Functional Programming with Overloading and Higher-Order Polymorphism

Mark P. Jones

Department of Computer Science, University of Nottingham, University Park, Nottingham NG7 2RD, UK.

Abstract. The Hindley/Milner type system has been widely adopted as a basis for statically typed functional languages. One of the main reasons for this is that it provides an elegant compromise between flexibility, allowing a single value to be used in different ways, and practicality, freeing the programmer from the need to supply explicit type information. Focusing on practical applications rather than implementation or theoretical details, these notes examine a range of extensions that provide more flexible type systems while retaining many of the properties that have made the original Hindley/Milner system so popular. The topics discussed, some old, but most quite recent, include higher-order polymorphism and type and constructor class overloading. Particular emphasis is placed on the use of these features to promote modularity and reusability.

1 Introduction

The Hindley/Milner type system [6, 19, 3], hereafter referred to as HM, represents a significant and highly influential step in the development of type systems for functional programming languages. In our opinion, the main reason for this is that it combines the following features in a single framework:

- **Type security**: soundness results guarantee that well-typed programs cannot 'go wrong'. This should be compared with the situation in dynamically typed languages like Scheme where run-time tests are often required to check that appropriate types of value are used in a particular context, and the execution of a program may terminate if these tests fail.
- **Flexibility**: polymorphism allows the use and definition of functions that behave uniformly over all types. This should be compared with the situation in monomorphically typed languages where it is sometimes necessary to produce several versions of a particular function or algorithm to deal with different types of values. Standard examples include swapping a pair of values, choosing the minimum of two values, sorting an array of values, etc.
- **Type inference**: there is an effective algorithm which can be used to determine that a given program term is well-typed and, in addition, to calculate its most general (principal) type, without requiring any type annotations in the source program. In practice, even though it is not required, programmers often choose to include explicit type information in a program as a form of

documentation. In this case, the programmer benefits from a useful consistency check that is obtained automatically by comparing the declared types with the results of the type inference algorithm.

- **Ease of implementation**: the type inference algorithm is easy to implement and behaves well in practice. Polymorphism itself is also easy to implement, for example, by using a uniform (or *boxed*) representation that is independent of the type of the values concerned.

As a result, HM has been used as a basis for several widely used functional languages including Hope [2], Standard ML [20], Miranda[1] [27] and Haskell [7].

The features listed above make HM an attractive choice for language designers, but we should also recognize that it has some significant limitations. In particular, while HM polymorphism allows the definition of functions that behave uniformly over all types, it does not permit:

- **Restricted polymorphism/overloading**: the use or definition of functions that are can be used for some, but not necessarily all, types, with potentially different behaviours in each case.
- **Higher-order polymorphism**: the use or definition of functions that behave uniformly over all type constructors.
- **Polymorphic arguments**: the use or definition of functions with polymorphic arguments that can be used at different instances in the body of the function.

These notes describe how the first two of these restrictions can be relaxed, while preserving many of the properties that have made HM so popular. The third item, to permit the use of function arguments with polymorphic components, is a topic of current research. For example, one approach that we are investigating is to use explicit type annotations to supplement the results of type inference. However, for reasons of space, this will not be addressed any further here.

Our main aim is to illustrate practical applications of these extended type systems using a variety of functional programming examples. To this end, we avoid the distraction of long technical discussions about either the underlying type theory or the implementation; these have already been covered in depth elsewhere. We place particular emphasis on the use of these extensions to promote modularity, extensibility and reusability at the level of the core language[2].

The main subjects of these notes are illustrated in Fig. 1. We start with a brief review of the original Hindley/Milner type system (Sect. 2). The first extension of HM that we consider is to support overloading using a system of *type classes*, as described in Sect. 3. Introduced, at least in the form used here, by Wadler and Blott [30], type classes have been adopted as part of the definition of the standard for the functional programming language Haskell [7]. Type classes are

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

[2] i.e. for programming in the small. These notes do not address the subject of modularity for programming in the large. Such goals are better met by powerful module systems, for example, the structures and functors of Standard ML.

particularly useful for describing the implementation of standard polymorphic operators such as equality, arithmetic and printing. We also include examples to show how they can be used to provide a flexible framework for other applications.

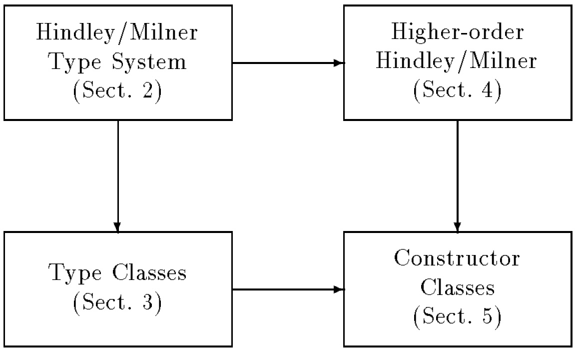

Fig. 1. A summary of the main subjects covered in these notes

Another way to extend HM is to make use of a form of higher-order polymorphism, i.e. polymorphism over type constructors as well as types. This is described in Sect. 4. The generalization to the higher-order case is surprisingly straightforward; it is most useful as a tool for specifying datatypes but it does not significantly increase the expressiveness of the type system as a whole.

However, there is a significant increase in expressiveness when we combine higher-order polymorphism with a class based overloading mechanism, leading to the system of *constructor classes* described in Sect. 5. For example, we show how constructor classes can be used to capture general patterns of recursion of a large family of datatypes, to support the use of monads and to construct modular programming language interpreters.

We assume familiarity with the basic techniques of functional programming, as described by Bird and Wadler [1] for example, and with the concrete syntax and use of Haskell [7] and/or Gofer [12]; these are the languages that were used to develop the examples shown in these notes.

2 The Hindley/Milner Type System

These notes assume that the reader is already familiar with the use of HM in languages like Standard ML or Haskell. However, it seems useful to start with a summary of what we consider the most important features of HM for the purposes of this paper.

The goal of the type system is to assign a type to each part of an input program, guaranteeing that execution of the program will not go wrong, i.e.

that it will not encounter a run-time type error. Terms that cannot be assigned a type will result in a compile-time type error.

One of the most striking differences between HM and many other type systems is the fact that the most general type of a term can be inferred without the need for type annotations. In some cases, the most general type is monomorphic:

```
not       :: Bool -> Bool
not False = True
not True  = False
```

In other cases, the most general type is polymorphic:

```
identity  :: a -> a
identity x = x
```

The type variable `a` appearing in the type of `identity` here represents an arbitrary type; if the argument `x` to `identity` has type `a`, then so will the result of `identity x`. Another simple example is the `length` function which is used to calculate the length of a list. One way to define `length` is as follows:

```
length        :: [a] -> Int
length []      = 0
length (x:xs) = 1 + length xs
```

In this example, the appearance of the type variable `a` in the type of `length` indicates that this single function `length` may be applied to any list, regardless of the type of values that it contains.

In some treatments of HM, the types of the `identity` and `length` functions about might be written more formally as $\forall a.a \to a$ and $\forall a.[a] \to Int$, respectively, so that polymorphic type variables are explicitly bound by a universal quantifier. These quantifiers are left implicit in the concrete syntax of Haskell. However, it is sometimes convenient to write the types of particular functions using the quantifier notation to emphasize the role of polymorphism.

3 Type Classes

The HM type system is convenient for many applications, but there are some important functions that cannot be given a satisfactory type. There are several well-rehearsed examples, including arithmetic and equality operators, which illustrate this point:

- If we treat addition as a monomorphic function of type `Int -> Int -> Int`, then it can be used to add integer values, but it is not as general as we might have hoped because it cannot also be used to add floating point quantities. On the other hand, it would not be safe to use a polymorphic type such as `a -> a -> a` for the addition operator because this allows `a` to be *any* type, but addition is only defined for numeric types.

– If we treat equality as a monomorphic function of type `T -> T -> Bool` for some type constructor `T`, then it is less general than we might have hoped because it cannot be used to compare values of other types. However, a polymorphic type like `a -> a -> Bool` would not be appropriate because it includes the case where `a` is a function type, and there is no computable equality for functional values.

In both of these examples we find ourselves in a position where monomorphic types are too restrictive and fully polymorphic types are too general. Type classes, described in some detail below, are an attempt to remedy such problems. This is achieved by providing an intermediate step between monomorphic and polymorphic types, i.e. by allowing the definition of values that can be used over a range of types, without requiring that they can be used over *all* types.

3.1 Basic principles

Type classes can be understood and used at several different levels. To begin with, we restrict our attention to the built-in classes of Haskell. Later, we will describe how these classes can be extended, and how new classes can be introduced.

