ML Modules and Haskell Type Classes: A Constructive Comparison

STEFAN WEHR

Institut für Informatik, Universität Freiburg,
Georges-Köhler-Allee 079, 79110 Freiburg i. Br., Germany
(e-mail: wehr@informatik.uni-freiburg.de*)*

MANUEL M. T. CHAKRAVARTY

School of Computer Science and Engineering, The University of New South Wales,
UNSW SYDNEY NSW 2052, Australia
(e-mail: chak@cse.unsw.edu.au*)*

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. Both translations preserve type correctness.

Building on the insights obtained from the translations, the article presents a thorough comparison between ML modules and Haskell type classes. Moreover, it evaluates 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.

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 (Kaes, 1988; Wadler & Blott, 1989) 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 (Schneider, 2000; Kahl & Scheffczyk, 2001; Chakravarty *et al.*, 2005b; Chakravarty *et al.*, 2005a; Rossberg, 2005) that there is some overlap in functionality between the module system of the programming language ML (Milner *et al.*, 1997), one of the most powerful module systems in widespread use, and the type class mechanism of the language Haskell (Peyton Jones, 2003), 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.

Contributions. Following the path just described, we make four main contributions:

- We devise two formal translations from ML modules to Haskell type classes and vice versa, prove that the translations preserve type correctness, and provide implementations for both.
- We use the insights obtained from the translations to compare ML modules with Haskell type classes thoroughly.
- We 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.
- We suggest extensions to both languages that address possible shortcomings identified by the translations.

In addition, we introduce a notion of *abstract associated types* as well as a rigorous formalisation of the type structure of four core languages Tiny-ML, Tiny-ML+, Tiny-HS, and Tiny-HS+, which are interesting in their own right.

Outline. The rest of this article is structured as follows. Section 2 introduces the key ideas of the translations through a series of examples. Section 3 then formalizes the translation from ML modules to Haskell type classes by translating Tiny-ML into Tiny-HS$^+$, two languages which are also defined in this section. Furthermore, the section gives a type-correctness preservation proof for the translation. Analogously, Section 4 defines the languages Tiny-HS and Tiny-ML$^+$, formalizes a translation between them, and shows that the translation preserves type correctness. Finally, Section 5 compares ML modules with Haskell type classes and concludes. The remainder of this introductory section presents related work.

Related Work. There is only little work on connecting modules with type classes. None of these works meet our goal of comparing ML modules with Haskell type classes based on formal translations.

Kahl and Scheffczyk (Kahl & Scheffczyk, 2001) propose 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. Kahl and Scheffczyk motivate and explain their extension in terms of OCaml's module system (Leroy, 1995; Leroy, 2000); they do not consider any kind of translation from ML modules to Haskell type classes or vice versa.

Shan (Shan, 2004) presents a formal translation from a sophisticated ML module calculus (Dreyer *et al.*, 2003) into System F_ω (Girard, 1972). The source ML module calculus is a unified formalism that covers a large part of the design space of ML modules. The target language System F_ω of Shan's translation can be encoded in Haskell extended with higher-rank types (Peyton Jones & Shields, 2004); however, this encoding is orthogonal to the type class system. Kiselyov builds on Shan's work and translates a particular applicative functor into Haskell with type classes (Kiselyov, 2004). However, he does not give a formal translation. Neither Shan nor Kiselyov consider translations from type classes to modules.

Schneider (Schneider, 2000) adds Haskell-style type classes to ML. His solution is conservative in the sense that type classes and modules remain two separate concepts. In particular, he does not encode type classes as modules. Translations in the opposite direction are not addressed in his work.

The work closest to ours is Dreyer and colleagues' modular reconstruction of type classes (Dreyer *et al.*, 2006). This work, which strictly speaking came after our own (Wehr, 2005), extends Harper & Stone's type-theoretic interpretation of modules (Harper & Stone, 2000) to include ad-hoc polymorphism in the style of Haskell type classes. Instead of adding an explicit notion of type classes to ML, certain forms of module signatures take the role of class declarations and matching modules may be nominated as being canonical for the purpose of overload resolution. The presented elaboration relation mirrors Haskell's notion of an evidence translation and is related to our translation of Haskell classes into ML modules. Dreyer and colleagues do not consider the converse direction of modelling modules by type classes.

2 Examples

We start with a few examples that demonstrate and motivate the key ideas behind the translations from ML modules to Haskell type classes (Section 2.1) and from Haskell type classes to ML modules (Section 2.2). We assume that you have a working knowledge of modules in Standard ML and of Haskell 98 type classes; other language features are explained along the way.

2.1 From modules to classes

The idea of the translation from ML modules to Haskell type classes is the following: signatures are modeled as type classes, structures and functors are translated

ML

```
structure IntSet = struct
                     type elem       = int
                     type set        = elem list
                     val  empty      = []
                     fun  member i s = List.exists (intEq i) s
                     fun  insert i s = if member i s then s else (i :: s)
                   end
```

Haskell

```
class IntSetSig 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
data IntSet = IntSet
instance IntSetSig IntSet where
    type Elem IntSet = Int
    type Set  IntSet = [Int]
    empty  _         = []
    member _ i s     = any (intEq i) s
    insert _ i s     = if member IntSet i s then s else (i : s)
```

Figure 1. ML structure for integer sets and its translation to Haskell

into instances of type classes, and type and value components of signatures and structures are mapped to associated type synonyms (Chakravarty *et al.*, 2005a) and class methods, respectively. We now substantiate the idea by presenting example translations of signatures and structures (Section 2.1.1), of abstract types (Section 2.1.2), and of functors (Section 2.1.3).

2.1.1 Translating signatures and structures

Our first example is shown in Figure 1. The ML code defines a structure `IntSet`, which implements sets of integers in terms of lists. The signature of `IntSet` is inferred implicitly in ML; however, we must represent it explicitly as a type class `IntSetSig` in Haskell.

The `type` declarations in this class introduce two associated type synonyms `Elem a` and `Set a`. The identities of such type synonyms depend on a particular instantiation of the class variable `a`. Hence, concrete definitions for `Elem` and `Set` are deferred to instance definitions of `IntSetSig`.

The data type `IntSet` corresponds to the name of the structure in ML. We translate the structure itself by defining an instance of `IntSetSig` for `IntSet`. The translation of the `insert` function shows that we encode the usage of the structure component `member` by indexing the method `member` with a value of type `IntSet`.

We use the same technique to translate qualified uses of structure components.

ML

```
structure IntSet' = IntSet :> sig
                                type elem   = int
                                type set
                                val  empty  : set
                                val  member : elem -> set -> bool
                                val  insert : elem -> set -> set
                              end
```

Haskell

```
data IntSet' = IntSet'
instance IntSetSig IntSet' where
    type     Elem IntSet' = Elem IntSet
    abstype  Set  IntSet' = Set IntSet
    empty    _            = empty IntSet
    member   _            = member IntSet
    insert   _            = insert IntSet
```

Figure 2. Sealed ML structure for integer sets and its translation to Haskell

For example, the ML expression `IntSet.insert 1 IntSet.empty` is written as `insert IntSet 1 (empty IntSet)` in Haskell.

2.1.2 Translating abstract types

The `IntSet` structure reveals to its clients that sets are implemented in term of lists. This is not always desirable; often, the type `set` should be kept abstract outside of the structure. Our next example (Figure 2) shows that we can achieve the desired effect in ML by sealing the `IntSet` structure with a signature that leaves the right-hand-side of `set` unspecified. Such signatures are called *translucent*, in contrast to *transparent* (all type components specified) and *opaque* (all type components unspecified) signatures.

Abstract types pose a problem to the translation because there is no obvious counterpart for them in Haskell. One possible solution would be to use Haskell's module system (Diatchki *et al.*, 2002). With this approach, a type can be made abstract by wrapping it in a `newtype` constructor and placing it in a separate module that hides the constructor. This solution is unsatisfactory for two reasons. Firstly, explicit conversion code is necessary to turn a value of the concrete type into a value of the abstract type and vice versa. Secondly, we do not want Haskell's module system to interfere with our comparison of ML modules and Haskell type classes.

Another commonly suggested solution to the problem of encoding abstract types are existentials (Mitchell & Plotkin, 1988). However, existentials are not adequate for expressing modular structure (MacQueen, 1986) because of their closed scoping discipline, so we do not pursue this approach any further.

Instead, we develop *abstract associated type synonyms* to represent abstract types in Haskell. Abstract associated type synonyms, an extension to associated type

synonyms, limit the scope of the right-hand side of an associated type synonym definition to 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.[1] We do not give a formalization of our extension at this point but defer it until Section 3.

The Haskell code in Figure 2 demonstrates how our extension is used to model abstract types in Haskell. The new keyword `abstype` introduces an abstract associated type synonym `Set` in the instance definition for `IntSet'`. The effect of using `abstype` is that the type equality `Set IntSet' = [Int]` is visible from within the instance definition, but not from outside.

Note that there is no explicit Haskell translation for the signature of the structure `IntSet'`. Instead, we reuse the type class `IntSetSig` from Figure 1. Such a reuse is possible because abstraction in Haskell is performed inside *instance* (and not class) definitions, which means that the signatures of the ML structures `IntSet` and `IntSet'`—differing only in whether the type component `set` is abstract or not—would be translated into equivalent type classes. (This is an important difference to ML, where abstraction is performed outside of structure definitions through translucent or opaque signatures. We come back to this difference and its implications in Section 5.1.)

2.1.3 Translating functors

So far, we only considered sets of integers. ML allows the definition of generic sets through functors, which act as functions from structures to structures (we do not consider higher-order functors here, see Section 3.5). Figure 3 shows such a functor. We removed the `elem` type component from the functor body to demonstrate a particular detail of the translation to Haskell.

The Haskell version defines two type classes `EqSig` and `MkSetSig` as the translations of the anonymous argument and result, respectively, The class `MkSetSig` is a multi-parameter type class, a widespread generalization of Haskell 98's single-parameter type classes. The first parameter `b` represents a possible implementation of the functor body, whereas the second parameter `a` corresponds to the functor argument; it is needed to access the associated type synonym `T` of the `EqSig` class in the body of `MkSetSig`. (Now it should become clear why we removed the `elem` type component from the functor body: If `E.t` did not appear in a value specification in the functor body, the necessity for the class parameter `a` would not occur.) Note that we cannot reuse the names `Set`, `empty`, `member`, and `insert` because type synonyms and class methods share a global namespace in Haskell.

The instance of `MkSetSig` for the fresh data type `MkSet` and some type variable `a` is the translation of the functor body. The constraint `EqSig a` in the instance

[1] Interestingly, this idea goes back to ML's `abstype` feature, which is nowadays essentially deprecated; a similar feature is also implemented in the Haskell interpreter Hugs (Jones & Peterson, 1999). In contrast to abstract associated type synonyms, the scope of the concrete definition of an abstract type has to be specified explicitly with these approaches.

ML

```
functor MkSet (E : sig type t  val eq : t -> t -> bool end) =
  struct
    type set        = E.t list
    val  empty      = []
    fun  member x s = List.exists (E.eq x) s
    fun  insert x s = if member x s then s else (x :: s)
  end :> sig
           type set
           val  empty  : set
           val  member : E.t -> set -> bool
           val  insert : E.t -> set -> set
         end
```

Haskell

```
class EqSig a where
    type T a
    eq :: a -> T a -> T a -> Bool
class EqSig a => MkSetSig 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 = MkSet
instance EqSig a => MkSetSig MkSet a where
    abstype Set' MkSet a  = [T a]
    empty'  _ _           = []
    member' _ a x s       = any (eq a x) s
    insert' _ a x s       = if member' MkSet a x s then s else (x : s)
```

Figure 3. ML functor for generic sets and its translation to Haskell

context is necessary because we use the associated type synonym `T` and the method `eq` in the instance body.

Figure 4 shows how we use the `MkSet` functor to construct a set implementation for strings. To translate the functor invocation to Haskell, we first define an appropriate `EqSig` instance. Then we define a type class `StringSetSig`, which corresponds to the (implicit) signature of the ML structure `StringSet`, and make the new type `StringSet` an instance of `StringSetSig`.

2.2 From classes to modules

The translation from Haskell type classes to ML modules encodes type classes as signatures and instances of type classes as functors that yield structures of these signatures. It makes use of two extensions to Standard ML, both of which are implemented in Moscow ML (Romanenko *et al.*, 2003): recursive functors (Crary *et al.*, 1999; Russo, 2001) are used to model recursive instance definitions, and first-class structures (Russo, 2000a) serve as dictionaries providing runtime evidence for type-class constraints.

ML

```
structure StringSet = MkSet (struct type t = string  val eq = stringEq end)
```

Haskell

```
data StringEq = StringEq
instance EqSig StringEq where
    type T StringEq = String
    eq   _          = stringEq
class StringSetSig a where
    type Set'' a
    empty''    :: a -> Set'' a
    member''   :: a -> String -> Set'' a -> Bool
    insert''   :: a -> String -> Set'' a -> Set'' a
data StringSet = StringSet
instance StringSetSig StringSet where
    type Set'' StringSet = Set'    MkSet StringEq
    empty''    _         = empty'  MkSet StringEq
    member''   _         = member' MkSet StringEq
    insert''   _         = insert' MkSet StringEq
```

Figure 4. Functor invocation in ML and its translation to Haskell

We first explain the usage of first-class structures as dictionaries (Section 2.2.1). Then we make the ideas of the translation more concrete by showing ML encodings of type class definitions (Section 2.2.2), of overloaded functions (Section 2.2.3), and of instance definitions (Section 2.2.4).

2.2.1 First-class structures as dictionaries

Dictionary translation (Wadler & Blott, 1989; Jones, 1994; Hall *et al.*, 1996; Faxén, 2002) is a technique frequently used to eliminate overloading introduced by type classes. Using this technique, type-class constraints are turned into extra parameters, so that evidence for these constraints can be passed explicitly at runtime. Evidence for a constraint comes as a dictionary that provides access to all methods of the constraint's type class.

The translation from Haskell type classes to ML modules is another application of dictionary translation. In our case, dictionaries are represented as first-class structures (Russo, 2000a), an extension to Standard ML that allows structures to be manipulated in the core language. (ML's core language comprises fundamental constructs such as primitive types and operations, functions, and binding facilities; the module language comprises constructs such as signatures, structures, and functors for decomposing programs into smaller units.)

Explicit conversion code is necessary to coerce a structure into a first-class structure and vice versa. Suppose S is a signature, and s is a structure of signature S. Then the construct `pack` s `as` S turns s into a first-class structure of type $<S>$. Such types are called package types. Conversely, the construct `open` e_1 `as` $X : S$ `in` e_2,

Haskell

```
class Eq a where (==) :: a -> a -> Bool
class Eq a => Ord a where (<) :: a -> a -> a
```

ML

```
signature Eq  = sig type t  val eq : t -> t -> bool end
signature Ord = sig
                  type t
                  val  lt      : t -> t -> t
                  val  superEq : <Eq where type t = t>
                end
```

Figure 5. Haskell type classes `Eq` and `Ord` and their translations to ML

Haskell

```
elem :: Eq a => a -> [a] -> Bool
elem x l = any ((==) x) l
```

ML

```
fun elem d (x:'a) l = open d as D : (Eq where type t = 'a)
                      in List.exists (D.eq x) l
```

Figure 6. Overloaded function in Haskell and its translation to ML

where the expression e_1 is expected to have type $<S>$, makes the structure contained in e_1 available in e_2 under the name X.

2.2.2 Translating type class definitions

Figure 5 shows two Haskell type classes `Eq` and `Ord`, which provide overloaded operators `==` and `<`. We translate these classes into ML signatures of the same name. Thereby, the type variable `a` in the class head is mapped to an opaque type specification `t`, and the methods of the class are translated into value specifications. The signature `Ord` has an additional value specification `superEq` to account for the superclass `Eq` of `Ord`. Consequently, `superEq` has type `<Eq where type t = t>` which represents a dictionary for `Eq` at type `t`.

2.2.3 Translating overloaded functions

Figure 6 shows the Haskell function `elem`, which uses the `==` operator of class `Eq`. Therefore, the constraint `Eq a` needs to be added to the (optional) type annotation of `elem` to limit the types that can be substituted for `a` to instances of `Eq`.

As already noted in Section 2.2.1, such a constraint is represented in the ML version of `elem` as an additional parameter `d` which abstracts explicitly over the dictionary for the constraint `Eq a`. Hence, the type of `elem` in ML is `<Eq where type t = 'a> -> 'a -> 'a list -> bool`.

Haskell

```
instance Eq Int where (==) = intEq
instance Ord Int where (<) = intLt
instance Eq a => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _      == _      = False
```

ML

```
functor EqInt  () = struct type t = int  val eq = intEq end
functor OrdInt () = struct
                      type t       = int
                      val  lt      = intLt
                      val  superEq = pack EqInt () as (Eq where type t = t)
                    end
rec functor EqList (X: Eq) : (Eq where type t = X.t list) =
      struct
        type t = X.t list
        fun  eq  []       []      = true
           | eq  (x::xs)  (y::ys) =
               open
                 pack EqList (X) as (Eq where type t = t)
               as Y : (Eq where type t = t)
               in X.eq x y && Y.eq xs ys
           | eq  _        _       = false
      end
```

Figure 7. Instance definitions in Haskell and their translations to ML

In the body of `elem`, we open the first-class structure `d` and bind the content to the structure variable `D`, so that we can access the equality comparison function as `D.eq`. Note that we cannot do without the type annotation `(x:'a)`: It introduces the lexically scoped type variable `'a` used in the signature required for opening `d`. (Lexically scoped type variables are part of Standard ML.)

2.2.4 Translating instance definitions

The last of our examples is the translation of instance definitions. The Haskell code in Figure 7 makes the type `Int` an instance of the type classes `Eq` and `Ord`. Furthermore, it specifies that lists can be compared for equality as long as the elements of the list can be compared for equality. This requirement is expressed as the constraint `Eq a` in the context of the instance definition for `Eq [a]`. (The constraints to the left of the double arrow `=>` are called *context*, whereas the part to the right is called *head*. The double arrow is omitted if the context is empty.)

The functors `EqInt` and `OrdInt` are translations of the instances `Eq Int` and `Ord Int`, respectively. These two functors do not take any arguments because the contexts of the corresponding instance definitions are empty. (We could use structures instead of functors in such cases; however, for reasons of consistency we decided to

use functors even if the instance context is empty.) The definition of the `superEq` component in `OrdInt` demonstrates that dictionaries are created by coercing structures into first-class structures.

The translation of the instance definition for `Eq [a]` is more interesting because the Haskell version is recursive (through the expression `xs == ys` in the second equation) and has a non-empty context. Consequently, the functor `EqList` is defined recursively and takes an argument `X` of signature `Eq` which corresponds to the constraint `Eq a` in the instance context.

Recursive functors (Crary *et al.*, 1999; Russo, 2001), an extension to Standard ML, are introduced with the keyword `rec;` the result signature is mandatory. The recursiveness of `EqList` is needed for the second case of the definition of `eq:` We invoke the functor recursively, pack the result as a first-class structure, immediately open this structure again, and bind the result to the variable `Y.` Now we can use `Y.eq` to compare `xs` and `ys` for equality. The combination of `pack`/`open` operations is necessary to interleave computations in the core language with computations in the module language; it is not possible to write `EqList(X).eq` directly.

