16 Functional Programming

16.1 Why functional programming?

In Section 1.8 we mentioned that both Church and Turing proposed models of computation long before the first computers were built. Turing machines are very similar to modern computers in that they are based on an *updatable store*, that is, a set of memory cells whose contents are modified when an instruction is executed. This is also known as the *von Neumann* architecture.

Church's formulation of a model of computation (called the *lambda calculus*) is entirely different, being based on the mathematical concept of functions. This formulation is completely equivalent to Turing's in terms of the computations that can be described, but as a practical formalism for computing, the functional approach has been much less popular. The language Lisp, developed in 1956, uses a functional approach to computation similar to the lambda calculus model, though it contains many features that encourage an imperative programming style.

During the 1980's further research into functional programming has resulted in languages that have a very clean theoretical basis and yet can be efficiently implemented. The main difference between modern functional programming languages and Lisp is that types and type checking are basic concepts of these languages, so both the reliability and efficiency of programs are greatly improved.

Many of the problems that we have in writing reliable programs stem directly from the use of an updatable store:

- Memory can be "smeared" because we are directly modifying memory cells (using array indices or pointers), rather than simply computing values.
- It is difficult to construct complex programs from components because subprograms can have side effects. Thus it can be impossible to understand the effect of a subprogram in isolation from the entire program.

Strong type checking and the encapsulation techniques of object-oriented programming can alleviate these problems but not entirely eliminate them. By adopting a functional approach, both of these problems disappear.

The discussion will be based on the popular language Standard ML though the concepts hold for other languages.

16.2 Functions

Functions are defined in ML by equating the name of a function with its formal parameter to an expression:

```
fun even n = (n mod 2 = 0)
```

the difference being that there are no global variables, no assignment,[1] no pointers and hence no side effects. Once a function has been defined, it can be *applied* to a value; the *evaluation* of the application will produce a result:

```
even 4 = true
even 5 = false
```

A type is associated with every function just as types are associated with variables in programming languages. The type of even is:

```
even: int -> bool
```

meaning that it maps a value of integer type into a value of Boolean type.

Expressions used in ML can contain conditions:

```
fun min (x,y) = if x < y then x else y
```

Evaluating an example of an application of the function gives:

```
min (4,5) =
(if x < y then x else y) (4,5) =
if 4 < 5 then 4 else 5 =
if true then 4 else 5 =
4
```

Note that this is not an if-statement but a conditional expression, similar to the C operator:

```
x < y ? x : y
```

What is the type of min? In functional programming, a function is considered to have exactly one argument;[2] if you need more than one, you must create a tuple (pair, triple, etc.) using the Cartesian product function. Thus (4,5) is of type int × int and the function min is of type:

```
min: (int × int) -> int
```

[1] ML actually does have imperative assignment in case you absolutely need it, but we will ignore this aspect of the language.

[2] It is common in functional programming to use the mathematical word argument, rather than the programming word parameter.

Instead of using tuples, you can define a function which will be applied one by one to each argument:

```
fun min_c x y = if x < y then x else y
```

This is called a *curried function*, named after the mathematician H.B. Curry. When this function is applied to a sequence of arguments, the first application creates another function which is then applied to the second.

The function min_c takes one integer argument and creates a new function, also of one argument:

```
min_c 4 = if 4 < y then 4 else y
```

This function can then be applied to another single argument:

```
min_c 4 5 =
(if 4 < y then 4 else y) 5 =
if 4 < 5 then 4 else 5 =
if true then 4 else 5 =
4
```

Curried functions can be used with *partial evaluation* to define new functions:

```
fun min_4 = min_c 4

min_4 5 =
(if 4 < y then 4 else y) 5 =
if 4 < 5 then 4 else 5 =
if true then 4 else 5 =
4
```

