A Taste of Functional Programming – 2

Madhavan Mukund

Madhavan Mukund has been on the Computer Science faculty at Chennai Mathematical Institute since 1992, where he is currently Professor. His research interests are primarily in formal methods for specifying and verifying computing systems. He is the Secretary of the Indian Association for Research in Computing Science (IARCS) and the National Coordinator of the Indian Computing Olympiad.

Functional programming has its roots in Alonzo Church's lambda calculus. In the first part[1] of this article, we explored some basic notions of functional programming using the language Haskell. We now examine some more advanced concepts, including polymorphism, infinite data types and computations, and user-defined data types.

1. Recap

Haskell is a typed functional programming language. Its built-in scalar types include `Int`, `Float`, `Char` and `Bool` for integers, floating point numbers, characters and boolean values, respectively. A function from type `A` to type `B` has type `A -> B`. Thus, the function `factorial` would have type `Int -> Int`. Multiple inputs are read one a time, using *currying*. For instance, a function `plus` that adds two integers would have type `Int -> Int -> Int` rather than `Int` $\times$ `Int -> Int`.

The basic collective type is a *list* – a sequence of values with a uniform underlying type. For instance, `[3,1,2,1]` is a list of `Int`. A list with underlying type `T` has type `[T]`, so the preceding example has type `[Int]`. Every list is built up from the empty list `[]` by repeated applications of the operator `:` that appends an element to the left of a list. Thus, the list `[3,1,2,1]` is internally represented as `3:(1:(2:(1:[])))`, or more simply as `3:1:2:1:[]`, since `:` associates to the right.

A Haskell program consists of a number of functions. Each function is specified using one or more definitions that are scanned from top to bottom. Pattern matching can be used to simplify definition by cases. For instance,

[1] A taste of functional programming – 1, *Resonance*, Vol.12, No.8, pp.27–48, 2007.

Keywords
Functional programming, polymorphism, currying, reductions.

here is an inductive definition of the `fibonacci` function, where the first line specifies the type of the function.

```
fibonacci :: Int -> Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

For functions on lists, pattern matching can be used to implicitly decompose a list into its first element, the *head*, and the rest of the list, the *tail*. Here is a function that inductively computes the length of a list of `[Int]`.

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

2. Polymorphism

Observe that the function `length` works in the same way for lists of any type. It would be wasteful to have to write a separate version of such a function for each different type of list. In Haskell, it is possible to say that a function works for multiple types by using type variables. For instance, we can write the type of `length` as `length::[a]->Int`. Here, the letter `a` in the type `[a]->Int` is a type variable. The type `[a]->Int` is to be read as "*for any underlying type* `a`, *this function is of type* `[a]->Int`".

In Haskell, it is possible to say that a function works for multiple types by using type variables. For instance, we can write the type of length as length::[a]->Int. Here, the letter a in the type [a]->Int is a type variable.

The function `reverse` reverses the order of elements in a list and the function `concat` "dissolves" one level of brackets in a nested list. Using type variables, we can specify the most general types of these functions as follows.

```
reverse :: [a] -> [a]
concat  :: [[a]] -> [a]
```

In these type definitions, it is significant that the same letter `a` appears on both sides of the `->`. This means, for

In a nutshell, overloading uses the same symbol to denote similar operations on different types, but the way the operation is implemented for each type is different. On the other hand, polymorphism refers to a single function definition with a fixed computation rule that works for multiple types in the same way.

instance, that the type of the list returned by `reverse` is the same as the type of the input list. In other words, all occurrences of a type variable in a type declaration must be instantiated to the same actual type.

Functions that work in the same way on different types are called *polymorphic*, which in Greek means 'taking different forms'. We must be careful to distinguish polymorphism of the type we have seen with lists from the ad hoc variety associated with overloading operators. For instance, in most programming languages, we write `+` to denote addition for both integers and floating point numbers. However, since the underlying representations used for the two kinds of numbers are completely different, we are actually using the same name (`+`, in this case) to designate functions that are computed in a different manner for different base types. This type of situation is more properly referred to as *overloading*.

In a nutshell, overloading uses the same symbol to denote similar operations on different types, but the way the operation is implemented for each type is different. On the other hand, polymorphism refers to a single function definition with a fixed computation rule that works for multiple types in the same way.

2.1 *Map and Filter*

We have seen the built-in function `map` that allows us to apply a function `f` "pointwise" to each element of a list, so that `map f [x0,x1,...,xk]` $\rightsquigarrow$ `[(f x0),(f x1), ...,(f xk)]`.

What is the type of `map`? The function `f` is in general of type `a->b`. The list that `map` operates on must be compatible with the function `f`, so it must be of type `[a]`. The list generated by `map` is of type `[b]`. Thus, we have `map::(a->b)->[a]->[b]`. Notice that there are two type variables, `a` and `b`, in the type definition for `map`. These can be independently instantiated to

different types, and these instantiations apply uniformly to all occurrences of `a` and `b`.

We have also seen that we can *filter* out values from a list. Let `l` be of type `[a]` and let `p` be a function from `a` to `Bool`. Then, we can define `filter` as follows.

