Typing Haskell in Haskell

Mark P. Jones
Oregon Graduate Institute of Science and Technology
mpj@cse.ogi.edu

Haskell Workshop Version: September 1, 1999

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 type-checker, 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 compilers and interpreters accept the same programs as other implementations. The informal specification in the Haskell report [10] leaves too much room for confusion and misinterpretation. This leads to genuine discrepancies between implementations, as many 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 in the research literature, but often abstract on specific features that are Haskell-like, but not Haskell-exact, and do not describe the complete type system. Moreover, these papers often use disparate and unfamiliar technical notation and concepts that may be hard for some Haskell programmers to understand.
- 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 specific 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 Haskell implementors, programmers and researchers, and that it will be a first step in eliminating most of the problems described above.

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. (We would, however, 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 doesn't 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 the specification in this paper could easily be recast in a more 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 complex, but modestly sized Haskell programs.

[1] Throughout, we use 'Haskell' as an abbreviation for 'Haskell 98'.

Another goal for this paper was 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 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; by comparison, the core parts of the Hugs typechecker takes some 90+ pages of C code.

Regrettably, length restrictions have prevented us from including many examples in this paper to illustrate the definitions at each stage. For the same reason, definitions of a few 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 not included in the typeset version of this 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 establish the correctness of our presentation here in any rigorous manner. We hope that this paper will stimulate discussion in the Haskell community, and would expect to make changes to the specification as we work towards some kind of consensus.
- There is no shortage of proposed extensions to the Haskell type system, some of which 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, 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 [5, 9, 2] will be needed to understand the algorithms presented here. Although we have aimed to keep our 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!

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*
predicate	$p, q, \ldots$	*Pred*
– 'deferred'	$d, \ldots$	
– 'retained'	$r, \ldots$	
qualified type	$qt, \ldots$	*QualType*
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*

Figure 1: Notational Conventions

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* library:

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

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 new identifiers dynamically using the *enumId* function:

```
type Id     =  String
enumId      :: Int → Id
enumId n    =  "v" ++ show n
```

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 $Char \to Bool$; kinds of the form $k_1 \to k_2$ 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 $* \to *$. Here, we will represent kinds as values of the following datatype:

```
data Kind  =  Star | Kfun Kind Kind
              deriving Eq
```

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 [3, 7].

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 $k_1 \to k_2$ to a type of kind k_1 produces a type of kind k_2:

```
data Type   =  TVar Tyvar
            |  TCon Tycon
            |  TAp Type Type
            |  TGen Int
               deriving Eq

data Tyvar  =  Tyvar Id Kind
               deriving Eq

data Tycon  =  Tycon Id Kind
               deriving Eq
```

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

```
tChar   =  TCon (Tycon "Char" Star)
tArrow  =  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.

Types of the form *TGen n* represent 'generic', or quantified type variables; their role is described in Section 8.

We do not provide a representation for type synonyms, assuming instead that they have been fully expanded before typechecking. It is always possible for an implementation to do this 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 two helper functions. The first provides a way to construct function types:

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

The second introduces an overloaded function, *kind*, that can be used to determine the kind of a type variable, type constant, or type expression:

```
class HasKind t where
  kind  ::  t → Kind
instance HasKind Tyvar where
  kind (Tyvar v k)  =  k
instance HasKind Tycon where
  kind (Tycon v k)  =  k
instance HasKind Type where
  kind (TCon tc)    =  kind tc
  kind (TVar u)     =  kind u
  kind (TAp t _)    =  case (kind t) of
                          (Kfun _ k)  →  k
```

Most of the cases here are straightforward. Notice, however, that we can calculate the kind of an application $(TAp\ t\ t')$ 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—which are just finite functions, mapping type variables to types—play a major role in type inference. In this paper, we represent substitutions using association lists:

```
type Subst  =  [(Tyvar, 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 can be 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:

```
nullSubst  ::  Subst
nullSubst  =   []
```

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—or to anything containing type 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:

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

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, specified by $\mathit{apply}\ (s_1 @@ s_2) = \mathit{apply}\ s_1\ .\ \mathit{apply}\ s_2$, can be defined more concretely 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 a result of type *Maybe Subst*:

$$
\begin{array}{lcl}
\mathit{merge} & :: & \mathit{Subst} \to \mathit{Subst} \to \mathit{Maybe}\ \mathit{Subst} \\
\mathit{merge}\ s_1\ s_2 & = & \textbf{if}\ \mathit{agree}\ \textbf{then}\ \mathit{Just}\ s\ \textbf{else}\ \mathit{Nothing} \\
\quad \textbf{where}\ \mathit{dom}\ s & = & \mathit{map}\ \mathit{fst}\ s \\
\qquad\qquad s & = & s_1 +\!\!+\, s_2 \\
\qquad\qquad \mathit{agree} & = & \mathit{all}\ (\backslash v \to \mathit{apply}\ s_1\ (\mathit{TVar}\ v) == \\
 & & \qquad\qquad \mathit{apply}\ s_2\ (\mathit{TVar}\ v)) \\
 & & \qquad (\mathit{dom}\ s_1\ \text{`}\mathit{intersect}\text{`}\ \mathit{dom}\ s_2)
\end{array}
$$

It is easy to check that both (@@) and *merge* produce kind-preserving results from kind-preserving arguments.

