Typing Haskell in Haskell∗

MARK P. JONES
Pacific Software Research Center
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
20000 NW Walker Road, Beaverton, OR 97006, USA
(*e-mail:* `mpj@cse.ogi.edu`)

Abstract

Haskell benefits from a sophisticated type system, but implementors, programmers, and researchers suffer because it has no formal description. To remedy this shortcoming, we present a Haskell program that implements a Haskell typechecker, thus providing a mathematically rigorous specification in a notation that is familiar to Haskell users. We expect this program to fill a serious gap in current descriptions of Haskell, both as a starting point for discussions about existing features of the type system, and as a platform from which to explore new proposals.

1 Introduction

Haskell[1] benefits from one of the most sophisticated type systems of any widely used programming language. Unfortunately, it also suffers because there is no formal specification of what the type system should be. As a result:

- It is hard for Haskell implementors to be sure that their systems accept the same programs as other implementations. The informal specification in the Haskell report (Peyton Jones & Hughes, 1999) leaves too much room for confusion and misinterpretation. This leads to genuine discrepancies between implementations, as subscribers to the Haskell mailing list will have seen.
- It is hard for Haskell programmers to understand the details of the type system and to appreciate why some programs are accepted when others are not. Formal presentations of most aspects of the type system are available, but they often abstract on specific features that are Haskell-like, but not Haskell-exact, and do not describe the complete type system. Moreover, these papers tend to use disparate and unfamiliar technical notation and concepts that may be difficult for some Haskell programmers to understand.

∗ An earlier version of this paper was presented at the Haskell workshop in Paris, France, on October 1, 2000. Both papers describe the same type system, but some significant parts of this version have been rewritten, restructured, or expanded to clarify the presentation.
[1] Throughout, we use ‘Haskell’ as an abbreviation for ‘Haskell 98’.

- It is hard for Haskell researchers to explore new type system extensions, or even to study usability issues that arise with the present type system such as the search for better type error diagnostics. Work in these areas requires a clear understanding of the type system and, ideally, a platform on which to build and experiment with prototype implementations. The existing Haskell implementations are not suitable for this (and were not intended to be): the nuts and bolts of a type system are easily obscured by the use of clever data structures and optimizations, or by the need to integrate smoothly with other parts of an implementation.

This paper presents a formal description of the Haskell type system using the notation of Haskell itself as a specification language. Indeed, the source code for this paper is itself an executable Haskell program that is passed through a custom preprocessor and then through LaTeX to obtain the typeset version. The type checker is available in source form on the Internet at `http://www.cse.ogi.edu/~mpj/thih/`. We hope that this will serve as a resource for the Haskell community, and that it will be a significant step in addressing the problems described previously.

One audience whose needs may not be particularly well met by this paper are researchers in programming language type systems who do not have experience of Haskell. (Of course, we encourage anyone in that position to learn more about Haskell!) Indeed, we do not follow the traditional route in such settings where the type system might first be presented in its purest form, and then related to a more concrete type inference algorithm by soundness and completeness theorems. Here, we deal only with type inference. It does not even make sense to ask if our algorithm computes 'principal' types: such a question requires a comparison between two different presentations of a type system, and we only have one. Nevertheless, we believe that our specification could be recast in a standard, type-theoretic manner and used to develop a presentation of Haskell typing in a more traditional style.

The code presented here can be executed with any Haskell system, but our primary goals have been clarity and simplicity, and the resulting code is not intended to be an efficient implementation of type inference. Indeed, in some places, our choice of representation may lead to significant overheads and duplicated computation. It would be interesting to try to derive a more efficient, but provably correct implementation from the specification given here. We have not attempted to do this because we expect that it would obscure the key ideas that we want to emphasize. It therefore remains as a topic for future work, and as a test to assess the applicability of program transformation and synthesis to modestly sized Haskell programs.

Another goal of this paper is to give as complete a description of the Haskell type system as possible, while also aiming for conciseness. For this to be possible, we have assumed that certain transformations and checks will have been made prior to typechecking, and hence that we can work with a much simpler abstract syntax than the full source-level syntax of Haskell would suggest. As we argue informally at various points in the paper, we do not believe that there would be any significant difficulty in extending our system to deal with the missing constructs. All of the fundamental components, including the thorniest aspects of Haskell typing, are addressed in the framework that we present here. Our specification does not attempt to deal with all of the issues that would occur in the implementation of a type checker in a full Haskell implementation. We do not tackle the problems of interfacing a typechecker with compiler front ends (to track source code locations in error diagnostics, for example) or back ends (to describe the implementation of overloading, for example), nor do we attempt to formalize any of the extensions that are implemented in current Haskell systems. This is one of things that makes our specification relatively concise (429 lines of Haskell code). By comparison, the core parts of the Hugs typechecker take some 90+ pages of C code.

Some examples are included in the paper to illustrate the datatypes and representations that are used. However, for reasons of space, the definitions of some constants that represent entities in the standard prelude, as well as the machinery that we use in testing to display the results of type inference, are included only in the electronic distribution, and not in the typeset version of the paper. Apart from those details, this paper gives the full source code.

We expect the program described here to evolve in at least three different ways.

- Formal specifications are not immune to error, and so it is possible that changes will be required to correct bugs in the code presented here. On the other hand, by writing our specification as a program that can be typechecked and executed with existing Haskell implementations, we have a powerful facility for detecting simple bugs automatically and for testing to expose deeper problems.
- As it stands, this paper just provides one more interpretation of the Haskell type system. We believe that it is consistent with the official specification, but because the latter is given only informally, we cannot prove the correctness of our program in a rigorous manner. Instead, we hope that our code, perhaps with some modifications, will eventually serve as a precise definition of the Haskell type system, capturing a consensus within the Haskell community. There is some evidence that this goal is already within reach: no discrepancies or technical changes have been discovered or reported in more than a year since the first version of this program was released.
- Many extensions of the Haskell type system have been proposed, and several of these have already been implemented in one or more of the available Haskell systems. Some of the better known examples of this include multiple-parameter type classes, existential types, rank-2 polymorphism, and extensible records. We would like to obtain formal descriptions for as many of these proposals as possible by extending the core specification presented here.

It will come as no surprise to learn that some knowledge of Haskell will be required to read this paper. That said, we have tried to keep the definitions and code as clear and simple as possible, and although we have made some use of Haskell overloading and do-notation, we have generally avoided using the more esoteric features of Haskell. In addition, some experience with the basics of Hindley-Milner style type inference (Hindley, 1969; Milner, 1978; Damas & Milner, 1982) will be needed to understand the algorithms presented here. Although we have aimed to keep our

Description	Symbol	Type
kind	$k, \ldots$	*Kind*
type constructor	$tc, \ldots$	*Tycon*
type variable	$v, \ldots$	*Tyvar*
– 'fixed'	$f, \ldots$	
– 'generic'	$g, \ldots$	
type	$t, \ldots$	*Type*
class	$c, \ldots$	*Class*
instance	$it, \ldots$	*Inst*
predicate	$p, q, \ldots$	*Pred*
– 'deferred'	$d, \ldots$	
– 'retained'	$r, \ldots$	
qualified type	$qt, \ldots$	*QualType*
class environment	$ce, \ldots$	*ClassEnv*
scheme	$sc, \ldots$	*Scheme*
substitution	$s, \ldots$	*Subst*
unifier	$u, \ldots$	*Subst*
assumption	$a, \ldots$	*Assump*
identifier	$i, \ldots$	*Id*
literal	$l, \ldots$	*Literal*
pattern	$pat, \ldots$	*Pat*
expression	$e, f, \ldots$	*Expr*
alternative	$alt, \ldots$	*Alt*
binding group	$bg, \ldots$	*BindGroup*

Fig. 1. Notational Conventions

presentation as simple as possible, some aspects of the problems that we are trying to address have inherent complexity or technical depth that cannot be side-stepped. In short, this paper will probably not be useful as a tutorial introduction to Hindley-Milner style type inference!

2 Preliminaries

For simplicity, we present the code for our typechecker as a single Haskell module. The program uses only a handful of standard prelude functions, like *map*, *concat*, *all*, *any*, *mapM*, etc., and a few operations from the *List* and *Monad* libraries:

```
module TypingHaskellInHaskell where
import List (nub, (\\), intersect, union, partition)
import Monad (msum)
```

For the most part, our choice of variable names follows the notational conventions set out in Figure 1. A trailing *s* on a variable name usually indicates a list. Numeric suffices or primes are used as further decoration where necessary. For example, we use k or k' for a kind, and ks or ks' for a list of kinds. The types and terms appearing in the table are described more fully in later sections. To distinguish the code for the typechecker from program fragments that are used to discuss its behavior, we typeset the former in an *italic* font, and the latter in a `typewriter` font.

Throughout this paper, we implement identifiers as strings, and assume that there is a simple way to generate identifiers from integers using the *enumId* function:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Id} & = & \mathit{String} \\ \\ \mathit{enumId} & :: & \mathit{Int} \rightarrow \mathit{Id} \\ \mathit{enumId}\ n & = & \text{“v”} +\!\!+\ \mathit{show}\ n \end{array}$$

The *enumId* function will be used in the definition of the *newTVar* operator in Section 10 to describe the allocation of fresh type variables during type inference. With the simple implementation shown here, we assume that variable names beginning with "`v`" do not appear in input programs.

3 Kinds

To ensure that they are valid, Haskell type constructors are classified into different *kinds*: the kind $*$ (pronounced 'star') represents the set of all simple (i.e., nullary) type expressions, like *Int* and $\mathit{Char} \rightarrow \mathit{Bool}$; kinds of the form $k_1 \rightarrow k_2$ represent type constructors that take an argument type of kind k_1 to a result type of kind k_2. For example, the standard list, *Maybe* and *IO* constructors all have kind $* \rightarrow *$. Here, we will represent kinds as values of the following datatype:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Kind} & = & \mathit{Star} \mid \mathit{Kfun}\ \mathit{Kind}\ \mathit{Kind} \\ & & \textbf{deriving}\ \mathit{Eq} \end{array}$$

Kinds play essentially the same role for type constructors as types do for values, but the kind system is clearly very primitive. There are a number of extensions that would make interesting topics for future research, including polymorphic kinds, subkinding, and record/product kinds. A simple extension of the kind system—adding a new *row* kind—has already proved to be useful for the Trex implementation of extensible records in Hugs (Gaster & Jones, 1996; Jones & Peterson, 1999).

4 Types

The next step is to define a representation for types. Stripping away syntactic sugar, Haskell type expressions are either type variables or constants (each of which has an associated kind), or applications of one type to another: applying a type of kind $k_1 \rightarrow k_2$ to a type of kind k_1 produces a type of kind k_2:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Type} & = & \mathit{TVar}\ \mathit{Tyvar} \mid \mathit{TCon}\ \mathit{Tycon} \mid \mathit{TAp}\ \mathit{Type}\ \mathit{Type} \mid \mathit{TGen}\ \mathit{Int} \\ & & \textbf{deriving}\ \mathit{Eq} \\ \\ \textbf{data}\ \mathit{Tyvar} & = & \mathit{Tyvar}\ \mathit{Id}\ \mathit{Kind} \\ & & \textbf{deriving}\ \mathit{Eq} \end{array}$$

```
data Tycon = Tycon Id Kind
             deriving Eq
```

This definition also includes types of the form *TGen n*, which represent 'generic' or quantified type variables. The only place where *TGen* values are used is in the representation of type schemes, which will be described in Section 8.

The following examples show how standard primitive datatypes are represented as type constants:

```
tUnit    = TCon (Tycon "()" Star)
tChar    = TCon (Tycon "Char" Star)
tInt     = TCon (Tycon "Int" Star)
tInteger = TCon (Tycon "Integer" Star)
tFloat   = TCon (Tycon "Float" Star)
tDouble  = TCon (Tycon "Double" Star)

tList    = TCon (Tycon "[]" (Kfun Star Star))
tArrow   = TCon (Tycon "(->)" (Kfun Star (Kfun Star Star)))
tTuple2  = TCon (Tycon "(,)" (Kfun Star (Kfun Star Star)))
```

A full Haskell compiler or interpreter might store additional information with each type constant—such as the the list of constructor functions for an algebraic datatype—but such details are not needed during typechecking.

More complex types are built up from constants and variables using the *TAp* constructor. For example, the representation for the type $Int \rightarrow [a]$ is as follows:

```
TAp (TAp tArrow tInt) (TAp tList (TVar (Tyvar "a" Star)))
```

We do not provide a representation for type synonyms, assuming instead that they have been fully expanded before typechecking. For example, the *String* type—a synonym for [*Char*]—is represented as:

```
tString :: Type
tString  = list tChar
```

It is always possible for an implementation to expand synonyms in this way because Haskell prevents the use of a synonym without its full complement of arguments. Moreover, the process is guaranteed to terminate because recursive synonym definitions are prohibited. In practice, however, implementations are likely to expand synonyms more lazily: in some cases, type error diagnostics may be easier to understand if they display synonyms rather than expansions.

