Type classes in Haskell

Cordelia Hall, Kevin Hammond, Simon Peyton Jones and Philip Wadler
University of Glasgow*

January 20, 1994

Abstract

This paper defines a set of type inference rules for resolving overloading introduced by type classes. Programs including type classes are transformed into ones which may be typed by the Hindley-Milner inference rules. In contrast to other work on type classes, the rules presented here relate directly to user programs. An innovative aspect of this work is the use of second-order lambda calculus to record type information in the program.

1 Introduction

A funny thing happened on the way to Haskell [HPW92]. The goal of the Haskell committee was to design a standard lazy functional language, applying existing, well-understood methods. To the committee's surprise, it emerged that there was no standard way to provide overloaded operations such as equality `(==)`, arithmetic `(+)`, and conversion to a string `(show)`.

Languages such as Miranda[1][Tur85] and Standard ML [MTH90, MT91] offer differing solutions to these problems. The solutions differ not only between languages, but within a language. Miranda uses one technique for equality (it is defined on all types – including abstract types on which it should be undefined!), another for arithmetic (there is only one numeric type), and a third for string conversion. Standard ML uses the same technique for arithmetic and string conversion (overloading must be resolved at the point of appearance), but a different one for equality (type variables that range only over equality types).

The committee adopted a completely new technique, based on a proposal by Wadler, which extends the familiar Hindley-Milner system [Mil78] with *type classes*. Type classes provide a uniform solution to overloading, including providing operations for equality, arithmetic, and string conversion. They generalise the idea of equality types from Standard ML, and subsume the approach to string conversion used in Miranda. This system was originally described by Wadler and Blott [WB89, Blo91], and a similar proposal was made independently by Kaes [Kae88].

The type system of Haskell is certainly its most innovative feature, and has provoked much discussion. There has been closely related work by Rouaix [Rou90] and Comack and Wright [CW90], and work directly inspired by type classes includes Nipkow and Snelting [NS91], Volpano and Smith [VS91], Jones [Jon92a, Jon93], Nipkow and Prehofer [NP93], Odersky and Läufer [OdLä91], Läufer [Läu92, Läu93], and Chen, Hudak and Odersky [CHO92].

The paper presents a source language (lambda calculus with implicit typing and with overloading) and a target language (polymorphic lambda calculus with explicit typing and without overloading). The semantics of the former is provided by translation into the latter, which has a well-known semantics [Hue 90]. Normally, one expects a theorem stating that the translation is sound, in that the translation *preserves* the meaning of programs. That is not possible here, as the translation *defines* the meaning of programs. It is a grave shortcoming of the system presented here is that there is no direct way of assigning meaning to a program, and it must be done indirectly via translation; but there appears to be no alternative. (Note, however, that [Kae88] does give a direct semantics for a slightly simpler form of overloading.)

The original type inference rules given in [WB89] were deliberately rather sparse, and were not intended to reflect the Haskell language precisely. As a result, there has been some confusion as to precisely how type classes in Haskell are defined.

1.1 Contributions of this paper

This paper spells out the precise definition of type classes in Haskell. These rules arose from a practical impetus: our attempts to build a compiler for Haskell. The rules were written to provide a precise specification of *what* type classes were, but we found that they also provided a blueprint for *how* to implement them.

This paper presents a simplified subset of the rules we derived. The full static semantics of Haskell[PW91] contains over 30 judgement forms and over 100 rules. The reader will be pleased to know that this paper simplifies the rules considerably, while maintaining their essence in so far as

*This work is supported by the SERC AQUA Project. Authors' address: Computing Science Dept, Glasgow University, 17 Lilybank Gdns., Glasgow, Scotland. Email: {cvh, kh, simonpj, wadler}@dcs.glasgow.ac.uk

[1]Miranda is a trademark of Research Software Limited.

type classes are concerned. The full rules are more complex because they deal with many additional syntactic features such as type declarations, pattern matching, and list comprehensions.

This paper shows how the static analysis phase of our Haskell compiler was derived by adopting directly the rules in the static semantics. This was generally a very straightforward task. In our earlier prototype compiler, and also in the prototype compilers constructed at Yale and Chalmers, subtleties with types caused major problems. Writing down the rules has enabled us to discover bugs in the various prototypes, and to ensure that similar errors cannot arise in our new compiler.

We have been inspired in our work by the formal semantics of Standard ML prepared by Milner, Tofte, and Harper [MTH90, MT91]. We have deliberately adopted many of the same techniques they use for mastering complexity.

This approach unites theory and practice. The industrial grade rules given here provide a useful complement to the more theoretical approaches of Wadler and Blott [WB89, Blo91], Nipkow and Snelting [NS91], Nipkow and Prehofer [NP93], and Jones [Jon92a, Jon93]. A number of simplifying assumptions made in those papers are not made here. Unlike [WB89], it is not assumed that each class has exactly one operation. Unlike [NS91], it is not assumed that the intersection of every pair of classes must be separately declared. Unlike [Jon92a], we deal directly with instance and class declarations. Each of those papers emphasises one aspect or another of the theory, while this paper stresses what we learned from practice. At the same time, these rules and the monad-based[Wad92] implementation they support provide a clean, 'high-level' specification for the implementation of a typechecker, unlike more implementation oriented papers [HaBl89, Aug93, Jon92b].

A further contribution of this work is the use of explicit polymorphism in the target language, as described in the next section.

1.2 A target language with explicit polymorphism

As in [WB89, NS91, Jon92a], the rules given here specify a translation from a *source* language with type classes to a *target* language without them. The translation implements type classes by introducing extra parameters to overloaded functions, which are instantiated at the calling point with dictionaries that define the overloaded operations.

The target language used here differs in that all polymorphism has been made explicit. In [WB89, NS91, Jon92a], the target language resembles the implicitly typed polymorphic lambda calculus of Hindley and Milner [Hin69, Mil78, DM82]. Here, the target language resembles the explicitly typed second-order polymorphic lambda calculus of Girard and Reynolds [Gir72, Rey74]. It has constructs for type abstraction and application, and each bound variable is labeled with its type.

The reason for using this as our target language is that it makes it easy to extract a type from any subterm. This greatly eases later stages of compilation, where certain optimisations depend on knowing a subterm's type. An alternative might be to annotate each subterm with its type, but our method has three advantages.

- It uses less space. Types are stored in type applications and with each bound variable, rather than at every subterm.
- It eases subsequent transformation. A standard and productive technique for compiling functional languages is to apply various transformations at intermediate phases [Pey87]. With annotations, each transformation must carefully preserve annotations on all subterms and add new annotations where required. With polymorphic lambda calculus, the usual transformation rules – e.g., β-reduction for type abstractions – preserve type information in a simple and efficient way.
- It provides greater generality. Our back end can deal not only with languages based on Hindley-Milner types (such as Haskell) but also languages based on the more general Girard-Reynolds types (such as Ponder).

The use of explicit polymorphism in our target language is one of the most innovative aspects of this work. Further, this technique is completely independent of type classes – it applies just as well to any language based on Hindley-Milner types.

1.3 Structure of the paper

This paper does not assume prior knowledge of type classes. However, the introduction given here is necessarily cursory; for further motivating examples, see the original paper by Wadler and Blott [WB89]. For a comparison of the Hindley-Milner and Girard-Reynolds systems, see the excellent summary by Reynolds [Rey85]. For a practicum on Hindley-Milner type inference, see the tutorials by Cardelli [Car87] or Hancock [Han87].

The remainder of this paper is organised as follows. Section 2 introduces type classes and our translation method. Section 3 describes the various notations used in presenting the inferences rules. The syntax of types, the source language, and the target language is given, and the various forms of environment used are discussed. Section 4 presents the inference rules. Rules are given for types, expressions, dictionaries, class declarations, instance declarations, and programs. Finally, Section 5 describes how these rules can be used directly in a monad-based implementation.

2 Type Classes

This section introduces type classes and defines the required terminology. Some simple examples based on equality and comparison operations are introduced. Some overloaded function definitions are given and we show how they translate. The examples used here will appear as running examples through the rest of the paper.

2.1 Classes and instances

A `class` declaration provides the names and type signatures of the class *operations*:

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

This declares that type `a` belongs to the class `Eq` if there is an operation `(==)` of type `a -> a -> Bool`. That is, `a` belongs to `Eq` if equality is defined for it.

An *instance* declaration provides a *method* that implements each class operation at a given type:

```
instance  Eq Int  where
  (==)  =  primEqInt
instance  Eq Char  where
  (==)  =  primEqChar
```

This declares that type `Int` belongs to class `Eq`, and that the implementation of equality on integers is given by `primEqInt`, which must have type `Int -> Int -> Bool`. Similarly for characters.

