Chapter 4

Functions

4.1 Functions as Templates

So far we just have the means to calculate the values of expressions, and to bind these values to variables for future reference. In this chapter we will introduce the ability to *abstract* the data from a calculation, leaving behind the bare *pattern* of the calculation. This pattern may then be *instantiated* as often as you like so that the calculation may be repeated with specified data values plugged in.

For example, consider the expression `2*(3+4)`. The data might be taken to be the values `2`, `3`, and `4`, leaving behind the pattern □ `*` (□ `+` □), with "holes" where the data used to be. We might equally well take the data to just be `2` and `3`, and leave behind the pattern □ `*` (□ `+ 4`). Or we might even regard `*` and `+` as the data, leaving `2` □ (`3` □ `4`) as the pattern! What is important is that a complete expression can be recovered by filling in the holes with chosen data.

Since a pattern can contain many different holes that can be independently instantiated, it is necessary to give *names* to the holes so that instantiation consists of plugging in a given value for all occurrences of a name in an expression. These names are, of course, just variables, and instantiation is just the process of substituting a value for all occurrences of a variable in a given expression. A pattern may therefore be viewed as a *function* of the variables that occur within it; the pattern is instantiated by *applying* the function to argument values.

This view of functions is similar to our experience from high school

algebra. In algebra we manipulate polynomials such as $x^2 + 2x + 1$ as a form of expression denoting a real number, with the variable x representing a fixed, but unknown, quantity. (Indeed, variables in algebra are sometimes called *unknowns*, or *indeterminates*.) It is also possible to think of a polynomial as a function on the real line: given a real number x, a polynomial determines a real number y computed by the given combination of arithmetic operations. Indeed, we sometimes write equations such as $f(x) = x^2 + 2x + 1$, to stand for the function f determined by the polynomial. In the univariate case we can get away with just writing the polynomial for the function, but in the multivariate case we must be more careful: we may regard the polynomial $x^2 + 2xy + y^2$ to be a function of x, a function of y, or a function of both x and y. In these cases we write $f(x) = x^2 + 2xy + y^2$ when x varies and y is held fixed, and $g(y) = x^2 + 2xy + y^2$ when y varies for fixed x, and $h(x, y) = x^2 + 2xy + y^2$, when both vary jointly.

In algebra it is usually left implicit that the variables x and y range over the real numbers, and that f, g, and h are functions on the real line. However, to be fully explicit, we sometimes write something like

$$f : \mathbb{R} \to \mathbb{R} \ : \ x \in R \mapsto x^2 + 2x + 1$$

to indicate that f is a function on the reals sending $x \in \mathbb{R}$ to $x^2 + 2x + 1 \in \mathbb{R}$. This notation has the virtue of separating the *name* of the function, f, from the function itself, the mapping that sends $x \in \mathbb{R}$ to $x^2 + 2x + 1$. It also emphasizes that functions are a kind of "value" in mathematics (namely, a certain set of ordered pairs), and that the variable f is bound to that value (*i.e.*, that set) by the declaration. This viewpoint is especially important once we consider operators, such as the differential operator, that map functions to functions. For example, if f is a differentiable function on the real line, the function $D\,f$ is its first derivative, another function on the real line. In the case of the function f defined above the function $D\,f$ sends $x \in \mathbb{R}$ to $2x + 2$.

4.2 Functions and Application

The treatment of functions in ML is very similar, except that we stress the *algorithmic* aspects of functions (*how* they determine values from arguments), as well as the *extensional* aspects (*what* they compute). As in

mathematics, a function in ML is a kind of value, namely a value of *function type* of the form *typ* `->` *typ′*. The type *typ* is the *domain type* (the type of arguments) of the function, and *typ′* is its *range type* (the type of its results). We compute with a function by *applying* it to an *argument* value of its domain type and calculating the *result*, a value of its range type. Function application is indicated by juxtaposition: we simply write the argument next to the function.

