$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{\textsf{new}\ \mathit{Wrap}^I(\mathcal{E}'''[e_2]).\mathit{wrapped}}_{=\mathcal{E}''[e_2].\mathit{wrapped}=\mathcal{E}[e_2]} \equiv \underbrace{\textsf{new}\ \mathit{Wrap}^J(\mathcal{E}_4[d_2]).\mathit{wrapped}}_{=\mathcal{E}'[d_2]} : T$$

- *Case* $\mathcal{E} = \mathcal{E}''.m(\overline{e'})$: Follows by inverting rule EQUIV-INVOKE and the I.H.
- *Case* $\mathcal{E} = e.m(\overline{v}, \mathcal{E}'', \overline{e'})$: Follows by inverting rule EQUIV-INVOKE and the I.H.
- *Case* $\mathcal{E} = \textsf{new}\ N(\overline{v}, \mathcal{E}'', \overline{e'})$:
  Case distinction on the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_1] \equiv d : T$.

- *Case* rule EQUIV-NEW-CLASS: Follows by the I.H.
- *Case* rule EQUIV-NEW-WRAP: Follows by the I.H.
- *Case* rule EQUIV-NEW-OBJECT-LEFT: Then

$$N = \mathit{Wrap}^I$$
$$\overline{v} = \bullet = \overline{e'}$$
$$\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}''[e_1] \equiv d : \mathit{Object}$$

Applying the I.H. yields the existence of $\mathcal{E}', d_1, d_2$ such that

$$d = \mathcal{E}'[d_1]$$
$$d_1 \longrightarrow_{\mathsf{iFJ}} d_2$$
$$\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}''[e_2] \equiv \mathcal{E}'[d_2] : \mathit{Object}$$

By rule EQUIV-NEW-OBJECT-LEFT, we also have

$$\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_2] \equiv \mathcal{E}'[d_2] : \mathit{Object}$$

- *Case* rule EQUIV-NEW-OBJECT-RIGHT: Then

$$d = \textsf{new } \mathit{Wrap}^I(d')$$
$$\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_1] \equiv d : \mathit{Object}$$

Applying the I.H. yields the existence of $\mathcal{E}''', d_1, d_2$ such that

$$d = \mathcal{E}'''[d_1]$$
$$d_1 \longrightarrow_{\mathsf{iFJ}} d_2$$
$$\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_2] \equiv \mathcal{E}'''[d_2] : \mathit{Object}$$

Define $\mathcal{E}' = \textsf{new } \mathit{Wrap}^I(\mathcal{E}''')$. Then, by rule EQUIV-NEW-OBJECT-RIGHT

$$\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_2] \equiv \mathcal{E}'[d_2] : \mathit{Object}$$

- *Case* other rule: Impossible.

End case distinction on the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} \mathcal{E}[e_1] \equiv d : T$.

- *Case* $\mathcal{E} = \textsf{cast}(U, \mathcal{E}'')$: Follows by inverting rule EQUIV-CAST and the I.H.
- *Case* $\mathcal{E} = \textsf{getdict}(I, \mathcal{E}'')$: Follows by inverting rule EQUIV-GETDICT and the I.H.
- *Case* $\mathcal{E} = \textsf{let}\, U\, x = \mathcal{E}''\, \textsf{in}\, e$: Follows by inverting rule EQUIV-LET and the I.H.

End case distinction on the form of $\mathcal{E}$. □

Proof of Theorem 4.16. From $e \longrightarrow_{\mathsf{iFJ}} e'$ we get by inverting rule DYN-CONTEXT the existence of an evaluation context $\mathcal{E}$ and expressions e_1, e_2 such that $e = \mathcal{E}[e_1]$ and $e' = \mathcal{E}[e_2]$ and $e_1 \longmapsto_{\mathsf{iFJ}} e_2$. Using Lemma C.3.22, we get the existence of an evaluation context $\mathcal{E}'$ and expressions d_1, d_2 such that $d = \mathcal{E}'[d_1]$ and $d_1 \longrightarrow_{\mathsf{iFJ}} d_2$ and $\Gamma \vdash_{\mathsf{iFJ}} e' \equiv \mathcal{E}'[d_2] : T$. By Lemma C.3.19 then $d \longrightarrow_{\mathsf{iFJ}} \mathcal{E}'[d_2]$. Defining $d' := \mathcal{E}'[d_2]$ finishes the proof. □

C.3.4 Proof of Theorem 4.18

Theorem 4.18 states that $\equiv$ is sound with respect to contextual equivalence.

Proof of Theorem 4.18. We get with Lemma C.3.9 that

$$\begin{gathered}\Gamma \vdash_{\mathsf{iFJ}} e_1 : T_1 \\ \vdash_{\mathsf{iFJ}} T_1 \leq T \\ \Gamma \vdash_{\mathsf{iFJ}} e_2 : T_2 \\ \vdash_{\mathsf{iFJ}} T_2 \leq T\end{gathered}$$

Now assume that d is an expression with $\Gamma, \chi : T \vdash_{\mathsf{iFJ}} d : U$ for some type U. With Lemma C.3.3 then

$$\Gamma, \chi : T \vdash_{\mathsf{iFJ}} d \equiv d : U$$

Thus, Theorem 4.15 yields

$$\Gamma \vdash_{\mathsf{iFJ}} [e_1/\chi]d \equiv [e_2/\chi]d : U$$

W.l.o.g., assume that $[e_1/\chi]d$ terminates; that is,

$$[e_1/\chi]d \longrightarrow_{\mathsf{iFJ}} d_1 \longrightarrow_{\mathsf{iFJ}} \dots \longrightarrow_{\mathsf{iFJ}} d_n$$

for some normal form d_n. We proceed by induction on n to show that $[e_2/\chi]d$ terminates as well.

- If $n = 0$ then $[e_1/\chi]d$ is already a normal form. Thus, $[e_2/\chi]d$ is also a normal form, otherwise Theorem 4.16 and Lemma C.3.4 would lead to a contradiction.
- If $n > 0$ then, with Theorem 4.16,

  $$\begin{gathered}[e_2/\chi]d \longrightarrow_{\mathsf{iFJ}} d'_1 \\ \Gamma \vdash_{\mathsf{iFJ}} d_1 \equiv d'_1 : U\end{gathered}$$

  Applying the I.H. proves that d'_1 terminates, so $[e_2/\chi]d$ terminates as well. □

C.3.5 Proof of Theorem 4.19

Theorem 4.19 states that translation and single-step evaluation in CoreGI$^\flat$ commute modulo wrappers.

Lemma C.3.23 (Transitivity of CoreGI$^\flat$ subtyping). *For all types T it holds that $\vdash^\flat T \leq T \rightsquigarrow \mathsf{nil}$.*

Proof. Easy because the relations $\trianglelefteq^\flat_{\mathsf{c}}$ and $\trianglelefteq^\flat_{\mathsf{i}}$ are reflexive. □

Lemma C.3.24. *If $\vdash^\flat I \leq T \rightsquigarrow \mathsf{nil}$ then either $T = Object$ or $T = J$ for some J with $I \trianglelefteq^\flat_{\mathsf{i}} J$.*

Proof. The derivation of $\vdash^\flat I \leq T \rightsquigarrow \mathsf{nil}$ must end with rule SUB-KERNEL$^\flat$. Thus, $\vdash^{\flat'} I \leq T$. The last rule in this derivation is either SUB-OBJECT$^\flat$ or SUB-IFACE$^\flat$. In both cases, the claim obviously holds. □

Lemma C.3.25. *If $\vdash^\flat T \leq N \rightsquigarrow I^?$ then $I^? = \mathsf{nil}$ and either $N = Object$ or $N = C, T = D$ for some C, D with $D \trianglelefteq^\flat_{\mathsf{c}} C$.*

Proof. The derivation of $\vdash^\flat T \leq N \rightsquigarrow I^?$ must end with rule SUB-KERNEL$^\flat$. Thus, $I^? = \mathsf{nil}$ and $\vdash^{\flat'} T \leq N$. Inspecting the rules defining this relation finishes the proof. □

Lemma C.3.26 (Transitivity of $\mathsf{CoreGI}^\flat$ kernel subtyping). *If* $\vdash^{\flat'} T \le U$ *and* $\vdash^{\flat'} U \le V$ *then* $\vdash^{\flat'} T \le V$.

Proof. It is straightforward to verify that the relations $\trianglelefteq^\flat_{\mathbf{c}}$ and $\trianglelefteq^\flat_{\mathbf{i}}$ are transitive. The original claim now follows by case distinction on the last rules in the derivations of $\vdash^{\flat'} T \le U$ and $\vdash^{\flat'} U \le V$. □

Definition C.3.27. The function $\mathsf{trans}(I^?, T, J^?)$ is defined as follows:

$$\mathsf{trans}(I^?, T, J^?) = \begin{cases} J^? & \text{if } J^? \neq \mathsf{nil} \\ T & \text{if } J^? = \mathsf{nil},\ I^? \neq \mathsf{nil}, \text{ and } T \neq \mathit{Object} \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

Lemma C.3.28. *If* $\vdash^\flat T \le U \rightsquigarrow I^?$ *and* $\vdash^\flat U \le V \rightsquigarrow J^?$ *then* $\vdash^\flat T \le V \rightsquigarrow \mathsf{trans}(I^?, V, J^?)$.

Proof. We proceed by cast distinction on the rules used to derive $\vdash^\flat T \le U \rightsquigarrow I^?$ and $\vdash^\flat U \le V \rightsquigarrow J^?$.
Case distinction on the rules used to derive $\vdash^\flat T \le U \rightsquigarrow I^?, \vdash^\flat U \le V \rightsquigarrow J^?$.

- *Case* rules SUB-KERNEL$^\flat$ / SUB-KERNEL$^\flat$: In this case, the claim follows from Lemma C.3.26.
- *Case* rules SUB-KERNEL$^\flat$ / SUB-IMPL$^\flat$: In this case, the claim follows with Lemma C.3.26 and rule SUB-IMPL$^\flat$.
- *Case* rules SUB-IMPL$^\flat$ / SUB-KERNEL$^\flat$: We then have

$$\begin{gathered} U = I \\ I^? = I \\ \vdash^{\flat'} T \le N \\ \textbf{implementation } I\ [N] \ldots \\ \vdash^{\flat'} I \le V \\ J^? = \mathsf{nil} \end{gathered} \tag{C.3.20}$$

  Case distinction on the form of V.

  - *Case* $V = \mathit{Object}$: Then $\mathsf{trans}(I, V, \mathsf{nil}) = \mathsf{nil}$ and $\vdash^\flat T \le \mathit{Object} \rightsquigarrow \mathsf{nil}$.
  - *Case* $V = C$: Impossible.
  - *Case* $V = J$: Then $I \trianglelefteq^\flat_{\mathbf{i}} J$ and $\mathsf{trans}(I, V, \mathsf{nil}) = V = J$. Using well-formedness criterion WF-IMPL-1 and Lemma C.3.26, an easy induction shows

$$\begin{gathered} \textbf{implementation } J\ [M] \ldots \\ \vdash^{\flat'} N \le M \end{gathered}$$

    By Lemma C.3.26 and (C.3.20) then $\vdash^{\flat'} T \le M$, so with rule SUB-IMPL$^\flat$

$$\vdash^\flat T \le J \rightsquigarrow J$$

  End case distinction on the form of V.

- *Case* rules SUB-IMPL$^\flat$ / SUB-IMPL$^\flat$: Then

$$\begin{gathered} U = I \\ I^? = I \\ V = J \\ J^? = J \\ \textbf{implementation } J\,[\,M\,]\,\ldots \\ \vdash^{\flat'} I \leq M \\ \mathsf{trans}(I^?, V, J^?) = J \end{gathered}$$

  By examining the rules defining the $\vdash^{\flat'} \cdot \leq \cdot$ relation, we see that $M = \mathit{Object}$. Thus, by rules SUB-OBJECT$^\flat$ and SUB-IMPL$^\flat$

$$\vdash^\flat T \leq J \rightsquigarrow J$$

End case distinction on the rules used to derive $\vdash^\flat T \leq U \rightsquigarrow I^?, \vdash^\flat U \leq V \rightsquigarrow J^?$. □

Lemma C.3.29. *If* $\mathsf{mtype}^\flat(m, T) = \mathit{msig} \rightsquigarrow I^?$ *and* $\vdash^\flat T' \leq T \rightsquigarrow J^?$ *then* $\mathsf{mtype}^\flat(m, T') = \mathit{msig} \rightsquigarrow I'^?$ *such that*

$$I'^? = \begin{cases} J' & \text{if } I^? = \mathsf{nil} \text{ and } J^? = J, \text{ where } J' \text{ such that } J \trianglelefteq^\flat_{\mathsf{i}} J' \\ I^? & \text{otherwise} \end{cases}$$

Moreover, $I'^? \neq I^?$ *implies that* $I'^? \neq \mathsf{nil}$ *is the interface that defines* m*.*

Proof. We proceed by case distinction on whether m is a class or interface method.
Case distinction on the form of m.

- *Case* $m = m^{\mathsf{c}}$: Then $I^? = \mathsf{nil}$ and $T = C$. From Lemma C.2.3 we know that $J^? = \mathsf{nil}$. With Lemma C.3.25 then $T' = C'$ for some C' such that $C' \trianglelefteq^\flat_{\mathsf{c}} C$. An easy induction on the derivation of $C' \trianglelefteq^\flat_{\mathsf{c}} C$ then shows

$$\mathsf{mtype}^\flat(m, C') = \mathit{msig}' \rightsquigarrow \mathsf{nil}$$

  Moreover, the premise of rule OK-OVERRIDE$^\flat$ ensures $\mathit{msig} = \mathit{msig}'$. Note that $\mathsf{nil} = I^? = I'^?$.

- *Case* $m = m^{\mathsf{i}}$: Hence, the derivation of $\mathsf{mtype}^\flat(m, T) = \mathit{msig} \rightsquigarrow I^?$ ends with rule MTYPE-IFACE$^\flat$, so we have

$$\begin{gathered} \textbf{interface } I \textbf{ extends } \overline{J}\,\{\,\overline{m : \mathit{msig}}\,\} \\ \vdash^\flat T \leq I \rightsquigarrow I^? \\ m = m_k \\ \mathit{msig} = \mathit{msig}_k \end{gathered}$$

  Case distinction on the form of $I^?$ and the form of $J^?$.

  - *Case* $I^? = \mathsf{nil}$ and $J^? \neq \mathsf{nil}$: Then $J^? = J$ for some J. By Lemma C.2.3 and Lemma C.2.5

$$\begin{gathered} T = J \\ J \trianglelefteq_{\mathsf{i}} I \end{gathered}$$

We then get $\vdash^\flat T' \leq I \rightsquigarrow I$ by Lemma C.3.28 (note $\mathsf{trans}(J^?, I, I^?) = \mathsf{trans}(J, I, \mathsf{nil}) = I$) so by rule MTYPE-IFACE$^\flat$

$$\mathsf{mtype}^\flat(m, T') = \mathit{msig} \rightsquigarrow I$$

and I is the interface defining m. Setting $I'^? := I$ finishes this case.

- *Case* $I^? \neq \mathsf{nil}$ or $J^? = \mathsf{nil}$: We get $\vdash^\flat T' \leq I \rightsquigarrow I^?$ by Lemma C.3.28 (note that $I^? = \mathsf{nil}$ implies $J^? = \mathsf{nil}$). The claim then follows by rule MTYPE-IFACE$^\flat$.

End case distinction on the form of $I^?$ and the form of $J^?$.

End case distinction on the form of m. □

Lemma C.3.30. *If* $\mathsf{fields}^\flat(C) = \overline{U\, f}$ *and* $\vdash^\flat T \leq C \rightsquigarrow I^?$ *then* $\mathsf{fields}^\flat(T) = \overline{U\, f}, \overline{V\, g}$ *and* $\overline{f}, \overline{g}$ *are pairwise disjoint.*

Proof. With $\vdash^\flat T \leq C \rightsquigarrow I^?$ and Lemma C.3.25 we get $I^? = \mathsf{nil}$, $T = D$, and $D \trianglelefteq^\flat_{\mathsf{c}} C$. A straightforward induction on the derivation of $D \trianglelefteq^\flat_{\mathsf{c}} C$ shows $\mathsf{fields}^\flat(T) = \overline{U\, f}, \overline{V\, g}$. The claim that $\overline{f}, \overline{g}$ are pairwise disjoint follows with well-formedness criterion WF$^\flat$-CLASS-1 □

Lemma C.3.31. *If* $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv \mathsf{wrap}(I^?, e_2) : T$ *and there exists* T', U *such that* $\vdash^\flat T' \leq T \rightsquigarrow I^?$ *and* $\vdash^\flat T \leq U \rightsquigarrow J^?$, *then it holds that* $\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J^?, e_1) \equiv \mathsf{wrap}(\mathsf{trans}(I^?, U, J^?), e_2) : U$.

Proof. We proceed by case distinction on the form of $J^?$.
Case distinction on the form of $J^?$.

- *Case* $J^? \neq \mathsf{nil}$: Then $J^? = J$ for some J and $\mathsf{trans}(I^?, U, J^?) = J$. By Lemma C.2.3 $U = J$. Assume that $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : \mathit{Object}$ holds. The claim then follows by rule EQUIV-NEW-WRAP. We now prove $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : \mathit{Object}$ by case distinction on the form of $I^?$.

  Case distinction on the form of $I^?$.

  - *Case* $I^? = \mathsf{nil}$: Then $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : T$ by the assumption and Lemma C.3.13 establishes the claim.
  - *Case* $I^? \neq \mathsf{nil}$: By Lemma C.2.3 $T = I$. Thus, the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv$ **new** $\mathit{Wrap}^I(e_2) : T$ (given in the assumption) must end with rule EQUIV-NEW-WRAP. Hence,

    $$e_1 = \textbf{new}\ \mathit{Wrap}^{I'}(e_1')$$
    $$\vdash_{\mathsf{iFJ}} I' \leq I$$
    $$\Gamma \vdash_{\mathsf{iFJ}} e_1' \equiv e_2 : \mathit{Object}$$

    We then get $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : \mathit{Object}$ by rule EQUIV-NEW-OBJECT-LEFT.

  End case distinction on the form of $I^?$.

- *Case* $J^? = \mathsf{nil}$: In this case, we perform another case distinction on the forms of $I^?$ and U.

  Case distinction on the forms of $I^?$ and U.

  - *Case* $I^? \neq \mathsf{nil}$ and $U \neq \mathit{Object}$: Thus, $I^? = I$ for some I and

    $$\mathsf{trans}(I^?, U, J^?) = U$$

    By Lemma C.2.3 and Lemma C.3.24 then

    $$T = I$$
    $$U = J \text{ for some } J$$
    $$I \trianglelefteq^\flat_{\mathsf{i}} J$$

Thus, the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv \textbf{new } \mathit{Wrap}^{I}(e_2) : T$ must end with an application of rule EQUIV-NEW-WRAP. Hence,

$$\begin{gathered} e_1 = \textbf{new } \mathit{Wrap}^{I'}(e_1') \\ \vdash_{\mathsf{iFJ}} I' \leq I \\ \Gamma \vdash_{\mathsf{iFJ}} e_1' \equiv e_2 : \mathit{Object} \end{gathered}$$

With $I \trianglelefteq_{\mathsf{i}}^{\flat} J$ we get by rule SUB-IFACE$^{\flat}$ and Lemma C.2.1 that $\vdash_{\mathsf{iFJ}} I \leq J$. With transitivity of subtyping we then have also $\vdash_{\mathsf{iFJ}} I' \leq J$. Thus, with rule EQUIV-NEW-WRAP

$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{e_1}_{=\mathsf{wrap}(J^?, e_1)} \equiv \underbrace{\textbf{new } \mathit{Wrap}^{J}(e_2)}_{=\mathsf{wrap}(\mathsf{trans}(I^?, U, J^?), e_2)} : \underbrace{J}_{=U}$$

as required.

- *Case* $I^? = \mathsf{nil}$ or $U = \mathit{Object}$: In this case, $\mathsf{trans}(I^?, U, J^?) = \mathsf{nil}$. Moreover, by Lemma C.2.2 $\vdash_{\mathsf{iFJ}} T \leq U$.
  - ∗ If $I^? = \mathsf{nil}$ then the claim follows with Lemma C.3.13.
  - ∗ If $I^? \neq \mathsf{nil}$ then $U = \mathit{Object}$, $I^? = I$ for some I, and, by Lemma C.2.3, $T = I$. Thus, the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv \textbf{new } \mathit{Wrap}^{I}(e_2) : T$ (from the assumption) must end with rule EQUIV-NEW-WRAP. Hence,

    $$\begin{gathered} e_1 = \textbf{new } \mathit{Wrap}^{I'}(e_1') \\ \Gamma \vdash_{\mathsf{iFJ}} e_1' \equiv e_2 : \mathit{Object} \end{gathered}$$

    We then get by rule EQUIV-NEW-OBJECT-LEFT that

    $$\Gamma \vdash_{\mathsf{iFJ}} e_1 \equiv e_2 : \underbrace{\mathit{Object}}_{=U}$$

End case distinction on the forms of $I^?$ and U.

End case distinction on the form of $J^?$. □

Lemma C.3.32. *If* $\Gamma \vdash_{\mathsf{iFJ}} e \equiv \mathsf{wrap}(I, e') : I$ *and* $\vdash_{\mathsf{iFJ}} I \leq J$ *then* $\Gamma \vdash_{\mathsf{iFJ}} e \equiv \mathsf{wrap}(J, e') : J$.

Proof. Case distinction on the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e \equiv \mathsf{wrap}(I, e') : I$.

- *Case* rule EQUIV-NEW-CLASS: Thus $e = \textbf{new } \mathit{Wrap}^{I}(e'')$, so with well-formedness criterion WF-IFJ-6 and the premise of the rule

  $$\Gamma \vdash_{\mathsf{iFJ}} e'' \equiv e' : \mathit{Object}$$

  It is now straightforward to verify that the claim follows by applying rule EQUIV-NEW-WRAP.
- *Case* rule EQUIV-NEW-WRAP: Then the claim follows with rule EQUIV-NEW-WRAP.
- *Case* any other rule: Impossible.

End case distinction on the last rule in the derivation of $\Gamma \vdash_{\mathsf{iFJ}} e \equiv \mathsf{wrap}(I, e') : I$. □

Lemma C.3.33. *If* $\mathsf{mtype}^{\flat}(m, T) = \mathit{msig} \rightsquigarrow \mathsf{nil}$ *then there exists a type* U *such that* $\vdash_{\mathsf{iFJ}} T \leq U$, $\mathsf{mtype}_{\mathsf{iFJ}}(m, U) = \mathit{msig}$, *and* $\mathsf{topmost}(U, m)$.

Proof. Follows with Lemma C.2.6 and Lemma C.3.2. □

Lemma C.3.34 (Substitution lemma for $\mathsf{CoreGI}^\flat$). *If $\Gamma, x : U \vdash^\flat e : T \leadsto d$ and $\Gamma \vdash^\flat e' : U' \leadsto d'$ with $\vdash^\flat U' \leq U \leadsto I^?$, then $\Gamma \vdash^\flat [e'/x]e : T' \leadsto d''$ with $\vdash^\flat T' \leq T \leadsto J^?$ and $\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]d \equiv \mathsf{wrap}(J^?, d'') : T$.*

Proof. We proceed by induction on the derivation of $\Gamma, x : U \vdash^\flat e : T \leadsto d$.
Case distinction on the last rule in the derivation of $\Gamma, x : U \vdash^\flat e : T \leadsto d$.

- *Case* rule EXP-VAR$^\flat$: Then $e = y = d$.

  Case distinction on whether or not $x = y$.

  – *Case* $x = y$: Then $[e'/x]e = e'$ and $T = U$. Thus, we have for $T' := U'$ and $d'' := d'$ and $J^? := I^?$ that

  $$\Gamma \vdash^\flat [e'/x]e : T' \leadsto d''$$
  $$\vdash^\flat T' \leq T \leadsto J^?$$

  With $\Gamma \vdash^\flat e' : U' \leadsto d'$ and Theorem 4.11 we get

  $$\Gamma \vdash_{\mathsf{iFJ}} d' : U'$$

  so with $\vdash^\flat U' \leq U \leadsto I^?$ and Lemma C.2.4

  $$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, d') : U''$$
  $$\vdash_{\mathsf{iFJ}} U'' \leq U$$

  for some U''. Moreover, $[\mathsf{wrap}(I^?, d')/x]d = \mathsf{wrap}(I^?, d')$, $T = U$, $I^? = J^?$, and $d'' = d'$, so with Lemma C.3.3

  $$\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]d \equiv \mathsf{wrap}(J^?, d'') : T$$

  as requested.

  – *Case* $x \neq y$: Then $[e'/x]e = e$ and $[\mathsf{wrap}(I^?, d')/x]d = d$. Define $d'' := d$, $T' := T$, and $J^? := \mathsf{nil}$, and we get by the assumptions and Lemma C.3.23

  $$\Gamma \vdash^\flat [e'/x]e : T' \leadsto d''$$
  $$\vdash^\flat T' \leq T \leadsto J^?$$

  Moreover, $\mathsf{wrap}(J^?, d'') = d$ and, with Theorem 4.11 $\Gamma \vdash_{\mathsf{iFJ}} d : T$, so with Lemma C.3.3

  $$\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]d \equiv \mathsf{wrap}(J^?, d'') : T$$

  End case distinction on whether or not $x = y$.

