A system of constructor classes: overloading and implicit higher-order polymorphism[†]

MARK P. JONES[*]

Department of Computer Science, Yale University, PO Box 208285, New Haven, CT 06520-8285, USA
(e-mail: jones-mark@cs.yale.edu*)*

Abstract

This paper describes a flexible type system that combines overloading and higher-order polymorphism in an implicitly typed language using a system of *constructor classes*—a natural generalization of type classes in Haskell. We present a range of examples to demonstrate the usefulness of such a system. In particular, we show how constructor classes can be used to support the use of monads in a functional language. The underlying type system permits higher-order polymorphism but retains many of the attractive features that have made Hindley/Milner type systems so popular. In particular, there is an effective algorithm that can be used to calculate principal types without the need for explicit type or kind annotations. A prototype implementation has been developed providing, amongst other things, the first concrete implementation of monad comprehensions known to us at the time of writing.

Capsule Review

The author describes a generalization of the Haskell type class concept. He allows not only to abstract from types in type expressions, but also from type constructors by using higher-order type variables. This leads to some kind of higher-order polymorphism. However, pure higher-order polymorphism is, as the author illustrates, hardly useful in practice, because it is too general. Type classes allow to constrain the higher-order type variables to particular type constructors. Jones calls these higher-order type classes *constructor classes*. They enable to overload a function on different type constructors.

To show the practical usefulness of this feature is the main topic of the paper. In particular, the author shows how constructor classes support the use of monads in a functional language. While in former presentations of monad applications the authors always had to rename the monad functions, depending on the particular monad, in this paper the functions are overloaded by using constructor classes. The author presents for example a general notation for monad comprehensions which generalizes the Haskell list comprehensions. This is especially

† A shorter version of this paper was presented at the *Conference on Functional Programming Languages and Computer Architecture, FPCA '93*, Copenhagen, Denmark, June 1993. This version expands on the conference paper, including a wider range of programming examples, and also some minor corrections.

[*] Authors current address is: Department of Computer Science, University of Nottingham, University Park, Nottingham, NG7 2RD, UK (email: mpj@cs.nott.ac.uk)

interesting because the comprehension syntax is concise and often used in mathematical descriptions.

Besides practical applications of constructor classes the author gives an overview of the formal treatment of the type system. He sketches a type inference algorithm that calculates principal types, without requiring any type annotations. This feature enables a powerful, but still manageable programming language.

1 Introduction

There have been many theoretical studies of higher-order polymorphism. In contrast with the type systems of core-ML and the polymorphic λ-calculus, this allows the type of a term to include variables that represent arbitrary type constructors of a particular kind. However, despite the increase in expressiveness, few practical programming languages have adopted these ideas. This is particularly true for implicitly-typed languages that rely on type inference rather than the explicit type annotations used in most formal treatments of higher-order systems. For example, the Standard ML module system (MacQueen, 1984) provides much of the power of higher-order polymorphism, but only when expressed in the module language itself (which requires explicit typing), not directly in the underlying core language.

One reason for this apparent lack of interest is that pure higher-order polymorphism is too general for practical applications. For example, it is hard to think of any useful functions with type $\forall a.\forall m.a \to m\ a$, because such a function must be able to produce values of type $m\ a$ for any type constructor m. The only possibility is the function $\lambda x.\bot$ which, apart from having almost no practical use, can be treated as having the more general type $\forall a.\forall b.a \to b$ without the need for higher-order polymorphism.

In fact, there are similar examples involving only ML-style polymorphism. For example, a function to test for membership in a list should not be treated as having type $\forall a.a \to List\ a \to Bool$ because it relies on the ability to compare values of type a, and there is no way to define a computable equality function for all such a, in particular, for those involving function spaces. However, some form of polymorphism is still appropriate, since the membership function could certainly be used for any types on which equality has been defined. In Haskell (Hudak *et al.*, 1992), this is dealt with using type classes, writing the type as $\forall a.Eq\ a \Rightarrow a \to List\ a \to Bool$. Equality types provide a similar solution for Standard ML (Milner *et al.*, 1990).

This paper describes an extension of the Haskell type system based on the notion of *constructor classes*, a natural generalization of type classes. While retaining the implicit typing of Haskell, the system supports a combination of higher-order polymorphism and overloading. A prototype implementation has been developed and has allowed us to explore a wide range of applications of constructor classes, some of which are described in the first part of the paper (Sections 2 and 3). For readers with an interest in more technical details, a second part (Sections 4 and 5) concentrates on a theoretical treatment of the type system. Finally, Section 6 concludes, and outlines areas for future work.

For related work on higher-order polymorphism, we refer the reader to Barendregt (1991), which provides a general framework for describing a variety of type systems, including some with higher-order polymorphism, and provides useful pointers to the literature. The type system we study here is perhaps closest to Barendregt's $\lambda\underline{\omega}$ or $\lambda\omega$, except that these systems are explicitly typed and, as a result, they are considerably more expressive.

2 An overloaded *map* function

Many functional programs use the *map* function to apply a function to each of the elements in a given list. The type and definition of this function as given in the Haskell standard prelude (Hudak *et al.*, 1992) are as follows:

$$\begin{array}{lcl} map & :: & (a \to b) \to [a] \to [b] \\ map\ f\ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\ x : map\ f\ xs \end{array}$$

It is well known that the *map* function satisfies the familiar laws:

$$\begin{array}{lcl} map\ id & = & id \\ map\ f\ .\ map\ g & = & map\ (f\ .\ g) \end{array}$$

A category theorist will recognize these observations as indicating that there is a functor from types to types whose object part maps any given type a to the list type $[a]$ and whose arrow part maps each function $f :: a \to b$ to the function $map\ f ::$ $[a] \to [b]$. A functional programmer will recognize that similar constructions are also used with a wide range of other data types, as illustrated by the following examples:

$$\begin{array}{lcl} \textbf{data}\ Tree\ a & = & Leaf\ a \mid Tree\ a :\hat{}: Tree\ a \\ mapTree & :: & (a \to b) \to (Tree\ a \to Tree\ b) \\ mapTree\ f\ (Leaf\ x) & = & Leaf\ (f\ x) \\ mapTree\ f\ (l :\hat{}: r) & = & mapTree\ f\ l :\hat{}: mapTree\ f\ r \\ & & \\ \textbf{data}\ Maybe\ a & = & Just\ a \mid Nothing \\ mapMaybe & :: & (a \to b) \to (Maybe\ a \to Maybe\ b) \\ mapMaybe\ f\ (Just\ x) & = & Just\ (f\ x) \\ mapMaybe\ f\ Nothing & = & Nothing \end{array}$$

Each of these functions has a similar type to that of *map* and satisfies similar functor laws. With this in mind, it seems a shame that we have to use different names for each of these variants.

A more attractive solution would allow the use of a single name, *map*, relying on the types of the objects involved to determine which particular version of the *map* function is required in a given situation. For example, it is clear that $map\ (1+)\ [1,2,3]$ should be a list, calculated using the original *map* function on lists, while $map\ (1+)\ (Just\ 1)$ should evaluate to $Just\ 2$ using *mapMaybe*.

Unfortunately, in a language using standard Hindley/Milner type inference, there is no way in which to assign a type to the *map* function that would allow it to be used in this way. Furthermore, even if typing were not an issue, use of the *map* function would be rather limited unless some additional mechanism was provided to allow the definition to be extended to include new datatypes perhaps distributed across a number of distinct program modules.

2.1 *An attempt to define* map *using type classes*

The ability to use a single function symbol with an interpretation that depends upon the type of its arguments is commonly known as *overloading*. While some authors dismiss overloading as a purely syntactic convenience, this is certainly not the case in Haskell, which has a flexible type system that supports both parametric polymorphism and overloading based on a system of *type classes* (Wadler and Blott, 1989). One of the most attractive features of this system is that, although each primitive overloaded operator will require a separate definition for each different argument type, there is no need for these to be in the same module.

Type classes in Haskell can be thought of as sets of types. The standard example is the class *Eq* that includes precisely those types whose elements can be compared using the (==) function. A simple definition might be:

$$\begin{array}{l} \textbf{class}\ Eq\ a\ \textbf{where} \\ \quad (==)\ ::\ a \rightarrow a \rightarrow Bool \end{array}$$

The equality operator can then be treated as having any of the types in the set $\{ a \rightarrow a \rightarrow Bool \mid a \in Eq \}$. The elements of a type class are defined by a collection of *instance declarations* which may be distributed across a number of distinct program modules. For the type class *Eq*, these would typically include definitions of equality for integers, characters, lists, pairs and user-defined datatypes. Only a single definition is required for functions defined either directly or indirectly in terms of overloaded primitives. For example, assuming a collection of instances as above, the *member* function defined by:

$$\begin{array}{lcl} member & :: & Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\ member\ x\ [\,] & = & False \\ member\ x\ (y : ys) & = & x == y\ \ ||\ member\ x\ ys \end{array}$$

can be used to test for membership in a list of integers, characters, lists, pairs, etc. See Hudak and Fasel (1992) and Wadler and Blott (1989) for further details about the use of type classes.

Unfortunately, the system of type classes is not sufficiently powerful to give a satisfactory treatment for the *map* function; to do so would require a class *Map* and a type expression $m(t)$ involving the type variable t such that the set $S = \{ m(t) \mid t \in Map \}$ includes (at least) the types:

$$\begin{array}{l} (a \rightarrow b) \rightarrow ([a] \rightarrow [b]) \\ (a \rightarrow b) \rightarrow (Tree\ a \rightarrow Tree\ b) \\ (a \rightarrow b) \rightarrow (Maybe\ a \rightarrow Maybe\ b) \end{array}$$

for arbitrary types a and b. The only possibility is to take $m(t) = t$ and choose *Map* as the set of types S for which the *map* function is required:

```
class Map t where  map  ::  t

instance Map ((a → b) → ([a] → [b])) where    ...
instance Map ((a → b) → (Tree a → Tree b)) where    ...
instance Map ((a → b) → (Maybe a → Maybe b)) where    ...
```

This syntax is not permitted in Haskell but, even if it were, it does not give a sufficiently accurate characterization of the type of *map*. For example, the principal type of *map v . map u* would be

$$(Map\ (a \rightarrow c \rightarrow e), Map\ (b \rightarrow e \rightarrow d)) \Rightarrow c \rightarrow d$$

where a and b are the types of u and v, respectively. This is complicated and does not enforce the condition that u and v have function types. Furthermore, the type is *ambiguous*; the type variable e does not appear to the right of the $\Rightarrow$ symbol or in the assumptions, and hence there is no way to find its value from the context in which the term is used. Under these conditions, we cannot guarantee a well-defined semantics for this expression; see Jones (1992b), for example. Other attempts to define the *map* function, for example using multiple parameter type classes, have also failed for essentially the same reasons.