16.3 Compound types

Lists

A list can be created from any previously defined type, in particular for predefined types like integer or Boolean. The lists:

```
[2, 3, 5, 7, 11]        [true, false, false]
```

are of types int list and bool list, respectively. Alternatively, a list can be created using *constructors*; the list constructors are [] for the empty list, and element :: list for a non-empty list created by adding an element to an existing list. Constructors can be used when defining functions by *pattern matching*:

```
fun   member [] e = false
|     member [e :: tail] e = true
|     member [e1 :: tail] e = member tail e
```

The type of member is:[3]

```
member: int list × int -> boolean
```

and it can be read as follows:

> When member is applied to a list L and (then) to an element e, the evaluation is based on cases according to the arguments: (1) if L is empty, e is not a member of L; (2) if e is the first element of L, then e is a member of L; (3) otherwise, e1, the first element of L, is not the same as e, so we (recursively) check to see if e is a member of the tail of L.

You do not have to declare the type of a function in ML; the compiler automatically infers the type of the function from the types of the arguments and the type of the result. If the compiler cannot infer the type, you will have to supply enough type declarations to disambiguate the expression. Type checking is static, meaning that when a function is applied to a value, a check is made at compile time that the type of the function matches the type of the argument.

Note that this function is recursive. Recursion is extremely important in functional programming languages; in the absence of "statements" it is the only way of creating loops in expression evaluation.

As a final example, let us show how to write the *insertion sort* algorithm in ML. You use this algorithm to sort your hand when playing cards: take each card one by one from the pile in front of you and place it in its correct place:

```
fun   insertion_sort [] = []
|     insertion_sort head :: tail =
          insert_element head insertion_sort tail
and
fun   insert_element x [] = [x]
|     insert_element x head :: tail =
          if x < head then x :: head :: tail
          else head :: (insert_element x tail)
```

These functions are of types:

```
insertion_sort: int list -> int list
insert_element: int -> int list -> int list
```

Once you get used to the notation it is easy to read such programs:

[3] Actually, the type is 't list × 't boolean, but type variables are not introduced until the next section.

A sorted empty list is the empty list. A non-empty list is sorted by taking the first element x, sorting the remainder of the list tail, and then inserting x in its proper place in the sorted version of the list.

An element x is inserted into an empty list by creating a list of a single element. To insert x into a non-empty list, compare x with the head of the list: (1) if x is less than head, make x the new first element of the list; (2) otherwise create a new list composed of head followed by the list created by inserting x in the remainder of the list.

Note that $->$ associates to the right:

```
insert_element: int -> (int list -> int list)
```

The function maps an integer into another function that maps integer lists into integer lists. Using partial evaluation, we can create new functions like:

```
fun insert_4 = insert_element 4
```

which is a function that inserts 4 into an integer list.

Compared with an imperative program for the same algorithm, there are no indices and no for-loops. Furthermore, it generalizes immediately to sorting objects of other types just by replacing the operator "¡" by an appropriate Boolean function for comparing two values of the type. No explicit pointers are needed to create the list; the pointers are implicit in the data representation. Of course, sorting lists in any language is less efficient than sorting an array in-place but for many applications using lists is practical.

Defining new types

Throughout the book, we have seen that defining new types is essential if a programming language is to model the real world. Modern functional programming languages also have this capability. Let us define a (recursive) type for trees whose nodes are labeled with integers:

```
datatype int tree =
        Empty
|       T of (int tree × int × int tree)
```

This is read:

int tree is a new data type whose values are: (1) the new constant value Empty, or (2) a value formed by the *constructor* T applied to a triple consisting of a tree, an integer and another tree.

Having defined the new type, we can write functions that process the tree. For example:

```
fun   sumtree Empty = 0
|     sumtree T(left, value, right) =
          (sumtree left) + value + (sumtree right)
```

adds the values labeling the nodes of the tree, and:

```
fun   mintree Empty = maxint
|     mintree T(left, value, right) =
          min left (min value (mintree right))
```

computes the minimum of all the values labeling the nodes, returning the largest integer maxint on an empty tree.

All the standard tree algorithms can be written in the same manner: define a new data type that matches the tree structure, and then write functions on the type. No explicit pointers or loops are required, just recursion and pattern matching.