- *Case* rule EXP-FIELD$^\flat$: Then $e = e_0.f$. We get from the premise of the rule

$$\Gamma, x : U \vdash^\flat e_0 : C \leadsto e_0'$$
$$\mathsf{fields}^\flat(C) = \overline{V\, f}^n$$
$$f_j = f$$
$$V_j = T$$
$$d = e_0'.f$$

Applying the I.H. and Lemma C.2.3 yields

$$\begin{gathered}\Gamma \vdash^\flat [e'/x]e_0 : C' \rightsquigarrow e''_0 \\ \vdash^\flat C' \leq C \rightsquigarrow \mathsf{nil} \\ \Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]e'_0 \equiv e''_0 : C \qquad \text{(C.3.21)}\end{gathered}$$

By Lemma C.3.30 we have

$$\mathsf{fields}^\flat(C') = \overline{V\,f}, \overline{V'\,f'}$$

Thus, by rule EXP-FIELD

$$\Gamma \vdash^\flat [e'/x]e : T : \underbrace{e''_0.f}_{=:d''}$$

By Lemma C.2.7 and Lemma C.3.1 we know that there exists some D such that

$$\begin{gathered}\vdash_{\mathsf{iFJ}} C \leq D \\ \mathsf{defines\text{-}field}(D, f) \\ \mathsf{fields}_{\mathsf{iFJ}}(D) = \overline{V\,f}^m \\ m \geq j\end{gathered}$$

With (C.3.21) and Lemma C.3.13

$$\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]e'_0 \equiv e''_0 : D$$

Thus, by rule EQUIV-FIELD

$$\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x](e'_0.f) \equiv e''_0.f : T$$

Noting that $d = e'_0.f$ and $d'' = e''_0.f$, defining $T' := T$ and $J^? := \mathsf{nil}$, and applying Lemma C.3.23 to get $\vdash^\flat T' \leq T \rightsquigarrow J^?$ finishes this case.

- *Case* rule EXP-INVOKE$^\flat$: Then $e = e_o.m(\overline{e})$. We get from the premise of the rule

$$\begin{gathered}\Gamma, x : U \vdash^\flat e_0 : T_0 \rightsquigarrow d_0 \\ \mathsf{mtype}^\flat(m, T_0) = \overline{V\,x} \to T \rightsquigarrow I^?_0 \qquad \text{(C.3.22)} \\ (\forall i)\ \Gamma, x : U \vdash^\flat e_i : V'_i \rightsquigarrow d_i \\ (\forall i)\ \vdash^\flat V'_i \leq V_i \rightsquigarrow I^?_i \\ d'_0 = \mathsf{wrap}(I^?_0, d_0) \\ (\forall i)\ d'_i = \mathsf{wrap}(I^?_i, d_i) \qquad \text{(C.3.23)}\end{gathered}$$

and we have $d = d'_0.m(\overline{d'})$.

In the following, we define $\varphi := [e'/x]$ and $\varphi' := [\mathsf{wrap}(I^?, d')/x]$.

Applying the I.H. yields

$$\begin{gathered}\Gamma \vdash^\flat \varphi e_0 : T'_0 \rightsquigarrow d''_0 \qquad \text{(C.3.24)} \\ \vdash^\flat T'_0 \leq T_0 \rightsquigarrow J^?_0 \qquad \text{(C.3.25)} \\ \Gamma \vdash_{\mathsf{iFJ}} \varphi' d_0 \equiv \mathsf{wrap}(J^?_0, d''_0) : T_0 \qquad \text{(C.3.26)} \\ (\forall i)\ \Gamma \vdash^\flat \varphi e_i : V''_i \rightsquigarrow d''_i \qquad \text{(C.3.27)} \\ (\forall i)\ \vdash^\flat V''_i \leq V'_i \rightsquigarrow J^?_i \\ (\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} \varphi' d_i \equiv \mathsf{wrap}(J^?_i, d''_i) : V'_i\end{gathered}$$

With Lemma C.3.28

$$(\forall i) \;\; \vdash^{\flat} V_i'' \leq V_i \leadsto \mathsf{trans}(J_i^?, V_i, I_i^?) \tag{C.3.28}$$

and with Lemma C.3.31

$$(\forall i) \; \Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I_i^?, \varphi' d_i) \equiv \mathsf{wrap}(\mathsf{trans}(J_i^?, V_i, I_i^?), d_i'') : V_i \tag{C.3.29}$$

Moreover, we get from Lemma C.3.29 that

$$\begin{gathered} \mathsf{mtype}^{\flat}(m, T_0') = \overline{V\,x} \to T \leadsto I_0'^? \\ I_0'^? = \begin{cases} J_0' & \text{if } I_0^? = \mathsf{nil} \text{ and } J_0^? = J_0 \text{ such that } J_0 \trianglelefteq_{\mathsf{i}}^{\flat} J_0' \\ I_0^? & \text{otherwise} \end{cases} \\ I_0'^? \neq I_0^? \text{ implies that } I_0'^? \neq \mathsf{nil} \text{ defines } m \end{gathered} \tag{C.3.30}$$

We get with (C.3.24), (C.3.30), (C.3.27), (C.3.28), and rule EXP-INVOKE$^{\flat}$ that

$$\Gamma \vdash^{\flat} \varphi(e_0.m(\overline{e})) : T \leadsto d_0'''.m(\overline{d'''}) \tag{C.3.31}$$

where

$$\begin{gathered} d_0''' = \mathsf{wrap}(I_0'^?, d_0'') \\ (\forall i) \; d_i''' = \mathsf{wrap}(\mathsf{trans}(J_i^?, V_i, I_i^?), d_i'') \end{gathered} \tag{C.3.32}$$

Our goal is now to prove that

$$\Gamma \vdash_{\mathsf{iFJ}} \varphi'(d_0'.m(\overline{d'})) \equiv d_0'''.m(\overline{d'''}) : T \tag{C.3.33}$$

Defining $d'' := d_0'''.m(\overline{d'''})$ and $J^? := \mathsf{nil}$ then finishes the claim because we have (C.3.31), $d = d_0'.m(\overline{d'})$, and $\vdash^{\flat} T \leq T \leadsto \mathsf{nil}$ by Lemma C.3.23.

We now prove (C.3.33).

Case distinction on $I_0^?$ and $J_0^?$.

- *Case* $J_0^? = J_0$ and $I_0^? = \mathsf{nil}$: Then $I_0'^? = J_0'$ for some J_0' defining m such that $J_0 \trianglelefteq_{\mathsf{i}}^{\flat} J_0'$. Thus, by definition of d_0' and d_0''' we get

  $$\begin{gathered} d_0' = d_0 \\ d_0''' = \mathsf{wrap}(J_0', d_0'') \end{gathered}$$

  With (C.3.25) and Lemma C.2.3 we get $T_0 = J_0$. By Lemma C.2.1, we know that $J_0 \trianglelefteq_{\mathsf{i}}^{\flat} J_0'$ implies $\vdash_{\mathsf{iFJ}} J_0 \leq J_0'$, so with (C.3.26) and Lemma C.3.32 we have

  $$\Gamma \vdash_{\mathsf{iFJ}} \varphi' d_0' \equiv \underbrace{\mathsf{wrap}(J_0', d_0'')}_{=d_0'''} : J_0'$$

  Because J_0' defines m, we have

  $$\mathsf{topmost}(J_0', m)$$

  With $T_0 = J_0$, $J_0 \trianglelefteq_{\mathsf{i}}^{\flat} J_0'$, Convention 4.2, and (C.3.22) it is easy to see that

  $$\mathsf{mtype}_{\mathsf{iFJ}}(m, J_0') = \overline{V\,x} \to T$$

  Using (C.3.29), (C.3.23), and (C.3.32) we get

  $$(\forall i) \; \Gamma \vdash_{\mathsf{iFJ}} \varphi' d_i' \equiv d_i''' : V_i$$

  Thus, rule EQUIV-INVOKE shows that (C.3.33) holds.

  - *Case* $J_0^? = \mathsf{nil}$ or $I_0^? \neq \mathsf{nil}$: In this case, we have $I_0'^? = I_0^?$.
    Case distinction on the form of $I_0^?$.
    - *Case* $I_0^? = I_0$: With (C.3.22), Lemma C.2.8, and the definition of $\mathsf{topmost}$, we see that

$$\begin{gathered}\vdash^\flat T_0 \leq I_0 \rightsquigarrow I_0 \qquad \text{(C.3.34)}\\ \mathsf{mtype}_{\mathsf{iFJ}}(m, I_0) = \overline{V\,x} \to T\\ \mathsf{topmost}(I_0, m)\end{gathered}$$

      With (C.3.26), (C.3.25), (C.3.34), and Lemma C.3.31, we get

$$\Gamma \vdash_{\mathsf{iFJ}} \varphi' \mathsf{wrap}(I_0, d_0) \equiv \mathsf{wrap}(\underbrace{\mathsf{trans}(J_0^?, I_0, I_0)}_{=I_0}, d_0'') : I_0$$

    - *Case* $I_0^? = \mathsf{nil}$: In this case also $J_0^? = \mathsf{nil}$. With (C.3.22) and Lemma C.3.33 we get the existence of a type W such that

$$\begin{gathered}\mathsf{mtype}_{\mathsf{iFJ}}(m, W) = \overline{V\,x} \to T\\ \mathsf{topmost}(W, m)\\ \vdash_{\mathsf{iFJ}} T_0 \leq W\end{gathered}$$

      Using (C.3.26), the fact that $I_0^? = \mathsf{nil} = J_0^?$, and Lemma C.3.13 we get

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I_0^?, \varphi' d_0) \equiv \mathsf{wrap}(I_0^?, d_0'') : W$$

    End case distinction on the form of $I_0^?$.
    In both cases, we have seen that there exists a type W such that

$$\begin{gathered}\Gamma \vdash_{\mathsf{iFJ}} \varphi' \overbrace{\mathsf{wrap}(I_0^?, d_0)}^{=d_0'} \equiv \overbrace{\mathsf{wrap}(I_0^?, d_0'')}^{=d_0'''} : W\\ \mathsf{mtype}_{\mathsf{iFJ}}(m, W) = \overline{V\,x} \to T\\ \mathsf{topmost}(W, m)\end{gathered}$$

    With (C.3.29) we get

$$(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} \varphi' d_i' \equiv d_i''' : V_i$$

    Using rule EQUIV-INVOKE, we conclude that (C.3.33) holds.
  End case distinction on $I_0^?$ and $J_0^?$.
  This finishes the proof of (C.3.33) and thus the proof of this case.

- *Case* rule EXP-NEW$^\flat$: Then $e = \textbf{new}\ N(\overline{e})$ and we get from the premise of the rule

$$\begin{gathered}(\forall i)\ \Gamma, x : U \vdash^\flat e_i : T_i \rightsquigarrow d_i\\ \vdash^\flat N\ \mathsf{ok}\\ \mathsf{fields}^\flat(N) = \overline{U\,f}\\ (\forall i)\ \vdash^\flat T_i \leq U_i \rightsquigarrow J_i^?\\ (\forall i)\ d_i' = \mathsf{wrap}(J_i^?, d_i)\\ d = \textbf{new}\ N(\overline{d'})\\ T = N\end{gathered}$$

Applying the I.H. yields for all suitable i

$$\begin{gathered}\Gamma \vdash^\flat [e'/x]e_i : T'_i \leadsto d''_i \\ \vdash^\flat T'_i \le T_i \leadsto {J'}^?_i \\ \Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]d_i \equiv \mathsf{wrap}({J'}^?_i, d''_i) : T_i\end{gathered}$$

By Lemma C.3.28, we get

$$(\forall i) \;\; \vdash^\flat T'_i \le U_i \leadsto \mathsf{trans}({J'}^?_i, U_i, J^?_i)$$

Define

$$(\forall i) \;\; d'''_i := \mathsf{wrap}(\mathsf{trans}({J'}^?_i, U_i, J^?_i), d''_i)$$

Now by rule EXP-NEW$^\flat$

$$\Gamma \vdash^\flat [e'/x]e : T \leadsto \underbrace{\mathbf{new}\, N(\overline{d'''})}_{=:d''}$$

Define $T' := T$ and $J^? := \mathsf{nil}$. Then by Lemma C.3.23 $\vdash^\flat T' \le T \leadsto J^?$. Moreover, by Lemma C.3.31

$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{\mathsf{wrap}(J^?_i, [\mathsf{wrap}(I^?, d')/x]d_i)}_{=[\mathsf{wrap}(I^?,d')/x]d'_i} \equiv d'''_i : U_i$$

Then by rule EQUIV-NEW-CLASS

$$\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]\, \underbrace{\mathbf{new}\, N(\overline{d'})}_{=d} \equiv \underbrace{\mathbf{new}\, N(\overline{d'''})}_{=\mathsf{wrap}(J^?,d'')} : T$$

- *Case* rule EXP-CAST$^\flat$: Then $e = (T)\, e_0$ and from the premise of the rule

  $$\begin{gathered}\vdash^\flat T \;\mathsf{ok} \\ \Gamma, x : U \vdash^\flat e_0 : V \leadsto d_0 \\ d = \mathbf{cast}(T, d_0)\end{gathered}$$

  Applying the I.H. yields

  $$\Gamma \vdash^\flat [e'/x]e_0 : V' \leadsto d'_0$$
  $$\vdash^\flat V' \le V \leadsto {J'}^? \tag{C.3.35}$$
  $$\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]d_0 \equiv \mathsf{wrap}({J'}^?, d'_0) : V \tag{C.3.36}$$

  We get with rule EXP-CAST$^\flat$

  $$\Gamma \vdash^\flat [e'/x]e : T \leadsto \underbrace{\mathbf{cast}(T, d'_0)}_{=:d''}$$

  Define $T' := T$ and $J^? := \mathsf{nil}$. Then by Lemma C.3.23 $\vdash^\flat T' \le T \leadsto J^?$.
  Obviously, $\vdash^\flat V \le \mathit{Object} \leadsto \mathsf{nil}$. Thus, we get with (C.3.35), (C.3.36), and Lemma C.3.31 that

  $$\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]d_0 \equiv \mathsf{wrap}(\mathsf{trans}({J'}^?, \mathit{Object}, \mathsf{nil}), d'_0) : \mathit{Object}$$

By Definition C.3.27, we have $\mathsf{trans}(J'^?, \mathit{Object}, \mathsf{nil}) = \mathsf{nil}$. Hence,

$$\Gamma \vdash_{\mathsf{iFJ}} [\mathsf{wrap}(I^?, d')/x]d_0 \equiv d_0' : \mathit{Object}$$

Rule EQUIV-CAST then yields

$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{\mathbf{cast}(T, [\mathsf{wrap}(I^?, d')/x]d_0)}_{=[\mathsf{wrap}(I^?, d')/x]d} \equiv \underbrace{\mathbf{cast}(T, d_0')}_{=\mathsf{wrap}(J^?, d'')} : T$$

as required.

End case distinction on the last rule in the derivation of $\Gamma, x : U \vdash^\flat e : T \rightsquigarrow d$. □

Lemma C.3.35. *If* $\Gamma \vdash^\flat e : T \rightsquigarrow d$ *then* $\mathsf{fv}(e) = \mathsf{fv}(d) \subseteq \mathsf{dom}(\Gamma)$.

Proof. Straightforward induction on the derivation of $\Gamma \vdash^\flat e : T \rightsquigarrow d$. □

Lemma C.3.36 (Multi-variable substitution lemma for $\mathsf{CoreGI}^\flat$). *If* $\Gamma, \overline{x : U}^n \vdash^\flat e : T \rightsquigarrow d$ *and, for all* $i \in [n]$, $\Gamma \vdash^\flat e_i : U_i' \rightsquigarrow d_i$ *and* $\vdash^\flat U_i' \leq U_i \rightsquigarrow I_i^?$, *then* $\Gamma \vdash^\flat [\overline{e/x}^n]e : T' \rightsquigarrow d'$ *with* $\vdash^\flat T' \leq T \rightsquigarrow J^?$ *and* $\Gamma \vdash_{\mathsf{iFJ}} [\overline{\mathsf{wrap}(I_i^?, d_i)/x_i}^{i \in [n]}]d \equiv \mathsf{wrap}(J^?, d') : T$.

Proof. We proceed by induction on n. If $n = 0$ then the claim follows from Lemma C.3.23, Lemma C.3.3, and Theorem 4.11. Suppose the claim already holds for n. Hence, for $\mathscr{M} = \{2, \ldots, n+1\}$

$$\Gamma, x_1 : U_1 \vdash^\flat [\overline{e_i/x_i}^{i \in \mathscr{M}}]e : T'' \rightsquigarrow d'' \tag{C.3.37}$$
$$\vdash^\flat T'' \leq T \rightsquigarrow J'^?$$
$$\Gamma, x_1 : U_1 \vdash_{\mathsf{iFJ}} [\overline{\mathsf{wrap}(I_i^?, d_i)/x_i}^{i \in \mathscr{M}}]d \equiv \mathsf{wrap}(J'^?, d'') : T \tag{C.3.38}$$

We now show the claim for $n+1$. Applying Lemma C.3.34 to (C.3.37) yields

$$\Gamma \vdash^\flat [e_1/x_1]([\overline{e_i/x_i}^{i \in \mathscr{M}}]e) : T' \rightsquigarrow d' \tag{C.3.39}$$
$$\vdash^\flat T' \leq T'' \rightsquigarrow J''^?$$
$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{[\mathsf{wrap}(I_1^?, d_1)/x_1]}_{=:\varphi} d'' \equiv \mathsf{wrap}(J''^?, d') : T''$$

Define $J^? := \mathsf{trans}(J''^?, T, J'^?)$. Then by Lemma C.3.28

$$\vdash^\flat T' \leq T \rightsquigarrow J^? \tag{C.3.40}$$

Thus, by Lemma C.3.31

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J'^?, \varphi d'') \equiv \mathsf{wrap}(J^?, d') : T \tag{C.3.41}$$

From the assumptions, Theorem 4.11, and Lemma C.2.4, we get

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I_1^?, d_1) : U_1''$$
$$\vdash_{\mathsf{iFJ}} U_1'' \leq U_1$$

We now apply Theorem 4.15 (together with Lemma C.3.3) to (C.3.38) and get

$$\Gamma \vdash_{\mathsf{iFJ}} \varphi[\overline{\mathsf{wrap}(I_i^?, d_i)/x_i}^{i \in \mathscr{M}}]d \equiv \varphi\mathsf{wrap}(J'^?, d'') : T \tag{C.3.42}$$

From the assumptions and Lemma C.3.35, we get $x_1 \notin \mathsf{fv}(\overline{e_i}^{i\in\mathscr{M}})$ and $x_1 \notin \mathsf{fv}(\overline{d_i}^{i\in\mathscr{M}})$. Thus

$$[e_1/x_1]([\overline{e_i/x_i}^{i\in\mathscr{M}}]e) = [\overline{e_i/x_i}^{i\in[n+1]}]e$$
$$\varphi([\overline{\mathsf{wrap}(I_i^?, d_i)/x_i}^{i\in\mathscr{M}}]d) = [\overline{\mathsf{wrap}(I_i^?, d_i)/x_i}^{i\in[n+1]}]d$$

Then (C.3.41) and (C.3.42) and Lemma C.3.11 yield

$$\Gamma \vdash_{\mathsf{iFJ}} [\overline{\mathsf{wrap}(I_i^?, d_i)/x_i}^{i\in[n+1]}]d \equiv \mathsf{wrap}(J^?, d') : T$$

The claim now follows with (C.3.39) and (C.3.40). □

Lemma C.3.37. *If $\Gamma \vdash^\flat v : T \leadsto e$ then e is a value.*

Proof. We proceed by induction on the derivation of $\Gamma \vdash^\flat v : T \leadsto e$. We know $v = \mathbf{new}\, N(\overline{v})$, so the derivation ends with rule EXP-NEW$^\flat$. Applying the I.H. yields that all arguments v_i translate to iFJ values w_i, so the resulting iFJ expression e is a value $\mathsf{\mathbf{new}}\, N(\overline{w})$. □

Lemma C.3.38. *Assume that m is a class method. If $\mathsf{mtype}^\flat(m, N) = msig \leadsto \mathsf{nil}$ and moreover $\mathsf{getmdef}^\flat(m, N) = mdef$ then $mdef = msig\,\{e\}$ and $\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = msig$ and $\mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = msig\,\{d\}$ such that $this : N, \overline{x : T} \vdash^\flat e : T' \leadsto d'$ and $\vdash^\flat T' \leq T \leadsto I^?$ and $d = \mathsf{wrap}(I^?, d')$.*

Proof. The claim "$mdef = msig\,\{e\}$" is obvious by the definitions of $\mathsf{mtype}^\flat$ and $\mathsf{getmdef}^\flat$. The claim "$\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = msig$" follows by Lemma C.2.6. The claim "$\mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = msig\,\{d\}$" holds by definition of $\mathsf{getmdef}_{\mathsf{iFJ}}$. The rest of the lemma holds by the premises of the rules OK-CDEF$^\flat$, OK-MDEF-IN-CLASS$^\flat$, and OK-MDEF$^\flat$. □

Lemma C.3.39. *If $\mathsf{mtype}^\flat(m, N) = msig \leadsto I$ and $\mathsf{getmdef}^\flat(m, N) = mdef$ then*

$$\mathbf{interface}\ I\ \mathbf{extends}\ \overline{J}\,\{\,\overline{m : msig}\,\}$$

such that $m = m_i$ for some i and $msig = msig_i$ and

$$\mathsf{least\text{-}impl}^\flat\{\mathbf{implementation}\ I\ [\,M\,]\ \ldots \mid N \trianglelefteq^\flat_{\mathsf{c}} M\}$$
$$= \mathbf{implementation}\ I\ [\,M\,]\,\{\,\overline{m : mdef}\,\}$$

for some M such that $N \trianglelefteq^\flat_{\mathsf{c}} M$ and $mdef = mdef_i = msig\,\{e\}$ for some e.

Proof. The derivation of $\mathsf{mtype}^\flat(m, N) = msig \leadsto I$ must end with rule MTYPE-IFACE$^\flat$. Inverting the rule and using Lemma C.2.3 show that m is defined in interface I. By Convention 4.1 we know that m cannot be defined in a class. Hence, the derivation of $\mathsf{getmdef}^\flat(m, N) = mdef$ ends with rule DYN-MDEF-IFACE$^\flat$. Inverting this rule, together with the premises of rules OK-IMPL$^\flat$ and IMPL-METH$^\flat$, proves the rest of the lemma. □

Lemma C.3.40. *Assume that a CoreGI$^\flat$ interface I defines a method m of arity k. Then it holds that $\mathsf{\mathbf{new}}\ Wrap^I(v).m(\overline{v}^k) \longrightarrow^+_{\mathsf{iFJ}} \mathsf{\mathbf{getdict}}(I, v).m(v, \overline{v}^k)$.*

Proof. Assume that m is defined as $\overline{T\,x}^k \to U$ in interface I. The translation of I in rules OK-IDEF$^\flat$ and WRAPPER-METHODS$^\flat$ then places a method definition

$$mdef = \overline{T\,x}^k \to U\,\{\,\mathsf{\mathbf{getdict}}(I, this.wrapped).m(this.wrapped, \overline{x})\}\,\}$$

in class $Wrap^I$. Thus, $\mathsf{getmdef}_{\mathsf{iFJ}}(m, Wrap^I) = mdef$. The claim now follows by an application of rule DYN-INVOKE-IFJ, followed by two applications of rule DYN-FIELD-IFJ. (Class $Wrap^I$ has a single field *wrapped* by well-formedness criterion WF-IFJ-6.) □

Lemma C.3.41. *For all class types N and M, it holds that $N \trianglelefteq^{\flat}_{\mathsf{c}} M$ if, and only if, $\vdash_{\mathsf{iFJ}} N \leq M$.*

Proof. If $N \trianglelefteq^{\flat}_{\mathsf{c}} M$ then $\vdash^{\flat'} N \leq M$ by rule $\textsc{sub-class}^{\flat}$, so $\vdash_{\mathsf{iFJ}} N \leq M$ by Lemma C.2.1. On the other hand, if $\vdash_{\mathsf{iFJ}} N \leq M$ then $\vdash_{\mathsf{iFJ\text{-}a}} N \leq M$ by Lemma C.1.3. Then $N \trianglelefteq^{\flat}_{\mathsf{c}} M$ by induction on the derivation of $\vdash_{\mathsf{iFJ\text{-}a}} N \leq M$. □

Lemma C.3.42. *If $N \trianglelefteq^{\flat}_{\mathsf{c}} M$ and*

$$\begin{aligned}\mathsf{least\text{-}impl}^{\flat}\{\textbf{implementation}\ I\ [\,M\,]\ \ldots \mid N \trianglelefteq^{\flat}_{\mathsf{c}} M\}&\\ = \textbf{implementation}\ I\ [\,M\,]\,\{\,\overline{m : \mathit{mdef}}\,\}&\end{aligned}$$

then $\textbf{getdict}(I, \textbf{new}\ N(\overline{v})) \longmapsto_{\mathsf{iFJ}} \textbf{new}\ \mathit{Dict}^{I,M}()$.