2.2 A solution using constructor classes

A much better approach is to notice that each of the types for which the *map* function is required is of the form:

$$(a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)$$

The variables a and b here represent arbitrary types, while f ranges over the set of type constructors for which a suitable *map* function has been defined. In particular, we would expect to include the list constructor (which we will write as *List*), *Tree* and *Maybe* as elements of this set which, motivated by our earlier comments, we will call *Functor*. With only a small extension to the Haskell syntax for type classes, this can be described by:

```
class Functor f where
  map  ::  (a → b) → (f a → f b)

instance Functor List where
  map f []       =  []
  map f (x : xs) =  f x : map f xs

instance Functor Tree where
  map f (Leaf x)   =  Leaf (f x)
  map f (l :^: r)  =  map f l :^: map f r

instance Functor Maybe where
  map f (Just x)  =  Just (f x)
  map f Nothing   =  Nothing
```

Functor is our first example of a *constructor class*. The following extract, taken from a session with the Gofer system (Jones, 1991), which includes support for constructor classes, illustrates how the definitions for *Functor* work in practice:

```
? map (1+) [1,2,3]
[2, 3, 4]
? map (1+) (Leaf 1 :^: Leaf 2)
Leaf 2 :^: Leaf 3
? map (1+) (Just 1)
Just 2
```

Furthermore, by specifying the type of *map* more precisely, we avoid the ambiguity problems mentioned above. For example, the expression $map\ v\ .\ map\ u$ has principal type $Functor\ f \Rightarrow f\ a \rightarrow f\ c$, provided that u has type $(a \rightarrow b)$ and that v has type $(b \rightarrow c)$. To see how this type is obtained, notice that, from the type of *map*, u and v must have function types of the form $(a \rightarrow b)$ and $(d \rightarrow c)$. Thus $map\ u$ has a type of the form $(f\ a \rightarrow f\ b)$ and $map\ v$ has a type of the form $(g\ d \rightarrow g\ c)$ for some instances f and g of the *Functor* class. For the composition of these two functions to be well formed, the range of the first, $f\ b$, must be the same as the domain of the second, $g\ d$. Hence $f = g$, $b = d$ and the composition has type $f\ a \rightarrow f\ c$, for any instance f of *Functor*.

2.3 The kind system

Each instance of *Functor* can be thought of as a function from types to types. It would be nonsense to allow the type *Int* of integers to be an instance of *Functor*, since the type $(a \rightarrow b) \rightarrow (Int\ a \rightarrow Int\ b)$ is obviously not well-formed. To avoid unwanted cases like this, we have to ensure that all of the elements in any given class are of the same kind. To this end, we formalize the notion of *kind*, writing $*$ for the kind of all types and $\kappa_1 \rightarrow \kappa_2$ for the kind of a constructor that takes something of kind κ_1 and returns something of kind κ_2. This choice of notation is motivated by Barendregt's description of generalized type systems (Barendregt, 1991). Instead of type expressions, we use a language of constructors given by:

$$\begin{array}{rcll} C & ::= & \chi & \textit{constants} \\ & | & a & \textit{variables} \\ & | & C\ C' & \textit{applications} \end{array}$$

This corresponds very closely to the way that most type expressions are already written in Haskell. For example, *Maybe a* is an application of the constant *Maybe* to the variable a. Each constructor constant has a corresponding kind. For example, writing $(\rightarrow)$ for the function space constructor and $(,)$ for pairing we have:

$$\begin{array}{lcl} Int,\ Float,\ () & :: & * \\ List,\ Maybe & :: & * \rightarrow * \\ (\rightarrow),\ (,) & :: & * \rightarrow * \rightarrow * \end{array}$$

The kinds of constructor applications are described by the rule:

$$\frac{C :: \kappa' \to \kappa \qquad C' :: \kappa'}{C\ C' :: \kappa}$$

The task of checking that a given type expression is well-formed can now be reformulated as the task of checking that a given constructor expression has kind $*$. In a similar way, all of the elements of a constructor class must have the same kind; for example, a constructor class constraint of the form *Functor* f is valid only if f is a constructor expression of kind $* \to *$.

The language of constructors is essentially a system of combinators without any reduction rules. It follows that standard techniques can be used to infer the kinds of constructor variables, constants introduced by new datatype definitions and the kind of constructors in any particular class. The important point is that there is no need—and indeed, in our current implementation, no opportunity—for the programmer to supply kind information explicitly. We regard this as a significant advantage, since it means that the programmer can avoid much of the complexity that might otherwise result from the need to annotate type expressions with kinds. The process of *kind inference* is described in more detail in Section 5.

The use of kinds is perhaps the most important aspect of our system, providing a loose semantic characterization of the elements in a constructor class. This is in contrast to the system of *parametric type classes* described by Chen *et al.* (1992), which addresses issues similar to those in this paper but relies on a more syntactic approach that involves a process of normalization. Note also that our system includes Haskell type classes as a special case; a type class is simply a constructor class for which each instance has kind $*$.

3 Monads as an application of constructor classes

Motivated by the work of Moggi (1989) and Spivey (1990), Wadler (1990; 1992) proposed a style of functional programming based on the use of *monads*. While the theory of monads had already been widely studied in the context of abstract category theory (MacLane, 1971), Wadler introduced the idea that monads could be used as a practical method for modelling so-called 'impure' features in a purely functional programming language. This section illustrates how constructor classes can be used to build a framework for programming in this style, identifying particular families of monads, and supporting an implementation of Wadler's notation for *monad comprehensions*.

3.1 A framework for programming with monads

One useful way to think about monads is as a means of representing computations. If m is a monad, then an object of type $(m\ a)$ represents a computation that is expected to produce a result of type a. The choice of monad reflects the possible use of particular programming language features as part of the computation; simple examples include state, exceptions and input/output. The distinction between computations of type *m a* and values of type *a* reflects the fact that the use of programming language features is a property of the computation itself, and not of the result that it produces.

Every monad provides at least two operations. First, there must be some way to return a result from a computation. We will represent this using a function:

```
result :: a → m a
```

with the intention that *result e* is the computation that returns the value *e* with no further effect. Note that the *result* function corresponds to the *unit* function in Wadler's presentations.

Second, there must be some way to combine computations. Taking the approach outlined by Wadler (1992), this might be described using a function:

```
bind :: m a → (a → m b) → m b
```

Writing *bind* as an infix operator, a simple way to understand an expression of the form *c* '*bind*' *f* is as a computation that runs *c*, passes its result *x* of type *a* to *f*, and runs the resulting computation *f x* to obtain a final result of type *b*. In many cases, this corresponds to the sequencing of one computation after another.

The description above leads us to the following definition for a constructor class of monads:

```
class Functor m ⇒ Monad m where
  result :: a → m a
  bind   :: m a → (a → m b) → m b
```

The expression *Functor m* ⇒ *Monad m* in the first line of the class declaration defines *Monad* as a subclass of *Functor* ensuring that, for any given monad, there will also be a corresponding instance of the overloaded *map* function. We should also mention that, for any monad *m*, the *result* and *bind* operators are expected to satisfy some simple algebraic laws that are not reflected in the class declaration above. This is described, for example, in Wadler (1992). A slightly more concise formulation of these properties, expressed in terms of an auxiliary function called *Kleisli composition*, will be given in Section 3.5.

One interesting application of monads is to model programs that make use of an internal state. Computations of this kind can be represented by functions of type $s \rightarrow (a, s)$, often referred to as *state transformers*, mapping an initial state to a pair containing the result and final state. To get this into the appropriate form for the system of constructor classes described in this paper, we introduce a new datatype:

```
data State s a = ST (s → (a,s))
```

The functor and monad structures for state transformers are as follows:

```
instance Functor (State s) where
  map f (ST st)  =  ST (\s → let  (x,s')  =  st s
                               in   (f x,s'))
instance Monad (State s) where
  result x   =  ST (\s → (x,s))
  m `bind` f =  ST (\s0 → let  ST m'   =  m
                                (x,s1)  =  m' s0
                                ST f'   =  f x
                                (y,s2)  =  f' s1
                          in    (y,s2))
```

For the benefit of those unfamiliar with this notation, we should mention that an expression of the form \x → *e* is just the Haskell concrete syntax for the lambda expression $\lambda x.e$. Notice that the *State* constructor has kind $* \to * \to *$ so that, for any state type *s*, the constructor *State s* has kind $* \to *$ as required for an instance of these classes. There is no need to assume a fixed state type as in Wadler's papers.

From a user's point of view, the most interesting properties of a monad are described, not by the *result* and *bind* operators, but by the additional operations that it supports. The following examples are often useful when working with state monads. The first can be used to 'run' a program given an initial state and discarding the final state, while the second might be used to implement an integer counter in a *State Int* monad:

```
startingWith               ::  State s a → s → a
ST m `startingWith` v      =   fst (m v)

incr                       ::  State Int Int
incr                       =   ST (\s → (s,s + 1))
```

To illustrate the use of state monads, consider the task of labelling each of the nodes in a binary tree with distinct integer values. One simple definition is:

```
label        ::  Tree a → Tree (a,Int)
label tree   =   fst (lab tree 0)
 where lab (Leaf n) c   =  (Leaf (n,c),c + 1)
       lab (l :^: r) c  =  (l' :^: r',c'')
                           where (l',c')   =  lab l c
                                 (r',c'')  =  lab r c'
```

This uses an explicit counter, the second parameter to *lab*, and great care must be taken to ensure that the appropriate counter value is used in each part of the program; simple errors, such as writing *c* in place of *c'* in the last line, are easily made but can be hard to detect.

One way in which to avoid these problems is to write the definition in a monadic style:

```
label          ::  Tree a → Tree (a, Int)
label tree     =   lab tree `startingWith` 0
 where lab (Leaf n)   =  incr                 `bind` \c →
                         result (Leaf (n,c))
       lab (l :^: r)  =  lab l                `bind` \l' →
                         lab r                `bind` \r' →
                         result (l' :^: r')
```

The layout used here follows Wadler (1992), and has a natural reading, with the result of each computation being passed on to subsequent steps by the *bind* operator. While this is a little longer than the previous version, the use of monad operations ensures that the correct counter value is passed from one part of the program to the next. There is no need to mention explicitly that a state monad is required: the use of *startingWith* and the initial value *0* are sufficient to determine the monad *State Int* needed for the *bind* and *result* operators. It is not necessary to distinguish between different versions of the monad operators *bind* and *result* as in Wadler (1992), or to rely on the use of explicit type declarations.