It may seem awkward to use recursive functors in ML to encode recursive Haskell functions. Indeed, for the example just discussed, a recursive ML function would be sufficient. In general, however, it is possible to write polymorphic recursive functions (Henglein, 1993) with Haskell type classes. For such cases, we definitely need to encode recursion in terms of recursive functors because polymorphic recursion is not available in the core language of Standard ML.

3 Formal Translation From Modules to Classes

After having presented several examples in the preceding section, we now develop a formal translation from ML modules to Haskell type classes. The translation from Haskell to ML is discussed afterwards in Section 4.

First, Section 3.1 defines the syntax and static semantics of the source language Tiny-ML, a small language that captures most of the essential features of Standard ML's module system. Section 3.2 continues by presenting the syntax and the type system of our target language Tiny-HS$^+$, which supports Haskell 98 type classes with some extensions. We then define in Section 3.3 a formal translation from Tiny-ML to Tiny-HS$^+$. Section 3.4 proves that every well-typed Tiny-ML program translates into a well-typed Tiny-HS$^+$ program. Finally, Section 3.5 discusses if and how the translation could handle extensions to Tiny-ML. We use the remainder of this section to explain some notational conventions used in the rest of the text.

An implementation of the translation is available from `http://www.informatik.uni-freiburg.de/~wehr/diplom/`. We do not discuss it here because the implementation techniques are standard (Wehr, 2005).

Overbar notation. The notation $\overline{x}^n$ stands for $x_1, \ldots, x_n$. We often omit the superscript n and simply write $\overline{x}$. Furthermore, $\overline{x_i}^{i \in \{i_1, \ldots i_n\}}$ is an abbreviation for $x_{i_1}, \ldots, x_{i_n}$. The separator of such enumerations might depend on the context.

Identifiers

$$\text{`a} \in \text{SimTypVar};\quad \text{c} \in \text{CoreId};\quad \text{T} \in \text{TyconId} = \{\rightarrow, \text{int}, \dots\}$$
$$\text{t} \in \text{TypId};\quad \text{x}, \text{y} \in \text{ValId};\quad \text{X} \in \text{StrId};\quad \text{F} \in \text{FunId}$$

Types

$$\begin{array}{rcl} \text{u} & ::= & \text{`a} \mid \text{T}^{\kappa}\,\overline{\text{u}}^{\kappa} \mid \text{t} \mid \text{X.t} \\ \text{v} & ::= & \forall \text{A.u} \\ \text{A}, \text{B} & \in & \textsc{Fin}(\text{SimTypVar}) \end{array}$$

Expressions

$$\text{e} ::= \text{c} \mid \lambda \text{c.e} \mid \text{e}\,\text{e} \mid \textbf{let}\ \text{c} = \text{e}\ \textbf{in}\ \text{e} \mid \text{x} \mid \text{X.x}$$

Signatures

$$\begin{array}{rcl} \text{B} & ::= & \textbf{type}\ \text{t}; \text{B} \mid \textbf{type}\ \text{t} = \text{u}; \text{B} \mid \textbf{val}\ \text{x} : \text{v}; \text{B} \mid \epsilon_{\text{B}} \\ \text{S} & ::= & \textbf{sig}\ \text{B}\ \textbf{end} \end{array}$$

Structures

$$\begin{array}{rcl} \text{b} & ::= & \textbf{type}\ \text{t} = \text{u}; \text{b} \mid \textbf{val}\ \text{x} = \text{e}; \text{b} \mid \epsilon_{\text{b}} \\ \text{ps} & ::= & \textbf{struct}\ \text{b}\ \textbf{end} \\ \text{s} & ::= & \text{ps} \mid \text{X} \mid \text{F}(\overline{\text{X}}) \end{array}$$

Programs

$$\text{prog} ::= \textbf{structure}\ \text{X} :> \text{S} = \text{s}; \text{prog} \mid \textbf{functor}\ \text{F}(\overline{\text{X} : \text{S}}) :> \text{S} = \text{ps}; \text{prog} \mid \epsilon_{\text{prog}}$$

Figure 8. Syntax of Tiny-ML

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

Finite maps. A finite map F from M to N is written $\{\overline{a_i \mapsto b_i}^{\,i \in [n]}\}$ where $a_i \in M$, $a_i \neq a_j$ for $i \neq j$, and $b_i \in N$; $\textsc{Dom}(F) \subseteq M$ and $\textsc{Img}(F) \subseteq N$ denote the domain and image of F, respectively. The set of all finite maps between M and N is written $M \xrightarrow{\text{fin}} N$. We use operations such as $\subseteq$, $\dot\cup$, and $\setminus$ on finite maps in the obvious way. Additionally, we define $F \mathbin{\overrightarrow{\cup}} G$ as the finite map $\{a \mapsto G(a) \mid a \in \textsc{Dom}(G)\} \dot\cup \{a \mapsto F(a) \mid a \in \textsc{Dom}(F) \setminus \textsc{Dom}(G)\}$ and $\textsc{Zip}(\{\overline{a}^m\}, \{\overline{b}^n\})$ as the finite map $\{\overline{a_i \mapsto b_i}^{\,i \in [\min(m,n)]}\}$. The notation $F, a \mapsto b$ is an abbreviation for $F \mathbin{\overrightarrow{\cup}} \{a \mapsto b\}$.

3.1 Tiny-ML

Tiny-ML is a subset (modulo some minor syntactic differences) of Mini-SML (Russo, 1998), which is in turn a simplified version of Standard ML. However, whereas Mini-SML models all important features of Standard ML's module language, Tiny-ML 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.[2] These features have been omitted to keep the translation from Tiny-ML to Tiny-HS$^+$ manageable (see Section 3.5).

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

3.1.1 Syntax

The syntax of Tiny-ML is shown in Figure 8. The identifiers sets introduced in this figure, as well as all other identifier sets introduced in the rest of this article, are assumed to be pair-wise disjoint and countably infinite. We write T^{κ} for a type constructor T of kind $\kappa \in \mathbb{N}$. It is sufficient to use natural number for kinds because higher-order types are not supported in Tiny-ML (and in Standard ML).

The type language of Tiny-ML distinguishes between simple types u and value types v. Simple types may refer to type components of structures, written t and $\mathsf{X.t}$. A value type is a simple type universally quantified over a finite set of simple type variables ‘a. The syntax of expressions is standard. Expressions can refer to value components of structures using the notation x and $\mathsf{X.x}$.

We identify types and expressions that differ only in their bound variables. Moreover, we often omit an empty set of universally quantified type variables; that is, we write u instead of $\forall\emptyset.\mathsf{u}$. The sets SimTyp and ValExp range over simple types and expressions, respectively.

Signature bodies B may contain opaque and transparent type components, written **type** t and **type** $\mathsf{t} = \mathsf{u}$, respectively, as well as value components, written **val** $\mathsf{x} : \mathsf{v}$. A signature expression S is just an encapsulated signature body. The set SigExp ranges over all S. The body b of a structure consists of a series of type and value components. Structure expressions come in two different forms: primitive structure expressions ps and (non-primitive) structure expressions s. The restricted structure expression language is justified because the translation to Haskell type classes requires that signature sealing is restricted to top-level structure and functor definitions, and that the right-hand side of a functor definition is a primitive structure expression. We omit structure and functor definitions without signature seals for simplicity.

Simple types, value types, and expressions are collectively referred to as the *core language* of Tiny-ML. Conversely, the *module language* encompasses signatures, structures, and functors.

3.1.2 Semantic Objects

The definition of Standard ML (Milner *et al.*, 1997) distinguishes between syntactic types and their semantic counterparts, which are called *semantic objects*. Tiny-ML follows the same approach; the definition of semantic objects is shown in Figure 9. We let $\mathcal{O}$ range over all semantic objects. In order to distinguish between syntactic constructs and semantic objects, we use an *italic font* for semantic objects.

Semantic objects introduce two new sorts of type variables. Semantic simple type variables ‘$a \in \mathit{SimTypVar}$ are the semantic counterpart of (syntactic) simple type variables ‘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.

Semantic types u and v correspond to simple and value types, respectively, except that type occurrences t and $\mathsf{X.t}$ are now represented by the right-hand sides of their

Identifiers

$$`a \in \mathit{SimTypVar}; \quad \alpha \in \mathit{TypVar}$$

Semantic types

$$\begin{array}{rcl} u & ::= & `a \mid T^{\kappa}\,\overline{u}^{\kappa} \mid \alpha \\ v & ::= & \forall A.u \\ A, B & \in & \textsc{Fin}(\mathit{SimTypVar}) \end{array}$$

Semantic structures, signatures, and functors

$$\begin{array}{rcl} \mathcal{S} & := & \{\mathcal{S}_{\mathrm{t}} \cup \mathcal{S}_{\mathrm{x}} \mid \mathcal{S}_{\mathrm{t}} \in \mathrm{TypId} \xrightarrow{\text{fin}} \mathit{SimTyp}, \mathcal{S}_{\mathrm{x}} \in \mathrm{ValId} \xrightarrow{\text{fin}} \mathit{ValTyp}\} \\ \mathcal{X} & ::= & \exists P.\mathcal{S} \\ \mathcal{L} & ::= & \Lambda P.\mathcal{S} \\ \mathcal{F} & ::= & \forall P.\overline{\mathcal{S}} \to \mathcal{X} \\ P, Q & \in & \textsc{Fin}(\mathit{TypVar}) \end{array}$$

Contexts

$$\mathcal{C} \quad := \quad \left\{ \begin{array}{c|l} \mathcal{C}_{\mathrm{c}} \cup \mathcal{C}_{`\mathrm{a}} \cup & \mathcal{C}_{\mathrm{c}} \in \mathrm{CoreId} \xrightarrow{\text{fin}} \mathit{ValTyp}, \mathcal{C}_{`\mathrm{a}} \in \mathrm{SimTypVar} \xrightarrow{\text{fin}} \mathit{SimTyp}, \\ \mathcal{C}_{\mathrm{t}} \cup \mathcal{C}_{\mathrm{x}} \cup & \mathcal{C}_{\mathrm{t}} \in \mathrm{TypId} \xrightarrow{\text{fin}} \mathit{SimTyp}, \mathcal{C}_{\mathrm{x}} \in \mathrm{ValId} \xrightarrow{\text{fin}} \mathit{ValTyp}, \\ \mathcal{C}_{\mathrm{X}} \cup \mathcal{C}_{\mathrm{F}} & \mathcal{C}_{\mathrm{X}} \in \mathrm{StrId} \xrightarrow{\text{fin}} \mathit{Str}, \mathcal{C}_{\mathrm{F}} \in \mathrm{FunId} \xrightarrow{\text{fin}} \mathit{Fun} \end{array} \right\}$$

Figure 9. Semantic objects for Tiny-ML

definitions (if available) or by semantic type variables. We let the sets *SimTyp* and *ValTyp* range over all u and v, respectively.

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, $\mathrm{x} \in \textsc{Dom}(\mathcal{S})$ ranges only over the value identifiers of $\textsc{Dom}(\mathcal{S}_{\mathrm{x}})$. The set *Str* ranges over all $\mathcal{S}$.

Existential semantic structures $\mathcal{X} = \exists P.\mathcal{S}$ are the types of structure expressions; the existentially quantified semantic type variables represent the abstract types introduced by the structure expression. Semantic signatures $\mathcal{L} = \Lambda Q.\mathcal{S}$ are the semantic counterpart of signature expressions. The parameters in Q stem from opaque type specifications. Semantic functors $\mathcal{F} = \forall P.\overline{\mathcal{S}} \to \mathcal{X}$ are types for functors. They are universally quantified because functors are polymorphic in the opaque type components of their argument signatures. The set *Fun* ranges over all $\mathcal{F}$.

A context $\mathcal{C}$ records information connected with various sort of identifiers. As with semantic structures, we often omit the subscript used to distinguish to different parts of a context when it is clear which part we mean. We finish this section with several definitions.

Definition 3.1 (Substitutions)**.** A substitution φ from semantic type variables to semantic simple types is an element of $\mathit{TypVar} \xrightarrow{\text{fin}} \mathit{SimTyp}$, written $[\overline{u/\alpha}]$. A substitution ϕ from semantic simple type variables to semantic simple types is an element of $\mathit{SimTypVar} \xrightarrow{\text{fin}} \mathit{SimTyp}$, written $[\overline{u/`a}]$. Substitution application is defined in the usual, capture-avoiding way.

Definition 3.2 (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 $\textsc{Dom}(\phi) = A$ such that $\phi(u) = u'$.

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

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

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

Classification of expressions $\boxed{\mathcal{C} \vdash \mathrm{e} : u \quad \mathcal{C} \vdash \mathrm{e} : v}$

$$\frac{\mathcal{C}(\mathrm{c}) = v \qquad v \succ u}{\mathcal{C} \vdash \mathrm{c} : u} \qquad \frac{\mathcal{C}(\mathrm{x}) = v \qquad v \succ u}{\mathcal{C} \vdash \mathrm{x} : u} \qquad \frac{\mathcal{C}(\mathrm{X})(\mathrm{x}) = v \qquad v \succ u}{\mathcal{C} \vdash \mathrm{X.x} : u}$$

$$\frac{\mathcal{C}, \mathrm{c} \mapsto u \vdash \mathrm{e} : u'}{\mathcal{C} \vdash \lambda \mathrm{c.e} : u \to u'} \qquad \frac{\mathcal{C} \vdash \mathrm{e}_1 : u_1 \to u_2 \qquad \mathcal{C} \vdash \mathrm{e}_2 : u_1}{\mathcal{C} \vdash \mathrm{e}_1\ \mathrm{e}_2 : u_2} \qquad \frac{\mathcal{C} \vdash \mathrm{e}_1 : v \qquad \mathcal{C}, \mathrm{c} \mapsto v \vdash \mathrm{e}_2 : u}{\mathcal{C} \vdash \textbf{let}\ \mathrm{c} = \mathrm{e}_1\ \textbf{in}\ \mathrm{e}_2 : u}$$

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

Figure 10. Typing judgments for Tiny-ML's core language

Definition 3.3 (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:

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

Enrichment is a pre-order closed under substitution (Milner, 1978; Russo, 1998). For value types, enrichment coincides with the definition of generic instances given by Damas and Milner (Damas & Milner, 1982).

Definition 3.4 (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 $\textsc{Dom}(\varphi) = P$ such that $\mathcal{S} \succcurlyeq \varphi(\mathcal{S}')$.

Definition 3.5 (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 $\mathrm{FV}^{\text{`}a}(\mathcal{O}) \subseteq \mathit{SimTypVar}$. We write the set of structure variables free in some syntactic construct $\mathbf{O}$ as $\mathrm{FV}^{\mathrm{X}}(\mathbf{O}) \subseteq \mathsf{StrId}$. Similarly, $\mathrm{FV}^{\mathrm{c}}(\mathbf{O}) \subseteq \mathsf{CoreId}$ denotes the set of core variables free in $\mathbf{O}$.

3.1.3 Static Semantics

We now specify the static semantics of Tiny-ML. The typing rules in this section are standard (Russo, 1998).

Figure 10 contains the typing judgments for Tiny-ML's core language. The judgments $\mathcal{C} \vdash \mathrm{u} \triangleright u$ and $\mathcal{C} \vdash \mathrm{v} \triangleright v$ relate syntactic types to their semantic counterparts;

Denotation of signatures $\boxed{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L} \quad \mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}}$

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

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

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

Classification of structures $\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{S} \quad \mathcal{C} \vdash \mathrm{ps} : \mathcal{X} \quad \mathcal{C} \vdash \mathrm{s} : \mathcal{X}}$

$$\frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \qquad \mathcal{C}, \mathrm{t} \mapsto u \vdash \mathrm{b} : \mathcal{S} \qquad \mathrm{t} \notin \textsc{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b} : \mathcal{S}, \mathrm{t} \mapsto u} \qquad \frac{\mathcal{C} \vdash \mathrm{e} : v \qquad \mathcal{C}, \mathrm{x} \mapsto v \vdash \mathrm{b} : \mathcal{S} \qquad \mathrm{x} \notin \textsc{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} : \mathcal{S}, \mathrm{x} \mapsto v} \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \emptyset}$$

$$\frac{\mathcal{C} \vdash \mathrm{b} : \mathcal{S}}{\mathcal{C} \vdash \textbf{struct}\ \mathrm{b}\ \textbf{end} : \exists\emptyset.\mathcal{S}} \qquad \frac{\mathcal{C}(\mathrm{X}) = \mathcal{S}}{\mathcal{C} \vdash \mathrm{X} : \exists\emptyset.\mathcal{S}} \qquad \frac{\mathcal{C}(\mathrm{X}_i) = \mathcal{S}_i \qquad \mathcal{C}(\mathrm{F}) = \forall Q.\overline{\mathcal{S}'}^{n} \to \mathcal{X} \qquad \mathcal{S}_i \succcurlyeq \varphi(\mathcal{S}'_i) \qquad \textsc{Dom}(\varphi) = Q \qquad (i \in [n])}{\mathcal{C} \vdash \mathrm{F}(\overline{\mathrm{X}}^{n}) : \varphi(\mathcal{X})}$$

Typing of programs $\boxed{\mathcal{C} \vdash \mathrm{prog}}$

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

$$\frac{\mathcal{C} \vdash_{fargs} \overline{\mathrm{X} : \mathrm{S}} \triangleright \forall P.\overline{\mathcal{S}} \qquad \mathcal{C}, \overline{\mathrm{X} \mapsto \mathcal{S}} \vdash_{seal} \mathrm{ps} :> \mathrm{S} : \mathcal{X} \qquad \mathrm{F} \notin \textsc{Dom}(\mathcal{C}) \qquad \mathcal{C}, \mathrm{F} \mapsto (\forall P.\overline{\mathcal{S}} \to \mathcal{X}) \vdash \mathrm{prog}}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}(\overline{\mathrm{X} : \mathrm{S}}) :> \mathrm{S} = \mathrm{ps}; \mathrm{prog}} \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{prog}}}$$

Auxiliaries $\boxed{\mathcal{C} \vdash_{seal} \mathrm{s} :> \mathrm{S} : \mathcal{X} \quad \mathcal{C} \vdash_{fargs} \overline{\mathrm{X} : \mathrm{S}} \triangleright \forall P.\overline{\mathcal{S}}}$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P'.\mathcal{S}' \qquad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \qquad P' \cap \mathrm{FV}^{\alpha}(\Lambda P.\mathcal{S}) = \emptyset \qquad \mathcal{S}' \succcurlyeq \varphi(\mathcal{S}) \qquad \textsc{Dom}(\varphi) = P}{\mathcal{C} \vdash_{seal} \mathrm{s} :> \mathrm{S} : \exists P.\mathcal{S}}$$

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

Figure 11. Typing judgments for Tiny-ML's module language

the judgments $\mathcal{C} \vdash \mathrm{e} : u$ and $\mathcal{C} \vdash \mathrm{e} : v$ assign monomorphic and polymorphic, respectively, semantic types to expressions.

