ML Modules and Haskell Type Classes:
A Constructive Comparison

Stefan Wehr

Diplomarbeit

Albert-Ludwigs-Universität Freiburg
Fakultät für Angewandte Wissenschaften
Institut für Informatik

November 2005

Erstgutachter: Prof. Dr. Peter Thiemann
Zweitgutachter und Betreuer: Dr. Manuel M. T. Chakravarty
(University of New South Wales)

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus veröffentlichten Schriften entnommen wurden, sind als solche gekennzeichnet. Die vorliegende Arbeit ist, weder komplett noch auszugsweise, zu keiner anderen Prüfung angefertigt worden.

Syndey, im November 2005 Stefan Wehr

Abstract

Researchers repeatedly observed that the module system of ML and the type class mechanism of Haskell are related. So far, this relationship has not been formally investigated. The work at hand fills this gap by presenting a constructive comparison between ML modules and Haskell type classes; that is, it introduces two formal translations from modules to type classes and vice versa, which enable a thorough comparison of the two concepts.

The source language of the first translation is a subset of Standard ML. The target language is Haskell with common extensions and one new feature, which was developed as part of this work. The second translation maps a subset of Haskell 98 to ML with well-established extensions. I prove that the translations preserve type correctness and provide implementations for both.

Building on the insights obtained from the translations, I present a thorough comparison between ML modules and Haskell type classes. Moreover, I evaluate to what extent the techniques used in the translations can be exploited for modular programming in Haskell and for programming with ad-hoc polymorphism in ML.

Zusammenfassung

Es wurde bereits mehrfach festgestellt, dass das ML Modulsystem und der Typklassenmechanismus von Haskell Ähnlichkeiten aufweisen. Bis zum heutigen Zeitpunkt wurde dieser Zusammenhang allerdings nicht formal untersucht. Die vorliegende Arbeit füllt diese Lücke durch einen konstruktiven Vergleich zwischen Modulen in ML und Typklassen in Haskell; die Arbeit entwickelt also zwei formale Übersetzungen zwischen Modulen und Typklassen, die einen detaillierten Vergleich zwischen den beiden Konzepten ermöglichen.

Die Quellsprache der ersten Übersetzung ist eine Untermenge von Standard ML, während als Zielsprache Haskell mit weitverbreiteten Erweiterungen sowie einer neuen, in dieser Arbeit entwickelten Erweiterung verwendet wird. Die zweite Übersetzung bildet eine Teilmenge von Haskell 98 auf ML mit etablierten Erweiterungen ab. Ich beweise, dass die Übersetzungen typkorrektheitserhaltend sind und stelle für beide Übersetzungen Implementierungen zur Verfügung.

Auf den durch die Übersetzungen gewonnenen Einsichten aufbauend, präsentiere ich einen detaillierten Vergleich zwischen Modulen in ML und Typklassen in Haskell. Desweiteren evaluiere ich, inwiefern die in den Übersetzungen verwendeten Techniken zur modularen Programmierung in Haskell und zum Programmieren mit ad-hoc Polymorphie in ML verwendet werden können.

Acknowledgments

Manuel Chakravarty accepted me as a student member of his research group and gave me the opportunity to write my diploma thesis in Sydney. I would like to thank Manuel for his excellent support, for his guidance, and for many useful discussions.

Peter Thiemann drew my attention to functional programming languages and supported me firmly throughout my studies. I would like to thank him for his help in organizing my diploma thesis abroad, for useful feedback on a rough draft version of this thesis, and for assessing the thesis.

Roman Leshchinskiy provided me with a workplace in his office, was always available for questions, and suggested various improvements to the presentation of the thesis. Gabriele Keller and Don Stewart let me use their minHS system as a basis for my implementations and provided valuable feedback on a draft of the thesis. I am also indebted to Sean Seefried and Simon Winwood for useful feedback on a draft of the thesis and to André Pang for providing me with an initial workplace.

Special thanks go to the whole Programming Languages & Systems research group at the University of New South Wales for their hospitality. I really enjoyed working in this vibrant `;-)` research group.

My wife Claudia not only printed and submitted the thesis in Freiburg, but also gave me a lot of mental support and motivation. Thank you! Last but not least, I would like to thank my parents Christine and Adam Heimann for their ongoing support.

Contents

List of Figures

Chapter 1.

Introduction

On first glance, module systems and type classes appear to be unrelated programming-language concepts: Module systems allow large programs to be decomposed into smaller, relatively independent units, whereas type classes [Kae88, WB89] provide a means for introducing ad-hoc polymorphism; that is, they give programmers the ability to define multiple functions or operators with the same name but different types. However, it has been repeatedly observed [Sch00, KS01, CKPM05, CKP05, Ros05] that there is some overlap in functionality between the module system of the programming language ML [MTHM97], one of the most powerful module systems in widespread use, and the type class mechanism of the language Haskell [Pey03], which constitutes a sophisticated approach to ad-hoc polymorphism.

It is natural to ask whether these observations rest on a solid foundation, or whether the overlap is only superficial. The standard approach to answer such a question is to devise two formal translations from modules to type classes and vice versa. The translations then pinpoint exactly the features that are easy, hard, or impossible to translate; thereby showing very clearly the differences and commonalities between the two concepts.

Such a constructive comparison between ML modules and Haskell type classes is particularly interesting because the strength of one language is a weak point of the other: ML has only very limited support for ad-hoc polymorphism, so translating Haskell type classes to ML modules could give new insights on how to program with this kind of polymorphism in ML. Conversely, the Haskell module system is weak, so an encoding of ML's powerful module system with type classes could open up new possibilities for modular programming in Haskell.

1.1. Goals

The concrete goals of this research are as follows:

- Devise two formal translations from ML modules to Haskell type classes and vice versa.
- Use the insights obtained from these translations to compare ML modules with Haskell type classes thoroughly.

- Investigate if and how the techniques used to encode ML modules in terms of Haskell type classes and vice versa can be exploited for modular programming in Haskell and for programming with ad-hoc polymorphism in ML, respectively.
- Suggest additions to both languages that address possible shortcomings identified by the translations.

1.2. Related work

I now describe the most relevant pieces of related work and show why none of these reach the goals outlined in the preceding listing. More related work is deferred to the following chapters.

Kahl and Scheffczyk propose in [KS01] a Haskell extension that enhances the late-binding capabilities of the type class system. They motivate the design of the extension by a comparison between modules in OCaml (an ML dialect [Ler00b]) and type classes. However, they do not develop any kind of formal translation, so the comparison stays rather superficial.

Shan [Sha04] presents a formal translation from a sophisticated ML module calculus [DCH03] into System F_ω [Gir72]. Although System F_ω can be encoded in Haskell extended with higher-rank types [PS04b], Shan's translation adds nothing of significance to the comparison between modules and type classes because the encoding of System F_ω is orthogonal to the type class system. Moreover, Shan does not give a translation for the opposite direction from type classes to modules.

In a posting to the Haskell mailing list [Kis04], Kiselyov attempts to "interpret some of Ken's [Shan] results in idiomatic Haskell with the full use of type classes". He does so by showing only a relatively small example, so it is unclear if his technique could be generalized to a formal translation from ML modules to Haskell type classes.

Schneider [Sch00] approaches the problem from a different angle: He presents type classes as an extension to ML, resulting in a language that supports both ML modules and Haskell-style type classes. However, Schneider does not translate one concept into the other.

1.3. Outline

The remainder of the thesis is organized in the following way:

Chapter 2. This chapter gives an introduction to the ML module system and formalizes two module calculi, namely Tiny-ML and Tiny-ML$^+$. Tiny-ML is a simplified version of Standard ML; the most notable feature missing in comparison with Standard ML is the ability to define nested structures. Tiny-ML$^+$ extends Tiny-ML with features from Standard ML, with recursive func-

tors [Rus01], and with first-class structures [Rus00a]. Chapter 2 also contains some conventions and definitions used throughout the thesis.

Chapter 3. In this chapter, I introduce Haskell type classes and define the two languages Tiny-HS and Tiny-HS$^+$. Tiny-HS features type classes in the style of Haskell 98; however, constructor classes, methods with constraints, and default methods are not supported. Tiny-HS$^+$ extends Tiny-HS with features from Haskell 98, with multi-parameter type classes [PJM97], and with associated type synonyms [CKP05]. The chapter also suggests abstract associated type synonyms as an extension to associated type synonyms.

Chapter 4. This chapter presents an encoding of ML modules with Haskell type classes by formalizing a translation from Tiny-ML to Tiny-HS$^+$. I prove that every type correct Tiny-ML program is translated into a type correct Tiny-HS$^+$ program, and discuss an implementation of the translation.[1]

Chapter 5. In this chapter, I show how Haskell type classes can be translated to ML modules by defining a mapping from Tiny-HS to Tiny-ML$^+$. Again, I prove that every type correct Tiny-HS program yields a type correct Tiny-ML$^+$ program, and demonstrate how the translation can be implemented.[1]

Chapter 6. The last chapter compares ML modules with Haskell type classes, outlines possible directions for future work, summarizes, and concludes.

[1] All material is available from `http://www.stefanwehr.de/diplom`.

Chapter 2.

ML modules

Modularization is essential for software development—it allows programs to be decomposed into small units (which are often called *modules*) that can be understood, developed, built, and maintained in isolation. Two key questions arise in the process of modularization.

- Which criteria should be used for decomposing a program into modules; that is, where is the borderline between different modules?
- How can the decomposition be realized in a concrete programming language?

The answer to the first question is beyond the scope of this work. It is discussed in great detail in the software-engineering literature; a good starting point is Parnas' article [Par72]. The module system of the programming language Standard ML [MTHM97] constitutes an answer to the second question; in fact, the ML module system is one of the most powerful in widespread use.

This chapter gives an introduction to the ML module system (Section 2.1) and formalizes two module calculi, namely Tiny-ML (Section 2.3) and Tiny-ML$^+$ (Section 2.4). Tiny-ML serves as the source language for the translation from ML modules to Haskell type classes, whereas Tiny-ML$^+$ is used as the target language for the translation in the opposite direction. (A rationale for using two different languages is given in Section 2.4.) Moreover, the chapter at hand contains some conventions and definitions used throughout the rest of the thesis (Section 2.2) and discusses work related to Tiny-ML and Tiny-ML$^+$ (Section 2.5).

2.1. Introduction to ML modules

To support proper modularization, a programming language should provide some, if not all, of the following features [Jon96]:

- A mechanism to support some form of namespace management.
- A mechanism for defining abstractions.
- A mechanism to enable the decomposition of large programs into small, reusable units in a way that is resistant to small changes in the program.

- A mechanism to support separate or at least incremental compilation. Separate compilation denotes the ability to compile some program unit without examining the rest of the program, whereas incremental compilation denotes the ability to compile some unit by examining only the interfaces and not the implementations of the units it depends on.[1]

We now discuss a series of examples that demonstrates how these features are realized in the module system of Standard ML. Many features of ML are not covered in this introduction. There are numerous tutorials, articles, and books that provide a more complete coverage of the topic [Pau96, Ler00a, HP04, Gil04].

2.1.1. Structures

Suppose we want to provide an implementation for sets of integers. Our first solution might look like this:[2]

```
structure IntSet1 =
  struct
    type elem   = int
    type set    = list elem
    val  empty  = []
    val  member = λi . λs . exists (λj . primIntEq i j) s
    val  insert = λi . λs . if member i s then s else (cons i s)
  end
```

The preceding piece of code introduces a new *structure* named IntSet_1, which implements sets in terms of lists by grouping together the types and values of the set interface. After the definition of IntSet_1, we can use the components of the structure via the dot notation. For example, to construct the set $\{0, 42\}$, we would write $\mathrm{IntSet}_1.\mathrm{insert}\ 42\ (\mathrm{IntSet}_1.\mathrm{insert}\ 0\ \mathrm{IntSet}_1.\mathrm{empty})$.

2.1.2. Signatures

The IntSet_1 structure defined in the preceding section reveals that sets are implemented in terms of lists to the clients of the structure. This is not always desirable; often, the type set should be kept abstract outside of the structure.

We can make types abstract by *sealing a structure with a signature*. Signatures describe the interfaces of structures. The following example shows how we can seal the IntSet_1 structure with a signature that keeps the type set abstract:

[1] The terminology is not standard. We use the terminology of Cardelli [Car97] and Harper & Pierce [HP04].

[2] The ML code presented here is not syntactically valid Standard ML code because arguments of type constructors are written after the type constructor and not in front of it. The reason for this syntactic deviation from Standard ML is to make the code more consistent with other code in this thesis.

```
structure IntSet2 = IntSet1 :> sig
                                 type elem     = int
                                 type set
                                 val  empty    : set
                                 val  member   : elem → set → bool
                                 val  insert   : elem → set → set
                               end
```

Clearly, you cannot seal a structure with some arbitrary signature. The structure has to *match* the signature in some way; that is, all components specified by the signature must be defined in the structure in a compatible way. We formalize signature matching in Section 2.3; for now, the intuition suffices.

It is important that the signature in the preceding example equates the elem type with int. Otherwise, elem would be abstract and we could not use the IntSet_2 structure at all. Type components like elem are called *transparent* because the definition of the type component in the underlying structure shines through the signature. Conversely, type components like set are called *opaque*. Signatures that contain only transparent (opaque) type components are also called transparent (opaque), whereas signatures that contain both transparent and opaque type components (like the signature in the preceding example) are called *translucent*.

2.1.3. Functors

Suppose we not only need sets of integers but also sets of (say) strings, and want to reuse the code of the IntSet_1 structure without duplicating it. We can do so by defining a function from structures to structures that creates a set implementation for every type that can be compared for equality. Such a function is called a *functor*.

```
functor MkSet (E : sig type t val eq : t → t → bool end) =
  struct
    type elem     = E.t
    type set      = list elem
    val  empty    = []
    val  member   = λx . λs . exists (λy . E.eq x y) s
    val  insert   = λx . λs . if member x s then s else (cons x s)
  end :> sig
           type elem     = E.t
           type set
           val  empty    : set
           val  member   : elem → set → bool
           val  insert   : elem → set → set
         end
```

The argument of the MkSet functor provides the types and values necessary to define the set implementation in the functor body. The signature for the argument is mandatory. We can now apply the MkSet functor to an appropriate argument

in order to get a working set implementation for strings. As with signature sealing, the actual argument of a functor application must match the signature of the argument in the functor definition.

structure StringEq =
  struct
    type t = string
    val eq = λs . λs′ . primStringEq s s′
  end

structure StringSet = MkSet (StringEq)

Functors in Standard ML are *generative*; that is, abstract types obtained by two different functor invocations are incompatible, even if the arguments used for the two invocations are compatible. Suppose we invoke the MkSet functor again:

structure StringSet′ = MkSet (StringEq)

The types StringSet.set and StringSet′.set are now incompatible, so an expression like StringSet.insert "ML" StringSet′.empty is not type correct.

2.1.4. Summary

In this section, we introduced four features essential for proper modularization and demonstrated how ML modules provide these features:

- Structures provide a mechanism for namespace management.
- Signatures provide a mechanism for abstraction.
- Functors provide a mechanism for code reuse.
- Incremental compilation is possible in Standard ML (as well as in Tiny-ML, defined in Section 2.3) because the signature language is powerful enough to describe the type of a structure exactly; that is, signatures are *fully syntactic* [Sha99].

Separate compilation is not possible with the module system of Standard ML because there is no way to fully specify the import list of a structure (i.e., the signatures of other structures and functors that are used inside the structure). The reason for this deficiency is that functors are not higher-order [MT94] in Standard ML; hence, we cannot parameterize over functors used inside a structure. However, several proposals exist that make separate compilation possible for ML [Ler94, HL94, Sha99].

2.2. General conventions and definitions

Before we define the first formal module calculus in Section 2.3, we need to agree on some general conventions and definitions. The conventions and definitions

presented in this section are used throughout the rest of the text. We first define a convenient notation for writing sequences of elements.

Definition 2.1 (Overbar). The notation $\overline{x}^n$ is short for $x_1, \ldots, x_n$. Furthermore, $\overline{x_y}^{y \in M}$ is an abbreviation for $x_{y_1}, \ldots, x_{y_n}$ where $M = \{y_1, \ldots y_n \mid n \in \mathbb{N}\}$. The syntactic element separating the items of the enumeration might be different from the comma used in this definition. It is always clear from the context which separator should be used.

Next, we define some notation dealing with sets.

Definition 2.2 (Sets). The notation $[n]$ denotes the set $\{1, \ldots, n\}$. The set of all finite subsets of a set M is written $\mathrm{Fin}(M) := \{N \mid N \subseteq M, N \text{ finite}\}$. $M \dot\cup N$ denotes the disjoint union of M and N, which implicitly assumes that $M \cap N = \emptyset$.

A finite map is a finite set of tuples that can be treated like a function. We now give a definition of finite maps and of various operations on finite maps.

Definition 2.3 (Finite maps). A finite map between two sets M and N is a finite set of tuples $F = \{(a, b) \mid a \in M, b \in N\}$ such that for $(a, b) \in F$ there is no $N \ni b' \neq b$, with $(a, b') \in F$. We often write $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ instead of $\{(a_1, b_1), \ldots (a_n, b_n)\}$. The set of all finite maps between M and N is written $M \xrightarrow{\text{fin}} N$. In addition to the regular set operations $\subseteq$, $=$, $\cap$, and $\setminus$, we define the following operations on finite maps $F, G \in (M \xrightarrow{\text{fin}} N)$:

$$
\begin{aligned}
F(a) &:= b \text{ if } (a, b) \in F && \textit{Element access} \\
\mathrm{Dom}(F) &:= \{a \in M \mid (a, b) \in F \text{ for some } b \in N\} && \textit{Domain of } F \\
\mathrm{Img}(F) &:= \{b \in N \mid (a, b) \in F \text{ for some } a \in M\} && \textit{Image of } F \\
F \dot\cup G &:= F \cup G \text{ if } \mathrm{Dom}(F) \cap \mathrm{Dom}(G) = \emptyset && \textit{Disjoint merge} \\
F \overrightarrow{\cup} G &:= \{(a, b) \in F \mid a \notin \mathrm{Dom}(G)\} \cup G && \textit{Right merge} \\
F, a \mapsto b &:= F \overrightarrow{\cup} \{(a, b)\} && \textit{Singleton merge} \\
\mathrm{zip}(\{\overline{a}^m\}, \{\overline{b}^n\}) &:= \{\overline{a_i \mapsto b_i}^{i \in [\min(m,n)]}\} && \textit{Zipping}
\end{aligned}
$$

2.3. Tiny-ML

This section formalizes the ML module calculus Tiny-ML, a simplified version of Russo's Mini-SML [Rus98], which is in turn a simplified version of Standard ML's module system [MTHM97]. Tiny-ML is not meant to be a new formalization of the ML module system; its sole purpose lies in its role as the source language for the translation from ML modules to Haskell type classes.

Tiny-ML is a subset (modulo some minor syntactic differences) of Mini-SML. It does not support the following Mini-SML features: nested structures, parameterizable type components, arbitrary structure expressions as functor arguments

and functor bodies, weak sealing, and data types.[3] These features have been omitted to keep the translation from Tiny-ML to Haskell type classes manageable (see Section 4.4 for details).

The formalization of Tiny-ML proceeds in four steps. First, we define the syntax of Tiny-ML in Section 2.3.1. Then, in Section 2.3.2, we define semantic objects, which play the role of types in the static semantics. Section 2.3.3 contains some more definitions needed to formalize the type system. Last but not least, Section 2.3.4 defines the typing judgments for Tiny-ML. All material in the following four sections is heavily inspired by the formalization of Mini-SML in Russo's dissertation [Rus98].

2.3.1. Syntax

Tiny-ML, like Mini-SML, is divided into a *core language* and a *module language*. The next two sections describe the syntax of the core and the module language. We let O range over all syntactic constructs.

Core language

The core language is an implicitly typed language; its syntax is shown in Figure 2.1. We assume the existence of three countably infinite and disjoint identifiers sets. Every type constructors $\mathrm{T} \in \mathrm{TyconId}$ is equipped with a *kind* $\kappa \in \mathbb{N}$, denoting the arity of T. We write T^{κ} for a type constructor T of kind κ. This simple definition of kinds is sufficient because higher-order types (like for example in System F_{ω} [Gir72]) are neither supported in Tiny-ML nor in Standard ML.

The type language of Tiny-ML distinguishes between simple types and value types. Simple types may contain type components of structures, written t and $\mathrm{X.t}$. Here, X and t denote structure and type identifiers, respectively, which are part of the module language introduced in the next section. A value type is a universally quantified simple type; we often write u for the value type $\mathrm{v} = \forall\emptyset.\mathrm{u}$.

The syntax of value expressions is standard. Expressions can refer to value components of structures via the notation x and $\mathrm{X.x}$, where x denotes a value identifier of the module language.

Module language

The explicitly typed module language supports structures, signatures, and functors as introduced in Section 2.1; its syntax is shown in Figure 2.2. We assume the existence of four countably infinite and disjoint identifiers sets.

The syntax of signature bodies and expressions, of structure bodies, and of programs is straightforward. More interesting is the syntax of structure expressions. We distinguish between four different forms of structure expressions:

[3] Mini-SML does not support data types directly, but they can be simulated with nested structures.

Figure 2.1. Syntax of Tiny-ML's core language

Identifiers

$\text{'a} \in \text{SimTypVar}$	simple type variables
$\text{c} \in \text{CoreId}$	core identifiers
$\text{T} \in \text{TyconId} = \{\rightarrow, \text{int}, \dots\}$	type constructor identifiers

Types

$\text{SimTyp} \ni \text{u} ::=$	'a	simple type variable
	$\mid \text{T}^{\kappa}\ \overline{\text{u}}^{\kappa}$	type constructor application
	$\mid \text{t}$	unqualified type occurrence
	$\mid \text{X.t}$	qualified type occurrence
$\text{ValTyp} \ni \text{v} ::=$	$\forall \text{A.u}$	value type
$\text{A, B} \in$	$\text{Fin}(\text{SimTypVar})$	set of simple type variables

Value expressions

$\text{ValExp} \ni \text{e} ::=$	c	core identifier
	$\mid \lambda\text{c.e}$	λ-abstraction
	$\mid \text{e e}$	application
	$\mid \textbf{let}\ \text{c} = \text{e}\ \textbf{in}\ \text{e}$	let-binding
	$\mid \text{x}$	unqualified value occurrence
	$\mid \text{X.x}$	qualified value occurrence

Figure 2.2. Syntax of Tiny-ML's module language

Identifiers

$t \in \mathrm{TypId}$	type identifiers
$x, y \in \mathrm{ValId}$	value identifiers
$X \in \mathrm{StrId}$	structure identifiers
$F \in \mathrm{FunId}$	functor identifiers

Signature bodies

$\mathrm{SigBod} \ni B ::=$	**type** t; B	opaque type specification
$\mid$	**type** $t = u$; B	transparent type specification
$\mid$	**val** $x : v$; B	value specification
$\mid$	ϵ_B	empty body

Signature expressions

$\mathrm{SigExp} \ni S ::=$	**sig** B **end**	encapsulated body

Structure bodies

$\mathrm{StrBod} \ni b ::=$	**type** $t = u$; b	type definition
$\mid$	**val** $x = e$; b	value definition
$\mid$	ϵ_b	empty body

Structure expressions

$\mathrm{PStrExp} \ni ps ::=$	**struct** b **end**	encapsulated body
$\mathrm{StrExp} \ni s ::=$	ps	primitive structure expression
$\mid$	X	structure identifier
$\mid$	$F(\overline{X})$	functor application
$\mathrm{PStrExp}_{:>} \ni ps_{:>} ::=$	ps	unsealed primitive structure expression
$\mid$	$ps :> S$	sealed primitive structure expression
$\mathrm{StrExp}_{:>} \ni s_{:>} ::=$	s	unsealed ordinary structure expression
$\mid$	$s :> S$	sealed ordinary structure expression

Programs

$\mathrm{Prog} \ni prog ::=$	**structure** $X = s_{:>}$; $prog$	structure definition
$\mid$	**functor** $F(\overline{X : S}) = ps_{:>}$; $prog$	functor definition
$\mid$	ϵ_{prog}	empty program

- Primitive structure expressions $\mathrm{ps} \in \mathrm{PStrExp}$
- Ordinary structure expressions $\mathrm{s} \in \mathrm{StrExp}$
- Sealed primitive structure expressions $\mathrm{ps}_{:>} \in \mathrm{PStrExp}_{:>}$
- Sealed structure expressions $\mathrm{s}_{:>} \in \mathrm{StrExp}_{:>}$

The hierarchic nesting of structure expressions may seem unnecessary. In fact, Mini-SML does not distinguish between different forms of structure expressions; instead, a much simpler grammar with only one hierarchy level is used. We need the hierarchically nested grammar in Tiny-ML because the translation from ML modules to Haskell type classes requires a distinction between the different forms of structure expressions.

2.3.2. Semantic objects

The definition of Standard ML [MTHM97] distinguishes between syntactic types and their semantic counterparts, which are called *semantic objects*. Mini-SML and Tiny-ML follow the same approach. Figure 2.3 gives the definition of semantic objects. We let $\mathcal{O}$ range over all semantic objects. In order to distinguish between syntactic constructs and semantic objects, we use a $\mathrm{roman\ font}$ for syntactic constructs and an $\mathit{italic\ font}$ for semantic objects.

Semantic objects introduce two new sorts of type variables, which are taken from countably infinite and disjoint identifier sets. Semantic simple type variables $'a \in \mathit{SimTypVar}$ are the semantic counterpart of (syntactic) simple type variables $'\mathrm{a}$. Semantic type variables $\alpha \in \mathit{TypVar}$ have no real syntactic counterpart. They represent abstract or unknown types introduced by opaque type specifications in signatures.

The semantic objects for the core language correspond directly to simple and value types, except that type occurrences t and $\mathrm{X.t}$ are now represented by the right-hand sides of their definitions (if available) or by semantic type variables. Note that we often write just u for a semantic value type $v = \forall\emptyset.u$.

Semantic structures $\mathcal{S}$ are finite maps consisting of two parts $\mathcal{S}_{\mathrm{t}}$ and $\mathcal{S}_{\mathrm{x}}$, which record the semantic objects for the type and value components of structure bodies, respectively. We omit the subscript used to distinguish the two parts when it is clear from context which part should be used. For example, $\mathrm{x} \in \mathrm{Dom}(\mathcal{S})$ ranges only over the value identifiers of $\mathrm{Dom}(\mathcal{S}_{\mathrm{x}})$.

Existential semantic structures $\exists P.\mathcal{S}$ are the types of structure expressions; the existentially quantified semantic type variables in P represent the abstract types introduced by a structure expression. Note that we sometimes write $\mathcal{S}$ instead of $\exists\emptyset.\mathcal{S}$.

Semantic signatures $\Lambda P.\mathcal{S}$ are the semantic counterpart of signature expressions. The variables in P stem from opaque type specifications in the signature expression. As with existential semantic structures, we sometimes write $\mathcal{S}$ instead of $\Lambda\emptyset.S$.

Figure 2.3. Semantic objects for Tiny-ML

Identifiers

$$
\begin{array}{rr}
{}'a \in \mathit{SimTypVar} & \text{semantic simple type variables} \\
\alpha \in \mathit{TypVar} & \text{semantic type variables}
\end{array}
$$

Semantic objects for the core language

$$
\begin{array}{rll}
\mathit{SimTyp} \ni u & ::= {}'a & \text{semantic simple type variable} \\
& \mid \; T^{\kappa}\,\overline{u}^{\kappa} & \text{semantic type constructor application} \\
& \mid \; \alpha & \text{semantic type variable} \\
\mathit{ValTyp} \ni v & ::= \forall A.u & \text{semantic value type} \\
A, B & \in \mathrm{Fin}(\mathit{SimTypVar}) & \text{set of semantic simple type variables}
\end{array}
$$

Semantic objects for the module language

$$
\mathit{Str} \ni \mathcal{S} := \left\{ \mathcal{S}_{\mathrm{t}} \cup \mathcal{S}_{\mathrm{x}} \;\middle|\; \begin{array}{l} \mathcal{S}_{\mathrm{t}} \in \mathrm{TypId} \xrightarrow{\text{fin}} \mathit{SimTyp}, \\ \mathcal{S}_{\mathrm{x}} \in \mathrm{ValId} \xrightarrow{\text{fin}} \mathit{ValTyp}, \end{array} \right\} \qquad \text{semantic structure}
$$

$$
\begin{array}{rll}
\mathit{ExStr} \ni \mathcal{X} & ::= \exists P.\mathcal{S} & \text{existential semantic structure} \\
\mathit{Sig} \ni \mathcal{L} & ::= \Lambda P.\mathcal{S} & \text{semantic signature} \\
\mathit{Fun} \ni \mathcal{F} & ::= \forall P.\overline{\mathcal{S}} \to \mathcal{X} & \text{semantic functor} \\
P, Q & \in \mathrm{Fin}(\mathit{TypVar}) & \text{set of semantic type variables}
\end{array}
$$

Contexts

$$
\mathit{CoreContext} \ni C := \left\{ C_{\mathrm{c}} \cup C_{\mathrm{a}} \;\middle|\; \begin{array}{l} C_{\mathrm{c}} \in \mathrm{CoreId} \xrightarrow{\text{fin}} \mathit{ValTyp}, \\ C_{\mathrm{a}} \in \mathrm{SimTypVar} \xrightarrow{\text{fin}} \mathit{SimTyp} \end{array} \right\}
$$

$$
\mathit{Context} \ni \mathcal{C} := \left\{ \begin{array}{l} C \cup \mathcal{C}_{\mathrm{t}} \cup \\ \mathcal{C}_{\mathrm{x}} \cup \mathcal{C}_{\mathrm{X}} \cup \mathcal{C}_{\mathrm{F}} \end{array} \;\middle|\; \begin{array}{l} C \in \mathit{CoreContext}, \\ \mathcal{C}_{\mathrm{t}} \in \mathrm{TypId} \xrightarrow{\text{fin}} \mathit{SimTyp}, \\ \mathcal{C}_{\mathrm{x}} \in \mathrm{ValId} \xrightarrow{\text{fin}} \mathit{ValTyp}, \\ \mathcal{C}_{\mathrm{X}} \in \mathrm{StrId} \xrightarrow{\text{fin}} \mathit{Str}, \\ \mathcal{C}_{\mathrm{F}} \in \mathrm{FunId} \xrightarrow{\text{fin}} \mathit{Fun} \end{array} \right\}
$$

Semantic functors $\forall P.\overline{\mathcal{S}} \to \mathcal{X}$ are the types of functors. They are universally quantified because functors are polymorphic in the opaque type components of their argument signatures.

Two different sorts of contexts are used to record information gathered during the typing process. A core context records information about core-language entities, whereas a context records information about core and module-language entities. As with semantic structures, we often omit the subscript used to distinguish to different parts of a (core) context when it is clear which part we mean.

We finish this section by remarking that we identify semantic objects up to renaming of bound variables.

2.3.3. Some definitions

Some more definitions are necessary before we can develop the typing judgments for Tiny-ML. We first define substitutions of semantic type variables and semantic simple type variables.

Definition 2.4 (Substitutions). A substitution φ from semantic type variables to semantic simple types is an element of $TypVar \xrightarrow{\text{fin}} SimTyp$. A substitution ϕ from semantic simple type variables to semantic simple types is an element of $SimTypVar \xrightarrow{\text{fin}} SimTyp$. We often write $[\overline{u_i/\alpha_i}]$ for a substitution $\varphi = \{\overline{\alpha_i \mapsto u_i}\}$ and $[\overline{u_i/'a_i}]$ for a substitution $\phi = \{\overline{'a_i \mapsto u_i}\}$. The application of a substitution is defined in the usual, capture-avoiding way.

We often need to relate semantic value types to semantic simple types. An important relation between them is generalization:

Definition 2.5 (Generalization of semantic simple types). A semantic value type $v = \forall A.u$ generalizes a semantic simple type u', written $v \succ u'$ if, and only if, there is a substitution ϕ with $\mathrm{Dom}(\phi) = A$ such that $\phi(u) = u'$.

The enrichment relation between semantic structures, which is defined next, plays an important role in deciding whether or not a structure matches a signature. Intuitively, a semantic structure $\mathcal{S}$ enriches a semantic structure $\mathcal{S}'$ if all type components of $\mathcal{S}'$ are defined identically in $\mathcal{S}$, and all value components of $\mathcal{S}'$ have a more polymorphic counterpart in $\mathcal{S}$. Formally, we define enrichment as follows:

Definition 2.6 (Enrichment). A semantic value type v enriches another semantic value type v', written $v \succcurlyeq v'$ if, and only if, for every semantic simple type u, $v \succ u$ whenever $v' \succ u$. The enrichment relation for semantic structures is defined as the least relation closed under the following rule:

$$\frac{\mathrm{Dom}(\mathcal{S}) \supseteq \mathrm{Dom}(\mathcal{S}') \qquad \mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t}) \text{ for all } \mathrm{t} \in \mathrm{Dom}(\mathcal{S}') \qquad \mathcal{S}(\mathrm{x}) \succcurlyeq \mathcal{S}'(\mathrm{x}) \text{ for all } \mathrm{x} \in \mathrm{Dom}(\mathcal{S}')}{\mathcal{S} \succcurlyeq \mathcal{S}'}$$

Enrichment is a pre-order closed under substitution [Mil78, Rus98].

The following lemma gives a different but equivalent formulation of enrichment for semantic value types. The alternative formulation coincides with the definition of generic instances in the article by Damas and Milner [DM82].

Lemma 2.7 (Equivalent formulation of value type enrichment). $\forall A.u \succcurlyeq \forall A'.u'$ *if, and only if,* $\mathrm{FV}^{'\alpha}(\forall A.u) \cap A' = \emptyset$ *and there exists a substitution* ϕ *with* $\mathrm{Dom}(\phi) \subseteq A$ *such that* $\phi(u) = u'$.

Proof. Straightforward. □

Signature matching was already introduced informally in Section 2.1. We can now define it formally.

Definition 2.8 (Signature matching). A semantic structure $\mathcal{S}$ matches a semantic signature $\mathcal{L} = \Lambda P.\mathcal{S}'$ if, and only if, there exists a substitution φ with $\mathrm{Dom}(\varphi) = P$ such that $\mathcal{S} \succcurlyeq \varphi(\mathcal{S}')$.

The last definition in this section specifies notations for denoting free variables.

Definition 2.9 (Free variables). The set of semantic type variables free in some semantic object $\mathcal{O}$ is written $\mathrm{FV}^{\alpha}(\mathcal{O}) \subseteq \mathit{TypVar}$. Similarly, the set of semantic simple type variables free in $\mathcal{O}$ is written $\mathrm{FV}^{'\alpha}(\mathcal{O}) \subseteq \mathit{SimTypVar}$. We write the set of structure variables free in some syntactic construct O as $\mathrm{FV}^{\mathrm{X}}(\mathrm{O}) \subseteq \mathrm{StrId}$. Similarly, $\mathrm{FV}^{\mathrm{c}}(\mathrm{O}) \subseteq \mathrm{CoreId}$ denotes the set of core variables free in O. The notion of free is defined in the usual way.

2.3.4. Typing judgments

The typing judgments for Tiny-ML come mainly in two different flavors. *Denotation judgments* are written $\mathcal{C} \vdash \mathrm{O} \triangleright \mathcal{O}$ for some syntactic object O and some semantic object $\mathcal{O}$; they relate simple types, value types, and signatures to semantic objects. *Classification judgments* have the form $\mathcal{C} \vdash \mathrm{O} : \mathcal{O}$; they assign semantic objects to value expressions and structure expressions.

Core language

The typing judgments for Tiny-ML's core language are shown in Figure 2.4. Rule (valtyp) uses the function zip from Definition 2.3 on page 9 to map quantified simple type variables to fresh semantic simple type variables. The generalization relation from Definition 2.5 on the previous page is used in rules (exp_{id}), (exp_{mod1}), and (exp_{mod2}) to instantiate the value type found in the context. The condition $\mathrm{FV}^{\alpha}(u) \subseteq \mathrm{FV}^{\alpha}(\mathcal{C})$ in rule (exp_{poly}) ensures that the type assigned to an expression contains only semantic type variables from the context. It does not affect the typability of an expression because semantic type variables not free in the context could be replaced by (say) semantic simple type variables.

Figure 2.4. Typing judgments for Tiny-ML's core language

Denotation of simple types $\boxed{\mathcal{C} \vdash \mathrm{u} \triangleright u}$

$$\frac{\mathcal{C}(\text{`a}) = u}{\mathcal{C} \vdash \text{`a} \triangleright u}\ (simtyp_{var}) \qquad \frac{\overline{\mathcal{C} \vdash \mathrm{u}_i \triangleright u_i}^{i \in [\kappa]}}{\mathcal{C} \vdash \mathrm{T}^{\kappa}\ \overline{\mathrm{u}}^{\kappa} \triangleright T^{\kappa}\ \overline{u}^{\kappa}}\ (simtyp_{tycon}) \qquad \frac{\mathcal{C}(\mathrm{t}) = u}{\mathcal{C} \vdash \mathrm{t} \triangleright u}\ (simtyp_{mod1})$$

$$\frac{\mathcal{C}(\mathrm{X})(\mathrm{t}) = u}{\mathcal{C} \vdash \mathrm{X.t} \triangleright u}\ (simtyp_{mod2})$$

Denotation of value types $\boxed{\mathcal{C} \vdash \mathrm{v} \triangleright v}$

$$\frac{B \cap \mathrm{FV}^{\text{`}a}(\mathcal{C}) = \emptyset \qquad |B| = |\mathrm{A}| \qquad \mathcal{C} \overrightarrow{\cup}\ \mathrm{zip}(\mathrm{A}, B) \vdash \mathrm{u} \triangleright u}{\mathcal{C} \vdash \forall \mathrm{A.u} \triangleright \forall B.u}\ (valtyp)$$

Monomorphic classification of value expressions $\boxed{\mathcal{C} \vdash \mathrm{e} : u}$

$$\frac{\mathcal{C}(\mathrm{c}) = v \qquad v \succ u}{\mathcal{C} \vdash \mathrm{c} : u}\ (exp_{id}) \qquad \frac{\mathcal{C}(\mathrm{x}) = v \qquad v \succ u}{\mathcal{C} \vdash \mathrm{x} : u}\ (exp_{mod1})$$

$$\frac{\mathcal{C}(\mathrm{X})(\mathrm{x}) = v \qquad v \succ u}{\mathcal{C} \vdash \mathrm{X.x} : u}\ (exp_{mod2}) \qquad \frac{\mathcal{C}, \mathrm{c} \mapsto u \vdash \mathrm{e} : u'}{\mathcal{C} \vdash \lambda \mathrm{c.e} : u \to u'}\ (exp_{abs})$$

$$\frac{\mathcal{C} \vdash \mathrm{e}_1 : u_1 \to u_2 \qquad \mathcal{C} \vdash \mathrm{e}_2 : u_1}{\mathcal{C} \vdash \mathrm{e}_1\ \mathrm{e}_2 : u_2}\ (exp_{app}) \qquad \frac{\mathcal{C} \vdash \mathrm{e}_1 : v \qquad \mathcal{C}, \mathrm{c} \mapsto v \vdash \mathrm{e}_2 : u}{\mathcal{C} \vdash \textbf{let}\ \mathrm{c} = \mathrm{e}_1\ \textbf{in}\ \mathrm{e}_2 : u}\ (exp_{let})$$

Polymorphic classification of value expressions $\boxed{\mathcal{C} \vdash \mathrm{e} : v}$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u \qquad \mathrm{FV}^{\alpha}(u) \subseteq \mathrm{FV}^{\alpha}(\mathcal{C}) \qquad A = \mathrm{FV}^{\text{`}a}(u) \setminus \mathrm{FV}^{\text{`}a}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{e} : \forall A.u}\ (exp_{poly})$$

Figure 2.5. Typing judgments for signatures and structures in Tiny-ML

Denotation of signature bodies $\boxed{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}$

$$\frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \qquad \mathcal{C}, \mathrm{t} \mapsto u \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S} \qquad P \cap \mathrm{FV}^{\alpha}(u) = \emptyset \qquad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{B} \triangleright \Lambda P.\mathcal{S}, \mathrm{t} \mapsto u}\ (\mathit{sigb}_{t=})$$

$$\frac{\mathcal{C}, \mathrm{t} \mapsto \alpha \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S} \qquad \alpha \notin \mathrm{FV}^{\alpha}(\mathcal{C}) \cup P \qquad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t}; \mathrm{B} \triangleright \Lambda(\{\alpha\} \cup P).\mathcal{S}, \mathrm{t} \mapsto \alpha}\ (\mathit{sigb}_{t})$$

$$\frac{\mathcal{C} \vdash \mathrm{v} \triangleright v \qquad \mathcal{C}, \mathrm{x} \mapsto v \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S} \qquad P \cap \mathrm{FV}^{\alpha}(v) = \emptyset \qquad \mathrm{x} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{val}\ \mathrm{x} : \mathrm{v}; \mathrm{B} \triangleright \Lambda P.\mathcal{S}, \mathrm{x} \mapsto v}\ (\mathit{sigb}_{v}) \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{B}} \triangleright \Lambda\emptyset.\emptyset}\ (\mathit{sigb}_{\epsilon})$$

Denotation of signature expressions $\boxed{\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}}$

$$\frac{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}{\mathcal{C} \vdash \textbf{sig}\ \mathrm{B}\ \textbf{end} \triangleright \mathcal{L}}\ (\mathit{sigexp})$$

Classification of structure bodies $\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{S}}$

$$\frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \qquad \mathcal{C}, \mathrm{t} \mapsto u \vdash \mathrm{b} : \mathcal{S} \qquad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b} : \mathcal{S}, \mathrm{t} \mapsto u}\ (\mathit{strb}_{t})$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : v \qquad \mathcal{C}, \mathrm{x} \mapsto v \vdash \mathrm{b} : \mathcal{S} \qquad \mathrm{x} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} : \mathcal{S}, \mathrm{x} \mapsto v}\ (\mathit{strb}_{v}) \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \emptyset}\ (\mathit{strb}_{\epsilon})$$

Classification of structure expressions $\boxed{\begin{array}{r}\mathcal{C} \vdash \mathrm{ps} : \mathcal{S}, \mathcal{C} \vdash \mathrm{s} : \mathcal{X} \\ \mathcal{C} \vdash \mathrm{ps}_{:>} : \mathcal{X}, \mathcal{C} \vdash \mathrm{s}_{:>} : \mathcal{X}\end{array}}$

$$\frac{\mathcal{C} \vdash \mathrm{b} : \mathcal{S}}{\mathcal{C} \vdash \textbf{struct}\ \mathrm{b}\ \textbf{end} : \mathcal{S}}\ (\mathit{strexp}_{\mathit{struct}}) \qquad \frac{\mathcal{C}(\mathrm{X}) = \mathcal{S}}{\mathcal{C} \vdash \mathrm{X} : \exists\emptyset.\mathcal{S}}\ (\mathit{strexp}_{\mathit{var}})$$

$$\frac{\overline{\mathcal{C}(\mathrm{X}_i) = \mathcal{S}_i}^{\,i \in [n]} \qquad \mathcal{C}(\mathrm{F}) = \forall Q.\overline{\mathcal{S}'}^{\,n} \to \mathcal{X}' \qquad \overline{\mathcal{S}_i \succcurlyeq \varphi(\mathcal{S}'_i)}^{\,i \in [n]} \qquad \mathrm{Dom}(\varphi) = Q \qquad \varphi(\mathcal{X}') = \mathcal{X}}{\mathcal{C} \vdash \mathrm{F}(\overline{\mathrm{X}}^{n}) : \mathcal{X}}\ (\mathit{strexp}_{\mathit{fapp}})$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P'.\mathcal{S}' \qquad P \cap \mathrm{FV}^{\alpha}(\Lambda P'.\mathcal{S}') = \emptyset \\ \mathcal{S} \succcurlyeq \varphi(\mathcal{S}') \qquad \mathrm{Dom}(\varphi) = P'\end{array}}{\mathcal{C} \vdash \mathrm{s} :> \mathrm{S} : \exists P'.\mathcal{S}'}\ (\mathit{strexp}_{\mathit{sealed}})$$

Module language

The typing judgments for Tiny-ML's module language are shown in Figure 2.5 (signature and structures) and in Figure 2.6 (programs). Rule $(sigb_t)$ is interesting because this is the place where fresh semantic type variables are added to the parameters of a semantic signature. Rules $(strexp_{fapp})$ and $(strexp_{sealed})$ use the enrichment relation $\succcurlyeq$ from Definition 2.6 to ensure signature matching. The rules $(sigb_{t=})$, $(sigb_v)$, $(strexp_{sealed})$, $(prog_{str})$, and $(funargs)$ have premises of the form $P \cap \mathrm{FV}^\alpha(\mathcal{O}) = \emptyset$ where $\mathcal{O}$ is some semantic object and P contains bound semantic type variables. These premises ensure that the semantic type variables in P cannot be intermixed with existing semantic type variables. We can always rename the variables bound in P so as to fulfill the premises.

The four classification judgments for the four different forms of structure expressions are specified by only five rules. This is possible because we can interpret the structure expression in the conclusion of a rule in different ways. For example, the structure expression **struct** b **end** in the conclusion of rule $(strexp_{struct})$ can be interpreted either as a primitive structure expression, a primitive sealed structure expression, an ordinary structure expression, or as a sealed ordinary structure expression. In all cases except the first, the semantic structure $\mathcal{S}$ in the conclusion of the rule must be interpreted as the existential semantic structure $\exists\emptyset.\mathcal{S}$.

Figure 2.6. Typing judgment for Tiny-ML programs

Programs $\boxed{\mathcal{C} \vdash \mathrm{prog}}$

$$\frac{P \cap \mathrm{FV}^\alpha(\mathcal{C}) = \emptyset \qquad \mathcal{C} \vdash \mathrm{s}_{:>} : \exists P.\mathcal{S} \qquad \mathrm{X} \notin \mathrm{Dom}(\mathcal{C}) \qquad \mathcal{C}, \mathrm{X} \mapsto \mathcal{S} \vdash \mathrm{prog}}{\mathcal{C} \vdash \textbf{structure}\ \mathrm{X} = \mathrm{s}_{:>}; \mathrm{prog}}\ (prog_{str})$$

$$\frac{\mathcal{C} \overset{\text{funargs}}{\vdash} \overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]} \triangleright \forall P.\overline{\mathcal{S}}^n \qquad \mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i\in[n]} \vdash \mathrm{ps}_{:>} : \mathcal{X} \qquad \mathrm{F} \notin \mathrm{Dom}(\mathcal{C}) \qquad \mathcal{C}, \mathrm{F} \mapsto (\forall P.\overline{\mathcal{S}}^n \to \mathcal{X}) \vdash \mathrm{prog}}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i\in[n]}) = \mathrm{ps}_{:>}; \mathrm{prog}}\ (prog_{fun})$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{prog}}}\ (prog_\epsilon)$$

Functor arguments $\boxed{\mathcal{C} \overset{\text{funargs}}{\vdash} \overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]} \triangleright \forall P.\overline{\mathcal{S}}^n}$

$$\frac{P_i \cap \mathrm{FV}^\alpha(\mathcal{C}) = \emptyset^{i\in[n]} \qquad \overline{\mathcal{C}, \overline{\mathrm{X}_j \mapsto \mathcal{S}_j}^{j\in[i-1]} \vdash \mathrm{S_i} \triangleright \wedge P_i.\mathcal{S}_i}^{i\in[n]} \qquad P_i \cap P_j = \emptyset^{i\neq j\in[n]} \qquad P = \cup_{i\in[n]} P_i}{\mathcal{C} \overset{\text{funargs}}{\vdash} \overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]} \triangleright \forall P.\overline{\mathcal{S}}^n}\ (funargs)$$

2.4. Tiny-ML$^+$

Tiny-ML is not suitable as the target language for the translation from Haskell type classes to ML modules because it misses certain features that are required for the translation or that make the presentation of the translation more readable. Therefore, we extend Tiny-ML with the features required and call the resulting language Tiny-ML$^+$. All extensions in Tiny-ML$^+$ are well-established and implemented in Moscow ML [RRK$^+$03], some are even part of Standard ML. The extensions are the following:

- First-class structures (not part of Standard ML, see [Rus00a])
- Recursive functors (not part of Standard ML, see [CHP99, Rus01])
- Lexically scoped type variables (part of Standard ML)
- Arbitrary structure expressions as functor arguments (part of Standard ML)
- Signature expressions with type realizations; that is, signature expressions of the form S **where type** $\mathrm{t} = \mathrm{u}$ (part of Standard ML)

We did not integrate these features into Tiny-ML because they are either not part of Standard ML (first-class structures and recursive functors), or they would complicate the translation from Tiny-ML to Haskell type classes without adding much to the comparison between ML modules and Haskell type classes; after all, lexically scoped type variables are not part of the module language, and the two remaining extensions would be only syntactic sugar for Tiny-ML.

We now discuss every extension separately. The syntactic changes between Tiny-ML and Tiny-ML$^+$ are displayed in Figure 2.7, changes to the semantic objects are shown in Figure 2.8, and the additional typing rules and typing judgments for Tiny-ML$^+$'s core and module language can be found in Figure 2.9 and Figure 2.10, respectively. We use the same symbols for Tiny-ML and Tiny-ML$^+$; this does not cause any problems because the rest of this chapter and the whole Chapter 5 uses Tiny-ML$^+$exclusively, whereas Chapter 4 uses only Tiny-ML.

2.4.1. First-class structures

The core and the module language of Tiny-ML are *stratified* in the sense that structures cannot be manipulated in the same way as ordinary values of the core language. First-class structures remove this stratification. The extension presented here is directly taken from Russo's work on first-class structures [Rus98, Rus00a, Rus00b].

The syntax of Tiny-ML$^+$ (Figure 2.7) contains three constructs in order to support first-class structures. Simple types $\mathrm{u} \in \mathrm{SimTyp}$ contain package types of the form <S>, which are used as the syntactic type of a first-class structure. Value

Figure 2.7. Syntax of Tiny-ML$^+$ (extends syntax in Figures 2.1 and 2.2)

Types

SimTyp $\ni$ u ::= ...		
	<S>	package type

Value expressions

ValExp $\ni$ e ::= ...		
	pack s **as** S	package introduction
	open e **as** X : S **in** e	package elimination
	let c : v = e **in** e	explicitly typed let-binding

Signature expressions

SigExp $\ni$ S ::= ...		
	S **where type** t = u	type realization

Structure bodies

StrBod $\ni$ b ::= ...		
	val x : v = e; b	explicitly typed value definition

Structure expressions

StrExp $\ni$ s ::= ...		
	$F(\bar{s})$	functor application with arbitrary arguments

Programs

Rfun $\ni$ rfun ::=	**functor** $F(\overline{X_i : S_i}^{i\in[n]}) : S = ps$	recursive functor definition
Rfuns $\ni$ rfuns ::=	rfun; rfuns	sequence of recursive functor definitions
	ϵ_{rfuns}	empty sequence
Prog $\ni$ prog ::=	...	
	rec rfuns; prog	recursive functor group

Figure 2.8. Semantic objects for Tiny-ML$^+$ (extend semantic objects in Figure 2.3)

Semantic objects for the core language

SimTyp $\ni$ *u* ::= ...		
	$<\mathcal{X}>$	semantic package type

Figure 2.9. Typing judgments for Tiny-ML$^+$'s core language
(extend typing judgments in Figure 2.4)

Denotation of simple types $\boxed{\mathcal{C} \vdash \mathrm{u} \triangleright u}$

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S}}{\mathcal{C} \vdash \text{<S>} \triangleright \text{<}\exists P.\mathcal{S}\text{>}} \; (simtyp_{pkg})^+$$

Monomorphic classification of value expressions $\boxed{\mathcal{C} \vdash \mathrm{e} : u}$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathrm{s} : \exists P'.\mathcal{S}' \qquad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \\ P' \cap \mathrm{FV}^{\alpha}(\Lambda P.\mathcal{S}) = \emptyset \qquad \mathcal{S}' \succcurlyeq \varphi(\mathcal{S}) \qquad \mathrm{Dom}(\varphi) = P\end{array}}{\mathcal{C} \vdash \textbf{pack}\ \mathrm{s}\ \textbf{as}\ \mathrm{S} : \text{<}\exists P.\mathcal{S}\text{>}} \; (exp_{pack})^+$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathrm{e} : \text{<}\exists P.\mathcal{S}\text{>} \qquad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \\ P \cap \mathrm{FV}^{\alpha}(\mathcal{C}) = \emptyset \qquad \mathcal{C}, \mathrm{X} \mapsto \mathcal{S} \vdash \mathrm{e}' : u \qquad P \cap \mathrm{FV}^{\alpha}(u) = \emptyset\end{array}}{\mathcal{C} \vdash \textbf{open}\ \mathrm{e}\ \textbf{as}\ \mathrm{X} : \mathrm{S}\ \textbf{in}\ \mathrm{e}' : u} \; (exp_{open})^+$$

$$\frac{\begin{array}{c}B \cap \mathrm{FV}'^{a}(\mathcal{C}) = \emptyset \qquad |\mathrm{A}| = |B| \qquad \mathcal{C}' = \mathcal{C} \overrightarrow{\cup} \mathrm{zip}(\mathrm{A}, B) \\ \mathcal{C}' \vdash \mathrm{u}_1 \triangleright u_1 \qquad \mathcal{C}' \vdash \mathrm{e}_1 : u_1 \qquad \mathcal{C}, \mathrm{c} \mapsto \forall B.u_1 \vdash \mathrm{e}_2 : u_2\end{array}}{\mathcal{C} \vdash \textbf{let}\ \mathrm{c} : \forall \mathrm{A}.\mathrm{u}_1 = \mathrm{e}_1\ \textbf{in}\ \mathrm{e}_2 : u_2} \; (exp_{let'})^+$$

expressions $\mathrm{e} \in \mathrm{ValExp}$ contain two constructs for package introduction and package elimination: **pack** s **as** S converts a structure expression s into a value expression of type <S>, and **open** e **as** X : S **in** e′ eliminates a value expression e of type <S> by binding the structure packaged inside e to the structure variable X in e′.

The new form of simple type <S> causes an extension to the language of semantic objects (Figure 2.8) and an additional rule to the denotation judgment for simple types (Figure 2.9): Semantic value types $u \in SimTyp$ can now have the form of a semantic package type <$\mathcal{X}$>, which is used by the new rule $(simtyp_{pkg})^+$ as the denotation of <S>. Moreover, the classification judgment for value expressions (Figure 2.9) is extended with the rules $(exp_{pack})^+$ and $(exp_{open})^+$ for typing package introduction and package elimination, respectively.

2.4.2. Recursive functors

In Tiny-ML, as well as in Standard ML, all structure and functor definitions must be strictly hierarchic; that is, no recursive definitions are allowed. We now discuss an extension based on Russo's recursive structures [Rus01] that allows us to define recursive functors. The design of this extension is oriented towards the usage of Tiny-ML$^+$ as the target language in the translation from type classes to modules.

The syntax of Tiny-ML$^+$ programs $\mathrm{prog} \in \mathrm{Prog}$ (Figure 2.7) contains a new form of top-level definition **rec** rfuns; prog, which introduces a group of recursive func-

Figure 2.10. Typing judgments for Tiny-ML$^+$'s module language (extend typing judgments in Figures 2.5 and 2.6)

Denotation of signature expressions $\boxed{\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}}$

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \qquad \mathcal{C} \vdash \mathrm{u} \triangleright u \qquad \mathcal{S}(\mathrm{t}) = \alpha \in P}{\mathcal{C} \vdash \mathrm{S}\ \mathbf{where\ type}\ \mathrm{t} = \mathrm{u} : \Lambda(P \setminus \{\alpha\}).[u/\alpha]\mathcal{S}}\ (sigexp_{patch})^+$$

Classification of structures bodies $\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{S}}$

$$\frac{\begin{array}{c} B \cap \mathrm{FV}^{'a}(\mathcal{C}) = \emptyset \qquad |\mathrm{A}| = |B| \qquad \mathcal{C}' = \mathcal{C} \overrightarrow{\cup}\ \mathrm{zip}(\mathrm{A}, B) \\ \mathcal{C}' \vdash \mathrm{u} \triangleright u \qquad \mathcal{C}' \vdash \mathrm{e} : u \qquad \mathcal{C}, \mathrm{x} \mapsto \forall B.u \vdash \mathrm{b} : \mathcal{S} \qquad \mathrm{x} \notin \mathrm{Dom}(\mathcal{S}) \end{array}}{\mathcal{C} \vdash \mathbf{var}\ \mathrm{x} : \forall \mathrm{A}.\mathrm{u} = \mathrm{e}; \mathrm{b}}\ (strb_{v'})^+$$

Classification of structure expressions $\boxed{\mathcal{C} \vdash \mathrm{s} : \mathcal{X}}$

$$\frac{\begin{array}{c} \overline{\mathcal{C} \vdash \mathrm{s}_i : \exists P_i.\mathcal{S}_i}^{i \in [n]} \qquad \mathcal{C}(\mathrm{F}) = \forall Q.\overline{\mathcal{S}'}^n \to \mathcal{X}' = \mathcal{F} \qquad \overline{\mathcal{S}_i \succcurlyeq \varphi(\mathcal{S}'_i)}^{i \in [n]} \\ \mathrm{Dom}(\varphi) = Q \qquad \varphi(\mathcal{X}') = \exists P.\mathcal{S} \qquad \overline{P_i \cap \mathrm{FV}^{\alpha}(\mathcal{F}) = \emptyset}^{i \in [n]} \end{array}}{\mathcal{C} \vdash \mathrm{F}(\overline{\mathrm{s}}^n) : \exists P \dot{\cup} \dot{\bigcup}_{i \in [n]} P_i.\mathcal{S}}\ (strexp_{fapp})^+$$

Denotation of recursive functors $\boxed{\mathcal{C} \vdash \mathrm{rfuns} \triangleright \mathcal{C}'}$

$$\frac{\begin{array}{c} \mathcal{C} \overset{\mathrm{funargs}}{\vdash} \overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]} \triangleright \forall Q.\overline{\mathcal{S}}^n \qquad \mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i \in [n]} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \\ \mathcal{C} \vdash \mathrm{rfuns} \triangleright \mathcal{C}' \qquad \mathrm{F} \notin \mathrm{Dom}(\mathcal{C}') \qquad \mathcal{F} = \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S} \end{array}}{\mathcal{C} \vdash \mathbf{functor}\ \mathrm{F}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]}) : \mathrm{S} = \mathrm{ps}; \mathrm{rfuns} \triangleright \mathcal{C}', \mathrm{F} \mapsto \mathcal{F}}\ (rfuns_{collect})^+$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{rfuns}} \triangleright \mathcal{C}}\ (rfuns_{collect_\epsilon})^+$$

Check of recursive functors $\boxed{\mathcal{C} \vdash \mathrm{rfuns}}$

$$\frac{\mathcal{C}(\mathrm{F}) = \forall Q.\overline{\mathcal{S}}^n \to \mathcal{X} \qquad \mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i \in [n]} \vdash \mathrm{s} : \mathcal{X} \qquad \mathcal{C} \vdash \mathrm{rfuns}}{\mathcal{C} \vdash \mathbf{functor}\ \mathrm{F}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]}) : \mathrm{S} = \mathrm{ps}; \mathrm{rfuns}}\ (rfuns_{check})^+$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{rfuns}}}\ (rfuns_{check_\epsilon})^+$$

Programs $\boxed{\mathcal{C} \vdash \mathrm{prog}}$

$$\frac{\mathcal{C} \vdash \mathrm{rfuns} \triangleright \mathcal{C}' \qquad \mathcal{C}' \vdash \mathrm{rfuns} \qquad \mathcal{C}' \vdash \mathrm{prog}}{\mathcal{C} \vdash \mathbf{rec}\ \mathrm{rfuns}; \mathrm{prog}}\ (prog_{rec})^+$$

tors. rfuns is a sequence of recursive functor definitions rfun of the form **functor** $\mathrm{F}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) : \mathrm{S} = \mathrm{s}$. The difference to an ordinary functor definition is the additional signature annotation : S, which specifies the expected signature of the functor body s. The signature S is used to generate forward declarations for all recursive functors of a group.