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 also lead to the most general type. 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 carefully chosen to ensure that, if two types have any unifying substitutions, then they also have a most general unifier, which can be calculated by a simple variant of Robinson's algorithm [11]. One of the main 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 any other mechanism that allows reductions or rewriting in the type language, will often make unification undecidable, non-unitary (meaning that there may not be most general unifiers), or both. This, for example, is why it is not possible to 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:

$$
\begin{array}{lcl}
\mathit{mgu} & :: & \mathit{Type} \to \mathit{Type} \to \mathit{Maybe}\ \mathit{Subst} \\
\mathit{varBind} & :: & \mathit{Tyvar} \to \mathit{Type} \to \mathit{Maybe}\ \mathit{Subst}
\end{array}
$$

Both of these return results using *Maybe* because unification is a partial function. However, because *Maybe* is a monad, the programming task can be simplified by using Haskell's monadic do-notation. The main unification algorithm is described by *mgu*, which uses the structure of its arguments to guide the calculation:

$$
\begin{array}{lcl}
\mathit{mgu}\ (\mathit{TAp}\ l\ r)\ (\mathit{TAp}\ l'\ r') & = & \textbf{do}\ s_1 \leftarrow \mathit{mgu}\ l\ l' \\
 & & \qquad s_2 \leftarrow \mathit{mgu}\ (\mathit{apply}\ s_1\ r) \\
 & & \qquad\qquad\qquad (\mathit{apply}\ s_1\ r') \\
 & & \qquad \mathit{Just}\ (s_2 @@ s_1) \\
\mathit{mgu}\ (\mathit{TVar}\ u)\ t & = & \mathit{varBind}\ u\ t \\
\mathit{mgu}\ t\ (\mathit{TVar}\ u) & = & \mathit{varBind}\ u\ t \\
\mathit{mgu}\ (\mathit{TCon}\ \mathit{tc}_1)\ (\mathit{TCon}\ \mathit{tc}_2) & & \\
\quad \mid\ \mathit{tc}_1 == \mathit{tc}_2 & = & \mathit{Just}\ \mathit{nullSubst} \\
\mathit{mgu}\ t_1\ t_2 & = & \mathit{Nothing}
\end{array}
$$

[2] The typeset version of the symbol $\mapsto$ is written `+->` in the concrete syntax of Haskell.

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:

$$\begin{array}{lllll} \mathit{varBind}\ u\ t & \mid & t == \mathit{TVar}\ u & = & \mathit{Just}\ \mathit{nullSubst} \\ & \mid & u\ \text{`}\mathit{elem}\text{`}\ \mathit{tv}\ t & = & \mathit{Nothing} \\ & \mid & \mathit{kind}\ u == \mathit{kind}\ t & = & \mathit{Just}\ (u \mapsto t) \\ & \mid & \mathit{otherwise} & = & \mathit{Nothing} \end{array}$$

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 $\mathit{apply}\ s\ t_1 = t_2$. 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:

$$\mathit{match} \ :: \ \mathit{Type} \to \mathit{Type} \to \mathit{Maybe}\ \mathit{Subst}$$

Matching follows the same pattern as unification, except that it uses *merge* rather than @@ in the case for type applications, and it does not allow binding of variables in t_2:

$$\begin{array}{lll} \mathit{match}\ (\mathit{TAp}\ l\ r)\ (\mathit{TAp}\ l'\ r') & = & \textbf{do}\ \mathit{sl} \leftarrow \mathit{match}\ l\ l' \\ & & \quad\ \ \mathit{sr} \leftarrow \mathit{match}\ r\ r' \\ & & \quad\ \ \mathit{merge}\ \mathit{sl}\ \mathit{sr} \\ \mathit{match}\ (\mathit{TVar}\ u)\ t & & \\ \quad \mid\ \mathit{kind}\ u == \mathit{kind}\ t & = & \mathit{Just}\ (u \mapsto t) \\ \mathit{match}\ (\mathit{TCon}\ \mathit{tc}_1)\ (\mathit{TCon}\ \mathit{tc}_2) & & \\ \quad \mid\ \mathit{tc}_1 == \mathit{tc}_2 & = & \mathit{Just}\ \mathit{nullSubst} \\ \mathit{match}\ t_1\ t_2 & = & \mathit{Nothing} \end{array}$$

7 Predicates and Qualified Types

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]:

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

Predicates themselves consist of a class name, and a type:

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

Haskell's classes represent sets of types. For example, a predicate $\mathit{IsIn}\ c\ t$ asserts that t is a member of the class c. It would be easy to extend the *Pred* datatype to allow other forms of predicate, as is done with Trex records in Hugs [7]. 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.

[3] The typeset version of the symbol $:\Rightarrow$ is written `:=>` in the concrete syntax of Haskell, and corresponds directly to the `=>` symbol that is used in instance declarations and in types.