```
filter : (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | (p x)     = x:(filter p xs)
                | otherwise = filter p xs
```

Here, the output of `filter` is a sublist of the original list, so the output list has the same type as the original list.

2.2 *Conditional Polymorphism*

Recall the function `sum` that adds up the elements of a list.

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

Clearly, this definition of `sum` would work for both lists of `Int` and lists of `Float`, so it would be appropriate to assign it a polymorphic type. However, it would be wrong to write `sum::[a]->a` because `sum` works only for lists whose underlying type supports addition. We need to assign `sum` a qualified polymorphic type of the form

```
sum :: [a] -> a, provided type a supports +
```

Analogously, consider the function `quicksort` to sort a list. If we use concrete types, we have to define separate versions of `quicksort` for each type of list, such as `iquicksort::[Int]->[Int]`, `fquicksort::[Float]->[Float]` and `ilistquicksort::[[Int]]->[[Int]]`. This is clearly undesirable because the actual definition of `quicksort` in all these cases is the same. The other option is to declare `quicksort` to be of type `[a]->[a]`.

Is this reasonable? Can we sort a list of functions of type `[Int->Int->Int]` such as `[plus, times, max]`?

The answer is that we can sort a list of values provided we can compare these values with each other. In other words, we need to assign the following type to `quicksort`:

```
quicksort :: [a] -> [a],
       provided we can compare values of type a
```

Haskell classifies types into subsets, called *type classes*, based on the additional properties satisfied by the types in a class. Each type class carries an obligation in the form of some underlying functions that a type needs to support to belong to the class[2]. For instance, the set of all types that support comparison functions `<`, `<=`,... is called `Ord`.

[2] This is analogous to a class implementing an interface in an object oriented language.

A subset X of a set Y can also be described in terms of its characteristic function f_X where $f_X(x) =$ True if and only if $x \in X$. Thus, we can think of `Ord` as a function that maps types to `Bool` and write `Ord T` to denote whether or not type `T` belongs to `Ord`. The type of `quicksort` now becomes:

```
quicksort :: (Ord a) => [a] -> [a]
```

Haskell classifies types into subsets, called *type classes*, based on the additional properties satisfied by the types in a class. Each type class carries an obligation in the form of some underlying functions that a type needs to support to belong to the class.

This is read as *"If* `a` *is in* `Ord`*, then* `quicksort` *is of type* `[a]->[a]`*"*. Note the double arrow notation to denote *"if ... then ..."*.

An even more basic type class is `Eq`, the set of all types that support checking for equality. Why is this a nontrivial type class? Recall that we have function types of the form `T1 -> T2`. One of the most fundamental results in the theory of computation is that it is *not* possible to effectively check whether two computable functions are equal for any nontrivial definition of equality (for instance, `f == g` if and only if for each input `x`, `f x == g x`). Thus, functions do not belong to `Eq`. A typical

example of a function that depends on `Eq` is the built-in function `elem` that checks if a value belongs to a list. Here is an inductive definition of `elem`.

```
elem x [] = False
elem x (y:ys) | x == y     = True
              | otherwise  = elem x ys
```

The most general type for `elem` is `elem::(Eq a)=>a->[a]->Bool`. Observe that a type can belong to `Ord` only if it belongs to `Eq`. This is because comparison involves not only the functions `<` and `>` but also `<=`, `>=`, ... which imply that we can check equality. Thus, as subsets of types, `Ord` is a subset of `Eq`. Alternatively, we have that `Ord a` implies `Eq a` for any type `a`.

Another typical type class in Haskell is `Num`, the collection of all types that supports "numeric" operations such as `+`, `-` and `*`. We observed earlier that the function `sum` that adds up the values in a list will work on any list whose underlying type supports addition. This means that we can assign the generic type `sum::(Num a)=>[a]->a`.

3. Type Inference

Haskell allows us to pass any type to a function, including another function. Consider the function `apply`, that takes as input a function `f` and a value `x` and returns the value `(f x)`. In other words, this function applies `f` to `x`. The definition of `apply` is very straightforward: `apply f x = f x`.

Haskell allows us to pass any type to a function, including another function. Consider the function apply, that takes as input a function f and a value x and returns the value (f x). In other words, this function applies f to x. The definition of apply is very straightforward: apply f x = f x.

What is the type of `apply`? The first argument is any function, so we can denote its type as `a -> b` for some arbitrary types `a` and `b`. The second argument x has to be fed as an input to `f`, so its type must be `a`. The output of apply is `f x`, which has type `b`. Thus, we have, `apply::(a->b)->a->b`.

What if we change the function to apply `f` twice to `x`,

as in the function `twice f x = f (f x)`? In this case, we see that the output `(f x)` is fed back as an input to `f`. This means that the input and output types `a` and `b` must be the same, so `f::a->a` and the type of `twice` is given by `twice::(a->a)->a->a`.

The analysis we did by hand when trying to deduce the type of `apply` and `twice` is built into Haskell. Thus, if we do not provide an explicit type for a function, Haskell will start with the most general assumption about the type and impose the constraints inferred from the function definitions to arrive at a final type.

4. Outermost Reduction and Infinite Data Structures

Computation in Haskell consists of rewriting expressions using function definitions. An expression may have multiple subexpressions that can be simplified. For instance, if `sqr x = x*x`, the expression `sqr (3+4)` may be simplified as either `sqr 7` or `(3+4)*(3+4)` depending on whether we use the definition of `sqr` or `+` first. In general, if the subexpressions that can be reduced overlap, as in this case, we can always unambiguously declare one to be *inside* the other – here `(3+4)` is inside `sqr (3+4)`.

Computation in Haskell consists of rewriting expressions using function definitions. An expression may have multiple subexpressions that can be simplified. For instance, if sqr x = x*x, the expression sqr (3+4) may be simplified as either sqr 7 or (3+4)*(3+4) depending on whether we use the definition of sqr or + first.

In Haskell, the outer expression is always simplified before the inner one. This *outermost* strategy is sometimes called *lazy* – it does not simplify the argument to a function until the value of the argument is actually needed in the evaluation of the function. On the other hand, *innermost* or *eager* reduction would always evaluate an argument before it is used. Outermost reduction sometimes produces a result when innermost reduction does not. For instance, given the definition `power x 0 = 1.0`, we have `power (8.0/0.0) 0` $\leadsto$ `1.0`, because the argument `x` does not need to be evaluated.

Outermost reduction also permits the definition of infi-

nite data structures. For instance, the list of all integers starting at `n` is given by the function