The values of function type consist of *primitive functions*, such as addition and square root, and *function expressions*, which are also called *lambda expressions*,[1] of the form

`fn` *var* `:` *typ* `=>` *exp*

The variable *var* is called the *parameter*, and the expression *exp* is called the *body*. It has type *typ*`->`*typ′* provided that *exp* has type *typ′* under the assumption that the parameter *var* has the type *typ*.

To apply such a function expression to an argument value *val*, we add the binding

`val` *var* `=` *val*

to the value environment, and evaluate *exp*, obtaining a value *val′*. Then the value binding for the parameter is removed, and the result value, *val′*, is returned as the value of the application.

For example, `Math.sqrt` is a primitive function of type `real->real` that may be applied to a real number to obtain its square root. For example, the expression `Math.sqrt 2.0` evaluates to `1.414` (approximately). We can, if we wish, parenthesize the argument, writing `Math.sqrt (2.0)` for the sake of clarity; this is especially useful for expressions such as `Math.sqrt (Math.sqrt 2.0)`. The square root function is built in. We may write the fourth root function as the following function expression:

```
fn x : real => Math.sqrt (Math.sqrt x)
```

It may be applied to an argument by writing an expression such as

```
(fn x : real => Math.sqrt (Math.sqrt x)) (16.0),
```

[1] For purely historical reasons.

which calculates the fourth root of `16.0`. The calculation proceeds by binding the variable `x` to the argument `16.0`, then evaluating the expression `Math.sqrt (Math.sqrt x)` in the presence of this binding. When evaluation completes, we drop the binding of `x` from the environment, since it is no longer needed.

Notice that we did not give the fourth root function a name; it is an "anonymous" function. We may give it a name using the declaration forms introduced in chapter 3. For example, we may bind the fourth root function to the variable `fourthroot` using the following declaration:

```
val fourthroot : real -> real =
    fn x : real => Math.sqrt (Math.sqrt x)
```

We may then write `fourthroot 16.0` to compute the fourth root of `16.0`.

This notation for defining functions quickly becomes tiresome, so ML provides a special syntax for function bindings that is more concise and natural. Instead of using the `val` binding above to define `fourthroot`, we may instead write

```
fun fourthroot (x:real):real = Math.sqrt (Math.sqrt x)
```

This declaration has the same meaning as the earlier `val` binding, namely it binds `fn x:real => Math.sqrt(Math.sqrt x)` to the variable `fourthroot`.

It is important to note that function applications in ML are evaluated according to the *call-by-value* rule: the arguments to a function are evaluated before the function is called. Put in other terms, functions are defined to act on values, rather than on unevaluated expressions. Thus, to evaluate an expression such as `fourthroot (2.0+2.0)`, we proceed as follows:

1. Evaluate `fourthroot` to the function value `fn x : real => Math.sqrt (Math.sqrt x)`.
2. Evaluate the argument `2.0+2.0` to its value `4.0`
3. Bind `x` to the value `4.0`.
4. Evaluate `Math.sqrt (Math.sqrt x)` to `1.414` (approximately).
    (a) Evaluate `Math.sqrt` to a function value (the primitive square root function).

(b) Evaluate the argument expression `Math.sqrt x` to its value, approximately `2.0`.

i. Evaluate `Math.sqrt` to a function value (the primitive square root function).

ii. Evaluate `x` to its value, `4.0`.

iii. Compute the square root of `4.0`, yielding `2.0`.

(c) Compute the square root of `2.0`, yielding `1.414`.

5. Drop the binding for the variable `x`.

Notice that we evaluate *both* the function and argument positions of an application expression — both the function and argument are expressions yielding values of the appropriate type. The value of the function position must be a value of function type, either a primitive function or a lambda expression, and the value of the argument position must be a value of the domain type of the function. In this case the result value (if any) will be of the range type of the function. Functions in ML are *first-class*, meaning that they may be computed as the value of an expression. We are not limited to applying only named functions, but rather may compute "new" functions on the fly and apply these to arguments. This is a source of considerable expressive power, as we shall see in the sequel.

