JavaGI: Generalized Interfaces for Java

Stefan Wehr[1], Ralf Lämmel[2], and Peter Thiemann[1]

[1] Institut für Informatik, Universität Freiburg
{wehr,thiemann}@informatik.uni-freiburg.de
[2] Microsoft Corp., Redmond
ralf.lammel@microsoft.com

Abstract JavaGI is an experimental language that extends Java 1.5 by generalizing the interface concept to incorporate the essential features of Haskell's type classes. In particular, generalized interfaces cater for retroactive and constrained interface implementations, binary methods, static methods in interfaces, default implementations for interface methods, interfaces over families of types, and existential quantification for interface-bounded types. As a result, many anticipatory uses of design patterns such as Adapter, Factory, and Visitor become obsolete; several extension and integration problems can be solved more easily. JavaGI's interface capabilities interact with subtyping (and subclassing) in interesting ways that go beyond type classes. JavaGI can be translated to Java 1.5. Its formal type system is derived from Featherweight GJ.

1 Introduction

What are the distinguishing characteristics of Haskell compared to other programming languages? An informed answer will eventually mention type classes.

Type classes have been invented for dealing with overloading in functional programming languages in a non–ad-hoc manner [20, 37]. To the surprise of their inventors, type classes provide powerful means for solving various software-design problems. For instance, sufficiently powerful type classe systems address various software extension and integration problems [23]—in fact, a range of problems for which previously a whole array of techniques and programming language extensions has been proposed.

The observation that type classes and Java-style interfaces are related is not new [32, 23]. In this context, the following questions arise: (i) What part of type-class expressiveness corresponds to interfaces? (ii) What is the exact value of additional type-class expressiveness for an OO language? (iii) What is a viable OO language design with interfaces that cover most if not all type-class expressiveness? The paper answers these questions by proposing the design of JavaGI (Java with Generalized Interfaces) as a language extension of Java 1.5.

Type classes vs. interfaces Let us recall Haskell's type classes and relate them to OO interfaces. A type class is a named abstraction for the signatures of *member functions* that share one or more type parameters. Here is a type class Connection for database connections with a member exec to execute SQL commands:

```
class Connection conn where
  exec :: conn −> String −> IO QueryResult
  {− further members elided −}
```

The type parameter conn abstracts over the implementing type. An *instance definition* instantiates the type parameter(s) of a type class and provides specific implementations of the member functions. Here is an instance for PostgreSQL, assuming the existence of a type PostgreSQLConnection and a function pgsqlExec of type PostgreSQLConnection −> String −> IO QueryResult:

```
instance Connection PostgreSQLConnection where
  exec = pgsqlExec
```

This kind of abstraction is familiar to OO programmers. A Java programmer would create a Connection interface and provide different classes that implement the interface, *e.g.*, a class for PostgreSQL:

```
interface Connection {
  QueryResult exec(String command);
  /∗ further members elided ∗/
}
class PostgreSQLConnection implements Connection {
  public QueryResult exec(String command) { ... }
}
```

Type-class–bounded vs. interface polymorphism The Connection example indicates that interfaces and type classes have some common ground. But there are differences such as the mechanics of using type classes and interfaces when devising signatures for functions and methods. Here is a Haskell function using the Connection type class for inserting a customer:

```
newCustomer conn customer = do let command = ...
                               exec conn command
```

The use of a member functions from a type class (such as exec) abstracts over different implementations. An implementation (*i.e.*, instance) is chosen on the grounds of the types of the used members. The selection happens at compile time, if types are sufficiently known. Otherwise, it is deferred till run time. Let us consider the (inferred or declared) Haskell signature for newCustomer:

```
newCustomer :: Connection a => a −> Customer −> IO QueryResult
```

The *type variable* a serves as a placeholder for a connection type, which is restricted by the *constraint* Connection a to be an instance of Connection. ('=>' separates all constraints from the rest of the signature.) In contrast, a Java signature treats the interface Connection as a type. Thus:

```
class UseConnection {
  static QueryResult newCustomer(Connection conn, Customer customer) {
    String command = ...;
    return conn.exec(command);
  }
}
```

To summarize, polymorphism based on Haskell's type classes uses (the names of) type classes to form bounds on type variables in function signatures, whereas

polymorphism based on Java's interfaces (or .NET's interfaces for that matter) uses (the names of) interfaces as types, while type variables still serve their role as type parameters for OO generics.

This difference has several consequences that we discuss in the paper. For instance, type-class–bounded polymorphism naturally provides access to the "identity" of an implementing type, thereby enabling, among others, binary methods. (A binary method [3] is a method with more than one argument of the implementing type.) For example, Haskell's type class Eq declares a binary method, (==), for equality:

class Eq a **where** (==) :: a −> a −> Bool

The interfaces of Java 1.5 (or .NET) cannot directly express that the types of the two formal arguments of the equality function must be identical because the type implementing the corresponding interface cannot be referenced. A nontrivial extension of Java with self types [4] addresses the problem. For Java 1.5, there is an encoding that requires a rather complicated, generic interface with recursive bounds [1].

Once we commit to a style that makes the implementing type of an interface explicit, it is natural to consider multiple implementing types. Such a generalization of interfaces corresponds to multi-parameter type classes in Haskell [30]. Accordingly, an implementation of a "multi-parameter" interface is not tied to a specific receiver type but rather to a family of interacting types.

Contributions

1. We generalize Java 1.5 interfaces to unleash the full power of type classes in JavaGI, thereby enabling retroactive and constrained interface implementations, binary methods, static methods in interfaces, default implementations for interface methods, and interfaces over families of types.
2. We conservatively extend Java's interface concept. We clarify that interface-oriented programming is sufficient for various scenarios of software extension and integration. We substantiate that interface orientation is of crucial help in mastering self types, family polymorphism, and class extension.
3. We retrofit Java's interface types as bounded existential types, where the bound is the interface. In general, constraint-bounded existential types are more powerful than interface types. JavaGI's existentials are non-intrusive because they come with implicit pack and unpack operations.
4. We exploit interesting feature interactions between interface polymorphism and OO subclassing to demonstrate that JavaGI goes beyond a mere transposition of Haskell type classes to Java. Further, interfaces over multiple types require an original grouping mechanism "per receiver".

Outline Sec. 2 motivates generalized interfaces and describes JavaGI's language constructs through a series of canonical examples. Sec. 3 describes a simple translation of JavaGI back to Java 1.5. Sec. 4 develops the type system of Core–JavaGI as an extension of Featherweight GJ [17]. Related work is discussed in Sec. 5. Finally, Sec. 6 concludes the paper and gives pointers to future work.

2 JavaGI by Examples

This section introduces JavaGI through a series of examples addressing common OO programming problems that cannot be addressed in the same satisfactory manner with just Java 1.5. As JavaGI is a conservative extension of Java, JavaGI code refers to common classes and interfaces from the Java 1.5 API.

2.1 Example: Retroactive Interface Implementation

Recall the interface for database connections from the introductory section. Suppose we have to make existing code (such as newCustomer from class UseConnection) work with a MySQL database where a library class MySQLConnection provides the desired functionality, although under a different name:

```
class MySQLConnection { QueryResult execCommand(String command) { ... } }
```

The library author was not aware of the Connection interface, and hence did not implement the interface for the MySQLConnection class. In Java, we cannot retroactively add such an implementation. Hence, we need to employ the Adapter pattern: a designated adapter class wraps a MySQLConnection object and implements Connection by delegating to MySQLConnection. This approach is tedious and suffers from problems like object schizophrenia [31].

Inspired by Haskell, JavaGI supports *retroactive interface implementation* such that the implementation of an interface does no longer need to be coupled with the implementing class. Here is the *implementation definition* for the Connection interface with the MySQLConnection class acting as the *implementing type* (enclosed in square brackets '[...]'):

```
implementation Connection [MySQLConnection] {
  QueryResult exec(String command) { return this.execCommand(command); }
}
```

In the body of the method exec, **this** has static type MySQLConnection and refers to the receiver of the method call. Thanks to the implementation definition just given, the newCustomer method can now use a MySQLConnection:

```
MySQLConnection conn = ...;
QueryResult result = UseConnection.newCustomer(conn, someCustomer);
```

2.2 Example: Preserved Dynamic Dispatch

Methods of a Java 1.5 interface are virtual, *i.e.*, subject to dynamic dispatch. JavaGI preserves this capability for methods of retroactive interface implementations. This expressiveness implies extensibility in the operation dimension so that we can solve the expression problem [36]. Compared to existing solutions in Java 1.5 (or C# 2.0) [33], JavaGI's solution is simple and more perspicuous.