Proof. By using Lemma C.3.41, by examining the translation of implementations (rule $\textsc{ok-impl}^{\flat}$), and by the definitions of $\mathsf{least\text{-}impl}^{\flat}$ and $\mathsf{mindict}_{\mathsf{iFJ}}$, it is easy to see that

$$\mathsf{mindict}_{\mathsf{iFJ}}\{\textbf{class}\ \mathit{Dict}^{I,M}\ \ldots \mid \vdash_{\mathsf{iFJ}} N \leq M\} = \textbf{class}\ \mathit{Dict}^{I,M}\ \ldots$$

Obviously, the class type N denotes a $\mathsf{CoreGI}^{\flat}$ class, so N is not a wrapper (Convention 4.4). Thus, $\mathsf{unwrap}(\textbf{new}\ N(\overline{v})) = \textbf{new}\ N(\overline{v})$, so the claim follows with rule $\textsc{dyn-getdict-ifj}$. □

Lemma C.3.43. *Suppose that the underlying iFJ program is in the image of the translation from CoreGI$^{\flat}$. If $\vdash^{\flat} N \leq I \rightsquigarrow I$ then there exists an iFJ class of the form* $\textbf{class}\ \mathit{Dict}^{I,M}\ \ldots$ *with $\vdash_{\mathsf{iFJ}} N \leq M$.*

Proof. The derivation of $\vdash^{\flat} N \leq I \rightsquigarrow I$ must end with rule $\textsc{sub-impl}^{\flat}$, so we have

$$\Vdash^{\flat} N\ \textbf{implements}\ I$$

Thus, there exists M and an $\textbf{implementation}\ I\ [\,M\,]\ \ldots$ such that $\vdash^{\flat'} N \leq M$. We have $\vdash_{\mathsf{iFJ}} N \leq M$ by Lemma C.2.1. The existence of $\textbf{class}\ \mathit{Dict}^{I,M}\ \ldots$ follows from the premise of rule $\textsc{ok-impl}^{\flat}$. □

Lemma C.3.44. *Suppose that the underlying iFJ program is in the image of the translation from CoreGI$^{\flat}$. If $\vdash_{\mathsf{iFJ}} N \leq I$ then N is a wrapper class.*

Proof. From $\vdash_{\mathsf{iFJ}} N \leq I$ we get by Lemma C.1.3 that $\vdash_{\mathsf{iFJ\text{-}a}} N \leq I$. This derivation must end with rule $\textsc{sub-alg-class-iface-ifj}$. Inverting the rule yields

$$\begin{gathered}\vdash_{\mathsf{iFJ\text{-}a}} N \leq C\\ \textbf{class}\ C\ \textbf{extends}\ M\ \textbf{implements}\ \overline{J}\ \ldots\\ \vdash_{\mathsf{iFJ\text{-}a}} J_i \leq I\end{gathered}$$

Now assume that N is not a wrapper class; that is, N appears in the $\mathsf{CoreGI}^{\flat}$ program of which the underlying iFJ program is the translation of. By examining rule $\textsc{ok-class}^{\flat}$ we see that C must also appear in this $\mathsf{CoreGI}^{\flat}$ program. However, then $\overline{J} = \bullet$ by rule $\textsc{ok-class}^{\flat}$. But this is a contradiction to $\vdash_{\mathsf{iFJ\text{-}a}} J_i \leq I$. Hence, N must be a wrapper class. □

Lemma C.3.45. *If $\emptyset \vdash^{\flat} e_1 : T \rightsquigarrow e_1'$ and $e_1 \longmapsto^{\flat} e_2$, then $e_1' \longrightarrow^{+}_{\mathsf{iFJ}} e_2'$ such that $\emptyset \vdash^{\flat} e_2 : T' \rightsquigarrow e_2''$ and $\vdash^{\flat} T' \leq T \rightsquigarrow I^{?}$ and $\emptyset \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^{?}, e_2'') \equiv e_2' : T$.*

Proof. *Case distinction* on the rule used for the reduction $e_1 \longmapsto^{\flat} e_2$.

- *Case* rule DYN-FIELD$^\flat$: Then

$$\begin{aligned} e_1 &= \mathbf{new}\, N(\overline{v}).f \\ \mathsf{fields}^\flat(N) &= \overline{U\, f} \\ f &= f_i \\ e_2 &= v_i \end{aligned} \tag{C.3.43}$$

The derivation of $\emptyset \vdash^\flat e_1 : T \rightsquigarrow e_1'$ must end with rule EXP-FIELD and its subderivation for $\mathbf{new}\, N(\overline{v})$ must end with rule EXP-NEW. Thus, with Lemma C.3.37 and because $\mathsf{fields}^\flat$ is deterministic (by Lemmas C.3.5 and C.2.7), we have

$$\emptyset \vdash^\flat \mathbf{new}\, N(\overline{v}) : N \rightsquigarrow \mathbf{new}\, N(\overline{w}) \tag{C.3.44}$$

$$N = C \text{ for some } C$$

$$\vdash^\flat N\ \mathsf{ok}$$

$$(\forall i)\ \emptyset \vdash^\flat v_i : V_i \rightsquigarrow w_i' \tag{C.3.45}$$

$$(\forall i)\ \vdash^\flat V_i \leq U_i \rightsquigarrow J_i^? \tag{C.3.46}$$

$$(\forall i)\ w_i = \mathsf{wrap}(J_i^?, w_i')$$

$$e_1' = \mathbf{new}\, N(\overline{w}).f$$

$$T = U_i$$

With Lemma C.2.7, we have $\mathsf{fields}_{\mathsf{iFJ}}(N) = \overline{U\, f}$, so by rule DYN-FIELD-IFJ

$$e_1' \longmapsto_{\mathsf{iFJ}} w_i$$

With rule DYN-CONTEXT-IFJ then for $e_2' := w_i$

$$e_1' \longrightarrow_{\mathsf{iFJ}} e_2'$$

Moreover, with (C.3.43), (C.3.45), $T' := V_i$, and $e_2'' := w_i'$

$$\emptyset \vdash^\flat e_2 : T' \rightsquigarrow e_2''$$

With (C.3.46) and $I^? := J_i^?$ we have

$$\vdash^\flat T' \leq T \rightsquigarrow I^?$$

With (C.3.45) and Theorem 4.11 we get $\emptyset \vdash_{\mathsf{iFJ}} w_i' : V_i$. With Lemma C.2.4 and (C.3.46) then $\emptyset \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J_i^?, w_i') : U_i'$ for some U_i' with $\vdash_{\mathsf{iFJ}} U_i' \leq U_i$. Obviously, $\mathsf{wrap}(J_i^?, w_i') = \mathsf{wrap}(I^?, e_2'')$ and $\mathsf{wrap}(J_i^?, w_i') = e_2'$, so with $T = U_i$ and Lemma C.3.3

$$\emptyset \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e_2'') \equiv e_2' : T$$

- *Case* rule DYN-INVOKE$^\flat$: Then

$$e_1 = v.m(\overline{v})$$

$$v = \mathbf{new}\, N(\overline{w}) \tag{C.3.47}$$

$$\mathsf{getmdef}^\flat(m, N) = \overline{T\, x} \to T\, \{e\}$$

$$e_2 = [v/\mathit{this}, \overline{v/x}]e$$

Obviously, the derivation of $\emptyset \vdash^\flat e_1 : T \leadsto e_1'$ must end with rule EXP-INVOKE$^\flat$. Hence, with Lemma C.3.37

$$\emptyset \vdash^\flat v : U \leadsto v' \tag{C.3.48}$$

$$\mathsf{mtype}^\flat(m, U) = \overline{V\,y} \to V \leadsto J^? \tag{C.3.49}$$

$$(\forall i)\ \emptyset \vdash^\flat v_i : U_i \leadsto v_i' \tag{C.3.50}$$

$$(\forall i)\ \vdash^\flat U_i \le V_i \leadsto J_i^? \tag{C.3.51}$$

$$(\forall i)\ v_i'' = \mathsf{wrap}(J_i^?, v_i')$$

$$v'' = \mathsf{wrap}(J^?, v')$$

$$e_1' = v''.m(\overline{v''})$$

With (C.3.47) and (C.3.48) we get by inverting rule EXP-NEW$^\flat$

$$U = N \tag{C.3.52}$$

$$(\forall i)\ \emptyset \vdash^\flat w_i : W_i \leadsto w_i'$$

$$\mathsf{fields}^\flat(N) = \overline{W'\,f}$$

$$(\forall i)\ \vdash^\flat W_i \le W_i' \leadsto J_i'^?$$

$$(\forall i)\ w_i'' = \mathsf{wrap}(J_i'^?, w_i')$$

$$v' = \textbf{new}\ N(\overline{w''})$$

Case distinction on the form of $J^?$.

- *Case* $J^? = \mathsf{nil}$: Assume that m is an interface method. Thus, the derivation of (C.3.49) ends with rule MTYPE-IFACE$^\flat$. Inverting this rule then yields $\vdash^\flat U \le J \leadsto \mathsf{nil}$ for some interface J. With (C.3.52) we then have $\vdash^\flat N \le J \leadsto \mathsf{nil}$, which is a contradiction to Lemma C.2.5. Hence, m is not an interface method but a class method.

  By Lemma C.3.38 we then get

$$\overline{T\,x} \to T = \overline{V\,y} \to V$$

$$\mathsf{mtype}_{\mathsf{iFJ}}(m, N) = \overline{T\,x} \to T$$

$$\mathsf{getmdef}_{\mathsf{iFJ}}(m, N) = \overline{T\,x} \to T\,\{d\}$$

$$\mathit{this} : N, \overline{x : T} \vdash^\flat e : T'' \leadsto d' \tag{C.3.53}$$

$$\vdash^\flat T'' \le T \leadsto J'^?$$

$$d = \mathsf{wrap}(J'^?, d')$$

  By rules DYN-INVOKE-IFJ and DYN-CONTEXT-IFJ we then have

$$\underbrace{v'.m(\overline{v''})}_{=e_1'} \longrightarrow_{\mathsf{iFJ}} \underbrace{\mathsf{wrap}(J'^?, [v'/\mathit{this}, \overline{v''/x}]d')}_{=:e_2'} \tag{C.3.54}$$

  Applying Lemma C.3.36 to (C.3.53), (C.3.50), (C.3.51), and (C.3.48) together with Lemma C.3.23 yield

$$\emptyset \vdash^\flat \overbrace{[v/\mathit{this}, \overline{v/x}]e}^{=e_2} : T' \leadsto e_2'' \tag{C.3.55}$$

$$\vdash^\flat T' \le T'' \leadsto J''^?$$

$$\emptyset \vdash_{\mathsf{iFJ}} [v'/\mathit{this}, \overline{v''/x}]d' \equiv \mathsf{wrap}(J''^?, e_2'') : T''$$

Define $I^? := \mathsf{trans}(J''^?, T, J'^?)$. By Lemma C.3.28 we then have

$$\vdash^\flat T' \leq T \leadsto I^? \tag{C.3.56}$$

Moreover, Lemma C.3.31 yields

$$\emptyset \vdash_{\mathsf{iFJ}} \underbrace{\mathsf{wrap}(J'^?, [v'/\mathit{this}, \overline{v''/x}]d')}_{=e_2'} \equiv \mathsf{wrap}(I^?, e_2'') : T$$

Applying Lemma C.3.4 to this equation and using (C.3.54), (C.3.55), and (C.3.56) then yields the desired result.

- *Case* $J^? = J$: By Lemma C.3.39 we get

$$\begin{gathered}
\textbf{interface } J \textbf{ extends } \overline{J} \,\{\, \overline{m : \mathit{msig}} \,\} \\
m = m_k \\
\mathit{msig}_k = \overline{V\, y} \to V \\
\begin{aligned}&\mathsf{least\text{-}impl}^\flat\{\textbf{implementation } J\ [\,M\,]\ \ldots \mid N \trianglelefteq^\flat_{\mathsf{c}} M\} \\ &\quad = \textbf{implementation } J\ [\,M\,]\,\{\,\overline{m : \mathit{mdef}}\,\}\end{aligned} \\
N \trianglelefteq^\flat_{\mathsf{c}} M \\
\overline{T\, x} \to T\,\{e\} = \mathit{mdef}_k \\
\overline{T\, x} \to T = \overline{V\, y} \to V
\end{gathered}$$

Moreover, we have

$$v'' = \textsf{\textbf{new}}\ \mathit{Wrap}^J(v')$$

By Lemma C.3.40 and Lemma C.3.42 we have

$$\begin{aligned}
e_1' &\longrightarrow^+_{\mathsf{iFJ}} \textsf{\textbf{getdict}}(J, v').m(v', \overline{v''}) \\
&\longrightarrow_{\mathsf{iFJ}} \textsf{\textbf{new}}\ \mathit{Dict}^{J,M}().m(v', \overline{v''}) \qquad\qquad (\text{C.3.57})
\end{aligned}$$

Using rules MTYPE-CLASS-BASE-IFJ, OK-MDEF$^\flat$, IMPL-METH$^\flat$, and OK-IMPL$^\flat$, it is straightforward to verify that

$$\begin{gathered}
\mathsf{getmdef}_{\mathsf{iFJ}}(m, \mathit{Dict}^{J,M}) = \mathit{Object}\, z, \overline{T\, x} \to T\,\{e'\} \\
\mathit{this} : M, \overline{x : T} \vdash^\flat e : T'' \leadsto e'' \qquad\qquad (\text{C.3.58}) \\
\vdash^\flat T'' \leq T \leadsto J'^? \qquad\qquad (\text{C.3.59}) \\
e' = \textsf{\textbf{let}}\ M\ z' = \textsf{\textbf{cast}}(M, z)\ \textsf{\textbf{in}}\ [z'/\mathit{this}]\mathsf{wrap}(J'^?, e'') \\
z, z' \text{ fresh}
\end{gathered}$$

Thus, we have

$$\begin{aligned}
&\textsf{\textbf{new}}\ \mathit{Dict}^{J,M}().m(v', \overline{v''}) \\
\longmapsto_{\mathsf{iFJ}}\ &[\textsf{\textbf{new}}\ \mathit{Dict}^{J,M}()/\mathit{this}, v'/z, \overline{v''/x}]e' && (\text{C.3.60}) \\
=\ &\textsf{\textbf{let}}\ M\ z' = \textsf{\textbf{cast}}(M, v')\ \textsf{\textbf{in}}\ [z'/\mathit{this}, \overline{v''/x}]\mathsf{wrap}(J'^?, e'') && (\text{C.3.61}) \\
\longrightarrow_{\mathsf{iFJ}}\ &\textsf{\textbf{let}}\ M\ z' = v'\ \textsf{\textbf{in}}\ [z'/\mathit{this}, \overline{v''/x}]\mathsf{wrap}(J'^?, e'') && (\text{C.3.62}) \\
\longmapsto_{\mathsf{iFJ}}\ &[v'/\mathit{this}, \overline{v''/x}]\mathsf{wrap}(J'^?, e'') && (\text{C.3.63})
\end{aligned}$$

(Reduction (C.3.60) follows by DYN-INVOKE-IFJ, equation (C.3.61) holds because z, z' are fresh and values like v' are closed, reduction (C.3.62) follows by DYN-CONTEXT-IFJ and DYN-CAST-IFJ, and reduction (C.3.63) follows by DYN-LET-IFJ.)

Together with (C.3.57) we have

$$e'_1 \longrightarrow^{+}_{\mathsf{iFJ}} \underbrace{\overbrace{[v'/\mathit{this}, \overline{v''/x}]}^{=:\varphi} \,\mathsf{wrap}(J'^{?}, e'')}_{=:e'_2} \tag{C.3.64}$$

With (C.3.58), (C.3.48), (C.3.50), (C.3.51), and Lemma C.3.36 we get

$$\emptyset \vdash^{\flat} \overbrace{[v/\mathit{this}, \overline{v/x}]e}^{=e_2} : T' \rightsquigarrow e''_2 \tag{C.3.65}$$

$$\vdash^{\flat} T' \leq T'' \rightsquigarrow J''^{?} \tag{C.3.66}$$

$$\emptyset \vdash_{\mathsf{iFJ}} \varphi e'' \equiv \mathsf{wrap}(J''^{?}, e''_2) : T''$$

By Lemma C.3.28 we get with (C.3.59) and (C.3.66) that

$$\vdash^{\flat} T' \leq T \rightsquigarrow \underbrace{\mathsf{trans}(J''^{?}, T, J'^{?})}_{:=I^{?}}$$

With Lemma C.3.31 then

$$\emptyset \vdash_{\mathsf{iFJ}} \underbrace{\mathsf{wrap}(J^{?}, \varphi e'')}_{=e'_2} \equiv \mathsf{wrap}(I^{?}, e''_2) : T$$

Then (C.3.64), (C.3.65), and Lemma C.3.4 finish this case.

End case distinction on the form of $J^{?}$.

- *Case* rule DYN-CAST$^{\flat}$: Then

$$e_1 = (U)\, v$$

$$e_2 = v = \mathbf{new}\, N(\overline{v}) \tag{C.3.67}$$

$$\vdash^{\flat} N \leq U \rightsquigarrow J^{?} \tag{C.3.68}$$

Obviously, the derivation of $\emptyset \vdash^{\flat} e_1 : T \rightsquigarrow e'_1$ ends with rule EXP-CAST$^{\flat}$. Hence, with Lemma C.3.37, we have

$$U = T \tag{C.3.69}$$

$$\vdash^{\flat} U \ \mathsf{ok}$$

$$\emptyset \vdash^{\flat} v : V \rightsquigarrow w \tag{C.3.70}$$

$$e'_1 = \mathbf{cast}(U, w)$$

With $v = \mathbf{new}\, N(\overline{v})$, we know that the derivation of (C.3.70) ends with an application of rule EXP-NEW$^\flat$. Inverting the rule yields

$$(\forall i)\ \emptyset \vdash^\flat v_i : T_i \rightsquigarrow w'_i$$

$$\vdash^\flat N\ \mathsf{ok}$$

$$\mathsf{fields}^\flat(N) = \overline{U\,f}$$

$$(\forall i)\ \vdash^\flat T_i \leq U_i \rightsquigarrow J^?_i$$

$$(\forall i)\ w_i = \mathsf{wrap}(J^?_i, w'_i)$$

$$V = N \tag{C.3.71}$$

$$w = \mathbf{new}\, N(\overline{w}) \tag{C.3.72}$$

By Convention 4.4, we know that N is not a wrapper class, so

$$\mathsf{unwrap}(w) = w$$

Moreover, we get with Theorem 4.11, Lemma C.2.4, Lemma C.2.7, and rule EXP-NEW-IFJ that

$$\emptyset \vdash_{\mathsf{iFJ}} w : N \tag{C.3.73}$$

Case distinction on whether or not $\vdash_{\mathsf{iFJ}} N \leq U$.

- *Case* $\vdash_{\mathsf{iFJ}} N \leq U$: Then by rule DYN-CAST-IFJ

  $$\underbrace{\mathbf{cast}(U, w)}_{=e'_1} \longmapsto_{\mathsf{iFJ}} \underbrace{w}_{=:e'_2}$$

  With (C.3.67), (C.3.70), (C.3.71), and (C.3.72), we get for $T' := N$ that

  $$\emptyset \vdash^\flat e_2 : T' \rightsquigarrow e'_2$$

  Moreover, we get for $I^? := J^?$ with (C.3.68) and (C.3.69) that

  $$\vdash^\flat T' \leq I \rightsquigarrow I^?$$

  Case distinction on the form of $I^?$.

  * *Case* $I^? = \mathsf{nil}$: Define $e''_2 := w$. Then $\mathsf{wrap}(I^?, e''_2) = w = e'_2$, so by (C.3.73), $\vdash_{\mathsf{iFJ}} N \leq U$, and Lemma C.3.3

    $$\emptyset \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e''_2) \equiv e'_2 : T$$

    as required.
  * *Case* $I^? = J$: Then, by Lemma C.2.3, $U = T = J$. Lemma C.3.44 applied to $\vdash_{\mathsf{iFJ}} N \leq U$ reveals that N is a wrapper class. But this contradicts Convention 4.4.

  End case distinction on the form of $I^?$.
- *Case* not $\vdash_{\mathsf{iFJ}} N \leq U$: With (C.3.68) and Lemma C.2.2 we get

  $$J^? = J$$

  for some J. With (C.3.68), (C.3.69), and Lemma C.2.3 then

  $$U = T = J \tag{C.3.74}$$

With (C.3.68) and Lemma C.3.43 then

$$\begin{array}{c}\textsf{\textbf{class}}\ \mathit{Dict}^{J,M}\ \ldots\\ \vdash_{\mathsf{iFJ}} N \le M\end{array}$$

By rule DYN-CAST-WRAP-IFJ then

$$\underbrace{\textsf{\textbf{cast}}(U, w)}_{=e_1'} \longmapsto_{\mathsf{iFJ}} \underbrace{\textsf{\textbf{new}}\ \mathit{Wrap}^{J}(w)}_{=:e_2'}$$

Define $T' := N$ and $e_2'' := w$. Then, with (C.3.67), (C.3.70), and (C.3.71),

$$\emptyset \vdash^{\flat} e_2 : T' \rightsquigarrow e_2''$$

For $I^? := J$, we get with (C.3.68) that

$$\vdash^{\flat} T' \le T \rightsquigarrow I^?$$

By (C.3.73) and Lemma C.3.3

$$\emptyset \vdash_{\mathsf{iFJ}} w \equiv w : \mathit{Object}$$

With rule EQUIV-NEW-WRAP and (C.3.74) then

$$\emptyset \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e_2'') \equiv e_2' : T$$

End case distinction on whether or not $\vdash_{\mathsf{iFJ}} N \le U$.

End case distinction on the rule used for the reduction $e_1 \longmapsto^{\flat} e_2$. □

Lemma C.3.46. *If $e \longmapsto^{\flat} e''$ then* $\mathsf{fv}(e) = \emptyset$.

Proof. Immediate by inspecting the rules defining the $\longmapsto^{\flat}$ evaluation relation. □

Lemma C.3.47. *If $\Gamma \vdash^{\flat} e : T \rightsquigarrow e'$ and $\mathsf{fv}(e) = \emptyset$ then $\emptyset \vdash^{\flat} e : T \rightsquigarrow e'$.*

Proof. Straightforward induction on the derivation of $\Gamma \vdash^{\flat} e : T \rightsquigarrow e'$. □

Lemma C.3.48 (Weakening for CoreGI$^{\flat}$ typing). *If $\Gamma \vdash^{\flat} e : T \rightsquigarrow e'$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash^{\flat} e : T \rightsquigarrow e'$.*

Proof. Straightforward induction on the derivation of $\Gamma \vdash^{\flat} e : T \rightsquigarrow e'$. □

Proof of Theorem 4.19. From $e_1 \longrightarrow_{\mathsf{iFJ}} e_2$ we get by inverting rule DYN-CONTEXT the existence of an evaluation context $\mathcal{E}$ and expressions d_1, d_2 such that $e_1 = \mathcal{E}[d_1]$ and $e_2 = \mathcal{E}[d_2]$. Thus, it suffices to show the following claim:

> *If $\Gamma \vdash^{\flat} \mathcal{E}[d_1] : T \rightsquigarrow e_1$ and $d_1 \longmapsto^{\flat} d_2$, then $e_1 \longrightarrow^{+}_{\mathsf{iFJ}} e_2$ such that $\Gamma \vdash^{\flat} \mathcal{E}[d_2] : T' \rightsquigarrow e_2'$ and $\vdash^{\flat} T' \le T \rightsquigarrow I^?$ and $\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e_2') \equiv e_2 : T$.*

The proof of this claim is by induction on $\mathcal{E}$.
Case distinction on the form of $\mathcal{E}$.

- *Case* $\mathcal{E} = \square$: In this case, we have with Lemma C.3.46 and Lemma C.3.47 that $\emptyset \vdash^{\flat} \mathcal{E}[d_1] : T \rightsquigarrow e_1$. The claim then follows from Lemma C.3.45, Lemma C.3.48, and Lemma C.3.20.

