Type classes in Haskell

Cordelia Hall, Kevin Hammond, Simon Peyton Jones
and Philip Wadler

Glasgow University

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

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

---

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.

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. It presents a simplified subset of the rules we derived. The full set of rules is given elsewhere [PW91], and contains over 30 judgement forms and over 100 rules, which deal with many additional syntactic features such as type declarations, pattern matching, and list comprehensions. 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.

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. We use the syntax `\ x ys -> e` to stand for λ`x`. λ`ys`. `e`.

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

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 context of the inferred signature would have been `(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) &&
```

Type variable	α			
Type contructor	χ			
Class name	κ			
Simple type	τ	$\rightarrow$	α	
		$\mid$	$\chi\ \tau_1\ \ldots\ \tau_k$	$(k \geq 0, k = \text{arity}(\chi))$
		$\mid$	$\tau' \rightarrow \tau$	
Overloaded type	ρ	$\rightarrow$	$\langle \kappa_1\ \tau_1, \ldots, \kappa_m\ \tau_m \rangle \Rightarrow \tau$	$(m \geq 0)$
Polymorphic type	σ	$\rightarrow$	$\forall \alpha_1 \ldots \alpha_l . \theta \Rightarrow \tau$	$(l \geq 0)$
Context	θ	$\rightarrow$	$\langle \kappa_1\ \alpha_1, \ldots, \kappa_m\ \alpha_m \rangle$	$(m \geq 0)$
Record Type	γ	$\rightarrow$	$\langle v_1 : \tau_1, \ldots, v_n : \tau_n \rangle$	$(n \geq 0)$

FIGURE 1. Syntax of types

```
( (==) (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.

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.

$program$	$\rightarrow$	$classdecls$; $instdecls$; exp	Programs
$classdecls$	$\rightarrow$	$classdecl_1 ; \ldots ; classdecl_n$	Class decls. $(n \geq 0)$
$instdecls$	$\rightarrow$	$instdecl_1 ; \ldots ; instdecl_n$	Instance decls. $(n \geq 0)$
$classdecl$	$\rightarrow$	`class` $\theta \Rightarrow \kappa\ \alpha$ `where` γ	Class declaration
$instdecl$	$\rightarrow$	`instance` $\theta \Rightarrow \kappa\ (\chi\ \alpha_1 \ldots \alpha_k)$ `where` $binds$	Instance decl. $(k \geq 0)$
$binds$	$\rightarrow$	$\langle var_1 = exp_1\ ,$ $\ldots\ ,\ var_n = exp_n \rangle$	$(n \geq 0)$
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

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

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 and Figure 3 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 used here differs in that all polymorphism has been made explicit. It has constructs for type abstraction and application, and each bound variable is labeled with its type. 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. The class types appearing in the translation denote monotypes; a formal translation of them appears in [HaHaPJW].

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** $\mathbf{exp}'$	Function application
	$\mid$	`let` **var** $=$ $\mathbf{exp}'$ `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	$\{\kappa :$ `class` $\theta \Rightarrow \kappa\ \alpha$ `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

3.3. Environments. The inference rules use a number of different environments, which are summarised in Figure 4. 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 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 $\overrightarrow{\oplus}$ combine environments. The former checks that the domains of its arguments are distinct, while the latter "shadows" its left

$$
\begin{array}{lll}
TE_0 = \{ & \texttt{Int}: 0, & \\
& \texttt{Bool}:0, & \\
& \texttt{List}:1 \} &
\end{array}
$$

$$
\begin{array}{lll}
CE_0 = \{ & \texttt{Eq} : \{\texttt{class Eq}\ \alpha\ \textbf{where}\ \langle \texttt{(==)}:\alpha \to \alpha \to \textbf{Bool}\rangle\ \}, & \\
& \texttt{Ord} : \{\texttt{class}\ \langle \texttt{Eq}\ \alpha\rangle \Rightarrow\ \texttt{Ord}\ \alpha & \\
& \qquad \textbf{where}\ \langle \texttt{(<)}:\alpha \to \alpha \to \textbf{Bool}, \texttt{(<=)}:\ \alpha \to \alpha \to \textbf{Bool}\rangle\ \}\ \} &
\end{array}
$$

