Type Classes in Haskell

CORDELIA V. HALL, KEVIN HAMMOND, SIMON L. PEYTON JONES,
and PHILIP L. WADLER
Glasgow University

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

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Language Classifications—*applicative languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Languages, Theory

Additional Key Words and Phrases: Functional programming, Haskell, type classes, types

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] [Turner 1985] and Standard ML [Milner and Tofte 1991; Milner et al. 1990] 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 Haskell committee adopted a completely new technique, based on a proposal by Wadler, which extends the familiar Hindley-Milner system [Milner 1978] with *type classes*. Type classes provide a uniform solution to overloading, including

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

This work is supported by the UK EPSRC AQUA, Parade, and Save Space grants, and by an SOED fellowship from the Royal Society of Edinburgh.

Authors' address: Computing Science Department, Glasgow University, 17 Lilybank Gardens, Glasgow, Scotland.

providing operations for equality, arithmetic, and string conversion. They generalize 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 [1989], and a similar proposal was made independently by Kaes [1988].

The type system of Haskell is certainly its most innovative feature and has provoked much discussion. There has been closely related work by Rouaix [1990] and Cormack and Wright [1990]; Nipkow and Snelting [1991] and Nipkow and Prehofer [1993] also describe type inference algorithms for type classes based on sorts; and work directly inspired by Haskell type classes includes the closely related Gofer type system with its multiparameter classes [Jones 1992a; 1992b; 1994], an extension of ML with type classes [Volpano and Smith 1991], constructor classes [Jones 1995], first-class abstract data types [Läufer 1992; 1993; Läufer and Odersky 1994; Odersky and Läufer 1991], and parametric type classes [Chen 1994; 1995; Chen et al. 1992].

This article 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 [Huet 1990]. 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 shortcoming of the system presented here in 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 Kaes [1988] does give a direct semantics for a slightly simpler form of overloading.)

The original type inference rules given by Wadler and Blott [1989] 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 Article

In comparison with previous work, this article makes several new contributions.

(1) With the few exceptions noted in Section 1.2, this article 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.

(2) This article presents a simplified subset of the rules we derived. The full static semantics of Haskell [Peyton Jones and Wadler 1991] contains over 30 judgment forms and over 100 rules. The reader will be pleased to know that this article 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.

(3) This article shows how the static analysis phase of our Haskell compiler was derived by adopting directly the rules in the static semantics. This was gener-

ally a very straightforward task. In our earlier prototype compiler, and 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 et al. [1990; 1991]. We have deliberately adopted many of the same techniques they use for mastering complexity.

(4) 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 [1989], Blott [1991], Nipkow and Snelting [1991], Nipkow and Prehofer [1993], and Jones [1992a; 1992b; 1993]. A number of simplifying assumptions made in those papers are not made here. Unlike Wadler and Blott [1989], it is not assumed that each class has exactly one operation. Unlike Nipkow and Snelting [1991], it is not assumed that the intersection of every pair of classes must be separately declared. Unlike Jones [1992a; 1992b], we deal directly with instance and class declarations. Each of those papers emphasizes one aspect or another of the theory, while this article stresses what we learned from practice. At the same time, these rules provide a clean, "high-level" specification for the implementation of a typechecker, unlike more implementation-oriented papers [Augustsson 1993; Hammond and Blott 1989].

A further contribution of this work is the use of explicit polymorphism in the target language, as described in Sections 1.3 and 5.2.

1.2 Outstanding Issues

Since this article does not attempt to be a complete semantics for type classes in Haskell, there are a number of issues which are not addressed here. These issues are addressed by the full static semantics [Peyton Jones and Wadler 1991]. The most important of these are:

(1) The rules do not allow polymorphic class methods: the only type variable which can appear in the type of a method is the one constrained by the class declaration. It would be straightforward to extend the rules to cover this case.

(2) There is no treatment of the Haskell *monomorphism* restriction, which was introduced in Haskell 1.1 for efficiency reasons. Essentially, the restriction limits the use of the GEN rule for value definitions with overloaded types, so that overloaded type variables are generalized only for function definitions or if an explicit type signature is provided. This improves sharing in the translation: since an overloaded value could be used at multiple types it would normally need to be recomputed once for each occurrence in the enclosing definition; the monomorphism restriction ensures that (1) the value is used only at one type and (2) that the result may therefore be reused. Papers by Hammond and Blott [1991] and Augustsson [1993] consider this in more detail.

(3) This article does not consider the use of *default types* to resolve certain type ambiguities. While the use of default types has been found to be invaluable in practice, they can technically lead to an incoherent semantics, as demonstrated by Blott [1991]. A good treatment of ambiguity in type classes can be found in Nipkow and Prehofer [1993].

1.3 A Target Language with Explicit Polymorphism

As in Wadler and Blott [1989], Nipkow and Snelting [1991], and Jones [1992a; 1992b], 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 Wadler and Blott [1989], Nipkow and Snelting [1991], and Jones [1992a; 1992b], the target language resembles the implicitly typed polymorphic lambda calculus of Hindley [1969] and Milner [1978]. Here, the target language resembles the explicitly typed second-order polymorphic lambda calculus of Girard [1972] and Reynolds [1974]. 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 optimizations depend on knowing a subterm's type. An alternative might be to annotate each subterm with its type, but our method has three advantages.

(1) It uses less space. Types are stored in type applications and with each bound variable, rather than at every subterm.

(2) It eases subsequent transformation. A standard and productive technique for compiling functional languages is to apply various transformations at intermediate phases [Peyton Jones 1987]. 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.

(3) 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.

1.4 Structure of the Article

This article 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 [1989]. For a comparison of the Hindley-Milner and Girard-Reynolds systems, see the excellent summary by Reynolds [1985]. For a practicum on Hindley-Milner type inference, see the tutorial by Cardelli [1987].

The remainder of this article is organized as follows. Section 2 introduces type classes and our translation method. Section 3 describes the various notations used in presenting the inference 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 6 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 article.

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`. The same is true for characters.

Under this system both `2+2 == 4` and `'a' == 'b'` are well typed. 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 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`. The two expressions `(member 1 [2,3])` or `(member 'a' ['c','a','t'])` are therefore well typed, but `(member sin [cos,tan])` is not, 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 generalizing this behavior.

