TYPE CLASSES IN HASKELL

CORDELIA HALL, KEVIN HAMMOND, SIMON PEYTON JONES, PHILIP WADLER
UNIVERSITY OF GLASGOW

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 an 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 [HPW91]. 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 [MTH, 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

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

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

[NS91], Volpano and Smith [VS91], and Jones [Jon92]. There has also been a good deal of traffic concerning type classes on the Haskell mailing list.

The original type inference rules given in [WB89] were deliberately rather spare, 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. We intend to publish the full set of rules as a formal static semantics of Haskell [PW91]. The full static semantics 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 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.

The static analysis phase of our Haskell compiler is now nearly complete. It was derived by adopting directly the rules in the static semantics, which 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 [MTH, MT91]. We have deliberately adopted many of the same techniques they use for mastering complexity.

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], and Jones [Jon92]. 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 [Jon92], 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.

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, Jon92], 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, Jon92], 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.

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 λ`x.` λ`ys. 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:

```
(==)         = \dEq  -> project$_1^1$ dEq
getEqFromOrd = \dOrd -> project$_1^3$ dOrd
(<)          = \dOrd -> project$_2^3$ dOrd
(<=)         = \dOrd -> project$_3^3$ dOrd
```

Here `project`$_i^n$ e selects the i'th component of a dicitionary with n entries, so that `project`$_i^n$ $\langle v_1, \ldots, v_n \rangle = v_i$. The selector `getEqFromOrd` extracts the dictionary of the superclass `Eq` from a dictionary for `Ord`, and the selectors `(<)` and `(<=)` extract the corresponding methods from a dictionary for `Ord`.

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.

Type variable	α			
Type contructor	χ			
Class name	κ			
Simple type	τ	$\rightarrow$	α	
		$\mid$	$\chi\ \tau_1\ \dots\ \tau_k$	$(k \geq 0, k = \text{arity}(\chi))$
		$\mid$	$\tau' \rightarrow \tau$	
Overloaded type	ρ	$\rightarrow$	$\langle \kappa_1\ \tau_1, \dots, \kappa_m\ \tau_m \rangle \Rightarrow \tau$	$(m \geq 0)$
Polymorphic type	σ	$\rightarrow$	$\forall \alpha_1 \dots \alpha_l . \theta \Rightarrow \tau$	$(l \geq 0)$
Context	θ	$\rightarrow$	$\langle \kappa_1\ \alpha_1, \dots, \kappa_m\ \alpha_m \rangle$	$(m \geq 0)$

FIGURE 1. Syntax of types

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

$$\forall \alpha . \langle \mathbf{Ord}\ \alpha \rangle \Rightarrow \alpha \rightarrow \mathbf{List}\ \alpha \rightarrow \text{Bool}.$$

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 $\tau = \alpha \rightarrow \mathbf{List}\ \alpha \rightarrow \text{Bool}$. Here **Ord** is a class name, **List** is a type constructor of arity 1, and **Bool** is a type constructor of arity 0.

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

$\mathit{program}$	$\rightarrow$	$\mathit{classdecls}\ ;\ \mathit{instdecls}\ ;\ \mathit{exp}$	Programs
$\mathit{classdecls}$	$\rightarrow$	$\mathit{classdecl}_1; \ldots ; \mathit{classdecl}_n$	Class declaration $(n \geq 0)$
$\mathit{instdecls}$	$\rightarrow$	$\mathit{instdecl}_1; \ldots ; \mathit{instdecl}_n$	Instance declaration $(n \geq 0)$
$\mathit{classdecl}$	$\rightarrow$	$\mathtt{class}\ \theta \Rightarrow \kappa\ \alpha$	Class declaration
		$\mathtt{where}\ \mathit{var}_1 : \tau_1\ ; \ldots ;\ \mathit{var}_m : \tau_m$	$(m \geq 0)$
$\mathit{instdecl}$	$\rightarrow$	$\mathtt{instance}\ \theta \Rightarrow \kappa\ (\chi\ \alpha_1 \ldots \alpha_k)$	Instance declaration
		$\mathtt{where}\ \mathit{var}_1 = \mathit{exp}_1\ ; \ldots ;\ \mathit{var}_m = \mathit{exp}_m$	$(k \geq 0,\ m \geq 0)$
exp	$\rightarrow$	var	Variable
	$\mid$	$\lambda\ \mathit{var}\ .\ \mathit{exp}$	Function abstraction
	$\mid$	$\mathit{exp}\ \mathit{exp}'$	Function application
	$\mid$	$\mathtt{let}\ \mathit{var} = \mathit{exp}'\ \mathtt{in}\ \mathit{exp}$	Local definition

FIGURE 2. Syntax of source programs

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

FIGURE 3. Syntax of target programs

to perform type abstraction and application. A program consists of a set of bindings, which may be mutually recursive, followed by an expression.