16.4 Higher-order functions

In functional programming, a function is an ordinary object with a type, so it can be an argument of other functions. For example, we can create a generic form of insert_element by simply adding a function compare as an additional argument:

```
fun   general_insert_element compare x [] = [x]
|     general_insert_element compare x head :: tail =
          if compare x head
          then x :: head :: tail
          else head :: (general_insert_element compare x tail)
```

If string_compare is a function from strings to Boolean:

```
string_compare: (string × string) -> bool
```

applying general_insert_element to this argument:

```
fun string_insert = general_insert_element   string_compare
```

gives a function of type:

```
string -> string list -> string list
```

Note that unlike imperative languages, this generalization is achieved naturally without any additional syntax or semantics like generic or template.

But what is the type of general_insert_element? Well, the first argument must be of type "function from a pair of anything to Boolean", the second argument must be of the same "anything", and the third argument is a list of "anything". *Type variables* are used as a shorthand for "anything" and the type of the function is thus:

```
general_insert_element: (('t × 't) -> bool) -> 't -> 't list
```

where type variables are written in ML as identifiers preceded by an apostrophe.

The use of *higher-order functions*, that is functions that have functions as arguments, is not limited to such static constructs as are generics. An extremely useful function is map:

```
fun    map f [] = []
|      map f head::tail = (f head)::(map f tail)
```

This function applies its first argument to a list of values producing a list of the results. For example:

```
map even [1, 3, 5, 2, 4, 6] = [false, false, false, true, true, true]
map min [(1,5), (4,2), (8,1)] = [1, 2, 1]
```

This is practically impossible to achieve in imperative languages; at most we could write a subprogram that receives a pointer to a function as an argument, but we would require different subprograms for each possible argument signature of the function argument.

Note that the construction is safe. The type of map is:

```
map: ('t1 -> 't2) -> 't1 list -> 't2 list
```

meaning that the elements of the argument list 't1 list must all be compatible with the argument of function 't1, and the result list 't2 list will consist only of elements of the function result type 't2.

Higher-order functions abstract away most of the control structures that we find essential in imperative languages. To give another example, the function accumulate compounds the function application instead of creating a list of results like map:

```
fun    accumulate f initial [] = initial
|      accumulate f initial head::tail = accumulate f (f initial head) tail
```

accumulate can be used to create a variety of useful functions. The functions

```
fun minlist = accumulate min maxint
fun sumlist = accumulate "+" 0
```

compute the minimum of an integer list and the sum of an integer list, respectively. For example:

```
minlist [3, 1, 2] =
accumulate min maxint [3, 1, 2] =
accumulate min (min maxint 3) [1, 2] =
accumulate min 3 [1, 2] =
accumulate min (min 3 1) [2] =
accumulate min 1 [2] =
accumulate min (min 1 2) [] =
accumulate min 1 [] =
1
```

Higher-order functions are not limited to lists; you can write functions that traverse trees and apply a function at each node. Furthermore, functions can be defined on type variables so that they can be used without change when you define new data types.

16.5 Lazy and eager evaluation

In imperative languages, we always assume that the actual parameters will be evaluated before calling the function:

C

```
n = min(j+k, (i+4)/m);
```

The technical term for this is *eager evaluation*. However, eager evaluation has its own problems which we encountered in if-statements (Section 6.2), and a special construct for short-circuit evaluation had to be defined:

Ada

```
if (N > 0) and then ((Sum / N) > M) then ...
```

How should a conditional expression:

```
if c then e1 else e2
```

be defined in a functional programming language? Under eager evaluation we would evaluate c, e1 and e2 and only then perform the conditional operation. Of course, this is not acceptable; the following expression fails if eager evaluation is used, since it is an error to take the head of an empty list:

```
if list = [] then [] else hd list
```

To solve this problem, ML has a special rule for evaluation of the if-function: the condition c is evaluated first, and only then is one of the two branches evaluated.

