Higher-order + Polymorphic = Reusable

Simon Thompson
Computing Laboratory, University of Kent
Canterbury, CT2 7NF, U.K.

Abstract

This paper explores how certain ideas in object oriented languages have their correspondents in functional languages. In particular we look at the analogue of the iterators of the C++ standard template library. We also give an example of the use of constructor classes which feature in Haskell 1.3 and Gofer.

1 Introduction

The combination of higher order and polymorphic functions in modern programming languages — chiefly in functional languages, but increasingly in object-oriented languages as well — makes them most suitable for software reuse. Polymorphism allows operations to be applied over whole classes of types, whilst function parameters mean that particular operations can be abstracted away, to be passed in as values on application. the first part of the paper provides a tutorial on this, in the Miranda[1] functional programming language.

Beyond this 'algorithmic' abstraction, there lies data abstraction: access to a particular type can be given (solely) through a signature of operations, hiding the concrete nature of the type. In the second half of the paper we show that using a higher-order, polymorphic interface signature we can describe the essential properties of various types, and using this approach, we can treat one type as if it were another. The examples in the paper include treating lists as trees, and trees as lists. The inspiration for this work came from the *iterators* of [Musser and Stepanov, 1994]. An iterator is an abstract index into a sequential structure, giving a uniform treatment of lists arrays and trees. The aim of the work here is to extend that approach in two ways. First, we wanted to see how to describe an iterator in a functional language – the list-like types. More importantly, we wanted to be able to take a different abstract view of types: the paradigm embodied by trees is divide and conquer, and we show that if we take a *tree-like* view of lists, the Quicksort algorithm can be developed in a very natural way.

Any language allowing higher order functions can be used to implement these ideas, but extra features can facilitate it further. In particular the constructor classes of Gofer,

[1]Miranda is a trade mark of Research Software Ltd.

[Jones, 1993], allow the recursion operators to be overloaded, avoiding the necessity of passing the appropriate parameters into applications. The paper concludes with a survey of the different language features which can be used to aid the implementation of these ideas.

I am grateful to Ian Utting for his patient explanations of some of the arcana of `C++`, and for pointing me in the direction of the Standard Template Library, [Stepanov and Lee, 1994], which is implemented using the iterator classes.

2 Functional programming

Modern functional programming languages like Miranda, Standard ML and Haskell [Turner, 1985, Milner *et al.*, 1990, Hudak *et al.*, 1992] are based on function definitions as *equations* or more generally as sequences of equations. For example, suppose we have already defined the function

```
sales :: num -> num
```

that is a function `sales` which takes a numerical argument (i.e. an argument of type `num`) and returns a number. We can then write a function to return the sum of the first `n` values of `sales` thus:

```
totalSales :: num -> num

totalSales 0 = 0
totalSales n = totalSales (n-1) + sales (n-1)
```

The first equation gives the value of `totalSales` on `0`, the second on non-zero values, illustrating the fact that the ordering of the defining equations is significant, the first one to apply being chosen.

Functions can have various alternatives on the right-hand side. If we want only to add the positive sales we may write

```
totalPosSales :: num -> num

totalPosSales 0 = 0
totalPosSales n
   = totalPosSales (n-1) + sales (n-1)   , if sales (n-1) > 0
   = totalPosSales (n-1)                 , otherwise
```

Finding the time of the maximum sales, we can write

```
maxMonth 0 = 0
maxMonth n = n      , if sales n > sales old
```

```
    = old    , otherwise
      where
      old = maxMonth (n-1)
```

in which the `where` clause attached to the second equation is used to make local definitions whose scope is that equation.

If `t` is a type, then `[t]` is the type of lists of items from type `t`, so, for example,

```
[2,3,4] :: [num]            [[2],[],[3,4]] :: [[num]]
```

Functions over lists are defined by equations just as above, so to sum a list of numbers we write

```
sum :: [num] -> num
sum []    = 0
sum (a:x) = a + sum x
```

On the left hand sides of the equations are seen *patterns* rather than variables. Patterns provide a case analysis: is the list empty, `[]` or not, `(a:x)`? In the latter case it has a first element, or head, `a` and remainder, or tail, `x`; these components can be used in calculating the result, thus pattern matching allows the selection of parts of structured data objects.

Thus far, what we have seen has much in common with traditional imperative programming languages: alternatives are provided by equations with multiple right hand sides, iteration by recursion, and local definitions and calculations by `where` clauses.