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

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

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{instance}\ \mathit{Types}\ \mathit{Pred}\ \textbf{where}} \\ \quad \mathit{apply}\ s\ (\mathit{IsIn}\ c\ t) & = & \mathit{IsIn}\ c\ (\mathit{apply}\ s\ t) \\ \quad \mathit{tv}\ (\mathit{IsIn}\ c\ t) & = & \mathit{tv}\ t \end{array}$$

7.1 Classes and Instances

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. We will assume that class names appearing in the original source code have been mapped to values of the following *Class* datatype prior to typechecking:

$$\begin{array}{lll} \textbf{data}\ \mathit{Class} & = & \mathit{Class}\ \{\mathit{name} :: \mathit{Id}, \\ & & \qquad\quad \mathit{super} :: [\mathit{Class}], \\ & & \qquad\quad \mathit{insts} :: [\mathit{Inst}]\} \\ \textbf{type}\ \mathit{Inst} & = & \mathit{Qual}\ \mathit{Pred} \end{array}$$

Values of type *Class* and *Inst* correspond to source level class and instance declarations, respectively. 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 member functions for the class, or their implementations in a particular instance.

A derived equality on *Class* is not useful because the data structures may be cyclic and so a test for structural equality might not terminate when applied to equal arguments. Instead, we use the *name* field to define an equality:

$$\begin{array}{l} \textbf{instance}\ \mathit{Eq}\ \mathit{Class}\ \textbf{where} \\ \quad c == c' \ = \ \mathit{name}\ c == \mathit{name}\ c' \end{array}$$

Apart from using a different keyword, Haskell class and instance declarations begin in the same way, with a clause of the form $\mathit{preds} \Rightarrow \mathit{pred}$ for some (possibly empty) 'context' *preds*, and a 'head' predicate *pred*. In a class declaration, the context is used to specify the immediate superclasses, which we represent more directly by the list of classes in the field *super*: If a type is an instance of a class c, then it must also be an instance of any superclasses of c. 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 $\mathit{bySuper}\ p$:

$$\begin{array}{l} \mathit{bySuper} \ :: \ \mathit{Pred} \to [\mathit{Pred}] \\ \mathit{bySuper}\ p@(\mathit{IsIn}\ c\ t) \\ \qquad\quad = \ p : \mathit{concat}\ (\mathit{map}\ \mathit{bySuper}\ \mathit{supers}) \\ \qquad\qquad \textbf{where}\ \mathit{supers} \ = \ [\mathit{IsIn}\ c'\ t \mid c' \leftarrow \mathit{super}\ c] \end{array}$$

The list $\mathit{bySuper}\ p$ may contain duplicates, but it will always be finite because restrictions in Haskell ensure that the superclass hierarchy is acyclic.

The final field in each *Class* structure, *insts*, is the list of instance declarations for that particular class. Each such instance declaration is represented by a clause $ps :\Rightarrow h$. Here, h is a predicate that describes the form of instances that the declaration can produce, while the context ps lists any constraints that it requires. We can use the following function to see if a particular predicate p can be deduced using a given instance. The result is either *Just ps*, where ps is a list of subgoals that must be established to complete the proof, or *Nothing* if the instance does not apply:

$$
\begin{array}{lcl}
\mathit{byInst} & :: & \mathit{Pred} \to \mathit{Inst} \to \mathit{Maybe}\ [\mathit{Pred}] \\
\mathit{byInst}\ p\ (ps :\Rightarrow h) & = & \mathbf{do}\ u \leftarrow \mathit{matchPred}\ h\ p \\
 & & \quad\ \mathit{Just}\ (\mathit{map}\ (\mathit{apply}\ u)\ ps)
\end{array}
$$

To see if an instance applies, we use one-way matching on predicates, which is implemented as follows:

$$
\begin{array}{lcl}
\mathit{matchPred} & :: & \mathit{Pred} \to \mathit{Pred} \to \mathit{Maybe}\ \mathit{Subst} \\
\multicolumn{3}{l}{\mathit{matchPred}\ (\mathit{IsIn}\ c\ t)\ (\mathit{IsIn}\ c'\ t')} \\
\quad \mid\ c == c' & = & \mathit{match}\ t\ t' \\
\quad \mid\ \mathit{otherwise} & = & \mathit{Nothing}
\end{array}
$$

We can find all the relevant instances for a given predicate $p = \mathit{IsIn}\ c\ t$ in $\mathit{insts}\ c$. So, if there are any ways to apply an instance to p, then we can find one using:

$$
\begin{array}{lcl}
\mathit{reducePred} & :: & \mathit{Pred} \to \mathit{Maybe}\ [\mathit{Pred}] \\
\mathit{reducePred}\ p@(\mathit{IsIn}\ c\ t) & = & \mathit{foldr}\ (|||)\ \mathit{Nothing}\ \mathit{poss} \\
\quad \mathbf{where}\ \mathit{poss} & = & \mathit{map}\ (\mathit{byInst}\ p)\ (\mathit{insts}\ c) \\
\qquad\quad \mathit{Nothing}|||y & = & y \\
\qquad\quad \mathit{Just}\ x|||y & = & \mathit{Just}\ x
\end{array}
$$

In fact, because Haskell prevents the definition of overlapping instances, we can be sure that, if *reducePreds* succeeds, then we have actually found the *only* applicable instance.