We can now write `2+2 == 4`, which returns `True`; or `'a' == 'b'`, which returns `False`. As usual, `x == y` abbreviates `(==) x y`. In our examples, we assume all numerals have type `Int`.

Functions that use equality may themselves be overloaded:

```
member = \ x ys -> not (null ys) &&
         (x == head ys || member x (tail ys))
```

This uses Haskell notation for lambda expressions: `\ x ys -> e` stands for λx. λys. e. In practice we would use pattern matching rather than `null`, `head`, and `tail`, but here we avoid pattern matching, since we give typing rules for expressions only. Extending to pattern matching is easy, but adds unnecessary complication.

The type system infers the most general possible signature for `member`:

```
member :: (Eq a) => a -> [a] -> Bool
```

The phrase `(Eq a)` is called a *context* of the type – it limits the types that `a` can range over to those belonging to class `Eq`. As usual, `[a]` denotes the type of lists with elements of type `a`. We can now inquire whether `(member 1 [2,3])` or `(member 'a' ['c','a','t'])`, but not whether `(member sin [cos,tan])`, since there is no instance of equality over functions. A similar effect is achieved in Standard ML by using equality type variables; type classes can be viewed as generalising this behaviour.

Instance declarations may themselves contain overloaded operations, if they are provided with a suitable context:

```
instance  (Eq a) => Eq [a]  where
  (==) = \ xs ys ->
             (null xs && null ys) ||
             ( not (null xs) && not (null ys) &&
               head xs == head ys &&
               tail xs == tail ys)
```

This declares that for every type `a` belonging to class `Eq`, the type `[a]` also belongs to class `Eq`, and gives an appropriate definition for equality over lists. Note that `head xs == head ys` uses equality at type `a`, while `tail xs == tail ys` recursively uses equality at type `[a]`. We can now ask whether `['c','a','t'] == ['d','o','g']`.

Every entry in a context pairs a class name with a type variable. Pairing a class name with a type is not allowed. For example, consider the definition:

```
palindrome xs  =  (xs == reverse xs)
```

The inferred signature is:

```
palindrome :: (Eq a) => [a] -> Bool
```

Note that the context is `(Eq a)`, not `(Eq [a])`.

2.2 Superclasses

A class declaration may include a context that specifies one or more *superclasses*:

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

This declares that type `a` belongs to the class `Ord` if there are operations `(<)` and `(<=)` of the appropriate type, and if `a` belongs to class `Eq`. Thus, if `(<)` is defined on some type, then `(==)` must be defined on that type as well. We say that `Eq` is a superclass of `Ord`.

The superclass hierarchy must form a directed acyclic graph. An instance declaration is valid for a class only if there are also instance declarations for all its superclasses. For example

```
instance  Ord Int  where
  (<)  =  primLtInt
  (<=) =  primLeInt
```

is valid, since `Eq Int` is already a declared instance.

Superclasses allow simpler signatures to be inferred. Consider the following definition, which uses both `(==)` and `(<)`:

```
search = \ x ys ->
     not (null ys) &&
     ( x == head ys || ( x < head ys &&
                         search x (tail ys))
```

The inferred signature is:

```
search :: (Ord a) => a -> [a] -> Bool
```

Without superclasses, the inferred signature would have had the context `(Eq a, Ord a)`.

2.3 Translation

The inference rules specify a translation of source programs into target programs where the overloading is made explicit.

Each instance declaration generates an appropriate corresponding *dictionary* declaration. The dictionary for a class contains dictionaries for all the superclasses, and methods for all the operators. Corresponding to the `Eq Int` and `Ord Int` instances, we have the dictionaries:

```
dictEqInt   = ⟨primEqInt⟩
dictOrdInt  = ⟨dictEqInt, primLtInt, primLeInt⟩
```

Here $\langle e_1, \ldots, e_n\rangle$ builds a dictionary. The dictionary for `Ord` contains a dictionary for its superclass `Eq` and methods for `(<)` and `(<=)`.

For each operation in a class, there is a *selector* to extract the appropriate method from the corresponding dictionary. For each superclass, there is also a selector to extract the superclass dictionary from the subclass dictionary. Corresponding to the `Eq` and `Ord` classes, we have the selectors:

```
(==)         = \ ((),==)                -> ==
getEqFromOrd = \ ((dictEq),(<,<=))      ->
                     dictEq
(<)          = \ ((dictEq),(<,<=))      -> <
(<=)         = \ ((dictEq),(<,<=))      -> <=
```

Each overloaded function has extra parameters corresponding to the required dictionaries. Here is the translation of `search`:

```
search = \ dOrd x ys ->
        not (null ys) &&
        ( (==) (getEqFromOrd dOrd) x (head ys) ||
          ( (<) dOrd x (head ys) &&
            search dOrd x (tail ys)))
```

Each call of an overloaded function supplies the appropriate parameters. Thus the term `(search 1 [2,3])` translates to `(search dictOrdInt 1 [2,3])`.

If an instance declaration has a context, then its translation has parameters corresponding to the required dictionaries. Here is the translation for the instance `(Eq a) => Eq [a]`:

```
dictEqList = \ dEq ->
  ⟨\ xs ys ->
     ( null xs && null ys ) ||
     ( not (null xs) && not (null ys) &&
       (==) dEq (head xs) (head ys)   &&
       (==) (dictEqList dEq) (tail xs) (tail ys))⟩
```

When given a dictionary for `Eq a` this yields a dictionary for `Eq [a]`. To get a dictionary for equality on list of integers, one writes `dictEqList dictEqInt`.

The actual target language used differs from the above in that it contains extra constructs for explicit polymorphism. See Section 3.2 for examples.

3 Notation

This section introduces the syntax of types, the source language, the target language, and the various environments that appear in the type inference rules.

3.1 Type syntax

Figure 1 gives the syntax of types. Types come in three flavours: simple, overloaded, and polymorphic.

Recall from the previous section the type signature for `search`,

`(Ord a) => a -> [a] -> Bool`,

which we now write in the form

$$\forall\alpha.\langle \mathbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \mathbf{List}\ \alpha \to \mathrm{Bool}.$$

This is a polymorphic type of the form $\sigma = \forall\alpha.\theta \Rightarrow \tau$ built from a context $\theta = \langle \mathbf{Ord}\ \alpha\rangle$ and a simple type $\tau = \alpha \to \mathbf{List}\ \alpha \to \mathrm{Bool}$. Here **Ord** is a class name, **List** is a type

$$
\begin{array}{llcll}
\text{Type variable} & \alpha & & & \\
\text{Type contructor} & \chi & & & \\
\text{Class name} & \kappa & & & \\
\text{Simple type} & \tau & \rightarrow & \alpha & \\
 & & | & \chi\ \tau_1\ \ldots\ \tau_k & (k \geq 0, k = \text{arity}(\chi)) \\
 & & | & \tau' \rightarrow \tau & \\
\text{Overloaded type} & \rho & \rightarrow & \langle \kappa_1\ \tau_1, \ldots, \kappa_m\ \tau_m \rangle \Rightarrow \tau & (m \geq 0) \\
\text{Polymorphic type} & \sigma & \rightarrow & \forall \alpha_1 \ldots \alpha_l . \theta \Rightarrow \tau & (l \geq 0) \\
\text{Context} & \theta & \rightarrow & \langle \kappa_1\ \alpha_1, \ldots, \kappa_m\ \alpha_m \rangle & (m \geq 0) \\
\text{Record Type} & \gamma & \rightarrow & \langle v_1 : \tau_1, \ldots, v_n : \tau_n \rangle & (n \geq 0)
\end{array}
$$

Figure 1: Syntax of types

constructor of arity 1, and **Bool** is a type constructor of arity 0.

The record type, γ, maps class operation names to their types, and appears in the source syntax for classes.

There is one subtlety. In an overloaded type ρ, entries between angle brackets may have the form $\kappa\ \tau$, whereas in a polymorphic type σ or a context θ entries are restricted to the form $\kappa\ \alpha$. The extra generality of overloaded types is required during the inference process.

3.2 Source and target syntax

Figure 2 gives the syntax of the source language. A program consists of a sequence of `class` and `instance` declarations, followed by an expression. The Haskell language also includes features such as type declarations and pattern matching, which have been omitted here for simplicity. The examples from the previous section fit the source syntax precisely.

Figure 3 gives the syntax of the target language. We write the nonterminals of translated programs in boldface: the translated form of *var* is **var** and of *exp* is **exp**. To indicate that some target language variables and expressions represent dictionaries, we also use **dvar** and **dexp**.