```
listfrom n = n: (listfrom (n+1))
```

After we rewrite `listfrom n`, the outermost expression is the one involving `:`. This is thus evaluated first, resulting in the initial `n` being generated. Haskell then tries to expand `listfrom (n+1)` which, in turn, generates `n+1` and `listfrom (n+2)` and so on. Thus, the output of `listfrom m` is the infinite list `[m, m+1,...]` which is denoted `[m..]` in Haskell.

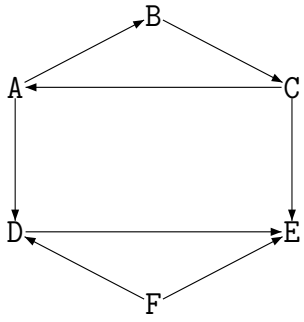
Why are infinite lists useful? It is often conceptually easier to define a function that returns an infinite list and extract a finite prefix to get a concrete value.

4.1 *Using Infinite Lists*

Why are infinite lists useful? It is often conceptually easier to define a function that returns an infinite list and extract a finite prefix to get a concrete value. We illustrate this by writing a function to check connectivity in a directed graph.

A directed graph is a collection of *vertices* with some pairs connected by oriented *edges*. *Figure* 1 is an example of a directed graph with six vertices. We can describe the graph using a function `edge`, as shown to the right of the graph.

Our goal is to construct a function `connected::Char->Char->Bool` to compute the pairs of vertices that are *connected* – that is, `connected x y` is `True` if and only




```
edge :: Char -> Char -> Bool
edge 'A' 'B' = True
edge 'A' 'D' = True
edge 'B' 'C' = True
edge 'C' 'A' = True
edge 'C' 'E' = True
edge 'D' 'E' = True
edge 'F' 'D' = True
edge 'F' 'E' = True
edge  x   y  = False
```

Figure 1. A graph in Haskell.

if there is a path from `x` to `y` using the given set of edges. We build up the set of paths inductively. Initially we have paths of length 0 – there is only one. Given a path of length k, we extend it to a path of length $k+1$ by adding an edge. We can represent a path as a list of nodes, so we have

```
extendpath :: [Char] -> [[Char]]
extendpath p = [p++c | c <- ['A'..'F'],
                                edge (last p) c]
```

where `last p` returns the last value in the list `p`. We can then map `extendpath` over the list of paths of length k to get the list of paths of length $k+1$.

```
extendall :: [[Char]] -> [[Char]]
extendall [] = [[c] | c <- ['A'..'F']]
extendall l  = [ll | p <- l, ll <- extend p]
```

The base case of `extendall` constructs paths that consist of a single node. If we start with the singleton list of empty paths `[[]]` and repeatedly apply `extendall`, we get lists with longer and longer paths.

To check if `x` and `y` are connected, we only need to check for paths without loops from `x` to `y` – that is, we can assume that the path from `x` to `y` does not visit an intermediate node `z` twice. If it did, we can excise the loop from `z` to `z` and get a shorter path that serves our purpose. If we have n nodes overall, a loop free path can have at most $n-1$ edges. This suggests that to check all pairs of connected nodes, it is sufficient to apply `extendall` n times to the initial list containing the empty path.

Haskell has a built-in function `iterate` such that `iterate f x` $\leadsto$ `[x,f x,f(f x),...]`. We can therefore write `iterate extendall [[]]` to generate the list we want. We can then extract the first n elements of this list (all paths of length upto $n-1$) as

```
firstn = take n (iterate extendall [[]])
```

Now, for each path in this list, we extract the start and end points as follows.