The situation would be much simpler if *lazy evaluation* were used, where an argument is evaluated only if it is needed, and only to the extent needed.[4] For example, we could define if as an ordinary function:

```
fun    if true x y = x
|      if false x y = y
```

When if is applied, the function is simply applied to its first argument, producing:

```
(if list=[]  []  hd list)  [] =
if []=[]  []  hd [] =
if true  []  hd [] =
[]
```

[4]For this reason lazy evaluation is also known as *call-by-need*.

and we don't attempt to evaluate hd [].

Lazy evaluation is similar to the *call-by-name* parameter passing mechanism in imperative languages, where the actual parameter is evaluated anew each time the formal parameter is used. The mechanism is problematic in imperative languages, because the possibility of side-effects makes it impossible to optimize by computing and storing an evaluation for reuse. In side-effect-free functional programming there is no problem, and languages using lazy evaluation (such as Miranda[5]) have been implemented. Lazy evaluation can be less efficient than eager evaluation, but it has significant advantages.

The main attraction of lazy evaluation is that it is possible to do incremental evaluation which can be used to program efficient algorithms. For example, consider a tree of integer values whose type we defined above. You may wish to program an algorithm which compares two trees to see if they have the same set of values under some ordering of the nodes. This can be written as follows:

```
fun   equal_nodes t1 t2 = compare_lists (tree_to_list t1) (tree_to_list t2)
```

The function tree_to_list traverses the tree and creates a list of the values in the nodes; compare_lists checks if two lists are equal. Under eager evaluation, both trees are completely transformed into lists before the comparison is done, even if the first nodes in the traversal are unequal! Under lazy evaluation, it is only necessary to evaluate the functions to the extent necessary to continue the computation.

The functions compare_lists and tree_to_list are defined as follows:[6]

```
fun   compare_lists [] [] = true
|     compare_lists head :: tail1 head :: tail2 = compare_lists tail1 tail2
|     compare_lists list1 list2 = false

fun   tree_to_list Empty = []
|     tree_to_list T(left, value, right) =
          value :: append (tree_to_list left) (tree_to_list right)
```

An example of lazy evaluation would proceed as follows (where we have abbreviated the functions as cmp and ttl, and the ellipsis indicates a very large subtree):

```
cmp   ttl T( T(Empty,4,Empty), 5, ... )
      ttl T( T(Empty,6,Empty), 5, ... ) =
cmp   5 :: append (ttl T(Empty,4,Empty)) (ttl ... )
      5 :: append (ttl T(Empty,6,Empty)) (ttl ... ) =
cmp   append (ttl T(Empty,4,Empty)) (ttl ... )
      append (ttl T(Empty,6,Empty)) (ttl ... ) =
      ...
cmp   4 :: append [] (ttl ... )
```

[5]Miranda is a trademark of Research Software Ltd.

[6]This traversal is called *preorder* because the root of a subtree is listed before the nodes in the subtree.

```
        6 :: append [] (ttl ...) =
false
```

By evaluating the arguments only as needed, the unnecessary traversal of the right-hand subtree is completely avoided. Programming tricks are needed to achieve the same effect in a language like ML that uses eager evaluation.

An additional advantage of lazy evaluation is that it lends itself to interactive and systems programming. The input, say from a terminal, is simply considered to be a potentially infinite list of values. The lazy evaluator never evaluates the entire list, of course; instead, whenever a value is needed, the head of the list is removed after prompting the user to supply a value.

16.6 Exceptions

An evaluation of an expression in ML can raise an exception. There are predefined exceptions, mainly for exceptions that are raised when computing with predefined types, such as division by zero or taking the head of an empty list. The programmer can also declare exceptions which may optionally have parameters:

```
exception BadParameter of int;
```

The exception can then be raised and handled:

```
fun only_positive n =
    if n <= 0 then raise BadParameter n
    else ...

val i = ...;
val j = only_positive i
    handle
        BadParameter 0 => 1;
        BadParameter n => abs n;
```

The function only_positive raises the exception BadParameter if the parameter is not positive. When the function is called, an exception handler is attached to the calling expression, specifying the value to be returned if the function raises the exception. This value can be used for further computation at the point where the exception was raised; in this case it is only used as the value returned from the function.