As another example, one of the most frequently used monads in functional programming is the list constructor, whose monad structure is described by the following declaration:

```
instance Monad List where
  result x      =  [x]
  xs `bind` f   =  concat (map f xs)
```

The *concat* function used in the definition of *bind* is the standard Haskell function for concatenating a list of lists.

As a model for computations, lists can be used to describe programs that produce multiple results, corresponding to a simple form of non-determinism. We will illustrate this with a simple algorithm for enumerating the permutations of a list. First, consider the task of choosing an arbitrary permutation of a non-empty list. One way in which to do this is to pick the first element of the permutation from the values given, and then choose a permutation of the remaining values. The list of all permutations can be obtained by repeating this process for all possible choices:

```
perm      ::  Eq a ⇒ [a] → [[a]]
perm []   =   result []
perm xs   =   pick xs            `bind` \x →
              perm (xs \\ [x])   `bind` \rest →
              result (x : rest)
```

Notice that this program uses lists in two ways, both as a data structure and as a monad. The simplest way to define the *pick* function used here would be the identity function on lists. Other choices of *pick* can be used, with a corresponding effect on the order in which the permutations are generated. For example, *pick = reverse* can be used to generate the permutations in reverse order.

Readers familiar with the notation of list comprehensions may have expected the last line in the definition of *perm* to be written as:

$$perm\ xs \quad = \quad [(x : rest) \mid x \leftarrow pick\ xs, rest \leftarrow perm\ (xs \setminus\setminus [x])]$$

It is no coincidence that this looks so similar to the first version using *bind* and *result*; we will see in Section 3.3 that the notation of list comprehensions can be interpreted more generally in other monads.

The *Tree* and *Maybe* datatypes defined in Section 2 can also be used as monads. For example, the monad structure for the *Maybe* datatype is given by the following declaration:

```
instance Monad Maybe where
  result             =  Just
  Just x `bind` f    =  f x
  Nothing `bind` f   =  Nothing
```

This monad can be used to describe computations which may produce a result, but which may also fail if, for example, an error condition occurs. We will see applications for similar monads in later sections of this paper. The *Maybe* monad is also used extensively in Spivey (1990), motivated by the problem of simplifying algebraic expressions using rewriting rules.

3.2 Using defaults

While the *bind* function described in the previous section is often convenient, it is not the only way to describe the combination of computations in a monad. For example, Wadler (1990) uses a *join* function as an alternative to *bind*. In the current framework, we can define *join* as a polymorphic function that can be used with any monad:

```
join      ::  Monad m => m (m a) -> m a
join xs   =   xs `bind` id
```

In some cases, it can be more convenient to define the structure of a monad using the *join* function instead of *bind*. One way to deal with this is to change the definition of *Monad* to include both of these functions:

```
class Functor m => Monad m where
  result     ::  a -> m a
  bind       ::  m a -> (a -> m b) -> m b
  join       ::  m (m a) -> m a

  x `bind` f  =  join (map f x)
  join x      =  x `bind` id
```

The default definitions for *bind* and *join* in the last two lines of the declaration describe how each of these functions may be defined in terms of the other. By including default definitions for both, only one of these functions, together with a suitable definition for *result*, is required to completely define an instance of *Monad*.

This is often quite convenient, for example, allowing us to define the monad structure for lists more concisely as:

```
instance Monad List where
  result x   =  [x]
  join       =  concat
```

However, it would be an error to omit definitions for both operators, since the default definitions are clearly circular. It is also important to realize that such errors will not be detected by the compiler.

The current implementation also supports the use of default instance declaration that can be used to define an instance of a class when no other suitable definition is included in a program. Once again, the convenience that this feature offers must be tempered with careful use. For example, in the class declarations above, we have explicitly specified that every instance of *Monad* must also be an instance of *Functor*. This is a reasonable assumption because the *result* and *bind* functions can be used to obtain a suitable definition for *map* if necessary; this construction can be expressed using the declaration:

```
instance Monad m => Functor m where
  map f m   =   m `bind` \x -> result (f x)
```

This default declaration can sometimes save the programmer from the need to specify both the *Functor* and *Monad* structure for a particular type constructor. However, since the definition for *map* provided here depends upon the *bind* function in the *Monad* class, we must be careful to ensure that *bind* does not in turn depend on *map* so as to avoid the possibility of circular definitions. Unfortunately, this means that neither of the two definitions for *Monad List* given above can be used in this way; we must either give an explicit definition for *Functor List*, or rewrite the definition for *Monad List* in the form:

```
instance Monad List where
  result x           =  [x]
  [] `bind` f        =  []
  (x : xs) `bind` f  =  f x ++ (xs `bind` f)
```

In some sense, the process of looping through the individual elements of a list is an essential part of both the *Functor* and *Monad* structures for *List*.

The examples in this section suggest that, while default definitions can be quite useful, they can also cause some awkward problems. In practice, it may be better to exclude *join* from the definition of the *Monad* class, or to extend the compiler with static checks that can be used to detect and avoid some careless errors.

3.3 Monad comprehensions

Several functional programming languages provide support for list comprehensions, enabling some common forms of computation with lists to be written in a concise form resembling the conventional mathematical notation for set comprehensions.

As described by Wadler (1990), the comprehension notation can be generalized to arbitrary monads, of which the list constructor is just one special case. In Wadler's notation, a monad comprehension is written using the syntax of a list comprehension with a superscript to indicate the monad in which the comprehension is to be interpreted. This is a little awkward, and makes the notation less powerful than might be hoped, since each comprehension is restricted to a particular monad. Using the overloaded operators described in Section 3.1, we have implemented a more flexible form of monad comprehension that relies on overloading rather than superscripts.

In our system, a monad comprehension is an expression of the form $[e \mid gs]$ where e is an expression and gs is a list of generators of the form $p \leftarrow exp$. As a special case, if gs is empty then the comprehension $[e \mid gs]$ is written as $[e]$. The implementation of monad comprehensions is based on the following translation of the comprehension notation in terms of the *result* and *bind* operators described in Section 3.1:

$$\begin{aligned} [e] &= \mathit{result}\ e \\ [e \mid p \leftarrow exp,\ gs] &= exp\ \text{`}\mathit{bind}\text{`}\ \backslash p \rightarrow [e \mid gs] \end{aligned}$$

In this notation, the *label* function from Section 3.1 can be rewritten as:

$$\begin{aligned} &\mathit{label} && :: \ \mathit{Tree}\ a \rightarrow \mathit{Tree}\ (a, \mathit{Int}) \\ &\mathit{label}\ \mathit{tree} && = \ \mathit{lab}\ \mathit{tree}\ \text{`}\mathit{startingWith}\text{`}\ 0 \\ &\quad \textbf{where}\ \mathit{lab}\ (\mathit{Leaf}\ n) && = \ [\mathit{Leaf}\ (n,c) \mid c \leftarrow \mathit{incr}] \\ &\quad\phantom{\textbf{where}}\ \mathit{lab}\ (l :\hat{}: r) && = \ [l' :\hat{}: r' \mid l' \leftarrow \mathit{lab}\ l,\ r' \leftarrow \mathit{lab}\ r] \end{aligned}$$

Applying the translation rules for monad comprehensions to this definition yields the previous definition in terms of *result* and *bind*. The principal advantage of the comprehension syntax is that it is often more concise and, in the author's opinion, sometimes more attractive.

3.4 Monads with a zero

Anyone familiar with the use of list comprehensions will know that it is also possible to include boolean guards in addition to generators in the definition of a list comprehension. Once again, Wadler showed that this was also possible in the more general setting of monad comprehensions, so long as we restrict such comprehensions to monads that include a special element *zero* satisfying a small number of laws. This can be dealt with in our framework by defining a subclass of *Monad*:

$$\begin{aligned} &\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}0\ m\ \textbf{where} \\ &\quad \mathit{zero} \ :: \ m\ a \end{aligned}$$

For example, both the *List* and *Maybe* monads have *zero* elements, given by the empty list and *Nothing* respectively:

$$\textbf{instance}\ \mathit{Monad}0\ \mathit{List}\ \textbf{where}\ \mathit{zero} \ = \ []$$

$$\textbf{instance}\ \mathit{Monad}0\ \mathit{Maybe}\ \textbf{where}\ \mathit{zero} \ = \ \mathit{Nothing}$$

Note that there are also some monads that do not have a zero element and hence cannot be defined as instances of *Monad0*. The *State s* monads described in Section 3.1 are a simple example of this.

Working in a monad with a zero, a comprehension involving a boolean guard can be implemented using the translation:

$$[\,e \mid \mathit{guard},\ \mathit{gs}\,] \quad = \quad \textbf{if}\ \mathit{guard}\ \textbf{then}\ [\,e \mid \mathit{gs}\,]\ \textbf{else}\ \mathit{zero}$$

Notice that, as far as the type system is concerned, the use of *zero* in the translation of a comprehension such as $[\,e \mid x \leftarrow xs,\ \mathit{guard}\,]$ automatically captures the restriction to monads with a zero. There is no need to introduce any additional mechanism to deal with this.

The inclusion of a *zero* element also allows a slightly different translation for generators in comprehensions:

$$\begin{array}{lcl} [\,e \mid p \leftarrow \mathit{exp},\ \mathit{gs}\,] & = & \mathit{exp}\ \text{`}\mathit{bind}\text{`}\ f \\ & & \quad \begin{array}{lcl} \textbf{where}\ f\ p & = & [\,e \mid \mathit{gs}\,] \\ f\ \_ & = & \mathit{zero} \end{array} \end{array}$$

This corresponds directly to the semantics of list comprehensions in Haskell 1.2 (Hudak *et al.*, 1992). The only time when there is any difference between this and the translation in Section 3.3 is when p is a *refutable* pattern which may not always match values generated by *exp*. For example, using the original translation, the expression $[\,x \mid [x] \leftarrow [[1],[\,],[2]]\,]$ evaluates to $[1]{+\!\!+}\bot$, whereas the corresponding list comprehension gives $[1, 2]$. To preserve the Haskell semantics for list comprehensions, the current implementation always uses the second translation to implement a generator containing a refutable pattern. Cases where a refutable pattern is required without being restricted to monads with a zero are easily dealt with by rewriting the generator as $\tilde{}p \leftarrow \mathit{exp}$; in Haskell, any pattern p can be treated as an irrefutable pattern by rewriting it as $\tilde{}p$. An alternative approach that we experimented with in earlier versions of this work (Jones, 1992c) is to use a slightly different syntax for monad comprehensions so that there is no clash with the semantics of Haskell list comprehensions.