7.2 Entailment

The information provided by class and instance declarations can be combined to define an *entailment* relation on predicates. As in the theory of qualified types [6], we write $ps \Vdash p$ to indicate that the predicate p will hold whenever all of the predicates in ps are satisfied. To make this more concrete, we define the following function[4]:

$$
\begin{array}{lcl}
(\Vdash) & :: & [\mathit{Pred}] \to \mathit{Pred} \to \mathit{Bool} \\
ps \Vdash p & = & \mathit{any}\ (p\ `\mathit{elem}`)\ (\mathit{map}\ \mathit{bySuper}\ ps)\ || \\
 & & \mathbf{case}\ \mathit{reducePred}\ p\ \mathbf{of} \\
 & & \quad \mathit{Nothing} \quad \to \quad \mathit{False} \\
 & & \quad \mathit{Just}\ qs \quad \to \quad \mathit{all}\ (ps \Vdash)\ qs
\end{array}
$$

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—are enough to guarantee that tests for entailment will terminate. Completeness of the algorithm is also important: will $ps \Vdash p$ 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 such that $ps \Vdash q$. 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 [10, 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 $ps \Vdash q$, 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.

[4]The typeset version of the symbol $\Vdash$ is written `||-` in the concrete syntax of Haskell.

8 Type Schemes

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

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

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 enforced statically by a careful choice of the representation for types and type schemes. However, after considering several other alternatives, we eventually settled for the representation shown here because it allows for simple implementations of equality and substitution. For example, because the implementation of substitution on *Type* ignores *TGen* values, we can be sure that there will be no variable capture problems in the following definition:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Types}\ \mathit{Scheme}\ \mathbf{where}} \\
\quad \mathit{apply}\ s\ (\mathit{Forall}\ ks\ qt) & = & \mathit{Forall}\ ks\ (\mathit{apply}\ s\ qt) \\
\quad \mathit{tv}\ (\mathit{Forall}\ ks\ qt) & = & \mathit{tv}\ qt
\end{array}
$$

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

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

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 very 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.

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*:

```
toScheme   :: Type → Scheme
toScheme t  = Forall [] ([] :⇒ t)
```

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:

```
data Assump = Id :>: Scheme
```

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

```
instance Types Assump where
  apply s (i :>: sc) = i :>: (apply s sc)
  tv (i :>: sc)      = tv sc
```

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:

```
find       :: Id → [Assump] → Scheme
find i as   = head [sc | (i' :>: sc) ← as, i == i']
```

We do not make any allowance here for the possibility that the variable i might not appear in *as*, and assume instead that a previous compiler pass will have detected any occurrences of unbound variables.

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 [12]. 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 *error* 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:

```
newtype TI a = TI (Subst → Int → (Subst, Int, a))

instance Monad TI where
  return x   = TI (\s n → (s, n, x))
  TI c >>= f = TI (\s n →
                   let (s', m, x) = c s n
                       TI fx      = f x
                   in fx s' m)

runTI           :: TI a → a
runTI (TI c)     = result
  where (s, n, result) = c nullSubst 0
```

We provide two operations that deal with the current substitution: *getSubst* returns the current substitution, while *unify* extends it with a most general unifier of its arguments:

```
getSubst    :: TI Subst
getSubst     = TI (\s n → (s, n, s))

unify       :: Type → Type → TI ()
unify t1 t2  = do s ← getSubst
                  case mgu (apply s t1) (apply s t2) of
                    Just u  → extSubst u
                    Nothing → error "unification"
```

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*:

```
extSubst    :: Subst → TI ()
extSubst s'  = TI (\s n → (s'@@s, n, ()))
```

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 or fresh type variable of a specified kind:

```
newTVar   :: Kind → TI Type
newTVar k  = TI (\s n →
                 let v = Tyvar (enumId n) k
                 in (s, n + 1, TVar v))
```

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

```
freshInst               :: Scheme → TI (Qual Type)
freshInst (Forall ks qt) = do ts ← mapM newTVar ks
                              return (inst ts qt)
```

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*. Although we use it at only this one place, it is still convenient to build up the definition of *inst* using overloading.

```
class Instantiate t where
  inst :: [Type] → t → t
instance Instantiate Type where
  inst ts (TAp l r) = TAp (inst ts l) (inst ts r)
  inst ts (TGen n)  = ts !! n
  inst ts t         = t
instance Instantiate a ⇒ Instantiate [a] where
  inst ts = map (inst ts)
instance Instantiate t ⇒ Instantiate (Qual t) where
  inst ts (ps :⇒ t) = inst ts ps :⇒ inst ts t
instance Instantiate Pred where
  inst ts (IsIn c t) = IsIn c (inst ts t)
```

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 our typing rules are expressed in terms of the following type synonym:

```
type Infer e t = [Assump] → e → TI ([Pred], t)
```

In more theoretical treatments, it would not be surprising to see the rules expressed in terms of judgments $P \mid A \vdash e : t$, where P is a set of predicates, A is a set of assumptions, e is an expression, and t is a corresponding type [6]. Judgments like this can be thought of as 4-tuples, and the typing rules themselves just correspond to a 4-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:

```
data Literal = LitInt Integer
             | LitChar Char
```

Type inference for literals is straightforward. For characters, we just return *typeChar*. 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.

```
tiLit               :: Literal → TI ([Pred], Type)
tiLit (LitChar _)   = return ([], tChar)
tiLit (LitInt _)    = do v ← newTVar Star
                         return ([IsIn cNum v], v)
```

For this last case, we assume the existence of a constant *cNum* :: *Class* to represent the Haskell class *Num*, but, for reasons of space, we do not present the definition here. It is straightforward to add additional cases for Haskell's floating point and *String* literals.