The typing judgments for the module language are shown in Figure 11. The judgments $\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}$ and $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$ relate signature bodies and signature expressions to semantic signatures. The rule for opaque type components is particularly interesting because it introduces new semantic type variables. The judgment $\mathcal{C} \vdash \mathrm{b} : \mathcal{S}$ assigns semantic structures to structure bodies, the judgments $\mathcal{C} \vdash \mathrm{ps} : \mathcal{X}$ and $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ assigns existential semantic structures to (primitive) structure expressions. The rule for functor application use the enrichment relation $\succcurlyeq$ to ensure proper signature

Identifiers

$\mathsf{C} \in \mathsf{ClassId}; \quad \mathsf{T} \in \mathsf{TyconId} = \{\rightarrow, \mathsf{Int}, \ldots\}; \quad \mathsf{S} \in \mathsf{ASynId}; \quad \mathsf{z} \in \mathsf{VarId}; \quad \mathsf{a}, \mathsf{b} \in \mathsf{TypVar}$

Types

$$\begin{array}{rcl} \tau & ::= & \mathsf{a} \mid \mathsf{T}^{\kappa}\,\overline{\tau}^{\kappa} \mid \eta \\ \eta & ::= & \mathsf{S}^{\kappa}\,\overline{\tau}^{\kappa} \\ \rho & ::= & \overline{\pi} \Rightarrow \tau \\ \sigma & ::= & \forall \mathsf{A}.\rho \\ \mathsf{A}, \mathsf{B} & \in & \textsc{Fin}(\mathsf{TypVar}) \end{array}$$

Constraints

$$\begin{array}{rcl} \pi & ::= & \mathsf{C}\,\overline{\tau} \mid \eta = \tau \\ \theta & ::= & \forall \mathsf{A}.\overline{\pi} \Rightarrow \mathsf{C}\,\overline{\tau} \mid \forall \mathsf{A}.\eta = \tau \end{array}$$

Expressions

$$\mathsf{w} ::= \mathsf{z} \mid \lambda \mathsf{z}.\mathsf{w} \mid \mathsf{w}_1\ \mathsf{w}_2 \mid \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2 \mid \mathsf{w} :: \sigma$$

Data types, classes, instances, and programs

$$\begin{array}{rcl} \mathsf{ddec} & ::= & \textbf{data}\ \mathsf{T} \\ \mathsf{cls} & ::= & \textbf{class}\ \forall \mathsf{A}.\overline{\mathsf{C}\,\mathsf{a}} \Rightarrow \mathsf{C}\,\mathsf{a}\ \textbf{where}\ \overline{\mathsf{tdec}}\ \ \overline{\mathsf{msig}} \\ \mathsf{tdec} & ::= & \textbf{type}\ \mathsf{S}^{\kappa}\,\overline{\mathsf{a}}^{\kappa} \\ \mathsf{msig} & ::= & \mathsf{m} :: \forall \mathsf{A}.\tau \\ \mathsf{inst} & ::= & \textbf{instance}\ \forall \mathsf{A}.\overline{\mathsf{C}\,\mathsf{a}} \Rightarrow \mathsf{C}\,\overline{\tau}\ \textbf{where}\ \overline{\mathsf{tdef}}\ \ \overline{\mathsf{mval}} \\ \mathsf{tdef} & ::= & \textbf{type}\ \mathsf{S}^{\kappa}\,\overline{\tau}^{\kappa} = \tau \mid \textbf{abstype}\ \mathsf{S}^{\kappa}\,\overline{\tau}^{\kappa} = \tau \\ \mathsf{mval} & ::= & \mathsf{m} = \mathsf{w} \\ \mathsf{pgm} & ::= & \overline{\mathsf{ddec}}\ \overline{\mathsf{cls}}\ \overline{\mathsf{inst}} \end{array}$$

Figure 12. Syntax of Tiny-HS$^+$

matching. Finally, the judgment $\mathcal{C} \vdash \mathsf{prog}$ states that the program prog is well-typed under context $\mathcal{C}$. It makes use of two auxiliary judgments $\mathcal{C} \vdash_{\mathit{seal}} \mathrm{s} :> \mathrm{S} : \mathcal{X}$ and $\mathcal{C} \vdash_{\mathit{fargs}} \overline{\mathrm{X} : \mathrm{S}} \triangleright \forall P.\overline{\mathcal{S}}$. The first judgment uses the enrichment relation to ensure that the structure expression s matches the prescribed signature S. The second judgment relates functor argument signatures to their denotation.

3.2 Tiny-HS$^+$

We now define Tiny-HS$^+$, the target language of our translation from ML modules to Haskell type classes. Tiny-HS$^+$ features type classes in the style of Haskell 98 extended with multi-parameter type classes (Peyton Jones *et al.*, 1997), associated type synonyms (Chakravarty *et al.*, 2005a), and abstract associated type synonyms. The latter are a contribution of our work; we already introduced them informally in Section 2.1.2.

3.2.1 Syntax

The syntax of Tiny-HS$^+$ is shown in Figure 12. There are three different forms of types: mono types τ, which may have the form of associated types η; qualified types ρ, which attach some constraints $\overline{\pi}$ to a mono type; and type schemes σ. We often write τ instead of $\Rightarrow \tau$ and ρ instead of $\forall\emptyset.\rho$. As in Tiny-ML, kinds κ are natural numbers (higher-order type constructors are not need for the translation).

We require that every Tiny-ML type constructor T^κ has a counterpart T^κ in Tiny-HS. The sets Typ, ATyp, and TypSc range over all τ, η, and σ, respectively.

A constraint π is either a type class constraint or an equality constraint. Constraints can be generalized to constraint schemes θ. The sets Constr and $\mathsf{ConstrSc}$ range over all π and θ, respectively. The syntax of expressions w is standard. We often write $\lambda\_.\mathsf{w}$ instead of $\lambda\mathsf{z}.\mathsf{w}$ provided z is not free in w. The set Exp ranges over all w.

A user-defined type constructor T is introduced through the definition **data** T. There are no value constructors for T because the translation uses such type constructors only in the form $\bot :: \mathsf{T}$. The definition of a type class C may also contain declarations of associated type synonyms. Consequently, instance definitions for C must provide implementations for the associated type synonyms of C. Such an implementation may either be concrete (**type**) or abstract (**abstype**). The effect of using **abstype** is that the type equality $\mathsf{S}^\kappa\,\overline{\tau}^\kappa = \tau'$ is only visible within the very instance definition.

3.2.2 Static Semantics

This section defines a type system for Tiny-HS$^+$. Before discussing the typing rules, we give some general definitions.

Definition 3.6 (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.

Definition 3.7 (Substitutions)**.** A substitution ψ from type variables to types is an element of $\mathsf{TypVar} \xrightarrow{\text{fin}} \mathsf{Typ}$, written $[\overline{\tau/\mathsf{a}}]$. Substitution application is defined in the usual, capture-avoiding way.

Definition 3.8 (Well-formed constraint schemes)**.** A constraint scheme $\theta = \forall\mathsf{A}.\ \overline{\pi} \Rightarrow \pi'$ is said to be well-formed if $\mathrm{FV}^{\mathsf{a}}(\overline{\pi}) \subseteq \mathrm{FV}^{\mathsf{a}}(\pi') = \mathsf{A}$.

Definition 3.9 (Tiny-HS$^+$ environments)**.** A variable environment $\Gamma \in \mathsf{VarId} \xrightarrow{\text{fin}} \mathsf{TypSc}$ is a mapping between term variables and type schemes. A constraint environment $\Theta \in \textsc{Fin}(\mathsf{Constr}) \cup \textsc{Fin}(\mathsf{ConstrSc})$ records constraints and constraint schemes.

Definition 3.10 (Superclasses for Tiny-HS$^+$)**.** The set of immediate superclasses of $\mathsf{C}\,\overline{\tau}$ in Θ is $\textsc{Sup}(\Theta, \mathsf{C}\,\overline{\tau}) := \{[\overline{\tau/\mathsf{a}}](\mathsf{C}'\,\overline{\mathsf{b}}) \mid \forall\mathsf{A}.\mathsf{C}\,\overline{\mathsf{a}} \Rightarrow \mathsf{C}'\,\overline{\mathsf{b}} \in \Theta\}$.

The first half of the typing judgments for Tiny-HS$^+$ are shown in Figure 13. The rules are standard (Chakravarty *et al.*, 2005a). The predicate $\Theta \vdash \tau$ states that τ is well-formed in the constraint environment Θ; the first rule ensures that an associated type synonym is applied only to types for which Θ is strong enough to derive the class declaring the synonym. The judgment $\Theta \Vdash \pi$ defines an entailment relation which makes this precise; that is, $\Theta \Vdash \pi$ holds whenever π is derivable from Θ. The last judgment in Figure 13, $\Theta; \Gamma \vdash \mathsf{w} : \sigma$, assigns an expression w the type scheme σ under the environments Θ and Γ. The second rule of this judgment is the most interesting one because it incorporates nonsyntactic type equalities.

Well-formedness $\boxed{\Theta \vdash \tau}$

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

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

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

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

$$\frac{}{\Theta \Vdash \tau = \tau} \qquad \frac{\Theta \Vdash \tau_2 = \tau_1}{\Theta \Vdash \tau_1 = \tau_2} \qquad \frac{\Theta \Vdash \tau_1 = \tau_2 \qquad \Theta \Vdash \tau_2 = \tau_3}{\Theta \Vdash \tau_1 = \tau_3} \qquad \frac{\Theta \Vdash [\tau_1/\mathsf{a}]\pi \qquad \Theta \Vdash \tau_1 = \tau_2}{\Theta \Vdash [\tau_2/\mathsf{a}]\pi}$$

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

$$\frac{\Gamma(\mathsf{z}) = \sigma}{\Theta; \Gamma \vdash \mathsf{z} : \sigma} \qquad \frac{\Theta; \Gamma \vdash \mathsf{w} : \tau' \qquad \Theta \Vdash \tau' = \tau}{\Theta; \Gamma \vdash \mathsf{w} : \tau} \qquad \frac{\Theta; \Gamma \vdash \mathsf{w}_1 : \tau' \to \tau \qquad \Theta; \Gamma \vdash \mathsf{w}_2 : \tau'}{\Theta; \Gamma \vdash \mathsf{w}_1\ \mathsf{w}_2 : \tau}$$

$$\frac{\Theta; \Gamma, \mathsf{z} \mapsto \tau' \vdash \mathsf{w} : \tau \qquad \Theta \vdash \tau'}{\Theta; \Gamma \vdash \lambda \mathsf{z}.\mathsf{w} : \tau' \to \tau} \qquad \frac{\Theta; \Gamma \vdash \mathsf{w}_1 : \sigma' \qquad \Theta; \Gamma, \mathsf{z} \mapsto \sigma' \vdash \mathsf{w}_2 : \sigma}{\Theta; \Gamma \vdash \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2 : \sigma}$$

$$\frac{\Theta; \Gamma \vdash \mathsf{w} : \sigma \qquad \mathrm{FV}^{\mathsf{a}}(\sigma) = \emptyset}{\Theta; \Gamma \vdash (\mathsf{w} :: \sigma) : \sigma} \qquad \frac{(\Theta \cup \{\pi\}); \Gamma \vdash \mathsf{w} : \rho}{\Theta; \Gamma \vdash \mathsf{w} : \pi \Rightarrow \rho} \qquad \frac{\Theta; \Gamma \vdash \mathsf{w} : \pi \Rightarrow \rho \qquad \Theta \Vdash \pi}{\Theta; \Gamma \vdash \mathsf{w} : \rho}$$

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

Figure 13. Well-formedness, entailment, and expression-typing judgments for Tiny-HS^{+}

The judgments for classes, instances, and programs in Figure 14 are influenced by the system of Chakravarty and others (2005a) and by Faxén's static semantics for Haskell (2002). Abstract associated type synonyms require an additional judgment $\mathsf{A} \vdash^{\mathrm{o,i}} \mathsf{tdef} : \theta$ that comes in two variants: $\mathsf{A} \vdash^{\mathrm{o}} \mathsf{tdef} : \theta$ collects constraint schemes for use outside of the instance defining tdef, so abstract associated type synonyms are equated with some fresh type constructor; $\mathsf{A} \vdash^{\mathrm{i}} \mathsf{tdef} : \theta$ collects constraint schemes for use inside the instance, so the true identities of all synonyms are revealed.

One difference between Faxén's system and the rules given here is that the rule of the judgment $\vdash \mathsf{prog} : \Theta; \Gamma$ does not define Θ and Γ recursively. Instead, it first builds up Θ and Γ using the judgments $\vdash \mathsf{cls} : \Theta; \Gamma$ and $\vdash \mathsf{inst} : \Theta$, and then checks class and instance definitions using the judgments $\Theta \vdash \mathsf{cls}$ and $\Theta; \Gamma \vdash \mathsf{inst}$, respectively.

Another important difference is that the entailment check for superclasses in the

Classes $\boxed{\vdash \mathsf{cls} : \Theta; \Gamma \quad \Theta \vdash \mathsf{cls}}$

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

$$\frac{\Theta \cup \{\mathsf{C}\,\overline{\mathsf{a}}\} \vdash \tau_i \qquad (i \in [n])}{\Theta \vdash \textbf{class}\ \forall \mathsf{A}.\overline{\pi} \Rightarrow \mathsf{C}\ \overline{\mathsf{a}}\ \textbf{where}\ \overline{\mathsf{tdec}}\ \overline{\mathsf{m}_i :: \forall \mathsf{A}_i.\tau_i}^{i\in[n]}}$$

Instances $\boxed{\vdash \mathsf{inst} : \Theta \quad \Theta; \Gamma \vdash \mathsf{inst}}$

$$\frac{\theta \text{ well-formed} \qquad \mathsf{A} \vdash^{\mathrm{o}} \mathsf{tdef}_i : \theta_i \qquad (i \in [n])}{\vdash \textbf{instance}\ \theta\ \textbf{where}\ \overline{\mathsf{tdef}}^n\ \overline{\mathsf{mval}} : \{\theta, \overline{\theta}^n\}} \qquad \frac{}{\mathsf{A} \vdash^{\mathrm{o,i}} \textbf{type}\ \mathsf{S}^\kappa\,\overline{\tau}^\kappa = \tau' : \forall \mathsf{A}.\mathsf{S}^\kappa\,\overline{\tau}^\kappa = \tau'}$$

$$\frac{\mathsf{T}^\kappa \text{ fresh}}{\mathsf{A} \vdash^{\mathrm{o}} \textbf{abstype}\ \mathsf{S}^\kappa\,\overline{\tau}^\kappa = \tau' : \forall \mathsf{A}.\mathsf{S}^\kappa\,\overline{\tau}^\kappa = \mathsf{T}^\kappa\,\overline{\tau}^\kappa} \qquad \frac{}{\mathsf{A} \vdash^{\mathrm{i}} \textbf{abstype}\ \mathsf{S}^\kappa\,\overline{\tau}^\kappa = \tau' : \forall \mathsf{A}.\mathsf{S}^\kappa\,\overline{\tau}^\kappa = \tau'}$$

$$\frac{\begin{array}{c}\Theta'; \Gamma; \overline{\tau} \vdash_{method} \mathsf{m}_i = \mathsf{w}_i \qquad \Theta'; \overline{\tau} \vdash_{tdef} \mathsf{tdef}_j \qquad \mathsf{A} \vdash^{\mathrm{i}} \mathsf{tdef}_j : \theta_j \\ \Theta' \setminus \{\forall \mathsf{A}.\overline{\pi} \Rightarrow \mathsf{C}\,\overline{\tau}\} \Vdash \mathsf{C}^{\mathsf{sup}}\,\overline{\tau'} \\ (\Theta' = \Theta \cup \{\overline{\theta}^m, \overline{\pi}\}, i \in [n], j \in [m], \mathsf{C}^{\mathsf{sup}}\,\overline{\tau'} \in \textsc{Sup}(\Theta, \mathsf{C}\,\overline{\tau}))\end{array}}{\Theta; \Gamma \vdash \textbf{instance}\ \forall \mathsf{A}.\overline{\pi} \Rightarrow \mathsf{C}\,\overline{\tau}\ \textbf{where}\ \overline{\mathsf{tdef}}^m\ \overline{\mathsf{m}_i = \mathsf{w}_i}^{i\in[n]}}$$

$$\frac{\Theta \vdash \tau'}{\Theta; \overline{\tau}^\kappa \vdash_{tdef} \textbf{type}\ \mathsf{S}^\kappa\,\overline{\tau}^\kappa = \tau'} \qquad \frac{\Theta \vdash \tau'}{\Theta; \overline{\tau}^\kappa \vdash_{tdef} \textbf{abstype}\ \mathsf{S}^\kappa\,\overline{\tau}^\kappa = \tau'}$$

$$\frac{\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\,\overline{\mathsf{a}} \Rightarrow \tau' \qquad \Theta; \Gamma \vdash \mathsf{w} : [\overline{\tau/\mathsf{a}}]\forall \mathsf{A}.\tau'}{\Theta; \Gamma; \overline{\tau} \vdash_{method} \mathsf{m} = \mathsf{w}}$$

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

$$\frac{\begin{array}{c}\vdash \mathsf{cls}_i : \Theta_i; \Gamma_i \qquad \vdash \mathsf{inst}_j : \Theta'_j \\ \Theta = \bigcup_{i\in[n]} \Theta_i \cup \bigcup_{j\in[m]} \Theta'_j \qquad \Gamma = \bigcup_{i\in[n]}^{\cdot} \Gamma_i \\ \Theta \vdash \mathsf{cls}_i \qquad \Theta; \Gamma \vdash \mathsf{inst}_j \qquad (i \in [n], j \in [m])\end{array}}{\vdash \overline{\mathsf{cls}}^n\ \overline{\mathsf{inst}}^m : \Theta; \Gamma} \qquad \frac{\vdash \mathsf{pgm} : \Theta; \Gamma}{\vdash \mathsf{pgm}}$$

Figure 14. Typing judgments for Tiny-HS$^+$ classes, instances, and programs

rule defining the judgment $\Theta; \Gamma \vdash \mathsf{inst}$ is performed without the constraint scheme $\forall \mathsf{A}.\overline{\pi} \Rightarrow \mathsf{C}\ \overline{\tau}$ introduced by the very instance, whereas the two other systems use the whole constraint environment Θ. This modification is necessary because the entailment check with the whole Θ is trivial, even for programs that miss some superclass definitions.[3]

[3] The relevant rule in Faxén's system (2002, Figure 26) is not wrong but leads to diverging dictionaries. We conjecture that our modification of Faxén's rule does not reject valid programs because Tiny-HS$^+$ (and Haskell 98) ensures that constraints in the instance context contain fewer type constructors than the constraint in the instance head, so the implication removed from Θ does not fire anyway. See Sulzmann *et al.* (2005) and Wehr (2005, p. 34–36) for details.