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

```
search = Λα. λdOrd:(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))
```

For the target language, the syntax of simple types is extended to include "dictionary types" of the form $\kappa\ \tau$. In the example above, $(\mathbf{Ord}\ \alpha)$ is a dictionary type. It can be thought of as a shorthand for the type $\langle \mathbf{Eq}\ \alpha, \alpha \rightarrow \alpha \rightarrow \mathbf{Bool}, \alpha \rightarrow \alpha \rightarrow \mathbf{Bool} \rangle$.

Environment	Notation	Type
Type variable environment	AE	$\{\alpha\}$
Type constructor environment	TE	$\{\chi : k\}$
Type class environment	CE	$\{\kappa : \forall\alpha.\ \theta \Rightarrow LVE\}$
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	$\{\mathbf{var} : \sigma\}$
Local variable environment	LVE	$\{\mathbf{var} : \tau\}$
Environment	E	$(AE, TE, CE, IE, LIE, VE)$
Declaration environment	DE	(CE, IE, VE)

FIGURE 4. Environments

$$
\begin{array}{lcll}
TE_0 & = & \{ & \texttt{Int}:\ 0,\\
 & & & \texttt{Bool}:0,\\
 & & & \texttt{List}:1\ \}\\[1ex]
CE_0 & = & \{ & \texttt{Eq}:\ \{\quad \texttt{(==)}:\ \forall\alpha.\ \langle\,\rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool}\ \},\\
 & & & \texttt{Ord}:\ \{\quad \texttt{(<)}:\ \ \forall\alpha.\ \langle\mathbf{Eq}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool},\\
 & & & \qquad\quad\ \ \texttt{(<=)}:\ \forall\alpha.\ \langle\mathbf{Eq}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool}\ \}\ \}\\[1ex]
IE_0 & = & \{ & \texttt{getEqFromOrd}:\ \ \forall\alpha.\ \langle\mathbf{Ord}\ \alpha\rangle \Rightarrow \mathbf{Eq}\ \alpha,\\
 & & & \texttt{dictEqInt}:\qquad \mathbf{Eq\ Int},\\
 & & & \texttt{dictEqList}:\quad\ \ \forall\alpha.\ \langle\mathbf{Eq}\ \alpha\rangle \Rightarrow \mathbf{Eq}\ (\mathbf{List}\ \alpha),\\
 & & & \texttt{dictOrdInt}:\quad\ \ \mathbf{Ord\ Int}\ \}\\[1ex]
VE_0 & = & \{ & \texttt{(==)}:\forall\alpha.\ \langle\mathbf{Eq}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool},\\
 & & & \texttt{(<)}:\ \forall\alpha.\ \langle\mathbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool},\\
 & & & \texttt{(<=)}:\forall\alpha.\ \langle\mathbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool}\ \}\\[1ex]
E_0 & = & \multicolumn{2}{l}{(\{\,\}, TE_0, CE_0, IE_0, \{\,\}, VE_0)}
\end{array}
$$

FIGURE 5. Initial environments

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* 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 contructor environment TE maps a type constructor χ to its arity k.
- The type class environment CE maps a class name κ to a value environment VE that specifies the type of each operator in the class. This is explained in more detail below.
- 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.

Given a local value environment LVE the notation

$$\forall \alpha_1 \ldots \alpha_k.\ \theta \Rightarrow LVE$$

stands for a value environment VE with the same domain as LVE such that if LVE **var** $=$ τ then VE **var** $= \forall \alpha_1 \ldots \alpha_k.\ \theta \Rightarrow \tau$. This is used when specifying the type of CE above, to make clear that each entry in the associated value environment has the same quantified type variable and context. We use a similar notation to generate an instance environment IE from a local instance environment LIE.

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.
- 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:

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

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 similarly for other environments.

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.