Consider a class hierarchy for binary trees with strings at the leaves:

```
abstract class BTree {}
class Leaf extends BTree { String information; }
class Node extends BTree { BTree left, right; }
```

Now suppose the classes for binary trees are in a compiled package, but we want to implement a count method on trees that returns the number of inner nodes. As we cannot add count to the classes, we introduce an interface Count with the count method and implement the interface for the tree hierarchy.

```
interface Count { int count(); }
implementation Count [BTree] { int count() { return 0; } } // works also for Leaf
implementation Count [Node] {
   int count() { return this.left.count() + this.right.count() + 1; }
}
class CountTest { int doCount(BTree t) { return t.count(); } }
```

In a recursive invocation of count in the implementation for Node, the static type of the receiver is BTree. Without dynamic dispatch, the recursive calls would count 0. Fortunately, JavaGI supports dynamic dispatch for retroactive interface implementations, so the recursive invocations return indeed the number of inner nodes of the subtrees.

The default implementation for BTree is required because retroactively added methods must not be abstract. See Sec. 3 for an explanation and further restrictions on the distribution of implementations over different compilation units.

Adding new operations and new data variants is straightforward. For a new operation, an interface and the corresponding implementations suffice.

```
// A new operation that collects the string information stored in the tree.
interface Collect { void collect(List<String> l); }
implementation Collect [BTree] { void collect(List<String> l) { return; } }
implementation Collect [Node] {
   void collect(List<String> l) { this.left.collect(l); this.right.collect(l); }
}
implementation Collect [Leaf] { void collect(List<String> l) { l.add(this.information); } }
```

A new data variant corresponds to a new subclass of BTree and interface implementations for existing operations, unless the default for the base class is acceptable.

```
// A new data variant that stores information in inner nodes.
class InformationNode extends Node {
   String information;
   void collect(List<String> l) { this.left.collect(l); l.add(this.information); this.right.collect(l); }
}
implementation Collect [InformationNode]
// The implementation of Count for Node also works for InformationNode.
```

JavaGI is amenable to another solution to the expression problem, which requires slightly more encoding effort. That is, we can transpose a Haskell-based recipe to JavaGI exploiting its regular type-class–like power [23], without taking any dependency on subtyping (and virtual methods): (i) designate an interface instead of a class as the root of all data variants; (ii) define data variants as generic classes that implement the root interface and are parameterized by the types of the immediate subcomponents; (iii) define subinterfaces of the root interface for the operations; (iv) provide implementations for all the data variants. This recipe does not require default implementations for the root of the hierarchy, and it does not put restrictions on the distribution over compilation units.

5

2.3 Example: Binary Methods

Java 1.5 defines a generic interface for comparing values:

```
interface Comparable<X> { int compareTo(X that); }
```

If we wanted to ensure that the (formal) argument type coincides with the type implementing the interface, then the above signature is too permissive. We would need to define compareTo as a *binary method* [3]. In Java 1.5, we can still constrain uses of the permissive signature of compareTo by a generic type with a recursive bound. For instance, consider a generic method max that computes the maximum of two objects using Comparable [2]:

```
<X extends Comparable<X>> X max(X x1, X x2) {
  if (x1.compareTo(x2) > 0) return x1;
    else return x2;
}
```

The recursive type bound X **extends** Comparable<X> expresses the intuition that the argument type of compareTo and the type implementing Comparable are the same. Any class C that is to be used with max must implement Comparable<C>.

In contrast, JavaGI supports binary methods (in interfaces) and enables the programmer to define the less permissive signature for compareTo directly:

```
interface MyComparable { int compareTo(This that); }
```

The type variable **This** (*cf.* compareTo's argument) is implicitly bound by the interface. It denotes the type implementing the interface. The type of compareTo results in a simpler and more comprehensible signature for the maximum method:

```
<X> X myMax(X x1, X x2) where X implements MyComparable {
  if (x1.compareTo(x2) > 0) return x1;
    else return x2;
}
```

The switch to a constraint-based notation **where** X **implements** MyComparable is reminiscent of .NET generics [21, 39]. To type check an implementation, **This** is replaced with the implementing type. Here is an implementation for Integer:

```
implementation MyComparable [Integer]
```

This implementation need not provide code for compareTo because Integer already has a method int compareTo(Integer that). Assuming type inference for generalized interfaces, we can call myMax as follows:

```
Integer i = myMax(new Integer(1), new Integer(2));
```

Let us bring subtyping into play. For instance, consider an implementation of MyComparable for Number, which is the (abstract) superclass of Integer.

```
implementation MyComparable [Number] {
  int compareTo(Number that) { /* Convert to double values and compare */ }
}
```

Suppose that x and y are of static type Number, so the call x.compareTo(y) is valid. Which version of compareTo should be invoked when both x and y have dynamic

type Integer? JavaGI takes an approach similar to multimethods [10] and selects
the most specific implementation dynamically, thereby generalizing the concept
of virtual method calls. Hence, the compareTo method of the implementation for
Integer is invoked. In all other cases, the compareTo version for Number is chosen
(assuming that there are no other implementations for subclasses of Number).

2.4 Example: Constrained Interface Implementations

If the elements of two given lists are comparable, then we expect the lists them-
selves to be comparable. JavaGI can express this implication with a constrained
interface implementation.

```
implementation<X> MyComparable [LinkedList<X>] where X implements MyComparable {
   int compareTo(LinkedList<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();
         int i = thisX.compareTo(thatX); // type checks because X implements MyComparable
         if (i != 0) return i;
      }
      if (thisIt.hasNext() && !thatIt.hasNext()) return 1;
      if (thatIt.hasNext() && !thisIt.hasNext()) return −1;
      return 0;
   }
}
```

If now x and y have type LinkedList<Integer>, then the call myMax(x, y) is valid.

   The implementation of MyComparable for LinkedList<X> is parameterized over
X, the type of list elements. The constraint X **implements** MyComparable of the
implementation makes the compareTo operation available on objects of type X
and ensures that only lists with comparable elements implement MyComparable.

   There is no satisfactory solution to the problem of constrained interface im-
plementations in Java 1.5. Here are two suboptimal solutions. (i) Implement
Comparable<LinkedList<X>> directly in class LinkedList<X>. But then we either
could no longer assemble lists with incomparable elements (if X has bound
Comparable<X>), or we would need run-time casts for the comparison of ele-
ments (if X is unbounded). (ii) Plan ahead and use a designated class, CmpList,
for lists of comparable elements.

```
class CmpList<X extends Comparable<X>> implements Comparable<CmpList<X>> { ... }
```

But this is another instance of the Adapter pattern with its well-known short-
comings. In addition, this technique results in a prohibitive proliferation of helper
classes such as CmpList because other interfaces than Comparable may exist.

2.5 Example: Static Interface Members

Many classes need to implement parsing such that instances can be constructed
from an external representation. (Likewise, in the XML domain, XML data needs
to be de-serialized.) Hence, we would like to define an interface of parseable types
with a parse method. However, parse cannot be defined as an instance method

because it behaves like an additional class constructor. To cater for this need, JavaGI (but not Java 1.5) admits static methods in interfaces:

```
interface Parseable { static This parse(String s); }
```

Again, **This** (in the result position) refers to the implementing type. For example, consider a generic method to process an entry in a web form (*cf.* class Form) using the method String getParameter(String name) for accessing form parameters:

```
class ParseableTest {
    <X> X processEntry(Form f, String pname) where X implements Parseable {
        String s = f.getParameter(pname);
        return Parseable[X].parse(s);
    }
    Integer parseMyParam(Form f) { return processEntry<Integer>(f, "integer parameter"); }
}
```

The expression Parseable[X].parse(s) invokes the static method parse of interface Parseable with X as the implementing type, indicated by square brackets '[...]'. The parseMyParam method requires a Parseable implementation for integers:

```
implementation Parseable [Integer] {
    static Integer parse(String s) { return new Integer(s); }
}
```

In Java 1.5, we would implement this functionality with the Factory pattern. The Java solution is more complicated than the solution in JavaGI because boilerplate code for the factory class needs to be written and an additional factory object must be passed around explicitly.

2.6 Example: Multi-Headed Interfaces

Traditional subtype polymorphism is insufficient to abstract over relations between conglomerations of objects and their methods. *Family polymorphism* [12] has been proposed as a corresponding generalization. It turns out that interfaces can be generalized in a related manner.

Consider the Observer pattern. There are two participating types: 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. The challenge in modeling this pattern in a reusable and type-safe way is the mutual dependency of subject and observer. That is, the subject has a register method which takes an observer as an argument, while the observer in turn has an update method which takes a subject as an argument.

JavaGI provides a suitable abstraction: *multi-headed interfaces*. While a classic OO interface concerns a single type, a multi-headed interface relates multiple implementing types and their methods. Such an interface can place mutual requirements on the methods of all participating types. The following multi-headed interface captures the Observer pattern:

```
interface ObserverPattern [Subject, Observer] {
    receiver Subject {
        List<Observer> getObservers();
```

```
        void register(Observer o) { getObservers().add(o); }
        void notify() { for (Observer o : getObservers()) o.update(this); }
    }
    receiver Observer { void update(Subject s); }
}
```

With multiple implementing types, we can no longer use the implicitly bound
type variable **This**. Instead, we have to name the implementing types explicitly
through type variables Subject and Observer. Furthermore, the interface groups
methods by receiver type because there is no obvious default.