```
connectedpairs  = [(head p, last p) | l <- firstn,
                                      p <- l]
```

Finally, we can define the function `connected`.

```
connected x y = (elem (x,y) connectedpairs)
```

Notice that we have not bothered about the fact that `extendall` generates paths that loop and do other unproductive things. For instance, the path `['A','B','C', 'A','B','C']` belongs to the sixth iteration of `extendall [[]]`, but it does not matter. All that we want is a guarantee that every pair `(x,y)` that is connected is enumerated by the nth step.

4.2 *Search Problems*

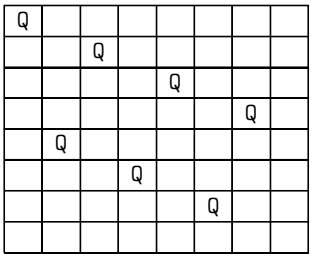
In search problems, there is no closed form for the answer and we need to systematically generate all possible solutions, undoing partial solutions whenever we reach a dead end. A classical problem of this sort is that of placing n queens on an $n \times n$ chessboard such that no two queens attack each other. Recall that two queens attack each other if they lie on the same row, column or diagonal.

From the problem description, it is immediate that in any solution to the problem, there is exactly one queen on each row (and also on each column). Thus, one strategy for solving the problem is the following:

- Place the first queen on some square of the first row.
- In each succeeding row, place a queen at the leftmost square that is not attacked by any of the earlier queens.

If we follow this strategy on an 8×8 board and place the first queen at the top left corner, after 7 moves, we arrive at the configuration shown in *Figure* 2. We find

Figure 2. The 8 queens problem.



that there is no valid position on the last row for the 8th queen, so we have to abandon this solution and try another one. This can be done in a systematic way by retrying the next possibility for the 7th queen and once again trying the 8th queen. If all possibilities for the 7th queen fail, we go back and try the next possibility for the 6th queen. This strategy is called *backtracking*.

We can represent an arrangement of queens as a list of integers, where the first integer is the column number of the first queen in the first row, the second integer is the column number of the second queen in the second row, ... Thus, the position in *Figure* 2 is described by the list `[1,3,5,7,2,4,6]`.

Given an arrangement of k queens, we can write a function that computes all valid extensions of this arrangement to $k+1$ queens, analogous to the function we wrote to extend paths of length k in a graph to paths of length $k+1$. We have to ensure that the new queen is not in the same column as any previously placed queen. We also calculate, using elementary arithmetic, that the new position is not on any diagonal that is attacked by any of the previous positions.

Given an arrangement of *k* queens, we can write a function that computes all valid extensions of this arrangement to *k*+1 queens, analogous to the function we wrote to extend paths of length *k* in a graph to paths of length *k*+1. We have to ensure that the new queen is not in the same column as any previously placed queen.

As in the paths example, let us give the name `extendall` to the function that computes all valid extensions of a list of arrangements. We can now solve the n queens problem by repeatedly applying the function `extendall` to the empty arrangement and picking up the values generated after the nth application. The following function computes all possible arrangements of n queens on an $n \times n$ board.

```
queens n = (iterate extendall [[]])!!(n+1)
```

where `l!!i` denotes the element at position `i` in list `l`, starting from position 0. The following returns just one such arrangement – the first one that is generated.

```
queensone n = head ((iterate extendall [])!!(n+1))
```

Notice that some of the positions after k iterations may have no valid extensions (like the arrangment of 7 queens above). This does not matter. If, at some stage, all arrangements die out as infeasible, we will get the value `[]` consisting of no valid arrangements (as opposed to `[[]]`, the list consisting of the empty arrangement) which will just repeat itself indefinitely.

It might appear that we need to generate all possible arrangements with 1,2,...,7 queens before we can obtain the first valid arrangement for 8 queens in `queensone`. However, Haskell's outermost evaluation strategy will actually expand the leftmost solution at each level, so it will compute the first solution with 8 queens in a depth-first, rather than a breadth-first, manner.

5. User-Defined Data Types

A data type is a collection of values with a collective name. For instance, the data type `Int` consists of the values {...,-2,-1,0,1,2,...}, while the data type `Bool` consists of the values {`False`,`True`}. Data types can be polymorphic and even recursively defined and hence of unbounded size – for example, lists. In Haskell, we can extend the set of built-in data types using the `data` statement.

5.1 *Enumerated Data Types*

This simplest form of data type is one consisting of a finite set of values. We can define such a type using the `data` statement, as follows.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri
                                    | Sat
```

Having introduced this new type, we can directly use it in functions such as:

```
weekend :: Day -> Bool
weekend Sun = True
weekend Sat = True
```

```
weekend d   = False
```

We can also write a function `nextday`.

```
nextday :: Day -> Day
nextday Sun = Mon
...
nextday Sat = Sun
```

What happens if we ask Haskell to evaluate `nextday Sat`? The answer is computed correctly as `Sun` but we get a message saying `Cannot find "show" function for Day`. Similarly, if we ask whether `Tue == Wed`, the response is `Cannot infer instance Eq Day`.