$$
\begin{array}{lll}
IE_0 = \{ & \texttt{getEqFromOrd}: & \forall\alpha.\ \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \textbf{Eq}\ \alpha, \\
& \texttt{dictEqInt}: & \textbf{Eq Int}, \\
& \texttt{dictEqList}: & \forall\alpha.\ \langle \textbf{Eq}\ \alpha\rangle \Rightarrow \textbf{Eq}\ (\textbf{List}\ \alpha), \\
& \texttt{dictOrdInt}: & \textbf{Ord Int}\ \}
\end{array}
$$

$$
\begin{array}{lll}
VE_0 = \{ & \texttt{(==)}:\forall\alpha.\ \langle \textbf{Eq}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}, & \\
& \texttt{(<)}:\ \forall\alpha.\ \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}, & \\
& \texttt{(<=)}:\forall\alpha.\ \langle \textbf{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \textbf{Bool}\ \} &
\end{array}
$$

$$E_0 = (\{\}, TE_0, CE_0, IE_0, \{\}, VE_0)$$

FIGURE 5. Initial environments

argument with its right:

$$
\begin{array}{l}
(ENV_1 \oplus ENV_2)\ var = \\
\left\{ \begin{array}{l} ENV_1\ var \text{ if } var \in dom(ENV_1) \text{ and } var \notin dom(ENV_2) \\ ENV_2\ var \text{ if } var \in dom(ENV_2) \text{ and } var \notin dom(ENV_1), \end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
(ENV_1 \stackrel{\rightarrow}{\oplus} ENV_2)\ var = \\
\left\{ \begin{array}{ll} ENV_1\ var & \text{if } var \in dom(ENV_1) \text{ and } var \notin dom(ENV_2) \\ ENV_2\ var & \text{if } var \in dom(ENV_2). \end{array} \right.
\end{array}
$$

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

There are three implicit side conditions associated with environments. 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. Every variable must appear in the environment. If $E\ var$ appears in a rule, then the side condition $var \in dom(E)$ is implied. 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

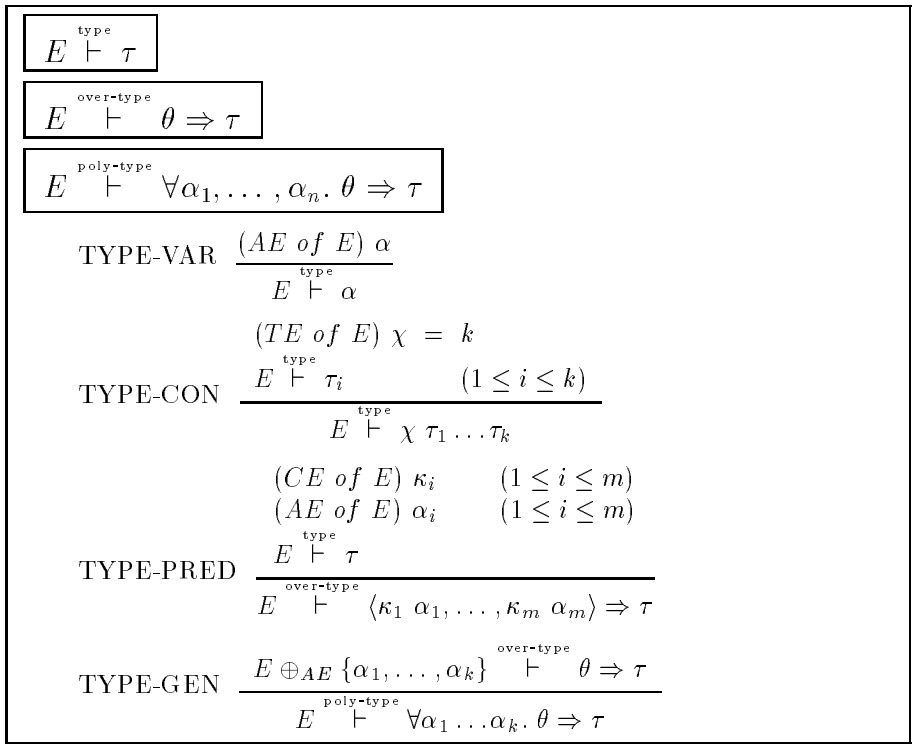

FIGURE 6. Rules for types

condition

$$\begin{array}{l} \forall\ \kappa_1\ (\chi_1\ \alpha_1 \ldots \alpha_m)\ \in\ IE_1. \\ \forall\ \kappa_2\ (\chi_2\ \alpha_1 \ldots \alpha_n)\ \in\ IE_2. \\ \kappa_1\ \neq\ \kappa_2\ \vee\ \chi_1\ \neq\ \chi_2 \end{array}$$

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. 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.