3.5 Kleisli composition

For any monad, we can define a *Kleisli composition* operator, similar to the usual composition of functions except that it also takes care of 'side-effects'. The general definition is as follows:

$$\begin{array}{lcl} (@@) & :: & \mathit{Monad}\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c) \\ f\ @@\ g & = & \mathit{join}\ .\ \mathit{map}\ f\ .\ g \end{array}$$

Categorists may recognize this from the standard construction of the Kleisli category of a monad (MacLane, 1971, §VI.5).

For example, in a monad of the form *State s*, the expression $f@@g$ denotes a state transformer in which the final state of the computation associated with g is used as the initial state for the computation associated with f. More precisely, for this particular kind of monad, the general definition given above is equivalent to:

$$\begin{array}{lllll} (f @@ g)\ x & = & ST\ (\backslash s_0 \rightarrow \textbf{let} & ST\ g' & = & g\ x \\ & & & (y, s_1) & = & g'\ s_0 \\ & & & ST\ f' & = & f\ y \\ & & & (z, s_2) & = & f'\ s_1 \\ & & \quad\quad\quad\ \ \textbf{in} & (z, s_2)) & & \end{array}$$

On the other hand, in the *List* monad, the Kleisli composition operator can be expressed using a list comprehension as:

$$(f @@ g)\ x \quad = \quad [z \mid y \leftarrow g\ x, z \leftarrow f\ y]$$

(In fact, if we interpret the right-hand side as a monad comprehension, then this is equivalent to the general definition of (@@) given above!) The greatest advantage of a general definition is that there is no need to construct new definitions of (@@) for every different monad. On the other hand, if specific definitions were required for some instances, perhaps in the interests of efficiency, we could simply include (@@) as a member function of *Monad* and use the generic definition as a default implementation.

Apart from practical applications, Kleisli composition can also be used to describe the properties of a monad. We mentioned, in Section 3.1, that the member functions of the *Monad* class are expected to satisfy some algebraic laws. An equivalent formulation using Kleisli composition is that, for any monad and any f, g and h of suitable types:

$$f @@ result = f, \qquad result @@ g = g, \qquad (f @@ g) @@ h = f @@ (g @@ h).$$

In other words, Kleisli composition is associative with *result* as both a left and right identity. This suggests an alternative formulation for the *Monad* class, with *result* and (@@) as member functions. The *bind* and *join* operators are easily defined in terms of (@@) using the equations x *'bind'* f = $(f @@ id)\,x$ and *join* = *id*@@*id*. Note that two distinct instances of the polymorphic *id* function are used in this definition of *join*; *id* is an identity of standard functional composition, but not of Kleisli composition. Given the simplicity of the laws above, a definition of *Monad* based on (@@) may well be useful for program transformation. However, defining the translation of the comprehension notation, or equivalent expressions using *bind*, in terms of Kleisli composition may be a little more awkward.

3.6 Generic operations on monads

The combination of polymorphism and constructor classes in our system makes it possible to define generic functions that can be used on a wide range of different monads. The Kleisli composition function described in the previous section is a good example of this. Many other examples occur as analogues of standard functions such as the *map* and *foldr* functions used in list processing:

$$\begin{array}{lcl}
mapl & :: & Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow ([a] \rightarrow m\ [b]) \\
mapl\ f\ [\,] & = & [\,[\,]\,] \\
mapl\ f\ (x : xs) & = & [\,y : ys \mid y \leftarrow f\ x,\ ys \leftarrow mapl\ f\ xs\,]
\end{array}$$

$$\begin{array}{lcl}
mfoldr & :: & Monad\ m \Rightarrow (a \rightarrow b \rightarrow m\ b) \rightarrow b \rightarrow [a] \rightarrow m\ b \\
mfoldr\ f\ a\ [\,] & = & result\ a \\
mfoldr\ f\ a\ (x : xs) & = & mfoldr\ f\ a\ xs\ \text{`}bind\text{`}\ (\backslash y \rightarrow f\ x\ y)
\end{array}$$

For example, the expression *mapl f xs* represents a computation whose result is the list obtained by applying *f* to each element of the list *xs*, starting on the left (i.e. moving from the front to the back of the list). Unlike the normal *map* function, the direction is significant because the function *f* may produce a 'side-effect'. The *mapl* function has applications in several different kinds of monads. For example, *mapl* might be used to accumulate statistics (in a state monad) or output messages to indicate progress (in a monad for I/O) while processing a list of elements. Note also that there is an obvious dual of *mapl* in which the elements of the list are processed in the reverse order, obtained by changing the order of the generators in the comprehension:

$$\begin{array}{lcl}
mapr & :: & Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow ([a] \rightarrow m\ [b]) \\
mapr\ f\ [\,] & = & [\,[\,]\,] \\
mapr\ f\ (x : xs) & = & [\,y : ys \mid ys \leftarrow mapr\ f\ xs,\ y \leftarrow fx\,]
\end{array}$$

As another useful example, the comprehension notation can also be used to define a generic version of Haskell's *filter* function that can be used in any monad with a zero:

$$\begin{array}{lcl}
filter & :: & Monad0\ m \Rightarrow (a \rightarrow Bool) \rightarrow m\ a \rightarrow m\ a \\
filter\ p\ xs & = & [\,x \mid x \leftarrow xs,\ p\ x\,]
\end{array}$$

3.7 A family of state monads

We have already described the use of monads to model programs with state using the *State* datatype in Section 3.1. The essential property of any such monad is the ability to update the state, and we might therefore consider a more general class of state monads given by:

$$\begin{array}{l}
\textbf{class}\ Monad\ (m\ s) \Rightarrow StateMonad\ m\ s\ \textbf{where} \\
\quad update\ \ ::\ \ (s \rightarrow s) \rightarrow m\ s\ s
\end{array}$$

An expression of the form *update f* denotes the computation that updates the state using *f* and returns the old state as its result. For example, the *incr* function described above can be defined as *update* $(1+)$ in this more general setting. Operations to set the state to a particular value or to return the current state are easily described in terms of *update*:

```
set      ::  StateMonad m s => s -> m s s
set v    =   update (\s -> v)

get      ::  StateMonad m s => m s s
get      =   update id
```

The *StateMonad* class has two parameters; the first should be a constructor of kind $(* \to * \to *)$, while the second gives the state type (of kind $*$); both are needed to specify the type of *update*. The implementation of *update* for a monad of the form *State s* is straightforward and provides us with our first instance of *StateMonad*:

```
instance StateMonad State s where
  update f  =  ST (\s -> (s,f s))
```

A rather more interesting family of state monads can be described using the following datatype definition:

```
data StateM m s a  =  STM (s -> m (a,s))
```

Note that the first parameter to *StateM* has kind $(* \to *)$, a significant extension from Haskell where all of the arguments to a type constructor must be types. This is another benefit of the kind system.

The functor and monad structure of a *StateM m s* constructor are given by:

```
instance Monad m => Functor (StateM m s) where
  map f (STM xs)  =  STM (\s -> [(f x,s') | ~(x,s') <- xs s])

instance Monad m => Monad (StateM m s) where
  result x        =  STM (\s -> [(x,s)])
  STM xs `bind` f =  STM (\s -> xs s `bind` \(x,s') ->
                                let STM f'  =  f x
                                in  f' s')
```

Note the condition that *m* is an instance of *Monad* in each of these definitions. The definition of *StateM m* as an instance of *StateMonad* is also straightforward:

```
instance StateMonad (StateM m) s where
  update f  =  STM (\s -> [(s,f s)])
```

Some means of starting a computation with a given initial state and returning any results in the parameterizing monad *m* is necessary to make use of *StateM m s* monads. This can be provided by the *runSTM* function:

```
runSTM             ::  Monad m => s -> StateM m s a -> m a
runSTM s (STM f)   =   [x | ~(x,s') <- f s]
```

Closely related to the *startingWith* function introduced in Section 3.1, we will illustrate the use of *runSTM* in Section 3.10.

It is also worth noting how a *StateM m s* monad can 'inherit' properties from the parameterizing monad *m*. For example, if *m* has a *zero* element, then so does *StateM m s*.

```
instance Monad0 m => Monad0 (StateM m s) where
  zero   =   STM (\s -> zero)
```

Support for monads like *StateM m s* seems to be an important step towards solving the problem of constructing monads by combining features from simpler monads, in this case combining the use of state with the features of an arbitrary monad *m*. Of course, we would prefer a more general method for combining arbitrary monads, but it does not seem that this is possible. However, given a large enough collection of examples like *StateM* that can be used to extend the features of an arbitrary monad with those of another specific kind of monad, it should still be possible to construct a wide range of combined monads in a fairly straightforward manner. See King and Wadler (1992), Jones and Duponcheel (1993) and Steele (1994) for related work in this area.

3.8 When computations fail: a class of error monads

Just as we defined a class of state monads by capturing the essential feature of such monads—the ability to update the state—we will introduce a class of monads for describing computations that might fail with a string as an error message or diagnostic:

```
class Monad m => ErrorMonad m where
  fail   ::   String -> m a
```

The *Error* datatype defined below, a variation on the *Maybe* datatype used in previous sections, together with a collection of instance declarations describing its *Functor* and *Monad* structure, provides one simple instance of the *ErrorMonad* class:

```
data Error a   =   Done a  |  Err String

instance Functor Error where
  map f (Done x)   =   Done (f x)
  map f (Err s)    =   Err s

instance Monad Error where
  result            =   Done
  Done x `bind` f   =   f x
  Err msg `bind` f  =   Err msg

instance ErrorMonad Error where
  fail   =   Err
```

In fact, further instances of *ErrorMonad* can be obtained for any monad with a zero by mapping any computation of the form *fail msg* to the *zero* element of the monad. Of course, the error message *msg* is lost in this process. This construction can be described by the following instance declaration:

```
instance Monad0 m => ErrorMonad m where
  fail msg   =  zero
```

This declaration is treated as a default, used in a similar way to the definition of instances of *Functor* from instances of *Monad* in Section 3.2.

Another useful property is that, if a *fail* function has been defined for the monad *m*, then there is a natural way to define a *fail* function for the parameterized state monad *StateM m s* introduced in the previous section:

```
instance ErrorMonad m => ErrorMonad (StateM m s) where
  fail msg   =   STM (\s -> fail msg)
```

For example, a constructor of the form *StateM Error Int* can be used to describe the combination of an integer state with the features of an *ErrorMonad*. This will be particularly useful in Section 3.10.

3.9 Monads and substitution

Up to this point, we have concentrated on the use of monads to describe computations. In fact, monads also have a useful interpretation as a general approach to substitution. This in turn provides another application for constructor classes.