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

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

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

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

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

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

$$\text{TYPE-GEN} \quad \frac{\begin{array}{l} AE \ = \ \{\alpha_1, \ldots, \alpha_k\} \\ E \oplus AE \overset{\text{over-type}}{\vdash} \theta \Rightarrow \tau \end{array}}{E \overset{\text{poly-type}}{\vdash} \forall \alpha_1 \ldots \alpha_k .\ \theta \Rightarrow \tau}$$

FIGURE 6. Rules for types

(2) Every variable must appear in the environment. If $E\ \mathit{var}$ appears in a rule, then the side condition $\mathit{var} \in \mathit{dom}(E)$ 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

$$\forall\ \kappa_1\ (\chi_1\ \alpha_1 \ldots \alpha_m)\ \in\ IE_1 .\ \forall\ \kappa_2\ (\chi_2\ \alpha_1 \ldots \alpha_n)\ \in\ IE_2 .\ \kappa_1\ \neq\ \kappa_2\ \vee\ \chi_1\ \neq\ \chi_2$$

is implied.

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. 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 $AE\ \mathit{of}\ E$ (as checked by rule TYPE-VAR), and all type constructors in τ must appear in $TE\ \mathit{of}\ E$ 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 $\forall\alpha.\ \langle\mathbf{Ord}\ \alpha\rangle \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathbf{Bool}$. Let $AE = \{\alpha\}$ and let E_0 be as in Figure 5. Then the following are valid judgements:

$$\begin{array}{ll}
(1) & E_0 \oplus AE \overset{\texttt{type}}{\vdash} \alpha \rightarrow \alpha \rightarrow \mathbf{Bool}, \\
(2) & E_0 \oplus AE \overset{\texttt{over-type}}{\vdash} \langle\mathbf{Ord}\ \alpha\rangle \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathbf{Bool}, \\
(3) & E_0 \overset{\texttt{poly-type}}{\vdash} \forall\alpha.\ \langle\mathbf{Ord}\ \alpha\rangle \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathbf{Bool}.
\end{array}$$

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

$$E \overset{\texttt{exp}}{\vdash} exp : \tau \ \rightsquigarrow\ \mathbf{exp}$$

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 $AE = \{\alpha\}$, $LIE = \{\texttt{dOrd} : \mathbf{Ord}\ \alpha\}$, and $VE = \{\texttt{x} : \alpha, \texttt{y} : \alpha\}$. Then the following are valid judgements:

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

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

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

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

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

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

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

$$\text{REL}\quad \frac{\begin{array}{ll} E \overset{\text{over-exp}}{\vdash} var : \langle \kappa_1\ \tau_1, \ldots, \kappa_m\ \tau_m \rangle \Rightarrow \tau \ \leadsto\ \mathbf{exp} & \\ E \overset{\text{dict}}{\vdash} \kappa_i\ \tau_i \ \leadsto\ \mathbf{dexp}_i & (1 \leq i \leq m) \end{array}}{E \overset{\text{exp}}{\vdash} var : \tau \ \leadsto\ \mathbf{exp}\ \mathbf{dexp}_1 \ldots \mathbf{dexp}_m}$$

$$\text{ABS}\quad \frac{\begin{array}{l} VE \ = \ \{\mathbf{var} : \tau'\} \\ E \overset{\rightarrow}{\oplus} VE \overset{\text{exp}}{\vdash} exp : \tau \ \leadsto\ \mathbf{exp} \end{array}}{E \overset{\text{exp}}{\vdash} \lambda var .\ exp : \tau' \rightarrow \tau \ \leadsto\ \lambda \mathbf{var} \colon \tau' .\ \mathbf{exp}}$$

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

$$\text{PRED}\quad \frac{\begin{array}{ll} (CE\ of\ E)\ \tau_i & (1 \leq i \leq m) \\ (AE\ of\ E)\ \alpha_i & (1 \leq i \leq m) \\ LIE \ = \ \{\mathbf{dvar}_1 : \kappa_1\ \alpha_1, \ldots, \mathbf{dvar}_m : \kappa_m\ \alpha_m\} & \\ E \oplus LIE \overset{\text{exp}}{\vdash} exp : \tau \ \leadsto\ \mathbf{exp} & \end{array}}{\begin{array}{l} E \overset{\text{over-exp}}{\vdash} exp : \langle \kappa_1\ \alpha_1, \ldots, \kappa_m\ \alpha_m \rangle \Rightarrow \tau \\ \quad \leadsto \\ \quad \lambda\ \mathbf{dvar}_1 : \kappa_1\ \alpha_1 .\ \ldots\ .\lambda\ \mathbf{dvar}_m : \kappa_m\ \alpha_m .\ \mathbf{exp} \end{array}}$$