The problem is that we have not associated the new data type with any type classes, including the most basic ones such as `Eq` and `Show`. The class `Show` consists of those types whose values can be displayed – that is, the value can be converted to a `String` and printed on screen. The most natural definitions for `Eq` and `Show` are that each value is distinct and equal only to itself and each value is displayed in the same way it is defined. We can include these "default" definitions for `Eq` and `Show` using the word `deriving` as follows:

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

In the same way, we can derive a definition for `Ord` – the default definition would order the values in the sequence they are presented, namely `Sun` $<$ `Mon` $<$ ... $<$ `Sat`.

5.2 *Data Types with Parameters*

We can go beyond finite enumerated types and describe data types with a parameter, as in the following example.

```
data Shape = Square Float | Circle Float | Rectangle Float Float
  deriving (Eq, Ord, Show)

size :: Shape -> Float
size (Square x)      = x
size (Circle r)      = r
size (Rectangle l w) = l+w
```

Each variant of `Shape` has a *constructor* – `Square`, `Circle` or `Rectangle`. Each constructor is attached to a group of values, which can vary from constructor to constructor. The values `Sun`, `Mon`, ... in the type `Day` are also constructors with zero values attached.

What happens when we derive `Eq` for `Shape`? At the level of `Shape`, this will ensure that `(Square x)` is equal to `(Square y)` provided `x == y` (thus, for the inner value, it *derives* equality from `Float`) but `(Square x)` is never equal to `(Circle y)`. Since we also derive `Ord`, we have `Square` < `Circle` < `Rectangle` so `(Square x)` < `(Circle y)` for all `x` and `y` and `(Circle z)` < `(Circle w)` if `z` < `w`.

5.3 *Polymorphic Data Types*

We can extend our definition of `Shape` to permit any numeric type as the parameter. Here is the corresponding definition.

```
data (Num a) => (Shape a) = Square a | Circle a | Rectangle a a
  deriving (Eq, Ord, Show)

size :: (Shape a) -> a
size (Square x)      = x
size (Circle r)      = r
size (Rectangle l w) = l+w
```

Note the conditional dependence on `Num a`. Observe also that we need to include the type parameter `a` in the name of the type – the data type is `Shape a` not just `Shape`.

The constructor Listof combines a value of type Int with a nested instance of Mylist. For example, a value of type Mylist corresponding to the list [1,3,2] is written Listof 1 (Listof 3 (Listof 2 Empty).

5.4 *Recursive Data Types*

We can have recursive data types. Here is an example.

```
data Mylist = Empty | Listof Int Mylist
```

Here the constructors are `Empty` and `Listof`. `Empty` has zero arguments and is hence a constant, representing the base case of the recursive type. The constructor `Listof` combines a value of type `Int` with a nested instance of `Mylist`. For example, a value of type `Mylist` corresponding to the list `[1,3,2]` is written `Listof 1 (Listof 3 (Listof 2 Empty)`. In Haskell's built-in definition of lists, `Empty` is written as `[]` and `Listof` is written as an infix constructor ":", so the value above becomes the more familiar `1:(3:(2:[])`.

It is a small step to extend `Mylist` to be polymorphic.

```
data Mylist a = Empty | Listof a (Mylist a)
```

Now, a term that uses the constructor `Listof` has a value of type `a` and a nested list of the same type. Note again that the full name of the type is `Mylist a`, not just `Mylist`.

We could use multiple types. For instance, here is a list in which types `a` and `b` alternate, beginning with a value of type `a`. The alternation of types is achieved by inverting the order of `a` and `b` in the nested copy of `Twolist`.

```
data Twolist a b = Empty | Listof a (Twolist b a)
```

5.5 *Stacks*

Haskell provides the built-in collective data type list. It is often convenient to have additional collective types. One such type is a stack. A stack is a structure in which we can add elements one at a time and remove elements one at a time such that the element removed first is the

one that was most recently added – a *last-in-first-out* structure. The insert operation is usually called `push` and the remove operation is usually called `pop`. Thus, we have:

```
push :: a -> (Stack a) -> (Stack a)
pop  :: Stack a -> (a,Stack a)
```

Notice that `pop` requires the stack to be nonempty and returns a pair of values – the element at the top of the stack and the resulting stack with this value removed.

We also add the following function that checks if the given stack is empty.

```
isempty :: (Stack a) -> Bool
```

We have yet to define how to represent a stack. Here is one possible definition:

```
data Stack a = Empty | St a (Stack a)
```

We can now instantiate the functions in terms of this definition.

```
push :: a -> (Stack a) -> (Stack a)
push x s = St x s

pop  :: Stack a -> (a,Stack a)
pop (St x s) = (x,s)

isempty :: (Stack a) -> Bool
isempty Empty = True
isempty s = False
```

Except for renaming constructors, our definition of the data type `Stack` is the same as that of the handcrafted list data type `Mylist` that we defined last time. This suggests that we could directly use the built-in list type and write

```
data Stack a = St [a]
```

A stack is a structure in which we can add elements one at a time and remove elements one at a time such that the element removed first is the one that was most recently added – a *last-in-first-out* structure.