Consequently, the type system of Tiny-ML$^+$'s module language (Figure 2.10) adds a new rule $(prog_{rec})^+$ to the judgment $\mathcal{C} \vdash \mathrm{prog}$ for checking programs, which first collects the semantic functors of a group of recursively defined functors in the extended context $\mathcal{C}'$, and then uses these semantic functors as forward declarations to check the type correctness of every functor body in the group. Two new judgments are defined for collecting semantic functors ($\mathcal{C} \vdash \mathrm{rfuns} \triangleright \mathcal{C}'$) and for checking functor bodies ($\mathcal{C} \vdash \mathrm{rfuns}$).

You might wonder how recursive functors can be used. After all, the module language prevents you syntactically from applying a functor inside a functor body. Hence, the only way to use recursive functors is through first-class structures of the core language, which we introduced in the preceding section. Here is an example that shows how recursive functors can encode polymorphic recursive functions; that is, functions which invoke themselves recursively at different types.

```
rec functor F (X : sig end) : sig val g : ∀{'a} . 'a → 'a end =
  struct
    val g = λx . open (pack F (struct end) as sig val g : ∀{'a} . 'a → 'a end)
                   as X : sig val g : ∀{'a} . 'a → 'a end
                   in let q = X.g true in let r = X.g 0 in x
  end
```

The recursive functor F defines a function g of the polymorphic type $\forall\{\text{'a}\}.\text{'a} \to \text{'a}$. Inside the body of g, the result of applying F to the empty structure is packaged as a first-class structure. We immediately unpack this first-class structure and bind the result to the structure variable X. Now we can invoke X.g at different types.

Recursive structures and functors may cause problems by introducing recursion on the type level. The extension for recursive functors proposed here does not allow definitions of recursive types because recursive functors can be used only on the value level, as demonstrated in the preceding example. You may also be concerned about the well-foundness of recursive functors. Here, well-foundness means that the definition of a recursive functor is evaluated without accessing one of the recursively defined functor variables. Although we have not defined a dynamic semantics for Tiny-ML$^+$, it should be clear that this cannot happen because the body of a functor is not evaluated until the functor is applied to some argument(s).

2.4.3. Lexically scoped type variables

Lexically scoped type variables (a feature of Standard ML, see also [PS04a]) are simple type variables that are introduced at an explicitly typed binding and can

be used in the subterm of the binding.[4] In some sense, the form of lexically scoped type variables found in ML and presented here resembles the explicit type-passing mechanism of System F [Gir72, Rey74].

In our case, the explicitly typed bindings are let-expressions and value specifications in structure expressions. Hence, the syntax of Tiny-ML$^+$ (Figure 2.7) extends the syntax of expressions $\mathrm{e} \in \mathrm{ValExp}$ with an explicitly typed let-binding of the form **let** $\mathrm{c} : \mathrm{v} = \mathrm{e}$ **in** e', and the syntax of structure bodies $\mathrm{b} \in \mathrm{StrBod}$ with an explicitly typed value definition of the form **val** $\mathrm{x} : \mathrm{v} = \mathrm{e}$; b. In both cases, v is the expected type of the expression e; the quantified simple type variables of v can be used inside e (e.g., in signature expressions for first-class structures).

The typing judgments for Tiny-ML$^+$ (Figures 2.9 and 2.10) contain two new rules in order to support these two syntactic constructs: Rule $(exp_{let'})^+$ extends the judgment $\mathcal{C} \vdash \mathrm{e} : u$ for classifying value expression, and rule $(strb_{v'})^+$ extends the judgment $\mathcal{C} \vdash \mathrm{b} : \mathcal{S}$ for classifying structure bodies.

2.4.4. Arbitrary functor arguments

In Tiny-ML, functors can be applied only to structure variables. Tiny-ML$^+$ removes this restriction by extending the language of structure expressions $\mathrm{s} \in \mathrm{StrExp}$ with a construct $\mathrm{F}(\bar{\mathrm{s}})$ (Figure 2.7). The typing judgment $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ is extended accordingly with a new rule $(strexp_{fapp})^+$ (Figure 2.10). The rule is similar to rule $(strexp_{fapp})$ for functor application in Tiny-ML (Figure 2.5 on page 18), we need only some extra side conditions that prevent semantic type variables of being intermixed accidentally.

2.4.5. Signature expressions with type realizations

Type realizations allow opaque type specifications in signature expressions to be turned into transparent type specifications. For example, the signature expression **sig type** t **end where type** $\mathrm{t} = \mathrm{int}$ with the type realization $\mathrm{t} = \mathrm{int}$ denotes the same semantic signature as the signature expression **sig type** $\mathrm{t} = \mathrm{int}$ **end**.

In order to support signature expressions with type realizations, the syntax of Tiny-ML$^+$ (Figure 2.7) extends signature expressions $\mathrm{S} \in \mathrm{SigExp}$ with a construct of the form S **where type** $\mathrm{t} = \mathrm{u}$. Additionally, a new rule $(sigexp_{patch})^+$ is added to the denotation judgment $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$ (Figure 2.10).

2.5. Related work

Mini-SML [Rus98], which is the basis of Tiny-ML, models all important features of Standard ML's module language [MTHM97]. It was developed to formulate the static semantics of Standard ML in a more type-theoretic, better comprehensible way. The static semantics of Standard ML's module language maintains a

[4] Standard ML supports also implicitly scoped type variables. We do not consider this variant here.

set of semantic type variables to keep track of all semantic type variables generated in a typing derivation. This set is threaded through the derivation tree in a global, state-like manner. Russo replaces this *global* generativity with a more type-theoretic concept, namely that of existential quantification, which can be seen as *local* generativity. Russo claims that his "state-less semantics provides a better conceptual understanding of the type structure of Standard ML" [Rus99]. Russo's formalization [Rus99] also demonstrates that the type structure of Standard ML does not need to be based on dependent types, as suggested by MacQueen [Mac86] and Harper & Mitchell [HM93].

The extension for recursive functors in Tiny-ML$^+$ is based on Russo's work on recursive structures [Rus01]. He uses a backpatching dynamic semantics (similar to that for letrec-expressions in Scheme [KCR98]) to handle computational effects during structure evaluation time. However, his static semantics does not guarantee well-founded recursion, which leads to runtime errors and unnecessary runtime costs. The extension for recursive functors presented here does not suffer from this problem.

Dreyer [Dre04] suggests a solution for ensuring well-founded recursion statically. He also uses a backpatching dynamic semantics, but keeps track of individual recursion variables in the typing derivation. This allows him to ensure well-founded recursion statically even in the presence of separate compilation.

Chapter 3.

Haskell type classes

The type systems of Standard ML [MTHM97] and Haskell 98 [Pey03] are both based on the Hindley/Milner system [Hin69, Mil78, DM82], which supports type inference and parametric polymorphism. However, Haskell extends this type system with *type classes* [Kae88, WB89], a feature not found in ML and the Hindley/Milner system. Type classes are a powerful approach to *ad-hoc polymorphism*. Ad-hoc polymorphism, often called *overloading*, allows the definition of a function to range over several different types; the function then behaves in a different way for each type. A typical example is the equality operator $==$; it is defined on integers, floats, strings, and so on, and compares values of these types in different ways.

This chapter gives an introduction to Haskell type classes (Section 3.1), formalizes the two languages Tiny-HS (Section 3.2) and Tiny-HS$^+$(Section 3.3), and discusses related work (Section 3.4). Tiny-HS is the source language for the translation from ML modules to Haskell type classes, whereas Tiny-HS$^+$ is the target language for the translation in the opposite direction. We discuss in Section 3.3 why we use two different languages.

3.1. Introduction to Haskell type classes

This section introduces Haskell 98 type classes by presenting a series of examples. The examples cover only the features of type classes relevant to this work; there are many tutorials, articles, and books that cover the topic in more detail [WB89, Tho99, HPF00, Pey03] and show advanced applications of type classes [Jon95b, McB02, KLS04].

Haskell permits the introduction of ad-hoc polymorphism by declaring functions as *methods* of a type class. *Instances* of the type class then provide concrete implementations for these methods. Here is how we might define a type class Eq with the equality operator $==$ as a method:

```
class Eq a where
  (==) :: a → a → Bool
```

This definition may be read as "some type a is an instance of the type class Eq if it defines the $==$ operator, which takes two values of type a and returns a boolean value". Now $==$ is available as a top-level operator and can be used for all types

belonging to the type class Eq. For example, we might define a function for testing membership of a list as follows:

```
elem :: Eq a ⇒ a → [a] → Bool
elem x (y : ys) = x == y ∨ elem x ys
elem x [ ]      = False
```

The (optional) type annotation reads "the function elem takes a value of type a, which must be an instance of the type class Eq, and a list of as, and returns a boolean value". The part Eq a of the type signature is called the *context* of the signature. The context lists *constraints* of the form C a for some type class C and some type variable a, which limit the possible types for a to instances of C. Types with such a context are called *qualified types*.

To make a type an instance of a class, we just have to provide a corresponding instance definition. For example, we can make the types Int and Char instances of Eq by the following definitions:

```
instance Eq Int where
  i == j = primIntEq i j
instance Eq Char where
  c == c′ = primCharEq c c′
```

Now we can use the == operator on integers and characters; moreover, we can also use the previously defined function elem on integers and characters:[1]

```
(1 == 42, 'c' == 's', elem 1 [0, 1, 2], elem 'a' "abc")
```

Now imagine that we want to implement equality for lists; that is, we want to make the list type an instance of class Eq. Clearly, we can compare a list for equality only if we can compare the elements of the list for equality. This requirement can be expressed by adding a context to an instance definition; thereby, we constrain the elements of the list to instances of Eq. The following example demonstrates this approach:

```
instance Eq a ⇒ Eq [a] where
  [ ]      == [ ]      = True
  (x : xs) == (y : ys) = x == y ∧ xs == ys
  _        == _        = False
```

The part Eq a to the left of the double arrow ⇒ is called the *instance context*, whereas the part Eq [a] to the right of the double arrow is called the *instance head* (in instance definitions like the one for Eq Int, the context is empty and Eq Int is the instance head). You should read the preceding instance definition as "a list [a], containing elements of type a, is an instance of the type class Eq only if a is an instance of Eq". We make use of a being an instance of Eq in the second equation of the implementation of ==; the left-hand side of the conjunction uses == on x and y, which both have type a. Note that we can also use == on the right-hand side

[1] The string "abc" is an abbreviation for the list of characters ['a', 'b', 'c'] in Haskell.

of the conjunction to compare xs and ys, even though these variables have type [a], the very type we are currently defining an Eq instance for.

Haskell also supports the extension of existing classes with new methods. This is possible by defining a new class as a *subclass* of an existing class. Imagine we want to overload the addition operator +, and all types supporting addition should also support equality comparison. We can implement these requirements by writing the following type class:

```
class Eq a ⇒ Num a where
  (+) :: a → a → a
```

Num is now a subclass of Eq, and Eq is a *superclass* of Num. A subclass supports all methods of its superclasses in addition to the methods defined in the subclass. Similar to instance definitions, the part to the left of the ⇒ is called the *class context*, whereas the part to the right is called the *class head*. We can only define Num instances for types that also belong to the Eq class. For example, we can write

```
instance Num Int where
  i + j = primIntAdd i j
```

but we could not make some type an instance of Num without making it an instance of Eq as well.

3.2. Tiny-HS

Tiny-HS is a simple formalization of the Haskell 98 [Pey03] type class system, supporting all important features except constructor classes [Jon95a], class methods with constraints, and default definitions for methods. These restrictions are necessary because Tiny-HS is the source language in the translation to ML modules (see Section 5.4 for details). The syntax and typing judgments for Tiny-HS are defined in Section 3.2.1 and Section 3.2.2, respectively. The material presented here is based on Jones' Overloaded ML [Jon94] and on Faxén's static semantics for Haskell [Fax02]. I especially recommend reading (parts of) Jones' dissertation if you are not familiar with type systems for qualified types.

3.2.1. Syntax

The syntax of Tiny-HS is shown in Figure 3.1. We assume that the identifier sets ClassId, TyconId, VarId, MethodId, and TypVar are countably infinite. We use different identifier sets for methods and ordinary variables because the translation to ML modules requires this distinction. Every type constructors $T \in \mathsf{TyconId}$ is equipped with a *kind* $\kappa \in \mathbb{N}$, denoting the arity of T. We write T^κ for a type constructor T of kind κ. Like for Tiny-ML, this simple definition of kinds is sufficient because higher-order types are not supported by Tiny-HS. However, unlike Standard ML, Haskell 98 supports higher-order types.

Figure 3.1. Syntax of Tiny-HS

Identifiers

$\mathsf{C} \in \mathsf{ClassId}$	class identifiers
$\mathsf{T} \in \mathsf{TyconId} = \{\rightarrow, \mathsf{Int}, \dots\}$	type constructor identifiers
$\mathsf{z} \in \mathsf{VarId}$	term variables
$\mathsf{m} \in \mathsf{MethodId}$	method variables
$\mathsf{a}, \mathsf{b} \in \mathsf{TypVar}$	type variables

Types

$\mathsf{Typ} \ni \tau ::= \mathsf{a}$	type variable
$\mid \mathsf{T}^{\kappa}\ \overline{\tau}^{\kappa}$	type constructor application
$\mathsf{QTyp} \ni \rho ::= \overline{\pi} \Rightarrow \tau$	qualified type
$\mathsf{TypSc} \ni \sigma ::= \forall \mathsf{A}.\rho$	type scheme
$\mathsf{A}, \mathsf{B} \in \mathrm{Fin}(\mathsf{TypVar})$	set of type variables

Constraints

$\mathsf{Constr} \ni \pi ::= \mathsf{C}\ \tau$	class constraint
$\mathsf{ConstrSc} \ni \theta ::= \forall \mathsf{A}.\overline{\pi} \Rightarrow \pi$	constraint scheme

Expressions

$\mathsf{Exp} \ni \mathsf{w} ::= \mathsf{z}$	term variable
$\mid \mathsf{m}$	method variable
$\mid \lambda \mathsf{z}.\mathsf{w}$	λ-abstraction
$\mid \mathsf{w}_1\ \mathsf{w}_2$	application
$\mid$ **let** $\mathsf{z} = \mathsf{w}_1$ **in** w_2	let-binding

Instances, classes, and programs

$\mathsf{Inst} \ni \mathsf{inst} ::=$ **instance** $\forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i \in [r]} \Rightarrow \mathsf{C}\ (\mathsf{T}^{\kappa}\ \overline{\tau}^{\kappa})$ **where** $\overline{\mathsf{mval}}$	instance definition
$\mathsf{Mval} \ni \mathsf{mval} ::= \mathsf{m} = \mathsf{w}$	method implementation
$\mathsf{Cls} \ni \mathsf{cls} ::=$ **class** $\forall \{\mathsf{a}\}.\overline{\mathsf{C}_i\ \mathsf{a}}^{i \in [r]} \Rightarrow \mathsf{C}\ \mathsf{a}$ **where** $\overline{\mathsf{msig}}$	class definition
$\mathsf{Msig} \ni \mathsf{msig} ::= \mathsf{m} :: \forall \mathsf{A}.\tau$	method signature
$\mathsf{Pgm} \ni \mathsf{pgm} ::= \overline{\mathsf{cls}}\ \overline{\mathsf{inst}}\ \mathsf{w}$	program

The syntax of types, constraints, expressions, instances, classes, and programs is fairly standard. We often write ρ for a type scheme $\sigma = \forall\emptyset.\rho$ and τ for a qualified type $\rho = \tau$. In code examples, we often omit universal quantifiers of classes, instances, and method signatures completely to enhance readability of the code. Two properties of the syntax are worth mentioning: Types $\tau \in \mathsf{Typ}$ only support saturated type constructor applications to prevent the construction of higher-order types, and method signatures $\mathsf{msig} \in \mathsf{Msig}$ do not contain a constraint part $\overline{\pi} \Rightarrow$ because the translation to ML modules cannot handle such constraints. Note that programs consist not only of class and instance definitions but also have a "main" expression w.

We need some definitions for reasoning about various syntactic constructs. We let O range over all syntactic constructs. First, we introduce the notion of free variables and substitutions.

Definition 3.1 (Free variables). $\mathrm{FV}^{\mathsf{a}}(\mathsf{O}) \subseteq \mathsf{TypVar}$ denotes the set of type variables free in some syntactic construct O. Similarly, $\mathrm{FV}^{\mathsf{z}}(\mathsf{O}) \subseteq \mathsf{VarId}$ denotes the set of term variables free in O. The notion of free is defined as usual.

Definition 3.2 (Substitutions). A substitution ψ from type variables to types is an element of $\mathsf{TypVar} \xrightarrow{\text{fin}} \mathsf{Typ}$. We often write $[\overline{\tau_i/\mathsf{a}_i}^{i\in[n]}]$ for a substitution $\psi = \{\mathsf{a}_i \mapsto \tau_i \mid i \in [n]\}$. Substitution application is defined in the usual, capture-avoiding way.

We need a condition that rules out ill-formed instance definitions like **instance** $\forall\{\mathsf{a}\}.\mathsf{Eq\ a} \Rightarrow \mathsf{Eq\ Int} \ldots$, where a type variable occurs in the instance context but is not mentioned in the instance head.

Definition 3.3 (Well-formed constraint schemes). A constraint scheme $\theta = \forall\mathsf{A}.\ \overline{\pi} \Rightarrow \pi'$ is said to be well-formed, if $\mathrm{FV}^{\mathsf{a}}(\overline{\pi}) \subseteq \mathrm{FV}^{\mathsf{a}}(\pi') = \mathsf{A}$.

As in Haskell 98, we need to disallow *ambiguous type schemes* to guarantee that the translation from type classes to ML modules is well-defined. Suppose we define the following type classes to convert values to strings and back:

```
class Show a where
  show :: a → String
class Read a where
  read :: String → a
```

What is then the type of the expression $\lambda\mathsf{z}.\ \mathsf{show}\ (\mathsf{read\ z})$? It is $\forall\{\mathsf{a}\}.\ (\mathsf{Read\ a}, \mathsf{Show\ a}) \Rightarrow \mathsf{String} \to \mathsf{String}$ with no way to tell which type we should pick for a. The problem is that a is mentioned only in the context of the type scheme. The following definition flags such type schemes as ambiguous.

Definition 3.4 (Unambiguous type schemes in Tiny-HS). A type scheme $\sigma = \forall\mathsf{A}.\overline{\pi} \Rightarrow \tau$ is considered unambiguous, if $\mathrm{FV}^{\mathsf{a}}(\overline{\pi}) \cap \mathsf{A} \subseteq \mathrm{FV}^{\mathsf{a}}(\tau)$. Otherwise, σ is called ambiguous.

3.2.2. Typing judgments

This section discusses the typing judgments for Tiny-HS. However, some preparatory definitions are necessary before we can discuss the judgments.

Preparations

We start by defining the environments used in the typing judgments.

Definition 3.5 (Environments for Tiny-HS). A variable environment $\hat{\Gamma}$ maps term and method variables to type schemes. A constraint environment $\hat{\Theta}$ consists of three components: $\hat{\Theta}^s$ records constraint schemes resulting from subclass definitions, $\hat{\Theta}^i$ is populated by constraint schemes originating from instance definitions, and $\hat{\Theta}^l$ is a set of local constraints added during a typing derivation. More formally, the environments are defined as follows:

$$\hat{\Gamma} \quad \in \quad \mathsf{VarId} \cup \mathsf{MethodId} \xrightarrow{\text{fin}} \mathsf{TypSc} \qquad \text{Variable environment}$$

$$\hat{\Theta} \quad := \quad \begin{pmatrix} \hat{\Theta}^s \in \mathrm{Fin}(\mathsf{ConstrSc}), \\ \hat{\Theta}^i \in \mathrm{Fin}(\mathsf{ConstrSc}), \\ \hat{\Theta}^l \in \mathrm{Fin}(\mathsf{Constr}) \end{pmatrix} \qquad \text{Constraint environment}$$

You may have noticed that the different components of $\hat{\Theta}$ are very similar. In fact, we could merge them into a single component. However, the translation from type classes to ML modules requires a distinction between the different components, so we introduce this distinction.

It is sometimes very convenient to treat $\hat{\Theta}$ as a set and not as a triple of sets. Therefore, we define the following notational convention:

Definition 3.6 (Set operations for constraint environments). The set operations $\subseteq$, $=$, $\cup$, $\cap$, and $\setminus$ are defined component-wise for a constraint environment $\hat{\Theta}$. Similarly, $\theta \in \hat{\Theta}$ is an abbreviation for "$\theta \in \hat{\Theta}^s$ or $\theta \in \hat{\Theta}^i$ or $\theta \in \hat{\Theta}^l$".

We need a way to determine the set of immediate superclasses of a class. The following definition specifies an operation that does exactly this. We shall see that the definition is reasonable once we discuss the typing judgment for class definitions.

Definition 3.7 (Superclasses for Tiny-HS). The set of immediate superclasses of C in $\hat{\Theta}$ is defined as $\mathrm{Sup}(\hat{\Theta}, \mathsf{C}) := \{\mathsf{C}' \mid \forall\{\mathsf{a}\}.\mathsf{C}\ \mathsf{a} \Rightarrow \mathsf{C}'\ \mathsf{a} \in \hat{\Theta}^s\}$.

Discussion of the typing judgments

The typing judgments for Tiny-HS are shown in Figures 3.2 and 3.3. The entailment judgment of Figure 3.2 is standard [Fax02, Figure 28]. Intuitively, a constraint environment $\hat{\Theta}$ entails some constraint π, written $\hat{\Theta} \Vdash \pi$, if π is contained in the

Figure 3.2. Judgments for entailment and expression typing in Tiny-HS

Entailment $\boxed{\hat{\Theta} \Vdash \pi}$

$$\frac{\pi \in \hat{\Theta}^l}{\hat{\Theta} \Vdash \pi}\ (elem_{entail}) \qquad \frac{\begin{array}{c}(\forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[n]} \Rightarrow \mathsf{C}\ \tau') \in \hat{\Theta}^i \\ \tau = \psi(\tau') \qquad \overline{\hat{\Theta} \Vdash \mathsf{C}_i\ \psi(\mathsf{a}_i)}^{i\in[n]} \qquad \mathrm{Dom}(\psi) = \mathsf{A}\end{array}}{\hat{\Theta} \Vdash \mathsf{C}\ \tau}\ (inst_{entail})$$

$$\frac{(\forall \mathsf{a}.\mathsf{C}^{\mathsf{sub}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathsf{sup}}\ \mathsf{a}) \in \hat{\Theta}^s \qquad \hat{\Theta} \Vdash \mathsf{C}^{\mathsf{sub}}\ \tau}{\hat{\Theta} \Vdash \mathsf{C}^{\mathsf{sup}}\ \tau}\ (super_{entail})$$

Expression typing $\boxed{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau}$

$$\frac{\hat{\Gamma}(\mathsf{z}) = \forall \mathsf{A}.\overline{\pi}^n \Rightarrow \tau' \qquad \psi = [\overline{\tau_{\mathsf{a}}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}}] \qquad \psi(\tau') = \tau \qquad \overline{\hat{\Theta} \Vdash \psi(\pi_i)}^{i\in[n]}}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{z} : \tau}\ (var)$$

$$\frac{\hat{\Gamma}(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau' \qquad \psi = [\overline{\tau_{\mathsf{a}}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}}] \qquad \psi(\tau') = \tau \qquad \hat{\Theta} \Vdash \mathsf{C}\ \psi(\mathsf{b})}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{m} : \tau}\ (method)$$

$$\frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_1 : \tau' \to \tau \qquad \hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_2 : \tau'}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_1\ \mathsf{w}_2 : \tau}\ (\to E) \qquad \frac{\hat{\Theta}; \hat{\Gamma}, \mathsf{z} \mapsto \tau' \vdash \mathsf{w} : \tau}{\hat{\Theta}; \hat{\Gamma} \vdash \lambda \mathsf{z}.\mathsf{w} : \tau' \to \tau}\ (\to I)$$

$$\frac{\begin{array}{c}\hat{\Theta}' := (\hat{\Theta}^s, \hat{\Theta}^i, \{\overline{\pi}\}) \\ \hat{\Theta}'; \hat{\Gamma} \vdash \mathsf{w}_1 : \tau' \qquad \sigma = \widehat{\mathrm{Gen}}(\hat{\Theta}', \hat{\Gamma}, \tau')\ \text{unambiguous} \qquad \hat{\Theta}; \hat{\Gamma}, \mathsf{z} \mapsto \sigma \vdash \mathsf{w}_2 : \tau\end{array}}{\hat{\Theta}; \hat{\Gamma} \vdash \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2 : \tau}\ (let)$$

Generalization

$$\widehat{\mathrm{Gen}}((\hat{\Theta}^s, \hat{\Theta}^i, \{\overline{\pi}\}), \hat{\Gamma}, \tau) := \forall((\mathrm{FV}^{\mathsf{a}}(\overline{\pi}) \cup \mathrm{FV}^{\mathsf{a}}(\tau)) \setminus (\mathrm{FV}^{\mathsf{a}}(\hat{\Gamma}) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Theta}^s) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Theta}^i))).\overline{\pi} \Rightarrow \tau$$

local part of $\hat{\Theta}$, or there is some instance definition whose head matches π and whose context is entailed by $\hat{\Theta}$, or π holds because of a subclass relation.

The typing judgment $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau$ (also show in Figure 3.2) assigns a type τ to an expression w under the environments $\hat{\Theta}$ and $\hat{\Gamma}$. It corresponds to the syntax-directed system presented by Jones [Jon94, Figure 3.2]; we do not use his declarative system [Jon94, Figure 3.1] because the type-directed translation from Haskell type classes to ML modules cannot be formulated in terms of this system (see Section 5.6 for a detailed discussion). The only additions to Jones' system are the rule (*method*) for method variables (necessary because the syntax of Tiny-HS differs between method and term variables), and a new premise for the rule (*let*), which ensures that σ is unambiguous. Note that the definition of generalization in Figure 3.2 is slightly different from Jones' definition: $\widehat{\mathrm{Gen}}(\hat{\Theta}, \hat{\Gamma}, \tau)$ not only generalizes over the type variables free in τ and not free in the environments, but also moves $\hat{\Theta}$'s local constraints into the context of the resulting type scheme.

The rules for instances, classes, and programs in Figure 3.3 are influenced by Faxén's static semantics for Haskell [Fax02]. One significant difference between Faxén's rules and the ones for Tiny-HS is that the rule (*prog*) does not define the constraint and variable environments recursively. Instead, it first collects the environments and then uses them to type check the instance definitions and the main expression.

The judgment $\vdash \mathsf{cls} : \hat{\Theta}; \hat{\Gamma}$ handles class definitions by collecting method signatures and information about superclasses in the variable and constraint environment, respectively. We now see why Definition 3.7 on page 32 works as expected: For every superclass $\mathsf{C}^{\mathsf{sup}}$ of a given class $\mathsf{C}^{\mathsf{sub}}$, an implication of the form $\forall\{\mathsf{a}\}.\ \mathsf{C}^{\mathsf{sub}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathsf{sup}}\ \mathsf{a}$ is added to the constraint environment.

Instance definitions are handled by two different judgments: The judgment $\vdash \mathsf{inst} : \hat{\Theta}$ collects constraint schemes introduced by instance definitions, and judgment $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{inst}$ ensures that all method implementations are type correct and that instances for all immediate superclasses are derivable.

This entailment check for superclasses in rule ($inst_{check}$) is performed differently than in Faxén's system because we remove the constraint scheme $\forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[r]} \Rightarrow \mathsf{C}\ \tau$ that was introduced by the very instance definition inst from $\hat{\Theta}^i$ before deriving instances for the immediate superclasses, whereas Faxén uses the whole $\hat{\Theta}^i$. We need to remove the constraint scheme to keep rule ($inst_{check}$) sound; otherwise, the entailment check is trivial, even for programs that miss some superclass definitions.[2] Let us look at the following incorrect program:

class $\forall\{\mathsf{a}\}$. $\mathsf{C}^{\mathsf{sup}}$ a **where**
class $\forall\{\mathsf{a}\}$. $\mathsf{C}^{\mathsf{sup}}$ a $\Rightarrow$ $\mathsf{C}^{\mathsf{sub}}$ a **where**
instance $\forall\emptyset$. $\mathsf{C}^{\mathsf{sub}}$ Int **where**
42

[2] I believe that the relevant rule in Faxén's system [Fax02, Figure 26] is incorrect. Interestingly, the corresponding rule in another paper [HHPW96, Figure 10] contains the same sort of mistake.

Figure 3.3. Typing judgments for Tiny-HS instances, classes, and programs

Instance definitions $\boxed{\vdash \mathsf{inst} : \hat{\Theta}}$

$$\frac{\theta \text{ well-formed}}{\vdash \textbf{instance } \theta \textbf{ where } \overline{\mathsf{mval}}^{n} : (\emptyset, \{\theta\}, \emptyset)} \; (\mathit{inst}_{\mathit{collect}})$$

$\boxed{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{inst}}$

$$\frac{\begin{array}{c}(\hat{\Theta}^s, \hat{\Theta}^i, \{\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[r]}\}); \hat{\Gamma}; \tau \overset{\text{method}}{\vdash} \mathsf{m}_i = \mathsf{w}_i \;\; (i \in [n]) \\ (\hat{\Theta}^s, \hat{\Theta}^i \setminus \{\forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[r]} \Rightarrow \mathsf{C}\ \tau\}, \{\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[r]}\}) \Vdash \mathsf{C}^{\mathsf{sup}}\ \tau \;\; (\mathsf{C}^{\mathsf{sup}} \in \mathrm{Sup}(\hat{\Theta}, \mathsf{C}))\end{array}}{\hat{\Theta}; \hat{\Gamma} \vdash \textbf{instance } \forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[r]} \Rightarrow \mathsf{C}\ \tau \textbf{ where } \overline{\mathsf{m}_i = \mathsf{w}_i}^{i\in[n]}} \; (\mathit{inst}_{\mathit{check}})$$

$\boxed{\hat{\Theta}; \hat{\Gamma}; \tau \overset{\text{method}}{\vdash} \mathsf{m} = \mathsf{w}}$

$$\frac{\begin{array}{c}\hat{\Gamma}(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau' \\ \mathsf{A} \cap (\mathrm{FV}^{\mathsf{a}}(\tau) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Theta}) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Gamma})) = \emptyset \qquad \hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : [\tau/\mathsf{b}]\tau'\end{array}}{\hat{\Theta}; \hat{\Gamma}; \tau \overset{\text{method}}{\vdash} \mathsf{m} = \mathsf{w}} \; (\mathit{inst}_{\mathit{check\text{-}method}})$$

Class definitions $\boxed{\vdash \mathsf{cls} : \hat{\Theta}; \hat{\Gamma}}$

$$\frac{\begin{array}{c}\mathsf{a} \notin \mathsf{A}_i \qquad \sigma_i = (\forall(\mathsf{A}_i \cup \{\mathsf{a}\}).\mathsf{C}\ \mathsf{a} \Rightarrow \tau_i) \text{ unambiguous} \\ \mathrm{FV}^{\mathsf{a}}(\sigma_i) = \emptyset \qquad (\text{for all } i \in [n])\end{array}}{\begin{array}{l}\vdash \textbf{class } \forall\{\mathsf{a}\}.\overline{\mathsf{C}_i\ \mathsf{a}}^{i\in[r]} \Rightarrow \mathsf{C}\ \mathsf{a} \textbf{ where } \overline{\mathsf{m}_i :: \forall \mathsf{A}_i.\tau_i}^{i\in[n]} \\ \quad : (\{\forall\{\mathsf{a}\}.\mathsf{C}\ \mathsf{a} \Rightarrow \mathsf{C}_i\ \mathsf{a} \mid i \in [r]\}, \emptyset, \emptyset) \\ \quad ; \{\mathsf{m}_i \mapsto \sigma_i \mid i \in [n]\}\end{array}} \; (\mathit{class})$$

Programs $\boxed{\vdash \mathsf{pgm}}$

$$\frac{\begin{array}{c}\overline{\vdash \mathsf{cls}_i : \hat{\Theta}_i; \hat{\Gamma}_i}^{i\in[n]} \qquad \overline{\vdash \mathsf{inst}_i : \hat{\Theta}'_i}^{i\in[m]} \\ \hat{\Theta} = \bigcup_{i\in[n]} \hat{\Theta}_i \cup \bigcup_{i\in[m]} \hat{\Theta}'_i \qquad \hat{\Gamma} = \dot{\bigcup}_{i\in[n]} \hat{\Gamma}_i \\ \overline{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{inst}_i}^{i\in[m]} \qquad \hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \mathsf{Int}\end{array}}{\vdash \overline{\mathsf{cls}}^{n}\ \overline{\mathsf{inst}}^{m}\ \mathsf{w}} \; (\mathit{prog})$$

We obtain the following constraint environment after collecting all constraint schemes: $\hat{\Theta} = (\{\forall\{\mathsf{a}\}.\mathsf{C}^{\mathsf{sub}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathsf{sup}}\ \mathsf{a}\}, \{\forall\emptyset.\mathsf{C}^{\mathsf{sub}}\ \mathsf{Int}\}, \emptyset)$. It should not be possible to get a derivation for $\hat{\Theta}; \hat{\Gamma} \vdash$ **instance** $\forall\emptyset.\mathsf{C}^{\mathsf{sub}}\ \mathsf{Int}$ **where** because $\mathsf{C}^{\mathsf{sup}}$ is a superclass of $\mathsf{C}^{\mathsf{sub}}$ but there is no instance definition for $\mathsf{C}^{\mathsf{sup}}\ \mathsf{Int}$. However, if we used the full constraint environment $\hat{\Theta}$ for the superclass entailment check, $\hat{\Theta} \Vdash \mathsf{C}^{\mathsf{sup}}\ \mathsf{Int}$ would be derivable by rules $(super_{entail})$ and $(inst_{entail})$, so $\hat{\Theta}; \hat{\Gamma} \vdash$ **instance** $\forall\emptyset.\mathsf{C}^{\mathsf{sub}}\ \mathsf{Int}$ **where** would be derivable. Hence, the incorrect program would type check! However, $(\hat{\Theta}^s, \hat{\Theta}^i \setminus \{\forall\emptyset.\mathsf{C}^{\mathsf{sub}}\ \mathsf{Int}\}, \emptyset) \Vdash \mathsf{C}^{\mathsf{sup}}\ \mathsf{Int}$ does not hold, so Tiny-HS' type system correctly rejects the offending program.

We now argue informally that our modification of Faxén's system does not reject valid programs. Suppose that $\mathsf{C}^{\mathsf{sub}}$ is a subclass of $\mathsf{C}^{\mathsf{sup}}$ and that we need to derive $\mathsf{C}^{\mathsf{sup}}\ \tau$ while checking an instance definition for $\mathsf{C}^{\mathsf{sub}}\ \tau$. In a type system that uses the whole constraint environment for checking superclasses (like Faxén's), we can derive $\mathsf{C}^{\mathsf{sub}}\ \psi(\tau)$ by rule $(inst_{entail})$ for any substitution ψ within a derivation of $\mathsf{C}^{\mathsf{sup}}\ \tau$. This is not possible in our system. However, we can never come into a situation where we would need $\mathsf{C}^{\mathsf{sub}}\ \psi(\tau)$ to entail the context of an instance definition for $\mathsf{C}^{\mathsf{sup}}\ \tau$ because $\mathsf{C}^{\mathsf{sub}}\ \psi(\tau)$ contains at least as many type constructors as $\mathsf{C}^{\mathsf{sup}}\ \tau$, but Tiny-HS and Haskell 98 ensure that a constraint in the instance context always contains fewer type constructors than the constraint in the instance head.

Another major difference between Faxén's system and the one presented here is that Faxén's rules have a lot more premises because his system is intended to be a complete language specification. However, these additional premises add only little to the discussion at hand and would make the rules presented here unnecessarily complicated. Nevertheless, we sometimes need programs to have properties not enforced by the typing rules. Therefore, we capture such properties in the following definition.

Definition 3.8 (Well-formed Tiny-HS programs). A Tiny-HS program is said to be well-formed if it meets the following requirements:

- Every class is defined at most once.
- Classes are used only after their definition.
- Instance heads do not overlap.

3.3. Tiny-HS$^+$

Tiny-HS cannot be used as the target language for the translation from ML modules to Haskell type classes; for example, one important feature missing is the counterpart for type components in ML structures. Therefore, we now introduce the language Tiny-HS$^+$, which extends Tiny-HS with Haskell 98 features, with well-known additions to Haskell 98, and with one feature developed as part of this work. The features are not included in Tiny-HS because they are either not

relevant to the translation from type classes to ML modules or they are not part of Haskell 98. The following features are new in Tiny-HS$^+$:

- User-defined data types (part of Haskell 98)
- Multi-parameter type classes (addition to Haskell 98, see [PJM97])
- Associated type synonyms (addition to Haskell 98, see [CKP05])
- Abstract associated type synonyms (extension to associated type synonyms, developed as part of this work)

We first discuss the motivation behind the extensions for multi-parameter type classes and (abstract) associated type synonyms in Sections 3.3.1 through 3.3.3. For reasons of space, we do not go much into detail; the interested reader is referred to the relevant literature [PJM97, CKP05]. Then we define the syntax of Tiny-HS$^+$ in Section 3.3.4 and discuss the typing judgments in Section 3.3.5.

3.3.1. Multi-parameter type classes

The type class definitions in the examples of Section 3.1 all have only a single type variable in the class head. It is relatively straightforward to allow type classes with multiple type variables in the class head. Consider the following example:

```
class Add a b c where
  add :: a → b → c
```

The class Add has three parameters, which are used to specify the operand types and the result type of the add operation. We can define instances for such multi-parameter type classes in the same way as we do for single-parameter type classes:

```
instance Add Int Float Float where
  add i f = primFloatAdd (intAsFloat i) f
```

The preceding instance definition specifies that we can add an integer and a float to get another float as result. We can also add an integer and a float and get an integer as result:

```
instance Add Int Float Int where
  add i f = primIntAdd i (floatAsInt f)
```

But maybe we do not want integers and floats being added in two different ways. Instead, it would be desirable if the types of the operands uniquely determined the result type. Multi-parameter type classes alone cannot solve this problem; one approach to specifying such dependencies are *associated type synonyms*, which are introduced in the following section.

3.3.2. Associated type synonyms

With associated type synonyms, we can define a new type class Add′ in such a way that the result type of the addition is uniquely determined by the types of the two operands:

```
class Add′ a b where
  type Result a b
  add′ :: a → b → Result a b
```

The result type is no longer part of the class head; instead, it is specified as a type synonym associated with the class. We define instances of such a class in much the same way as before, we only need to define the associated type synonym of the class in addition to its method.

```
instance Add′ Int Float where
  type Result Int Float = Float
  add′ i f = primFloatAdd (intAsFloat i) f
```

Associated type synonyms introduce an equality relation on types that is not based purely on syntactic equality. Consider the types Result Int Float and Float in the preceding example. Clearly, the two types are syntactically different. Nevertheless, they can be used interchangeably because the associated type synonym definition in the instance for Add′ *defines them as equal*.

3.3.3. Abstract associated type synonyms

None of the features of Haskell type classes seen so far provides a way to make certain types abstract. But a translation of ML's module system to Haskell type classes definitely needs to be able to handle abstract types! After all, abstract types are one of the key features of a module system.

One possible solution to this problem would be to use Haskell's module system [DJH02]. We have not introduced Haskell's module system, but all you need to know in order to understand this solution is that Haskell lets you hide the constructors of algebraic data types in the export list of a module. Hence, a type can be made abstract by wrapping it in an algebraic data type[3] in a separate module that does not export the data constructors of the data type. This solution is unsatisfactory for two reasons. First of all, explicit conversion code is necessary to turn a value of the concrete type into a value of the abstract type and vice versa. Such a conversion is not necessary when abstract types are implemented using the ML module system as demonstrated in Section 2.1. The second reason for not using Haskell's module system is that I want to compare ML modules with Haskell type classes and not ML modules with "Haskell type classes plus the Haskell module system".

[3] In Haskell, you would probably use a `newtype`.

Therefore, I propose *abstract associated type synonyms* as an extension to associated type synonyms, and use them to implement abstract types in Haskell. Interestingly, the idea behind abstract associated type synonyms goes back to ML's `abstype` feature, which is nowadays essentially deprecated; a similar feature is also implemented in the Haskell interpreter Hugs [JP99, Section 7.3.5]. The idea is the following: To make a type synonym abstract, we only have to limit the scope of the right-hand side of its definition. In ML and Hugs, this is done by explicitly stating the constructs that are allowed to access the concrete definition. With abstract associated type synonyms, the scope is determined implicitly by the instance defining the synonym: Inside the instance, the right–hand side is visible, but outside it is hidden; that is, the associated type synonym is equated with some fresh type constructor.

Let us illustrate abstract associated type synonyms by implementing sets of integers with type classes, where the concrete type used to implement sets is hidden from the clients of the implementation.[4] We first define a type class SET that defines the set interface independent from the concrete element type:

```
class SET a where
  type Elem a
  type Set a
  empty  :: a → Set a
  member :: a → Elem a → Set a → Bool
  insert :: a → Elem a → Set a → Set a
```

The type variable a in the class definition is only used to index the associated type synonyms Elem and Set, and the methods of the class. We cannot do without a in the method signatures; otherwise, the signatures would be ambiguous (ambiguity in the presence of associated type synonyms will be formally defined in Definition 3.9 on page 42). Note that the type variable a is not the type of the set implementation; instead, the set implementation is represented by the associated type synonym Set. This is important for being able to use an abstract associated type synonym to keep the concrete implementation type hidden.

We need to define a data type that plays the role of a in the instance definition for SET. We leave the right-hand side of the following data type definition empty to emphasize that we never examine values of this type.

```
data IntSet
```

Now we can make IntSet an instance of SET. The keyword **abstype** is used to introduce an abstract associated type synonym.[5]

[4] You may notice that the example is more or less a translation of the IntSet_2 example from page 6. However, a comparison between the ML and Haskell code would hamper the discussion at hand. The translation from ML to Haskell is extensively covered in Chapter 4.

[5] In a real implementation, the syntax **type** Set IntSet **hiding** [Int] would be desirable because it avoids the new keyword **abstype**. Thanks to Donald Steward for pointing this out.

```
instance SET IntSet where
  type Elem IntSet    = Int
  abstype Set IntSet = [Int]
  empty   _           = []
  member  _ x s       = exists (λy . primIntEq x y) s
  insert  _ x s       = if member (⊥ :: IntSet) x s then s else (x : s)
```

Note that we never examine a value of type IntSet; hence, it is safe to use the diverging value $\bot$ in the definition of insert. In fact, the IntSet type only directs the type checker in selecting the right instance. The effect of using an abstract associated type synonym for Set is that the type equality Set IntSet = [Int] is not visible outside the instance definition, so that clients cannot treat sets like lists. For example, we can write insert ($\bot$ :: IntSet) 1 (empty ($\bot$ :: IntSet)), but insert ($\bot$ :: IntSet) 1 [] does not type check. However, the type equality is visible inside the instance definition; this is crucial so as to type check the methods of the instance.

3.3.4. Syntax

The syntax of Tiny-HS$^+$ is shown in Figure 3.4. We use the same symbols for Tiny-HS and Tiny-HS$^+$, sometimes with a different meaning. This does not cause any problems because it is always clear from the context which meaning is relevant: The rest of this chapter and the whole Chapter 4 uses Tiny-HS$^+$ exclusively, whereas Chapter 5 uses only Tiny-HS.

Tiny-HS$^+$'s syntax is very similar to the syntax of Tiny-HS. Method identifiers are no longer needed because we do not differ between method and term variables anymore. We require that the set TyconId contains a type constructor T^κ for every Tiny-ML type constructor T^κ, so that we can translate Tiny-ML types correctly into Tiny-HS$^+$ types. There is a new identifier set ASynId for identifiers of associated type synonyms S. Along the lines of type constructors, associated type synonyms are equipped with a kind $\kappa \in \mathbb{N}$. The kind of an associated type synonym corresponds to the number of parameters of the class declaring the synonym. This is a restriction of the system of Chakravarty et al. [CKP05], where an associated type synonym may have more parameters than the class that declares it. Tiny-HS$^+$ does not need this greater generality.

The syntax of types $\tau \in$ Typ is extended with associated type synonym applications $\eta \in$ ATyp. An associated type synonym application must always be saturated in order to keep type inference decidable.

Constraints $\pi \in$ Constr and constraint schemes $\theta \in$ ConstrSc contain not only class constraints but also equality constraints. Equality constraints are used to model nonsyntactic type equalities. Note that a class constraint C $\overline{\tau}$ can have multiple parameters in Tiny-HS$^+$.

The syntax of expressions w $\in$ Exp is extended with a wildcard λ-abstraction of the form λ_.w. Additionally, expressions can now have type annotations, written (w :: σ).

Figure 3.4. Syntax of Tiny-HS$^+$

Identifiers

$C \in \mathsf{ClassId}$	class identifiers
$T \in \mathsf{TyconId} = \{\rightarrow, \mathsf{Int}, \dots\}$	type constructor identifiers
$S \in \mathsf{ASynId}$	associated type synonym identifiers
$z \in \mathsf{VarId}$	term variables
$a, b \in \mathsf{TypVar}$	type variables

Types

$\mathsf{Typ} \ni \tau$	$::= a \mid T^\kappa\ \overline{\tau}^\kappa$	(as for Tiny-HS)
	$\mid \eta$	associated type synonym
$\mathsf{ATyp} \ni \eta$	$::= S^\kappa\ \overline{\tau}^\kappa$	associated type synonym application
$\mathsf{QTyp} \ni \rho$	$::= \overline{\pi} \Rightarrow \tau$	qualified type
$\mathsf{TypSc} \ni \sigma$	$::= \forall A.\rho$	type scheme
A, B	$\in \mathrm{Fin}(\mathsf{TypVar})$	set of type variables

Constraints

$\mathsf{Constr} \ni \pi$	$::= C\ \overline{\tau}$	class constraint
	$\mid \eta = \tau$	equality constraint
$\mathsf{ConstrSc} \ni \theta$	$::= \forall A.\overline{\pi} \Rightarrow C\ \overline{\tau}$	constraint scheme
	$\mid \forall A.\eta = \tau$	equality constraint scheme

Expressions

$\mathsf{Exp} \ni w$	$::= z \mid \lambda z.w \mid w_1\ w_2 \mid \mathbf{let}\ z = w_1\ \mathbf{in}\ w_2$	(as for Tiny-HS)
	$\mid \lambda\_.w$	wildcard λ-abstraction
	$\mid (w :: \sigma)$	type annotation

Instances, classes, data types, and programs

$\mathsf{Inst} \ni \mathsf{inst}$	$::= \mathbf{instance}\ \forall A.\overline{\overline{C_i}\ \overline{a_i}}^{i\in[r]} \Rightarrow C\ \overline{\tau}\ \mathbf{where}$ $\overline{\mathsf{tdef}}\ \overline{\mathsf{mval}}$	instance definition
$\mathsf{Tdef} \ni \mathsf{tdef}$	$::= \mathbf{type}\ S^\kappa\ \overline{\tau}^\kappa = \tau'$	associated type synonym definition
	$\mid \mathbf{abstype}\ S^\kappa\ \overline{\tau}^\kappa = \tau'$	abstract associated type synonym definition
$\mathsf{Mval} \ni \mathsf{mval}$	$::= m = w$	method implementation
$\mathsf{Cls} \ni \mathsf{cls}$	$::= \mathbf{class}\ \forall A.\overline{\overline{C_i}\ \overline{a_i}}^{i\in[r]} \Rightarrow C\ \overline{a}\ \mathbf{where}$ $\overline{\mathsf{tdec}}\ \overline{\mathsf{msig}}$	class definition
$\mathsf{Tdec} \ni \mathsf{tdec}$	$::= \mathbf{type}\ S^\kappa\ \overline{a}^\kappa$	associated type synonym declaration
$\mathsf{Msig} \ni \mathsf{msig}$	$::= m :: \forall A.\tau$	method signature
$\mathsf{Ddec} \ni \mathsf{ddec}$	$::= \mathbf{data}\ T$	data type definition
$\mathsf{Pgm} \ni \mathsf{pgm}$	$::= \overline{\mathsf{ddec}}\ \overline{\mathsf{cls}}\ \overline{\mathsf{inst}}$	program

Instance definitions $\mathsf{inst} \in \mathsf{Inst}$ in Tiny-HS$^+$ can have multiple parameters in the instance context and the instance head, and they have to provide definitions for the associated type synonyms of the class. Definitions of associated type synonyms come in two flavors: they either define a (non–abstract) associated type synonym using the keyword **type**, or they define an abstract associated type synonym using the keyword **abstype**. Perhaps surprisingly, equality constraints in instance contexts are not needed for the translation from modules to type classes, so they have been omitted from Tiny-HS$^+$. Note that it is not possible to ensure syntactically that instance heads contain at least one type constructor. However, without this condition, type checking might be undecidable for Tiny-HS$^+$. Therefore, we assume implicitly that all instance heads of a Tiny-HS$^+$ program contain at least one type constructor.

Along the lines of instances, class definitions $\mathsf{cls} \in \mathsf{Cls}$ can have multiple parameters in the class context and head. They may also contain associated type synonym declarations $\mathsf{tdec} \in \mathsf{Tdec}$.

The new top-level definition **data** T defines a new type constructor T of kind 0. Strictly speaking, such data type constructor definitions are not valid Haskell 98 because they do not define any data constructors. However, we do not need data constructors in the translation from ML modules to Tiny-HS$^+$, so we omitted them for clarity.

We need to refine the definition of ambiguity in the presence of associated type synonyms. For example, the type scheme $\forall\{\mathsf{a}\}.\mathsf{C}\ \mathsf{a} \Rightarrow \mathsf{S}\ \mathsf{a}$ is ambiguous because knowing that $\mathsf{S}\ \mathsf{a} = \tau$ for some τ does not tell us to which instance the definition of S belongs. However, according to the old definition of ambiguity on page 31, the type scheme would be unambiguous.

Definition 3.9 (Unambiguous type schemes in Tiny-HS$^+$). In the presence of associated type synonyms, we call a type scheme $\sigma = \forall \mathsf{A}.\rho$ unambiguous if $\mathrm{FV}^{\mathsf{a}}(\rho) \cap \mathsf{A} \subseteq \mathrm{Fixv}(\rho)$. Here, the set of fixed type variables $\mathrm{Fixv}(\rho)$ of a qualified type ρ is defined as follows:

$$\begin{aligned}
\mathrm{Fixv}(\mathsf{a}) &= \{\mathsf{a}\} \\
\mathrm{Fixv}(\mathsf{T}^\kappa\ \overline{\tau}^\kappa) &= \cup_{i\in[\kappa]}\, \mathrm{Fixv}(\tau_i) \\
\mathrm{Fixv}(\mathsf{S}^\kappa\ \overline{\tau}^\kappa) &= \emptyset \\
\mathrm{Fixv}(\mathsf{C}\ \overline{\tau} \Rightarrow \rho) &= \mathrm{Fixv}(\rho) \\
\mathrm{Fixv}((\eta = \tau) \Rightarrow \rho) &= \mathrm{Fixv}(\tau) \cup \mathrm{Fixv}(\rho)
\end{aligned}$$

3.3.5. Typing judgments

Before discussing the typing judgments for Tiny-HS$^+$, we need to adopt the definitions for environments and superclasses from Section 3.2.2. The following definition of environments for Tiny-HS$^+$ drops the somewhat artificial distinction between the three components of a constraint environment.

Figure 3.5. Judgment for well-formedness of types in Tiny-HS$^+$

$\boxed{\hat{\Theta} \vdash \tau}$

$$\frac{\overline{\hat{\Theta} \vdash \tau_i}^{\,i \in [\kappa]} \qquad \hat{\Theta} \Vdash \mathsf{C}\ \overline{\tau}^{\kappa}\ (\mathsf{S} \text{ is an associated type of } \mathsf{C})}{\hat{\Theta} \vdash \mathsf{S}^{\kappa}\ \overline{\tau}^{\kappa}}\ (wf_{syn})^+$$

$$\frac{\mathsf{T} \text{ is a builtin or user-defined type constructor of kind } \kappa \qquad \overline{\hat{\Theta} \vdash \tau_i}^{\,i \in [\kappa]}}{\hat{\Theta} \vdash \mathsf{T}^{\kappa}\ \overline{\tau}^{\kappa}}\ (wf_{tycon})^+ \qquad \frac{}{\hat{\Theta} \vdash \mathsf{a}}\ (wf_{var})^+$$

Definition 3.10 (Tiny-HS$^+$ environments). A variable environment $\hat{\Gamma} \in \mathsf{VarId} \xrightarrow{\text{fin}} \mathsf{TypSc}$ is a mapping between term variables and type schemes. A constraint environment $\hat{\Theta} \in \mathsf{Fin}(\mathsf{Constr}) \cup \mathsf{Fin}(\mathsf{ConstrSc})$ records constraints and constraint schemes.

The definition of superclasses must take multi-parameter type classes into account because a class can now have superclasses with fewer parameters than the class itself. Therefore, it is necessary to relate superclasses to the parameters of the class.

Definition 3.11 (Superclasses for Tiny-HS$^+$). The set of immediate superclasses of $\mathsf{C}\ \overline{\tau}^n$ in $\hat{\Theta}$ is $\mathrm{Sup}(\hat{\Theta}, \mathsf{C}\ \overline{\tau}^n) := \{[\overline{\tau_i/\mathsf{a}_i}^{\,i \in [n]}](\mathsf{C}'\ \overline{\mathsf{b}}^m) \mid \forall \mathsf{A}.\mathsf{C}\ \overline{\mathsf{a}}^n \Rightarrow \mathsf{C}'\ \overline{\mathsf{b}}^m \in \hat{\Theta}\}$.

The typing judgments for Tiny-HS$^+$ are shown in Figures 3.5 through 3.8. The judgments are a combination of the judgments given by Chakravarty et al. [CKP05] and the judgments for Tiny-HS from Section 3.2.2.

Well-formedness, entailment, and expression typing

The judgment in Figure 3.5 defines a well-formedness predicate $\hat{\Theta} \vdash \tau$ on types. $(wf_{syn})^+$ is the interesting rule; it ensures that an associated type synonym is applied only to types for which $\hat{\Theta}$ is strong enough to derive the class declaring the synonym.

The judgment $\hat{\Theta} \Vdash \pi$ for entailment is shown in Figure 3.6. The Tiny-HS rules $(elem_{entail})$, $(inst_{entail})$, and $(super_{entail})$ are merged into a single rule $(mp_{entail})^+$, and there are new rules for deriving nonsyntactic type equalities provoked by associated type synonyms.

The judgment $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \sigma$, also shown in Figure 3.6, assigns a type scheme σ to the expression w under the environments $\hat{\Theta}$ and $\hat{\Gamma}$. The rules are taken unchanged from [CKP05], with the addition of rule $(wildcard)^+$ and the ambiguity checks in rules $(let)^+$ and $(sig)^+$. The important rule of this judgment is rule $(conv)^+$, which incorporates nonsyntactic type equalities.