The target language uses explicit polymorphism. It gives the type of bound variables in function abstractions, and it includes constructs to build and select from dictionaries, and to perform type abstraction and application. A program consists of a set of bindings, which may be mutually recursive, followed by an expression.

Notice that no class types appear in the translation. Given an environment E as defined below, context and record types are converted into monotypes by the function *tran*, defined as:

$$
\begin{array}{l}
tran\ E\ (\theta)\ =\ (tran\ E\ (\kappa_1\ \alpha_1), \ldots, tran\ E\ (\kappa_n\ \alpha_n)) \\
\qquad\qquad \textbf{where}\ \ \theta\ =\ (\kappa_1\ \alpha_1, \ldots, \kappa_n\ \alpha_n)
\end{array}
$$

$$
\begin{array}{l}
tran\ E\ (\kappa\ \alpha)\ =\ (tran\ E\ (\theta), tran\ E\ (\gamma)) \\
\qquad\qquad \textbf{where}\ \ (CE\ of\ E)\ \kappa\ = \\
\qquad\qquad\qquad\quad \texttt{class}\ \theta\ \Rightarrow\ \kappa\ \alpha\ \texttt{where}\ \gamma
\end{array}
$$

$$
\begin{array}{l}
tran\ E\ (\gamma)\ =\ (\tau_1, \ldots, \tau_n) \\
\qquad\qquad \textbf{where}\ \ \gamma\ =\ (v_1 : \tau_1, \ldots, v_n : \tau_n)
\end{array}
$$

This allows us to remove class types from the translation entirely.

As an example, here is the translation of `search` from Section 2.3, amended to make all polymorphism explicit:

```
search =
  Λα. λ dOrd:tran E₀ (Ord α). λx:α. λys:[α].
  not (null α ys) &&
  ( (==) α (getOrdFromEq α dOrd) x
           (head α ys) ||
  ( (<) α dOrd x (head α ys) &&
    search α dOrd x (tail α ys)))
```

3.3 Environments

The inference rules use a number of different environments, which are summarised in Figure 4.

The environment contains sufficient information to verify that all type variables, type constructors, class names, and individual variables appearing in a type or expression are valid. Environments come in two flavours, map environments and compound environments.

A map environment associates names with information. We write *ENV name* = *info* to indicate that environment *ENV*

$program$	$\rightarrow$	$classdecls$; $instdecls$; exp	Programs
$classdecls$	$\rightarrow$	$classdecl_1 ; \ldots ; classdecl_n$	Class declaration ($n \geq 0$)
$instdecls$	$\rightarrow$	$instdecl_1 ; \ldots ; instdecl_n$	Instance declaration ($n \geq 0$)
$classdecl$	$\rightarrow$	`class` $\theta \Rightarrow \kappa\ \alpha$ `where` γ	Class declaration
$instdecl$	$\rightarrow$	`instance` $\theta \Rightarrow \kappa\ (\chi\ \alpha_1 \ldots \alpha_k)$ `where` $binds$	Instance declaration ($k \geq 0$)
$binds$	$\rightarrow$	$\langle var_1 = exp_1\ , \ldots ,\ var_n = exp_n \rangle$	($n \geq 0$)
exp	$\rightarrow$	var	Variable
	$\mid$	$\lambda\ var$. exp	Function abstraction
	$\mid$	$exp\ exp'$	Function application
	$\mid$	`let` $var = exp'$ `in` exp	Local definition

Figure 2: Syntax of source programs

maps name *name* to information *info*. If the information is not of interest, we just write *ENV name* to indicate that *name* is in the domain of *ENV*. The type of a map environment is written in the symbolic form $\{name : info\}$.

We have the following map environments.

- The type variable environment AE contains each type variable name α that may appear in a valid type. This is the one example of a degenerate map, where there is no information associated with a name. We write $AE\ \alpha$ to indicate that α is in AE.
- The type constructor environment TE maps a type constructor χ to its arity k.
- The type class environment CE maps a class name κ to a class declaration, which contains all of the required type information.
- The instance environment IE maps a dictionary variable **dvar** to its corresponding type. The type indicates that **dvar** is a polymorphic function that expects one dictionary for each entry in θ, and returns a dictionary for $\kappa\ \tau$.
- The local instance environment LIE is similar, except the associated type is more restricted. Here the type indicates that **dvar** is a dictionary for $\kappa\ \tau$.
- The variable environment IE maps a variable **var** to its associated polymorphic type σ.
- The local variable environment LIE is similar, except the associated type is a simple type τ.

Environments corresponding to the examples in Section 2 are shown in Figure 5.

A compound environment consists of a tuple of other environments. We have the following compound environments.

- Most judgements use an environment E consisting of a type variable, a type constructor, a type class, an instance, a local instance, and a variable environment.
- Top level rules such as those for class and instance declarations use an initial version PE of the environment E. This contains an empty AE and LIE.
- The judgements for class declarations produces a declaration environment DE consisting of a type class, an instance, and a variable environment.

Again, these are summarised in Figure 4.

We write $VE\ of\ E$ to extract the type environment VE from the compound environment E, and similarly for other components of compound environments.

The operations $\oplus$ and $\vec{\oplus}$ combine environments. The former checks that the domains of its arguments are distinct, while the latter "shadows" its left argument with its right:

$$(ENV_1 \oplus ENV_2)\ var = \begin{cases} ENV_1\ var \text{ if } var \in dom(ENV_1) \text{ and } var \notin dom(ENV_2) \\ ENV_2\ var \text{ if } var \in dom(ENV_2) \text{ and } var \notin dom(ENV_1), \end{cases}$$

$$(ENV_1 \vec{\oplus} ENV_2)\ var = \begin{cases} ENV_1\ var & \text{if } var \in dom(ENV_1) \text{ and } var \notin dom(ENV_2) \\ ENV_2\ var & \text{if } var \in dom(ENV_2). \end{cases}$$

For brevity, we write $E_1 \oplus E_2$ instead of a tuple of the sums of the components of E_1 and E_2; and we write $E \oplus VE$ to combine VE into the appropriate component of E; and

program	$\rightarrow$	`letrec` **bindset** `in` **exp**	Program
bindset	$\rightarrow$	$\mathbf{var}_1 \; = \; \mathbf{exp}_1; \ldots ; \mathbf{var}_n \; = \; \mathbf{exp}_n$	Binding set ($n \geq 0$)
exp	$\rightarrow$	**var**	Variable
	$\mid$	λ **pat**. **exp**	Function abstraction
	$\mid$	**exp exp**$'$	Function application
	$\mid$	`let` **var** $=$ **exp**$'$ `in` **exp**	Local definition
	$\mid$	$\langle \mathbf{exp}_1, \ldots, \mathbf{exp}_n \rangle$	Dictionary formation ($n \geq 0$)
	$\mid$	$\Lambda \alpha_1 \ldots \alpha_n$. **exp**	Type abstraction ($n \geq 1$)
	$\mid$	**exp** $\tau_1 \ldots \tau_n$	Type application ($n \geq 1$)
pat	$\rightarrow$	**var** : τ	
	$\mid$	$(\mathbf{pat}_1, \ldots, \mathbf{pat}_n)$	($n \geq 0$)

Figure 3: Syntax of target programs

Environment	Notation	Type
Type variable environment	AE	$\{\alpha\}$
Type constructor environment	TE	$\{\chi : k\}$
Type class environment	CE	$\{\kappa : \texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ \gamma\}$
Instance environment	IE	$\{\mathbf{dvar} : \forall \alpha_1 \ldots \alpha_k .\ \theta \Rightarrow \kappa\ \tau\}$
Local instance environment	LIE	$\{\mathbf{dvar} : \kappa\ \tau\}$
Variable environment	VE	$\{\mathbf{var} : \sigma\}$
Environment	E	$(AE, TE, CE, IE, LIE, VE)$
Top level environment	PE	$(\{\}, TE, CE, IE, \{\}, VE)$
Declaration environment	DE	(CE, IE, VE)

Figure 4: Environments

similarly for other environments. Sometimes we specify the contents of an environment explicitly and write $\oplus_{ENV}$.

There are three implicit side conditions associated with environments:

1. Variables may not be declared twice in the same scope. If $E_1 \oplus E_2$ appears in a rule, then the side condition $dom(E_1) \cap dom(E_2) = \emptyset$ is implied.
2. Every variable must appear in the environment. If $E\ var$ appears in a rule, then the side condition $var \in dom(E)$ is implied.
3. At most one instance can be declared for a given class and given type constructor. If $IE_1 \oplus IE_2$ appears in a rule, then the side condition
$$\begin{array}{l} \forall\ \kappa_1\ (\chi_1\ \alpha_1 \ldots \alpha_m)\ \in\ IE_1. \\ \forall\ \kappa_2\ (\chi_2\ \alpha_1 \ldots \alpha_n)\ \in\ IE_2. \\ \quad \kappa_1\ \neq\ \kappa_2\ \vee\ \chi_1\ \neq\ \chi_2 \end{array}$$
is implied.