Structures

$$\begin{array}{rcl} \mathrm{b} & ::= & \textbf{type}\ \mathrm{t} = \mathrm{u}^{\langle u \rangle}; \mathrm{b} \mid \textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} \mid \epsilon_{\mathrm{b}} \\ \mathrm{s} & ::= & \mathrm{ps} \mid \mathrm{X}^{\langle \mathcal{S} \rangle} \mid \mathrm{F}^{\langle k, \mathcal{F} \rangle}(\overline{\mathrm{X}^{\langle \mathcal{S} \rangle}}) \end{array}$$

Programs

$$\begin{array}{rcl} \mathrm{prog} & ::= & \textbf{structure}\ \mathrm{X} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle} = \mathrm{s}^{\langle \exists P.\mathcal{S} \rangle}; \mathrm{prog} \\ & \mid & \textbf{functor}\ \mathrm{F}^{\langle \mathcal{F} \rangle}(\overline{\mathrm{X} : \mathrm{S}}) :> \mathrm{S}^{\langle \Lambda P.\mathcal{S} \rangle} = \mathrm{ps}^{\langle \mathcal{S} \rangle}; \mathrm{prog} \\ & \mid & \epsilon_{\mathrm{prog}} \end{array}$$

Figure 15. Syntax of Annotated Tiny-ML (extends syntax in Figure 8)

In contrast to Chakravarty and colleagues' work (2005a), the rewrite system on associated type synonyms implied by our type system is possibly nonterminating because we assume a superset of the whole constraint environment Θ when checking the well-formedness of associated type synonym definitions (premise $\Theta'; \overline{\tau} \vdash_{tdef} \mathsf{tdef}_j$ of the rule defining the judgment $\Theta; \Gamma \vdash \mathsf{inst}$), whereas they do not assume constraint schemes resulting from instance definitions. We cannot use the their termination condition because it would rule out translations of certain Tiny-ML programs (see Section 5.1).

3.3 Formal Translation

The translation from Tiny-ML to Tiny-HS$^+$ proceeds in two steps. The first step attaches typing information to certain Tiny-ML terms (Section 3.3.1). In the second step, Tiny-ML program with type annotations are translated into Tiny-HS$^+$ programs (Sections 3.3.2 through 3.3.5).

3.3.1 Preliminaries

In this section, we define *Annotated Tiny-ML*, a variant of Tiny-ML with type annotations. Additionally, we present facilities for manipulating Tiny-ML identifiers and define the environments used in the translation.

Annotated Tiny-ML. The only difference between Annotated Tiny-ML and Tiny-ML is that certain syntactic constructs are annotated with typing information in form of semantic objects, written as superscripts enclosed in $\langle\rangle$. Hence, Figure 15 shows only those syntactic catogories of Annotated Tiny-ML that contain constructs with such annotations; the remaining syntactic categories are the same as for Tiny-ML (c.f.Figure 8).

The translation from Tiny-ML to Annotated Tiny-ML is left implicit because it is obvious how to add the annotations while constructing a typing derivation for a Tiny-ML program. In the following, we assume that all annotations result from a valid typing derivation.

Identifier manipulation. We assume the existence of several functions for translating Tiny-ML into Tiny-HS$^+$ identifiers and for generating fresh Tiny-ML identifiers.

The signatures of these functions and a short-hand notation for function application are defined as follows:

$\mathsf{StrId} \to \mathsf{TyconId}$	T^{X}	$\mathsf{FunId} \times \mathsf{TypId} \to \mathsf{ASynId}$	$\mathsf{S}^{\mathrm{F,t}}$
$\mathsf{StrId} \to \mathsf{ClassId}$	C^{X}	$\mathsf{FunId} \times \mathsf{ValId} \to \mathsf{VarId}$	$\mathsf{z}^{\mathrm{F,x}}$
$\mathsf{StrId} \times \mathsf{TypId} \to \mathsf{ASynId}$	$\mathsf{S}^{\mathrm{X.t}}$	$\mathsf{FunId} \times \mathbb{N} \times \mathsf{TypId} \to \mathsf{ASynId}$	$\mathsf{S}^{\mathrm{F},i,\mathrm{t}}$
$\mathsf{StrId} \times \mathsf{ValId} \to \mathsf{VarId}$	$\mathsf{z}^{\mathrm{X.y}}$	$\mathsf{FunId} \times \mathbb{N} \times \mathsf{ValId} \to \mathsf{VarId}$	$\mathsf{z}^{\mathrm{F},i,\mathrm{x}}$
$\mathsf{FunId} \to \mathsf{TyconId}$	T^{F}	$\mathsf{CoreId} \to \mathsf{VarId}$	z^{c}
$\mathsf{FunId} \times \mathbb{N} \to \mathsf{TyconId}$	$\mathsf{T}^{\mathrm{F},k}$	$\mathit{SimTypVar} \to \mathsf{TypVar}$	$\mathsf{a}^{`a}$
$\mathsf{FunId} \to \mathsf{ClassId}$	C^{F}	$\mathsf{StrId} \to \mathsf{StrId}$	$\mathrm{X}^{\star}$
$\mathsf{FunId} \to \mathsf{ClassId}$	$\mathsf{C}^{\mathrm{F,arg}}$	$\mathsf{FunId} \to \mathsf{FunId}$	$\mathrm{F}^{\star}$

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

Environments. An occurrence environment Φ maps value occurrences and semantic type variables to their Tiny-HS$^+$ translation. Hence, the Tiny-HS$^+$ translation of value occurrences x and $\mathrm{X.y}$ is given by $\Phi(\mathrm{x})$ and $\Phi(\mathrm{X}, \mathrm{y})$, respectively, and $\Phi(\alpha)$ denotes the translation of a semantic type variable α. Formally, Φ consists of three parts; however, we omit the sub- and superscripts used to distinguish these parts when no confusion can arise. The definition of Φ is as follows:

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

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 $\mathbf{type}\ \mathrm{t} = \mathrm{u}^{\langle u \rangle}$, $\Omega(\mathrm{t})$ is the translation of u. Similarly, $\Omega(\mathrm{x})$ is the translation of the expression e for some value definition $\mathbf{val}\ \mathrm{x} = \mathrm{e}$. We define Ω as follows (again, subscripts are often omitted):

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

3.3.2 Translation of semantic types and value expressions

Figure 16 shows the translation functions $\mathfrak{T}_u$, $\mathfrak{T}_v$, and $\mathfrak{E}$ that translate—given an occurrence environment Φ—semantic simple types, semantic value types, and Tiny-ML expressions into Tiny-HS$^+$ types, type schemes, and expressions, respectively. A semantic simple type variable $`a$ is translated into the corresponding Tiny-HS$^+$

$$\boxed{\mathfrak{T}_u[\![u]\!]\Phi = \tau \quad \mathfrak{T}_v[\![v]\!]\Phi = \sigma}$$

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

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

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

Figure 16. Translation of semantic types and value expressions

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

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

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

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

Figure 17. Translation of structure bodies and unsealed structure expressions

type variable $\mathsf{a}^{`a}$ by using the appropriate identifier manipulation function defined in the preceding section. 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 $\mathrm{X}.\mathrm{y}$.

3.3.3 Translation of structure bodies and structure expressions

The translation function $\mathfrak{S}_\mathrm{b}$, defined in Figure 17, takes an occurrence environment Φ and a structure body b and returns a code environment Ω that contains the Tiny-HS$^+$ code for the components of b. The definition of $\mathfrak{S}_\mathrm{b}$ is proper only if Φ contains all semantic type variables used in b, all value identifiers used in b, and

pairs (X, y) for all $\mathrm{X}.\mathrm{y}$ in b. 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 identifiers introduced by preceding definitions. This invariant is maintained by all translation functions that use $\mathfrak{S}_{\mathsf{b}}$.

Figure 17 also defines a function $\mathfrak{S}$ that translates a structure expression s 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 $\mathfrak{S}_{\mathsf{b}}$.

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

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 the translation to Tiny-HS$^+$, we have to make this matching explicit by creating a new instance of type class $\mathsf{C}^{\mathrm{F,arg}}$, which is the translation of the functor argument signatures. (We shall see how the type class $\mathsf{C}^{\mathrm{F,arg}}$ is defined once we discuss the translation of functor definitions in Section 3.3.5; for now, it suffices to know that $\mathsf{C}^{\mathrm{F,arg}}$ 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 $\mathcal{S}'_i$.) The data type $\mathsf{T}^{\mathrm{F},k}$ 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 that is unique among all other data types in the program.

The code environment Ω of the result triple contains the Tiny-HS$^+$ code for all components of the functor's result signature $\mathcal{S}'$. To understand the definition of Ω, you need to know that the result signature of a functor definition is translated into a two-parameter type class C^{F} that declares associated type synonyms $\mathsf{S}^{\mathrm{F,t}}$ and methods $\mathsf{z}^{\mathrm{F,x}}$ for all components of the body; furthermore, the body of a functor definition is translated into an instance $\mathsf{C}^{\mathrm{F,arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}$. Hence, $\mathsf{S}^{\mathrm{F,t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F},k}$ is the translation of a type component t, and $\mathsf{z}^{\mathrm{F,x}}\ (\bot :: \mathsf{T}^{\mathrm{F}})\ (\bot :: \mathsf{T}^{\mathrm{F},k})$ is the translation of a value component x, where t and x appear in the structure resulting from the functor application.

3.3.4 Translation of structure definitions

The function $\mathfrak{X}$ in Figure 18 translates a structure definition without a signature seal into a triple of data type, class, and instance definitions.[4] For a structure definition **structure** $\mathrm{X} = \mathrm{s}^{\langle\exists P.\mathcal{S}\rangle}$, the type class C^{X} in the result triple is the translation of $\mathcal{S}$, and the instance $\mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}$ is the translation of s.

[4] Strictly speaking, Annotated Tiny-ML does not support structure definitions without signature seals; however, we can regard a structure definition **structure** $\mathrm{X} = \mathrm{s}^{\langle\exists P.\mathcal{S}\rangle}$ as an abbreviation for **structure** $\mathrm{X} :> S^{\langle\Lambda P.\mathcal{S}\rangle} = \mathrm{s}^{\langle\exists P.\mathcal{S}\rangle}$ where S has denotation $\Lambda P.\mathcal{S}$. Moreover, $\mathfrak{X}[\![$**structure** $\mathrm{X} = \mathrm{s}^{\langle\exists P.\mathcal{S}\rangle}]\!]\Phi$ can be seen as a suggestive notation for $\mathfrak{X}[\![\mathrm{s}]\!]\mathrm{X}\langle\exists P.\mathcal{S}\rangle\Phi$.

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

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

Figure 18. Translation of structure definitions without signature seals

The definition of the occurrence environment Φ' uses the operation $\textsc{Pick}(\mathcal{S}, \alpha)$ that yields a type identifier such that $\mathcal{S}(\textsc{Pick}(\mathcal{S}, \alpha)) = \alpha$. Section 3.4.1 proves that every usage of the pick operation is well-defined. Formally, PICK is defined as follows:

Definition 3.11 (The pick operation)**.** The operation $\textsc{Pick}(\overline{\mathcal{S}}^n, \alpha) = \langle i, \mathrm{t}\rangle$ selects the lexicographically smallest $\langle i, \mathrm{t}\rangle$ with $i \in [n]$ and $\mathrm{t} \in \textsc{Dom}(\mathcal{S}_i)$ such that $\mathcal{S}_i(\mathrm{t}) = \alpha$. We write $\textsc{Pick}(\mathcal{S}, \alpha) = \mathrm{t}$ if $n = 1$.

Figure 19 shows how regular structure definitions (with signature seals) are translated. The function $\mathfrak{P}$ translates an Annotated Tiny-ML program into a triple of Tiny-HS$^+$ data type, class, and instance definitions. To obtain the final Tiny-HS$^+$ program, we simply concatenate the definitions of the triple. (The definition of $\mathfrak{P}$ for functor definitions is given in the next section.)

To translate a structure definition, we split, at least conceptually, a structure definition of the form **structure** $\mathrm{X} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S}\rangle} = \mathrm{s}^{\langle \exists P'.\mathcal{S}'\rangle}$ into two structure definitions **structure** $\mathrm{X}^\star = \mathrm{s}^{\langle \exists P'.\mathcal{S}'\rangle}$ and **structure** $\mathrm{X} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S}\rangle} = \mathrm{X}^{\star\langle \mathcal{S}'\rangle}$. The definition of $\mathrm{X}^\star$ is then translated using the function $\mathfrak{X}$ just defined. For the translation of X, we define a type class C^{X} as the translation of the semantic structure $\mathcal{S}$ and create an instance $\mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}$. 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. If t is selected by the PICK operation then it is translated into an abstract associated type synonym; if t is not selected by PICK it is translated into a regular associated type synonym that propagates the type equality with the corresponding abstract synonym. (Strictly speaking, the propagation of type equalities is not needed because a newly introduced semantic type variables is always represented by the same asso-

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

$$
\begin{array}{l}
\mathfrak{P}[\![\textbf{structure}\ \mathrm{X} :> \mathrm{S}^{\langle \Lambda P.\mathcal{S}\rangle} = \mathrm{s}^{\langle \exists P'.\mathcal{S}'\rangle}; \mathrm{prog}]\!]\Phi = \\
\quad \langle\ \textbf{data}\ \mathsf{T}^{\mathrm{X}}\ \overline{\mathsf{ddec}}\ \overline{\mathsf{ddec}'} \\
\quad ,\ \overline{\mathsf{cls}} \\
\quad\ \ \textbf{class}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{a}\ \textbf{where} \\
\quad\quad \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{a} \qquad\qquad\qquad\qquad\qquad\qquad \mathrm{t} \in \textsc{Dom}(\mathcal{S}) \\
\quad\quad \mathsf{z}^{\mathrm{X.y}} :: \forall \mathsf{B}_{\mathrm{y}}.\mathsf{a} \rightarrow \tau_{\mathrm{y}} \qquad\qquad\qquad\qquad \mathrm{y} \in \textsc{Dom}(\mathcal{S}) \\
\quad\ \ \overline{\mathsf{cls}'} \\
\quad ,\ \textbf{instance}\ \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}\ \textbf{where} \\
\quad\quad \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} \quad\ \ = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} \qquad \mathrm{t} \in \textsc{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \notin P \\
\quad\quad \textbf{abstype}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathsf{S}^{\mathrm{X}^\star.\mathrm{t}}\ \mathsf{T}^{\mathrm{X}^\star} \qquad \mathrm{t} \in \textsc{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t} = \textsc{Pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})) \\
\quad\quad \textbf{type}\ \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} \quad\ \ = \mathsf{S}^{\mathrm{X.t}'}\ \mathsf{T}^{\mathrm{X}} \qquad\quad \mathrm{t} \in \textsc{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t} \neq \mathrm{t}' = \textsc{Pick}(\mathcal{S}, \mathcal{S}(\mathrm{t})) \\
\quad\quad \mathsf{z}^{\mathrm{X.y}} \qquad\qquad\quad\ = \lambda\_.\mathsf{z}^{\mathrm{X}^\star.\mathrm{y}}\ (\bot :: \mathsf{T}^{\mathrm{X}^\star}) \quad \mathrm{y} \in \textsc{Dom}(\mathcal{S}) \\
\quad\ \ \overline{\mathsf{inst}}\ \overline{\mathsf{inst}'}\rangle \\
\quad \textit{where} \\
\quad\quad \langle\overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}}\rangle \qquad = \mathfrak{X}[\![\textbf{structure}\ \mathrm{X}^\star = \mathrm{s}^{\langle \exists P'.\mathcal{S}'\rangle}]\!]\Phi \\
\quad\quad \langle\overline{\mathsf{ddec}'}, \overline{\mathsf{cls}'}, \overline{\mathsf{inst}'}\rangle \quad = \mathfrak{P}[\![\mathrm{prog}]\!]\Phi' \\
\quad\quad \Phi' \qquad\qquad\qquad\quad = \Phi\ \dot{\cup}\ \{\alpha \mapsto \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} \mid \alpha \in P, \mathrm{t} = \textsc{Pick}(\mathcal{S}, \alpha)\} \\
\quad\quad \qquad\qquad\qquad\qquad\quad\ \ \dot{\cup}\ \{(\mathrm{X}, \mathrm{y}) \mapsto \mathsf{z}^{\mathrm{X.y}}\ (\bot :: \mathsf{T}^{\mathrm{X}}) \mid \mathrm{y} \in \textsc{Dom}(\mathcal{S})\} \\
\quad\quad \forall \mathsf{B}_{\mathrm{y}}.\tau_{\mathrm{y}} \qquad\qquad\quad = \mathfrak{T}_v[\![\mathcal{S}(\mathrm{y})]\!]\Phi'' \\
\quad\quad \Phi'' \qquad\qquad\qquad\quad = \Phi\ \dot{\cup}\ \{\alpha \mapsto \mathsf{S}^{\mathrm{X.t}}\ \mathsf{a} \mid \alpha \in P, \mathrm{t} = \textsc{Pick}(\mathcal{S}, \alpha)\} \\
\quad\quad \mathsf{a} \in \mathsf{FreshTypVars}
\end{array}
$$

Figure 19. Translation of structure definitions (see Figure 21 for the other case of $\mathfrak{P}$)

ciated type synonym, namely the one selected by Pick. However, the propagation is mandatory for functor definitions so we consistently perform it also for structure definitions.)

3.3.5 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 structure definitions, we first define a function $\mathfrak{F}$ that translates functor definitions without signature seals[5] (Figure 20); a functor definition with signature seal is then split into two functor definitions, one without and one with signature seal. The extra parameter F' to $\mathfrak{F}$ represents the functor identifier of the original definition. Notice that $\mathfrak{F}$ requires that the semantic structure $\mathcal{S}$ on the left-hand and right-hand side of the functor definition must match; all well-typed programs satisfy this criteria.

The triple of data type, class, and instance definitions returned by $\mathfrak{F}$ contains a new class $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$, which is the translation of all argument signatures $\overline{\mathcal{S}}^n$. We already discussed on page 24 that instances of the class $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$ are used to translate functor applications. The triple also defines another type class C^{F} as the translation of the

[5] Functor definitions without signature seals can be modeled the same way as structure definitions without signature seals.

