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

Mark P. Jones
Yale University, Department of Computer Science,
P.O. Box 2158 Yale Station, New Haven, CT 06520-2158.
jones-mark@cs.yale.edu

Abstract

This paper describes a flexible type system which 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 wide range of examples which 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 many of the attractive features that have made the use of Hindley/Milner type systems so popular. In particular, there is an effective algorithm which 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.

1 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 [6] 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:

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

Each of these functions has a similar type to that of the original *map* and also satisfies the functor laws given above. 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 *mapOpt*.

Unfortunately, in a language using standard Hindley/Milner type inference, there is no way 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.

1.1 An attempt to define *map* using type classes

The ability to use a single function symbol with an interpretation that depends on 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* [13]. 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* which includes pre-

cisely those types whose elements can be compared using the (==) function. A simple definition might be:

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

The equality operator can then be treated as having any of the types in the set $\{\, a \to a \to Bool \mid a \in Eq \,\}$. 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:

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

can be used to test for membership in a list of integers, characters, lists, pairs, etc. See [5, 13] 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 $S = \{\, m(t) \mid t \in Map \,\}$ includes (at least) the types:

$$\begin{array}{l}(a \to b) \to ([a] \to [b])\\ (a \to b) \to (Tree\ a \to Tree\ b)\\ (a \to b) \to (Opt\ a \to Opt\ b)\end{array}$$

(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:

$$\begin{array}{l}\textbf{class } Map\ t \textbf{ where}\\ \quad map \ :: \ t\\ \textbf{instance } Map\ ((a \to b) \to ([a] \to [b]))\ \textbf{where}\\ \quad \ldots\\ \textbf{instance } Map\ ((a \to b) \to (Tree\ a \to Tree\ b))\ \textbf{where}\\ \quad \ldots\\ \textbf{instance } Map\ ((a \to b) \to (Opt\ a \to Opt\ b))\ \textbf{where}\\ \quad \ldots\end{array}$$

This syntax is not permitted in the current syntax of 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 j . map i* would be

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

where a and b are the types of i and j respectively. This is complicated and does not enforce the condition that i and j 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). Under these conditions, we cannot guarantee a well-defined semantics for this expression (see [8], for example). Other attempts to define the *map* function, for example using multiple parameter type classes, have also failed for essentially the same reasons.

1.2 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:

$$(a \to b) \to (f\ a \to f\ b).$$

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 *Opt* 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:

$$\begin{array}{l}\textbf{class } Functor\ f \textbf{ where}\\ \quad map \ :: \ (a \to b) \to (f\ a \to f\ b)\\ \textbf{instance } Functor\ List \textbf{ where}\\ \quad map\ f\ [\,] \ = \ [\,]\\ \quad map\ f\ (x : xs) \ = \ f\ x : map\ f\ xs\\ \textbf{instance } Functor\ Tree \textbf{ where}\\ \quad map\ f\ (Leaf\ x) \ = \ Leaf\ (f\ x)\\ \quad map\ f\ (l :\hat{}: r) \ = \ map\ f\ l :\hat{}: map\ f\ r\\ \textbf{instance } Functor\ Opt \textbf{ where}\\ \quad map\ f\ (Just\ x) \ = \ Just\ (f\ x)\\ \quad map\ f\ Nothing \ = \ Nothing\end{array}$$

Functor is our first example of a *constructor class*. The following extract (taken from a session with the Gofer system 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* function more precisely, we avoid the ambiguity problems mentioned above. For example, the principal type of *map j . map i* is simply $Functor\ f \Rightarrow f\ a \to f\ c$ provided that i has type $(a \to b)$ and that j has type $(b \to c)$.

1.3 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 $(a \to b) \to (Int\ a \to Int\ b)$ 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.

Our approach to this problem is to formalize the notion of *kind* writing $*$ for the kind of all types and $\kappa_1 \to \kappa_2$ for the kind of a constructor which 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 [1]. Instead of type expressions, we use a language of constructors given by:

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

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

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

The kinds of constructor applications are described by the rule:

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

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 only valid if f is a constructor expression of kind $* \rightarrow *$.

The language of constructors is essentially a system of combinators without any reduction rules. As such, standard techniques can be used to infer the kinds of constructor variables, constructor constants introduced by new datatype definitions and the kind of the elements held in any particular constructor 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 4.

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, Hudak and Odersky [3] which addresses similar issues to 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 $*$.

2 Monads as an application of constructor classes

Motivated by the work of Moggi [10] and Spivey [12], Wadler [14, 15] has 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, Wadler introduced the idea that monads could be used as a practical method for modeling so-called 'impure' features in a purely functional programming language. The examples in this section illustrate that the use of constructor classes can be particularly convenient for programming in this style.