A queue is a *first-in-first-out* structure. Like a stack, it has basic operations to add and remove elements, but the element that is removed is the one that was added earliest.

We need a constructor to associate with the value `[a]`, but otherwise all the functions we use are derived from the structure of lists.

```
push x (St xs) =  St (x:xs)
pop  (St (x:xs)) =  (x,St xs)
isempty (St l) = (l == [])
```

5.6 *Queues*

A queue is a *first-in-first-out* structure. Like a stack, it has basic operations to add and remove elements, but the element that is removed is the one that was added earliest. Here are the operations that we would like to perform on queues:

```
addq :: a -> (Queue a) -> (Queue a)
removeq :: (Queue a) -> (a,Queue a)
isemptyq :: (Queue a) -> Bool
```

Observe that the signatures are the same as those for the functions `push`, `pop` and `isempty` that we defined for stacks, modulo the new name of the datatype. We can again implement a queue using a list, as follows.

```
data Queue a = Qu [a]

addq x (Qu xs) = (Qu xs ++ [x])
removeq (Qu (x:xs) = (x,Qu xs)
isempty (Qu l) = (l == [])
```

Adding an element to a queue takes time proportional to the size of the queue. Removing an element takes constant time. In contrast, in a stack, both push and pop take constant time, independent of the size of the stack.

Here, `removeq` and `isemptyq` have essentially the same definition as `pop` and `isempty` for stacks. Only `addq` is different – the new element is appended at the end of the list rather than at the beginning.

This is an important difference – adding an element to a queue takes time proportional to the size of the queue. Removing an element takes constant time. In contrast, in a stack, both `push` and `pop` take constant time, independent of the size of the stack.

Suppose we push and pop n elements in a stack. This will take $O(n)$ time, regardless of the way the pushes and pops are interleaved. On the other hand, for a queue, if we first do n `addq`'s and then n `removeq`'s, it takes time $O(n^2)$ to build up the queue, since each `addq` takes time proportional to the length of the queue. We could, of course, reverse the representation and add elements to the front of the list and remove them from the rear. Then, `addq` would be a constant time operation while `removeq` would take time proportional to the length of the list.

Can we do better? Can we find an implementation of a queue in which n `addq`'s and n `removeq`'s take $O(n)$ time, regardless of the order in which these operations appear? We imagine that we break a queue into two parts and use a separate list to represent the front and the rear. Since we remove elements from the front portion, the first element in the front should be at the head. We add elements to the end of the rear, so, to avoid traversing the rear portion each time we add a new element, we maintain the rear portion in reverse, with the end of the queue at the head of the list.

Here is the data declaration and the definition of `addq`.

```
data Queue a = Nuqu [a] [a]
addq x (Nuqu ys zs)  = Nuqu ys (x:zs)
```

Recall that `zs` represents the rear of the queue, in reverse, so the last element of the queue is at the head of `zs`, and `x` is added before this element.

How about `removeq`? If the left list is nonempty, we just extract its head. If it is empty, we reverse the entire rear into the front and then extract its head.

```
removeq (Nuqu (x:xs) ys)  = (x,Nuqu xs ys)
removeq (Nuqu [] ys)  =
            removeq (Nuqu (reverse ys) [])
```

A *tree* is a structure in which each node has multiple successors, called *children*. There is a special node, called the *root*, from which the tree begins.

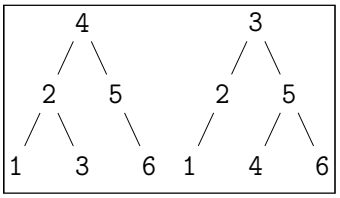
Why is this any better? After all, after adding n elements the queue would be `Nuqu [] [xn,...,x2,x1]`, so the first `removeq` will take $O(n)$ time, since `reverse` takes $O(n)$ time. Note, however, that the $O(n)$ time taken to extract `x1` also transfers `[x2,..,xn]` to the front of the queue. Thus, after one `removeq`, we have `Nuqu [x2,...,xn] []`. The next $n-1$ `removeq` operations take only $O(1)$ time each. In this way, overall, adding n elements and then removing them takes only $O(n)$ operations. The $O(n)$ cost of extracting the first element can be thought of as *amortized*, or spread out, over the next $n-1$ `removeq` operations.

5.7 *Trees*

A *tree* is a structure in which each node has multiple successors, called *children*. There is a special node, called the *root*, from which the tree begins. The root is usually drawn as the topmost node in the tree. Every node other than the root has a unique *parent* (the node of which it is a *child*), and hence a unique path back to the root following parent links. *Figure* 3 shows two examples of trees.

In the trees we have drawn, a value is stored at each node. As in lists, these values have a uniform type – `Int`, in the examples above. A bottom level node with no children is called a *leaf node*. Non-leaf nodes are called *internal nodes*. Internal nodes in a tree need not have a uniform number of children. For instance, the node with value `5` in the left tree has only one child while the node with value `2` has two children. The order of the children is important. In the trees we have drawn, each node has up to two children and the two children are oriented as *left* and *right*. Thus, for the tree on the left, `2` is the left child of the root `4` and `3` is the right child of `2`. Notice that though `5` has only one child, `6`, this is a right child, not a left child.