In some rules, types in the source syntax constrain the environments generated from them. This is stated explicitly by the *determines* relation, defined as:

$$\tau \text{ determines } AE \iff ftv(\tau) \; = \; AE$$

$$\theta \text{ determines } LIE \iff \theta \; = \; ran(LIE)$$

4 Rules

This section gives the inference rules for the various constructs in the source language. We consider in turn types, expressions, dictionaries, class declarations, instance declarations, and full programs.

4.1 Types

The rules for types are shown in Figure 6. The three judgement forms defined are summarised in the upper left corner.

$$\begin{aligned}
TE_0 &= \{\ \texttt{Int}: 0,\ \texttt{Bool}: 0,\ \texttt{List}: 1\ \} \\
CE_0 &= \{\ \texttt{Eq} : \{\texttt{class Eq}\ \alpha\ \textbf{where}\ \langle \texttt{(==)}: \alpha \to \alpha \to \textbf{Bool}\rangle\ \}, \\
&\qquad \texttt{Ord} : \{\texttt{class}\ \langle \texttt{Eq}\ \alpha\rangle \Rightarrow\ \texttt{Ord}\ \alpha \\
&\qquad\qquad \textbf{where}\ \langle \texttt{(<)}: \alpha \to \alpha \to \textbf{Bool}, \texttt{(<=)}: \alpha \to \alpha \to \textbf{Bool}\rangle\ \}\ \} \\
IE_0 &= \{\ \texttt{getEqFromOrd} : \forall\alpha.\ \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \textbf{Eq}\ \alpha, \\
&\qquad \texttt{dictEqInt} : \textbf{Eq Int}, \\
&\qquad \texttt{dictEqList} : \forall\alpha.\ \langle \textbf{Eq}\ \alpha\rangle \Rightarrow \textbf{Eq}\ (\textbf{List}\ \alpha), \\
&\qquad \texttt{dictOrdInt} : \textbf{Ord Int}\ \} \\
VE_0 &= \{\ \texttt{(==)}: \forall\alpha.\ \langle \textbf{Eq}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}, \\
&\qquad \texttt{(<)}: \forall\alpha.\ \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}, \\
&\qquad \texttt{(<=)}: \forall\alpha.\ \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}\ \} \\
E_0 &= (\{\}, TE_0, CE_0, IE_0, \{\}, VE_0)
\end{aligned}$$

Figure 5: Initial environments

A judgement of the form

$$E \overset{\text{type}}{\vdash} \tau$$

holds if in environment E the simple type τ is valid. In particular, all type variables in τ must appear in $AE\ of\ E$ (as checked by rule TYPE-VAR), and all type constructors in τ must appear in $TE\ of\ E$ with the appropriate arity (as checked by rule TYPE-CON). The other judgements act similarly for overloaded types and polymorphic types.

Here are some steps involved in validating the type $\forall\alpha.\ \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}$. Let $AE = \{\alpha\}$ and let E_0 be as in Figure 5. Then the following are valid judgements:

(1) $E_0 \oplus AE \overset{\text{type}}{\vdash} \alpha \to \alpha \to \textbf{Bool}$,

(2) $E_0 \oplus AE \overset{\text{over-type}}{\vdash} \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}$,

(3) $E_0 \overset{\text{poly-type}}{\vdash} \forall\alpha.\ \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}$.

Judgement (1) yields (2) via TYPE-PRED, and judgement (2) yields (3) via TYPE-GEN.

The type inference rules are designed to ensure that all types that arise are valid, given that all types in the initial environment are valid. In particular, if all types appearing in the CE, IE, LIE, and VE components of E are valid with respect to E, this property will be preserved throughout the application of the rules.

4.2 Expressions

The rules for expressions are shown in Figure 7. A judgement of the form

$$E \overset{\text{exp}}{\vdash} exp : \tau \rightsquigarrow \textbf{exp}$$

holds if in environment E the expression exp has simple type τ and yields the translation **exp**. The other two judgements act similarly for overloaded and polymorphic types.

The rules are very similar to those for the Hindley-Milner system. The rule TAUT handles variables, the rule LET handle let binding, the rules ABS and COMB introduce and eliminate function types, and the rules GEN and SPEC introduce and eliminate type quantifiers. The new rules PRED and REL introduce and eliminate contexts. Just as the rule GEN shrinks the type variable environment AE, the rule PRED shrinks the local instance environment LIE.

Here are some steps involved in typing the phrase `\ x y -> x < y`. Let E_0 be as in Figure 5, and let $AE = \{\alpha\}$, $LIE = \{\texttt{dOrd} : \textbf{Ord}\ \alpha\}$, and $VE = \{\texttt{x} : \alpha, \texttt{y} : \alpha\}$.

$$\boxed{E \overset{\text{exp}}{\vdash} exp : \tau \leadsto \mathbf{exp}}$$

$$\boxed{E \overset{\text{over-exp}}{\vdash} exp : \rho \leadsto \mathbf{exp}}$$

$$\boxed{E \overset{\text{poly-exp}}{\vdash} exp : \sigma \leadsto \mathbf{exp}}$$

$$\text{TAUT}\quad \frac{(VE\ of\ E)\ \mathbf{var} = \sigma}{E \overset{\text{poly-exp}}{\vdash} var : \sigma \leadsto \mathbf{var}}$$

$$\text{SPEC}\quad \frac{\begin{array}{ll} E \overset{\text{poly-exp}}{\vdash} var : \forall\alpha_1 \ldots \alpha_k.\theta \Rightarrow \tau \leadsto \mathbf{var} & \\ E \overset{\text{type}}{\vdash} \tau_i & (1 \le i \le k) \end{array}}{E \overset{\text{over-exp}}{\vdash} var : (\theta \Rightarrow \tau)[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k] \leadsto \mathbf{var}\ \tau_1 \ldots \tau_k}$$

$$\text{REL}\quad \frac{\begin{array}{l} E \overset{\text{over-exp}}{\vdash} var : \theta \Rightarrow \tau \leadsto \mathbf{exp} \\ E \overset{\text{dicts}}{\vdash} \theta \leadsto \mathbf{dexps} \end{array}}{E \overset{\text{exp}}{\vdash} var : \tau \leadsto \mathbf{exp\ dexps}}$$

$$\text{ABS}\quad \frac{E \overset{\rightarrow}{\oplus}_{VE} \{\mathbf{var} : \tau'\} \overset{\text{exp}}{\vdash} exp : \tau \leadsto \mathbf{exp}}{E \overset{\text{exp}}{\vdash} \lambda var.\ exp : \tau' \rightarrow \tau \leadsto \lambda\mathbf{var}\!: \tau'.\ \mathbf{exp}}$$

$$\text{COMB}\quad \frac{\begin{array}{l} E \overset{\text{exp}}{\vdash} exp : \tau' \rightarrow \tau \leadsto \mathbf{exp} \\ E \overset{\text{exp}}{\vdash} exp' : \tau' \leadsto \mathbf{exp}' \end{array}}{E \overset{\text{exp}}{\vdash} (exp\ exp') : \tau \leadsto (\mathbf{exp\ exp}')}$$

$$\text{PRED}\quad \frac{\begin{array}{ll} E \oplus LIE \overset{\text{dicts}}{\vdash} \theta \leadsto \mathbf{dpat} & \theta \text{ determines } LIE \\ E \oplus LIE \overset{\text{exp}}{\vdash} exp : \tau \leadsto \mathbf{exp} & \end{array}}{E \overset{\text{over-exp}}{\vdash} exp : \theta \Rightarrow \tau \leadsto \lambda\ \mathbf{dpat} : tran\ E\ (\theta)\ .\ \mathbf{exp}}$$

$$\text{GEN}\quad \frac{E \oplus_{AE} \{\alpha_1, \ldots, \alpha_k\} \overset{\text{over-exp}}{\vdash} exp : \theta \Rightarrow \tau \leadsto \mathbf{exp}}{E \overset{\text{poly-exp}}{\vdash} exp : \forall\alpha_1 \ldots \alpha_k.\ \theta \Rightarrow \tau \leadsto \Lambda\alpha_1 \ldots \alpha_k\ .\ \mathbf{exp}}$$

$$\text{LET}\quad \frac{\begin{array}{l} E \overset{\text{poly-exp}}{\vdash} exp' : \sigma \leadsto \mathbf{exp}' \\ E \overset{\rightarrow}{\oplus}_{VE} \{\mathbf{var} : \sigma\} \overset{\text{exp}}{\vdash} exp : \tau \leadsto \mathbf{exp} \end{array}}{E \overset{\text{exp}}{\vdash} \texttt{let}\ var\ =\ exp'\ \texttt{in}\ exp : \tau \leadsto \texttt{let}\ \mathbf{var}\ =\ \mathbf{exp}'\ \texttt{in}\ \mathbf{exp}}$$