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:

```
data Pat = PVar Id
         | PLit Literal
         | PCon Assump [Pat]
```

A *PVar i* pattern matches any value, and binds the result to the variable *i*. A *PLit l* pattern matches only the particular value denoted by the literal *l*. 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 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 of Haskell 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 it would need to be extended to account for patterns using labeled fields, and for $(n + k)$ patterns. Neither of these cause any substantial problems, but they do add a little 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 :: Pat → TI ([Pred], [Assump], Type)
```

Note that we do not need to pass in a list of assumptions here; by definition, any occurence 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.

```
tiPat (PVar i) = do v ← newTVar Star
                    return ([], [i :>: toScheme v], v)
```

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.

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

```
tiPat (PLit l) = do (ps, t) ← tiLit l
                    return (ps, [], t)
```

The case for constructed patterns is slightly more complex:

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

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.

$$
\begin{array}{l}
\mathit{tiPats} \qquad :: \ \ [\mathit{Pat}] \rightarrow \mathit{TI}\ ([\mathit{Pred}],\ [\mathit{Assump}],\ [\mathit{Type}]) \\
\mathit{tiPats}\ \mathit{pats} \ \ = \\
\quad \mathbf{do}\ \mathit{psasts} \leftarrow \mathit{mapM}\ \mathit{tiPat}\ \mathit{pats} \\
\qquad \mathbf{let}\ \mathit{ps} \ = \ [p \mid (\mathit{ps},\ \_,\ \_) \leftarrow \mathit{psasts},\ p \leftarrow \mathit{ps}] \\
\qquad\quad\ \ \mathit{as} \ = \ [a \mid (\_,\ \mathit{as},\ \_) \leftarrow \mathit{psasts},\ a \leftarrow \mathit{as}] \\
\qquad\quad\ \ \mathit{ts} \ = \ [t \mid (\_,\ \_,\ t) \leftarrow \mathit{psasts}] \\
\qquad \mathit{return}\ (\mathit{ps},\ \mathit{as},\ \mathit{ts})
\end{array}
$$

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

Our next step is to describe type inference for expressions, represented by the *Expr* datatype:

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

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.

Haskell has a much richer syntax of expressions, but they can all be translated into *Expr* values. For example, a lambda expression like `\x->e` can be rewritten using a local definition as `let f x = e in f`, where `f` is a new variable. Similar translations are used for case expressions.

Type inference for expressions is quite straightforward:

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

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*:

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

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 building a new list as' of assumptions for any bound variables, and use it to infer types for each of the patterns, 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*:

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

In practice, we will often need to 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:

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

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 prominent role. Even in an unextended Haskell implementation, this could still help to improve the quality of type error messages.

11.5 Context Reduction

We have seen how lists of predicates are accumulated during type inference. In this section, we will describe how those predicates are used to construct inferred types. The Haskell report [10] 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.

To understand the basic problem, suppose that we have run *tiExpr* over the body of a function f to obtain a set of predicates ps and a type t. At this point we could infer a type for f 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 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 $n \geq 0$). Predicates that do not fit this pattern must be broken down using *reducePred*. In some cases, this will result in predicates being eliminated. In others, where *reducePred* fails, it will indicate that a predicate is unsatisfiable, and will trigger an error diagnostic.
- Some of the predicates in ps may be repeated or, more generally, entailed by the other members of ps. These predicates can safely be deleted, leading to smaller and simpler inferred types.
- 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 [6, Section 6.1.4]. These predicates should be 'deferred' to the enclosing bindings.
- Some of the predicates in ps could be ambiguous, and might require defaulting to avoid a type error.