We end this section with the definition of a few helper functions. The first three provide simple ways to construct function, list, and pair types, respectively:

```
infixr    4 `fn`
fn         :: Type -> Type -> Type
a `fn` b    = TAp (TAp tArrow a) b

list       :: Type -> Type
list t      = TAp tList t
```

$$\begin{array}{lcl} \mathit{pair} & :: & \mathit{Type} \to \mathit{Type} \to \mathit{Type} \\ \mathit{pair}\ a\ b & = & \mathit{TAp}\ (\mathit{TAp}\ \mathit{tTuple2}\ a)\ b \end{array}$$

We also define an overloaded function, *kind*, that can be used to determine the kind of a type variable, type constant, or type expression:

$$\begin{array}{l} \textbf{class}\ \mathit{HasKind}\ t\ \textbf{where} \\ \quad \mathit{kind}\ ::\ t \to \mathit{Kind} \\ \textbf{instance}\ \mathit{HasKind}\ \mathit{Tyvar}\ \textbf{where} \\ \quad \mathit{kind}\ (\mathit{Tyvar}\ v\ k) \ = \ k \\ \textbf{instance}\ \mathit{HasKind}\ \mathit{Tycon}\ \textbf{where} \\ \quad \mathit{kind}\ (\mathit{Tycon}\ v\ k) \ = \ k \\ \textbf{instance}\ \mathit{HasKind}\ \mathit{Type}\ \textbf{where} \\ \quad \mathit{kind}\ (\mathit{TCon}\ tc) \ = \ \mathit{kind}\ tc \\ \quad \mathit{kind}\ (\mathit{TVar}\ u) \ = \ \mathit{kind}\ u \\ \quad \mathit{kind}\ (\mathit{TAp}\ t\ \_) \ = \ \textbf{case}\ (\mathit{kind}\ t)\ \textbf{of} \\ \qquad\qquad\qquad\qquad (\mathit{Kfun}\ \_\ k) \ \to \ k \end{array}$$

Most of the cases here are straightforward. Notice, however, that we can calculate the kind of an application $(\mathit{TAp}\ t\ t')$ using only the kind of its first argument t: Assuming that the type is well-formed, t must have a kind $k' \to k$, where k' is the kind of t' and k is the kind of the whole application. This shows that we need only traverse the leftmost spine of a type expression to calculate its kind.

5 Substitutions

Substitutions—finite functions, mapping type variables to types—play a major role in type inference. In this paper, we represent substitutions using association lists:

$$\textbf{type}\ \mathit{Subst} \ = \ [(\mathit{Tyvar},\ \mathit{Type})]$$

To ensure that we work only with well-formed type expressions, we will be careful to construct only *kind-preserving* substitutions in which variables are mapped only to types of the same kind. The simplest substitution is the null substitution, represented by the empty list, which is obviously kind-preserving:

$$\begin{array}{lcl} \mathit{nullSubst} & :: & \mathit{Subst} \\ \mathit{nullSubst} & = & [\,] \end{array}$$

Almost as simple are the substitutions $(u \mapsto t)$[2] that map a single variable u to a type t of the same kind:

$$\begin{array}{lcl} (\mapsto) & :: & \mathit{Tyvar} \to \mathit{Type} \to \mathit{Subst} \\ u \mapsto t & = & [(u,\ t)] \end{array}$$

This is kind-preserving if, and only if, $\mathit{kind}\ u = \mathit{kind}\ t$.

Substitutions can be applied to types—and, in fact, to any other value with type

[2] The ‘maps to’ symbol $\mapsto$ is written as `+->` in the concrete syntax of Haskell.

components—in a natural way. This suggests that we overload the operation to *apply* a substitution so that it can work on different types of object:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Types}\ t\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{apply} & :: & \mathit{Subst} \to t \to t \\
\mathit{tv} & :: & t \to [\mathit{Tyvar}]
\end{array}
\end{array}
$$

In each case, the purpose of applying a substitution is the same: To replace every occurrence of a type variable in the domain of the substitution with the corresponding type. We also include a function *tv* that returns the set of type variables (i.e., *Tyvar*s) appearing in its argument, listed in order of first occurrence (from left to right), with no duplicates. The definitions of these operations for *Type* are as follows:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Types}\ \mathit{Type}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{apply}\ s\ (\mathit{TVar}\ u) & = & \textbf{case}\ \mathit{lookup}\ u\ s\ \textbf{of} \\
 & & \quad \begin{array}{lcl} \mathit{Just}\ t & \to & t \\ \mathit{Nothing} & \to & \mathit{TVar}\ u \end{array} \\
\mathit{apply}\ s\ (\mathit{TAp}\ l\ r) & = & \mathit{TAp}\ (\mathit{apply}\ s\ l)\ (\mathit{apply}\ s\ r) \\
\mathit{apply}\ s\ t & = & t \\
 & & \\
\mathit{tv}\ (\mathit{TVar}\ u) & = & [u] \\
\mathit{tv}\ (\mathit{TAp}\ l\ r) & = & \mathit{tv}\ l\ \text{`}\mathit{union}\text{`}\ \mathit{tv}\ r \\
\mathit{tv}\ t & = & [\,]
\end{array}
\end{array}
$$

It is straightforward (and useful!) to extend these operations to work on lists:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Types}\ a \Rightarrow \mathit{Types}\ [a]\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{apply}\ s & = & \mathit{map}\ (\mathit{apply}\ s) \\
\mathit{tv} & = & \mathit{nub}\ .\ \mathit{concat}\ .\ \mathit{map}\ \mathit{tv}
\end{array}
\end{array}
$$

The *apply* function can be used to build more complex substitutions. For example, composition of substitutions, satisfying $\mathit{apply}\ (s_1 @@ s_2) = \mathit{apply}\ s_1\ .\ \mathit{apply}\ s_2$, can be defined using:

$$
\begin{array}{lcl}
\textbf{infixr} & 4 & @@ \\
(@@) & :: & \mathit{Subst} \to \mathit{Subst} \to \mathit{Subst} \\
s_1 @@ s_2 & = & [(u,\ \mathit{apply}\ s_1\ t) \mid (u,\ t) \leftarrow s_2] +\!\!+\ s_1
\end{array}
$$

We can also form a 'parallel' composition $s_1 +\!\!+\ s_2$ of two substitutions s_1 and s_2, but the result is left-biased because bindings in s_1 take precedence over any bindings for the same variables in s_2. For a more symmetric version of this operation, we use a *merge* function, which checks that the two substitutions agree at every variable in the domain of both and hence guarantees that $\mathit{apply}\ (s_1 +\!\!+\ s_2) = \mathit{apply}\ (s_2 +\!\!+\ s_1)$. Clearly, this is a partial function, which we reflect by arranging for *merge* to return its result in a monad, using the standard *fail* function to provide a string diagnostic in cases where the function is undefined.

$$
\begin{array}{lcl}
\mathit{merge} & :: & \mathit{Monad}\ m \Rightarrow \mathit{Subst} \rightarrow \mathit{Subst} \rightarrow m\ \mathit{Subst} \\
\mathit{merge}\ s_1\ s_2 & = & \textbf{if}\ \mathit{agree}\ \textbf{then}\ \mathit{return}\ (s_1 +\!\!+\, s_2)\ \textbf{else}\ \mathit{fail}\ \text{“}\texttt{merge fails}\text{”} \\
\quad \textbf{where}\ \mathit{agree} & = & \mathit{all}\ (\backslash v \rightarrow \mathit{apply}\ s_1\ (\mathit{TVar}\ v) == \mathit{apply}\ s_2\ (\mathit{TVar}\ v)) \\
 & & \qquad (\mathit{map}\ \mathit{fst}\ s_1\ \text{‘}\mathit{intersect}\text{‘}\ \mathit{map}\ \mathit{fst}\ s_2)
\end{array}
$$

It is easy to check that both (@@) and *merge* produce kind-preserving results from kind-preserving arguments. In the next section, we will see how the first of these composition operators is used to describe unification, while the second is used in the formulation of a matching operation.

6 Unification and Matching

The goal of unification is to find a substitution that makes two types equal—for example, to ensure that the domain type of a function matches up with the type of an argument value. However, it is also important for unification to find as 'small' a substitution as possible because that will lead to most general types. More formally, a substitution s is a *unifier* of two types t_1 and t_2 if $\mathit{apply}\ s\ t_1 = \mathit{apply}\ s\ t_2$. A *most general unifier*, or *mgu*, of two such types is a unifier u with the property that any other unifier s can be written as $s'@@u$, for some substitution s'.

The syntax of Haskell types has been chosen to ensure that, if two types have any unifying substitutions, then they have a most general unifier, which can be calculated by a simple variant of Robinson's algorithm (Robinson, 1965). One of the reasons for this is that there are no non-trivial equalities on types. Extending the type system with higher-order features (such as lambda expressions on types), or with other mechanisms that allow reductions or rewriting in the type language, could make unification undecidable, non-unitary (meaning that there may not be most general unifiers), or both. This, for example, is why Haskell does not allow type synonyms to be partially applied (and interpreted as some restricted kind of lambda expression).

The calculation of most general unifiers is implemented by a pair of functions:

$$
\begin{array}{lcl}
\mathit{mgu} & :: & \mathit{Monad}\ m \Rightarrow \mathit{Type} \rightarrow \mathit{Type} \rightarrow m\ \mathit{Subst} \\
\mathit{varBind} & :: & \mathit{Monad}\ m \Rightarrow \mathit{Tyvar} \rightarrow \mathit{Type} \rightarrow m\ \mathit{Subst}
\end{array}
$$

These functions return results in a monad, capturing the fact that unification is a partial function. The main algorithm is described by *mgu*, using the structure of its arguments to guide the calculation:

$$
\begin{array}{lcl}
\mathit{mgu}\ (\mathit{TAp}\ l\ r)\ (\mathit{TAp}\ l'\ r') & = & \textbf{do}\ s_1 \leftarrow \mathit{mgu}\ l\ l' \\
 & & \quad\ \ s_2 \leftarrow \mathit{mgu}\ (\mathit{apply}\ s_1\ r)\ (\mathit{apply}\ s_1\ r') \\
 & & \quad\ \ \mathit{return}\ (s_2@@s_1) \\
\mathit{mgu}\ (\mathit{TVar}\ u)\ t & = & \mathit{varBind}\ u\ t \\
\mathit{mgu}\ t\ (\mathit{TVar}\ u) & = & \mathit{varBind}\ u\ t \\
\mathit{mgu}\ (\mathit{TCon}\ tc_1)\ (\mathit{TCon}\ tc_2) & & \\
\quad \mid\ tc_1 == tc_2 & = & \mathit{return}\ \mathit{nullSubst} \\
\mathit{mgu}\ t_1\ t_2 & = & \mathit{fail}\ \text{“}\texttt{types do not unify}\text{”}
\end{array}
$$

The *varBind* function is used for the special case of unifying a variable u with a type t. At first glance, one might think that we could just use the substitution $(u \mapsto t)$ for this. In practice, however, tests are required to ensure that this is valid, including an 'occurs check' (u '*elem*' $tv\ t$) and a test to ensure that the substitution is kind-preserving:

$$\begin{array}{lll} varBind\ u\ t \mid t == TVar\ u & = & return\ nullSubst \\ \quad\quad\quad\quad \mid u \text{ `}elem\text{` } tv\ t & = & fail\ \text{“}\texttt{occurs check fails}\text{”} \\ \quad\quad\quad\quad \mid kind\ u\ /=\ kind\ t & = & fail\ \text{“}\texttt{kinds do not match}\text{”} \\ \quad\quad\quad\quad \mid otherwise & = & return\ (u \mapsto t) \end{array}$$

In the following sections, we will also make use of an operation called *matching* that is closely related to unification. Given two types t_1 and t_2, the goal of matching is to find a substitution s such that $apply\ s\ t_1 = t_2$. Because the substitution is applied only to one type, this operation is often described as *one-way* matching. The calculation of matching substitutions is implemented by a function:

$$match \quad :: \quad Monad\ m \Rightarrow Type \rightarrow Type \rightarrow m\ Subst$$

Matching follows the same pattern as unification, except that it uses *merge* rather than @@ to combine substitutions, and it does not allow binding of variables in t_2:

$$\begin{array}{lll} match\ (TAp\ l\ r)\ (TAp\ l'\ r') & = & \textbf{do}\ sl \leftarrow match\ l\ l' \\ & & \quad\ sr \leftarrow match\ r\ r' \\ & & \quad\ merge\ sl\ sr \\ match\ (TVar\ u)\ t \mid kind\ u == kind\ t & = & return\ (u \mapsto t) \\ match\ (TCon\ tc_1)\ (TCon\ tc_2) & & \\ \quad\quad\quad\quad\quad \mid tc_1 == tc_2 & = & return\ nullSubst \\ match\ t_1\ t_2 & = & fail\ \text{“}\texttt{types do not match}\text{”} \end{array}$$

7 Type Classes, Predicates and Qualified Types

One of the most unusual features of the Haskell type system, at least in comparison to those of other polymorphically typed languages like ML, is the support that it provides for *type classes*. Described by Wadler and Blott (1989) as a general mechanism that subsumes several ad-hoc forms of overloading, type classes have found many uses (and, sometimes, abuses!) in the ten years since they were introduced. A significant portion of the code presented in this paper, particularly in this section, is needed to describe the handling of type classes in Haskell. (Of course, type classes are not the only source of complexity. The treatment of mixed implicit and explicit typing, mutually recursive bindings, and pattern matching—which are often elided in more theoretical presentations—are also significant contributors, as is the extra level of detail and precision that is needed in executable code.)