Figure 3.6. Judgments for entailment and expression typing in Tiny-HS$^+$

Entailment $\boxed{\hat{\Theta} \Vdash \pi}$

$$\frac{(\forall \mathsf{A}.\overline{\pi}^n \Rightarrow \mathsf{C}\ \overline{\tau'}) \in \hat{\Theta} \qquad \overline{\hat{\Theta} \Vdash \psi(\pi_i)}^{i\in[n]} \qquad \mathrm{Dom}(\psi) = \mathsf{A}}{\hat{\Theta} \Vdash \mathsf{C}\ \overline{\tau}} \ (mp_{entail})^+$$

$$\frac{(\forall \mathsf{A}.\eta' = \tau') \in \hat{\Theta} \qquad \psi(\eta' = \tau') = (\eta = \tau) \qquad \mathrm{Dom}(\psi) = \mathsf{A}}{\hat{\Theta} \Vdash \eta = \tau} \ (eqdef_{entail})^+ \qquad \frac{}{\hat{\Theta} \Vdash \tau = \tau} \ (eqrefl_{entail})^+$$

$$\frac{\hat{\Theta} \Vdash \tau_2 = \tau_1}{\hat{\Theta} \Vdash \tau_1 = \tau_2} \ (eqsymm_{entail})^+ \qquad \frac{\hat{\Theta} \Vdash \tau_1 = \tau_2 \qquad \hat{\Theta} \Vdash \tau_2 = \tau_3}{\hat{\Theta} \Vdash \tau_1 = \tau_3} \ (eqtrans_{entail})^+$$

$$\frac{\hat{\Theta} \Vdash [\tau_1/\mathsf{a}]\pi \qquad \hat{\Theta} \Vdash \tau_1 = \tau_2}{\hat{\Theta} \Vdash [\tau_2/\mathsf{a}]\pi} \ (eqsubst_{entail})^+$$

Expression typing $\boxed{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \sigma}$

$$\frac{\hat{\Gamma}(\mathsf{z}) = \sigma}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{z} : \sigma} \ (var)^+ \qquad \frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau' \qquad \hat{\Theta} \Vdash \tau' = \tau}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau} \ (conv)^+$$

$$\frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_1 : \tau' \to \tau \qquad \hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_2 : \tau'}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_1\ \mathsf{w}_2 : \tau} \ (\to E)^+$$

$$\frac{\hat{\Theta}; \hat{\Gamma}, \mathsf{z} \mapsto \tau' \vdash \mathsf{w} : \tau \qquad \hat{\Theta} \vdash \tau_1}{\hat{\Theta}; \hat{\Gamma} \vdash \lambda \mathsf{z}.\mathsf{w} : \tau' \to \tau} \ (\to I)^+ \qquad \frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau \qquad \hat{\Theta} \vdash \tau_1}{\hat{\Theta}; \hat{\Gamma} \vdash \lambda\_.\mathsf{w} : \tau' \to \tau} \ (wildcard)^+$$

$$\frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_1 : \sigma' \qquad \hat{\Theta}; \hat{\Gamma}, \mathsf{z} \mapsto \sigma' \vdash \mathsf{w}_2 : \sigma \qquad \sigma' \text{ unambiguous}}{\hat{\Theta}; \hat{\Gamma} \vdash \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2 : \sigma} \ (let)^+$$

$$\frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \sigma \qquad \mathrm{FV}^{\mathsf{a}}(\sigma) = \emptyset \qquad \sigma \text{ unambiguous}}{\hat{\Theta}; \hat{\Gamma} \vdash (\mathsf{w} :: \sigma) : \sigma} \ (sig)^+$$

$$\frac{(\hat{\Theta} \cup \{\pi\}); \hat{\Gamma} \vdash \mathsf{w} : \rho}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \pi \Rightarrow \rho} \ (\Rightarrow I)^+ \qquad \frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \pi \Rightarrow \rho \qquad \hat{\Theta} \Vdash \pi}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \rho} \ (\Rightarrow E)^+$$

$$\frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \forall \mathsf{A}.\rho \qquad \mathsf{a} \notin (\mathrm{FV}^{\mathsf{a}}(\hat{\Theta}) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Gamma}))}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \forall \mathsf{A} \cup \{\mathsf{a}\}.\rho} \ (\forall I)^+$$

$$\frac{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \forall \mathsf{A} \cup \{\mathsf{a}\}.\rho \qquad \hat{\Theta} \vdash \tau}{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : [\tau/\mathsf{a}](\forall \mathsf{A}.\rho)} \ (\forall E)^+$$

Figure 3.7. Typing judgments for Tiny-HS$^+$ instance definitions

Collecting type equalities $\boxed{\mathsf{A} \vdash^{\mathrm{o,i}} \mathsf{tdef} : \theta}$

$$\frac{}{\mathsf{A} \vdash^{\mathrm{o,i}} \textbf{type}\ \mathsf{S}^\kappa\ \overline{\tau}^\kappa = \tau' : \forall \mathsf{A}.\mathsf{S}^\kappa\ \overline{\tau}^\kappa = \tau'}\ (\mathit{type})^+$$

$$\frac{\mathsf{T}^\kappa \text{ fresh}}{\mathsf{A} \vdash^{\mathrm{o}} \textbf{abstype}\ \mathsf{S}^\kappa\ \overline{\tau}^\kappa = \tau' : \forall \mathsf{A}.\mathsf{S}^\kappa\ \overline{\tau}^\kappa = \mathsf{T}^\kappa\ \overline{\tau}^\kappa}\ (\mathit{abstype}_{\mathit{outside}})^+$$

$$\frac{}{\mathsf{A} \vdash^{\mathrm{i}} \textbf{abstype}\ \mathsf{S}^\kappa\ \overline{\tau}^\kappa = \tau' : \forall \mathsf{A}.\mathsf{S}^\kappa\ \overline{\tau}^\kappa = \tau'}\ (\mathit{abstype}_{\mathit{inside}})^+$$

Collecting instance constraints $\boxed{\vdash \mathsf{inst} : \hat{\Theta}}$

$$\frac{\theta \text{ well-formed} \qquad \overline{\mathsf{A} \vdash^{\mathrm{o}} \mathsf{tdef}_i : \theta_i}^{i \in [n]}}{\vdash \textbf{instance}\ \theta\ \textbf{where}\ \overline{\mathsf{tdef}}^n\ \overline{\mathsf{mval}} : \{\theta, \overline{\theta}^n\}}\ (\mathit{inst}_{\mathit{collect}})^+$$

Checking instance definitions $\boxed{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{inst}}$

$$\frac{\begin{array}{c}\hat{\Theta}'; \hat{\Gamma}; \overline{\tau} \overset{\text{method}}{\vdash} \mathsf{m}_i = \mathsf{w}_i \ \ (\text{for all } i \in [n]) \\ \hat{\Theta}'; \overline{\tau} \overset{\text{tdef}}{\vdash} \mathsf{tdef}_i \ \ (\text{for all } i \in [m]) \\ \hat{\Theta}' \setminus \{\forall \mathsf{A}.\overline{\pi}^r \Rightarrow \mathsf{C}\ \overline{\tau}\} \Vdash \mathsf{C}^{\mathsf{sup}}\ \overline{\tau'} \ \ (\text{for all } \mathsf{C}^{\mathsf{sup}}\ \overline{\tau'} \in \mathrm{Sup}(\hat{\Theta}, \mathsf{C}\ \overline{\tau})) \\ \hat{\Theta}' = \hat{\Theta} \cup \{\overline{\theta}^m, \overline{\pi}^r\} \qquad \overline{\mathsf{A} \vdash^{\mathrm{i}} \mathsf{tdef}_i : \theta_i}^{i \in [m]}\end{array}}{\hat{\Theta}; \hat{\Gamma} \vdash \textbf{instance}\ \forall \mathsf{A}.\overline{\pi}^r \Rightarrow \mathsf{C}\ \overline{\tau}\ \textbf{where}\ \overline{\mathsf{tdef}}^m\ \overline{\mathsf{m}_i = \mathsf{w}_i}^{i \in [n]}}\ (\mathit{inst}_{\mathit{check}})^+$$

$\boxed{\hat{\Theta}; \overline{\tau} \overset{\text{tdef}}{\vdash} \mathsf{tdef}}$

$$\frac{\hat{\Theta} \vdash \tau'}{\hat{\Theta}; \overline{\tau}^\kappa \overset{\text{tdef}}{\vdash} \textbf{type}\ \mathsf{S}^\kappa\ \overline{\tau}^\kappa = \tau'}\ (\mathit{inst}_{\mathit{check\text{-}type}})^+$$

$$\frac{\hat{\Theta} \vdash \tau'}{\hat{\Theta}; \overline{\tau}^\kappa \overset{\text{tdef}}{\vdash} \textbf{abstype}\ \mathsf{S}^\kappa\ \overline{\tau}^\kappa = \tau'}\ (\mathit{inst}_{\mathit{check\text{-}abstype}})^+$$

$\boxed{\hat{\Theta}; \hat{\Gamma}; \overline{\tau}^k \overset{\text{method}}{\vdash} \mathsf{m} = \mathsf{w}}$

$$\frac{\hat{\Gamma}(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \overline{\mathsf{a}}^k \Rightarrow \tau' \qquad \hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : [\overline{\tau_i/\mathsf{a}_i}^{i \in [k]}]\forall \mathsf{A}.\tau'}{\hat{\Theta}; \hat{\Gamma}; \overline{\tau}^k \overset{\text{method}}{\vdash} \mathsf{m} = \mathsf{w}}\ (\mathit{inst}_{\mathit{check\text{-}method}})^+$$

Instance definitions

The judgments for instance definitions are given in Figure 3.7. They are similar to those for Tiny-HS, except that they also deal with (abstract) associated type synonyms. The rules are complicated by the necessity to handle abstract associated type synonyms differently depending on whether the type equalities implied by them are used outside or inside an instance definition.

The judgment $\mathsf{A} \vdash^{\mathrm{o,i}} \mathsf{tdef} : \theta$ collects equality schemes, which introduce new nonsyntactic type equalities. There are two different versions of this judgment: version $\mathsf{A} \vdash^{\mathrm{o}} \mathsf{tdef} : \theta$ collects constraint schemes for use outside of the instance defining tdef, whereas version $\mathsf{A} \vdash^{\mathrm{i}} \mathsf{tdef} : \theta$ collects constraint schemes for use inside the instance. Consequently, rule $(\mathit{abstype}_{\mathit{outside}})^+$ equates an abstract associated type synonym with some fresh type constructor, whereas rule $(\mathit{abstype}_{\mathit{inside}})^+$ reveals the true identify of the synonym. The type variables in A are supposed to be the universally quantified type variables of the instance.

Rule $(\mathit{inst}_{\mathit{collect}})^+$ then uses the external variant $\mathsf{A} \vdash^{\mathrm{o}} \mathsf{tdef} : \theta$ to collect the constraint schemes resulting from the instance because these constraint schemes might be used outside the instance. However, rule $(\mathit{inst}_{\mathit{check}})^+$ extends the constraint environment $\hat{\Theta}$ with constraint schemes collected by the internal variant $\mathsf{A} \vdash^{\mathrm{i}} \mathsf{tdef} : \theta$ because it uses the extended environment $\hat{\Theta}'$ for checking methods, associated type synonym definitions, and superclass entailment inside the instance. As for Tiny-HS, we remove the constraint scheme resulting from the very instance from $\hat{\Theta}'$ before checking superclass entailment. Note that it is not possible that we remove accidentally some constraint scheme resulting from a subclass definition because instance heads of Tiny-HS$^+$ programs are required to contain at least one type constructor.

The well-formedness check $\hat{\Theta}'; \overline{\tau} \overset{\mathsf{tdef}}{\vdash} \mathsf{tdef}_i$ for associated type synonyms in rule $(\mathit{inst}_{\mathit{check}})^+$ assumes the whole constraint environment $\hat{\Theta}$. This is an important difference to Chakravarty and colleagues' system, which assumes only the superclass part of $\hat{\Theta}$ to avoid nonterminating associated type synonym definitions. However, their termination condition is not compatible with the way associated type synonyms are used in the translation from modules to type classes. We discuss on page 108 that the translation from modules to type classes does not produce nonterminating associated type synonym definitions; the formalization of a termination condition compatible with the translation is regarded as future work.

Class definitions and programs

The judgments for class definitions and programs are shown in Figure 3.8. They are a natural generalization of the corresponding judgments for Tiny-HS (see Figure 3.3). We add a new judgment $\hat{\Theta} \vdash \mathsf{cls}$ that checks whether the method signatures of cls are well-formed. Another new judgment $\vdash \mathsf{pgm} : \hat{\Theta}; \hat{\Gamma}$ is added because it is useful for the proofs in Chapter 4.

Figure 3.8. Typing judgments for Tiny-HS$^+$ class definitions and programs

Class definitions $\boxed{\vdash \mathsf{cls} : \hat{\Theta}; \hat{\Gamma}}$

$$\frac{\begin{array}{c}\forall \mathsf{A}.\overline{\pi}^r \Rightarrow \mathsf{C}\ \overline{\mathsf{a}}\ \text{well-formed} \\ \mathsf{A} \cap \mathsf{A}_i = \emptyset \qquad \sigma_i = \forall \mathsf{A}_i \cup \{\overline{\mathsf{a}}\}.\mathsf{C}\ \overline{\mathsf{a}} \Rightarrow \tau_i\ \text{unambiguous} \\ \mathrm{FV}^{\mathsf{a}}(\sigma_i) = \emptyset \qquad (\text{for all } i \in [n])\end{array}}{\begin{array}{l}\vdash \textbf{class}\ \forall \mathsf{A}.\overline{\pi}^r \Rightarrow \mathsf{C}\ \overline{\mathsf{a}}\ \textbf{where}\ \overline{\mathsf{tdec}}\ \overline{\mathsf{m}_i :: \forall \mathsf{A}_i.\tau_i}^{i\in[n]} \\ \quad : \{\forall \mathsf{A}.\mathsf{C}\ \overline{\mathsf{a}} \Rightarrow \pi_i \mid i \in [r]\}; \{\mathsf{m}_i \mapsto \sigma_i \mid i \in [n]\}\end{array}}\ (\mathit{class}_{\mathit{collect}})^+$$

$\boxed{\hat{\Theta} \vdash \mathsf{cls}}$

$$\frac{\hat{\Theta} \cup \{\mathsf{C}\ \overline{\mathsf{a}}\} \vdash \tau_i \ \ (\text{for all } i \in [n])}{\hat{\Theta} \vdash \textbf{class}\ \forall \mathsf{A}.\overline{\pi}^r \Rightarrow \mathsf{C}\ \overline{\mathsf{a}}\ \textbf{where}\ \overline{\mathsf{tdec}}\ \overline{\mathsf{m}_i :: \forall \mathsf{A}_i.\tau_i}^{i\in[n]}}\ (\mathit{class}_{\mathit{check}})^+$$

Programs $\boxed{\vdash \mathsf{pgm} : \hat{\Theta}; \hat{\Gamma}}$

$$\frac{\begin{array}{c}\overline{\vdash \mathsf{cls}_i : \hat{\Theta}_i; \hat{\Gamma}_i}^{i\in[n]} \qquad \overline{\vdash \mathsf{inst}_i : \hat{\Theta}'_i}^{i\in[m]} \\ \hat{\Theta} = \bigcup_{i\in[n]} \hat{\Theta}_i \cup \bigcup_{i\in[m]} \hat{\Theta}'_i \qquad \hat{\Gamma} = \dot{\bigcup}_{i\in[n]} \hat{\Gamma}_i \\ \overline{\hat{\Theta} \vdash \mathsf{cls}_i}^{i\in[n]} \qquad \overline{\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{inst}_i}^{i\in[m]}\end{array}}{\vdash \overline{\mathsf{cls}}^n\ \overline{\mathsf{inst}}^m : \hat{\Theta}; \hat{\Gamma}}\ (\mathit{prog}')^+$$

$\boxed{\vdash \mathsf{pgm}}$

$$\frac{\vdash \mathsf{pgm} : \hat{\Theta}; \hat{\Gamma}}{\vdash \mathsf{pgm}}\ (\mathit{prog})^+$$

3.4. Related work

Functional dependencies [Jon00a] are an alternative to associated type synonyms. They solve the problem of specifying dependencies among class parameters by adapting the notion of functional dependencies from database theory to Haskell type classes. For example, the class Add′ from Section 3.1 written with functional dependencies looks like this:

```
class Add' a b c | a b ⇝ c where
  add' :: a → b → c
```

Functional dependencies are well-explored, widely used, and available in major Haskell systems like GHC [ghc05] and Hugs [hug05], whereas associated type synonyms are a relatively recent development, for which only a prototype imple-

mentation[6] exists. Why did we then choose associated type synonyms instead of functional dependencies? The big advantage of associated type synonyms over functional dependencies is that you refer to an associated type synonym *by name*, whereas functional dependent class parameters are referred to *by position* in the class head. This advantage becomes even more relevant if we look ahead to Chapter 4 where type components of ML structures—which are also referred to by name—are translated into associated type synonyms.

Another form of associating types with type classes is to use data types instead of type synonyms. With such *associated data types* [CKPM05], we could write the Add′ class and a corresponding instance in the following way:

```
class Add′ a b where
  data Result a b
  add′ :: a → b → Result a b
instance Add′ Int Float where
  data Result Int Float = FloatResult Float
  add′ i f = FloatResult (primFloatAdd (intAsFloat i) f)
```

You see that associated data types introduce new type constructors (just as ordinary data types); hence, we have to wrap the result of the floatAdd function with the data constructor FloatResult. This property makes associated data types unsuitable for simulating type components of ML structures because type components of structures are type synonyms and do not introduce new types. However, ML also allows the definition of data types as structure components, a feature we have not introduced in Chapter 2. Associated data types would then correspond to data type components of ML structures.

[6] Available from http://www.cse.unsw.edu.au/~chak/papers/CKP05.html.

Chapter 4.

From modules to classes

Now we have set the scene and are ready to develop the translation from ML modules to Haskell type classes. To give you an intuition of how the translation works, Section 4.1 shows how a programmer would translate a particular piece of ML code to Haskell. Section 4.2 then presents the formal translation from Tiny-ML to Tiny-HS$^+$. In Section 4.3, we prove that every well-typed Tiny-ML program translates into a well-typed Tiny-HS$^+$ program. Section 4.4 explains why some features found in Standard ML and some extensions to Standard ML cannot be translated to Haskell type classes. Finally, Section 4.5 describes an implementation of the translation, and Section 4.6 discusses related work.

4.1. Example translation

Seeing how a programmer translates ML modules to Haskell type classes helps a lot to grasp the general idea of the formal translation. Therefore, we first discuss an example of such a manual translation, which is shown in Figure 4.1; the result of applying the formal translation to this example can be found in Appendix A.

The Tiny-ML code in Figure 4.1(a) is a slightly modified version of the MkSet functor example from Section 2.1. The difference is that the functor body does not define a separate type component for set elements; instead, it uses directly the type E.t provided by the functor argument. This modification is necessary to demonstrate a particular detail of the translation.

The Tiny-HS$^+$ version is shown in Figure 4.1(b).[1] We first translate the anonymous functor argument signature into a type class EQ. The type variable a is only used to index the associated type synonym T and the method eq; the type signature of eq would be ambiguous without the extra argument of type a. (We already saw this technique on page 39.)

The next step is to translate the anonymous result signature of the MkSet functor into a Haskell type class MK_SET. The class MK_SET has two parameters: the first parameter b represents a possible implementation of the functor body, and the second parameter a corresponds to the functor argument; it is needed to access the associated type synonym T of the EQ class, which is used in the type signatures of the methods member and insert as the translation of the type E.t. Now

[1] We bend the syntax of Tiny-HS$^+$ at some points to make the code more readable.

Figure 4.1. Translating modules to type classes by hand

(a) Example in Tiny-ML

```
functor MkSet (E : sig type t val eq : t → t → bool end) =
  struct
    type set       = list E.t
    val  empty     = []
    val  member    = λx . λs . exists (λy . E.eq x y) s
    val  insert    = λx . λs . if member x s then s else (cons x s)
  end :> sig
           type set
           val  empty  : set
           val  member : E.t → set → bool
           val  insert : E.t → set → set
         end
structure IntEq =
  struct
    type t  = int
    val eq  = λi . λj . primIntEq i j
  end
structure IntSet = MkSet (IntEq)
```

(b) Example translated to Tiny-HS$^+$ by hand

```
class EQ a where
  type T a
  eq :: a → T a → T a → Bool
class EQ a ⇒ MK_SET b a where
  type Set b a
  empty  :: b → a → Set b a
  member :: b → a → T a → Set b a → Bool
  insert :: b → a → T a → Set b a → Set b a
data MkSet
instance EQ a ⇒ MK_SET MkSet a where
  abstype Set MkSet a = [T a]
  empty  _ _     = []
  member _ a x s = exists (λy . eq a x y) s
  insert _ a x s = if member (⊥ :: MkSet) a x s then s else (x : s)
data IntEq
instance EQ IntEq where
  type T IntEq = Int
  eq _ i j     = primIntEq i j
```

Figure 4.2. Analogies between ML modules and Haskell type classes

ML	*Haskell*
structure signature	one-parameter type class
structure	instance of the corresponding type class
functor argument signature	one-parameter type class
functor result signature	two-parameter type class, which is a subclass of the functor argument signature
functor	instance of the functor result signature with the functor argument signature in the instance context
structure/functor name	data type
type specification	associated type synonym declaration
type definition	associated type synonym definition
type occurrence	associated type synonym applied to appropriate argument(s)
value specification	method signature
value definition	method implementation
value occurrence	method applied to appropriate argument(s)

you can see why the example presented here is a modification of the example in Section 2.1: If E.t did not appear in a value specification in the functor body, the second parameter of MK_SET would not be needed.

Now that we have translated the functor's argument and result signatures into Haskell type classes, we translate the functor body into an instance of type class MK_SET. We first define a data type MkSet, which corresponds to the name of the functor in ML. The instance definition itself is straightforward. The first parameter of the type class is filled with the name MkSet, whereas the second parameter—representing the functor argument—is left as a type variable. The constraint EQ a in the instance context is necessary because we use the associated type synonym T and the method eq of EQ in the definition of the abstract associated type synonym Set and the method member, respectively. The definition of insert shows that we use a method of the MK_SET instance from inside the instance the same way as from outside. Note that it is safe to use the diverging $\bot$ value because member (as well as all other methods of the instance) does not examine its first argument.

The translation of the functor application $\mathrm{MkSet}(\mathrm{IntEq})$ is straightforward: We only need to provide an appropriate EQ instance. There is no counterpart of the IntSet structure in Haskell. For example, to construct the singleton set $\{1\}$, we simply write insert $(\bot :: \mathsf{MkSet})$ $(\bot :: \mathsf{IntEq})$ 1 $(\mathsf{empty}(\bot :: \mathsf{MkSet})(\bot :: \mathsf{IntEq}))$. This also demonstrates that structures in Haskell are first-class by default because structures are just arbitrary values of a certain type.

Figure 4.2 summarizes the analogies between ML modules and Haskell type classes we encountered so far. A more complete comparison between ML modules

and Haskell type classes is deferred until Section 6.1.

4.2. Formal translation

The development of the formal translation from ML modules to Haskell type classes proceeds in two steps. Section 4.2.1 first gives some preparatory definitions; one of the things we define is a variant of Tiny-ML with type annotations. The actual translation in Section 4.2.2 then translates a Tiny-ML program with type annotations into a Tiny-HS$^+$ program. The translation from Tiny-ML to Tiny-HS$^+$ is then obtained by first annotating a Tiny-ML program using an extended version of Tiny-ML's typing judgments, and then translating the annotated program to Tiny-HS$^+$.

4.2.1. Preparations

This section prepares the translation by defining facilities for manipulating identifiers, by defining the environments used in the translation, and by defining a type-annotated variant of Tiny-ML, which is the actual source language of the translation. The typing judgments for Tiny-ML can be extended easily to add the type annotations to a Tiny-ML program.

Identifier manipulation

During the translation, we need to map Tiny-ML to Tiny-HS$^+$ identifiers and we need access to fresh identifiers. Figure 4.3 lists functions for this purpose and defines an intuitive shorthand notation for function application. The idea behind these functions becomes clear once we use them in the translation; for now, we just require that all functions are injective, and that their images are pairwise disjoint.

Moreover, the images of the last two functions are required to be disjoint from the structure and functor identifiers of the program under translation because we use these functions to generate fresh identifiers. Clearly, the two functions depend on the given Tiny-ML program.

We also postulate the existence of a set of fresh Tiny-HS$^+$ term variables and a set of fresh Tiny-HS$^+$ type variables, $\mathsf{FreshVarIds} \subseteq \mathsf{VarId}$ and $\mathsf{FreshTypVars} \subseteq \mathsf{TypVar}$, respectively. The sets $\mathsf{FreshVarIds}$ and $\mathsf{FreshTypVars}$ are required to be disjoint from the images of the functions in Figure 4.3, and must contain at least two elements.

Environments

Figure 4.4 shows the environments used in the translation from Tiny-ML to Tiny-HS$^+$. An occurrence environment Φ maps value occurrences and semantic type variables to the appropriate Tiny-HS$^+$ constructs. The Tiny-HS$^+$ translation of

Figure 4.3. Identifier manipulation functions

Function signature	*Function application*
$\mathrm{StrId} \to \mathsf{TyconId}$	T^{X}
$\mathrm{StrId} \to \mathsf{ClassId}$	C^{X}
$\mathrm{StrId} \times \mathrm{TypId} \to \mathsf{ASynId}$	$\mathsf{S}^{\mathrm{X.t}}$
$\mathrm{StrId} \times \mathrm{ValId} \to \mathsf{VarId}$	$\mathsf{z}^{\mathrm{X.y}}$
$\mathrm{FunId} \to \mathsf{TyconId}$	T^{F}
$\mathrm{FunId} \times \mathbb{N} \to \mathsf{TyconId}$	$\mathsf{T}^{\mathrm{F},k}$
$\mathrm{FunId} \to \mathsf{ClassId}$	C^{F}
$\mathrm{FunId} \to \mathsf{ClassId}$	$\mathsf{C}^{\mathrm{F,arg}}$
$\mathrm{FunId} \times \mathrm{TypId} \to \mathsf{ASynId}$	$\mathsf{S}^{\mathrm{F},\mathrm{t}}$
$\mathrm{FunId} \times \mathrm{ValId} \to \mathsf{VarId}$	$\mathsf{z}^{\mathrm{F},\mathrm{x}}$
$\mathrm{FunId} \times \mathbb{N} \times \mathrm{TypId} \to \mathsf{ASynId}$	$\mathsf{S}^{\mathrm{F},i,\mathrm{t}}$
$\mathrm{FunId} \times \mathbb{N} \times \mathrm{ValId} \to \mathsf{VarId}$	$\mathsf{z}^{\mathrm{F},i,\mathrm{x}}$
$\mathrm{CoreId} \to \mathsf{VarId}$	z^{c}
$\mathit{SimTypVar} \to \mathsf{TypVar}$	a'^{a}
$\mathrm{StrId} \to \mathrm{StrId}$	$\mathrm{X}^{\star}$
$\mathrm{FunId} \to \mathrm{FunId}$	$\mathrm{F}^{\star}$

Figure 4.4. Environments

Occurrence environment

$$\Phi := \left\{ \Phi^{l}_{\mathrm{x}} \cup \Phi^{g}_{\mathrm{x}} \cup \Phi_{\alpha} \;\middle|\; \begin{array}{l} \Phi^{l}_{\mathrm{x}} \in \mathrm{ValId} \xrightarrow{\text{fin}} \mathsf{Exp}, \\ \Phi^{g}_{\mathrm{x}} \in \mathrm{StrId} \times \mathrm{ValId} \xrightarrow{\text{fin}} \mathsf{Exp}, \\ \Phi_{\alpha} \in \mathit{TypVar} \xrightarrow{\text{fin}} \mathsf{ATyp}. \end{array} \right\}$$

Code environment

$$\Omega := \left\{ \Omega_{\mathrm{t}} \cup \Omega_{\mathrm{x}} \;\middle|\; \begin{array}{l} \Omega_{\mathrm{t}} \in \mathrm{TypId} \xrightarrow{\text{fin}} \mathsf{Typ}, \\ \Omega_{\mathrm{x}} \in \mathrm{ValId} \xrightarrow{\text{fin}} \mathsf{Exp} \end{array} \right\}$$

Figure 4.5. Syntax of Annotated Tiny-ML
(extends syntax in Figure 2.1 and changes syntax in Figure 2.2)

Structure bodies

$$\begin{aligned} \mathrm{StrBod} \ni \mathrm{b} ::= & \ldots \\ \mid & \ \textbf{type}\ \mathrm{t} = \mathrm{u}^{\langle u \rangle}; \mathrm{b} \end{aligned}$$

Structure expressions

$$\begin{aligned} \mathrm{StrExp} \ni \mathrm{s} ::= & \ldots \\ \mid & \ \mathrm{X}^{\langle \mathcal{S} \rangle} \\ \mid & \ \mathrm{F}^{\langle k, \mathcal{F} \rangle}(\overline{\mathrm{X}_i^{\langle \mathcal{S}_i \rangle}}^{i \in [n]}) \end{aligned} \qquad (k \in \mathbb{N},\ k \text{ unique among all functor applications in the program})$$

Sealed structure expressions

$$\begin{aligned} \mathrm{PStrExp}_{:>} \ni \mathrm{ps}_{:>} ::= & \ \mathrm{ps}^{\langle \exists P.\mathcal{S} \rangle} \\ \mid & \ \mathrm{ps}^{\langle \exists P'.\mathcal{S}' \rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle} \\ \mathrm{StrExp}_{:>} \ni \mathrm{s}_{:>} ::= & \ \mathrm{s}^{\langle \exists P.\mathcal{S} \rangle} \\ \mid & \ \mathrm{s}^{\langle \exists P'.\mathcal{S}' \rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle} \end{aligned}$$

Programs

$$\begin{aligned} \mathrm{Prog} \ni \mathrm{prog} ::= & \ldots \\ \mid & \ \textbf{functor}\ \mathrm{F}^{\langle \mathcal{F} \rangle}\ (\overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]}) = \mathrm{ps}_{:>}; \mathrm{prog} \end{aligned}$$

value occurrences x and $\mathrm{X.y}$ is given by $\Phi(\mathrm{x})$ and $\Phi(\mathrm{X},\mathrm{y})$, respectively. According to Figure 4.2, $\Phi(\mathrm{x})$ and $\Phi(\mathrm{X},\mathrm{y})$ are methods applied to appropriate arguments. The Tiny-HS$^+$ translation of a semantic type variable α is given by $\Phi(\alpha)$. Semantic type variables represent type occurrences, which are (according to Figure 4.2) translated into associated type synonyms applied to appropriate arguments. Hence, $\Phi(\alpha)$ is an element of ATyp. We shall see in Section 4.2.2 that the translation operates on semantic objects and not on syntactic types; therefore, we do not need to record information about type occurrences t and $\mathrm{X.t}$.

A code environment Ω maps type and value identifiers to the Tiny-HS$^+$ translations of the simple types and value expressions bound by the identifiers. For some type definition **type** $\mathrm{t} = \mathrm{u}$, $\Omega(\mathrm{t})$ is the translation of u, where u is the denotation of u. (We shall see in the next section how we get this u.) $\Omega(\mathrm{x})$ is the translation of the expression e for some value definition **val** $\mathrm{x} = \mathrm{e}$.

Annotated Tiny-ML

We already saw in the preceding section that the translation operates on semantic objects and not on syntactic types. Therefore, we need access to the semantic objects of several syntactic constructs during the translation.

For example, we said that $\Omega(\mathrm{t})$ is the translation of u, where u is the denotation of u for some type definition **type** $\mathrm{t} = \mathrm{u}$. In order to get the denotation of u (and the semantic objects of other syntactic constructs), we define in Figure 4.5 a type-annotated variant of Tiny-ML called *Annotated Tiny-ML*. The type annotations are written as superscripts enclosed in $\langle\rangle$. Note that we only need to annotate module language constructs. The functor identifier F in some functor application $\mathrm{F}(\overline{\mathrm{X}}^n)$ is not only annotated with the corresponding semantic functor $\mathcal{F}$, but also with some $k \in \mathbb{N}$ that is required to be distinct from the ks used in the annotations of all other functor applications in the program.

It is obvious how to add the annotations while constructing a typing derivation for a Tiny-ML program, so the rules are not shown here. In the following, we implicitly assume that the annotations result from a valid typing derivation.

4.2.2. The translation

This section presents the translation from Annotated Tiny-ML to Tiny-HS$^+$. The translation is organized as a set of functions, which map Annotated Tiny-ML constructs to the appropriate Tiny-HS$^+$constructs. We now discuss these functions in a bottom-up fashion; that is, we begin with the translation functions for value expressions and semantic types (where a semantic type is either a semantic simple type or a semantic value type), then continue with the translation of structure expressions, and finally discuss the translation of whole programs.

Figure 4.6. Translation of semantic types and value expressions

$$\boxed{\mathfrak{T}_u[\![u]\!]\Phi = \tau}$$

$$\begin{aligned}
\mathfrak{T}_u[\![`a]\!]\Phi &= \mathsf{a}^{`a} \\
\mathfrak{T}_u[\![T^\kappa\ \overline{u}^\kappa]\!]\Phi &= \mathsf{T}^\kappa\ \overline{\mathfrak{T}_u[\![u]\!]\Phi}^{i\in\kappa} \\
\mathfrak{T}_u[\![\alpha]\!]\Phi &= \Phi(\alpha)
\end{aligned}$$

$$\boxed{\mathfrak{T}_v[\![v]\!]\Phi = \sigma}$$

$$\mathfrak{T}_v[\![\forall A.u]\!]\Phi = \forall\{\mathsf{a}^{`a} \mid `a \in A\}.\mathfrak{T}_u[\![u]\!]\Phi$$

$$\boxed{\mathfrak{E}[\![\mathrm{e}]\!]\Phi = \mathsf{w}}$$

$$\begin{aligned}
\mathfrak{E}[\![\mathrm{c}]\!]\Phi &= \mathsf{z}^{\mathrm{c}} \\
\mathfrak{E}[\![\lambda\mathrm{c}.\mathrm{e}]\!]\Phi &= \lambda\mathsf{z}^{\mathrm{c}}.\mathfrak{E}[\![\mathrm{e}]\!]\Phi \\
\mathfrak{E}[\![\mathrm{e}_1\ \mathrm{e}_2]\!]\Phi &= \mathfrak{E}[\![\mathrm{e}_1]\!]\Phi\ \mathfrak{E}[\![\mathrm{e}_2]\!]\Phi \\
\mathfrak{E}[\![\textbf{let}\ \mathrm{c} = \mathrm{e}_1\ \textbf{in}\ \mathrm{e}_2]\!]\Phi &= \textbf{let}\ \mathsf{z}^{\mathrm{c}} = \mathfrak{E}[\![\mathrm{e}_1]\!]\Phi\ \textbf{in}\ \mathfrak{E}[\![\mathrm{e}_2]\!]\Phi \\
\mathfrak{E}[\![\mathrm{y}]\!]\Phi &= \Phi(\mathrm{y}) \\
\mathfrak{E}[\![\mathrm{X}.\mathrm{y}]\!]\Phi &= \Phi(\mathrm{X},\mathrm{y})
\end{aligned}$$

Translation of semantic types and value expressions

Figure 4.6 shows the translation functions $\mathfrak{T}_u$, $\mathfrak{T}_v$, and $\mathfrak{E}$, which translate semantic simple types, semantic value types, and value expressions into Tiny-HS$^+$ types, type schemes, and expressions, respectively. A semantic simple type variable $`a$ is translated into the corresponding Tiny-HS$^+$ type variable $\mathsf{a}^{`a}$ by using the appropriate identifier manipulation function from Figure 4.3. A semantic type variable α is translated into the Tiny-HS$^+$ type $\Phi(\alpha)$.

The translation of value expressions is straightforward as well: z^{c} is used as the translation of core variables c, and the occurrence environment Φ provides translations of value occurrences y and $\mathrm{X}.\mathrm{y}$.

Translation of structure bodies and unsealed structure expressions

Figure 4.7 shows the translation functions $\mathfrak{S}_\mathrm{b}$ and $\mathfrak{S}$ for structure bodies b and unsealed structure expressions s, respectively. The domain of the occurrence environment Φ used in the definitions of $\mathfrak{S}_\mathrm{b}$ and $\mathfrak{S}$ is expected to contain at least the following elements:

- All semantic type variables used in b or s.
- All value identifiers used in b or s.
- A pair (X,y) for all value identifiers y defined by some structure X provided X is used in b or s.

Figure 4.7. Translation of structure bodies and unsealed structure expressions

$$\boxed{\mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi = \Omega}$$

$$\begin{aligned}
\mathfrak{S}_{\mathrm{b}}[\![\mathbf{type}\ \mathrm{t} = \mathrm{u}^{\langle u\rangle}; \mathrm{b}]\!]\Phi &= \mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi, \mathrm{t} \mapsto \mathfrak{T}_u[\![u]\!]\Phi \\
\mathfrak{S}_{\mathrm{b}}[\![\mathbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b}]\!]\Phi &= \mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi, \mathrm{x} \mapsto \mathfrak{E}[\![\mathrm{e}]\!]\Phi \\
\mathfrak{S}_{\mathrm{b}}[\![\epsilon_{\mathrm{b}}]\!]\Phi &= \emptyset
\end{aligned}$$

$$\boxed{\mathfrak{S}[\![\mathrm{s}]\!]\Phi = \langle \Omega, \overline{\mathsf{ddec}}, \overline{\mathsf{inst}}\rangle}$$

$$\begin{aligned}
\mathfrak{S}[\![\mathbf{struct}\ \mathrm{b}\ \mathbf{end}]\!]\Phi &= \langle \mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi, \epsilon, \epsilon\rangle \\
\mathfrak{S}[\![\mathrm{X}^{\langle\mathcal{S}\rangle}]\!]\Phi &= \langle \{\mathrm{t} \mapsto \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S})\} \cup \\
&\qquad \{\mathrm{y} \mapsto \Phi(\mathrm{X}, \mathrm{y}) \mid \mathrm{y} \in \mathrm{Dom}(\mathcal{S})\}, \epsilon, \epsilon\rangle \\
\mathfrak{S}[\![\mathrm{F}^{\langle k, \forall Q.\overline{\mathcal{S}'}^n \to \exists P.\mathcal{S}'\rangle}(\overline{\mathrm{X}_i^{\langle\mathcal{S}_i\rangle}}^{i\in[n]})]\!]\Phi &= \langle \Omega, \mathsf{ddec}, \mathsf{inst}\rangle \\
\textit{where}\ \mathsf{ddec} &= \mathbf{data}\ \mathsf{T}^{\mathrm{F},k} \\
\mathsf{inst} &= \mathbf{instance}\ \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{T}^{\mathrm{F},k}\ \mathbf{where} \\
&\qquad \mathbf{type}\ \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{T}^{\mathrm{F},k} = \mathfrak{T}_u[\![\mathcal{S}_i(\mathrm{t})]\!]\Phi \quad i \in [n], \mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i') \\
&\qquad \mathsf{z}^{\mathrm{F},i,y} \qquad\qquad = \lambda_.\Phi(\mathrm{X}_i, \mathrm{y}) \quad i \in [n], \mathrm{y} \in \mathrm{Dom}(\mathcal{S}_i') \\
\Omega &= \{\mathrm{t} \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}')\} \cup \\
&\qquad \{\mathrm{x} \mapsto \mathsf{z}^{\mathrm{F},\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}})\ (\bot :: \mathsf{T}^{\mathrm{F},k}) \mid \mathrm{x} \in \mathrm{Dom}(\mathcal{S}')\}
\end{aligned}$$

In particular, the value identifiers defined by b must already be contained in Φ because the right-hand side of a value definition may use value identifiers introduced by earlier value definitions. The translation functions that make use of $\mathfrak{S}_{\mathrm{b}}$ and $\mathfrak{S}$ ensure that this precondition on Φ holds.

A structure body b is translated by $\mathfrak{S}_{\mathrm{b}}$ into a code environment Ω that contains the Tiny-HS$^+$ code for the components of b. $\mathfrak{S}_{\mathrm{b}}$ uses the annotation of a type definition to retrieve the semantic simple type corresponding to the right-hand side of the definition.

$\mathfrak{S}$ translates an unsealed structure expression into a triple consisting of a code environment Ω, and two sequences of data type and instance definitions. The case for enclosed structure bodies **struct** b **end** is straightforward because we can use the function $\mathfrak{S}_{\mathrm{b}}$.

The case for structure variables $\mathrm{X}^{\langle\mathcal{S}\rangle}$ is slightly more interesting. Conceptually, we simulate expanding the structure variable X into a structure body and returning the translation of this structure body; that is, the code environment of $\mathfrak{S}[\![\mathrm{X}^{\mathcal{S}}]\!]\Phi$ is the same as

$$\mathfrak{S}_{\mathrm{b}}[\![\overline{\mathbf{type}\ \mathrm{t} = \mathrm{X.t}^{\langle\mathcal{S}(\mathrm{t})\rangle}}^{\mathrm{t}\in\mathrm{Dom}(\mathcal{S})}\ \overline{\mathbf{val}\ \mathrm{y} = \mathrm{X.y}}^{\mathrm{y}\in\mathrm{Dom}(\mathcal{S})}]\!]\Phi.$$

The only case for which the sequences of data type and instance definitions of the result triple is not empty is the one for functor applications. In Tiny-ML, the actual functor arguments are matched implicitly against the argument signatures of the functor. In Tiny-HS$^+$, we have to make this matching explicit by creating a

new instance of the type class $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$, which is the translation of the functor argument signatures. We shall see how the type class $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$ is defined once we discuss the translation of functor definitions; for now, it suffices to know that $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$ is a single-parameter class that declares associated type synonyms $\mathsf{S}^{\mathrm{F},i,\mathrm{t}}$ and methods $\mathsf{z}^{\mathrm{F},i,\mathrm{x}}$ for all type and value components of all functor argument signatures $\mathcal{S}'_i$. We implement these associated type synonyms and methods by translating the semantic simple types found in the semantic structures $\mathcal{S}_i$ of the actual arguments, and by looking up the relevant value occurrences in the occurrence environment Φ, respectively.

The data type used in the instance head needs to be defined as well. We use the natural number k from the functor annotation to create an identifier for the data type that is unique among all other data types in the program.

The code environment Ω contains the Tiny-HS$^+$ code for all components of the functor's result signature $\mathcal{S}'$. We have not seen yet how functor definitions are translated, but all you need to know to understand the definition of Ω is the following: The body of a functor definition is translated into a two-parameter type class C^{F} that declares associated type synonyms $\mathsf{S}^{\mathrm{F},\mathrm{t}}$ and methods $\mathsf{z}^{\mathrm{F},\mathrm{x}}$ for all components of the body; the translation is done in the style of the example discussed in Section 4.1. Furthermore, an instance of this class is defined for T^{F} and some type variable a provided a is an instance of $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$. Therefore, $\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k}$ is the translation of a type component t, and $\mathsf{z}^{\mathrm{F},\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}})\ (\bot :: \mathsf{T}^{\mathrm{F},k})$ is the translation of a value component x of the result of the functor application.

Translation of structure definitions

We now discuss the translation of structure definitions; the next section then deals with the translation of functor definitions. However, before we can tackle the translation, we first have to clarify to which Tiny-HS$^+$ type a fresh semantic type variable introduced by some structure definition should be mapped.

We have already seen that the translation operates on semantic objects and not on syntactic types. Therefore, we have to extend the current occurrence environment whenever a new semantic type variable is introduced. Clearly, we should bind the new semantic type variable to an application of the associated type synonym that corresponds to the type component that introduced the semantic type variable. But we do not necessarily know which type component introduced the semantic type variable! Consider the following structure definition:

structure $\mathrm{X} =$ **struct type** $\mathrm{t} = \mathrm{int}$ **type** $\mathrm{s} = \mathrm{int}$ **end** :> **sig type** t **type** $\mathrm{s} = \mathrm{t}$ **end**

The semantic structure for X is $\mathcal{S} = \{\mathrm{t} \mapsto \alpha, \mathrm{s} \mapsto \alpha\}$ and that is all the translation knows. We cannot tell from looking at $\mathcal{S}$ whether t or s introduced α. Luckily, it does not really matter which type identifier we choose, so we can define an operation pick that selects some type identifier among the candidates. We must define pick slightly more general, so that it is also possible to select a type identifier among candidates from several semantic structures.

Definition 4.1 (The pick operation). The operation $\mathrm{pick}(\overline{\mathcal{S}}^n, \alpha) = \langle i, \mathrm{t} \rangle$ selects the lexicographically smallest $\langle i, \mathrm{t} \rangle$ with $i \in [n]$ and $\mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i)$ such that $\mathcal{S}_i(\mathrm{t}) = \alpha$. We write $\mathrm{pick}(\mathcal{S}, \alpha) = \mathrm{t}$ if $n = 1$.

Clearly, $\mathrm{pick}(\overline{\mathcal{S}}^n, \alpha)$ is only well-defined if there is some $i \in [n]$ and some $\mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i)$ such that $\mathcal{S}_i(\mathrm{t}) = \alpha$. For the rest of the translation, we use the pick operation without worrying about its well-definedness; instead, we prove in Section 4.3.1 that the whole translation (and so every usage of the pick operation) is well-defined.

Now we can turn our attention to the translation of structure definitions. We first define a function $\mathfrak{X}$ in Figure 4.8 that translates structure definitions **structure** $\mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S} \rangle}$ with unsealed right-hand sides. $\mathfrak{X}$ returns a triple of data type, class, and instance definitions. These definitions define a new type class C^{X}, and make the new data type T^{X} an instance of this class. The class C^{X} is the translation of the semantic structure $\mathcal{S}$; the general idea behind this translation was already discussed in Section 4.1. In order to translate a value type $\mathcal{S}(\mathrm{y})$ into a Tiny-HS$^+$ type scheme $\forall \mathsf{B}_{\mathrm{y}}.\tau_{\mathrm{y}}$, we need to extend the occurrence environment Φ with the semantic type variables in P because $\mathcal{S}(\mathrm{y})$ might contain these variables. The pick operation from Definition 4.1 is used to choose the appropriate associated type synonyms for these semantic type variables.

The instance definition for C^{X} T^{X} binds the translations of the type and value components of the structure expression s. We use the previously defined function $\mathfrak{S}$ to translate s. Note that Φ'' already contains the translation for occurrences of value components defined by s. However, Φ'' does not contain the semantic type variables in P. These semantic type variables are not needed because the semantic objects in the annotations of s can only contain variables from P if s has the form **struct** b **end**; but then P is empty.

Figure 4.9 shows the two cases for structure definitions of the function $\mathfrak{P}$ that translates an Annotated Tiny-ML program into a triple of data type, class, and instance definitions, which can be assembled to form a Tiny-HS$^+$ program. The two cases of $\mathfrak{P}$ for functor definitions are discussed in the next section.

There are two cases for structure definitions because we differentiate between unsealed and sealed right-hand sides. We cannot handle the two cases uniformly because code inside a sealed structure expression possibly knows more about the structure expression than code that accesses the sealed structure expression from outside. However, the method implementations of an instance definition in Tiny-HS$^+$ know—apart from the true identities of abstract associated type synonyms—only as much as code outside of the instance definition. Consider the following Tiny-ML example:

```
structure Y =
  struct
    val f = λc . c
    val b = f true
  end :> sig
```

Figure 4.8. Translation of structure definitions with unsealed right-hand sides

$$\boxed{\mathfrak{X}[\![\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S}\rangle}]\!]\Phi = \langle \overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}}\rangle}$$

$$\begin{array}{lll}
\mathfrak{X}[\![\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S}\rangle}]\!]\Phi = & & \\
\quad \langle \textbf{data}\ \mathsf{T}^{\mathrm{X}}\ \overline{\mathsf{ddec}} & & \\
\quad , \textbf{class}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{a}\ \textbf{where} & & \\
\qquad \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{a} & & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\
\qquad \mathsf{z}^{\mathrm{X.y}} :: \forall \mathsf{B}_{\mathrm{y}}.\mathsf{a} \to \tau_{\mathrm{y}} & & \mathrm{y} \in \mathrm{Dom}(\mathcal{S}) \\
\quad , \textbf{instance}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where} & & \\
\qquad \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \Omega(\mathrm{t}) & & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\
\qquad \mathsf{z}^{\mathrm{X.y}} \quad\quad = \lambda_.\Omega(\mathrm{y}) & & \mathrm{y} \in \mathrm{Dom}(\mathcal{S}) \\
\quad \overline{\mathsf{inst}}\rangle & & \\
\quad \textit{where} & & \\
\qquad \forall \mathsf{B}_{\mathrm{y}}.\tau_{\mathrm{y}} & = \mathfrak{T}_v[\![\mathcal{S}(\mathrm{y})]\!]\Phi' & \\
\qquad \Phi' & = \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{X.t}}\ \mathsf{a} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\} & \\
\qquad \langle \Omega, \overline{\mathsf{ddec}}, \overline{\mathsf{inst}}\rangle & = \mathfrak{S}[\![\mathrm{s}]\!]\Phi'' & \\
\qquad \Phi'' & = \Phi \dot\cup \{\mathrm{y} \mapsto \mathsf{z}^{\mathrm{X.y}}\ (\bot :: \mathsf{T}^{\mathrm{X}}) \mid \mathrm{y} \in \mathrm{Dom}(\mathcal{S})\} & \\
\qquad \mathsf{a} \in \mathsf{FreshTypVars} & &
\end{array}$$

$$\begin{array}{l}
\quad\quad\quad \textbf{val}\ \mathrm{f} : \mathrm{int} \to \mathrm{int} \\
\quad\quad\quad \textbf{val}\ \mathrm{b} : \mathrm{bool} \\
\quad\quad \textbf{end}
\end{array}$$

If we translated the example naively to Tiny-HS$^+$, we would end up with a program that does not type check:

$$\begin{array}{l}
\textbf{data}\ \mathsf{T}^{\mathrm{Y}} \\
\textbf{class}\ \mathsf{C}^{\mathrm{Y}}\ \mathsf{a}\ \textbf{where} \\
\quad \mathsf{f} :: \mathsf{a} \to \mathsf{Int} \to \mathsf{Int} \\
\quad \mathsf{b} :: \mathsf{a} \to \mathsf{Bool} \\
\textbf{instance}\ \mathsf{C}^{\mathrm{Y}}\ \mathsf{T}^{\mathrm{Y}}\ \textbf{where} \\
\quad \mathsf{f} = \lambda_\ .\ \lambda \mathsf{z}\ .\ \mathsf{z} \\
\quad \mathsf{b} = \lambda_\ .\ \mathsf{f}\ (\bot :: \mathsf{T}^{\mathrm{Y}})\ \mathsf{True}
\end{array}$$

We now continue with the explanation of the two cases for structure definitions (Figure 4.9). The case for structure definitions with an unsealed right-hand side is straightforward because all work is done by the previously defined function $\mathfrak{X}$. Note that we extend the occurrence environment Φ with the semantic type variables and the value identifiers introduced by the structure definition when translating the rest of the program.

The case for structure definitions with a sealed right-hand side is more involved. Conceptually, we simulate splitting the structure definition

Figure 4.9. Translation of structure definitions

$$\boxed{\mathfrak{P}[\![\mathrm{prog}]\!]\Phi = \langle \overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}} \rangle}$$

$$\mathfrak{P}[\![\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S} \rangle}; \mathrm{prog}]\!]\Phi = \langle \overline{\mathsf{ddec}}\ \overline{\mathsf{ddec}'}, \overline{\mathsf{cls}}\ \overline{\mathsf{cls}'}, \overline{\mathsf{inst}}\ \overline{\mathsf{inst}'} \rangle$$

where

$$\begin{array}{lll}
\langle \overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}} \rangle & = \mathfrak{X}[\![\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S} \rangle}]\!]\Phi & \\
\langle \overline{\mathsf{ddec}'}, \overline{\mathsf{cls}'}, \overline{\mathsf{inst}'} \rangle & = \mathfrak{P}[\![\mathrm{prog}]\!]\Phi' & \\
\Phi' & = \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\} & \\
 & \quad \dot\cup \{(\mathrm{X}, \mathrm{y}) \mapsto \mathsf{z}^{\mathrm{X.y}}\ (\bot :: \mathsf{T}^{\mathrm{X}}) \mid \mathrm{y} \in \mathrm{Dom}(\mathcal{S})\} &
\end{array}$$

$$\mathfrak{P}[\![\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P'.\mathcal{S}' \rangle} :> \mathrm{S}^{\langle \wedge P.\mathcal{S} \rangle}; \mathrm{prog}]\!]\Phi =$$

$$\begin{array}{lll}
\langle \textbf{data}\ \mathsf{T}^{\mathrm{X}}\ \overline{\mathsf{ddec}}\ \overline{\mathsf{ddec}'} & & \\
, \overline{\mathsf{cls}} & & \\
\quad \textbf{class}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{a}\ \textbf{where} & & \\
\qquad \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{a} & & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\
\qquad \mathsf{z}^{\mathrm{X.y}} :: \forall \mathsf{B}_{\mathrm{y}}.\mathsf{a} \to \tau_{\mathrm{y}} & & \mathrm{y} \in \mathrm{Dom}(\mathcal{S}) \\
\quad \overline{\mathsf{cls}'} & & \\
\quad , \textbf{instance}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where} & & \\
\qquad \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} & = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \notin P \\
\qquad \textbf{abstype}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} & = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \\
 & & \mathrm{t} = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})) \\
\qquad \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} & = \mathsf{S}^{\mathrm{X.t}'}\ \mathsf{T}^{\mathrm{X}} & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \\
 & & \mathrm{t}' = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})), \mathrm{t}' \neq \mathrm{t} \\
\qquad \mathsf{z}^{\mathrm{X.y}} & = \lambda_.\mathsf{z}^{\mathrm{X}^\star.\mathrm{y}}\ (\bot :: \mathsf{T}^{\mathrm{X}^\star}) & \mathrm{y} \in \mathrm{Dom}(\mathcal{S}) \\
\quad \overline{\mathsf{inst}}\ \overline{\mathsf{inst}'} \rangle & &
\end{array}$$

where

$$\begin{array}{lll}
\langle \overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}} \rangle & = \mathfrak{X}[\![\textbf{structure}\ \mathrm{X}^\star = \mathrm{s}^{\langle \exists P'.\mathcal{S}' \rangle}]\!]\Phi & \\
\langle \overline{\mathsf{ddec}'}, \overline{\mathsf{cls}'}, \overline{\mathsf{inst}'} \rangle & = \mathfrak{P}[\![\mathrm{prog}]\!]\Phi' & \\
\Phi' & = \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\} & \\
 & \quad \dot\cup \{(\mathrm{X}, \mathrm{y}) \mapsto \mathsf{z}^{\mathrm{X.y}}\ (\bot :: \mathsf{T}^{\mathrm{X}}) \mid \mathrm{y} \in \mathrm{Dom}(\mathcal{S})\} & \\
\forall \mathsf{B}_{\mathrm{y}}.\tau_{\mathrm{y}} & = \mathfrak{T}_v[\![\mathcal{S}(\mathrm{y})]\!]\Phi'' & \\
\Phi'' & = \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{X.t}}\ \mathsf{a} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\} & \\
\mathsf{a} \in \mathsf{FreshTypVars} & &
\end{array}$$

$$\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P'.\mathcal{S}' \rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle}$$

into two structure definitions

$$\textbf{structure}\ \mathrm{X}^\star = \mathrm{s}^{\langle \exists P'.\mathcal{S}' \rangle}$$
$$\textbf{structure}\ \mathrm{X}\ = \mathrm{X}^{\star\langle \mathcal{S}' \rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle}.$$

The structure definition for $\mathrm{X}^\star$ is translated using the function $\mathfrak{X}$. Then we define a class C^{X}, which is the translation of the semantic structure $\mathcal{S}$, and make the new data type T^{X} an instance of this class. The instance definition simply copies all methods and those associated type synonyms, which correspond to type components that do not introduce new abstract types, from the translation of $\mathrm{X}^\star$. Type components t with $\mathcal{S}(\mathrm{t}) \in P$ introduce new abstract types, so they must be treated differently. There are two sorts of type components introducing abstract types: those which are selected by the pick operation to represent some semantic type variable, and those which are not selected. The former are translated into abstract associated type synonyms, whereas the latter are translated into regular associated type synonyms. The definitions of these regular associated type synonyms propagate the relevant type equalities between type components selected by pick and those which are not.

Strictly speaking, the propagation of type equalities is not needed because the newly introduced semantic type variables are always represented by the associated type synonym that is selected by pick (this can be seen from the definition of Φ'). However, if the formal translation is used as a model for a manual translation, the propagation of such type equalities gives us the freedom to translate a semantic type variable into applications of different associated type synonyms; hence, we can maintain a closer match between the original Tiny-ML source program and the translated Tiny-HS$^+$ program.

Figure 4.9 contains associated type synonym definitions that would be rejected by Chakravarty and colleagues' system [CKP05] because they might not terminate. For example, the definition **type** $\mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star}$ is not valid in their system because $\mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}$ is an associated type synonym of class $\mathsf{C}^{\mathrm{X}^\star}$ but $\mathsf{C}^{\mathrm{X}^\star}\ \mathsf{T}^{\mathrm{X}^\star}$ is not derivable without using the instance definitions of the program. Tiny-HS$^+$ accepts the definition because it uses the instance definitions of the program when checking the well-formedness of associated type synonym definitions. Note that such associated type synonym definitions, which are well-formed in Tiny-HS$^+$ but not in the system proposed by Chakravarty et al., do not occur only in Figure 4.9 but at all places where the right-hand side of an associated type synonym definition is the translation of a type u that contains a type variable α. We discuss in Section 6.1.1 that all associated type synonym definitions in a Tiny-HS$^+$ program, which results from the translation of a Tiny-ML program, are terminating.

Translation of functor definitions

The translation of functor definitions is the last piece missing to complete the translation from Annotated Tiny-ML to Tiny-HS$^+$. Similar to the translation of struc-

Figure 4.10. Translation of functor definitions with unsealed right-hand sides

$$\boxed{\mathfrak{F}[\![\textbf{functor}\ \mathrm{F}^{\langle\forall Q.\overline{\mathcal{S}}^n\to\mathcal{S}\rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) = \mathrm{ps}^{\langle\mathcal{S}\rangle}]\!]\mathrm{F}'\Phi = \langle\overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}}\rangle}$$

$$\begin{array}{l}
\mathfrak{F}[\![\textbf{functor}\ \mathrm{F}^{\langle\forall Q.\overline{\mathcal{S}}^n\to\mathcal{S}\rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) = \textbf{struct}\ \mathrm{b}\ \textbf{end}^{\langle\mathcal{S}\rangle}]\!]\mathrm{F}'\Phi = \\
\quad\langle \textbf{data}\ \mathsf{T}^{\mathrm{F}} \\
\quad, \textbf{class}\ \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a}\ \textbf{where} \\
\qquad \textbf{type}\ \mathsf{S}^{\mathrm{F}',i,\mathrm{t}}\ \mathsf{a} && i \in [n], \mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i) \\
\qquad \mathsf{z}^{\mathrm{F}',i,\mathrm{x}} :: \forall \mathrm{B}_{\mathrm{x},i}.\mathsf{a} \to \tau_{\mathrm{x},i} && i \in [n], \mathrm{x} \in \mathrm{Dom}(\mathcal{S}_i) \\
\quad \textbf{class}\ \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}\ \textbf{where} \\
\qquad \textbf{type}\ \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{b}\ \mathsf{a} && \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\
\qquad \mathsf{z}^{\mathrm{F},\mathrm{x}} :: \forall \mathrm{B}_{\mathrm{x}}.\mathsf{b} \to \mathsf{a} \to \tau_{\mathrm{x}} && \mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \\
\quad, \textbf{instance}\ \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\ \textbf{where} \\
\qquad \textbf{type}\ \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \Omega(\mathrm{t}) && \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\
\qquad \mathsf{z}^{\mathrm{F},\mathrm{x}} = \lambda_.\lambda \mathsf{z}.\Omega(\mathrm{x}) && \mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \\
\quad\rangle \\
\quad \textit{where} \\
\qquad \forall \mathrm{B}_{\mathrm{x},i}.\tau_{\mathrm{x},i} = \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi' \\
\qquad \forall \mathrm{B}_{\mathrm{x}}.\tau_{\mathrm{x}} = \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi' \\
\qquad \Phi' = \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F}',i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}}, \alpha)\} \\
\qquad \Omega = \mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi'' \\
\qquad \Phi'' = \Phi' \dot\cup \{(\mathrm{X}_i, \mathrm{y}) \mapsto \mathsf{z}^{\mathrm{F}',i,\mathrm{y}}\ \mathsf{z} \mid i \in [n], \mathrm{y} \in \mathrm{Dom}(\mathcal{S}_i)\} \\
\qquad\qquad \dot\cup \{\mathrm{x} \mapsto \mathsf{z}^{\mathrm{F},\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}})\ \mathsf{z} \mid \mathrm{x} \in \mathrm{Dom}(\mathcal{S})\} \\
\qquad \mathsf{z} \in \mathsf{FreshVarIds}, \mathsf{a} \neq \mathsf{b} \in \mathsf{FreshTypVars}.
\end{array}$$

ture definitions, we first define in Figure 4.10 a function $\mathfrak{F}$ that translates functor definitions with unsealed right-hand sides. This function takes an extra functor identifier F'. We shall see later why this extra functor identifier is needed. $\mathfrak{F}$ accepts only functor definitions for which the semantic structure $\mathcal{S}$ in the annotation on the left-hand and on the right-hand side are equal. All well-typed functor definitions satisfy this criteria.

The triple of data type, class, and instance definitions returned by $\mathfrak{F}$ define a new class $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$, which is the translation of the argument signatures $\overline{\mathcal{S}}^n$. We already discussed on page 57 that instances of the class $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$ are used to translate functor applications. Note that we accumulate *all* argument signatures into a single type class. To translate a value type $\mathcal{S}_i(\mathrm{x})$ to a Tiny-HS$^+$ type scheme $\forall \mathrm{B}_{\mathrm{x},i}.\tau_{\mathrm{x},i}$, we need to extend the occurrence environment with the semantic type variables in Q.

We also define a new type class C^{F} and make the new data type T^{F} an instance of C^{F}. The class C^{F} is the translation of the functor result signature $\mathcal{S}$. $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$ has to be a superclass of C^{F} because $\mathcal{S}(\mathrm{x})$ might contain semantic type variables from Q, which are represented by applications of associated type synonyms declared

in $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$. The instance definition for C^{F} uses the code obtained by translating the functor body b with the function $\mathfrak{S}_{\mathrm{b}}$ defined in Figure 4.7. Note that we extend the occurrence environment with the semantic type variables and the value components introduced by the functor arguments, and with the value components defined in the functor body.

Figure 4.11 shows the two missing cases of the program translation function $\mathfrak{P}$. This function puts, similar to $\mathfrak{F}$, certain restrictions on the annotations of the functor definitions it accepts. All type correct functor definitions fulfill these restrictions.

The case for functor definitions with an unsealed right-hand side is trivial because all work is done by the function $\mathfrak{F}$. Note that we do not need to extend the occurrence environment Φ for translating the rest of the program because a functor definition per se does not introduce any new semantic type variables or value identifiers.

