Functions as Data

Michael P. Fourman

February 2, 2010

In ML functions are first-class objects: they can be passed as arguments, returned as values, and incorporated as components of compound values. Sometimes this can be an effective way of representing data; in particular, it allows a common interface to be provided for similar functions on differently represented datatypes. By including these functions as part of the representation of an object, we can implement some features of an "object-oriented" programming style in ML. We will return to this use of functions as data later in the course. We can also use functions to represent infinite datastructures, such as the sequence of all prime numbers. Again, we will return to this topic later in the course.

In this note, we are more interested in the use of functional representations to provide "executable specifications" of some datatypes that we will later implement in a more traditional fashion. We give some examples that make use of functions as values.

Dictionaries

Our first example is an implementation of the `Dictionary` signature. Consider a dictionary, like the telephone dictionary, with strings as keys, and numbers as entries. Given a string, we can use the dictionary to lookup the corresponding number. This gives us a (partial) function from strings to numbers. We can use functions to implement dictionaries directly. .

A more common implementation of a dictionary is as an *association list,* a list of associated pairs. Mathematicians call the set of such pairs the *graph* of the function; this may be confusing, as in computer science we use the term graph for a related, but more general, notion.

```
signature DictSig =
sig
   type Key
   type Item
   type  Dict
   exception Lookup
   val empty  :  Dict
   val lookup :  Dict -> Key -> Item
   val remove : Key *  Dict ->  Dict
   val enter  : (Key * Item) *  Dict -> Dict
end
```

Figure 1: A signature for Dictionaries

```
structure Dict : DictSig = struct
    exception Lookup
    type Key  = string
    type Item = int
    type Dict = string -> item
    val empty = fn _ => raise Lookup

    fun lookup d k = d k

    fun remove (k, d) =
        fn k' => if k' = k then raise Lookup
                           else d k'

    fun enter ((k, e), d) =
        fn k' => if k' = k then e
                           else d k'
end;
```

Figure 2: A Dictionary implemented as a function

```
structure Dict : DictSig =
struct
    exception Lookup

    type Key  = string
    type Item = int
    type Dict = (string * int) list

    val empty = []

    fun lookup [] k       = raise Lookup
      | lookup ((k', e) :: t) k = if k' = k then e else lookup t k

    fun remove (k, []) = []
      | remove (k, (k',e) :: t) = if k = k' then t
                                            else (k', e) :: remove k t

    fun enter ((k, e), d) = (k,e) :: remove k d
end;
```

Figure 3: A Dictionary implemented as an association list

Sets

A set may be represented by a boolean-valued function, or *predicate,* of type `Item -> bool`, that tells us whether a given item is a member of the set. This gives an implementation of most of the set operations. However, we

```
infix 4 ==
functor FNSET( type Item val == : Item * Item -> bool ) =
struct
    type Item = Item
    type Set  = Item -> bool

    val empty  = fn _ => false
    fun member  s e = s e

    fun insert(e, s) =
        fn e' => if e == e' then true else s e'

    fun delete(e, []) =
        fn e' => if e == e' then false else s e'

    fun union (s, t)     = fn e => s e orelse t e

    fun intersect( s, t) = fn e => s e andalso t e
end;
```

Figure 4: Sets implemented as predicates

cannot implement `IsEmpty`, since there is no way to test whether a given function will always return the answer false. One benefit of this representation is that it allows us to represent infinite sets, such as the set of even numbers.