Figure 7: Rules for expressions

$$\boxed{E \overset{\text{type}}{\vdash} \tau}$$

$$\boxed{E \overset{\text{over-type}}{\vdash} \theta \Rightarrow \tau}$$

$$\boxed{E \overset{\text{poly-type}}{\vdash} \forall \alpha_1, \ldots, \alpha_n.\ \theta \Rightarrow \tau}$$

$$\text{TYPE-VAR}\quad \frac{(AE\ of\ E)\ \alpha}{E \overset{\text{type}}{\vdash} \alpha}$$

$$\text{TYPE-CON}\quad \frac{\begin{array}{ll} (TE\ of\ E)\ \chi\ =\ k & \\ E \overset{\text{type}}{\vdash} \tau_i & (1 \le i \le k) \end{array}}{E \overset{\text{type}}{\vdash} \chi\ \tau_1 \ldots \tau_k}$$

$$\text{TYPE-PRED}\quad \frac{\begin{array}{ll} (CE\ of\ E)\ \kappa_i & (1 \le i \le m) \\ (AE\ of\ E)\ \alpha_i & (1 \le i \le m) \\ E \overset{\text{type}}{\vdash} \tau & \end{array}}{E \overset{\text{over-type}}{\vdash} \langle \kappa_1\ \alpha_1, \ldots, \kappa_m\ \alpha_m \rangle \Rightarrow \tau}$$

$$\text{TYPE-GEN}\quad \frac{E \oplus_{AE} \{\alpha_1, \ldots, \alpha_k\} \overset{\text{over-type}}{\vdash} \theta \Rightarrow \tau}{E \overset{\text{poly-type}}{\vdash} \forall \alpha_1 \ldots \alpha_k.\ \theta \Rightarrow \tau}$$

Figure 6: Rules for types

Then the following are valid judgements:

$$\begin{array}{lll}
(1) & E_0 \oplus AE \oplus LIE \oplus VE \overset{\text{exp}}{\vdash} & \texttt{x < y} : \mathbf{Bool} \\
& & \leadsto \\
& & \texttt{(<)}\ \alpha\ \texttt{dOrd x y} \\
(2) & E_0 \oplus AE \oplus LIE \overset{\text{exp}}{\vdash} & \texttt{\textbackslash\ x y -> x < y} \\
& & : \\
& & \alpha \to \alpha \to \mathbf{Bool} \\
& & \leadsto \\
& & \lambda\texttt{x} : \alpha.\ \lambda\texttt{y} : \alpha.\ \texttt{(<)}\ \alpha\ \texttt{dOrd x y} \\
(3) & E_0 \oplus AE \overset{\text{over-exp}}{\vdash} & \texttt{\textbackslash\ x y -> x < y} \\
& & : \\
& & \langle \texttt{Ord}\ \alpha \rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool} \\
& & \leadsto \\
& & \lambda\texttt{dOrd} : tranE_0(\text{Ord}\ \alpha). \\
& & \lambda\texttt{x} : \alpha.\ \lambda\texttt{y} : \alpha.\ \texttt{(<)}\ \alpha\ \texttt{dOrd x y} \\
(4) & E_0 \overset{\text{poly-exp}}{\vdash} & \texttt{\textbackslash\ x y -> x < y} \\
& & : \\
& & \forall\alpha.\ \langle \texttt{Ord}\ \alpha \rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool} \\
& & \leadsto \\
& & \Lambda\alpha.\ \lambda\texttt{dOrd} : tranE_0(\text{Ord}\ \alpha). \\
& & \lambda\texttt{x} : \alpha.\ \lambda\texttt{y} : \alpha.\ \texttt{(<)}\ \alpha\ \texttt{dOrd x y}
\end{array}$$

Judgement (1) yields (2) via ABS, judgement (2) yields (3) via PRED, and judgment (3) yields (4) via GEN.

As is usual with such rules, one is required to use prescience to guess the right initial environments. For the SPEC and GEN rules, the method of transforming prescience into an algorithm is well known: one generates equations relating types during the inference process, then solves these equations via unification. For the PRED and REL rules, a similar method of generating equations can be derived.

4.3 Dictionaries

The inference rules for dictionaries are shown in Figure 8. A judgement of the form

$$E \overset{\text{dict}}{\vdash} \kappa\ \tau \leadsto \mathbf{dexp}$$

holds if in environment E there is an instance of class κ at type τ given by the dictionary **dexp**. The other two judgements act similarly for overloaded and polymorphic instances.

The two DICT-TAUT rules find instances in the IE and LIE component of the environment. The DICT-SPEC rule instantiates a polymorphic dictionary, by applying it to a type. Similarly, the DICT-REL rule instantiates an overloaded dictionary, by applying it to other dictionaries, themselves derived by recursive application of the dictionary judgement.

The rule that translates an entire context into a tuple of the corresponding dictionaries has the judgement form

$$E \overset{\text{dicts}}{\vdash} \theta \leadsto \mathbf{dexps}$$

Here is how to derive a dictionary for the instance of class **Eq** at type **Int**. Let E_0 be as in Figure 5. Then the following judgements hold:

$$\begin{array}{ll}
(1) & E_0 \overset{\text{poly-dict}}{\vdash} \forall\alpha.\ \langle \mathbf{Eq}\ \alpha \rangle \Rightarrow \mathbf{Eq}\ (\mathbf{List}\ \alpha) \leadsto \texttt{dictEqList} \\
(2) & E_0 \overset{\text{over-dict}}{\vdash} \langle \mathbf{Eq\ Int} \rangle \Rightarrow \mathbf{Eq}\ (\mathbf{List\ Int}) \leadsto \texttt{dictEqList}\ \mathbf{Int} \\
(3) & E_0 \overset{\text{dict}}{\vdash} \mathbf{Eq\ Int} \leadsto \texttt{dictEqInt} \\
(4) & E_0 \overset{\text{dict}}{\vdash} \mathbf{Eq}\ (\mathbf{List\ Int}) \leadsto \texttt{dictEqList}\ \mathbf{Int}\ \texttt{dictEqInt}
\end{array}$$

Judgement (1) holds via DICT-TAUT-IE, judgement (2) follows from (1) via DICT-SPEC, judgement (3) holds via DICT-TAUT-IE, and judgement (4) follows from (2) and (3) via DICT-REL.

Note that the dictionary rules correspond closely to the TAUT, SPEC, and REL rules for expressions.

$$\boxed{E \overset{\text{dict}}{\vdash} \kappa\ \tau \leadsto \mathbf{dexp}}$$

$$\boxed{E \overset{\text{over-dict}}{\vdash} \theta \Rightarrow \kappa\ \tau \leadsto \mathbf{dexp}}$$

$$\boxed{E \overset{\text{poly-dict}}{\vdash} \forall\alpha_1 \ldots \alpha_n.\ \theta \Rightarrow \kappa\ \tau \leadsto \mathbf{dexp}}$$

$$\boxed{E \overset{\text{dicts}}{\vdash} \theta \leadsto \mathbf{dexps}}$$

$$\text{DICT-TAUT-LIE}\quad \frac{(LIE\ of\ E)\ \mathbf{dvar} = \kappa\ \alpha}{E \overset{\text{dict}}{\vdash} \kappa\ \alpha \leadsto \mathbf{dvar}}$$

$$\text{DICT-TAUT-IE}\quad \frac{(IE\ of\ E)\ \mathbf{dvar} = \forall\alpha_1 \ldots \alpha_n.\ \theta \Rightarrow \kappa\ (\chi\ \alpha_1 \ldots \alpha_n)}{E \overset{\text{poly-dict}}{\vdash} \forall\alpha_1 \ldots \alpha_n.\ \theta \Rightarrow \kappa\ (\chi\ \alpha_1 \ldots \alpha_n) \leadsto \mathbf{dvar}}$$

$$\text{DICT-SPEC}\quad \frac{E \overset{\text{poly-dict}}{\vdash} \forall\alpha_1 \ldots \alpha_n.\theta \Rightarrow \kappa\ \tau \leadsto \mathbf{dexp}}{E \overset{\text{over-dict}}{\vdash} (\theta \Rightarrow \kappa\ \tau)[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \leadsto \mathbf{dexp}\ \tau_1 \ldots \tau_n}$$

$$\text{DICT-REL}\quad \frac{\begin{array}{l} E \overset{\text{over-dict}}{\vdash} \theta \Rightarrow \kappa\ \tau \leadsto \mathbf{dexp} \\ E \overset{\text{dicts}}{\vdash} \theta \leadsto \mathbf{dexps} \end{array}}{E \overset{\text{dict}}{\vdash} \kappa\ \tau \leadsto \mathbf{dexp\ dexps}}$$

$$\text{DICTS}\quad \frac{E \overset{\text{dict}}{\vdash} \kappa_i\ \tau_i \leadsto \mathbf{dexp}_i \qquad\qquad (1 \leq i \leq n)}{E \overset{\text{dicts}}{\vdash} \langle \kappa_1\ \tau_1, \ldots, \kappa_n\ \tau_n \rangle \leadsto \langle \mathbf{dexp}_1, \ldots, \mathbf{dexp}_n \rangle}$$