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 vice versa*, 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]`. The expression `['c','a','t'] == ['d','o','g']` is therefore well typed.

Every entry in a context pairs a class name with a type variable. Pairing a class name with a more specific 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])`. The simpler context can be inferred from the context for the instance declaration, since whenever the type `[a]` belongs to class `Eq`, the type `a` must also belong to class `Eq`.

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 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` from Section 2.2:

```
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]` from Section 2.1:

```
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 flavors: 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 $\forall\alpha.\langle \mathbf{Ord}\ \alpha\rangle \Rightarrow \alpha \rightarrow \mathbf{List}\ \alpha \rightarrow \mathbf{Bool}$. 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 $\tau = \alpha \rightarrow \mathbf{List}\ \alpha \rightarrow \mathbf{Bool}$. There are also record types, γ, which map class operation names to their types. These

Type variable	α			
Type contructor	χ			
Class name	κ			
Simple type	τ	$\rightarrow$	α	
		$\mid$	$\chi\ \tau_1 \ldots \tau_n$	$(n \geq 0,\ n = \text{arity}(\chi))$
		$\mid$	$\tau' \rightarrow \tau$	
Overloaded type	ρ	$\rightarrow$	$\phi \Rightarrow \tau$	
Polymorphic type	σ	$\rightarrow$	$\forall \alpha_1 \ldots \alpha_n\ .\ \theta \Rightarrow \tau$	$(n \geq 0)$
Context	θ	$\rightarrow$	$\langle \kappa_1\ \alpha_1, \ldots, \kappa_n\ \alpha_n \rangle$	$(n \geq 0)$
Saturated Context	ϕ	$\rightarrow$	$\langle \kappa_1\ \tau_1, \ldots, \kappa_n\ \tau_n \rangle$	$(n \geq 0)$
Record Type	γ	$\rightarrow$	$\langle v_1 : \tau_1, \ldots, v_n : \tau_n \rangle$	

Fig. 1. Syntax of types.

program	$\rightarrow$	*classdecls* ; *instdecls* ; *exp*	Programs
classdecls	$\rightarrow$	$\mathit{classdecl}_1; \ldots; \mathit{classdecl}_n$	Class declaration $(n \geq 0)$
instdecls	$\rightarrow$	$\mathit{instdecl}_1; \ldots; \mathit{instdecl}_n$	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	$\rightarrow$	$\langle \mathit{var}_1 = \mathit{exp}_1\ , \ldots,\ \mathit{var}_n = \mathit{exp}_n \rangle$	$(n \geq 0)$
exp	$\rightarrow$	*var*	Variable
	$\mid$	λ *var* . *exp*	Function abstraction
	$\mid$	*exp* *exp*$'$	Function application
	$\mid$	`let` *var* = *exp*$'$ `in` *exp*	Local definition

Fig. 2. Syntax of source programs.

appear in the source syntax for classes. 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 or a saturated context ϕ, entries may have the form $\kappa\ \tau$, whereas in a polymorphic type σ or a context θ entries are restricted to the form $\kappa\ \alpha$ (as with the `palindrome` example from Section 2.1. 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

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

Fig. 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	$\{\mathbf{dvar} : \forall \alpha_1 \ldots \alpha_n .\ \theta \Rightarrow \kappa\ \tau\}$
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)

Fig. 4. Environments.

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. Type inference rules for this language appear in Section 5.

Notice that no class types appear in the translation. Given an environment E as defined below, a context θ or record type γ can be converted into a monotype by $tran\ E\ \theta$ or $tran\ E\ \gamma$. The $tran$ function is defined in Section 4.7.

3.3 Environments

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

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

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

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

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

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

Fig. 5. Initial environments.

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 flavors: 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 constructor environment TE maps each 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 $\kappa\ \tau$. This environment is used to record information about globally visible dictionaries and dictionary selectors.

—The local instance environment LIE is similar, except that the associated type is monomorphic. Here the type indicates that **dvar** is a dictionary for $\kappa\ \tau$. This environment is used to record information about specific dictionaries.

—The variable environment VE maps a variable **var** to its associated polymorphic type σ.

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 judgments 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 judgments for class declarations produce a declaration environment DE consisting of a type class, an instance, and a variable environment.

Again, these are summarized 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 $\overset{\rightarrow}{\oplus}$ combine environments. The former checks that the domains of its arguments are distinct, while the latter "shadows" its left argument with its right:

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

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

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 similarly for other environments. Sometimes we specify the components 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

$$\forall\, \kappa_1\ (\chi_1\ \alpha_1 \ldots \alpha_m)\ \in\ IE_1.\ \forall\, \kappa_2\ (\chi_2\ \beta_1 \ldots \beta_n)\ \in\ IE_2.\ \ \kappa_1 \neq \kappa_2\ \vee\ \chi_1 \neq \chi_2$$

is implied.

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

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

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

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

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

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

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

Fig. 6. Type formation rules.

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

$$\begin{aligned} \tau \text{ determines } AE &\iff ftv(\tau) \ = \ AE \\ \theta \text{ determines } LIE &\iff \theta \ = \ ran(LIE). \end{aligned}$$

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 judgment forms defined are summarized in the upper left corner. A judgment 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 judgments 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 judgments:

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

Judgment (1) yields (2) via TYPE-PRED, and judgment (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 Figures 7–8. A judgment 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 judgments 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 handles let bindings; 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. Unlike the original Hindley-Milner system and most of its derivatives, we have chosen to restrict type variable generalization and specialization: our rules do not allow spurious type variables to be introduced by GEN to be removed by SPEC. This is partly a matter of taste, in that it more closely reflects the type algorithm, but it also reduces the size of the translation and eliminates the possibility that alternative but equivalent translations could be produced by these rules.

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

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

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

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

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

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

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

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

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

Fig. 7. Rules for expressions, part 1.