7.1 Basic Definitions

A Haskell type class can be thought of as a set of types (of some particular kind), each of which supports a certain collection of *member functions* that are specified as part of the class declaration. The types in each class (known as *instances*) are specified by a collection of instance declarations. Haskell types can be *qualified* by adding a (possibly empty) list of *predicates*, or class constraints, to restrict the ways in which type variables are instantiated[3]:

$$\begin{array}{lcl}\textbf{data}\ \mathit{Qual}\ t & = & [\mathit{Pred}] :\Rightarrow t \\ & & \textbf{deriving}\ \mathit{Eq}\end{array}$$

In a value of the form $ps :\Rightarrow t$, we refer to ps as the *context* and to t as the *head*. Predicates themselves consist of a class identifier and a type; a predicate of the form $\mathit{IsIn}\ i\ t$ asserts that t is a member of the class named i:

$$\begin{array}{lcl}\textbf{data}\ \mathit{Pred} & = & \mathit{IsIn}\ \mathit{Id}\ \mathit{Type} \\ & & \textbf{deriving}\ \mathit{Eq}\end{array}$$

For example, using the *Qual* and *Pred* datatypes, the type $(\mathit{Num}\ a) \Rightarrow a \to \mathit{Int}$ can be represented by:

$$[\mathit{IsIn}\ \text{“}\texttt{Num}\text{”}\ (\mathit{TVar}\ (\mathit{Tyvar}\ \text{“}\texttt{a}\text{”}\ \mathit{Star}))] :\Rightarrow (\mathit{TVar}\ (\mathit{Tyvar}\ \text{“}\texttt{a}\text{”}\ \mathit{Star})\ \text{‘}\mathit{fn}\text{‘}\ \mathit{tInt})$$

It would be easy to extend the *Pred* datatype to allow other forms of predicate, as is done with Trex records in Hugs (Jones & Peterson, 1999). Another frequently requested extension is to allow classes to accept multiple parameters, which would require a list of *Type*s rather than the single *Type* in the definition above.

The extension of *Types* to the *Qual* and *Pred* datatypes is straightforward:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Types}\ t \Rightarrow \mathit{Types}\ (\mathit{Qual}\ t)\ \textbf{where}} \\
\quad \mathit{apply}\ s\ (ps :\Rightarrow t) & = & \mathit{apply}\ s\ ps :\Rightarrow \mathit{apply}\ s\ t \\
\quad \mathit{tv}\ (ps :\Rightarrow t) & = & \mathit{tv}\ ps\ \text{‘}\mathit{union}\text{‘}\ \mathit{tv}\ t \\
\\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Types}\ \mathit{Pred}\ \textbf{where}} \\
\quad \mathit{apply}\ s\ (\mathit{IsIn}\ i\ t) & = & \mathit{IsIn}\ i\ (\mathit{apply}\ s\ t) \\
\quad \mathit{tv}\ (\mathit{IsIn}\ i\ t) & = & \mathit{tv}\ t
\end{array}$$

The tasks of calculating most general unifiers and matching substitutions on types also extend naturally to predicates:

$$\begin{array}{lcl}
\mathit{mguPred}, \mathit{matchPred} & :: & \mathit{Pred} \to \mathit{Pred} \to \mathit{Maybe}\ \mathit{Subst} \\
\mathit{mguPred} & = & \mathit{lift}\ \mathit{mgu} \\
\mathit{matchPred} & = & \mathit{lift}\ \mathit{match} \\
\\
\multicolumn{3}{l}{\mathit{lift}\ m\ (\mathit{IsIn}\ i\ t)\ (\mathit{IsIn}\ i'\ t')} \\
\quad \mid\ i == i' & = & m\ t\ t' \\
\quad \mid\ \mathit{otherwise} & = & \mathit{fail}\ \text{“}\texttt{classes differ}\text{”}
\end{array}$$

We will represent each class by a pair of lists, one containing the name of each

[3] The symbol $:\Rightarrow$ (pronounced ‘then’) is written as `:=>` in the concrete syntax of Haskell, and corresponds directly to the `=>` symbol that is used in instance declarations and in types.

superclass, and another containing an entry for each instance declaration:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Class} & = & ([\mathit{Id}], [\mathit{Inst}]) \\ \textbf{type}\ \mathit{Inst} & = & \mathit{Qual\ Pred} \end{array}$$

For example, a simplified version of the standard Haskell class *Ord* might be described by the following value of type *Class*:

$$\begin{array}{l} ([\text{“}\mathtt{Eq}\text{”}], [[\,] :\Rightarrow \mathit{IsIn}\ \text{“}\mathtt{Ord}\text{”}\ \mathit{tUnit}, \\ \qquad\quad [\,] :\Rightarrow \mathit{IsIn}\ \text{“}\mathtt{Ord}\text{”}\ \mathit{tChar}, \\ \qquad\quad [\,] :\Rightarrow \mathit{IsIn}\ \text{“}\mathtt{Ord}\text{”}\ \mathit{tInt}, \\ \qquad\quad [\mathit{IsIn}\ \text{“}\mathtt{Ord}\text{”}\ (\mathit{TVar}\ (\mathit{Tyvar}\ \text{“}\mathtt{a}\text{”}\ \mathit{Star})), \\ \qquad\quad\ \mathit{IsIn}\ \text{“}\mathtt{Ord}\text{”}\ (\mathit{TVar}\ (\mathit{Tyvar}\ \text{“}\mathtt{b}\text{”}\ \mathit{Star}))] \\ \qquad\qquad :\Rightarrow \mathit{IsIn}\ \text{“}\mathtt{Ord}\text{”}\ (\mathit{pair}\ (\mathit{TVar}\ (\mathit{Tyvar}\ \text{“}\mathtt{a}\text{”}\ \mathit{Star})) \\ \qquad\qquad\qquad\qquad\qquad\qquad\ (\mathit{TVar}\ (\mathit{Tyvar}\ \text{“}\mathtt{b}\text{”}\ \mathit{Star})))]) \end{array}$$

This structure captures the fact that *Eq* is a superclass of *Ord* (the only one in fact), and lists four instance declarations for the unit, character, integer, and pair types (if a and b are in *Ord*, then (a, b) is also in *Ord*). Of course, this is only a fraction of the list of *Ord* instances that are defined in the full Haskell prelude. Only the details that are needed for type inference are included in these representations. A full Haskell implementation would need to store additional information for each declaration, such as the list of member functions for each class and details of their implementations in each particular instance.

7.2 Class Environments

The information provided by the class and instance declarations in a given program can be captured by a class environment of type:

$$\begin{array}{lcl} \textbf{data}\ \mathit{ClassEnv} & = & \mathit{ClassEnv}\ \{\mathit{classes} :: \mathit{Id} \to \mathit{Maybe\ Class}, \\ & & \qquad\qquad\quad \mathit{defaults} :: [\mathit{Type}]\} \end{array}$$

The *classes* component in a *ClassEnv* value is a partial function that maps identifiers to *Class* values (or to *Nothing* if there is no class corresponding to the specified identifier). We define helper functions *super* and *insts* to extract the list of superclass identifiers, and the list of instances, respectively, for a class name i in a class environment *ce*:

$$\begin{array}{lcl} \mathit{super} & :: & \mathit{ClassEnv} \to \mathit{Id} \to [\mathit{Id}] \\ \mathit{super}\ \mathit{ce}\ i & = & \textbf{case}\ \mathit{classes}\ \mathit{ce}\ i\ \textbf{of}\ \mathit{Just}\ (\mathit{is}, \mathit{its}) \ \ \to \ \ \mathit{is} \\ & & \\ \mathit{insts} & :: & \mathit{ClassEnv} \to \mathit{Id} \to [\mathit{Inst}] \\ \mathit{insts}\ \mathit{ce}\ i & = & \textbf{case}\ \mathit{classes}\ \mathit{ce}\ i\ \textbf{of}\ \mathit{Just}\ (\mathit{is}, \mathit{its}) \ \ \to \ \ \mathit{its} \end{array}$$

These functions are intended to be used only in cases where it is known that the class i is defined in the environment *ce*. In some cases, this condition might be guaranteed by static analysis prior to type checking. Alternatively, we can resort to a dynamic check by testing *defined* (*classes ce i*) before applying either function.

The function *defined* used here is defined as follows[4]:

$$\begin{array}{lcl} \mathit{defined} & :: & \mathit{Maybe}\ a \to \mathit{Bool} \\ \mathit{defined}\ (\mathit{Just}\ x) & = & \mathit{True} \\ \mathit{defined}\ \mathit{Nothing} & = & \mathit{False} \end{array}$$

We will also define a helper function, *modify*, to describe how a class environment can be updated to reflect a new binding of a *Class* value to a given identifier:

$$\begin{array}{lcl} \mathit{modify} & :: & \mathit{ClassEnv} \to \mathit{Id} \to \mathit{Class} \to \mathit{ClassEnv} \\ \mathit{modify}\ ce\ i\ c & = & ce\ \{\mathit{classes} = \backslash j \to \mathbf{if}\ i == j\ \mathbf{then}\ \mathit{Just}\ c \\ & & \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{classes}\ ce\ j\} \end{array}$$

The *defaults* component of a *ClassEnv* value is used to provide a list of types for defaulting, as described in Section 11.5.1. Haskell allows programmers to specify a value for this list using a `default` declaration; if no explicit declaration is given, then a `default (Integer,Double)` declaration is assumed. It is easy to describe this using the *ClassEnv* type. For example, $ce\ \{\mathit{defaults} = [\mathit{tInt}]\}$ is the result of modifying a class environment *ce* to reflect the presence of a `default (Int)` declaration. Further discussion of defaulting is deferred to Section 11.5.1.

In the remainder of this section, we will show how to build an appropriate class environment for a given program, starting from an (almost) empty class environment, and extending it as necessary to reflect the effect of each class or instance declaration in the program. The initial class environment is defined as follows:

$$\begin{array}{lcl} \mathit{initialEnv} & :: & \mathit{ClassEnv} \\ \mathit{initialEnv} & = & \mathit{ClassEnv}\ \{\mathit{classes} = \backslash i \to \mathit{fail}\ \text{“}\texttt{class not defined}\text{”}, \\ & & \qquad\qquad\quad \mathit{defaults} = [\mathit{tInteger}, \mathit{tDouble}]\} \end{array}$$

As we process each class or instance declaration in a program, we transform the initial class environment to add entries, either for a new class, or for a new instance, respectively. In either case, there is a possibility that the new declaration might be incompatible with the previous declarations, attempting, for example, to redefine an existing class or instance. For this reason, we will describe transformations of a class environment as functions of the *EnvTransformer* type, using a *Maybe* type to allow for the possibility of errors:

$$\mathbf{type}\ \mathit{EnvTransformer} \quad = \quad \mathit{ClassEnv} \to \mathit{Maybe}\ \mathit{ClassEnv}$$

The sequencing of multiple transformers can be described by a (forward) composition operator (<:>):

$$\begin{array}{lcl} \mathbf{infixr} & 5 & <:> \\ (<:>) & :: & \mathit{EnvTransformer} \to \mathit{EnvTransformer} \to \mathit{EnvTransformer} \\ (f <:> g)\ ce & = & \mathbf{do}\ ce' \leftarrow f\ ce \\ & & \quad\ \ g\ ce' \end{array}$$

[4] The same function appears in the standard *Maybe* library, but with a less intuitive name: *isJust*.

Some readers will recognize this as a special case of the more general Kleisli composition operator; without the type declaration, the definition given here would work for any monad and for any element types, not just for *Maybe* and *ClassEnv*.

To add a new class to an environment, we must check that there is not already a class with the same name, and that all of the named superclasses are already defined. This is a simple way of enforcing Haskell's restriction that the superclass hierarchy be acyclic. Of course, in practice, it will be necessary to topologically sort the set of class declarations in a program to determine a suitable ordering; any cycles in the hierarchy will typically be detected at this stage.

```
addClass                              :: Id -> [Id] -> EnvTransformer
addClass i is ce
    | defined (classes ce i)          = fail "class already defined"
    | any (not . defined . classes ce) is = fail "superclass not defined"
    | otherwise                       = return (modify ce i (is, []))
```

For example, we can describe the effect of the class declarations in the Haskell prelude using the following transformer:

```
addPreludeClasses :: EnvTransformer
addPreludeClasses  = addCoreClasses <:> addNumClasses
```

This definition breaks down the set of standard Haskell classes into two separate pieces. The core classes are described as follows:

```
addCoreClasses ::   EnvTransformer
addCoreClasses =    addClass "Eq" []
               <:>  addClass "Ord" ["Eq"]
               <:>  addClass "Show" []
               <:>  addClass "Read" []
               <:>  addClass "Bounded" []
               <:>  addClass "Enum" []
               <:>  addClass "Functor" []
               <:>  addClass "Monad" []
```

The hierarchy of numeric classes is captured separately in the following definition:

```
addNumClasses  ::   EnvTransformer
addNumClasses  =    addClass "Num" ["Eq", "Show"]
               <:>  addClass "Real" ["Num", "Ord"]
               <:>  addClass "Fractional" ["Num"]
               <:>  addClass "Integral" ["Real", "Enum"]
               <:>  addClass "RealFrac" ["Real", "Fractional"]
               <:>  addClass "Floating" ["Fractional"]
               <:>  addClass "RealFloat" ["RealFrac", "Floating"]
```

To add a new instance to a class, we must check that the class to which the instance applies is defined, and that the new instance does not overlap with any previously declared instance:

```
addInst                        :: [Pred] → Pred → EnvTransformer
addInst ps p@(IsIn i _) ce
  | not (defined (classes ce i)) = fail "no class for instance"
  | any (overlap p) qs           = fail "overlapping instance"
  | otherwise                    = return (modify ce i c)
   where its = insts ce i
         qs  = [q | (_ :⇒ q) ← its]
         c   = (super ce i, (ps :⇒ p) : its)
```

Two instances for a class are said to *overlap* if there is some predicate that is a substitution instance of the heads of both instance declarations. It is easy to test for overlapping predicates using the functions that we have defined previously:

```
overlap       :: Pred → Pred → Bool
overlap p q   =  defined (mguPred p q)
```

This test covers simple cases where a program provides two instance declarations for the same type (for example, two declarations for `Eq Int`), but it also covers cases where more interesting overlaps occur (for example, between the predicates `Eq [Int]` and `Eq [a]`, or between predicates `Eq (a,Bool)` and `Eq (Int,b)`). In each case, the existence of an overlap indicates the possibility of a semantic ambiguity, with two applicable instance declarations, and no clear reason to prefer one over the other. This is why Haskell treats such overlaps as an error. Extensions to Haskell to support overlapping instances in certain special cases have been considered elsewhere; they appear to have interesting applications, but also have some potentially troublesome impact on program semantics (Peyton Jones *et al.*, 1997). We will not consider such issues further in this paper.