Suppose that a value of type *m v* represents a collection of terms with 'variables' of type *v*. Then a function of type $w \rightarrow m\ v$ can be thought of as a substitution, mapping variables of type *w* to terms over *v*. For example, consider the representation of a simple language of types constructed from type variables and the function space constructor using the datatype:

```
data Type v   =   TVar v | Fun (Type v) (Type v)
```

The corresponding instances of *Functor* and *Monad* are as follows:

```
instance Functor Type where
  map f (TVar v)    =  TVar (f v)
  map f (Fun d r)   =  Fun (map f d) (map f r)

instance Monad Type where
  result v          =  TVar v
  TVar v `bind` f   =  f v
  Fun d r `bind` f  =  Fun (d `bind` f) (r `bind` f)
```

In this setting, the *map* function gives a systematic renaming of the variables in a term (there are no bound variables), while the *result* function corresponds to the null substitution that maps each variable to the term that represents it. If *t* :: *Type v* and *s* is a substitution of type $v \rightarrow$ *Type v*, then *t 'bind' s* gives the result of applying the substitution *s* to the term *t*, replacing each occurrence of a variable *v* in *t* with the corresponding term *s v* in the result. In other words, application of a substitution to a term is captured by the function:

$$\begin{array}{lcl} apply & :: & Monad\ m \Rightarrow (v \to m\ w) \to (m\ v \to m\ w) \\ apply\ s\ t & = & t\ \text{`}bind\text{`}\ s \end{array}$$

Note also that the Kleisli composition (@@) operator described in Section 3.5 gives the standard composition of substitutions. The fact that composition of substitutions is associative with the null substitution as both a left and right identity is a special case of the laws for Kleisli composition. Furthermore, using the same laws, and the definition for *bind* in terms of (@@), it is a simple exercise to show that:

$$apply\ result = id \qquad \text{and} \qquad apply\ (s@@t) = apply\ s\ .\ apply\ t$$

for any s and t of the appropriate types. In particular, for any monad m and any type v, the *apply* function is a monoid homomorphism from $\langle v \to m\ v,\ (@@),\ result\rangle$ to $\langle m\ v \to m\ v,\ (.), id\rangle$.

In most cases, the same type will be used to represent variables in both the domain and range of a substitution. We introduce a type synonym to capture this and to make some type signatures a little easier to read.

$$\textbf{type}\ Subst\ m\ v \quad = \quad v \to m\ v$$

Note that *Subst* has kind $(* \to *) \to * \to *$; we do not restrict the arguments of a type synonym to be constructors of kind $*$. On the other hand, unlike datatype constructors, it is an error to use a type synonym constructor without the correct number of arguments. These issues are discussed in more detail in Section 5.2.

One of the simplest kinds of substitution, which will be denoted by $v \mapsto t$, is a function that maps the variable v to the term t but leaves all other variables fixed:

$$\begin{array}{lcl} (\mapsto) & :: & (Eq\ v, Monad\ m) \Rightarrow v \to m\ v \to Subst\ m\ v \\ (v \mapsto t)\ w & = & \textbf{if}\ v == w\ \textbf{then}\ t\ \textbf{else}\ [w] \end{array}$$

The type signature shown here is the principal type for the $(\mapsto)$ operator, and could also have been inferred automatically by the type system. The class constraints $(Eq\ v, Monad\ m)$ indicate that, while $(\mapsto)$ is defined for an arbitrary monad, it can be used only in cases where the values representing variables can be tested for equality.

The following definition gives an implementation of the standard unification algorithm for values of type *Type v*. This illustrates the use of monads both as a means of describing substitutions and as a model for computations, in this case, in an *ErrorMonad*:

$$\begin{array}{lcl} unify\ (TVar\ v)\ (TVar\ w) & = & \textbf{if}\ v == w\ \textbf{then}\ [result]\ \textbf{else}\ [v \mapsto TVar\ w] \\ unify\ (TVar\ v)\ t & = & varBind\ v\ t \\ unify\ t\ (TVar\ v) & = & varBind\ v\ t \\ unify\ (Fun\ d\ r)\ (Fun\ e\ s) & = & [s_2@@s_1 \mid s_1 \leftarrow unify\ d\ e, \\ & & \qquad\qquad s_2 \leftarrow unify\ (apply\ s_1\ r) \\ & & \qquad\qquad\qquad\qquad (apply\ s_1\ s)] \end{array}$$

The only way that unification can fail is if we try to bind a variable to a term that contains that variable. A test for this condition, often referred to as the *occurs check*, is included in the auxiliary function *varBind*:

```
varBind v t  | occurs v t  =  fail "unification fails"
             | otherwise   =  [v ↦ t]
              where occurs v (TVar w)   =  v == w
                    occurs v (Fun d r)  =  occurs v d || occurs v r
```

The types of *unify* and *varBind* were not included in the definitions above, but can be inferred by the type checker, giving:

```
unify    :: (Eq v, ErrorMonad m) ⇒ Type v → Type v → m (Subst Type v)
varBind  :: (Eq v, ErrorMonad m) ⇒ v → Type v → m (Subst Type v)
```

In both cases, the class constraint *Eq v* reflects the use of the (==) operator to compare values used to represent variables. The second constraint, *ErrorMonad m*, occurs as a result of the use of the *fail* function in *varBind*. As a result of the call to *varBind*, the same constraint is propagated into the type of *unify*.

3.10 A simple application: type inference

To illustrate how some of the classes and functions introduced above might actually be used in practice, this section describes an implementation of Milner's type inference algorithm $\mathscr{W}$. We will not attempt to explain in detail how the algorithm works or to justify its formal properties since these are already well-documented, for example in Milner (1978) and Damas and Milner (1982).

The purpose of the type checker is to determine types for the terms of a simple λ-calculus represented by the data type:

```
data Term  =  Var Name             variables, v
           |  Ap Term Term         applications, M N
           |  Lam Name Term        abstractions, λv.M
```

The names of term variables are represented by values of type *Name*; this will typically be a synonym for *String*. For the types themselves, we use the representation introduced in the previous section, with type variables represented by integer values so that it is easy to generate 'new' type variables as the algorithm proceeds. For example, given the identity function represented by *Lam x* (*Var x*) :: *Term*, we expect the type inference algorithm to produce a result of the form *Fun n n* :: *Type Int* for some (arbitrary) type variable of the form *n* = *TVar m*.

At each stage, the type inference algorithm maintains a collection of assumptions about the types currently assigned to free variables. The simplest way to represent this is using an association list of type *Assoc Name* (*Type Int*) with the following, more general, implementation:

```
data Assoc v t              =  Assoc [(v,t)]

empty                       ::  Assoc v t
empty                       =  Assoc []

extend                      ::  v → t → Assoc v t → Assoc v t
extend v t (Assoc as)       =  Assoc ((v,t) : as)

lookup                      ::  (Eq v, ErrorMonad m) ⇒ v → Assoc v t → m t
lookup v (Assoc as)         =  foldr find (fail "unbound variable") as
  where find (w,t) alt      | w == v     =  [t]
                            | otherwise  =  alt

 instance Functor (Assoc v) where
   map f (Assoc as)    =   Assoc [(a,f b) | (a,b) ← as]
```

As the names suggest, *empty* represents the empty association list, *extend* is used to add a new binding, and *lookup* is used to search for a binding, raising an error if no corresponding value is found. We have also defined an instance of the *Functor* class that allows us to apply a function to each of the values held in the list, without changing the keys.

The type inference algorithm behaves almost like a function taking assumptions a and a term m as inputs, and producing a pair consisting of a substitution s and a type t as its result. The intention here is that t will be the principal type of m under the assumptions obtained by applying s to a. The complete algorithm is given by the following definition, with an equation for each different kind of *Term*:

```
infer a (Var v)     =  lookup v a                           `bind`\t →
                       result (result,t)
infer a (Lam v e)   =  newVar                               `bind`\b →
                       infer (extend v (TVar b) a) e        `bind`\(s,t) →
                       result (s, s b `Fun` t)
infer a (Ap l r)    =  infer a l                            `bind`\(s,lt) →
                       infer (map (apply s) a) r            `bind`\(t,rt) →
                       newVar                               `bind`\b →
                       unify (apply t lt) (rt `Fun` TVar b) `bind`\u →
                       result (u@@t@@s, u b)
```

The reason for writing this algorithm in a monadic style is that it is not quite functional. There are two reasons for this; first, it is necessary to generate 'new' variables during type checking. This is usually dealt with informally in presentations of type inference, but a more concrete approach is necessary for a practical implementation. For the purposes of this algorithm, we use a *StateMonad* with an integer state to represent the next unused type variable. New variables are generated using the function *newVar* = *update* (*1* +).

The second reason for using the monadic style is that the algorithm may fail, either because the term contains a variable not bound in the assumptions a, or because the unification algorithm fails.

Both of these are reflected by the class constraints in the type of *infer*, indicating that an instance of both *StateMonad* and *ErrorMonad* is required to use the type inference algorithm:

$$\begin{array}{lll} \textit{infer} & :: & (\textit{ErrorMonad}\ (m\ \textit{Int}), \textit{StateMonad}\ m\ \textit{Int}) \Rightarrow \\ & & \quad \textit{Assoc Name}\ (\textit{Type Int}) \rightarrow \\ & & \quad\quad \textit{Term} \rightarrow \\ & & \quad\quad\quad m\ \textit{Int}\ (\textit{Subst Type Int}, \textit{Type Int}) \end{array}$$

On the other hand, the constraints do not single out any particular monad satisfying these conditions. This is important because we may want to be able to use *infer* as part of a larger program that requires other features such as I/O. In specific cases, it is often possible for the type system to determine which monad is required as a result of the non-generic monad operators that are involved. The *typecheck* function defined below illustrates this point:

$$\begin{array}{lll} \textit{typecheck} & :: & \textit{Term} \rightarrow \textit{Error String} \\ \textit{typecheck} & = & \textit{map}\ (\textit{show}\ .\ \textit{snd})\ .\ \textit{runSTM}\ \ 0\ .\ \textit{infer empty} \end{array}$$

In this case, given the use of *Error* in the type assigned to *typecheck* and of *runSTM* in the body of the definition, it follows that the *infer* function in this definition will require a monad of the form *StateM Error Int*. To make the output from *typecheck* more readable, we apply the *show* function to each output type which has been defined so that, for example, *Fun* (*TVar 0*) (*TVar 0*) will be displayed as the string `"0 -> 0"`.