3 Higher-order functions

The function `totalSales` returns running totals for the function `sales`; if we wish to do the same for `newSales` we have to write rewrite the function `totalSales`. All that changes is the function called. Instead of demanding a re-write, we can *abstract* from the particular function to be summed, and make this a parameter, thus:

```
total :: (num -> num) -> num -> num

total f 0 = 0
total f n = total f (n-1) + f (n-1)
```

The running totals of particular functions are then given by

```
totalSales    = total sales
```

```
totalNewSales = total newSales
```

where we have *partially applied* the function `total` to the first of its arguments; the result is itself a function, which when given a number returns a number. We call a function *higher order* when it takes a function as argument or returns a function as result; `total` is higher-order in both senses.

The abstraction here turns a particular operation, `sales`, into a parameter, `f`; the calculation of running totals can then be *reused* in many situations, both of similar sales calculations and in completely different contexts.

In a similar way, we might round all numbers in a list to the closest integer,

```
roundList :: [num] -> [num]

roundList []    = []
roundList (a:x) = round a : roundList x

round v = entier (v + 0.5)
```

The operation of rounding is obviously only one of a whole class of possible functions to be applied to every element of a list. The general function to 'map' a function is

```
mapNum :: (num -> num) -> [num] -> [num]

mapNum f []    = []
mapNum f (a:x) = f a : mapNum f x
```

and

```
roundList = mapNum round
```

This Section has shown how abstracting a particular sort of behaviour from a function increases its generality: not only is it applicable to forming its original target, but also it can be used to form a host of other functions.

4 Polymorphism

Suppose we want the length of a list of numbers. We can write

```
# []    = 0                                                    (1)
# (a:x) = # x + 1
```

(Note that function application binds more tightly then operator application, so that `# x + 1` equals `(# x)+1`.) The definition contains no reference to the type of objects in the list argument, and so it is applicable to a list of *any* type. Its type is

```
# :: [*] -> num                                          (2)
```

in which the '`*`' is a *type variable*. Just as for an ordinary variable, a type variable is used to signify that the property is valid for any *instance* of the variable. In this case, `(2)` implies that

```
# :: [[num]] -> num
# :: [bool]  -> num
```

to take two examples. Types containing variables, such as `(2)`, are called *polymorphic*; alternatively we might call the definition `(1)` *generic*.

Other objects of polymorphic type are `[] :: [*]` and

```
swap :: (*,**) -> (**,*)
swap (a,b) = (b,a)
```

where the elements in a pair have their order swapped. For instance, `swap (True,3)` is `(3,True)`; from this example it is apparent that the type variables `*` and `**` can be given different instances, so that

```
swap :: (bool,num) -> (num,bool)
```

in the example seen above.

Polymorphism supports re-use: a function intended to give the length of numerical lists can be used over *any* list type, for example.

5 Higher-Order + Polymorphic

Separately, we have seen that higher-order functions and polymorphic functions each support reuse. Together, we have *general* functions of considerable power. In Section 3 we defined a higher-order function to 'map' numeric functions along lists of numbers. Nowhere in the definition is it apparent that the lists are numerical, and we have the general definition

```
map :: (* -> **) -> [*] -> [**]

map f []    = []
map f (a:x) = f a : map f x
```

We have two dimensions of generality here:

- The type of list along which the operation is applied is arbitrary, as is the result type of the operation.

- Once the input and output types are known, there is a choice of operations to be applied: the choice is given as an argument when `map` is applied.

Forming general functions

How do we find the definitions of general functions? Often it can be seen as a two-stage process, which we illustrate with the `sum` function from Section 2.

```
sum :: [num] -> num
sum []    = 0
sum (a:x) = a + sum x
```

What can become a parameter here?

- The operation we use to combine the values, `(+)` is a function of type `num -> num -> num`
- The start value, `0`, is a number.