Figure 3. Trees.



We will typically look at *binary trees*, in which each node has up to two children. Here is one way to describe a binary data over an arbitrary type `a`.

```
data BTree a = Nil | Node (BTree a) a (BTree a)
```

Lists have a linear structure, so there is only one measure of size for a list, the length of the list. Trees are two dimensional, so we consider two quantities:

- *Size* : the number of nodes in the tree
- *Height* : the length of the longest path from a root to a leaf

We can compute both of these quantities inductively, as with lists.

Lists have a linear structure, so there is only one measure of size for a list, the length of the list. Trees are two dimensional, so we consider two quantities:
Size: the number of nodes in the tree;
Height : the length of the longest path from a root to a leaf.

```
size :: (Btree a) -> Int
size Nil = 0
size (Node t1 x t2) = 1 + (size t1) + (size t2)

height :: (Btree a) -> Int
height Nil = 0
height (Node t1 x t2) = 1 + max (height t1) (height t2)
```

5.8 *Binary Search Trees*

An important use of binary trees is to store values that we may want to look up later. For instance, a binary search tree could be used to store a dictionary of words. A binary search tree is a tree with no duplicate values that satisfies the following property at every node v: all values in the subtree rooted at v that are smaller than the value stored at v lie in the left subtree of v and all values in the subtree rooted at v that are larger than the value stored at v lie in the right subtree of v. To emphasize that the values in the tree can be ordered, we elaborate slightly on the Haskell definition of binary trees to describe search trees.

A binary search tree is a tree with no duplicate values that satisfies the following property at every node *v*: all values in the subtree rooted at *v* that are smaller than the value stored at *v* lie in the left subtree of *v* and all values in the subtree rooted at *v* that are larger than the value stored at *v* lie in the right subtree of *v*.

```
data (Ord a) => STree a = Nil | Node (STree a)
                                     a (STree a)
```

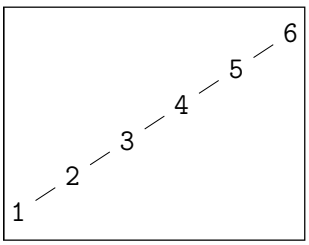
Observe that the structure of an `STree` is identical to that of a normal `BTree`, but there is a type class dependence on `Ord`, similar to the one we have seen for polymorphic functions such as `quicksort`. We have changed the name of the data structure from `BTree a` to `STree a` to emphasize that we are working with *search* trees rather than ordinary binary trees.

Both the trees in *Figure* 3 are examples of search trees over the values `[1,2,3,4,5,6]`. Though these two trees look reasonably well balanced, this is not always the case. For instance, *Figure* 4 shows a highly unbalanced search tree over the same set of values.

To find a value in a binary search tree, we start at the root. At each node, if we have not already found the value we are looking for, we can use the search tree property to decide whether to search in the right subtree or the left subtree. We keep walking down the tree in this fashion till we find the value we seek or we reach a leaf node from where we cannot descend further. Thus, each lookup in a binary search tree traverses, in the worst case, a single path from the root to a leaf node.

How much time does it take to look up a value in a balanced search tree with n nodes? A tree is balanced if at each node the size of the left subtree differs from the size of the right subtree by at most 1. Initially, we search for the value in the entire tree, with n nodes. If we do not find the value at the root, we search either the left or the right subtree. Since the tree is balanced, the number of nodes in each of these subtrees is at most $\frac{n}{2}$. In this way, we successively search trees of size n, $\frac{n}{2}$, $\frac{n}{4}$, ... till we reach a leaf node, a subtree of size 1. The length of this sequence is clearly bounded by $\log n$ – in other words, the height of a balanced search tree with n nodes is $\log n$.

Figure 4. A skewed search tree.



Here is a Haskell definition of the search procedure we just described:

```
findtree :: (Stree a) -> a -> Bool
findtree Nil x = False
findtree (Node tleft y tright) x
  | x == y    = True
  | x < y     = findtree tleft x
  | otherwise = findtree tright x
```

Search trees are not static objects. In general, we have to insert new values into search trees and remove stale values from search trees.

Where should we insert a value into a search tree? From the definition of a search tree, there is only one possibility. Search for the value in the tree. If it already exists, there is nothing to be done. Otherwise, we reach a leaf node. This is the same path that we would have to follow to find the new value after it has been inserted. So, insert the new value as a left or right child of the leaf node where the unsuccessful search terminates.

Where should we insert a value into a search tree? From the definition of a search tree, there is only one possibility. Search for the value in the tree. If it already exists, there is nothing to be done. Otherwise, we reach a leaf node. So, insert the new value as a left or right child of the leaf node where the unsuccessful search terminates.

```
inserttree :: (Stree a) -> a -> (Stree a)
inserttree Nil x = Node Nil x Nil
inserttree (Node tleft y tright) x
  | x == y    = Node tleft y tright
  | x < y     = Node (inserttree tleft x) y tright
  | otherwise = Node tleft y (inserttree tright x)
```

Clearly, the maximum number of steps required to insert a value into a search tree is equal to the length of the longest path in the tree. Thus, if the search tree is balanced and has n nodes, `inserttree` takes time $\log n$, but will not in general preserve the balanced structure of the tree.