The example illustrates that generalized interfaces may contain default im-
plementations for methods, which are inherited by all implementations that do
not override them. The default implementations for register and notify rely on
the list of observers returned by getObservers to store and retrieve registered ob-
servers. (Default implementations in interfaces weaken the distinction between
interface and implementation. They are not essential to JavaGI's design, but they
proved useful in Haskell.)

Here are two classes to participate in the Observer pattern:

```
class Model { // designated subject class
    private List<Display> observers = new ArrayList<Display>();
    List<Display> getObservers() { return observers; }
}
class Display { } // designated observer class
```

An implementation of ObserverPattern only needs to define update:

```
implementation ObserverPattern [Model, Display] {
    receiver Display { void update (Model m) { System.out.println("model has changed"); } }
}
```

All other methods required by the interface are either implemented by the par-
ticipating classes or inherited from the interface definition.

The genericUpdate method of the following test class uses the constraint
[S,O] **implements** ObserverPattern to specify that the type parameters S and O
must together implement the ObserverPattern interface.

```
class MultiheadedTest {
    <S,O> void genericUpdate(S subject, O observer) where [S,O] implements ObserverPattern {
        observer.update(subject);
    }
    void callGenericUpdate() { genericUpdate(new Model(), new Display()); }
}
```

The Observer pattern can also be implemented in Java 1.5 using generics
for the subject and observer roles with complex, mutually referring bounds. In
fact, the subject part must be encoded as a generic class (as opposed to a generic
interface) to provide room for the default methods of subjects. A concrete subject
class must then extend the generic subject class, which has to be planned ahead
and is even impossible if the concrete subject class requires another superclass.

The notation for single-headed interfaces used so far is just syntactic sugar.
For example, JavaGI's MyComparable interface (Sec. 2.3) is fully spelled out as:

```
interface MyComparable [This] { receiver This { int compareTo(This that); } }
```

2.7 Example: Bounded Existential Types

In the preceding examples, we have used interfaces such as Connection (Sec. 2.1) and List (Sec. 2.2) as if they were types. This view aligns well with Java 1.5. But multi-headed interfaces, introduced in the preceding section, do not fit this scheme. For instance, simply using ObserverPattern as a type does not make sense.

To this end, JavaGI supports *bounded existential types* in full generality. Take **exists** X **where** [Model,X] **implements** ObserverPattern . X as an example. This bounded existential type (*existential* for short) comprises objects that acts as an observer for class Model. Here is an example that calls the update method on such objects:

```
class ExistentialTest {
  void updateObserver((exists X where [Model,X] implements ObserverPattern . X) observer) {
    observer.update(new Model()); /* implicit unpacking */
  }
  void callUpdateObserver() { updateObserver(new Display()); /* implicit conversion */ }
}
```

The example also demonstrates that existential values are implicitly unpacked (*e.g.*, the update method is invoked directly on an object of existential type), and that objects are implicitly converted into an existential value (*e.g.*, an object of type Display is used directly as an argument to updateObserver).

This treatment of bounded existential types generalizes the requirement for backwards compatibility with Java interface types, which are only syntactic sugar for existentials. For instance, the type Connection is expanded into **exists** X **where** X **implements** Connection . X, whereas type List<Observer> is an abbreviation for **exists** X **where** X **implements** List<Observer> . X.

Support for multi-headed interfaces is not the only good reason to have bounded existential types. They have other advantages over Java interface types:

- They allow the general composition of interface types. For example, the type **exists** X **where** X **implements** Count **and** X **implements** Connection . X is the intersection of types that implement both the Count and Connection interfaces. Java 1.5 can denote such types only in the bound of a generic type variable as in X **extends** Count & Connection.
- They encompass Java wildcards [35]. Consider List<? **extends** Connection>, a Java 1.5 type comprising all values of type List<X> where X **extends** Connection. (This type is different from the fully heterogeneous type List<Connection>, where each list element may have a different type.) In JavaGI, this type is denoted as **exists** X **where** X **implements** Connection . List<X>. Torgersen, Ernst, and Hansen [34] investigate the relation between wildcards and existentials.

3 Translation to Java

This section sketches a translation from JavaGI to Java 1.5, which follows the scheme for translating Haskell type classes to System F [15]. We first demonstrate the general idea (Sec. 3.1), then explain the encoding of retroactively defined methods (Sec. 3.2) and of multi-headed interfaces (Sec. 3.3), show how existentials are translated (Sec. 3.4), and finally discuss interoperability with Java (Sec. 3.5).

3.1 Translating the Basics

We first explain the general idea of the translation under the simplifying assumption that the implementing types of an implementation definition provide all methods required, so that the definition itself contains only static methods.[1]

An interface $I\langle\overline{X}\rangle[\overline{Y}]$ is translated into a *dictionary interface* $I^{\mathrm{dict}}\langle\overline{X},\overline{Y}\rangle$. The interface I^{dict} supports the same method names as I, but static methods of I are mapped to non-static methods and I's non-static methods for implementing type Y_i get a new first argument this\$ of type Y_i.

As an example, consider the translation of the interfaces MyComparable and Parseable (Sec. 2.3 and 2.5).

```
interface MyComparableDict<This> { int compareTo(This this$, This that); }
interface ParseableDict<This> { This parse(String s); }
```

A constraint $[\overline{T}]$ **implements** $I\langle\overline{U}\rangle$ in a class or method signature is translated into an additional constructor or method argument, respectively, that has type $I^{\mathrm{dict}}\langle\overline{T},\overline{U}\rangle$. The additional argument allows the class or method to access a *dictionary object* (*i.e.*, an instance of the dictionary class) that provides all methods available through the constraint. In case of a class constraint, the dictionary object is stored in an instance variable.

For example, here is the translation of myMax and processEntry (Sec. 2.3 and 2.5):

```
<X> X myMax(X x1, X x2, MyComparableDict<X> dict) {
  if (dict.compareTo(x1, x2) > 0) return x1; else return x2;
}
<X> X processEntry(Form f, String pname, ParseableDict<X> dict) {
  String s = f.getParameter(pname); return dict.parse(s);
}
```

A definition **implementation**$\langle\overline{X}\rangle\, I\langle\overline{T}\rangle[\overline{U}]\,\{\dots\}$ (ignoring constrained implementations for a moment) is translated into a *dictionary class* $C^{\mathrm{dict},\overline{U}}\langle\overline{X}\rangle$ that implements $I^{\mathrm{dict}}\langle\overline{T},\overline{U}\rangle$. Methods of I^{dict} corresponding to static methods in the original interface are implemented by translating their bodies (discussed shortly). The remaining methods of I^{dict} are implemented by delegating the call to the corresponding implementing type, which is available through the argument this\$.

The next example shows the translation of the implementation definitions for MyComparable and Parseable with implementing type Integer (Sec. 2.3 and 2.5).

```
class MyComparableDict_Integer implements MyComparableDict<Integer> {
  public int compareTo(Integer this$, Integer that) {
    return this$.compareTo(that); // Integer has a compareTo method
  }
}
class ParseableDict_Integer implements ParseableDict<Integer> {
  public Integer parse(String s) { return new Integer(s); }
}
```

---

[1] The following conventions apply: I ranges over interface names; I^{dict} and $C^{\mathrm{dict},\overline{U}}$ represent fresh interface and class names, respectively; X and Y range over type variables; T and U range over types; overbar notation denotes sequencing (*e.g.*, $\overline{T}$ denotes $T_1,\dots,T_n$).

A constrained implementation is translated similarly, with every constraint $[\overline{T}]$ **implements** $I\langle\overline{U}\rangle$ giving rise to an extra constructor argument and an instance variable of type $I^{\mathrm{dict}}\langle\overline{T},\overline{U}\rangle$ for the dictionary class. These dictionary objects serve the same purpose as the additional arguments introduced for constraints in class or method signatures.

The translation of the implementation of MyComparable for LinkedList (Sec.2.4) looks as follows:

```
class MyComparableDict_LinkedList<X> implements MyComparableDict<LinkedList<X>> {
  private MyComparableDict<X> dict;
  MyComparableDict_LinkedList(MyComparableDict<X> dict) { this.dict = dict; }
  public int compareTo(LinkedList<X> this$, LinkedList<X> that) {
    /* ... */ X thisX = thisIt.next(); X thatX = thatIt.next();
             int i = this.dict.compareTo(thisX, thatX); /* ... */
  }
}
```

The translation of statements and expressions considers every instantiation of a class and every invocation of a method. If the class or method signature contains constraints, the translation supplies appropriate dictionary arguments.