To deal with all of these issues, we use a process of *context reduction* whose purpose is to compute, from a given set of predicates ps, a set of 'deferred' predicates ds and a set of 'retained' predicates rs. Only retained predicates will be included in inferred types. The complete process is described by the following function:

$$
\begin{array}{l}
reduce\ \ ::\ \ [Tyvar] \to [Tyvar] \to [Pred] \to ([Pred], [Pred]) \\
reduce\ fs\ gs\ ps\ \ =\ \ (ds, rs') \\
\quad \textbf{where}\ (ds, rs)\ \ =\ \ split\ fs\ ps \\
\qquad\qquad\ rs'\ \ =\ \ useDefaults\ (fs +\!\!+\ gs)\ rs
\end{array}
$$

The first stage of this algorithm, which we call context splitting, is implemented by *split* and is described in Section 11.5.1. Its purpose is to separate the deferred predicates from the retained predicates, using *reducePred* as necessary. The second stage implemented by *useDefaults*, is described in Section 11.5.2. Its purpose is to eliminate ambiguities in the retained predicates, whenever possible. The *fs* and *gs* parameters specify appropriate sets of 'fixed' and 'generic' type variables, respectively. The former is just the set of variables appearing free in the assumptions, while the latter is the set of variables over which we would like to quantify. Any variable in ps that is not in either fs or gs may cause ambiguity, as we describe in Section 11.5.2.

11.5.1 Context Splitting

We will describe the process of splitting a context as the composition of three functions, each corresponding to one of the bulletted items at the beginning of Section 11.5.

$$
\begin{array}{lll}
split & :: & [Tyvar] \to [Pred] \to ([Pred], [Pred]) \\
split\ fs & = & partition\ (all\ (\text{`}elem\text{`}\ fs)\ .\ tv) \\
 & . & simplify\ [\,] \\
 & . & toHnfs
\end{array}
$$

The first stage of this pipeline, implemented by *toHnfs*, uses *reducePred* to break down any inferred predicates into the form that Haskell requires:

$$
\begin{array}{lll}
toHnfs & :: & [Pred] \to [Pred] \\
toHnfs & = & concat\ .\ map\ toHnf
\end{array}
$$

$$
\begin{array}{l}
toHnf\ \ ::\ \ Pred \to [Pred] \\
toHnf\ p\ \ = \\
\quad \textbf{if}\ inHnf\ p \\
\qquad \textbf{then}\ [p] \\
\qquad \textbf{else case}\ reducePred\ p\ \textbf{of} \\
\qquad\qquad Nothing \ \to\ error\ \text{“}\texttt{context reduction}\text{”} \\
\qquad\qquad Just\ ps \ \to\ toHnfs\ ps
\end{array}
$$

The name *toHnfs* is motivated by similarities with the concept of *head-normal forms* in λ-calculus. The test to determine whether a given predicate is in the appropriate form is implemented by the following function:

$$
\begin{array}{l}
inHnf\ \ ::\ \ Pred \to Bool \\
inHnf\ (IsIn\ c\ t)\ \ =\ \ hnf\ t \\
\quad \textbf{where}\ hnf\ (TVar\ v)\ \ =\ \ True \\
\qquad\qquad\ hnf\ (TCon\ tc)\ \ =\ \ False \\
\qquad\qquad\ hnf\ (TAp\ t\ \_)\ \ =\ \ hnf\ t
\end{array}
$$

The second stage of the pipeline uses information about superclasses to eliminate redundant predicates. More precisely, if the list produced by *toHnfs* contains some predicate

p, then we can eliminate any occurrence of a predicate from *bySuper* p in the rest of the list. As a special case, this also means that we can eliminate any repeated occurrences of p, which always appears as the first element in *bySuper* p. This process is implemented by the *simplify* function, using an accumulating parameter to bulid up the final result:

$$\begin{array}{lcl} \mathit{simplify} & :: & [\mathit{Pred}] \to [\mathit{Pred}] \to [\mathit{Pred}] \\ \mathit{simplify}\ \mathit{rs}\ [\,] & = & \mathit{rs} \\ \mathit{simplify}\ \mathit{rs}\ (p : \mathit{ps}) & = & \mathit{simplify}\ (p : (\mathit{rs} \setminus\setminus \mathit{qs}))\ (\mathit{ps} \setminus\setminus \mathit{qs}) \\ \quad \textbf{where}\ \mathit{qs} & = & \mathit{bySuper}\ p \\ \qquad\quad \mathit{rs} \setminus\setminus \mathit{qs} & = & [r \mid r \leftarrow \mathit{rs},\ r\ \text{`}\mathit{notElem}\text{`}\ \mathit{qs}] \end{array}$$

Note that we have used a modified version of the $(\setminus\setminus)$ operator; with the standard Haskell semantics for $(\setminus\setminus)$, we could not guarantee that all duplicate entries would be removed.

The third stage of context reduction uses *partition* to separate deferred predicates—i.e., those containing only fixed variables—from retained predicates. The set of fixed variables is passed in as the *fs* argument to *split*.

11.5.2 Applying Defaults

A type scheme $P \Rightarrow t$ is said to be *ambiguous* if P contains generic variables that do not also appear in t. From theoretical studies [1, 6], we know that we cannot guarantee a well-defined semantics for any term with an ambiguous type, which is why Haskell will not allow programs containing such terms. In this section, we describe the mechanisms that are used to detect ambiguity, and the defaulting mechanism that it uses to try to eliminate ambiguity.