Using similar techniques we may define functions with arbitrary domain and range. For example, the following are all valid function declarations:

```
fun srev (s:string):string = implode (rev (explode s))
fun pal (s:string):string = s ^ (srev s)
fun double (n:int):int = n + n
fun square (n:int):int = n * n
fun halve (n:int):int = n div 2
fun is_even (n:int):bool = (n mod 2 = 0)
```

Thus `pal "ot"` evaluates to the string `"otto"`, and `is_even 4` evaluates to `true`.

4.3 Binding and Scope, Revisited

A function expression of the form

`fn` *var*`:`*typ* `=>` *exp*

binds the variable *var* within the body *exp* of the function. Unlike `val` bindings, function expressions bind a variable without giving it a specific value. The value of the parameter is only determined when the function is applied, and then only temporarily, for the duration of the evaluation of its body.

It is worth reviewing the rules for binding and scope of variables that we introduced in chapter 3 in the presence of function expressions. As before we adhere to the principle of *static scope*, according to which variables are taken to refer to the *nearest enclosing binding* of that variable, whether by a `val` binding or by a `fn` expression.

Thus, in the following example, the occurrences of `x` in the body of the function `f` refer to the parameter of `f`, whereas the occurrences of `x` in the body of `g` refer to the preceding `val` binding.

```
val x:real = 2.0
fun f(x:real):real = x+x
fun g(y:real):real = x+y
```

Local `val` bindings may shadow parameters, as well as other `val` bindings. For example, consider the following function declaration:

```
fun h(x:real):real =
    let val x:real = 2.0 in x+x end * x
```

The inner binding of `x` by the `val` declaration shadows the parameter `x` of `h`, but *only* within the body of the `let` expression. Thus the last occurrence of `x` refers to the parameter of `h`, whereas the preceding two occurrences refer to the inner binding of `x` to `2.0`.

The phrases "inner" and "outer" binding refer to the *logical structure*, or *abstract syntax* of an expression. In the preceding example, the body of `h` lies "within" the scope of the parameter `x`, and the expression `x+x` lies within the scope of the `val` binding for `x`. Since the occurrences of `x` within the body of the `let` lie within the scope of the inner `val` binding, they are taken to refer to that binding, rather than to the parameter. On the other hand the last occurrence of `x` does not lie within the scope of the `val` binding, and hence refers to the parameter of `h`.

In general the names of parameters do not matter; we can rename them at will without affecting the meaning of the program, provided that we

simultaneously (and consistently) rename the binding occurrence and all uses of that variable. Thus the functions `f` and `g` below are completely equivalent to each other:

```
fun f(x:int):int = x*x
fun g(y:int):int = y*y
```

A parameter is just a placeholder; its name is not important.

Our ability to rename parameters is constrained by the static scoping rule. We may rename a parameter to whatever we'd like, provided that we don't change the way in which uses of a variable are resolved. For example, consider the following situation:

```
val x:real = 2.0
fun h(y:real):real = x+y
```

The parameter `y` to `h` may be renamed to `z` without affecting its meaning. However, we may *not* rename it to `x`, for doing so changes its meaning! That is, the function

```
fun h'(x:real):real = x+x
```

does *not* have the same meaning as `h`, because now both occurrences of `x` in the body of `h'` refer to the parameter, whereas in `h` the variable `x` refers to the outer `val` binding, whereas the variable `y` refers to the parameter.

While this may seem like a minor technical issue, it is essential that you master these concepts now, for they play a central, and rather subtle, role later on.

4.4 Sample Code

Here is the complete code for this chapter.

Chapter 5

Products and Records

5.1 Product Types

A distinguishing feature of ML is that aggregate data structures, such as tuples, lists, arrays, or trees, may be created and manipulated with ease. In contrast to most familiar languages it is not necessary in ML to be concerned with allocation and deallocation of data structures, nor with any particular representation strategy involving, say, pointers or address arithmetic. Instead we may think of data structures as first-class values, on a par with every other value in the language. Just as it is unnecessary to think about "allocating" integers to evaluate an arithmetic expression, it is unnecessary to think about allocating more complex data structures such as tuples or lists.