The case for functor definitions with a sealed right-hand side is more complicated. Along the lines of the case for structure definitions, we conceptually simulate splitting the functor definition

$$\textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S}\rangle}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i\in[n]}) = \mathrm{ps}^{\langle \mathcal{S}'\rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S}\rangle}$$

into two functor definitions[2]

$$\begin{aligned}&\textbf{functor}\ \mathrm{F}^{\star\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}'\rangle}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i\in[n]}) &&= \mathrm{ps}^{\langle \mathcal{S}'\rangle}\\ &\textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S}\rangle}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i\in[n]}) &&= \mathrm{F}^{\star}(\overline{\mathrm{X}}^n) :> \mathrm{S}^{\langle \Lambda P.\mathcal{S}\rangle}.\end{aligned}$$

The functor definition for $\mathrm{F}^{\star}$ is translated by the function $\mathfrak{F}$. Now we can see why the function $\mathfrak{F}$ takes an extra functor identifier as a parameter: The functor argument signatures need to be translated into a type class $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$, whereas the functor result signature must be translated into a type class $\mathsf{C}^{\mathrm{F}^{\star}}$. $\mathfrak{F}$ generates an instance definition **instance** $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$ a $\Rightarrow$ $\mathsf{C}^{\mathrm{F}^{\star}}$ $\mathsf{T}^{\mathrm{F}^{\star}}$ a **where** ... as well.

Then we define a class C^{F} as the translation of the functor result signature $\mathcal{S}$. The instance definition **instance** $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$ a $\Rightarrow$ C^{F} T^{F} a **where** ... simply copies all methods and those associated type synonyms, which correspond to type components that do not introduce new abstract types, from the translation of $\mathrm{F}^{\star}$. Type components that introduce abstract types are translated along the lines of the translation of structure definitions with sealed right-hand sides (see page 62).

However, this time we could not do without propagating type equalities between abstract types. Consider the following example:

```
functor F (X : sig end) =
  struct
    type s = int
```

[2]The functor application on the right-hand side of the second functor definition is not valid Annotated Tiny-ML because the annotations for $\mathrm{F}^{\star}$ and for the arguments X_i are missing. However, we do not really split the functor definition in the translation; the code should only illustrate the idea behind the translation.

Figure 4.11. Translation of functor definitions

$$\boxed{\mathfrak{P}[\![\mathrm{prog}]\!]\Phi = \langle \overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}}\rangle}$$

$$\mathfrak{P}[\![\textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}\rangle}(\overline{\mathrm{X_i : S_i}}^{i\in[n]}) = \mathrm{ps}^{\langle \mathcal{S}\rangle}; \mathrm{prog}]\!]\Phi =$$
$$\langle \overline{\mathsf{ddec}}\ \overline{\mathsf{ddec}'}, \overline{\mathsf{cls}}\ \overline{\mathsf{cls}'}, \overline{\mathsf{inst}}\ \overline{\mathsf{inst}'}\rangle$$
where
$$\begin{aligned}
\langle \overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}}\rangle &= \mathfrak{F}[\![\textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}\rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) = \mathrm{ps}^{\langle \mathcal{S}\rangle}]\!]\mathrm{F}\Phi \\
\langle \overline{\mathsf{ddec}'}, \overline{\mathsf{cls}'}, \overline{\mathsf{inst}'}\rangle &= \mathfrak{P}[\![\mathrm{prog}]\!]\Phi
\end{aligned}$$

$$\mathfrak{P}[\![\textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S}\rangle}(\overline{\mathrm{X_i : S_i}}^{i\in[n]}) = \mathrm{ps}^{\langle \mathcal{S}'\rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S}\rangle}; \mathrm{prog}]\!]\Phi =$$

$$\begin{array}{lll}
\langle\, \textbf{data}\ \mathsf{T^F}\ \overline{\mathsf{ddec}}\ \overline{\mathsf{ddec}'} & & \\
,\ \overline{\mathsf{cls}} & & \\
\quad \textbf{class}\ \mathsf{C^{F,arg}\ a \Rightarrow C^F\ b\ a}\ \textbf{where} & & \\
\qquad \overline{\textbf{type}\ \mathsf{S^{F,t}\ b\ a}} & & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\
\qquad \overline{\mathsf{z^{F,x}} :: \forall \mathsf{B_x.b \to a} \to \tau_{\mathrm{x}}} & & \mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \\
\quad \overline{\mathsf{cls}'} & & \\
,\ \textbf{instance}\ \mathsf{C^{F,arg}\ a \Rightarrow C^F\ T^F\ a}\ \textbf{where} & & \\
\qquad \overline{\textbf{type}\ \mathsf{S^{F,t}\ T^F\ a}} & = \mathsf{S^{F^\star,t}\ T^{F^\star}\ a} & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \notin P \\
\qquad \overline{\textbf{abstype}\ \mathsf{S^{F,t}\ T^F\ a}} & = \mathsf{S^{F^\star,t}\ T^{F^\star}\ a} & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P \\
 & & \mathrm{t} = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})) \\
\qquad \overline{\textbf{type}\ \mathsf{S^{F,t}\ T^F\ a}} & = \mathsf{S^{F,t'}\ T^F\ a} & \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P \\
 & & \mathrm{t}' = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})), \mathrm{t}' \neq \mathrm{t} \\
\qquad \overline{\mathsf{z^{F,x}}} & = \lambda_.\lambda \mathsf{z.z^{F^\star,x}}\ (\bot :: \mathsf{T^{F^\star}})\ \mathsf{z} & \mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \\
\quad \overline{\mathsf{inst}}\ \overline{\mathsf{inst}'}\rangle & &
\end{array}$$

where
$$\begin{aligned}
\langle \overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}}\rangle &= \mathfrak{F}[\![\textbf{functor}\ \mathrm{F}^{\star\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}'\rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) = \mathrm{ps}^{\langle \mathcal{S}'\rangle}]\!]\mathrm{F}\Phi \\
\langle \overline{\mathsf{ddec}'}, \overline{\mathsf{cls}'}, \overline{\mathsf{inst}'}\rangle &= \mathfrak{P}[\![\mathrm{prog}]\!]\Phi \\
\forall \mathsf{B_x}.\tau_{\mathrm{x}} &= \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi' \\
\Phi' &= \Phi \mathbin{\dot\cup} \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}}, \alpha)\} \\
&\quad \mathbin{\dot\cup} \{\alpha \mapsto \mathsf{S^{F,t}\ b\ a} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\}
\end{aligned}$$
$$\mathsf{z} \in \mathsf{FreshVarIds}, \mathsf{a} \neq \mathsf{b} \in \mathsf{FreshTypVars}$$

```
    type t = int
    val  x = 0
  end :> sig type s type t = s val x : s end
structure Empty = struct end
structure Y = F (Empty) :> sig type t val x : t end
```

If we translated the example to Tiny-HS$^+$ without translating the type equality between s and t, then the translation of the right-hand side of the definition of Y did not type check.

4.3. Formal properties

After having defined the translation from Tiny-ML to Tiny-HS$^+$ in the preceding section, we now prove formal properties of it. The main results are that the translation is well-defined for every type correct program (Section 4.3.1) and that the result of the translation is a type correct Tiny-HS$^+$ program given the source program is type correct (Section 4.3.2). Taken together, this means that every type correct Tiny-ML program translates into a type correct Tiny-HS$^+$ program. However, it does not mean that the translation is sound. To prove soundness, we would also need to relate the dynamic behavior of a source program to its translation. Nevertheless, the results presented here are a strong indication that the translation is indeed sound. The material in this section is not important for understanding the rest of thesis, so you may skip this section if you are not interested in the formal details.

There is one issue we need to discuss before we can start with the proofs: The translation from Tiny-ML to Tiny-HS$^+$ first translates a Tiny-ML program into an Annotated Tiny-ML program, and then translates the Annotated Tiny-ML program into a Tiny-HS$^+$ program. However, we have not formalized the translation from Tiny-ML to Annotated Tiny-ML because it is obvious how to define such a translation based on Tiny-ML's typing judgments. Therefore, we assume implicitly that every Annotated Tiny-ML fragment complies with the typing derivation for the corresponding Tiny-ML fragment under discussion.

4.3.1. Well-definedness

It is not obvious that the translation is well-defined for every well-typed program. Here are some examples for what could go wrong:

- $\mathrm{pick}(\mathcal{S}, \alpha)$ is not well-defined of there exists no $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$ with $\mathcal{S}(\mathrm{t}) = \alpha$.
- $\mathfrak{E}[\![\mathrm{e}]\!]\Phi$ is not well-defined if e contains value occurrences not covered by Φ.
- $\mathfrak{T}_u[\![u]\!]\Phi$ is not well-defined if $\mathrm{FV}^\alpha(u) \not\subseteq \mathrm{Dom}(\Phi)$.

Therefore, we prove in this section that the translation is indeed well-defined for every type correct program. We start with some definitions and lemmata that are important for proving the well-definedness of usages of the pick operation.

Definition 4.2 (Solvability). A signature $\mathcal{S}$ is said to be solvable with respect to $P \subseteq \mathit{TypVar}$, written $\mathrm{Solv}(\mathcal{S}, P)$, if for all $\alpha \in P$ there exists some $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$ such that $\mathcal{S}(\mathrm{t}) = \alpha$.

Lemma 4.3 (Solvability of signature expressions). *If* $\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S}$, *then we have also* $\mathrm{Solv}(\mathcal{S}, P)$.

Proof. Simple induction on the rules defining $\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}$ and $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$. □

Remark. Definition 4.2 and Lemma 4.3 correspond to Definition 4.2 and Lemma 4.3 in Russo's thesis [Rus98, page 122], respectively.

Definition 4.4 (Groundness). A semantic functor $\mathcal{F} = \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S}$ is ground if $\mathrm{Solv}(\mathcal{S}, P)$, and there exist sets Q_i with $Q = \cup_{i \in [n]} Q_i$ such that $\mathrm{Solv}(\mathcal{S}_i, Q_i)$ for all $i \in [n]$. A context $\mathcal{C}$ is ground if $\mathcal{C}(\mathrm{F})$ is ground for all $\mathrm{F} \in \mathrm{Dom}(\mathcal{C})$.

Remark. The preceding definition of groundness is similar to Definition 4.5 in Russo's thesis [Rus98]. The difference is that Russo does not postulate solvability of the result signature $\exists P.\mathcal{S}$.

Lemma 4.5 (Solvability of structure expressions). *Let* $\mathcal{C}$ *be ground. If* $\mathcal{C} \vdash \mathrm{s}_{:>} : \exists P.\mathcal{S}$, *then* $\mathrm{Solv}(\mathcal{S}, P)$.

Proof. Straightforward rule induction. □

Remark. Lemma 4.5 would not hold if we allowed arbitrary structure expressions (and not only structure variables) as functor arguments.

The following lemma states that denotation and classification judgments do not introduce new type variables. The lemma is taken from Russo's thesis as well.

Lemma 4.6 (Free type variables and typing judgments).

- $\mathcal{C} \vdash \mathrm{u} \triangleright u$ *implies* $\mathrm{FV}^\alpha(u) \subseteq \mathrm{FV}^\alpha(\mathcal{C})$ *and* $\mathrm{FV}^{'a}(u) \subseteq \mathrm{FV}^{'a}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{v} \triangleright v$ *implies* $\mathrm{FV}^\alpha(v) \subseteq \mathrm{FV}^\alpha(\mathcal{C})$ *and* $\mathrm{FV}^{'a}(v) \subseteq \mathrm{FV}^{'a}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}$ *implies* $\mathrm{FV}^\alpha(\mathcal{L}) \subseteq \mathrm{FV}^\alpha(\mathcal{C})$ *and* $\mathrm{FV}^{'a}(\mathcal{L}) \subseteq \mathrm{FV}^{'a}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$ *implies* $\mathrm{FV}^\alpha(\mathcal{L}) \subseteq \mathrm{FV}^\alpha(\mathcal{C})$ *and* $\mathrm{FV}^{'a}(\mathcal{L}) \subseteq \mathrm{FV}^{'a}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{e} : v$ *implies* $\mathrm{FV}^\alpha(v) \subseteq \mathrm{FV}^\alpha(\mathcal{C})$ *and* $\mathrm{FV}^{'a}(v) \subseteq \mathrm{FV}^{'a}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{b} : \mathcal{S}$ *implies* $\mathrm{FV}^\alpha(\mathcal{S}) \subseteq \mathrm{FV}^\alpha(\mathcal{C})$ *and* $\mathrm{FV}^{'a}(\mathcal{S}) \subseteq \mathrm{FV}^{'a}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{s}_{:>} : \mathcal{X}$ *implies* $\mathrm{FV}^\alpha(\mathcal{X}) \subseteq \mathrm{FV}^\alpha(\mathcal{C})$, *provided* $\mathcal{C}$ *is ground.*

- $\mathcal{C} \vdash \mathrm{s}_{:>} : \mathcal{X}$ *implies* $\mathrm{FV}'^{a}(\mathcal{X}) \subseteq \mathrm{FV}'^{a}(\mathcal{C})$.

Proof. All claims are proved by straightforward rule inductions. □

The following definition relates a context $\mathcal{C}$ to an occurrence environment Φ.

Definition 4.7 (Validity of occurrence environments). Φ is said to be valid with respect to $\mathcal{C}$, written $\mathrm{Valid}(\mathcal{C}, \Phi)$, if $\mathrm{FV}^{\alpha}(\mathcal{C}) \subseteq \mathrm{Dom}(\Phi)$ and $\{\mathrm{x} \mid \mathrm{x} \in \mathrm{Dom}(\mathcal{C})\} \cup \{(\mathrm{X}, \mathrm{y}) \mid \mathrm{X} \in \mathrm{Dom}(\mathcal{C}), \mathrm{y} \in \mathrm{Dom}(\mathcal{C}(\mathrm{X}))\} \subseteq \mathrm{Dom}(\Phi)$.

Now we can prove the well-definedness of all translation functions except the program translation function $\mathfrak{P}$.

Lemma 4.8 (Well-definedness of expression translation). $\mathfrak{E}[\![\mathrm{e}]\!]\Phi$ *is well-defined if* $\mathcal{C} \vdash \mathrm{e} : u$ *and* $\mathrm{Valid}(\mathcal{C}, \Phi)$.

Proof. Straightforward induction on the derivation of $\mathcal{C} \vdash \mathrm{e} : u$. □

Lemma 4.9 (Well-definedness of type translation). $\mathfrak{T}_u[\![u]\!]\Phi$ *and* $\mathfrak{T}_v[\![v]\!]\Phi$ *are well-defined if* $\mathrm{FV}^{\alpha}(u) \subseteq \mathrm{Dom}(\Phi)$ *and* $\mathrm{FV}^{\alpha}(v) \subseteq \mathrm{Dom}(\Phi)$, *respectively.*

Proof. Obvious. □

Lemma 4.10 (Well-definedness of $\mathfrak{S}_{\mathrm{b}}$). $\mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi$ *is well-defined provided* $\mathcal{C} \vdash \mathrm{b} : \mathcal{S}$, $\mathrm{Valid}(\mathcal{C}, \Phi)$, *and* $\mathrm{Dom}(\mathcal{S}_{\mathrm{x}}) \subseteq \mathrm{Dom}(\Phi)$.

Proof. By induction on the structure of b. See Appendix B, page 115. □

Lemma 4.11 (Well-definedness of structure expression translation). $\mathfrak{S}[\![\mathrm{s}]\!]\Phi$ *is well-defined if* $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$, $\mathrm{Valid}(\mathcal{C}, \Phi)$, *and* $\mathrm{Dom}(\mathcal{S}_{\mathrm{x}}) \subseteq \mathrm{Dom}(\Phi)$.

Proof. By structural induction on s. See Appendix B, page 115. □

Lemma 4.12. *If* $\mathfrak{S}[\![\mathrm{s}]\!]\Phi = \langle \Omega, \overline{\mathsf{ddec}}, \overline{\mathsf{inst}} \rangle$ *and* $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$, *then* $\mathrm{Dom}(\Omega) = \mathrm{Dom}(\mathcal{S})$.

Proof. Straightforward structural induction on s. □

Lemma 4.13 (Well-definedness of translation of structure definitions with unsealed right-hand sides). $\mathfrak{X}[\![\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \mathcal{X} \rangle}]\!]\Phi$ *is well-defined if* $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$, $\mathrm{Valid}(\mathcal{C}, \Phi)$, *and* $\mathcal{C}$ *ground.*

Proof. See Appendix B, page 116. □

Lemma 4.14 (Well-definedness of translation of functor definitions with unsealed right-hand sides). $\mathfrak{F}[\![\textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^{n} \to \mathcal{S} \rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i \in [n]}) = \mathrm{ps}^{\langle \mathcal{S} \rangle}]\!]\mathrm{F}'\Phi$ *is well-defined if* $\mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i \in [n]} \vdash \mathrm{b} : \mathcal{S}$, $\mathcal{C}$ *and* $\forall Q.\overline{\mathcal{S}}^{n} \to \mathcal{S}$ *ground, and* $\mathrm{Valid}(\mathcal{C}, \Phi)$.

Proof. See Appendix B, page 116. □

We need another simple lemma before we can prove the well-definedness of the program translation function $\mathfrak{P}$.

Lemma 4.15 (Properties of helper judgment for functor arguments). *If* $\mathcal{C} \overset{funargs}{\vdash} \overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]} \triangleright \forall P.\overline{\mathcal{S}}^n$, *then* $\mathrm{FV}^\alpha(\forall P.\overline{\mathcal{S}}^n) \subseteq \mathrm{FV}^\alpha(\mathcal{C})$ *and for all* $i \in [n]$ *exist* P_i *with* $P = \cup_{i\in[n]} P_i$ *such that* $\mathrm{Solv}(\mathcal{S}_i, P_i)$.

Proof. We prove the first proposition by induction on n using Lemma 4.6. The second proposition follows directly from Lemma 4.3. □

Now we can state and prove the main result of this section.

Theorem 4.16 (Well-definedness of program translation). $\mathfrak{P}[\![\mathrm{prog}]\!]\Phi$ *is well-defined if* $\mathcal{C} \vdash \mathrm{prog}$, $\mathcal{C}$ *ground, and* $\mathrm{Valid}(\mathcal{C}, \Phi)$.

Proof. By structural induction on prog. See Appendix B, page 116. □

Corollary 4.17. $\mathfrak{P}[\![\mathrm{prog}]\!]\emptyset$ *is well-defined if* $\emptyset \vdash \mathrm{prog}$.

Proof. Follows directly from Theorem 4.16. □

4.3.2. Type correctness

This section proves that the result of the translation is a type correct Tiny-HS$^+$ program provided the source program is a type correct Tiny-ML program. In the following, all usages of the translation functions and all usages of the pick operation are well-defined. The well-definedness property is easy to verify, so we do not mention it explicitly.

We first have to relate the environments $\hat{\Theta}$ and $\hat{\Gamma}$ used in the typing judgments for Tiny-HS$^+$ to the context $\mathcal{C}$ used in the typing judgments for Tiny-ML.

Definition 4.18 (Equivalence of Tiny-HS$^+$ environments and Tiny-ML contexts). $\hat{\Theta}, \hat{\Gamma}$ are equivalent to $\mathcal{C}$ modulo Φ, written $\hat{\Theta}; \hat{\Gamma} \equiv^\Phi \mathcal{C}$, if the following conditions hold:

1. $\mathcal{C}$ ground
2. $\mathrm{Valid}(\mathcal{C}, \Phi)$
3. $\mathrm{FV}^{\mathsf{z}}(\Phi) \cap \{\mathsf{z}^{\mathrm{c}} \mid \mathrm{c} \in \mathrm{CoreId}\} = \emptyset$
4. For $\mathrm{x} \in \mathrm{Dom}(\mathcal{C})$: $\hat{\Theta}; \hat{\Gamma} \vdash \Phi(\mathrm{x}) : \mathfrak{T}_v[\![\mathcal{C}(\mathrm{x})]\!]\Phi$
5. For $\mathrm{X} \in \mathrm{Dom}(\mathcal{C})$, $\mathrm{y} \in \mathrm{Dom}(\mathcal{C}(\mathrm{X}))$: $\hat{\Theta}; \hat{\Gamma} \vdash \Phi(\mathrm{X}, \mathrm{y}) : \mathfrak{T}_v[\![\mathcal{C}(\mathrm{X})(\mathrm{y})]\!]\Phi$
6. For $\mathrm{c} \in \mathrm{Dom}(\mathcal{C})$: $\hat{\Gamma}(\mathsf{z}^{\mathrm{c}}) = \mathfrak{T}_v[\![\mathcal{C}(\mathrm{c})]\!]\Phi$
7. For $\mathrm{F} \in \mathrm{Dom}(\mathcal{C})$ with $\mathcal{C}(\mathrm{F}) = \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S}$:

7.1. For all $i \in [n], \mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i)$: $\mathsf{S}^{\mathrm{F},i,\mathrm{t}}$ is an associated type synonym of type class $\mathsf{C}^{\mathrm{F},arg}$

7.2. For all $i \in [n]$, $\mathrm{x} \in \mathrm{Dom}(\mathcal{S}_i)$: $\hat{\Gamma}(\mathsf{z}^{\mathrm{F},i,\mathrm{x}}) = \forall \mathsf{A} \dot\cup \{\mathsf{a}\}.\mathsf{C}^{\mathrm{F},arg}\ \mathsf{a} \Rightarrow \mathsf{a} \to \tau$, where $\forall \mathsf{A}.\tau = \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi'$, $\Phi' := \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}}, \alpha)\}$

7.3. $(\forall\{a\}.\mathsf{C}^{\mathrm{F},arg}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}) \in \hat{\Theta}$ where T^{F} is a user-defined data constructor of kind 0

7.4. For all $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$: $\mathsf{S}^{\mathrm{F},t}$ is an associated type synonym of type class C^{F} and $\hat{\Theta} \Vdash [\tau/\mathsf{a}](\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi'')$ for all τ, where $\Phi'' := \Phi' \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\}$

7.5. For all $\mathrm{x} \in \mathrm{Dom}(\mathcal{S})$: $\hat{\Gamma}(\mathsf{z}^{\mathrm{F},\mathrm{x}}) = \forall \mathsf{A} \dot\cup \{\mathsf{a}, \mathsf{b}\}.\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{b} \to \mathsf{a} \to \tau$, where $\forall \mathsf{A}.\tau = \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi'''$, $\Phi''' := \Phi' \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{b}\ \mathsf{a} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\}$

7.6. $\mathrm{Sup}(\hat{\Theta}, \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}) = \emptyset$, $\mathrm{Sup}(\hat{\Theta}, \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}) = \{\mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}\}$

We extend to well-formedness predicate for Tiny-HS$^+$ types (see Figure 3.5 on page 43) to whole occurrence environments. This enables us to state and prove a lemma about the well-formedness of translated types.

Definition 4.19 (Well-formedness of occurrence environments). Φ is said to be well-formed under $\hat{\Theta}$, written $\hat{\Theta} \vdash \Phi$, if $\hat{\Theta} \vdash \Phi(\alpha)$ for all $\alpha \in \mathrm{Dom}(\Phi)$.

Lemma 4.20 (Well-formedness of translated types). *If* $\mathfrak{T}_u[\![u]\!]\Phi = \tau$ *and* $\hat{\Theta} \vdash \Phi$, *then* $\hat{\Theta} \vdash \tau$.

Proof. Straightforward induction on the structure of u. Note that we postulated in Section 3.3.4 that every Tiny-ML type constructor T^κ has a builtin Tiny-HS$^+$ counterpart T^κ. □

The next lemma proves that it does not matter whether we first apply a substitution to a Tiny-ML type and then translate the resulting type, or whether we translate the type first and then apply a corresponding substitution on the result.

Lemma 4.21 (Type translation and substitutions). *Suppose* ϕ *is a substitution from SimTypVar to SimTyp and* Φ *is an occurrence environment with* $\mathrm{FV}^\alpha(\phi) \subseteq \mathrm{Dom}(\Phi)$, $\mathrm{FV}^{\mathsf{a}}(\Phi) \cap \{\mathsf{a}'^a \mid `a \in \mathrm{Dom}(\phi)\} = \emptyset$. *Define* $\psi := \{\mathsf{a}'^a \mapsto \mathfrak{T}_u[\![\phi(`\mathrm{a})]\!]\Phi \mid `a \in \mathrm{Dom}(\phi)\}$. *Then we have* $\mathfrak{T}_u[\![\phi(u)]\!]\Phi = \psi(\mathfrak{T}_u[\![u]\!]\Phi)$ *for all* u *with* $\mathrm{FV}^\alpha(u) \subseteq \mathrm{Dom}(\Phi)$.

Proof. Straightforward induction on the structure of u. □

The following lemma is a simple weakening lemma.

Lemma 4.22 (Weakening).

- $\hat{\Theta} \Vdash \pi$ *and* $\hat{\Theta} \subseteq \hat{\Theta}'$ *imply* $\hat{\Theta}' \Vdash \tau$.

- $\hat{\Theta} \vdash \tau$ *and* $\hat{\Theta} \subseteq \hat{\Theta}'$ *imply* $\hat{\Theta}' \vdash \tau$.
- $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \sigma$, $\hat{\Theta} \subseteq \hat{\Theta}'$, *and* $\hat{\Gamma} \subseteq \hat{\Gamma}'$ *imply* $\hat{\Theta}'; \hat{\Gamma}' \vdash \mathsf{w} : \sigma$.

Proof. All claims are proved by rule induction. For the case $(\forall I)^+$ of the last claim, we safely assume that $(\mathsf{A} \cup \{\mathsf{a}\}) \cap (\mathrm{FV}^{\mathsf{a}}(\hat{\Theta}') \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Gamma}')) = \emptyset$ where $\sigma = \forall \mathsf{A} \cup \{\mathsf{a}\}.\rho$. □

The next lemma states that if we can assign some type scheme to an expression, then we can also assign instances of the type scheme to the expression.

Lemma 4.23 (Instantiation of type schemes). *If* $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \forall \mathsf{A}.\rho$ *and* ψ *is a substitution with* $\mathrm{Dom}(\psi) = \mathsf{A}$ *such that* $\hat{\Theta} \vdash \psi(\mathsf{a})$ *for all* $\mathsf{a} \in \mathsf{A}$, *then* $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \psi(\rho)$.

Proof. We can safely assume that $\mathsf{A} \cap \mathrm{FV}^{\mathsf{a}}(\psi) = \emptyset$. The proof is now by induction on $|\mathsf{A}|$ using rule $(\forall E)^+$. □

Now we can prove that type correct Tiny-ML expressions are translated into type correct Tiny-HS$^+$ expressions.

Lemma 4.24 (Type correctness of translated expressions). *If* $\mathfrak{E}[\![\mathrm{e}]\!]\Phi = \mathsf{w}$ *and* $\mathcal{C} \vdash \mathrm{e} : u$, *then* $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau$ *and* $\mathfrak{T}_u[\![u]\!]\Phi = \tau$, *provided the following assumptions hold:*

- $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi} \mathcal{C}$
- $\hat{\Theta} \vdash \Phi$
- $\mathrm{FV}^{\alpha}(u) \subseteq \mathrm{Dom}(\Phi)$

Proof. By induction on the structure of e. See Appendix B on page 118. □

Corollary 4.25 (Type correctness of translated, polymorphic expressions). *If we have* $\mathfrak{E}[\![\mathrm{e}]\!]\Phi = \mathsf{w}$ *and* $\mathcal{C} \vdash \mathrm{e} : v$, *then we have also* $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \sigma$ *and* $\mathfrak{T}_v[\![v]\!]\Phi = \sigma$, *provided* $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi} \mathcal{C}$ *and* $\hat{\Theta} \vdash \Phi$ *hold.*

Proof. Follows directly from Lemma 4.24, Lemma 4.6, rule (exp_{poly}), and rule $(\forall I)^+$. □

The next lemma states that the elements of a code environment Ω returned by the translation function $\mathfrak{S}_{\mathrm{b}}$ have the "expected properties".

Lemma 4.26 (Type correctness of translated structure bodies). *Suppose*

- $\mathcal{C} \vdash \mathrm{b} : \mathcal{S}$
- $\mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi = \Omega$
- $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi} \mathcal{C}$
- $\hat{\Theta} \vdash \Phi$

- $\hat{\Theta}; \hat{\Gamma} \vdash \Phi(\mathrm{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi$ *for all* $x \in \mathrm{Dom}(\mathcal{S})$

Then we have

- $\hat{\Theta} \vdash \Omega(\mathrm{t})$ *for all* $\mathrm{t} \in \mathrm{Dom}(\Omega)$
- $\Omega(\mathrm{t}) = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi$ *for all* $\mathrm{t} \in \mathrm{Dom}(\Omega)$
- $\hat{\Theta}; \hat{\Gamma} \vdash \Omega(\mathrm{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi$ *for all* $\mathrm{x} \in \mathrm{Dom}(\Omega)$

Proof. By structural induction on b. See Appendix B, page 120. □

The following three lemmata prove various propositions about the type translation function $\mathfrak{T}_u$.

Lemma 4.27 (Type translation, type equality, and substitutions). *Suppose* $\hat{\Theta} \Vdash \Phi(\alpha) = \mathfrak{T}_u[\![\varphi(\alpha)]\!]\Phi$ *for all* $\alpha \in \mathrm{Dom}(\varphi)$*, where* φ *is a substitution from TypVar to SimTyp such that* $\mathrm{Dom}(\varphi) \cup \mathrm{FV}^{\alpha}(\varphi) \subseteq \mathrm{Dom}(\Phi)$*. Then* $\hat{\Theta} \Vdash \mathfrak{T}_u[\![u]\!]\Phi = \mathfrak{T}_u[\![\varphi(u)]\!]\Phi$*, provided* $\mathrm{FV}^{\alpha}(u) \subseteq \mathrm{Dom}(\Phi)$.

Proof. Straightforward induction over the structure of u. □

Lemma 4.28 (Type translation and type equality). *If* $\mathrm{Dom}(\Phi) = \mathrm{Dom}(\Phi')$*,* $\hat{\Theta} \Vdash \Phi(\alpha) = \Phi'(\alpha)$ *for all* $\alpha \in \mathrm{Dom}(\Phi)$*, and* $\mathrm{FV}^{\alpha}(u) \subseteq \mathrm{Dom}(\Phi)$*, then* $\hat{\Theta} \Vdash \mathfrak{T}_u[\![u]\!]\Phi = \mathfrak{T}_u[\![u]\!]\Phi'$.

Proof. Straightforward structural induction over u. □

Lemma 4.29 (Type translation and free simple type variables). *Let* u *be a semantic simple type with* $\mathrm{FV}^{\alpha}(u) \subseteq \mathrm{Dom}(\Phi)$*. Then we have* $\mathrm{FV}^{\mathsf{a}}(\mathfrak{T}_u[\![u]\!]\Phi) \subseteq \mathrm{FV}^{\mathsf{a}}(\Phi) \cup \{\mathsf{a}^{`a} \mid `a \in \mathrm{FV}^{`a}(u)\}$*. Similarly, for a semantic simple type scheme* $v$ *with* $\mathrm{FV}^{\alpha}(v) \subseteq \mathrm{Dom}(\Phi)$ *we have* $\mathrm{FV}^{\mathsf{a}}(\mathfrak{T}_v[\![v]\!]\Phi) \subseteq \mathrm{FV}^{\mathsf{a}}(\Phi) \cup \{\mathsf{a}^{`a} \mid `a \in \mathrm{FV}^{`a}(v)\}$.

Proof. Straightforward structural inductions on u and v, respectively. □

The next two lemmata are important because they connect the enrichment relation of Tiny-ML with type assignments in Tiny-HS^{+}. They basically state that if some expression has type σ then it also has type σ', provided σ and σ' are translations of v and v', respectively, and v enriches v'.

Lemma 4.30 (Typing and value type enrichment). *Suppose* $v \succcurlyeq v'$ *and* $\mathrm{FV}^{\alpha}(v') \subseteq \mathrm{Dom}(\Phi)$*. If* $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \mathfrak{T}_v[\![v]\!]\Phi$ *and* $\hat{\Theta} \vdash \Phi$*, then* $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \mathfrak{T}_v[\![v']\!]\Phi$.

Proof. See Appendix B, page 121. □

Lemma 4.31 (Typing and structure enrichment). *Suppose* $\mathcal{S} \succcurlyeq \varphi(\mathcal{S}')$*,* $\mathrm{x} \in \mathrm{Dom}(\mathcal{S}')$*,* $\hat{\Theta}; \hat{\Gamma} \vdash \mathrm{w} : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi$*, and* $\hat{\Theta} \vdash \Phi$*. Then we have also* $\hat{\Theta}; \hat{\Gamma} \vdash \mathrm{w} : \mathfrak{T}_v[\![\mathcal{S}'(\mathrm{x})]\!]\Phi'$ *for all* $\Phi' = \Phi \dot{\cup} \{\alpha \mapsto \tau_\alpha \mid \alpha \in \mathrm{Dom}(\varphi), \hat{\Theta} \Vdash \tau_\alpha = \mathfrak{T}_u[\![\varphi(\alpha)]\!]\Phi\}$ *provided* $\mathrm{Dom}(\varphi) \cap \mathrm{Dom}(\Phi) = \emptyset$ *and* $\mathrm{FV}^{\alpha}(\varphi) \cup (\mathrm{FV}^{\alpha}(\mathcal{S}') \setminus \mathrm{Dom}(\varphi)) = \mathrm{Dom}(\Phi)$.

Proof. See Appendix B, page 122. □

The preceding two lemmata make it possible to prove that the translation of unsealed structure expressions yields the "desired result".

Lemma 4.32 (Type correctness of translated structure expressions). *Given*

- $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$
- $\mathfrak{S}[\![\mathrm{s}]\!](\Phi \dot\cup \Phi') = \langle \Omega, \overline{\mathsf{ddec}}, \overline{\mathsf{inst}}^m \rangle$
- $\hat\Theta; \hat\Gamma \equiv^{\Phi \dot\cup \Phi'} \mathcal{C}$
- $\hat\Theta \vdash \Phi$
- $\mathrm{Dom}(\mathcal{S}_\mathrm{x}) = \mathrm{Dom}(\Phi')$
- $\hat\Theta; \hat\Gamma \vdash \Phi'(\mathrm{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!](\Phi \dot\cup \Phi'')$ *for all* $\mathrm{x} \in \mathrm{Dom}(\mathcal{S})$ *and some* Φ'' *with* $\Phi'' = \{\alpha \mapsto \tau_\alpha \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha), \hat\Theta \Vdash \tau_\alpha = \Omega(\mathrm{t}), \mathrm{FV}^\mathsf{a}(\tau_\alpha) = \emptyset\}$
- $\mathrm{FV}^{'a}(\mathcal{C}) \cup \mathrm{FV}^\mathsf{a}(\Phi) = \emptyset$

Then $\vdash \mathsf{inst}_i : \hat\Theta_i$ *for all* $i \in [m]$, *and with* $\hat\Theta' := \hat\Theta \cup \bigcup_{i\in[m]} \hat\Theta_i$

- $\hat\Theta'; \hat\Gamma \vdash \mathsf{inst}_i$ *for all* $i \in [m]$
- $\hat\Theta' \vdash \Omega(\mathrm{t})$ *for all* $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$
- $\hat\Theta' \Vdash \Omega(\mathrm{t}) = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!](\Phi \dot\cup \Phi'')$ *for all* $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$
- $\hat\Theta'; \hat\Gamma \vdash \Omega(\mathrm{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!](\Phi \dot\cup \Phi'')$ *for all* $\mathrm{x} \in \mathrm{Dom}(\mathcal{S})$

Proof. The proof is by structural induction on s. The interesting case is the one for functor application. See Appendix B on page 122 for a detailed proof. □

We need two more definitions and a simple lemma before we can prove the type correctness of the translation for whole programs. The following definition introduces a convenient notation for the triples returned by functions $\mathfrak{X}$, $\mathfrak{F}$, and $\mathfrak{P}$.

Definition 4.33 (Program vector). A program vector pv is a three-element vector $\langle \overline{\mathsf{ddec}}, \overline{\mathsf{inst}}, \overline{\mathsf{cls}} \rangle$. We define the following operations on program vectors:

- $\overrightarrow{\mathsf{pv}}$ denotes the program $\overline{\mathsf{ddec}}\ \overline{\mathsf{inst}}\ \overline{\mathsf{cls}}$ obtained by concatenating the elements of the program vector pv.
- $\mathsf{pv} \oplus \mathsf{pv}'$ is defined as the concatenation of the program vectors pv and pv′; that is, $\langle \overline{\mathsf{ddec}}, \overline{\mathsf{inst}}, \overline{\mathsf{cls}} \rangle \oplus \langle \overline{\mathsf{ddec}'}, \overline{\mathsf{inst}'}, \overline{\mathsf{cls}'} \rangle := \langle \overline{\mathsf{ddec}}\ \overline{\mathsf{ddec}'}, \overline{\mathsf{inst}}\ \overline{\mathsf{inst}'}, \overline{\mathsf{cls}}\ \overline{\mathsf{cls}'} \rangle$.
- $\mathsf{pv} \overrightarrow{\oplus} \mathsf{pv}'$ is a short hand notation for $\overrightarrow{\mathsf{pv} \oplus \mathsf{pv}'}$.

The next definition connects a Tiny-ML context $\mathcal{C}$ with a Tiny-HS$^+$ program vector pv. Intuitively, pv provides $\mathcal{C}$ if pv is the translation of the Tiny-ML program whose type information is contained in $\mathcal{C}$.

Definition 4.34 (Tiny-ML context provider). A program vector pv provides $\mathcal{C}$ through Φ at $\hat{\Theta}$, $\hat{\Gamma}$if $\vdash \overrightarrow{\mathsf{pv}} : \hat{\Theta}; \hat{\Gamma}$, $\hat{\Theta} \vdash \Phi$, and $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi} \mathcal{C}$. If the specific $\hat{\Theta}$ and $\hat{\Gamma}$ do not matter, we just say that pv provides $\mathcal{C}$ through Φ.

We now extend the weakening lemma 4.22 to work for whole programs.

Lemma 4.35 (Weakening for whole programs). *Suppose*

- $\vdash \overline{\mathsf{cls}}^{n}\ \overline{\mathsf{inst}}^{m} : \hat{\Theta}; \hat{\Gamma}$
- $\vdash \mathsf{cls}_i : \hat{\Theta}_i; \hat{\Gamma}_i$ *for* $i \in \{n+1, \ldots, n+k\} =: N$
- $\vdash \mathsf{inst}_i : \hat{\Theta}'_i$ *for* $i \in \{m+1, \ldots, m+l\} =: M$
- $\hat{\Gamma}, \hat{\Gamma}_{n+1}, \ldots, \hat{\Gamma}_{n+k}$ *pairwise disjoint*

Define $\hat{\Theta}' := \hat{\Theta} \cup \bigcup_{i \in N} \hat{\Theta}_i \cup \bigcup_{i \in M} \hat{\Theta}'_i$ *and* $\hat{\Gamma}' := \hat{\Gamma} \cup \bigcup_{i \in N} \hat{\Gamma}_i$. *If now* $\hat{\Theta}' \vdash \mathsf{cls}_i$ *for* $i \in N$ *and* $\hat{\Theta}'; \hat{\Gamma}' \vdash \mathsf{inst}_i$ *for* $i \in M$, *then* $\vdash \overline{\mathsf{cls}}^{n+k}\ \overline{\mathsf{inst}}^{m+l} : \hat{\Theta}'; \hat{\Gamma}'$.

Proof. Follows from Lemma 4.22. □

The remainder of this section contains the type correctness proof for translated programs. Corollary 4.39 states the main result.

Lemma 4.36 (Type correctness of translated structure definitions with unsealed right-hand sides). *If*

- $\mathfrak{X}[\![\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S} \rangle}]\!]\Phi = \mathsf{pv}$
- $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$
- *there exists some* pv' *that provides* $\mathcal{C}$ *through* Φ
- $\mathrm{FV}^{`a}(\mathcal{C}) \cup \mathrm{FV}^{\mathsf{a}}(\Phi) = \emptyset$
- $\{(\mathrm{X}, \mathrm{y}) \mid \mathrm{y} \in \mathrm{ValId}\} \cap \Phi = \emptyset$

then

- $\mathsf{pv}' \oplus \mathsf{pv}$ *provides* $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$ *through* $\widetilde{\Phi} = \Phi \dot{\cup} \{\alpha \mapsto \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\} \dot{\cup} \{(\mathrm{X}, \mathrm{y}) \mapsto \mathsf{z}^{\mathrm{X.y}}\ (\bot :: \mathsf{T}^{\mathrm{X}}) \mid \mathrm{y} \in \mathrm{Dom}(\mathcal{S})\}$ *at* $\hat{\Theta}$,$\hat{\Gamma}$
- $\hat{\Theta} \Vdash \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}$
- $\mathsf{S}^{\mathrm{X.t}}$ *is an associated type synonym of class* C^{X} *for all* $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$
- $\hat{\Theta} \Vdash \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\widetilde{\Phi}$ *for all* $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$

Proof. See Appendix B, page 126. □

Lemma 4.37 (Type correctness of translated functor definitions with unsealed right-hand sides). *If*

- $\mathfrak{F}[\![\textbf{functor } \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}\rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) = \mathrm{ps}^{\langle \mathcal{S}\rangle}]\!]\mathrm{F}'\Phi = \mathsf{pv}$
- $\mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i\in[n]} \vdash \mathrm{ps} : \mathcal{S}$
- *there exists some* pv' *that provides* $\mathcal{C}$ *through* Φ
- $\mathrm{FV}^{'a}(\mathcal{C}) \cup \bigcup_{i\in[n]} \mathrm{FV}^{'a}(\mathcal{S}_i) \cup \mathrm{FV}^{\mathsf{a}}(\Phi) = \emptyset$

then

1. $\vdash \mathsf{pv}' \oplus \mathsf{pv} : \hat{\Theta}; \hat{\Gamma}$
2. $\hat{\Theta} \vdash \Phi$
3. *For all* $i \in [n], \mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i)$: $\mathsf{S}^{\mathrm{F}',i,\mathrm{t}}$ *is an associated type synonym of type class* $\mathsf{C}^{\mathrm{F}',arg}$
4. *For all* $i \in [n]$, $\mathrm{x} \in \mathrm{Dom}(\mathcal{S}_i)$: $\hat{\Gamma}(\mathsf{z}^{\mathrm{F}',i,\mathrm{x}}) = \forall \mathsf{A} \dot\cup \{\mathsf{a}\}.\mathsf{C}^{\mathrm{F}',arg}\ \mathsf{a} \Rightarrow \mathsf{a} \to \tau$, *where* $\forall \mathsf{A}.\tau = \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi'$, $\Phi' := \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F}',i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}}, \alpha)\}$
5. $(\forall\{\mathsf{a}\}.\mathsf{C}^{\mathrm{F}',arg}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}) \in \hat{\Theta}$ *where* T^{F} *is a user-defined data constructor of kind 0*
6. *For all* $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$: $\mathsf{S}^{\mathrm{F},t}$ *is an associated type synonym of type class* C^{F} *and* $(\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F},t}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi') \in \hat{\Theta}$
7. *For all* $\mathrm{x} \in \mathrm{Dom}(\mathcal{S})$: $\hat{\Gamma}(\mathsf{z}^{\mathrm{F},\mathrm{x}}) = \forall \mathsf{A} \dot\cup \{\mathsf{a},\mathsf{b}\}.\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{b} \to \mathsf{a} \to \tau$, *where* $\forall \mathsf{A}.\tau = \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi'$
8. $\mathrm{Sup}(\hat{\Theta}, \mathsf{C}^{\mathrm{F}',arg}\ \mathsf{a}) = \emptyset$, $\mathrm{Sup}(\hat{\Theta}, \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}) = \{\mathsf{C}^{\mathrm{F}',arg}\ \mathsf{a}\}$

Proof. See Appendix B, page 128 □

Theorem 4.38 (Type correctness of translated programs). *If*

- $\mathfrak{P}[\![\mathrm{prog}]\!]\Phi = \mathsf{pv}$
- $\mathcal{C} \vdash \mathrm{prog}$
- *there is some* pv' *that provides* $\mathcal{C}$ *through* Φ
- $\mathrm{FV}^{'a}(\mathcal{C}) \cup \mathrm{FV}^{\mathsf{a}}(\Phi) = \emptyset$
- $\{(\mathrm{X}, \mathrm{y}) \mid \mathrm{X} \in \mathrm{StrId} \setminus \mathrm{Dom}(\mathcal{C}), \mathrm{y} \in \mathrm{ValId}\} \cap \mathrm{Dom}(\Phi) = \emptyset$

then $\vdash \mathsf{pv}' \overrightarrow{\oplus}\ \mathsf{pv}$

Proof. The proof is by structural induction on prog. See Appendix B on page 131 for a detailed proof. □

Corollary 4.39. *If* $\mathfrak{P}[\![\mathrm{prog}]\!]\emptyset = \mathsf{pv}$ *and* $\emptyset \vdash \mathrm{prog}$, *then* $\vdash \overrightarrow{\mathsf{pv}}$.

Proof. Follows directly from Theorem 4.38. □

4.4. Restrictions on the source language Tiny-ML

We already noticed at the beginning of Section 2.3 that Tiny-ML does not support some features of Standard ML's module system, namely nested structures, parameterizable type components, arbitrary structure expressions as functor arguments and functor bodies, weak sealing, and data types. Moreover, Tiny-ML does not support higher-order functors [MT94] and applicative functors [Ler95], two widespread extensions to Standard ML. These features are not supported either because the translation to Tiny-HS$^+$ could not handle them, or because they would only complicate the translation without adding much to the comparison between ML modules and Haskell type classes. We now discuss if and how we could extend the translation to cope with these features.

Nested structures. Nested structures in ML would correspond to nested classes and nested instances, which are not supported by Haskell 98 or any extension. One possibility to translate nested structures would be to lift them to the top level. However, this would probably require a nontrivial transformation because nested structures may refer to components of the structure that contains them. I have not investigated this problem any further.

Parameterizable type components. In Standard ML, type components can be parameterized over simple type variables. This feature has been omitted from Tiny-ML to keep the translation simple. It is unproblematic to translate a parameterized type component to an associated type synonym that has more parameters than the class that declares it. (This feature has been omitted from Tiny-HS$^+$ but is supported by the system of Chakravarty et al. [CKP05].)

Arbitrary structure expressions as functor arguments. The translation of functor applications with arbitrary functor arguments is possible by binding the functor arguments to fresh structure variables and replacing the arguments with these variables.

Arbitrary structure expressions as functor bodies. Support for arbitrary structure expressions as functor bodies seems not to be possible. The problematic case is a functor body consisting of a functor application that uses an argument of the original functor; that is, something like **functor** $\mathrm{F(X) = G(X)}$. The problem is that

the translation to Tiny-HS$^+$ generates a new instance definition for the functor application G(X). However, the Tiny-HS$^+$ counterpart of the functor argument X is only available *inside* a class or instance definition, but nested classes and instances are not supported by Haskell 98 or any extension.

Weak sealing. Weak sealing only hides type and value components; it does not introduce new abstract types. Extending the translation with support for weak sealing is straightforward.

Data types. In Standard ML, signatures and structures can also contain data type components. Such data types correspond to associated data types, a Haskell extension suggested by Chakravarty et al. [CKPM05].

Higher-order functors. Higher-order functors [MT94] are functors that take other functors as parameters or return them as result. Higher-order functors pose a serious problem to the translation. A possible solution for functors taking other functors as arguments could be universal quantifiers in class contexts. For example, the class corresponding to a higher-order functor HF, which takes a functor F as argument, could be written **class** $(\forall c.\ \mathsf{S}\ c \Rightarrow \mathsf{F}\ a\ c) \Rightarrow \mathsf{HF}\ b\ a$ **where** ..., where S is the translation of F's argument signature. Such universal quantifiers in class contexts are mentioned by Peyton Jones et al. [PJM97, Section 5.2]. However, the authors state that universal quantification in constraints would mean a "substantial complication" of the type system and therefore reject the extension. However, Rossberg and Sulzmann show in [RS02, Section 4.3] that such universal quantifications can be encoded in the Haskell-like language Chameleon [SW05].

Applicative functors. Functors in Standard ML are generative; that is, they generate new abstract types every time they are invoked. Functors translated to Tiny-HS$^+$ behave generatively as well because a fresh type constructor and a fresh instance of the functor argument class is generated for each invocation. Applicative functors [Ler95] yield compatible abstract types when applied to compatible arguments. The translation to Tiny-HS$^+$ can be extended to handle applicative functors; only some extra bookkeeping is needed to avoid the generation of fresh type constructors and instances when a functor is applied to compatible arguments for the second time.

4.5. Implementation

A Haskell implementation of the translation from Tiny-ML to Tiny-HS$^+$ is available from `http://www.stefanwehr.de/diplom`. The Tiny-ML examples in Chapter 2 and the example in Figure 4.1(a) were checked against this implementation.

In fact, Appendix A shows the (slightly edited) Tiny-HS$^+$ code generated by the implementation for the ML code in Figure 4.1(a).

The type annotations for Tiny-ML are obtained by running the Moscow ML interpreter [RRK+03] on a pretty-printed and slightly modified version of the Tiny-ML abstract syntax tree. The interpreter outputs a textual form of the semantic objects, which can be parsed to get the type annotations.

The implementation of the translation itself is merely a Haskell version of the translation functions from Section 4.2. The result of the translation is an abstract syntax tree for PHRaC[3], which implements the target language Tiny-HS$^+$. PHRaC was developed by Gabriele Keller, Donald Stewart, and the author of this work.

4.6. Related work

Section 1.2 of the introduction already mentioned work related to the translation from ML modules to Haskell type classes. This sections contains a more detailed discussion of this work, and discusses other approaches to modular programming in Haskell.

Kahl and Scheffczyk propose in [KS01] *named instances* for Haskell type classes. Named instances allow the definition of more than one instance for the same type; the instances are then distinguished by their name. Such named instances are not used automatically in resolving overloading; however, the programmer can customize overloading resolution by supplying them explicitly. My translation from ML modules to Haskell type classes represents the name of a structure or functor as a data type. Kahl and Scheffczyk's extension would make it possible to avoid this detour because names are directly available in their language. However, associated type synonyms are not part of their extension. Kahl and Scheffczyk motivate and explain their extension in terms of OCaml's [Ler00b] module system; they do not consider any kind of translation from ML modules to Haskell type classes.

Shan [Sha04] presents a formal translation from a sophisticated ML module calculus [DCH03] into System F_ω [Gir72]. The source ML module calculus is a unified formalism that covers a large part of the design space of ML modules. In contrast, my source language Tiny-ML supports only basic features of the ML module system. The target language System F_ω of Shan's translation can be encoded in Haskell extended with higher-rank types [PS04b]; however, this encoding is orthogonal to the type class system. Shan implements abstract types using existential quantification in contrast to abstract associated type synonyms used in my work. Kiselyov builds on Shan's work and presents a Haskell example with type classes of an applicative translucent functor [Kis04]. However, he does not give a formal translation.

[3] PHRaC is available from `http://www.cse.unsw.edu.au/~chak/papers/CKP05.html`; however, the distribution from `http://www.stefanwehr.de/diplom` already contains PHRaC.

Jones suggests in [Jon96] an approach to modular programming that is different from the approach taken by ML: signatures cannot contain type components, but they may be parameterized over type variables; structures are then simply polymorphic records. Abstraction can be performed by using some sort of quantifier, which is left unspecified by Jones' theory. The approach of parameterized signatures is interesting because it avoids the use of a separate module language, which means that modules are first-class by default and functors are just ordinary functions.

Building on the ideas of Jones and other existing concepts, Shields and Peyton Jones [SP02] extend Haskell's core language in such a way that it can be used as a module language. They use existential quantification to encode abstract types and introduce a new construct to open existentials at the top level. Their resulting module system offers first-class modules, type abstraction, generative functors, type sharing, incremental compilation, and recursive and nested signatures and structures.

Chapter 5.

From classes to modules

The translation from Haskell type classes to ML modules, which we develop in this chapter, is very similar to other evidence translations [WB89, Jon94, HHPW96, Fax02] that make ad-hoc polymorphism introduced by type classes explicit; in our case, first-class structures are used as runtime evidence for constraints. We first discuss an example translation in Section 5.1, and then develop the formal translation from Tiny-HS to Tiny-ML$^+$ in Section 5.2. Section 5.3 proves that all well-formed and type correct Tiny-HS programs are translated into type correct Tiny-ML$^+$ programs. We discuss why full Haskell 98 type classes cannot be translated to ML modules in Section 5.4. An implementation of the translation is the topic of Section 5.5, and related work is discussed in Section 5.6.

5.1. Example translation

Starting with an example translation helps a lot to understand the general idea behind the translation from type classes to ML modules. The Tiny-HS code of the example is shown in Figure 5.1;[1] it is a slightly modified version of the example presented in Section 3.1. The expression at the end of the program is the main expression.

The translation to Tiny-ML$^+$ is shown in Figure 5.2. We first define abbreviations for the signatures representing the type classes Eq and Num.[2] A type class is translated into a signature with a single opaque type specification t, which corresponds to the type variable in the class head, and with value specifications corresponding to the methods of the class. Classes with superclasses have additional value specifications, one for every immediate superclass. The type of such a superclass value is that of a first-class structure of the superclass signature. For example, the value specification superEq in the signature Num represents Num's superclass Eq. Note that we use a type realization **where type** $\mathrm{t} = \mathrm{t}$ to make sure that the type specifications in the superclass and subclass signature are compatible.

All instance definitions in a Tiny-HS program are translated into a single group of recursive functors, which contains a functor definition for every instance definition. The arguments of such a functor correspond to the constraints in the context

[1] We bend Tiny-HS's syntax slightly to avoid unnecessary clutter.

[2] Signature abbreviations are supported in Standard ML but not in Tiny-ML. We can eliminate the abbreviations by replacing every signature identifier with its right-hand side.

Figure 5.1. Translating type classes to modules by hand: Tiny-HS code

```
class Eq a where
  eq :: a → a → Bool
class Eq a ⇒ Num a where
  plus :: a → a → a
instance Eq Int where
  eq = primIntEq
instance Num Int where
  plus = primIntAdd
instance Eq a ⇒ Eq [a] where
  eq l l′ = if null l ∧ null l′ then True else
            if null l ∨ null l′ then False else
            eq (head l) (head l′) ∧ eq (tail l) (tail l′)
let p = λx . λy . eq x y ∧ eq [plus x y] [plus y x] in
  if p 1 42 then 1 else 2
```

of the instance definition; we translate an instance with an empty context into a functor with an empty argument list rather than into a structure because Tiny-ML$^+$ does not support recursive structures. The functor definitions have to be mutually recursive because instances may be mutually recursive. Note that the group of recursive functors is not necessarily minimal; it would be straightforward but tedious to extend the translation with a dependency analysis so as to generate a set of minimal recursive functor groups.

The functors EqInt and NumInt are the translations of the instances Eq Int and Eq Num, respectively. We use the previously defined signature abbreviations together with a type realization as the mandatory type annotations for the functor bodies. The superEq value in NumInt's body is defined as a first-class structure that contains the result of invoking the EqInt functor.

The functor EqList is the translation of the instance definition for equality of lists. The functor argument X : Eq corresponds to the constrain Eq a in the instance head. The type occurrence X.t in the type realization **where type** t = list X.t and in the type definition **type** t = list X.t reflects the connection between the type variable a in the instance context and the type [a] in the instance head.

The definition of eq shows how the functor argument X is used on the value level. You see that the EqList functor is applied inside its own body so as to invoke eq recursively. We already saw this trick in Section 2.4.2.

So far, we did not see how values of qualified types are translated. Generally, a value of some qualified Tiny-HS type C $\tau \Rightarrow \rho$ is translated into a λ-abstraction that takes a *dictionary* as an extra argument. The dictionary provides runtime evidence for the constraint C τ; it contains the methods of class C at type τ. In our case, the dictionary is a first-class structure of type <S **where type** t = u>, where S is the

Figure 5.2. Translating type classes to modules by hand: Tiny-ML$^+$ code

```
signature Eq = sig type t val eq : t → t → bool end
signature Num = sig
                  type t
                  val plus      : t → t → t
                  val superEq : <Eq where type t = t>
                end
rec
  functor EqInt () : Eq where type t = int =
    struct type t = int val eq = primIntEq end
  functor NumInt () : Num where type t = int =
    struct
      type t          = int
      val plus        = primIntAdd
      val superEq = pack EqInt () as Eq where type t = int
    end
  functor EqList (X : Eq) : Eq where type t = list X.t =
    struct
      type t   = list X.t
      val   eq = λl . λl′ .
        if null l ∧ null l′ then true else
        if null l ∨ null l′ then false else
        X.eq (head l) (head l′)
         ∧ (open (pack EqList (X) as Eq where type t = list X.t)
              as Y : Eq where type t = list X.t
              in Y.eq (tail l) (tail l′))
    end
structure Main =
  struct val main = let p : ∀{'a} . <Num where type t = 'a> → 'a → 'a → bool
                        = λd . λx . λy .
                            open d as N : Num where type t = 'a in
                            open N.superEq as E : Eq where type t = 'a in
                            open (pack EqList (E) as Eq where type t = list 'a)
                              as L : Eq where type t = list 'a
                            in E.eq x y ∧ L.eq [N.plus x y] [N.plus y x]
                    in    if p (pack NumInt () as Num where type t = int) 1 42
                          then 1 else 2
  end
```

signature corresponding to C, and u is the translation of τ.

You might wonder why we do not use Standard ML records as dictionaries. In fact, records could be used as dictionaries for the example discussed in this section. But in general, method signatures may contain universally quantified type variables that are different from the type variable in the class head. We would need *polymorphic* records to represent dictionaries of such type classes adequately. However, record entries in Standard ML are monomorphic; hence, we use first-class structures as dictionaries, whose value components might be polymorphic.

The translation of the main expression **let** p = ... **in** ... exemplifies how a value of a qualified type is encoded in Tiny-ML$^+$. In Tiny-HS, the let-bound variable p has type $\forall\{\mathsf{a}\}.\ \mathsf{Num\ a} \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$. Consequently, the value p in Tiny-ML$^+$ has type $\forall\{\text{'a}\}.\ \langle\mathrm{Num}\ \textbf{where type}\ \mathrm{t} = \text{'a}\rangle \to \text{'a} \to \text{'a} \to \mathrm{bool}$. The type annotation for p is essential; it makes the type variable 'a accessible in the body of p. We bind the dictionary of type <Num **where type** t = 'a> to an extra dictionary parameter d; the **open** construct uses d to bring structure variables N, E, and L into scope, which give access to the overloaded values necessary to translate the body of p. Note that we use the type variable 'a, introduced by the type annotation for p, in the signatures required by **open**. The translation of the application p 1 42 constructs the additional dictionary parameter by packaging the result of applying the functor NumInt as a first-class structure.

The formal translation presented in the following section produces a result very similar to the manual translation shown here. Three things are slightly different in the output of the formal translation:

- Signature abbreviations are not used; instead, the whole signature is repeated at every point of use.
- The functors representing instance definitions have an additional argument, which represents type variables free in the instance head but not constrained by the instance context. An empty signature is used if there are no such type variables. In our case, none of the three instance definitions in Figure 5.1 has such type variables, so we omitted the empty arguments in Figure 5.2 for clarity.
- Dictionaries (i.e., first-class structures) are re-opened inside every subexpression that accesses an overloaded operation. In Figure 5.2, we reduced the number of **open** constructs to make the code more readable.