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

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

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.

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

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

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.

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

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

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).

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

$\boxed{E \overset{\text{over-exp}}{\vdash} exp : \rho \rightsquigarrow \textbf{exp}}$

$\boxed{E \overset{\text{poly-exp}}{\vdash} exp : \sigma \rightsquigarrow \textbf{exp}}$

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

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

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

$$\text{ABS}\ \frac{E \overset{\rightarrow}{\oplus}_{VE} \{\textbf{var} : \tau'\} \overset{\text{exp}}{\vdash} exp : \tau \rightsquigarrow \textbf{exp}}{E \overset{\text{exp}}{\vdash} \lambda var.\ exp : \tau' \to \tau \rightsquigarrow \lambda \textbf{var}\textbf{:}\ \tau'.\ \textbf{exp}}$$

$$\text{COMB}\ \frac{\begin{array}{l} E \overset{\text{exp}}{\vdash} exp : \tau' \to \tau \rightsquigarrow \textbf{exp} \\ E \overset{\text{exp}}{\vdash} exp' : \tau' \rightsquigarrow \textbf{exp}' \end{array}}{E \overset{\text{exp}}{\vdash} (exp\ exp') : \tau \rightsquigarrow (\textbf{exp}\ \textbf{exp}')}$$

$$\text{PRED}\ \frac{\begin{array}{ll} E \oplus LIE \overset{\text{dicts}}{\vdash} \theta \rightsquigarrow \textbf{dpat} & \theta \text{ determines } LIE \\ E \oplus LIE \overset{\text{exp}}{\vdash} exp : \tau \rightsquigarrow \textbf{exp} & \end{array}}{E \overset{\text{over-exp}}{\vdash} exp : \theta \Rightarrow \tau \rightsquigarrow \lambda\ \textbf{dpat} : \theta\ .\ \textbf{exp}}$$