2.1 A framework for programming with monads

The basic motivation for the use of monads is the need to distinguish between computations and the values that they produce. If m is a monad then an object of type $(m\ a)$ represents a computation which is expected to produce a value of type a. These types reflect the fact that the use of particular programming language features in a given calculation is a property of the computation itself and not of the result that it produces.

Taking the approach outlined in [15] we introduce a constructor class of monads using the definition:

$$\begin{array}{l} \textbf{class}\ Functor\ m \Rightarrow Monad\ m\ \textbf{where} \\ \quad \begin{array}{lll} result & :: & a \rightarrow m\ a \\ bind & :: & m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\ join & :: & m\ (m\ a) \rightarrow m\ a \end{array} \\ \\ \quad \begin{array}{lcl} x\ `bind`\ f & = & join\ (map\ f\ x) \\ join\ x & = & x\ `bind`\ id \end{array} \end{array}$$

The expression $Functor\ m \Rightarrow Monad\ m$ 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. The use of a hierarchy of classes enables us to capture the fact that not every instance of $Functor$ can be treated as an instance of $Monad$ in a natural way.

By including default definitions for $bind$ and $join$ we only need to give a definition for one of these (in addition to a definition for $result$) to completely define an instance of $Monad$. This is often quite convenient. On the other hand, it would be an error to omit definitions for both operators since the default definitions are clearly circular. We should also mention that the member functions in an instance of $Monad$ are expected to satisfy a number of laws which are not reflected in the class definition above. See [15] for further details.

The following declaration defines the standard monad structure for the list constructor $List$ which can be used to describe computations producing multiple results, corresponding to a simple form of non-determinism:

$$\begin{array}{l} \textbf{instance}\ Monad\ List\ \textbf{where} \\ \quad \begin{array}{lcl} result\ x & = & [x] \\ join & = & foldr\ (+\!\!+)\ [\,] \end{array} \end{array}$$

Another interesting use 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. In order to get this into the appropriate form for the system of constructor classes described in this paper, we introduce a new datatype:

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

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

$$\begin{array}{l} \textbf{instance}\ Functor\ (State\ s)\ \textbf{where} \\ \quad map\ f\ (ST\ st) \quad = \quad ST\ (\backslash s \rightarrow \textbf{let} \quad (x, s') \quad = \quad st\ s \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in} \quad (f\ x, s')) \\ \textbf{instance}\ Monad\ (State\ s)\ \textbf{where} \\ \quad \begin{array}{lcl} result\ x & = & ST\ (\backslash s \rightarrow (x, s)) \\ ST\ m\ `bind`\ f & = & ST\ (\backslash s \rightarrow \textbf{let}\ \ (x, s') = m\ s \\ & & \qquad\qquad\quad\ \ ST\ f' = f\ x \\ & & \qquad\quad \textbf{in}\ \ f'\ s') \end{array} \end{array}$$

Notice that the $State$ constructor has kind $* \rightarrow * \rightarrow *$ and that the declarations above define $State\ s$ as a monad and

functor for any state type s (and hence *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*, *bind* and *join* 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 labeling 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 (represented by 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.

An alternative definition, using a state monad and following the layout suggested in [15], can be written as follows:

```
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')
```

While this program is perhaps 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* (or indeed, the use of *incr* on its own) 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*, *result* and *join* as in [15] or to rely on the use of explicit type declarations.

2.2 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 standard syntax for set comprehensions in mathematics. In [14], Wadler made the observation that 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 but 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 the previous section, we have implemented a more flexible form of monad comprehension which relies on overloading rather than superscripts. At the time of writing, this is the only concrete implementation of monad comprehensions known to us.

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 the previous section:

```
[ e ]                      =  result e
[ e | p ← exp, gs ]        =  exp `bind` \ p → [ e | gs ]
```

In this notation, the *label* function from the previous section can be rewritten as:

```
label          ::  Tree a → Tree (a, Int)
label tree     =   lab tree `startingWith` 0
 where
   lab (Leaf n)   =  [ Leaf (n, c) | c ← incr ]
   lab (l :^: r)  =  [ l' :^: r' | l' ← lab l, r' ← lab r ]
```

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.