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

$$\mathfrak{F}[\![\textbf{functor } \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}\rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) = \textbf{struct b end}^{\langle \mathcal{S}\rangle}]\!]\mathrm{F}'\Phi =$$

$\langle$ **data** T^{F}

, **class** $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$ a **where**

type $\mathsf{S}^{\mathrm{F}',i,\mathrm{t}}$ a	$i \in [n], \mathrm{t} \in \textsc{Dom}(\mathcal{S}_i)$
$\mathsf{z}^{\mathrm{F}',i,\mathrm{x}} :: \forall \mathsf{B}_{\mathrm{x},i}.\mathsf{a} \to \tau_{\mathrm{x},i}$	$i \in [n], \mathrm{x} \in \textsc{Dom}(\mathcal{S}_i)$

class $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$ a $\Rightarrow$ C^{F} b a **where**

type $\mathsf{S}^{\mathrm{F},\mathrm{t}}$ b a	$\mathrm{t} \in \textsc{Dom}(\mathcal{S})$
$\mathsf{z}^{\mathrm{F},\mathrm{x}} :: \forall \mathsf{B}_{\mathrm{x}}.\mathsf{b} \to \mathsf{a} \to \tau_{\mathrm{x}}$	$\mathrm{x} \in \textsc{Dom}(\mathcal{S})$

, **instance** $\mathsf{C}^{\mathrm{F}',\mathrm{arg}}$ a $\Rightarrow$ C^{F} T^{F} a **where**

type $\mathsf{S}^{\mathrm{F},\mathrm{t}}$ T^{F} a	$= \Omega(\mathrm{t})$	$\mathrm{t} \in \textsc{Dom}(\mathcal{S})$
$\mathsf{z}^{\mathrm{F},\mathrm{x}}$	$= \lambda\_.\lambda\mathsf{z}.\Omega(\mathrm{x})$	$\mathrm{x} \in \textsc{Dom}(\mathcal{S})$

$\rangle$

where

$$\begin{array}{lcl}
\forall \mathsf{B}_{\mathrm{x},i}.\tau_{\mathrm{x},i} & = & \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi' \\
\forall \mathsf{B}_{\mathrm{x}}.\tau_{\mathrm{x}} & = & \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi' \\
\Phi' & = & \Phi \,\dot\cup\, \{\alpha \mapsto \mathsf{S}^{\mathrm{F}',i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i,\mathrm{t}\rangle = \textsc{Pick}(\overline{\mathcal{S}}, \alpha)\} \\
\Omega & = & \mathfrak{S}_{\mathrm{b}}[\![\mathrm{b}]\!]\Phi'' \\
\Phi'' & = & \Phi' \,\dot\cup\, \{(\mathrm{X}_i, \mathrm{y}) \mapsto \mathsf{z}^{\mathrm{F}',i,\mathrm{y}}\ \mathsf{z} \mid i \in [n], \mathrm{y} \in \textsc{Dom}(\mathcal{S}_i)\} \\
 & & \quad \dot\cup\, \{\mathrm{x} \mapsto \mathsf{z}^{\mathrm{F},\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}})\ \mathsf{z} \mid \mathrm{x} \in \textsc{Dom}(\mathcal{S})\}
\end{array}$$

$\mathsf{z} \in \mathsf{FreshVarIds}, \mathsf{a} \neq \mathsf{b} \in \mathsf{FreshTypVars}$

Figure 20. Translation of functor definitions without signature seals

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

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

$\langle$ **data** T^{F} $\overline{\mathsf{ddec}}$ $\overline{\mathsf{ddec}'}$

, $\overline{\mathsf{cls}}$

class $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$ a $\Rightarrow$ C^{F} b a **where**

type $\mathsf{S}^{\mathrm{F},\mathrm{t}}$ b a	$\mathrm{t} \in \textsc{Dom}(\mathcal{S})$
$\mathsf{z}^{\mathrm{F},\mathrm{x}} :: \forall \mathsf{B}_{\mathrm{x}}.\mathsf{b} \to \mathsf{a} \to \tau_{\mathrm{x}}$	$\mathrm{x} \in \textsc{Dom}(\mathcal{S})$

$\overline{\mathsf{cls}'}$

, **instance** $\mathsf{C}^{\mathrm{F},\mathrm{arg}}$ a $\Rightarrow$ C^{F} T^{F} a **where**

type $\mathsf{S}^{\mathrm{F},\mathrm{t}}$ T^{F} a	$= \mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}$ $\mathsf{T}^{\mathrm{F}^\star}$ a	$\mathrm{t} \in \textsc{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \notin P$
abstype $\mathsf{S}^{\mathrm{F},\mathrm{t}}$ T^{F} a	$= \mathsf{S}^{\mathrm{F}^\star,\mathrm{t}}$ $\mathsf{T}^{\mathrm{F}^\star}$ a	$\mathrm{t} \in \textsc{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t} = \textsc{Pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))$
type $\mathsf{S}^{\mathrm{F},\mathrm{t}}$ T^{F} a	$= \mathsf{S}^{\mathrm{F},\mathrm{t}'}$ T^{F} a	$\mathrm{t} \in \textsc{Dom}(\mathcal{S}), \mathcal{S}(\mathrm{t}) \in P, \mathrm{t} \neq \mathrm{t}' = \textsc{Pick}(\mathcal{S}, \mathcal{S}(\mathrm{t}))$
$\mathsf{z}^{\mathrm{F},\mathrm{x}}$	$= \lambda\_.\mathsf{z}^{\mathrm{F}^\star,\mathrm{x}}\ (\bot :: \mathsf{T}^{\mathrm{F}^\star})$	$\mathrm{x} \in \textsc{Dom}(\mathcal{S})$

$\overline{\mathsf{inst}}$ $\overline{\mathsf{inst}'}\rangle$

where

$$\begin{array}{lcl}
\langle \overline{\mathsf{ddec}}, \overline{\mathsf{cls}}, \overline{\mathsf{inst}}\rangle & = & \mathfrak{F}[\![\textbf{functor } \mathrm{F}^{\star\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S}'\rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i\in[n]}) = \mathrm{ps}^{\langle \mathcal{S}'\rangle}]\!]\mathrm{F}\Phi \\
\langle \overline{\mathsf{ddec}'}, \overline{\mathsf{cls}'}, \overline{\mathsf{inst}'}\rangle & = & \mathfrak{P}[\![\mathrm{prog}]\!]\Phi \\
\forall \mathsf{B}_{\mathrm{x}}.\tau_{\mathrm{x}} & = & \mathfrak{T}_v[\![\mathcal{S}(\mathrm{x})]\!]\Phi' \\
\Phi' & = & \Phi \,\dot\cup\, \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i,\mathrm{t}\rangle = \textsc{Pick}(\overline{\mathcal{S}}, \alpha)\} \\
 & & \quad \dot\cup\, \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{b}\ \mathsf{a} \mid \alpha \in P, \mathrm{t} = \textsc{Pick}(\mathcal{S}, \alpha)\}
\end{array}$$

$\mathsf{a} \neq \mathsf{b} \in \mathsf{FreshTypVars}$

Figure 21. Translation of functor definitions (see Figure 19 for the other case of $\mathfrak{P}$)

functor result signature $\mathcal{S}$, and an instance $\mathsf{C}^{\mathsf{F}',\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathsf{F}}\ \mathsf{T}^{\mathsf{F}}\ \mathsf{a}$ as the translation of the functor body ps. The class $\mathsf{C}^{\mathsf{F}',\mathrm{arg}}$ has to be a superclass of C^{F} because type variables from Q, represented as associated type synonyms of $\mathsf{C}^{\mathsf{F}',\mathrm{arg}}$, might appear free in $\mathcal{S}$.

Figure 21 shows the missing case of the program translation function $\mathfrak{P}$. Notice that all type correct programs fulfill the restriction imposed on the annotations of the functor definitions accepted by $\mathfrak{P}$. Along the lines of the case for structure definitions, $\mathfrak{P}$ simulates splitting a single functor definition into two functor definitions, one without and one with signature seal.

The functor definition without signature seal is translated by the function $\mathfrak{F}$. This results in two type classes $\mathsf{C}^{\mathsf{F},\mathrm{arg}}$ and $\mathsf{C}^{\mathsf{F}^\star}$, where the later type class represents the unsealed functor result signature. The function $\mathfrak{P}$ itself translates the definition with signature seal into a type class C^{F} and an instance $\mathsf{C}^{\mathsf{F},\mathrm{arg}}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathsf{F}}\ \mathsf{T}^{\mathsf{F}}\ \mathsf{a}$, in a way similar to the case for structure definitions. This time, however, the propagation of type equalities between abstract associated type synonyms representing type components selected by Pick and associated type synonyms representing components not selected by Pick is essential to obtain a type correct Tiny-HS$^+$ program. The reason for this is that the translation of the rest program may use both sorts of type synonyms.

3.4 Formal Properties

This section establishes two important properties of the translation from Tiny-ML to Tiny-HS$^+$: it is well-defined (Section 3.4.1) and preserves type correctness (Section 3.4.2).

3.4.1 Well-definedness

It is not difficult to prove that the translation is well-defined for every type correct Tiny-ML program. Hence, we only show the main definitions and propositions. All auxiliary lemmata and all proofs can be found in the first author's diploma thesis (Wehr, 2005).

The definitions of solvability and groundness, given next, formalize the intuition that fresh semantic type variables are introduced only by opaque type components. Both definitions are taken from Russo's thesis (Russo, 1998).

Definition 3.12 (Solvability)**.** A semantic structure $\mathcal{S}$ is said to be solvable with respect to $P \subseteq \mathit{TypVar}$, written $\textsc{Solv}(\mathcal{S}, P)$, iff for all $\alpha \in P$ there exists some $\mathrm{t} \in \textsc{Dom}(\mathcal{S})$ such that $\mathcal{S}(\mathrm{t}) = \alpha$.

Definition 3.13 (Groundness)**.** A semantic functor $\mathcal{F} = \forall Q.\overline{\mathcal{S}}^n \to \exists P.\mathcal{S}$ is ground iff $\textsc{Solv}(\mathcal{S}, P)$, and there exist sets Q_i with $Q = \cup_{i\in[n]} Q_i$ such that $\textsc{Solv}(\mathcal{S}_i, Q_i)$ for all $i \in [n]$. A context $\mathcal{C}$ is ground if $\mathcal{C}(\mathrm{F})$ is ground for all $\mathrm{F} \in \textsc{Dom}(\mathcal{C})$.

We now prove that well-typed signature and structure expressions are solvable. This property is essential for proving the well-definedness of uses of the Pick operation.

Lemma 3.14 (Solvability of signature and structure expressions)**.** *If* $\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S}$ *then* $\textsc{Solv}(\mathcal{S}, P)$*. If* $\mathcal{C}$ *ground and* $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$ *then* $\textsc{Solv}(\mathcal{S}, P)$*.*

Proof. By rule induction.

The next definition introduces a predicate $\textsc{Valid}(\mathcal{C}, \Phi)$ that holds whenever the occurrence environment Φ provides translations of all type variables, unqualified value identifiers, and qualified value identifiers bound in the context $\mathcal{C}$.

Definition 3.15 (Validity of occurrence environments)**.** An occurrence environment Φ is said to be valid with respect to a context $\mathcal{C}$, written $\textsc{Valid}(\mathcal{C}, \Phi)$, iff $\mathrm{FV}^{\alpha}(\mathcal{C}) \cup \{\mathrm{x} \mid \mathrm{x} \in \textsc{Dom}(\mathcal{C})\} \cup \{(\mathrm{X}, \mathrm{y}) \mid \mathrm{X} \in \textsc{Dom}(\mathcal{C}), \mathrm{y} \in \textsc{Dom}(\mathcal{C}(\mathrm{X}))\} \subseteq \textsc{Dom}(\Phi)$.

Finally, we show that the translation from Tiny-ML to Tiny-HS$^+$ is well-defined for type correct programs.

Theorem 3.16 (Well-definedness of program translation)**.** *The translation* $\mathfrak{P}[\![\mathsf{prog}]\!]\Phi$ *is well-defined if* $\mathcal{C} \vdash \mathsf{prog}$*,* $\mathcal{C}$ *ground, and* $\textsc{Valid}(\mathcal{C}, \Phi)$*.*

Proof. By structural induction on prog.

An immediate corollary of the theorem is that $\mathfrak{P}[\![\mathsf{prog}]\!]\emptyset$ is well-defined if $\vdash \mathsf{prog}$.

3.4.2 Type correctness

This section presents the key arguments required to show that the translation preserves type correctness. Technical lemmata and proof details have been omitted; they can be found elsewhere (Wehr, 2005).

The first lemma shows that translated types are well-formed.

Lemma 3.17 (Well-formedness of translated types)**.** *If* $\mathfrak{T}_u[\![u]\!]\Phi = \tau$ *and* $\Theta \vdash \Phi(\alpha)$ *for all* $\alpha \in \textsc{Dom}(\Phi)$ *then* $\Theta \vdash \tau$*.*

Proof. Induction on the structure of u.

To establish similar results for other translation functions, we define a matching relation $\approx$ between Tiny-HS$^+$ environments Θ and Γ and Tiny-ML contexts $\mathcal{C}$. Matching is mediated by an occurrence environment Φ.

Definition 3.18 (Matching)**.** Tiny-HS$^+$ environments Θ and Γ match a Tiny-ML context $\mathcal{C}$ with respect to an occurrence environment Φ, written $\Theta; \Gamma \approx^{\Phi} \mathcal{C}$, iff $\mathcal{C}$ ground and $\textsc{Valid}(\mathcal{C}, \Phi)$ and $\Theta \vdash \Phi(\alpha)$ and $\Theta; \Gamma \vdash \Phi(\mathrm{x}) : \mathfrak{T}_v[\![\mathcal{C}(\mathrm{x})]\!]\Phi$ and $\Theta; \Gamma \vdash \Phi(\mathrm{X}, \mathrm{y}) : \mathfrak{T}_v[\![\mathcal{C}(\mathrm{X})(\mathrm{y})]\!]\Phi$ and $\Gamma(\mathsf{z}^{\mathrm{c}}) = \mathfrak{T}_v[\![\mathcal{C}(\mathrm{c})]\!]\Phi$ for all $\alpha \in \textsc{Dom}(\Phi)$ and $\mathrm{x}, \mathrm{X}, \mathrm{c} \in \textsc{Dom}(\mathcal{C})$ and $\mathrm{y} \in \textsc{Dom}(\mathcal{C}(\mathrm{X}))$. Furthermore, the following must hold for all $\mathrm{F} \in \textsc{Dom}(\mathcal{S})$ with $\mathcal{C}(\mathrm{F}) = \forall Q.\overline{\mathcal{S}}^{n} \to \exists P.\mathcal{S}$:

- for all $i \in [n], \mathrm{t} \in \textsc{Dom}(\mathcal{S}_i)$: $\mathsf{S}^{\mathrm{F},i,\mathrm{t}}$ is an associated type synonym of class $\mathsf{C}^{\mathrm{F},arg}$
- for all $i \in [n]$, $\mathrm{x} \in \textsc{Dom}(\mathcal{S}_i)$: $\Gamma(\mathsf{z}^{\mathrm{F},i,\mathrm{x}}) = \forall \mathsf{A} \,\dot\cup\, \{\mathsf{a}\}.\mathsf{C}^{\mathrm{F},arg}\ \mathsf{a} \Rightarrow \mathsf{a} \to \tau$ where $\forall \mathsf{A}.\tau = \mathfrak{T}_v[\![\mathcal{S}_i(\mathrm{x})]\!]\Phi'$ and $\Phi' = \Phi \,\dot\cup\, \{\alpha \mapsto \mathsf{S}^{\mathrm{F},i,\mathrm{t}}\ \mathsf{a} \mid \alpha \in Q, \langle i, \mathrm{t}\rangle = \textsc{Pick}(\overline{\mathcal{S}}, \alpha)\}$
- $(\forall\{a\}.\mathsf{C}^{\mathrm{F},arg}\ \mathsf{a} \Rightarrow \mathsf{C}^{\mathrm{F}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a}) \in \Theta$ where T^{F} is a user-defined data constructor of kind 0

- for all $\mathsf{t} \in \textsc{Dom}(\mathcal{S})$: $\mathsf{S}^{\mathrm{F},t}$ is an associated type synonym of type class C^{F} and $\Theta \Vdash [\tau/\mathsf{a}](\mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} = \mathfrak{T}_u[\![\mathcal{S}(\mathsf{t})]\!]\Phi'')$ for all τ where $\Phi'' = \Phi' \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{T}^{\mathrm{F}}\ \mathsf{a} \mid \alpha \in P, \mathsf{t} = \textsc{Pick}(\mathcal{S}, \alpha)\}$
- for all $\mathsf{x} \in \textsc{Dom}(\mathcal{S})$: $\Gamma(\mathsf{z}^{\mathrm{F},\mathrm{x}}) = \forall \mathsf{A} \dot\cup \{\mathsf{a}, \mathsf{b}\}.\mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{b} \to \mathsf{a} \to \tau$ where $\forall \mathsf{A}.\tau = \mathfrak{T}_v[\![\mathcal{S}(\mathsf{x})]\!]\Phi'''$ and $\Phi''' = \Phi' \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{F},\mathrm{t}}\ \mathsf{b}\ \mathsf{a} \mid \alpha \in P, \mathsf{t} = \textsc{Pick}(\mathcal{S}, \alpha)\}$
- $\textsc{Sup}(\Theta, \mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}) = \emptyset$, $\textsc{Sup}(\Theta, \mathsf{C}^{\mathrm{F}}\ \mathsf{b}\ \mathsf{a}) = \{\mathsf{C}^{\mathrm{F},\mathrm{arg}}\ \mathsf{a}\}$

For technical reasons, we also require $\mathrm{FV}^{\mathsf{z}}(\Phi) \cap \{\mathsf{z}^{\mathsf{c}} \mid \mathsf{c} \in \mathsf{CoreId}\} = \emptyset$.

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

Lemma 3.19 (Type correctness of translated expressions)**.** *Suppose* $\Theta; \Gamma \approx^{\Phi} \mathcal{C}$. *If* $\mathcal{C} \vdash \mathsf{e} : u$ *and* $\mathrm{FV}^{\alpha}(u) \subseteq \textsc{Dom}(\Phi)$ *then* $\Theta; \Gamma \vdash \mathfrak{E}[\![\mathsf{e}]\!]\Phi : \mathfrak{T}_u[\![\mathsf{u}]\!]\Phi$. *Moreover, if* $\mathcal{C} \vdash \mathsf{e} : v$ *then* $\Theta; \Gamma \vdash \mathfrak{E}[\![\mathsf{e}]\!]\Phi : \mathfrak{T}_v[\![\mathsf{v}]\!]\Phi$.

Proof. By induction on the structure of e.

The next lemma shows that the translation of a structure body b yields a code environment Ω that correctly reflects the semantic structure $\mathcal{S}$ of b.

Lemma 3.20 (Type correctness of translated structure bodies)**.** *Suppose* $\mathcal{C} \vdash \mathsf{b} : \mathcal{S}$ *and* $\mathfrak{S}_{\mathsf{b}}[\![\mathsf{b}]\!]\Phi = \Omega$ *and* $\Theta; \Gamma \approx^{\Phi} \mathcal{C}$ *and* $\mathsf{x} \in \textsc{Dom}(\Phi)$ *with* $\Theta; \Gamma \vdash \Phi(\mathsf{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathsf{x})]\!]\Phi$ *for all* $\mathsf{x} \in \textsc{Dom}(\mathcal{S})$. *Then* $\Theta \vdash \Omega(\mathsf{t})$ *and* $\Omega(\mathsf{t}) = \mathfrak{T}_u[\![\mathcal{S}(\mathsf{t})]\!]\Phi$ *and* $\Theta; \Gamma \vdash \Omega(\mathsf{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathsf{x})]\!]\Phi$ *for all* $\mathsf{t}, \mathsf{x} \in \textsc{Dom}(\Omega)$.

Proof. By structural induction on b.

We now show that the instance definitions returned by the translation function for structure expressions are type correct and that the resulting code environment correctly reflects the components of the structure expression.

Lemma 3.21 (Type correctness of translated structure expressions)**.** *Suppose* $\mathcal{C} \vdash \mathsf{s} : \exists P.\mathcal{S}$ *and* $\mathfrak{S}[\![\mathsf{s}]\!](\Phi \dot\cup \Phi') = \langle \Omega, \overline{\mathsf{ddec}}, \overline{\mathsf{inst}}^m \rangle$ *with* $\textsc{Dom}(\mathcal{S}_{\mathsf{x}}) = \textsc{Dom}(\Phi')$ *and* $\mathrm{FV}^{`a}(\mathcal{C}) \cup \mathrm{FV}^{\mathsf{a}}(\Phi) = \emptyset$. *Moreover, suppose* $\Theta; \Gamma \approx^{\Phi \dot\cup \Phi'} \mathcal{C}$ *and* $\Theta; \Gamma \vdash \Phi'(\mathsf{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathsf{x})]\!](\Phi \dot\cup \Phi'')$ *for all* $\mathsf{x} \in \textsc{Dom}(\mathcal{S})$ *and some* $\Phi'' = \{\alpha \mapsto \tau_\alpha \mid \alpha \in P, \mathsf{t} = \textsc{Pick}(\mathcal{S}, \alpha), \Theta \Vdash \tau_\alpha = \Omega(\mathsf{t}), \mathrm{FV}^{\mathsf{a}}(\tau_\alpha) = \emptyset\}$.