To illustrate how the *addInst* function might be used, the following definition shows how the standard prelude class environment can be extended to include the four instances for *Ord* from the example in Section 7.1:

```
exampleInsts  :: EnvTransformer
exampleInsts  =  addPreludeClasses
             <:> addInst [] (IsIn "Ord" tUnit)
             <:> addInst [] (IsIn "Ord" tChar)
             <:> addInst [] (IsIn "Ord" tInt)
             <:> addInst [IsIn "Ord" (TVar (Tyvar "a" Star)),
                          IsIn "Ord" (TVar (Tyvar "b" Star))]
                         (IsIn "Ord" (pair (TVar (Tyvar "a" Star))
                                           (TVar (Tyvar "b" Star))))
```

The Haskell report imposes some further restrictions on class and instance declarations that are not enforced by the definitions of *addClass* and *addInst*. For example, the superclasses of a class should have the same kind as the class itself; the parameters of any predicates in an instance context should be type variables, each of which should appear in the head of the instance; and the type appearing in the head of an instance should consist of a type constructor applied to a sequence of distinct type variable arguments. Because these conditions have no direct impact on type checking, and because they are straightforward but tedious to verify, we have chosen not to include tests for them here, and instead assume that they have been checked during static analysis prior to type checking.

7.3 Entailment

In this section, we describe how class environments can be used to answer questions about which types are instances of particular classes. More generally, we consider the treatment of *entailment*: given a predicate p and a list of predicates ps, our goal is to determine whether p will hold whenever all of the predicates in ps are satisfied. In the special case where $p = \mathit{IsIn}\ i\ t$ and $ps = [\,]$, this amounts to determining whether t is an instance of the class i. In the theory of qualified types (Jones, 1992), assertions like this are captured using judgements of the form $ps \Vdash p$; we use a different notation here—the *entail* function that is defined at the end of this section—to make the dependence on a class environment explicit.

As a first step, we can ask how information about superclasses and instances can be used independently to help reason about entailments. For example, if a type is an instance of a class i, then it must also be an instance of any superclasses of i. Hence, using only superclass information, we can be sure that, if a given predicate p holds, then so too must all of the predicates in the list *bySuper* p:

```
bySuper   ::  ClassEnv -> Pred -> [Pred]
bySuper ce p@(IsIn i t)
          =   p : concat [bySuper ce (IsIn i' t) | i' <- super ce i]
```

The list *bySuper ce p* may contain duplicates, but it will always be finite because of the restriction that the superclass hierarchy is acyclic.

Next we consider how information about instances can be used. Of course, for a given predicate $p = \mathit{IsIn}\ i\ t$, we can find all the directly relevant instances in a class environment ce by looking in *insts ce i*. As we have seen, individual instance declarations are mapped into clauses of the form $ps :\Rightarrow h$. The head predicate h describes the general form of instances that can be constructed from this declaration, and we can use *matchPred* to determine whether this instance is applicable to the given predicate p. If it is applicable, then matching will return a substitution u, and the remaining subgoals are the elements of *map* (*apply* u) ps. The following function uses these ideas to determine the list of subgoals for a given predicate:

```
byInst                    ::  ClassEnv -> Pred -> Maybe [Pred]
byInst ce p@(IsIn i t)    =   msum [tryInst it | it <- insts ce i]
  where tryInst (ps :=> h)  =  do u <- matchPred h p
                                  Just (map (apply u) ps)
```

The *msum* function used here comes from the standard *Monad* library, and returns the first defined element in a list of *Maybe* values; if there are no defined elements in the list, then it returns *Nothing*. Because Haskell prevents overlapping instances, there is at most one applicable instance for any given p, and we can be sure that the first defined element will actually be the *only* defined element in this list.

The *bySuper* and *byInst* functions can be used in combination to define a general entailment operator, *entail*. Given a particular class environment *ce*, the intention here is that *entail ce ps p* will be *True* if, and only if, the predicate p will hold whenever all of the predicates in *ps* are satisfied:

$$
\begin{array}{lcl}
\mathit{entail} & :: & \mathit{ClassEnv} \to [\mathit{Pred}] \to \mathit{Pred} \to \mathit{Bool} \\
\mathit{entail}\ \mathit{ce}\ \mathit{ps}\ p & = & \mathit{any}\ (p\ \text{`}\mathit{elem}\text{`})\ (\mathit{map}\ (\mathit{bySuper}\ \mathit{ce})\ \mathit{ps})\ || \\
 & & \textbf{case}\ \mathit{byInst}\ \mathit{ce}\ p\ \textbf{of} \\
 & & \quad \mathit{Nothing} \ \to\ \mathit{False} \\
 & & \quad \mathit{Just}\ \mathit{qs} \ \ \to\ \mathit{all}\ (\mathit{entail}\ \mathit{ce}\ \mathit{ps})\ \mathit{qs}
\end{array}
$$

The first step here is to determine whether p can be deduced from *ps* using only superclasses. If that fails, we look for a matching instance and generate a list of predicates *qs* as a new goal, each of which must, in turn, follow from *ps*.

Conditions specified in the Haskell report—namely that the class hierarchy is acyclic and that the types in any instance declaration are strictly smaller than those in the head—translate into conditions on the values for the *ClassEnv* that can be passed in as *ce*, and these are enough to guarantee that tests for entailment will terminate. Completeness of the algorithm is also important: will *entail ce ps p* always return *True* whenever there is a way to prove p from *ps*? In fact our algorithm does not cover all possible cases: it does not test to see if p is a superclass of some other predicate q for which *entail ce ps q* is *True*. Extending the algorithm to test for this would be very difficult because there is no obvious way to choose a particular q, and, in general, there will be infinitely many potential candidates to consider. Fortunately, a technical condition in the Haskell report (Peyton Jones & Hughes, 1999; Condition 1 on Page 47) reassures us that this is not necessary: if p can be obtained as an immediate superclass of some predicate q that was built using an instance declaration in an entailment *entail ce ps q*, then *ps* must already be strong enough to deduce p. Thus, although we have not formally proved these properties, we believe that our algorithm is sound, complete, and guaranteed to terminate.

7.4 Context Reduction

Class environments also play an important role in an aspect of the Haskell type system that is known as *context reduction*. The basic goal of context reduction is to reduce a list of predicates to an equivalent but, in some sense, simpler list. The Haskell report (Peyton Jones & Hughes, 1999) provides only informal hints about this aspect of the Haskell typing, where both pragmatics and theory have important parts to play. We believe therefore that this is one of the areas where a more formal specification will be particularly valuable.

One way to simplify a list of predicates is to simplify the type components of individual predicates in the list. For example, given the instance declarations in the Haskell standard prelude, we could replace any occurrences of predicates like `Eq [a]`, `Eq (a,a)`, or `Eq ([a],Int)` with `Eq a`. This is valid because, for any choice of `a`, each one of these predicates holds if, and only if, `Eq a` holds. Notice that, in some cases, an attempt to simplify type components—for example, by replacing `Eq (a, b)` with `(Eq a, Eq b)`—may increase the number of predicates in the list. The extent to which simplifications like this are used in a system of qualified types has an impact on the implementation and performance of overloading in practical systems (Jones, 1992; Chapter 7). In Haskell, however, the decisions are made for us by a syntactic restriction that forces us to simplify predicates until we obtain types in a kind of 'head-normal form'. This terminology is motivated by similarities with the concept of *head-normal forms* in λ-calculus. More precisely, the syntax of Haskell requires class arguments to be of the form $v\ t_1\ \ldots\ t_n$, where v is a type variable, and $t_1,\ldots,t_n$ are types (and $n \geq 0$). The following function allows us to determine whether a given predicate meets these restrictions:

```
inHnf                :: Pred -> Bool
inHnf (IsIn c t)      = hnf t
  where hnf (TVar v)  = True
        hnf (TCon tc) = False
        hnf (TAp t _) = hnf t
```

Predicates that do not fit this pattern must be broken down using *byInst*. In some cases, this will result in predicates being eliminated altogether. In others, where *byInst* fails, it will indicate that a predicate is unsatisfiable, and will trigger an error diagnostic. This process is captured in the following definition:

```
toHnfs               :: Monad m => ClassEnv -> [Pred] -> m [Pred]
toHnfs ce ps          = do pss <- mapM (toHnf ce) ps
                           return (concat pss)

toHnf                :: Monad m => ClassEnv -> Pred -> m [Pred]
toHnf ce p | inHnf p  = return [p]
           | otherwise = case byInst ce p of
                           Nothing -> fail "context reduction"
                           Just ps -> toHnfs ce ps
```

Another way to simplify a list of predicates is to reduce the number of elements that it contains. There are several ways that this might be achieved: by removing duplicates (e.g., reducing `(Eq a, Eq a)` to `Eq a`); by eliminating predicates that are already known to hold (e.g., removing any occurrences of `Num Int`); or by using superclass information (e.g., reducing `(Eq a, Ord a)` to `Ord a`). In each case, the reduced list of predicates, is equivalent to the initial list, meaning that all the predicates in the first will be satisfied if, and only if, all of the predicates in the second are satisfied. The simplification algorithm that we will use here is based on the observation that a predicate p in a list of predicates $(p : ps)$ can be eliminated if p is entailed by ps. As a special case, this will eliminate duplicated predicates: if p is repeated in ps, then it will also be entailed by ps. These ideas are used in the following definition of the *simplify* function, which loops through each predicate in the list and uses an accumulating parameter to build up the final result. Each time we encounter a predicate that is entailed by the others, we remove it from the list.

$$
\begin{array}{lll}
simplify & :: & ClassEnv \to [Pred] \to [Pred] \\
simplify\ ce & = & loop\ [\,] \\
\quad \mathbf{where}\ loop\ rs\ [\,] & & \qquad\qquad\qquad\qquad = \ rs \\
\qquad\qquad loop\ rs\ (p : ps) & \mid & entail\ ce\ (rs +\!\!+ ps)\ p \ = \ loop\ rs\ ps \\
 & \mid & otherwise \qquad\qquad\quad = \ loop\ (p : rs)\ ps
\end{array}
$$

Now we can describe the particular form of context reduction used in Haskell as a combination of *toHnfs* and *simplify*. Specifically, we use *toHnfs* to reduce the list of predicates to head-normal form, and then simplify the result:

$$
\begin{array}{lll}
reduce & :: & Monad\ m \Rightarrow ClassEnv \to [Pred] \to m\ [Pred] \\
reduce\ ce\ ps & = & \mathbf{do}\ qs \leftarrow toHnfs\ ce\ ps \\
 & & \qquad return\ (simplify\ ce\ qs)
\end{array}
$$

As a technical aside, we note that there is some redundancy in the definition of *reduce*. The *simplify* function is defined in terms of *entail*, which makes use of the information provided by both superclass and instance declarations. The predicates in *qs*, however, are guaranteed to be in head-normal form, and hence will not match instance declarations that satisfy the syntactic restrictions of Haskell. It follows that we could make do with a version of *simplify* that used only the following function in determining (superclass) entailments:

$$
\begin{array}{lll}
scEntail & :: & ClassEnv \to [Pred] \to Pred \to Bool \\
scEntail\ ce\ ps\ p & = & any\ (p\ \text{`}elem\text{`})\ (map\ (bySuper\ ce)\ ps)
\end{array}
$$

8 Type Schemes

Type schemes are used to describe polymorphic types, and are represented using a list of kinds and a qualified type:

$$
\begin{array}{lll}
\mathbf{data}\ Scheme & = & Forall\ [Kind]\ (Qual\ Type) \\
 & & \mathbf{deriving}\ Eq
\end{array}
$$

There is no direct equivalent of *Forall* in the syntax of Haskell. Instead, implicit quantifiers are inserted as necessary to bind free type variables.