Figure 8: Rules for dictionaries

4.4 Class declarations

The rule for class declarations is given in Figure 9. Although the rule looks formidable, its workings are straightforward.

A judgement of the form

$$PE \overset{\text{classdecl}}{\vdash} classdecl : DE \leadsto \mathbf{bindset}$$

holds if in environment PE the class declaration *classdecl* is valid, generating new environment DE and yielding translation **bindset**. In the compound environment $DE = (CE, IE, VE)$, the class environment CE has one entry that describes the class itself, the instance environment IE has one entry for each superclass of the class (given the class dictionary, it selects the appropriate superclass dictionary) and the value environment VE has one entry for each operator of the class (given the class dictionary, it selects the appropriate method).

For example, the class declaration for `Ord` given in Section 2.2 yields the `Ord` component of CE_0, the `getEqFromOrd` component of IE_0, and the `(<)` and `(<=)` components of VE_0, as found in Figure 5. The binding set generated by the rule is as shown in Section 2.3.

4.5 Instance declarations

The rule for instance declarations is given in Figure 11. Again the rule looks formidable, and again its workings are straightforward.

A judgement of the form

$$PE \overset{\text{instdecl}}{\vdash} instdecl : IE \leadsto \mathbf{bindset}$$

holds if in environment PE the instance declaration *instdecl* is valid, generating new environment IE and yielding trans-

$$\boxed{E \overset{\text{classdecl}}{\vdash} classdecl : DE \leadsto \mathbf{bindset}}$$

$$\text{CLASS}\quad \frac{\begin{array}{ll} PE \oplus AE \overset{\text{type}}{\vdash} \alpha & \alpha \text{ determines } AE \\ PE \oplus AE \oplus LIE \overset{\text{dicts}}{\vdash} \theta \leadsto \mathbf{dpat} & \theta \text{ determines } LIE \\ PE \oplus AE \overset{\text{sigs}}{\vdash} \gamma \leadsto \mathbf{mpat} & \end{array}}{PE \overset{\text{classdecl}}{\vdash} \begin{array}{l} \texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ \gamma \\ : \\ (\{\kappa : \texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ \gamma\}, \\ \ \{\mathbf{dvar} : \Lambda\ \alpha.\ \langle \kappa\ \alpha \rangle \Rightarrow \kappa'\ \tau' \mid \mathbf{dvar} : \kappa'\ \tau' \in LIE\}, \\ \ \{\mathbf{var} : \Lambda\ \alpha.\ \langle \kappa\ \alpha \rangle \Rightarrow \tau \mid \mathbf{var} : \tau \in \gamma\}) \\ \leadsto \\ \ \{\mathbf{dvar} = \Lambda\ \alpha.\ \lambda\ (\mathbf{dpat}, \mathbf{mpat}) : (tran\ PE\ (\theta), tran\ PE\ (\gamma))\ .\ \mathbf{dvar} \\ \qquad\qquad \mid \mathbf{dvar} \in dom(LIE)\} \cup \\ \ \{\mathbf{var} = \Lambda\ \alpha.\ \lambda\ (\mathbf{dpat}, \mathbf{mpat}) : (tran\ PE\ (\theta), tran\ PE\ (\gamma))\ .\ \mathbf{var} \\ \qquad\qquad \mid \mathbf{var} \in dom(\gamma)\} \end{array}}$$

Figure 9: Rule for class declarations

$$\boxed{E \overset{\text{sigs}}{\vdash} sigs \leadsto \mathbf{sigs}}$$

$$\text{SIGS}\quad \frac{E \overset{\text{type}}{\vdash} \tau_i \qquad\qquad (1 \leq i \leq m)}{E \overset{\text{sigs}}{\vdash} \langle var_1 : \tau_1, \ldots, var_m : \tau_m \rangle \leadsto \langle var_1, \ldots, var_m \rangle}$$

Figure 10: Rule for class signatures

$$\boxed{E \overset{\text{instdecl}}{\vdash} instdecl : IE \leadsto \mathbf{bindset}}$$

$$\text{INST}\quad \frac{\begin{array}{ll} (CE\ of\ PE)\ \kappa = \texttt{class}\ \theta' \Rightarrow \kappa\ \alpha\ \texttt{where}\ \gamma' & \\ PE \oplus AE \overset{\text{type}}{\vdash} \tau & \tau \text{ determines } AE \\ PE \oplus AE \oplus LIE \overset{\text{dicts}}{\vdash} \theta \leadsto \mathbf{dpat} & \theta \text{ determines } LIE \\ PE \oplus AE \oplus LIE \overset{\text{dicts}}{\vdash} \theta'[\tau/\alpha] \leadsto \mathbf{dexp} & \\ PE \oplus AE \oplus LIE \overset{\text{binds}}{\vdash} binds : \gamma'[\tau/\alpha] \leadsto \mathbf{binds} & \end{array}}{PE \overset{\text{instdecl}}{\vdash} \begin{array}{l} \texttt{instance}\ \theta \Rightarrow \kappa\ \tau\ \texttt{where}\ binds \\ : \\ \{\mathbf{dvar} = \forall\ dom(AE).\ \theta \Rightarrow \kappa\ \tau\} \\ \leadsto \\ \mathbf{dvar} = \Lambda\ dom(AE).\ \lambda\ \mathbf{dpat} : tran\ PE\ (\theta).\ \langle \mathbf{dexp}, \mathbf{binds} \rangle \end{array}}$$

Figure 11: Rule for instance declarations

$$\boxed{E \overset{\text{binds}}{\vdash} binds : \gamma \leadsto \mathbf{binds}}$$

$$\text{BINDS}\quad \frac{E \overset{\text{exp}}{\vdash} exp_i : \mathbf{exp}_i \leadsto \tau_i \qquad\qquad (1 \leq i \leq m)}{\begin{array}{ll} E \overset{\text{binds}}{\vdash} & \langle var_1 = exp_1, \ldots, var_m = exp_m \rangle \\ & : \\ & \langle var_1 : \tau_1, \ldots, var_m : \tau_m \rangle \\ & \leadsto \\ & \langle \mathbf{var}_1 = \mathbf{exp}_1, \ldots, \mathbf{var}_m = \mathbf{exp}_m \rangle \end{array}}$$

Figure 12: Rule for instance bindings

lation **bindset**. The instance environment IE contains a single entry corresponding to the instance declaration, and the **bindset** contains a single binding. If the header of the instance declaration is $\theta \Rightarrow \kappa\ \tau$, then the corresponding instance is a function that expects one dictionary for each entry in θ, and returns a dictionary for the instance.

The first line looks up the superclasses and record type of the instance class. Line (2) sets AE to contain the type variables in τ. Line (3) sets LIE to contain the types in θ, the instance context, and builds the pattern for the dictionary parameters. Line (4) checks that the superclasses are satisfied by the LIE and builds the dictionaries for those superclasses. Finally, line (5) checks that the method definitions have precisely the types of the class operations instantiated by the instance type, and builds the method translations.

For example, the instance declarations for `Eq Int`, `(Eq a) => Eq [a]`, and `Ord Int` yield the `dictEqInt`, `dictEqList`, and `dictOrdInt` components of IE_0 as found in Figure 5, and the bindings generated by the rule are as shown in Section 2.3.

4.6 Programs

Figure 13 gives the rules for declaration sequences and programs.

The order of the class declarations is significant, because at the point of a class declaration all its superclasses must already be declared. (This guarantees that the superclass hierarchy forms a directed acyclic graph.) Further, all class declarations must come before all instance declarations.