Then $\vdash \mathsf{inst}_i : \Theta_i$ *for all* $i \in [m]$. *Furthermore,* $\Theta'; \Gamma \vdash \mathsf{inst}_i$ *and* $\Theta' \vdash \Omega(\mathsf{t})$ *and* $\Theta' \Vdash \Omega(\mathsf{t}) = \mathfrak{T}_u[\![\mathcal{S}(\mathsf{t})]\!](\Phi \dot\cup \Phi'')$ *and* $\Theta'; \Gamma \vdash \Omega(\mathsf{x}) : \mathfrak{T}_v[\![\mathcal{S}(\mathsf{x})]\!](\Phi \dot\cup \Phi'')$ *for* $\Theta' = \Theta \cup \bigcup_{i \in [m]} \Theta_i$ *and* $i \in [m]$ *and* $\mathsf{t}, \mathsf{x} \in \textsc{Dom}(\mathcal{S})$.

Proof. By structural induction on s.

To lighten the notation, we let the symbol pv range over Tiny-HS$^+$ *program vectors* $\langle \overline{\mathsf{ddec}}, \overline{\mathsf{inst}}, \overline{\mathsf{cls}} \rangle$. The binary operation $\oplus$ is the element-wise concatenation of program vectors and $\overrightarrow{\mathsf{pv}}$ denotes the Tiny-HS program consisting of pv's elements.

To connect a Tiny-ML context $\mathcal{C}$ with a Tiny-HS$^+$ program vector pv, we say that pv provides $\mathcal{C}$ through Φ at Θ and Γ iff $\vdash \overrightarrow{\mathsf{pv}} : \Theta; \Gamma$ and $\Theta; \Gamma \approx^{\Phi} \mathcal{C}$. When the specific Θ and Γ do not matter we simply say that pv provides $\mathcal{C}$ through Φ. Intuitively, pv provides $\mathcal{C}$ through Φ if pv is the translation of a program whose type information is contained in $\mathcal{C}$ and Φ is a mediating occurrence environment.

If a program vector pv provides $\mathcal{C}$ through Φ and if a structure expression s is classified as $\exists P.\mathcal{S}$ under $\mathcal{C}$, then the program vector pv', obtained by translating the structure definition **structure** $\mathrm{X} = \mathrm{s}$, combined with pv should provide the extended context $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$ through an extension of Φ. The following lemma formalizes this intuition.

Lemma 3.22 (Type correctness of translated structure definitions without signature seals)**.** *Suppose* $\mathfrak{X}[\![\textbf{structure } \mathrm{X} = \mathrm{s}^{\langle \exists P.\mathcal{S} \rangle}]\!]\Phi = \mathsf{pv}'$ *and* $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$ *with* $\mathrm{FV}^{`a}(\mathcal{C}) \cup \mathrm{FV}^{\mathsf{a}}(\Phi) = \{(\mathrm{X}, \mathrm{y}) \mid \mathrm{y} \in \mathrm{ValId}\} \cap \Phi = \emptyset$*. If* pv *provides* $\mathcal{C}$ *through* Φ *then* $\mathsf{pv} \oplus \mathsf{pv}'$ *provides* $\mathcal{C}, \mathrm{X} \mapsto \mathcal{S}$ *through* $\Phi' = \Phi \dot\cup \{\alpha \mapsto \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} \mid \alpha \in P, \mathrm{t} = \textsc{Pick}(\mathcal{S}, \alpha)\} \dot\cup \{(\mathrm{X}, \mathrm{y}) \mapsto \mathsf{z}^{\mathrm{X.y}}\ (\bot :: \mathsf{T}^{\mathrm{X}}) \mid \mathrm{y} \in \textsc{Dom}(\mathcal{S})\}$ *at* Θ *and* Γ *such that* $\Theta \Vdash \mathsf{C}^{\mathrm{X}}\ \mathsf{T}^{\mathrm{X}}$ *and* $\mathsf{S}^{\mathrm{X.t}}$ *is an associated type synonym of class* C^{X} *with* $\Theta \Vdash \mathsf{S}^{\mathrm{X.t}}\ \mathsf{T}^{\mathrm{X}} = \mathfrak{T}_u[\![\mathcal{S}(\mathrm{t})]\!]\Phi'$ *for all* $\mathrm{t} \in \textsc{Dom}(\mathcal{S})$*.*

Proof. Follows from Lemma 3.21 and some lemmata proved elsewhere (Wehr, 2005).

The next lemma does the same job for functor definitions.

Lemma 3.23 (Type correctness of translated functor definitions without signature seals)**.** *Suppose that* $\mathfrak{F}[\![\textbf{functor } \mathrm{F}^{\langle \forall Q.\overline{\mathcal{S}}^n \to \mathcal{S} \rangle}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i \in [n]})]\!] = \mathrm{ps}^{\langle \mathcal{S} \rangle}]\!]\mathrm{F}'\Phi = \mathsf{pv}'$ *and that* $\mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i \in [n]} \vdash \mathrm{ps} : \mathcal{S}$ *with* $\mathrm{FV}^{`a}(\mathcal{C}) \cup \bigcup_{i \in [n]} \mathrm{FV}^{`a}(\mathcal{S}_i) \cup \mathrm{FV}^{\mathsf{a}}(\Phi) = \emptyset$*. If* pv *provides* $\mathcal{C}$ *through* Φ *then the following holds:*

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

Proof. Similar to the proof of Lemma 3.22.

We are now in the position to prove the main result of this section:

Theorem 3.24 (Type correctness of translated programs)**.** *Suppose* $\mathfrak{P}[\![\mathrm{prog}]\!]\Phi = \mathsf{pv}'$ *and* pv *provides* $\mathcal{C}$ *through* Φ *such that* $\mathrm{FV}^{`a}(\mathcal{C}) \cup \mathrm{FV}^{\mathsf{a}}(\Phi) = \textsc{Dom}(\Phi) \cap \{(\mathrm{X}, \mathrm{y}) \mid \mathrm{X} \in \mathrm{StrId} \setminus \textsc{Dom}(\mathcal{C}), \mathrm{y} \in \mathrm{ValId}\} = \emptyset$*. If* $\mathcal{C} \vdash \mathrm{prog}$ *then* $\vdash \overrightarrow{\mathsf{pv} \oplus \mathsf{pv}'}$*.*

Proof. By structural induction on prog.

A direct corollary of the theorem is that $\vdash \overrightarrow{\mathsf{pv}}$ whenever $\mathfrak{P}[\![\mathrm{prog}]\!] = \mathsf{pv}$ and $\emptyset \vdash \mathrm{prog}$.

3.5 Lifting Restrictions

Tiny-ML does not support all features of Standard ML's module system and none of its extensions. The following paragraphs discuss if and how some of the restrictions could be lifted.

Nested structures. Structures in Tiny-ML must not contain other structures because such nested structures would correspond to nested classes and nested instances, which are not supported by Haskell 98 or any extension. One possibility to translate nested structures is to lift them to the top level. However, this requires a nontrivial transformation because nested structures may refer to components of their containing structure.

Arbitrary structure expressions as functor bodies. Tiny-ML supports only functor bodies of the form **struct** ... **end**. If arbitrary structure expressions were allowed as functor bodies, the translation would have to deal with definitions such as **functor** F(X) :> S = G(X). The problem with this example 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 available only *inside* the class and instance definitions representing F; hence, it cannot be used in the instance for G(X) because nested classes and instances are not supported by Haskell.

Higher-order functors. Higher-order functors (MacQueen & Tofte, 1994) are not supported in Tiny-ML because they pose a problem to the translation. Consider a higher-order functor H, which takes a functor F as argument. Let C^{H} and C^{F} be the type classes representing the result signatures of H and F, respectively, and let $\mathsf{C}^{\mathsf{F,arg}}$ be the translation of F's argument signature. The classes C^{H} and C^{F} both have two parameters, where the second parameter represents the argument of the corresponding functor. Hence, the definition of C^{H} would start with **class** $\forall\{\mathsf{a},\mathsf{b}\}.(\forall\mathsf{c}.\ \mathsf{C}^{\mathsf{F,arg}}\ \mathsf{c} \Rightarrow \mathsf{C}^{\mathsf{F}}\ \mathsf{a}\ \mathsf{c}) \Rightarrow \mathsf{C}^{\mathsf{H}}\ \mathsf{b}\ \mathsf{a}$. However, constraint schemes in class contexts are not supported by Haskell 98 or any extension. They are briefly mentioned by Peyton Jones and colleagues (1997, Section 5.2) but rejected because they would lead to a "substantial complication" of the type system. However, Rossberg and Sulzmann (2002, Section 4.3) show that such universal quantifications can be encoded in the Haskell-like language Chameleon (Sulzmann & Wazny, 2005).

Features omitted for simplicity. Some features were omitted from Tiny-ML to keep the translation manageable; it is straightforward to encode them with Haskell type classes:

- *Parameterizable type components:* It is straightforward to translate parameterizable type components into associated type synonyms with more parameters than the class that declares them, a feature supported by the system of Chakravarty *et al.* (2005a).
- *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.

- *Algebraic data types.* In Standard ML, signatures and structures can also contain algebraic data type definitions. Such definitions correspond—to a certain extent—to associated data types, a Haskell extension suggested by Chakravarty *et al.* (2005b). There are two main differences: Firstly, an ML signature also defines value constructors for an algebraic data type, whereas a type class only declares the data type and defers value constructors to instance definitions. Secondly, a type specification of a signature may be implemented as a data type in a structure, whereas Haskell strictly distinguishes between type synonyms and algebraic data types.

4 Formal Translation From Classes to Modules

This section develops the formal translation from Haskell type classes to ML modules. In Section 4.1, we introduce our source language Tiny-HS, which is a subset of Tiny-HS$^+$ and of Haskell 98. Section 4.2 then defines the target language Tiny-ML$^+$ as an extension of Tiny-ML. The translation itself is presented in Section 4.3. In Section 4.4, we prove that type-correct Tiny-HS programs translate into type-correct Tiny-ML$^+$ programs. Finally, Section 4.5 discusses why full Haskell 98 type classes cannot be translated to ML modules.

An implementation of the translation is available from `http://www.informatik.uni-freiburg.de/~wehr/diplom/`. We do not discuss it here because the implementation techniques are standard (Wehr, 2005).

4.1 Tiny-HS

Tiny-HS is a simple formalization of the Haskell 98 (Peyton Jones, 2003) type class system, supporting all important features except constructor classes (Jones, 1995), class methods with constraints, and default definitions for methods. These restrictions are necessary to keep the translation from Tiny-HS to ML modules feasible (see Section 4.5 for details).

The syntax of Tiny-HS, shown in Figure 22, is a slightly modified subset of Tiny-HS$^+$'s syntax. We introduce a designated identifier set for methods because the translation treats them special. To ensure a coherent translation we need to rule out expressions with ambiguous type.[6]

Definition 4.1 (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.

[6] The classical example for an expression with ambiguous type is λz. show (read z) with show :: Show a ⇒ a → String and read :: Read a ⇒ String → a. Its type is ∀{a}. (Read a, Show a) ⇒ String → String with no way to tell which type we should pick for a.

Identifiers

$\mathsf{C} \in \mathsf{ClassId}; \quad \mathsf{T} \in \mathsf{TyconId} = \{\rightarrow, \mathsf{Int}, \ldots\}; \quad \mathsf{z} \in \mathsf{VarId}; \quad \mathsf{m} \in \mathsf{MethodId}; \quad \mathsf{a}, \mathsf{b} \in \mathsf{TypVar}$

Types

$$\begin{array}{rcl} \tau & ::= & \mathsf{a} \mid \mathsf{T}^{\kappa}\,\overline{\tau}^{\kappa} \\ \rho & ::= & \overline{\pi} \Rightarrow \tau \\ \sigma & ::= & \forall \mathsf{A}.\rho \\ \mathsf{A}, \mathsf{B} & \in & \textsc{Fin}(\mathsf{TypVar}) \end{array}$$

Constraints

$$\begin{array}{rcl} \pi & ::= & \mathsf{C}\,\tau \\ \theta & ::= & \forall \mathsf{A}.\overline{\pi} \Rightarrow \pi \end{array}$$

Expressions

$$\mathsf{w} ::= \mathsf{z} \mid \mathsf{m} \mid \lambda \mathsf{z}.\mathsf{w} \mid \mathsf{w}_1\ \mathsf{w}_2 \mid \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2$$

Classes, instances, and programs

$$\begin{array}{rcl} \mathsf{cls} & ::= & \textbf{class}\ \forall\{\mathsf{a}\}.\overline{\mathsf{C}_i\,\mathsf{a}}^{\,i\in[r]} \Rightarrow \mathsf{C}\,\mathsf{a}\ \textbf{where}\ \overline{\mathsf{msig}} \\ \mathsf{msig} & ::= & \mathsf{m} :: \forall \mathsf{A}.\tau \\ \mathsf{inst} & ::= & \textbf{instance}\ \forall \mathsf{A}.\overline{\mathsf{C}\,\mathsf{a}} \Rightarrow \mathsf{C}\,(\mathsf{T}^{\kappa}\,\overline{\tau}^{\kappa})\ \textbf{where}\ \overline{\mathsf{mval}} \\ \mathsf{mval} & ::= & \mathsf{m} = \mathsf{w} \\ \mathsf{pgm} & ::= & \overline{\mathsf{cls}}\ \overline{\mathsf{inst}}\ \mathsf{w} \end{array}$$

Figure 22. Syntax of Tiny-HS

We do not specify a static semantics for Tiny-HS explicitly because the translation in Section 4.3 is type-directed, which means that it does not only define the Tiny-ML$^+$counterpart of a Tiny-HS construct, but also assigns a type to it.

4.2 Tiny-ML$^+$

Tiny-ML is not suitable as 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. Hence, we extend Tiny-ML by adding first-class structures (Russo, 2000a), recursive functors (Crary *et al.*, 1999; Russo, 2001), lexically scoped type variables, arbitrary structure expressions as functor arguments, and signature expressions with type realizations (*e.g.*, S **where type** t = u). The resulting language is called Tiny-ML$^+$.

First-class structures and recursive functors are not part of Standard ML, so we omitted them from Tiny-ML. Both features are well-established and implemented in Moscow ML (Romanenko *et al.*, 2003). The three remaining features are part of Standard ML. We did not integrate them into Tiny-ML because they would complicate the translation from Tiny-ML to Tiny-HS$^+$ without giving more insights.

The syntactic changes between Tiny-ML and Tiny-ML$^+$ are displayed in Figure 23, which also shows the changes to semantic objects. The additional typing rules and typing judgments can be found in Figure 24. The following two sections now explains the two new Tiny-ML$^+$ features which are not part of Standard ML.

Types
u ::= $\cdots$ | <S>

Expressions
e ::= $\cdots$ | **pack** s **as** S | **open** e **as** X : S **in** e | **let** c : v = e **in** e

Signatures
S ::= $\cdots$ | S **where type** t=u

Structures
b ::= $\cdots$ | **val** x : v = e; b
s ::= $\cdots$ | F($\overline{\text{s}}$)

Programs
rfun ::= **functor** F($\overline{\text{X : S}}$) :> S = ps
rfuns ::= rfun; rfuns | ϵ_{rfuns}
prog ::= $\cdots$ | **rec** rfuns; prog

Semantic types
u ::= $\ldots$ | $<\mathcal{X}>$

Figure 23. Syntax and semantic objects for Tiny-ML$^+$ (extends Figures 8 and 9)

4.2.1 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 (Russo, 1998; Russo, 2000a; Russo, 2000b).

Figure 23 contains four constructs to support first-class structures. Simple types u now contain package types <S>, the syntactic types of first-class structures. The semantic counterpart is $<\mathcal{X}>$. The expression form **pack** s **as** S introduces a package type; its elimination form is **open** e **as** X : S **in** e′ which binds the structure packaged inside e to the structure variable X in e′. Figure 24 contains rules for relating a package type to its denotation and for typing package introduction and elimination.

4.2.2 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 (Russo, 2001) that allows us to define recursive functors. The design of this extension is oriented towards the use of Tiny-ML$^+$ as target language of the translation from type classes to modules.

A group of recursive functors is introduced by a new form of top-level definition **rec** rfuns; prog. Typing a group of recursive functors proceeds in two steps: judgment $\mathcal{C} \vdash \text{rfuns} \triangleright \mathcal{C}'$ first collects the semantic functors of the group, and judgment $\mathcal{C} \vdash \text{rfuns}$ then uses these them as forward declarations to check the type correctness of every functor body in the group.

The only way to use recursive functors is through first-class structures of the

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

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S}}{\mathcal{C} \vdash <\mathrm{S}> \triangleright <\exists P.\mathcal{S}>}$$

Classification of expressions $\boxed{\mathcal{C} \vdash \mathrm{e} : u}$

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

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

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

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

Classification of structures $\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{S} \quad \mathcal{C} \vdash \mathrm{s} : \mathcal{X}}$

$$\frac{\begin{array}{c}\mathcal{C}' \vdash \mathrm{u} \triangleright u \qquad \mathcal{C}' \vdash \mathrm{e} : u \qquad \mathcal{C}, \mathrm{x} \mapsto \forall B.u \vdash \mathrm{b} : \mathcal{S} \\ \mathcal{C}' = \mathcal{C} \overrightarrow{\cup} \textsc{Zip}(\mathrm{A}, B) \qquad B \cap \mathrm{FV}^{`a}(\mathcal{C}) = \emptyset \qquad |\mathrm{A}| = |B| \qquad \mathrm{x} \notin \textsc{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \textbf{val}\ \mathrm{x} : \forall \mathrm{A}.\mathrm{u} = \mathrm{e}; \mathrm{b}}$$

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

Recursive functors $\boxed{\mathcal{C} \vdash \mathrm{rfuns} \triangleright \mathcal{C}' \quad \mathcal{C} \vdash \mathrm{rfuns}}$

$$\frac{\mathcal{C} \vdash_{fargs} \overline{\mathrm{X} : \mathrm{S}} \triangleright \forall Q.\overline{\mathcal{S}} \quad \mathcal{C}, \overline{\mathrm{X} \mapsto \mathcal{S}} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \quad \mathcal{C} \vdash \mathrm{rfuns} \triangleright \mathcal{C}' \quad \mathrm{F} \notin \textsc{Dom}(\mathcal{C}')}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}(\overline{\mathrm{X} : \mathrm{S}}) :> \mathrm{S} = \mathrm{ps}; \mathrm{rfuns} \triangleright \mathcal{C}', \mathrm{F} \mapsto \forall Q.\overline{\mathcal{S}} \to \exists P.\mathcal{S}} \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{rfuns}} \triangleright \mathcal{C}}$$

$$\frac{\mathcal{C}(\mathrm{F}) = \forall Q.\overline{\mathcal{S}} \to \mathcal{X} \qquad \mathcal{C}, \overline{\mathrm{X} \mapsto \mathcal{S}} \vdash_{seal} \mathrm{ps} :> \mathrm{S} : \mathcal{X} \qquad \mathcal{C} \vdash \mathrm{rfuns}}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}(\overline{\mathrm{X} : \mathrm{S}}) :> \mathrm{S} = \mathrm{ps}; \mathrm{rfuns}} \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{rfuns}}}$$

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

$$\frac{\mathcal{C} \vdash \mathrm{rfuns} \triangleright \mathcal{C}' \qquad \mathcal{C}' \vdash \mathrm{rfuns} \qquad \mathcal{C}' \vdash \mathrm{prog}}{\mathcal{C} \vdash \textbf{rec}\ \mathrm{rfuns}; \mathrm{prog}}$$

Figure 24. Typing judgments for Tiny-ML^{+} (extends Figures 10 and 11)

core language. This restriction precludes recursion on the type level. Tiny-ML^{+}'s recursive functors are well-founded (*i.e.*, the definition of a recursive functor is evaluated without accessing one of the recursively defined variables) because the body of a functor is not evaluated until the functor is applied to some argument.