In a type scheme *Forall ks qt*, each type of the form *TGen n* that appears in the qualified type *qt* represents a generic, or universally quantified type variable whose kind is given by $ks\,!!\,n$. This is the only place where we will allow *TGen* values to appear in a type. We had originally hoped that this restriction could be captured statically by a careful choice of the representation for types and type schemes. Unfortunately, we have not yet found a satisfactory way to enforce this, and, after considering several alternatives, we have settled for the representation shown here because it allows for simple implementations of equality and substitution. For example, the implementation of *apply* on *Type* values ignores *TGen* values, so we can be sure that there will be no variable capture problems in the following definition:

$$\begin{array}{lcl} \textbf{instance}\ \mathit{Types}\ \mathit{Scheme}\ \textbf{where} & & \\ \quad \mathit{apply}\ s\ (\mathit{Forall}\ \mathit{ks}\ \mathit{qt}) & = & \mathit{Forall}\ \mathit{ks}\ (\mathit{apply}\ s\ \mathit{qt}) \\ \quad \mathit{tv}\ (\mathit{Forall}\ \mathit{ks}\ \mathit{qt}) & = & \mathit{tv}\ \mathit{qt} \end{array}$$

Type schemes are constructed by quantifying a qualified type *qt* with respect to a list of type variables *vs*:

$$\begin{array}{lcl} \mathit{quantify} & :: & [\mathit{Tyvar}] \to \mathit{Qual}\ \mathit{Type} \to \mathit{Scheme} \\ \mathit{quantify}\ \mathit{vs}\ \mathit{qt} & = & \mathit{Forall}\ \mathit{ks}\ (\mathit{apply}\ s\ \mathit{qt}) \\ \quad \textbf{where}\ \mathit{vs}' & = & [v \mid v \leftarrow \mathit{tv}\ \mathit{qt},\ v\ \text{`}\mathit{elem}\text{`}\ \mathit{vs}] \\ \qquad\qquad \mathit{ks} & = & \mathit{map}\ \mathit{kind}\ \mathit{vs}' \\ \qquad\qquad s & = & \mathit{zip}\ \mathit{vs}'\ (\mathit{map}\ \mathit{TGen}\ [0..]) \end{array}$$

Note that the order of the kinds in *ks* is determined by the order in which the variables *v* appear in *tv qt*, and not by the order in which they appear in *vs*. So, for example, the leftmost quantified variable in a type scheme will always be represented by *TGen* 0. By insisting that type schemes are constructed in this way, we obtain a unique canonical form for *Scheme* values. This is important because it means that we can test whether two type schemes are the same—for example, to determine whether an inferred type agrees with a declared type—using Haskell's derived equality, and without having to implement more complex tests for α-equivalence.

In practice, we sometimes need to convert a *Type* into a *Scheme* without adding any qualifying predicates or quantified variables. For this special case, we can use the following function instead of *quantify*:

$$\begin{array}{lcl} \mathit{toScheme} & :: & \mathit{Type} \to \mathit{Scheme} \\ \mathit{toScheme}\ t & = & \mathit{Forall}\ [\,]\ ([\,] :\Rightarrow t) \end{array}$$

To complete our description of type schemes, we need to be able to instantiate the quantified variables in *Scheme* values. In fact, for the purposes of type inference, we only need the special case that instantiates a type scheme with fresh type variables. We therefore defer further description of instantiation to Section 10 where the mechanisms for generating fresh type variables are introduced.

9 Assumptions

Assumptions about the type of a variable are represented by values of the *Assump* datatype, each of which pairs a variable name with a type scheme:

$$\textbf{data}\ \mathit{Assump} \quad = \quad \mathit{Id} :\!>\!: \mathit{Scheme}$$

Once again, we can extend the *Types* class to allow the application of a substitution to an assumption:

$$\begin{array}{lcl} \textbf{instance}\ \mathit{Types}\ \mathit{Assump}\ \textbf{where} & & \\ \quad \mathit{apply}\ s\ (i :\!>\!: \mathit{sc}) & = & i :\!>\!: (\mathit{apply}\ s\ \mathit{sc}) \\ \quad \mathit{tv}\ (i :\!>\!: \mathit{sc}) & = & \mathit{tv}\ \mathit{sc} \end{array}$$

Thanks to the instance definition for *Types* on lists (Section 5), we can also use the *apply* and *tv* operators on the lists of assumptions that are used to record the type of each program variable during type inference. We will also use the following function to find the type of a particular variable in a given set of assumptions:

$$
\begin{array}{lcl}
\mathit{find} & :: & \mathit{Monad}\ m \Rightarrow \mathit{Id} \rightarrow [\mathit{Assump}] \rightarrow m\ \mathit{Scheme} \\
\mathit{find}\ i\ [\,] & = & \mathit{fail}\ (\text{“}\texttt{unbound identifier: }\text{”} +\!\!+\, i) \\
\mathit{find}\ i\ ((i' :\!>\!: sc) : as) & = & \textbf{if}\ i == i'\ \textbf{then}\ \mathit{return}\ sc\ \textbf{else}\ \mathit{find}\ i\ as
\end{array}
$$

This definition allows for the possibility that the variable i might not appear in as. In practice, occurrences of unbound variables will probably have been detected in earlier compiler passes.

10 A Type Inference Monad

It is now quite standard to use monads as a way to hide certain aspects of ‘plumbing’ and to draw attention instead to more important aspects of a program’s design (Wadler, 1992). The purpose of this section is to define the monad that will be used in the description of the main type inference algorithm in Section 11. Our choice of monad is motivated by the needs of maintaining a ‘current substitution’ and of generating fresh type variables during typechecking. In a more realistic implementation, we might also want to add error reporting facilities, but in this paper the crude but simple *fail* function from the Haskell prelude is all that we require. It follows that we need a simple state monad with only a substitution and an integer (from which we can generate new type variables) as its state:

$$
\textbf{newtype}\ \mathit{TI}\ a \;=\; \mathit{TI}\ (\mathit{Subst} \rightarrow \mathit{Int} \rightarrow (\mathit{Subst},\ \mathit{Int},\ a))
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{TI}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{return}\ x & = & \mathit{TI}\ (\backslash s\ n \rightarrow (s,\ n,\ x)) \\
\mathit{TI}\ f \ \texttt{>>=}\ g & = & \mathit{TI}\ (\backslash s\ n \rightarrow \textbf{case}\ f\ s\ n\ \textbf{of} \\
 & & \qquad\qquad (s',\ m,\ x) \ \rightarrow\ \textbf{let}\ \mathit{TI}\ gx \ = \ g\ x \\
 & & \qquad\qquad\qquad\qquad\qquad\ \ \textbf{in}\ gx\ s'\ m)
\end{array}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{runTI} & :: & \mathit{TI}\ a \rightarrow a \\
\mathit{runTI}\ (\mathit{TI}\ f) & = & x\ \textbf{where}\ (s,\ n,\ x) \ = \ f\ \mathit{nullSubst}\ 0
\end{array}
$$

The *getSubst* operation returns the current substitution, while *unify* extends it with a most general unifier of its arguments:

$$
\begin{array}{lcl}
\mathit{getSubst} & :: & \mathit{TI}\ \mathit{Subst} \\
\mathit{getSubst} & = & \mathit{TI}\ (\backslash s\ n \rightarrow (s,\ n,\ s)) \\
 & & \\
\mathit{unify} & :: & \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{TI}\ () \\
\mathit{unify}\ t_1\ t_2 & = & \textbf{do}\ s \leftarrow \mathit{getSubst} \\
 & & \qquad u \leftarrow \mathit{mgu}\ (\mathit{apply}\ s\ t_1)\ (\mathit{apply}\ s\ t_2) \\
 & & \qquad \mathit{extSubst}\ u
\end{array}
$$

For clarity, we define the operation that extends the substitution as a separate function, even though it is used only here in the definition of *unify*:

$$
\begin{array}{lcl}
extSubst & :: & Subst \rightarrow TI\ () \\
extSubst\ s' & = & TI\ (\backslash s\ n \rightarrow (s'@@s,\ n,\ ()))
\end{array}
$$

Overall, the decision to hide the current substitution in the *TI* monad makes the presentation of type inference much clearer. In particular, it avoids heavy use of *apply* every time an extension is (or might have been) computed.

There is only one primitive that deals with the integer portion of the state, using it in combination with *enumId* to generate a new type variable of a specified kind:

$$
\begin{array}{lcl}
newTVar & :: & Kind \rightarrow TI\ Type \\
newTVar\ k & = & TI\ (\backslash s\ n \rightarrow \mathbf{let}\ v\ =\ Tyvar\ (enumId\ n)\ k \\
 & & \qquad\qquad\quad \mathbf{in}\ (s,\ n+1,\ TVar\ v))
\end{array}
$$

One place where *newTVar* is useful is in instantiating a type scheme with new type variables of appropriate kinds:

$$
\begin{array}{lcl}
freshInst & :: & Scheme \rightarrow TI\ (Qual\ Type) \\
freshInst\ (Forall\ ks\ qt) & = & \mathbf{do}\ ts \leftarrow mapM\ newTVar\ ks \\
 & & \qquad return\ (inst\ ts\ qt)
\end{array}
$$

The structure of this definition guarantees that *ts* has exactly the right number of type variables, and each with the right kind, to match *ks*. Hence, if the type scheme is well-formed, then the qualified type returned by *freshInst* will not contain any unbound generics of the form *TGen n*. The definition relies on an auxiliary function *inst*, which is a variation of *apply* that works on generic variables. In other words, *inst ts t* replaces each occurrence of a generic variable *TGen n* in *t* with *ts !! n*. It is convenient to build up the definition of *inst* using overloading:

$$
\begin{array}{l}
\mathbf{class}\ Instantiate\ t\ \mathbf{where} \\
\quad inst\ \ ::\ \ [Type] \rightarrow t \rightarrow t \\
\mathbf{instance}\ Instantiate\ Type\ \mathbf{where} \\
\quad inst\ ts\ (TAp\ l\ r)\ \ =\ \ TAp\ (inst\ ts\ l)\ (inst\ ts\ r) \\
\quad inst\ ts\ (TGen\ n)\ \ =\ \ ts\ !!\ n \\
\quad inst\ ts\ t\qquad\qquad =\ \ t \\
\mathbf{instance}\ Instantiate\ a \Rightarrow Instantiate\ [a]\ \mathbf{where} \\
\quad inst\ ts\ \ =\ \ map\ (inst\ ts) \\
\mathbf{instance}\ Instantiate\ t \Rightarrow Instantiate\ (Qual\ t)\ \mathbf{where} \\
\quad inst\ ts\ (ps :\Rightarrow t)\ \ =\ \ inst\ ts\ ps :\Rightarrow inst\ ts\ t \\
\mathbf{instance}\ Instantiate\ Pred\ \mathbf{where} \\
\quad inst\ ts\ (IsIn\ c\ t)\ \ =\ \ IsIn\ c\ (inst\ ts\ t)
\end{array}
$$

11 Type Inference

With this section we have reached the heart of the paper, detailing our algorithm for type inference. It is here that we finally see how the machinery that has been built up in earlier sections is actually put to use. We develop the complete algorithm in stages, working through the abstract syntax of the input language from the simplest part (literals) to the most complex (binding groups). Most of the typing rules are expressed by functions whose types are simple variants of the following synonym:

$$\textbf{type}\ \mathit{Infer}\ e\ t \quad = \quad \mathit{ClassEnv} \to [\mathit{Assump}] \to e \to \mathit{TI}\ ([\mathit{Pred}],\ t)$$

In more theoretical treatments, it would not be surprising to see the rules expressed in terms of judgments $\Gamma; P \mid A \vdash e : t$, where Γ is a class environment, P is a set of predicates, A is a set of assumptions, e is an expression, and t is a corresponding type (Jones, 1992). Judgments like this can be thought of as 5-tuples, and the typing rules themselves just correspond to a 5-place relation. Exactly the same structure shows up in types of the form *Infer e t*, except that, by using functions, we distinguish very clearly between input and output parameters.

11.1 Literals

Like other languages, Haskell provides special syntax for constant values of certain primitive datatypes, including numerics, characters, and strings. We will represent these *literal* expressions as values of the *Literal* datatype:

$$\begin{array}{lcl}\textbf{data}\ \mathit{Literal} & = & \mathit{LitInt}\ \mathit{Integer} \\ & \mid & \mathit{LitChar}\ \mathit{Char} \\ & \mid & \mathit{LitRat}\ \mathit{Rational} \\ & \mid & \mathit{LitStr}\ \mathit{String}\end{array}$$

Type inference for literals is straightforward. For characters, we just return *tChar*. For integers, we return a new type variable v together with a predicate to indicate that v must be an instance of the *Num* class. The cases for *String* and floating point literals follow similar patterns:

$$\begin{array}{lcl}\mathit{tiLit} & :: & \mathit{Literal} \to \mathit{TI}\ ([\mathit{Pred}],\ \mathit{Type}) \\ \mathit{tiLit}\ (\mathit{LitChar}\ \_) & = & \mathit{return}\ ([\,],\ \mathit{tChar}) \\ \mathit{tiLit}\ (\mathit{LitInt}\ \_) & = & \textbf{do}\ v \leftarrow \mathit{newTVar}\ \mathit{Star} \\ & & \quad\ \ \mathit{return}\ ([\mathit{IsIn}\ \text{“}\texttt{Num}\text{”}\ v],\ v) \\ \mathit{tiLit}\ (\mathit{LitStr}\ \_) & = & \mathit{return}\ ([\,],\ \mathit{tString}) \\ \mathit{tiLit}\ (\mathit{LitRat}\ \_) & = & \textbf{do}\ v \leftarrow \mathit{newTVar}\ \mathit{Star} \\ & & \quad\ \ \mathit{return}\ ([\mathit{IsIn}\ \text{“}\texttt{Fractional}\text{”}\ v],\ v)\end{array}$$

11.2 Patterns

Patterns are used to inspect and deconstruct data values in lambda abstractions, function and pattern bindings, list comprehensions, do notation, and case expressions. We will represent patterns using values of the *Pat* datatype:

$$\begin{array}{lcl} \textbf{data}\ Pat & = & PVar\ Id \\ & | & PWildcard \\ & | & PAs\ Id\ Pat \\ & | & PLit\ Literal \\ & | & PNpk\ Id\ Integer \\ & | & PCon\ Assump\ [Pat] \end{array}$$

A *PVar i* pattern matches any value and binds the result to the variable *i*. A *PWildcard* pattern, corresponding to an underscore _ in Haskell syntax, matches any value, but does not bind any variables. A pattern of the form (*PAs i pat*), known as an "as-pattern" and written using the syntax *i*@*pat* in Haskell, binds the variable *i* to any value that matches the pattern *pat*, while also binding any variables that appear in *pat*. A *PLit l* pattern matches only the particular value denoted by the literal *l*. A pattern (*PNpk i k*) is an $(n + k)$ pattern, which matches any positive integral value m that is greater than or equal to k, and binds the variable i to the difference $(m - k)$. Finally, a pattern of the form *PCon a pats* matches only values that were built using the constructor function *a* with a sequence of arguments that matches *pats*. We use values *a* of type *Assump* to represent constructor functions; all that we really need for typechecking is the type, although the name is useful for debugging. A full implementation would store additional details, such as arity, and use this to check that constructor functions in patterns are always fully applied.