16.7 Environments

In addition to defining functions and evaluating expressions, an ML program can contain declarations:

```
val i = 20
val s = "Hello world"
```

Thus ML has a *store*, but unlike imperative languages this store is not updatable; in the example, it is not possible to "assign" a different value to i or s. If we now execute:

```
val i = 35
```

a new named value would be created, hiding the old value but not replacing the contents of i with a new value. ML declarations are similar to const declarations in C in that an object is created but cannot be modified; however, an ML redeclaration hides the previous one while in C it is illegal to redeclare an object in the same scope.

Block structuring is possible by making declarations local to a definition or an expression. The syntax for localization within an expression is shown in the following example which computes the roots of a quadratic equation using a local declaration for the discriminant:

```
val a = 1.0 and b = 2.0 and c = 1.0
let
    D = b*b – 4.0*a*c
in
    ( (–b+D)/2.0*a, (–b–D)/2.0*a )
end
```

Each declaration *binds* a value to a name. The set of all bindings in force at any time is called an *environment*, and we say that an expression is evaluated in the context of an environment. We actually discussed environments at length in the context of scope and visibility in imperative languages; the difference is that bindings cannot be modified in a functional programming environment.

It is an easy extension to include abstract data types within an environment. This is done by attaching a set of functions to a type declaration:

```
abstype int tree =
        Empty
|       T of (int tree × int × int tree)
with
    fun sumtree t = ...
    fun equal_nodes t1 t2 = ...
end
```

The meaning of this declaration is that only the listed functions have access to constructors of the abstract type, similar to a private type in Ada or a C++ class with private components. Furthermore, the abstract type may be parameterized with a type variable:

```
abstype 't tree = ...
```

which is similar to creating generic abstract data types in Ada.

ML includes a very flexible system for defining and manipulating modules. The basic concept is the *structure* which encapsulates an environment consisting of declarations (types and functions), in a way that is similar to a C++ class or an Ada package that defines an abstract data type. However, in ML a structure is itself an object that has a type called a *signature*. Structures can be manipulated using *functors* which are functions that map one structure to another. This is a generalization of the generic concept which maps package or class templates to concrete types. Functors can be used to hide or share information in structures. The details of these concepts are beyond the scope of the book and the interested reader is referred to textbooks on ML.

Functional programming languages can be used to write concise, reliable programs for applications that deal with complex data structures and algorithms. Even if the efficiency of the functional program is not acceptable, it can still be used as a prototype or as a working specification of the final program.

16.8 Exercises

1. What is the type of the curried function min_c?

   ```
   fun min_c x y = if x < y then x else y
   ```

2. Infer the types of sumtree and mintree.

3. Write out the definition of general_insert_element in words.

4. Write a function for appending lists, and then show how the same function can be defined using accumulate.

5. Write a function that takes a list of trees and returns a list of the minimum values in each tree.

6. Infer the types of compare_lists and tree_to_list.

7. What does the following program do? What is the type of the function?

   ```
   fun    filter f [] = []
   |      filter f h::t = h::(filter f t),        if f h = true
   |      filter f h::t = filter f t,             otherwise
   ```

8. The standard deviation of a sequence of numbers $(x_1, \ldots, x_n)$ is defined as the square root of the averages of the squares of the numbers minus the square of the average. Write an ML program that computes the standard deviation of a list of numbers. Hint: use map and accumulate.

9. Write an ML program to test for perfect numbers; $n > 2$ is a perfect number if the factors of n (not including n) add up to n. For example, $1+2+4+7+14 = 28$. The outline of the program is as follows:

```
fun isperfect n =
    let fun addfactors ...
in addfactors(n div 2) = n end;
```

10. Compare exceptions in ML with exceptions in Ada, C++ and Eiffel.