5.2. Formal translation

The formal translation from Tiny-HS to Tiny-ML$^+$ extends to typing judgments for Tiny-HS with an additional output parameter, which gives the result of the translation. Additionally, we introduce new judgments for translating Tiny-HS types. We first make some preparations in Section 5.2.1 before we define the translation judgments in Section 5.2.2.

Figure 5.3. Identifier manipulation functions

Function signature	*Function application*
$\mathsf{MethodId} \to \mathrm{ValId}$,	x^{m}
$\mathsf{ClassId} \to \mathrm{ValId}$,	x^{C}
$\mathsf{VarId} \to \mathrm{CoreId}$,	c^{z}
$\mathsf{TypVar} \to \mathrm{TypId}$,	t^{a}
$\mathsf{TypVar} \to \mathrm{SimTypVar}$,	$\mathrm{`a}^{\mathsf{a}}$
$\mathsf{TypVar} \to \mathit{SimTypVar}$,	$`a^{\mathsf{a}}$

5.2.1. Preparations

Figure 5.3 shows injective functions for converting Tiny-HS identifiers to Tiny-ML$^+$ identifiers, and defines an intuitive shorthand notation for function application. We require that the images of all functions are pairwise disjoint. Furthermore, we postulate the existence of a set of fresh core identifiers, $\mathrm{FreshCoreIds} \subseteq \mathrm{CoreId}$, such that $\mathrm{FreshCoreIds} \cap \{\mathrm{c}^{\mathsf{z}} \mid \mathsf{z} \in \mathsf{VarId}\} = \emptyset$.

We need to adjust the original definition of environments (Definition 3.5 on page 32) because constraint environments carry extra information for the translation; we also need two additional environments.

Definition 5.1 (Environments for the translation from Tiny-HS to Tiny-ML$^+$). A variable environment Γ maps term and method variables to type schemes, just as the variable environment $\hat{\Gamma}$ in the original type system does. A constraint environment Θ is similar to a constraint environment $\hat{\Theta}$ in the original Tiny-HS type system but provides access to additional information: Θ^s contains constraint schemes resulting from subclass definitions, just as $\hat{\Theta}^s$ does; Θ^i maps constraint schemes originating from instance definitions to the names of the functors the instances are translated to; Θ^l records for constraints added during the translation the expressions providing evidence for the constraints. A type environment Σ maps Tiny-HS type variables to Tiny-ML$^+$ simple types. Finally, a signature environment Δ maps classes to signature expressions that represent the classes in Tiny-ML$^+$. The environment Δ is necessary because we do not use signature abbreviations in the formal translation. The environments are defined as follows:

$$
\begin{array}{lcll}
\Gamma & \in & \mathsf{VarId} \cup \mathsf{MethodId} \xrightarrow{\text{fin}} \mathsf{TypSc} & \text{Variable environment} \\
\Theta & := & \begin{pmatrix} \Theta^s \in \mathrm{Fin}(\mathsf{ConstrSc}), \\ \Theta^i \in \mathsf{ConstrSc} \xrightarrow{\text{fin}} \mathrm{FunId}, \\ \Theta^l \in \mathsf{Constr} \xrightarrow{\text{fin}} \mathrm{Exp} \end{pmatrix} & \text{Constraint environment} \\
\Sigma & \in & \mathsf{TypVar} \xrightarrow{\text{fin}} \mathrm{SimTyp} & \text{Type environment} \\
\Delta & \in & \mathsf{ClassId} \xrightarrow{\text{fin}} \mathrm{SigExp} & \text{Signature environment}
\end{array}
$$

It is sometimes convenient to treat Θ as a finite map and not as a triple containing a set and two finite maps. Hence, we define the following notation:

Figure 5.4. Translation of types

Translation of type schemes $\boxed{\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}}$

$$\frac{\Delta; \Sigma \overrightarrow{\cup} \{\mathsf{a} \mapsto \text{`}\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\} \vdash \rho \rightsquigarrow \mathrm{u}}{\Delta; \Sigma \vdash \forall \mathsf{A}.\rho \rightsquigarrow \forall\{\text{`}\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}.\mathrm{u}} \; (\mathit{typtrans}_{\mathit{scheme}})^t$$

Translation of qualified types $\boxed{\Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}}$

$$\frac{\Sigma \vdash \tau \rightsquigarrow \mathrm{u} \qquad \Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}' \qquad \mathsf{C} \in \mathrm{Dom}(\Delta)}{\Delta; \Sigma \vdash \mathsf{C}\ \tau \Rightarrow \rho \rightsquigarrow <\Delta(\mathsf{C}) \textbf{ where type } \mathrm{t} = \mathrm{u}> \to \mathrm{u}'} \; (\mathit{typtrans}_{\mathit{qual}})^t$$

$$\frac{\Sigma \vdash \tau \rightsquigarrow \mathrm{u}}{\Delta; \Sigma \vdash \tau \rightsquigarrow \mathrm{u}} \; (\mathit{typtrans}_{\mathit{qual}'})^t$$

Translation of monotypes $\boxed{\Sigma \vdash \tau \rightsquigarrow \mathrm{u}}$

$$\frac{\Sigma \vdash \tau_i \rightsquigarrow \mathrm{u}_i \ (i \in [\kappa])}{\Sigma \vdash \mathsf{T}^{\kappa}\ \overline{\tau}^{\kappa} \rightsquigarrow \mathsf{T}^{\kappa}\ \overline{\mathrm{u}}^{\kappa}} \; (\mathit{typtrans}_{\mathit{tycon}})^t \qquad \frac{\Sigma(\mathsf{a}) = \mathrm{u}}{\Sigma \vdash \mathsf{a} \rightsquigarrow \mathrm{u}} \; (\mathit{typtrans}_{\mathit{tyvar}})^t$$

Definition 5.2 (Operations on constraint environments). The operations $\dot{\cup}$ and $\overrightarrow{\cup}$ for finite maps are defined component-wise for constraint environments:

$$\begin{aligned} \Theta \,\dot{\cup}\, \Theta' &:= (\Theta^s \cup \Theta^{s\prime}, \Theta^i \,\dot{\cup}\, \Theta^{i\prime}, \Theta^l \,\dot{\cup}\, \Theta^{l\prime}) \\ \Theta \,\overrightarrow{\cup}\, \Theta' &:= (\Theta^s \cup \Theta^{s\prime}, \Theta^i \,\overrightarrow{\cup}\, \Theta^{i\prime}, \Theta^l \,\overrightarrow{\cup}\, \Theta^{l\prime}) \end{aligned}$$

5.2.2. The translation

The translation of Tiny-HS types into Tiny-ML$^+$ types is shown in Figure 5.4. The judgments $\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$ and $\Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}$ need the signature environment Δ because rule $(\mathit{typtrans}_{\mathit{qual}})^t$ translates a constraint $\mathsf{C}\ \tau$ into an appropriate package type. $\Delta(\mathsf{C})$ is the signature representing the class C in Tiny-ML$^+$.

The entailment judgment $\Delta; \Sigma; \Theta \Vdash \pi \rightsquigarrow \mathrm{e}$ is shown in Figure 5.5; the expression e is a dictionary that provides evidence for the constraint π. Rule $(\mathit{elem}_{\mathit{entail}})^t$ is trivial because the evidence is already contained in the local part of the constraint environment Θ.

Rule $(\mathit{inst}_{\mathit{entail}})^t$ handles the case where some instance definition provides evidence for π. We convert the dictionaries e_i for the constraints in the instance context into structure variables X_i by using the **open** construct. Then we apply the functor F, which is the translation of the instance, and package the result as a first-class structure. The extra argument for F represents the type variables free in the instance head but not constrained by the instance context.

Rule $(\mathit{super}_{\mathit{entail}})^t$ derives evidence for a superclass from a subclass constraint.

Figure 5.5. Entailment with translation

$$\boxed{\Delta; \Sigma; \Theta \Vdash \pi \rightsquigarrow \mathrm{e}}$$

$$\frac{\Theta^l(\pi) = \mathrm{e}}{\Delta; \Sigma; \Theta \Vdash \pi \rightsquigarrow \mathrm{e}}\ (\mathit{elem}_{\mathit{entail}})^t$$

$$\frac{\begin{array}{c}\mathsf{F} = \Theta^i(\forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[r]} \Rightarrow \mathsf{C}\ \tau') \qquad \tau = \psi(\tau') \qquad \Delta; \Sigma; \Theta \Vdash \mathsf{C}_i\ \psi(\mathsf{a}_i) \rightsquigarrow \mathrm{e}_i \\ \Sigma \vdash \tau \rightsquigarrow \mathrm{u} \qquad \Sigma \vdash \psi(\mathsf{a}_i) \rightsquigarrow \mathrm{u}_i \qquad \Sigma \vdash \psi(\mathsf{b}) \rightsquigarrow \mathrm{u}_\mathsf{b}\ \ (\mathsf{b} \in \mathsf{B}) \\ \mathrm{Dom}(\psi) = \mathsf{A} \qquad \mathsf{B} = \mathrm{FV}^\mathsf{a}(\tau') \setminus \{\mathsf{a}_i \mid i \in [r]\} \\ \mathsf{C}, \mathsf{C}_i \in \mathrm{Dom}(\Delta) \qquad \overline{\mathrm{X}}^r \text{ pairwise distinct and fresh}\end{array}}{\Delta; \Sigma; \Theta \Vdash \mathsf{C}\ \tau \rightsquigarrow \begin{array}{l}\overline{\textbf{open}\ \mathrm{e}_i\ \textbf{as}\ (\mathrm{X}_i : \Delta(\mathsf{C}_i)\ \textbf{where type}\ \mathrm{t} = \mathrm{u}_i)\ \textbf{in}}^{i\in[r]} \\ \quad \textbf{pack}\ \mathrm{F}(\overline{\mathrm{X}}^r, \textbf{struct}\ \overline{\textbf{type}\ \mathrm{t}^\mathsf{b} = \mathrm{u}_\mathsf{b}}^{\mathsf{b}\in\mathsf{B}}\ \textbf{end}) \\ \quad \textbf{as}\ (\Delta(\mathsf{C})\ \textbf{where type}\ \mathrm{t} = \mathrm{u})\end{array}}\ (\mathit{inst}_{\mathit{entail}})^t$$

$$\frac{(\forall \mathsf{a}.\mathsf{C}^{\mathsf{sub}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathsf{sup}}\ \mathsf{a}) \in \Theta^s \qquad \Delta; \Sigma; \Theta \Vdash \mathsf{C}^{\mathsf{sub}}\ \tau \rightsquigarrow \mathrm{e} \qquad \Sigma \vdash \tau \rightsquigarrow \mathrm{u}}{\Delta; \Sigma; \Theta \Vdash \mathsf{C}^{\mathsf{sup}}\ \tau \rightsquigarrow \textbf{open}\ \mathrm{e}\ \textbf{as}\ (\mathrm{X} : \Delta(\mathsf{C}^{\mathsf{sub}})\ \textbf{where type}\ \mathrm{t} = \mathrm{u})\ \textbf{in}\ \mathrm{X}.\mathrm{x}^{\mathsf{C}^{\mathsf{sup}}}}\ (\mathit{super}_{\mathit{entail}})^t$$

Hence, the resulting expression opens the first-class structure for the subclass constraint and selects the dictionary $\mathrm{x}^{\mathsf{C}^{\mathsf{sup}}}$ for the superclass from it.

The typing and translation judgment $\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \tau$ is defined in Figure 5.6; here, the Tiny-ML$^+$ expression e is the translation of the Tiny-HS expression w. A variable z is translated by rule $(\mathit{var})^t$ to the corresponding core variable c^z applied to dictionaries for the constraints in z's type.

Rule $(\mathit{method})^t$ handles method variables m. The type of a method is always of the form $\forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau'$, where C is the class declaring the method, because methods in Tiny-HS cannot have additional constraints. We use the entailment judgment to get the dictionary e for C at the right type. Then we open the first-class structure e and select the right method x^m from it.

Rules $(\rightarrow E)^t$ and $(\rightarrow I)^t$ are straightforward. Rule $(\mathit{let})^t$ is more interesting. We define an extended type environment Σ' that contains the quantified type variables A of σ, where σ is the generalization of τ'. It may seem strange that we then use Σ' to derive the very type τ'; but keep in mind that the translation system does not describe a concrete algorithm, so we have the freedom to "guess" the correct Σ'. We have to annotate the translated let-binding with the translation of σ because the subexpression e_1 may contain signature or structure expressions that use some of the simple type variables $\mathrm{'a}^\mathsf{a}$ for $\mathsf{a} \in \mathsf{A}$.

The definition of generalization in Figure 5.6 is the same as for Tiny-HS in Figure 3.2. However, there is a minor problem: The order of constraints in a qualified type $\overline{\pi} \Rightarrow \tau$ becomes now important because dictionary parameters are passed in the same order as the constraints $\overline{\pi}$ are written. But in the definition of gener-

Figure 5.6. Translation of expressions

$$\boxed{\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \tau}$$

$$\frac{\Gamma(\mathsf{z}) = \forall \mathsf{A}.\overline{\pi}^{n} \Rightarrow \tau' \qquad \psi = [\overline{\tau_{\mathsf{a}}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}}] \qquad \psi(\tau') = \tau \qquad \overline{\Delta; \Sigma; \Theta \Vdash \psi(\pi_i) \rightsquigarrow \mathrm{e}_i}^{i\in[n]}}{\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{z} \rightsquigarrow \mathrm{c}^{\mathsf{z}}\ \overline{\mathrm{e}}^{n} : \tau} \ (var)^t$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau' \qquad \psi = [\overline{\tau_{\mathsf{a}}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}}] \\ \psi(\tau') = \tau \qquad \Delta; \Sigma; \Theta \Vdash \mathsf{C}\ \tau_{\mathsf{b}} \rightsquigarrow \mathrm{e} \qquad \Sigma \vdash \tau_{\mathsf{b}} \rightsquigarrow \mathrm{u}_{\mathsf{b}} \qquad \mathsf{C} \in \mathrm{Dom}(\Delta)\end{array}}{\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{m} \rightsquigarrow \begin{array}{l}\textbf{open}\ \mathrm{e}\ \textbf{as}\ (\mathrm{X} : \Delta(\mathsf{C})\ \textbf{where type}\ \mathrm{t} = \mathrm{u}_{\mathsf{b}}) \\ \textbf{in}\ \mathrm{X.x}^{\mathsf{m}}\end{array} : \tau} \ (method)^t$$

$$\frac{\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w}_1 \rightsquigarrow \mathrm{e}_1 : \tau' \to \tau \quad \Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w}_2 \rightsquigarrow \mathrm{e}_2 : \tau'}{\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w}_1\ \mathsf{w}_2 \rightsquigarrow \mathrm{e}_1\ \mathrm{e}_2 : \tau} \ (\to E)^t$$

$$\frac{\Sigma; \Theta; \Gamma, \mathsf{z} \mapsto \tau' \vdash \mathsf{w} \rightsquigarrow \mathsf{e} : \tau}{\Delta; \Sigma; \Theta; \Gamma \vdash \lambda \mathsf{z}.\mathsf{w} \rightsquigarrow \lambda \mathrm{c}^{\mathsf{z}}.\mathrm{e} : \tau' \to \tau} \ (\to I)^t$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma \overrightarrow{\cup} \{\mathsf{a} \mapsto \text{`}\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\} \\ \Theta' = (\Theta^s, \Theta^i, \{\overline{\pi_i \mapsto \mathrm{c}_i}^{i\in[n]}\}); \qquad \Delta; \Sigma'; \Theta'; \Gamma \vdash \mathsf{w}_1 \rightsquigarrow \mathrm{e}_1 : \tau' \\ \mathrm{c}_i \in \mathrm{FreshCoreIds}, \qquad \mathrm{Gen}(\Theta', \Gamma, \tau') = \forall \mathsf{A}.\rho = \sigma \text{ unambiguous} \\ \Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v} \qquad \Delta; \Sigma; \Theta; \Gamma, \mathsf{z} \mapsto \sigma \vdash \mathsf{w}_2 \rightsquigarrow \mathrm{e}_2 : \tau\end{array}}{\Delta; \Sigma; \Theta; \Gamma \vdash \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2 \rightsquigarrow \textbf{let}\ \mathrm{c}^{\mathsf{z}} : \mathrm{v} = \overline{\lambda \mathrm{c}}^{n}.\mathrm{e}_1\ \textbf{in}\ \mathrm{e}_2 : \tau} \ (let)^t$$

Generalization

$$\mathrm{Gen}((\Theta^s, \Theta^i, \{\overline{\pi_i \mapsto \mathrm{e}_i}^{i\in[n]}\}), \Gamma, \tau) := \forall((\mathrm{FV}^{\mathsf{a}}(\overline{\pi}^{n}) \cup \mathrm{FV}^{\mathsf{a}}(\tau)) \setminus (\mathrm{FV}^{\mathsf{a}}(\Gamma) \cup \mathrm{FV}^{\mathsf{a}}(\Theta^s) \cup \mathrm{FV}^{\mathsf{a}}(\Theta^i))).\overline{\pi}^{n} \Rightarrow \tau$$

Figure 5.7. Translation of instance definitions

Constraint collection $\boxed{\vdash \mathsf{inst} \rightsquigarrow \Theta}$

$$\frac{\theta \text{ well-formed} \qquad \mathrm{F} \text{ fresh}}{\vdash \textbf{instance } \theta \textbf{ where } \overline{\mathsf{mval}}^{n} \rightsquigarrow (\emptyset, \{\theta \mapsto \mathrm{F}\}, \emptyset)} \; (\mathit{inst}_{\mathit{collect}})^t$$

Instance translation $\boxed{\Delta; \Theta; \Gamma \vdash \mathsf{inst} \rightsquigarrow \mathrm{rfun}}$

$$\frac{\begin{array}{c}
\Delta; \Sigma; \Theta'; \Gamma; \mathsf{A}; \tau \overset{\text{method}}{\vdash} \mathsf{m}_i = \mathsf{w}_i \rightsquigarrow \mathrm{e}_i : \mathrm{v}_i \;\; (i \in [n]) \\
\mathrm{sup} = \{\langle \mathsf{C}^{\mathsf{sup}}, \mathrm{e}^{\mathsf{sup}} \rangle \mid \mathsf{C}^{\mathsf{sup}} \in \mathrm{Sup}(\Theta, \mathsf{C}), \\
\qquad \Sigma; (\Theta'^{s}, \Theta'^{i} \setminus \{\theta \mapsto \mathrm{F}\}, \Theta'^{l}) \Vdash \mathsf{C}^{\mathsf{sup}}\ \tau \rightsquigarrow \mathrm{e}^{\mathsf{sup}}\} \\
\Theta' = \left(\Theta^{s}, \Theta^{i}, \{\mathsf{C}_i\ \mathsf{a}_i \mapsto \textbf{pack } \mathrm{X}_i \textbf{ as } \mathrm{S}_i \mid i \in [r]\}\right) \\
\Sigma = \{\mathsf{a}_i \mapsto \mathrm{X}_i.\mathrm{t} \mid i \in [r], \mathsf{a}_i \neq \mathsf{a}_j \text{ for all } j \in [i-1]\} \cup \{\mathsf{b} \mapsto \mathrm{Y}.\mathrm{t}^{\mathsf{b}} \mid \mathsf{b} \in \mathsf{B}\} \\
\Sigma \vdash \tau \rightsquigarrow \mathrm{u} \qquad \mathsf{B} = \mathrm{FV}^{\mathsf{a}}(\tau) \setminus \{\mathsf{a}_i \mid i \in [r]\} \\
\mathrm{S}_i = \begin{cases} \Delta(\mathsf{C}_i) & \text{if } \mathsf{a}_i \neq \mathsf{a}_j \text{ for all } j \in [i-1], \\ \Delta(\mathsf{C}_i) \textbf{ where type } \mathrm{t} = \mathrm{X}_j.\mathrm{t} & \text{if } \mathsf{a}_i = \mathsf{a}_j \text{ for some } j \in [i-1]. \end{cases} \\
\overline{\mathrm{X}}^{r}, \mathrm{Y} \text{ pairwise distinct and fresh} \\
\mathrm{F} = \Theta^{i}(\theta) \qquad \theta = \forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i \in [r]} \Rightarrow \mathsf{C}\ \tau
\end{array}}{\begin{array}{l}
\Delta; \Theta; \Gamma \vdash \textbf{instance } \forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i \in [r]} \Rightarrow \mathsf{C}\ \tau \textbf{ where } \overline{\mathsf{m}_i = \mathsf{w}_i}^{i \in [n]} \\
\quad \rightsquigarrow \textbf{functor } \mathrm{F}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i \in [r]}, \mathrm{Y} : \textbf{sig } \overline{\textbf{type } \mathrm{t}^{\mathsf{b}}}^{\mathsf{b} \in \mathsf{B}} \textbf{ end}) \\
\qquad : \Delta(\mathsf{C}) \textbf{ where type } \mathrm{t} = \mathrm{u} = \\
\qquad \textbf{struct} \\
\qquad\quad \textbf{type } \mathrm{t} = \mathrm{u} \\
\qquad\quad \overline{\textbf{val } \mathrm{x}^{\mathsf{m}_i} : \mathrm{v}_i = \mathrm{e}_i}^{i \in [n]} \\
\qquad\quad \overline{\textbf{val } \mathrm{x}^{\mathsf{C}^{\mathsf{sup}}} = \mathrm{e}^{\mathsf{sup}}}^{\langle \mathsf{C}^{\mathsf{sup}}, \mathrm{e}^{\mathsf{sup}} \rangle \in \mathrm{sup}} \\
\qquad \textbf{end}
\end{array}} \; (\mathit{inst}_{\mathit{check}})^t$$

Method translation $\boxed{\Delta; \Sigma; \Theta; \Gamma; \tau \overset{\text{method}}{\vdash} \mathsf{m} = \mathsf{w} \rightsquigarrow \mathrm{e} : \mathrm{v}}$

$$\frac{\begin{array}{c}
\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau' \qquad \mathsf{A} \cap (\mathrm{FV}^{\mathsf{a}}(\tau) \cup \mathrm{FV}^{\mathsf{a}}(\Theta) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma)) = \emptyset \\
\Delta; \Sigma'; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : [\tau/\mathsf{b}]\tau' \qquad \Sigma' \vdash [\tau/\mathsf{b}]\tau' \rightsquigarrow \mathrm{u}' \\
\mathrm{v} = \forall \{\text{`}\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}.\mathrm{u}' \qquad \Sigma' = \{\mathsf{a} \mapsto \text{`}\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}
\end{array}}{\Delta; \Sigma; \Theta; \Gamma; \tau \overset{\text{method}}{\vdash} \mathsf{m} = \mathsf{w} \rightsquigarrow \mathrm{e} : \mathrm{v}} \; (\mathit{inst}_{\mathit{check\text{-}method}})^t$$

Figure 5.8. Translation of class definitions and programs

Translation of class definitions $\boxed{\Delta \vdash \mathsf{cls} \rightsquigarrow \Delta; \Theta; \Gamma}$

$$\frac{\begin{array}{c}\mathsf{a} \notin \mathsf{A}_i \qquad \sigma_i := (\forall(\mathsf{A}_i \cup \{\mathsf{a}\}).\mathsf{C}\ \mathsf{a} \Rightarrow \tau_i)\ \text{unambiguous} \\ \mathrm{FV}^{\mathsf{a}}(\sigma_i) = \emptyset \qquad \emptyset; \{\mathsf{a} \mapsto \mathbf{t}\} \vdash \forall \mathsf{A}_i.\tau_i \rightsquigarrow \mathrm{v}_i \qquad (\text{for all } i \in [n]) \\ \mathrm{S} = \mathbf{sig} \\ \quad \mathbf{type}\ \mathbf{t} \\ \quad \overline{\mathbf{val}\ \mathrm{x}^{\mathsf{m}_i} : \mathrm{v}_i}^{i \in [n]} \\ \quad \overline{\mathbf{val}\ \mathrm{x}^{\mathsf{C}_i} : <\Delta(\mathsf{C}_i)\ \mathbf{where\ type}\ \mathrm{t} = \mathrm{t}>}^{i \in [r]} \\ \mathbf{end}\end{array}}{\begin{array}{l}\Delta \vdash \mathbf{class}\ \forall\{\mathsf{a}\}.\overline{\mathsf{C}_i\ \mathsf{a}}^{i \in [r]} \Rightarrow \mathsf{C}\ \mathsf{a}\ \mathbf{where}\ \overline{\mathsf{m}_i :: \forall \mathsf{A}_i.\tau_i}^{i \in [n]} \\ \quad \rightsquigarrow \{\mathsf{C} \mapsto \mathrm{S}\} \\ \quad ; (\{\forall\{\mathsf{a}\}.\mathsf{C}\ \mathsf{a} \Rightarrow \mathsf{C}_i\ \mathsf{a} \mid i \in [r]\}, \emptyset, \emptyset) \\ \quad ; \{\mathsf{m}_i \mapsto \sigma_i \mid i \in [n]\}\end{array}}\ (class)^t$$

Translation of programs $\boxed{\vdash \mathsf{pgm} \rightsquigarrow \mathrm{prog}}$

$$\frac{\begin{array}{c}\overline{\dot\cup_{j \in [i-1]} \Delta_j \vdash \mathsf{cls}_i \rightsquigarrow \Delta_i; \Theta_i; \Gamma_i}^{i \in [n]} \qquad \overline{\vdash \mathsf{inst}_i \rightsquigarrow \Theta'_i}^{i \in [m]} \\ \Delta = \dot\cup_{i \in [n]} \Delta_i \qquad \Theta = \dot\bigcup_{i \in [n]} \Theta_i \,\dot\cup\, \dot\bigcup_{i \in [m]} \Theta'_i \qquad \Gamma = \dot\bigcup_{i \in [n]} \Gamma_i \\ \overline{\Delta; \Theta; \Gamma \vdash \mathsf{inst}_i \rightsquigarrow \mathrm{rfun}_i}^{i \in [m]} \qquad \Delta; \emptyset; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \mathsf{Int}\end{array}}{\begin{array}{l}\vdash \overline{\mathsf{cls}}^{n}\ \overline{\mathsf{inst}}^{m} \rightsquigarrow \mathbf{rec}\ \overline{\mathrm{rfun}}^{m}; \\ \qquad \mathbf{structure\ Main} = \mathbf{struct\ val\ main} = \mathrm{e}\ \mathbf{end}\end{array}}\ (prog)^t$$

alization, we take constraints from an unordered set and use them in a qualified type! We can solve this problem by agreeing on some ordering relation on Constr, the set of all constraints; the ordering relation is then used implicitly to order the constraints before forming the qualified type.

Figure 5.7 shows the translation judgments for instance definitions. The judgment $\vdash \mathsf{inst} \rightsquigarrow \Theta$ just collects the constraint scheme of an instance definition and associates it with a fresh functor identifier. The judgment $\Delta; \Theta; \Gamma \vdash \mathsf{inst} \rightsquigarrow \mathrm{rfun}$ uses this functor identifier to generate a new recursive functor definition. The functor arguments X_i correspond to the constraints $\mathsf{C}_i\ \mathsf{a}_i$ in the instance context. The purpose of the extra parameter Y was mentioned several times before: It represents the type variables free in the instance head but not constrained by the instance context.

In the premise of rule $(inst_{check})^t$, we first translate the method implementations and derive dictionaries for the immediate superclasses. The results are used to define the value components $\mathrm{x}^{\mathsf{m}_i}$ and $\mathrm{x}^{\mathsf{C}^{\mathsf{sup}}}$ of the functor body. The explicit type annotations v_i for the value components $\mathrm{x}^{\mathsf{m}_i}$ are needed because e_i itself might contain some of the universally quantified simple type variables of v_i. The con-

straint environment Θ' is extended with the constraints from the instance context bound to the functor arguments X_i packaged as first-class structures. The type environment Σ maps type variables to type components of functor arguments: a type variable a constrained by the instance context is bound to $X_i.t$, where $C_i\ a_i$ is the first constraint with $a = a_i$; a type variable b free in the instance head but not constrained by the instance context is bound to $Y.t^b$. Note that the signature expressions S_i correctly model sharing introduced by constraints on the same type variable.

The translation judgment $\Delta \vdash \mathsf{cls} \rightsquigarrow \Delta; \Theta; \Gamma$ for class definitions is shown in Figure 5.8. Its main task is to construct the signature S as the translation of the class C. Figure 5.8 contains also the translation judgment $\vdash \mathsf{pgm} \rightsquigarrow \mathrm{prog}$ for whole programs. We first collect the environments resulting from class and instance definitions. Then we translate the instance definitions into recursive functors and the Tiny-HS main expression into a Tiny-ML$^+$ expression. Finally, we form a group of recursive functors and define a main structure.

5.3. Formal properties

This section proves formal properties of the translation from Tiny-HS to Tiny-ML$^+$. We first prove in Section 5.3.1 that the translation of well-formed programs is sound and complete with respect to the type system of Tiny-HS defined in Section 3.2.2. Then we show in Section 5.3.2 that the translation of a well-formed and type correct Tiny-HS program yields a type correct Tiny-ML$^+$ program. Taken together, this proves that every type correct and well-formed Tiny-HS program translates into a type correct Tiny-ML$^+$ program. It does not prove that the translation is sound because we do not show that executing a Tiny-HS program and its translation yields the same result. However, knowing that the translation preserves type correctness is a strong indication that the translation is indeed sound. You may skip this section if you are not interested in the formal details because they are not important for understanding the rest of the thesis.

5.3.1. Soundness and completeness with respect to Tiny-HS' type system

It is obvious that a program that is type correct with respect to the type-directed translation defined in Section 5.2 is also type correct with respect to Tiny-HS' type system from Section 3.2.2 *(soundness)*. Therefore, we do not need to state and prove soundness explicitly in this section.

However, it is not so clear that every program that is well-formed and type correct according to the system in Section 3.2.2 is also type correct according to the translation in Section 5.2 *(completeness)*. In particular, the type translation $\Sigma \vdash \tau \rightsquigarrow u$ may rule out certain programs because we can only construct a derivation if $\mathrm{FV}^a(\tau) \subseteq \mathrm{Dom}(\Sigma)$ holds.

We now prove completeness of the translation with respect to the original Tiny-HS type system, provided the program under translation is well-formed. The main result of the section is found in Theorem 5.12. We begin with some basic definitions.

Definition 5.3 (Well-formed constraint environments). A constraint environment Θ is called well-formed if all $\theta \in \mathrm{Dom}(\Theta^i)$ are well-formed.

Definition 5.4 (Unambiguous variable environments). A variable environment Γ is called unambiguous if all $\sigma \in \mathrm{Img}(\Gamma)$ are unambiguous.

Definition 5.5 (Comparing Θ with $\hat{\Theta}$). We write $\Theta \subseteq \hat{\Theta}$ for $\Theta^s \subseteq \hat{\Theta}^s$, $\mathrm{Dom}(\Theta^i) \subseteq \hat{\Theta}^i$, and $\mathrm{Dom}(\Theta^l) \subseteq \hat{\Theta}^l$. $\hat{\Theta} \subseteq \Theta$ is defined analogously. $\Theta = \hat{\Theta}$ stands for $\Theta \subseteq \hat{\Theta}$ and $\hat{\Theta} \subseteq \Theta$.

Definition 5.6 (Collecting class identifiers). The function $\mathrm{CS} : \mathsf{Typ} \cup \mathsf{QTyp} \cup \mathsf{TypSc} \cup \mathsf{Constr} \cup \mathsf{ConstrSc} \to \mathrm{Fin}(\mathsf{ClassId})$ collects all class identifiers of a type or constraint. It is defined in the obvious way.

The following lemma formulates conditions that ensure that a Tiny-HS type can be translated into a Tiny-ML$^+$ type.

Lemma 5.7 (Type translation).

- *If* $\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$, *then there is some* u *such that* $\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$.
- *If* $\mathrm{FV}^{\mathsf{a}}(\rho) \subseteq \mathrm{Dom}(\Sigma)$ *and* $\mathrm{CS}(\rho) \subseteq \mathrm{Dom}(\Delta)$, *then there is some* u *such that* $\Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}$.
- *If* $\mathrm{FV}^{\mathsf{a}}(\sigma) \subseteq \mathrm{Dom}(\Sigma)$ *and* $\mathrm{CS}(\sigma) \subseteq \mathrm{Dom}(\Delta)$, *then there is some* v *such that* $\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$.

Proof. Simple induction on the structure of τ, ρ and τ. □

The next lemma proves completeness of the entailment relation defined in Figure 5.5 on page 87 with respect to the relation defined in Figure 3.2 on page 33.

Lemma 5.8 (Completeness of entailment). *If* $\hat{\Theta} \Vdash \pi$, *then there exists some* e *such that* $\Delta; \Sigma; \Theta \Vdash \pi \rightsquigarrow \mathrm{e}$, *provided*

- $\hat{\Theta} \subseteq \Theta$, Θ *well-formed*
- $\mathrm{FV}^{\mathsf{a}}(\pi) \subseteq \mathrm{Dom}(\Sigma)$
- $\mathrm{CS}(\pi) \cup \mathrm{CS}(\Theta) \subseteq \mathrm{Dom}(\Delta)$

Proof. By induction on the derivation of $\hat{\Theta} \Vdash \pi$. See Appendix B, page 137. □

We need two more lemmata to prove completeness of the expression-translation judgment in Figure 5.6 on page 88. The first lemma is a standard substitution lemma. The second lemma is a strengthening lemma, which enables us to remove certain constraints from a constraint environment without making an entailment or expression-translation derivation invalid.

Lemma 5.9 (Substitution lemma). *If* $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau$ *and* $\psi \in \mathsf{TypVar} \to \mathsf{Typ}$ *is a substitution, then* $\psi(\hat{\Theta}); \psi(\hat{\Gamma}) \vdash \mathsf{w} : \psi(\tau)$.

Proof. See [Jon94, p. 24,133]. Note that unambiguity of type schemes is preserved under substitution (the additional premise "σ unambiguous" in rule (*let*) is the only significant difference between Jones' syntax directed system and ours). □

Lemma 5.10 (Constraint strengthening). *If* $\hat{\Theta} \Vdash \pi$ *and* $\mathsf{C} \notin (\mathrm{CS}(\pi) \cup \mathrm{CS}(\hat{\Theta}^s) \cup \mathrm{CS}(\hat{\Theta}^i))$, *then* $(\hat{\Theta}^s, \hat{\Theta}^i, \hat{\Theta}^l \setminus \{\mathsf{C}\ \tau'\}) \Vdash \pi$ *for any* τ'.

If $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau$ *and* $\mathsf{C} \notin (\mathrm{CS}(\hat{\Gamma}) \cup \mathrm{CS}(\hat{\Theta}^s) \cup \mathrm{CS}(\hat{\Theta}^i))$, *then* $(\hat{\Theta}^s, \hat{\Theta}^i, \hat{\Theta}^l \setminus \{\mathsf{C}\ \tau'\}); \hat{\Gamma} \vdash \mathsf{w} : \tau$ *for any* τ'.

Proof. The first part is proved by rule induction, the second part by structural induction on w. □

We now prove completeness of the expression-translation judgment defined in Figure 5.6 on page 88 with respect to the typing judgment in Figure 3.2 on page 33.

Lemma 5.11 (Completeness of typing). *If* $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \tau$, *then there exists some* e *such that* $\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \tau$, *provided*

- $\hat{\Theta} = \Theta$, Θ *well-formed*
- $\hat{\Gamma} = \Gamma$, Γ *unambiguous*
- $\mathrm{FV}^{\mathsf{a}}(\tau) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma) \cup \mathrm{FV}^{\mathsf{a}}(\Theta) \subseteq \mathrm{Dom}(\Sigma)$
- $\mathrm{CS}(\Theta) \cup \mathrm{CS}(\Gamma) \subseteq \mathrm{Dom}(\Delta)$

Proof. By structural induction on w. See Appendix B, page 138. □

Finally, we prove completeness of the program translation judgment defined in Figure 5.8 on page 90 with respect to the original Tiny-HS judgment defined in Figure 3.3 on page 35.

Theorem 5.12 (Soundness and completeness of program judgment). *Suppose* pgm *is well-formed according to Definition 3.8. Then* $\vdash \mathsf{pgm} \rightsquigarrow \mathrm{prog}$ *implies* $\vdash \mathsf{pgm}$.

Proof. See Appendix B, page 140. □

5.3.2. Type correctness

The purpose of this section is to prove that the translation from Tiny-HS to Tiny-ML$^+$ produces only type correct programs provided that the source program is well-formed and type correct.

We first define a notation for translating Tiny-HS types into semantic Tiny-ML$^+$ types. Such a notation is useful because the typing judgments for Tiny-ML$^+$ (see Figure 2.9 and Figure 2.10) are formulated in terms of semantic objects. However, the type translation defined in Figure 5.4 translates Tiny-HS types into syntactic Tiny-ML$^+$ types.

Definition 5.13 (Semantic type translation). $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$ is an abbreviation for $\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$ and $\mathcal{C} \vdash \mathrm{u} \triangleright u$. We call the semantic object u the semantic translation of the source type τ. Similarly, $\Delta; \Sigma; \mathcal{C} \vdash \rho \rightsquigarrow \mathrm{u}$ means $\Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}$ and $\mathcal{C} \vdash \mathrm{u} \triangleright u$. Finally, $\Delta; \Sigma; \mathcal{C} \vdash \sigma \rightsquigarrow v$ is short for $\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$ and $\mathcal{C} \vdash \mathrm{v} \triangleright v$.

We now prove uniqueness and existence of semantic type translations.

Lemma 5.14 (Uniqueness of semantic type translations).

- *If* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$ *and* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u'$, *then* $u = u'$.
- *If* $\Delta; \Sigma; \mathcal{C} \vdash \rho \rightsquigarrow u$ *and* $\Delta; \Sigma; \mathcal{C} \vdash \rho \rightsquigarrow u'$, *then* $u = u'$.
- *If* $\Delta; \Sigma; \mathcal{C} \vdash \sigma \rightsquigarrow v$ *and* $\Delta; \Sigma; \mathcal{C} \vdash \sigma \rightsquigarrow v'$, *then* $v = v'$.

Proof. Simple rule inductions. □

Lemma 5.15 (Existence of semantic type translations).

- *If* $\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$ *and for all* $\mathsf{a} \in \mathrm{FV}^{\mathsf{a}}(\tau)$ *there exists some* u' *such that* $\mathcal{C} \vdash \Sigma(\mathsf{a}) \triangleright u'$, *then there is some* u *with* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$.
- *If* $\mathrm{CS}(\rho) \subseteq \mathrm{Dom}(\Delta)$, $\mathrm{FV}^{\mathsf{a}}(\rho) \subseteq \mathrm{Dom}(\Sigma)$, *and for all* $\mathsf{a} \in \mathrm{FV}^{\mathsf{a}}(\rho)$ *there exists some* u' *such that* $\mathcal{C} \vdash \Sigma(\mathsf{a}) \triangleright u'$, *then there is some* u *with* $\Delta; \Sigma; \mathcal{C} \vdash \rho \rightsquigarrow \mathrm{u}$.
- *If* $\mathrm{CS}(\sigma) \subseteq \mathrm{Dom}(\Delta)$, $\mathrm{FV}^{\mathsf{a}}(\sigma) \subseteq \mathrm{Dom}(\Sigma)$, *and for all* $\mathsf{a} \in \mathrm{FV}^{\mathsf{a}}(\sigma)$ *there exists some* u' *such that* $\mathcal{C} \vdash \Sigma(\mathsf{a}) \triangleright u'$, *then there is some* v *with* $\Delta; \Sigma; \mathcal{C} \vdash \sigma \rightsquigarrow v$.

Proof. Simple rule inductions. □

It is often more convenient to translate a Tiny-HS type directly into a semantic Tiny-ML$^+$ type. Hence, we introduce a new translation judgment and prove that it is equivalent to a semantic type translation.

Definition 5.16 (Direct semantic type translation). Let $\mathcal{T} \in \mathsf{TypVar} \xrightarrow{\text{fin}} \mathit{SimTyp}$. We call $\mathcal{T} \vdash \tau \rightsquigarrow u$ the direct semantic translation of type τ into the semantic object u, and define the translation as follows:

$$\frac{\mathcal{T} \vdash \tau_i \rightsquigarrow u_i \quad (i \in [\kappa])}{\mathcal{T} \vdash \mathsf{T}^{\kappa}\, \overline{\tau}^{\kappa} \rightsquigarrow T^{\kappa}\, \overline{u}^{\kappa}} \qquad \frac{\mathcal{T}(\mathsf{a}) = u}{\mathcal{T} \vdash \mathsf{a} \rightsquigarrow u}$$

Lemma 5.17 (Equivalence of semantic type translations). *If* $\mathcal{T} \vdash \tau \rightsquigarrow u$ *and* $\Sigma; \mathcal{C} \vdash \mathsf{a} \rightsquigarrow \mathcal{T}(\mathsf{a})$ *for all* $\mathsf{a} \in \mathrm{Dom}(\mathcal{T})$*, then* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$*. If* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$*, then* $\mathcal{T} \vdash \tau \rightsquigarrow u$ *provided* $\mathcal{T} \subseteq \{\mathsf{a} \mapsto u_{\mathsf{a}} \mid \mathsf{a} \in \mathrm{FV}^{\mathsf{a}}(\tau), \Sigma; \mathcal{C} \vdash \mathsf{a} \rightsquigarrow u_{\mathsf{a}}\}$.

Proof. Simple induction on the structure of τ. □

We now prove existence and uniqueness of direct semantic type translations.

Corollary 5.18 (Existence and uniqueness of direct semantic type translations). *If* $\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\mathcal{T})$*, then there is some* u *such that* $\mathcal{T} \vdash \tau \rightsquigarrow u$*. If* $\mathcal{T} \vdash \tau \rightsquigarrow u$ *and* $\mathcal{T} \vdash \tau \rightsquigarrow u'$*, then* $u = u'$.

Proof. Follows from Lemmata 5.14, 5.15, and 5.17. □

The following lemma states that direct semantic type translations are preserved under substitutions.

Lemma 5.19 (Substitution lemma for direct semantic type translations). *If* $\mathcal{T} \vdash \tau \rightsquigarrow u$ *and* φ *is some substitution from* TypVar *to* SimTyp*, then* $\varphi(\mathcal{T}) \vdash \tau \rightsquigarrow \varphi(u)$.

Proof. Straightforward rule induction. □

The following three rather technical lemmata state properties of direct semantic type translations.

Lemma 5.20. *Let* $\mathcal{T} \dot\cup \mathcal{T}' \vdash \tau \rightsquigarrow u$*,* $\mathcal{T}$ *bijective,* $\mathrm{Img}(\mathcal{T}) \subseteq \mathit{SimTypVar}$*, and* $\mathrm{FV}^{`a}(\mathcal{T}') \cap \mathrm{Img}(\mathcal{T}) = \emptyset$*. Let* ψ *be a substitution with* $\mathrm{Dom}(\psi) = \mathrm{Dom}(\mathcal{T})$*,* $\mathrm{FV}^{\mathsf{a}}(\psi) \subseteq \mathrm{Dom}(\mathcal{T}')$*. Then* $\mathcal{T}' \vdash \psi(\tau) \rightsquigarrow \phi(u)$*, where* $\phi := \{`a \mapsto u_{`a} \mid `a \in \mathrm{Img}(\mathcal{T}), \mathcal{T}' \vdash \psi(\mathcal{T}^{-1}(`a)) \rightsquigarrow u_{`a}\}$.

Proof. By induction on τ. See Appendix B, page 140. □

Lemma 5.21. *Let* $\mathcal{T} \dot\cup \mathcal{T}' \vdash \tau \rightsquigarrow u$*,* $\mathcal{T}$ *bijective,* $\mathrm{Img}(\mathcal{T}) \subseteq \mathit{TypVar}$*, and* $\mathrm{FV}^{\alpha}(\mathcal{T}') \cap \mathrm{Img}(\mathcal{T}) = \emptyset$*. Let* ψ *be a substitution with* $\mathrm{Dom}(\psi) = \mathrm{Dom}(\mathcal{T})$*,* $\mathrm{FV}^{\mathsf{a}}(\psi) \subseteq \mathrm{Dom}(\mathcal{T}')$*. Then* $\mathcal{T}' \vdash \psi(\tau) \rightsquigarrow \varphi(u)$*, where* $\varphi := \{\alpha \mapsto u_\alpha \mid \alpha \in \mathrm{Img}(\mathcal{T}), \mathcal{T}' \vdash \psi(\mathcal{T}^{-1}(\alpha)) \rightsquigarrow u_\alpha\}$.

Proof. See proof of Lemma 5.20. □

Lemma 5.22. *Let* $\mathcal{T} \vdash \tau \rightsquigarrow u$*,* $\mathcal{T}$ *bijective,* $\mathrm{Img}(\mathcal{T}) \subseteq \mathit{TypVar}$*, and* $\mathrm{Dom}(\mathcal{T}) = \mathrm{FV}^{\mathsf{a}}(\tau)$*. Let* ψ *be a substitution with* $\mathrm{Dom}(\psi) = \mathrm{FV}^{\mathsf{a}}(\tau)$*. Given* $\mathcal{T}'$ *such that* $\mathrm{FV}^{\mathsf{a}}(\psi) \subseteq \mathrm{Dom}(\mathcal{T}')$*, then* $\mathcal{T}' \vdash \psi(\tau) \rightsquigarrow \varphi(u)$ *where* $\varphi := \{\alpha \mapsto u_\alpha \mid \alpha \in \mathrm{Img}(\mathcal{T}), \mathcal{T}' \vdash \psi(\mathcal{T}^{-1}(\alpha)) \rightsquigarrow u_\alpha\}$.

Proof. We first note that φ is well-defined because of Corollary 5.18 and the given assumptions. The proof itself is by induction over the structure of τ. □

The following definition introduces the important notion of well-typed signature environments. Intuitively, a signature environment Δ is well-typed with respect to a constraint environment Θ and a variable environment Γ if the signatures in Δ correctly reflect the information contained in Θ and Γ.

Definition 5.23 (Well-typed signature environments). A signature environment Δ is said to be well-typed with respect to a constraint environment Θ and a variable environment Γ (short: Δ well-typed w.r.t. Θ, Γ), if the following holds for every $\mathsf{C} \in \mathrm{Dom}(\Delta)$:

1. $\emptyset \vdash \Delta(\mathsf{C}) \triangleright \Lambda\{\alpha\}.\mathcal{S}, \mathrm{FV}^{\alpha}(\mathcal{S}) \subseteq \{\alpha\}$
2. $\mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) = \alpha$
3. For all $\mathsf{m} \in \mathrm{Dom}(\Gamma)$ with $\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\,\mathsf{b} \Rightarrow \tau$: $\mathrm{x}^{\mathsf{m}} \in \mathrm{Dom}(\mathcal{S}), \mathrm{FV}^{`a}(\mathcal{S}(\mathrm{x}^{\mathsf{m}})) = \emptyset$, and $\mathcal{S}(\mathrm{x}^{\mathsf{m}}) = \forall\{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}.u$ with $\{\mathsf{b} \mapsto \alpha\} \dot{\cup} \{\mathsf{a} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\} \vdash \tau \rightsquigarrow u$
4. For all $\mathsf{C}' \in \mathrm{Sup}(\Theta, \mathsf{C})$: $\mathsf{C}' \in \mathrm{Dom}(\Delta), \emptyset \vdash \Delta(\mathsf{C}') \triangleright \Lambda\{\alpha'\}.\mathcal{S}', \mathrm{x}^{\mathsf{C}'} \in \mathrm{Dom}(\mathcal{S})$, $\mathrm{FV}^{`a}(\mathcal{S}(\mathrm{x}^{\mathsf{C}'})) = \emptyset$, and $\mathcal{S}(\mathrm{x}^{\mathsf{C}'}) = <[\alpha/\alpha']\mathcal{S}'>$
5. $\mathcal{S}$ contains no other elements

We call Δ well-typed if only conditions 1 and 2 hold.

The next definition introduces a notation for the denotation of dictionaries.

Definition 5.24 (Denotation of dictionaries). If Δ is well-typed, then the denotation of a structure representing the dictionary of an instance of class C at type u is written $\mathcal{S}_{\Delta}(\mathsf{C}, u)$, and defined as

$$\mathcal{S}_{\Delta}(\mathsf{C}, u) := [u/\alpha]\mathcal{S}$$

where $\Delta(\mathsf{C}) = \Lambda\{\alpha\}.\mathcal{S}$.

We can move substitutions "inside" the denotation of dictionaries, as formalized in the following lemma.

Lemma 5.25 ($\mathcal{S}_{\Delta}$ and substitutions). *If Δ is well-typed, and φ is a substitution from $TypVar$ to $SimTyp$, then $\varphi(\mathcal{S}_{\Delta}(\mathsf{C}, u)) = \mathcal{S}_{\Delta}(\mathsf{C}, \varphi(u))$. Similarly, if ϕ is a substitution from $SimTypVar$ to $SimTyp$, then $\phi(\mathcal{S}_{\Delta}(\mathsf{C}, u)) = \mathcal{S}_{\Delta}(\mathsf{C}, \phi(u))$.*

Proof. Δ is well-typed, so we have $\Delta(\mathsf{C}) = \Lambda\{\alpha\}.\mathcal{S}$ and $\mathrm{FV}^{\alpha}(\mathcal{S}) \subseteq \{\alpha\}$. Hence, $\varphi(\mathcal{S}_{\Delta}(\mathsf{C}, u)) = \varphi([u/\alpha]\mathcal{S}) = [\varphi(u)/\alpha]\mathcal{S} = \mathcal{S}_{\Delta}(\mathsf{C}, \varphi(u))$.

The proof of the second claim is similar, we just note that $\mathrm{FV}^{`a}(\mathcal{S}) = \emptyset$ because Δ is well-typed. □

The following three lemmata allow us to reason about the denotations of frequently used Tiny-ML^{+} constructs.

Lemma 5.26 (Denotation of type realizations in signature expressions). *If* Δ *well-typed, and* $\mathcal{C} \vdash \mathrm{u} \triangleright u$*, then* $\mathcal{C} \vdash \Delta(\mathsf{C})$ **where type** $\mathrm{t} = \mathrm{u} \triangleright \mathcal{S}_\Delta(\mathsf{C}, u)$*.*

Proof. Follows directly from the assumptions and rule $(sigexp_{patch})^+$. □

Lemma 5.27 (Denotation of superclass components). *Let* Δ *be well-typed w.r.t.* Θ, Γ*. If* $\mathsf{C}' \in \mathrm{Sup}(\Theta, \mathsf{C})$ *and* $\mathcal{S} = \mathcal{S}_\Delta(\mathsf{C}, u)$*, then* $\mathrm{x}^{\mathsf{C}'} \in \mathrm{Dom}(\mathcal{S})$ *and* $\mathcal{S}(\mathrm{x}^{\mathsf{C}'}) = <\mathcal{S}_\Delta(\mathsf{C}', u)>$*.*

Proof. Follows directly from the assumptions and Definition 5.23. □

Lemma 5.28 (Denotation of method components). *Let* Δ *be well-typed w.r.t.* Θ, Γ*. If* $\mathsf{m} \in \mathrm{Dom}(\Gamma)$*,* $\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau$*,* $\mathcal{S} = \mathcal{S}_\Delta(\mathsf{C}, u)$*, then* $\mathrm{x}^{\mathsf{m}} \in \mathrm{Dom}(\mathcal{S})$*, and* $\mathcal{S}(\mathrm{x}^{\mathsf{m}}) = \forall \{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}.u'$ *where* $\{\mathsf{b} \mapsto u\} \dot\cup \{\mathsf{a} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\} \vdash \tau \rightsquigarrow u'$*.*

Proof. Follows from the well-typedness of Δ and Lemma 5.19. □

The next two lemmata describe the form of translated type schemes and of translated function types.

Lemma 5.29 (Translation of type schemes). *If* $\Delta; \Sigma; \mathcal{C} \vdash \sigma \rightsquigarrow v$ *and* $\sigma = \forall \mathsf{A}.\ \overline{\mathsf{C}_i\ \tau_i}^{i \in [n]} \Rightarrow \tau$*, then* $v = \forall \{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}.\overline{<\mathcal{S}_\Delta(\mathsf{C}_i, u_i)>}^{i \in [n]} \to u$*, such that* $\Sigma'; \mathcal{C}' \vdash \tau_i \rightsquigarrow u_i$ *and* $\Sigma'; \mathcal{C}' \vdash \tau \rightsquigarrow u$ *with* $\Sigma' := \Sigma \overrightarrow{\cup} \{\mathsf{a} \mapsto `\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}$*,* $\mathcal{C}' := \mathcal{C} \overrightarrow{\cup} \{`\mathrm{a}^{\mathsf{a}} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}$*.*

Proof. Straightforward induction on the rules defining $\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$, $\Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}$, $\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$, $\mathcal{C} \vdash \mathrm{u} \triangleright u$, and $\mathcal{C} \vdash \mathrm{v} \triangleright v$. We use Lemma 5.26 for the case $(typtrans_{qual})^t$. □

Lemma 5.30 (Translation of function types). *If* $\mathcal{T} \vdash \tau_1 \to \tau_2 \rightsquigarrow u$*, then* $u = u_1 \to u_2$ *with* $\mathcal{T} \vdash \tau_1 \rightsquigarrow u_1$ *and* $\mathcal{T} \vdash \tau_2 \rightsquigarrow u_2$*.*

Proof. Straightforward rule induction. □

The following two lemmata are a simple weakening and strengthening lemma.

Lemma 5.31 (Weakening).

1. *If* $\mathcal{C} \vdash \mathrm{u} \triangleright u$ *and* $\mathrm{X} \notin \mathrm{FV}^{\mathrm{X}}(\mathrm{u})$*, then* $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S} \vdash \mathrm{u} \triangleright u$ *for arbitrary* $\mathcal{S}$*.*
2. *If* $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{S}$ *and* $\mathrm{X} \notin \mathrm{FV}^{\mathrm{X}}(\mathrm{S})$*, then* $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S}' \vdash \mathrm{S} \triangleright \mathcal{S}$ *for arbitrary* $\mathcal{S}'$*.*
3. *If* $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ *and* $\mathrm{X} \notin \mathrm{FV}^{\mathrm{X}}(\mathrm{s})$*, then* $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S} \vdash \mathrm{s} : \mathcal{X}$ *for arbitrary* $\mathcal{S}$*.*
4. *If* $\mathcal{C} \vdash \mathrm{e} : u$ *and* $\mathrm{X} \notin \mathrm{FV}^{\mathrm{X}}(\mathrm{e})$*, then* $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S} \vdash \mathrm{e} : u$ *for arbitrary* $\mathcal{S}$*.*
5. *If* $\mathcal{C} \vdash \mathrm{e} : u$ *and* $\mathrm{c} \notin \mathrm{FV}^{\mathrm{c}}(\mathrm{e})$*, then* $\mathcal{C}, \mathrm{c} \mapsto v \vdash \mathrm{e} : u$ *for arbitrary* v*.*

Proof. Obvious. □

Lemma 5.32 (Strengthening). *If* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$*, then* $\Sigma; \mathcal{C} \setminus (\mathrm{c}, \mathcal{C}(\mathrm{c})) \vdash \tau \rightsquigarrow \mathrm{u}$ *for arbitrary* $\mathrm{c} \in \mathrm{Dom}(\mathcal{C})$.

Proof. Trivial. □

Before we can start with the actual correctness proofs, we need to establish a connection between a Tiny-ML$^+$ context $\mathcal{C}$ and the environments Δ, Σ, and Θ used in the Tiny-HS typing judgments. Intuitively, $\mathcal{C}$ is compatible with Δ, Σ, and Θ if $\mathcal{C}$ correctly reflects the information recorded in Δ, Σ, and Θ.

Definition 5.33 (Compatibility of Tiny-ML$^+$ contexts). Let Δ be well-typed w.r.t. Θ and some arbitrary Γ, let Θ be well-formed, $\mathrm{CS}(\Theta) \subseteq \mathrm{Dom}(\Delta)$, and suppose that $\mathrm{FV}^{\mathsf{a}}(\Theta^l) \subseteq \mathrm{Dom}(\Sigma)$ for a type environment Σ. A context $\mathcal{C}$ is said to be compatible with Δ, Σ, and Θ, if the following conditions hold:

1. For all $\mathrm{u} \in \mathrm{Img}(\Sigma)$, there is some u such that $\mathcal{C} \vdash \mathrm{u} \triangleright u$.
2. For all $(\mathsf{C}\ \tau, \mathrm{e}) \in \Theta^l$, we have $\mathrm{FV}^{\mathrm{c}}(\mathrm{e}) \subseteq \mathrm{FreshCoreIds}$, and $\mathcal{C} \vdash \mathrm{e} : <\mathcal{S}_\Delta(\mathsf{C}, u)>$ with $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$.
3. For all $\theta := (\forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i \in [r]} \Rightarrow \mathsf{C}\ \tau) \in \mathrm{Dom}(\Theta^i)$, we have $\Theta^i(\theta) = \mathrm{F} \in \mathrm{Dom}(\mathcal{C})$, $\mathcal{C}(\mathrm{F}) = \forall P.\overline{\mathcal{S}}^{r+1} \to \mathcal{S}$, and with $\mathsf{B} := \mathrm{FV}^{\mathsf{a}}(\tau) \setminus \{\overline{\mathsf{a}}^r\}$, the following conditions hold:
   - 3.1. $\mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i)$ for all $i \in [r]$,
   - 3.2. $P = \{\mathcal{S}_i(\mathrm{t}) \mid i \in [r+1], \mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i)\}$
   - 3.3. $\mathcal{S}_i = \mathcal{S}_\Delta(\mathsf{C}_i, \mathcal{S}_i(\mathrm{t}))$ for all $i \in [r]$
   - 3.4. $\mathcal{S}_i(\mathrm{t}) = \mathcal{S}_j(\mathrm{t})$ iff $\mathsf{a}_i = \mathsf{a}_j$ for $i, j \in [r]$
   - 3.5. $\mathrm{Dom}(\mathcal{S}_{r+1}) = \{\mathrm{t}^{\mathsf{b}} \mid \mathsf{b} \in \mathsf{B}\}$ and $\mathcal{S}_{r+1}(\mathrm{t}^{\mathsf{b}}) \neq \mathcal{S}_{r+1}(\mathrm{t}^{\mathsf{b}'})$ if $\mathsf{b} \neq \mathsf{b}'$
   - 3.6. $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$
   - 3.7. $\mathcal{S} = \mathcal{S}_\Delta(\mathsf{C}, u)$, where $\{\mathsf{a}_i \mapsto \mathcal{S}_i(\mathrm{t}) \mid i \in [r]\} \dot\cup \{\mathsf{b} \mapsto \mathcal{S}_{r+1}(\mathrm{t}^{\mathsf{b}}) \mid \mathsf{b} \in \mathsf{B}\} \vdash \tau \rightsquigarrow u$
   - 3.8. $\mathrm{FV}^{\alpha}(\mathcal{C}(\mathrm{F})) = \emptyset$

The following lemma formalizes the intuition that the expression resulting from an entailment derivation is a first-class structure of the "right" signature.