The Haskell *standard prelude* is a large library of useful types, type classes, and functions, that is automatically imported into every Haskell program. The prelude datatypes include Booleans (`Bool`), integers (fixed precision `Int` and arbitrary precision `Integer`), rationals (`Ratio`), complex numbers (`Complex`), floating point values (single precision `Float` and double precision `Double`), characters (`Char`), lists, tuples, arrays, etc.

The prelude also defines a number of *type classes*, which can be thought of as sets of types whose members are referred to as the *instances* of the class. If `C` is the name of a class and `a` is a type, then we write `C a` to indicate that `a` is an instance of `C`. Each type class is in fact associated with a collection of operators and this has an influence on the choice of names. For example, the `Eq` class contains types whose elements can be tested for equality, while the class `Ord` contains types whose elements are ordered. We will return to this again below, but for the time being, we will continue to think of classes as sets of types.

The instances of a class are defined by a collection of *instance declarations*. For example, the instances of the `Eq` class are described by the declarations:

```
instance Eq Bool
instance Eq Char
instance Eq Int
instance Eq Integer
instance Eq Float
instance Eq Double
instance Eq a => Eq [a]
instance (Eq a, Eq b) => Eq (a,b)
instance (Eq a, Eq b, Eq c) => Eq (a,b,c)
```

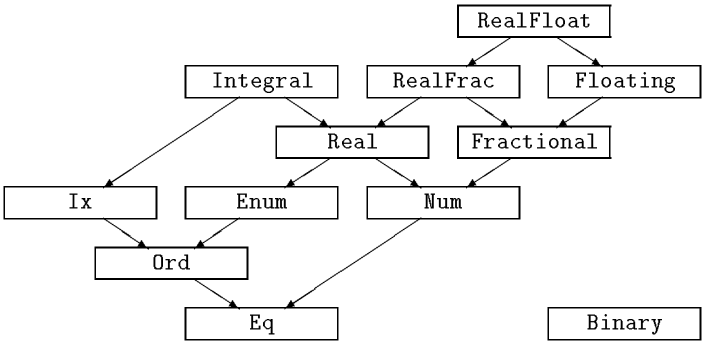

Fig. 2. The hierarchy of standard Haskell type classes

```
instance (Eq a, Eq b, Eq c, Eq d) => Eq (a,b,c,d)
...
```

The first few lines indicate that the types `Bool`, `Char`, `Int`, `Integer`, `Float` and `Double` are instances of `Eq` class. The remaining declarations include a *context* to the left of the `=>` symbol. For example, the `instance Eq a => Eq [a]` declaration can be read as indicating that, if `a` is an instance of `Eq`, then so is the list type `[a]`. The very first declaration tells us that `Bool` is an instance of `Eq`, and hence so are `[Bool]`, `[[Bool]]`, ...

More formally, the effect of these instance declarations is to define `Eq` as the smallest solution of the equation:

$$\begin{aligned}\texttt{Eq} = {} & \{\,\texttt{Bool}, \texttt{Char}, \texttt{Int}, \texttt{Integer}, \texttt{Float}, \texttt{Double}\,\} \cup \\ & \{\,\texttt{[}\tau\texttt{]} \mid \tau \in \texttt{Eq}\,\} \cup \\ & \{\,\texttt{(}\tau_1,\ \tau_2\texttt{)} \mid \tau_1, \tau_2 \in \texttt{Eq}\,\} \cup \\ & \{\,\texttt{(}\tau_1,\ \tau_2,\ \tau_3\texttt{)} \mid \tau_1, \tau_2, \tau_3 \in \texttt{Eq}\,\} \cup \\ & \{\,\texttt{(}\tau_1,\ \tau_2,\ \tau_3,\ \tau_4\texttt{)} \mid \tau_1, \tau_2, \tau_3, \tau_4 \in \texttt{Eq}\,\} \cup \\ & \ldots\end{aligned}$$

The Haskell prelude defines a number of other classes, as illustrated in Fig. 2. Not all of the standard classes are infinite like `Eq`. For example, the prelude includes instance declarations which defines the classes `Integral` and `RealFloat` of integer and floating point number types, respectively, to be equivalent to:

$$\begin{aligned}\texttt{Integral} &= \{\,\texttt{Int}, \texttt{Integer}\,\} \\ \texttt{RealFloat} &= \{\,\texttt{Float}, \texttt{Double}\,\}\end{aligned}$$

The prelude also specifies inclusions between different classes; these are illustrated by arrows in Fig. 2. For example, the `Ord` class is a subset of `Eq`: every instance of `Ord` is also an instance of `Eq`. These inclusions are described by a collection of *class declarations* like the following:

```
class Eq a
class (Eq a) => Ord a
class (Eq a, Text a) => Num a
...
```

The last declaration shown here specifies that `Num` is a subset of both `Eq` and `Text`[3]. The inclusions between classes are verified by the compiler, and are of most use in reasoning about whether a particular type is an instance of a given class.

Finally, on top of the type, class, and instance declarations, the standard prelude defines a large collection of primitive values and general purpose functions. Some of the values defined in the prelude have monomorphic types:

```
not     :: Bool -> Bool             -- Boolean negation
ord     :: Char -> Int              -- Character to ASCII code
```

Others have polymorphic types:

```
(++)    :: [a] -> [a] -> [a]        -- List append
length :: [a] -> Int                -- List length
```

There are also a number of functions with restricted polymorphic types:

```
(==)    :: Eq a => a -> a -> Bool  -- Test for equality
min     :: Ord a => a -> a -> a    -- Find minimum
show    :: Text a => a -> String   -- Convert to string
(+)     :: Num a => a -> a -> a    -- Addition
```

We refer to these types as being restricted because they include type class constraints. For instance, the first example tells us that the equality operator, `(==)`, can be treated as a function of type `a -> a -> Bool`. But the choice for `a` is not arbitrary; the context `Eq a` will only be satisfied if `a` is an instance of `Eq`. Thus we can use `'a'=='b'` to compare character values, or `[1,2,3]==[1,2,3]` to compare lists of integers, but we cannot use `id == id`, where `id` is the identity function, because the class `Eq` does not contain any function types. In a similar way, the `(+)` operator can be used to add two integer values or two floating point numbers because these are all instances of `Num`, but it cannot be used to add two lists, say, because Haskell does not include lists in the `Num` class; any attempt to add two list values will result in a compile-time type error.

Class constraints may also appear in the types of user-defined functions that make use, either directly or indirectly of prelude functions with restricted polymorphic types. For example, consider the following definitions:

```
> member xs x  = any (x==) xs
> subset xs ys = all (member ys) xs
```

[3] This aspect of Haskell syntax can sometimes be confusing. It might have been better if the roles of the expressions on the left and right hand side of `=>` were reversed so that `Num a => (Eq a, Text a)` could be read as an implication; if `a` is an instance of `Num`, then `a` is also an instance of `Eq` and `Text`.

The definition of `member` takes a list `xs` of type `[a]` and a value `x` of type `a`, and returns a boolean value indicating whether `x` is a member of `xs`; i.e. whether any element of `xs` is equal to `x`. Since `(==)` is used to compare values of type `a`, it is necessary to restrict our choice of `a` to instances of `Eq`. In a similar way, it follows that `subset` must also have a restricted polymorphic type because it makes use of the `member` function. Hence the types of these two functions are:

```
> member      :: Eq a => [a] -> a -> Bool
> subset      :: Eq a => [a] -> [a] -> Bool
```

These functions can now be used to work with lists of type `[a]` for any instance `a` of `Eq`. But what if we want to work with user-defined datatypes that were not mentioned in the prelude? In Haskell, this can be dealt with by including a list of classes as part of the datatype definition. For example:

```
> data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
>            deriving (Eq, Ord, Text)
```

The second line, `deriving (Eq, Ord, Text)`, is a request to the compiler to extend the three named classes to include the `Day` datatype, and to generate appropriate versions of any overloaded operators for values of type `Day`. For example:

```
  ? member [Mon,Tue,Wed,Thu,Fri] Wed
  True
  ? subset [Mon,Sun] [Mon,Tue,Wed,Thu,Fri]
  False
  ?
```

Instances of a type class that are obtained in this way are described as *derived instances*. In the general case, a derived instance may require a context. For example, the following datatype definition:

```
> data Either a b = Left a | Right b  deriving (Eq, Ord)
```

will result in two derived instances:

```
  instance (Eq a, Eq b) => Eq (Either a b)
  instance (Ord a, Ord b) => Ord (Either a b)
```

3.2 Defining instances

The simple approach to type classes described above works quite well until you run into a situation where either you want to include a new datatype in a class for which derived instances are either not permitted[4] or not suitable because

[4] Haskell only permits derived instances of `Eq`, `Ord`, `Text`, `Ix`, `Enum`, and `Binary`. In some cases, there are additional restrictions on the form of the datatype definition when a derived instance is requested.

the rules for generating versions of overloaded functions do not give the desired semantics. For example, suppose that we define a set datatype using lists to store the members of each set, but without worrying about duplicate values or about the order in which the elements are listed. A datatype definition like:

```
data Set a = Set [a]  deriving (Eq)
```

would result in an implementation of equality satisfying:

```
Set xs == Set ys  =  xs == ys
```

where the equality on the right hand side is the equality on lists. Thus the sets `Set [1,2]` and `Set [2,1,2]` would be treated as being distinct because their element lists differ, even though they are intended to represent the same set.