- *Case* $\mathcal{E} = \mathcal{E}'.f$: Thus, the derivation of $\Gamma \vdash^{\flat} \mathcal{E}[d_1] : T \rightsquigarrow e_1$ ends with rule EXP-FIELD$^{\flat}$, so we have

$$\begin{gathered}
\Gamma \vdash^{\flat} \mathcal{E}'[d_1] : C \rightsquigarrow e'_1 \\
e_1 = e'_1.f \\
\mathsf{fields}^{\flat}(C) = \overline{V\ f} \\
f = f_i \\
T = V_i
\end{gathered} \tag{C.3.75}$$

Applying the I.H. yields

$$\begin{gathered}
e'_1 \longrightarrow_{\mathsf{iFJ}} e''_2 \\
\Gamma \vdash^{\flat} \mathcal{E}'[d_2] : U \rightsquigarrow e'''_2 \\
\vdash^{\flat} U \leq C \rightsquigarrow J^{?} \\
\Gamma \vdash^{\flat} \mathsf{wrap}(J^{?}, e'''_2) \equiv e''_2 : C
\end{gathered} \tag{C.3.76}$$

By Lemma C.3.25 $J^{?} = \mathsf{nil}$ and $U = M$ for some M.
We get by Lemma C.3.19

$$\underbrace{e'_1.f}_{=e_1} \longrightarrow^{+}_{\mathsf{iFJ}} \underbrace{e''_2.f}_{=e_2}$$

By Lemma C.3.30

$$\mathsf{fields}^{\flat}(U) = \overline{V\ f}, \overline{V'\ f'}$$

Thus, by EXP-FIELD$^{\flat}$

$$\Gamma \vdash^{\flat} \underbrace{\mathcal{E}'[d_2].f}_{=\mathcal{E}[d_2]} : T \rightsquigarrow \underbrace{e'''_2.f}_{=:e'_2}$$

We get for $T' := T$ and $I^{?} := \mathsf{nil}$ by Lemma C.3.23 that

$$\vdash^{\flat} T' \leq T \rightsquigarrow I^{?}$$

With (C.3.75), Lemma C.2.7, and Lemma C.3.1 we get the existence of C' such that

$$\begin{gathered}
\vdash_{\mathsf{iFJ}} C \leq C' \\
\mathsf{defines\text{-}field}(C', f_i) \\
\mathsf{fields}_{\mathsf{iFJ}}(C') = \overline{U\ f}^{n} \\
n \geq i
\end{gathered}$$

With $J^{?} = \mathsf{nil}$, (C.3.76), and Lemma C.3.13 we get

$$\Gamma \vdash_{\mathsf{iFJ}} e'''_2 \equiv e''_2 : C'$$

Thus, we get by rule EQUIV-FIELD

$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{e'_2}_{=e'''_2.f=\mathsf{wrap}(I^{?},e'_2)} \equiv \underbrace{e_2}_{=e''_2.f} : T$$

- *Case* $\mathcal{E} = \mathcal{E}'.m(\overline{d'})$: Thus, the derivation of $\Gamma \vdash^\flat \mathcal{E}[d_1] : T \rightsquigarrow e_1$ ends with rule EXP-INVOKE$^\flat$, so we have

$$\Gamma \vdash^\flat \mathcal{E}'[d_1] : U \rightsquigarrow e_0$$

$$\mathsf{mtype}^\flat(m, U) = \overline{V\,x} \to T \rightsquigarrow J^? \tag{C.3.77}$$

$$(\forall i)\ \Gamma \vdash^\flat d'_i : V'_i \rightsquigarrow d''_i \tag{C.3.78}$$

$$(\forall i)\ \vdash^\flat V'_i \leq V_i \rightsquigarrow I^?_i \tag{C.3.79}$$

$$(\forall i)\ d'''_i = \mathsf{wrap}(I^?_i, d''_i)$$

$$e_1 = \mathsf{wrap}(J^?, e_0).m(\overline{d'''})$$

Applying the I.H. yields

$$e_0 \longrightarrow^+_{\mathsf{iFJ}} e'_0$$

$$\Gamma \vdash^\flat \mathcal{E}'[d_2] : U' \rightsquigarrow e''_0$$

$$\vdash^\flat U' \leq U \rightsquigarrow J'^? \tag{C.3.80}$$

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J'^?, e''_0) \equiv e'_0 : U \tag{C.3.81}$$

By Lemma C.3.29 we have

$$\mathsf{mtype}^\flat(m, U') = \overline{V\,x} \to T \rightsquigarrow J''^? \tag{C.3.82}$$

$$J''^? = \begin{cases} I & \text{if } J^? = \mathsf{nil} \text{ and } J'^? = J \text{ where } I \text{ such that } J \trianglelefteq^\flat_{\mathsf{i}} I \\ J^? & \text{otherwise} \end{cases}$$

Thus, by rule EXP-INVOKE$^\flat$

$$\Gamma \vdash^\flat \underbrace{\mathcal{E}'[d_2].m(\overline{d'})}_{=\mathcal{E}[d_2]} : \underbrace{T}_{=:T'} \rightsquigarrow \underbrace{\mathsf{wrap}(J''^?, e''_0).m(\overline{d'''})}_{=:e'_2}$$

Moreover, by Lemma C.3.19

$$\underbrace{\mathsf{wrap}(J^?, e_0).m(\overline{d'''})}_{=e_1} \longrightarrow^+_{\mathsf{iFJ}} \underbrace{\mathsf{wrap}(J^?, e'_0).m(\overline{d'''})}_{=e_2}$$

We get by Lemma C.3.23 for $I^? := \mathsf{nil}$ that

$$\vdash^\flat T' \leq T \rightsquigarrow I^?$$

We still need to prove

$$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{\mathsf{wrap}(J''^?, e''_0).m(\overline{d'''})}_{=\mathsf{wrap}(I^?, e'_2)} \equiv \underbrace{\mathsf{wrap}(J^?, e'_0).m(\overline{d'''})}_{=e_2} : T \tag{C.3.83}$$

From (C.3.78), (C.3.79), Theorem 4.11, Lemma C.2.4, Lemma C.3.13, and Lemma C.3.3 we get

$$(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} d'''_i \equiv d'''_i : V_i$$

We next show the following three claims:

(i) $\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J''^?, e''_0) \equiv \mathsf{wrap}(J^?, e'_0) : U''$ *for some* U''

(ii) $\mathsf{topmost}(U'', m)$

(iii) $\mathsf{mtype}_{\mathsf{iFJ}}(m, U'') = \overline{V\,x} \to T$

Then (C.3.83) follows with rule EQUIV-INVOKE.

Case distinction on $J^?$ and $J'^?$.

- *Case* $J^? = \mathsf{nil}$ and $J'^? = J$ for some J: Then $J''^? = I$ for some I such that $J \trianglelefteq^{\flat}_{\mathbf{i}} I$. By (C.3.82), Lemma C.2.8, and the definition of $\mathsf{topmost}$ then

  $$\vdash^{\flat} U' \leq I \leadsto I$$
  $$\mathsf{mtype}_{\mathsf{iFJ}}(m, I) = \overline{V\,x} \to T$$
  $$\mathsf{topmost}(I, m)$$

  Defining $U'' := I$ proves claims (ii) and (iii). Lemma C.2.3, (C.3.80), and $J'^? = J$ imply $U = J$. Thus, (C.3.81), Lemma C.3.4, and Lemma C.3.32 yield

  $$\Gamma \vdash_{\mathsf{iFJ}} \textbf{new}\ \mathit{Wrap}^{I}(e''_0) \equiv e'_0 : I$$

  This proves claim (i).

- *Case* $J^? \neq \mathsf{nil}$ or $J'^? = \mathsf{nil}$: Then $J''^? = J^?$.

  Case distinction on the form of $J'^?$.

  * *Case* $J'^? = \mathsf{nil}$: First, assume $J^? \neq \mathsf{nil}$; that is, $J^? = J$ for some J. From (C.3.77), Lemma C.2.8, and the definition of $\mathsf{topmost}$ then

    $$\vdash^{\flat} U \leq J \leadsto J$$
    $$\mathsf{mtype}_{\mathsf{iFJ}}(m, J) = \overline{V\,x} \to T$$
    $$\mathsf{topmost}(J, m)$$

    Defining $U'' := J$ proves claims (ii) and (iii). From (C.3.81) and Lemma C.3.13 we get

    $$\Gamma \vdash_{\mathsf{iFJ}} e''_0 \equiv e'_0 : \mathit{Object}$$

    Hence, with rule EQUIV-NEW-WRAP

    $$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{\textbf{new}\ \mathit{Wrap}^{J}(e''_0)}_{=\mathsf{wrap}(J''^?, e''_0)} \equiv \underbrace{\textbf{new}\ \mathit{Wrap}^{J}(e'_0)}_{=\mathsf{wrap}(J^?, e'_0)} : \underbrace{J}_{=U''}$$

    which is what we need to prove claim (i).

    Now assume $J^? = \mathsf{nil}$. By Lemma C.3.33 and (C.3.77) we get the existence of U'' such that

    $$\vdash_{\mathsf{iFJ}} U \leq U''$$
    $$\mathsf{mtype}_{\mathsf{iFJ}}(m, U'') = \overline{V\,x} \to T$$
    $$\mathsf{topmost}(U'', m)$$

    This proves claims (ii) and (iii). Claim (i) follows from (C.3.81), $J^? = J'^? = J''^? = \mathsf{nil}$, and Lemma C.3.13

- *Case* $J'^? \neq \mathsf{nil}$: Then $J^? \neq \mathsf{nil}$; that is, $J^? = J$ for some J. From (C.3.77), Lemma C.2.8, and the definition of topmost then

$$\vdash^\flat U \leq J \rightsquigarrow J$$
$$\mathsf{mtype}_{\mathsf{iFJ}}(m, J) = \overline{V\,x} \to T$$
$$\mathsf{topmost}(J, m)$$

Defining $U'' := J$ now proves claims (ii) and (iii). We get from (C.3.81) and Lemma C.3.4 that

$$\Gamma \vdash_{\mathsf{iFJ}} e_0' \equiv \mathsf{wrap}(J'^?, e_0'') : U$$

With (C.3.80), $\vdash^\flat U \leq J \rightsquigarrow J$, and Lemma C.3.31 then

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J, e_0') \equiv \mathsf{wrap}(\underbrace{\mathsf{trans}(J'^?, J, J)}_{=J}, e_0'') : J$$

With $U'' = J = J^? = J''^?$ and Lemma C.3.4 we finally get claim (i).

End case distinction on the form of $J'^?$.

End case distinction on $J^?$ and $J'^?$.

- *Case* $\mathcal{E} = d.m(\overline{v}, \mathcal{E}', \overline{d'})$: W.l.o.g., $\overline{v} = \bullet$. We know that the derivation of $\Gamma \vdash^\flat \mathcal{E}[d_1] : T \rightsquigarrow e_1$ ends with rule EXP-INVOKE$^\flat$, so we have

$$\Gamma \vdash^\flat d : U \rightsquigarrow d' \tag{C.3.84}$$
$$\mathsf{mtype}^\flat(m, U) = V_0\,x_0, \overline{V\,x} \to T \rightsquigarrow J^? \tag{C.3.85}$$
$$\Gamma \vdash^\flat \mathcal{E}'[d_1] : V_0' \rightsquigarrow d_0$$
$$\vdash^\flat V_0' \leq V_0 \rightsquigarrow I_0^?$$
$$d_0' = \mathsf{wrap}(I_0^?, d_0)$$
$$(\forall i)\ \Gamma \vdash^\flat d_i' : V_i' \rightsquigarrow d_i''$$
$$(\forall i)\ \vdash^\flat V_i' \leq V_i \rightsquigarrow I_i^?$$
$$(\forall i)\ d_i''' = \mathsf{wrap}(I_i^?, d_i'')$$
$$e_1 = \mathsf{wrap}(J^?, d).m(d_0', \overline{d'''})$$

Applying the I.H. yields

$$d_0 \longrightarrow^+_{\mathsf{iFJ}} d_0''$$
$$\Gamma \vdash^\flat \mathcal{E}'[d_2] : U \rightsquigarrow d_0'''$$
$$\vdash^\flat U \leq V_0' \rightsquigarrow I_0'^?$$
$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I_0'^?, d_0''') \equiv d_0'' : V_0' \tag{C.3.86}$$

By Lemma C.3.19 we get

$$e_1 \longrightarrow^+_{\mathsf{iFJ}} \underbrace{\mathsf{wrap}(J^?, d).m(\mathsf{wrap}(I_0^?, d_0''), \overline{d'''})}_{=:e_2}$$

By Lemma C.3.28

$$\vdash^\flat U \leq V_0 \rightsquigarrow \mathsf{trans}(I_0'^?, V_0, I_0^?)$$

By rule EXP-INVOKE$^\flat$

$$\Gamma \vdash^\flat \underbrace{\mathcal{E}[d_2]}_{=d.m(\mathcal{E}'[d_2],\overline{d'})} : \underbrace{T}_{=:T'} \rightsquigarrow \underbrace{\mathsf{wrap}(J^?, d).m(\mathsf{wrap}(\mathsf{trans}(I_0'^?, V_0, I_0^?), d_0'''), \overline{d'''})}_{=:e_2'}$$

We get by Lemma C.3.23 for $I^? := \mathsf{nil}$ that

$$\vdash^\flat T' \leq T \rightsquigarrow I^?$$

From (C.3.84), we get by Theorem 4.11 that

$$\Gamma \vdash_{\mathsf{iFJ}} d' : U$$

Case distinction on the form of $J^?$.

- *Case* $J^? = \mathsf{nil}$: Then by Lemma C.3.33 for some U'

  $$\vdash_{\mathsf{iFJ}} U \leq U'$$
  $$\mathsf{mtype}_{\mathsf{iFJ}}(m, U') = V_0\, x_0, \overline{V\, x} \to T$$
  $$\mathsf{topmost}(U', m)$$

  By Lemma C.3.3

  $$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J^?, d') \equiv \mathsf{wrap}(J^?, d') : U'$$

- *Case* $J^? \neq \mathsf{nil}$: Then $J^? = J$ for some J. Hence, by Lemma C.2.8 and the definition of topmost

  $$\vdash^\flat U \leq J \rightsquigarrow J$$
  $$\mathsf{mtype}_{\mathsf{iFJ}}(m, J) = V_0\, x_0, \overline{V\, x} \to T$$
  $$\mathsf{topmost}(J, m)$$

  By Lemma C.2.4

  $$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J^?, d') : U''$$

  for some type U'' with $\vdash_{\mathsf{iFJ}} U'' \leq J$. Thus, by Lemma C.3.3

  $$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J^?, d') \equiv \mathsf{wrap}(J^?, d') : J$$

End case distinction on the form of $J^?$.
In both cases, we have found a type U' such that

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J^?, d') \equiv \mathsf{wrap}(J^?, d') : U'$$
$$\mathsf{mtype}_{\mathsf{iFJ}}(m, U') = V_0\, x_0, \overline{V\, x} \to T$$
$$\mathsf{topmost}(U', m)$$

By Theorem 4.11, Lemma C.3.3, Lemma C.3.13, and Lemma C.2.4

$$(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} d_i''' \equiv d_i''' : V_i$$

We further get by Lemma C.3.4, Lemma C.3.31, and (C.3.86)

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(\mathsf{trans}(I_0'^?, V_0, I_0^?), d_0''') \equiv \mathsf{wrap}(I_0^?, d_0'') : V_0$$

Thus, by rule EQUIV-INVOKE

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e_2') \equiv e_2 : T$$

as required.

- *Case* $\mathcal{E} = \mathbf{new}\, N(\overline{v}, \mathcal{E}', \overline{d'})$: W.l.o.g., $\overline{v} = \bullet$. We know that the derivation of $\Gamma \vdash^\flat \mathcal{E}[d_1] : T \leadsto e_1$ must end with rule EXP-NEW$^\flat$, so we have

$$\begin{gathered}
\mathsf{fields}^\flat(N) = U_0\, f_0, \overline{U\, f} \\
\Gamma \vdash^\flat \mathcal{E}'[d_1] : U_0' \leadsto d_0 \\
\vdash^\flat U_0' \leq U_0 \leadsto I_0^? \\
d_0' = \mathsf{wrap}(I_0^?, d_0) \\
(\forall i)\ \Gamma \vdash^\flat d_i' : U_i' \leadsto d_i'' \\
(\forall i)\ \ \vdash^\flat U_i' \leq U_i \leadsto I_0^? \\
(\forall i)\ d_i''' = \mathsf{wrap}(I_0^?, d_i'') \\
e_1 = \mathbf{new}\, N(d_0', \overline{d'''}) \\
T = N
\end{gathered}$$

Applying the I.H. yields

$$\begin{gathered}
d_0 \longrightarrow^+_{\mathsf{iFJ}} d_0'' \\
\Gamma \vdash^\flat \mathcal{E}'[d_2] : U_0'' \leadsto d_0''' \\
\vdash^\flat U_0'' \leq U_0' \leadsto I_0'^? \\
\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I_0'^?, d_0''') \equiv d_0'' : U_0'
\end{gathered}$$

By Lemma C.3.19 we get

$$e_1 \longrightarrow^+_{\mathsf{iFJ}} \underbrace{\mathbf{new}\, N(\mathsf{wrap}(I_0^?, d_0''), \overline{d'''})}_{=:e_2}$$

By Lemma C.3.28

$$\vdash^\flat U_0'' \leq U_0 \leadsto \mathsf{trans}(I_0'^?, U_0, I_0^?)$$

We then get by rule EXP-NEW$^\flat$

$$\Gamma \vdash^\flat \underbrace{\mathcal{E}[d_2]}_{=\mathbf{new}\, N(\mathcal{E}'[d_2], \overline{d'})} : \underbrace{N}_{=:T'} \leadsto \underbrace{\mathbf{new}\, N(\mathsf{wrap}(\mathsf{trans}(I_0'^?, U_0, I_0^?), d_0'''), \overline{d'''})}_{=:e_2'}$$

We get by Lemma C.3.23 for $I^? := \mathsf{nil}$ that

$$\vdash^\flat T' \leq T \leadsto I^?$$

By Theorem 4.11, Lemma C.3.3, Lemma C.3.13, and Lemma C.2.4

$$(\forall i)\ \Gamma \vdash_{\mathsf{iFJ}} d_i''' \equiv d_i''' : U_i$$

We further get by Lemma C.3.4, Lemma C.3.31, and (C.3.86)

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(\mathsf{trans}(I_0'^?, U_0, I_0^?), d_0''') \equiv \mathsf{wrap}(I_0^?, d_0'') : U_0$$

Finally, by rule EQUIV-NEW-CLASS

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^?, e_2') \equiv e_2 : T$$

- *Case* $\mathcal{E} = (V)\,\mathcal{E}'$: We know that the derivation of $\Gamma \vdash^{\flat} \mathcal{E}[d_1] : T \leadsto e_1$ must end with rule EXP-CAST$^{\flat}$, so we have

  $$\begin{gathered} \Gamma \vdash^{\flat} \mathcal{E}'[d_1] : U \leadsto e_1' \\ e_1 = \mathbf{cast}(V, e_1') \\ T = V \\ \vdash^{\flat} T \ \mathsf{ok} \end{gathered}$$

  Applying the I.H. yields

  $$\begin{gathered} e_1' \longrightarrow^{+}_{\mathsf{iFJ}} e_2'' \\ \Gamma \vdash^{\flat} \mathcal{E}'[d_2] : U' \leadsto e_2''' \\ \vdash^{\flat} U' \leq U \leadsto J^{?} \\ \Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J^{?}, e_2''') \equiv e_2'' : U \qquad \text{(C.3.87)} \end{gathered}$$

  By Lemma C.3.19 we get

  $$e_1 \longrightarrow^{+}_{\mathsf{iFJ}} \underbrace{\mathbf{cast}(T, e_2'')}_{=:e_2}$$

  By rule EXP-CAST$^{\flat}$ we have

  $$\Gamma \vdash^{\flat} \mathcal{E}[d_2] : T \leadsto \underbrace{\mathbf{cast}(T, e_2''')}_{=:e_2'}$$

  Case distinction on the form of $J^{?}$.

  - *Case* $J^{?} = \mathsf{nil}$: Then by (C.3.87) and Lemma C.3.13

    $$\Gamma \vdash_{\mathsf{iFJ}} e_2''' \equiv e_2'' : \mathit{Object}$$

  - *Case* $J^{?} = J$: Then $U = J$ by Lemma C.2.3. From (C.3.87) then, by inverting rule EQUIV-NEW-WRAP,

    $$\begin{gathered} e_2'' = \mathbf{new}\ \mathit{Wrap}^{J'}(\hat{e}) \\ \vdash_{\mathsf{iFJ}} J' \leq J \\ \vdash_{\mathsf{iFJ}} e_2''' \equiv \hat{e} : \mathit{Object} \end{gathered}$$

    By rule EQUIV-NEW-OBJECT-RIGHT then

    $$\Gamma \vdash_{\mathsf{iFJ}} e_2''' \equiv e_2'' : \mathit{Object}$$

  End case distinction on the form of $J^{?}$.
  In both cases, we get for $I^{?} := \mathsf{nil}$ that

  $$\Gamma \vdash_{\mathsf{iFJ}} \underbrace{\mathbf{cast}(T, e_2''')}_{=\mathsf{wrap}(I^{?}, e_2')} \equiv \underbrace{\mathbf{cast}(T, e_2'')}_{=e_2} : T$$

  by rule EQUIV-CAST. Moreover, with $T' := T$ we get by Lemma C.3.23 that

  $$\vdash^{\flat} T' \leq T \leadsto I^{?}$$

End case distinction on the form of $\mathcal{E}$. □

C.3.6 Proof of Theorem 4.20

Theorem 4.20 states that translation and multi-step evaluation commute modulo wrappers.

Proof of Theorem 4.20. By induction on the length n of the evaluation sequence $e_0 \longrightarrow^{\flat *} e_n$.

- $n = 0$. In this case, the claim follows by Lemma C.3.23, Theorem 4.11, and Lemma C.3.3.
- $n > 0$. Then $e_0 \longrightarrow^{\flat} e_1 \longrightarrow^{\flat *} e_n$. The diagram in Figure 4.28 sketches how we complete the proof in this case. We first show that the individual parts of the diagram commute.

  (a) Commutativity of (a) follows from Theorem 4.19:

$$e'_0 \longrightarrow^{+}_{\mathsf{iFJ}} e'_1 \tag{C.3.88}$$

$$\Gamma \vdash^{\flat} e_1 : T'' \rightsquigarrow e''_1 \tag{C.3.89}$$

$$\vdash^{\flat} T'' \leq T \rightsquigarrow J^{?} \tag{C.3.90}$$

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J^{?}, e''_1) \equiv e'_1 : T \tag{C.3.91}$$

  (b) Applying the I.H. to $e_1 \longrightarrow^{\flat *} e_n$ and (C.3.89) yields commutativity of (b):

$$e''_1 \longrightarrow^{*}_{\mathsf{iFJ}} d$$

$$\Gamma \vdash^{\flat} e_n : T' \rightsquigarrow e' \tag{C.3.92}$$

$$\vdash^{\flat} T' \leq T'' \rightsquigarrow J'^{?} \tag{C.3.93}$$

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J'^{?}, e') \equiv d : T'' \tag{C.3.94}$$

  (c) Part (c) of the diagram commutes by (possibly repeated) applications of Lemma C.3.19 to $e''_1 \longrightarrow^{*}_{\mathsf{iFJ}} d$:

$$\mathsf{wrap}(J^{?}, e''_1) \longrightarrow^{*}_{\mathsf{iFJ}} \mathsf{wrap}(J^{?}, d)$$

  (d) Applying Theorem 4.16 to (C.3.91) proves that (d) also commutes:

$$e'_1 \longrightarrow^{*}_{\mathsf{iFJ}} e \tag{C.3.95}$$

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(J^{?}, d) \equiv e : T \tag{C.3.96}$$

  Next, we note that (C.3.88) and (C.3.95) imply

$$e'_0 \longrightarrow^{*}_{\mathsf{iFJ}} e \tag{C.3.97}$$

  Then we define $I^{?} := \mathsf{trans}(J'^{?}, T, J^{?})$. By Lemma C.3.28, (C.3.90), and (C.3.93) then

$$\vdash^{\flat} T' \leq T \rightsquigarrow I^{?} \tag{C.3.98}$$

  Together with (C.3.94), (C.3.90), (C.3.93), Lemma C.3.4, and Lemma C.3.31 we then have

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^{?}, e') \equiv \mathsf{wrap}(J^{?}, d) : T$$

  Finally, using Lemma C.3.11 and (C.3.96) yields

$$\Gamma \vdash_{\mathsf{iFJ}} \mathsf{wrap}(I^{?}, e') \equiv e : T \tag{C.3.99}$$

  The claim now follows from (C.3.97), (C.3.92), (C.3.98), and (C.3.99). □

C.4 Relating CoreGI$^\flat$ and CoreGI

This section presents all details of the proof that CoreGI$^\flat$ is a subset of CoreGI. It implicitly assumes that all syntactic CoreGI entities mentioned are restricted and that the underlying CoreGI program is the image according to $\mathcal{B}_\mathrm{p}$ of the underlying CoreGI$^\flat$ program.

C.4.1 Proof of Theorem 4.24

Theorem 4.24 states that subtyping in CoreGI$^\flat$ and restricted CoreGI is equivalent.

Lemma C.4.1. *If $N \trianglelefteq^\flat_\mathbf{c} N'$ then $\mathcal{B}_\mathrm{t}[\![N]\!] \trianglelefteq_\mathbf{c} \mathcal{B}_\mathrm{t}[\![N']\!]$ and $\Delta \vdash \mathcal{B}_\mathrm{t}[\![N]\!] \leq \mathcal{B}_\mathrm{t}[\![N']\!]$ for any Δ. Furthermore, If $K \trianglelefteq^\flat_\mathbf{i} K'$ then $\mathcal{B}_\mathrm{t}[\![K]\!] \trianglelefteq_\mathbf{i} \mathcal{B}_\mathrm{t}[\![K']\!]$ and $\Delta \vdash \mathcal{B}_\mathrm{t}[\![K]\!] \vdash \mathcal{B}_\mathrm{t}[\![K']\!]$ for any Δ.*

Proof. By rule inductions. □

Lemma C.4.2. *If $\vdash^{\flat'} T \leq U$ then $\Delta \vdash \mathcal{B}_\mathrm{t}[\![T]\!] \leq \mathcal{B}_\mathrm{t}[\![U]\!]$ and $\Delta \vdash_\mathrm{q}{}' \mathcal{B}_\mathrm{t}[\![T]\!] \leq \mathcal{B}_\mathrm{t}[\![U]\!]$ for any Δ.*

Proof. Follows with Lemma C.4.1. □

Lemma C.4.3. *If $N \trianglelefteq_\mathbf{c} N'$ then $\mathcal{B}_\mathrm{t}^{-1}[\![N]\!] \trianglelefteq^\flat_\mathbf{c} \mathcal{B}_\mathrm{t}^{-1}[\![N']\!]$. Moreover, if $I \trianglelefteq_\mathbf{i} I'$ then $\mathcal{B}_\mathrm{t}^{-1}[\![I]\!] \trianglelefteq^\flat_\mathbf{i} \mathcal{B}_\mathrm{t}^{-1}[\![I']\!]$.*

Proof. By rule inductions. □

Lemma C.4.4. *If $\emptyset \vdash_\mathrm{q}{}' T \leq U$ then $\vdash^{\flat'} \mathcal{B}_\mathrm{t}^{-1}[\![T]\!] \leq \mathcal{B}_\mathrm{t}^{-1}[\![U]\!]$.*

Proof. Follows with Lemma C.4.3. □

Proof of Theorem 4.24. The first part follows easily using Lemma C.4.2. For the second part, we have $\emptyset \vdash_\mathrm{q} V \leq W$ with Theorem 3.12. The claim then follows using Lemma C.4.3, Lemma C.4.4, and Lemma B.1.7. □

C.4.2 Proof of Theorem 4.25

Theorem 4.25 states that the dynamic semantics of CoreGI$^\flat$ and restricted CoreGI is equivalent.

Lemma C.4.5. *If $\vdash^\flat N \leq M$ then $N \trianglelefteq^\flat_\mathbf{c} M$.*

Proof. Obviously, the derivation of $\vdash^\flat N \leq M$ ends with rule SUB-KERNEL$^\flat$. Hence, $\vdash^{\flat'} N \leq M$. If this derivation ends with rule SUB-CLASS$^\flat$ then we are done. Otherwise, it ends with rule SUB-OBJECT$^\flat$, so $M = \mathit{Object}$. The claim then holds because every class ultimately inherits from Object. □

Lemma C.4.6 (Equivalence of dynamic method lookup).