Here are translations of sample invocations of myMax and of parseMyParam (Sec. 2.3, 2.4, and 2.5).[2]

```
void invokeMyMax(LinkedList<Integer> x, LinkedList<Integer> y) {
  Integer j = myMax(new Integer(1), new Integer(2), new MyComparableDict_Integer());
  LinkedList<Integer> z = myMax(x, y, new MyComparableDict_LinkedList<Integer>(
                                       new MyComparableDict_Integer()));
}
Integer parseMyParam(Form f) {
  return processEntry(f, "integer parameter", new ParseableDict_Integer());
}
```

3.2 Translating Retroactively Defined Methods

The preservation of dynamic dispatch is the main complication in the translation of retroactively defined methods (*e.g.*, non-static methods of retroactive interface implementations). For example, consider the implementations of Count for BTree and its subclass Node from Sec. 2.2. The translation creates a dictionary interface CountDict and two dictionary classes CountDict_BTree and CountDict_Node. Because count is a retroactively defined method, its invocations in the implementation for Node must be translated such that an instance of CountDict_BTree acts as the receiver. But this means that the definition of count in CountDict_BTree must take care of dynamically dispatching to the correct target code.

MultiJava's strategy for implementing external method families [11] solves the problem. The dictionary CountDict_BTree uses the **instanceof** operator to dispatch on this$, which represents the receiver of the call in the untranslated code. If this$ is an instance of Node, the call is delegated to an instance of CountDict_Node.

---

[2] Repeated allocations of dictionary objects can be avoided by using a caching mechanism. For simplicity, we omit this optimization in this section.

```
// Dictionary interface corresponding to JavaGI's Count interface
interface CountDict<This> { int count(This this$); }
// Implementations of the dictionary interface
class CountDict_BTree implements CountDict<BTree> {
  public int count(BTree this$) {
    if (this$ instanceof Node) return new CountDict_Node().count( (Node) this$);
    else return 0;
  }
}
class CountDict_Node implements CountDict<Node> {
  public int count(Node this$) {
    return new CountDict_BTree().count(this$.left) +
           new CountDict_BTree().count(this$.right) + 1;
  }
}
```

Fig. 1: Translation of interface Count and its implementations (Sec. 2.2) to Java 1.5.

Otherwise, the code for BTree is used. If there are further arguments of the implementing type, then they must be included in the dispatch.

Fig. 1 contains an application of this strategy. For simplicity, the code ignores the possibility that some subclass of Node or Leaf overrides count internally. The MultiJava paper explains how this situation is handled.

To allow for modular compilation, we impose two restrictions. (i) Retroactively defined methods must not be abstract. (ii) If an implementation of interface I in compilation unit U retroactively adds a method to class C, then U must contain either C's definition or any implementation of I for a superclass of C.

The first restriction corresponds to MultiJava's restriction R2 [11, p. 542]. It ensures that there is a default case for the **instanceof** tests on this$ (**return** 0; in Fig. 1). The second restriction guarantees that all possible branches for the **instanceof** tests can be collected in a modular way. It corresponds to MultiJava's restriction R3 [11, p. 543].

3.3   Translating Multi-headed Interfaces

Fig. 2 shows the translation of the multi-headed interface ObserverPattern (Sec. 2.6). The dictionary interface ObserverPatternDict has two type parameters, one for each implementing type. Grouping by receiver type, in JavaGI achieved through the **receiver** keyword, translates to different types for the first argument this$.

The abstract class ObserverPatternDef contains those methods for which default implementations are available. A class such as ObserverPattern_ModelDisplay, which implements the dictionary interface, inherits from the abstract class to avoid code duplication and to allow redefinition of default methods.

3.4   Translating Bounded Existential Types

The translation of a bounded existential type is a wrapper class that stores a witness object and the dictionaries for the constraints in instance variables. The

```
// Dictionary interface corresponding to JavaGI's ObserverPattern interface
// (without default methods)
interface ObserverPatternDict<Subject,Observer> {
  // methods with receiver type Subject
  List<Observer> getObservers(Subject this$);
  void register(Subject this$, Observer o);
  void notify(Subject this$);
  // methods with receiver type Observer
  void update(Observer this$, Subject s);
}
// Abstract class holding the default methods of the ObserverPattern interface
abstract class ObserverPatternDef<Subject,Observer>
                implements ObserverPatternDict<Subject,Observer> {
  public void register(Subject this$, Observer o) { getObservers(this$).add(o); }
  public void notify(Subject this$) { for (Observer o : getObservers(this$)) update(o, this$); }
}
// Implementation of the dictionary interface
class ObserverPatternDict_ModelDisplay extends ObserverPatternDef<Model, Display> {
  public List<Display> getObservers(Model this$) { return this$.getObservers(); }
  public void update(Display this$, Model m) { System.out.println("model has changed"); }
}
// Test code
class MultiheadedTest {
  <S,O> void genericUpdate(S subject, O observer, ObserverPatternDict<S,O> dict) {
    dict.update(observer, subject);
  }
  void callGenericUpdate() {
    genericUpdate(new Model(), new Display(), new ObserverPatternDict_ModelDisplay());
  }
}
```

Fig. 2: Translation of the multi-headed interface ObserverPattern, its implementation, and its use (Sec. 2.6) to Java 1.5.

types of these instance variables are obtained by translating the original witness type and the constraints from JavaGI to Java, replacing every existentially quantified type variable with Object.

Fig. 3 shows the class Exists1 resulting from translating the JavaGI type **exists X where X implements** Count . X. The class uses a static, generic create method to return new Exists1 instances for a witness of type X and a CountDict<X> dictionary, because a constructor cannot be generic unless the class is generic.

A method call on an existential value is translated to Java by invoking the corresponding method on one of the existential's dictionaries, passing the witness as first argument. For example, if x has type **exists X where X implements** Count . X in JavaGI, then x has type Exists1 in Java, and the call x.count() is translated into x.dict.count(x.witness).

3.5 Interoperability with Java

The main source of incompatibility between JavaGI and Java is that Java uses interface names as types, whereas JavaGI uses them only in constraints (if we

```
class Exists1 {
  // Type Object stands for the existentially quantified type variable
  Object witness; CountDict<Object> dict;
  private Exists1(Object witness, CountDict<Object> dict) {
    this.witness = witness; this.dict = dict;
  }
  // "type−safe" constructor method
  static <X> Exists1 create(X witness, CountDict<X> dict) {
    return new Exists1(witness, (CountDict) dict); // CountDict is a raw type
  }
}
```

Fig. 3: Java 1.5 class corresponding to the bounded existential type **exists** X **where** X **implements** Count . X in JavaGI.

ignore JavaGI's syntactic sugar). As discussed in Sec. 2.7, a Java interface type I is interpreted as the existential **exists** X **where** X **implements** I . X. We face two problems: (i) it must be possible to pass objects of this type from JavaGI to a Java method expecting arguments of type I; (ii) JavaGI must be prepared to invoke methods of I on objects coming from the Java world, *i.e.*, objects that are not instances of the existential's wrapper class but implement I in the Java-sense.

How are these problems solved? Let $T = $ **exists** $\overline{X}$ **where** $\overline{Q}$. Y where $\overline{Q}$ is a sequence of constraints and class C be the translation of T. To solve the first problem, C implements each Java interface $I\langle\overline{U}\rangle$ that appears in $\overline{Q}$ as Y **implements** $I\langle\overline{U}\rangle$. To solve the second problem, we adjust the translation for method calls on objects of existential type. If a method declared in a Java interface is called on an object of type T (assuming a suitable constraint in $\overline{Q}$), the modified translation leaves the call unchanged because the object may not be an instance of C. The translated call is also valid for C because C implements the Java interface. In all other cases, the translation treats the call as in Sec. 3.4.

Wrapping objects is problematic because operations involving object identity (*e.g.*, ==) and type tests (*e.g.*, **instanceof**) in Java code may no longer work as expected. (Translating such operations away does not work because they may be contained in some external Java library not under our control.) It is, however, somewhat unavoidable if retroactive interface implementation is to be supported and changing the JVM is not an option. (For example, the translation of expanders [38] to Java suffers from the same problem.) Eugster's uniform proxies [14] might solve the problem for type tests because they would allow the wrapper to be a subclass of the run-time class of the witness and to implement all interfaces the witness implements through a regular Java **implements** clause.