Now suppose that we also have a function *parse* :: *String* → *Error Term* that attempts to parse its argument string as a term t, returning *Done* t if successful, or *Err* 'syntax error' if not. It is easy to define such a function using standard techniques such as combinator or monadic parsers. A natural application of Kleisli composition is to combine parsing with typechecking:

$$\begin{array}{lll} \textit{typeOf} & :: & \textit{String} \rightarrow \textit{Error String} \\ \textit{typeOf} & = & \textit{typecheck}\ @@\ \textit{parse} \end{array}$$

The following extracts are taken from a session with Gofer, showing how these definitions work in practice. In the first two examples, we ask the system to calculate types for the identity function $\lambda x.x$ and the S combinator, $\lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$:

```
? typeOf "(\\x.x)"
Done "0 -> 0"
? typeOf "(\\x y z.(x z (y z)))"
Done "(2 -> 4 -> 5) -> (2 -> 4) -> 2 -> 5"
?
```

More interesting, perhaps, are the following examples, illustrating each of the different ways that an error can occur during either parsing or type checking:

```
? typeOf "(x"
Err "syntax error"
? typeOf "x"
Err "unbound variable"
? typeOf "(\\x.(x x))"
Err "unification fails"
?
```

The type checker described here implements a very simple form of the standard type inference algorithm; we have not made any attempt to deal with local definitions introduced by **let** constructs, with instantiation and generalization of polymorphic types or with any form of overloading. However, it should be clear that this implementation can be modified to deal with these extensions. We should also mention that most practical implementations of type inference are based on Milner's algorithm $\mathscr{J}$ since this can be implemented using a single substitution that is only modified during unification. This, too, can be dealt with using the techniques described here, with a state type (*Int*, *Subst Type Int*) that records both the next 'new' variable and the current substitution.

4 A formal treatment of the type system

Up to this point, we have concentrated mostly on the applications of constructor classes. In this section and the next, we turn our attention to providing a more formal description of the underlying type system. In particular, we show how constructor classes can be supported in a language where type inference is used to replace the need for explicit type annotations. This is particularly interesting from a theoretical point of view, since the type system includes both overloading and higher-order polymorphism, for example, allowing universal quantification over constructors of kind $* \rightarrow *$.

For reasons of space, many of the technical details have been omitted from this paper. However, the definitions and overall approach used here are very closely based on our earlier work with *qualified types*, except that we allow predicates over type constructors in addition to predicates over types. This previous work is described in Jones (1992a) and documented more fully in Jones (1992b).

4.1 Constructors, substitutions and predicates

For the formal work with the system of constructor classes it is convenient to annotate each constructor expression with its kind. Thus for each kind κ, we have a collection of constructors C^κ, including *constructor variables* α^κ, given by the grammar:

$$\begin{array}{rcll} C^\kappa & ::= & \chi^\kappa & \textit{constants} \\ & | & \alpha^\kappa & \textit{variables} \\ & | & C^{\kappa' \to \kappa}\, C^{\kappa'} & \textit{applications} \end{array}$$

The apparent mismatch between these explicitly kinded constructors and the implicit kinding used in the preceding sections will be addressed in Section 5. Note that, other than requiring that the function space constructor $\rightarrow$ be included as an element of $C^{*\rightarrow*\rightarrow*}$, we do not make any assumption about the constructor constants χ^{κ} in the grammar above.

A *substitution* is a mapping from variables to constructors. Any such function can be extended in a natural way to give a mapping from constructors to constructors. For the purposes of this work, we will restrict ourselves to the use of *kind-preserving substitutions* that map each variable to a constructor of the same kind. A simple induction shows that each of the collections C^{κ} is closed with respect to such substitutions.

A constructor class represents a set of constructors or, more generally, when the class has multiple parameters, a relation between constructors. The kinds of the elements in the relations for a given class P are specified by a tuple of kinds $(\kappa_1, \ldots, \kappa_n)$ called the *arity* of P. For example, any standard type class has arity $(*)$, the *Functor* and *Monad* classes have arity $(* \rightarrow *)$ and the *StateMonad* class has arity $(* \rightarrow * \rightarrow *, *)$. A *predicate*, or *class constraint*, is an expression of the form $P\ C_1\ \ldots\ C_n$ (where each $C_i \in C^{\kappa_i}$) representing the assertion that the constructors $C_1, \ldots, C_n$ are related by P.

The properties of predicates are captured abstractly by an entailment relation $P \Vdash Q$ between finite sets of predicates. The only restrictions that we make on the $\Vdash$ relation are that it is transitive, closed under substitution and monotonic in the sense that $P \Vdash Q$ whenever $P \supseteq Q$. (In practice, some additional restrictions on the definition of $\Vdash$ are necessary to ensure decidability of type checking. We will return to this point in Section 4.5.) The precise definition of entailment is determined by the class and instance declarations that appear in a given program. The following examples are based on the definitions given in the preceding sections:

$$\begin{array}{lll}
\emptyset & \Vdash & \{\mathit{Functor}\ \mathit{List}, \mathit{Monad}\ \mathit{List}\} \\
\{\mathit{Monad}\ m^{*\rightarrow*}\} & \Vdash & \{\mathit{Functor}\ m^{*\rightarrow*}\} \\
\{\mathit{StateMonad}\ m^{*\rightarrow*}\ s^{*}\} & \Vdash & \{\mathit{Monad}\ (m^{*\rightarrow*}\ s^{*})\}
\end{array}$$

The first of these axioms expresses the fact that *List* is an instance of the *Functor* and *Monad* classes, while the remaining two examples appear as a result of superclass declarations. For example, the second axiom indicates that every instance of the *Monad* class must also be an instance of *Functor*. Unlike some descriptions of type class based type inference, there is no need for any special encoding of superclasses since these already fit naturally into our framework.

4.2 *Types and terms*

Following the definition of types and type schemes in ML, we use a structured language of types, with the principal restriction being the inability to support functions with either polymorphic or overloaded arguments:

$$\begin{array}{lcll} \tau & ::= & C^{*} & \textit{types} \\ \rho & ::= & P \Rightarrow \tau & \textit{qualified types} \\ \sigma & ::= & \forall T.\rho & \textit{type schemes} \end{array}$$

The symbols P and T used here range over finite sequences of predicates and constructor variables, respectively.

It will also be convenient to introduce some abbreviations for qualified types and type schemes. In particular, if $\rho = (P \Rightarrow \tau)$ and $\sigma = \forall T.\rho$, then we write $\pi \Rightarrow \rho$ and $\forall\alpha.\sigma$ as abbreviations for $(\{\pi\} \cup P) \Rightarrow \tau$ and $\forall(\{\alpha\} \cup T).\rho$, respectively.

For program terms, we use the standard Core-ML language based on simple untyped λ-calculus with the addition of a *let* construct to enable the definition and use of polymorphic/overloaded terms:

$$E ::= x \mid EF \mid \lambda x.E \mid \mathbf{let}\ x = E\ \mathbf{in}\ F.$$

The symbol x ranges over a given set of *(term) variables*.

4.3 Typing rules

A *type assignment* is a (finite) set of pairs of the form $x : \sigma$ in which no term variable x appears more than once. If A is a type assignment, then we write $dom\ A = \{x \mid (x : \sigma) \in A\}$, and if x is a term variable with $x \notin dom\ A$, then we write $A, x : \sigma$ as an abbreviation for the type assignment $A \cup \{x : \sigma\}$. The type assignment obtained from A by removing any typing statement for the variable x is denoted A_x.

A *typing* is an expression of the form $P \mid A \vdash E : \sigma$ representing the assertion that a term E has type σ when the predicates in P are satisfied and the types of free variables in E are as specified in the type assignment A. The set of all derivable typings is defined by the rules in Fig. 1. Note the use of the symbols τ, ρ and σ to restrict the application of certain rules to specific sets of type expressions.

Most of these are similar to the usual rules for Core-ML; only the rules $(\Rightarrow I)$ and $(\Rightarrow E)$ for dealing with qualified types and the $(\forall I)$ rule for polymorphic generalization involve the predicate set. An expression of the form $CV(X)$ denotes the set of constructor variables appearing free in X. For example, in rule $(\forall I)$, the condition $\alpha^{\kappa} \notin CV(A) \cup CV(P)$ is needed to ensure that we do not universally quantify over a variable that is constrained either by A or P.

4.4 Type inference

The rules in Fig. 1 are useful as a means of explaining and understanding the type system, but they are not suitable as a basis for a type inference algorithm: There are many different ways in which the rules might be applied to find the type of a given term and it is not always clear which (if any) will give the best result. An alternative set of rules that avoids these problems is presented in Section 4.4.2 following a preliminary description of constructor unification.

$$(var)\quad \frac{(x:\sigma)\in A}{P\,|\,A\vdash x:\sigma} \qquad\qquad (let)\quad \frac{\begin{array}{c}P\,|\,A\vdash E:\sigma\\ Q\,|\,A_x,x:\sigma\vdash F:\tau\end{array}}{P\cup Q\,|\,A\vdash(\mathbf{let}\ x=E\ \mathbf{in}\ F):\tau}$$

$$(\rightarrow E)\quad \frac{\begin{array}{c}P\,|\,A\vdash E:\tau'\rightarrow\tau\\ P\,|\,A\vdash F:\tau'\end{array}}{P\,|\,A\vdash EF:\tau} \qquad\qquad (\rightarrow I)\quad \frac{P\,|\,A_x,x:\tau'\vdash E:\tau}{P\,|\,A\vdash\lambda x.E:\tau'\rightarrow\tau}$$

$$(\Rightarrow E)\quad \frac{\begin{array}{c}P\,|\,A\vdash E:\pi\Rightarrow\rho\\ P\Vdash\{\pi\}\end{array}}{P\,|\,A\vdash E:\rho} \qquad\qquad (\Rightarrow I)\quad \frac{P\cup\{\pi\}\,|\,A\vdash E:\rho}{P\,|\,A\vdash E:\pi\Rightarrow\rho}$$

$$(\forall E)\quad \frac{\begin{array}{c}P\,|\,A\vdash E:\forall\alpha^\kappa.\sigma\\ C\in C^\kappa\end{array}}{P\,|\,A\vdash E:[C/\alpha^\kappa]\sigma} \qquad\qquad (\forall I)\quad \frac{\begin{array}{c}P\,|\,A\vdash E:\sigma\\ \alpha^\kappa\notin CV(A)\cup CV(P)\end{array}}{P\,|\,A\vdash E:\forall\alpha^\kappa.\sigma}$$

Fig. 1. ML-like typing rules for constructor classes.

4.4.1 Unification of constructor expressions

Unification algorithms are often required in type inference algorithms to ensure that the argument type of a function coincides with the type of the argument that it is applied to. In the context of this paper, we need to deal with unification of constructors which is a little more tricky than unification of simple types since we need to keep track of the kinds of the constructors involved. Nevertheless, the following presentation follows the standard approach, as introduced by Robinson (1965), extended to deal with the use of kind-preserving substitutions.