2.3 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*:

```
class Monad m ⇒ Monad0 m where
  zero  ::  m a
```

For example, the *List* monad has the empty list as a *zero* element:

```
instance Monad0 List where
  zero  =  []
```

Note that not there are also some monads which do not have a zero element and hence cannot be defined as instances of *Monad0*. The *State s* monads described in Section 2 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 | guard, gs ]  =  if guard then [ e | gs ] else zero
```

Notice that, as far as the type system is concerned, the use of *zero* in the translation of a comprehension such as $[\,e \mid x \leftarrow xs,\ b\,]$ 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:

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

This corresponds directly to the semantics of list comprehensions in the current version of Haskell. The only time when there is any difference between this and the translation in Section 2.2 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 $\tilde{}p \leftarrow exp$. (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 [9] is to use a slightly different syntax for monad comprehensions so that there is no clash with the semantics of Haskell list comprehensions.

2.4 Generic operations on monads

The combination of polymorphism and constructor classes in our system makes it possible to define generic functions which can be used on a wide range of different monads. A simple example of this is the *Kleisli composition* for an arbitrary monad, similar to the usual composition of functions except that it also takes care of 'side effects'. The general definition is as follows:

$$\begin{array}{l} (@@) \ :: \ Monad\ m \Rightarrow (b \to m\ c) \to (a \to m\ b) \to (a \to m\ c) \\ f\ @@\ g\ =\ join\ .\ map\ f\ .\ g \end{array}$$

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:

$$\begin{array}{lcll} (f@@g)\ a & = & ST\ (\backslash s_0 \to \textbf{let} & ST\ g' = g\ a \\ & & & (b, s_1) = g'\ s_0 \\ & & & ST\ f' = f\ b \\ & & & (c, s_2) = f'\ s_1 \\ & & \textbf{in} & (c, s_2)) \end{array}$$

The biggest advantage of the generic 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.

Generic operations can also be defined using the comprehension notation:

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

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 kinds of monad with obvious examples involving state and output.

The comprehension notation can also be used to define a generalization of Haskell's *filter* function which works in an arbitrary monad with a zero:

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

There are many other general purpose functions that can be defined in the current framework and used in arbitrary monads. Unfortunately, space prohibits the inclusion of any further examples here.

2.5 A family of state monads

We have already described the use of monads to model programs with state using the *State* datatype in Section 2. 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:

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

An expression of the form $update\ f$ denotes the computation which 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 return the current state are easily described in terms of *update*.

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*:

$$\begin{array}{l} \textbf{instance}\ StateMonad\ State\ s\ \textbf{where} \\ \quad update\ f\ \ = \ ST\ (\backslash s \to\ (s, f\ s)) \end{array}$$

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

$$\textbf{data}\ StateM\ m\ s\ a\ \ = \ \ STM\ (s \to m\ (a, s))$$

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` \(xs, 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)])
```

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.

2.6 Monads and substitution

The previous sections have concentrated on the use of monads to describe computations. Monads also have a useful interpretation as a general approach to substitution. This in turn provides another application for constructor classes.

Taking a fairly general approach, a substitution can be considered as a function $s :: v \to t\ w$ where the types v and w represent sets of variables and the type $t\ a$ represents a set of terms, typically involving elements of type a. If t is a monad and $x :: t\ v$, then x `` `bind` `` s gives the result of applying the substitution s to the term x by replacing each occurrence of a variable v in x with the corresponding term $s\ v$ in the result. For example:

```
instance Monad Tree where
  result x          =  Leaf x
  Leaf n `bind` s   =  s n
  (l :^: r) `bind` s  =  (l `bind` s) :^: (r `bind`s)
```

With this interpretation in mind, the Kleisli composition (@@) in Section 2.4 is just the standard way of composing substitutions, while the *result* function corresponds to a null substitution. The fact that (@@) is associative with *result* as both a left and right identity follows from the standard algebraic properties of a monad.

3 A formal treatment of the type system

Having outlined a number of different examples motivating the use of constructor classes, this section provides a more formal description of the underlying type system. In particular, we show how the use of constructor classes is suitable for use 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 higher-order polymorphism (for example, allowing universal quantification over constructors of kind $* \to *$) and overloading.

For reasons of space, many of the technical details have been omitted from this presentation. 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 [7] and documented more fully in [8].

3.1 Constructors, substitutions and predicates

In order to work more formally with the use of constructor classes it is convenient to explicitly 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) of kind 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 4. Note that, other than requiring that the function space constructor $\to$ be included as an element of $C^{*\to*\to*}$, 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* which 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 $(* \to *)$ and the *StateMonad* class has arity $(* \to *, *)$. 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. For our purposes, the $\Vdash$ relation must be transitive, closed under substitution and monotonic in the sense that $P \Vdash Q$ whenever $P \supseteq Q$. 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:

$$\begin{array}{lcl} P & \Vdash & \{\textit{Functor List}, \textit{Monad List}\} \\ \{\textit{Monad}\ m^{*\to*}\} & \Vdash & \{\textit{Functor}\ m^{*\to*}\} \\ \{\textit{StateMonad}\ m^{*\to*}\ s^*\} & \Vdash & \{\textit{Monad}\ (m^{*\to*}\ s^*)\} \end{array}$$

In practice, some restrictions on the definition of $\Vdash$ will be needed to ensure decidability of type checking. We will return to this point in Section 3.5.