(*i*) *If $\mathsf{getmdef}^\flat(m^\mathrm{c}, N) = \mathit{mdef}$ then $\mathsf{getmdef}^\mathrm{c}(m^\mathrm{c}, \mathcal{B}_\mathrm{t}[\![N]\!]) = \mathcal{B}_\mathrm{md}[\![\mathit{mdef}]\!]$.*

(*ii*) *If $\mathsf{getmdef}^\flat(m^\mathrm{i}, N) = \mathit{mdef}$ then $\mathsf{getmdef}^\mathrm{i}(m^\mathrm{i}, \mathcal{B}_\mathrm{t}[\![N]\!], \overline{N}) = \mathcal{B}_\mathrm{md}[\![\mathit{mdef}]\!]$ for any CoreGI types $\overline{N}$.*

(*iii*) *If $\mathsf{getmdef}^\mathrm{c}(m^\mathrm{c}, N) = \mathit{mdef}$ then $\mathsf{getmdef}^\flat(m^\mathrm{c}, \mathcal{B}_\mathrm{t}^{-1}[\![N]\!]) = \mathcal{B}_\mathrm{md}^{-1}[\![\mathit{mdef}]\!]$.*

(*iv*) *If $\mathsf{getmdef}^\mathrm{i}(m^\mathrm{i}, N, \overline{N}) = \mathit{mdef}$ then $\mathsf{getmdef}^\flat(m^\mathrm{i}, \mathcal{B}_\mathrm{t}^{-1}[\![N]\!]) = \mathcal{B}_\mathrm{md}^{-1}[\![\mathit{mdef}]\!]$.*

Proof. Claims (i) and (iii) follow by rule inductions.

Claim (ii) follows by inverting rule DYN-MDEF-IFACE$^\flat$ and Lemma C.4.1.

Claim (iv) follows by inverting rule DYN-MDEF-IFACE and Lemmas 4.24 and C.4.5. □

Lemma C.4.7. *If* $\mathsf{fields}^\flat(N) = \overline{U\ f}$ *then* $\mathsf{fields}(\mathcal{B}_{\mathrm{t}}\llbracket N\rrbracket) = \overline{\mathcal{B}_{\mathrm{t}}\llbracket U_i\rrbracket\ f}$*. Furthermore, if* $\mathsf{fields}(N) = \overline{U\ f}$ *then* $\mathsf{fields}^\flat(\mathcal{B}_{\mathrm{t}}^{-1}\llbracket N\rrbracket) = \overline{\mathcal{B}_{\mathrm{t}}^{-1}\llbracket U_i\rrbracket\ f}$*.*

Proof. By rule inductions. □

Proof of Theorem 4.25. We prove (i) and (ii) by case distinctions on the reduction rules used, relying on Lemma C.4.7, Lemma C.4.6, and Theorem 4.24. Then (iii) and (iv) follow from (i) and (ii). □

C.4.3 Proof of Theorem 4.26

Theorem 4.26 states that expression typing in CoreGI$^\flat$ and restricted CoreGI is equivalent.

Lemma C.4.8 (Equivalence of well-formedness of types).

(i) *If* $\vdash^\flat T\ \mathsf{ok}$ *then* $\Delta \vdash \mathcal{B}_{\mathrm{t}}\llbracket T\rrbracket\ \mathsf{ok}$ *for any* Δ*.*

(ii) *If* $\emptyset \vdash T\ \mathsf{ok}$ *then* $\vdash^\flat \mathcal{B}_{\mathrm{t}}^{-1}\llbracket T\rrbracket\ \mathsf{ok}$*.*

Proof. By case distinctions on the last rules used in the derivations given. □

Lemma C.4.9. *If* $\vdash^\flat T \le I$ *then* $\Delta \vdash_{\mathrm{q}}{}' \mathcal{B}_{\mathrm{t}}\llbracket T\rrbracket \le U$ *and* $\Delta \Vdash^{?}_{\mathrm{a}} U\ \mathbf{implements}\ I{<}\bullet{>} \rightarrowtriangle U\ \mathbf{implements}\ I{<}\bullet{>}$ *for any* Δ *and some* U*.*

Furthermore, $\emptyset \vdash_{\mathrm{q}}{}' T \le U$ *and* $\emptyset \Vdash^{?}_{\mathrm{a}} U\ \mathbf{implements}\ I{<}\bullet{>} \rightarrowtriangle U\ \mathbf{implements}\ I{<}\bullet{>}$ *imply* $\vdash^\flat \mathcal{B}_{\mathrm{t}}^{-1}\llbracket T\rrbracket \le I$

Proof. Assume $\vdash^\flat T \le I$. If the corresponding derivation ends with rule SUB-KERNEL$^\flat$, then $T = J$ and $J \trianglelefteq^\flat_{\mathrm{i}} I$. Define $U := \mathcal{B}_{\mathrm{t}}\llbracket I\rrbracket$. Then $\Delta \vdash_{\mathrm{q}}{}' \mathcal{B}_{\mathrm{t}}\llbracket T\rrbracket \le U$ for any Δ by Lemma C.4.1 and rule SUB-Q-ALG-IFACE. Furthermore, $\Delta \Vdash^{?}_{\mathrm{a}} U\ \mathbf{implements}\ I{<}\bullet{>} \rightarrowtriangle U\ \mathbf{implements}\ I{<}\bullet{>}$ by rule ENT-NIL-ALG-IFACE$_2$. If the derivation of $\vdash^\flat T \le I$ ends with rule SUB-IMPL$^\flat$ then we have $\vdash^{\flat\prime} T \le N$ and **implementation** $I\ [N]\ \ldots$, so defining $U := \mathcal{B}_{\mathrm{t}}\llbracket N\rrbracket$ yields $\Delta \vdash_{\mathrm{q}}{}' \mathcal{B}_{\mathrm{t}}\llbracket T\rrbracket \le U$ for any Δ by Lemma C.4.2 and $\Delta \Vdash^{?}_{\mathrm{a}} U\ \mathbf{implements}\ I{<}\bullet{>} \rightarrowtriangle U\ \mathbf{implements}\ I{<}\bullet{>}$ for any Δ by rule ENT-NIL-ALG-IMPL.

Assume $\emptyset \vdash_{\mathrm{q}}{}' T \le U$ and $\emptyset \Vdash^{?}_{\mathrm{a}} U\ \mathbf{implements}\ I{<}\bullet{>} \rightarrowtriangle U\ \mathbf{implements}\ I{<}\bullet{>}$. If the derivation of the latter ends with ENT-NIL-ALG-IMPL, then we get the existence of **implementation** $I\ [N]\ \ldots$ with $\emptyset \vdash_{\mathrm{q}}{}' U \le N$. Then Lemma B.1.7 and Lemma C.4.4 yield $\vdash^{\flat\prime} \mathcal{B}_{\mathrm{t}}^{-1}\llbracket T\rrbracket \le \mathcal{B}_{\mathrm{t}}^{-1}\llbracket N\rrbracket$, so rule SUB-IMPL$^\flat$ gives us $\vdash^\flat \mathcal{B}_{\mathrm{t}}^{-1}\llbracket T\rrbracket \le I$ as required. If the last rule in the derivation of $\emptyset \Vdash^{?}_{\mathrm{a}} U\ \mathbf{implements}\ I{<}\bullet{>} \rightarrowtriangle U\ \mathbf{implements}\ I{<}\bullet{>}$ is either rule ENT-NIL-ALG-IFACE$_1$ or rule ENT-NIL-ALG-IFACE$_2$ (rule ENT-NIL-ALG-ENV is impossible), then we have $\emptyset \vdash_{\mathrm{q}}{}' U \le I{<}\bullet{>}$, so the claim follows with Lemma B.1.7, Lemma C.4.4, and rule SUB-KERNEL$^\flat$. □

Lemma C.4.10 (Equivalence of method types).

(i) *If* $\mathsf{mtype}^\flat(m, T) = \mathit{msig}$ *then* $\mathsf{a\text{-}mtype}_\Delta(m, \mathcal{B}_{\mathrm{t}}\llbracket T\rrbracket, \overline{T}) = \mathcal{B}_{\mathrm{ms}}\llbracket \mathit{msig}\rrbracket$ *for any* Δ *and any* $\overline{T}$*.*

(ii) *If* $\mathsf{a\text{-}mtype}_\emptyset(m, T, \overline{T}) = \mathit{msig}$ *then* $\mathsf{mtype}^\flat(m, \mathcal{B}_{\mathrm{t}}^{-1}\llbracket T\rrbracket) = \mathcal{B}_{\mathrm{ms}}^{-1}\llbracket \mathit{msig}\rrbracket$*.*

Proof. If m is a class method, then both claims follow by rule inductions. Otherwise, m is an interface method. The first claim then follows by inverting rule MTYPE-IFACE$^\flat$ and using Lemma C.4.9; the second claim follows by inverting rule ALG-MTYPE-IFACE and using Lemma C.4.9. □

Proof of Theorem 4.26. For the first claim, we prove $\Delta; \mathcal{B}_\mathrm{t}\,[\![\Gamma]\!] \vdash_\mathrm{a} \mathcal{B}_\mathrm{e}\,[\![e]\!] : \mathcal{B}_\mathrm{t}\,[\![T]\!]$ for any Δ. This proof is by rule induction, using Lemma C.4.7, Lemma C.4.10, Theorem 4.24, and Lemma C.4.8. Then (i) follows with Theorem 3.35 and Lemma C.4.8.

The second claim first uses Theorem 3.36 to obtain $\emptyset; \Gamma \vdash_\mathrm{a} e : U'$ for some U' with $\emptyset \vdash U' \leq T$. A straightforward rule induction, using Lemma C.4.7, Lemma C.4.10, Theorem 4.24, and Lemma C.4.8, then yields $\mathcal{B}_\mathrm{t}^{-1}[\![\Gamma]\!] \vdash^\flat \mathcal{B}_\mathrm{e}^{-1}[\![e]\!] : \mathcal{B}_\mathrm{t}^{-1}[\![U']\!]$. Define $U := \mathcal{B}_\mathrm{t}^{-1}[\![U']\!]$. Then $\vdash^\flat U \leq \mathcal{B}_\mathrm{t}^{-1}[\![T]\!]$ by Theorem 4.24. □

C.4.4 Proof of Theorem 4.27

Theorem 4.27 states that program typing in CoreGI$^\flat$ and restricted CoreGI is equivalent.

Lemma C.4.11 (Equivalence of well-formedness criteria).

(*i*) *If a CoreGI$^\flat$ program prog fulfills all of CoreGI$^\flat$'s well-formedness criteria, then $\mathcal{B}_\mathrm{p}\,[\![prog]\!]$ fulfills all of CoreGI's well-formedness criteria.*

(*ii*) *If a CoreGI program prog fulfills all of CoreGI's well-formedness criteria, then $\mathcal{B}_\mathrm{p}^{-1}[\![prog]\!]$ fulfills all of CoreGI$^\flat$'s well-formedness criteria.*

Proof. Straightforward. The proof that WF$^\flat$-IMPL-1 implies WF-IMPL-1 is by induction on sup as mentioned in WF-IMPL-1, using Lemma C.4.5 and Lemma C.4.1. The implication from WF-IMPL-1 to WF$^\flat$-IMPL-1 follows by Lemma B.2.8 and Theorem 4.24. □

Lemma C.4.12.

(*i*) *Assume that the underlying CoreGI$^\flat$ program is well-typed and that class C contains a definition of method m with signature msig. If* override-ok$^\flat(m : msig, C)$ *then* override-ok$_\Delta(m : \mathcal{B}_\mathrm{ms}\,[\![msig]\!]\,, \mathcal{B}_\mathrm{t}\,[\![C]\!])$ *for any Δ.*

(*ii*) *If the underlying CoreGI program has invariant return types and* override-ok$_\emptyset(m : msig, N)$ *and $N \neq Object$ then* override-ok$^\flat(m : \mathcal{B}_\mathrm{ms}^{-1}[\![msig]\!], \mathcal{B}_\mathrm{t}^{-1}[\![N]\!])$.

Proof. We prove both claims separately.

(i) Define $N := \mathcal{B}_\mathrm{t}\,[\![C]\!]$. Assume $\Delta \vdash N \leq N'$ and mtype$_\Delta(m, N') = msig'$. We now show $\mathcal{B}_\mathrm{ms}\,[\![msig]\!] = msig'$. Then the claim follows by rule OK-OVERRIDE. With $\Delta \vdash N \leq N'$ we get $\Delta \vdash_\mathrm{q} N \leq N'$ by Theorem 3.12, so obviously $N \trianglelefteq_\mathbf{c} N'$. If $N = N'$ then $\mathcal{B}_\mathrm{ms}\,[\![msig]\!] = msig'$ trivially holds. Assume $N \neq N'$. Then there exists a class D such that

$$\textbf{class } D \textbf{ extends } N'$$
$$N \trianglelefteq_\mathbf{c} D\texttt{<}\bullet\texttt{>}$$

Because the underlying CoreGI$^\flat$ is well-typed, a straightforward induction on the derivation of $N \trianglelefteq_\mathbf{c} D\texttt{<}\bullet\texttt{>}$ shows that

$$\textsf{override-ok}^\flat(m : msig, D) \tag{C.4.1}$$

With mtype$_\Delta(m, N') = msig'$ and the fact that m must be a class method, we get that N' defines m with signature $msig'$. Thus,

$$\textsf{mtype}^\flat(m, \mathcal{B}_\mathrm{t}^{-1}[\![N']\!]) = \mathcal{B}_\mathrm{ms}^{-1}[\![msig']\!] \rightsquigarrow \textsf{nil}$$

With (C.4.1) then

$$msig = \mathcal{B}_\mathrm{ms}^{-1}[\![msig']\!]$$

Theorem 4.22 then yields $\mathcal{B}_\mathrm{ms}\,[\![msig]\!] = msig'$ as required.

(ii) Because $N \neq \mathit{Object}$ we have $N = C\texttt{<}\bullet\texttt{>}$ and

$$\textbf{class } C\texttt{<}\bullet\texttt{>} \textbf{ extends } M \ \ldots$$

Assume $\mathsf{mtype}^\flat(m, \mathcal{B}_\mathrm{t}^{-1}[\![M]\!]) = \mathit{msig}' \rightsquigarrow \mathsf{nil}$. It is easy to verify that this implies the existence of M' such that $\emptyset \vdash M \leq M'$ and $\mathsf{mtype}_\emptyset(m, M') = \mathcal{B}_\mathrm{ms}[\![\mathit{msig}']\!]$. We get from the assumption $\mathsf{override\text{-}ok}_\emptyset(m : \mathit{msig}, N)$, so inverting rule OK-OVERRIDE yields $\mathit{msig} = \mathcal{B}_\mathrm{ms}[\![\mathit{msig}']\!]$ because the underlying CoreGI program has invariant return types. But then $\mathcal{B}_\mathrm{ms}^{-1}[\![\mathit{msig}]\!] = \mathit{msig}'$ by Theorem 4.22, so $\mathsf{override\text{-}ok}^\flat(m : \mathcal{B}_\mathrm{ms}^{-1}[\![\mathit{msig}]\!], C)$ follows via rule OK-OVERRIDE$^\flat$. □

Proof of Theorem 4.27. Easy, using Theorem 4.24, Lemma C.4.8, Theorem 4.26, Lemma C.4.11, and Lemma C.4.12. □

D

Formal Details of Chapter 5

D.1 Interfaces as Implementing Types

This section contains the proofs of Theorem 5.3 (undecidability of subtyping in IIT), Theorem 5.6 (Restriction 5.5 ensures decidability of subtyping in IIT), and Theorem 5.8 (Restriction 5.7 implies Restriction 5.5).

D.1.1 Proof of Theorem 5.3

Theorem 5.3 states the subtyping in IIT is decidable. This section completes the proof sketch for this theorem from Section 5.1.2.

The following lemma proves basic properties of the encoding scheme for words over Σ:

Lemma D.1.1. *Suppose $\eta, \zeta \in \Sigma^*$ and T is a type.*

(i) $[\![\eta]\!] = [\![\zeta]\!]$ *if, and only if,* $\eta = \zeta$.

(ii) $\eta \mathbin{\#} (\zeta \mathbin{\#} T) = \eta\zeta \mathbin{\#} T$.

(iii) $\eta \mathbin{\#} [\![\zeta]\!] = [\![\eta\zeta]\!]$.

Proof. Straightforward. □

The next lemma ensures that the types occurring in a derivation of

$$\vdash_{\mathsf{i}} \mathbb{S}\texttt{<}[\![\eta_i]\!], [\![\zeta_i]\!]\texttt{>} \leq \mathbb{G}$$

are of a certain form. Metavariables $\mathfrak{I}$ and $\mathfrak{J}$ range over (possible empty) sequences of indices, and $\mathfrak{I}\mathfrak{J}$ is the concatenation of $\mathfrak{I}$ and $\mathfrak{J}$. For $\mathfrak{I} = i_1 \dots i_r$, the notation $\eta_{\mathfrak{I}}$ denotes the word $\eta_{i_1} \dots \eta_{i_r}$. We implicitly assume a fixed PCP instance $\mathcal{P} = \{(\eta_1, \zeta_1), \dots, (\eta_n, \zeta_n)\}$ such that the underlying IIT program is the encoding thereof (according to the encoding defined in the proof sketch for Theorem 5.3 from Section 5.1.2).

Lemma D.1.2. *Suppose $\vdash_{\mathsf{i}} T \leq W$. Let U and V be types such that neither $\mathbb{S}$ nor $\mathbb{G}$ occur in U or V. Assume that either $T = \mathbb{S}\texttt{<}U, V\texttt{>}$ or $T = \mathbb{G}$. Then one of the following holds:*

(a) $W = \mathbb{S}\texttt{<}U, V\texttt{>}$*, or*

(b) $W = \mathbb{S}\texttt{<}\eta_{\mathfrak{I}} \mathbin{\#} U, \zeta_{\mathfrak{I}} \mathbin{\#} V\texttt{>}$ *for a non-empty sequence* $\mathfrak{I}$*, or*

(c) $W = \mathbb{G}$*.*

With the additional assumption that $W = \mathbb{G}$*, one of the following holds:*

(a) $T = \mathbb{G}$*, or*

(b) $U = V$*, or* $\eta_{\mathfrak{I}} \mathbin{\#} U = \zeta_{\mathfrak{I}} \mathbin{\#} V$ *for some non-empty sequence* $\mathfrak{I}$*.*

Proof. We prove the first claim by induction on the derivation of $\vdash_{\mathrm{i}} T \leq W$.
Case distinction on the last rule used.

- *Case* rule IIT-REFL: Then $T = W$, so the claim follows trivially.
- *Case* rule IIT-TRANS: Then $\vdash_{\mathrm{i}} T \leq V$ and $\vdash_{\mathrm{i}} V \leq W$ for some V. Applying the I.H. to $\vdash_{\mathrm{i}} T \leq V$ gives us that one of the following holds:

  (a) $V = \mathbb{S}\texttt{<}U, V\texttt{>}$, or

  (b) $V = \mathbb{S}\texttt{<}\eta_{\mathfrak{I}'} \mathbin{\#} U, \zeta\#_{\mathfrak{I}'}V\texttt{>}$ for some non-empty sequence $\mathfrak{I}'$, or

  (c) $V = \mathbb{G}$.

  The claim now follows by applying the I.H. to $\vdash_{\mathrm{i}} V \leq W$, possibly using Lemma D.1.1(ii).
- *Case* rule IIT-IMPL: Then

$$\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{U}\texttt{>}\ [J\texttt{<}\overline{T}\texttt{>}]$$
$$T = [\overline{V/X}]J\texttt{<}\overline{T}\texttt{>}$$
$$W = [\overline{V/X}]I\texttt{<}\overline{U}\texttt{>}$$

  There are two possibilities:

  - The implementation is defined by (5.1) on page 113:

$$\overline{X} = X, Y$$
$$I\texttt{<}\overline{U}\texttt{>} = \mathbb{S}\texttt{<}\eta_i \mathbin{\#} X, \zeta_i \mathbin{\#} Y\texttt{>}$$
$$J\texttt{<}\overline{T}\texttt{>} = \mathbb{S}\texttt{<}X, Y\texttt{>}$$

    Hence, T is of the form $\mathbb{S}\texttt{<}U, V\texttt{>}$, so $[\overline{V/X}] = [U/X, V/Y]$. Thus, $W = \mathbb{S}\texttt{<}\eta_i \mathbin{\#} U, \zeta_i \mathbin{\#} V\texttt{>}$.
  - The implementation is defined by (5.2) on page 113. In this case, $W = \mathbb{G}$.

End case distinction on the last rule used.

The proof of the second claim is also by induction on the derivation of $\vdash_{\mathrm{i}} T \leq W$.
Case distinction on the last rule used.

- *Case* rule IIT-REFL: Trivial.
- *Case* rule IIT-TRANS: Then $\vdash_{\mathrm{i}} T \leq V$ and $\vdash_{\mathrm{i}} V \leq W$ for some V. We now apply the first part of this lemma to $\vdash_{\mathrm{i}} T \leq V$ and get that for V either (a), (b), or (c) from case IIT-TRANS in the proof of the first part holds. We now can apply the I.H. for the current part of the proof to $\vdash_{\mathrm{i}} V \leq W$ and get that one of the following holds:

  (a) $V = \mathbb{G}$. Then the claim follows by applying the I.H. to $\vdash_{\mathrm{i}} T \leq V$.

  (b) Either $U = V$ or, with Lemma D.1.1(ii), $\eta_{\mathfrak{I}} \# U = \zeta_{\mathfrak{I}} \# V$ for some non-empty sequence $\mathfrak{I}$. But this is exactly what we need to prove.

- *Case* rule IIT-IMPL: Then

$$\begin{gathered}\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{U}\texttt{>}\,[J\texttt{<}\overline{T}\texttt{>}]\\ T = [\overline{V/X}]J\texttt{<}\overline{T}\texttt{>}\\ W = [\overline{V/X}]I\texttt{<}\overline{U}\texttt{>}\end{gathered}$$

  Because $W = \mathbb{G}$ we know that the implementation definition defined by (5.2) on page 113 must have been used. Thus

$$\begin{gathered}\overline{X} = X\\ J\texttt{<}\overline{T}\texttt{>} = \mathbb{S}\texttt{<}X, X\texttt{>}\end{gathered}$$

  But then $U = V$ as required.

End case distinction on the last rule used. □

Proof of Theorem 5.3. To complete the proof sketch for Theorem 5.3 from Section 5.1.2, we still need to verify to following claim:

> *The PCP instance* $\mathcal{P} = \{(\eta_1, \zeta_1), \ldots, (\eta_n, \zeta_n)\}$ *has a solution if and only if there exists* $i \in \{1, \ldots, n\}$ *such that* $\vdash_{\mathrm{i}} \mathbb{S}\texttt{<}[\![\eta_i]\!], [\![\zeta_i]\!]\texttt{>} \leq \mathbb{G}$ *is derivable.*

We prove the two implications separately.

“⇒”: We first show for any non-empty sequence of indices $i_1 \ldots i_k$ that

$$\vdash_{\mathrm{i}} \mathbb{S}\texttt{<}[\![\eta_{i_k}]\!], [\![\zeta_{i_k}]\!]\texttt{>} \leq \mathbb{S}\texttt{<}[\![\eta_{i_1} \ldots \eta_{i_k}]\!], [\![\zeta_{i_1} \ldots \zeta_{i_k}]\!]\texttt{>} \tag{D.1.1}$$

The proof is by induction on k. The base case ($k = 1$) follows from reflexivity of subtyping. For the inductive step, the induction hypothesis yields

$$\vdash_{\mathrm{i}} \mathbb{S}\texttt{<}[\![\eta_{i_{k+1}}]\!], [\![\zeta_{i_{k+1}}]\!]\texttt{>} \leq T \tag{D.1.2}$$

where $T = \mathbb{S}\texttt{<}[\![\eta_{i_2} \ldots \eta_{i_{k+1}}]\!], [\![\zeta_{i_2} \ldots \zeta_{i_{k+1}}]\!]\texttt{>}$. Choosing a suitable implementation definition from (5.1) on page 113, we get with Lemma D.1.1(iii) and rule IIT-IMPL that

$$\vdash_{\mathrm{i}} T \leq \mathbb{S}\texttt{<}[\![\eta_{i_1} \ldots \eta_{i_{k+1}}]\!], [\![\zeta_{i_1} \ldots \zeta_{i_{k+1}}]\!]\texttt{>}$$

Claim (D.1.1) now follows with (D.1.2) and transitivity of subtyping.

Now suppose that $\mathfrak{I} = i_1 \ldots i_r$ is a solution to $\mathcal{P}$. Then we have from (D.1.1)

$$\vdash_{\mathrm{i}} \mathbb{S}\texttt{<}[\![\eta_{i_r}]\!], [\![\zeta_{i_r}]\!]\texttt{>} \leq \mathbb{S}\texttt{<}[\![\eta_{\mathfrak{I}}]\!], [\![\zeta_{\mathfrak{I}}]\!]\texttt{>}$$

Because $\eta_{\mathfrak{I}} = \zeta_{\mathfrak{I}}$ we get $[\![\eta_{\mathfrak{I}}]\!] = [\![\zeta_{\mathfrak{I}}]\!]$ by Lemma D.1.1(i), so implementation definition (5.2) on page 113 yields together with rule IIT-IMPL and transitivity of subtyping

$$\vdash_{\mathrm{i}} \mathbb{S}\texttt{<}[\![\eta_{i_r}]\!], [\![\zeta_{i_r}]\!]\texttt{>} \leq \mathbb{G}$$

as required.