4.3 Formal Translation

The translation from Tiny-HS to Tiny-ML$^+$, which we develop in this chapter, is very similar to other evidence translations (Wadler & Blott, 1989; Jones, 1994; Hall *et al.*, 1996; Faxén, 2002) 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 give some definitions in Section 4.3.1 before we define the type-directed translation judgments in Sections 4.3.2 and 4.3.3.

4.3.1 Preliminaries

Identifier manipulation. Similar to the translation from modules to classes, we define injective functions with an intuitive shorthand notation for function application that convert Tiny-HS identifiers to Tiny-ML$^+$ identifiers:

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

We require that the images of all functions are pairwise disjoint. Furthermore, we postulate the existence of a set of fresh core identifiers, $\mathsf{FreshCoreIds} \subseteq \mathsf{CoreId}$, such that $\mathsf{FreshCoreIds} \cap \{\mathrm{c}^{\mathsf{z}} \mid \mathsf{z} \in \mathsf{VarId}\} = \emptyset$.

Environments. The translation from Tiny-HS to Tiny-ML$^+$ uses four different environments:

- A variable environment $\Gamma \in \mathsf{VarId} \cup \mathsf{MethodId} \xrightarrow{\text{fin}} \mathsf{TypSc}$ maps term and method variables to type schemes.
- A type environment $\Sigma \in \mathsf{TypVar} \xrightarrow{\text{fin}} \mathsf{SimTyp}$ maps Tiny-HS type variables to Tiny-ML$^+$ simple types.
- A constraint environment $\Theta = (\Theta^s, \Theta^i, \Theta^l)$ consists of three components: $\Theta^s \in \textsc{Fin}(\mathsf{ConstrSc})$ records constraint schemes resulting from subclass definitions; $\Theta^i \in \mathsf{ConstrSc} \xrightarrow{\text{fin}} \mathsf{FunId}$ maps constraint schemes originating from instance definitions to the names of the functors the instances are translated to; $\Theta^l \in \mathsf{Constr} \xrightarrow{\text{fin}} \mathsf{Exp}$ maps constraints to expressions providing evidence for the constraints.
- A signature environment $\Delta \in \mathsf{ClassId} \xrightarrow{\text{fin}} \mathsf{SigExp}$ maps classes to signature expressions that represent the classes in Tiny-ML$^+$.

It is sometimes convenient to treat Θ as a finite map and not as a triple. Hence, we extend the finite map operations $\dot\cup$ and $\overrightarrow{\cup}$ to constraint environments such that $\Theta \mathbin{\dot\cup} \Theta' = (\Theta^s \cup \Theta^{s\prime}, \Theta^i \mathbin{\dot\cup} \Theta^{i\prime}, \Theta^l \mathbin{\dot\cup} \Theta^{l\prime})$ and $\Theta \mathbin{\overrightarrow{\cup}} \Theta' = (\Theta^s \cup \Theta^{s\prime}, \Theta^i \mathbin{\overrightarrow{\cup}} \Theta^{i\prime}, \Theta^l \mathbin{\overrightarrow{\cup}} \Theta^{l\prime})$.

4.3.2 Translating types, constraints, and expressions

Figure 25 shows how types, constraints, and expressions are translated. The translation of type schemes and monotypes is straightforward. More interesting is the

Translation of types $\boxed{\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v} \quad \Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u} \quad \Sigma \vdash \tau \rightsquigarrow \mathrm{u}}$

$$\frac{\Delta; \Sigma \overrightarrow{\cup} \{\mathsf{a} \mapsto \text{`}\mathsf{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\} \vdash \rho \rightsquigarrow \mathrm{u}}{\Delta; \Sigma \vdash \forall \mathsf{A}.\rho \rightsquigarrow \forall\{\text{`}\mathsf{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\}.\mathrm{u}} \qquad \frac{\Sigma \vdash \tau \rightsquigarrow \mathrm{u} \qquad \Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}' \qquad \mathsf{C} \in \textsc{Dom}(\Delta)}{\Delta; \Sigma \vdash \mathsf{C}\,\tau \Rightarrow \rho \rightsquigarrow <\Delta(\mathsf{C})\ \textbf{where type}\ \mathbf{t} = \mathrm{u}> \to \mathrm{u}'}$$

$$\frac{\Sigma \vdash \tau_i \rightsquigarrow \mathrm{u}_i\ (i \in [\kappa])}{\Sigma \vdash \mathsf{T}^{\kappa}\ \overline{\tau}^{\kappa} \rightsquigarrow \mathsf{T}^{\kappa}\ \overline{\mathrm{u}}^{\kappa}} \qquad \frac{\Sigma(\mathsf{a}) = \mathrm{u}}{\Sigma \vdash \mathsf{a} \rightsquigarrow \mathrm{u}}$$

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

$$\frac{\Theta^{l}(\pi) = \mathrm{e}}{\Delta; \Sigma; \Theta \Vdash \pi \rightsquigarrow \mathrm{e}}$$

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

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

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

$$\frac{\begin{array}{c}\Gamma(\mathsf{z}) = \forall \mathsf{A}.\overline{\pi}^{n} \Rightarrow \tau' \qquad \psi(\tau') = \tau \\ \Delta; \Sigma; \Theta \Vdash \psi(\pi_i) \rightsquigarrow \mathrm{e}_i \qquad \textsc{Dom}(\psi) = \mathsf{A} \qquad (i \in [n])\end{array}}{\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{z} \rightsquigarrow \mathsf{c}^{\mathsf{z}}\ \overline{\mathrm{e}}^{n} : \tau}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\,\mathsf{b} \Rightarrow \tau' \qquad \psi(\tau') = \tau \\ \Delta; \Sigma; \Theta \Vdash \mathsf{C}\,\psi(\mathsf{b}) \rightsquigarrow \mathrm{e} \qquad \Sigma \vdash \psi(\mathsf{b}) \rightsquigarrow \mathrm{u} \qquad \mathsf{C} \in \textsc{Dom}(\Delta) \qquad \textsc{Dom}(\psi) = \mathsf{A}\end{array}}{\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{m} \rightsquigarrow \textbf{open}\ \mathrm{e}\ \textbf{as}\ (\mathrm{X} : \Delta(\mathsf{C})\ \textbf{where type}\ \mathbf{t} = \mathrm{u})\ \textbf{in}\ \mathrm{X}.\mathrm{x}^{\mathsf{m}} : \tau}$$

$$\frac{\begin{array}{c}\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w}_1 \rightsquigarrow \mathrm{e}_1 : \tau' \to \tau \\ \Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w}_2 \rightsquigarrow \mathrm{e}_2 : \tau'\end{array}}{\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w}_1\ \mathsf{w}_2 \rightsquigarrow \mathrm{e}_1\ \mathrm{e}_2 : \tau} \qquad \frac{\Sigma; \Theta; \Gamma, \mathsf{z} \mapsto \tau' \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \tau}{\Delta; \Sigma; \Theta; \Gamma \vdash \lambda \mathsf{z}.\mathsf{w} \rightsquigarrow \lambda \mathsf{c}^{\mathsf{z}}.\mathrm{e} : \tau' \to \tau}$$

$$\frac{\begin{array}{c}\Delta; \Sigma'; \Theta'; \Gamma \vdash \mathsf{w}_1 \rightsquigarrow \mathrm{e}_1 : \tau' \qquad \textsc{Gen}(\Theta', \Gamma, \tau') = \forall \mathsf{A}.\rho = \sigma \text{ unambiguous} \\ \Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v} \qquad \Delta; \Sigma; \Theta; \Gamma, \mathsf{z} \mapsto \sigma \vdash \mathsf{w}_2 \rightsquigarrow \mathrm{e}_2 : \tau \qquad \Sigma' = \Sigma \overrightarrow{\cup} \{\mathsf{a} \mapsto \text{`}\mathsf{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A}\} \\ \Theta' = (\Theta^{s}, \Theta^{i}, \{\overline{\pi_i \mapsto \mathrm{c}_i}^{i \in [n]}\}); \qquad \overline{\mathrm{c}}^{n} \in \text{FreshCoreIds and pairwise distinct}\end{array}}{\Delta; \Sigma; \Theta; \Gamma \vdash \textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2 \rightsquigarrow \textbf{let}\ \mathrm{c}^{\mathsf{z}} : \mathrm{v} = \overline{\lambda \mathrm{c}}^{n}.\mathrm{e}_1\ \textbf{in}\ \mathrm{e}_2 : \tau}$$

$$\textsc{Gen}((\Theta^{s}, \Theta^{i}, \{\overline{\pi \mapsto \mathrm{e}}\}), \Gamma, \tau) := \forall \mathsf{A}.\overline{\pi} \Rightarrow \tau$$
$$\text{where } \mathsf{A} = (\mathrm{FV}^{\mathsf{a}}(\overline{\pi}) \cup \mathrm{FV}^{\mathsf{a}}(\tau)) \setminus (\mathrm{FV}^{\mathsf{a}}(\Gamma) \cup \mathrm{FV}^{\mathsf{a}}(\Theta^{s}) \cup \mathrm{FV}^{\mathsf{a}}(\Theta^{i}))$$

Figure 25. Translation of types, constraints, and expression

translation of qualified types: A type $\mathsf{C}\,\tau \Rightarrow \rho$ translates into a function type whose domain is the type of $\mathsf{C}\,\tau$ dictionaries. We encode a dictionary for $\mathsf{C}\,\tau$ as a first-class structure of signature $\Delta(\mathsf{C})$ such that the type component **t** is equal to to the translation of τ. Here, $\Delta(\mathsf{C})$ is the translation of C and **t** represents the class variable in C's definition. (We use a bold **t** to emphasize that it is a designated type identifier.) The premise $\Delta; \Sigma \vdash \rho \rightsquigarrow \mathsf{u}'$ of the translation rule for qualified types should be read as $\Sigma \vdash \tau \rightsquigarrow \mathsf{u}'$ if $\rho = \tau$.

The judgment $\Delta; \Sigma; \Theta \Vdash \pi \rightsquigarrow \mathsf{e}$ states that π is entailed by the environments and gives evidence for this fact in form of a dictionary expression e. The type of e is that of a π dictionary. The second rule of this judgment 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 to F represents the type variables free in the instance head but not constrained by the instance context. The last rule of the judgment derives evidence for a superclass from a subclass constraint. Hence, the resulting expression opens the first-class structure for the subclass constraint and selects the dictionary $\mathsf{x}^{\mathsf{C}^{\mathsf{sup}}}$ for the superclass.

The judgment $\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathsf{e} : \tau$ assigns type τ to the Tiny-HS expression w and defines the Tiny-ML$^+$expression e as its translation. The interesting rules are those for methods and let-expressions.

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 e and project the value component x^{m}, giving the translation of m, from it.

To type and translate let-expressions of the form $\textbf{let}\ \mathsf{z} = \mathsf{w}_1\ \textbf{in}\ \mathsf{w}_2$ we need to guess the quantified type variables A and the constraints π_i of σ (the generalized type of w_1) because the extended environments Σ' and Θ' are used in the typing/translation derivation for w_1. The translated let-expression needs to be annotated with v (the translation of σ) because the subexpression e_1 may contain some of the simple type variables $\text{`}\mathsf{a}^{\mathsf{a}}$ for $\mathsf{a} \in \mathsf{A}$.

The definition of generalization needs to be careful to preserve the ordering of constraints in a qualified type $\overline{\pi} \Rightarrow \tau$ because dictionary parameters are passed in the same order as the constraints $\overline{\pi}$ are written. We ensure proper ordering by implicitly using an order relation on the whole set of constraints.

4.3.3 Translating instances, classes, and programs

Figure 26 shows the translation judgments for instances, classes, and programs. The judgment $\vdash \mathsf{inst} \rightsquigarrow \Theta$ just collects the constraint scheme implied by an instance definition and associates it with a fresh functor identifier. The judgment $\Delta; \Theta; \Gamma \vdash \mathsf{inst} \rightsquigarrow \mathsf{rfun}$ 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

Translation of instances $\boxed{\vdash \mathsf{inst} \rightsquigarrow \Theta \quad \Delta;\Theta;\Gamma \vdash \mathsf{inst} \rightsquigarrow \mathsf{rfun}}$

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

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

$$\frac{\begin{array}{c}\Gamma(\mathsf{m}) = \forall \mathsf{A}.\mathsf{C}\,\mathsf{b} \Rightarrow \tau' \\ \Delta;\Sigma';\Theta;\Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : [\tau/\mathsf{b}]\tau' \qquad \Sigma' \vdash [\tau/\mathsf{b}]\tau' \rightsquigarrow \mathrm{u}' \qquad \mathrm{v} = \forall\{\text{`}\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\}.\mathrm{u}' \\ \Sigma' = \{\mathsf{a} \mapsto \text{`}\mathrm{a}^{\mathsf{a}} \mid \mathsf{a} \in \mathsf{A} \setminus \{\mathsf{b}\}\} \qquad \mathsf{A} \cap (\mathrm{FV}^{\mathsf{a}}(\tau) \cup \mathrm{FV}^{\mathsf{a}}(\Theta) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma)) = \emptyset\end{array}}{\Delta;\Sigma;\Theta;\Gamma;\tau \vdash_{\mathit{method}} \mathsf{m} = \mathsf{w} \rightsquigarrow \mathrm{e} : \mathrm{v}}$$

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

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

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

$$\frac{\begin{array}{c}\mathsf{pgm} = \overline{\mathsf{cls}}^{\,n}\ \overline{\mathsf{inst}}^{\,m} \qquad \mathsf{pgm} \text{ well-formed} \\ \dot\cup_{j\in[i-1]}\Delta_j \vdash \mathsf{cls}_i \rightsquigarrow \Delta_i;\Theta_i;\Gamma_i \qquad \vdash \mathsf{inst}_k \rightsquigarrow \Theta'_k \\ \Delta = \dot\cup_{i\in[n]}\Delta_i \qquad \Theta = \dot\bigcup_{i\in[n]}\Theta_i \,\dot\cup\, \dot\bigcup_{k\in[m]}\Theta'_k \qquad \Gamma = \dot\bigcup_{i\in[n]}\Gamma_i \\ \Delta;\Theta;\Gamma \vdash \mathsf{inst}_k \rightsquigarrow \mathrm{rfun}_k \qquad \Delta;\emptyset;\Theta;\Gamma \vdash \mathsf{w} \rightsquigarrow \mathrm{e} : \mathsf{Int} \qquad (i \in [n], k \in [m])\end{array}}{\begin{array}{l}\vdash \mathsf{pgm} \rightsquigarrow \textbf{rec } \overline{\mathrm{rfun}}^{\,m}; \\ \qquad \textbf{structure Main} = \textbf{struct val main} = \mathrm{e} \textbf{ end}\end{array}}$$

Figure 26. Translation of instances, classes, and programs

instance context. The extra parameter Y represents the type variables free in the instance head but not constrained by the instance context.

The rule defining the judgment $\Delta;\Theta;\Gamma \vdash \mathsf{inst} \rightsquigarrow \mathsf{rfun}$ translates the method

implementations and derives 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 annotation v_i for $\mathrm{x}^{\mathsf{m}_i}$ is needed because e_i might contain some of the universally quantified type variables of v_i. The constraint 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 $\mathrm{X}_i.\mathbf{t}$, where $\mathsf{C}_i\,\mathsf{a}_i$ is the first constraint with $\mathsf{a} = \mathsf{a}_i$; a type variable b free in the instance head but not constrained by the instance context is bound to $\mathrm{Y}.\mathrm{t}^{\mathsf{b}}$. Notice 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 mainly constructs the signature S as the translation of cls. The rule defining the translation judgment for programs, $\vdash \mathsf{pgm} \rightsquigarrow \mathsf{prog}$, first collects the environments resulting from class and instance definitions. Then it translates the instance definitions into recursive functors and the Tiny-HS main expression into a Tiny-ML$^+$ expression. Finally, it constructs a group of recursive functors and defines the main structure. The rule is restricted to well-formed Tiny-HS programs; here, a Tiny-HS program is well-formed if and only if every class is defined at most once and used only after its definition, and if instance heads do not overlap.

4.4 Formal Properties

This section proves that the translation from Tiny-HS to Tiny-ML$^+$ produces only type correct programs provided the source program is type correct. We omitted detailed proofs and technical lemmata; they can be found elsewhere (Wehr, 2005).

It is convenient to introduce some notation for expressing that a Tiny-HS type translates into a semantic object of Tiny-ML$^+$:

- $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$ stands for $\Sigma \vdash \tau \rightsquigarrow \mathrm{u}$ and $\mathcal{C} \vdash \mathrm{u} \triangleright u$.
- $\Delta; \Sigma; \mathcal{C} \vdash \rho \rightsquigarrow \mathrm{u}$ stands for $\Delta; \Sigma \vdash \rho \rightsquigarrow \mathrm{u}$ and $\mathcal{C} \vdash \mathrm{u} \triangleright u$.
- $\Delta; \Sigma; \mathcal{C} \vdash \sigma \rightsquigarrow v$ stands for $\Delta; \Sigma \vdash \sigma \rightsquigarrow \mathrm{v}$ and $\mathcal{C} \vdash \mathrm{v} \triangleright v$.

At some points, we also use a translation judgment $\mathcal{T} \vdash \tau \rightsquigarrow u$ defined by

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

where $\mathcal{T} \in \mathsf{TypVar} \xrightarrow{\text{fin}} \mathit{SimTyp}$. It is easy to show that we can replace every use of $\mathcal{T} \vdash \tau \rightsquigarrow u$ with an application of $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$.

A well-typed signature environment Δ correctly reflects the class definitions of a Tiny-HS program. The following definition makes this notion precise.

Definition 4.2 (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 and Γ), iff $\emptyset \vdash \Delta(\mathsf{C}) \triangleright \Lambda\{\alpha\}.\mathcal{S}$ for all $\mathsf{C} \in \textsc{Dom}(\Delta)$ such that

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

If only the first condition is relevant, we call Δ simply well-typed.

The next step is to define a notion of consistency between Tiny-HS environments and Tiny-ML$^+$ contexts. In the following, we call a constraint environment Θ well-formed iff all $\theta \in \Theta^s \cup \textsc{Dom}(\Theta^i)$ are well-formed. Similarly, a variable Γ environment is called unambiguous iff all $\sigma \in \textsc{Img}(\Gamma)$ are unambiguous. Moreover, we define a function $\textsc{ClassIds}(\cdot)$ which yields the set of class identifiers in a Tiny-HS construct. Finally, we introduce the notation $\mathcal{S}_\Delta(\mathsf{C}, u)$ for the semantic object of a structure representing the dictionary of an instance $\mathsf{C}\,\tau$ where u is the semantic object of τ's translation; we define $\mathcal{S}_\Delta(\mathsf{C}, u)$ as $[u/\alpha]\mathcal{S}$ for $\Delta(\mathsf{C}) = \Lambda\{\alpha\}.\mathcal{S}$.