Then the following are valid judgments:

$$(1)\qquad E_0 \oplus AE \oplus LIE \oplus VE \overset{\text{exp}}{\vdash} \texttt{x < y} : \mathbf{Bool} \leadsto \texttt{(<)}\ \alpha\ \texttt{dOrd x y}$$

$$(2)\qquad E_0 \oplus AE \oplus LIE \overset{\text{exp}}{\vdash} \texttt{\textbackslash\ x y -> x < y} : \alpha \to \alpha \to \mathbf{Bool} \leadsto \lambda\texttt{x} : \alpha.\ \lambda\texttt{y} : \alpha.\ \texttt{(<)}\ \alpha\ \texttt{dOrd x y}$$

$$(3)\qquad E_0 \oplus AE \overset{\text{over-exp}}{\vdash} \texttt{\textbackslash\ x y -> x < y} : \langle\texttt{Ord}\ \alpha\rangle \Rightarrow \alpha \to \alpha \to \mathbf{Bool} \leadsto \lambda\ \texttt{dOrd} : (tran\ E_0\ (\text{Ord}\ \alpha)) .\ \lambda\ \texttt{x} : \alpha .\ \lambda\texttt{y} : \alpha .\ \texttt{(<)}\ \alpha\ \texttt{dOrd x y}$$

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

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

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

Fig. 8. Rules for expressions, part 2.

$$(4)\qquad E_0 \overset{\text{poly-exp}}{\vdash} \texttt{\\ x y -> x < y} \ : \ \forall \alpha .\ \langle \texttt{Ord}\ \alpha \rangle \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathbf{Bool}$$
$$\leadsto$$
$$\Lambda\ \alpha\ .\ \ \lambda\ \texttt{dOrd}\ :\ (tran\ E_0\ (\text{Ord}\ \alpha))\ .$$
$$\lambda\ \texttt{x}\ :\ \alpha\ .\ \ \lambda \texttt{y}\ :\ \alpha\ .\ \ \texttt{(<)}\ \alpha\ \texttt{dOrd x y}$$

Judgment (1) yields (2) via ABS; judgment (2) yields (3) via PRED; and judgment (3) yields (4) via GEN. The *tran* function is defined in Section 4.7.

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 and 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 9. A judgment 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 judgments 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 judgment.

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

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

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

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

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

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

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

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

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

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

Fig. 9. Rules for dictionaries.

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

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 judgments hold:

(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) $E_0 \overset{\text{dict}}{\vdash} \mathbf{Eq}\ (\mathbf{List\ Int}) \leadsto \texttt{dictEqList}\ \mathbf{Int}\ \texttt{dictEqInt}$

$$\boxed{E \overset{\text{sigs}}{\vdash} \mathit{sigs} \rightsquigarrow \mathbf{sigs}}$$

$$\text{SIGS}\ \frac{E \overset{\text{type}}{\vdash} \tau_i \qquad (1 \leq i \leq m)}{E \overset{\text{sigs}}{\vdash} \langle \mathit{var}_1 : \tau_1, \ldots, \mathit{var}_m : \tau_m \rangle \rightsquigarrow \langle \mathit{var}_1, \ldots, \mathit{var}_m \rangle}$$

Fig. 10. Rule for class signatures.

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

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

Fig. 11. Rule for class declarations.

Judgment (1) holds via DICT-TAUT-IE; judgment (2) follows from (1) via DICT-SPEC; judgment (3) holds via DICT-TAUT-IE; and judgment (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.

4.4 Class Declarations

The rule for class declarations is given in Figure 11. Although the rule looks formidable, its workings are straightforward.

A judgment of the form

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

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

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

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

Fig. 12. Rule for instance declarations.

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

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

Fig. 13. Rule for instance bindings.

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 12. Again the rule looks formidable, and again its workings are straightforward.

A judgment of the form

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

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**

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

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

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

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

$$\boxed{PE \overset{\text{program}}{\vdash} \mathit{program} : \tau \rightsquigarrow \mathbf{exp}}$$

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

Fig. 14. Rules for declaration sequences and programs.

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 14 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.

The order of the instance declarations is, however, irrelevant, because all instance declarations may be mutually recursive. Mutual recursion of polymorphic functions does not 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 *instdecls* 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.

4.7 Converting Contexts/Records to Monotypes

Given an environment E, context and record types can be converted into monotypes using the function *tran*, defined below.

$$\begin{aligned}
\mathit{tran}\ E\ (\phi) &= \langle \mathit{tran}\ E\ (\kappa_1\ \tau_1), \ldots, \mathit{tran}\ E\ (\kappa_n\ \tau_n)\rangle \\
&\quad \textbf{where}\ \ \phi\ =\ \langle \kappa_1\ \tau_1, \ldots, \kappa_n\ \tau_n\rangle \\
\mathit{tran}\ E\ (\kappa\ \tau) &= \langle \mathit{tran}\ E\ (\theta\ [\tau/\alpha]),\ \mathit{tran}\ E\ (\gamma\ [\tau/\alpha])\rangle \\
&\quad \textbf{where}\ \ (\mathit{CE}\ \mathit{of}\ E)\ \kappa\ =\ (\texttt{class}\ \theta\ \Rightarrow\ \kappa\ \alpha\ \texttt{where}\ \gamma) \\
\mathit{tran}\ E\ (\gamma) &= \langle \tau_1, \ldots, \tau_n\rangle\ \ \textbf{where}\ \ \gamma\ =\ \langle v_1 : \tau_1, \ldots, v_n : \tau_n\rangle
\end{aligned}$$

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:

$$\begin{aligned}
&\text{search}\ = \Lambda\alpha.\quad \lambda\,\text{dOrd} : (\mathit{tran}\ E_0\ \langle \textbf{Ord}\ \alpha\rangle)\ .\quad \lambda \text{x} : \alpha.\quad \lambda\text{ys} : [\alpha]. \\
&\quad \text{not (null}\ \alpha\ \text{ys)}\ \&\&\ (\ (==)\ \alpha\ (\text{getOrdFromEq}\ \alpha\ \text{dOrd})\ \text{x}\ (\text{head}\ \alpha\ \text{ys})\ ||\\
&\qquad\qquad\qquad\qquad (\ (<)\ \alpha\ \text{dOrd x}\ (\text{head}\ \alpha\ \text{ys})\ \&\&\ \text{search}\ \alpha\ \text{dOrd x}\ (\text{tail}\ \alpha\ \text{ys})))
\end{aligned}$$

The translation of the dictionary, *tran* E_0 $\langle \textbf{Ord}\ \alpha\rangle$ is:

$$\begin{aligned}
&\mathit{tran}\ E_0\ \langle \textbf{Ord}\ \alpha\rangle\ \ = \\
&\quad \langle\ (\ \ \langle\ \ (\ \langle\rangle, \langle \alpha \to \alpha \to \textbf{Bool}\rangle\ ),\ \ \langle \alpha \to \alpha \to \textbf{Bool}, \alpha \to \alpha \to \textbf{Bool}\rangle\ \ )\ \rangle
\end{aligned}$$

5. SOUNDNESS, COMPLETENESS, AND PRINCIPAL TYPING RESULTS

We do not attempt to prove soundness and completeness results here, but it is a useful exercise to state the form of these results and to sketch their proofs. Equivalent results have been proved by Blott [1991] and Jones [1992b] for their respective type systems, which are broadly similar to that presented here. It is thus expected that their proofs will carry over in large part to this system.