Conversely, the order of the instance declarations is irrelevant, because all instance declarations may be mutually recursive. Mutual recursion of polymorphic functions doesn't cause the problems you might expect, because the instance declaration explicitly provides the needed type information.

These differences are reflected in the different forms of the CDECLS and IDECLS rules. That instance declarations are mutually recursive is indicated by line (2) of the PROG rule, where the same environment IE appears on the left and right of the *instdecl* rule.

In Haskell, the source text need not be so ordered. A preprocessing phase performs a dependency analysis and places the declarations in a suitable order.

5 Implementing the rules

Our major goal in writing the type rules was to provide a basis for an implementation.

In writing the rules and the implementation, we had the picture in Figure 2 in our minds. Each syntactic non-terminal in the grammar of the language (such as expressions, *exp*) corresponds one-to-one with a judgement form (such as $\overset{\text{exp}}{\vdash}$). The implementation was designed so that each judgement form in turn corresponds one-to-one with a function (such as `tcExpr`, the function which typechecks expressions).

The structure extends further. Each *production* in the grammar corresponds one-to-one with a type *rule*, and that in turn corresponds one-to-one with a case in the definition of the appropriate function. For example, the production

```
exp -> exp_1 exp_2
```

corresponds to the rule `COMB`, and to a case in the definition of `tcExpr` which begins

```
tcExpr (App e1 e2) ... = ...
```

Lastly, the type of the function that corresponds to a judgement form is systematically derived from the signature of that judgement form. For example, the signature of $\overset{\text{exp}}{\vdash}$ is

$$\boxed{E \overset{\text{exp}}{\vdash} exp : \tau \leadsto \mathbf{exp}}$$

$$\boxed{PE \overset{\text{classdecls}}{\vdash} classdecls : DE \rightsquigarrow \mathbf{bindset}}$$

$$\text{CDECLS}\quad \frac{PE \oplus DE_1 \oplus \ldots \oplus DE_{i-1} \overset{\text{classdecl}}{\vdash} classdecl_i : DE_i \rightsquigarrow \mathbf{bindset}_i \quad (1 \le i \le n)}{\begin{array}{ll} PE \overset{\text{classdecls}}{\vdash} & classdecl_1 \; ; \; \ldots \; ; \; classdecl_n : DE_1 \oplus \ldots \oplus DE_n \\ & \rightsquigarrow \\ & \mathbf{bindset}_1 ; \; \ldots \; ; \mathbf{bindset}_n \end{array}}$$

$$\boxed{PE \overset{\text{instdecls}}{\vdash} instdecls : IE \rightsquigarrow \mathbf{bindset}}$$

$$\text{IDECLS}\quad \frac{PE \overset{\text{instdecl}}{\vdash} instdecl_i : IE_i \rightsquigarrow \mathbf{bindset}_i \quad (1 \le i \le n)}{\begin{array}{ll} PE \overset{\text{instdecls}}{\vdash} & instdecl_1 \; ; \; \ldots \; ; \; instdecl_n : (IE \text{ of } PE) \oplus IE_1 \oplus \ldots \oplus IE_n \\ & \rightsquigarrow \\ & \mathbf{bindset}_1 ; \; \ldots \; ; \mathbf{bindset}_n \end{array}}$$

$$\boxed{PE \overset{\text{program}}{\vdash} program : (DE, \tau) \rightsquigarrow \mathbf{exp}}$$

$$\text{PROG}\quad \frac{\begin{array}{ll} (1) & PE \overset{\text{classdecls}}{\vdash} classdecls : DE \rightsquigarrow \mathbf{bindset}_C \\ (2) & PE \oplus DE \oplus IE \overset{\text{instdecls}}{\vdash} instdecls : IE \rightsquigarrow \mathbf{bindset}_I \\ (3) & PE \oplus DE \oplus IE \overset{\text{exp}}{\vdash} exp : \tau \rightsquigarrow \mathbf{exp} \end{array}}{\begin{array}{ll} PE \overset{\text{program}}{\vdash} & classdecls \; ; \; instdecls \; ; \; exp : (DE \oplus IE, \tau) \\ & \rightsquigarrow \\ & \texttt{letrec } \mathbf{bindset}_C ; \; \mathbf{bindset}_I \texttt{ in } \mathbf{exp} \end{array}}$$

Figure 13: Rules for declaration sequences and programs

and the type of `tcExpr` is

```
tcExpr :: Env -> Expr -> TcM (Expr, Type)
```

That is, `tcExpr` takes an environment and an expression to be typechecked, and returns the translated expression, along with its inferred type.

Notice that the monad `TcM` appears in the result type of `tcExpr`. One might have thought that `tcExpr` would have type `Env -> Expr -> (Expr, Type)`, a function from expressions to types, but quite a lot of "plumbing" is required behind the scenes. Firstly, if there is a type error in the program, the type-inference process can *fail* when type-checking `fun` or `arg`, or during the unification step. Secondly, there must be a *unique-name supply* from which to manufacture a new type variable. Thirdly, there must be some way of collecting and displaying *error messages*. Fourthly, the unification process works by incrementally augmenting a *substitution* mapping type variables to types. All of these would usually be handled in an imperative language by side effects, or by some sort of exception-handling mechanism. We handle them all using a monad, `TcM`, which is defined as

```
type TcM a =   Uniq -> Subst ->
                        Maybe (a, Uniq, Subst)

data Maybe a = Nothing | Just a
```

A tutorial on monads as they are used in the compiler appears in the appendix.

Programming using monads has a number of benefits. Firstly, *the plumbing is implicit*, which makes the program much easier to read and write.

Secondly, *because the plumbing is all encapsulated in one place, it is very easy to modify*. For example, well over a year after the type checker was working we added an error recovery mechanism, so that a single type error would not cause the entire type-checker to halt. It is now possible to recover from the error, and gather several accurate type error messages in one run of the compiler. This was achieved in a single afternoon's work, modifying very localised parts of the type checker.

5.1 The typechecker implementation

The only difficulty in translating rules into code is deciding which components of the judgement signature should go *into* the function as its arguments, and what should come *out* of the function as its result. This is not an easy question, because type rules are deliberately relational in style, and therefore ambiguous about the direction of data flow. In translating them into functions, these data flow directions must be made explicit.

The major change needed to implement a functional version of these rules requires computing the *LIE* directly. This is done in each of the expression rules, which combine *LIE*s calculated for the leaves of the abstract syntax tree. The dictionary rules are explicitly passed an *LIE*, rather than building it up relationally in the incoming environment.

5.1.1 Application

Let `Expr` be the data-type for abstract syntax, and `SimpleType`, `OverType` and `PolyType` represent the types τ, θ, σ. The types of `exp`, `over_exp` and `dicts` are

```
exp :: E -> Expr -> TcM (Expr, SimpleType, LIE)
over_exp :: E -> Expr -> TcM (Expr, OverType, LIE)
dicts :: E -> LIE  -> TcM (Expr, Theta)
```

To start off with, we present the (slightly simplified) code from the compiler for type-checking an application:

```
tcExpr (Ap fun arg)
 = tcExpr fun `thenTcM` (\(fun',fun_ty,lie_fun) ->
   tcExpr arg `thenTcM` (\(arg',arg_ty,lie_arg) ->
   newTyVar   `thenTcM` (\ res_ty ->

   unify fun_ty (arg_ty `arrow` res_ty)
       `thenTcM` (\ () ->

   returnTcM (Ap fun' arg', res_ty,
              plusLIE lie_fun lie_arg)
   ))))
```

At an informal level this code should be legible even by readers with no experience of type inference. It can be read like this: "To typecheck an application `Ap fun arg`, first typecheck `fun`, inferring the type `fun_ty`, then typecheck `arg`, inferring the type `arg_ty`. Now invent a fresh type variable `res_ty`, and unify `fun_ty` with `arg_ty -> res_ty`. Finally, return `res_ty` as the type of the application."

5.1.2 `REL` and `PRED`

Next, we'll show how the two new expression rules, *REL* and *PRED* can be implemented using monads, and then sketch the rest of the implementation.