Most Haskell patterns have a direct representation in *Pat*, but extensions would be needed to account for patterns using labeled fields. This is not difficult, but adds some complexity, which we prefer to avoid in this presentation.

Type inference for patterns has two goals: To calculate a type for each bound variable, and to determine what type of values the whole pattern might match. This leads us to look for a function:

$$tiPat \quad :: \quad Pat \to TI\ ([Pred],\ [Assump],\ Type)$$

Note that we do not need to pass in a list of assumptions here; by definition, any occurrence of a variable in a pattern would hide rather than refer to a variable of the same name in an enclosing scope.

For a variable pattern, *PVar i*, we just return a new assumption, binding *i* to a fresh type variable.

$$\begin{array}{lcl} tiPat\ (PVar\ i) & = & \textbf{do}\ v \leftarrow newTVar\ Star \\ & & \qquad return\ ([\,],\ [i :\!>\!:\ toScheme\ v],\ v) \end{array}$$

Haskell does not allow multiple use of any variable in a pattern, so we can be sure that this is the first and only occurrence of *i* that we will encounter in the pattern. Wildcards are typed in the same way except that we do not need to create a new assumption:

$$\begin{array}{lcl} tiPat\ PWildcard & = & \textbf{do}\ v \leftarrow newTVar\ Star \\ & & \qquad return\ ([\,],\ [\,],\ v) \end{array}$$

To type an as-pattern *PAs i pat*, we calculate a set of assumptions and a type for the *pat* pattern, and then add an extra assumption to bind i:

$$
\begin{array}{lcl}
\mathit{tiPat}\ (\mathit{PAs}\ i\ \mathit{pat}) & = & \mathbf{do}\ (\mathit{ps},\ \mathit{as},\ t) \leftarrow \mathit{tiPat}\ \mathit{pat} \\
 & & \quad \mathit{return}\ (\mathit{ps},\ (i :\!>\!: \mathit{toScheme}\ t) : \mathit{as},\ t)
\end{array}
$$

For literal patterns, we use *tiLit* from the previous section:

$$
\begin{array}{lcl}
\mathit{tiPat}\ (\mathit{PLit}\ l) & = & \mathbf{do}\ (\mathit{ps},\ t) \leftarrow \mathit{tiLit}\ l \\
 & & \quad \mathit{return}\ (\mathit{ps},\ [\,],\ t)
\end{array}
$$

The rule for $(n + k)$ patterns does not fix a type for the bound variable, but adds a predicate to constrain the choice to instances of the *Integral* class:

$$
\begin{array}{lcl}
\mathit{tiPat}\ (\mathit{PNpk}\ i\ k) & = & \mathbf{do}\ t \leftarrow \mathit{newTVar}\ \mathit{Star} \\
 & & \quad \mathit{return}\ ([\mathit{IsIn}\ \text{“}\mathtt{Integral}\text{”}\ t],\ [i :\!>\!: \mathit{toScheme}\ t],\ t)
\end{array}
$$

The case for constructed patterns is slightly more complex:

$$
\begin{array}{lcl}
\mathit{tiPat}\ (\mathit{PCon}\ (i :\!>\!: \mathit{sc})\ \mathit{pats}) & = & \mathbf{do}\ (\mathit{ps},\ \mathit{as},\ \mathit{ts}) \leftarrow \mathit{tiPats}\ \mathit{pats} \\
 & & \quad t' \leftarrow \mathit{newTVar}\ \mathit{Star} \\
 & & \quad (\mathit{qs} :\Rightarrow t) \leftarrow \mathit{freshInst}\ \mathit{sc} \\
 & & \quad \mathit{unify}\ t\ (\mathit{foldr}\ \mathit{fn}\ t'\ \mathit{ts}) \\
 & & \quad \mathit{return}\ (\mathit{ps} \mathbin{+\!\!+} \mathit{qs},\ \mathit{as},\ t')
\end{array}
$$

First we use the *tiPats* function, defined below, to calculate types *ts* for each subpattern in *pats* together with corresponding lists of assumptions in *as* and predicates in *ps*. Next, we generate a new type variable t' that will be used to capture the (as yet unknown) type of the whole pattern. From this information, we would expect the constructor function at the head of the pattern to have type $\mathit{foldr}\ \mathit{fn}\ t'\ \mathit{ts}$. We can check that this is possible by instantiating the known type *sc* of the constructor and unifying.

The *tiPats* function is a variation of *tiPat* that takes a list of patterns as input, and returns a list of types (together with a list of predicates and a list of assumptions) as its result.

$$
\begin{array}{lcl}
\mathit{tiPats} & :: & [\mathit{Pat}] \rightarrow \mathit{TI}\ ([\mathit{Pred}],\ [\mathit{Assump}],\ [\mathit{Type}]) \\
\mathit{tiPats}\ \mathit{pats} & = & \mathbf{do}\ \mathit{psasts} \leftarrow \mathit{mapM}\ \mathit{tiPat}\ \mathit{pats} \\
 & & \quad \mathbf{let}\ \mathit{ps} = \mathit{concat}\ [\mathit{ps}' \mid (\mathit{ps}',\ \_,\ \_) \leftarrow \mathit{psasts}] \\
 & & \quad \phantom{\mathbf{let}}\ \mathit{as} = \mathit{concat}\ [\mathit{as}' \mid (\_,\ \mathit{as}',\ \_) \leftarrow \mathit{psasts}] \\
 & & \quad \phantom{\mathbf{let}}\ \mathit{ts} = [t \mid (\_,\ \_,\ t) \leftarrow \mathit{psasts}] \\
 & & \quad \mathit{return}\ (\mathit{ps},\ \mathit{as},\ \mathit{ts})
\end{array}
$$

We have already seen how *tiPats* was used in the treatment of *PCon* patterns above. It is also useful in other situations where lists of patterns are used, such as on the left hand side of an equation in a function definition.

11.3 Expressions

Next we describe type inference for expressions, represented by the *Expr* datatype:

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Expr} & = & \mathit{Var}\ \mathit{Id} \\
 & | & \mathit{Lit}\ \mathit{Literal} \\
 & | & \mathit{Const}\ \mathit{Assump} \\
 & | & \mathit{Ap}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \mathit{Let}\ \mathit{BindGroup}\ \mathit{Expr}
\end{array}
$$

The *Var* and *Lit* constructors are used to represent variables and literals, respectively. The *Const* constructor is used to deal with named constants, such as the constructor or selector functions associated with a particular datatype or the member functions that are associated with a particular class. We use values of type *Assump* to supply a name and type scheme, which is all the information that we need for the purposes of type inference. Function application is represented using the *Ap* constructor, while *Let* is used to represent let expressions. (Note that the definition of the *BindGroup* type, used here to represent binding groups, will be delayed to Section 11.6.3.) Of course, Haskell has a much richer syntax of expressions—which includes λ-abstractions, case expressions, conditionals, list comprehensions, and do-notation—but they all have simple translations into *Expr* values. For example, a λ-expression like `\x->e` can be rewritten using a local definition as `let f x = e in f`, where `f` is a new variable.

Type inference for expressions is quite straightforward:

$$
\begin{array}{lcl}
\mathit{tiExpr} & :: & \mathit{Infer}\ \mathit{Expr}\ \mathit{Type} \\
\mathit{tiExpr}\ \mathit{ce}\ \mathit{as}\ (\mathit{Var}\ i) & = & \textbf{do}\ \mathit{sc} \leftarrow \mathit{find}\ i\ \mathit{as} \\
 & & \quad (\mathit{ps} :\Rightarrow t) \leftarrow \mathit{freshInst}\ \mathit{sc} \\
 & & \quad \mathit{return}\ (\mathit{ps},\ t) \\
\mathit{tiExpr}\ \mathit{ce}\ \mathit{as}\ (\mathit{Const}\ (i :>: \mathit{sc})) & = & \textbf{do}\ (\mathit{ps} :\Rightarrow t) \leftarrow \mathit{freshInst}\ \mathit{sc} \\
 & & \quad \mathit{return}\ (\mathit{ps},\ t) \\
\mathit{tiExpr}\ \mathit{ce}\ \mathit{as}\ (\mathit{Lit}\ l) & = & \textbf{do}\ (\mathit{ps},\ t) \leftarrow \mathit{tiLit}\ l \\
 & & \quad \mathit{return}\ (\mathit{ps},\ t) \\
\mathit{tiExpr}\ \mathit{ce}\ \mathit{as}\ (\mathit{Ap}\ e\ f) & = & \textbf{do}\ (\mathit{ps},\ \mathit{te}) \leftarrow \mathit{tiExpr}\ \mathit{ce}\ \mathit{as}\ e \\
 & & \quad (\mathit{qs},\ \mathit{tf}) \leftarrow \mathit{tiExpr}\ \mathit{ce}\ \mathit{as}\ f \\
 & & \quad t \leftarrow \mathit{newTVar}\ \mathit{Star} \\
 & & \quad \mathit{unify}\ (\mathit{tf}\ \text{`}\mathit{fn}\text{`}\ t)\ \mathit{te} \\
 & & \quad \mathit{return}\ (\mathit{ps} +\!\!+\ \mathit{qs},\ t) \\
\mathit{tiExpr}\ \mathit{ce}\ \mathit{as}\ (\mathit{Let}\ \mathit{bg}\ e) & = & \textbf{do}\ (\mathit{ps},\ \mathit{as}') \leftarrow \mathit{tiBindGroup}\ \mathit{ce}\ \mathit{as}\ \mathit{bg} \\
 & & \quad (\mathit{qs},\ t) \leftarrow \mathit{tiExpr}\ \mathit{ce}\ (\mathit{as}' +\!\!+\ \mathit{as})\ e \\
 & & \quad \mathit{return}\ (\mathit{ps} +\!\!+\ \mathit{qs},\ t)
\end{array}
$$

The final case here, for *Let* expressions, uses the function *tiBindGroup* presented in Section 11.6.3, to generate a list of assumptions as' for the variables defined in bg. All of these variables are in scope when we calculate a type t for the body e, which also serves as the type of the whole expression.

11.4 Alternatives

The representation of function bindings in following sections uses *alternatives*, represented by values of type *Alt*:

$$\mathbf{type}\ Alt \quad = \quad ([Pat],\ Expr)$$

An *Alt* specifies the left and right hand sides of a function definition. With a more complete syntax for *Expr*, values of type *Alt* might also be used in the representation of lambda and case expressions.

For type inference, we begin by using *tiPats* to infer a type for each of the patterns, and to build a new list as' of assumptions for any bound variables, as described in Section 11.2. Next, we calculate the type of the body in the scope of the bound variables, and combine this with the types of each pattern to obtain a single (function) type for the whole *Alt*:

$$\begin{array}{lcl}
tiAlt & :: & Infer\ Alt\ Type \\
tiAlt\ ce\ as\ (pats,\ e) & = & \mathbf{do}\ (ps,\ as',\ ts) \leftarrow tiPats\ pats \\
 & & \quad\ \ (qs,\ t) \leftarrow tiExpr\ ce\ (as' +\!\!+\ as)\ e \\
 & & \quad\ \ return\ (ps +\!\!+\ qs,\ foldr\ fn\ t\ ts)
\end{array}$$

In practice, we will often run the typechecker over a list of alternatives, *alts*, and check that the returned type in each case agrees with some known type t. This process can be packaged up in the following definition:

$$\begin{array}{lcl}
tiAlts & :: & ClassEnv \to [Assump] \to [Alt] \to Type \to TI\ [Pred] \\
tiAlts\ ce\ as\ alts\ t & = & \mathbf{do}\ psts \leftarrow mapM\ (tiAlt\ ce\ as)\ alts \\
 & & \quad\ \ mapM\ (unify\ t)\ (map\ snd\ psts) \\
 & & \quad\ \ return\ (concat\ (map\ fst\ psts))
\end{array}$$

Although we do not need it here, the signature for *tiAlts* would allow an implementation to push the type argument inside the checking of each *Alt*, interleaving unification with type inference instead of leaving it to the end. This is essential in extensions like the support for rank-2 polymorphism in Hugs where explicit type information plays a key role. Even in an unextended Haskell implementation, this could still help to improve the quality of type error messages. Of course, we can still use *tiAlts* to infer a type from scratch. All this requires is that we generate and pass in a fresh type variable v in the parameter t to *tiAlts*, and then inspect the value of v under the current substitution when it returns.