5.1 The Type System

Some results can be obtained directly from the type system. The most important of these is that the type system has principal types. This relies on definitions of generality for polymorphic types and on the notion of a canonical form for contexts.

5.1.1 *Canonical Contexts.* There is a canonical form for contexts and saturated contexts, which is unique up to the ordering of $\kappa\ \tau$ pairs.

Definition 1. A *canonical context*, $\langle \kappa_1\ \tau_1, \ldots, \kappa_n\ \tau_n \rangle$ is one where

(1) all $\kappa\ \tau$ pairs have the form $\kappa\ \alpha$;

(2) no class κ appearing in a $\kappa\ \alpha$ constraint is a *superclass* of a class κ' appearing in another constraint $\kappa'\ \alpha$ on the same type variable α; and

(3) there are no duplicate $\kappa\ \alpha$ pairs.

Definition 2. A *superclass* of some class κ is

(1) any class appearing in the context of the class definition for κ; or

(2) a *superclass* of any such class.

The canonical form of a context is produced by the *canon* function. This transforms a saturated context ϕ into a minimal context θ by:

—generating the transitive closure of $\kappa\ \tau$ pairs under the defined instance hierarchy;

—removing all trivially satisified constraints $\kappa\ \tau$ where τ is not a type variable;

—removing all superclasses for each $\kappa\ \alpha$ pair; and

—finally eliminating all remaining duplicate constraints.

Note that the type rules (DICT-TAUT-IE in conjunction with REL and SPEC) ensure that for all $\kappa\ \tau$ constraints, τ is an instance of κ, so there is no possibility of failure here.

$$\mathit{canon}\ E\ \phi \quad = \quad \mathit{mincontext}\ E\ \phi' \quad \textbf{where} \quad (\mathit{dpat}, \phi') \quad = \quad \mathit{dicts}_{\mathit{simpl}}\ E\ \{\}\ \phi$$

The *canon* function requires two subsidiary functions: $\mathit{dicts}_{\mathit{simpl}}$ and *mincontext*. The $\mathit{dicts}_{\mathit{simpl}}$ function is used to generate the transitive closure of ϕ. Only the global instance environment component of E is actually used.

```
$\mathit{dicts}_{\mathit{simpl}}\ E\ \mathit{LIE}\ \langle \kappa_1\ \tau_1, \ldots, \kappa_n\ \tau_n \rangle \quad =$
    let $(\textbf{dpat}_1, \langle \kappa_{11}\ \tau_{11}, \ldots, \kappa_{1j}\ \tau_{1j} \rangle) \quad = \quad \mathit{dict}_{\mathit{cxt}}\ E\ \mathit{LIE}\ \kappa_1\ \tau_1$   in
    ...
    let $(\textbf{dpat}_n, \langle \kappa_{n1}\ \tau_{n1}, \ldots, \kappa_{nk}\ \tau_{nk} \rangle) \quad = \quad \mathit{dict}_{\mathit{cxt}}\ E\ \mathit{LIE}\ \kappa_n\ \tau_n$   in
    $(\ [\![\ (\textbf{dpat}_1, \ldots, \textbf{dpat}_n)\ ]\!], \langle \kappa_{11}\ \tau_{11}, \ldots, \kappa_{1j}\ \tau_{1j}, \ldots, \kappa_{n1}\ \tau_{n1}, \ldots, \kappa_{nk}\ \tau_{nk} \rangle)$
```

```
$\mathit{dict}_{\mathit{cxt}}\ E\ \mathit{LIE}\ \kappa\ \alpha \quad =$
    let $\textbf{dvar} \quad = \quad \mathit{lookup}_{\mathit{LIE}}\ \kappa\ \alpha$   in
    $(\ [\![\ \textbf{dvar}\ ]\!], \langle \kappa\ \alpha \rangle)$
```

$$
\begin{array}{lll}
\mathit{dict}_{cxt}\ E\ \mathit{LIE}\ \kappa\ (\chi\ \tau_1\ \ldots\ \tau_n) & = & \\
\quad \texttt{let}\ (\mathbf{dvar}, \forall\ \alpha_1 \ldots \alpha_n\ .\ \theta\ \Rightarrow\ \tau) & = & \mathit{lookup}_{IE}\ (\mathit{IE\ of\ E})\ \kappa\ \chi \qquad \texttt{in} \\
\quad \texttt{let}\ (\mathbf{dpats}, \phi) & = & \mathit{dicts}_{simpl}\ E\ (\theta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]) \quad \texttt{in} \\
\quad (\ [\![\ \texttt{(}\ \mathbf{dvar\ dpats}\ \texttt{)}\ ]\!], \phi) & &
\end{array}
$$

The *mincontext* function "minimizes" a saturated context by eliminating duplicate $\kappa\ \alpha$ pairs and all $\kappa\ \tau$ pairs.

$$
\begin{array}{c}
\mathit{mincontext}\ E\ \phi\ =\ \langle \kappa_1\ \alpha_1, \ldots, \kappa_n\ \alpha_n \rangle\ \text{ such that }\ \kappa_i\ \alpha_i\ \in\ \phi\ \text{ and} \\
\mathit{subclasses}\ E\ \kappa_i\ \cap\ \phi\ =\ \emptyset\ \text{ for all } i \text{ with } 1 \leq i \leq n \quad \text{and} \\
\kappa_i\ \alpha_i \neq \kappa_j\ \alpha_j \quad \text{for all } i, j \text{ with } 1 \leq i \leq j \leq n, i \neq j
\end{array}
$$

$$
\begin{array}{l}
\mathit{subclasses}\ E\ \kappa\ = \\
\quad \{\ \kappa' \text{ such that } (\kappa' : \langle \ldots, \kappa\ \alpha, \ldots \rangle\ \Rightarrow\ \kappa'\ \alpha\ \mathbf{where}\ \gamma)\ \in\ (\mathit{CE\ of\ E})\ \}
\end{array}
$$

5.1.2 *Generality of Types.* In order to present the principal type result, it is necessary to introduce an ordering relation on types, $\geq$, such that $\sigma \geq \sigma'$ means σ is no less general than σ'.