In situations like this, it is possible for a programmer to provide their own semantics for the overloaded operators associated with a particular class. To start with, we need to take a more careful look at the full definition of the `Eq` class:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x == y)
```

This indicates that, to include a type `a` as an instance of the `Eq` class, the programmer must supply definitions for the `(==)` and `(/=)` functions, both of type `a -> a -> Bool`. In fact, the final line eases the programmers task a little by providing a *default definition* for `(/=)` that will be used if the programmer does not give a suitable definition of their own. As a result, all that the programmer has to do is to provide a definition for `(==)`; i.e. to define what it means for two values of type `a` to be equal.

Returning to the example above, we can define the set datatype as:

```
> data Set a = Set [a]
```

and we use the following in place of a derived instance:

```
> instance Eq a => Eq (Set a) where
>    Set xs == Set ys  =  subset xs ys && subset ys xs
```

This properly captures the intended semantics of set equality, i.e. that two sets are equal precisely when each is a subset of the other, indicating that they have the same members.

It is important to notice that a class can be arbitrarily extended to include new instances, without any modification to the original class definition. This gives a high degree of extensibility and modularity in many cases.

3.3 Defining classes

We have now seen how a programmer can use either derived instances or their own implementations to specify the instances of one the standard Haskell classes. This may be all that some programmers will ever need to know about Haskell type classes. However, for some applications, it is useful for a programmer to be able to define new classes. We will give a number of examples to illustrate this point below.

In defining a new class, the first step is to decide exactly what common properties we expect the instances to share, and to decide how this should be reflected in the choice of the operators listed in the class declaration. However, it is important to recognize that overloading is only appropriate if the meaning of a symbol is uniquely determined by the types of the values that are involved. For instance, some might consider the following example, using classes to describe monoids, as an abuse of the system because monoid structures are not uniquely determined by type.

```
class Monoid a where
  e  :: a
  op :: a -> a -> a

instance Monoid [a] where
  e  = []    -- Empty list
  op = (++)  -- List append

instance Monoid (a -> a) where
  e  = id    -- Identity function
  op = (.)   -- Function composition

instance Monoid Int where
  e  = 0
  op = (+)
```

The final instance declaration here is particularly difficult to justify; there is another equally good way to define a monoid structure on `Int`egers using `e=1` and `op=(*)`, i.e. multiplication. There does not seem to be any good reason why we should favour either one of these alternatives over the other.

We hope that the reader will find that most of the applications of type classes in this paper, and of constructor classes in later sections, are well suited to overloading, with a single natural implementation for each instance of a particular overloaded operator.

Trees. From search trees to the representation of parsed terms in a compiler, trees, of one form or another, must rate as one of the most widely used data structures in functional programming. There are many different kinds of tree structure, with variations such as the number of branches out of each node, and the type of values used as labels. The following datatype definitions help to illustrate the point:

- Simple binary trees, with a value of type a at each leaf node.

```
> data BinTree a = Leaf a
>                | BinTree a :^: BinTree a
```

- Labelled trees with a value of type a at each leaf node, and a value of type l at each interior node:

```
> data LabTree l a = Tip a
>                  | LFork l (LabTree l a) (LabTree l a)
```

- Binary search trees, with data values of type a in the body of the tree. These values would typically be used in conjunction with an ordering on the elements of type a in order to locate a particular item in the tree.

```
> data STree a = Empty
>              | Split a (STree a) (STree a)
```

- Rose trees, in which each node is labelled with a value of type a, and may have an arbitrary number of subtrees:

```
> data RoseTree a = Node a [RoseTree a]
```

- Abstract syntax, for example, the following datatype might be used to represent λ-expressions in a simple interpreter. In this case, the leaf nodes correspond to variables while the interior nodes represent either applications or abstractions:

```
> type Name = String
> data Term = Var Name        -- variable
>           | Ap  Term Term   -- application
>           | Lam Name Term   -- lambda abstraction
```

On the other hand, there are some strong similarities between these datatypes, and many familiar concepts, for example, depth, size, paths, subtrees, etc. can be used with any of these different kinds of tree.

Consider the task of calculating the depth of a tree. Normally, it would be necessary to write a different version of the depth calculation for each different kind of tree structure that we are interested in. However, using type classes it is possible to take a more general approach by defining a class of tree-like data types. Starting with the observation that, whichever datatype we happen to be using, every tree has a number of subtrees, we are lead to the following simple characterization of tree-like data structures:

```
> class Tree t where
>     subtrees :: t -> [t]
```

In words, `subtrees t` generates the list of (proper) subtrees of a given tree, `t`. There are many properties of trees that this does not address, for example, the use of labels, but of course, these are exactly the kind of things that we need to ignore to obtain the desired level of generality.

The following instance declarations can be used to include each of the five tree-like data structures listed above as an instance of the `Tree` class:

```
> instance Tree (BinTree a) where
>     subtrees (Leaf n)  = []
>     subtrees (l :^: r) = [l,r]

> instance Tree (LabTree l a) where
>    subtrees (Tip x)       = []
>    subtrees (LFork x l r) = [l,r]

> instance Tree (STree a) where
>     subtrees Empty         = []
>     subtrees (Split x l r) = [l,r]

> instance Tree (RoseTree a) where
>     subtrees (Node x gts) = gts

> instance Tree Term where
>     subtrees (Var _)   = []
>     subtrees (Ap f x)  = [f,x]
>     subtrees (Lam v b) = [b]
```

With these definitions in place, we can start to construct a library of useful functions that can be applied to any kind of tree that has been included in the `Tree` class. For example, the following definitions can be used to determine the depth and the size (i.e. the number of nodes) in any given tree:

```
> depth  :: Tree t => t -> Int
> depth   = (1+) . foldl max 0 . map depth . subtrees

> size   :: Tree t => t -> Int
> size    = (1+) . sum . map size . subtrees
```

There are more efficient ways to describe these calculations for particular kinds of tree. For example, the definition of `size` for a `BinTree` could be simplified to:

```
  size (Leaf n)  = 1
  size (l :^: r) = size l + size r
```

without constructing the intermediate list of `subtrees`. However, it is entirely possible that this more efficient implementation could be obtained automatically in a compiler, for example, by generating specialized versions of overloaded functions [11].

Another simple example of an algorithm that can be applied to many different kinds of tree is the process of calculating the list of paths from the root node to each of the leaves. In specific cases, we might be tempted to use sequences of labels, or sequences of directions such as 'left' and 'right' to identify a particular path in the tree. Neither of these is possible in our more general framework. Instead, we will identify each path with the corresponding sequence of subtrees. This leads to the following definition:

```
> paths              :: Tree t => t -> [[t]]
> paths t | null br   = [ [t] ]
>         | otherwise = [ t:p | b<-br, p<-paths b ]
>           where br = subtrees t
```

The definitions of depth-first and breadth-first search can also be expressed in our current framework, each yielding a list of subtrees in some appropriate order:

```
> dfs    :: Tree t => t -> [t]
> dfs t   = t : concat (map dfs (subtrees t))

> bfs    :: Tree t => t -> [t]
> bfs     = concat . lev
>  where lev t = [t] : foldr cat [] (map lev (subtrees t))
>        cat   = combine (++)

> combine                :: (a -> a -> a) -> ([a] -> [a] -> [a])
> combine f (x:xs) (y:ys) = f x y : combine f xs ys
> combine f []     ys     = ys
> combine f xs     []     = xs
```

The depth-first algorithm given here is straightforward. We refer the reader to [8] for further details and explanation of the breadth-first algorithm. It may seem strange to define functions that return the complete list of every subtree in a given tree. But this approach is well-suited to a lazy language where the list produced by the search may not be fully evaluated. For example, if `p` is some predicate on trees, then we might use the function:

```
  head . filter p . dfs
```

to find the first node in a depth first search of a tree that satisfies `p`, and, once it has been found, there will not be any need to continue the search.

As a final example, we sketch the implementation of a function for drawing character-based diagrams of arbitrary tree values. This might, for example, be useful as a way of visualizing the results of simple tree-based algorithms. The following examples show the output of the function for two different kinds of tree:

```
? drawTree ((Leaf 1 :^: Leaf 2) :^: (Leaf 3 :^: Leaf 4))
--@--@--1
  |  |
  |  `--2
  |
  `--@--3
     |
     `--4

? drawTree (Lam "f" (Ap (Ap (Var "f") (Var "x")) (Var "y")))
--\f--@--@--f
      |  |
      |  `--x
      |
      `--y

?
```

The tree-drawing algorithm is based on a function:

```
> drawTree' :: Tree t => (t -> String) -> t -> [String]
```

The first argument of `drawTree'` is a function of type `(t -> String)` that produces a text string corresponding to the label (if any) of the root node of a tree of type `t`. The second argument of `drawTree'` is the tree itself. The result of the function is a list of strings, each corresponding to a single line of output, that can be combined using the standard `unlines` function to produce a single string with a newline character after each line.

To save the trouble of specifying a labelling function for `drawTree`, we define a subclass of `Tree` that provides appropriate functions for labelling and drawing:

```
> class Tree t => DrawTree t where
>     drawTree :: t -> String
>     labTree  :: t -> String
>
>     drawTree  = unlines . drawTree' labTree
```

For example, the instance declaration that we use for the `Term` datatype is as follows:

```
> instance DrawTree Term where
>     labTree (Var v)    = v
>     labTree (Ap _ _)   = "@"
>     labTree (Lam v _) = "\\"++v
```

We leave the construction of `drawTree'` and the definition of instances of the `DrawTree` class for the other tree types defined above as an exercise for the reader.

Duality and the De Morgan Principle. Our next example is inspired by the work of Turner [26] to extend the concept of duality on Boolean algebras, and the well-known De Morgan principle, to the list datatype. We start by defining a class `Dual` of types with a function `dual` that maps values to appropriate duals:

```
> class Dual a where
>     dual :: a -> a
```

The only property that we will require for an instance of `Dual` is that the corresponding implementation of `dual` is self-inverse:

```
  dual . dual = id
```

The easiest way to deal with classes constrained by laws such as this is to treat the laws as proof obligations for each instance of the class that is defined, assuming that the laws are satisfied for each of the subinstances involved.

The first example of duality is the inversion of boolean values given by:

```
> instance Dual Bool where
>     dual = not
```

For example, `dual True = False` and `dual False = True`. It is easy to see that this declaration satisfies the self-inverse property since because `not . not` is the identity on booleans.

To make any further progress, we need to extend the concept of duality to function values:

```
> instance (Dual a, Dual b) => Dual (a -> b) where
>     dual f = dual . f . dual
```

The proof that this satisfies the self-inverse law is straightforward:

```
  dual (dual f)
    = { definition of dual, twice }
      dual . dual . f . dual . dual
    = { Assuming dual . dual = id for Dual a, Dual b }
      id . f . id
    = { ((.),id) monoid }
      f
```

The `dual` function distributes over application and composition of functions:

```
 dual (f x)    =  (dual f) (dual x)
 dual (f . g)  =  dual f . dual g
```

We leave formal verification of these properties as a straightforward exercise for the reader. These laws can be used to calculate duals. For example, consider the definition of conjunction in the Haskell standard prelude:

```
    True  && x    = x
    False && x    = False
```

Applying `dual` to both sides of each equation and simplifying, we obtain:

```
dual (&&) False x = x
dual (&&) True  x = True
```

which shows that `dual (&&) = (||)`, i.e. that disjunction (or) is the dual of conjunction (and), as we would expect from the standard version of De Morgan's theorem for boolean values.

There are a variety of other applications of duality. Turner's work was motivated by the duality on finite lists that arises from the list reverse function:

```
> instance Dual a => Dual [a] where
>     dual = reverse . map dual
```

If we restrict our attention to finite lists, then `reverse . reverse` is the identity function and it is easy to show that this definition satisfies the self-inverse law. We can make direct use of `dual` in calculations such as:

```
? dual head [1..10]          -- dual head = last
10
? dual tail [1..10]          -- dual tail = init
[1, 2, 3, 4, 5, 6, 7, 8, 9]
? dual (++) [1,2] [3,4]      -- dual (++) = flip (++)
[3, 4, 1, 2]
?
```

The `flip` function referred to in the last example is the Haskell equivalent of the classical W combinator that switches the order of the arguments to a curried function:

```
flip       ::  (a -> b -> c) -> (b -> a -> c)
flip f x y  =  f y x
```

This can also be used to illustrate the use of the duals of the Haskell prelude functions `foldl` an `foldr`, as in the following:

```
? foldl (flip (:)) [] [1..4]
[4, 3, 2, 1]
? dual foldr (:) [] [1..4]
[4, 3, 2, 1]
?
```

In general, the two fold functions are related by the formulae:

```
dual foldr  =  foldl . flip
dual foldl  =  foldr . flip
```

We refer the reader to the text by Bird and Wadler [1] for further discussion on the relationship between `foldl` and `foldr`, and on duality for lists.

To conclude our comments about duality, we extend the framework to include integers with unary minus as the `dual` function:

```
> instance Dual Int where
>     dual = negate
```

For example:

```
? dual (+) 3 4    -- dual (+) = (+)
7
? dual max 3 5    -- dual max = min
3
? dual min 3 5    -- dual min = max
5
?
```

Computing with Lattices. A lattice is a partially ordered set with a top and a bottom value in which every pair of elements has a meet (greatest lower bound) and a join (least upper bound). There are many applications for lattices in computer science, particularly in studies of semantics and program analysis. Motivated by the study of *frontiers* and their use in *strictness analysis*, Jones [9] developed a general framework for computing with (finite) lattices using type classes. The result is an elegant system that includes a range of different types of lattice and extends easily to accommodate other kinds of lattice needed for particular applications. This compares very favourably with an earlier implementation of the same ideas that did not use type classes and, because of the limitations imposed by HM, was less robust, more awkward to work with, and harder to extend.

The most important part of Jones' framework is the definition of a class of lattices:

```
> class Eq a => Lattice a where
>     bottom, top :: a
>     meet, join  :: a -> a -> a
>     lt          :: a -> a -> Bool
>     x `lt` y     = (x `join` y) == y
```

The `lt` function, written here as an infix operator, is used to describe the partial order on the elements of the lattice. The default definition for `lt` shows how it can be defined in terms of the `join` and equality operators.

The `Bool` datatype gives one of the simplest examples of a lattice, with meet and join corresponding to conjunction and disjunction, respectively:

```
> instance Lattice Bool where
>     bottom = False
>     top    = True
>     meet   = (&&)
>     join   = (||)
```

Note that we ignore any improper elements of lattice types, in this case, just the bottom element $\perp$ of type `Bool`, since these values cannot be used without risking abnormal- or non-termination.

As a slightly more complex example, we can define the lattice structure of a product of two lattices using the declaration:

```
> instance (Lattice a, Lattice b) => Lattice (a,b) where
>     bottom            = (bottom,bottom)
>     top               = (top,top)
>     (x,y) `meet` (u,v) = (x `meet` u, y `meet` v)
>     (x,y) `join` (u,v) = (x `join` u, y `join` v)
```

It is possible to extend the `Lattice` class with other kinds of lattice, such as lattices of subsets, lattices of frontiers, lifted lattices, and lattices of functions.

We will use the problem of defining the least fixed point operator as an illustration of the use of the `Lattice` class. It is well-known that, if `f` is a monotonic function[5] on some lattice `a`, then `f` has a least fixed point which can be obtained as the limit of the sequence:

```
  iterate f bottom = [ bottom, f bottom, f (f bottom), ...
```

Assuming that the lattice in question is finite, the limit will be the first (and only) repeated value in this sequence. This translates directly to an algorithm for calculating the least fixed point, `fix f`:

```
> fix                :: Lattice a => (a -> a) -> a
> fix f               = firstRepeat (iterate f bottom)