4 A Formal Type System for JavaGI

This section formalizes a type system for the language Core–JavaGI, which captures the main ingredients of JavaGI and supports all essential features presented in Sec. 2. The static semantics of Core–JavaGI is based on Featherweight GJ (FGJ [17]) and on Wild FJ [34].

$$
\begin{array}{rcl}
prog & ::= & \overline{def}\ e \\
def & ::= & cdef \mid idef \mid impl \\
cdef & ::= & \textbf{class } C\langle \overline{X}\rangle \textbf{ extends } N \textbf{ where } \overline{Q}\,\{\,\overline{T\,f}\ \overline{m:mdef}\,\} \\
idef & ::= & \textbf{interface } I\langle \overline{X}\rangle\,[\overline{X}]\textbf{ where } \overline{Q}\,\{\,\overline{m:\textbf{static } msig}\ \overline{itsig}\,\} \\
impl & ::= & \textbf{implementation}\langle \overline{X}\rangle\ K\ [\overline{T}\,]\textbf{ where } \overline{Q}\,\{\,\overline{m:\textbf{static } mdef}\ \overline{itdef}\,\} \\
itsig & ::= & \textbf{receiver }\{\overline{m:msig}\} \qquad\qquad S,T,U,V ::= X \mid N \mid \exists \overline{X}\textbf{ where }\overline{Q}\,.\,T \\
itdef & ::= & \textbf{receiver }\{\overline{m:mdef}\} \qquad\qquad N \qquad ::= C\langle \overline{T}\rangle \mid Object \\
msig & ::= & \langle \overline{X}\rangle\,\overline{T\,x} \to T\textbf{ where }\overline{Q} \qquad\qquad K \qquad ::= I\langle \overline{T}\rangle \\
mdef & ::= & msig\,\{e\} \qquad\qquad\qquad\qquad\quad P,Q \qquad ::= \overline{T}\,\textbf{implements } K \\
e & ::= & x \mid e.f \mid e.m\langle \overline{T}\rangle(\overline{e}) \mid K[\overline{T}\,].m\langle \overline{T}\rangle(\overline{e}) \mid \textbf{new } N(\overline{e}) \mid (N)\,e
\end{array}
$$

$$X,Y,Z \in TyvarName \quad C,D \in ClassName \quad I,J \in IfaceName$$
$$m \in MethodName \qquad f \in FieldName \qquad x \in VarName$$

Fig. 4: Syntax of Core–JavaGI.

4.1 Syntax

Fig. 4 shows the syntax of Core–JavaGI.[3] A program *prog* consists of a sequence of class definitions *cdef*, interface definitions *idef*, implementation definitions *impl*, and a "main" expression *e*. We omit constructors from class definitions, and assume that every field name is defined at most once. Methods of classes are written as $m:mdef$ where *mdef* is a method signature *msig* with a method body. An interface definition contains static method signatures $m:\textbf{static } msig$ and signatures *itsig* for methods supported by particular implementing types. Implementation definitions provide the corresponding implementations $m:\textbf{static } mdef$ and *itdef*.

Core–JavaGI does not support default methods in interface definitions. Moreover, signatures *itsig* and definitions *itdef* refer to their implementing type by position. We assume that the name of a method defined in some interface is unique across method definitions in classes and other interfaces. Interface implementations must provide explicit definitions for all methods of all implementing types. It is straightforward but tedious to lift these restrictions.

A method signature *msig* has the form $\langle \overline{X}\rangle\,\overline{T\,x} \to T\textbf{ where }\overline{Q}$ where $\overline{T}$ are the argument types and T is the result type. Types T in Core–JavaGI are either type variables X, class types N, or bounded existential types $\exists \overline{X}\textbf{ where }\overline{Q}\,.\,T$. The latter are considered equivalent up to renaming of bound type variables, reordering of type variables and constraints, addition and removal of unused type variables and constraints, and merging of variable-disjoint, adjacent existential quantifiers. Finally, $\exists\,\cdot\,\textbf{ where }\cdot\,.\,T$ is equivalent to T, so that every type can be written as an existential $\exists \overline{X}\textbf{ where }\overline{Q}\,.\,T$ where T is a type variable or class.

K abbreviates an instantiated interface $I\langle \overline{T}\rangle$ and P,Q range over constraints. Core–JavaGI does not support class bounds in constraints.

---

[3] The notation $\overline{\xi}^{\,n}$ (or $\overline{\xi}$ for short) denotes the sequence $\xi_1,\dots,\xi_n$ for some syntactic construct ξ, $\cdot$ denotes the empty sequence. At some points, we interpret $\overline{\xi}$ as the set $\{\xi_1,\dots,\xi_n\}$. We assume that the identifier sets *TyvarName*, *ClassName*, *IfaceName*, *MethodName*, *FieldName*, and *VarName* are countably infinite and pairwise disjoint.

$$\boxed{\Theta; \Delta \vdash T \text{ ok} \qquad \Theta; \Delta \vdash K[\overline{T}] \text{ ok} \qquad \Theta; \Delta \vdash Q \text{ ok} \qquad \Theta; \Delta \vdash [\overline{T/X}] \text{ ok under } \overline{Q}}$$

$$\frac{X \in \Delta}{\Theta; \Delta \vdash X \text{ ok}} \qquad \Theta; \Delta \vdash \textit{Object} \text{ ok} \qquad \frac{\Theta; \Delta, \overline{X}, \overline{Q} \vdash \overline{Q}, T \text{ ok}}{\Theta; \Delta \vdash \exists \overline{X} \textbf{ where } \overline{Q} . T \text{ ok}}$$

$$\frac{\textbf{class } C\langle \overline{X}\rangle \textbf{ extends } N \textbf{ where } \overline{Q} \{\dots\} \in \Theta}{\Theta; \Delta \vdash [\overline{T/X}] \text{ ok under } \overline{Q}} \qquad \frac{\textbf{interface } I\langle \overline{X}\rangle [\overline{Y}] \textbf{ where } \overline{Q} \{\dots\} \in \Theta}{\Theta; \Delta \vdash [\overline{S/X}, \overline{T/Y}] \text{ ok under } \overline{Q}}$$

$$\frac{}{\Theta; \Delta \vdash C\langle \overline{T}\rangle \text{ ok}} \qquad \frac{}{\Theta; \Delta \vdash I\langle \overline{S}\rangle[\overline{T}] \text{ ok}}$$

$$\frac{\Theta; \Delta \vdash K[\overline{T}] \text{ ok}}{\Theta; \Delta \vdash \overline{T} \textbf{ implements } K \text{ ok}} \qquad \frac{(\forall i)\ \Theta; \Delta \Vdash [\overline{T/X}]Q_i \qquad \Theta; \Delta \vdash \overline{T} \text{ ok}}{\Theta; \Delta \vdash [\overline{T/X}] \text{ ok under } \overline{Q}}$$

Fig. 5: Well-formedness.

$$\boxed{\Theta; \Delta \Vdash Q \qquad \Theta; \Delta \vdash T \leq T}$$

$$\frac{Q \in \Delta}{\Theta; \Delta \Vdash Q} \qquad \frac{\forall \overline{X} . \overline{Q} \Rightarrow P \in \Theta \qquad (\forall i)\ \Theta; \Delta \vdash [\overline{T/X}] \text{ ok under } Q_i}{\Theta; \Delta \Vdash [\overline{T/X}]P}$$

$$\Theta; \Delta \vdash T \leq T \qquad \Theta; \Delta \vdash T \leq \textit{Object} \qquad \frac{\Theta; \Delta \vdash S \leq T \qquad \Theta; \Delta \vdash T \leq U}{\Theta; \Delta \vdash S \leq U}$$

$$\frac{\textbf{class } C\langle \overline{X}\rangle \textbf{ extends } N \dots \in \Theta}{\Theta; \Delta \vdash C\langle \overline{T}\rangle \leq [\overline{T/X}]N} \qquad \frac{\Theta; \Delta, \overline{X}, \overline{Q} \vdash T \leq U \qquad \Theta; \Delta \vdash U \text{ ok}}{\Theta; \Delta \vdash \exists \overline{X} \textbf{ where } \overline{Q} . T \leq U}$$

$$\frac{(\forall i)\ \Theta; \Delta \Vdash [\overline{U/X}]Q_i \qquad \Theta; \Delta \vdash \exists \overline{X} \textbf{ where } \overline{Q} . T \text{ ok}}{\Theta; \Delta \vdash [\overline{U/X}]T \leq \exists \overline{X} \textbf{ where } \overline{Q} . T}$$

Fig. 6: Entailment and subtyping.

Core–JavaGI expressions e are very similar to FGJ expressions. The new expression form $K[\overline{T}].m(\overline{e})$ invokes static interface methods. The types $\overline{T}$ select the implementation of method m. The target type of a cast must be a class type N, so that constraints need not be checked at runtime.[4]

4.2 Typing Judgments

The typing judgments of Core–JavaGI make use of three different environments. A program environment Θ is a set of program definitions *def* and constraint schemes of the form $\forall \overline{X} . \overline{Q} \Rightarrow P$. The constraint schemes result from the **interface** and **implementation** definitions of the program (to be defined in Fig. 9). A type environment Δ is a sequence of type variables and constraints. Its domain, written $\mathsf{dom}(\Delta)$, consists of only the type variables. The extension of a type environment is written $\Delta, \overline{X}, \overline{Q}$ assuming $\mathsf{dom}(\Delta) \cap \overline{X} = \emptyset$. A variable environment Γ is a finite mapping from variables to types, written $\overline{x : T}$. The extension of a variable environment is denoted by $\Gamma, x : T$ assuming $x \notin \mathsf{dom}(\Gamma)$.