We first introduce *substitutions*, idempotent environments mapping type variables to types, with the usual conventions. Thus if S is $\{\ \alpha\ :\ \tau\ \}$, then, for example:

$$
\begin{array}{rlll}
 & S\ \ (\chi\ \alpha\ \texttt{Int}\ \alpha) & = \chi\ \tau\ \texttt{Int}\ \tau & \\
\text{and} & S\ (S\ \alpha) & = S\ \alpha & = \tau
\end{array}
$$

A saturated context ϕ is at least as general as another saturated context ϕ' if it is entirely contained within the canonical form of that context. We use the notation $\phi\ \subseteq\ \phi'$ to mean that ϕ contains no duplicate entries, and every entry in ϕ also appears in ϕ'. Thus, a smaller context is more general than a larger one, and the empty context is most general of all.

$$\phi\ \geq\ \phi' \quad \text{if}\ \ \phi\ \subseteq\ \mathit{canon}\ E\ (\phi')$$

(Note that generality for saturated contexts depends on the prevailing instance environment, IE.)

Now it is possible to define a generality relation $\geq$ over polymorphic types,

$$\forall\ \alpha_1\ \ldots\ \alpha_n\ .\ \phi\ \Rightarrow\ \tau\ \ \geq\ \ \forall\ \beta_1\ \ldots\ \beta_n\ .\ \phi'\ \Rightarrow\ \tau'$$

if there exists a substitution S such that $S\ \tau\ =\ \tau'$ and $(S\ \phi)\ \geq\ \phi'$.

5.1.3 *Principal Types.* A type τ is the *principal type* of p under PE if, and only if,

(1) $PE\ \overset{\text{program}}{\vdash}\ p\ :\ \tau\ \leadsto\ \mathbf{exp}$; and

(2) if, for some τ', $PE\ \overset{\text{program}}{\vdash}\ p\ :\ \tau'\ \leadsto\ \mathbf{exp}$, then $\tau \geq \tau'$.

Definition 3. A typing $PE\ \overset{\text{program}}{\vdash}\ p\ :\ \tau\ \leadsto\ \mathbf{exp}$ is a *principal typing* for program p if whenever $PE\ \overset{\text{program}}{\vdash}\ p\ :\ \tau'\ \leadsto\ \mathbf{exp}$, we have $\tau \geq \tau'$.

Definition 4. A typing $E\ \overset{\text{poly-exp}}{\vdash}\ \mathit{exp}\ :\ \sigma\ \leadsto\ \mathbf{exp}$ is a *principal typing* for expression exp if whenever $E\ \overset{\text{poly-exp}}{\vdash}\ \mathit{exp}\ :\ \sigma'\ \leadsto\ \mathbf{exp}$, we have $\sigma \geq \sigma'$.

$$\boxed{E \overset{\Lambda}{\vdash} \mathbf{exp} : \delta}$$

$$\boxed{E \overset{\Lambda\text{-bind}}{\vdash} \mathbf{exp} : VE}$$

$$\Lambda\text{-PROG} \quad \frac{\begin{array}{c} E \oplus VE \overset{\Lambda\text{-bind}}{\vdash} \mathbf{bindset} : VE \\ E \oplus VE \overset{\Lambda}{\vdash} \mathbf{exp} : \tau \end{array}}{E \overset{\Lambda}{\vdash} \texttt{letrec}\ \mathbf{bindset}\ \texttt{in}\ \mathbf{exp} : \tau}$$

$$\Lambda\text{-BINDS} \quad \frac{E \overset{\Lambda}{\vdash} \mathbf{exp}_i : \tau_i \qquad (1 \le i \le n)}{E \overset{\Lambda\text{-bind}}{\vdash} \mathbf{var}_1 = \mathbf{exp}_1; \ldots; \mathbf{var}_n = \mathbf{exp}_n : \{\mathbf{var}_1 : \tau_1, \ldots, \mathbf{var}_n : \tau_n\}}$$

Fig. 15. Rules for type calculus Λ: Programs and bindings.

Statement (Principal Types). *If there exists a typing for program p,* $PE \overset{program}{\vdash} p : \tau \leadsto \mathbf{exp}$ *then there exists a principal typing for program p,* $PE \overset{program}{\vdash} p : \tau' \leadsto \mathbf{exp}$.

Lemma (Principal Types for Expressions). *If there exists a typing for expression exp,* $E \overset{poly\text{-}exp}{\vdash} exp : \sigma \leadsto \mathbf{exp}$ *then there exists a principal typing for expression exp,* $E \overset{poly\text{-}exp}{\vdash} exp : \sigma' \leadsto \mathbf{exp}$.

Proof Sketch. The lemma can be proved by structural induction on programs and expressions. □

5.2 The Translation

As noted above, since the translation defines the semantics of the type system a simple soundness result is not meaningful. It is, however, possible to prove the soundness of the translation with respect to the types derived by the type system.

To prove these results, we need to refer to the type calculus for the second-order polymorphic lambda calculus, Λ, on the target language defined in Figure 3. A judgment of the form

$$PE \overset{\Lambda}{\vdash} \mathbf{exp} : \delta$$

holds if in environment PE the type δ is valid. Note that, unlike Reynolds' second-order polymorphic lambda calculus, but in common with the Hindley-Milner type system, type system Λ only permits type variables to be quantified at the outermost level of a type. The type structure for second-order types differs from that used for first-order polymorphic types in that:

(1) there are no overloaded second-order types;

(2) simple types include record types, which describe the types of dictionaries.