5.1.1 Tuples

This chapter is concerned with the simplest form of aggregate data structure, the *n-tuple*. An n-tuple is a finite ordered sequence of values of the form

$(val_1, \ldots, val_n)$,

where each val_i is a value. A 2-tuple is usually called a *pair*, a 3-tuple a *triple*, and so on.

An n-tuple is a value of a *product type* of the form

$typ_1 * \ldots * typ_n$.

Values of this type are n-tuples of the form

$(val_1, \ldots, val_n)$,

where val_i is a value of type typ_i (for each $1 \leq i \leq n$).

Thus the following are well-formed bindings:

```
val pair : int * int = (2, 3)
val triple : int * real * string = (2, 2.0, "2")
val quadruple
    : int * int * real * real
    = (2,3,2.0,3.0)
val pair_of_pairs
    : (int * int) * (real * real)
    = ((2,3),(2.0,3.0))
```

The nesting of parentheses matters! A pair of pairs is not the same as a quadruple, so the last two bindings are of distinct values with distinct types.

There are two limiting cases, $n = 0$ and $n = 1$, that deserve special attention. A 0-tuple, which is also known as a *null tuple*, is the empty sequence of values, `()`. It is a value of type `unit`, which may be thought of as the 0-tuple type.[1] The null tuple type is surprisingly useful, especially when programming with effects. On the other hand there seems to be no particular use for 1-tuples, and so they are absent from the language.

As a convenience, ML also provides a general *tuple expression* of the form

$(exp_1, \ldots, exp_n)$,

where each exp_i is an arbitrary expression, not necessarily a value. Tuple expressions are evaluated from left to right, so that the above tuple expression evaluates to the tuple value yielding the tuple value

$(val_1, \ldots, val_n)$,

provided that exp_1 evaluates to val_1, exp_2 evaluates to val_2, and so on. For example, the binding

[1] In Java (and other languages) the type `unit` is misleadingly written `void`, which suggests that the type has *no* members, but in fact it has exactly one!

```
val pair : int * int = (1+1, 5-2)
```

binds the value `(2, 3)` to the variable `pair`.

Strictly speaking, it is not essential to have tuple expressions as a primitive notion in the language. Rather than write

$(exp_1, \ldots, exp_n)$,

with the (implicit) understanding that the exp_i's are evaluated from left to right, we may instead write

```
let val x1 = exp1
    val x2 = exp2
    ⋮
    val xn = expn
in  (x1,...,xn) end
```

which makes the evaluation order explicit.

5.1.2 Tuple Patterns

One of the most powerful, and distinctive, features of ML is the use of *pattern matching* to access components of aggregate data structures. For example, suppose that *val* is a value of type

```
(int * string) * (real * char)
```

and we wish to retrieve the first component of the second component of *val*, a value of type `real`. Rather than explicitly "navigate" to this position to retrieve it, we may simply use a generalized form of value binding in which we select that component using a pattern:

```
val ((_, _), (r:real, _)) = val
```

The left-hand side of the `val` binding is a tuple pattern that describes a pair of pairs, binding the first component of the second component to the variable `r`. The underscores indicate "don't care" positions in the pattern — their values are not bound to any variable. If we wish to give names to all of the components, we may use the following value binding:

```
val ((i:int, s:string), (r:real, c:char)) = val
```

If we'd like we can even give names to the first and second components of the pair, without decomposing them into constituent parts:

```
val (is:int*string,rc:real*char) = val
```

The general form of a value binding is

`val` *pat* `=` *exp*,

where *pat* is a *pattern* and *exp* is an expression. A pattern is one of three forms:

1. A *variable pattern* of the form *var*:*typ*.
2. A *tuple pattern* of the form $(pat_1, \ldots, pat_n)$, where each pat_i is a pattern. This includes as a special case the null-tuple pattern, `()`.
3. A *wildcard pattern* of the form `_`.