---

[4] Interoperability with Java requires support for casts to certain bounded existential types whose constraints are easily checkable at runtime.

$$\boxed{\mathsf{mtype}_{\Theta;\Delta}(m, T) = msig \qquad \mathsf{smtype}_{\Theta;\Delta}(m, K[\overline{T}]) = msig}$$

$$\frac{\textbf{class } C\langle\overline{X}\rangle \textbf{ extends } N \textbf{ where } \overline{Q} \{\ldots \overline{m : msig \{e\}}\} \in \Theta}{\mathsf{mtype}_{\Theta;\Delta}(m_j, C\langle\overline{T}\rangle) = [\overline{T/X}]msig_j}$$

$$\frac{\Theta; \Delta \Vdash \overline{T} \textbf{ implements } I\langle\overline{V}\rangle}{\textbf{interface } I\langle\overline{X}\rangle\,[\overline{Y}] \textbf{ where } \overline{Q}\{\ldots \overline{itsig}\} \in \Theta \qquad itsig_j = \textbf{receiver } \{\overline{m : msig}\}}{\mathsf{mtype}_{\Theta;\Delta}(m_k, T_j) = [\overline{V/X}, \overline{T/Y}]msig_k}$$

$$\frac{\Theta; \Delta \Vdash \overline{T} \textbf{ implements } I\langle\overline{S}\rangle \quad \textbf{interface } I\langle\overline{X}\rangle\,[\overline{Y}] \textbf{ where } \overline{Q}\{\overline{m : \textbf{static } msig} \ldots\} \in \Theta}{\mathsf{smtype}_{\Theta;\Delta}(m_j, I\langle\overline{S}\rangle[\overline{T}]) = [\overline{S/X}, \overline{T/Y}]msig_j}$$

Fig. 7: Method types.

Fig. 5 establishes well-formedness predicates on types, instantiated interfaces with implementing types, constraints, and substitutions. The judgment $\Theta; \Delta \vdash [\overline{T/X}]$ ok under $\overline{Q}$ ensures that the (capture avoiding) substitution $[\overline{T/X}]$ replaces $\overline{X}$ with well-formed types $\overline{T}$ that respect the constraints $\overline{Q}$. Its definition uses the entailment judgment $\Theta; \Delta \Vdash Q$ discussed next. We abbreviate multiple well-formedness predicates $\Theta; \Delta \vdash \xi_1$ ok, $\ldots$, $\Theta; \Delta \vdash \xi_n$ ok to $\Theta; \Delta \vdash \overline{\xi}$ ok.

Fig. 6 defines the entailment and the subtyping relation. Entailment $\Theta; \Delta \Vdash Q$ establishes the validity of constraint Q. A constraint is only valid if it is either contained in the local type environment Δ, or if it is implied by a constraint scheme of the program environment Θ. There is no rule that allows us to conclude T **implements** I if all we know is T' **implements** I for some supertype T' of T. Such a conclusion would be unsound because the implementing type of I might appear in the result type of some method.[5] Similarly, a constraint such as $(\exists X \textbf{ where } X \textbf{ implements } I \,.\, X) \textbf{ implements } I$ is only valid if there exists a corresponding implementation definition; otherwise, invocations of methods with the implementing type of I in argument position would be unsound.[6]

The subtyping judgment $\Theta; \Delta \vdash T \leq U$ is similar to FGJ, except that there is a top rule $\Theta; \Delta \vdash T \leq \textit{Object}$ and two rules for bounded existential types. These two rules allow for implicit conversions between existential and non-existential values. The first allows opening an existential on the left-hand side if the quantified type variables are sufficiently fresh (guaranteed by the premise $\Theta; \Delta \vdash U$ ok). The second rule allows abstracting over types that fulfill the constraints of the existential on the right-hand side.

The relation $\mathsf{mtype}_{\Theta;\Delta}(m, T) = msig$, defined in Fig. 7, determines the signature of method m invoked on a receiver with static type T. The first rule is similar to the corresponding rules in FJG, except that it does not ascend the inheritance tree; instead, Core–JavaGI allows for subsumption on the receiver type (see the typing rules for expressions in Fig. 8). The second rule handles the case

---

[5] To ensure interoperability with Java, we generate suitable implementation definitions for all Java classes that "inherit" the implementation of an interface from a superclass.

[6] Interoperability with Java requires us to generate implementation definitions for I and all its superinterfaces with implementing type $\exists X \textbf{ where } X \textbf{ implements } I \,.\, X$.

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

$$\frac{\Theta; \Delta; \Gamma \vdash e : T \qquad \mathsf{bound}(T) = \exists \overline{X} \textbf{ where } \overline{Q} . N \qquad \mathsf{fields}_\Theta(N) = \overline{U\,f}}{\Theta; \Delta; \Gamma \vdash e.f_j : \exists \overline{X} \textbf{ where } \overline{Q} . U_i}$$

$$\frac{\begin{array}{c} \Theta; \Delta; \Gamma \vdash e_0 : T_0 \qquad \Theta; \Delta \vdash T_0 \leq \exists \overline{X} \textbf{ where } \overline{Q} . T_0' \\ (\forall i)\ \Theta; \Delta; \Gamma \vdash e_i : S_i \qquad \mathsf{mtype}_{\Theta; \Delta, \overline{X}, \overline{Q}}(m, T_0') = \langle \overline{Y} \rangle \overline{U\,x} \to U \textbf{ where } \overline{P} \\ \Theta; \Delta \vdash \overline{V} \textsf{ ok} \qquad (\forall i)\ \Theta; \Delta, \overline{X}, \overline{Q} \Vdash [\overline{V/Y}]P_i \qquad (\forall i)\ \Theta; \Delta, \overline{X}, \overline{Q} \vdash S_i \leq [\overline{V/Y}]U_i \end{array}}{\Theta; \Delta; \Gamma \vdash e_0.m\langle \overline{V} \rangle(\overline{e}) : \exists \overline{X} \textbf{ where } \overline{Q} . [\overline{V/Y}]U}$$

$$\frac{\begin{array}{c} \Theta; \Delta \vdash K[\overline{T}] \textsf{ ok} \qquad \mathsf{smtype}_{\Theta; \Delta}(m, K[\overline{T}]) = \langle \overline{X} \rangle \overline{U\,x} \to U \textbf{ where } \overline{Q} \\ \Theta; \Delta \vdash [\overline{V/X}] \textsf{ ok under } \overline{Q} \qquad (\forall i)\ \Theta; \Delta; \Gamma \vdash e_i : S_i \qquad (\forall i)\ \Theta; \Delta \vdash S_i \leq [\overline{V/X}]U_i \end{array}}{\Theta; \Delta; \Gamma \vdash K[\overline{T}].m\langle \overline{V} \rangle(\overline{e}) : [\overline{V/X}]U}$$

Fig. 8: Expression typing. The remaining rules are similar to those for FGJ [17] and thus omitted.

of invoking a method defined in an interface implemented by the receiver. The judgment $\mathsf{smtype}_{\Theta; \Delta}(m, K[\overline{T}]) = msig$, also shown in Fig. 7, defines the type of a static method invoked on instantiated interface K for implementing types $\overline{T}$.

The typing judgment for expressions $\Theta; \Delta; \Gamma \vdash e : T$ (Fig. 8) assigns type T to expression e under the environments Θ, Δ, and Γ. Its definition is very similar to the corresponding FGJ judgment, so we show only those rules that are new (rule for static method invocation) or significantly different (rules for field lookup and non-static method invocation). Following Wild FJ [34], the rules for field lookup and non-static method invocation propagate the existentially bounded type variables and the constraints of the receiver type to the conclusion to ensure proper scoping. Furthermore, the rule for non-static method invocation allows subsumption on the receiver type. This change was necessary because entailment does not take subtyping into account (see the discussion on page 18), so without subsumption on the receiver type, it would not be possible to invoke a retroactively defined method on a receiver whose static type is a subtype of the type used in the corresponding implementation definition.

In the rule for field lookup, $\mathsf{fields}_\Theta(N)$ denotes the fields of N and its superclasses (as in FGJ), and $\mathsf{bound}(T)$ denotes the upper bound of T. It is defined as $\mathsf{bound}(N) = N$, $\mathsf{bound}(X) = Object$, and $\mathsf{bound}(\exists \overline{X} \textbf{ where } \overline{Q} . T) = \exists \overline{X} \textbf{ where } \overline{Q} . \mathsf{bound}(T)$.