The new syntax for second-order types is shown in Figure 18.

$$\boxed{E \overset{\Lambda}{\vdash} \mathbf{exp} \ : \ \delta}$$

$$\Lambda\text{-TAUT} \quad \frac{(VE\ \mathit{of}\ E)\ \mathbf{var} = \delta}{E \overset{\Lambda}{\vdash} \mathbf{var} \ : \ \delta}$$

$$\Lambda\text{-ABS} \quad \frac{\begin{array}{c} E \overset{\Lambda\text{-pat}}{\vdash} \mathbf{pat} \ : \ VE \\ E \oplus_{VE} VE \overset{\Lambda}{\vdash} \mathbf{exp} \ : \ \tau \end{array}}{E \overset{\Lambda}{\vdash} \lambda\, \mathbf{pat}\ .\ \mathbf{exp} \ : \ \tau}$$

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

$$\Lambda\text{-LET} \quad \frac{\begin{array}{c} E \overset{\Lambda}{\vdash} \mathbf{exp}' \ : \ \delta \\ E \overset{\rightarrow}{\oplus} \{\mathbf{var} : \delta\} \overset{\Lambda}{\vdash} \mathbf{exp} \ : \ \tau \end{array}}{E \overset{\Lambda}{\vdash} \mathtt{let}\ \mathbf{var} \ = \ \mathbf{exp}'\ \mathtt{in}\ \mathbf{exp} \ : \ \tau}$$

$$\Lambda\text{-RECORD} \quad \frac{E \overset{\Lambda}{\vdash} \mathbf{exp}_i \ : \ \tau_i \qquad\qquad\qquad (1 \leq i \leq n)}{E \overset{\Lambda}{\vdash} \langle \mathbf{exp}_1, \ldots, \mathbf{exp}_n \rangle \ : \ \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\Lambda\text{-TYPE-ABS} \quad \frac{E \oplus_{AE} \{\alpha_1, \ldots, \alpha_n\} \overset{\Lambda}{\vdash} \mathbf{exp} \ : \ \tau}{E \overset{\Lambda}{\vdash} \Lambda\ \alpha_1 \ldots \alpha_n\ .\ \mathbf{exp} \ : \ \forall\ \alpha_1 \ldots \alpha_n\ .\ \tau}$$

$$\Lambda\text{-TYPE-COMB} \quad \frac{E \overset{\Lambda}{\vdash} \mathbf{exp} \ : \ \forall\ \alpha_1 \ldots \alpha_n\ .\ \tau}{E \overset{\Lambda}{\vdash} \mathbf{exp}\ \tau_1 \ldots \tau_n \ : \ \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}$$

Fig. 16. Rules for type calculus Λ: Expressions.

Typing rules for expressions in our target language are given in Figures 15–17. Note that there are no equivalents to SPEC or GEN, which in our previous rules specialized universally quantified types or introduced new type variables. Instead, all type variables are explicitly quantified by expressions of the form $\Lambda\ \alpha_1 \ldots \alpha_n\ .\ \mathbf{exp}$ and substituted by expressions of the form $\mathbf{exp}\ \tau_1 \ldots \tau_n$. Since the second-order types cannot be overloaded, there are also no equivalents to REL or PRED.

$$\boxed{E \overset{\Lambda\text{-pat}}{\vdash} \mathbf{pat} \ : \ VE}$$

$$\Lambda\text{-PAT-VAR} \quad \frac{}{E \overset{\Lambda\text{-pat}}{\vdash} (\mathbf{var} : \tau) \ : \ \{\mathbf{var} : \tau\}}$$

$$\Lambda\text{-PAT-REC} \quad \frac{E \overset{\Lambda\text{-pat}}{\vdash} \mathbf{pat}_i \ : \ VE_i \qquad\qquad (1 \leq i \leq n)}{E \overset{\Lambda\text{-pat}}{\vdash} (\mathbf{pat}_1, \ldots, \mathbf{pat}_n) \ : \ VE_1 \oplus \ldots \oplus VE_n}$$

Fig. 17. Rules for type calculus Λ: Patterns.

$$\begin{array}{llll}
\text{Second-Order Polymorphic Type} & \delta & \to \ \forall\alpha_1 \ldots \alpha_n \ . \ \tau & (n \geq 0) \\
\text{Simple type} & \tau & \to \ \alpha & \\
 & & \mid \ \chi \ \tau_1 \ldots \tau_n & (n \geq 0, \ n = \text{arity}(\chi)) \\
 & & \mid \ \tau' \to \tau & \\
 & & \mid \ \langle \tau_1, \ldots, \tau_n \rangle & (n \geq 0)
\end{array}$$

Fig. 18. Syntax of second-order types.

5.2.1 *Type Soundness.* Given a typing in our type calculus, then the type of the translation can be directly related to the type given by that typing. To state this formally, we first need to introduce a property relating first-order polymorphic types to second-order types.

Definition 5. A polymorphic type σ has the second-order translation δ, written $\sigma^* \ = \ \delta$, as follows:

$$\begin{array}{lll}
(\ \forall\alpha_1 \ldots \alpha_n \ . \ \rho \ )^* & = \forall\alpha_1 \ldots \alpha_n \ . \ \tau', & \text{if} \ \ \rho^* \ = \ \tau' \\
(\ \langle\rangle \Rightarrow \tau \ )^* & = \tau & \\
(\ \langle \kappa_1 \ \tau_1, \ldots, \kappa_n \ \tau_n \rangle \Rightarrow \tau \ )^* & = \langle \mathit{dicttype} \ \kappa_1 \ \tau_1, \ldots, \mathit{dicttype} \ \kappa_n \ \tau_n \rangle \ \to \ \tau &
\end{array}$$

This uses a function to convert class names into dictionary types, *dicttype*.

$$\begin{array}{l}
\mathit{dicttype} \ \kappa \ \tau \quad = \quad \langle \tau_1, \ldots, \tau_n \rangle \ [\tau/\alpha] \\
\quad \mathbf{where} \quad [\![\ \texttt{class} \ \theta \ \Rightarrow \ \kappa \ \alpha \ \texttt{where} \ \langle v_1 : \tau_1, \ldots, v_n : \tau_n \rangle \ ]\!] \quad = \quad CE \ \kappa
\end{array}$$

STATEMENT (TYPE SOUNDNESS FOR PROGRAMS). *Given a typing*

$$PE \overset{\mathit{program}}{\vdash} \mathit{program} \ : \ \tau \ \leadsto \ \mathbf{exp},$$

then there exists a typing $PE \overset{\Lambda}{\vdash} \mathbf{exp} \ : \ \tau'$*. Furthermore* $\tau \ = \ \tau'$*.*

This result relies on the following lemma for expressions and similar lemmas for declaration bindings.