11.5 From Types to Type Schemes

We have seen how lists of predicates are accumulated during type inference; now we will describe how those predicates are used to construct inferred types. This process is sometimes referred to as *generalization* because the goal is always to calculate the most general types that are possible. In a standard Hindley-Milner system, we can usually calculate most general types by quantifying over all relevant type variables that do not appear in the assumptions. In this section, we will describe how this process is modified to deal with the predicates in Haskell types.

To understand the basic problem, suppose that we have run the type checker over the body of a function h to obtain a list of predicates ps and a type t. At this point, to obtain the most general result, we could infer a type for h by forming the qualified type $qt = (ps :\Rightarrow t)$ and then quantifying over any variables in qt that do not appear in the assumptions. While this is permitted by the theory of qualified types, it is often not the best thing to do in practice. For example:

- The list ps can often be simplified using the context reduction process described in Section 7.4. This will also ensure that the syntactic restrictions of Haskell are met, requiring all predicates to be in head-normal form.
- Some of the predicates in ps may contain only 'fixed' variables (i.e., variables appearing in the assumptions), so including those constraints in the inferred type will not be of any use (Jones, 1992; Section 6.1.4). These predicates should be 'deferred' to the enclosing bindings.
- Some of the predicates in ps could result in *ambiguity*, and require defaulting to avoid a type error. This aspect of Haskell's type system will be described shortly in Section 11.5.1.

In this paper we use a function called *split* to address these issues. For the situation described previously where we have inferred a type t and a list of predicates ps for a function h, we can use *split* to rewrite and break ps into a pair $(ds,\ rs)$ of deferred predicates ds and 'retained' predicates rs. The predicates in rs will be used to form an inferred type $(rs :\Rightarrow t)$ for h, while the predicates in ds will be passed out as constraints to the enclosing scope. We use the following definition for *split*:

```
split             :: Monad m => ClassEnv -> [Tyvar] -> [Tyvar] -> [Pred]
                                  -> m ([Pred], [Pred])
split ce fs gs ps  = do ps' <- reduce ce ps
                        let (ds, rs)  =  partition (all (`elem` fs) . tv) ps'
                        rs' <- defaultedPreds ce (fs ++ gs) rs
                        return (ds, rs \\ rs')
```

In addition to a list of predicates ps, the *split* function is parameterized by two lists of type variables. The first, fs, specifies the set of 'fixed' variables, which are just the variables appearing free in the assumptions. The second, gs, specifies the set of variables over which we would like to quantify; for the example above, it would just be the variables in $(tv\ \ t \setminus\setminus fs)$. It is possible for ps to contain variables that are not in either fs or gs (and hence not in the parameter $(fs +\!\!+\ gs)$ that is passed to *defaultedPreds*). In Section 11.5.1 we will see that this is an indication of ambiguity.

There are three stages in the *split* function, corresponding directly to the three points listed previously. The first stage uses *reduce* to perform context reduction. The second stage uses the standard prelude function *partition* to identify the deferred predicates, ds; these are just the predicates in ps' that contain only fixed type variables. The third stage determines whether any of the predicates in rs should be eliminated using Haskell's defaulting mechanism, and produces a list of all such predicates in rs'. Hence the final set of retained predicates is produced by the expression $rs \setminus\setminus rs'$ in the last line of the definition.

11.5.1 Ambiguity and Defaults

In the terminology of Haskell (Peyton Jones & Hughes, 1999; Section 4.3.4), a type scheme $ps \Rightarrow t$ is *ambiguous* if ps contains generic variables that do not also appear in t. This condition is important because theoretical studies (Blott, 1991; Jones, 1992) have shown that, in the general case, we can only guarantee a well-defined semantics for a term if its most general type is not ambiguous. As a result, expressions with ambiguous types are considered ill-typed in Haskell and will result in a static error. The following definition shows a fairly typical example illustrating how ambiguity problems can occur:

```
stringInc x = show (read x + 1)
```

The intention here is that a string representation of a number will be parsed (using the prelude function `read`), incremented, and converted back to a string (using the prelude function `show`). But there is a genuine ambiguity because there is nothing to specify which type of number is intended, and because different choices can lead to different semantics. For example, `stringInc "1.5"` might produce a result of `"2.5"` if floating point numbers are used, or a parse error (or perhaps a result of `"2"`) if integers are used. This semantic ambiguity is reflected by a syntactic ambiguity in the inferred type of `stringInc`:

```
stringInc :: (Read a, Num a) => String -> String
```

(There is no `Show a` constraint here because `Show` is a superclass of `Num`.) A programmer can fix this particular problem quite easily by picking a particular type for `a`, and by adding an appropriate type annotation:

```
stringInc x = show (read x + 1 :: Int)
```

Practical experience suggests that ambiguities like this tend to occur quite infrequently in real Haskell code. Moreover, when ambiguities are detected, the error diagnostics that are generated can often be useful in guiding programmers to genuine problems in their code. However, the designers of Haskell felt that, in some situations involving numeric types—and particularly involving overloaded numeric literals—the potential for ambiguity was significant enough to become quite a burden on programmers. Haskell's `default` mechanism was therefore introduced as a pragmatic compromise that is convenient—because it automates the task of picking types for otherwise ambiguous variables—but also dangerous—because it involves making choices about the semantics of a program in ways that are not always directly visible to programmers. For this latter reason, the use of defaulting is restricted so that it will only apply under certain, fairly restrictive circumstances.

The remainder of this section explains in more detail how ambiguities in Haskell programs can be detected and, when appropriate, eliminated by a suitable choice of defaults. The first step is to identify any sources of ambiguity. Suppose, for example, that we are about to qualify a type with a list of predicates ps and that vs lists all known variables, both fixed and generic. An ambiguity occurs precisely if there is a type variable that appears in ps but not in vs (i.e., in $tv\ ps \setminus\setminus vs$). The goal of defaulting is to bind each ambiguous type variable v to a monotype t. The type t must be chosen so that all of the predicates in ps that involve v will be satisfied once t has been substituted for v. The following function calculates the list of ambiguous variables and pairs each one with the list of predicates that must be satisfied by any choice of a default:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Ambiguity} & = & (\mathit{Tyvar},\ [\mathit{Pred}]) \\ \\ \mathit{ambiguities} & :: & \mathit{ClassEnv} \to [\mathit{Tyvar}] \to [\mathit{Pred}] \to [\mathit{Ambiguity}] \\ \mathit{ambiguities}\ \mathit{ce}\ \mathit{vs}\ \mathit{ps} & = & [(v,\ \mathit{filter}\ (\mathit{elem}\ v\ .\ \mathit{tv})\ \mathit{ps}) \mid v \leftarrow \mathit{tv}\ \mathit{ps} \setminus\setminus \mathit{vs}] \end{array}$$

Given one of these pairs (v, qs), and as specified by the Haskell report (Peyton Jones & Hughes, 1999; Section 4.3.4), defaulting is permitted if, and only if, all of the following conditions are satisfied:

- All of the predicates in qs are of the form $\mathit{IsIn}\ c\ (\mathit{TVar}\ v)$ for some class c.
- At least one of the classes involved in qs is a standard numeric class. The list of these class names is provided by a constant:

$$\begin{array}{lcl} \mathit{numClasses} & :: & [\mathit{Id}] \\ \mathit{numClasses} & = & [\text{“}\texttt{Num}\text{”},\ \text{“}\texttt{Integral}\text{”},\ \text{“}\texttt{Floating}\text{”},\ \text{“}\texttt{Fractional}\text{”}, \\ & & \ \text{“}\texttt{Real}\text{”},\ \text{“}\texttt{RealFloat}\text{”},\ \text{“}\texttt{RealFrac}\text{”}] \end{array}$$

- All of the classes involved in qs are standard classes, defined either in the standard prelude or standard libraries. Again, the list of these class names is provided by a constant:

$$\begin{array}{lcl} \mathit{stdClasses} & :: & [\mathit{Id}] \\ \mathit{stdClasses} & = & [\text{“}\texttt{Eq}\text{”},\ \text{“}\texttt{Ord}\text{”},\ \text{“}\texttt{Show}\text{”},\ \text{“}\texttt{Read}\text{”},\ \text{“}\texttt{Bounded}\text{”},\ \text{“}\texttt{Enum}\text{”},\ \text{“}\texttt{Ix}\text{”}, \\ & & \ \text{“}\texttt{Functor}\text{”},\ \text{“}\texttt{Monad}\text{”},\ \text{“}\texttt{MonadPlus}\text{”}] \mathbin{+\!\!+} \mathit{numClasses} \end{array}$$

- That there is at least one type in the list of default types for the enclosing module that is an instance of all of the classes mentioned in qs. The first such type will be selected as the default. The list of default types can be obtained from a class environment by using the *defaults* function that was described in Section 7.2.

These conditions are captured rather more succinctly in the following definition, which we use to calculate the candidates for resolving a particular ambiguity:

$$\begin{array}{lcl} \mathit{candidates} & :: & \mathit{ClassEnv} \to \mathit{Ambiguity} \to [\mathit{Type}] \\ \mathit{candidates}\ \mathit{ce}\ (v,\ \mathit{qs}) & = & [t' \mid \textbf{let}\ \mathit{is} \ = \ [i \mid \mathit{IsIn}\ i\ t \leftarrow \mathit{qs}] \\ & & \qquad\qquad\ \ \mathit{ts} \ = \ [t \mid \mathit{IsIn}\ i\ t \leftarrow \mathit{qs}], \\ & & \qquad \mathit{all}\ ((\mathit{TVar}\ v) ==)\ \mathit{ts}, \\ & & \qquad \mathit{any}\ (\text{`}\mathit{elem}\text{`}\ \mathit{numClasses})\ \mathit{is}, \\ & & \qquad \mathit{all}\ (\text{`}\mathit{elem}\text{`}\ \mathit{stdClasses})\ \mathit{is}, \\ & & \qquad t' \leftarrow \mathit{defaults}\ \mathit{ce}, \\ & & \qquad \mathit{all}\ (\mathit{entail}\ \mathit{ce}\ [\,])\ [\mathit{IsIn}\ i\ t' \mid i \leftarrow \mathit{is}]] \end{array}$$

If *candidates* returns an empty list for a given ambiguity, then defaulting cannot be applied to the corresponding variable, and the ambiguity cannot be avoided. On the other hand, if the result is a non-empty list *ts*, then we will be able to substitute *head ts* for *v* and remove the predicates in *qs* from *ps*. The calculations for the defaulting substitution, and for the list of predicates that it eliminates follow very similar patterns, which we capture by defining them in terms of a single, higher-order function:

```
withDefaults      :: Monad m => ([Ambiguity] -> [Type] -> a)
                        -> ClassEnv -> [Tyvar] -> [Pred] -> m a
withDefaults f ce vs ps
  | any null tss  = fail "cannot resolve ambiguity"
  | otherwise     = return (f vps (map head tss))
   where vps = ambiguities ce vs ps
         tss = map (candidates ce) vps
```

The *withDefaults* function takes care of picking suitable defaults, and of checking whether there are any ambiguities that cannot be eliminated. If defaulting succeeds, then the list of predicates that can be eliminated is obtained by concatenating the predicates in each *Ambiguity* pair:

```
defaultedPreds :: Monad m => ClassEnv -> [Tyvar] -> [Pred] -> m [Pred]
defaultedPreds  = withDefaults (\vps ts -> concat (map snd vps))
```

In a similar way, the defaulting substitution can be obtained by zipping the list of variables together with the list of defaults:

```
defaultSubst :: Monad m => ClassEnv -> [Tyvar] -> [Pred] -> m Subst
defaultSubst  = withDefaults (\vps ts -> zip (map fst vps) ts)
```

One might wonder why the defaulting substitution is useful to us here; if the ambiguous variables don't appear anywhere in the assumptions or in the inferred types, then applying this substitution to those components would have no effect. In fact, we will only need *defaultSubst* at the top-level, when type inference for an entire module is complete (Peyton Jones & Hughes, 1999; Section 4.5.5, Rule 2). In this case, it is possible that Haskell's infamous 'monomorphism restriction' (see Section 11.6.2) may prevent generalization over some type variables. But Haskell does not allow the types of top-level functions to contain unbound type variables. Instead, any remaining variables are considered ambiguous, even if they appear in inferred types; the substitution is needed to ensure that they are bound correctly.

11.6 Binding Groups

Our last remaining technical challenge is to describe typechecking for binding groups. This area is neglected in most theoretical treatments of type inference, often being regarded as a simple exercise in extending basic ideas. In Haskell, at least, nothing could be further from the truth! With interactions between overloading, polymorphic recursion, and the mixing of both explicitly and implicitly typed bindings, this is the most complex, and most subtle component of type inference. We will start by describing the treatment of explicitly typed bindings and implicitly typed bindings as separate cases, and then show how these can be combined.

11.6.1 Explicitly Typed Bindings

The simplest case is for explicitly typed bindings, each of which is described by the name of the function that is being defined, the declared type scheme, and the list of alternatives in its definition:

$$\textbf{type}\ \mathit{Expl} \quad = \quad (\mathit{Id},\ \mathit{Scheme},\ [\mathit{Alt}])$$

Haskell requires that each Alt in the definition of a given identifier has the same number of left-hand side arguments, but we do not need to enforce that here.