Here is the implementation of the *REL* rule. A new *LIE* is created from new variable names given by the monad state.

```
exp e (Var v)
 = over_exp e (Var v) `thenTcM`
             (\ (expr, Arrow theta simple, _) ->

   mkNewLIE theta     `thenTcM` (\ lie         ->
   dicts e lie        `thenTcM` (\ (dexprs, _) ->

   returnTcM (mkDictApp expr dexprs, simple,
              lie))))
```

The implementation of the *PRED* rule is more complex. Notice that the *LIE* inferred for an expression may constrain free type variables in the type environment; the corresponding dictionaries should not appear in the outgoing *LIE*. Thus `lie_expr` must be split into two pieces by `splitLIE`.

```
over_exp e expr
 = exp e expr `thenTcM` (\ (expr', simple,
                             lie_expr        ) ->

   (let (lie_theta, lie_enclosing) =
                     splitLIE simple e lie_expr
    in

    dicts e lie_theta `thenTcM` (\(dpat,theta) ->
    returnTcM (mkLambda dpat (tran e theta) expr',
               Arrow theta simple,
               lie_enclosing))))
```

The rest of the expression rules and the dictionary rules provide a straightforward implementation. The circularity in the top-level rules, which requires that instance definitions be based on top-level bindings and which in turn depend on instance definitions, can be implemented by using two passes.

6 Conclusions

This paper makes two contributions. First, it presents a minimal, readable set of inference rules to handle type classes in Haskell, derived from the full static semantics [PW91]. An important feature of this style of presentation is that it scales up well to a description of the entire Haskell language, as we have found in practice.

Second, the paper has shown how these rules can be directly implemented using monads. This style has been applied to the full static semantics in order to construct the type checker used in the Glasgow Haskell compiler, as well as virtually all other passes in the compiler. It has undoubtedly saved us from initially making countless bookkeeping errors, and continues to pay off as we maintain our code and train students to work with it.

Together, these form a useful bridge between more traditional theoretical papers on type theory and papers describing particular compiler implementations.

References

[Aug93] L. Augustsson, Implementing Haskell Overloading. In *Functional Programming and Computer Architecture*, Copenhagen, June 1993.

[Blo91] S. Blott, *Type classes.* Ph.D. Thesis, Glasgow University, 1991.

[Car87] L. Cardelli, Basic Polymorphic Typechecking. *Science of Computer Programming*, Vol. 8, 1987, pp. 147–172.

[CHO92] K. Chen, P. Hudak, and M. Odersky, Parametric Type Classes. In *Lisp and Functional Programming*, 1992, pp. 170–181.

[CW90] G. V. Comack and A. K. Wright, Type dependent parameter inference. In *Programming Language Design and Implementation*, White Plains, New York, June 1990, ACM Press.

[DM82] L. Damas and R. Milner, Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, Albuquerque, N.M., January 1982.

[Gir72] J.-Y. Girard, *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure.* Ph.D. thesis, Université Paris VII, 1972.

[HaBl89] K. Hammond and S. Blott, Implementing Haskell Type Classes. In *1989 Glasgow Workshop on Functional Programming*, Fraserburgh, Scotland, September 1989, Springer-Verlag WICS, pp. 266–286.

[Han87] P. Hancock, Chapters 8 and 9 In S. L. Peyton Jones, *The implementation of functional programming languages*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1987.

[Hin69] R. Hindley, The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc. 146*, December 1969, pp. 29–60, .

[HPW92] P. Hudak, S. L. Peyton Jones, and P. Wadler, editors, Report on the Programming Language Haskell, Version 1.2. *SIGPLAN Notices*, May 1992.

[Hue 90] Gerard Huet, editor, Logical Foundations of Functional Programming, Addison Wesley, 1990. See Part II, Polymorphic Lambda Calculus, especially the introduction by Reynolds.

[Jon92a] M. P. Jones, A theory of qualified types. In *European Symposium on Programming*, Rennes, February 1992, LNCS 582, Springer-Verlag.

[Jon92b] M. P. Jones, Efficient Implementation of Type Class Overloading. Dept. of Computing Science, Oxford University.

[Jon93] M. P. Jones, A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. In *Functional Programming and Computer Architecture*, Copenhagen, June 1993, pp. 52–61.

[Kae88] S. Kaes, Parametric polymorphism. In *European Symposium on Programming*, Nancy, France, March 1988, LNCS 300, Springer-Verlag.

[Läu92] Polymorphic Type Inference and Abstract Data Types. K. Läufer, Ph.D. Thesis, New York University, 1992.

[Läu93] An Extension of Haskell with First-Class Abstract Types. K. Läufer, Technical Report, Loyola University of Chicago, 1993.

[MTH90] R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML*, MIT Press, Cambridge, Massachusetts, 1990.

[MT91] R. Milner and M. Tofte, *Commentary on Standard ML*, MIT Press, Cambridge, Massachusetts, 1991.

[Mil78] R. Milner, A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 1978, pp. 348–375.

[NP93] T. Nipkow and C. Prehofer, Type Checking Type Classes. In *ACM Symposium on Principles of Programming Languages*, January 1993, pp. 409–418.

[NS91] T. Nipkow and G. Snelting, Type Classes and Overloading Resolution via Order-Sorted Unification. In *Functional Programming Languages and Computer Architecture*, Boston, August 1991, LNCS 523, Springer-Verlag.

[OdLä91] M. Odersky and K. Läufer, Type classes are signatures of abstract types. Technical Report, IBM TJ Watson Research Centre, May 1991.

[Pey87] S. L. Peyton Jones, *The implementation of functional programming languages*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1987.

[PW91] S. L. Peyton Jones and P. Wadler, A static semantics for Haskell. Department of Computing Science, Glasgow University, May 1991.

[Rey74] J. C. Reynolds, Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, LNCS 19, Springer-Verlag.

[Rey85] J. C. Reynolds, Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.

[Rou90] F. Rouaix, Safe run-time overloading. In *ACM Symposium on Principles of Programming Languages*, San Francisco, January 1990, ACM Press.

[Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985, LNCS 201, Springer-Verlag, pp. 1–16.

[VS91] D. M. Volpano and G. S. Smith, On the complexity of ML typability with overloading. In *Functional Programming Languages and Computer Architecture*, Boston, August 1991, LNCS 523, Springer-Verlag.

[Wad92] P. L. Wadler, The essence of functional programming. In *ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[WB89] P. L. Wadler and S. Blott, How to make ad-hoc polymorphism less ad hoc, In *ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989, pp. 60–76.

A A tutorial on compiler monads

A monad is an abstract data type with a set of operations, typically 'return', and 'then' [Wad92]. We'll refer to 'functions which use monads' as monad *functions*, as distinct from monad operations. If `f` has type `a -> b`, then the corresponding monad function `f'` has type `a -> M b`. These functions are given access to the monad state only by monad operations.

Here is a typical compiler monad which passes around unique variable names. The supply must be passed in and out of each monad operation. The name supply is seen only by the monad operation `getName` which removes one name and passes the depleted supply back to the monad.

```
type NameSupply = [Int]
type M result   = NameSupply ->
                  (NameSupply, result)
```

```
getName :: M Name
getName (n:name_supply) = (name_supply, n)
```

The operations 'returnM' and 'thenM' do the plumbing; the simplest of these is 'returnM'.

```
returnM :: result -> M result
returnM result name_supply = (name_supply, result)
```

The most complex monad operation is 'thenM'. In general, its type shows that it takes a computation of type `a`, and a function (continuation) from a value of type `a` to a computation of type `b`, yielding a computation of type `b`. It works as follows:

1. perform the first computation to yield a value of type `a`;
2. apply the continuation to this to get a value of type `b`;
3. this value is combined with the effects of the two computations in an appropriate way, and this combination is a computation of type `b`.

For example, the implementation of 'thenM' for this particular monad is:

```
thenM :: M a -> (a -> M b) -> M b
thenM e k name_supply
 = case (e name_supply) of
    (next_name_supply, result) ->
         k result next_name_supply
```

The expression `e` receives the name supply, which it can access and update, returning the new supply. Its value is passed to the continuation, which gets the new name supply. Since nothing needs to be done to combine the value of the continuation with the monad state, the result of 'thenM' is just the application of the continuation. Another monad might look at the state returned by this application and combine it with the first state which was returned by evaluating `e`.

One particularly useful feature of monads is that they can also propagate errors. To do this, we alter `M` as follows:

```
type ErrorType         = String
data MaybeErr res err  = Succeeded res |
                         Failed err

type M result = NameSupply ->
                MaybeErr (NameSupply, result)
                         ErrorType
```

The corresponding monad operations are:

```
returnM :: result -> M result
returnM result name_supply
  = Succeeded (name_supply, result)
```

```
thenM :: M a -> (a -> M b) -> M b
thenM e k name_supply
 = case (e name_supply) of
    Succeeded (next_name_supply, result)
         -> k result next_name_supply
    Failed err -> Failed err
```

Notice that if evaluation of `e` fails, then the continuation is never applied.

We add a new monad operation to introduce failures:

```
failM :: ErrorType -> M a
failM err = Failed err
```