Lemma 5.34 (Type correctness of entailment with translation). *If* $\Delta; \Sigma; \Theta \Vdash \mathsf{C}\ \tau \rightsquigarrow \mathrm{e}$*, then* $\mathcal{C} \vdash \mathrm{e} : <\mathcal{S}_\Delta(\mathsf{C}, u)>$ *with* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$*, provided*

- Δ *well-typed w.r.t.* Θ *and some arbitrary* Γ
- Θ *well-formed*
- $\mathrm{CS}(\Theta) \cup \{\mathsf{C}\} \subseteq \mathrm{Dom}(\Delta)$

- $\mathrm{FV}^{\mathsf{a}}(\Theta^{l}) \cup \mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$
- *$\mathcal{C}$ compatible with Δ, Σ, and Θ*

Proof. The proof is by rule induction. See Appendix B on page 141 for a detailed proof. □

The next two lemmata state that if we can translate a type, then this type does not contain any new type variables or class identifiers.

Lemma 5.35 (Type translation and free type variables).

- *If* $\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$, *then* $\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$.
- *If* $\Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}$, *then* $\mathrm{FV}^{\mathsf{a}}(\rho) \subseteq \mathrm{Dom}(\Sigma)$.
- *If* $\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$, *then* $\mathrm{FV}^{\mathsf{a}}(\sigma) \subseteq \mathrm{Dom}(\Sigma)$.

Proof. Straightforward induction on the structure of τ, ρ, and σ. □

Lemma 5.36 (Type translation and class identifiers). *If* $\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$, *then* $\mathrm{CS}(\sigma) \subseteq \mathrm{Dom}(\Delta)$.

Proof. Immediate from the rule $(\mathit{typtrans}_{\mathit{qual}})^{t}$. □

Similarly, if we can derive some constraint, then this constraint does not contain new type variables.

Lemma 5.37 (Entailment with translation and free type variables). *If* $\Delta; \Sigma; \Theta \Vdash \pi \rightsquigarrow \mathrm{e}$ *and* $\mathrm{FV}^{\mathsf{a}}(\Theta) \subseteq \mathrm{Dom}(\Sigma)$, *then* $\mathrm{FV}^{\mathsf{a}}(\pi) \subseteq \mathrm{Dom}(\Sigma)$.

Proof. Straightforward induction on the derivation of $\Delta; \Sigma; \Theta \Vdash \pi \rightsquigarrow \mathrm{e}$, using Lemma 5.35 for the cases $(\mathit{inst}_{\mathit{entail}})^{t}$ and $(\mathit{super}_{\mathit{entail}})^{t}$. □

We also need a substitution lemma for the expression translation.

Lemma 5.38 (Substitution lemma for expression translation). *Let ψ be a substitution from* TypVar *to* Typ. *If* $\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \tau$, $\mathrm{FV}^{\mathsf{a}}(\Theta) \subseteq \mathrm{Dom}(\Sigma)$, *and* $\mathrm{Dom}(\psi) \cap \mathrm{Dom}(\Sigma) = \emptyset$, *then* $\Delta; \Sigma; \Theta; \psi(\Gamma) \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \psi(\tau)$.

Proof. By induction on the structure of w. See Appendix B, page 143. □

Now we can prove that a translated expression has the translated type of the source expression.

Lemma 5.39 (Type correctness of expression translation). *If* $\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \tau$, *then* $\mathcal{C} \vdash \mathrm{e} : u$ *with* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$, *provided*

1. *Δ well-typed w.r.t. Θ, Γ*
2. *Θ well-formed*

3. Γ *unambiguous*, $\mathrm{FV}^{\mathsf{a}}(\Gamma(\mathsf{m})) = \emptyset$ *for all* $\mathsf{m} \in \mathrm{Dom}(\Gamma)$
4. $\mathrm{CS}(\Theta) \cup \mathrm{CS}(\Gamma) \subseteq \mathrm{Dom}(\Delta)$
5. $\mathrm{FV}^{\mathsf{a}}(\Theta) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma) \cup \mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$
6. $\mathcal{C}$ *compatible with* Δ, Σ, *and* Θ
7. *If* $\Gamma(\mathsf{z}) = \sigma$, *then* $\mathcal{C}(\mathrm{c}^{\mathsf{z}}) = v$ *with* $\Delta; \Sigma; \mathcal{C} \vdash \sigma \rightsquigarrow v$

Proof. By induction on the structure of w. See Appendix B, page 146. □

The following lemma states that Tiny-ML$^+$ judgments do not introduce new type variables. It is very similar to Lemma 4.6 on page 67, except that the lemma here is for Tiny-ML$^+$ and not for Tiny-ML.

Lemma 5.40 (Free variables and Tiny-ML$^+$ typing judgments).

- $\mathcal{C} \vdash \mathrm{u} \triangleright u$ *implies* $\mathrm{FV}^{\alpha}(u) \subseteq \mathrm{FV}^{\alpha}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{v} \triangleright v$ *implies* $\mathrm{FV}^{\alpha}(v) \subseteq \mathrm{FV}^{\alpha}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}$ *implies* $\mathrm{FV}^{\alpha}(\mathcal{L}) \subseteq \mathrm{FV}^{\alpha}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$ *implies* $\mathrm{FV}^{\alpha}(\mathcal{L}) \subseteq \mathrm{FV}^{\alpha}(\mathcal{C})$.
- $\mathcal{C} \overset{\mathit{funargs}}{\vdash} \overline{\mathrm{X}_i : \mathrm{S}_i}^{i \in [n]} \triangleright \forall P.\overline{\mathcal{S}}^n$ *implies* $\mathrm{FV}^{\alpha}(\forall P.\overline{\mathcal{S}}^n) \subseteq \mathrm{FV}^{\alpha}(\mathcal{C})$.

Proof. All propositions except the last one are proved by parallel induction on the term size. The last proposition is proved by induction on n. □

The following lemma shows that the translation of an instance definition, which is a recursive functor, defines the signature for the functor body in a way that correctly reflects the instance definition.

Lemma 5.41 (Denotation of recursive functors resulting from instance definition translation). *If* $\Delta; \Theta; \Gamma \vdash \mathsf{inst} \rightsquigarrow \mathrm{rfun}, \vdash \mathsf{inst} \rightsquigarrow \Theta'$ *with* $\Theta' \subseteq \Theta$, *and* Δ *is well-typed w.r.t.* Θ *and* Γ, *then* $\emptyset \vdash \mathrm{rfun} \triangleright \mathcal{C}$ *such that* $\mathcal{C}$ *is compatible with* Δ, $\Sigma = \emptyset$, *and* Θ'.

Proof. See Appendix B, page 151 □

We also want to show that the body of such a recursive functor matches the signature of the functor body. We need two more lemmata before we can prove this proposition.

Lemma 5.42 (Type translation and substitutions). *If* $\Sigma; \mathcal{C} \vdash [\tau'/\mathsf{a}]\tau \rightsquigarrow u$ *and* $\Sigma; \mathcal{C} \vdash \tau' \rightsquigarrow u'$, *then* $\{\mathsf{a} \mapsto u'\} \dot\cup \{\mathsf{b} \mapsto u_{\mathsf{b}} \mid \mathsf{b} \in \mathrm{FV}^{\mathsf{a}}(\tau) \setminus \{\mathsf{a}\}, \Sigma; \mathcal{C} \vdash \mathsf{b} \rightsquigarrow u_{\mathsf{b}}\} \vdash \tau \rightsquigarrow u$.

Proof. Straightforward structural induction on τ. □

Lemma 5.43 (Properties of method translation). *If* $\Delta; \Sigma; \Theta; \Gamma; \tau \overset{method}{\vdash} \mathsf{m} = \mathsf{w} \rightsquigarrow \mathrm{e} : \mathrm{v}$ *and*

- Δ *well-typed w.r.t.* Θ, Γ
- Θ *well-formed*
- Γ *unambiguous,* $\mathrm{Dom}(\Gamma) \subseteq \mathsf{MethodId}$
- $\mathrm{CS}(\Theta) \cup \mathrm{CS}(\Gamma) \subseteq \mathrm{Dom}(\Delta)$
- $\mathrm{FV}^{\mathsf{a}}(\Theta) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma) = \emptyset$, $\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$
- $\mathcal{C}$ *compatible with* Δ, Σ, *and* Θ
- $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$

then we also have

1. $\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau'$
2. $\mathrm{v} = \forall \{`\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}.\mathrm{u}'$
3. $\{\mathsf{b} \mapsto u\} \dot\cup \{\mathsf{a} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\} \vdash \tau' \rightsquigarrow u'$
4. $\{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\} \cap \mathrm{FV}^{`a}(\mathcal{C}) = \emptyset$

and for $\mathcal{C}' := \mathcal{C}, \{`\mathrm{a}^{\mathsf{a}} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}$

5. $\mathcal{C}' \vdash \mathrm{u}' \triangleright u'$
6. $\mathcal{C}' \vdash \mathrm{e} : u'$

Proof. See Appendix B, page 152. □

Now we can prove that the body of a recursive functor that is the translation of an instance definition matches the signature of the functor body.

Lemma 5.44 (Classification of recursive functors resulting from instance definition translation). *If* $\Delta; \Theta; \Gamma \vdash \mathsf{inst} \rightsquigarrow \mathrm{rfun}$, *and*

1. Δ *well-typed w.r.t.* Θ, Γ
2. Θ *well-formed*
3. Γ *unambiguous,* $\mathrm{Dom}(\Gamma) \subseteq \mathsf{MethodId}$
4. $\mathrm{CS}(\Theta) \cup \mathrm{CS}(\Gamma) \subseteq \mathrm{Dom}(\Delta)$
5. $\mathrm{FV}^{\mathsf{a}}(\Theta) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma) = \emptyset$
6. $\mathcal{C}$ *compatible with* Δ, $\Sigma = \emptyset$, *and* Θ

7. $\emptyset \vdash \mathrm{rfun} \triangleright \{\mathrm{F} \mapsto \mathcal{C}(\mathrm{F})\}, \mathcal{C}(\mathrm{F}) = \forall P.\overline{\mathcal{S}}^{r+1} \to \mathcal{S}$

8. $\mathrm{rfun} = \textbf{functor}\ \mathrm{F}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[r]}, \mathrm{Y} : \mathrm{S}_\mathrm{Y}) : \mathrm{S} = \mathrm{s}$

then we have that $\mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i\in[r]}, \mathrm{Y} \mapsto \mathcal{S}_{r+1} \vdash \mathrm{s} : \mathcal{S}$.

Proof. See Appendix B, page 153 □

The following lemma shows that the translation judgment for classes (Figure 5.8 on page 90) produces only well-typed class environments.

Lemma 5.45 (Well-typedness of class environments). *If* $\Delta \vdash \mathsf{cls} \rightsquigarrow \Delta'; \Theta'; \Gamma', \mathsf{C}$ *is the class defined by* cls*, and*

- Δ *well-typed w.r.t. some* Θ, Γ
- $\mathrm{Dom}(\Delta) \cap \mathrm{Dom}(\Delta') = \emptyset$
- $\mathrm{Sup}(\Theta, \mathsf{C}) = \emptyset$
- $\mathrm{Dom}(\Gamma) \cap \mathrm{Dom}(\Gamma') = \emptyset$ *and there is no* m *with* $\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \rho$

then $\Delta \dot\cup \Delta'$ *is well-typed w.r.t.* $\Theta \dot\cup \Theta'$ *and* $\Gamma \dot\cup \Gamma'$.

Proof. See Appendix B, page 154. □

Eventually, we are able to prove the main result of this section.

Theorem 5.46 (Type correctness of program translation). *If* $\vdash \mathsf{pgm} \rightsquigarrow \mathrm{prog}$ *and* pgm *well-formed, then* $\emptyset \vdash \mathrm{prog}$.

Proof. See Appendix B, page 154. □

5.4. Restrictions on the source language Tiny-HS

As already mentioned at the beginning of Section 3.2, the source language Tiny-HS of our translation to ML modules does not support constructor classes, class methods with constraints, and default definitions for methods. We now discuss why these restriction are necessary, and if and how we could extend the translation so as to remove the restrictions.

Constructor classes. A constructor class [Jon95a] is a type class that ranges over (possibly higher-order) type constructors. The translation to ML modules cannot support constructor classes because ML does not support higher-order type constructors. You might argue that we could at least translate constructor classes ranging over first-order type constructors, because parameterizable type components could be used as the translation of first-order type constructors. Indeed, it is straightforward to write the signature corresponding to such a type class. For example, a type class for monads

```
class ∀{m} . Monad m where
  bind   :: ∀{a, b} . m a → (a → m b) → m b
  return :: ∀{a} . a → m a
```

might be translated into the following signature (parameterizable type components, which are not supported in Tiny-ML$^+$, are written **type** t $\overline{\text{'a}}$):

```
signature Monad =
  sig
    type m 'a
    val  bind   : ∀{'a, 'b} . m 'a → ('a → m 'b) → m 'b
    val  return : ∀{'a} . 'a → m 'a
  end
```

However, we run into serious problems when we try to translate a type scheme like ∀{a, b, m}.Monad m ⇒ m a → m b → m b. We cannot translate the constraint Monad m into a package type <Monad **where type** m = 'm> because the type variable 'm would then be of higher order. The type

```
∀{'a, 'ma, 'b, 'mb} . <Monad where type m 'a = 'ma where type m 'b = 'mb>
                        → 'ma → 'mb → 'mb
```

looks like a solution, but it is invalid because the second type realization **where type** m 'b = 'mb refers to the now transparent type component m. What we would really need is the ability to name the content of the package type <Monad>; that is, something like

```
∀{'a, 'b} . <X : Monad> → X.m 'a → X.m 'b → X.m 'b
```

Neither Tiny-ML$^+$ nor Russo's proposal for first-class structures [Rus00a] supports such named package types. Another possible approach to the problem, which seems to work even for higher-order constructor classes, is to simulate higher-order types with functors. Both approaches are considered interesting future work.

Class methods with constraints. In Haskell 98, method signatures may contain constraints, which may lead to recursive classes. Here is a rather artificial example:

```
class ∀{a} . C a where
  foo :: ∀{b} . C b ⇒ b → a
```

We cannot translate such a recursive class into an ML signature because the resulting signature would be recursive as well. Inventing some syntax for binding recursive signature variables, we could write the signature for C as follows:

μZ.**sig type** t **val** foo: $\forall\{$'b$\}$.<Z **where type** t = 'b> $\rightarrow$ 'b $\rightarrow$ t **end**

Crary et al. [CHP99] introduced the notion of *recursively dependent signatures*; however, in their setting, the recursion variable is a structure, not a signature variable.

Default definitions for methods. In Haskell 98, type class may provide default definitions for methods. The translation could handle such default definitions by copying the translated code of a default definition to the translations of those instance definitions that do not overwrite the default definition. However, it is not possible to put default definitions directly into signatures because signatures cannot contain code for value components.

5.5. Implementation

A Haskell implementation of the translation from Tiny-HS to Tiny-ML$^+$ is available from `http://www.stefanwehr.de/diplom`. The implementation is based on Jones' "Typing Haskell in Haskell" [Jon00b] and on the overloading-resolution algorithm described by Peterson and Jones [PJ93]. Moscow ML [RRK+03] acts as the target language of the implementation because it supports all features of Tiny-ML$^+$. The example in Figures 5.1 and 5.2 was checked against the implementation.

A minor problem in implementing the translation is how to construct the type environment Σ' in the premise of rule $(let)^t$ in Figure 5.6 on page 88. We already discussed that we would need to guess the correct Σ', but this is not feasible in an implementation. We can solve the problem by maintaining the type environment only for those Tiny-HS type variables a that are not mapped to their "trivial" Tiny-ML$^+$ counterpart 'a^a. Thereby, we can use Σ instead of Σ' in the premise of rule $(let)^t$.

5.6. Related work

A common approach to describe the meaning of programs with ad-hoc polymorphism is to make ad-hoc polymorphism explicit by passing around evidence values (dictionaries) at runtime. Wadler & Blott [WB89] introduce Haskell type classes and present an evidence translation into a language that resembles the implicitly typed, polymorphic λ-calculus [Hin69, Mil78, DM82]. Their source language assumes that each type class has exactly one method.

Jones [Jon94] presents a general theory of qualified types, of which type classes in Haskell are a specific application. Jones defines three equivalent type systems for qualified types: the first type system is declarative, the second system uses

syntax-directed rules, and the third system is an algorithmic system. Jones uses these type systems to define type-directed translations into a version of the polymorphic λ-calculus extended with constructs for evidence application and abstraction. Jones' approach to qualified types is more general than type classes; therefore, he does not deal directly with type classes and instances, and treats evidence values as an abstract concept.

Hall et al. [HHPW96] present an evidence translation for a source language that supports all important features of type classes in Haskell. Their target language is similar to System F [Gir72, Rey74]. Faxén [Fax02] gives a static semantics for Haskell 98, which includes an evidence translation into a variant of System F_ω [Gir72].

The translation from Tiny-HS to Tiny-ML$^+$ (and so the type system of Tiny-HS) is based on Jones' syntax directed system [Jon94, Figure 3.2]. We now discuss why his (probably better known) declarative system [Jon94, Figure 3.1] cannot be used for the translation. Rule $(\mathit{method})^t$ for method variables in Figure 5.6 on page 88 exemplifies the problem. The declarative system has different rules for variables, for quantifier elimination, and for constraint elimination. If the translation was based on this system, we would lookup a method variable in the environment and instantiate the type of the method to a monotype at *two different* places in the derivation tree. But then we could no longer construct the translation of the method variable because part of the information necessary to construct the translation (i.e., the name of the method) is available at the place where we lookup the variable in the environment, but needed at the place where we fully instantiate the type of the method.

The translations presented by Hall et al. and Faxén map methods to ordinary top-level bindings that select the appropriate component from a dictionary passed to them. Translating methods in such a way would allow us to use Jones' declarative system. However, we cannot translate methods to top-level bindings because selecting a component from a first-class structure (i.e., a dictionary) requires opening the first-class structure. But the type information necessary for opening the first-class structure is only available at the point where the method is actually used, and cannot be passed to a top-level binding. Passing type information around is not a problem in the translations presented by Hall et al. and Faxén because their target languages support explicit type abstraction and application.

Support for ad-hoc polymorphism is very limited in Standard ML [MTHM97]: Only a few overloaded operators are supported, overloading has to be resolvable at compile time, and user-defined functions cannot be overloaded. The equality operator is an exception from this rule: The programmer can use it to define overloaded functions, and overloading introduced by the equality operator can be resolved at runtime.

Schneider [Sch00] adds Haskell-style type classes to ML. His solution is conservative in the sense that type classes and modules remain two separate concepts. He does not translate type classes into modules.

Chapter 6.

Discussion

The last chapter of the thesis discusses and summarizes the results of the preceding chapters. Section 6.1 presents a thorough comparison between ML modules and Haskell type classes. Section 6.2 outlines possible topics for future research based on the material of this work. Finally, Section 6.3 summarizes the contributions and concludes.

6.1. ML modules and Haskell type classes: a comparison

Chapter 4 and Chapter 5 presented formal translations from ML modules to Haskell type classes and vice versa. Building on the insights obtained by developing these translations, we now draw a detailed comparison between ML modules and Haskell type classes.

The comparison proceeds in two steps. Section 6.1.1 compares the two concepts viewing Haskell type classes as a replacement for ML modules. Then we change the standpoint and view ML modules as an alternative to Haskell type classes; the comparison from this perspective is presented in Section 6.1.2. Some of the issues mentioned in these two sections were already discussed earlier; we repeat them here for the sake of completeness.

6.1.1. Classes as modules

The translation from ML modules to Haskell type classes in Chapter 4 demonstrates that Haskell type classes can be used to simulate certain aspects of the ML module system: Signatures are translated into type classes, structures and functors are modeled as instances of the type classes corresponding to the signatures of the structures and functors, and type and value components of signatures and structures are translated into associated type synonyms (not part of Haskell 98 [Pey03], see [CKP05]) and type class methods, respectively. The translation also discloses several differences between the two concepts, which are discussed in the following paragraphs.

Namespace management. ML modules provide proper namespace management, whereas Haskell type classes do not: Two different type classes cannot define two

associated type synonyms or two methods with the same name (unless the two classes are defined in different Haskell modules).

Signature and structure components. Signatures and structures in ML may contain all sorts of language constructs: values, types, exceptions, substructures, and so on. Type classes and instances in Haskell 98 may contain only methods; extensions to Haskell 98 also allow type synonyms [CKP05] and data types [CKPM05]. However, there exists no extension that allows nested type classes and instances.

Sequential versus recursive definitions. Definitions in ML are type checked and evaluated sequentially, with special support for recursive data types and recursive functions. In particular, recursive definitions of type components in structures are not possible because a type component can be used only *after* its definition.

In Haskell, all top-level definitions are mutually recursive, so Chakravarty et al. [CKP05] need extra conditions to avoid nonterminating associated type synonym definitions. However, their termination conditions disallow certain associated type synonym definitions that are needed for the translation from ML modules to Haskell type classes. Hence, we could not use their conditions in the target language of this translation. Nevertheless, all associated type synonym definitions in the translation from ML modules to Haskell type classes are terminating because type components in structures (the counterpart of associated type synonyms) cannot be defined recursively, as described in the preceding paragraph.

Implicit versus explicit signatures. In ML, signatures of structures are inferred implicitly. In Haskell, the type class to which an instance definition belongs has to be stated explicitly. However, we saw in Section 2.4 that we need explicit signatures in ML as well once we introduce recursive functors, so the difference between implicit and explicit signatures boils down to the difference between sequential and recursive definitions, which we discussed in the preceding point of our comparison.

Anonymous versus named signatures. Signatures in ML are essentially anonymous because named signatures can be removed from the language without losing expressiveness. Haskell type classes (which are the counterpart of ML signatures) cannot be anonymous.

Structural versus nominal signature matching. The difference between anonymous and named signatures becomes relevant if we compare signature matching in ML with its Haskell counterpart. In ML, matching a structure against a signature is performed by comparing the components of the structure with the components of the signature; the names of the structure and the signature—if present at all—do not matter. This sort of signature matching is often called *structural*

matching (here, the term "structural" is not to be confused with a structure in the ML-sense).

The Haskell analogon of signature matching is verifying whether the type representing a structure is an instance of the type class representing the signature. The name of a class plays an important role in deciding whether or not some type is an instance of the class. Therefore, we can characterize the Haskell analogon of signature matching as *nominal*.

Translucent versus transparent signatures. A key feature of the ML module system are translucent signatures: They allow fine-grained control over how much type information is propagated, and they are essential to support fully syntactic signatures [Sha99]. Signatures in Haskell (i.e., type classes) are transparent: The definitions of associated type synonyms in instances are always visible through a type class.[1] The difference between translucent and transparent signatures becomes relevant in the following two points of our comparison.

Abstraction. In ML, abstraction is performed by sealing a structure with a translucent or opaque signature. Haskell supports only transparent signatures, so abstraction has to be performed in instance definitions. We used abstract associated type synonyms, a contribution of the work at hand, for this purpose.

Separate compilation. We already saw in Chapter 2 that Standard ML [MTHM97] supports incremental compilation, and that there are extensions supporting separate compilation [Ler94, HL94, Sha99]. Incremental compilation, not to mention separate compilation, is not possible for Haskell type classes (if we regard them as a replacement for ML modules) because type classes are transparent, so it is impossible to write fully syntactic signatures with them. Interestingly, the situation for Haskell type classes with respect to separate/incremental compilation is the same as for ML modules before the introduction of transparent type components and strong sealing [Ler94, HL94].

Unsealed and sealed view. A sealed structure body may look different depending on whether we view the body from inside or outside the signature seal: Inside, more values and types may be visible, some types may be concrete, and some values may have a more polymorphic type than outside.

With Haskell type classes, the same set of types and values is visible, and a value has the same type, regardless of whether we view the instance from inside or outside. See page 59 for an example illustrating this difference.

[1] Associated type synonyms can have default definitions in type classes. The difference to transparent type components in ML signatures is that a default definition of an associated type synonym can be overwritten with an arbitrary type in the instances of the class, whereas a transparent type component has to be defined equivalently in a structure matching the signature.

First–class structures. First–class structures are a nontrivial extension to Standard ML. In Haskell, we get first–class structures for free because a structure is represented as an arbitrary value of a certain type (see Section 4.1).

6.1.2. Modules as classes

In Chapter 5, we demonstrated how to simulate ad-hoc polymorphism introduced by Haskell type classes with ML modules: Type classes are translated into signatures, instances are mapped to recursive functors (an extension to Standard ML, see [Rus01]), and first-class structures (an extension to Standard ML, see [Rus00a]) are used as dictionaries providing runtime evidence for type class constraints. However, ML modules cannot replace Haskell type classes completely. We now discuss the points missing to make ML modules a coequal replacement for Haskell type classes.

Implicit versus explicit overloading resolution. Overloading in Haskell is resolved implicitly by the compiler. If type classes are simulated with ML modules, overloading has to be resolved explicitly by the programmer, which leads to awkward and overly verbose code (see Figure 5.2 on page 83 for an example).

Constructor classes. Constructor classes in Haskell cannot be translated to ML because higher-order types are not supported in ML. Type checking in the presence of higher-order types is decidable in Haskell because Haskell maintains a clear distinction between *data types* (which introduce new type constructors that can be applied partially yielding higher-order types) and *type synonyms* (which introduce only abbreviations for existing types that cannot be applied partially).

ML does not have this clear distinction between data types and type synonyms. On the one hand, this gives programmers the freedom to specify some type component of a signature as a type synonym, and to implement the same component in a structure matching the signature as a data type. On the other hand, the missing distinction makes it impossible to support higher-order types in ML. See Section 5.4 for further details.

Recursive classes. Type classes in Haskell may be recursive; that is, a class can be used in a constraint for a method of the same class. We cannot translate such recursive classes to ML because signatures cannot be recursive. See Section 5.4 for further details.

Default definitions for methods. Haskell type classes may contain default definitions for methods. Such default definitions cannot be translated properly to ML because signatures specify only the types of value components and cannot contain implementations of value components.

6.2. Future work

This section outlines topics based on the results of this work that I consider worthwhile for future research.

ML with type classes. The translation from Haskell type classes to ML modules shows that it is feasible to integrate Haskell-style type classes smoothly into ML module systems with support for first-class structures: Type classes become designated signatures (maybe with some restrictions on the content of the signatures) and instances are represented by a special form of structure and functor definitions. This approach is particularly interesting because it naturally allows for two useful features [KS04] not found in Haskell's type class system:

- Type classes and instances can be nested because they are just a special form of signatures, structures, and functors.
- A syntax for explicit dictionary application and abstraction can be provided because dictionaries are available to the programmer as first-class structures.

Constructor classes. Constructor classes are an important feature of Haskell's type class system. However, they are difficult to translate to ML because ML does not support higher-order types. Section 5.4 discussed two approaches to the problem; it would be worthwhile to develop a proper solution.

Flexible termination conditions for associated type synonyms. The target language of the translation from ML modules to Haskell type classes allows nonterminating associated type synonym definitions (see Section 3.3.5) because the termination conditions suggested by Chakravarty et al. [CKP05] are not suitable for the translation from modules to type classes. It would be beneficial to find more flexible termination conditions that are compatible with the way associated type synonyms are used in the translation from modules to type classes.

Recursive signatures. The translation from Haskell type classes to ML modules demonstrates that recursive signatures together with first-class structures are useful (see also Section 5.4). However, no formalization of recursive signatures exists yet. It would be worthwhile to develop such a formalization.

6.3. Summary and conclusions

I demonstrated how ML modules can be translated to Haskell type classes, proved that the translation preserves type correctness, and implemented the translation. The source language of the translation is a subset of Standard ML, the most important feature missing is the ability to define nested structures. The target language

is a subset of Haskell 98 extended with multi-parameter type classes and (abstract) associated type synonyms. Abstract associated type synonyms, another contribution of this work, are used to translate abstract types in ML adequately to Haskell. I believe that it is feasible to use the general idea behind the translation for practical programming because a lot of the overhead introduced by the formal translation can be avoided when writing the Haskell code by hand, as demonstrated in an example of such a manual translation (Figure 4.1 on page 50).

Furthermore, I showed that Haskell type classes can be translated into ML modules by using first-class structures as runtime evidence for type class constraints. I proved that the translation preserves type correctness, and provided an implementation of the translation. The source language of the translation is a subset of Haskell 98, which does not support constructor classes, class methods with constraints, and default definitions for methods. The target language is a subset of Standard ML extended with first-class structures and recursive functors. I do not recommend writing programs in the style of the translation by hand because too much syntactic overhead is introduced by explicit dictionary abstraction and application, and by opening and packaging first-class structures (see Figure 5.2 on page 83 for an example). However, the translation provides a good starting point for integrating type classes into the ML module system.

Finally, I presented a thorough comparison between ML modules and Haskell type classes, which fills a serious gap in the literature because it is the first comparison between the two concepts that is based on formal translations. The comparison shows that there are also significant differences between modules and type classes.

Appendix A.

Code

Example translation from Tiny-ML to Tiny-HS^{+}

The following Tiny-HS^{+} code is the result of the formal translation developed in Section 4.2 applied to the Tiny-ML code from Figure 4.1(a). The code was obtained by using the implementation of the translation discussed in Section 4.5.

```
data T^MkSet*
data T^MkSet
data T^IntEq
data T^MkSet,1
data T^IntSet

class C^MkSet,arg a where
  type S^MkSet,1,t a
  z^MkSet,1,eq :: a → S^MkSet,1,t a → S^MkSet,1,t a → Bool

class C^MkSet,arg a ⇒ C^MkSet* b a where
  type S^MkSet*,set b a
  z^MkSet*,empty  :: b → a → [c]
  z^MkSet*,member :: b → a → S^MkSet,1,t a → [S^MkSet,1,t a] → Bool
  z^MkSet*,insert :: b → a → S^MkSet,1,t a → [S^MkSet,1,t a] → [S^MkSet,1,t a]

class C^MkSet,arg a ⇒ C^MkSet b a where
  type S^MkSet,set b a
  z^MkSet,empty  :: b → a → S^MkSet,set b a
  z^MkSet,member :: b → a → S^MkSet,1,t a → S^MkSet,set b a → Bool
  z^MkSet,insert :: b → a → S^MkSet,1,t a → S^MkSet,set b a → S^MkSet,set b a

class C^IntEq a where
  type S^IntEq.t a
  z^IntEq.eq :: a → Int → Int → Bool

class C^IntSet a where
  type S^IntSet.set a
  z^IntSet.empty  :: a → S^IntSet.set a
  z^IntSet.member :: a → Int → S^IntSet.set a → Bool
  z^IntSet.insert :: a → Int → S^IntSet.set a → S^IntSet.set a
```

```
instance C^MkSet,arg a ⇒ C^MkSet⋆ T^MkSet⋆ a where
  type S^MkSet⋆,set T^MkSet⋆ a = [S^MkSet,1,t a]
  z^MkSet⋆,empty  = λ_ . λz . []
  z^MkSet⋆,member = λ_ . λz . λx . λs . exists (λy . z^MkSet,1,eq z x y) s
  z^MkSet⋆,insert = λ_ . λz . λx . λs .
                      if z^MkSet⋆,member (⊥ :: T^MkSet⋆) z x s then s else (x : s)
instance C^MkSet,arg a ⇒ C^MkSet T^MkSet a where
  abstype S^MkSet,set T^MkSet a = S^MkSet⋆,set T^MkSet⋆ a
  z^MkSet,empty  = λ_ . λz . z^MkSet⋆,empty (⊥ :: T^MkSet⋆) z
  z^MkSet,member = λ_ . λz . z^MkSet⋆,member (⊥ :: T^MkSet⋆) z
  z^MkSet,insert = λ_ . λz . z^MkSet⋆,insert (⊥ :: T^MkSet⋆) z
instance C^IntEq T^IntEq where
  type S^IntEq.t T^IntEq = Int
  z^IntEq.eq = λ_ . λi . λj . i == j
instance C^MkSet,arg T^MkSet,1 where
  type S^MkSet,1,t T^MkSet,1 = Int
  z^MkSet,1,eq = λ_ . z^IntEq.eq (⊥ :: T^IntEq)
instance C^IntSet T^IntSet where
  type S^IntSet.set T^IntSet = S^MkSet,set T^MkSet T^MkSet,1
  z^IntSet.empty  = λ_ . z^MkSet,empty (⊥ :: T^MkSet) (⊥ :: T^MkSet,1)
  z^IntSet.member = λ_ . z^MkSet,member (⊥ :: T^MkSet) (⊥ :: T^MkSet,1)
  z^IntSet.insert = λ_ . z^MkSet,insert (⊥ :: T^MkSet) (⊥ :: T^MkSet,1)
```

Appendix B.

Long proofs

Proofs for Section 4.3.1

Proof of Lemma 4.10

By induction on the structure of b.

Case $\mathrm{b} = \epsilon_\mathrm{b}$: Trivial.

Case $\mathrm{b} = \textbf{type}\,\mathrm{t} = \mathrm{u}^{\langle u \rangle}; \mathrm{b}'$: We have $\mathcal{C} \vdash \mathrm{u} \triangleright u$ because of $\mathcal{C} \vdash \mathrm{s} : \mathcal{S}$. Hence, $\mathrm{FV}^\alpha(u) \overset{\text{Lemma 4.6}}{\subseteq} \mathrm{FV}^\alpha(\mathcal{C}) \overset{\mathrm{Valid}(\mathcal{C},\Phi)}{\subseteq} \mathrm{Dom}(\Phi)$. Now the claim follows by Lemma 4.9 and the induction hypothesis.

Case $\mathrm{b} = \textbf{val}\,\mathrm{x} = \mathrm{e}; \mathrm{b}'$: We have $\mathcal{C} \vdash \mathrm{e} : \mathcal{S}(\mathrm{x})$ because of $\mathcal{C} \vdash \mathrm{s} : \mathcal{S}$. Now the proposition follows with Lemma 4.8 and the induction hypothesis. □

Proof of Lemma 4.11

By structural induction on s.

Case $\mathrm{s} = \textbf{struct}\,\mathrm{b}\,\textbf{end}$: Immediate from Lemma 4.10.

Case $\mathrm{s} = \mathrm{X}^{\langle \mathcal{S} \rangle}$: The last rule in the derivation of $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ must be $(\mathit{strexp}_{\mathit{var}})$, hence $\mathcal{C}(\mathrm{X}) = \mathcal{S}$. The claim now follows from the assumption $\mathrm{Valid}(\mathcal{C}, \Phi)$ and Lemma 4.9.

Case $\mathrm{s} = \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}'}^n \to \mathcal{X} \rangle}(\overline{\mathrm{X}_i^{\langle \mathcal{S}_i \rangle}}^{i \in [n]})$: The last rule in the derivation of $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ must be $(\mathit{strexp}_{\mathit{fapp}})$. We get from the premise of this rule

$$\begin{gathered} \mathcal{S}_i \succcurlyeq \varphi(\mathcal{S}_i') \quad (i \in [n]) \\ \mathcal{C}(\mathrm{X}_i) = \mathcal{S}_i \end{gathered}$$

Now we have $\mathrm{Dom}(\mathcal{S}_i') \subseteq \mathrm{Dom}(\mathcal{S}_i)$, so $\mathfrak{T}_u[\![\mathcal{S}_i(\mathrm{t})]\!]\Phi$ (in the definition of the associated type synonym $\mathsf{S}^{\mathrm{F},i,\mathrm{t}}$) is well-defined for all $\mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i')$ by Lemma 4.9 and $\mathrm{Valid}(\mathcal{C}, \Phi)$, and $\Phi(\mathrm{X}_i, \mathrm{y})$ (in the definition of method $\mathsf{z}^{\mathrm{F},i,\mathrm{y}}$) is well-defined for all $\mathrm{y} \in \mathrm{Dom}(\mathcal{S}_i')$ because $\mathrm{Valid}(\mathcal{C}, \Phi)$. □

Proof of Lemma 4.13

It is straightforward to verify that all usages of Ω, $\mathfrak{T}_v$, $\mathfrak{S}$ and pick in the definition of $\mathfrak{X}$ are well-defined:

- $\Omega(\mathrm{t})$ in the definition of the associated type synonym $\mathsf{S}^{\mathrm{X.t}}$ and $\Omega(\mathrm{y})$ in the definition of the method $\mathsf{z}^{\mathrm{X.y}}$ are well-defined by Lemma 4.12.
- $\mathfrak{T}_v[\![\mathcal{S}(\mathrm{y})]\!]\Phi'$ and $\mathfrak{S}[\![\mathrm{s}]\!]\Phi''$ are well-defined by Lemmata 4.9 and 4.11, respectively.
- $\mathrm{pick}(\mathcal{S}, \alpha)$ in the definition of Φ' and Φ'' is well-defined by Lemma 4.5. □

Proof of Lemma 4.14

It is straightforward to verify that all usages of Ω, $\mathfrak{T}_u$, $\mathfrak{T}_v$, $\mathfrak{S}$ and pick in the definition of $\mathfrak{F}$ are well-defined:

- $\Omega(\mathrm{t})$ in the definition of the associated type synonym $\mathsf{S}^{\mathrm{F,t}}$ and $\Omega(\mathrm{x})$ in the definition of the method $\mathsf{z}^{\mathrm{F,x}}$ are well-defined by Lemma 4.12.
- $\mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi'$ and $\mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi'$ are well-defined by Lemma 4.9.
- $\mathrm{pick}(\overline{\mathcal{S}}, \alpha)$ in the definition of Φ' is well-defined because $\forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}$ is ground.
- $\mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi''$ is well-defined by Lemma 4.11. Notice that Φ'' is valid with respect to $\mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i\in[n]}$. □

Proof of Theorem 4.16

By structural induction on prog. We perform an additional case analysis on the form of the right-hand side of a program definition, so that we can distinguish between sealed and unsealed structure expressions.

Case $\mathrm{prog} = \textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S}\rangle}; \mathrm{prog}'$: The last rule in the derivation of $\mathcal{C} \vdash \mathrm{prog}$ must be $(\mathit{prog}_{\mathit{str}})$. We get from the premise of this rule:

$$\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$$
$$\mathcal{C}, \mathrm{X} \mapsto \mathcal{S} \vdash \mathrm{prog}'$$

The well-definedness is now straightforward to check:

- $\mathfrak{X}[\![\textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S}\rangle}]\!]\Phi$ is well-defined by Lemma 4.13.
- $\mathfrak{P}[\![\mathrm{prog}']\!]\Phi'$ is well-defined by the induction hypothesis because Φ' is valid w.r.t. $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$.
- $\mathrm{pick}(\mathcal{S}, \alpha)$ in the definition of Φ' is well-defined by Lemma 4.5.

CASE $\mathrm{prog} = \textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P'.\mathcal{S}' \rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle}; \mathrm{prog}'$: Again, the last rule in the derivation must be $(\mathit{prog}_{\mathit{str}})$. We get from the premise of this rule and from the premise of rule $(\mathit{strexp}_{\mathit{sealed}})$:

$$\begin{gathered}\mathcal{C} \vdash \mathrm{s} :> \mathrm{S} : \exists P.\mathcal{S} \\ \mathcal{C} \vdash \mathrm{s} : \exists P'.\mathcal{S}' \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \\ \mathcal{C}, \mathrm{X} \mapsto \mathcal{S} \vdash \mathrm{prog}'\end{gathered}$$

The well-definedness is now easy to check:

- $\mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))$ in the definition of the associated type synonym $\mathrm{S}^{\mathrm{X.t}}$ is well-defined because $\mathcal{S}(\mathrm{t}) \in P$.
- $\mathfrak{X}[\![\textbf{structure}\ \mathrm{X}^\star = \mathrm{s}^{\langle \exists P'.\mathcal{S}' \rangle}]\!]\Phi$ is well-defined by Lemma 4.13.
- $\mathfrak{P}[\![\mathrm{prog}']\!]\Phi'$ is well-defined by the induction hypothesis because Φ' is valid w.r.t. $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$.
- $\mathrm{pick}(\mathcal{S}, \alpha)$ in the definition of Φ' and Φ'' is well-defined by Lemma 4.3.
- $\mathfrak{T}_v[\![\mathcal{S}(\mathrm{y})]\!]\Phi''$ is well-defined by Lemma 4.9.

CASE $\mathrm{prog} = \textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S} \rangle}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]}) = \mathrm{ps}^{\langle \mathcal{S} \rangle}; \mathrm{prog}'$: The last rule in the derivation must be $(\mathit{prog}_{\mathit{fun}})$. We get from the premise of this rule:

$$\begin{gathered}\mathcal{C} \overset{\text{funargs}}{\vdash} \overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]} \triangleright \forall Q.\overline{\mathcal{S}}^n \\ \mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i \in [n]} \vdash \mathrm{ps} : \mathcal{S} \\ \mathcal{C}, \mathrm{F} \mapsto \underbrace{\forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}}_{=:\mathcal{F}} \vdash \mathrm{prog}'\end{gathered}$$

Now the well-definedness is straightforward to verify:

- $\mathfrak{F}[\![\textbf{functor}\ \mathrm{F}^{\langle \mathcal{F} \rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i \in [n]}) = \mathrm{ps}^{\langle \mathcal{S} \rangle}]\!]\mathrm{F}\Phi$ is well-defined by Lemma 4.14. Notice that $\mathcal{F}$ is ground by Lemma 4.15.
- $\mathfrak{P}[\![\mathrm{prog}']\!]\Phi$ is well-defined by the induction hypothesis. Notice that Φ is valid w.r.t. $\mathcal{C}, \mathrm{F} \mapsto \mathcal{F}$ because $\mathrm{FV}^\alpha(\mathcal{C}) = \mathrm{FV}^\alpha(\mathcal{C}, \mathrm{F} \mapsto \mathcal{F})$ by Lemmata 4.6 and 4.15.

CASE $\mathrm{prog} = \textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S} \rangle}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]}) = \mathrm{ps}^{\langle \mathcal{S}' \rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle}; \mathrm{prog}'$: Again, the last rule in the derivation must be $(\mathit{prog}_{\mathit{fun}})$. We get from the premise of this

rule and from the premise of rule $(\mathit{strexp}_{\mathit{sealed}})$:

$$\mathcal{C} \overset{\text{funargs}}{\vdash} \overline{\mathrm{X_i} : \mathrm{S_i}}^{i\in[n]} \triangleright \forall Q.\overline{\mathcal{S}}^n$$
$$\underbrace{\mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i\in[n]}}_{=:\mathcal{C}'} \vdash \mathrm{ps} : \mathcal{S}'$$
$$\mathcal{C}' \vdash \mathrm{S} \triangleright \wedge P.\mathcal{S}$$
$$\mathcal{C}, \mathrm{F} \mapsto \underbrace{\forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S}}_{=:\mathcal{F}} \vdash \mathrm{prog}'$$

The well-definedness is now straightforward to verify:

- $\mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))$ in the definition of the associated type synonym $\mathrm{S}^{\mathrm{F,t}}$ is well-defined because $\mathcal{S}(\mathrm{t}) \in P$.
- $\mathfrak{F}[\![\textbf{functor}\ \mathrm{F}^{\star\langle\mathcal{F}'\rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) = \mathrm{ps}^{\langle\mathcal{S}'\rangle}]\!]\mathrm{F}\Phi$ is well-defined by Lemma 4.14 ($\mathcal{F}' := \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}'$ is ground for the same reason as in the preceding case).
- $\mathfrak{P}[\![\mathrm{prog}']\!]\Phi$ is well-defined by the induction hypothesis (Φ is valid w.r.t. $\mathcal{C}, \mathrm{F} \mapsto \mathcal{F}$ for the same reason as in the preceding case).
- $\mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi'$ is well-defined by Lemma 4.9.
- $\mathrm{pick}(\overline{\mathcal{S}}, \alpha)$ and $\mathrm{pick}(\mathcal{S}, \alpha)$ in the definition of Φ' and Φ'' are well-defined by Lemmata 4.15 and 4.3, respectively. □

Proofs for Section 4.3.2

Proof of Lemma 4.24

By induction on the structure of e:

CASE $\mathrm{e} = \mathrm{c}$: From the assumption $\mathcal{C} \vdash \mathrm{c} : u$, we get with rule $(\mathit{exp}_{\mathit{id}})$

$$\mathcal{C}(\mathrm{c}) = v = \forall A.u'$$
$$\phi(u') = u \quad \text{where } \mathrm{Dom}(\phi) = A$$

W.l.o.g., $A \subseteq \mathrm{FV}^{`a}(u')$. Hence, with the assumption $\mathrm{FV}^{\alpha}(u) \subseteq \mathrm{Dom}(\Phi)$,

$$\mathrm{FV}^{\alpha}(\phi) \subseteq \mathrm{Dom}(\Phi)$$

We now define

$$\psi := \{\mathsf{a}^{`a} \mapsto \mathfrak{T}_u[\![\phi(`a)]\!]\Phi \mid `a \in A\}$$

We obtain by the assumption $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi} \mathcal{C}$:

$$\hat{\Gamma}(\mathsf{z}^{\mathrm{c}}) = \mathfrak{T}_v[\![v]\!]\Phi = \forall\{\mathsf{a}^{`a} \mid `a \in A\}.\mathfrak{T}_u[\![u']\!]\Phi \tag{1}$$

W.l.o.g., $\{\mathsf{a}^{`a} \mid `a \in A\} \cap \mathrm{FV}^{\mathsf{a}}(\Phi) = \emptyset$. Furthermore, $\mathrm{FV}^{\alpha}(u') \subseteq \mathrm{FV}^{\alpha}(u) \subseteq \mathrm{Dom}(\Phi)$, so with Lemma 4.21

$$\mathfrak{T}_u[\![u]\!]\Phi = \mathfrak{T}_u[\![\phi(u')]\!]\Phi = \psi(\mathfrak{T}_u[\![\mathrm{u}']\!]\Phi) \tag{2}$$

We get $\hat{\Theta} \vdash \mathfrak{T}_u[\![\phi(`a)]\!]\Phi$ for all $`a \in A$ with Lemma 4.20, so

$$\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{z}^{\mathrm{c}} : \mathfrak{T}_u[\![u]\!]\Phi$$

by rule $(var)^+$, by Equations 1 and 2, and by Lemma 4.23.

Case $\mathrm{e} = \lambda \mathrm{c}.\mathrm{e}'$: The last rule in the derivation of $\mathcal{C} \vdash \lambda \mathrm{c}.\mathrm{e}' : u$ must be (exp_{abs}). Hence, we have

$$u = u_1 \to u_2$$
$$\mathcal{C}, \mathrm{c} \mapsto u_1 \vdash \mathrm{e}' : u_2$$

Let us define

$$\tau_1 := \mathfrak{T}_u[\![u_1]\!]\Phi$$

To be able to apply the induction hypothesis, we need to show

$$\hat{\Theta}; \hat{\Gamma}, \mathsf{z}^{\mathrm{c}} \mapsto \tau_1 \equiv^{\Phi} \mathcal{C}, \mathrm{c} \mapsto u_1 \tag{3}$$

From the assumption $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi} \mathcal{C}$ we get $\mathrm{FV}^{\mathsf{z}}(\Phi) \cap \{\mathsf{z}^{\mathrm{c}} \mid \mathrm{c} \in \mathrm{CoreId}\} = \emptyset$. Now Equation 3 follows from the definition of τ_1 and the weakening lemma 4.22. Hence

$$\hat{\Theta}; \hat{\Gamma}, \mathsf{z}^{\mathrm{c}} \mapsto \tau_1 \vdash \mathfrak{E}[\![\mathrm{e}']\!]\Phi : \mathfrak{T}_u[\![u_2]\!]\Phi$$

by the induction hypothesis and $\hat{\Theta} \vdash \tau_1$ by Lemma 4.20. Therefore, rule $(\to I)^+$ can be used to derive the desired result.

Case $\mathrm{e} = \mathrm{e}_1\ \mathrm{e}_2$: Follows directly from the induction hypothesis and rule $(\to E)^+$.

Case $\mathrm{e} = \mathbf{let}\ \mathrm{c} = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2$: The last rule in the derivation of $\mathcal{C} \vdash \mathrm{e} : u$ must be (exp_{let}), therefore

$$\mathcal{C} \vdash \mathrm{e}_1 : v = \forall A.u' \tag{4}$$
$$\mathcal{C}, \mathrm{e} \mapsto v \vdash \mathrm{e}_2 : u$$

Equation 4 gives us

$$\mathcal{C} \vdash \mathrm{e}_1 : u'$$
$$A \cap \mathrm{FV}^{`a}(\mathcal{C}) = \emptyset$$

We can safely assume that

$$\{\mathsf{a}^{`a} \mid `a \in A\} \cap (\mathrm{FV}^{\mathsf{a}}(\hat{\Theta}) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Gamma})) = \emptyset$$

Now the induction hypothesis and repeated applications of rule $(\forall I)^+$ yield

$$\hat{\Theta}; \hat{\Gamma} \vdash \mathfrak{E}[\![\mathrm{e}_1]\!]\Phi : \underbrace{\mathfrak{T}_v[\![v]\!]\Phi}_{=:\sigma}$$

The same argumentation as in the case "$\mathrm{e} = \lambda \mathrm{c}.\mathrm{e}$" can be used that show that

$$\hat{\Theta}; \hat{\Gamma}, \mathsf{z}^{\mathrm{c}} \mapsto \sigma \equiv^{\Phi} \mathcal{C}, \mathrm{c} \mapsto v$$

Hence, by the induction hypothesis

$$\hat{\Theta}; \hat{\Gamma}, \mathsf{z}^{\mathrm{c}} \mapsto \sigma \vdash \mathfrak{E}[\![\mathrm{e}_2]\!]\Phi : \mathfrak{T}_u[\![u]\!]\Phi$$

The claim now follows from rule $(let)^+$ (σ is unambiguous because it does not contain any constraints).

CASE $\mathrm{e} = \mathrm{x}$, $\mathrm{e} = \mathrm{X}.\mathrm{x}$: Analogous to the case "e = c". □

Proof of Lemma 4.26

The proof is by structural induction on b.

CASE $\mathrm{b} = (strb_\epsilon)$: Obvious.

CASE $\mathrm{b} = \textbf{type}\ \mathrm{t} = \mathrm{u}^{\langle u \rangle}; \mathrm{b}'$: The last rule in the derivation of $\mathcal{C} \vdash \mathrm{b} : \mathcal{S}$ must be $(strb_t)$. Hence, we have

$$\begin{gathered} \mathcal{C} \vdash \mathrm{u} \triangleright u \\ \mathcal{C}, \mathrm{t} \mapsto u \vdash \mathrm{b}' : \mathcal{S}' \\ \mathcal{S} = \mathcal{S}', \mathrm{t} \mapsto u \end{gathered} \tag{5}$$

Now by definition of $\mathfrak{S}_\mathrm{b}$

$$\Omega(\mathrm{t}) = \mathfrak{T}_u[\![u]\!]\Phi = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi$$

With Lemma 4.20, we get

$$\hat{\Theta} \vdash \Omega(\mathrm{t})$$

Finally, we prove the claim by applying the induction hypothesis to Equation 5. Notice that $\mathrm{FV}^\alpha(\mathcal{C}, \mathrm{t} \mapsto u) = \mathrm{FV}^\alpha(\mathcal{C})$ by Lemma 4.6, so $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi} \mathcal{C}, \mathrm{t} \mapsto u$.

CASE $\mathrm{b} = \textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b}'$: The last rule in the derivation of $\mathcal{C} \vdash \mathrm{b} : \mathcal{S}$ must be $(strb_v)$, so we have from the premise of this rule

$$\mathcal{C} \vdash \mathrm{e} : v \tag{6}$$

$$\begin{gathered} \underbrace{\mathcal{C}, \mathrm{x} \mapsto v}_{=:\mathcal{C}'} \vdash \mathrm{b}' : \mathcal{S}' \\ \mathcal{S} = \mathcal{S}', \mathrm{x} \mapsto v \end{gathered} \tag{7}$$

By Equation 6 and Corollary 4.25 we obtain

$$\hat{\Theta};\hat{\Gamma} \vdash \underbrace{\mathfrak{E}[\![\mathrm{e}]\!]\Phi}_{=\Omega(\mathrm{x})} : \underbrace{\mathfrak{T}_v[\![v]\!]\Phi}_{\mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi}$$

Applying the induction hypothesis to Equation 7 finishes the proof. Notice that $\hat{\Theta};\hat{\Gamma} \equiv^{\Phi} \mathcal{C}'$ holds because $\hat{\Theta};\hat{\Gamma} \vdash \Phi(\mathrm{x}) : \mathfrak{T}_v[\![\mathcal{C}'(\mathrm{x})]\!]\Phi$ by the assumptions and $\mathrm{FV}^{\alpha}(\mathcal{C}') = \mathrm{FV}^{\alpha}(\mathcal{C})$ by Lemma 4.6. □

Proof of Lemma 4.30

Let us define

$$\forall A.u := v$$
$$\forall A'.u' := v'$$

We can safely assume that

$$\begin{gathered}\mathrm{FV}^{\mathsf{a}}(\Phi) \cap \{\mathsf{a}^{`a} \mid `a \in A\} = \emptyset \\ (\mathrm{FV}^{\mathsf{a}}(\hat{\Theta}) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Gamma})) \cap \{\mathsf{a}^{`a} \mid `a \in A'\} = \emptyset \\ A \subseteq \mathrm{FV}^{`a}(u)\end{gathered} \tag{8}$$

We get from $v \succcurlyeq v'$ by Lemma 2.7

$$\phi(u) = u'$$
$$\mathrm{Dom}(\phi) = A$$

Now we have by Lemma 4.21

$$\begin{gathered}\mathfrak{T}_u[\![\phi(u)]\!]\Phi = \psi(\mathfrak{T}_u[\![u]\!]\Phi) \\ \psi = \{\mathsf{a}^{`a} \mapsto \mathfrak{T}_u[\![\phi(`a)]\!]\Phi \mid `a \in A\}\end{gathered} \tag{9}$$

By Lemma 4.20

$$\hat{\Theta} \vdash \psi(\mathsf{a}) \quad (\mathsf{a} \in \mathrm{Dom}(\psi))$$

By the assumption $\hat{\Theta};\hat{\Gamma} \vdash \mathsf{w} : \mathfrak{T}_v[\![v]\!]\Phi$, by the definition of $\mathfrak{T}_v$, and by Lemma 4.23

$$\hat{\Theta};\hat{\Gamma} \vdash \mathsf{w} : \underbrace{\psi(\mathfrak{T}_u[\![u]\!]\Phi)}_{\overset{\text{Equation 9}}{=}\ \mathfrak{T}_u[\![u']\!]\Phi}$$

Equation 8 allows us to apply rule $(\forall I)^+$ repeatedly, so we finally get

$$\hat{\Theta};\hat{\Gamma} \vdash \mathsf{w} : \mathfrak{T}_v[\![v']\!]\Phi$$

□

Proof of Lemma 4.31

We have $\mathcal{S}(\mathrm{x}) \succcurlyeq \varphi(\mathcal{S}'(\mathrm{x}))$, so by Lemma 4.30

$$\hat{\Theta}; \hat{\Gamma} \vdash \mathrm{w} : \mathfrak{T}_v[\![\varphi(\mathcal{S}'(\mathrm{x}))]\!]\Phi$$

By construction of Φ' and Lemma 4.27

$$\hat{\Theta} \Vdash \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{x})]\!]\Phi' = \underbrace{\mathfrak{T}_u[\![\varphi(\mathcal{S}'(\mathrm{x}))]\!]\Phi'}_{= \mathfrak{T}_u[\![\varphi(\mathcal{S}'(\mathrm{x}))]\!]\Phi}$$

The claim now follows by rule $(conv)^+$. □

Proof of Lemma 4.32

The proof is by structural induction on s. The interesting case is the one for functor application.

Case s = **struct** b **end**: Follows directly from Lemma 4.26 and from $m = 0$.

Case $\mathrm{s} = \mathrm{X}^{\langle \mathcal{S} \rangle}$: We have $\mathcal{C}(\mathrm{X}) = \mathcal{S}$ and $P = \emptyset$. The claim for the inst_i holds trivially because $m = 0$. The two claims for $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$ follow from Lemma 4.20 and from the definition of Ω. The claim for $\mathrm{x} \in \mathrm{Dom}(\mathcal{S})$ follows from the definition of Ω and from the assumption $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi \dot{\cup} \Phi'} \mathcal{C}$.

Case $\mathrm{s} = \mathrm{F}^{\langle k, \forall Q.\overline{\mathcal{S}'}^n \to \exists P.\mathcal{S}' \rangle}(\overline{\mathrm{X}_i^{\langle \mathcal{S}_i \rangle}}^{i \in [n]})$: We first establish some general properties. The last rule in the derivation of $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$ must be $(strexp_{fapp})$. We get from the premise of this rule

$$\begin{gathered}
\mathcal{C}(\mathrm{X}_i) = \mathcal{S}_i \\
\mathcal{C}(\mathrm{F}) = \forall Q.\overline{\mathcal{S}'}^n \to \exists P.\mathcal{S}' \\
\mathcal{S}_i \succcurlyeq \varphi(\mathcal{S}'_i) \\
\varphi(\exists P.\mathcal{S}') = \exists P.\mathcal{S} \\
\mathrm{Dom}(\varphi) = Q
\end{gathered}$$

We can safely assume that $Q \cap P = \emptyset$ and $\mathrm{FV}^\alpha(\varphi) \cap P = \emptyset$, hence

$$\varphi(\mathcal{S}') = \mathcal{S} \tag{10}$$

$\mathcal{S}_i \succcurlyeq \varphi(\mathcal{S}'_i)$ implies

$$\mathcal{S}_i(\mathrm{t}) = \varphi(\mathcal{S}'_i(\mathrm{t}))$$

$\mathcal{C}$ is ground, so we get

$$\varphi = \{\alpha \mapsto \mathcal{S}_i(\mathrm{t}) \mid \alpha \in Q, \langle i, t \rangle = \mathrm{pick}(\overline{\mathcal{S}'}, \alpha)\} \tag{11}$$

Let inst be defined as in the relevant case of $\mathfrak{S}$. Clearly,

$$\vdash \mathsf{inst} : \hat{\Theta}''$$
$$\hat{\Theta}'' := \{\mathsf{C}^{\mathrm{F},arg}\ \mathsf{T}^{\mathrm{F},k}\} \cup \{\mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{T}^{\mathrm{F},k} = \mathfrak{T}_u[\![\mathcal{S}_i(\mathrm{t})]\!]\Phi \mid i \in [n], \mathrm{t} \in \mathrm{Dom}(\mathcal{S}'_i)\}$$

Note that Φ' has no effect in the translation of types.