“⇐”: Given that $\vdash_{\mathrm{i}} \mathbb{S}\texttt{<}[\![\eta_i]\!], [\![\zeta_i]\!]\texttt{>} \leq \mathbb{G}$ is derivable for some $i \in \{1, \ldots, n\}$, we get from Lemma D.1.2 that either $[\![\eta_i]\!] = [\![\zeta_i]\!]$ or that there exists a non-empty sequence $\mathfrak{I}$ such that $\eta_{\mathfrak{I}} \mathbin{\#} [\![\eta_i]\!] = \zeta_{\mathfrak{I}} \mathbin{\#} [\![\zeta_i]\!]$. For the first case, we have $\eta_i = \zeta_i$ by Lemma D.1.1(i); for the second case, we get $[\![\eta_{\mathfrak{I}}\eta_i]\!] = [\![\zeta_{\mathfrak{I}}\zeta_i]\!]$ by Lemma D.1.1(iii), and $\eta_{\mathfrak{I}}\eta_i = \zeta_{\mathfrak{I}}\zeta_i$ by Lemma D.1.1(i). Hence, $\mathcal{P}$ has a solution. □

Figure D.1 Subtyping for IIT without transitivity rule.

$\boxed{\vdash_{\mathrm{i}}' T \leq U}$

IIT-REFL'

$$\vdash_{\mathrm{i}}' T \leq T$$

IIT-IMPL'

$$\frac{[\overline{V/X}]J<\overline{U}> \neq T \qquad \textbf{implementation}<\overline{X}>\ I<\overline{T}>\ [J<\overline{U}>] \qquad \vdash_{\mathrm{i}}' [\overline{V/X}]I<\overline{T}> \leq T}{\vdash_{\mathrm{i}}' [\overline{V/X}]J<\overline{U}> \leq T}$$

D.1.2 Proof of Theorem 5.6

Theorem 5.6 states that subtyping in IIT is decidable under Restriction 5.5. Figure D.1 defines the relation $\vdash_{\mathrm{i}}' T \leq U$, a variant of the subtyping relation of IIT without a built-in transitivity rule. We first verify that $\vdash_{\mathrm{i}} T \leq U$ and $\vdash_{\mathrm{i}}' T \leq U$ are equivalent.

Lemma D.1.3. *If* $\vdash_{\mathrm{i}}' T \leq U$ *then* $\vdash_{\mathrm{i}} T \leq U$*.*

Proof. Straightforward induction on the derivation of $\vdash_{\mathrm{i}}' T \leq U$. □

Lemma D.1.4. *If* $\vdash_{\mathrm{i}}' T \leq U$ *and* $\vdash_{\mathrm{i}}' U \leq V$ *then* $\vdash_{\mathrm{i}}' T \leq V$

Proof. Follows by induction on the derivation of $\vdash_{\mathrm{i}}' T \leq U$. □

Lemma D.1.5. *If* $\vdash_{\mathrm{i}} T \leq U$ *then* $\vdash_{\mathrm{i}}' T \leq U$*.*

Proof. Follows by case distinction on the last rule in the derivation of $\vdash_{\mathrm{i}} T \leq U$, making use of Lemma D.1.4 if this rule is IIT-TRANS. □

Next, we check that $\Vdash_{\mathrm{ia}} T \leq U$ and $\vdash_{\mathrm{i}}' T \leq U$ are equivalent.

Lemma D.1.6. *If* $\Vdash_{\mathrm{ia}} T \leq U$ *then* $\vdash_{\mathrm{i}}' T \leq U$*.*

Proof. A straightforward rule induction shows that $\mathscr{G} \Vdash_{\mathrm{ia}} T \leq U$ implies $\vdash_{\mathrm{i}}' T \leq U$ for any $\mathscr{G}$. Inverting $\Vdash_{\mathrm{ia}} T \leq U$ yields $\{T\} \Vdash_{\mathrm{ia}} T \leq U$, so the claim holds. □

Lemma D.1.7. *If* $\vdash_{\mathrm{i}}' T \leq U$ *then* $\Vdash_{\mathrm{ia}} T \leq U$*.*

Proof. Let $\mathcal{D}_1$ be the derivation of $\vdash_{\mathrm{i}}' T \leq U$ and let $\mathcal{D}_2$ be the immediate subderivation of $\mathcal{D}_1$, let $\mathcal{D}_3$ be the immediate subderivation of $\mathcal{D}_2$, and so on. It is easy to verify that all $\mathcal{D}_i$ have the form $\vdash_{\mathrm{i}}' T_i \leq U$ for types $T = T_1, \ldots, T_n$. We may safely assume that all types $T_1, \ldots, T_n$ are pairwise disjoint. (Otherwise, there are two derivations with identical conclusions, so we simply replace the larger derivation with the smaller one.) With these considerations in place, a straightforward induction shows that $\vdash_{\mathrm{i}}' T \leq U$ implies $\{T\} \Vdash_{\mathrm{ia}} T \leq U$. Thus, we get $\Vdash_{\mathrm{ia}} T \leq U$ by rule IIT-ALG-SUB. □

Proof of Theorem 5.6. With Lemmas D.1.3, D.1.5, D.1.6, and D.1.7, it follows that $\vdash T \leq U$ and $\vdash_{\mathrm{ia}} T \leq U$ are equivalent. Thus, we only need to verify that the algorithm induced by $\vdash_{\mathrm{ia}} T \leq U$ terminates. Suppose that $\mathscr{G} \vdash_{\mathrm{ia}} T' \leq U'$ is a subderivation in an attempt to prove the original goal $\vdash_{\mathrm{ia}} T \leq U$. A straightforward induction on the number of rule applications needed to reach the subderivation shows that $\mathscr{G} \subseteq \mathscr{S}_T$. Thus, $|\mathscr{S}_T| - |\mathscr{G}| \in \mathbb{N}$. Furthermore, rule IIT-ALG-IMPL ensures that the measure $|\mathscr{S}_T| - |\mathscr{G}|$ decreases when moving from the conclusion to the premise. Hence, the algorithm induced by $\vdash_{\mathrm{ia}} T \leq U$ terminates. □

D.1.3 Proof of Theorem 5.8

Theorem 5.8 states that Restriction 5.7 implies Restriction 5.5.

Assume that $\mathit{def}_1, \ldots, \mathit{def}_n$ are the implementation definitions of the underlying IIT program. Define a graph $G = (\mathscr{V}, \mathscr{E})$ such that

$$\begin{aligned} \mathscr{V} &= \{\mathit{def}_1, \ldots, \mathit{def}_n\} \\ \mathscr{E} &= \{(\mathit{def}, \mathit{def}') \in \mathscr{V} \times \mathscr{V} \mid \text{if } \mathit{def} = \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ J\texttt{<}\overline{U}\texttt{>}\,[I\texttt{<}\overline{T}\texttt{>}] \\ &\qquad \text{then } \mathit{def}' = \textbf{implementation}\texttt{<}\overline{Y}\texttt{>}\ I'\texttt{<}\overline{W}\texttt{>}\,[J\texttt{<}\overline{V}\texttt{>}]\} \end{aligned}$$

G is acyclic because Restriction 5.7 holds. Thus, there exists an upper bound $L \in \mathbb{N}$ on the length of any path in G.

In the following, write $T \xrightarrow{\mathit{def}} U$ if, and only if,

$$\mathit{def} = \textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\,[J\texttt{<}\overline{U}\texttt{>}]$$

and there exists a substitution $[\overline{V/X}]$ with $[\overline{V/X}]J\texttt{<}\overline{U}\texttt{>} = T$ and $[\overline{V/X}]I\texttt{<}\overline{T}\texttt{>} = U$. It is straightforward to verify that $U \in \mathscr{S}_T$ if, and only if, there exists a path $\mathit{def}_1, \ldots, \mathit{def}_m$ in G such that $T \xrightarrow{\mathit{def}_1} \ldots \xrightarrow{\mathit{def}_m} U$.

Define the *size* of types and implementation definitions as follows:

$$\begin{aligned} \mathsf{size}(X) &= 1 \\ \mathsf{size}(I\texttt{<}\overline{T}^{k}\texttt{>}) &= 1 + \sum_{i=1}^{k} \mathsf{size}(T_i) \\ \mathsf{size}(\textbf{implementation}\texttt{<}\overline{X}\texttt{>}\ J\texttt{<}\overline{U}\texttt{>}\,[I\texttt{<}\overline{T}\texttt{>}]) &= \mathsf{size}(J\texttt{<}\overline{U}\texttt{>}) \end{aligned}$$

Then $T \xrightarrow{\mathit{def}} U$ implies $\mathsf{size}(U) \leq \mathsf{size}(\mathit{def}) \cdot \mathsf{size}(T) + \mathsf{size}(\mathit{def})$. If now $\delta \in \mathbb{N}$ is an upper bound on the size of all implementation definitions of the underlying program, then $T \xrightarrow{\mathit{def}_1} \ldots \xrightarrow{\mathit{def}_m} U$ implies that $\mathsf{size}(U) \leq \delta^m \cdot \mathsf{size}(T) + \sum_{i=1}^{m} \delta^i$. Thus, $U \in \mathscr{S}_T$ implies $\mathsf{size}(U) \leq \delta^L \cdot \mathsf{size}(T) + \sum_{i=1}^{L} \delta^i$, so the set $\mathscr{S}_T$ is finite because there exist only finitely many types with a bounded size. □

D.2 Bounded Existential Types with Lower and Upper Bounds

This section contains the proofs of Theorem 5.17 (undecidability of subtyping in EXuplo), Theorem 5.19 (decidability of subtyping in EXuplo without lower bounds), and Theorem 5.21 (decidability of subtyping in EXuplo without upper bounds and with only variable-bounded existentials).

D.2.1 Proof of Theorem 5.17

Theorem 5.17 states that subtyping in EXuplo is undecidable. We first show that $\Delta \vdash_{\mathrm{ex}} T \leq U$ if, and only if, $\Delta \vdash_{\mathrm{ex}}{}' T \leq U$.

Lemma D.2.1. *For all types T, $\Delta \vdash_{\text{ex}}' T \leq T$.*

Proof. The only interesting case is $T = \exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,N$. Then we have

$$\textsc{exuplo-open'}\ \dfrac{\textsc{exuplo-abstract'}\ \dfrac{N = N \qquad (\forall i)\ \Delta, \overline{P} \Vdash_{\text{ex}}' P_i}{\Delta, \overline{P} \vdash_{\text{ex}}' N \leq \exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,N} \qquad \overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset}{\Delta \vdash_{\text{ex}}' \exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,N \leq \exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,N}$$

It is easy to verify that $\Delta \Vdash' P$ for any $P \in \Delta$. □

Definition D.2.2. The *size* of an EXuplo type or constraint is defined as follows:

$$\begin{aligned}
\mathsf{size}(X) &= 1\\
\mathsf{size}(C\texttt{<}\overline{T}\texttt{>}) &= 1 + \mathsf{size}(\overline{T})\\
\mathsf{size}(\mathit{Object}) &= 1\\
\mathsf{size}(\exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,N) &= 1 + \mathsf{size}(\overline{P}) + \mathsf{size}(N)\\
\mathsf{size}(X\,\textbf{extends}\,T) &= \mathsf{size}(T)\\
\mathsf{size}(X\,\textbf{super}\,T) &= \mathsf{size}(T)
\end{aligned}$$

The notation $\mathsf{size}(\overline{\xi})$ abbreviates $\sum_i \mathsf{size}(\xi_i)$.

Lemma D.2.3. *If $\Delta \vdash_{\text{ex}}' T \leq U$ and $\Delta \vdash_{\text{ex}}' U \leq V$ then $\Delta \vdash_{\text{ex}}' T \leq V$.*

Proof. The proof makes essential use of the fact that type variables do not have both lower and upper bounds and that only type variables may occur as type arguments of generic classes. Define the *domain* of a type environment Δ as $\mathsf{dom}(\Delta) = \{X \mid X\,\textbf{extends}\,T \in \Delta \text{ or } X\,\textbf{super}\,T \in \Delta\}$, and the *range* of a type environment Δ as $\mathsf{rng}(\Delta) = \{T \mid X\,\textbf{extends}\,T \in \Delta \text{ or } X\,\textbf{super}\,T \in \Delta\}$.

We strengthen the claim as follows:

> *Let $n \in \mathbb{N}$.*
>
> *(i) Assume $\mathsf{size}(U) = n$. If $\Delta \vdash_{\text{ex}}' T \leq U$ and $\Delta \vdash_{\text{ex}}' U \leq V$, then $\Delta \vdash_{\text{ex}}' T \leq V$.*
>
> *(ii) Assume $\mathsf{size}(\overline{P}) = n$. If $\Delta', \overline{P} \vdash_{\text{ex}}' W_1 \leq W_2$ and $[\overline{Y/X}]\Delta' \Vdash_{\text{ex}}' [\overline{Y/X}]P$ for all $P \in \overline{P}$ and $\overline{X} \cap \mathsf{dom}(\Delta') = \emptyset$, then $[\overline{Y/X}]\Delta' \vdash_{\text{ex}}' [\overline{Y/X}]W_1 \leq [\overline{Y/X}]W_2$.*

We now prove that claims (i) and (ii) hold for all $n \in \mathbb{N}$ by complete induction. Suppose $n \in \mathbb{N}$ and assume the I.H. stating that

$$\text{(i) and (ii) hold for all } n' \in \mathbb{N} \text{ with } n' < n. \tag{D.2.1}$$

We now have to prove that (i) and (ii) hold for n.

(i) We prove claim (i) by induction on the combined size of the derivations of $\Delta \vdash_{\text{ex}}' T \leq U$ and $\Delta \vdash_{\text{ex}}' U \leq V$. We perform a case analysis on the last rules used in these derivations. The following tables lists all possible combinations; the rows contain the last rule used in $\Delta \vdash_{\text{ex}}' T \leq U$, the columns the last rule used in $\Delta \vdash_{\text{ex}}' U \leq V$. (The table omits the prefix "EXUPLO-" from the rule names.)

	Refl'	Object'	Extends'	Super'	Open'	Abstract'
Refl'	✓	✓	✓	✓	✓	✓
Object'	✓	✓	↯	✓	↯	(a)
Extends'	✓	✓	✓	✓	✓	✓
Super'	✓	✓	(b)	✓	↯	↯
Open'	✓	✓	✓	✓	✓	✓
Abstract'	✓	✓	↯	✓	(c)	↯

Cases marked with ✓ are trivial or follow directly from the inner induction hypothesis; cases marked with ↯ can never occur because they put conflicting constraints on the form of U. We now deal with the remaining cases.

(a) Then $U = \mathit{Object}$ and $V = \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N$. Further, the premise of rule EXUPLO-ABSTRACT' requires $\mathit{Object} = [\overline{Y/X}]N$, so $N = \mathit{Object}$. But this contradicts Restriction 5.13.

(b) Then $U = X$ and Δ contains an lower and upper bound for X. This is a contradiction to Restriction 5.15.

(c) Then $T = M$ and $U = \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N$ and

$$\frac{M = [\overline{Y/X}]N \qquad (\forall i)\ \Delta \Vdash_{\mathrm{ex}}{}' [\overline{Y/X}]P_i}{\Delta \vdash_{\mathrm{ex}}{}' M \leq \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N} \qquad \frac{\Delta, \overline{P} \vdash_{\mathrm{ex}}{}' N \leq V \qquad \mathsf{ftv}(\Delta, V) \cap \overline{X} = \emptyset}{\Delta \vdash_{\mathrm{ex}}{}' \exists\overline{X}\,\mathbf{where}\,\overline{P}\,.\,N \leq V}$$

We have

$$\mathsf{size}(\overline{P}) < \mathsf{size}(U) = n$$

With (D.2.1) we then get

$$[\overline{Y/X}]\Delta \vdash_{\mathrm{ex}}{}' [\overline{Y/X}]N \leq [\overline{Y/X}]V$$

Because $T = [\overline{Y/X}]N$ and $\overline{X} \cap \mathsf{ftv}(\Delta, V) = \emptyset$, we have

$$\Delta \vdash_{\mathrm{ex}}{}' T \leq V$$

as required.

(ii) We proceed by induction on the derivation $\mathcal{D}$ of $\Delta', \overline{P} \vdash_{\mathrm{ex}}{}' W_1 \leq W_2$. We have already proved (i) for n, so with (D.2.1)

$$\text{(i) holds for all } n' \in \mathbb{N} \text{ with } n' \leq n \tag{D.2.2}$$

Case distinction on the last rule used in $\mathcal{D}$.

- *Case* rule EXUPLO-REFL': Follows with Lemma D.2.1.
- *Case* rule EXUPLO-OBJECT': Trivial.
- *Case* rule EXUPLO-EXTENDS': We then have $W_1 = X$ and

$$\frac{X\,\mathbf{extends}\,W_2' \in \Delta', \overline{P} \qquad \Delta', \overline{P} \vdash_{\mathrm{ex}}{}' W_2' \leq W_2}{\Delta', \overline{P} \vdash_{\mathrm{ex}}{}' X \leq W_2}$$

Applying the inner I.H. yields

$$[\overline{Y/X}]\Delta' \vdash_{\mathrm{ex}}{}' [\overline{Y/X}]W_2' \leq [\overline{Y/X}]W_2 \tag{D.2.3}$$

– If $X\,\mathbf{extends}\,W_2' \in \overline{P}$ then

$$[\overline{Y/X}]\Delta' \vdash_{\mathrm{ex}}{}' [\overline{Y/X}]X \leq [\overline{Y/X}]W_2' \tag{D.2.4}$$

by the assumption. We also have

$$\mathsf{size}([\overline{Y/X}]W_2') = \mathsf{size}(W_2') \leq \mathsf{size}(\overline{P}) = n$$

Using (D.2.2) on (D.2.4) and (D.2.3) yields

$$[\overline{Y/X}]\Delta' \vdash_{\mathrm{ex}}{}' [\overline{Y/X}]X \leq [\overline{Y/X}]W_2$$

as required.

  - If X **extends** $W_2' \in \Delta'$ then $[\overline{Y/X}]X = X$ because $\overline{X} \cap \mathsf{dom}(\Delta) = \emptyset$. With (D.2.3) and rule exuplo-extends’, we get the required result.

- *Case* rule exuplo-super’: Follows analogously.
- *Case* rule exuplo-open’: Then $W_1 = \exists\overline{Z}\,\mathbf{where}\,\overline{Q}\,.\,N$ and

$$\frac{\Delta', \overline{P}, \overline{Q} \vdash_{\mathrm{ex}}{}' N \leq W_2 \qquad \overline{Z} \cap \mathsf{ftv}(\Delta', \overline{P}, W_2) = \emptyset}{\Delta', \overline{P} \vdash_{\mathrm{ex}}{}' \exists\overline{Z}\,\mathbf{where}\,\overline{Q}\,.\,N \leq W_2}$$

  Because the $\overline{Z}$ are sufficiently fresh, we may assume

$$[\overline{Y/X}](\exists\overline{Z}\,\mathbf{where}\,\overline{Q}\,.\,N) = \exists\overline{Z}\,\mathbf{where}\,([\overline{Y/X}]\overline{Q})\,.\,([\overline{Y/X}]N)$$
$$\overline{Z} \cap \mathsf{ftv}([\overline{Y/X}]\Delta, [\overline{Y/X}]W_2) = \emptyset$$

  Using the inner I.H. yields

$$[\overline{Y/X}](\Delta', \overline{Q}) \vdash_{\mathrm{ex}}{}' [\overline{Y/X}]N \leq [\overline{Y/X}]W_2$$

  Thus with exuplo-open’

$$[\overline{Y/X}]\Delta' \vdash_{\mathrm{ex}}{}' [\overline{Y/X}](\exists\overline{Z}\,\mathbf{where}\,\overline{Q}\,.\,N) \leq [\overline{Y/X}]W_2$$

- *Case* rule exuplo-abstract’: Then $W_2 = \exists\overline{Z}\,\mathbf{where}\,\overline{Q}\,.\,N$ and

$$\frac{W_1 = [\overline{Y'/Z}]N \qquad (\forall i)\ \Delta', \overline{P} \Vdash_{\mathrm{ex}}{}' [\overline{Y'/Z}]Q_i}{\Delta', \overline{P} \vdash_{\mathrm{ex}}{}' W_1 \leq \exists\overline{Z}\,\mathbf{where}\,\overline{Q}\,.\,N}$$

  Using the inner I.H., we can easily verify that

$$(\forall i)\ [\overline{Y/X}]\Delta' \Vdash_{\mathrm{ex}}{}' [\overline{Y/X}][\overline{Y'/Z}]Q_i$$

  Because the $\overline{Z}$ are sufficiently fresh, we may assume

$$[\overline{Y/X}](\exists\overline{Z}\,\mathbf{where}\,\overline{Q}\,.\,N) = \exists\overline{Z}\,\mathbf{where}\,([\overline{Y/X}]\overline{Q})\,.\,([\overline{Y/X}]N)$$
$$\overline{Z} \cap \overline{Y} = \emptyset$$

  Moreover, for $\varphi = [\overline{[\overline{Y/X}]Y'/Z}]$, we have

$$[\overline{Y/X}][\overline{Y'/Z}]N = \varphi[\overline{Y/X}]N$$
$$[\overline{Y/X}][\overline{Y'/Z}]\overline{Q} = \varphi[\overline{Y/X}]\overline{Q}$$

  Hence,

$$[\overline{Y/X}]W_1 = \varphi[\overline{Y/X}]N$$
$$(\forall i)\ [\overline{Y/X}]\Delta' \Vdash_{\mathrm{ex}}{}' \varphi[\overline{Y/X}]Q_i$$

  The claim now follows with rule exuplo-abstract’.

End case distinction on the last rule used in $\mathcal{D}$.

This finishes the proof of (D.2.1). □

Now we can prove that $\Delta \vdash_{\mathrm{ex}} T \leq U$ and $\Delta \vdash_{\mathrm{ex}}{}' T \leq U$ coincide.

Lemma D.2.4. *$\Delta \vdash_{\text{ex}} T \le U$ if, and only, if $\Delta \vdash_{\text{ex}}{}' T \le U$.*

Proof. Both directions of the lemma are proved by a straightforward induction on the derivation given. For the "⇒" direction, we note two things:

- When the derivation of $\Delta \vdash_{\text{ex}} T \le U$ ends with rule EXUPLO-TRANS, we apply the I.H. to the two subderivations and combine the two resulting derivations using Lemma D.2.3.
- When the derivation of $\Delta \vdash_{\text{ex}} T \le U$ ends with rule EXUPLO-ABSTRACT, we have $N = [\overline{T/X}]M$ as a premise. But the corresponding rule EXUPLO-ABSTRACT' requires $N = [\overline{Y/X}]M$. We can easily show $\overline{T} = \overline{Y}$ for some $\overline{Y}$ because N has the form $C\texttt{<}\overline{Z}\texttt{>}$ (see the syntax in Figure 5.3). □

Our next goal is to show that $[\![\Omega]\!]^- \vdash_{\text{ex}}{}' [\![\tau]\!]^- \le [\![\tau']\!]^+$ implies $\Omega \vdash_D \tau \le \tau'$. Before proving this fact, we need to establish some more lemmas. In the following, we use the notation $\mathcal{D} :: \mathcal{J}$ to denote that $\mathcal{D}$ is a derivation for judgment $\mathcal{J}$ and define $\mathsf{height}(\mathcal{D})$ as the height of $\mathcal{D}$.

Lemma D.2.5. *Suppose $X \notin \mathsf{ftv}(\Delta, T, U, V)$. If either $\mathcal{D} :: \Delta, X\,\textbf{super}\,T \vdash_{\text{ex}}{}' U \le V$ or $\mathcal{D} :: \Delta, X\,\textbf{extends}\,T \vdash_{\text{ex}}{}' U \le V$, then $\mathcal{D}' :: \Delta \vdash_{\text{ex}}{}' U \le V$ with $\mathsf{height}(\mathcal{D}) = \mathsf{height}(\mathcal{D}')$.*

Proof. Straightforward induction on $\mathcal{D}$. □

Lemma D.2.6.

(i) If $\mathcal{D} :: \Delta, X\,\textbf{super}\,T \vdash_{\text{ex}}{}' U \le X$ with $X \notin \mathsf{ftv}(\Delta, T, U)$, then $\mathcal{D}' :: \Delta \vdash_{\text{ex}}{}' U \le T$ with $\mathsf{height}(\mathcal{D}') \le \mathsf{height}(\mathcal{D})$.

(ii) If $\mathcal{D} :: \Delta, X\,\textbf{extends}\,T \vdash_{\text{ex}}{}' X \le U$ with $X \notin \mathsf{ftv}(\Delta, T, U)$, then $\mathcal{D}' :: \Delta \vdash_{\text{ex}}{}' T \le U$ with $\mathsf{height}(\mathcal{D}') \le \mathsf{height}(\mathcal{D})$.

Proof.

(i) Induction on $\mathcal{D}$.

Case distinction on the last rule of $\mathcal{D}$.

- *Case* rule EXUPLO-REFL': Impossible.
- *Case* rule EXUPLO-OBJECT': Impossible.
- *Case* rule EXUPLO-EXTENDS': Follows by I.H. and rule EXUPLO-EXTENDS'.
- *Case* rule EXUPLO-SUPER': Then $\Delta, X\,\textbf{super}\,T \vdash_{\text{ex}}{}' U \le T$ from the premise and the claim follows with Lemma D.2.5.
- *Case* rule EXUPLO-OPEN': Then $U = \exists\overline{Y}\,\textbf{where}\,\overline{Q}\,.\,N$ and

$$\text{EXUPLO-OPEN'}\;\dfrac{\text{EXUPLO-SUPER'}\;\dfrac{\mathcal{D}_1 :: \Delta, X\,\textbf{super}\,T, \overline{Q} \vdash_{\text{ex}}{}' N \le T}{\Delta, X\,\textbf{super}\,T, \overline{Q} \vdash_{\text{ex}}{}' N \le X} \qquad \overline{Y} \cap \mathsf{ftv}(\Delta, X, T) = \emptyset}{\mathcal{D} :: \Delta, X\,\textbf{super}\,T \vdash_{\text{ex}}{}' \exists\overline{Y}\,\textbf{where}\,\overline{Q}\,.\,N \le X}$$

We have $X \notin \mathsf{ftv}(\overline{Q}, N)$ because $X \notin \mathsf{ftv}(U)$. With Lemma D.2.5

$$\mathcal{D}_1' :: \Delta, \overline{Q} \vdash_{\text{ex}}{}' N \le T$$
$$\mathsf{height}(\mathcal{D}_1) = \mathsf{height}(\mathcal{D}_1')$$

The claim now follows with rule EXUPLO-OPEN'.