> firstRepeat        :: Eq a => [a] -> a
> firstRepeat (x:xs) = if x==head xs then x else firstRepeat xs
```

Building on examples like these, Jones [9] shows how to define general tools for computing with lattices, including an algorithm to enumerate the elements of a finite lattice. It is beyond the scope of these notes to give any further details of these examples here.

4 A Higher-order Hindley/Milner Type System

We have already seen examples showing how HM allows the programmer to generalize with respect to types, suggesting that a polymorphic function has a uniform implementation for a range of different types. For example, the type of the `length` function in Sect. 2 is $\forall a.[a] \to Int$; this reflects the fact that the elements of a list do not play a part in the calculation of its length. However, HM does not allow us to generalize with respect to type constructors, for example to define a function:

$$\texttt{size} :: \forall t.\forall a.t(a) \to Int.$$

that could be used to give some measure of the size of an object of type $(t\ a)$ for any type constructor t, and any type a (for instance, we might expect that

[5] In the notation used here, this means that `f x `lt` f y`, whenever `x `lt` y`.

`length` would be a special case of `size`, using the list type constructor in place of the variable t).

At first glance, we may be concerned that a generalization of HM to support this weak form of *higher-order polymorphism* would quickly run into technical difficulties. For example, standard type inference algorithms require the use of a *unification* algorithm to determine when two types are equal. In the higher-order case, we need to be able to compare type constructors which might seem to imply a need for higher-order unification, known to be undecidable. In fact, the generalization of HM to support higher-order polymorphism that is sketched here is surprisingly straightforward. Many of the technical properties of HM, and their proofs, carry over with little or no change. In particular, there is an effective type inference algorithm, based on a (decidable) kinded, first-order unification process[6]. To the best of our knowledge, the only place where this has been described in the past is as an integral part of the system of constructor classes [10] which is the subject of the next section. Our goal here is to highlight the fact that the higher-order extension is independent of any use of overloading.

The extension rests on the use of a *kind* system:

$$\begin{array}{rll} \kappa ::= & * & \textit{monotypes} \\ \mid & \kappa_1 \to \kappa_2 & \textit{function kinds} \end{array}$$

Kinds are used to identify particular families of type constructors in much the same way as types are used to describe collections of values. The $*$ kind represents the set of all monotypes, i.e. nullary type constructors, while the kind $\kappa_1 \to \kappa_2$ represents constructors that take something of kind κ_1 and return something of kind κ_2. For each kind κ, we have a collection of constructors C^κ (including constructor variables α^κ) of kind κ given by:

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

This corresponds very closely to the way that most type expressions are already written in Haskell. For example, `List a` is an application of the constructor constant `List` to the constructor variable `a`. In addition, each constructor constant has a corresponding kind. For example, writing `(->)` for the function space constructor and `(,)` for pairing we have:

```
Int, Float, ()      :: *
List, BinTree       :: * -> *
(->), (,), LabTree :: * -> * -> *
```

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 $*$. The apparent mismatch between the explicitly kinded constructor expressions

[6] This is possible because the language of constructors is built up from constants and applications; in particular, there are no abstractions.

specified above and the implicit kinding used in examples can be resolved by a process of kind inference; i.e. by using standard techniques to infer kinds without the need for programmer supplied kind annotations [10].

Given this summary of the technical issues, we turn our attention to applications of the extended type system. Here, we find that, by itself, higher-order polymorphism is often too general for practical examples. For example, in the case of the `size` function described above, it is hard to construct a definition for any interesting functions of type $\forall t.\forall \alpha.t(\alpha) \rightarrow Int$[7] because we need a definition that will work for *any* type constructor t, and *any* type a. The only possibilities are functions of the form $\lambda x.n$ where n is an integer constant, all of which can be treated as having the more general type $\forall a.a \rightarrow Int$ without the need for higher-order polymorphism.

Even so, higher-order types are still useful, particularly as a means of specifying new datatypes where we can use a mixture of types and type constructors as parameters.

```
data Mu f   = In (f (Mu f))

data NatF s = Zero | Succ s
type Nat    = Mu NatF

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

The first three examples here can be used to provide a general framework for constructing recursive datatypes and corresponding recursion schemes. The fourth example is used to describe a parameterized state monad. Both of these examples will be described in the following section.

The reader may like to check the following kinds for each of the type constructors introduced above.

```
Mu     :: (* -> *) -> *
NatF   :: * -> *
Nat    :: *
StateT :: * -> (* -> *) -> * -> *
```

All of these kinds can be determined automatically without the use of kind annotations.

As a final comment, it is worth noting that the implementation of this form of higher-order polymorphism is straightforward, and that experience with practical implementations, for example, Gofer, suggests that it is also natural from a programmer's perspective.

[7] observation that this argument is based on an implicit assumption that we do not have any extra constants that were not included in HM. Adding suitable constants with types that involve higher-order polymorphism would make the type system described here much more powerful.

5 Constructor Classes

Type class overloading and higher-order polymorphism are independent extensions of HM. In this section, we give a number of examples to illustrate the expressiveness of a system that combines these two ideas. Previously, we have used classes to represent sets of types, i.e. constructors of kind $*$, but in this section, we will use classes to represent sets of constructors of any fixed kind κ. We will refer to these sets as *constructor classes* [10], including the type classes of Sect. 3 as a special case.

5.1 Functors

We begin our discussion of constructor classes with a now standard example. Consider the familiar `map` function that can be used to apply a function to each element in a list of values:

```
map          :: (a -> b) -> (List a -> List b)
map f []      = []
map f (x:xs) = f x : map f xs
```

It is well known that `map` satisfies the following laws:

```
map id        = id
map f . map g = map (f . g)
```

Many functional programmers will be aware that it is possible to define variants of `map`, each satisfying very similar laws, for many other datatypes. Such constructions have also been widely studied in the context of category theory where the observations here might be summarized by saying that the list type constructor `List`, together with the `map` function correspond to a *functor*. This is an obvious application for overloading because the implementation of a particular variant of `map` (if it exists) is uniquely determined by the choice of the type constructor that it involves.

Overloading map. Motivated by the discussion above, we define a constructor class, `Functor` with the following definition:

```
> class Functor f where
>     fun :: (a -> b) -> (f a -> f b)
```

Note that we have used the name `fun` to avoid a conflict with the prelude `map` function. Renaming the 'functor' laws above gives:

```
fun id        = id
fun f . fun g = fun (f . g)
```

The following datatypes will be used in later parts of these notes, and all of them can be treated as functors:

```
> data Id a     = Id a
> type List     = [ ]
> data Maybe a  = Just a | Nothing
> data Error a  = Ok a | Fail String
> data Writer a = Result String a
> type Read r   = (r ->)
```

The syntax in the final example may need a little explanation; `(r->)` is just a more attractive way of writing the partial application of constructors `((->) r)`. The whole declaration tells us that the expression `Read r` should be treated as a synonym for `(r->)`, and hence that `(a->b)`, `((a->) b)`, and `Read a b` are equivalent ways of writing the same type constructor. In this case, the **type** keyword is something of a misnomer since `(r->)`, and hence also `Read r`, has kind `(*->*)` rather than just `*`.

The functor structures for each of these datatypes are captured by the following definitions:

```
> instance Functor Id where
>     fun f (Id x)  = Id (f x)

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

> instance Functor Maybe where
>     fun f (Just x) = Just (f x)
>     fun f Nothing  = Nothing

> instance Functor Error where
>     fun f (Ok x)   = Ok (f x)
>     fun f (Fail s) = Fail s

> instance Functor Writer where
>     fun f (Result s x) = Result s (f x)

> instance Functor (r->) where
>     fun f g = f . g
```

Again, we would draw special attention to the final example. As functional programmers, we tend to think of mapping a function over the elements of a list as being a very different kind of operation to composing two functions. But, in fact, they are both instances of a single concept. This means that, in future functional languages, we could dispense with the use of two different symbols for these two concepts. We might have, for example:

```
  f . (xs ++ ys)  =   (f . xs) ++ (f . ys)
  (f . g) . xs    =   f . (g . xs)
  id . x          =   x
```

Recursion schemes: Functional programming with bananas and lenses. Functions like `map` are useful because they package up a particular pattern of computation in a convenient form as a higher-order function. Algorithms expressed in terms of `map` are often quite because they hide the underlying recursion over the structure of a list and may be more useful in program calculation where standard, but general laws for `map` can be used in place of inductive proof. The `foldr` function is another well known example of this, again from the theory of lists:

```
foldr            :: (a -> b -> b) -> b -> List a -> b
foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

As with `map`, there are variants of this function for other datatypes. For example, the fold function for the `RoseTree` datatype is:

```
> foldRT              :: (a -> [b] -> b) -> RoseTree a -> b
> foldRT f (Node a xs) = f a (map (foldRT f) xs)
```

Given that `foldr` and `foldRT` don't even have the same number of parameters, it will probably seem unlikely that we will be able to use overloading to view these two functions as instances of a single concept.

In fact, it is possible to do just this, provided that we are prepared to adopt a more uniform way of defining recursive datatypes. These ideas have already been widely studied from a categorical perspective where datatypes are constructed as fixed points of functors. The general version of a fold function is often described as a *catamorphism* and there is a dual notion of an *anamorphism*. It is common to use the notation $(\!|\phi|\!)$ for a catamorphism, and $[\![\psi]\!]$ for an anamorphism. Inspired by the shape of the brackets used here, the use of these operators has been described as 'functional programming with bananas and lenses' [17]. The remainder of this section shows how these ideas can be implemented directly using constructor classes. These ideas are dealt with in more detail elsewhere in this volume. A more detailed overview of our implementation can be found elsewhere [18].

We start by defining a datatype for constructing fixed points of unary type constructors:

```
> data Mu f  = In (f (Mu f))
```

Ideally, we would like to view the `In` constructor as an isomorphism of `f (Mu f)` and `Mu f` with the inverse isomorphism given by:

```
> out       :: Mu f -> f (Mu f)
> out (In x) = x
```

Unfortunately, the semantics of Haskell treats `In` as a non-strict constructor, so these functions are not actually isomorphisms. We will not concern ourselves any further with this rather technical point here, except to note that there have been several proposals to extend Haskell with mechanisms that would allow us to define these functions as true isomorphisms.

Now, choosing an appropriate functor as a parameter, we can use the `Mu` constructor to build recursive types:

– Natural numbers: the datatype `Nat` of natural numbers is defined as the fixed point of a functor `NatF`:

```
> type Nat    = Mu NatF
> data NatF s = Zero | Succ s

> instance Functor NatF where
>     fun f Zero        = Zero
>     fun f (Succ x) = Succ (f x)
```

For convenience, we define names for the zero natural number and for the successor function:

```
> zero   :: Nat
> zero    = In Zero

> succ   :: Nat -> Nat
> succ x  = In (Succ x)
```

For example, the number 1 is represented by `one = succ zero`.

– Lists of integers: Following the same pattern as above, we define the type `IntList` as the fixed point of a functor `IntListF`, and we introduce convenient names for the constructors:

```
> type IntList    = Mu IntListF
> data IntListF a = Nil | Cons Int a

> instance Functor IntListF where
>     fun f Nil        = Nil
>     fun f (Cons n x) = Cons n (f x)

> nil       = In Nil
> cons x xs = In (Cons x xs)
```

– Rose trees: Again, we follow a similar pattern:

```
> type RoseTree a    = Mu (RoseTreeF a)
> data RoseTreeF a b = Node a [b]

> instance Functor (RoseTreeF a) where
>     fun f (Node x ys) = Node x (map f ys)

> node     :: a -> [RoseTree a] -> RoseTree a
> node x ys = In (Node x ys)
```

The general definitions of catamorphisms and anamorphisms can be expressed directly in this framework, writing `cata phi` and `ana psi` for $(\!|\phi|\!)$ and $[\![\psi]\!]$, respectively:

```
> cata        :: Functor f => (f a -> a) -> Mu f -> a
> cata phi     = phi . fun (cata phi) . out

> ana         :: Functor f => (a -> f a) -> a -> Mu f
> ana  psi     = In . fun (ana psi) . psi
```

To illustrate the use of these recursions schemes, consider the following definitions for arithmetic on natural numbers (addition, multiplication and exponentiation):

```
> addNat n m   = cata (\fa -> case fa of
>                              Zero   -> m
>                              Succ x -> succ x) n
> mulNat n m   = cata (\fa -> case fa of
>                              Zero   -> zero
>                              Succ x -> addNat m x) n
> expNat n m   = cata (\fa -> case fa of
>                              Zero   -> one
>                              Succ x -> mulNat n x) m
```

The same recursion schemes can be used with other datatypes as shown by the following implementations of functions to calculate the length of a list of integers and to append two lists. The final example uses an anamorphism to construct an infinite list of integers:

```
> len          = cata (\fa -> case fa of
>                              Nil       -> zero
>                              Cons z zs -> succ zs)

> append xs ys = cata (\fa -> case fa of
>                              Nil       -> ys
>                              Cons z zs -> cons z zs) xs

> intsFrom     = ana (\n -> Cons n (n+1))
```

5.2 Monads

Motivated by the work of Moggi [21] and Spivey [24], Wadler [29, 28] has proposed a style of functional programming based on the use of *monads*. Wadler's main contribution was to show that monads, previously studied in depth in the context of abstract category theory [16], could be used as a practical method for structuring functional programming, and particularly for modelling 'impure' features in a purely functional setting.

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 use an expression of the form `result e` to represent a computation that returns the value `e` with no further effect, where `result` is a function:

```
result :: a -> m a
```

corresponding to the `unit` function in Wadler's presentations.

Second, to describe the way that computations can be combined, we use a function:

```
bind :: m a -> (a -> m b) -> m b
```

Writing `bind` as an infix operator, we can think of ``c `bind` f`` as a computation which runs `c`, passes the result `x` of type `a` to `f`, and runs the computation `f x` to obtain a final result of type `b`. In many cases, this corresponds to sequencing of one computation after another.

The description above leads 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 monad operators, `result` and `bind`, are required to satisfy some simple algebraic laws, that are not reflected in this class declaration. For further information, we refer the reader to the more detailed presentations of monadic programming in this volume.

One well-known application of monads is to model programs that make use of an internal state. Computations of this kind can be represented by *state transformers*, i.e. by functions of type `s -> (a,s)`, mapping an initial state to a result value paired with the final state. For the system of constructor classes in this paper, state transformers can be represented using the 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
>     fun 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 (\s -> let ST m'  = m
>                                (x,s1) = m' s
>                                ST f'  = f x
>                                (y,s2) = f' s1
>                            in  (y,s2))
```

Note that the `State` constructor has kind `* -> * -> *` so that, for any state type `s`, the constructor `State s` has kind `* -> *` as required for instances of the `Functor` and `Monad` classes. We refer the reader to other sources [28, 29, 10] for examples illustrating the use of state monads.

Many of the datatypes that we described as functors in the previous section can also be given a natural monadic structure:

- The identity monad has little practical use on its own, but provides a trivial base case for use with the monad transformers that are described in later sections.

```
> instance Monad Id where
>     result       = Id
>     Id x `bind` f = f x
```

- The list monad is useful for describing computations that may produce a sequence of zero or more results.

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

- The `Maybe` monad has been used to model programs that either produce a result (by returning a value of the form `Just e`) or raise an exception (by returning a value of the form `Nothing`).

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

- The `Error` monad is closely related to the `Maybe` datatype, but attaches a string error message to any computation that does not produce a value.

```
> instance Monad Error where
>     result           = Ok
>     Ok x     `bind` f = f x
>     Fail msg `bind` f = Fail msg
```

– The `Writer` monad is used to allow a program to produce both an output string[8] and a return value.

```
> instance Monad Writer where
>     result x             = Result "" x
>     Result s x `bind` f = Result (s ++ s') y
>                           where Result s' y = f x
```

– A `Reader` monad is used to allow a computation to access the values held in some enclosing environment (represented by the type `r` in the following definitions).

```
> instance Monad (r->) where
>     result x           = \r -> x
>     x `bind` f         = \r -> f (x r) r
```

As a passing comment, it is interesting to note that these two functions are just the standard `K` and `S` combinators of combinatory logic.

Operations on Monads. 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, for example, to permit access to the state, or to deal with input/output. It would be quite easy to run through the list of monads above and provide a small catalogue of useful operators for each one. For example, we might include an operator to update the state in a `State` monad, or to output a value in a `Writer` monad, or to signal an error condition in an `Error` monad.

In fact, we will take a more forward-thinking approach and use the constructor class mechanisms to define different families of monads, each of which supports a particular collection of simple primitives. The benefit of this is that, later, we will want to consider monads that are simultaneously instances of several different classes, and hence support a combination of different primitive features. This same approach has proved to be very flexible in other recent work [10, 15].

In these notes, we will make use of the following classes of monad:

– **State monads**: The principal characteristic of state based computations is that there is a way to access and update the state. We will represent these two features by a single `update` operator that applies a user supplied function to update the current state, returning the old state as its result.

```
> class Monad m => StateMonad m s where
>     update :: (s -> s) -> m s
```

[8] Note that, for a serious implementation of `Writer`, it would be better to use functions of type `ShowS = String -> String` as the output component of the `Writer` monad in place of the strings used here. This is a well-known trick to avoid the worst-case quadratic behaviour of nested calls to the append operator, `(++)`.

The `State s` monad described above is an obvious example of a `StateMonad`:

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

Simple uses of a state monad include maintaining an integer counter:

```
> incr    :: StateMonad m Int => m Int
> incr     = update (1+)
```

or generating a sequence of pseudo-random numbers, in this case using the algorithm suggested by Park and Miller [23]:

```
> random  :: StateMonad m Int => Int -> m Int
> random n = update min_stand `bind` \m ->
>            result (m `mod` n)

> min_stand  :: Int -> Int
> min_stand n = if test > 0 then test else test + 2147483647
>               where test = 16807 * lo - 2836 * hi
>                     hi   = n `div` 127773
>                     lo   = n `mod` 127773
```

– **Error monads**: The main feature of this class of monads is the ability for a computation to fail, producing an error message as a diagnostic.