Lemma (Type Soundness for Expressions). *Given a typing* $E \overset{over\text{-}exp}{\vdash} exp : \sigma \leadsto \mathbf{exp}$ *then there exists a typing* $E \overset{\Lambda}{\vdash} \mathbf{exp} : \delta$. *Furthermore* $\sigma^* = \delta$.

Proof Sketch. The lemma can be proved by structural induction on the rules for programs and expressions and by reference to the typings for the translated terms. □

6. IMPLEMENTING A TYPE INFERENCE ALGORITHM

This section sketches the implementation of a type inference algorithm based on the rules above. For simplicity, we will follow the conventional practice of defining an explicit substitution-based algorithm (as described by Cardelli [1987], for example), though we note that, in practice, graph-based algorithms are often more efficient. Hammond [1991] describes a graph-based algorithm using monads [Wadler 1992] that is similar to the one we use in the Glasgow Haskell compiler.

When deriving an inference algorithm from type rules such as these, it is common practice to define a set of functions that each implement a single rule or related set of rules. The arguments to the function are those items that appear to the left of the turnstile in the rule, while the results of the function are those items that appear to the right of the turnstile. So, given a rule which gives the types of expressions that yield polymorphic types (of type σ), whose form is

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

then the arguments to the inference function are a type environment E and an expression *exp*, and the outputs of the function are a polytype σ and a translated expression **exp**.

Since our rules have been defined relationally, however, there are some places where this simple scheme is insufficient. We will now consider each of these cases below.

The first issue that must be addressed is how to resolve the circularity in the rules for instance declarations and programs.

$$\text{PROG}\quad \frac{\begin{array}{l}(1)\ \ldots\\ (2)\ PE \oplus DE \oplus IE \overset{\text{instdecls}}{\vdash} instdecls : IE \leadsto \mathbf{bindset}_I\\ (3)\ \ldots\end{array}}{PE \overset{\text{program}}{\vdash} \ldots}$$

Here the same instance environment, IE, that is returned from the *instdecls* rule is also injected into the environment that is passed to it as an argument. The solution we adopt is to simply use a fixpoint in the function that implements the PROGRAM rule. This uses laziness in an essential way.

Second, we need to decide how to relate dictionaries to types in the PRED rule.

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

The translated dictionary pattern, **dpat**, depends on the local instance environment LIE that is an input to the *dicts* rule. This therefore needs to be generated as a result of function that implements the *exp* rule.

The final issue is how to implement the inference rules for dictionaries, *dicts*:

$$E \overset{\text{dicts}}{\vdash} \phi \rightsquigarrow \mathbf{dexps}$$

These are somewhat problematic because they are used in three distinct ways throughout the rules. We therefore need to provide three different functions that each implement one of these three uses.

The rules are used to produce

(1) a dictionary pattern and a local instance environment from an environment and a context in INST and CLASS

$$\begin{array}{lrcl}
\text{(INST)} & E \oplus LIE \overset{\text{dicts}}{\vdash} \theta \rightsquigarrow \mathbf{dpat} & & \theta \text{ determines } LIE \\
\text{(CLASS)} & PE \oplus AE \oplus LIE \overset{\text{dicts}}{\vdash} \theta \rightsquigarrow \mathbf{dpat} & & \theta \text{ determines } LIE
\end{array}$$

in which case, the *dicts* function takes an environment E and a context θ as its arguments and produces a translated dictionary pattern **dpat** and a local instance environment LIE as its results;

(2) the corresponding dictionary expression from the augmented environment and a context in INST and REL

$$\begin{array}{lrll}
\text{(INST)} & PE \oplus AE \oplus LIE \overset{\text{dicts}}{\vdash} & \theta'[\tau/\alpha] & \rightsquigarrow \mathbf{dexp} \\
\text{(REL)} & E \overset{\text{dicts}}{\vdash} & \phi & \rightsquigarrow \mathbf{dexps}
\end{array}$$

in which case, the *dicts* function takes an environment E and a saturated context ϕ as its arguments and produces a translated dictionary expression **dexp** as its result;

(3) or a context and the corresponding dictionary pattern in PRED

$$\begin{array}{lrcl}
\text{(PRED)} & E \oplus LIE \overset{\text{dicts}}{\vdash} \theta \rightsquigarrow \mathbf{dpat} & & \theta \text{ determines } LIE
\end{array}$$

in which case the *dicts* function takes an environment E and a local instance environment LIE as its arguments and produces a translated dictionary pattern **dpat** and a context θ as its results.

Notice in the third case, that despite the ordering in the determines relation, LIE is an argument to $dicts_{cxt}$, and θ is a result.

Only the instance environment components (both global and local) of the environment are used in each case. These rules use the global and local instance environments in the opposite way from their normal use as environments: matches are made on the co-domain of the environment rather than its domain.

6.1 Optimizations

There are several optimizations that can be applied to improve the translation described above [Augustsson 1993; Hammond and Blott 1989; Jones 1992c]. For example,

—Specialized versions could be produced for each overloaded function, so that each use of that function can be compiled without introducing dictionaries.

—Methods could be "inlined."

—Dictionaries could be partially applied in order to reduce the number of times that dictionary functions are applied at run-time.

—Parameterized dictionaries can be lifted from recursive overloaded definitions, so avoiding reconstructing them in each recursive call.

—The well-known *common sub-expression elimination* optimisation, which involves replacing two or more identical sub-expressions in an expression by a single shared sub-expression, can be used to avoid constructing identical dictionaries several times.

These optimizations have already been at least partially adopted by several of the more optimizing Haskell compilers, and there is evidence that significant performance improvements can result [Augustsson 1993].

7. CONCLUSIONS

The main contribution of this article is that it presents a minimal, readable set of inference rules to handle type classes in Haskell, derived from the full static semantics of Haskell [Peyton Jones and Wadler 1991]. We have treated most of the essential characteristics of Haskell, including a direct treatment of the type class hierarchy (this is a minor theoretical point, but it is a practical advantage to be able to express errors in terms of the classes the programmer has used). We have also briefly described how the type rules can be implemented. For simplicity, we have omitted discussions of the monomorphism restriction, ambiguity (default resolution), and default methods. These are dealt with at length in the full static semantics and elsewhere, e.g., Blott [1991] and Nipkow and Prehofer [1993].