Rewriting, we have

```
foldr :: (num->num->num) -> num -> [num] -> num

foldr f st []    = st                                       (3)
foldr f st (a:x) = f a (foldr f st x)

sum = foldr (+) 0
```

The operation is called `foldr` because it involves *folding* in the function `f` from the *right*. For example,

$$\texttt{foldr (+) 0 [}v_1, v_2, \ldots, v_n\texttt{]} = v_1 + (v_2 + \ldots + (v_n + 0)\ldots)$$

After transforming from the particular definition to `sum` a list, the definition `(3)` becomes polymorphic, since there is no longer any reference to the lists being made up of numbers. We have

```
foldr :: (*->*->*) -> * -> [*] -> *
```

so that we can join a list of lists by folding in the operator `++` which joins together two lists:

```
concat :: [[*]] -> [*]
```

```
concat = foldr (++) []

concat [[2,3],[],[4,1]]
= [2,3] ++ ([] ++ ([4,1] ++ []))
= [2,3,4,1]
```

In fact, the full generality[2] of the type is

```
foldr :: (*->**->**) -> ** -> [*] -> **
```

In other words, there is no reason for the function folded in to return a value of the same type as the elements of the list. For example, we can define the length function by folding in `addOne`

```
# = foldr addOne 0
    where
    addOne val len = len + 1
```

General *vs* polymorphic

It is instructive to examine the polymorphic functions provided in the Miranda 'standard environment'. A few are first-order: they return the length of a list, reverse a list or divide a list into parts by taking or dropping a number of elements; all these functions are 'structural', in that their operation is independent of the elements of the list.[3]

The majority of the library functions over lists will examine elements, and these functions are higher-order, since the way in which the elements are examined is packaged as a function or functions, passed into the general function when it is applied.[4]

This analysis is based on examining a functional library, but there is every reason to believe that in an imperative context a similar conclusion will be reached. [Kershenbaum *et al.*, 1988] discusses the role of higher order imperative programs.

The way in which instantiation takes place is assumed here to be function application; an argument can be made for replacing higher-order functions by parametrised modules, [Goguen, 1990], but this mechanism appears to be simultaneously more cumbersome and less powerful; we discuss this further in Section 9 below.

[2] This greater than anticipated generality is not an isolated phenomenon.

[3] This intuitive characterisation can be made formal; the functions are strict in the *spine* the list, but are lazy in the elements themselves.

[4] In general these functions will be strict in the elements of their list arguments. For instance, `map f` is as strict in the elements as is the function `f`.

6 The essence of lists

The type `[*]` is characterised by its constructors

```
[]  :: [*]
(:) :: * -> [*] -> [*]
```

the list `[2,12]` being built thus:

```
2 : (12 : [])
```

Every list can be seen as arising in this way, by repeatedly adding elements to the empty list. How are lists used? A typical definition has the form

```
g []    = st
g (a:x) = ... a ... x ... g x ...
```

where the right-hand side of the definition of `g (a:x)` uses the components `a` and `x` as well as the value of g on the tail, `g x`. Rewriting this as a function application, we have

```
g []    = st
g (a:x) = h (g a) a x
```

As a higher order function, we define

```
fold :: (** -> * -> [*] -> **) -> ** -> [*] -> **
fold h st []    = st
fold h st (a:x) = h (fold h st x) a x
```

which can be seen as a generalisation of the `foldr` function, in that `h` takes the tail `x` as an argument in addition to the head of the list and the recursive call of the folding function.[5]

Using `fold`, `empty = []` and `cons = (:)` we can define any list processing function. For example,

```
tail =  fold tailPart (error "tail")
        where
        tailPart v a x = x
map g = fold h empty
        where
        h v a x = cons (g a) v
```

[5]Note that the `fold` function here is *not* the same as the `fold` in the Miranda standard environment.

From another, more abstract, point of view, we can see *any* family of types `t *` as being a list of `*` if we have objects

```
empty :: t *
cons  :: * -> t * -> t *                                    (4)
fold  :: (** -> * -> t * -> **) -> ** -> t * -> **
```

These objects can become *parameters* to the definitions of `tail`, `map`, and so on, giving

```
mapGen :: ( t * ,
            * -> t * -> t * ,
            (** -> * -> t * -> **) -> ** -> t * -> ** ) ->
          (*** -> ****) -> t *** -> t ****
mapGen (empty,cons,fold) g
      = fold h empty
        where
        h v a x = cons (g a) v
```

To summarise, we have shown in this Section that a family of types `t *` is *list-like* if we can define functions conforming to the signature `(4)`. Given such a family, we can define all the general functions over lists over the types in the family.

The novelty of this approach is that `t *` can be any family of types: we shall see in the next Section that trees and error types conform to this signature.