We begin with two fairly standard definitions. A kind-preserving substitution S is a *unifier* of two constructors $C, C' \in C^\kappa$ if $SC = SC'$. A kind-preserving substitution U is called a *most general unifier* for the constructors $C, C' \in C^\kappa$ if:

- U is a unifier for C and C', and
- every unifier S of C and C' can be written in the form RU for some kind-preserving substitution R.

Writing $C \overset{U}{\sim}_\kappa C'$ for the assertion that U is a unifier of the constructors $C, C' \in C^\kappa$, the rules in Fig. 2 define a *unification algorithm* that can be used to calculate a unifier for a given pair of constructors. It is straightforward to verify that any substitution U obtained using these rules is indeed a unifier for the corresponding pair of constructors.

Notice that there are two distinct ways in which the unification algorithm may fail; first in the rules (*bindVar*) and (*bindVar'*) where unification would result in an infinite constructor (i.e. producing a regular tree). Second, the unification of a constructor of the form CC' with another constructor of the form DD' is possible only if C and D can be unified which in turn requires that these two constructors have the same kind, which must be of the form $\kappa' \rightarrow \kappa$. This is a consequence of the fact that there are no non-trivial equivalences between constructor expressions. This

(*idVar*)	$\alpha \overset{id}{\sim}_{\kappa} \alpha$	
(*idConst*)	$\chi \overset{id}{\sim}_{\kappa} \chi$	
(*bindVar*)	$\alpha \overset{[C/\alpha]}{\sim}_{\kappa} C$	$\alpha \notin CV(C)$
(*bindVar'*)	$C \overset{[C/\alpha]}{\sim}_{\kappa} \alpha$	$\alpha \notin CV(C)$
(*apply*)	$\dfrac{C \overset{S}{\sim}_{\kappa' \to \kappa} D \quad SC' \overset{S'}{\sim}_{\kappa'} SD'}{CC' \overset{S'S}{\sim}_{\kappa} DD'}$	

Fig. 2. Kind-preserving unification.

property would be lost if we had included abstractions over constructor variables in the language of constructors, requiring the use of higher-order unification and ultimately leading to undecidability in the type system.

With these observations, we can use standard techniques to prove:

Theorem 1
If there is a unifier for two given constructors $C, C' \in C^{\kappa}$, then $C \overset{U}{\sim}_{\kappa} C'$ using the rules in Fig. 2 for some U and this substitution is a most general unifier for C and C'. Conversely, if no unifier exists, then the unification algorithm fails.

The rules in Fig. 2 require that the two constructors involved at each stage in the unification must have the same kind. In practice, however, it is only necessary to check that the constructor C has the same kind as the variable α in the rules (*bindVar*) and (*bindVar'*), ensuring that the substitution $[C/\alpha]$ in these rules is kind-preserving.

The process of finding the kind of the constructor C to compare with the kind of the variable α to which it will be bound can be implemented reasonably efficiently. Suppose that $C = H\ C_1 \ldots C_n$ where H is either a variable or a constant. Since C is a well-formed constructor, H must have kind of the form $\kappa_1 \to \ldots \to \kappa_n \to \kappa$ where κ_i is the kind of the corresponding constructor C_i. It follows that C has kind κ. Thus the only information needed to find the kind of C is the kind of the head H and the number of arguments n that it is applied to. There is no need for any more sophisticated form of kind inference in this situation.

The need to check the kinds of constructors in this way certainly increases the cost of unification. On the other hand, we would expect that, in many cases, this would be significantly less than that of the occurs check—$\alpha \notin CV(C)$—which will typically involve a complete traversal of C.

4.4.2 *A type inference algorithm*

The rules in Fig. 3 provide an alternative to those of Fig. 1 with a single rule for each syntactic construct in the language of terms. Each judgment is an expression of the form $P \mid TA \overset{W}{\vdash} E : \tau$ where P is a set of predicates, T is a (kind-preserving)

$$(var)^{\mathrm{W}} \quad \frac{(x : \forall \alpha_i^{\kappa_i}.P \Rightarrow \tau) \in A \qquad \beta_i^{\kappa_i} \text{ new}}{[\beta_i^{\kappa_i}/\alpha_i^{\kappa_i}]P \mid A \overset{\mathrm{W}}{\vdash} x : [\beta_i^{\kappa_i}/\alpha_i^{\kappa_i}]\tau}$$

$$(\to E)^{\mathrm{W}} \quad \frac{P \mid TA \overset{\mathrm{W}}{\vdash} E : \tau \qquad Q \mid T'TA \overset{\mathrm{W}}{\vdash} F : \tau' \qquad \alpha^* \text{ new} \qquad T'\tau \overset{U}{\sim}_* \tau' \to \alpha^*}{U(T'P \cup Q) \mid UT'TA \overset{\mathrm{W}}{\vdash} EF : U\alpha^*}$$

$$(\to I)^{\mathrm{W}} \quad \frac{P \mid T(A_x, x : \alpha^*) \overset{\mathrm{W}}{\vdash} E : \tau \qquad \alpha^* \text{ new}}{P \mid TA \overset{\mathrm{W}}{\vdash} \lambda x.E : T\alpha^* \to \tau}$$

$$(let)^{\mathrm{W}} \quad \frac{P \mid TA \overset{\mathrm{W}}{\vdash} E : \tau \qquad P' \mid T'(TA_x, x : Gen(TA, P \Rightarrow \tau)) \overset{\mathrm{W}}{\vdash} F : \tau'}{P' \mid T'TA \overset{\mathrm{W}}{\vdash} (\textbf{let } x = E \textbf{ in } F) : \tau'}$$

Fig. 3. Type inference algorithm W.

substitution, A is a type assignment, E is a term and τ is an element of C^*. The *Gen* function used in the rule $(let)^{\mathrm{W}}$ is defined by $Gen(A, \rho) = \forall(CV(\rho) \setminus CV(A)).\rho$ and is used to calculate the generalization of a qualified type ρ with respect to the assumptions A.

These rules can be interpreted as an algorithm for calculating typings. Starting with a term E and an assignment A, we can use the structure of E to guide the calculation of values for P, T and τ such that $P \mid TA \overset{\mathrm{W}}{\vdash} E : \tau$. The only way that this process can ever fail is if E contains a free variable that is not mentioned in A or if one of the unifications fails.

The type inference algorithm has several important properties. First of all, the typing that it calculates is valid with respect to the original set of typing rules:

Theorem 2
If $P \mid TA \overset{\mathrm{W}}{\vdash} E : \tau$, then $P \mid TA \vdash E : \tau$.

Given a term E and an assignment A it is particularly useful to find a concise characterization of the set of pairs $(P \mid \sigma)$ such that $P \mid A \vdash E : \sigma$. This information might, for example, be used to determine if E is well-typed (i.e. if the set of pairs is non-empty) or to validate a programmer supplied type signature for E.

Following the approach of Damas and Milner (1982), this can be achieved by defining an ordering on the set of all pairs $(P \mid \sigma)$ to describe when one pair is more general than another. We can then show that the set of all $(P \mid \sigma)$ for which a given term is well-typed is equal to the set of all pairs that are less than a *principal type*, calculated using the type inference algorithm. This allows us to establish the following theorem:

Theorem 3
Let E be a term and A an arbitrary type assignment. The following conditions are equivalent:

- E is well-typed under A.
- $P \mid TA \overset{\mathrm{W}}{\vdash} E : \tau$ for some P and τ and there is a substitution R such that $RTA = A$.
- E has a principal type under A.

The definition of the ordering between pairs $(P \mid \sigma)$ and the full proof of the above theorem are essentially the same as used in Jones (1992b), to which we refer the reader for further details.

4.5 Coherence and decidability of type checking

Up to this point, we have not made any attempt to discuss how programs in the current system of constructor classes might be implemented. One fairly general approach is to find a *translation* for each source term in which overloaded functions have additional *evidence* parameters that make the use of overloading explicit. Different forms of evidence values can be used. For example, in theoretical work, it might be sensible to use predicates themselves as evidence, while practical work might benefit from a more concrete approach, such as the *dictionary* passing scheme proposed by Wadler and Blott (1989).

To justify this approach, it is important to show that any two potentially distinct translations of a given term are semantically equivalent. Following Breazu-Tannen *et al.* (1989), we refer to this as a *coherence* property of the type system. As we hinted in Section 2.1, the same problem occurs in Haskell unless we restrict our attention to terms with unambiguous type schemes. Fortunately, the same solution works for the system described in this paper; we can show that, if the principal type scheme $\forall \alpha_i . P \Rightarrow \rho$ of a given term satisfies $\{\alpha_i\} \cap CV(P) \subseteq CV(\rho)$, then all translations of that term are equivalent.

This restriction also simplifies the conditions needed to ensure decidability of type checking. In particular, we can show that type checking for terms with unambiguous principal types is decidable if and only if, for each choice of P and Q, the task of determining whether $P \Vdash Q$ is decidable. Simple syntactic conditions on the form of instance declarations can be used to guarantee this property. The same approach is used in the definition of Haskell (Hudak *et al.*, 1992).

Once again, we refer the reader to Jones (1992b) for further details and background on the issues raised in this section.

5 Kind inference

The biggest difference between the formal type system described in Section 4 and its 'user interface' described in the opening sections of the paper is the need to annotate constructor variables with their kinds. As we have already indicated, we regard the fact that the programmer does not supply kind information explicitly as a significant advantage of the system. At the same time, this also means an implementation of this systems needs to be able to determine:

1. The kind of each user-defined constructor χ.
2. The arity of each constructor class P.
3. The kind of each universally quantified variable in a type scheme. This is necessary so that we can generate new variables of the appropriate kind when a type scheme is instantiated using $(var)^{W}$.

Fortunately, given the simple structure of the languages of constructors and kinds, it is relatively straightforward to calculate suitable values in each of these cases using a process of *kind inference*. Treating the set of constructors as a system of combinators, we can use standard techniques—analogous to type inference—to discover constraints on the kinds of each object appearing in a given (unannotated) constructor expression and solve these constraints to obtain the required kinds.