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

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

FIGURE 7. Rules for expressions

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

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

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

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

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

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

$$\text{DICT-REL}\quad \frac{\begin{array}{l} E \overset{\text{over-dict}}{\vdash} \langle\kappa_1\ \tau_1, \ldots, \kappa_n\ \tau_n\rangle \Rightarrow \kappa\ \tau \leadsto \mathbf{dexp} \\ E \overset{\text{dict}}{\vdash} \kappa_i\ \tau_i \leadsto \mathbf{dexp}_i \qquad\qquad (1 \leq i \leq n) \end{array}}{E \overset{\text{dict}}{\vdash} \kappa\ \tau \leadsto \mathbf{dexp}\ \mathbf{dexp}_1 \ldots \mathbf{dexp}_n}$$

FIGURE 8. Rules for dictionaries

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.

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:

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

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.

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

$$\text{CLASS}\quad \frac{\begin{array}{lll} (1) & \theta = \langle \kappa_1\ \alpha, \ldots \kappa_l\ \alpha\rangle & \\ (2) & (CE\ of\ E)\ \kappa_i & (1 \le i \le l) \\ (3) & AE = \{\alpha\} & \\ (4) & E \oplus AE \overset{\text{type}}{\vdash} \tau_i & (1 \le i \le n) \\ (5) & LIE = \{\mathbf{dvar}_1 : \kappa_1\ \alpha, \ldots, \mathbf{dvar}_n : \kappa_n \alpha\} & \\ (6) & LVE = \{\mathbf{var}_1 : \tau_1, \ldots, \mathbf{var}_n : \tau_n\} & \end{array}}{\begin{array}{ll} E \overset{\text{classdecl}}{\vdash} & \texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ var_1 : \tau_1, \ldots, var_n : \tau_n \\ & : \\ & (\{\kappa : \forall\alpha.\ \theta \Rightarrow LVE\},\ \forall\alpha.\ \langle\kappa\ \alpha\rangle \Rightarrow LIE, \forall\alpha.\ \langle\kappa\ \alpha\rangle \Rightarrow LVE) \\ & \leadsto \\ & \begin{array}{lcl} \mathbf{dvar}_1 & = & \lambda\ \mathbf{dvar} : \kappa\ \alpha.\ \texttt{project}_1^{m+n}\ \mathbf{dvar}; \\ \ldots & & \\ \mathbf{dvar}_m & = & \lambda\ \mathbf{dvar} : \kappa\ \alpha.\ \texttt{project}_m^{m+n}\ \mathbf{dvar}; \\ \mathbf{var}_1 & = & \lambda\ \mathbf{dvar} : \kappa\ \alpha.\ \texttt{project}_{m+1}^{m+n}\ \mathbf{dvar}; \\ \ldots & & \\ \mathbf{var}_n & = & \lambda\ \mathbf{dvar} : \kappa\ \alpha.\ \texttt{project}_{m+n}^{m+n}\ \mathbf{dvar} \end{array} \end{array}}$$

FIGURE 9. Rule for class declarations

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

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

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

holds if in environment E 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 9. Again the rule looks formidable, and again its workings are straightforward.