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

The `Error` datatype used above is a simple example of an `ErrorMonad`:

```
> instance ErrorMonad Error where
>     fail  = Fail
```

– **Writer monads**: The most important feature of a writer monad is the ability to output messages.

```
> class Monad m => WriterMonad m where
>     write :: String -> m ()

> instance WriterMonad Writer where
>     write msg = Result msg ()
```

– **Reader monads**: A class of monads for describing computations that consult some fixed environment:

```
> class Monad m => ReaderMonad m r where
>     env    :: r -> m a -> m a
>     getenv :: m r
```

```
> instance ReaderMonad (r->) r where
>     env e c = \_ -> c e
>     getenv  = id
```

To illustrate why this approach is so attractive, consider the following definition:

```
> nxt m = update (m+)   `bind` \n ->
>         if n > 0 then write ("count = " ++ show n)
>                  else fail "count must be positive"
```

The `nxt` function uses a combination of features: state, error and output. This is reflected in the inferred type:

```
  (WriterMonad m, ErrorMonad m, StateMonad m Int) => Int -> m ()
```

In this example, the type inference mechanism records the combination of features that are required for a particular computation, without committing to a particular monad `m` that happens to meet these constraints[9]. This last point is important for two reasons. First, because we may want to use `nxt` in a context where some additional features are required, resulting in an extra constraint on `m`. Second, because there may be several ways to combine a particular combination of features with corresponding variations in semantics. Clearly, it is preferable to retain control over this, rather than leaving the type system to make an arbitrary choice on our behalf.

Monads as substitutions. 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 -> 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                   -- Type variable
>             | TInt                     -- Integer type
>             | Fun (Type v) (Type v)    -- Function type
```

For convenience, we define an instance of the `Text` class to describe how such type expressions will be displayed:

```
 instance Text v => Text (Type v) where
>    showsPrec p (TVar v)  = shows v
>    showsPrec p TInt      = showString "Int"
>    showsPrec p (Fun l r) = showParen (p>0) str
>     where str = showsPrec 1 l . showString " -> " . shows r
```

[9] In fact, none of the monad examples that we have seen so far are instances of all of these classes. The process of constructing new monads which do satisfy all of the class constraints listed here will be described later in these notes.

The functor and monad structure of the `Type` constructor are as follows:

```
> instance Functor Type where
>     fun f (TVar v)  = TVar (f v)
>     fun f TInt      = TInt
>     fun f (Fun d r) = Fun (fun f d) (fun f r)

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

In this setting, the `fun` function gives a systematic renaming of the variables in a term (there are no bound variables), while `result` corresponds to the null substitution that maps each variable to the term for that variable. If `t` has type `Type v` and `s` is a substitution of type `v -> 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:

```
> apply    :: Monad m => (a -> m b) -> (m a -> m b)
> apply s t = t `bind` s
```

Note that this operator can be used with any monad, not just the `Type` constructor that we are discussing here. Composition of substitutions also corresponds to a more general operator, called *Kleisli composition*, that can be used with arbitrary monads. Written here as the infix operator `(@@)`, Kleisli composition can be defined as:

```
> (@@)     :: Monad m => (a -> m b) -> (c -> m a) -> (c -> m b)
> f @@ g    = join . fun f . g

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

Apart from its use in the definition of `(@@)`, the `join` operator defined here can also be used an alternative to `bind` for combining computations [28].

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.

```
> type Subst m v = v -> m v
```

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

```
> (>>)        :: (Eq v, Monad m) => v -> m v -> Subst m v
> (v >> t) w  = if v==w then t else result w
```

The type signature shown here is the most general type of the `(>>)` operator, and could also have been inferred automatically by the type system. The class constraints `(Eq v, Monad m)` indicate that, while `(>>)` is defined for arbitrary monads, 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`:

```
> unify TInt      TInt      = result result
> unify (TVar v)  (TVar w)  = result (if v==w then result
>                                              else v >> TVar w)
> unify (TVar v)  t         = varBind v t
> unify t         (TVar v)  = varBind v t
> unify (Fun d r) (Fun e s) = unify d e         `bind` \s1 ->
>                             unify (apply s1 r)
>                                   (apply s1 s) `bind` \s2 ->
>                             result (s2 @@ s1)
>  unify t1        t2        = fail ("Cannot unify " ++ show t1 ++
>                                   " with " ++ show t2)
```

The only way that unification can fail is if we try to bind a variable to a type 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     = if (v `elem` vars t)
>                     then fail "Occurs check fails"
>                     else result (v>>t)
>                   where vars (TVar v)   = [v]
>                         vars TInt       = []
>                         vars (Fun d r)  = vars d ++ vars r
```

A Simple Application: A Type Inference Algorithm. To illustrate how some of the classes and functions introduced above might be used in practice, we will describe a simple monadic implementation of Milner's type inference algorithm $\mathcal{W}$. 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 [19, 3].

The purpose of the type checker is to determine types for the terms of a simple λ-calculus represented by the `Term` datatype introduced in Section 3.3:

```
> type Name = String
> data Term = Var Name       -- variable
>           | Ap  Term Term  -- application
>           | Lam Name Term  -- lambda abstraction
>           | Num Int        -- numeric literal
```

We will also use the representation of types described above 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 term `Lam x (Var x)`, we expect the algorithm to produce a result of the form `Fun n n :: Type Int` for some (arbitrary) type variable `n = TVar m`.

At each stage, the type inference algorithm maintains a collection of assumptions about the types currently assigned to free variables. This can be described by an environment mapping variable names to types and represented using association lists:

```
> data Env t = Ass [(Name,t)]

> emptyEnv           :: Env t
> emptyEnv            = Ass []

> extend             :: Name -> t -> Env t -> Env t
> extend v t (Ass as) = Ass ((v,t):as)

> lookup             :: ErrorMonad m => Name -> Env t -> m t
> lookup v (Ass as)   = foldr find err as
>  where find (w,t) alt = if w==v then result t else alt
>        err            = fail ("Unbound variable: " ++ v)

> instance Functor Env where
>     fun f (Ass as) = Ass [ (n, f t) | (n,t) <- as ]
```

As the names suggest, `emptyEnv` 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 `e` 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 `e` 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)
>   = incr                                `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 (fun (apply s) a) r         `bind` \(t,rt) ->
>     incr                              `bind` \b      ->
>     unify (apply t lt) (rt `Fun` TVar b) `bind` \u   ->
>     result (u @@ t @@ s, u b)

> infer a (Num n)
>   = result (result, TInt)
```

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 `incr`.

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:

```
infer :: (ErrorMonad m, StateMonad m Int) =>
           Env (Type Int) ->
             Term ->
               m (Subst Type Int, Type Int)
```

Our problem now is that to make any use of `infer`, we need to construct a monad `m` that satisfies these constraints. It is possible to deal with such problems on a case-by-case basis, but it is obviously more attractive to use more general tools if possible. This is the problem that we turn our attention to in the following sections.

Combining Monads. While we can give some nice examples to illustrate the use of one particular set of features, for example, the use of state in a state monad, real programs typically require a combination of several different features. It is therefore quite important to develop systematic techniques for combining groups of features in a single monad.

In recent years, there have been several investigations into techniques for combining monads in functional programming languages[10]. Examples of this include the work of King and Wadler [14], and of Jones and Duponcheel [13]

[10] In fact, much of this work is a rediscovery of ideas that have already been developed by category theorists, albeit in a more abstract manner that is perhaps less accessible to some computer scientists.

to investigate the conditions under which a pair of monads `m` and `n` can be composed. In the following definitions, we adapt the `swap` construction of Jones and Duponcheel to the framework used in these notes. For reasons of space, we do not give any formal proof or motivation for these techniques here. We urge the reader not to be too distracted by the formal definitions shown below, focusing instead on the main objective which is to construct composite monads.

To begin with, it is useful to define two different forms of composition; forwards (`FComp`) and backwards (`BComp`):

```
> data FComp m n a = FC (n (m a))
> data BComp m n a = BC (m (n a))

> unBC (BC x) = x
> unFC (FC x) = x
```

It may seem strange to provide both forms of composition here since any value of type `FComp m n a` corresponds in an obvious way to a value of type `BComp n m a`, and vice versa. However, it is useful to have both forms of composition when we consider partial applications; the constructors `FComp m` and `BComp m` are not equivalent.

The functor structure for the two forms of composition are straightforward:

```
> instance (Functor m, Functor n) => Functor (FComp m n) where
>     fun f (FC c) = FC (fun (fun f) c)

> instance (Functor m, Functor n) => Functor (BComp m n) where
>     fun f (BC c) = BC (fun (fun f) c)
```

These two definitions rely on the overloading mechanisms to determine which version of the `fun` operator is used for a particular occurrence.

Two monads `m` and `n` can be 'composed' if there is a function:

```
  swap :: m (n a) -> n (m a)
```

satisfying certain laws set described by Jones and Duponcheel [13]. Fixing the monad `m` and using `n` to represent an arbitrary monad, it follows that the forward composition `FComp m n` is a monad if `m` is an instance of the class:

```
> class Monad m => Into m where
>     into :: Monad n => m (n a) -> n (m a)
```

and the `into` function satisfies the laws for `swap`. We refer to this operator as `into` because it allows us to push the monad `m` into an arbitrary computation represented by a monad `n`. Given this function, the structure of the composite monad is given by:

```
> instance (Into m, Monad n) => Monad (FComp m n) where
>     result x      = FC (result (result x))
>     FC c `bind` f = FC ((fun join . join . fun f') c)
>                      where f' = into . fun (unFC . f)
```

For example, any forward composition of one of either the `Maybe`, `Error` or `Writer` monads with another arbitrary monad can be obtained using the following instances of `Into`:

```
> instance Into Maybe where
>     into Nothing  = result Nothing
>     into (Just c) = fun Just c

> instance Into Error where
>     into (Fail msg) = result (Fail msg)
>     into (Ok c)     = fun Ok c

> instance Into Writer where
>     into (Result s c) = c `bind` \x -> result (Result s x)
```

In a similar way, for any fixed monad `m` and an arbitrary monad `n`, the backward composition `BComp m n` is a monad if `m` is an instance of the class:

```
> class Monad m => OutOf m where
>    outof :: Monad n => n (m a) -> m (n a)
```

and the `outof` operator satisfies the laws for `swap`. In this case, the monad structure can be described by the definition:

```
> instance (OutOf m, Monad n) => Monad (BComp m n) where
>     result x      = BC (result (result x))
>     BC c `bind` f = BC ((fun join . join . fun f') c)
>                     where f' = outof . fun (unBC . f)
```

For example, any backward composition of a reader monad and another arbitrary monad, yields a monad:

```
> instance OutOf (r ->) where
>    outof c = \r -> c `bind` \f -> result (f r)
```

Monad Transformers. Notice that, rather than allowing us to combine two arbitrary monads, all of the examples above use one fixed monad to transform another arbitrary monad. In other words, the following constructors can be understood as *monad transformers*, each having kind `(* -> *) -> (* -> *)` and mapping a monad to a new transformed monad that includes some extra features:

```
> type MaybeT    = FComp Maybe
> type ErrorT    = FComp Error
> type WriterT   = FComp Writer
> type ReaderT r = BComp (r ->)
```

The possibility of using monad transformers had previously been suggested by Moggi [22], leading independently to the use of *pseudomonads* in Steele's work on the construction of modular interpreters [25], and to a Scheme implementation by Espinosa [4, 5]. The problem of implementing monad transformers in a strongly typed language has been addressed by Liang, Hudak and Jones [15] using constructor classes.

We can define a class of monad transformers using the definition:

```
> class MonadT t where
>     lift :: Monad m => m a -> t m a
```

The intention here is that `lift` embeds a computation in the `m` monad into the extended monad `t m`, without using any of the extra features that it supports. Partial applications of both forward and backward compositions give rise to monad transformers, including the four examples above as special cases:

```
> instance Into m => MonadT (FComp m) where
>     lift = FC . fun result

> instance OutOf m => MonadT (BComp m) where
>     lift = BC . result
```

There are also examples of monad transformers that are not easily expressed as compositions. A standard example of this is the following definition of a state monad transformer:

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

> instance Monad m => Functor (StateT s m) where
>     fun f (STM xs) = STM (\s -> xs s `bind` \(x,s') ->
>                                 result (f x, s'))

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

> instance MonadT (StateT s) where
>     lift c = STM (\s -> c `bind` \x -> result (x,s))
```

In fact, this defines a family of monad transformers, each of which takes the form `StateT s` for some state type `s`.

Previously, we have defined classes of monads for describing computations involving state, errors, writers and readers, but we have only defined one instance of each with no overlap between the different classes. Using monad transformers, we can extend these classes with new instances, and construct monads that are simultaneously instances of several different classes. We will illustrate this with two examples, leaving the task of extending some of the other classes introduced above to include transformed monads as an exercise for the reader.

– The state monad transformer: The following instance declaration indicates that we can apply `StateT s` to an arbitrary monad `m` to obtain a monad with a state component of type `s`:

```
> instance Monad m => StateMonad (StateT s m) s where
>     update f = STM (\s -> result (s, f s))
```

On the other hand, if `m` is a monad with a state component of type `s`, then so is the result of applying an arbitrary transformer to `m`:

```
> instance (MonadT t, StateMonad m s) => StateMonad (t m) s where
>     update f = lift (update f)
```

These two instance definitions overlap; a monad of the form `StateT s m` matches both of the monad constructors to the right of the `=>` symbol. In Gofer, these conflicts are dealt with by choosing the most specific instance that matches the given constructor.

– The Error monad transformer: Following a similar pattern to the declarations above, the following definitions tell us that applying `ErrorT` to any monad or an arbitrary transformer to an `ErrorMonad` will produce an `ErrorMonad`:

```
> instance Monad m => ErrorMonad (ErrorT m) where
>     fail msg = FC (result (fail msg))

> instance (MonadT t, ErrorMonad m) => ErrorMonad (t m) where
>     fail msg = lift (fail msg)
```

Now, at last, we have the tools that we need to combine monads to satisfy particular sets of constraints. For example, for the type inference algorithm described above, we need to find a monad `m` satisfying the constraints:

```
(ErrorMonad m, StateMonad m Int)
```

We have at least two different ways to construct a suitable monad:

– `ErrorT (State Int)`: in this case, `m` is equivalent to the monad:

```
ES a = Int -> (Error a, Int)
```

With this combination of state and error handling it is possible to return a modified state, even if an error occurs.

– `StateT Int Error`, in this case, `m` is equivalent to the monad:

```
SE a = Int -> Error (a,Int)
```

With this combination of state and error handling the final state will only be returned if the computation does not produce an error.

This example shows how monad transformers can be used to combine several different features in a single monad, with the flexibility to choose an appropriate semantics for a particular application.

References

1. R. Bird and P. Wadler. *Introduction to functional programming.* Prentice Hall, 1988.
2. R.M. Burstall, D.B MacQueen, and D.T. Sanella. Hope: An experimental applicative language. In *The 1980 LISP Conference*, pages 136–143, Stanford, August 1980.
3. L. Damas and R. Milner. Principal type schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming languages*, pages 207–212, Albuquerque, N.M., January 1982.
4. David Espinosa. Modular denotational semantics. Unpublished manuscript, December 1993.
5. David Espinosa. Building interpreters by transforming stratified monads. Unpublished manuscript, June 1994.
6. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
7. P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.
8. Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms, an exercise in the arithmetic of folds and zips. Programming Research Group, Oxford, December 1992.
9. Mark P. Jones. Computing with lattices: An application of type classes. *Journal of Functional Programming*, 2(4), October 1992.
10. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, New York, June 1993. ACM Press.
11. Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida*, June 1994. To appear.
12. Mark P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994.
13. M.P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, New Haven, Connecticut, USA, December 1993.
14. D.J. King and P. Wadler. Combining monads. In *Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming*, Ayr, Scotland, 1992. Springer Verlag Workshops in Computer Science.
15. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995.
16. S. MacLane. *Categories for the working mathematician.* Graduate texts in mathematics, 5. Springer-Verlag, 1971.
17. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th ACM conference on Functional Programming Languages and Computer Architecture*, pages 124–144, New York, 1991. Springer-Verlag. Lecture Notes in Computer Science, 523.

18. Erik Meijer and Mark P. Jones. Gofer goes bananas. In preparation, 1994.
19. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.
20. Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.
21. E. Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, Asilomar, California, 1989.
22. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.
23. Stephen K Park and Keith W Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, Oct 1988.
24. M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1), 1990.
25. Guy L. Steele Jr. Building interpreters by composing monads. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, Portland, OR, January 1994.
26. D.A. Turner. Duality and De Morgan principles for lists. In W. Feijen, N. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business, A Birthday Salute to Edsger W. Dijkstra*, pages 390–398. Springer-Verlag, 1990.
27. D.A. Turner. An overview of Miranda. In David Turner, editor, *Research Topics in Functional Programming*, chapter 1, pages 1–16. Addison Wesley, 1990.
28. P. Wadler. Comprehending monads. In *ACM conference on LISP and Functional Programming*, Nice, France, 1990.
29. P. Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–14, Jan 1992.
30. P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Jan 1989.

This article was processed using the LaTeX macro package with LLNCS style