3.2 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:

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

(P and T 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.

Terms are written using the standard language based on simple untyped λ-calculus with the addition of a *let* construct to enable the definition and use of polymorphic (and in this case, overloaded) terms:

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

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

3.3 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 Figure 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 the ML type system; 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 $CV(X)$ is used to denote the set of all the 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 which is constrained either by the type assignment A or the predicate set P.

3.4 Type inference

The rules in Figure 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 which avoids these problems is presented in Section 3.4.2 following a preliminary description of unification for constructors in Section 3.4.1.

$$(var)\quad \frac{(x:\sigma) \in A}{P \mid A \vdash x : \sigma}$$

$$(\rightarrow E)\quad \frac{P \mid A \vdash E : \tau' \rightarrow \tau \quad P \mid A \vdash F : \tau'}{P \mid A \vdash EF : \tau}$$

$$(\rightarrow I)\quad \frac{P \mid A_x, x:\tau' \vdash E : \tau}{P \mid A \vdash \lambda x.E : \tau' \rightarrow \tau}$$

$$(let)\quad \frac{P \mid A \vdash E : \sigma \quad Q \mid A_x, x:\sigma \vdash F : \tau}{P \cup Q \mid A \vdash (\textbf{let } x = E \textbf{ in } F) : \tau}$$

$$(\Rightarrow E)\quad \frac{P \mid A \vdash E : \pi \Rightarrow \rho \quad P \Vdash \{\pi\}}{P \mid A \vdash E : \rho}$$

$$(\Rightarrow I)\quad \frac{P \cup \{\pi\} \mid A \vdash E : \rho}{P \mid A \vdash E : \pi \Rightarrow \rho}$$

$$(\forall E)\quad \frac{P \mid A \vdash E : \forall\alpha^\kappa.\sigma \quad C \in C^\kappa}{P \mid A \vdash E : [C/\alpha^\kappa]\sigma}$$

$$(\forall I)\quad \frac{P \mid A \vdash E : \sigma \quad \alpha^\kappa \notin CV(A) \cup CV(P)}{P \mid A \vdash E : \forall\alpha^\kappa.\sigma}$$

Figure 1: ML-like typing rules for constructor classes

3.4.1 Unification of constructor expressions

Unification algorithms are often required in type inference algorithms to ensure that 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 [11]) 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 Figure 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 con-

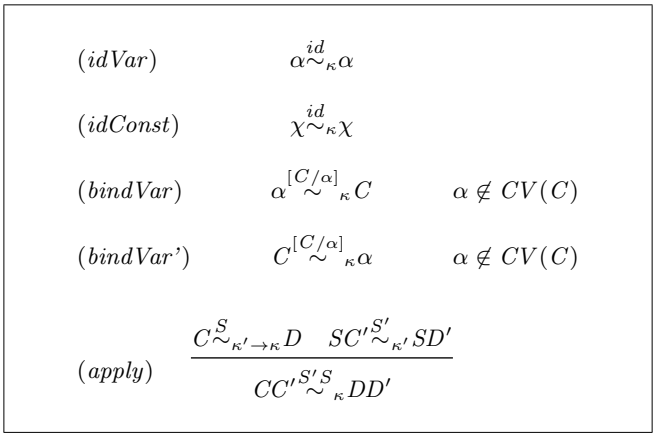

Figure 2: Kind-preserving unification

structor of the form DD' is only possible 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 $\kappa' \to \kappa$. This is a consequence of the fact that there are non non-trivial equivalences between constructor expressions. This 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.

Given these observations, we can use standard techniques to prove the following theorem:

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 Figure 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 Figure 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'*) (in other words, to ensure 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 relatively 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.

3.4.2 A type inference algorithm

The rules in Figure 3 provide an alternative to those of Figure 1 with a single rule for each syntactic construct in the language of terms. Each judgement is an expression of the form $P \mid TA \vdash^{\mathrm{W}} E : \tau$ where P is a set of predicates, T is a (kind-preserving) substitution, A is a type assignment, E is a term and τ is an element of C^*.