Item (1) will be dealt with more fully in Sections 5.1 and 5.2. To illustrate the basic ideas for an example involving items (2) and (3), recall the definition of the constructor class *Functor* from Section 2.2:

$$\begin{array}{l} \textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \\ \quad \mathit{map}\ ::\ (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b) \end{array}$$

Using the fact that $(\rightarrow) :: * \rightarrow * \rightarrow *$, and that both a and b are used as arguments to $\rightarrow$ in the expression $(a \rightarrow b)$, it follows that both of these variables must have kind $*$. By a similar argument, $f\ a$ also has kind $*$ and hence f must have kind $* \rightarrow *$ as expected. Thus *map* has type:

$$\forall f^{*\rightarrow *}.\forall a^{*}.\forall b^{*}.\mathit{Functor}\ f \Rightarrow (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)$$

and the *Functor* class has arity $(* \rightarrow *)$.

5.1 Datatype definitions

Many programs include definitions of new datatypes and it is important to determine suitable kinds for the corresponding type constructors. The general form of a datatype declaration in Haskell is:

$$\textbf{data}\ \chi\ a_1\ \ldots\ a_m \quad = \quad \mathit{constr}_1 \ \mid\ \ldots\ \mid\ \mathit{constr}_n$$

This introduces a new constructor χ that expects m arguments, represented by the (distinct) variables $a_1, \ldots, a_m$. Each *constr* on the right-hand side is an expression of the form $F\ \tau_1\ \ldots\ \tau_n$ that allows the symbol F to be used as a function of type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \chi\ a_1\ \ldots\ a_m$ to construct values of the new type.

In our more general setting, we can treat χ as a constructor of kind:

$$\kappa_1 \rightarrow \ldots \rightarrow \kappa_m \rightarrow *$$

where $\kappa_1,\ldots,\kappa_m$ are kinds for $a_1, \ldots, a_m$, respectively, calculated by the kind inference process.

We have already seen several examples above for which the kind of the type constructor may be determined by this process. In some cases, the definition of a type constructor does not uniquely determine its kind; just as some λ-terms can be assigned polymorphic types, some type constructors can be treated as having polymorphic kinds. For example, given the definition:

$$\textbf{data}\ \mathit{Fork}\ a \quad = \quad \mathit{Prong}\ \mid\ \mathit{Split}\ (\mathit{Fork}\ a)\ (\mathit{Fork}\ a)$$

we can infer that *Fork* has kind $\kappa \rightarrow *$ for any kind κ. In the current implementation, we avoid the need to deal with polymorphic kinds by replacing any unknown part of an inferred kind with $*$. Hence the *Fork* constructor will actually be treated as having kind $* \rightarrow *$. This is consistent with the interpretation of datatype definitions in Haskell where all variables are expected to have kind $*$.

5.2 Synonym definitions

In addition to defining new datatypes, it is common for a program to introduce new names for existing types using a type synonym declarations of the form:

$$\textbf{type}\ \chi\ a_1\ \ldots\ a_m \quad = \quad rhs$$

The intention here is that any type expression of the form $\chi\ C_1\ \ldots\ C_m$ abbreviates the type obtained from *rhs* by replacing each occurrence of a variable a_i with the corresponding constructor C_i. We do not allow a type synonym constructor χ to be used without the full number of arguments. This ensures that we do not invalidate the conditions needed to establish the coherence property described in Section 4.5. In addition, following the definition of Haskell, it is not permitted to define mutually recursive type synonyms without an intervening datatype definition. These conditions guarantee that it is always possible to expand any given type expression to eliminate all type synonyms. However, for practical purposes, it is sensible to calculate a kind for each synonym constructor χ and hence avoid the need to expand synonyms during kind inference. For example, given the definitions:

$$\begin{array}{lcl} \textbf{type}\ Church\ a & = & (a \rightarrow a) \rightarrow (a \rightarrow a) \\ \textbf{type}\ Subst\ t\ v & = & v \rightarrow t\ v \end{array}$$

we find that, for the purposes of kind inference:

$$\begin{array}{lcl} Church & :: & * \rightarrow * \\ Subst & :: & (* \rightarrow *) \rightarrow * \rightarrow *. \end{array}$$

6 Conclusions and future work

Based on the examples in this paper and our initial experience with the prototype implementation, we believe that there are many useful applications for a system of constructor classes. Of course, since the use of constructor classes is likely to lead to a greater use of overloading in typical programs, further work to investigate new techniques for the efficient implementation of type and constructor class overloading would be particularly useful.

The use of constructor classes is compatible with the current treatment of type classes in Haskell, but some further extensions to the language may also be helpful. For example, the ability to generate instances of the *Functor* class automatically would certainly be useful, extending the 'derived instance' mechanism in Haskell (Hudak *et al.*, 1992, Appendix E). Some small extensions are necessary to support separate compilation. In particular, if a module exports a datatype constructor χ without any value constructor functions, then it will not be possible to infer the kind of χ. The obvious solution is to include the kind of χ explicitly as part of the module interface specification.

There are two noticeable restrictions in the current system. First, while the decision to exclude any form of abstraction from the language of constructors is essential to ensure the tractability of the whole system, it also places some limitations on the programmer. For example, having defined:

$$\textbf{data}\ State\ s\ a \quad = \quad ST\ (s \rightarrow (a,s))$$

we were able to define *State* as a functor in its second argument. This would not have been possible if the two parameters s and a on the left-hand side been written in the reverse order. Of course, this problem can always be avoided by defining a new data type. Indeed, this was precisely the motivation for originally introducing the *State* data type, since the constructor expression $s \rightarrow (a,s)$ is not in a suitable form. Nevertheless, it would certainly be useful to find a less cumbersome technique.

The second problem can be illustrated by considering the task of building a class of set constructors. One possible class declaration for this might be:

```
class Functor set => Set set where
  emptySet         :: set a
  singleton        :: a -> set a
  union, intersect :: set a -> set a -> set a
  member           :: a -> set a -> Bool
  setToList        :: set a -> List a
```

Unfortunately, this declaration does not reflect the fact that, for some concrete implementations of sets, additional constraints must be placed on the type of values that can be held in a set. For example, if sl :: $ListSet\ a$ is a set, represented by a list of values, then the *member* function can be used to test for membership in sl only if a is an instance of *Eq*. On the other hand, if st :: $TreeSet\ a$ is a set, implemented using a binary search tree, then a stronger condition, requiring a to be an instance of the class *Ord* so that its values can be ordered, is necessary. Since these constraints are properties of the individual constructors rather than the class, it is appropriate that they should be reflected as part of the datatype definitions:

```
data Eq a => ListSet a   =  ...
data Ord a => TreeSet a  =  ...
```

Our intention here is to view *Eq* and *Ord* as new kinds, each a subkind of $*$, and to treat *ListSet* and *TreeSet* as having kinds $Eq \rightarrow *$ and $Ord \rightarrow *$, respectively (in fact, this syntax is already supported in Haskell, but with a somewhat different interpretation). Extending these ideas to the type system described in this paper would require some form of subkinding. We hope that experience with the current implementation will allow us to decide whether examples like the *Set* class above occur sufficiently often in practice to warrant the additional complexity.

The system of parametric type classes proposed in Chen *et al.* (1992) also addresses some of the problems that we have described in this paper. The most important feature of parametric classes is their ability to express particular forms of dependency between the parameters in a multi-parameter class. This might, for example, provide us with an alternative solution to the *Set* class problem described above, using instance declarations of the form:

instance *Eq a* ⇒ *ListSet a* :: *Set a* **where** ...

Nevertheless, we believe that the ability to express dependencies between constructors is largely orthogonal to our use of higher-order constructor variables. It seems reasonable to expect that a system of parametric classes could be extended along the lines described in this paper to obtain a system of parametric constructor classes that supports the features of both systems.

Acknowledgments

This work was supported in part by a grant from ARPA, contract number N00014-91-J-4043. My thanks also to Phil Wadler and to the three anonymous referees for their help in improving the presentation of this paper.

References

Barendregt, H. (1991) Introduction to generalized type systems. *Journal of Functional Programming*, **1**(2).

Breazu-Tannen, V., Coquand, T., Gunter, C. A. and Scedrov, A. (1989) Inheritance and coercion. *IEEE Symposium on Logic in Computer Science.*

Chen, K., Hudak, P. and Odersky, M. (1992) Parametric type classes (Extended abstract). *ACM Conf. LISP and Functional Programming*, San Francisco, CA, June.

Damas, L. and Milner, R. (1982) Principal type schemes for functional programs. *8th Ann. ACM Symposium on Principles of Programming languages.*

Hudak, P. and Fasel, J. (1992) A gentle introduction to Haskell. *ACM SIGPLAN Notices*, **27**(5).

Hudak, P., Peyton Jones, S. L. and Wadler, P. (eds.) (1992) Report on the programming language Haskell, version 1.2. *ACM SIGPLAN Notices*, **27**(5).

Jones, M. P. (1991) Introduction to Gofer—included as part of the Gofer distribution, available by anonymous ftp from `nebula.cs.yale.edu` in the directory `pub/haskell/gofer`.

Jones, M. P. (1992a) A theory of qualified types. *European symposium on programming. Lecture Notes in Computer Science 582.* Springer-Verlag.

Jones, M. P. (1992b) Qualified types: Theory and Practice. DPhil thesis, Oxford University Computing Laboratory.

Jones, M. P. (1992c) Programming with constructor classes (preliminary summary). *Proc. Fifth Annual Glasgow Workshop on Functional Programming*, Ayr, Scotland. Springer-Verlag.

Jones, M. P. and Duponcheel, L. (1993) Composing Monads. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-1004.

King, D. J. and Wadler, P. (1992) Combining monads. *Proc. Fifth Annual Glasgow Workshop on Functional Programming*, Ayr, Scotland. Springer-Verlag.

MacLane, S. (1971) *Categories for the working mathematician. Graduate Texts in Mathematics* 5. Springer-Verlag.

MacQueen, D. B. (1984) Modules for Standard ML. *Conf. Record 1984 ACM Symposium on Lisp and Functional Programming.*

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

Milner, R., Tofte, M. and Harper, R. (1990) *The definition of Standard ML.* MIT Press.

Moggi, E. (1989) Computational lambda-calculus and monads. *IEEE Symposium on Logic in Computer Science*, Asilomar, CA.

Robinson, J. A. (1965) A machine-oriented logic based on the resolution principle. *Journal ACM*, **12**.

Spivey, M. (1990) A functional theory of exceptions. *Science of Computer Programming*, **14**(1).

Steele, G. L., Jr. (1994) Building interpreters by composing monads. *21st Ann. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon.

Wadler, P. and Blott, S. (1989) How to make ad-hoc polymorphism less ad-hoc. *16th Ann. ACM Symposium on Principles of Programming Languages*, Austin, TX.

Wadler, P. (1990) Comprehending Monads. *ACM Conf. LISP and Functional Programming*, Nice, France.

Wadler, P. (1992) The essence of functional programming. *19th Ann. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Santa Fe, NM.