This characterisation is inspired by the *iterators* of the Standard Template Library for `C++`, [Stepanov and Lee, 1994, Musser and Stepanov, 1994]. Iterators are an abstraction from indices (or pointers), allowing a walk through a sequential structure. List-like types are analogue of these in a functional setting, and the tree-like types discussed in Section 8 generalise them in providing a 'divide and conquer' interface to a data structure, which is not provided (at least directly) by iterators.

7 Views of data: list-like types

We have various examples of list-like types, which we enumerate now.

Trees as lists

Given the definition

```
tree * ::= Leaf | Node * (tree *) (tree *)
```

we can make a sequential traversal of these trees. The definitions of `empty`, `cons` and `fold` follow now.

```
empty = Leaf

cons a Leaf            = Node a Leaf Leaf
cons a (Node b t1 t2) = Node b (cons a t1) t2
```

As can be seen from the definition, in this case the traversal is in pre-order; other traversals of the trees give rise to other definitions and therefore other ‘views’ of trees as lists.

```
fold f st Leaf = st
fold f st t    = f (fold f st t') b t'
                 where
                 (b,t') = splitTree t
```

and the splitting up of the general tree is given by

```
splitTree :: tree * -> (*,tree *)
splitTree Leaf            = error "splitTree"
splitTree (Node a Leaf t) = (a,t)
splitTree (Node a t1 t2)  = (b,Node a t1' t2)
                            where
                            (b,t1') = splitTree t1
```

As we remarked above, other traversals give rise to other `cons` and `fold` functions; whatever the case, the family of general list processing functions will be available.

Snoc lists

Elements are added to the end of a list rather than the start by `cons`, and folding is also done from that end. The requisite definitions are

```
empty = []
cons  = snoc
fold  = foldAlt
```

where

```
snoc a []    = [a]
snoc a (b:x) = b : snoc a x

foldAlt f st [] = st
foldAlt f st x  = f (foldAlt f st x') a x'
                  where
```

```
                  (a,x') = split x

split :: [*] -> (*,[*])
split []    = error "split"
split [a]   = (a,[])
split (a:x) = (b,(a:x'))
              where (b,x') = split x
```

Other examples

Other examples are given by other traversals of trees, ordered traversals of lists, and the error types

```
err * ::= Error | OK *
```

We leave the definitions as exercises for the reader.

8 Tree-like types

A typical recursion over the type

```
tree * ::= Leaf | Node * (tree *) (tree *)
```

has the form

```
h Leaf            = st
h (Node a t1 t2) = ...h t1...h t2...a...t1...t2...
```

and so is an application of

```
treeRec :: (** -> ** -> * -> tree * -> tree * -> **) ->
           ** -> tree * -> **
treeRec f st Leaf = st
treeRec f st (Node a t1 t2)
  = f (treeRec f st t1) (treeRec f st t2) a t1 t2
```

A *tree-like* type `t *` carries the operations

```
leaf    :: t *
node    :: * -> t * -> t * -> t *
```

```
treeRec :: (** -> ** -> * -> t * -> t * -> **) -> ** -> t * -> **
```

and using these operations we can form a sorting algorithm:

```
tSort :: t * -> [*]
tSort = treeRec mergeVal []
        where
        mergeVal sort1 sort2 val t1 t2 = mVal sort1 sort2 val
```

where the function `mVal :: [*] -> [*] -> * -> [*]` takes two sorted lists and a value and merges them into a single sorted list. Its definition simultaneously generalises a merge of two sorted lists and the insertion of an element into a sorted list, and is left as an exercise for the reader.

The essence of a recursion over a tree is that is works by *divide and conquer*, so that viewing an arbitrary type as a tree will give divide and conquer algorithms over that type. Our first example is (of course) trees themselves, but we can also view lists as trees.

Lists as trees

We define, over lists

```
leaf         = []
node a l1 l2 = l1 ++ [a] ++ l2
```

and for recursion,

```
treeRec f st [] = st
treeRec f st l  = f v1 v2 a t1 t2
                  where
                  v1 = treeRec f st t1
                  v2 = treeRec f st t2
                  (a,t1,t2) = listToTree l
```

and lists are bisected thus

```
listToTree :: [*] -> (*,[*],[*])

listToTree []    = error "listToTree"
listToTree (a:x) = (a,l1,l2)
                   where
                   n  = #x div 2
                   l1 = take n x
```

```
            l2 = drop n x
```

Other views of lists as trees are possible. If, for instance, we re-define `listToTree` so that

```
listToTree (a:x) = (a, [ b | b<-x ; b<=a ] , [ b | b<-x ; b>a ])
```

then the function to flatten a tree

```
flatten = treeRec joinUp []

joinUp flat1 flat2 a t1 t2 = flat1 ++ [a] ++ flat2
```

becomes the *quicksort* function.