Fig. 9 defines the program typing rules. They differ from FGJ because FGJ defines only a method and a class typing judgment. The first part of the figure defines well-formedness of method definitions and a subtyping relation on method signatures.[7] The next three parts check that class, interface, and implementation definitions are well-formed. The last part defines the two judgments $def \Rightarrow \Theta$ and $\vdash prog$ ok. The first judgment collects the constraint schemes resulting from the definitions in the program: a class definition contributes no constraint schemes, an interface definition contributes constraint schemes for all of its superinterfaces because every implementation of the interface must respect

---

[7] Interoperability with Java requires a second relation specifying covariant return types, only.

$$\boxed{\Theta; \Delta; \Gamma \vdash mdef \text{ ok} \qquad \Theta; \Delta \vdash msig \leq msig}$$

$$\frac{\Delta' = \Delta, \overline{X}, \overline{Q} \qquad \Theta; \Delta' \vdash \overline{T}, U, \overline{Q} \text{ ok} \qquad \Theta; \Delta'; \Gamma, \overline{x : T} \vdash e :_{\leq} U}{\Theta; \Delta; \Gamma \vdash \langle \overline{X} \rangle \, \overline{T \, x} \rightarrow U \text{ where } \overline{Q} \, \{e\} \text{ ok}}$$

$$\frac{(\forall i) \; \Theta; \Delta, \overline{X}, \overline{Q} \Vdash [\overline{X/Y}]P_i \qquad (\forall i) \; \Theta; \Delta, \overline{Y}, \overline{P} \Vdash [\overline{Y/X}]Q_i}{(\forall i) \; \Theta; \Delta, \overline{X}, \overline{Q} \vdash [\overline{X/Y}]U_i \leq T_i \qquad (\forall i) \; \Theta; \Delta, \overline{X}, \overline{Q} \vdash T \leq [\overline{X/Y}]U}{\Theta; \Delta \vdash \langle \overline{X} \rangle \, \overline{T \, x} \rightarrow T \text{ where } \overline{Q} \leq \langle \overline{Y} \rangle \, \overline{U \, x} \rightarrow U \text{ where } \overline{P}}$$

$$\boxed{\Theta; \Delta \vdash m : mdef \text{ ok in } N \qquad \text{override-ok}_{\Theta; \Delta}(m : msig, N) \qquad \Theta \vdash cdef \text{ ok}}$$

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

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

$$\frac{\Delta = \overline{X}, \overline{Q} \qquad \Theta; \Delta \vdash N, \overline{Q}, \overline{T} \text{ ok} \qquad (\forall i) \; \Theta; \Delta \vdash m_i : mdef_i \text{ ok in } C\langle \overline{X} \rangle}{\Theta \vdash \textbf{class } C\langle \overline{X} \rangle \textbf{ extends } N \textbf{ where } \overline{Q} \, \{\overline{T \, f} \; \overline{m : mdef}\} \text{ ok}}$$

$$\boxed{\Theta; \Delta \vdash msig \text{ ok} \qquad \Theta; \Delta \vdash itsig \text{ ok} \qquad \Theta \vdash idef \text{ ok}}$$

$$\frac{\Delta' = \Delta, \overline{X}, \overline{Q} \qquad \Theta; \Delta' \vdash \overline{T}, U, \overline{Q} \text{ ok}}{\Theta; \Delta \vdash \langle \overline{X} \rangle \, \overline{T \, x} \rightarrow U \text{ where } \overline{Q} \text{ ok}} \qquad \frac{(\forall i) \; msig_i \text{ ok}}{\Theta; \Delta \vdash \textbf{receiver } \{\overline{m : msig}\} \text{ ok}}$$

$$\frac{\Delta = \overline{X}, \overline{Y}, \overline{Q} \qquad \Theta; \Delta \vdash \overline{Q}, \overline{msig}, \overline{itsig} \text{ ok}}{\Theta \vdash \textbf{interface } I\langle \overline{X} \rangle \, [\overline{Y}] \textbf{ where } \overline{Q} \, \{\overline{m : \textbf{static } msig} \; \overline{itsig}\} \text{ ok}}$$

$$\boxed{\Theta; \Delta \vdash mdef \text{ implements } msig \qquad \Theta; \Delta \vdash itdef \text{ implements } itsig \qquad \Theta \vdash impl \text{ ok}}$$

$$\frac{\Theta; \Delta; \Gamma \vdash msig \, \{e\} \text{ ok} \qquad \Theta; \Delta \vdash msig \leq msig'}{\Theta; \Delta; \Gamma \vdash msig \, \{e\} \text{ implements } msig'}$$

$$\frac{(\forall i) \; \Theta; \Delta; \Gamma \vdash mdef_i \text{ implements } msig_i}{\Theta; \Delta; \Gamma \vdash \textbf{receiver } \{\overline{m : mdef}\} \text{ implements } \textbf{receiver } \{\overline{m : msig}\}}$$

$$\frac{\Delta = \overline{X}, \overline{Q} \qquad \Theta' = \Theta \setminus \{\forall \overline{X} . \overline{Q} \Rightarrow \overline{S} \textbf{ implements } I\langle \overline{T} \rangle\}}{\Theta'; \Delta \vdash I\langle \overline{T} \rangle [\overline{S}], \overline{Q} \text{ ok} \qquad \textbf{interface } I\langle \overline{Y} \rangle \, [\overline{Z}] \textbf{ where } \overline{P} \, \{\overline{m : \textbf{static } msig} \; \overline{itsig}\} \in \Theta}{(\forall i) \; \Theta; \Delta; \emptyset \vdash mdef_i \text{ implements } [\overline{T/Y}, \overline{S/Z}]msig_i}{(\forall i) \; \Theta; \Delta; this : S_i \vdash itdef_i \text{ implements } [\overline{T/Y}, \overline{S/Z}]itsig_i}{\Theta \vdash \textbf{implementation}\langle \overline{X} \rangle \, I\langle \overline{T} \rangle \, [\overline{S}] \textbf{ where } \overline{Q} \, \{\overline{m : \textbf{static } mdef} \; \overline{itdef}\} \text{ ok}}$$

$$\boxed{def \Mapsto \Theta \qquad \vdash prog \text{ ok}}$$

$$cdef \Mapsto \emptyset$$

$$\textbf{interface } I\langle \overline{X} \rangle \, [\overline{Y}] \textbf{ where } \overline{Q}^n \, \{\dots\} \Mapsto \{\forall \overline{XY} . \overline{Y} \textbf{ implements } I\langle \overline{X} \rangle \Rightarrow Q_i \mid 1 \leq i \leq n\}$$

$$\textbf{implementation}\langle \overline{X} \rangle \, K \, [\overline{T}] \textbf{ where } \overline{Q} \, \{\dots\} \Mapsto \{\forall \overline{X} . \overline{Q} \Rightarrow \overline{T} \textbf{ implements } K\}$$

$$\frac{(\forall i) \; def_i \Mapsto \Theta_i \qquad \Theta = \overline{def} \cup \cup_i \Theta_i}{(\forall i) \; \Theta \vdash def_i \text{ ok} \qquad \text{well-founded}(\Theta) \qquad \text{no-overlap}(\Theta) \qquad \Theta; \cdot; \cdot \vdash e : T}{\vdash \overline{def} \; e \text{ ok}}$$

Fig. 9: Program typing.

the superinterface constraints, and an implementation definition contributes a single constraint scheme. The judgment ⊢ *prog* ok first collects all constraint schemes, then checks the definitions of the program, and finally types the main expression. The predicate well-founded(Θ) only holds if the class and interface hierarchies of program Θ are acyclic. The predicate no-overlap(Θ) ensures that program Θ does not contain overlapping implementation definitions, *i.e.*, no implementation definition in Θ is a substitution instance of some other implementation definition.

5   Related Work

PolyTOIL [7] and $\mathcal{LOOM}$ [6] are both object-oriented languages with a MyType type as needed for binary methods: an occurrence of MyType refers to the type of **this**. PolyTOIL achieves type safety by separating inheritance from subtyping, whereas $\mathcal{LOOM}$ drops subtyping completely. However, both languages support matching, which is more general than subtyping. The language LOOJ [4] integrates MyType into Java. It ensures type safety through exact types that prohibit subtype polymorphism. Compared with these languages, JavaGI does not support MyType in classes but only in interfaces. As a consequence, JavaGI allows unrestricted subtype polymorphism on classes; only invocations of binary methods on receivers with existential type are disallowed. 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. $\mathcal{LOOM}$ supports "hash types", which can be interpreted as match-bounded existential types in the same way as JavaGI's interface types are interpreted as interface-bounded existential types. Hash types, though, are tagged explicitly.

The multi-headed interfaces of JavaGI enable a statically safe form of family polymorphism (dating back to BETA's [24] virtual types). Other work on family polymorphism either use path-dependent types [12], virtual classes [13], or a generalized form of MyType [5] that deals with a mutually recursive system of classes. Scala's abstract types together with self type annotations [28, 27] can also be used for family polymorphism. Helm and collaborators' contracts [16] specify how groups of interdependent objects should cooperate, thus allowing some form of family polymorphism.

JavaGI's generalization of Java interfaces is systematically inspired by Haskell's type-class mechanism [37, 29, 30]: (multi-headed) interface and implementation definitions in JavaGI play the role of (multi-parameter) type classes and instance definitions in Haskell (so far without functional dependencies). A notable difference between JavaGI and Haskell is that Haskell does not have the notion of classes and objects in the object-oriented sense, so methods are not tied to a particular class or object. Thus, methods of Haskell type classes correspond to static methods of JavaGI's interfaces; there is no Haskell correspondence to JavaGI's non-static interface methods. Another difference is the absence of subtyping in Haskell, which avoids the question how subtyping and instance definitions should interact. However, Haskell supports type inference, whereas JavaGI

requires explicit type annotations. Finally, Haskell's existentials are notoriously inconvenient since they are bound to data-type constructors and lack implicit pack and unpack operations.