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. It is also straightforward to extend the rules to more esoteric cases, such as classes constraining more than one overloaded type [Hammond 1993].

ACKNOWLEDGEMENTS

We would like to thank those who have read and commented on previous drafts of this article, including the anonymous referees.

REFERENCES

Augustsson, L. 1993. Implementing Haskell overloading. In *Proceedings of the 1993 Conference on Functional Programming and Computer Architecture* (Copenhagen, Denmark). ACM, New York, 65–73.

Blott, S. 1991. Type classes. Ph.D. thesis, Dept. of Computing Science, Glasgow Univ., Glasgow, Scotland.

Cardelli, L. 1987. Basic polymorphic typechecking. *Sci. Comput. Program. 8*, 147–172.

Chen, K. 1994. Semantics and coherence for parametric type classes. Res. Rep., YALEU/DCS/RR-1003, Dept. of Computer Science, Yale Univ., New Haven, Conn. June.

Chen, K. 1995. A parametric extension of Haskell's type classes. Ph. D. thesis, Dept. of Computer Science, Yale Univ., New Haven, Conn.

CHEN, K., HUDAK, P., AND ODERSKY, M. 1992. Parametric type classes. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming.* ACM, New York, 170–181.

CORMACK, G. V. AND WRIGHT, A. K. 1990. Type dependent parameter inference. In *Proceedings of the 1990 ACM Conference on Programming Language Design and Implementation* (White Plains, N.Y.), ACM. New York, 127–136.

GIRARD, J.-Y. 1972. Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure. Thèse de doctorat d'état, Université Paris VII, Paris, France.

HAMMOND, K. 1991. Efficient type inference using monads. In *1991 Glasgow Workshop on Functional Programming* (Ullapool, Scotland). Workshops in Computing Science, Springer-Verlag, Berlin, 146–157.

HAMMOND, K. 1993. Extended type classes, Int. Memo., Dept. of Computer Science, Glasgow Univ., Glasgow, Scotland.

HAMMOND, K. AND BLOTT, S. 1989. Implementing Haskell type classes. In *1989 Glasgow Workshop on Functional Programming* (Fraserburgh, Scotland). Workshops in Computing Science, Springer-Verlag, Berlin, 266–286.

HINDLEY, R. 1969. The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc. 146*, 29–60.

HUDAK, P., PEYTON JONES, S. L., AND WADLER, P.L., Eds., 1992. Report on the programming language Haskell, version 1.2. *ACM SIGPLAN Not. 27*, 5.

HUET, G., Ed.. 1990. *Logical Foundations of Functional Programming.* Addison-Wesley, Reading, Mass.

JONES, M. P. 1992a. A theory of qualified types. In *Proceedings of the 1992 European Symposium on Programming* (Rennes, France). Lecture Notes in Computer Science, vol. 582. Springer-Verlag, Berlin.

JONES, M.P. 1992b. A theory of qualified types. D.Phil. thesis, Programming Research Group, Oxford Univ., Oxford, England. Published by Oxford University Press, 1994.

JONES, M. P. 1992c. Efficient implementation of type class overloading. Int. Memo., Programming Research Group, Oxford Univ., Oxford, England.

JONES, M.P. 1993. Partial Evaluation for dictionary-free overloading. Res. Rep., YALEU/DCS/RR-959, Dept. of Computer Science, Yale Univ., New Haven, Conn. Apr.

JONES, M. P. 1994. Simplifying and improving qualified types. Res. Rep., YALEU/DCS/RR-1040, Dept. of Computer Science, Yale Univ., New Haven, Conn. June.

JONES, M. P. 1995. A system of constructor classes: Overloading and implicit higher-order polymorphism. *J. Func. Program. 5*, 1, 1–37.

KAES, S. 1988. Parametric polymorphism. In *Proceedings of the 1988 European Symposium on Programming* (Nancy, France). Lecture Notes in Computer Science, vol. 300. Springer-Verlag, Berlin.

LÄUFER, K. 1992. Polymorphic type inference and abstract data types. Ph.D. thesis, New York Univ., New York.

LÄUFER, K. 1993. An extension of Haskell with first-class abstract types. Tech. Rep., Loyola Univ. Chicago, Ill.

LÄUFER, K. AND ODERSKY, M. 1994. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst. 16*, 5, 1411–1430.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, Mass.

MILNER, R. AND TOFTE, M. 1991. *Commentary on Standard ML.* MIT Press, Cambridge, Mass.

NIPKOW, T. AND PREHOFER, C. 1993. Type checking type classes. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages.* ACM, New York, 409–418.

NIPKOW, T. AND SNELTING, G. 1991. Type classes and overloading resolution via order-sorted unification. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture* (Boston, Mass.). Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 1–14.

ODERSKY, M. AND LÄUFER, K. 1991. Type classes are signatures of abstract types. Tech. Rep., IBM T.J. Watson Research Center, Yorktown Heights, N.Y. May.

PEYTON JONES, S. L. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, N.J.

PEYTON JONES, S.L. AND WADLER, P.L. 1991. A static semantics for Haskell. Int. Memo., Dept. of Computing Science, Glasgow Univ., Glasgow, Scotland.

REYNOLDS, J. C. 1974. Towards a theory of type structure. In *Proceedings of the Colloque sur la Programmation.* Lecture Notes in Computer Science, vol. 19. Springer-Verlag, Berlin, 408–425.

REYNOLDS, J. C. 1985. Three approaches to type structure. In *Mathematical Foundations of Software Development.* Lecture Notes in Computer Science, vol. 185. Springer-Verlag, Berlin, 97–138.

ROUAIX, F. 1990. Safe run-time overloading. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages* (San Francisco, Calif.). ACM, New York, 355–366.

TURNER, D. A. 1985. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture* (Nancy, France). Lecture Notes in Computer Science, vol. 201. Springer-Verlag, Berlin, 1–16.

VOLPANO, D. M. AND SMITH, G. S. 1991. On the complexity of ML typability with overloading. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture* (Boston, Mass.). Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 15–28.

WADLER, P. L. 1992. The essence of functional programming. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages* (Albuquerque, N.M.). ACM, New York, 1–14.

WADLER, P. L. AND BLOTT, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages* (Austin, Tex.). ACM, New York, 60–76.

Received March 1994; accepted May 1995