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 $P \mid TA \vdash^{\mathrm{W}} E : \tau$. The only way that this process can ever fail is if E contains a free variable which 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 $P \mid TA \vdash^{\mathrm{W}} E : \tau$, 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 [4], 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 which 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.*
- *$P \mid TA \vdash^{\mathrm{W}} E : \tau$ 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 [7, 8] to which we refer the reader for further details.

3.5 Coherence and decidability of type checking

Upto 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 which make the use of overloading explicit. Different forms of evidence value 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 use of *dictionaries* as proposed by Wadler and Blott [13].

In order to justify this approach, it is important to show that any two potentially distinct translations of a given term are semantically equivalent. Following [2], we refer

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

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

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

$$(let)^{\mathrm{W}} \quad \frac{P \mid TA \overset{\mathrm{W}}{\vdash} E : \tau \quad P' \mid T'(TA_x, x{:}Gen(TA, P \Rightarrow \tau)) \overset{\mathrm{W}}{\vdash} F : \tau'}{P' \mid T'TA \overset{\mathrm{W}}{\vdash} (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau'} \qquad \text{where } Gen(A, \rho) = \forall(CV(\rho) \setminus CV(A)).\rho$$

Figure 3: Type inference algorithm W

to this as a *coherence* property of the type system. As we hinted in Section 1.1, 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 [6]).

Once again, we refer the reader to [8] for further details and background on the issues raised in this section.

4 Kind inference

The biggest difference between the formal type system described in Section 3 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, and
3. the kind of each universally quantified variable in a type scheme, needed to generate new variables of the appropriate kind when a type scheme is instantiated using $(var)^{\mathrm{W}}$.

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 then solve these constraints to obtain the required kinds.

Item (1) will be dealt with more fully in Sections 4.1 and 4.2. To illustrate the basic ideas for an example involving items (2) and (3), recall the definition of the constructor class *Functor* from Section 1.2:

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

Using the fact that $(\to) :: * \to * \to *$, and that both a and b are used as arguments to $\to$ in the expression $(a \to b)$, 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 $* \to *$ as expected. Thus *map* has type:

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

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

4.1 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:

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

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 $F\ \tau_1\ \ldots\ \tau_n$ which allows the symbol F to be used as a function of type $\tau_1 \to \ldots \to \tau_n \to \chi\ a_1\ \ldots\ a_m$ to construct values of the new type.

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

$$\kappa_1 \to \ldots \to \kappa_m \to *$$

where $\kappa_1, \ldots, \kappa_m$ are appropriate kinds for $a_1, \ldots, a_m$ respectively that can be determined by a process of kind inference.

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:

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

we can infer that *Fork* has kind $\kappa \to *$ 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 $* \to *$. This is consistent with the interpretation of datatype definitions in Haskell where all variables are expected to have kind $*$.

4.2 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 declaration of the form:

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

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. The current implementation does not allow the use of 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 3.5. 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 types involving synonyms during the kind inference process. For example, given the synonym definitions:

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

we find that, for the purposes of kind inference:

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

5 Additional comments

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 decision to exclude any form of abstraction from the language of constructors is essential to ensure the tractability of the whole system. At the same time, this also results in some limitations for the programmer. For example, having defined:

$$\textbf{data}\ \mathit{State}\ s\ a\ =\ \mathit{ST}\ (s \to (a, s))$$

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 introducing the *State* data type since the constructor expression $s \to (a, s)$ is not in a suitable form.

References

[1] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, volume 1, part 2, 1991.

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

[3] K. Chen, P. Hudak, and M. Odersky. Parametric type classes (Extended abstract). *ACM conference on LISP and Functional Programming*, San Francisco, California, June 1992.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *8th Annual ACM Symposium on Principles of Programming languages*, 1982.

[5] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN notices*, 27, 5, May 1992.

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

[7] M.P. Jones. A theory of qualified types. In *European symposium on programming*. Springer Verlag LNCS 582, 1992.

[8] M.P. Jones. Qualified types: Theory and Practice. D. Phil. Thesis. Programming Research Group, Oxford University Computing Laboratory. July 1992.

[9] M.P. Jones. Programming with constructor classes (preliminary summary). In *Draft Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming*, Ayr, Scotland, July 1992.

[10] E. Moggi. Computational lambda-calculus and monads. *IEEE Symposium on Logic in Computer Science*, Asilomar, California, June 1989.

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

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

[13] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *16th ACM annual symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[14] P. Wadler. Comprehending Monads. *ACM conference on LISP and Functional Programming*, Nice, France, June 1990.

[15] P. Wadler. The essence of functional programming. In *19th Annual Symposium on Principles of Programming Languages*, Santa Fe, New Mexico, January 1992.