Let us define

$$\hat{\Theta}' := \hat{\Theta} \cup \hat{\Theta}''$$

We then have

$$\hat{\Theta}' \Vdash \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{T}^{\mathrm{F},k}$$
$$\hat{\Theta}' \Vdash \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{T}^{\mathrm{F},k} = \mathfrak{T}_u[\![\mathcal{S}_i(\mathrm{t})]\!]\Phi \quad (i \in [n], \mathrm{t} \in \mathrm{Dom}(\mathcal{S}'_i)) \tag{12}$$

We get from the assumption $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi \dot\cup \Phi'} \mathcal{C}$ that

$$(\forall \mathsf{a}.\mathsf{C}^{\mathrm{F},arg}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}) \in \hat{\Theta}$$

so we have by rule $(mp_{entail})^+$

$$\hat{\Theta}' \Vdash \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k} \tag{13}$$

Our first goal is to prove that $\hat{\Theta}'; \hat{\Gamma} \vdash \mathsf{inst}$. We first note that the $\hat{\Theta}'$ defined in the premise of rule $(inst_{check})^+$ is not different from the $\hat{\Theta}'$ defined here because inst has an empty context and defines no abstract associated type synonyms. The checks for entailment of super classes in the premise of the rule succeed trivially because we have $\mathrm{Sup}(\hat{\Theta}, \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}) = \emptyset$ by $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi \dot\cup \Phi'} \mathcal{C}$. Furthermore, all associated type synonyms defined in inst are well-formed by $\hat{\Theta} \vdash \Phi$ and Lemma 4.22. The last thing to check is the type correctness of the right-hand sides of the methods $\mathsf{z}^{\mathrm{F},i,y}$ for $i \in [n]$, $\mathrm{y} \in \mathrm{Dom}(\mathcal{S}'_i)$. We get from $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi \dot\cup \Phi'} \mathcal{C}$ and Lemma 4.22

$$\hat{\Gamma}(\mathsf{z}^{\mathrm{F},i,y}) = \forall \mathsf{A} \dot\cup \{\mathsf{a}\}.\mathsf{C}^{\mathrm{F},arg}\ \mathsf{a} \Rightarrow \mathsf{a} \to \tau$$
$$\forall \mathsf{A}.\tau = \mathfrak{T}_v[\![\mathcal{S}'_i(\mathrm{y})]\!]\Phi_1$$
$$\Phi_1 =: \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}'}, \alpha)\}$$
$$\hat{\Theta}'; \hat{\Gamma} \vdash \Phi(\mathrm{X}_i, \mathrm{y}) : \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{y})]\!]\Phi$$

By $\mathcal{S}_i \succcurlyeq \varphi(\mathcal{S}'_i)$, Lemma 4.31, and Equations 11 and 12

$$\hat{\Theta}'; \hat{\Gamma} \vdash \Phi(\mathrm{X}_i, \mathrm{y}) : \mathfrak{T}_v[\![\mathcal{S}'_i(\mathrm{y})]\!]\Phi'_1$$
$$\Phi'_1 := \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{T}^{\mathrm{F},k} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}'}, \alpha)\}$$

We have also

$$\forall \mathsf{A}.[\mathsf{T}^{\mathrm{F},k}/\mathsf{a}]\tau = [\mathsf{T}^{\mathrm{F},k}/\mathsf{a}](\mathfrak{T}_v[\![\mathcal{S}'_i(\mathrm{y})]\!]\Phi_1) \overset{\mathrm{FV}^{'a}(\mathcal{C})=\emptyset}{=}$$
$$\mathfrak{T}_v[\![\mathcal{S}'_i(\mathrm{y})]\!]([\mathsf{T}^{\mathrm{F},k}/\mathsf{a}]\Phi_1) \overset{\mathrm{FV}^{\mathrm{a}}(\Phi)=\emptyset}{=} \mathfrak{T}_v[\![\mathcal{S}'_i(\mathrm{y})]\!]\Phi'_1$$

hence

$$\hat{\Theta}'; \hat{\Gamma} \vdash \Phi(\mathrm{X}_i, \mathrm{y}) : \forall \mathsf{A}.[\mathsf{T}^{\mathrm{F},k}/\mathsf{a}]\tau$$

Applications of rules $(\forall E)^+$, $(\mathit{wildcard})^+$, and $(\forall I)^+$ yield

$$\hat{\Theta}'; \hat{\Gamma} \vdash \lambda\_.\Phi(\mathrm{X}_i, \mathrm{y}) : [\mathsf{T}^{\mathrm{F},k}/\mathsf{a}](\forall \mathsf{A}.\mathsf{a} \to \tau)$$

But this is exactly what we need in the premise of rule $(\mathit{inst}_{\mathit{check\text{-}method}})^+$, so we have

$$\hat{\Theta}'; \hat{\Gamma} \vdash \mathsf{inst}$$

The proof of the second proposition is straightforward: Because $\mathsf{S}^{\mathrm{F},\mathrm{t}}$ is an associated type synonym of class C^{F} and T^{F} is an user-defined type constructor of kind 0 (this follows from the assumption $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi \dot\cup \Phi'} \mathcal{C}$), we get by Equation 13, rule $(\mathit{wf}_{\mathit{tycon}})^+$, and rule $(\mathit{wf}_{\mathit{syn}})^+$

$$\hat{\Theta}' \vdash \Omega(\mathrm{t}) \quad (\text{for all } t \in \mathrm{Dom}(\mathcal{S}) = \mathrm{Dom}(\Omega))$$

We now prove the third claim, that is $\hat{\Theta}' \Vdash \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k} = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!](\Phi \dot\cup \Phi'')$ for all $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$. From the assumption $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi \dot\cup \Phi'} \mathcal{C}$ we get

$$\hat{\Theta}' \Vdash \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k} = \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\Phi_2 \tag{14}$$

$$\begin{aligned}\Phi_2 := \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{T}^{\mathrm{F},k} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}}, \alpha)\} \\ \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\}\end{aligned}$$

Let us define

$$\Phi_2' := \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{T}^{\mathrm{F},k} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}}, \alpha)\} \dot\cup \Phi''$$

By definition of Φ''

$$\begin{gathered}\mathrm{Dom}(\Phi_2) = \mathrm{Dom}(\Phi_2') \\ \hat{\Theta}' \Vdash \Phi_2(\alpha) = \Phi_2'(\alpha) \quad (\text{for all } \alpha \in \mathrm{Dom}(\Phi_2))\end{gathered}$$

We now get by Lemma 4.28 and by Equation 10

$$\hat{\Theta}' \Vdash \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi_2' = \mathfrak{T}_u[\![\varphi(\mathcal{S}'(\mathrm{t}))]\!]\Phi_2 \tag{15}$$

$\Phi \subseteq \Phi_2$, so by definition of $\hat{\Theta}''$ and by Equations 11 and 12

$$\hat{\Theta}' \Vdash \Phi_2(\alpha) = \mathfrak{T}_u[\![\varphi(\alpha)]\!]\Phi_2 \quad (\text{for all } \alpha \in \mathrm{Dom}(\varphi))$$

We obtain by using Lemma 4.27

$$\hat{\Theta}' \Vdash \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\Phi_2 = \mathfrak{T}_u[\![\varphi(\mathcal{S}'(\mathrm{t}))]\!]\Phi_2 \tag{16}$$

Now we have by Equations 14, 15, and 16 that

$$\hat{\Theta}' \Vdash \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k} = \underbrace{\mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!](\Phi \dot\cup \Phi'')}_{\overset{\mathrm{FV}^{\alpha}(\mathcal{S})\cap Q=\emptyset}{=} \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi_2'}$$

We now turn our attention to proving the last proposition, that is $\hat{\Theta}'; \hat{\Gamma} \vdash \mathsf{z}^{\mathrm{F},\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}})\ (\bot :: \mathsf{T}^{\mathrm{F},k}) : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!](\Phi \dot\cup \Phi'')$ for all $\mathrm{x} \in \mathrm{Dom}(\mathcal{S})$. We have from the assumption $\hat{\Theta}; \hat{\Gamma} \equiv^{\Phi \dot\cup \Phi'} \mathcal{C}$

$$\begin{gathered}
\hat{\Gamma}(\mathsf{z}^{\mathrm{F},\mathrm{x}}) = \forall \mathsf{A} \dot\cup \{\mathsf{a},\mathsf{b}\}.\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{b} \to \mathsf{a} \to \tau \\
\forall \mathsf{A}.\tau = \mathfrak{T}_v[\![\mathcal{S}'(\mathrm{x})]\!]\Phi_3 \\
\Phi_3 := \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}'}, \alpha)\} \\
\dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{b}\ \mathsf{a} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}', \alpha)\} \\
\forall B.u := \mathcal{S}'(\mathrm{x}) \\
\mathsf{A} = \{\mathsf{a}^{`a} \mid `a \in B\}
\end{gathered}$$

We can safely assume that

$$\begin{gathered}
(\mathsf{A} \cup \{\mathsf{a},\mathsf{b}\}) \cap (\mathrm{FV}^{\mathsf{a}}(\hat{\Theta}) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Gamma})) = \emptyset \\
\{\mathsf{a},\mathsf{b}\} \cap \mathrm{FV}^{\mathsf{a}}(\Phi) = \emptyset
\end{gathered}$$

By using rules $(\forall E)^+$, $(\Rightarrow E)^+$, $(\to E)^+$, and Equation 13, we can derive

$$\hat{\Theta}'; \hat{\Gamma} \vdash \underbrace{\mathsf{z}^{\mathrm{F},\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}})\ (\bot :: \mathsf{T}^{\mathrm{F},k})}_{=:\ \mathsf{w}} : \underbrace{[\mathsf{T}^{\mathrm{F},k}/\mathsf{a}, \mathsf{T}^{\mathrm{F}}/\mathsf{b}]\,\tau}_{=:\ \psi} \tag{17}$$

Let us define

$$\begin{aligned}
\Phi_3' := \psi(\Phi_3) = \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{T}^{\mathrm{F},k} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}'}, \alpha)\} \\
\dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}', \alpha)\}
\end{aligned}$$

Because $\Phi \subseteq \Phi_3'$, we get from Equations 11 and 12

$$\hat{\Theta}' \Vdash \Phi_3'(\alpha) = \mathfrak{T}_u[\![\varphi(\alpha)]\!]\Phi_3' \quad (\text{for all } \alpha \in \mathrm{Dom}(\varphi))$$

Hence, by Lemma 4.27

$$\hat{\Theta}' \Vdash \mathfrak{T}_u[\![u]\!]\Phi_3' = \underbrace{\mathfrak{T}_u[\![\varphi(u)]\!]\Phi_3'}_{=:\ \tau'} \tag{18}$$

W.l.o.g. $\{\mathsf{a}^{`a}, \mathsf{b}^{`\mathrm{a}}\} \cap \mathrm{FV}^{`a}(u) = \emptyset$, so we have

$$\psi(\tau) = \psi(\mathfrak{T}_u[\![u]\!]\Phi_3) = \mathfrak{T}_u[\![u]\!]\Phi_3' \tag{19}$$

Using Equations 17, 18, 19, we can derive with rule $(conv)^+$

$$\hat{\Theta}'; \hat{\Gamma} \vdash \mathsf{w} : \tau'$$

Because $\mathsf{A} \cap (\mathrm{FV}^{\mathsf{a}}(\hat{\Theta}) \cup \mathrm{FV}^{\mathsf{a}}(\hat{\Gamma})) = \emptyset$, we have by rule $(\forall I)^+$

$$\hat{\Theta}'; \hat{\Gamma} \vdash \mathsf{w} : \forall \mathsf{A}.\tau'$$

Moreover,

$$\mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi \overset{\Phi \subseteq \Phi'_3}{=\!=} \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi'_3 \overset{\mathcal{S}=\varphi(\mathcal{S}')}{=\!=} \mathfrak{T}_v[\![\forall B.\varphi(u)]\!]\Phi'_3 = \forall \mathsf{A}.\mathfrak{T}_u[\![\varphi(u)]\!]\Phi'_3 = \forall \mathsf{A}.\tau'$$

Hence

$$\hat{\Theta}'; \hat{\Gamma} \vdash \mathsf{w} : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!](\Phi \dot{\cup} \Phi'') \qquad \square$$

Proof of Lemma 4.36

Let Φ', Φ'', and $\forall \mathsf{B}_\mathrm{y}.\tau_\mathrm{y}$ be defined as in the body of $\mathfrak{X}$. We have from the assumptions

$$\begin{gathered} \vdash \overrightarrow{\mathsf{pv}'} : \hat{\Theta}'; \hat{\Gamma}' \\ \hat{\Theta}' \vdash \Phi \\ \hat{\Theta}'; \hat{\Gamma}' \equiv^{\Phi} \mathcal{C} \end{gathered} \tag{20}$$

and from the definition of $\mathfrak{X}$

$$\langle \Omega, \overline{\mathsf{ddec}}, \overline{\mathsf{inst}}^m \rangle = \mathfrak{S}[\![\mathrm{s}]\!]\Phi'' \tag{21}$$

By Lemma 4.29

$$\vdash \textbf{class } \mathsf{C}^{\mathrm{X}}\ \mathsf{a}\ \textbf{where} \ldots : \emptyset; \hat{\Gamma}'' \tag{22}$$

$$\hat{\Gamma}'' := \{\mathsf{z}^{\mathrm{X.y}} \mapsto \forall \mathsf{B}_\mathrm{y} \cup \{\mathsf{a}\}.\mathsf{C}^{\mathrm{X}}\ \mathsf{a} \Rightarrow \mathsf{a} \to \tau_\mathrm{y} \mid \mathrm{y} \in \mathrm{Dom}(\mathcal{S})\}$$

Clearly,

$$\vdash \textbf{instance } \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where} \ldots : \hat{\Theta}'' \tag{23}$$

$$\hat{\Theta}'' := \{\mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\} \cup \{\mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \Omega(\mathrm{t}) \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S})\} \tag{24}$$

Let us define

$$\hat{\Theta}''' := \hat{\Theta}' \cup \hat{\Theta}''$$
$$\hat{\Gamma} := \hat{\Gamma}' \dot{\cup} \hat{\Gamma}''$$

We get from Definition 4.18 and the weakening lemma 4.22

$$\hat{\Theta}''';\hat{\Gamma} \equiv^{\Phi''} \mathcal{C} \tag{25}$$

Moreover

$$\hat{\Theta}''' \vdash \Phi'' \tag{26}$$

Because $\hat{\Gamma}(z^{X.y})$ is closed it is easy to verify that

$$\hat{\Theta}''';\hat{\Gamma} \vdash z^{X.y}\ (\bot :: T^X) : [T^X/a](\forall B_y.\tau_y)$$

so we have with $FV^{'a}(\mathcal{C}) \cup FV^a(\Phi) = \emptyset$ that

$$\hat{\Theta}''';\hat{\Gamma} \vdash \Phi''(y) : \mathfrak{T}_v[\![\mathcal{S}(y)]\!]\Phi''' \tag{27}$$
$$\Phi''' := \Phi \dot\cup \{\alpha \mapsto S^{X.t}\ T^X \mid \alpha \in P, t = \mathrm{pick}(\mathcal{S},\alpha)\}$$

We can now apply Lemma 4.32 (Φ' and Φ'' mentioned in the assumptions of the lemma correspond to $\Phi'' \setminus \Phi$ and $\Phi''' \setminus \Phi$, respectively; moreover, Equations 21, 24, 25, and 27 give us the nontrivial assumptions) and obtain

$$\vdash \mathsf{inst}_i : \hat{\Theta}'_i \quad (i \in [m]) \tag{28}$$
$$\hat{\Theta} := \hat{\Theta}''' \cup \bigcup_{i\in[m]} \hat{\Theta}'_i$$
$$\hat{\Theta};\hat{\Gamma} \vdash \mathsf{inst}_i \quad (i \in [m]) \tag{29}$$
$$\hat{\Theta} \vdash \Omega(t) \quad (t \in \mathrm{Dom}(\mathcal{S})) \tag{30}$$
$$\hat{\Theta} \Vdash \Omega(t) = \mathfrak{T}_u[\![\mathcal{S}(t)]\!]\Phi''' \quad (t \in \mathrm{Dom}(\mathcal{S})) \tag{31}$$
$$\hat{\Theta};\hat{\Gamma} \vdash \Omega(y) : \mathfrak{T}_v[\![\mathcal{S}(y)]\!]\Phi''' \quad (y \in \mathrm{Dom}(\mathcal{S})) \tag{32}$$

We now show that the class and instance definitions for C^X produced by $\mathfrak{X}$ pass the rules $(class_{check})^+$ and $(inst_{check})^+$, respectively. Let us first prove the case for class definitions. We have

$$\hat{\Theta} \cup \{C^X\ a\} \vdash \Phi'$$

so by Lemma 4.20

$$\hat{\Theta} \cup \{C^X\ a\} \vdash \forall B_y.\tau_y$$

Therefore

$$\hat{\Theta} \vdash \textbf{class}\ C^X\ a\ \textbf{where}\ \ldots \tag{33}$$

Now we prove the case for the instance definition. From Equation 32 and from $FV^{'a}(\mathcal{C}) = FV^a(\Phi) = \emptyset$ we get

$$\hat{\Theta};\hat{\Gamma} \vdash \Omega(y) : [T^X/a](\mathfrak{T}_v[\![\mathcal{S}(y)]\!]\Phi') \quad (y \in \mathrm{Dom}(\mathcal{S}))$$

We can easily derive that

$$\hat{\Theta}; \hat{\Gamma} \vdash \lambda\_.\Omega(\mathrm{y}) : [\mathsf{T}^{\mathrm{X}}/\mathsf{a}](\forall \mathsf{B}_{\mathrm{y}}.\mathsf{a} \to \tau_{\mathrm{y}})$$

Hence

$$\hat{\Theta}; \hat{\Gamma} \overset{\text{method}}{\vdash} \mathsf{z}^{\mathrm{X.y}} = \lambda\_.\Omega(\mathrm{y})$$

Furthermore, we get by Equation 30

$$\hat{\Theta}; \mathsf{T}^{\mathrm{X}} \overset{\text{tdef}}{\vdash} \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \Omega(\mathrm{t})$$

Hence

$$\hat{\Theta}; \hat{\Gamma} \vdash \textbf{instance}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where} \ldots \tag{34}$$

Using Equations 20, 22, 23, 28, 29, 33, 34, and Lemma 4.35, we can show that

$$\vdash \mathsf{pv}' \overrightarrow{\oplus}\ \mathsf{pv} : \hat{\Theta}; \hat{\Gamma}$$

Obviously,

$$\hat{\Theta}; \hat{\Gamma} \vdash \widetilde{\Phi}(\mathrm{X}, \mathrm{y}) : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{y})]\!]\widetilde{\Phi} \quad (\mathrm{y} \in \mathrm{Dom}(\mathcal{S}))$$

hence

$$\hat{\Theta}; \hat{\Gamma} \equiv^{\widetilde{\Phi}} \mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$$

Moreover,

$$\hat{\Theta} \vdash \widetilde{\Phi}$$

This proves that $\mathsf{pv}' \overrightarrow{\oplus}\ \mathsf{pv}$ provides $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$ through $\widetilde{\Phi}$ at $\hat{\Theta}, \hat{\Gamma}$. The claim $\hat{\Theta} \Vdash \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\widetilde{\Phi}$ follows from Equations 24 and 31. The two remaining claims hold trivially. □

Proof of Lemma 4.37

Let Φ', Φ'', $\forall \mathsf{B}_{\mathrm{x},i}.\tau_{\mathrm{x},i}$, $\forall \mathsf{B}_{\mathrm{x}}.\tau_{\mathrm{x}}$, and z be defined as in the body of $\mathfrak{F}$. We have from the assumptions for $\mathrm{ps} = \textbf{struct}\ \mathrm{b}\ \textbf{end}$

$$\vdash \overrightarrow{\mathsf{pv}'} : \hat{\Theta}'; \hat{\Gamma}'$$

$$\hat{\Theta}' \vdash \Phi \tag{35}$$

$$\hat{\Theta}'; \hat{\Gamma}' \equiv^{\Phi} \mathcal{C}$$

$$\mathcal{C}' \vdash \mathrm{b} : \mathcal{S} \tag{36}$$

$$\mathcal{C}' := \mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i \in [n]}$$

and from the definition of $\mathfrak{F}$

$$\Omega = \mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi'' \tag{37}$$

By Lemma 4.29

$$\begin{gathered}
\vdash \textbf{class } \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a}\ \textbf{where} \ldots : \emptyset; \hat{\Gamma}'' \\
\hat{\Gamma}'' := \{\mathsf{z}^{\mathrm{F}',i,\mathrm{x}} \mapsto \forall \mathsf{B}_{\mathrm{x},i} \cup \{\mathsf{a}\}.\mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{a} \to \tau_{\mathrm{x},i} \mid i \in [n], \mathrm{x} \in \mathrm{Dom}(\mathcal{S}_i)\} \\
\vdash \textbf{class } \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}\ \textbf{where} \ldots : \hat{\Theta}''; \hat{\Gamma}''' \\
\hat{\Theta}'' := \{\forall\{\mathsf{a},\mathsf{b}\}.\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a}\} \\
\hat{\Gamma}''' := \{\mathsf{z}^{\mathrm{F},\mathrm{x}} \mapsto \forall \mathsf{B}_{\mathrm{x}} \cup \{\mathsf{a},\mathsf{b}\}.\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{b} \to \mathsf{a} \to \tau_{\mathrm{x}}
\end{gathered}$$

Clearly,

$$\begin{gathered}
\vdash \textbf{instance } \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\ \textbf{where} \ldots : \hat{\Theta}''' \\
\hat{\Theta}''' := \{\forall\{\mathsf{a}\}.\mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\} \cup \\
\{\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \Omega(\mathrm{t}) \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S})\}
\end{gathered} \tag{38}$$

Let us define

$$\begin{gathered}
\hat{\Theta} := \hat{\Theta}' \cup \hat{\Theta}'' \cup \hat{\Theta}''' \\
\hat{\Theta}_1 := \hat{\Theta} \cup \{\mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a}\} \\
\hat{\Gamma} := \hat{\Gamma}' \dot{\cup} \hat{\Gamma}'' \dot{\cup} \hat{\Gamma}''' \\
\hat{\Gamma}_1 := \hat{\Gamma}, \mathsf{z} \mapsto \mathsf{a}
\end{gathered}$$

Because $\hat{\Gamma}_1(\mathsf{z}^{\mathrm{F}',i,\mathrm{x}})$ is closed we can derive easily

$$\hat{\Theta}_1; \hat{\Gamma}_1 \vdash \mathsf{z}^{\mathrm{F}',i,\mathrm{x}}\ \mathsf{z} : \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi'$$

hence

$$\hat{\Theta}_1; \hat{\Gamma}_1 \equiv^{\Phi''} \mathcal{C}' \tag{39}$$

Moreover,

$$\hat{\Theta}_1 \vdash \mathsf{S}^{\mathrm{F}',i,\mathrm{t}}\ \mathsf{a}$$

hence

$$\hat{\Theta}_1 \vdash \Phi'' \tag{40}$$

Because $\hat{\Gamma}_1(\mathsf{z}^{\mathrm{F},\mathrm{x}})$ is closed it is easy to check that

$$\hat{\Theta}_1; \hat{\Gamma}_1 \vdash \mathsf{z}^{\mathrm{F},\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}})\ \mathsf{z} : [\mathsf{T}^{\mathrm{F}}/\mathsf{b}]\forall \mathsf{B}_{\mathrm{x}}.\tau_{\mathrm{x}}$$

so we have for all $\mathrm{x} \in \mathrm{Dom}(\mathcal{S})$

$$\hat{\Theta}_1; \hat{\Gamma}_1 \vdash \Phi''(\mathrm{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi'' \tag{41}$$

We can now apply Lemma 4.26 to $\mathfrak{S}_\mathrm{b}[\![\mathrm{b}]\!]\Phi''$ (Equations 36, 37, 39, 40, and 41 are exactly the assumptions of this lemma) and obtain

$$\hat{\Theta}_1 \vdash \Omega(\mathrm{t}) \quad (\mathrm{t} \in \mathrm{Dom}(\Omega) = \mathrm{Dom}(\mathcal{S})) \tag{42}$$

$$\Omega(\mathrm{t}) = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi'' \quad (\mathrm{t} \in \mathrm{Dom}(\mathcal{S})) \tag{43}$$

$$\hat{\Theta}_1; \hat{\Gamma}_1 \vdash \Omega(\mathrm{x}) : \mathfrak{T}_u[\![\mathcal{S}(\mathrm{x})]\!]\Phi'' \quad (\mathrm{x} \in \mathrm{Dom}(\mathcal{S})) \tag{44}$$

We now show that the class and instance definitions for $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$and C^{F} produced by $\mathfrak{F}$ pass rules $(\mathit{class}_{\mathit{check}})^+$ and $(\mathit{inst}_{\mathit{check}})^+$, respectively. Let us begin with the case for the class definitions. We have by Lemma 4.20

$$\hat{\Theta}_1 \vdash \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi'$$

hence with Lemma 4.29

$$\hat{\Theta} \vdash \textbf{class } \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a}\ \textbf{where} \ldots \tag{45}$$

With similar arguments we can show that

$$\hat{\Theta} \vdash \textbf{class } \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}\ \textbf{where} \ldots \tag{46}$$

Let us now check the instance definition for C^{F}. Using Equation 44, we can derive easily

$$\hat{\Theta}_1; \hat{\Gamma} \vdash \lambda\_.\lambda\mathsf{z}.\Omega(\mathrm{x}) : [\mathsf{T}^{\mathrm{F}}/\mathsf{b}](\forall \mathsf{B}_\mathrm{x}.\mathsf{b} \to \mathsf{a} \to \tau_\mathrm{x})$$

hence

$$\hat{\Theta}_1; \hat{\Gamma} \overset{\text{method}}{\vdash} \mathsf{z}^{\mathrm{F},\mathrm{x}} = \lambda\_.\lambda\mathsf{z}.\Omega(\mathrm{x})$$

Furthermore, we get by Equation 42

$$\hat{\Theta}_1; \mathsf{T}^{\mathrm{F}}\ \mathsf{a} \overset{\text{tdef}}{\vdash} \textbf{type } \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \Omega(\mathrm{t})$$

We have also

$$\hat{\Theta}_1 \setminus \{\forall\{\mathsf{a}\}.\mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\} \Vdash \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a}$$

hence

$$\hat{\Theta}; \hat{\Gamma} \vdash \textbf{instance } \mathsf{C}^{\mathrm{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\ \textbf{where} \ldots \tag{47}$$

Now we can show that all claims of the lemma hold: the first claim, namely $\vdash \mathsf{pv}' \overrightarrow{\oplus}\ \mathsf{pv} : \hat{\Theta}; \hat{\Gamma}$, follows by Lemma 4.35 and by Equations 45, 46, 47; the second claim $\hat{\Theta} \vdash \Phi$ follows by the weakening lemma 4.22 and Equation 35; the third claim is trivial; the fourth claim follows by definition of $\hat{\Gamma}$; the fifth claim follows from Equation 38; the sixth claim follows from Equations 38 and 43; the seventh claim holds by definition of $\hat{\Gamma}$; and the eighth claim is obvious. □

Proof of Theorem 4.38

The proof is by structural induction on prog.

CASE $\mathrm{prog} = \textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S}\rangle}; \mathrm{prog}'$: Follows directly from the induction hypothesis and Lemma 4.36.

CASE $\mathrm{prog} = \textbf{structure}\ \mathrm{X} = \mathrm{s}^{\langle \exists P'.\mathcal{S}'\rangle} :> \mathrm{S}^{\langle \wedge P.\mathcal{S}\rangle}; \mathrm{prog}'$: We get from the premises of rules ($prog_{str}$) and ($strexp_{sealed}$):

$$\begin{gathered}
\mathcal{C} \vdash \mathrm{s} : \exists P'.\mathcal{S}' \\
\mathcal{C} \vdash \mathrm{S} \triangleright \wedge P.\mathcal{S} \\
\mathcal{S}' \succcurlyeq \varphi(\mathcal{S}) \\
\mathrm{Dom}(\varphi) = P \\
\mathrm{X} \notin \mathrm{Dom}(\mathcal{C}) \\
\mathcal{C}, \mathrm{X} \mapsto \mathcal{S} \vdash \mathrm{prog}'
\end{gathered}$$

For $\mathsf{pv}'' = \mathfrak{X}[\![\textbf{structure}\ \mathrm{X}^\star = \mathrm{s}^{\langle \exists P'.\mathcal{S}'\rangle}]\!]\Phi$ we get by Lemma 4.36 that

$$\begin{gathered}
\mathsf{pv}' \oplus \mathsf{pv}'' \text{ provides } \mathcal{C}' \text{ through } \widetilde{\Phi} \text{ at } \hat{\Theta}, \hat{\Gamma} \\
\mathcal{C}' := \mathcal{C}, \mathrm{X}^\star \mapsto \mathcal{S} \\
\widetilde{\Phi} := \Phi \,\dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} \mid \alpha \in P', \mathrm{t} = \mathrm{pick}(\mathcal{S}', \alpha)\} \\
\dot\cup \{(\mathrm{X}^\star, \mathrm{y}) \mapsto \mathsf{z}^{\mathrm{X}^\star.\mathrm{y}}\ (\bot :: \mathsf{T}^{\mathrm{X}^\star}) \mid \mathrm{y} \in \mathrm{Dom}(\mathcal{S}')\} \\
\hat{\Theta} \Vdash \mathsf{C}^{\mathrm{X}^\star}\ \mathsf{T}^{\mathrm{X}^\star} \\
\hat{\Theta} \vdash \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} \quad (\mathrm{t} \in \mathrm{Dom}(\mathcal{S}')) \qquad (48) \\
\hat{\Theta} \Vdash \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} = \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\widetilde{\Phi} \quad (\mathrm{t} \in \mathrm{Dom}(\mathcal{S}')) \qquad (49)
\end{gathered}$$

For the rest of the proof of this case, we assume that Φ', Φ'', and $\forall \mathsf{B}_\mathrm{y}.\tau_\mathrm{y}$ are defined as in the relevant case of $\mathfrak{P}$. We get from Lemma 4.29

$$\begin{gathered}
\vdash \textbf{class}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{a}\ \textbf{where} \ldots : \emptyset; \hat{\Gamma}' \\
\hat{\Gamma}' := \{\mathsf{z}^{\mathrm{X.y}} \mapsto \forall \mathsf{B}_\mathrm{y} \cup \{\mathsf{a}\}.\mathsf{C}^{\mathrm{X}}\ \mathsf{a} \Rightarrow \mathsf{a} \to \tau_\mathrm{y} \mid \mathrm{y} \in \mathrm{Dom}(\mathcal{S})\}
\end{gathered}$$

Clearly,

$$\begin{gathered}
\vdash \textbf{instance}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where} \ldots : \hat{\Theta}' \\
\hat{\Theta}' := \{\mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\} \cup \{\mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \notin P\} \cup \\
\{\mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{T}^{\mathsf{S}^{\mathrm{X.t}}}\ \mathsf{T}^{\mathrm{X}} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})), \mathsf{T}^{\mathsf{S}^{\mathrm{X.t}}} \text{ fresh}\} \\
\cup \{\mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{S}^{\mathrm{X.t}'}\ \mathsf{T}^{\mathrm{X}} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t}' = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})), \mathrm{t}' \neq \mathrm{t}\}
\end{gathered}$$

Let us define

$$\hat{\Theta}_1 := \hat{\Theta} \cup \hat{\Theta}'$$
$$\hat{\Gamma}_1 := \hat{\Gamma} \dot{\cup} \hat{\Gamma}'$$

We get from $\hat{\Theta} \vdash \Phi$ that

$$\hat{\Theta}_1 \cup \{\mathsf{C}\ \mathsf{a}\} \vdash \Phi'$$

hence

$$\hat{\Theta}_1 \vdash \textbf{class}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{a}\ \textbf{where} \ldots$$

We now need to check that $\hat{\Theta}_1; \hat{\Gamma}_1 \vdash \textbf{instance}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where} \ldots$ holds. Let us first define

$$\hat{\Theta}_2 := \hat{\Theta}_1 \cup \{\mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))\} \quad (50)$$

$\mathsf{pv}' \oplus \mathsf{pv}''$ provides $\mathcal{C}'$ through $\widetilde{\Phi}$ at $\hat{\Theta}$,$\hat{\Gamma}$, hence $\hat{\Theta}; \hat{\Gamma} \equiv^{\widetilde{\Phi}} \mathcal{C}'$, so we get by the weakening lemma 4.22 for all $\mathrm{y} \in \mathrm{Dom}(\mathcal{S})$

$$\hat{\Theta}_2; \hat{\Gamma}_1 \vdash \underbrace{\mathsf{z}^{\mathrm{X}^\star.\mathrm{y}}\ (\bot :: \mathsf{T}^{\mathrm{X}^\star})}_{\mathsf{w}_\mathrm{y}} : \mathfrak{T}_v[\![\mathcal{S}'(\mathrm{y})]\!]\widetilde{\Phi}$$

We can conclude from $\mathcal{S}' \succcurlyeq \varphi(\mathcal{S})$ that

$$\varphi = \{\alpha \mapsto \mathcal{S}'(\mathrm{t}) \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\}$$

We get from Equation 49

$$\hat{\Theta}_2 \Vdash \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\widetilde{\Phi} \quad (\mathrm{t} \in \mathrm{Dom}(\mathcal{S}))$$

hence

$$\hat{\Theta}_2 \Vdash \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathfrak{T}_u[\![\varphi(\alpha)]\!]\widetilde{\Phi} \quad (\alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha))$$

We get $\hat{\Theta}_2 \vdash \widetilde{\Phi}$ from Equation 48, so we can use Lemma 4.31 to derive

$$\hat{\Theta}_2; \hat{\Gamma}_1 \vdash \mathsf{w}_\mathrm{y} : \mathfrak{T}_v[\![\mathcal{S}(\mathrm{y})]\!]\Phi'$$

Hence

$$\hat{\Theta}_2; \hat{\Gamma}_1 \vdash \lambda\_.\mathsf{w}_\mathrm{y} : [\mathsf{T}^{\mathrm{X}}/\mathsf{a}](\forall \mathsf{B}_\mathrm{y}.\mathsf{a} \to \tau_\mathrm{y})$$

This gives us

$$\hat{\Theta}_2; \hat{\Gamma}_1 \overset{\text{method}}{\vdash} \mathsf{z}^{\mathrm{X.y}} = \lambda\_.\mathsf{w}_\mathrm{y}$$

Moreover, we get from Equation 48

$$\hat{\Theta}_2; \mathsf{T}^{\mathrm{X}} \overset{\text{tdef}}{\vdash} \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star}$$
$$\hat{\Theta}_2; \mathsf{T}^{\mathrm{X}} \overset{\text{tdef}}{\vdash} \textbf{abstype}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star}$$

and because $(\mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}) \in \hat{\Theta}_2$

$$\hat{\Theta}_2; \mathsf{T}^{\mathrm{X}} \overset{\text{tdef}}{\vdash} \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{S}^{\mathrm{X.t'}}\ \mathsf{T}^{\mathrm{X}}$$

Now we can use rule $(inst_{check})^+$ to conclude that

$$\hat{\Theta}_1; \hat{\Gamma}_1 \vdash \textbf{instance}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where}\ \ldots$$

Finally, we define

$$\mathsf{pv}''' := \langle \textbf{data}\ \mathsf{T}^{\mathrm{X}}, \textbf{class}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{a}\ \textbf{where}\ \ldots, \textbf{instance}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where}\ \ldots \rangle$$

It is easy to verify that $\mathsf{pv}' \oplus \mathsf{pv}'' \oplus \mathsf{pv}'''$ provides $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$ through Φ', hence we can apply to induction hypothesis to $\mathfrak{P}[\![\mathrm{prog}']\!]\Phi'$ to derive

$$\vdash \mathsf{pv}' \oplus \mathsf{pv}$$

Case $\mathrm{prog} = \textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S} \rangle}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]}) = \mathrm{ps}^{\langle \mathcal{S} \rangle}; \mathrm{prog}'$: Follows directly from the induction hypothesis and Lemma 4.37.

Case $\mathrm{prog} = \textbf{functor}\ \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S} \rangle}(\overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]}) = \mathrm{ps}^{\langle \mathcal{S}' \rangle} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle}; \mathrm{prog}'$: We get from the premises of rules $(prog_{fun})$ and $(strexp_{sealed})$:

$$\mathcal{C} \overset{\text{funargs}}{\vdash} \overline{\mathrm{X_i} : \mathrm{S_i}}^{i \in [n]} \triangleright \forall Q.\overline{\mathcal{S}}^n$$
$$\underbrace{\mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i \in [n]}}_{=:\mathcal{C}'} \vdash \mathrm{ps} : \mathcal{S}'$$
$$\mathcal{C}' \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S}$$
$$\mathcal{S}' \succcurlyeq \varphi(\mathcal{S})$$
$$\mathrm{Dom}(\varphi) = P$$
$$\mathcal{C}, \mathrm{F} \mapsto \underbrace{\forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S}}_{=:\mathcal{F}} \vdash \mathrm{prog}'$$

In the following text we assume that Φ', $\forall \mathrm{B_x}.\tau_{\mathrm{x}}$, and z are defined as in the relevant case of $\mathfrak{P}$. Additionally, we define

$$\mathsf{pv}'' = \mathfrak{F}[\![\mathrm{ps}^{\langle \mathcal{S}' \rangle}]\!]\mathrm{F}^{\star\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}' \rangle}\overline{\mathrm{X}}^n \mathrm{F} \Phi$$
$$\Phi'' := \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i, \mathrm{t} \rangle = \mathrm{pick}(\overline{\mathcal{S}}, \alpha)\}$$

We obtain by Lemma 4.37 the following facts:

1. $\vdash \mathsf{pv}' \oplus \mathsf{pv} : \hat{\Theta}; \hat{\Gamma}$
2. $\hat{\Theta} \vdash \Phi$
3. For all $i \in [n], \mathrm{t} \in \mathrm{Dom}(\mathcal{S}_i)$: $\mathsf{S}^{\mathrm{F},i,\mathrm{t}}$ is an associated type synonym of type class $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$
4. For all $i \in [n], \mathrm{x} \in \mathrm{Dom}(\mathcal{S}_i)$: $\hat{\Gamma}(\mathsf{z}^{\mathrm{F},i,\mathrm{x}}) = \forall \mathsf{B}_{\mathrm{x},i} \dot{\cup} \{\mathsf{a}\}.\mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{a} \to \tau_{\mathrm{x},i}$ where $\forall \mathsf{B}_{\mathrm{x},i}.\tau_{\mathrm{x},i} = \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi''$
5. $(\forall\{\mathsf{a}\}.\mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}^\star}\ \mathsf{T}^{\mathrm{F}^\star}\ \mathsf{a}) \in \hat{\Theta}$ where $\mathsf{T}^{\mathrm{F}^\star}$ is an user-defined type constructor of kind 0
6. For all $\mathrm{t} \in \mathrm{Dom}(\mathcal{S}')$: $\mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}$ is an associated type synonym of type class $\mathsf{C}^{\mathrm{F}^\star}$ and $(\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}\ \mathsf{T}^{\mathrm{F}^\star}\ \mathsf{a} = \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\Phi'') \in \hat{\Theta}$
7. For all $\mathrm{x} \in \mathrm{Dom}(\mathcal{S}')$: $\hat{\Gamma}(\mathsf{z}^{\mathrm{F}^\star,\mathrm{x}}) = \forall \mathsf{B}_{\mathrm{x},\star} \dot{\cup} \{\mathsf{a},\mathsf{b}\}.\mathsf{C}^{\mathrm{F}^\star}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{b} \to \mathsf{a} \to \tau_{\mathrm{x},\star}$, $\forall \mathsf{B}_{\mathrm{x},\star}.\tau_{\mathrm{x},\star} = \mathfrak{T}_v[\![\mathcal{S}'(\mathrm{x})]\!]\Phi''$
8. $\mathrm{Sup}(\hat{\Theta}, \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}) = \emptyset, \mathrm{Sup}(\hat{\Theta}, \mathsf{C}^{\mathrm{F}^\star}\ \mathsf{b}\ \mathsf{a}) = \{\mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}\}$

We get with Lemma 4.29

$$\vdash \textbf{class}\ \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}\ \textbf{where} \ldots : \hat{\Theta}'; \hat{\Gamma}'$$
$$\hat{\Theta}' := \{\forall\{\mathsf{a},\mathsf{b}\}.\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}\}$$
$$\hat{\Gamma}' := \{\mathsf{z}^{\mathrm{F},\mathrm{x}} \mapsto \forall \mathsf{B}_{\mathrm{x}} \cup \{\mathsf{a},\mathsf{b}\}.\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{b} \to \mathsf{a} \to \tau_{\mathrm{x}} \mid \mathrm{x} \in \mathrm{Dom}(\mathcal{S})\}$$

Moreover,

$$\vdash \textbf{instance}\ \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\ \textbf{where} \ldots : \hat{\Theta}''$$
$$\hat{\Theta}'' := \{\forall\{\mathsf{a}\}.\mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\} \cup$$
$$\{\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}\ \mathsf{T}^{\mathrm{F}^\star}\ \mathsf{a} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \notin P\} \cup$$
$$\{\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{T}^{\mathsf{S}^{\mathrm{F},\mathrm{t}}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})), \mathsf{T}^{\mathsf{S}^{\mathrm{F},\mathrm{t}}}\ \mathrm{fresh}\} \cup$$
$$\{\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{S}^{\mathrm{F},\mathrm{t}'}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t}' = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})), \mathrm{t}' \neq \mathrm{t}\}$$

Let us define

$$\hat{\Theta}_1 := \hat{\Theta} \cup \hat{\Theta}' \cup \hat{\Theta}''$$
$$\hat{\Gamma}_1 := \hat{\Gamma} \dot{\cup} \hat{\Gamma}'$$

We get from $\hat{\Theta} \vdash \Phi$ that

$$\Theta_1 \cup \{\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}\} \vdash \Phi'$$

hence

$$\hat{\Theta}_1 \vdash \textbf{class}\ \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}\ \textbf{where} \ldots$$

We now need to check that $\hat{\Theta}_1; \hat{\Gamma}_1 \vdash \textbf{instance}\ \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\ \textbf{where} \ldots$ holds. Let us first define

$$\hat{\Theta}_2 := \hat{\Theta}_1 \cup \{\mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}\} \cup$$
$$\{\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}\ \mathsf{T}^{\mathrm{F}^\star}\ \mathsf{a} \mid \mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))\}$$
$$\hat{\Gamma}_2 := \hat{\Gamma}_1, \mathsf{z} \mapsto \mathsf{a}$$

Facts 5 and 7 obtained by Lemma 4.37 give us

$$\hat{\Theta}_2; \hat{\Gamma}_2 \vdash \underbrace{\mathsf{z}^{\mathrm{F}^\star,\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}^\star})\ \mathsf{z}}_{=:\ \mathsf{w}_\mathrm{x}} : \mathfrak{T}_v[\![\mathcal{S}'(\mathrm{x})]\!]\Phi''$$

We get from fact 6 and by definition of $\hat{\Theta}_2$

$$\hat{\Theta}_2 \Vdash \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\Phi'' \quad (\mathrm{t} \in \mathrm{Dom}(\mathcal{S}))$$

$\mathcal{S}' \succcurlyeq \varphi(\mathcal{S})$ yields

$$\varphi = \{\alpha \mapsto \mathcal{S}'(\mathrm{t}) \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\} \tag{51}$$

hence

$$\hat{\Theta}_2 \Vdash \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathfrak{T}_u[\![\varphi(\alpha)]\!]\Phi'' \quad (\alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha))$$

Clearly, $\hat{\Theta}_2 \vdash \Phi''$, so we can now use Lemma 4.31 to conclude that

$$\hat{\Theta}_2; \hat{\Gamma}_2 \vdash \mathsf{w}_\mathrm{x} : \mathfrak{T}_u[\![\mathcal{S}(\mathrm{x})]\!]\Phi'''$$
$$\Phi''' := \Phi'' \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\}$$

We now get

$$\hat{\Theta}_2; \hat{\Gamma}_1 \vdash \lambda\_.\lambda\mathsf{z}.\mathsf{w}_\mathrm{x} : [\mathsf{T}^{\mathrm{F}}/\mathsf{b}](\forall \mathsf{B}_\mathrm{x}.\mathsf{b} \to \mathsf{a} \to \tau_\mathrm{x})$$

This gives us

$$\hat{\Theta}_2; \hat{\Gamma}_1 \overset{\text{method}}{\vdash} \mathsf{z}^{\mathrm{F},\mathrm{x}} = \lambda\_.\lambda\mathsf{z}.\mathsf{w}_\mathrm{x}$$

Furthermore, we get from fact 5 that

$$\hat{\Theta}_2; \mathsf{T}^{\mathrm{F}}\mathsf{a} \overset{\text{tdef}}{\vdash} \textbf{type}\ \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}\ \mathsf{T}^{\mathrm{F}^\star}\ \mathsf{a}$$
$$\hat{\Theta}_2; \mathsf{T}^{\mathrm{F}}\mathsf{a} \overset{\text{tdef}}{\vdash} \textbf{abstype}\ \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}\ \mathsf{T}^{\mathrm{F}^\star}\ \mathsf{a}$$

Moreover,

$$\hat{\Theta}_2; \mathsf{T}^{\mathrm{F}}\mathsf{a} \overset{\text{tdef}}{\vdash} \textbf{type}\ \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{S}^{\mathrm{F},\mathrm{t}'}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}$$

Now we use rule $(inst_{check})^+$ to derive

$$\hat{\Theta}_1; \hat{\Gamma}_1 \vdash \textbf{instance}\ \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\ \textbf{where}\ldots$$

We now define

$$\begin{aligned}\mathsf{pv}''' = \langle &\textbf{data}\ \mathsf{T}^{\mathrm{F}}, \textbf{class}\ \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}\ \textbf{where}\ldots,\\ &\textbf{instance}\ \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}\ \textbf{where}\ldots\rangle\end{aligned}$$

We get by Lemma 4.35

$$\vdash \mathsf{pv}' \overrightarrow{\oplus}\ \mathsf{pv}'' \overrightarrow{\oplus}\ \mathsf{pv}''' : \hat{\Theta}_1; \hat{\Gamma}_1$$

From $\hat{\Theta} \subseteq \hat{\Theta}_1$ and Lemma 4.22 we get

$$\hat{\Theta}_1 \vdash \Phi$$

To prove that $\mathsf{pv}' \oplus \mathsf{pv}'' \oplus \mathsf{pv}'''$ provides $\mathcal{C}, \mathrm{F} \mapsto \mathcal{F}$ through Φ at $\hat{\Theta}_1, \hat{\Gamma}_1$, we still need to show $\hat{\Theta}_1; \hat{\Gamma}_1 \equiv^{\Phi} \mathcal{C}, \mathrm{F} \mapsto \mathcal{F}$. But condition 1 of Definition 4.18 holds by Lemmata 4.3 and 4.15, conditions 2 and 3 hold trivially, conditions 4 – 6 hold by the weakening lemma 4.22, and condition 7 needs to be checked only for the newly introduced F. Conditions 7.1 – 7.3, 7.5, and 7.6 are straightforward to check, the only difficulty is verifying condition 7.4, namely that $\hat{\Theta}_1 \Vdash [\tau/\mathsf{a}](\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi''')$ holds for all $\mathrm{t} \in \mathrm{Dom}(\mathcal{S})$ and all τ. We proceed by case analysis:

CASE $\mathcal{S}(\mathrm{t}) \in P$: If $\mathrm{t} = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))$, then $\mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi''' = \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}$, so the claim holds trivially. If $\mathrm{t} \neq \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))$, then $\mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi''' = \mathsf{S}^{\mathrm{F},\mathrm{t}'}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}$ with $\mathrm{t}' = \mathrm{pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))$. The definition of $\hat{\Theta}'' \subseteq \hat{\Theta}_1$ gives us $(\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{S}^{\mathrm{F},\mathrm{t}'}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}) \in \hat{\Theta}_1$, so the claim follows now by rule $(eqdef_{entail})^+$, $\mathfrak{T}_u[\![\mathcal{S}(\mathrm{t}')]\!]\Phi''' = \mathsf{S}^{\mathrm{F},\mathrm{t}'}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}$, and $\mathcal{S}(\mathrm{t}') = \mathcal{S}(\mathrm{t})$.

CASE $\mathcal{S}(\mathrm{t}) \notin P$: Let us define

$$\begin{aligned}\Phi_1 &:= \Phi \dot{\cup} \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \tau \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \mathrm{pick}(\overline{\mathcal{S}}, \alpha)\}\\ \Phi_2 &:= \Phi_1 \dot{\cup} \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \tau \mid \alpha \in P, \mathrm{t} = \mathrm{pick}(\mathcal{S}, \alpha)\}\end{aligned}$$

Because $\mathrm{FV}^{\mathsf{a}}(\Phi) = \emptyset$ we have

$$\begin{aligned}\Phi_1 &= [\tau/\mathsf{a}]\Phi''\\ \Phi_2 &= [\tau/\mathsf{a}]\Phi'''\end{aligned}$$

By Equation 51 we have $\mathfrak{T}_u[\![\varphi(\alpha)]\!]\Phi_2 = \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\Phi_2$ for all $\alpha \in \mathrm{Dom}(\varphi)$, hence we get by Lemma 4.27

$$\hat{\Theta}_1 \Vdash \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi_2 = \underbrace{\mathfrak{T}_u[\![\varphi(\mathcal{S}(\mathrm{t}))]\!]\Phi_2}_{=\mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\Phi_1}$$

We have from fact 6 and the definition of $\hat{\Theta}''$

$$(\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}\ \mathsf{T}^{\mathrm{F}^\star}\ \mathsf{a} = \mathfrak{T}_u[\![\mathcal{S}'(\mathrm{t})]\!]\Phi'') \in \hat{\Theta}_1$$
$$(\forall\{\mathsf{a}\}.\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}\ \mathsf{T}^{\mathrm{F}^\star}\ \mathsf{a}) \in \hat{\Theta}_1$$

Now the claim follows with rules $(\mathit{eqdef}_{\mathit{entail}})^+$ and $(\mathit{eqtrans}_{\mathit{entail}})^+$. Note that $\mathrm{FV}^{'a}(\mathcal{S}') = \mathrm{FV}^{'a}(\mathcal{S}) = \emptyset$.

Now we have proved that $\mathsf{pv}' \oplus \mathsf{pv}'' \oplus \mathsf{pv}'''$ provides $\mathcal{C}, \mathrm{F} \mapsto \mathcal{F}$ through Φ, so we can apply the induction hypothesis to derive

$$\mathsf{pv}' \overrightarrow{\oplus}\ \mathsf{pv} \qquad \square$$

Proofs for Section 5.3.1

Proof of Lemma 5.8

We proceed by induction on the derivation of $\hat{\Theta} \Vdash \pi$. Let $\pi = \mathsf{C}\ \tau$.

CASE $(\mathit{elem}_{\mathit{entail}})$: Trivial because $\hat{\Theta}^l \subseteq \mathrm{Dom}(\Theta^l)$.

CASE $(\mathit{inst}_{\mathit{entail}})$: We get from the premise of this rule

$$(\forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[r]} \Rightarrow \mathsf{C}\ \tau') \in \hat{\Theta}^i \subseteq \mathrm{Dom}(\Theta^i)$$
$$\tau = \psi(\tau')$$
$$\hat{\Theta} \Vdash \mathsf{C}_i\ \psi(\mathsf{a}_i)$$

Θ is well-formed, hence

$$\mathrm{FV}^{\mathsf{a}}(\psi(\mathsf{a}_i)) \subseteq \mathrm{FV}^{\mathsf{a}}(\psi(\tau')) = \mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$$

We can now apply the induction hypothesis to derive

$$\Delta; \Sigma; \Theta \Vdash \mathsf{C}_i\ \psi(\mathsf{a}_i) \rightsquigarrow \mathrm{e}_i$$

Lemma 5.7 gives us

$$\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$$
$$\Sigma \vdash \psi(\mathsf{a}_i) \rightsquigarrow \mathrm{u}_i$$
$$\Sigma \vdash \psi(\mathsf{b}) \rightsquigarrow \mathrm{u}_\mathsf{b}$$

Now the claim follows by rule $(\mathit{inst}_{\mathit{entail}})^t$.

CASE ($super_{entail}$): We get from the premise of this rule

$$(\forall \mathsf{A}.\mathsf{C}^{\mathsf{sub}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathsf{sup}}\ \mathsf{a}) \in \hat{\Theta}^s \subseteq \Theta^s$$

By the induction hypothesis, we obtain

$$\Delta; \Sigma; \Theta \Vdash \mathsf{C}^{\mathsf{sub}}\ \tau \rightsquigarrow \mathrm{e}$$

$\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$, so by Lemma 5.7

$$\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$$

Applying rule $(super_{entail})^t$ finishes this proof. □

Proof of Lemma 5.11

We proof Lemma 5.11 by structural induction over w and make use of the syntax–directed form of the typing rules.

CASE $\mathsf{w} = \mathsf{z}$: The last typing rule used in the derivation of $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{z} : \tau$ must be (var). The premise of this rule gives us

$$\Gamma(\mathsf{z}) = \hat{\Gamma}(\mathsf{z}) = \forall \mathsf{A}.\overline{\pi}^n \Rightarrow \tau'$$
$$\hat{\Theta} \Vdash \psi(\pi_i)$$

Γ is unambiguous, hence

$$\mathrm{FV}^{\mathsf{a}}(\psi(\pi_i)) \subseteq \mathrm{FV}^{\mathsf{a}}(\tau) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma) \subseteq \mathrm{Dom}(\Sigma)$$

Now by Lemma 5.8

$$\Delta; \Sigma; \Theta \Vdash \psi(\pi_i) \rightsquigarrow \mathrm{e}_i$$

Applying rule $(var)^t$ yields

$$\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \tau.$$

CASE $\mathsf{w} = \mathsf{m}$: This case is very similar to the first case. We use rules $(method)$ and $(method)^t$ instead of (var) and $(var)^t$. $\Sigma \vdash \psi(\tau_{\mathsf{b}}) \rightsquigarrow \mathrm{u}$ follows from Lemma 5.7.

CASE $\mathsf{w} = \mathsf{w}_1\ \mathsf{w}_2$: The last typing rule used must be $(\rightarrow E)$. From the premise we get

$$\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_1 : \tau' \rightarrow \tau$$
$$\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w}_2 : \tau'$$

We cannot apply the induction hypothesis directly because it might be that

$$\mathsf{A} := \mathrm{FV}^{\mathsf{a}}(\tau') \setminus \mathrm{Dom}(\Sigma) \neq \emptyset$$

Therefore, we define a substitution

$$\psi =: [\overline{\mathsf{Int}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}}]$$

and then use the substitution lemma 5.9 to obtain

$$\hat{\Theta};\hat{\Gamma} \vdash \mathsf{w}_1 : \psi(\tau') \to \tau$$
$$\hat{\Theta};\hat{\Gamma} \vdash \mathsf{w}_2 : \psi(\tau')$$

Note that ψ does not affect $\hat{\Theta}$, $\hat{\Gamma}$, and τ because

$$\mathrm{FV}^{\mathsf{a}}(\tau) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma) \cup \mathrm{FV}^{\mathsf{a}}(\Theta) \subseteq \mathrm{Dom}(\Sigma)$$

Now we can use the induction hypothesis and rule $(\to E)^t$ to derive the desired result.

CASE $\mathsf{w} = \lambda\mathsf{z}.\mathsf{w}'$: Follows directly from the induction hypothesis and rule $(\to I)^t$.

CASE $\mathsf{w} = (\textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2)$: The last rule in the derivation must be (let). We get from the premise of this rule

$$\hat{\Theta}' = (\hat{\Theta}^s, \hat{\Theta}^i, \{\overline{\pi}\})$$
$$\hat{\Theta}';\hat{\Gamma} \vdash \mathsf{w}_1 : \tau' \tag{52}$$
$$\sigma = \widehat{\mathrm{Gen}}(\hat{\Theta}', \hat{\Gamma}, \tau') \text{ unambiguous}$$
$$\hat{\Theta};\hat{\Gamma}, \mathsf{z} \mapsto \sigma \vdash \mathsf{w}_2 : \tau \tag{53}$$

Using constraint strengthening (Lemma 5.10), we can safely assume that

$$\mathrm{CS}(\overline{\pi}) \subseteq \mathrm{Dom}(\Delta)$$

By definition of Σ' in the premise of rule $(let)^t$

$$\mathrm{FV}^{\mathsf{a}}(\overline{\pi}) \cup \mathrm{FV}^{\mathsf{a}}(\tau') \subseteq \mathrm{Dom}(\Sigma')$$

Now we can apply the induction hypothesis to Equation 52 and get

$$\Delta;\Sigma';(\Theta^s,\Theta^i,\{\overline{c_i \mapsto \pi_i}^{i\in[n]}\});\Gamma \vdash \mathsf{w}_1 \rightsquigarrow \mathrm{e}_1 : \tau'$$

$\hat{\Theta}' = \Theta'$ and $\hat{\Gamma} = \Gamma$, hence

$$\widehat{\mathrm{Gen}}(\hat{\Theta}', \hat{\Gamma}, \tau') = \mathrm{Gen}(\Theta', \Gamma, \tau')$$

Clearly, $\mathrm{FV}^{\mathsf{a}}(\sigma) \subseteq \mathrm{Dom}(\Sigma)$, so by Lemma 5.7

$$\Delta;\Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$$

Applying the induction hypothesis to Equation 53 yields

$$\Delta;\Sigma;\Theta;\Gamma, \mathsf{z} \mapsto \sigma \vdash \mathsf{w}_2 \rightsquigarrow \mathrm{e}_2 : \tau$$

Rule $(let)^t$ now gives the desired result. □

Proof of Theorem 5.12

We know that $\vdash \mathsf{pgm}$ holds. The last premise of rule $(prog)^t$, $\Delta; \emptyset; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \mathsf{Int}$, follows from Lemma 5.11 and the premise $\hat{\Theta}; \hat{\Gamma} \vdash \mathsf{w} : \mathsf{Int}$ of rule $(prog)$.

We can establish a similar correspondence for the other premises of rule $(prog)^t$ even though the premises in the translation rules for instance and class definitions are stronger than the premises in the original typing rules. The stronger premises in the translation rules result from using the type translation judgments $\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$ and $\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$ (in rules $(inst_{check})^t$, $(inst_{check\text{-}method})^t$, and $(class)^t$), from accessing the class environment Δ (in $(inst_{check})^t$ and $(class)^t$), and from accessing the instance part of the constraint environment Θ^i (in $(inst_{check})^t$). However, these additional premises are fulfilled whenever the remaining premises of a rule are fulfilled:

- All usages of the type translation judgments are successful by definition of the relevant type environment Σ and because $\mathrm{FV}^{\mathsf{a}}(\Gamma) = \emptyset$ for all Γ produced by rule $(class)^t$.
- Every access to the class environment Δ is well-defined because the well-formedness of pgm implies that classes can be used only after their definition.
- The access to $\hat{\Theta}^i$ in rule $(inst_{check})^t$ is well-defined because there is also an application of rule $(inst_{collect})^t$ to the same instance definition in the derivation of $\vdash \mathsf{pgm} \rightsquigarrow \mathrm{prog}$.

Furthermore, the existence of an entailment or typing derivation in the translation system given a derivation in the original system is guaranteed by Lemma 5.8 and Lemma 5.11, respectively. □

Proofs for Section 5.3.2

Proof of Lemma 5.20

We proof the lemma by induction over the structure of τ.

CASE $\tau = \mathsf{a}$: If $\mathsf{a} \in \mathrm{Dom}(\mathcal{T})$, then $\mathcal{T} \dot\cup \mathcal{T}' \vdash \mathsf{a} \rightsquigarrow \mathcal{T}(\mathsf{a}) =: {}'a$. Because $\mathsf{a} \in \mathrm{Dom}(\psi)$, and $\mathrm{FV}^{\mathsf{a}}(\psi) \subseteq \mathcal{T}'$, we have $\mathcal{T}' \vdash \psi(\mathsf{a}) \rightsquigarrow \phi({}'a)$ by definition of ϕ and Corollary 5.18.

If $\mathsf{a} \notin \mathrm{Dom}(\mathcal{T})$, then $\mathsf{a} \in \mathrm{Dom}(\mathcal{T}')$ by Definition 5.16, thus $\mathcal{T} \dot\cup \mathcal{T}' \vdash \mathsf{a} \rightsquigarrow \mathcal{T}'(\mathsf{a})$. Furthermore, we have $\psi(\mathsf{a}) = \mathsf{a}$, and $\phi(\mathcal{T}'(\mathsf{a})) = \mathcal{T}'(\mathsf{a})$ because $\mathrm{FV}^{{}'a}(\mathcal{T}') \cap \mathrm{Dom}(\phi) = \emptyset$. Hence $\mathcal{T}' \vdash \psi(\mathsf{a}) \rightsquigarrow \phi(\mathcal{T}'(\mathsf{a}))$.