- *Case* rule EXUPLO-ABSTRACT': Impossible.

End case distinction on the last rule of $\mathcal{D}$.

(ii) *Case distinction* on the last rule of $\mathcal{D}$.

- *Case* rule EXUPLO-REFL’: Impossible.
- *Case* rule EXUPLO-OBJECT’: Trivial.
- *Case* rule EXUPLO-EXTENDS’: Then $\Delta, X\,\textbf{extends}\,T \vdash_{\text{ex}}{}' T \leq U$ from the premise and the claim follows with Lemma D.2.5.
- *Case* rule EXUPLO-SUPER’: Follows by I.H. and rule EXUPLO-SUPER’.
- *Case* rule EXUPLO-OPEN’: Impossible.
- *Case* rule EXUPLO-ABSTRACT’: Impossible.

End case distinction on the last rule of $\mathcal{D}$. □

Lemma D.2.7. *Let τ^- and σ^+ be $F^D_\leq$ types. Then $\llbracket\tau\rrbracket^- \neq \llbracket\sigma\rrbracket^+$.*

Proof. Obvious. □

Lemma D.2.8. *If $\llbracket\Omega\rrbracket^- \vdash_{\text{ex}}{}' \llbracket\tau\rrbracket^- \leq \llbracket\tau'\rrbracket^+$ then $\Omega \vdash_D \tau \leq \tau'$.*

Proof. Let $\llbracket\Omega\rrbracket^- = \Delta$, $\llbracket\tau\rrbracket^- = T$, and $\llbracket\tau'\rrbracket^+ = U$. Proceed by induction on the given derivation. *Case distinction* on the last rule of this derivation.

- *Case* rule EXUPLO-REFL’: Then $T = U$ so $\llbracket\tau\rrbracket^- = \llbracket\tau'\rrbracket^+$ which is impossible by Lemma D.2.7.

- *Case* rule EXUPLO-OBJECT’: Then $\tau' = \mathsf{Top}$ and the claim follows by D-TOP.

- *Case* rule EXUPLO-EXTENDS’: Then $T = X^\alpha$ and $\tau = \alpha$ and

  $$\frac{X\,\textbf{extends}\,T' \in \Delta \qquad \Delta \vdash_{\text{ex}}{}' T' \leq U}{\Delta \vdash_{\text{ex}}{}' X^\alpha \leq U}$$

  Because $\Delta = \llbracket\Omega\rrbracket^-$, we have $T' = \llbracket\sigma\rrbracket^-$ and $\Omega(\alpha) = \sigma^-$. Applying the I.H. yields

  $$\Omega \vdash_D \sigma \leq \tau'$$

  so the claim follows by rule D-VAR.

- *Case* rule EXUPLO-SUPER’: Impossible because n-positive types are not variables.

- *Case* rule EXUPLO-OPEN’: Hence $T = \exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,N$ and

  $$\frac{\Delta, \overline{P} \vdash_{\text{ex}}{}' N \leq T \qquad \overline{X} \cap \mathsf{ftv}(\Delta, U) = \emptyset}{\Delta \vdash_{\text{ex}}{}' \exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,N \leq U}$$

  From $T = \llbracket\tau\rrbracket^-$ we have

  $$\begin{aligned} \tau &= \forall\alpha_0 \ldots \alpha_n\,.\,\neg\,\sigma \\ T &= \neg\,\overbrace{\exists X^{\alpha_0} \ldots X^{\alpha_n}\,Y\,\textbf{where}\,Y\,\textbf{extends}\,\llbracket\sigma\rrbracket^+ \\ &\qquad .\,\mathbb{C}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>}}^{=T'} \\ &= \exists X\,\textbf{where}\,X\,\textbf{super}\,T'\,.\,\mathbb{D}\texttt{<}X\texttt{>} \end{aligned}$$

From $U = \llbracket \tau' \rrbracket^+$ we get that either $U = \mathit{Object}$ (then $\tau' = \mathsf{Top}$ and we are done) or that

$$\begin{aligned}
\tau' &= \forall \alpha_0 {\leq} \tau_0^- \ldots \alpha_n {\leq} \tau_n^- \,.\, \neg\, \sigma' \\
U &= \neg\, \overbrace{\begin{aligned}[t]\exists X^{\alpha_0} \ldots X^{\alpha_n}\, Y \,\mathbf{where}\, X^{\alpha_0} \,\mathbf{extends}\, \llbracket \tau_0 \rrbracket^- \ldots \\ X^{\alpha_n} \,\mathbf{extends}\, \llbracket \tau_n \rrbracket^- \\ Y \,\mathbf{extends}\, \llbracket \sigma' \rrbracket^- \\ .\, \mathbb{C}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>}\end{aligned}}^{=U'} \\
&= \exists X \,\mathbf{where}\, X \,\mathbf{super}\, U' \,.\, \mathbb{D}\texttt{<}X\texttt{>}
\end{aligned}$$

From $\Delta \vdash_{\text{ex}}{}' \; T \leq U$ we get by inverting the rules:

$$\text{EXUPLO-OPEN'}\;\dfrac{\text{EXUPLO-ABSTRACT'}\;\dfrac{\text{EXUPLO-SUPER'}\;\dfrac{\mathcal{D} :: \Delta, X \,\mathbf{super}\, T' \vdash_{\text{ex}}{}' \; U' \leq X}{\Delta, X \,\mathbf{super}\, T' \vdash_{\text{ex}}{}' \; X \,\mathbf{super}\, U'}}{\Delta, X \,\mathbf{super}\, T' \vdash_{\text{ex}}{}' \; \mathbb{D}\texttt{<}X\texttt{>} \leq \exists X \,\mathbf{where}\, X \,\mathbf{super}\, U' \,.\, \mathbb{D}\texttt{<}X\texttt{>}} \qquad X \notin \mathsf{ftv}(\Delta, U)}{\begin{aligned}\Delta \vdash_{\text{ex}}{}' \; \exists X \,\mathbf{where}\, X \,\mathbf{super}\, T' \,.\, \mathbb{D}\texttt{<}X\texttt{>} \leq \\ \exists X \,\mathbf{where}\, X \,\mathbf{super}\, U' \,.\, \mathbb{D}\texttt{<}X\texttt{>}\end{aligned}}$$

We have $X \notin \mathsf{ftv}(\Delta, T', U')$ so with Lemma D.2.6

$$\begin{gathered}
\mathcal{D}' :: \Delta \vdash_{\text{ex}}{}' \; U' \leq T' \\
\mathsf{height}(\mathcal{D}') \leq \mathsf{height}(\mathcal{D})
\end{gathered}$$

$\mathcal{D}'$ must end with rule EXUPLO-OPEN'. Define

$$\begin{gathered}
\Delta' = \Delta, X^{\alpha_0} \,\mathbf{extends}\, \llbracket \tau_0 \rrbracket^-, \ldots, X^{\alpha_n} \,\mathbf{extends}\, \llbracket \tau_n \rrbracket^- \\
\Delta'' = \Delta', Y \,\mathbf{extends}\, \llbracket \sigma' \rrbracket^-
\end{gathered}$$

Inverting the rules yields

$$\text{EXUPLO-OPEN'}\;\dfrac{\text{EXUPLO-ABSTRACT'}\;\dfrac{\ldots \qquad \text{EXUPLO-EXTENDS}\;\dfrac{\mathcal{D}'' :: \Delta'' \vdash_{\text{ex}}{}' \; Y \leq \llbracket \sigma \rrbracket^+}{\Delta'' \Vdash_{\text{ex}}{}' \; Y \,\mathbf{extends}\, \llbracket \sigma \rrbracket^+} \qquad \ldots}{\Delta'' \vdash_{\text{ex}}{}' \; \mathbb{C}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>} \leq T'}}{\mathcal{D}' :: \Delta \vdash_{\text{ex}}{}' \; U' \leq T'}$$

We have $Y \notin \mathsf{ftv}(\Delta', \llbracket \sigma' \rrbracket^-, \llbracket \sigma \rrbracket^+)$. Hence with Lemma D.2.6

$$\begin{gathered}
\mathcal{D}''' :: \Delta' \vdash_{\text{ex}}{}' \; \llbracket \sigma' \rrbracket^- \leq \llbracket \sigma \rrbracket^+ \\
\mathsf{height}(\mathcal{D}''') \leq \mathsf{height}(\mathcal{D}'')
\end{gathered}$$

Because $\mathcal{D}'''$ is smaller than the initial derivation, we can apply the I.H. and get

$$\Omega, \alpha_0 {\leq} \tau_0 \ldots \alpha_n {\leq} \tau_n \vdash_D \sigma' \leq \sigma$$

Then with rule D-ALL-NEG

$$\Omega \vdash_D \forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma \leq \forall \alpha_0 {\leq} \tau_0 \ldots \alpha_n {\leq} \tau_n \,.\, \neg\, \sigma'$$

as required.

- *Case* rule EXUPLO-ABSTRACT': Impossible because no class type N is in the image of the $\llbracket \cdot \rrbracket^-$ translation. □

Figure D.2 Constraint specificity.

$$\boxed{\Delta \vdash_{\mathrm{ex}} P \precsim Q}$$

$$\textsc{con-spec-upper}\ \dfrac{\Delta \vdash_{\mathrm{ex}} T \le T'}{\Delta \vdash_{\mathrm{ex}} X\ \mathbf{extends}\ T \precsim X\ \mathbf{extends}\ T'} \qquad \textsc{con-spec-lower}\ \dfrac{\Delta \vdash_{\mathrm{ex}} T' \le T}{\Delta \vdash_{\mathrm{ex}} X\ \mathbf{super}\ T \precsim X\ \mathbf{super}\ T'}$$

$$\boxed{\Delta \vdash_{\mathrm{ex}} \overline{P} \precsim \overline{Q}}$$

$$\textsc{con-spec-multi}\ \dfrac{(\forall i \in [n], \exists j \in [m])\ \Delta, \Delta_i \vdash_{\mathrm{ex}} P_j \precsim Q_i \text{ with } \Delta_i \subseteq \overline{P}}{\Delta \vdash_{\mathrm{ex}} \overline{P}^m \precsim \overline{Q}^n}$$

End case distinction on the last rule of this derivation.

The next three lemmas are required to prove that $\Omega \vdash_D \tau \le \sigma$ implies $[\![\Omega \vdash_D \tau \le \sigma]\!]$. We first prove a standard weakening lemma.

Lemma D.2.9. *If $\Delta \vdash_{\mathrm{ex}} T \le U$ and $\Delta \subseteq \Delta'$ then $\Delta' \vdash_{\mathrm{ex}} T \le U$.*

Proof. Straightforward induction on the derivation given. □

The next lemma shows that the negation operator for EXuplo types allows us to swap the left- and right-hand sides of a subtyping judgment.

Lemma D.2.10. *If $\Delta \vdash_{\mathrm{ex}} U \le T$ then $\Delta \vdash_{\mathrm{ex}} \neg T \le \neg U$.*

Proof. We have

$$\begin{aligned} \neg T &= \exists X\ \mathbf{where}\ X\ \mathbf{super}\ T\,.\,\mathbb{D}\texttt{<}X\texttt{>} \\ \neg U &= \exists X\ \mathbf{where}\ X\ \mathbf{super}\ U\,.\,\mathbb{D}\texttt{<}X\texttt{>} \end{aligned}$$

Assume $\Delta \vdash_{\mathrm{ex}} U \le T$. Then $\Delta, X\ \mathbf{super}\ T \vdash_{\mathrm{ex}} U \le T$ with Lemma D.2.9. Hence

$$\textsc{exuplo-open}\ \dfrac{\textsc{exuplo-abstract}\ \dfrac{\textsc{exuplo-super}\ \dfrac{\textsc{exuplo-super}\ \dfrac{\Delta, X\ \mathbf{super}\ T \vdash_{\mathrm{ex}} U \le T}{\Delta, X\ \mathbf{super}\ T \vdash_{\mathrm{ex}} U \le X}}{\Delta, X\ \mathbf{super}\ T \vdash_{\mathrm{ex}} X\ \mathbf{super}\ U}}{\Delta, X\ \mathbf{super}\ T \vdash_{\mathrm{ex}} \mathbb{D}\texttt{<}X\texttt{>} \le \exists X\ \mathbf{where}\ X\ \mathbf{super}\ U\,.\,\mathbb{D}\texttt{<}X\texttt{>}} \qquad X \notin \mathsf{ftv}(\Delta, \neg U)}{\Delta \vdash_{\mathrm{ex}} \exists X\ \mathbf{where}\ X\ \mathbf{super}\ T\,.\,\mathbb{D}\texttt{<}X\texttt{>} \le \exists X\ \mathbf{where}\ X\ \mathbf{super}\ U\,.\,\mathbb{D}\texttt{<}X\texttt{>}}$$

□

The relation $\Delta \vdash_{\mathrm{ex}} \overline{P} \precsim \overline{Q}$, defined in Figure D.2, expresses that the constraints $\overline{P}$ are more specific than the constraints $\overline{Q}$. We now connect $\precsim$ with subtyping on existentials.

Lemma D.2.11. *If $\Delta \vdash_{\mathrm{ex}} \overline{P} \precsim \overline{Q}$ then $\Delta \vdash_{\mathrm{ex}} \exists\overline{X}\ \mathbf{where}\ \overline{P}\,.\,N \le \exists\overline{X}\ \mathbf{where}\ \overline{Q}\,.\,N$.*

Proof. It is easy to see that $\Delta \vdash_{\text{ex}} \overline{P} \precsim \overline{Q}$ implies $\Delta, \overline{P} \vdash_{\text{ex}} Q$ for all $Q \in \overline{Q}$. Then we have

$$\textsc{exuplo-open}\ \dfrac{\begin{array}{c}\textsc{exuplo-abstract}\ \dfrac{(\forall i)\ \Delta, \overline{P} \Vdash_{\text{ex}} Q_i}{\Delta, \overline{P} \vdash_{\text{ex}} N \leq \exists \overline{X}\,\textbf{where}\,\overline{Q}\,.\,N} \\ \overline{X} \cap \mathsf{ftv}(\Delta, \exists \overline{X}\,\textbf{where}\,\overline{Q}\,.\,N) = \emptyset\end{array}}{\Delta \vdash_{\text{ex}} \exists \overline{X}\,\textbf{where}\,\overline{P}\,.\,N \leq \exists \overline{X}\,\textbf{where}\,\overline{Q}\,.\,N} \qquad \square$$

Now we are ready to prove undecidability of subtyping in EXuplo.

Proof of Theorem 5.17. We need to prove the following claim

$\Omega \vdash_D \tau \leq \tau'$ *if, and only if,* $\llbracket \Omega \vdash_{\text{ex}} \tau \leq \tau' \rrbracket$.

We prove the two directions of the claim separately.

"$\Rightarrow$": Assume $\Omega \vdash_D \tau \leq \tau'$. We proceed by induction on the derivation of $\Omega \vdash_D \tau \leq \tau'$. *Case distinction* on the last rule used.

- *Case* rule D-TOP: Then $\llbracket \tau' \rrbracket^+ = \mathit{Object}$ and the claim is obvious.
- *Case* rule D-VAR: Then $\tau = \alpha$ and

$$\dfrac{\Omega \vdash_D \Omega(\alpha) \leq \tau' \qquad \tau' \neq \mathsf{Top}}{\Omega \vdash_D \alpha \leq \tau'}$$

  Then

$$X^\alpha\,\textbf{extends}\,\llbracket \Omega(\alpha) \rrbracket^- \in \llbracket \Omega \rrbracket^-$$

  and by the I.H.

$$\llbracket \Omega \rrbracket^- \vdash_{\text{ex}} \llbracket \Omega(\alpha) \rrbracket^- \leq \llbracket \tau' \rrbracket^+$$

  The claim now follows with rules EXUPLO-EXTENDS and EXUPLO-TRANS.

- *Case* rule D-ALL-NEG: Then

$$\dfrac{\Omega, \alpha_0 \leq \tau_0 \ldots \alpha_n \leq \tau_n \vdash_D \sigma' \leq \sigma}{\Omega \vdash_D \underbrace{\forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma}_{=\tau} \leq \underbrace{\forall \alpha_0 \leq \tau_0 \ldots \alpha_n \leq \tau_n \,.\, \neg\, \sigma'}_{=\tau'}}$$

  and

$$\begin{aligned} \llbracket \tau \rrbracket^- &= \neg\ \overbrace{\exists X^{\alpha_0} \ldots X^{\alpha_n}\, Y\,\textbf{where}\,Y\,\textbf{extends}\,\llbracket \sigma \rrbracket^+ \,.\, \mathbb{C}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>}}^{=T} \\ \llbracket \tau' \rrbracket^+ &= \neg\ \overbrace{\exists X^{\alpha_0} \ldots X^{\alpha_n}\, Y\,\textbf{where}\,X^{\alpha_0}\,\textbf{extends}\,\llbracket \tau_0 \rrbracket^- \ldots X^{\alpha_n}\,\textbf{extends}\,\llbracket \tau_n \rrbracket^-\ Y\,\textbf{extends}\,\llbracket \sigma' \rrbracket^- \,.\, \mathbb{C}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>}}^{=U} \end{aligned}$$

  Define

$$\begin{aligned} \Delta &= \llbracket \Omega \rrbracket^- \\ \Delta' &= \Delta, X^{\alpha_0}\,\textbf{extends}\,\llbracket \tau_0 \rrbracket^- \ldots X^{\alpha_n}\,\textbf{extends}\,\llbracket \tau_n \rrbracket^- \end{aligned}$$

Note that $[\![\Omega, \alpha_0 \leq \tau_0 \ldots \alpha_n \leq \tau_n]\!]^- = \Delta'$.
We must show $\Delta \vdash_{\text{ex}} \neg T \leq \neg U$. By applying the I.H. we get

$$\Delta' \vdash_{\text{ex}} [\![\sigma']\!]^- \leq [\![\sigma]\!]^+$$

Thus

$$\Delta' \vdash_{\text{ex}} Y\,\textbf{extends}\,[\![\sigma']\!]^- \precsim Y\,\textbf{extends}\,[\![\sigma]\!]^+$$

Hence

$$\begin{aligned}\Delta \vdash_{\text{ex}}\ & X^{\alpha_0}\,\textbf{extends}\,[\![\tau_0]\!]^- \ldots X^{\alpha_n}\,\textbf{extends}\,[\![\tau_n]\!]^-, Y\,\textbf{extends}\,[\![\sigma']\!]^- \\ & \precsim\ Y\,\textbf{extends}\,[\![\sigma]\!]^+\end{aligned}$$

By Lemma D.2.11

$$\Delta \vdash_{\text{ex}} U \leq T$$

By Lemma D.2.10

$$\Delta \vdash_{\text{ex}} \neg T \leq \neg U$$

End case distinction on the last rule used.

"$\Leftarrow$": Assume $[\![\Omega \vdash_D \tau \leq \tau']\!]$. Let

$$\begin{aligned}\Delta &= [\![\Omega]\!]^- \\ T &= [\![\tau]\!]^- \\ U &= [\![\tau']\!]^+\end{aligned}$$

Hence, $\Delta \vdash_{\text{ex}} T \leq U$. By Lemma D.2.4 we then have $\Delta \vdash_{\text{ex}}{}' T \leq U$. Thus, by Lemma D.2.8, $\Omega \vdash_{\text{ex}} \tau \leq \tau'$. □

D.2.2 Proof of Theorem 5.19

Theorem 5.19 states that subtyping in EXuplo becomes decidable if all type environments involved are contractive and if support for lower bounds is dropped. Lemma D.2.4 proves equivalence of $\Delta \vdash_{\text{ex}} T \leq U$ and $\Delta \vdash_{\text{ex}}{}' T \leq U$, so we only need to prove that the algorithm induced by the rules defining the judgment $\Delta \vdash_{\text{ex}}{}' T \leq U$ terminates.

Define

$$\begin{aligned}\mathsf{weight}''_\Delta(X) &:= 1 + \mathsf{max}\{\mathsf{weight}''_\Delta(T) \mid X\,\textbf{extends}\,T \in \Delta\} \\ \mathsf{weight}''_\Delta(N) &:= 1 \\ \mathsf{weight}''_\Delta(\exists\overline{X}\,\textbf{where}\,\overline{P}\,.\,N) &:= 1\end{aligned}$$

This definition is proper (i.e., terminates) because Δ is contractive. Using Definition D.2.2, which defines the size of EXuplo types and constraints, specify a measure μ on subtyping judgments as follows:

$$\mu(\Delta \vdash_{\text{ex}}{}' T \leq U) = (\mathsf{size}(U), \mathsf{weight}''_\Delta(T), \mathsf{size}(T)) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

Then the measure μ decreases according to the usual lexicographic ordering on triples of natural numbers when moving from conclusions to premises in a derivation of $\Delta \vdash_{\text{ex}}{}' T \leq U$.
Case distinction on the last rule in the derivation of $\Delta \vdash_{\text{ex}}{}' T \leq U$.

- *Case* rule EXUPLO-EXTENDS': Then $T = X$ and the premise contains the recursive invocation $\Delta \vdash_{\mathrm{ex}}{}' T' \leq U$ with $X\, \mathbf{extends}\, T' \in \Delta$. In this case, the measure decreases because $\mathsf{size}(U) = \mathsf{size}(U)$ and $\mathsf{weight}''_{\Delta}(T) > \mathsf{weight}''_{\Delta}(T')$.
- *Case* rule EXUPLO-OPEN': Then $T = \exists \overline{X}\, \mathbf{where}\, \overline{P} \,.\, N$ and the premise contains the recursive invocation $\Delta, \overline{P} \vdash_{\mathrm{ex}}{}' N \leq U$. In this case, the measure decreases because $\mathsf{size}(U) = \mathsf{size}(U)$, $\mathsf{weight}''_{\Delta}(T) = \mathsf{weight}''_{\Delta,\overline{P}}(N)$, and $\mathsf{size}(T) > \mathsf{size}(N)$.
- *Case* rule EXUPLO-ABSTRACT': Then $T = N$, $U = \exists \overline{X}\, \mathbf{where}\, \overline{P} \,.\, M$, and $N = [\overline{Y/X}]M$. Assume $P \in \overline{P}$ with $P = V\, \mathbf{extends}\, W$. ($P$ cannot be a **super**-constraint because lower bounds are not supported.) The premise now contains the recursive invocation $\Delta \vdash_{\mathrm{ex}}{}' [\overline{Y/X}]V \leq [\overline{Y/X}]W$. In this case, the measure decreases because $\mathsf{size}(U) > \mathsf{size}(P) = \mathsf{size}(W) = \mathsf{size}([\overline{Y/X}]W)$.
- *Case* rule EXUPLO-SUPER': Impossible because lower bounds are not supported.
- *Case* any other rule: Irrelevant because no recursive invocations are present.

End case distinction on the last rule in the derivation of $\Delta \vdash_{\mathrm{ex}}{}' T \leq U$. □

D.2.3 Proof of Theorem 5.21

Theorem 5.21 states that subtyping in EXuplo becomes decidable if all type environments involved are contractive, if support for upper bounds is dropped, and if all existentials are variable-bounded. As in the preceding section, it suffices to show that the algorithm induced by the rules defining the judgment $\Delta \vdash_{\mathrm{ex}}{}' T \leq U$ terminates.

Define

$$\begin{aligned} \mathsf{weight}'''_{\Delta}(X) &:= 1 + \max\{\mathsf{weight}'''_{\Delta}(T) \mid X\, \mathbf{super}\, T \in \Delta\} \\ \mathsf{weight}'''_{\Delta}(N) &:= 1 \\ \mathsf{weight}'''_{\Delta}(\exists \overline{X}\, \mathbf{where}\, \overline{P} \,.\, N) &:= 1 \end{aligned}$$

This definition is proper (i.e., terminates) because Δ is contractive. Using Definition D.2.2, which defines the size of EXuplo types and constraints, specify a measure μ on subtyping judgments as follows:

$$\mu(\Delta \vdash_{\mathrm{ex}}{}' T \leq U) = (\mathsf{size}(T), \mathsf{weight}'''_{\Delta}(U)) \in \mathbb{N} \times \mathbb{N}$$

Then the measure μ decreases according to the usual lexicographic ordering on pairs of natural numbers when moving from conclusions to premises in a derivation of $\Delta \vdash_{\mathrm{ex}}{}' T \leq U$.
Case distinction on the last rule in the derivation of $\Delta \vdash_{\mathrm{ex}}{}' T \leq U$.

- *Case* rule EXUPLO-SUPER': Then $U = X$ and the premise contains the recursive invocation $\Delta \vdash_{\mathrm{ex}}{}' T \leq U'$ with $X\, \mathbf{super}\, U' \in \Delta$. In this case, the measure decreases because $\mathsf{size}(T) = \mathsf{size}(T)$ and $\mathsf{weight}'''_{\Delta}(U) > \mathsf{weight}'''_{\Delta}(U')$.
- *Case* rule EXUPLO-OPEN': Then $T = \exists \overline{X}\, \mathbf{where}\, \overline{P} \,.\, N$ and the premise contains the recursive invocation $\Delta \vdash_{\mathrm{ex}}{}' N \leq U$. In this case, the measure decreases because $\mathsf{size}(T) > \mathsf{size}(N)$.
- *Case* rule EXUPLO-ABSTRACT': Then $T = N$, $U = \exists \overline{X}\, \mathbf{where}\, \overline{P} \,.\, M$, and $N = [\overline{Y/X}]M$. Assume $P \in \overline{P}$ with $P = V\, \mathbf{super}\, W$. ($P$ cannot be an **extends**-constraint because lower bounds are not supported.) All existentials are variable-bounded, so $W = Z$ for some Z. The premise now contains the recursive invocation $\Delta \vdash_{\mathrm{ex}}{}' [\overline{Y/X}]Z \leq [\overline{Y/X}]V$. With Restriction 5.13 and $N = [\overline{Y/X}]M$, we get $\mathsf{size}(T) = \mathsf{size}(N) > 1$. Thus, the measure decreases because $\mathsf{size}(T) > \mathsf{size}([\overline{Y/X}]Z)$.