The maximum number of steps required to insert a value into a search tree is equal to the length of the longest path in the tree.

How do we delete a value from a tree? We interpret *"delete* `x` *from* `t`*"* as *"delete* `x` *from* `t` *if the value exists in* `t`*"*. Suppose we want to delete a value `x` from a tree

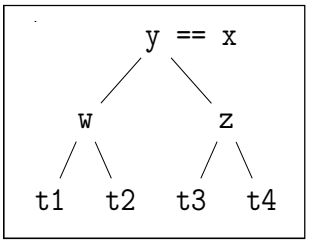


Figure 5. Deleting a value from a search tree.

whose root is y. If x < y, we inductively delete x from the left subtree of y. Similarly, if x > y, we inductively delete x from the right subtree of y. So, the interesting case is when y == x, as shown in *Figure* 5.

If we remove y, we have a hole at the root of the tree. It is tempting to move either w (or z) into this place and recursively delete w from the left subtree (or z from the right subtree). However, this would not preserve the structure of the tree – for instance, if we move w up to the root, values in the tree t2, which are bigger than w, will end up to the left of w.

The correct solution is to move the largest value from the left subtree of y (or the smallest value from the right subtree of y) in place of y. The largest value in a search tree can be found easily, by following the rightmost path in the tree. Removing this value from a tree is also a relatively easy operation. Here is a function that removes the maximum value from a nonempty tree, returning both the value and the modified tree, after deletion.

```
deletemax :: (STree a) -> (a,STree a)
deletemax (Node t1 y Nil) = (y,t1)
deletemax (Node t1 y t2) = (z, Node t1 y tz)
  where (z,tz) = deletemax t2
```

We can now rewrite deletetree as follows:

```
deletetree :: (Stree a) -> a -> (Stree a)
deletetree Nil x = Nil
deletetree (Node tleft y tright) x
 | x < y = Node (deletetree tleft x) y tright
 | x > y = Node tleft y (deletetree tright x)

-- In all cases below, we must have x == y

deletetree (Node Nil y tright) x   = tright
deletetree (Node tleft y tright) x =
                              Node tz z tright
  where (z,tz) = deletemax tleft
```

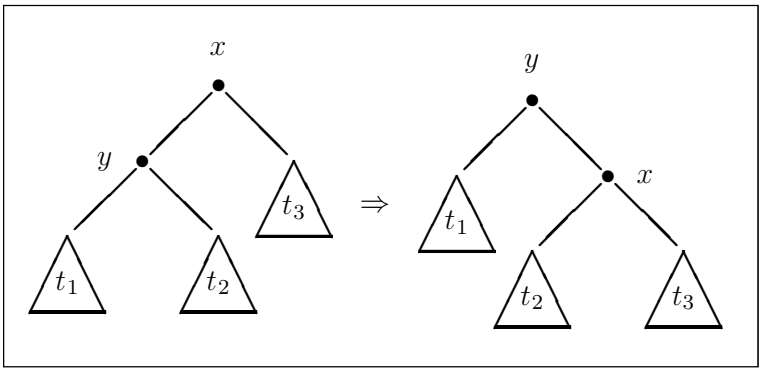


Figure 6. A tree rotation.

5.9 *Balanced Binary Search Trees*

In general, the functions `insert` and `delete` that we have described may result in skewed trees where the height is not logarithmic in the size. We can inductively maintain balance in a tree by appropriately *rotating* parts of a tree after each update. A typical rotation is shown in *Figure* 6. We will not go into the details of how to use such rotations to maintain a balanced search tree. However, it is instructive to note that the restructuring shown in *Figure* 6 can be directly rendered in Haskell as follows, again emphasizing the clarity that is achieved with a declarative style of programming.

```
rotateright (Node x (Node y t1 t2) t3) =
              Node y t1 (Node x t2 t3)
```

6. Conclusion

This concludes our two-part introduction to functional programming in Haskell[3]. We have seen that functional programming allows us to describe complex computations in a declarative manner. Haskell's type mechanism supports a sophisticated version of conditional polymorphism. Outermost evaluation allows us to describe infinite data structures, which can be fruitfully used in solving search problems. Finally, we have seen that we can define and manipulate user-defined data types quite naturally.

[3] A number of compilers and interpreters are available in the public domain for Haskell. The easiest to install and use is the interpreter Hugs, which runs on both Windows and Unix based systems. There is an active website, http://www.haskell.org, that has pointers to software, tutorials, books and reference material on Haskell.

Address for Correspondence
Madhavan Mukund
Chennai Mathematical Institute
H1, SIPCOT IT Park
Padur PO
Siruseri 603 103, India
Email:madhavan@cmi.ac.in
http://www.cmi.ac.in/~madhavan