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

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

FIGURE 7. Rules for expressions

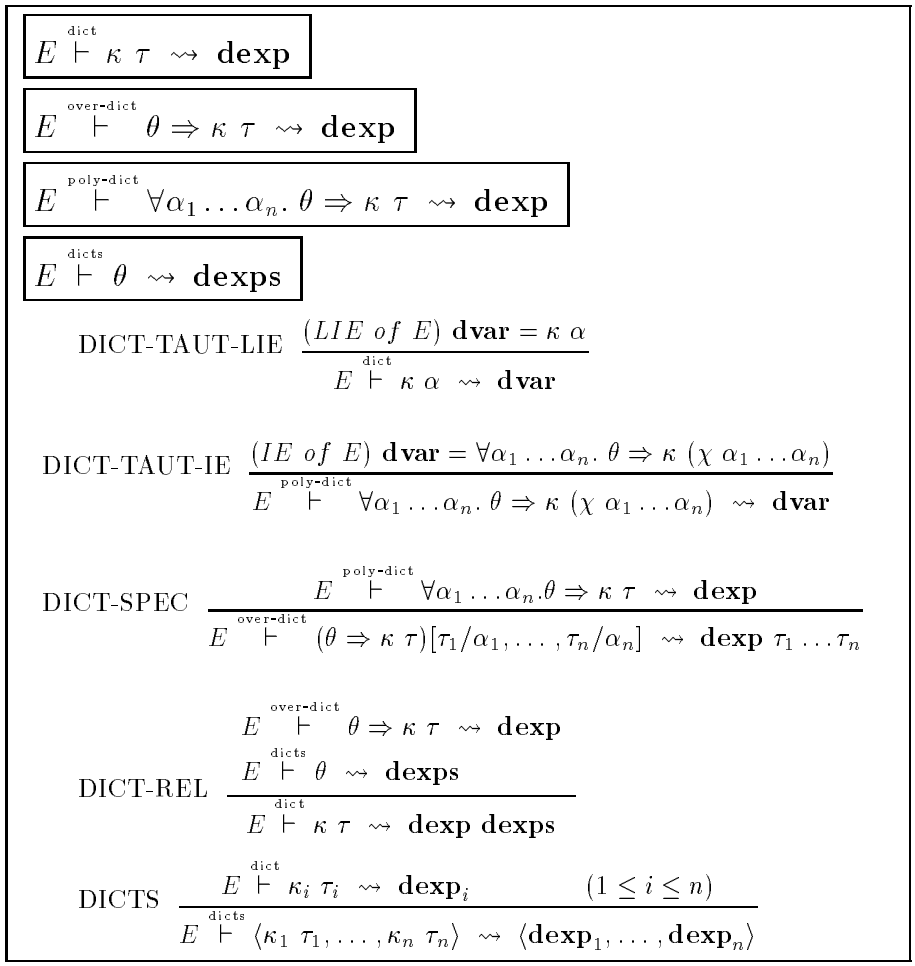

FIGURE 8. Rules for dictionaries

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

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

holds if in environment PE 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.

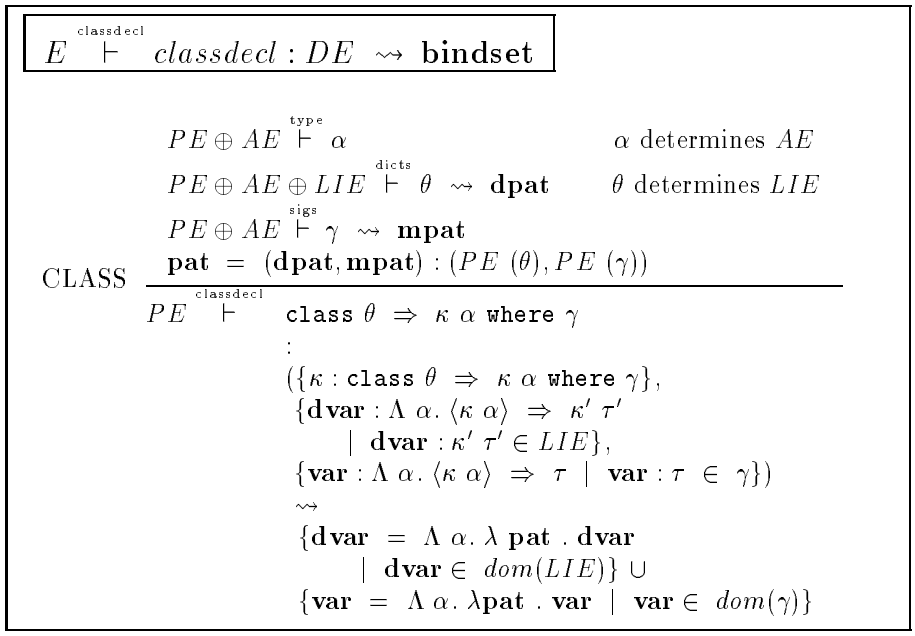

FIGURE 9. Rule for class declarations

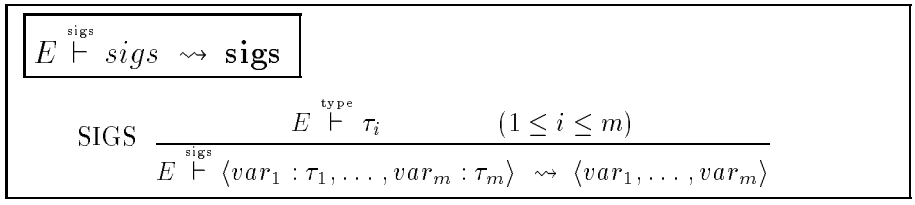

FIGURE 10. Rule for class signatures

4.6. Programs. We omit the rules for declaration sequences and programs; these appear in the full technical report [HaHaPJW].

5. Conclusions

This paper 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. We have found in practice that 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

$$\boxed{E \overset{\text{instdecl}}{\vdash} \mathit{instdecl} : IE \rightsquigarrow \textbf{bindset}}$$

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

FIGURE 11. Rule for instance declarations

$$\boxed{E \overset{\text{binds}}{\vdash} \mathit{binds} : \gamma \rightsquigarrow \textbf{binds}}$$

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

FIGURE 12. Rule for instance bindings

continues to pay off as we maintain our code and train students to work with it.

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.

[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.

[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.

[HaHaPJW] C.V. Hall, K. Hammond, S.L.Peyton Jones and P. Wadler, Type Classes In Haskell. Department of Computing Science, Glasgow University, Jan 1994.

[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.

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

[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.