Definition 4.3 (Consistency of Tiny-ML$^+$ contexts)**.** Let Δ be well-typed w.r.t. Θ and Γ, let Θ be well-formed, let Γ be unambiguous, and suppose that $\textsc{ClassIds}(\Theta) \cup \textsc{ClassIds}(\Gamma) \subseteq \textsc{Dom}(\Delta)$ and $\mathrm{FV}^{\mathsf{a}}(\Theta) \cup \mathrm{FV}^{\mathsf{a}}(\Gamma) \subseteq \textsc{Dom}(\Sigma)$ for a type environment Σ. A context $\mathcal{C}$ is said to be consistent with Δ, Σ, Θ, and Γ, if and only if the following conditions hold:

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

We first prove that the translation of a constraint π is indeed a dictionary for π.

Lemma 4.4 (Type correctness of constraint translation)**.** *Let $\mathcal{C}$ be consistent with Δ, Σ, Θ, and some arbitrary Γ. Suppose $\mathsf{C} \in \textsc{Dom}(\Delta)$ and $\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \textsc{Dom}(\Sigma)$. If $\Delta; \Sigma; \Theta \Vdash \mathsf{C}\,\tau \rightsquigarrow \mathsf{e}$, then $\mathcal{C} \vdash \mathsf{e} : <\mathcal{S}_\Delta(\mathsf{C}, u)>$ with $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$.*

Proof. By rule induction.

The next step is to show that translating a Tiny-HS expression of type τ yields a Tiny-ML$^+$ expression of type u where u is the semantic object of τ's translation.

Lemma 4.5 (Type correctness of expression translation)**.** *Suppose* $\mathcal{C}$ *is consistent with* Δ*,* Σ*,* Θ*, and* Γ *such that* $\mathrm{FV}^{\mathsf{a}}(\Gamma(\mathsf{m})) = \emptyset$ *for all* $\mathsf{m} \in \textsc{Dom}(\Gamma)$ *and* $\Delta; \Sigma; \mathcal{C} \vdash \Gamma(\mathsf{z}) \rightsquigarrow \mathcal{C}(\mathsf{c}^{\mathsf{z}})$ *for all* $\mathsf{z} \in \textsc{Dom}(\Gamma)$*. If* $\Delta; \Sigma; \Theta; \Gamma \vdash \mathsf{w} \rightsquigarrow \mathsf{e} : \tau$ *and* $\mathrm{FV}^{\mathsf{a}}(\tau) \subseteq \textsc{Dom}(\Sigma)$*, then* $\mathcal{C} \vdash \mathsf{e} : u$ *with* $\Sigma; \mathcal{C} \vdash \tau \rightsquigarrow u$*.*

Proof. By induction on the structure of w.

The following lemma shows that the recursive functor resulting as the translation of an instance definition correctly reflects the instance and type checks.

Lemma 4.6 (Type correctness of instance translation)**.** *Suppose that* $\Delta; \Theta; \Gamma \vdash \mathsf{inst} \rightsquigarrow \mathsf{rfun}$ *and that* $\mathcal{C}$ *is consistent with* Δ*,* $\Sigma = \emptyset$*,* Θ*, and* Γ*.*

- *If* $\vdash \mathsf{inst} \rightsquigarrow \Theta'$ *with* $\Theta' \subseteq \Theta$*, then* $\emptyset \vdash \mathsf{rfun} \triangleright \{\mathrm{F} \mapsto \mathcal{C}(\mathrm{F})\}$ *and* $\{\mathrm{F} \mapsto \mathcal{C}(\mathrm{F})\}$ *is consistent with* Δ*,* $\Sigma = \emptyset$*,* Θ'*, and* Γ*.*
- *If* $\emptyset \vdash \mathsf{rfun} \triangleright \{\mathrm{F} \mapsto \mathcal{C}(\mathrm{F})\}$ *and* $\mathsf{rfun} = \textbf{functor}\ \mathrm{F}(\overline{\mathrm{X}_i : \mathrm{S}_i}^{i \in [r]}, \mathrm{Y} : \mathrm{S}_{\mathrm{Y}}) :> \mathrm{S} = \mathrm{ps}$ *and* $\mathcal{C}(\mathrm{F}) = \forall P.\overline{\mathcal{S}}^{r+1} \to \mathcal{S}$*, then* $\mathcal{C}, \overline{\mathrm{X}_i \mapsto \mathcal{S}_i}^{i \in [r]}, \mathrm{Y} \mapsto \mathcal{S}_{r+1} \vdash_{\mathit{seal}} \mathrm{ps} :> \mathrm{S} : \mathcal{S}$*.*

Proof. The proofs of the two propositions are straightforward but require some technical lemmata (Wehr, 2005).

We now show that the translation judgment for classes produces only well-typed class environments.

Lemma 4.7 (Well-typedness of class environments)**.** *Suppose* $\Delta \vdash \mathsf{cls} \rightsquigarrow \Delta'; \Theta'; \Gamma'$ *and let* C *be the class defined by* cls*. If* Δ *is well-typed w.r.t. some* Θ *and some* Γ *such that* $\textsc{Sup}(\Theta, \mathsf{C}) = \emptyset$ *and* $\textsc{Dom}(\Delta) \cap \textsc{Dom}(\Delta') = \textsc{Dom}(\Gamma) \cap \textsc{Dom}(\Gamma') = \emptyset$ *and* $\Gamma(\mathsf{m}) \neq \forall \mathsf{A}.\mathsf{C}\ \mathsf{b} \Rightarrow \rho$ *for all* m*, then we have that* $\Delta \dot\cup \Delta'$ *is well-typed w.r.t.* $\Theta \dot\cup \Theta'$ *and* $\Gamma \dot\cup \Gamma'$*.*

Proof. Straightforward.

Eventually, we are able to prove the main result of this section.

Theorem 4.8 (Type correctness of program translation)**.** *If* $\vdash \mathsf{pgm} \rightsquigarrow \mathsf{prog}$*, then* $\emptyset \vdash \mathsf{prog}$*.*

Proof. Follows from Lemmata 4.5, 4.6, and 4.7.

4.5 Lifting Restrictions

The source language Tiny-HS of our translation does not support all features of Haskell 98 type classes. We now discuss why these restriction are necessary and—if possible—how we could extend the translation to remove them.

Constructor classes. Constructor classes (Jones, 1995) range over (possibly higher-order) type constructors. Higher-order type constructors are not supported by Standard ML, so we also omitted them from our source language Tiny-HS. At first sight, it seems possible to translate constructor classes ranging over first-order type constructors. For example, the `Monad` class from the Haskell 98 prelude could be written in ML as follows:

```
signature Monad =
  sig
    type 'a m
    val  >>=    : 'a m -> ('a -> 'b m) -> 'b m
    val  return : 'a -> 'a m
  end
```

However, we run into serious problems when we try to translate a type scheme such as `Monad m => m a -> m b`. Here are two non-solutions:

- The type `<Monad where type m = 'm> -> 'a 'm -> 'b 'm` is illegal because the type variable `'m` ranges over a higher-order type constructor.
- The type `<Monad where type 'a m = 'c and type 'b m = 'd> -> 'c -> 'd` is illegal as well because the second type realization `type 'b m = 'd` refers to the now transparent type component `m`

To solve the problem properly, we would need to name the content of a package type; then, something like `<X:Monad> -> 'a X.m -> 'b X.m -> 'b X.m` could do.

Neither Tiny-ML$^+$ nor Russo's proposal for first-class structures (Russo, 2000a) support 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. However, this encoding would be very heavy-weight.

Class methods with constraints. Method signatures in Haskell 98 may contain constraints, which may lead to recursive classes. Here is a rather artificial example:

```
class C a where
    foo :: C b => b -> a
```

We cannot translate such a recursive class into an ML signature because the resulting signature would be recursive as well.[7] Inventing some syntax for binding recursive signature variables, we could write the signature for `C` as follows:

```
fix Z. sig
         type t
         val  foo : <Z where type t = 'b> -> 'b -> t
       end
```

Default definitions for methods. In Haskell 98, type classes 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.

[7] Crary, Harper, and Puri (1999) introduced the notion of *recursively dependent signatures*; however, in their setting, the recursion variable is a structure, not a signature variable.

5 Discussion

After having developed the formal translations from ML modules to Haskell type classes and vice versa, we present in Section 5.1 a thorough comparison between the two concepts. Section 5.2 summarizes our contributions and concludes.

5.1 ML modules and Haskell type classes: a comparison

Sections 3 and 4 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 5.1.1 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 5.1.2.

5.1.1 Classes as modules

The translation from ML modules to Haskell type classes in Section 3 demonstrates that Haskell type classes can be used to simulate certain aspects of the ML module system. 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 (in the same Haskell module) cannot define two members of the same name.

Signature and structure components. Signatures and structures in ML may contain all sorts of language constructs, including substructures. Type classes and instances in Haskell 98 may contain only methods; extensions to Haskell 98 also allow type synonyms (Chakravarty *et al.*, 2005a) and data types (Chakravarty *et al.*, 2005b). 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. (Chakravarty *et al.*, 2005a) need extra conditions to ensure a terminating rewrite system on associated type synonyms. For our purpose, their termination conditions are too restrictive; for example, translating an ML program in which a structure defines a type component in terms of an opaque type component of another structure would result in an ill-typed Haskell program. Hence, our target language does not enforce a terminating rewrite system, which ultimately leads to undecidable type inference. However, for the class of programs arising as the translation of ML

programs, the rewrite system is terminating because the sequential nature of type components carries over to associated type synonym definitions.

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, once we introduce recursive modules, we need explicit signatures in ML as well, so the difference between implicit and explicit signatures goes back 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 cannot be anonymous.

Structural versus nominal signature matching. The difference between anonymous and named signatures becomes relevant when 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 (Shao, 1999). Signatures in Haskell (*i.e.*, type classes) are transparent: The definitions of associated type synonyms in instances are always visible through a type class. 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. Standard ML supports incremental compilation, and there are extensions supporting separate compilation (Leroy, 1994; Harper & Lillibridge, 1994; Shao, 1999). 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 (Leroy, 1994; Harper & Lillibridge, 1994).

Unsealed and sealed view. A sealed structure may look different depending on whether we view the structure 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.

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.

5.1.2 Modules as classes

In Section 4, we demonstrated how to simulate ad-hoc polymorphism introduced by Haskell type classes with ML modules. 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 (see also Section 3.5).

Implicit versus explicit overloading resolution. Overloading in Haskell is resolved implicitly by the compiler. When type classes are simulated with ML modules, overloading has to be resolved explicitly by the programmer, which leads to awkward and overly verbose code.

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 in 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 because type inference then would need to rely on undecidable higher-oder unification (Goldfarb, 1981).

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.

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.

5.2 Summary and conclusions

In this work, we demonstrated how ML modules can be translated to Haskell type classes, proved that the translation preserves type correctness, and provided an implementation of 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 to Haskell. Our practical experience suggests that it is feasible to use the general idea behind the translation for practical programming because some of the overhead introduced by the formal translation is avoided when writing the Haskell code by hand.

Furthermore, we showed that Haskell type classes can be translated into ML modules by using first-class structures as runtime evidence for type class constraints. We proved that this translation also preserves type correctness and implemented it. 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. It is not recommended writing programs in the style of the translation by hand because too much syntactic overhead is introduced by explicit dictionary abstraction and application. However, the translation provides a good starting point for integrating type classes into the ML module system.

Finally, we 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.

References

Chakravarty, Manuel M. T., Keller, Gabriele, & Peyton Jones, Simon. 2005a (Sept.). Associated type synonyms. *Pages 241–253 of: ACM SIGPLAN International Conference on Functional Programming (ICFP), Tallinn, Estonia.*

Chakravarty, Manuel M. T., Keller, Gabriele, Peyton Jones, Simon, & Marlow, Simon. 2005b (Jan.). Associated types with class. *Pages 1–13 of: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California.*

Crary, Karl, Harper, Robert, & Puri, Sidd. 1999 (May). What is a recursive module? *Pages 50–63 of: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia*.

Damas, Luis, & Milner, Robin. 1982 (Jan.). Principal type schemes for functional programs. *Pages 207–212 of: ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*.

Diatchki, Iavor S., Jones, Mark P., & Hallgren, Thomas. 2002 (Oct.). A formal specification of the Haskell 98 module system. *Pages 17–28 of: ACM Haskell Workshop, Pittsburgh, Pennsylvania*.

Dreyer, Derek, Crary, Karl, & Harper, Robert. 2003 (Jan.). A type system for higher-order modules. *Pages 236–249 of: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*.

Dreyer, Derek, Harper, Robert, Chakravarty, Manuel M. T., & Keller, Gabriele. (2006). *Modular type classes*. Tech. rept. TR-2006-03. University of Chicago.

Faxén, Karl-Filip. (2002). A static semantics for Haskell. *Journal of Functional Programming*, **12**(4&5), 295–357.

Girard, Jean-Yves. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, University of Paris VII. Summary in J. E. Fenstad, editor, *Scandinavian Logic Symposium*, pages 63–92, North-Holland, 1971.

Goldfarb, Warren D. (1981). The undecidability of the second-order unification problem. *Theoretical Computer Science*, **13**(2), 225–230.

Hall, Cordelia, Hammond, Kevin, Peyton Jones, Simon, & Wadler, Philip. (1996). Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, **18**(2), 109–138.

Harper, Robert, & Lillibridge, Mark. 1994 (January). A type-theoretic approach to higher-order modules with sharing. *Pages 123–137 of: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon*.

Harper, Robert, & Stone, Chris. (2000). A type-theoretic interpretation of Standard ML. Plotkin, Gordon, Stirling, Colin, & Tofte, Mads (eds), *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press.

Henglein, Fritz. (1993). Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, **15**(2), 253–289.

Jones, Mark P. (1994). *Qualified types: Theory and practice*. Cambridge University Press.

Jones, Mark P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, **5**(1), 1–35.

Jones, Mark P., & Peterson, John. (1999). *The Hugs 98 user manual*. Available from `http://www.haskell.org/hugs/`.

Kaes, Stefan. (1988). Parametric overloading in polymorphic programming languages. *Pages 131–144 of: European Symposium on Programming (ESOP), Nancy, France*. Lecture Notes in Computer Science, vol. 300. Springer-Verlag.

Kahl, Wolfram, & Scheffczyk, Jan. 2001 (Sept.). Named instances for Haskell type classes. *ACM Haskell Workshop, Firenze, Italy*, informal proceedings. Technical Report UU-CS-2001-23, Institute of Information and Computer Sciences, Utrecht University.

Kiselyov, Oleg. 2004 (Aug.). *Applicative translucent functors in Haskell*. Post to the Haskell mailing list, `http://www.haskell.org/pipermail/haskell/2004-August/014463.html`.

Leroy, Xavier. 1994 (January). Manifest types, modules and separate compilation. *Pages 109–122 of: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon*.

Leroy, Xavier. 1995 (Jan.). Applicative functors and fully transparent higher-order modules. *Pages 142–153 of: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California.*

Leroy, Xavier. (2000). *The Objective Caml system: Documentation and user's manual.* With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

MacQueen, David B. (1986). Using dependent types to express modular structure. *Pages 277–286 of: ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida.*

MacQueen, David B., & Tofte, Mads. (1994). A semantics for higher-order functors. *European Symposium on Programming (ESOP), Edinburgh, Scotland.* Lecture Notes in Computer Science, vol. 788. Springer-Verlag.

Milner, Robin. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**(Aug.), 348–375.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The definition of Standard ML,* revised edition. MIT Press.

Mitchell, John C., & Plotkin, Gordon D. (1988). Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, **10**(3), 470–502. Summary in *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, 1985.

Peyton Jones, Simon (ed). (2003). *Haskell 98 language and libraries: the revised report.* Cambridge University Press.

Peyton Jones, Simon, & Shields, Mark. 2004 (Apr.). *Practical type inference for arbitrary-rank types.* `http://research.microsoft.com/~simonpj/papers/putting/index.htm`.

Peyton Jones, Simon, Jones, Mark, & Meijer, Erik. 1997 (June). Type classes: An exploration of the design space. *ACM Haskell Workshop, Amsterdam, The Netherlands*, informal proceedings.

Romanenko, Sergei, Russo, Claudio, Kokholm, Niels, Larsen, Ken Friis, & Sestoft, Peter. (2003). *Moscow ML homepage.* `http://www.dina.dk/~sestoft/mosml.html`.

Rossberg, Andreas. 2005 (May). *Re: CTM Chapter 4.* Post to the Alice-users mailing list, `http://www.ps.uni-sb.de/pipermail/alice-users/2005/000466.html`.

Rossberg, Andreas, & Sulzmann, Martin. (2002). *Beyond type classes.* Tech. rept. Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany. `http://www.ps.uni-sb.de/Papers/paper_info.php?label=btclasses`.

Russo, Claudio V. (1998). *Types for modules.* Ph.D. thesis, Edinburgh University, Edinburgh, Scotland. LFCS Thesis ECS–LFCS–98–389.

Russo, Claudio V. (2000a). First-class structures for Standard ML. *Pages 336–350 of: European Symposium on Programming (ESOP), Berlin, Germany.* Lecture Notes in Computer Science, vol. 1782. Springer-Verlag.

Russo, Claudio V. (2000b). First-class structures for Standard ML. *Nordic journal of computing*, **7**(4), 348–374.

Russo, Claudio V. 2001 (Sept.). Recursive structures for Standard ML. *Pages 50–61 of: ACM SIGPLAN International Conference on Functional Programming (ICFP), Firenze, Italy.*

Schneider, Gerhard. 2000 (June). *ML mit Typklassen.* Diplomarbeit. Fachrichtung Informatik, Universität des Saarlandes. `http://www.ps.uni-sb.de/Papers/abstracts/Schneider2000.html`.

Shan, Chung-chieh. 2004 (July). *Higher-order modules in System F_ω and Haskell.* `http://www.eecs.harvard.edu/~ccshan/xlate/`.

Shao, Zhong. 1999 (Sept.). Transparent modules with fully syntactic signatures. *Pages 220–232 of: ACM SIGPLAN International Conference on Functional Programming (ICFP), Paris, France.*

Sulzmann, Martin, & Wazny, Jeremy. (2005). *Chameleon.* `http://www.comp.nus.edu.sg/~sulzmann/chameleon/`.

Sulzmann, Martin, Wazny, Jeremy, & Stuckey, Peter J.. (2005). *Co-induction, type improvement and cyclic superclasses in type class proofs.* `http://www.comp.nus.edu.sg/~sulzmann/type-class-proofs.pdf`.

Wadler, Philip, & Blott, Stephen. 1989 (Jan.). How to make *ad-hoc* polymorphism less *ad hoc. Pages 60–76 of: ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas.*

Wehr, Stefan. 2005 (Nov.). *ML modules and Haskell type classes: A constructive comparison.* Diplomarbeit. Albert-Ludwigs-Universität Freiburg, Fakultät für Angewandte Wissenschaften, Institut für Informatik. `http://www.informatik.uni-freiburg.de/~wehr/diplom/`.