CASE $\tau = \mathsf{T}^\kappa\ \overline{\tau}^\kappa$: Directly from the induction hypothesis. □

Proof of Lemma 5.34

We prove the lemma by rule induction.

CASE $(\mathit{entailElem})^t$: Obvious from the compatibility of $\mathcal{C}$ with Θ.

CASE $(\mathit{inst}_{\mathit{entail}})^t$: We have from the premise of the rule

$$\begin{gathered}\Sigma \vdash \tau \rightsquigarrow \mathrm{u} \\ \Sigma \vdash \psi(\mathsf{a}_i) \rightsquigarrow \mathrm{u}_i \quad \text{for } i \in [r] \\ \Sigma \vdash \psi(\mathsf{b}) \rightsquigarrow \mathrm{u}_\mathsf{b} \quad \text{for } \mathrm{b} \in \mathrm{B}, \text{ where B is defined as in the premise}\end{gathered}$$

Let us define

$$\mathrm{s}_\mathrm{B} := \mathbf{struct}\ \overline{\mathbf{type}\ \mathrm{t}^\mathsf{b} = \mathrm{u}_\mathsf{b}}^{\mathrm{b}\in\mathrm{B}}\ \mathbf{end}$$

Firstly, we prove that the subexpression $\mathrm{F}(\overline{\mathrm{X}}^r, \mathrm{s}_\mathrm{B})$ of the result expression e in the conclusion of the rule has type $\mathcal{S}_\Delta(\mathsf{C}, u)$ in an appropriate context. Lemma 5.14, 5.15, and the compatibility of $\mathcal{C}$ with Σ give us

$$\mathcal{C} \vdash \mathrm{u} \triangleright u \tag{54}$$
$$\mathcal{C} \vdash \mathrm{u}_i \triangleright u_i \tag{55}$$
$$\mathcal{C} \vdash \mathrm{u}_\mathsf{b} \triangleright u_\mathsf{b} \tag{56}$$

We now define

$$\mathcal{C}' := \mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_\Delta(\mathsf{C}_i, u_i)}^{i\in[r]}$$

With Equation 56, Lemma 5.31, and the rules $(\mathit{strexp}_{\mathit{struct}})$and (strb_t), we get

$$\mathcal{C}' \vdash \mathrm{s}_\mathrm{B} : \underbrace{\{\mathrm{t}^\mathsf{b} \mapsto \mathrm{u}_\mathsf{b} \mid \mathrm{b} \in \mathrm{B}\}}_{=:\ \mathcal{S}_\mathrm{B}}$$

$\mathcal{C}(\mathrm{F}) = \forall P.\overline{\mathcal{S}}^{r+1} \to \mathcal{S}$ follows from the compatibility of $\mathcal{C}$; P, $\overline{\mathcal{S}}^{r+1}$, and $\mathcal{S}$ are supposed to be as in Definition 5.33. Our goal is to use rule $(\mathit{strexp}_{\mathit{fapp}})$ to derive $\mathcal{C}' \vdash \mathrm{F}(\overline{\mathrm{X}}^r, \mathrm{s}_\mathrm{B}) : \mathcal{S}_\Delta(\mathsf{C}, u)$. We first have to show that $\mathcal{C}'(\mathrm{X}_i) \succcurlyeq \varphi(\mathcal{S}_i)$ for $i \in [r]$, and that $\mathcal{S}_\mathrm{B} \succcurlyeq \varphi(\mathcal{S}_{r+1})$, where φ is a substitution with $\mathrm{Dom}(\varphi) = P$. We define φ as follows:

$$\varphi := \{\mathcal{S}_i(\mathrm{t}) \mapsto u_i \mid i \in [r]\} \dot\cup \{\mathcal{S}_{r+1}(\mathrm{t}^\mathsf{b}) \mapsto u_\mathsf{b} \mid \mathrm{b} \in \mathrm{B}\}$$

Note that $\mathcal{S}_i(\mathrm{t}) = \mathcal{S}_j(\mathrm{t})$ implies $u_i = u_j$, and that $\mathrm{Dom}(\varphi) = P$. This follows from the compatibility of $\mathcal{C}$. Now we have with Lemma 5.25

$$\begin{aligned}\mathcal{C}'(\mathrm{X}_i) = \mathcal{S}_\Delta(\mathsf{C}_i, u_i) = \mathcal{S}_\Delta(\mathsf{C}_i, \varphi(\mathcal{S}_i(\mathrm{t}))) = \varphi(\mathcal{S}_i) \\ \mathcal{S}_\mathrm{B} = \{\mathrm{t}^\mathsf{b} \mapsto \varphi(\mathcal{S}_{r+1}(\mathrm{t}^\mathsf{b})) \mid \mathrm{b} \in \mathrm{B}\} = \varphi(\mathcal{S}_{r+1})\end{aligned}$$

and so we can use $(\mathit{strexp}_{\mathit{fapp}})$ to conclude that

$$\mathcal{C}' \vdash \mathrm{F}(\overline{\mathrm{X}}^r, \mathrm{s_B}) : \varphi(\mathcal{S})$$

We still need to show that $\varphi(\mathcal{S}) = \mathcal{S}_\Delta(\mathsf{C}, u)$. For

$$\mathcal{T} := \{\mathsf{a}_i \mapsto \mathcal{S}_i(\mathrm{t}) \mid i \in [r]\} \dot\cup \{\mathsf{b} \mapsto \mathcal{S}_{r+1}(\mathrm{t}^{\mathsf{b}}) \mid \mathrm{b} \in \mathrm{B}\}$$
$$\mathcal{T} \vdash \tau' \rightsquigarrow u'$$

we get from Definition 5.33 and Lemma 5.25

$$\varphi(\mathcal{S}) = \varphi(\mathcal{S}_\Delta(\mathsf{C}, u')) = \mathcal{S}_\Delta(\mathsf{C}, \varphi(u'))$$

and so we only need to prove that $u = \varphi(u')$. We now write Equations 54, 55, 56, and the substitution φ in a slightly different way:

$$\mathcal{T}' := \{\mathsf{a} \mapsto u_{\mathsf{a}} \mid \mathsf{a} \in \mathrm{Dom}(\Sigma), \Sigma; \mathcal{C} \vdash \mathsf{a} \rightsquigarrow u_{\mathsf{a}}\}$$
$$\mathcal{T}' \vdash \psi(\tau') \rightsquigarrow u$$
$$\mathcal{T}' \vdash \psi(\mathsf{a}_i) \rightsquigarrow u_i$$
$$\mathcal{T}' \vdash \psi(\mathsf{b}) \rightsquigarrow u_{\mathsf{b}}$$
$$\varphi = \{\alpha \mapsto u_\alpha \mid \alpha \in \mathrm{Img}(\mathcal{T}), \mathcal{T}' \vdash \psi(\mathcal{T}^{-1}(\alpha)) \rightsquigarrow u_\alpha\}$$

Now we can apply Lemma 5.22 to derive

$$\mathcal{T}' \vdash \psi(\tau') \rightsquigarrow \varphi(u')$$

The uniqueness of direct semantic type translations (Corollary 5.18) gives us $u = \varphi(u')$, so that we have now proved

$$\mathcal{C}' \vdash \mathrm{F}(\overline{\mathrm{X}}^r, \mathrm{s_B}) : \mathcal{S}_\Delta(\mathsf{C}, u) \tag{57}$$

The second step is to show that $\mathcal{C}' \vdash \mathrm{e}_{\mathit{pack}} : <\mathcal{S}_\Delta(\mathsf{C}, u)>$ where we define $\mathrm{e}_{\mathit{pack}}$ as follows:

$$\mathrm{e}_{\mathit{pack}} := \mathbf{pack}\ \mathrm{F}(\overline{\mathrm{X}}^r, \mathrm{s_B})\ \mathbf{as}\ (\Delta(\mathsf{C})\ \mathbf{where\ type}\ \mathrm{t} = \mathrm{u})$$

Using Equation 54, Lemma 5.26 and 5.31, we get

$$\mathcal{C}' \vdash \Delta(\mathsf{C})\ \mathbf{where\ type}\ \mathrm{t} = \mathrm{u} \triangleright \mathcal{S}_\Delta(\mathsf{C}, u)$$

and so rule $(\mathit{exp}_{\mathit{pack}})^+$ and Equation 57 allow us to derive

$$\mathcal{C}' \vdash \mathrm{e}_{\mathit{pack}} : <\mathcal{S}_\Delta(\mathsf{C}, u)> \tag{58}$$

The last part of the proof for the case $(\mathit{inst}_{\mathit{entail}})^t$ is easy: We just have to make sure that typing the nested **open** subexpressions of the result $\overline{\mathbf{open} \ldots \mathbf{in}}\ \mathrm{e}_{\mathit{pack}}$ properly builds up the context $\mathcal{C}'$. For $i \in [r]$, we have from the premise of the rule

$$\Delta; \Sigma; \Theta \Vdash \mathsf{C}_i\ \psi(\mathsf{a}_i) \rightsquigarrow \mathrm{e}_i$$

so that we can use the induction hypothesis, the weakening lemma 5.31, and Lemma 5.26 to conclude that

$$\mathcal{C}^i \vdash \mathrm{e}_i : <\mathcal{S}_\Delta(\mathsf{C}_i, u_i)>$$
$$\mathcal{C}^i \vdash \Delta(\mathsf{C}_i) \textbf{ where type } \mathrm{t} = \mathrm{u}_i \triangleright \mathcal{S}_\Delta(\mathsf{C}_i, u_i)$$

where $\mathcal{C}^i$ is defined as

$$\mathcal{C}^i := \mathcal{C}, \overline{\mathrm{X}_j \mapsto <\mathcal{S}_\Delta(\mathsf{C}_j, u_j)>}^{j\in[i-1]}$$

Now repeated applications of rule $(exp_{open})^+$ and Equation 58 allow us to conclude that $\mathcal{C} \vdash \mathrm{e} : <\mathcal{S}_\Delta(\mathsf{C}, u)>$.

CASE $(entailSuper)^t$: We get from the premise of the rule:

$$\Delta; \Sigma; \Theta \Vdash \mathsf{C}^{\mathsf{sub}}\ \tau \rightsquigarrow \mathrm{e}$$
$$\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$$

Using the induction hypothesis, we obtain

$$\mathcal{C} \vdash \mathrm{e} : \overbrace{\mathcal{S}_\Delta(\mathsf{C}^{\mathsf{sub}}, u)}^{=:\mathcal{S}}$$
$$\mathcal{C} \vdash \mathrm{u} \triangleright u$$

Now with Lemma 5.26

$$\mathcal{C} \vdash \Delta(\mathsf{C}^{\mathsf{sub}}) \textbf{ where type } \mathrm{t} = \mathrm{u} \triangleright \mathcal{S}$$

Because of Lemma 5.27, we have

$$\mathrm{x}^{\mathsf{C}^{\mathsf{sup}}} \in \mathrm{Dom}(\mathcal{S})$$
$$\mathcal{S}(\mathrm{x}^{\mathsf{C}^{\mathsf{sup}}}) = <\mathcal{S}_\Delta(\mathsf{C}^{\mathsf{sup}}, u)>$$

The claim now follows with rules $(exp_{open})^+$ and (exp_{mod2}). □

Proof of Lemma 5.38

We proof the lemma by structural induction over w.

CASE $\mathsf{w} = \mathsf{z}$: We get from the premise of rule $(var)^t$:

$$\Gamma(\mathsf{z}) = \forall \mathsf{A}.\overline{\pi}^n \Rightarrow \tau'$$
$$\widetilde{\psi} = [\overline{\tau_\mathsf{a}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}}]$$
$$\widetilde{\psi}(\tau') = \tau$$
$$\Delta; \Sigma; \Theta \Vdash \widetilde{\psi}(\pi_i) \rightsquigarrow \mathrm{e}_i$$

We can safely assume that

$$\mathsf{A} \cap \mathrm{FV}^{\mathsf{a}}(\psi) = \emptyset \tag{59}$$
$$\mathsf{A} \cap \mathrm{Dom}(\psi) = \emptyset \tag{60}$$
$$\mathsf{A} \cap \mathrm{FV}^{\mathsf{a}}(\widetilde{\psi}) = \emptyset$$

Let us define

$$\mathsf{A}' := \mathrm{FV}^{\mathsf{a}}(\overline{\pi}^{n}) \cap \mathsf{A}$$
$$\widetilde{\psi}' := [\overline{\tau_{\mathsf{a}}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}'}]$$

Then we have

$$\Delta; \Sigma; \Theta \Vdash \widetilde{\psi}'(\pi_i) \rightsquigarrow \mathrm{e}_i \tag{61}$$

and with Lemma 5.37

$$\mathrm{FV}^{\mathsf{a}}(\widetilde{\psi}'(\pi_i)) \subseteq \mathrm{Dom}(\Sigma)$$

hence

$$\mathrm{FV}^{\mathsf{a}}(\pi_i) \subseteq \mathrm{Dom}(\Sigma) \cup \mathsf{A}' \subseteq \mathrm{Dom}(\Sigma) \cup \mathsf{A} \tag{62}$$
$$\mathrm{FV}^{\mathsf{a}}(\widetilde{\psi}') \subseteq \mathrm{Dom}(\Sigma)$$
$$\mathrm{FV}^{\mathsf{a}}(\widetilde{\psi}') \cap \mathrm{Dom}(\psi) = \emptyset$$

Using Equations 59, 60, 61, 62, we obtain

$$\mathrm{FV}^{\mathsf{a}}(\pi_i) \cap \mathrm{Dom}(\psi) = \emptyset$$
$$\psi(\pi_i) = \pi_i$$
$$\Delta; \Sigma; \Theta \Vdash \widetilde{\psi}'(\psi(\pi_i)) \rightsquigarrow \mathrm{e}_i$$

By defining

$$\widetilde{\psi}'' := \widetilde{\psi}' \dot{\cup} \{\mathsf{a} \mapsto \psi(\widetilde{\psi}(\mathsf{a})) \mid \mathsf{a} \in \mathsf{A} \setminus \mathsf{A}'\}$$

we get from the definition of A'

$$\Delta; \Sigma; \Theta \Vdash \widetilde{\psi}''(\psi(\pi_i)) \rightsquigarrow \mathrm{e}_i \tag{63}$$

Because $\mathsf{A} \cap \mathrm{FV}^{\mathsf{a}}(\psi) = \emptyset$, we also have

$$\psi(\Gamma)(\mathsf{z}) = \forall \mathsf{A}.\overline{\psi(\pi_i)}^{i\in[n]} \Rightarrow \psi(\tau')$$

Thus, by Equation 63 and rule $(var)^t$, we obtain

$$\Delta; \Sigma; \Theta; \psi(\Gamma) \vdash \mathsf{z} \rightsquigarrow \mathrm{c}^{\mathsf{z}}\ \overline{\mathrm{e}} : \widetilde{\psi}''(\psi(\tau'))$$

To finish this part of the proof, we still need to show that $\widetilde{\psi}''(\psi(\tau')) = \psi(\tau) = \psi(\widetilde{\psi}(\tau'))$. We do this by induction over τ':

CASE $\tau' = \mathsf{a}$:

- Suppose $\mathsf{a} \notin \mathsf{A}$.
  $\widetilde{\psi}''(\psi(\mathsf{a})) = \psi(\mathsf{a}) \quad (\mathrm{Dom}(\widetilde{\psi}'') = \mathsf{A}, \mathsf{A} \cap \mathrm{FV}^{\mathsf{a}}(\psi) = \emptyset)$
  $\psi(\widetilde{\psi}(\mathsf{a})) = \psi(\mathsf{a}) \quad (\mathrm{Dom}(\widetilde{\psi}) = \mathsf{A})$
- Suppose $\mathsf{a} \in \mathsf{A}$, $\mathsf{a} \in \mathsf{A}'$.
  $\widetilde{\psi}''(\psi(\mathsf{a})) = \widetilde{\psi}''(\mathsf{a}) = \widetilde{\psi}'(\mathsf{a}) \quad (\mathsf{A} \cap \mathrm{Dom}(\psi) = \emptyset)$
  $\psi(\widetilde{\psi}(\mathsf{a})) = \psi(\widetilde{\psi}'(\mathsf{a})) = \widetilde{\psi}'(\mathsf{a}) \quad (\mathrm{FV}^{\mathsf{a}}(\widetilde{\psi}') \cap \mathrm{Dom}(\psi) = \emptyset)$
- Suppose $\mathsf{a} \in \mathsf{A}$, $\mathsf{a} \notin \mathsf{A}'$.
  $\widetilde{\psi}''(\psi(\mathsf{a})) = \widetilde{\psi}''(\mathsf{a}) = \psi(\widetilde{\psi}(\mathsf{a})) \quad (\mathsf{A} \cap \mathrm{Dom}(\psi) = \emptyset)$

CASE $\tau' = \mathsf{T}^{\kappa}\ \overline{\tau}^{\kappa}$: Immediate from the induction hypothesis.

CASE $\mathsf{w} = \mathsf{m}$: Analogously to case $\mathsf{w} = \mathsf{z}$. Note that $\Sigma \vdash \widetilde{\psi}(\psi(\mathsf{b})) \rightsquigarrow \mathrm{u}_{\mathsf{b}}$ follows from $\psi(\mathsf{b}) = \mathsf{b}$.

CASE $\mathsf{w} = \mathsf{w}_1\ \mathsf{w}_2$: Follows directly with the induction hypothesis.

CASE $\mathsf{w} = \lambda \mathsf{z}.\mathsf{w}'$: Follows directly with the induction hypothesis.

CASE $\mathsf{w} = \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2$: We get from the premise of rule $(let)^t$:

$$\Sigma' = \Sigma \overrightarrow{\cup} \{\mathsf{a} \mapsto {}'a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}$$

$$\Theta' = (\Theta^s, \Theta^i, \{\overline{\mathsf{C}_i\ \tau_i \mapsto \mathrm{c}_i}^{i \in [n]}) \tag{64}$$

$$\Delta; \Sigma'; \Theta'; \Gamma \vdash \mathsf{w}_1 \rightsquigarrow \mathrm{e}_1 : \tau' \tag{65}$$

$$\mathrm{Gen}(\Theta', \Gamma, \tau') = \forall \mathsf{A}.\rho = \sigma$$

$$\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v} \tag{66}$$

$$\Delta; \Sigma; \Theta; \Gamma, \mathsf{z} \mapsto \sigma \vdash \mathsf{w}_2 \rightsquigarrow \mathrm{e}_2 : \tau \tag{67}$$

By definition of Gen and by Equation 64, we obtain

$$\mathrm{FV}^{\mathsf{a}}(\Theta') \subseteq \mathrm{Dom}(\Sigma')$$

W.l.o.g., $\mathsf{A} \cap \mathrm{FV}^{\mathsf{a}}(\psi)$, hence

$$\mathrm{FV}^{\mathsf{a}}(\psi) \cap \mathrm{Dom}(\Sigma') = \emptyset$$

Now we can apply the induction hypothesis to Equation 65 and obtain

$$\Delta; \Sigma'; \Theta'; \psi(\Gamma) \vdash \mathsf{w}_1 \rightsquigarrow \mathrm{e}_1 : \psi(\tau')$$

From Equation 66 and Lemma 5.35, we get $\mathrm{FV}^{\mathsf{a}}(\sigma) \subseteq \mathrm{Dom}(\Sigma)$, hence $\mathrm{FV}^{\mathsf{a}}(\sigma) \cap \mathrm{Dom}(\psi) = \emptyset$, and so

$$\Delta; \Sigma \vdash \psi(\sigma) \rightsquigarrow \mathrm{v}$$

Applying the induction hypothesis to Equation 67 yields

$$\Delta; \Sigma; \Theta; \psi(\Gamma, \mathsf{z} \mapsto \sigma) \vdash \mathsf{w}_2 \rightsquigarrow \mathrm{e}_2 : \psi(\tau)$$

Now the proposition follows by rule $(let)^t$. □

Proof of Lemma 5.39

We prove the theorem by induction on the structure of w.

CASE w = z: The last rule in the derivation must be $(var)^t$. We get from the premise of this rule:

$$\Gamma(\mathsf{z}) = \forall \mathsf{A}.\overline{\mathsf{C}_i\ \tau_i'}^{i\in[n]} \Rightarrow \tau'$$
$$\psi = [\overline{\tau_\mathsf{a}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}}]$$
$$\psi(\tau') = \tau$$
$$\Delta; \Sigma; \Theta \Vdash \mathsf{C}\ \psi(\tau_i') \rightsquigarrow \mathrm{e} \tag{68}$$

Without loss of generality,

$$\mathsf{A} \cap \mathrm{FV}^{\mathsf{a}}(\psi) = \emptyset$$

Because $\Gamma(\mathsf{z})$ is unambiguous, we can also assume that

$$\mathsf{A} \subseteq \mathrm{FV}^{\mathsf{a}}(\tau')$$

The assumption $\Delta; \Sigma; \mathcal{C} \vdash \sigma \rightsquigarrow \mathcal{C}(\mathrm{c}^{\mathsf{z}})$ and Lemma 5.29 give us

$$\mathcal{C}(\mathrm{c}^{\mathsf{z}}) = \forall \underbrace{\{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}}_{=:B}.\overline{<\mathcal{S}_\Delta(\mathsf{C}_i, u_i')>}^{i\in[n]} \to u'$$
$$\Sigma'; \mathcal{C}' \vdash \tau_i' \rightsquigarrow u_i' \tag{69}$$
$$\Sigma'; \mathcal{C}' \vdash \tau' \rightsquigarrow u' \tag{70}$$
$$\Sigma' := \Sigma \overrightarrow{\cup} \{\mathsf{a} \mapsto `\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}$$
$$\mathcal{C}' := \mathcal{C} \overrightarrow{\cup} \{`\mathrm{a}^{\mathsf{a}} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}$$

We now define

$$\mathcal{T} := \{\mathsf{a} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}$$
$$\mathcal{T}' := \{\mathsf{a} \mapsto \widetilde{u} \mid \mathsf{a} \in \mathrm{FV}^{\mathsf{a}}(\sigma) \cup \mathrm{FV}^{\mathsf{a}}(\psi), \Sigma; \mathcal{C} \vdash \mathsf{a} \rightsquigarrow \widetilde{u}\}$$

Using Equations 69 and 70, and Lemma 5.17, we get

$$\mathcal{T} \dot\cup \mathcal{T}' \vdash \tau_i' \rightsquigarrow u_i'$$
$$\mathcal{T} \dot\cup \mathcal{T}' \vdash \tau' \rightsquigarrow u'$$

W.l.o.g., $B = \mathrm{Img}(\mathcal{T}) \cap \mathrm{FV}^{`a}(\mathcal{T}') = \emptyset$, and so with Lemma 5.20

$$\mathcal{T}' \vdash \psi(\tau_i') \rightsquigarrow \phi(u_i') \tag{71}$$
$$\mathcal{T}' \vdash \psi(\tau') \rightsquigarrow \phi(u') \tag{72}$$

for

$$\phi = \{`a^{\mathsf{a}} \mapsto u_{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}, \mathcal{T}' \vdash \psi(\mathsf{a}) \rightsquigarrow u_{\mathsf{a}}\}$$

From Equations 68 and 71, and Lemma 5.34, we get

$$\mathcal{C} \vdash \mathrm{e}_i : <\mathcal{S}_\Delta(\mathsf{C}_i, \phi(u_i'))>$$

We also have

$$\mathcal{C}(\mathrm{c}^{\mathsf{z}}) \succ \phi(\overline{<\mathcal{S}_\Delta(\mathsf{C}_i, u_i')>}^{i\in[n]} \to u') \overset{\text{Lemma 5.25}}{=} \overline{<\mathcal{S}_\Delta(\mathsf{C}_i, \phi(u_i'))>}^{i\in[n]} \to \phi(u') \quad (73)$$

Using rules (exp_{id}) and (exp_{app}), we can derive that

$$\mathcal{C} \vdash \mathrm{c}^{\mathsf{z}}\ \overline{\mathrm{e}}^n : \phi(u')$$

Equation 72 and Lemma 5.17 give us

$$\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow \phi(u')$$

which finishes this case of the proof.

CASE $\mathsf{w} = \mathsf{m}$: The last rule in the derivation must be $(method)^t$. We get from the premise of this rule:

$$\begin{gathered} \Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau' \\ \psi = [\overline{\tau_{\mathsf{a}}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}}] \\ \psi(\tau') = \tau \\ \Delta; \Sigma; \Theta \Vdash \mathsf{C}\ \tau_{\mathsf{b}} \rightsquigarrow \mathrm{e} \\ \Sigma \vdash \tau_{\mathsf{b}} \rightsquigarrow \mathrm{u}_{\mathsf{b}} \end{gathered} \quad (74)$$

Γ is unambiguous and $\mathrm{FV}^{\mathsf{a}}(\Gamma(\mathsf{m})) = \emptyset$, so we can safely assume that

$$\begin{gathered} \mathrm{FV}^{\mathsf{a}}(\tau') = \mathsf{A} \\ \mathrm{FV}^{\mathsf{a}}(\tau) = \mathrm{FV}^{\mathsf{a}}(\psi(\tau')) = \mathrm{FV}^{\mathsf{a}}(\psi) \end{gathered}$$

From the assumptions we get $\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \mathrm{Dom}(\Sigma)$, so the following definition makes sense:

$$\mathcal{T} := \{\widetilde{\mathsf{a}} \mapsto \widetilde{u_{\mathsf{a}}} \mid \widetilde{\mathsf{a}} \in \mathrm{FV}^{\mathsf{a}}(\tau), \Sigma; \mathcal{C} \vdash \widetilde{\mathsf{a}} \rightsquigarrow \widetilde{u_{\mathsf{a}}}\}$$

Using Corollary 5.18, we can find unique u and u_{a} for $\mathsf{a} \in \mathsf{A}$, such that

$$\begin{gathered} \mathcal{T} \vdash \tau \rightsquigarrow u \\ \mathcal{T} \vdash \tau_{\mathsf{a}} \rightsquigarrow u_{\mathsf{a}} \end{gathered} \quad (75)$$

Lemma 5.34 and Equation 74 give us

$$\mathcal{C} \vdash \mathrm{e} : < \underbrace{\mathcal{S}_\Delta(\mathsf{C}, u_\mathsf{b})}_{=:\mathcal{S}} >$$

and with Lemma 5.26, we get

$$\mathcal{C} \vdash \Delta(\mathsf{C}) \textbf{ where type } \mathrm{t} = \mathrm{u}_\mathsf{b} : \mathcal{S}$$

Let us now define

$$\mathcal{C}' := \mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$$

If we can show that $\mathcal{C}' \vdash \mathrm{X.x}^\mathsf{m} : u$, then we can use rule $(exp_{open})^+$to finish this case of the proof.

In order to prove $\mathcal{C}' \vdash \mathrm{X.x}^\mathsf{m} : u$, we need to show $\mathcal{S}(\mathrm{x}^\mathsf{m}) \succ u$. Δ is well-typed, so

$$\Delta(\mathsf{C}) = \Lambda\{\alpha\}.\mathcal{S}'$$
$$\mathcal{S}'(\mathrm{x}^\mathsf{m}) = \forall \overbrace{\{`a^\mathsf{a} \mid \mathsf{a} \in \underbrace{\mathsf{A} \setminus \{\mathsf{b}\}}_{=:\mathsf{A}'}\}}^{=:B}.u'$$
$$\{\mathsf{b} \mapsto \alpha\} \dot\cup \underbrace{\{\mathsf{a} \mapsto `a^\mathsf{a} \mid \mathsf{a} \in \mathsf{A}'\}}_{=:\mathcal{T}'} \vdash \tau' \rightsquigarrow u'$$

By renaming bound variables, we can assume that

$$\mathsf{A}' \cap \mathrm{FV}^\mathsf{a}(\tau) = \emptyset \tag{76}$$
$$\mathrm{FV}^{`a}(u_\mathsf{a}) \cap B = \emptyset \tag{77}$$
$$\mathrm{FV}^{`a}(\widetilde{u_\mathsf{a}}) \cap B = \emptyset \tag{78}$$

Equation 76 implies $\mathrm{Dom}(\mathcal{T}) \cap \mathrm{Dom}(\mathcal{T}') = \emptyset$, so

$$\mathcal{T} \dot\cup \mathcal{T}' \vdash [\tau_\mathsf{b}/\mathsf{b}]\tau' \rightsquigarrow [u_\mathsf{b}/\alpha]u'$$

Equation 78 ensures that the requirements of Lemma 5.20 are met (the role of $\mathcal{T}$ and $\mathcal{T}'$ are swapped in this lemma), so we can show that

$$\mathcal{T} \vdash [\overline{\tau_\mathsf{a}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}'}][\tau_\mathsf{b}/\mathsf{b}]\tau' \rightsquigarrow \underbrace{\overline{[u_\mathsf{a}/`a^\mathsf{a}]}^{\mathsf{a}\in\mathsf{A}'}}_{=:\varphi}[u_\mathsf{b}/\alpha]u'$$

Because of $\mathrm{FV}^\mathsf{a}(\tau_\mathsf{b}) \cap \mathsf{A}' = \emptyset$, we have

$$[\overline{\tau_\mathsf{a}/\mathsf{a}}^{\mathsf{a}\in\mathsf{A}'}][\tau_\mathsf{b}/\mathsf{b}]\tau' = \psi(\tau') = \tau$$

and so by Corollary 5.18 and Equation 75

$$u = \varphi([u_b/\alpha]u') \tag{79}$$

We know that $\mathcal{S} = [u_b/\alpha]\mathcal{S}'$, hence

$$\mathcal{S}(x^m) = [u_b/\alpha](\forall B.u')$$

Using Equation 77, we get

$$\mathcal{S}(x^m) = \forall B.([u_b/\alpha]u')$$

Equation 79 and $\mathrm{Dom}(\varphi) = B$ now let us conclude that

$$\mathcal{S}(x^m) \succ u$$

CASE $w = w_1\ w_2$: The last rule in the derivation must be $(\rightarrow E)^t$. We can safely assume that $\mathrm{FV}^a(\tau') \subseteq \mathrm{Dom}(\Sigma)$ because of the substitution lemma 5.38. Now the proposition follows directly from the induction hypothesis, Lemma 5.30, and rule (exp_{app}).

CASE $w = \lambda z.w'$: We have

$$\Delta; \Sigma; \Theta; \Gamma, z \mapsto \tau' \vdash w' \rightsquigarrow e' : \tau \tag{80}$$

from the premise of rule $(\rightarrow I)^t$. From the assumptions, $\mathrm{FV}^a(\tau') \subseteq \mathrm{Dom}(\Sigma)$, so with Lemmata 5.14 and 5.15

$$\Sigma; \mathcal{C} \vdash \tau' \rightsquigarrow u' \tag{81}$$

Let us define

$$\mathcal{C}' := \mathcal{C}, c^z \mapsto u'$$

Now we can apply the induction hypothesis with $\mathcal{C} = \mathcal{C}'$ to Equation 80 and get

$$\begin{gathered} \mathcal{C}' \vdash e' : u \\ \Sigma; \mathcal{C}' \vdash \tau \rightsquigarrow u \end{gathered} \tag{82}$$

Rule (exp_{abs}) gives us

$$\mathcal{C} \vdash \lambda c^z.e' : u' \rightarrow u$$

Using Equations 81 and 82, together with the strengthening lemma 5.32, we get

$$\Sigma; \mathcal{C} \vdash \tau' \rightarrow \tau \rightsquigarrow u' \rightarrow u$$

CASE $\mathsf{w} = \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2$: We get from the premise of rule $(let)^t$:

$$\Sigma' = \Sigma \overrightarrow{\cup} \{\mathsf{a} \mapsto \text{`}a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}$$
$$\Theta' = (\Theta^s, \Theta^i, \{\overline{\mathsf{C}_i\ \tau_i \mapsto \mathrm{c}_i}^{i\in[n]}), \mathrm{c}_i \in \mathrm{FreshCoreIds}$$
$$\Delta; \Sigma'; \Theta'; \Gamma \vdash \mathsf{w}_1 \rightsquigarrow \mathrm{e}_1 : \tau' \tag{83}$$
$$\mathrm{Gen}(\Theta', \Gamma, \tau') = \forall \mathsf{A}.\rho = \sigma$$
$$\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v} \tag{84}$$
$$\Delta; \Sigma; \Theta; \Gamma, \mathsf{z} \mapsto \sigma \vdash \mathsf{w}_2 \rightsquigarrow \mathrm{e}_2 : \tau \tag{85}$$

From the definition of Gen and A, we have

$$\mathrm{FV}^{\mathsf{a}}(\tau') \cup \mathrm{FV}^{\mathsf{a}}(\overline{\tau}) \subseteq \mathsf{A} \cup \mathrm{FV}^{\mathsf{a}}(\Theta) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma) \subseteq \mathrm{Dom}(\Sigma') \tag{86}$$

Hence, there exists unique u_i (Lemma 5.14, 5.15), such that

$$\Sigma'; \mathcal{C}' \vdash \tau_i \rightsquigarrow u_i$$

Let us now define

$$\mathcal{C}' := \mathcal{C}, \overline{\text{`}\mathrm{a}^{\mathsf{a}} \mapsto \text{`}a^{\mathsf{a}}}^{\mathsf{a}\in\mathsf{A}}$$
$$\mathcal{C}'' := \mathcal{C}', \overline{\mathrm{c}_i \mapsto \mathcal{S}_\Delta(\mathsf{C}_i, u_i)}^{i\in[n]}$$

We want to apply the induction hypothesis with $\mathcal{C} = \mathcal{C}'$ to Equation 83. But first, we have to make sure that all assumptions hold (we only mention the nontrivial ones here):

- $\mathrm{CS}(\Theta') \subseteq \mathrm{Dom}(\Delta)$ because $\{\overline{\mathsf{C}}\} = \mathrm{CS}(\sigma) \overset{\text{Lemma 5.36}}{\subseteq} \mathrm{Dom}(\Delta)$
- $\mathrm{FV}^{\mathsf{a}}(\Theta') \cup \mathrm{FV}^{\mathsf{a}}(\tau') \subseteq \mathrm{Dom}(\Sigma)$ because of Equation 86
- $\mathcal{C}''$ is compatible with Δ, Σ', Θ' because $\mathcal{C}'' \vdash \text{`}\mathrm{a}^{\mathsf{a}} \triangleright \text{`}a^{\mathsf{a}}$ for all $\mathsf{a} \in \mathsf{A}$, $\mathcal{C}'' \vdash \mathrm{c}_i : \mathcal{S}_\Delta(\mathsf{C}_i, u_i)$, and $\mathrm{c}_i \in \mathrm{FreshCoreIds}$

Thus, the induction hypothesis can be applied and yields together with the strengthening lemma 5.32

$$\mathcal{C}'' \vdash \mathrm{e}_1 : u'$$
$$\Sigma', \mathcal{C}' \vdash \tau' \rightsquigarrow u'$$

Using Equation 86, we get

$$\mathrm{FV}^{\mathsf{a}}\ \sigma \subseteq \mathrm{Dom}(\Sigma)$$

so there exists some unique v such that

$$\mathcal{C} \vdash \mathrm{v} \triangleright v \tag{87}$$

We have $\sigma = \forall \mathsf{A}.\rho$, and

$$\rho = \overline{\mathsf{C}_i\ \tau_i}^{i\in[n]} \Rightarrow \tau' \tag{88}$$

from the definition of Gen, so with Lemma 5.29 and Equation 84

$$v = \forall\{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}.\overline{\mathcal{S}_\Delta(\mathsf{C}_i, u_i)}^{i\in[n]} \to u' \tag{89}$$

Hence, with rule (exp_{abs})

$$\mathcal{C}' \vdash \lambda \bar{\mathrm{c}}^n.\mathrm{e}_1 : \overline{\mathcal{S}_\Delta(\mathsf{C}_i, u_i)}^{i\in[n]} \to u' \tag{90}$$

Now define

$$\mathcal{C}''' := \mathcal{C}, \mathrm{c}^{\mathsf{z}} \mapsto v$$

Let us apply the induction hypothesis with $\mathcal{C} = \mathcal{C}'''$ to Equation 85. Note that, for all $(\mathsf{C}'\ \tau', \mathrm{e}') \in \Theta^l$ with $\mathcal{C} \vdash \mathrm{e}' : u'$, we also have $\mathcal{C}''' \vdash \mathrm{e}' : u'$ (weakening lemma 5.31 together with $\mathrm{c}^{\mathsf{z}} \notin \mathrm{FreshCoreIds}$). The other assumptions hold trivially, so we get

$$\mathcal{C}''' \vdash \mathrm{e}_2 : u \tag{91}$$

$$\Sigma; \mathcal{C}''' \vdash \tau \rightsquigarrow u \tag{92}$$

W.l.o.g., $\{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\} \cap \mathrm{FV}^{`a}(\mathcal{C}) = \emptyset$. The remaining premises of rule $(exp_{let'})^+$ follow from Equations 87, 90, and 91, so we can now derive

$$\mathcal{C} \vdash \mathbf{let}\ \mathrm{c}^{\mathsf{z}} : \mathrm{v} = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2 : u$$

Equation 92, together with the strengthening lemma 5.32, finally gives us $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$. □

Proof of Lemma 5.41

We get from $\vdash \mathsf{inst} \rightsquigarrow \Theta'$

$$\Theta' = (\emptyset, \{\theta \mapsto \mathrm{F}\}, \emptyset)$$

Hence, we need to check only that $\emptyset \vdash \mathrm{rfun} \triangleright \mathcal{C}$, and that condition 3 of Definition 5.33 (compatibility of Tiny-ML$^+$ contexts) holds for $\mathcal{C}(\mathrm{F})$. We know by looking at rule $(inst_{check})^t$ that rfun and inst must be of the following form:

$$\begin{aligned}\mathrm{rfun} = \mathbf{functor}\ \mathrm{F}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[r]}, \mathrm{Y} : \overbrace{\mathbf{sig}\ \overline{\mathbf{type}\ \mathrm{t}^{\mathsf{b}}}^{\mathsf{b}\in\mathsf{B}}\ \mathbf{end}}^{=:\mathrm{S_Y}}) \\ : \Delta(\mathsf{C})\ \mathbf{where\ type}\ \mathrm{t} = \mathrm{u} = \ldots\end{aligned}$$

$$\mathsf{inst} = \mathbf{instance}\ \underbrace{\forall \mathsf{A}.\overline{\mathsf{C}_i\ \mathsf{a}_i}^{i\in[r]} \Rightarrow \mathsf{C}\ \tau}_{=:\theta}\ \mathbf{where}\ \ldots$$

We get by definition of S_i and u in the premise of this rule, by Lemma 5.26, and by the well-definedness of Δ

$$\emptyset \overset{\text{funargs}}{\vdash} \overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[r]}, \mathrm{Y} : \mathrm{S}_\mathrm{Y} \triangleright \forall P.\overline{\mathcal{S}}^{r+1} \tag{93}$$

$$\{\mathrm{X}_i \mapsto \mathcal{S}_i \mid i \in [r]\} \dot\cup \{\mathrm{Y} \mapsto \mathcal{S}_{r+1}\} \vdash \Delta(\mathsf{C}) \textbf{ where type } \mathrm{t} = \mathrm{u} \triangleright \mathcal{S} \tag{94}$$

such that for $\forall P.\overline{\mathcal{S}}^{r+1}$ and $\mathcal{S}$ the conditions 3.1 – 3.7 of Definition 5.33 hold. Now by Equations 93 and 94, and by rule $(\mathit{rfuns}_{\mathit{collect}})^+$

$$\emptyset \vdash \mathrm{rfun} \triangleright \underbrace{\{\mathrm{F} \mapsto \forall P.\overline{\mathcal{S}}^{r+1} \to \mathcal{S}\}}_{=:\mathcal{C}}$$

Hence, by Lemma 5.40

$$\mathrm{FV}^\alpha(\mathcal{C}(\mathrm{F})) = \emptyset$$

This gives us condition 3.8 of Definition 5.33, so $\mathcal{C}$ is compatible with Δ, $\Sigma = \emptyset$, and Θ'. □

Proof of Lemma 5.43

Propositions 1 and 2 follow from the premise of rule $(\mathit{inst}_{\mathit{check\text{-}method}})^t$, proposition 3 follows from $\mathrm{FV}^\mathsf{a}(\Gamma) = \emptyset$, and proposition 4 can be fulfilled by renaming bound type variables. We still need to prove propositions 5 and 6. We get from the premise of rule $(\mathit{inst}_{\mathit{check\text{-}method}})^t$

$$\begin{gathered}\Sigma' = \{\mathsf{a} \mapsto \text{`}\mathrm{a}^\mathsf{a} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\} \\ \Delta; \Sigma'; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : [\tau/\mathsf{b}]\tau'\end{gathered} \tag{95}$$

We now want to apply Lemma 5.39 to Equation 95. Assumptions 1 – 4 and 7 of this lemma hold trivially, assumption 5 follows with

$$\mathrm{FV}^\mathsf{a}([\tau/\mathsf{b}]\tau') \overset{\mathrm{FV}^\mathsf{a}(\Gamma)=\emptyset}{\subseteq} \mathrm{FV}^\mathsf{a}(\tau) \cup \mathsf{A} \setminus \{\mathsf{b}\} \overset{\mathrm{FV}^\mathsf{a}(\tau)\subseteq\Sigma}{\subseteq} \mathrm{Dom}(\Sigma')$$

and assumption 6 follows by construction of $\mathcal{C}'$. Hence by Lemma 5.39

$$\begin{gathered}\mathcal{C}' \vdash \mathrm{e} : u'' \\ \Sigma'; \mathcal{C}' \vdash [\tau/\mathsf{b}]\tau' \rightsquigarrow u''\end{gathered} \tag{96}$$

We get $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$ from the assumptions, so we can apply Lemma 5.42 to Equation 96 and obtain with Lemma 5.14 and proposition 3 of this lemma that

$$u' = u'' \tag{97}$$

This proves proposition 6 of this lemma. Proposition 5 follows by Equations 96 and 97 because we get $\Sigma' \vdash [\tau/\mathsf{b}]\tau' \rightsquigarrow \mathrm{u}'$ from the premise of rule $(\mathit{inst}_{\mathit{check\text{-}method}})^t$. □

Proof of Lemma 5.44

We assume that all symbols for which no explicit definition is given here are defined as in rule $(inst_{check})^t$. We first define

$$\mathrm{S_Y} = \mathbf{sig}\ \mathbf{type}\ \overline{\mathrm{t}^{\mathsf{b}}}^{\mathsf{b}\in\mathsf{B}}\ \mathbf{end}$$
$$\mathrm{S} = \Delta(\mathsf{C})\ \mathbf{where}\ \mathbf{type}\ \mathrm{t} = u$$

$\emptyset \vdash \mathrm{rfun} \triangleright \{\mathrm{F} \mapsto \mathcal{C}(\mathrm{F})\}$ gives us

$$\emptyset \overset{\text{funargs}}{\vdash} \overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[r]}, \mathrm{Y} : \mathrm{S_Y} \triangleright \forall P.\overline{\mathcal{S}}^{r+1}$$
$$\underbrace{\overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i\in[r]}, \mathrm{Y} \mapsto \mathcal{S}_{r+1}}_{=:\,\mathcal{C}'} \vdash \mathrm{S} \triangleright \mathcal{S}$$

$\mathcal{C}$ is compatible with Δ, $\Sigma = \emptyset$, and Θ, hence

$$\mathcal{S} = \mathcal{S}_\Delta(\mathsf{C}, u)$$
$$\{\mathsf{a}_i \mapsto \mathcal{S}_i(\mathrm{t}) \mid i \in [r]\} \dot\cup \{\mathsf{b} \mapsto \mathcal{S}_{r+1}(\mathrm{t}^{\mathsf{b}}) \mid \mathsf{b} \in \mathsf{B}\} \vdash \tau \rightsquigarrow u$$
$$\mathcal{S}_i(\mathrm{t}) = \mathcal{S}_j(\mathrm{t}) \text{ iff } \mathsf{a}_i = \mathsf{a}_j \text{ for all } i, j \in [r]$$

Now by Lemma 5.17 and by definition of Σ in the premise of rule $(inst_{check})^t$

$$\Sigma; \mathcal{C}' \vdash \tau \rightsquigarrow u$$

so that we also have

$$\Sigma; \mathcal{C}'' \vdash \tau \rightsquigarrow u \tag{98}$$
$$\mathcal{C}'' := \mathcal{C} \dot\cup \mathcal{C}'$$

For $\Gamma(\mathsf{m}_i) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \tau_i$, we get by Lemma 5.28

$$\mathcal{S}(\mathrm{x}^{\mathsf{m}_i}) = \forall\{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}.u_i \tag{99}$$
$$\{\mathsf{b} \mapsto u\} \dot\cup \{\mathsf{a} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\} \vdash \tau_i \rightsquigarrow u_i$$

We now want to apply Lemma 5.43 to the usages of the $\overset{\text{method}}{\vdash}$ judgment in the premise of rule $(inst_{check})^t$. Δ is well-typed w.r.t. Θ' and Γ because $\mathrm{Sup}(\Theta, \mathsf{C}) = \mathrm{Sup}(\Theta', \mathsf{C})$. $\mathcal{C}''$ is compatible with Δ, Σ, and Θ' because $\mathcal{C}'' \vdash \mathbf{pack}\ \mathrm{X}_j\ \mathbf{as}\ \mathrm{S}_j : <\mathcal{S}_\Delta(\mathsf{C}_j, \mathcal{S}_j(\mathrm{t}))>$ and $\Sigma; \mathcal{C}'' \vdash \mathsf{a}_j \rightsquigarrow \mathcal{S}_j(\mathrm{t})$. The other assumptions of the lemma hold trivially, so we get

$$\mathcal{C}''' := \mathcal{C}'', \{`\mathrm{a}^{\mathsf{a}} \mapsto `a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}$$
$$\mathcal{C}''' \vdash \mathrm{e}_i : u_i \tag{100}$$
$$\mathcal{C}''' \vdash \mathrm{u}_i \triangleright u_i \tag{101}$$
$$\mathrm{v}_i = \forall\{`\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}.\mathrm{u}_i \tag{102}$$
$$\{`a^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\} \cap \mathrm{FV}^{`a}(\mathcal{C}'') = \emptyset \tag{103}$$

For $\mathsf{C}^{\mathsf{sup}} \in \mathrm{Sup}(\Theta, \mathsf{C})$, we get by Lemma 5.27

$$\mathcal{S}(\mathrm{x}^{\mathsf{C}^{\mathsf{sup}}}) = <\mathcal{S}_\Delta(\mathsf{C}^{\mathsf{sup}}, u)> \tag{104}$$

and by Lemma 5.34

$$\mathcal{C}'' \vdash \mathrm{e}^{\mathsf{sup}} : <\mathcal{S}_\Delta(\mathsf{C}^{\mathsf{sup}}, u)> \tag{105}$$

By using Equation 98 with rule $(strb_t)$, Equations 99 – 103 with rule $(strb_{v'})^+$, and Equations 104, 105 with rule $(strb_v)$, we can eventually conclude that

$$\mathcal{C}'' \vdash \mathrm{s} : \mathcal{S}$$ □

Proof of Lemma 5.45

Let us first define

$$\Theta'' := \Theta \dot\cup \Theta'$$
$$\Gamma'' := \Gamma \dot\cup \Gamma'$$

Δ is well-typed w.r.t. Θ'' and Γ'' because $\mathsf{C} \notin \mathrm{Dom}(\Delta)$. Therefore, we need to check the conditions of Definition 5.23 (well-typedness of class environments) only for C (notice that $\mathrm{Dom}(\Delta') = \{\mathsf{C}\}$). Conditions 1 and 2 hold by definition of S in the premise of rule $(class)^t$.

We need to check condition 3 only for $\mathsf{m} \in \mathrm{Dom}(\Gamma')$ because there is no $\mathsf{m} \in \mathrm{Dom}(\Gamma)$ of the form $\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \rho$. Obviously, condition 3 holds for all $\mathsf{m} \in \mathrm{Dom}(\Gamma')$ by construction of the v_i in the premise of rule $(class)^t$.

Similarly, condition 4 needs to be checked only for $\mathsf{C}' \in \mathrm{Sup}(\Theta', \mathsf{C})$ because $\mathrm{Sup}(\Theta, \mathsf{C}) = \emptyset$. Now suppose $\mathsf{C}' \in \mathrm{Sup}(\Theta', \mathsf{C})$. Then there exists some $i \in [r]$ such that $\mathsf{C}' = \mathsf{C}_i$. We have $\mathsf{C}_i \in \mathrm{Dom}(\Delta)$ by the premise of rule $(class)^t$. The other requirements of condition 4 hold by definition of S, by Lemma 5.26, and by rule $(simtyp_{pkg})^+$.

Condition 5 holds trivially by construction of S. This finishes the proof of the lemma. □

Proof of Theorem 5.46

We get from the premise of rule $(prog)^t$:

$$\dot\cup_{j\in[i-1]}\Delta_j \vdash \mathsf{cls}_i \rightsquigarrow \Delta_i; \Theta_i; \Gamma_i \quad (i \in [n]) \tag{106}$$
$$\vdash \mathsf{inst}_i \rightsquigarrow \Theta'_i \quad (i \in [m]) \tag{107}$$
$$\Delta = \dot\cup_{i\in[n]}\Delta_i$$
$$\Theta = \dot\bigcup_{i\in[n]}\Theta_i \dot\cup \dot\bigcup_{i\in[m]}\Theta'_i$$
$$\Gamma = \dot\bigcup_{i\in[n]}\Gamma_i$$
$$\Delta; \Theta; \Gamma \vdash \mathsf{inst}_i \rightsquigarrow \mathrm{rfun}_i \quad (i \in [m]) \tag{108}$$
$$\Delta; \emptyset; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \mathsf{Int}$$

By Equation 106, Lemma 5.45, the well-formedness of pgm, and a straightforward induction on n, we get that Δ is well-formed w.r.t. $\dot\cup_{i\in[n]}\Theta_i$ and Γ. The Θ'_i do not contribute to $\mathrm{Sup}(\Theta, \mathsf{C})$, so we have also

$$\Delta \text{ well-typed w.r.t. } \Theta, \Gamma \tag{109}$$

By Lemma 5.41 and Equations 107, 108 we get for all $i \in [m]$

$$\emptyset \vdash \mathrm{rfun}_i \triangleright \mathcal{C}_i$$
$$\mathcal{C}_i \text{ compatible with } \Delta, \Sigma = \emptyset, \Theta'_i$$

Now we have by rule $(rfuns_{collect})^+$

$$\emptyset \vdash \overline{\mathrm{rfun}}^m \triangleright \underbrace{\dot\bigcup_{i\in[m]} \mathcal{C}_i}_{=:\,\mathcal{C}} \tag{110}$$

The Θ_i do not contribute to the instance part of Θ, hence

$$\mathcal{C} \text{ compatible with } \Delta, \Sigma = \emptyset, \Theta \tag{111}$$

Now we can apply Lemma 5.44 to Equation 108 (assumptions 1, 6, and 7 follow from Equations 109, 111, and 110, respectively; the remaining assumptions hold trivially) and obtain with rule $(rfuns_{check})^+$

$$\mathcal{C} \vdash \overline{\mathrm{rfun}}^m \tag{112}$$

Now by Lemma 5.39 (assumption 1 and 6 follow by Equations 109 and 111, respectively; the remaining assumptions are easy to verify)

$$\mathcal{C} \vdash \mathrm{e} : \mathrm{int} \tag{113}$$

Finally, by rule $(prog_{str})$ and Equation 113, and by rule $(prog_{rec})^+$ and Equations 110, 112, we get

$$\emptyset \vdash \mathrm{prog}$$

This finishes the proof of the theorem. □

Bibliography

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 266–277, January 1997. 1

[CHP99] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia*, pages 50–63, May 1999. 2.4, 5.4

[CKP05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Tallinn, Estonia*, pages 241–253, September 2005. 1, 1.3, 3.3, 3.3.4, 3.3.5, 3.3.5, 4.2.2, 4.4, 6.1.1, 6.1.1, 6.1.1, 6.2

[CKPM05] 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), Long Beach, California*, pages 1–13, January 2005. 1, 3.4, 4.4, 6.1.1

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 236–249, January 2003. 1.2, 4.6

[DJH02] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A formal specification of the Haskell 98 module system. In *ACM Haskell Workshop, Pittsburgh, Pennsylvania*, pages 17–28, October 2002. 3.3.3

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 207–212, January 1982. 2.3.3, 3, 5.6

[Dre04] Derek Dreyer. A type system for well-founded recursion. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Venice, Italy*, pages 293–305, January 2004. 2.5

[Fax02] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4&5):295–357, 2002. 3.2, 3.2.2, 3.2.2, 2, 5, 5.6

[ghc05] The Glasgow Haskell Compiler, 2005. http://www.haskell.org/ghc/. 3.4

[Gil04] Stephen Gilmore. Programming in Standard ML '97: An on-line tutorial, 2004. http://www.dcs.ed.ac.uk/home/stg/NOTES/. 2.1

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, University of Paris VII, 1972. Summary in J. E. Fenstad, editor, *Scandinavian Logic Symposium*, pages 63–92, North-Holland, 1971. 1.2, 2.3.1, 2.4.3, 4.6, 5.6

[HHPW96] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996. 2, 5, 5.6

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. 3, 5.6

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, January 1994. 2.1.4, 6.1.1

[HM93] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993. An earlier version appeared in *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, under the title "The Essence of ML" (Mitchell and Harper), January 1988. 2.5

[HP04] Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module system. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8. MIT Press, 2004. 2.1, 1

[HPF00] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98, 2000. http://haskell.org/tutorial/index.html. 3.1

[hug05] Hugs, 2005. http://www.haskell.org/hugs/. 3.4

[Jon94] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994. 3.2, 3.2.2, 5, 5.3.1, 5.6

[Jon95a] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995. 3.2, 5.4

[Jon95b] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, May 1995. 3.1

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, January 1996. 2.1, 4.6

[Jon00a] Mark P. Jones. Type classes with functional dependencies. In *European Symposium on Programming (ESOP), Berlin, Germany*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 2000. 3.4

[Jon00b] Mark P. Jones. Typing Haskell in Haskell, November 2000. `http://www.cse.ogi.edu/~mpj/thih/`. 5.5

[JP99] Mark P. Jones and John Peterson. The Hugs 98 user manual, 1999. Available from `http://www.haskell.org/hugs/`. 3.3.3

[Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *European Symposium on Programming (ESOP), Nancy, France*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1988. 1, 3

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. 2.5

[Kis04] Oleg Kiselyov. Applicative translucent functors in Haskell, August 2004. Post to the Haskell mailing list, `http://www.haskell.org/pipermail/haskell/2004-August/014463.html`. 1.2, 4.6

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *ACM Haskell Workshop, Snowbird, Utah*, pages 96–107, September 2004. 3.1

[KS01] Wolfram Kahl and Jan Scheffczyk. Named instances for Haskell type classes. In *ACM Haskell Workshop, Firenze, Italy*, informal proceedings, September 2001. Technical Report UU-CS-2001-23, Institute of Information and Computer Sciences, Utrecht University. 1, 1.2, 4.6

[KS04] Oleg Kiselyov and Chung-chieh Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *ACM Haskell Workshop, Snowbird, Utah*, pages 33–44, September 2004. 6.2

[Ler94] Xavier Leroy. Manifest types, modules and separate compilation. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 109–122, January 1994. 2.1.4, 6.1.1

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 142–153, January 1995. 4.4, 4.4

[Ler00a] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000. 2.1

[Ler00b] Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`. 1.2, 4.6

[Mac86] David MacQueen. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 277–286, January 1986. 2.5

[McB02] Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375–392, 2002. 3.1

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978. 2.6, 3, 5.6

[MT94] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming (ESOP), Edinburgh, Scotland*, volume 788 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1994. 2.1.4, 4.4, 4.4

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*, Revised edition. MIT Press, 1997. 1, 2, 2.3, 2.3.2, 2.5, 3, 5.6, 6.1.1

[Par72] David Parnas. The criteria to be used in decomposing systems into modules. *Communications of the ACM*, 14(1):221–227, 1972. 2

[Pau96] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996. 2.1

[Pey03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003. 1, 3, 3.1, 3.2, 6.1.1

[PJ93] John Peterson and Mark P. Jones. Implementing type classes. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico*, pages 227–236, June 1993. 5.5

[PJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: An exploration of the design space. In *ACM Haskell Workshop, Amsterdam, The Netherlands*, informal proceedings, June 1997. 1.3, 3.3, 4.4

[PS04a] Simon Peyton Jones and Mark Shields. Lexically scoped type variables. http://research.microsoft.com/Users/simonpj/papers/scoped-tyvars/, March 2004. 2.4.3

[PS04b] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. http://research.microsoft.com/~simonpj/papers/putting/index.htm, April 2004. 1.2, 4.6

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Paris, France*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974. 2.4.3, 5.6

[Ros05] Andreas Rossberg. Re: CTM Chapter 4, May 2005. Post to the Alice-users mailing list, http://www.ps.uni-sb.de/pipermail/alice-users/2005/000466.html. 1

[RRK+03] Sergei Romanenko, Claudio Russo, Niels Kokholm, Ken Friis Larsen, and Peter Sestoft. Moscow ML homepage, 2003. http://www.dina.dk/~sestoft/mosml.html. 2.4, 4.5, 5.5

[RS02] Andreas Rossberg and Martin Sulzmann. Beyond type classes. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany, 2002. http://www.ps.uni-sb.de/Papers/paper_info.php?label=btclasses. 4.4

[Rus98] Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, Edinburgh, Scotland, 1998. LFCS Thesis ECS–LFCS–98–389. 2.3, 2.6, 2.4.1, 2.5, 4.3.1, 4.3.1

[Rus99] Claudio V. Russo. Non-dependent types for Standard ML modules. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP), Paris, France*, pages 80–97, September 1999. 2.5

[Rus00a] Claudio V. Russo. First-class structures for Standard ML. In *European Symposium on Programming (ESOP), Berlin, Germany*, volume 1782 of *Lecture Notes in Computer Science*, pages 336–350. Springer-Verlag, 2000. 1.3, 2.4, 2.4.1, 5.4, 6.1.2

[Rus00b] Claudio V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, 2000. 2.4.1

[Rus01] Claudio V. Russo. Recursive structures for Standard ML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Firenze, Italy*, pages 50–61, September 2001. 1.3, 2.4, 2.4.2, 2.5, 6.1.2

[Sch00] Gerhard Schneider. *ML mit Typklassen*, June 2000. Diplomarbeit. Fachrichtung Informatik, Universität des Saarlandes. http://www.ps.uni-sb.de/Papers/abstracts/Schneider2000.html. 1, 1.2, 5.6

[Sha99] Zhong Shao. Transparent modules with fully syntactic signatures. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Paris, France*, pages 220–232, September 1999. 2.1.4, 6.1.1, 6.1.1

[Sha04] Chung-chieh Shan. Higher-order modules in System F_ω and Haskell. http://www.eecs.harvard.edu/~ccshan/xlate/, July 2004. 1.2, 4.6

[SP02] Mark B. Shields and Simon Peyton Jones. First class modules for Haskell. In *International Workshop on Foundations of Object-Oriented Languages (FOOL), Portland, Oregon*, informal proceedings, pages 28–40, January 2002. 4.6

[SW05] Martin Sulzmann and Jeremy Wazny. Chameleon, 2005. http://www.comp.nus.edu.sg/~sulzmann/chameleon/. 4.4

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999. 3.1

[WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 60–76, January 1989. 1, 3, 3.1, 5, 5.6