Suppose 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*. To determine whether defaults can be applied, we compute a triple $(v,\ qs,\ ts)$ for each ambiguous variable v. In each case, *qs* is the list of predicates in *ps* that involve v, and *ts* is the list of types that could be used as a default value for v:

$$\begin{array}{l} \mathit{ambig}\ \ ::\ \ [\mathit{Tyvar}] \to [\mathit{Pred}] \to [(\mathit{Tyvar},\ [\mathit{Pred}],\ [\mathit{Type}])] \\ \mathit{ambig}\ \mathit{vs}\ \mathit{ps} \\ \qquad =\ \ [(v,\ \mathit{qs},\ \mathit{defs}\ v\ \mathit{qs}) \mid \\ \qquad\qquad\quad v \leftarrow \mathit{tv}\ \mathit{ps} \setminus\setminus \mathit{vs}, \\ \qquad\qquad\quad \textbf{let}\ \mathit{qs}\ \ =\ \ [p \mid p \leftarrow \mathit{ps},\ v\ \text{`}\mathit{elem}\text{`}\ \mathit{tv}\ p]] \end{array}$$

If the *ts* component of any one of these triples turns out to be empty, then defaulting cannot be applied to the corresponding variable, and the ambiguity cannot be avoided. On the other hand, if *ts* is non-empty, then we will be able to substitute *head ts* for v and remove the predicates in *qs* from *ps*.

Given one of these triples $(v,\ qs,\ ts)$, and as specified by the Haskell report [10, 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 *IsIn* c (*TVar* v) for some class c.
- All of the classes involved in *qs* are standard classes, defined either in the standard prelude or standard libraries. We assume that the list of these classes is provided by a constant *stdClasses* :: [*Class*].
- At least one of the classes involved in *qs* is a numeric class. Again, we assume that the list of these classes is provided by a constant *numClasses* :: [*Class*].
- 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 in *qs*. We assume that this list of types is provided in a constant *defaults* :: [*Type*].

These conditions are captured rather more succinctly in the following definition, which we use to calculate the third component of each triple:

$$\begin{array}{lcl} \mathit{defs} & :: & \mathit{Tyvar} \to [\mathit{Pred}] \to [\mathit{Type}] \\ \mathit{defs}\ v\ \mathit{qs} & = & [t \mid \mathit{all}\ ((\mathit{TVar}\ v) ==)\ \mathit{ts}, \\ & & \quad\ \mathit{all}\ (\text{`}\mathit{elem}\text{`}\ \mathit{stdClasses})\ \mathit{cs}, \\ & & \quad\ \mathit{any}\ (\text{`}\mathit{elem}\text{`}\ \mathit{numClasses})\ \mathit{cs}, \\ & & \quad\ t \leftarrow \mathit{defaults}, \\ & & \quad\ \mathit{and}\ [[\,] \Vdash \mathit{IsIn}\ c\ t \mid c \leftarrow \mathit{cs}]] \\ \quad \textbf{where}\ \mathit{cs} & = & [c \mid (\mathit{IsIn}\ c\ t) \leftarrow \mathit{qs}] \\ \qquad\qquad \mathit{ts} & = & [t \mid (\mathit{IsIn}\ c\ t) \leftarrow \mathit{qs}] \end{array}$$

The defaulting process can now be described by the following function, which generates an error if there are any ambiguous type variables that cannot be defaulted:

$$\begin{array}{lcl} \mathit{useDefaults} & :: & [\mathit{Tyvar}] \to [\mathit{Pred}] \to [\mathit{Pred}] \\ \mathit{useDefaults}\ \mathit{vs}\ \mathit{ps} & & \\ \quad \mid\ \mathit{any}\ \mathit{null}\ \mathit{tss} & = & \mathit{error}\ \text{“}\mathtt{ambiguity}\text{”} \\ \quad \mid\ \mathit{otherwise} & = & \mathit{ps} \setminus\setminus \mathit{ps}' \\ \qquad \textbf{where}\ \mathit{ams} & = & \mathit{ambig}\ \mathit{vs}\ \mathit{ps} \\ \qquad\qquad\quad \mathit{tss} & = & [\mathit{ts} \mid (v,\ \mathit{qs},\ \mathit{ts}) \leftarrow \mathit{ams}] \\ \qquad\qquad\quad \mathit{ps}' & = & [p \mid (v,\ \mathit{qs},\ \mathit{ts}) \leftarrow \mathit{ams},\ p \leftarrow \mathit{qs}] \end{array}$$

A modified version of this process is required at the top-level, when type inference for an entire module is complete [10, Section 4.5.5, Rule 2]. In this case, *any* remaining type variables are considered ambiguous, and we need to arrange for defaulting to return a substitution mapping any such variables to their defaulted types:

$$\begin{array}{lcl} \mathit{topDefaults} & :: & [\mathit{Pred}] \to \mathit{Maybe}\ \mathit{Subst} \\ \mathit{topDefaults}\ \mathit{ps} & & \\ \quad \mid\ \mathit{any}\ \mathit{null}\ \mathit{tss} & = & \mathit{Nothing} \\ \quad \mid\ \mathit{otherwise} & = & \mathit{Just}\ (\mathit{zip}\ \mathit{vs}\ (\mathit{map}\ \mathit{head}\ \mathit{tss})) \\ \qquad \textbf{where}\ \mathit{ams} & = & \mathit{ambig}\ [\,]\ \mathit{ps} \\ \qquad\qquad\quad \mathit{tss} & = & [\mathit{ts} \mid (v,\ \mathit{qs},\ \mathit{ts}) \leftarrow \mathit{ams}] \\ \qquad\qquad\quad \mathit{vs} & = & [v \mid (v,\ \mathit{qs},\ \mathit{ts}) \leftarrow \mathit{ams}] \end{array}$$

11.6 Binding Groups

The main technical challenge in this paper is to describe typechecking for binding groups. This area is neglected in most theoretical treatments of 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:

```
type Expl = (Id, Scheme, [Alt])
```

Haskell requires that each *Alt* in the definition of any given value has the same number of arguments in each left-hand side, but we do not need to enforce that restriction 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 [4, 8], we will assume that the declared typing for i is already included in the assumptions when we call the following function:

```
tiExpl                 :: [Assump] -> Expl -> TI [Pred]
tiExpl as (i, sc, alts) =
   do (qs :=> t) <- freshInst sc
      ps <- tiAlts as alts t
      s <- getSubst
      let qs'      = apply s qs
          t'       = apply s t
          ps'      = [p | p <- apply s ps, not (qs' ||- p)]
          fs       = tv (apply s as)
          gs       = tv t' \\ fs
          (ds, rs) = reduce fs gs ps'
          sc'      = quantify gs (qs' :=> t')
      if sc /= sc' then
          error "signature too general"
        else if not (null rs) then
          error "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 *reduce* 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 simple way to test for this condition:

```
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 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 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) = reduce 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 [4, 8].) Next we use the process of context reduction 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 *reduce*, 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:

type *BindGroup* $=$ ([*Expl*], [[*Impl*]])

The first component in each such pair lists any explicitly typed bindings in the group, while the second component breaks down any remaining bindings into a sequence of smaller, implicitly typed binding groups, arranged in dependency order. In choosing our representation for the abstract syntax, 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* in an appropriate manner. For a correct implementation of the semantics specified in the Haskell report, we must place all of the implicitly typed bindings in a single group, even if a more refined decomposition would be possible. In addition, if that group is restricted, then we must also check that none of the explicitly typed bindings in the same *BindGroup* have any predicates in their type. With hindsight, these seem like strange restrictions that we might prefer to avoid in any further 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 [10, 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, at first glance, 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:

$$
\begin{array}{l}
\mathit{tiBindGroup} \qquad\qquad :: \quad \mathit{Infer\ BindGroup}\ [\mathit{Assump}] \\
\mathit{tiBindGroup}\ \mathit{as}\ (\mathit{es},\ \mathit{iss}) \ = \\
\quad \textbf{do let}\ \mathit{as}' \ = \ [v :>: \mathit{sc} \mid (v,\ \mathit{sc},\ \mathit{alts}) \leftarrow \mathit{es}] \\
\qquad (\mathit{ps},\ \mathit{as}'') \leftarrow \mathit{tiSeq}\ \mathit{tiImpls}\ (\mathit{as}' +\!\!+\ \mathit{as})\ \mathit{iss} \\
\qquad \mathit{qs} \leftarrow \mathit{mapM}\ (\mathit{tiExpl}\ (\mathit{as}'' +\!\!+\ \mathit{as}' +\!\!+\ \mathit{as}))\ \mathit{es} \\
\qquad \mathit{return}\ (\mathit{ps} +\!\!+\ \mathit{concat}\ \mathit{qs},\ \mathit{as}'' +\!\!+\ \mathit{as}')
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{tiSeq} \;\; :: \;\; \mathit{Infer}\ \mathit{bg}\ [\mathit{Assump}] \rightarrow \mathit{Infer}\ [\mathit{bg}]\ [\mathit{Assump}] \\
\mathit{tiSeq}\ \mathit{ti}\ \mathit{as}\ [\,] \\
\qquad = \;\; \mathit{return}\ ([\,], [\,]) \\
\mathit{tiSeq}\ \mathit{ti}\ \mathit{as}\ (\mathit{bs} : \mathit{bss}) \\
\qquad = \;\; \mathbf{do}\ (\mathit{ps}, \mathit{as}') \leftarrow \mathit{ti}\ \mathit{as}\ \mathit{bs} \\
\qquad\qquad\quad (\mathit{qs}, \mathit{as}'') \leftarrow \mathit{tiSeq}\ \mathit{ti}\ (\mathit{as}' +\!\!+ \mathit{as})\ \mathit{bss} \\
\qquad\qquad\quad \mathit{return}\ (\mathit{ps} +\!\!+ \mathit{qs}, \mathit{as}'' +\!\!+ \mathit{as}')
\end{array}
$$

11.6.4 Top-level Binding Groups

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

$$\mathbf{type}\ \mathit{Program} \;\; = \;\; [\mathit{BindGroup}]$$

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.

$$
\begin{array}{l}
\mathit{tiProgram} \qquad\quad :: \;\; [\mathit{Assump}] \rightarrow \mathit{Program} \rightarrow [\mathit{Assump}] \\
\mathit{tiProgram}\ \mathit{as}\ \mathit{bgs} \;\; = \;\; \mathit{runTI}\ \$ \\
\quad \mathbf{do}\ (\mathit{ps}, \mathit{as}') \leftarrow \mathit{tiSeq}\ \mathit{tiBindGroup}\ \mathit{as}\ \mathit{bgs} \\
\qquad s \leftarrow \mathit{getSubst} \\
\qquad \mathbf{let}\ ([\,], \mathit{rs}) \;\; = \;\; \mathit{split}\ [\,]\ (\mathit{apply}\ s\ \mathit{ps}) \\
\qquad \mathbf{case}\ \mathit{topDefaults}\ \mathit{rs}\ \mathbf{of} \\
\qquad\quad \mathit{Just}\ s' \quad \rightarrow \;\; \mathit{return}\ (\mathit{apply}\ (s'@@s)\ \mathit{as}') \\
\qquad\quad \mathit{Nothing} \;\; \rightarrow \;\; \mathit{error}\ \text{“}\texttt{top-level ambiguity}\text{”}
\end{array}
$$

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 [7]. 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 Lennart Augustsson, Stephen Eldridge, Tim Sheard, Andy Gordon, and from an anonymous referee. The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-96-C-0161.

References

[1] S. M. Blott. *An approach to overloading with polymorphism*. PhD thesis, Department of Computing Science, University of Glasgow, July 1991. (draft version).

[2] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming languages*, pages 207–212, Albuquerque, NM, January 1982.

[3] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Computer Science, University of Nottingham, November 1996.

[4] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[5] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[6] M. P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.

[7] M. P. Jones and J. C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs/`.

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

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

[10] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, February 1999. Available from `http://www.haskell.org/definition/`.

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

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