A judgement of the form

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

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

$$\text{INST}\quad \frac{\begin{array}{ll}
(1) & \tau = \chi\ \alpha_1 \ldots \alpha_k \\
(2) & (TE\ of\ E)\ \chi = k \\
(3) & AE = \{\alpha_1, \ldots, \alpha_k\} \\
(4) & \theta' = \langle \kappa'_1\ \tau'_1, \ldots, \kappa'_l\ \tau'_l \rangle \\
(5) & (CE\ of\ E)\ \kappa_i \qquad\qquad (1 \leq i \leq l) \\
(6) & AE\ \tau'_i \qquad\qquad (1 \leq i \leq l) \\
(7) & (CE\ of\ E)\ \kappa = \forall\alpha.\ \langle \kappa_1\ \alpha, \ldots, \kappa_m\ \alpha \rangle \Rightarrow \{\mathbf{var}_1 : \tau_1, \ldots, \mathbf{var}_n : \tau_n\} \\
(8) & LIE = \{\mathbf{dvar}_1 : \kappa'_1\ \tau'_1, \ldots, \mathbf{dvar}_l : \kappa'_l\ \tau'_l\} \\
(9) & E \oplus AE \oplus LIE \overset{\text{dict}}{\vdash} \kappa_i\ \tau \leadsto \mathbf{dexp}_i \qquad\qquad (1 \leq i \leq m) \\
(10) & E \oplus AE \oplus LIE \overset{\text{exp}}{\vdash} exp_i : \tau_i[\tau/\alpha] \leadsto \mathbf{exp}_i \qquad\qquad (1 \leq i \leq n)
\end{array}}{\begin{array}{ll}
E \overset{\text{instdecl}}{\vdash} & \texttt{instance}\ \theta \Rightarrow \kappa\ \tau\ \texttt{where}\ var_1 = exp_1, \ldots, var_m = exp_n \\
& : \\
& \{\mathbf{dvar} : \forall\alpha_1 \ldots \alpha_k.\ \theta \Rightarrow \kappa\ \tau\} \\
& \leadsto \\
& \mathbf{dvar} = \Lambda\alpha_1 \ldots \alpha_k\ . \\
& \qquad\qquad \lambda\mathbf{dvar}_1 : \kappa'_1\ \tau'_1.\ \ldots\ \lambda\mathbf{dvar}_l : \kappa'_l\ \tau'_l. \\
& \qquad\qquad\quad \langle \mathbf{dexp}_1, \ldots, \mathbf{dexp}_m, \mathbf{exp}_1, \ldots, \mathbf{exp}_n \rangle
\end{array}}$$

FIGURE 10. Rule for instance declarations

holds if in environment E the instance declaration *instdecl* is valid, generating new environment IE and yielding translation **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.

Line (1) ensures that τ has the form $\chi\ \alpha_1 \ldots \alpha_k$. Line (3) sets AE to contain the type variables in τ, line (4) extracts the entries $\kappa'_i\ \tau'_i$ from the context θ, and line (6) guarantees that each τ'_i in the entry is one of the free type variables of τ. The other lines check that the given methods correspond to those required by the class, and generate the required dictionaries for the superclasses.

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 11 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

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

$$\text{CDECLS}\quad \frac{E \oplus DE_1 \oplus \ldots \oplus DE_{i-1} \overset{\text{classdecl}}{\vdash} \mathit{classdecl}_i : DE_i \leadsto \mathbf{bindset}_i \ (1 \leq i \leq n)}{\begin{array}{l} E \overset{\text{classdecls}}{\vdash} \mathit{classdecl}_1 \ ; \ \ldots \ ; \ \mathit{classdecl}_n : DE_1 \oplus \ldots \oplus DE_n \\ \leadsto \\ \mathbf{bindset}_1 ; \ \ldots \ ; \mathbf{bindset}_n \end{array}}$$

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

$$\text{IDECLS}\quad \frac{E \overset{\text{instdecl}}{\vdash} \mathit{instdecl}_i : IE_i \leadsto \mathbf{bindset}_i \ (1 \leq i \leq n)}{\begin{array}{l} E \overset{\text{instdecls}}{\vdash} \mathit{instdecl}_1 \ ; \ \ldots \ ; \ \mathit{instdecl}_n : (IE \text{ of } E) \oplus IE_1 \oplus \ldots \oplus IE_n \\ \leadsto \\ \mathbf{bindset}_1 ; \ \ldots \ ; \mathbf{bindset}_n \end{array}}$$

$$\boxed{E \overset{\text{program}}{\vdash} \mathit{program} : (DE, \tau) \leadsto \mathbf{exp}}$$

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

FIGURE 11. Rules for declaration sequences and programs

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.

This concludes our presentation of the inference rules for type classes in Haskell. An important feature of this style of presentation is that it scales up well to a description of the entire Haskell language. The rules have been of immense help in constructing our new Haskell compiler.

REFERENCES

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

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

[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 *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.

[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*, pp. 29–60, December 1969.

[HPW91] P. Hudak, S. L. Peyton Jones, and P. Wadler, editors, Report on the Programming Language Haskell, Dept. of Computing Science, University of Glasgow, August (1991).

[Jon92] M. P. Jones, A theory of qualified types. In *European Symposium on Programming*, Rennes, February 1992, LNCS (to appear), Springer-Verlag.

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

[MTH] 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*, pp. 348–375, 1978.

[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.

[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 *Principles of Programming Languages*, San Francisco, January 1990, ACM Press.

[Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.

[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.

[WB89] P. L. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc", *Proc 16th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January (1989), pp. 60–76.