9 Type abstractions and language mechanisms

The last three Sections have shown how to take an abstract view of data structures. Not only do we have libraries of general functions over lists, trees and so on, but these libraries can be extended, by further parametrisation, to list-like and tree-like types. As examples, we showed that lists can be tree-like, and trees list-like, according to the kind of recursion we wish to perform. This Section looks at the mechanisms by which this view of abstraction can be supported in various programming languages.

Parameter passing

In a language with higher-order functions, we can define the library functions such as `map` relative to a triple of parameters, as we did for `mapGen` in Section 6 above.

This method calls for explicit parameters to be given at each invocation of `mapGen`, but has the advantage that `mapGen` can in a single scope be used over different types, or different recursors over the same type. An instance of the latter might be to consider ordinary recursion (`fold`) and snoc-recursion (`foldAlt`) in the same scope.

Constructor classes

The Gofer language supports constructor classes, [Jones, 1993], which generalise the type classes of Haskell by (essentially) allowing the classification of type constructors rather than types. Our classifications of lists and trees in this paper give exactly constructor classes. Using a hybrid notation (`*`'s for type variables), we can give the class of tree-like types thus:

```
class TreeRec t where
leaf    :: t *
node    :: * -> t * -> t * -> t *
treeRec :: (** -> ** -> * -> t * -> t * -> **) -> ** -> t * -> **
```

We can then in a single scope give various *instantiations* of the class. We can declare lists as tree-like thus

```
instance TreeRec List where
 leaf = []
 node = ...
 treeRec = ...
```

The definition of `tSort` using `leaf`, `node` and `treeRec` will then be applicable over lists.

Note, however, that only one instance declaration per type is allowed in any single scope. Using this mechanism, therefore, does not allow us to view lists as trees in two different ways in a single scope.

Abstract data types

Many programming languages allow declarations of abstract data types, by means of a specified signature to which an implementation is bound. Note that only one implementation is allowed per scope, and that the abstraction given by such a type is *closed*: no operations are allowed on the type beyond those in the signature. This *proscriptive* approach does not fit well with the *permissive* tenor of this work, where we use the tree-like nature of a type to allow certain sorts of definition, rather than to prevent all others.

Structures and signatures

The approach of binding structures to signatures in Standard ML is permissive, but will not allow the *overloading* of multiple bindings per scope. The same restriction applies to parametrised modules, as in [Goguen, 1990].

Dynamic binding

In most object-oriented languages there are mechanisms for dynamic binding. In `C++`, for example, a virtual class forces all its subclasses to provide operations, such as those of a tree-like type. Each subclass can implement these in different ways, but objects of any of these subclasses are all considered to be of the virtual class. A list of objects of such a class will have to associate operations with values in the list dynamically, since a single operation will not in general serve all objects of the list.

10 Conclusion

The aim of this paper has been to show that the twin features of higher-order functions and polymorphism (or generics, in other terminology) support a programming style in which re-use of software is encouraged. We illustrated the proposition with the example of data abstraction — the treatment of one type as if it were structured like another

— which supports a general, ‘algorithm oriented’ style, as introduced in [Musser and Stepanov, 1994].

References

[Goguen, 1990] Joseph Goguen. Higher-order functions considered unnecessary for higher-order programming. In David A. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[Hudak *et al.*, 1992] Paul Hudak, Simon Peyton Jones, and Philip Wadler (Editors). Report on the Programming Language Haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[Jones, 1993] Mark Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA 93 – Functional Programming Languages and Computer Architecture*. ACM Press, 1993.

[Kershenbaum *et al.*, 1988] Aaron Kershenbaum, David Musser, and Alexander Stepanov. Higher order imperative programming. Technical report, Computer Science Department, Rensselaer Polytechnic Institute, 1988.

[Milner *et al.*, 1990] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Musser and Stepanov, 1994] David Musser and Alexander Stepanov. Algorithm-oriented Generic Libraries. *Software — Practice and Experience*, 24, 1994.

[Stepanov and Lee, 1994] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical report, Hewlett-Packard Laboratories, Palo Alto, USA, 1994.

[Turner, 1985] David A. Turner. Miranda: a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1985.