- *Case* rule EXUPLO-EXTENDS': Impossible because upper bounds are not supported.
- *Case* any other rule: Irrelevant because no recursive invocations are present.

End case distinction on the last rule in the derivation of $\Delta \vdash_{\mathrm{ex}}{}' T \leq U$. □

Bibliography and Index

Bibliography

[1] Eric Allen, Joseph J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. Modular multiple dispatch with multiple inheritance. In *ACM Symposium on Applied Computing (SAC)*, pages 1117–1121, Seoul, Korea, 2007. ACM Press.

[2] Davide Ancona and Elena Zucca. True modules for Java-like languages. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 354–380, Budapest, Hungary, 2001. Springer-Verlag.

[3] Apache Software Foundation. Apache Tomcat, 2009. `http://tomcat.apache.org/`.

[4] Apple Inc. The Objective-C programming language, 2009. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

[5] Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[6] AspectJ Team. The AspectJ development environment guide, 2009. `http://www.eclipse.org/aspectj/doc/released/devguide/index.html`.

[7] AspectJ Team. The AspectJ programming guide, 2009. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

[8] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[9] Bruce H. Barnes and Terry B. Bollinger. Making reuse cost-effective. *IEEE Software*, 8(1):13–24, 1991.

[10] Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & Half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Revised 3/02, Ohio State University, 2002. `http://www.csc.lsu.edu/~gb/Brew/Publications/HalfNHalf.pdf`.

[11] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, 2004.

[12] Alexander Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems & Structures*, 31(3–4):107–126, 2005.

[13] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 177–189, San Diego, CA, USA, 2005. ACM Press.

[14] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Joint Modular Languages Conference (JMLC)*, volume 2789 of *Lecture Notes in Computer Science*, pages 122–131, Klagenfurt, Austria, 2003. Springer-Verlag.

[15] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2–3):83–108, 2008.

[16] Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas Priesnitz. A comparison of C++ concepts and Haskell type classes. In *ACM SIGPLAN Workshop on Generic Programming*, pages 37–48, Victoria, BC, Canada, 2008. ACM Press.

[17] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[18] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, Portland, OR, USA, 2006. ACM Press.

[19] Barry W. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[20] Daniel Bonniot. Using kinds to type partially-polymorphic methods. *Electronic Notes in Theoretical Computer Science*, 75:21–40, 2003.

[21] Daniel Bonniot, Bryn Keller, and Francis Barber. The Nice user's manual, 2003. http://nice.sourceforge.net/manual.html.

[22] Viviana Bono, Ferruccio Damiani, and Elena Giachino. On traits and types in a Java-like setting. In *IFIP International Conference On Theoretical Computer Science (TCS)*, pages 367–382, Milano, Italy, 2008. Springer-Verlag.

[23] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 302–315, Paris, France, 1997. ACM Press.

[24] Gilad Bracha. Generics in the Java programming language, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[25] Gilad Bracha and William Cook. Mixin-based inheritance. In *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*, pages 303–311, Ottawa, ON, Canada, 1990. ACM Press.

[26] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, Canada, 1998. ACM Press.

[27] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (fifth edition), 2008. http://www.w3.org/TR/REC-xml.

[28] Manfred Broy, Wassiou Sitou, and Tony Hoare, editors. *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Science for Peace and Security Series - D: Information and Communication Security*. IOS Press BV, 2009.

[29] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[30] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 389–413, Oslo, Norway, 2004. Springer-Verlag.

[31] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, Brussels, Belgium, 1998. Springer-Verlag.

[32] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good ”match” for object-oriented languages. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127, Jyväskylä, Finland, 1997. Springer-Verlag.

[33] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *Lecture Notes in Computer Science*, pages 27–51, Åarhus, Denmark, 1995. Springer-Verlag.

[34] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2):225–290, 2003.

[35] Martin Büchi and Wolfgang Weck. Compound types for Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 362–373, Vancouver, BC, Canada, 1998. ACM Press.

[36] Nicholas Cameron and Sophia Drossopoulou. On subtyping, wildcards, and existential types. In *International Workshop on Formal Techniques for Java-like Programs (FTfJP)*, pages 1–7, Genova, Italy, 2009. ACM Press.

[37] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 2–26, Paphos, Cyprus, 2008. Springer-Verlag.

[38] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an existential types model for Java wildcards. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, informal proceedings, pages 1–13, 2007. http://cs.nju.edu.cn/boyland/ftjp/proceedings.pdf.

[39] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 273–280, London, UK, 1989. ACM Press.

[40] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[41] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 241–253, Tallinn, Estonia, 2005. ACM Press.

[42] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–13, Long Beach, CA, USA, 2005. ACM Press.

[43] Craig Chambers. Object-oriented multi-methods in Cecil. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, 1992.

[44] Craig Chambers and Gary T. Leavens. BeCecil, a core object-oriented language with block structure and multimethods: Semantics and typing. Technical Report TR-96-12-02, University of Washington, Department of Computer Science and Engineering, 1996.

[45] Craig Chambers and the Cecil Group. The Cecil language: Specification and rationale, version 3.2, 2004. http://www.cs.washington.edu/research/projects/cecil/pubs/cecil-spec.html.

[46] Juan Chen. Decidable subclassing-bounded quantification. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 37–46, Long Beach, CA, USA, 2005. ACM Press.

[47] James Clark and Steve DeRose. XML path language (XPath), version 1.0, 1999. http://www.w3.org/TR/xpath.

[48] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 121–134, Vancouver, BC, Canada, 2007. ACM Press.

[49] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 130–145, Minneapolis, MN, USA, 2000. ACM Press.

[50] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, 2006.

[51] William R. Cook. A proposal for making Eiffel type-safe. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 57–70, Nottingham, UK, 1989. Cambridge University Press.

[52] William R. Cook. Object-oriented programming versus abstract data types. In *REX School/Workshop on Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178, Noordwijkerhout, The Netherlands, 1991. Springer-Verlag.

[53] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 125–135, San Francisco, CA, USA, 1990. ACM Press.

[54] O.-J. Dahl, B. Myrhaug, and K. Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computing Center, Oslo, Norway, 1970. Revised version 1984.

[55] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 156–168, Austin, TX, USA, 1995. ACM Press.

[56] Tom DeMarco. *Why Does Software Cost So Much?* Dorset House Publishing, 1995.

[57] Dom4j — An open source XML framework for Java, 2008. http://www.dom4j.org/.

[58] Stéphane Ducasse. Putting traits in perspective. In *International Conference on Software Engineering (ICSE)*, volume 5634 of *Lecture Notes in Computer Science*, pages 5–8. Springer-Verlag, 2009.

[59] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

[60] Eclipse — An open development platform, 2009. http://www.eclipse.org/.

[61] Eclipse Foundation. Eclipse public license, 2004. http://www.eclipse.org/legal/epl-v10.html.

[62] Eclipse Foundation. Eclipse compiler for Java, 2008. http://download.eclipse.org/eclipse/downloads/drops/R-3.4.1-200809111700/index.php.

[63] ECMA International. Standard 334: C# language specification, 2nd edition, 2002. http://www.ecma-international.org/publications/standards/Ecma-334-arch.htm.

[64] ECMA International. Standard 334: C# language specification, 3rd edition, 2005. http://www.ecma-international.org/publications/standards/Ecma-334-arch.htm.

[65] ECMA International. Standard 335: Common language infrastructure, 4th edition, 2006. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[66] Burak Emir, Andrew Kennedy, Claudio V. Russo, and Dachuan Yu. Variance and generalized constraints for C# generics. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303, Nantes, France, 2006. Springer-Verlag.

[67] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Åarhus, Denmark, 1999.

[68] Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326, Budapest, Hungary, 2001. Springer-Verlag.

[69] Erik Ernst. Higher-order hierarchies. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 303–329, Darmstadt, Germany, 2003. Springer-Verlag.

[70] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 270–282, Charleston, SC, USA, 2006. ACM Press.

[71] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211, Brussels, Belgium, 1998. Springer-Verlag.

[72] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montreal, QC, Canada, 1998. ACM Press.

[73] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[74] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(02):145–205, 2007.

[75] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 115–134, Anaheim, CA, USA, 2003. ACM Press.

[76] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–152, Montreal, QC, CA, 2007. ACM Press.

[77] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1–2):75–96, 1998.

[78] Martin Giese. The Java pretty printer library, 2007. http://jpplib.sourceforge.net/.

[79] Joseph Gil and Itay Maman. Whiteoak: Introducing structural typing into Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 73–90, Nashville, TN, USA, 2008. ACM Press.

[80] Jean-Yves Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur*. PhD thesis, University of Paris VII, 1972.

[81] Adele Goldberg and David Robson. *Smalltalk 80: The Language*. Addison-Wesley, 1989.

[82] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[83] Douglas Gregor. Generic programming in ConceptC++, 2008. http://www.generic-programming.org/languages/conceptcpp/.

[84] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 291–310, Portland, OR, USA, 2006. ACM Press.

[85] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[86] William Harrison and Harold Ossher. Subject-oriented programming: A critique of pure objects. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 411–428, Washington, D.C., USA, 1993. ACM Press.

[87] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*, pages 169–180, Ottawa, ON, Canada, 1990. ACM Press.

[88] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[89] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 707 of *Lecture Notes in Computer Science*, pages 36–56. Springer-Verlag, 1993.

[90] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[91] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cJ: Enhancing Java with safe type conditions. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 185–198, Vancouver, BC, Canada, 2007. ACM Press.

[92] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[93] Oliver Hummel and Colin Atkinson. The managed adapter pattern: Facilitating glue code generation for component reuse. In *International Conference on Software Reuse (ICSR)*, pages 211–224, Falls Church, VA, USA, 2009. Springer-Verlag.

[94] Jason Hunter and Brett McLaughlin. JDOM, 2007. http://www.jdom.org/.

[95] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Information and Computation*, 177(1):56–89, 2002.

[96] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[97] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 113–132, Montreal, QC, CA, 2007. ACM Press.

[98] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, USA, 1986. ACM Press.

[99] Ivar Jacobson. *Object-Oriented Software Engineering*. Addison-Wesley, 1993. Revised printing.

[100] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In *International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830 of *Lecture Notes in Computer Science*, pages 228–244, Erfurt, Germany, 2003. Springer-Verlag.

[101] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, San Diego, CA, USA, 2005. ACM Press.

[102] Jaxen — A universal Java XPath engine, 2008. http://jaxen.codehaus.org/.

[103] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 52–61, Copenhagen, Denmark, 1993. ACM Press.

[104] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[105] Mark P. Jones. Type classes with functional dependencies. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244, Berlin, Germany, 2000. Springer-Verlag.

[106] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of Ariane. *IEEE Computer*, 30(1):129–130, 1997.

[107] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *European Symposium on Programming (ESOP)*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144, Nancy, France, 1988. Springer-Verlag.

[108] Tetsuo Kamina and Tetsuo Tamai. Lightweight scalable components. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 145–154, Salzburg, Austria, 2007. ACM Press.

[109] Tetsuo Kamina and Tetsuo Tamai. Lightweight dependent classes. In *ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE)*, pages 113–124, Nashville, TN, USA, 2008. ACM Press.

[110] Ralph Keller and Urs Hölzle. Binary component adaptation. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329, Brussels, Belgium, 1998. Springer-Verlag.

[111] Andrew Kennedy and Claudio Russo. Generalized algebraic data types and object-oriented programming. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–40, San Diego, CA, USA, 2005. ACM Press.

[112] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Snowbird, UT, USA, 2001. ACM Press.

[113] Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, informal proceedings, 2007. http://foolwood07.cs.uchicago.edu/program/kennedy-abstract.html.

[114] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, 2001. Springer-Verlag.

[115] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[116] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system, 2005. http://homepages.cwi.nl/~ralf/OOHaskell/.

[117] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *ACM SIGPLAN Haskell Workshop*, pages 96–107, Snowbird, UT, USA, September 2004.

[118] Oleg Kiselyov and Chung-chieh Shan. Functional pearl: Implicit configurations—or, type classes reflect the values of types. In *ACM SIGPLAN Haskell Workshop*, pages 33–44, Snowbird, UT, USA, September 2004.

[119] Ralf Lämmel and Klaus Ostermann. Software extension and integration with type classes. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 161–170, Portland, OR, USA, 2006. ACM Press.

[120] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. *The Computer Journal*, 43(6):469–481, 2000.

[121] Gary T. Leavens and Todd D. Millstein. Multiple dispatch as dispatch on tuples. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 374–387, Vancouver, BC, Canada, 1998. ACM Press.

[122] Xavier Leroy. The Objective Caml system release 3.11, 2008. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[123] Nancy Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[124] Wayne C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 1994.

[125] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[126] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 30(2):1–32, 2008.

[127] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU reference manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[128] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. Theta reference manual, preliminary version, 1995. http://www.pmg.csail.mit.edu/papers/thetaref.ps.gz.

[129] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[130] Vasily Litvinov. *Constraint-bounded polymorphism: An expressive and practical type system for object-oriented languages*. PhD thesis, University of Washington, 2003.

[131] Vassily Litvinov. Contraint-based polymorphism in Cecil: Towards a practical and static type system. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 388–411, Vancouver, BC, Canada, 1998. ACM Press.

[132] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–406, New Orleans, LA, USA, 1989. ACM Press.

[133] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[134] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 260–284, Paphos, Cyprus, 2008. Springer-Verlag.

[135] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? An empirical study. In *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 95–111, York, United Kingdom, 2009. Springer-Verlag.

[136] Michael Mattsson, Jan Bosch, and Mohamed E. Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10):80–87, 1999.

[137] Karl Mazurak and Steve Zdancewic. Type inference for Java 5: Wildcards, F-bounds, and undecidability, 2006. http://www.cis.upenn.edu/~stevez/note.html.

[138] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fasioned Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–222, Tampa Bay, FL, USA, 2001. ACM Press.

[139] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *Workshop on C++ Template Programming*, informal proceedings, 2000. http://www.oonumerics.org/tmpw00/mcnamara.pdf.

[140] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[141] Bertrand Meyer. Static typing. *ACM SIGPLAN OOPS Messenger*, 6(4):20–29, 1995.

[142] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[143] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 52–67, Seattle, WA, USA, 2002. ACM Press.

[144] Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In Mehmet Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2000.

[145] Microsoft Corporation. Component object model (COM), 2009. http://www.microsoft.com/com.

[146] Todd Millstein. Practical predicate dispatch. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364, Vancouver, BC, Canada, 2004. ACM Press.

[147] Todd Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and modular predicate dispatch for Java. *ACM Transactions on Programming Languages and Systems*, 31(2):1–54, 2009.

[148] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 224–240, Anaheim, CA, USA, 2003. ACM Press.

[149] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303, Lisbon, Portugal, 1999. Springer-Verlag.

[150] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[151] Markus Mohnen. Interfaces with default implementations in Java. In *Conference on the Principles and Practice of Programming in Java (PPPJ)*, pages 35–40, Dublin, Ireland, 2002. ACM Press.

[152] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 423–438, Nashville, TN, USA, 2008. ACM Press.

[153] James Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[154] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 563–582, Nashville, TN, USA, 2008. ACM Press.

[155] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, Inc., 2004.

[156] Nathan Myers. A new and useful template technique: "traits". In Stanley B. Lippman, editor, *C++ gems*, pages 451–457. SIGS Publications, Inc., 1996.

[157] MzScheme — Core virtual machine for PLT Scheme, 2009. http://www.plt-scheme.org/software/mzscheme/.

[158] National Institute of Standards and Technology. Software errors cost U.S. economy $59.5 billion annually, 2002. http://www.nist.gov/public_affairs/releases/n02-10.htm.

[159] Peter Naur and Brian Randell. Software engineering: Report of a conference sponsored by the NATO science committee, 1969. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[160] Peter G. Neumann. The risks digest, 2009. http://catless.ncl.ac.uk/Risks.

[161] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 99–115, Vancouver, BC, Canada, 2004. ACM Press.

[162] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–36, Portland, OR, USA, 2006. ACM Press.

[163] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 457–474, Nashville, TN, USA, 2008. ACM Press.

[164] Object Management Group. Common object request broker architecture (CORBA), version 3.1, 2008. http://www.omg.org/spec/CORBA/3.1.

[165] Object Management Group. Unified modeling language (UML), infrastructure specification, version 2.2, 2009. http://www.omg.org/spec/UML/2.2/.

[166] Martin Odersky. The Scala language specification, version 2.7, 2009. Draft, http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[167] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, informal proceedings, 2005. http://homepages.inf.ed.ac.uk/wadler/fool/program/10.html.

[168] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 41–58, San Diego, CA, USA, 2005. ACM Press.

[169] Harold Ossher and Peri Tarr. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In *International Conference on Software Engineering (ICSE)*, pages 687–688, Los Angeles, CA, USA, 1999. ACM Press.

[170] Harold Ossher and Peri Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *International Conference on Software Engineering (ICSE)*, pages 734–737, Limerick, Ireland, 2000. ACM Press.

[171] Klaus Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, 2008. http://www.jot.fm/issues/issue_2008_01/article4/.

[172] Claus H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 407–417, New Orleans, LA, USA, 1989. ACM Press.

[173] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[174] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: An exploration of the design space. In *Haskell Workshop*, Amsterdam, The Netherlands, 1997.

[175] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.

[176] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[177] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[178] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. Open multi-methods for C++. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 123–134, Salzburg, Austria, 2007. ACM Press.

[179] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Åarhus University, Denmark, 1981.

[180] Martin Plümicke. Java type unification with wildcards. In *International Workshop on Unification (UNIF)*, Paris, France, 2007. http://www.lsv.ens-cachan.fr/Events/rdp07/unif.html.

[181] Martin Plümicke. Typeless programming in Java 5.0 with wildcards. In *Internation Symposium on the Principles and Practice of Programming in Java (PPPJ)*, pages 73–82, Lisboa, Portugal, 2007. ACM Press.

[182] Emil L. Post. A variant of a recursivley unsolvable problem. *Bulletin of the American Mathematical Society*, 53:264–268, 1946.

[183] Xin Qi and Andrew C. Myers. Sharing classes between families. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 281–292, Dublin, Ireland, 2009. ACM Press.

[184] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–308, Charleston, SC, USA, 2006. ACM Press.

[185] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[186] Didier Rémy and Jérôme Vouillon. On the (un)reality of virtual types, 1998. http://gallium.inria.fr/~remy/work/virtual/virtual.ps.gz.

[187] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Paris, France, 1974. Springer-Verlag.

[188] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schumann, editor, *New Directions in Algorithmic Languages*. INRIA, 1975. Reprinted in [189].

[189] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 13–23. MIT Press, 1994. Originally published in [188].

[190] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, Copenhagen, Denmark, 2002. IEEE Computer Society Press.

[191] Winston W. Royce. Managing the development of large software systems: Concepts and techniques. In *International Conference on Software Engineering (ICSE)*, pages 328–338, Monterey, CA, USA, 1987. ACM Press.

[192] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 40–53, Paris, France, 1997. ACM Press.

[193] Chieri Saito and Atsushi Igarashi. Self type constructors. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 263–282, Orlando, FL, USA, 2009. ACM Press.

[194] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(3):285–331, 2008.

[195] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[196] Vijay Saraswat. Report on the programming language X10, version 2.0, 2009. `http://dist.codehaus.org/x10/documentation/languagespec/x10-200.pdf`.

[197] James Sasitorn and Robert Cartwright. Component NextGen: A sound and expressive component framework for Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 153–170, Montreal, QC, CA, 2007. ACM Press.

[198] K. Chandra Sekharaiah and D. Janaki Ram. Object schizophrenia problem in object role system design. In *International Conference on Object-Oriented Information Systems (OOIS)*, pages 494–506, Montpellier, France, 2002. Springer-Verlag.

[199] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Programming Language*. Addison-Wesley, 1997.

[200] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 73–84, Chicago, IL, USA, 2005. ACM Press.

[201] Jeremy G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, 2005.

[202] Jeremy G. Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[203] Charles Smith and Sophia Drossopoulou. Chai: Typed traits in java. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 543–576, Glasgow, Scotland, 2005. Springer-Verlag.

[204] Daniel Smith and Robert Cartwright. Java type inference is broken: Can we fix it? In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 505–524, Nashville, TN, USA, 2008. ACM Press.

[205] Guy Steele. *Common LISP: The Language*. Digital Press, 2nd edition, 1990.

[206] Alexander Stepanov and Meng Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1995.

[207] David Stoutamire and Stephen Omohundro. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, 1996.

[208] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: Core design and semantic definition. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 499–514, Montreal, QC, CA, 2007. ACM Press.

[209] Martin Sulzmann. Extracting programs from type class proofs. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 97–108, Venice, Italy, 2006. ACM Press.

[210] Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17(1):83–129, 2007.

[211] Sun Microsystems. The collections framework, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/collections/.

[212] Sun Microsystems. Java 2 platform standard edition 5.0 API specification, 2004. http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[213] Sun Microsystems. Enterprise Java Beans Specification 3.0, 2006. http://java.sun.com/products/ejb/docs.html.

[214] Sun Microsystems. JSR 277: Java module system, 2006. http://jcp.org/en/jsr/detail?id=277.

[215] Sun Microsystems. Java servlet specification, version 2.5, 2007. http://java.sun.com/products/servlet/.

[216] Sun Microsystems. JavaBeans API specification, version 1.01, 2007. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html.

[217] Sun Microsystems. Project Fortress website, 2008. http://projectfortress.sun.com/.

[218] Sun Microsystems. Java platform standard edition, 2009. http://java.sun.com/javase/.

[219] Clemens Szyperski. Independently extensible systems — Software engineering potential and challenges. In *Australasian Computer Science Conference (ACSC)*, Melbourne, Australia, 1996.

[220] Clemens Szyperski. *Component Software*. Addison-Wesley, 2nd edition, 2002.

[221] Clemens Szyperski, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of Sather. In *International Conference on Programming Languages and Systems Architecture*, volume 782 of *Lecture Notes in Computer Science*, pages 208–227, Zürich, Switzerland, March 1994. Springer-Verlag.

[222] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy, editors. *Ada 2005 Reference Manual. Language and Standard Libraries*, volume 4348 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[223] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE)*, pages 107–119, Los Angeles, CA, USA, 1999. ACM Press.

[224] Peter Thiemann. An embedded domain-specific language for type-safe server-side Web-scripting. *ACM Transactions on Internet Technology*, 5(1):1–46, 2005.

[225] Peter Thiemann and Stefan Wehr. Interface types for Haskell. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5356 of *Lecture Notes in Computer Science*, pages 256–272, Bangalore, India, 2008. Springer-Verlag.

[226] Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, Jyväskylä, Finland, 1997. Springer-Verlag.

[227] Mads Torgersen. The expression problem revisited — Four new solutions using generics. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 123–143, Oslo, Norway, 2004. Springer-Verlag.

[228] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, informal proceedings, 2005. http://homepages.inf.ed.ac.uk/wadler/fool/program/14.html.

[229] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, 2004. http://www.jot.fm/issues/issue_2004_12/article5/.

[230] Valery Trifonov and Scott Smith. Subtyping constrained types. In *International Symposium on Static Analysis (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365, Aachen, Germany, 1996. Springer-Verlag.

[231] V-Modell XT, version 1.3, 2009. http://www.v-modell-xt.de/.

[232] Mirko Viroli. On the recursive generation of parametric types. Technical Report DEIS-LIA-00-002, Università di Bologna, 2000.

[233] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 146–165, Minneapolis, MN, USA, 2000. ACM Press.

[234] W3C. XHTML 1.0, the extensible hypertext markup language (2nd edition), 2002. http://www.w3.org/TR/html/.

[235] Philip Wadler. The expression problem, 1998. Post to the Java Genericity mailing list.

[236] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 60–76, Austin, TX, USA, 1989. ACM Press.

[237] Alessandro Warth, Milan Stanojevic, and Todd Millstein. Statically scoped object adaptation with expanders. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 37–56, Portland, OR, USA, 2006. ACM Press.

[238] Stefan Wehr. Problem with superclass entailment in “A Static Semantics for Haskell”, 2005. Post to the Haskell mailinglist, http://www.haskell.org//pipermail/haskell/2005-October/016695.html.

[239] Stefan Wehr. JavaGI homepage, 2009. http://www.informatik.uni-freiburg.de/~wehr/javagi.

[240] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized interfaces for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372, Berlin, Germany, 2007. Springer-Verlag.

[241] Stefan Wehr and Peter Thiemann. Subtyping existential types. In *Workshop on Formal Techniques for Java-like Programs (FTfJP),* informal proceedings, pages 125–136, 2008. http://www-sop.inria.fr/everest/events/FTfJP08/ftfjp08.pdf.

[242] Stefan Wehr and Peter Thiemann. JavaGI in the battlefield: Practical experience with generalized interfaces. In *ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE)*, pages 65–74, Denver, CO, USA, 2009. ACM Press.

[243] Stefan Wehr and Peter Thiemann. On the decidability of subtyping with bounded existential types. In *Asian Symposium on Programming Languages and Systems (APLAS)*, Seoul, Korea, 2009.

[244] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[245] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .NET common language runtime. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 39–51, Venice, Italy, 2004. ACM Press.

[246] Matthias Zenger. Keris: Evolving software with extensible modules. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):333–362, 2005.

Index