Type inference for an explicitly typed binding is fairly easy; we need only check that the declared type is valid, and do not need to infer a type from first principles. To support the use of polymorphic recursion (Henglein, 1993; Kfoury *et al.*, 1993), we will assume that the declared typing for i is already included in the assumptions when we call the following function:

```
tiExpl :: ClassEnv → [Assump] → Expl → TI [Pred]
tiExpl ce as (i, sc, alts)
        = do (qs :⇒ t) ← freshInst sc
             ps ← tiAlts ce as alts t
             s ← getSubst
             let qs' = apply s qs
                 t'  = apply s t
                 fs  = tv (apply s as)
                 gs  = tv t' \\ fs
                 sc' = quantify gs (qs' :⇒ t')
                 ps' = filter (not . entail ce qs') (apply s ps)
             (ds, rs) ← split ce fs gs ps'
             if sc /= sc' then
                 fail "signature too general"
               else if not (null rs) then
                 fail "context too weak"
               else
                 return ds
```

This code begins by instantiating the declared type scheme sc and checking each alternative against the resulting type t. When all of the alternatives have been processed, the inferred type for i is $qs' :\Rightarrow t'$. If the type declaration is accurate, then this should be the same, up to renaming of generic variables, as the original type $qs :\Rightarrow t$. If the type signature is too general, then the calculation of sc' will result in a type scheme that is more specific than sc and an error will be reported.

In the meantime, we must discharge any predicates that were generated while checking the list of alternatives. Predicates that are entailed by the context qs' can be eliminated without further ado. Any remaining predicates are collected in ps' and passed as arguments to *split* along with the appropriate sets of fixed and generic variables. If there are any retained predicates after context reduction, then an error is reported, indicating that the declared context is too weak.

11.6.2 Implicitly Typed Bindings

Two complications occur when we deal with implicitly typed bindings. The first is that we must deal with groups of mutually recursive bindings as a single unit rather than inferring types for each binding one at a time. The second is Haskell's monomorphism restriction, which restricts the use of overloading in certain cases.

A single implicitly typed binding is described by a pair containing the name of the variable and a list of alternatives:

```
type Impl = (Id, [Alt])
```

The monomorphism restriction is invoked when one or more of the entries in a list of implicitly typed bindings is simple, meaning that it has an alternative with no left-hand side patterns. The following function provides a way to test for this:

```
restricted    :: [Impl] -> Bool
restricted bs  = any simple bs
  where simple (i, alts) = any (null . fst) alts
```

Type inference for groups of mutually recursive, implicitly typed bindings is described by the following function:

```
tiImpls          :: Infer [Impl] [Assump]
tiImpls ce as bs  = do ts <- mapM (\_ -> newTVar Star) bs
                       let is    = map fst bs
                           scs   = map toScheme ts
                           as'   = zipWith (:>:) is scs ++ as
                           altss = map snd bs
                       pss <- sequence (zipWith (tiAlts ce as') altss ts)
                       s <- getSubst
                       let ps' = apply s (concat pss)
                           ts' = apply s ts
                           fs  = tv (apply s as)
                           vss = map tv ts'
                           gs  = foldr1 union vss \\ fs
                       (ds, rs) <- split ce fs (foldr1 intersect vss) ps'
                       if restricted bs then
                           let gs'  = gs \\ tv rs
                               scs' = map (quantify gs' . ([] :=>)) ts'
                           in return (ds ++ rs, zipWith (:>:) is scs')
                         else
                           let scs' = map (quantify gs . (rs :=>)) ts'
                           in return (ds, zipWith (:>:) is scs')
```

In the first part of this process, we extend *as* with assumptions binding each identifier defined in *bs* to a new type variable, and use these to type check each alternative in each binding. This is necessary to ensure that each variable is used with the same type at every occurrence within the defining list of bindings. (Lifting this restriction makes type inference undecidable (Henglein, 1993; Kfoury *et al.*, 1993).) Next we use *split* to break the inferred predicates in ps' into a list of deferred predicates *ds* and retained predicates *rs*. The list *gs* collects all the generic variables that appear in one or more of the inferred types ts', but not in the list *fs* of fixed variables. Note that a different list is passed to *split*, including only variables that appear in *all* of the inferred types. This is important because all of those types will eventually be qualified by the same set of predicates, and we do not want any of the resulting type schemes to be ambiguous. The final step begins with a test to see if the monomorphism restriction should be applied, and then continues to calculate an assumption containing the principal types for each of the defined values. For an unrestricted binding, this is simply a matter of qualifying over the retained predicates in *rs* and quantifying over the generic variables in *gs*. If the binding group is restricted, then we must defer the predicates in *rs* as well as those in *ds*, and hence we can only quantify over variables in *gs* that do not appear in *rs*.

11.6.3 Combined Binding Groups

Haskell requires a process of *dependency analysis* to break down complete sets of bindings—either at the top-level of a program, or within a local definition—into the smallest possible groups of mutually recursive definitions, and ordered so that no group depends on the values defined in later groups. This is necessary to obtain the most general types possible. For example, consider the following fragment from a standard prelude for Haskell:

```
foldr f a (x:xs) = f x (foldr f a xs)
foldr f a []     = a
and xs           = foldr (&&) True xs
```

If these definitions were placed in the same binding group, then we would not obtain the most general possible type for `foldr`; all occurrences of a variable are required to have the same type at each point within the defining binding group, which would lead to the following type for `foldr`:

```
(Bool -> Bool -> Bool) -> Bool -> [Bool] -> Bool
```

To avoid this problem, we need only notice that the definition of `foldr` does not depend in any way on `&&`, and hence we can place the two functions in separate binding groups, inferring first the most general type for `foldr`, and then the correct type for `and`.

In the presence of explicitly typed bindings, we can refine the dependency analysis process a little further. For example, consider the following pair of bindings:

```
f   :: Eq a => a -> Bool
f x  = (x==x) || g True
g y  = (y<=y) || f True
```

Although these bindings are mutually recursive, we do not need to infer types for `f` and `g` at the same time. Instead, we can use the declared type of `f` to infer a type:

```
g   :: Ord a => a -> Bool
```

and then use this to check the body of `f`, ensuring that its declared type is correct.

Motivated by these observations, we will represent Haskell binding groups using the following datatype:

$$\textbf{type}\ \mathit{BindGroup} \quad = \quad ([\mathit{Expl}], [[\mathit{Impl}]])$$

The first component in each such pair lists any explicitly typed bindings in the group. The second component provides an opportunity to break down the list of any implicitly typed bindings into several smaller lists, arranged in dependency order. In other words, if a binding group is represented by a pair $(es, [is_1, ..., is_n])$, then the implicitly typed bindings in each is_i should depend only on the bindings in $es, is_1, \ldots, is_i$, and not on any bindings in is_j when $j > i$. (Bindings in es could depend on any of the bindings in the group, but will presumably depend on at least those in is_n, or else the group would not be minimal. Note also that if es is empty, then n must be 1.) In choosing this representation, we have assumed that dependency analysis has been carried out prior to type checking, and that the bindings in each group have been organized into values of type *BindGroup* as appropriate. In particular, by separating out implicitly typed bindings as much as possible, we can potentially increase the degree of polymorphism in inferred types. For a correct implementation of the semantics specified in the Haskell report, a simpler but less flexible approach is required: all implicitly typed bindings must be placed in a single list, even if a more refined decomposition would be possible. In addition, if the group is restricted, then we must also ensure that none of the explicitly typed bindings in the same *BindGroup* have any predicates in their type, even though this is not strictly necessary. With hindsight, these are restrictions that we might prefer to avoid in any future revision of Haskell.

A more serious concern is that the Haskell report does not indicate clearly whether the previous example defining `f` and `g` should be valid. At the time of writing, some implementations accept it, while others do not. This is exactly the kind of problem that can occur when there is no precise, formal specification! Curiously, however, the report does indicate that a modification of the example to include an explicit type for `g` would be illegal. This is a consequence of a throw-away comment specifying that all explicit type signatures in a binding group must have the same context up to renaming of variables (Peyton Jones & Hughes, 1999; Section 4.5.2). This is a syntactic restriction that can easily be checked prior to type checking. Our comments here, however, suggest that it is unnecessarily restrictive.

In addition to the function bindings that we have seen already, Haskell allows variables to be defined using pattern bindings of the form $pat = expr$. We do not need to deal directly with such bindings because they are easily translated into the simpler framework used in this paper. For example, a binding:

```
(x,y) = expr
```

can be rewritten as:

```
nv = expr
x  = fst nv
y  = snd nv
```

where `nv` is a new variable. The precise definition of the monomorphism restriction in Haskell makes specific reference to pattern bindings, treating any binding group that includes one as restricted. So it may seem that the definition of restricted binding groups in this paper is not quite accurate. However, if we use translations as suggested here, then it turns out to be equivalent: even if the programmer supplies explicit type signatures for `x` and `y` in the original program, the translation will still contain an implicitly typed binding for the new variable `nv`.

Now, at last, we are ready to present the algorithm for type inference of a complete binding group, as implemented by the following function:

```
tiBindGroup                  :: Infer BindGroup [Assump]
tiBindGroup ce as (es, iss)  =
  do let as'  =  [v :>: sc | (v, sc, alts) <- es]
     (ps, as'') <- tiSeq tiImpls ce (as' ++ as) iss
     qss <- mapM (tiExpl ce (as'' ++ as' ++ as)) es
     return (ps ++ concat qss, as'' ++ as')
```

The structure of this definition is quite straightforward. First we form a list of assumptions as' for each of the explicitly typed bindings in the group. Next, we use this to check each group of implicitly typed bindings, extending the assumption set further at each stage. Finally, we return to the explicitly typed bindings to verify that each of the declared types is acceptable. In dealing with the list of implicitly typed binding groups, we use the following utility function, which typechecks a list of binding groups and accumulates assumptions as it runs through the list:

```
tiSeq                    :: Infer bg [Assump] -> Infer [bg] [Assump]
tiSeq ti ce as []        =  return ([], [])
tiSeq ti ce as (bs : bss) =  do (ps, as') <- ti ce as bs
                                (qs, as'') <- tiSeq ti ce (as' ++ as) bss
                                return (ps ++ qs, as'' ++ as')
```

11.6.4 Top-level Binding Groups

At the top-level, a Haskell program can be thought of as a list of binding groups:

```
type Program  =  [BindGroup]
```

Even the definitions of member functions in class and instance declarations can be included in this representation; they can be translated into top-level, explicitly typed bindings. The type inference process for a program takes a list of assumptions giving the types of any primitives, and returns a set of assumptions for any variables.

```
tiProgram            :: ClassEnv -> [Assump] -> Program -> [Assump]
tiProgram ce as bgs  =  runTI $
                        do (ps, as') <- tiSeq tiBindGroup ce as bgs
                           s <- getSubst
                           rs <- reduce ce (apply s ps)
                           s' <- defaultSubst ce [] rs
                           return (apply (s'@@s) as')
```

This completes our presentation of the Haskell type system.

12 Conclusions

We have presented a complete Haskell program that implements a type checker for the Haskell language. In the process, we have clarified certain aspects of the current design, as well as identifying some ambiguities in the existing, informal specification.

The type checker has been developed, type-checked, and tested using the "Haskell 98 mode" of Hugs 98 (Jones & Peterson, 1999). The full program includes many additional functions, not shown in this paper, to ease the task of testing, debugging, and displaying results. We have also translated several large Haskell programs—including the Standard Prelude, the Maybe and List libraries, and the source code for the type checker itself—into the representations described in Section 11, and successfully passed these through the type checker. As a result of these and other experiments we have good evidence that the type checker is working as intended, and in accordance with the expectations of Haskell programmers.

We believe that this typechecker can play a useful role, both as a formal specification for the Haskell type system, and as a testbed for experimenting with future extensions.

Acknowledgments

This paper has benefited from feedback from Johan Nordlander, Tom Pledger, Lennart Augustsson, Stephen Eldridge, Tim Sheard, Andy Gordon, and from an anonymous referee. The research reported in this paper was supported, in part, by the USAF Air Materiel Command, contract # F19628-96-C-0161, and by the National Science Foundation grants CCR-9703218 and CCR-9974980.

References

Blott, Stephen M. (1991). *An approach to overloading with polymorphism*. Ph.D. thesis, Department of Computing Science, University of Glasgow.

Damas, L., & Milner, R. (1982). Principal type schemes for functional programs. *Pages 207–212 of: 9th Annual ACM Symposium on Principles of Programming Languages.*

Gaster, Benedict R., & Jones, Mark P. (1996). *A polymorphic type system for extensible records and variants*. Technical Report NOTTCS-TR-96-3. Computer Science, University of Nottingham.

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

Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, **146**, 29–60.

Jones, Mark P. (1992). *Qualified types: Theory and practice*. Ph.D. thesis, Programming Research Group, Oxford University Computing Laboratory. Published by Cambridge University Press, November 1994.

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

Kfoury, A.J., Tiuryn, J., & Urzyczyn, P. (1993). Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, **15**(2), 290–311.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**(3).

Peyton Jones, Simon, & Hughes, John (eds). (1999). *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language.* Available from `http://www.haskell.org/definition/`.

Peyton Jones, Simon, Jones, Mark, & Meijer, Erik. (1997). Type classes: Exploring the design space. *Proceedings of the second haskell workshop.*

Robinson, J.A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, **12**.

Wadler, P. (1992). The essence of functional programming (invited talk). *Pages 1–14 of: Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages.*

Wadler, P., & Blott, S. (1989). How to make *ad hoc* polymorphism less *ad hoc*. *Pages 60–76 of: Proceedings of 16th ACM Symposium on Principles of Programming Languages.*