Siek and collaborators have developed a related notion of concepts for grouping and organizing requirements on a type [32]. In particular, they have also formalized this notion in F^G, an extension of System F, receiving inspiration from Haskell type classes. F^G also includes associated types (*i.e.*, types functionally depending on other types). In contrast, JavaGI supports self types, bounded existential types, defaults for interface methods, and it interacts with subtyping. It has been noted that a limited form of concepts can be also realized with C#'s interface support [19], while the primary application domain of concepts (*i.e.*, generic programming) requires extra support for associated types and constraint propagation. We note that constraint propagation [19] is related to our notion of constraint entailment.

There is an impressive number of approaches for some form of open classes—means to extend existing classes. The approaches differ with regard to the "extension time" and the restrictions imposed on extensions. Partial classes in C♯ 2.0 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 [25] support full separate compilation, but the added methods cannot be virtual, and members other than methods cannot be added. Aspect-oriented language implementations such as AspectJ [22] typically support some sort of open classes based on a global program analysis, a byte-code–level weaving technique, or more dynamic approaches.

MultiJava [11] is a conservative Java extension that adds open classes and multimethods. We adopted MultiJava's implementation strategy to account for retroactive interface implementations and for implementing binary methods [3] by specializing the argument types in subclasses. The design of Relaxed Multi-Java [26] might help to lift the restrictions imposed by our compilation strategy. Expanders [38] comprise an extra language construct (next to classes and interfaces) for adding new state, methods and superinterfaces to existing classes in a modular manner. JavaGI does not support state extension. Expanders do not deal with family polymorphism, static interface methods, binary methods, and some other aspects of JavaGI.

The expander paper [38] comprises an excellent related work discussion looping in all kinds of approaches that are more or less remotely related to class extensions: mixins, traits, nested inheritance, and Scala views.

6   Conclusion and Future Work

We have described JavaGI, a language that generalizes Java's interfaces in various dimensions to enable clearer program designs, stronger static typing, and extra forms of software extension and integration. Our generalization is based on Haskell's type class mechanism. The design of JavaGI shows that the combina-

tion of type classes and bounded existential types with implicit pack and unpack operations subsumes Java-like interfaces. We have watched out for feature interactions with existing uses of interfaces, subtyping, and subclassing. In particular, JavaGI is the first satisfactory example of a language where type classes (interfaces) and subtyping coexist. In this language-design process, we realized that a convenient form of existential quantification needs to become part of the extended Java type system. All of the scenarios that JavaGI can handle have been previously identified in other work; however, using separate language extensions with unclear interaction. There is no single proposal that would match the expressiveness of JavaGI. Hence, we do not apply for an originality award but we hope to score with the uniformity and simplicity of generalized interfaces.

The formalization of JavaGI presented in this article consists of only a type system for a core language. In future work, we would like to complete the formalization. In particular, this includes the specification of an operational semantics, its soundness proof, an algorithmic formulation of subtyping, the adoption of Java's inference algorithm for type parameters, the completeness proof for representing Java generics in JavaGI's existential-based type system, and a proper formalization of the translation to Java. Furthermore, we are working on a prototype compiler for JavaGI from which we also expect real-world data on the overhead caused by the translation semantics. We also would like to lift the restrictions imposed by our compilation strategy, and we are investigating state extension for JavaGI. Another challenging aspect is the potential of multiple type parameters of generalized interfaces (both implementing types and regular type parameters). Such multiplicity has triggered advanced extensions for Haskell's type classes [18, 9, 8] to restrict and direct instance selection and type inference. In the context of JavaGI, the existing restriction for generics—that a certain type can implement a generic interface for only one type instantiation—may be sufficient for practical purposes.

References

1. G. Bracha. Generics in the Java programming language. `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`, July 2004.
2. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. 13th ACM Conf. OOPSLA*, pages 183–200, Vancouver, BC, Canada, Oct. 1998. ACM Press, New York.
3. K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.
4. K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. In M. Odersky, editor, *18th ECOOP*, volume 3086 of *LNCS*, pages 389–413, Oslo, Norway, June 2004. Springer.

5. K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In E. Jul, editor, *12th ECOOP*, number 1445 in LNCS, pages 523–549, Brussels, Belgium, July 1998. Springer.

6. K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In M. Aksit and S. Matsuoka, editors, *11th ECOOP*, number 1241 in LNCS, pages 104–127, Jyväskylä, Finland, June 1997. Springer.

7. K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Trans. Prog. Lang. and Systems*, 25(2):225–290, 2003.

8. M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In B. C. Pierce, editor, *Proc. Intl. Conf. Functional Programming 2005*, pages 241–253, Tallinn, Estonia, Sept. 2005. ACM Press, New York.

9. M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In M. Abadi, editor, *Proc. 32nd ACM Symp. POPL*, pages 1–13, Long Beach, CA, USA, Jan. 2005. ACM Press.

10. C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *6th ECOOP*, volume 615 of *LNCS*, pages 33–56. Springer, 1992.

11. C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Prog. Lang. and Systems*, 28(3):517–575, 2006.

12. E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *15th ECOOP*, number 2072 in LNCS, pages 303–326, Budapest, Hungary, June 2001. Springer.

13. E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In S. Peyton Jones, editor, *Proc. 33rd ACM Symp. POPL*, pages 270–282, Charleston, South Carolina, USA, Jan. 2006. ACM Press.

14. P. Eugster. Uniform proxies for Java. In *Proc. 21th ACM Conf. OOPSLA*, pages 139–152, Portland, OR, USA, 2006. ACM Press, New York.

15. C. V. Hall, K. Hammond, S. L. P. Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Prog. Lang. and Systems*, 18(2):109–138, 1996.

16. R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Conf. OOPSLA / ECOOP*, SIGPLAN Notices 25(10), pages 169–180, Ottawa, Canada, Oct. 1990.

17. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. and Systems*, 23(3):396–450, May 2001.

18. M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Proc. 9th European Symp. Programming*, number 1782 in LNCS, pages 230–244, Berlin, Germany, Mar. 2000. Springer.

19. J. Järvi, J. Willcock, and A. Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *Proc. 20th ACM Conf. OOPSLA*, pages 1–19, New York, NY, USA, 2005. ACM Press.

20. S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. of the 2nd European Symp. Programming*, number 300 in LNCS, pages 131–144. Springer, 1988.

21. A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *Proc. 2001 PLDI*, pages 1–12, Snowbird, UT, United States, June 2001. ACM Press, New York, USA.

22. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *15th ECOOP*, number 2072 in LNCS, pages 327–353, Budapest, Hungary, June 2001. Springer.

23. R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *GPCE '06*, pages 161–170, New York, NY, USA, 2006. ACM Press.

24. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

25. Microsoft Corp. C# Version 3.0 Specification, May 2006. `http://msdn2.microsoft.com/en-us/vcsharp/aa336745.aspx`.

26. T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proc. 18th ACM Conf. OOPSLA*, pages 224–240, Anaheim, CA, USA, 2003. ACM Press, New York.

27. M. Odersky. The scala language specification version 2.0, Nov. 2006. Draft, `http://scala.epfl.ch/docu/files/ScalaReference.pdf`.

28. M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. 20th ACM Conf. OOPSLA*, pages 41–58, San Diego, CA, USA, 2005. ACM Press, New York.

29. S. Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

30. S. Peyton Jones, M. Jones, and E. Meijer. Type classes: An exploration of the design space. In J. Launchbury, editor, *Proc. of the Haskell Workshop*, Amsterdam, The Netherlands, June 1997.

31. K. C. Sekharaiah and D. J. Ram. Object schizophrenia problem in object role system design. In *OOIS '02: Proc. 8th Int. Conf. Object-Oriented Inf. Systems*, pages 494–506, London, UK, 2002. Springer-Verlag.

32. J. Siek and A. Lumsdaine. Essential language support for generic programming. In *Proc. 2005 ACM Conf. PLDI*, pages 73–84, New York, NY, USA, June 2005. ACM Press.

33. M. Torgersen. The expression problem revisited — four new solutions using generics. In M. Odersky, editor, *18th ECOOP*, volume 3086 of *LNCS*, Oslo, Norway, June 2004. Springer.

34. M. Torgersen, E. Ernst, and C. P. Hansen. Wild FJ. In *International Workshop on Foundations of Object-Oriented Languages,informal proceedings*, 2005.

35. M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the java programming language. *Journal of Object Technology*, 3(11):97–116, Dec. 2004.

36. P. Wadler. The expression problem, 1998. Posted on Java Genericity mailing list.

37. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th ACM Symp. POPL*, pages 60–76, Austin, Texas, Jan. 1989. ACM Press.

38. A. Warth, M. Stanojevic, and T. Millstein. Statically scoped object adaptation with expanders. In *Proc. 21th ACM Conf. OOPSLA*, pages 37–56, Portland, OR, USA, 2006. ACM Press, New York.

39. D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In X. Leroy, editor, *Proc. 31st ACM Symp. POPL*, pages 39–51, Venice, Italy, Jan. 2004. ACM Press.