The type of a pattern is determined by an inductive analysis of the form of the pattern:

1. A variable pattern *var*:*typ* is of type *typ*.
2. A tuple pattern $(pat_1, \ldots, pat_n)$ has type $typ_1 * \cdots * typ_n$, where each pat_i is a pattern of type typ_i. The null-tuple pattern `()` has type `unit`.
3. The wildcard pattern `_` has any type whatsoever.

A value binding of the form

`val` *pat* `=` *exp*

is well-typed iff *pat* and *exp* have the same type; otherwise the binding is ill-typed and is rejected.

For example, the following bindings are well-typed:

```
val (m:int, n:int) = (7+1,4 div 2)
val (m:int, r:real, s:string) = (7, 7.0, "7")
val ((m:int,n:int), (r:real, s:real)) = ((4,5),(3.1,2.7))
val (m:int, n:int, r:real, s:real) = (4,5,3.1,2.7)
```

In contrast, the following are ill-typed:

```
val (m:int,n:int,r:real,s:real) = ((4,5),(3.1,2.7))
val (m:int, r:real) = (7+1,4 div 2)
val (m:int, r:real) = (7, 7.0, "7")
```

Value bindings are evaluated using the *bind-by-value* principle discussed earlier, except that the binding process is now more complex than before. First, we evaluate the right-hand side of the binding to a value (if indeed it has one). This happens regardless of the form of the pattern — the right-hand side is *always* evaluated. Second, we perform *pattern matching* to determine the bindings for the variables in the pattern.

The process of matching a value against a pattern is defined by a set of rules for reducing bindings with complex patterns to a set of bindings with simpler patterns, stopping once we reach a binding with a variable pattern. The rules are as follows:

1. The variable binding `val` *var* `=` *val* is irreducible.

2. The wildcard binding `val _ =` *val* is discarded.

3. The tuple binding

   `val (`pat_1`,...,`pat_n`) =`
   `(`val_1`,...,`val_n`)`

   is reduced to the set of n bindings

   `val` pat_1 `=` val_1

   ⋮

   `val` pat_n `=` val_n

   In the case that $n = 0$ the tuple binding is simply discarded.

These simplifications are repeated until all bindings are irreducible, which leaves us with a set of variable bindings that constitute the result of pattern matching.

For example, evaluation of the binding

```
val ((m:int,n:int), (r:real, s:real)) = ((2,3),(2.0,3.0))
```

proceeds as follows. First, we compose this binding into the following two bindings:

```
val (m:int, n:int) = (2,3)
and (r:real, s:real) = (2.0,3.0).
```

Then we decompose each of these bindings in turn, resulting in the following set of four atomic bindings:

```
val m:int = 2
and n:int = 3
and r:real = 2.0
and s:real = 3.0
```

At this point the pattern-matching process is complete.

5.2 Record Types

Tuples are most useful when the number of positions is small. When the number of components grows beyond a small number, it becomes difficult to remember which position plays which role. In that case it is more natural to attach a *label* to each component of the tuple that mediates access to it. This is the notion of a *record type*.

A record type has the form

$\{lab_1 : typ_1, \ldots, lab_n : typ_n\}$,

where $n \geq 0$, and all of the labels lab_i are distinct. A *record value* has the form

$\{lab_1 = val_1, \ldots, lab_n = val_n\}$,

where val_i has type typ_i. A *record pattern* has the form

$\{lab_1 = pat_1, \ldots, lab_n = pat_n\}$

which has type

$\{lab_1 : typ_1, \ldots, lab_n : typ_n\}$

provided that each pat_i has type typ_i.

A record value binding of the form

```
val
 {lab1=pat1, ..., labn=patn} =
 {lab1=val1, ..., labn=valn}
```

is decomposed into the following set of bindings

```
val pat1 = val1
and ...
and patn = valn.
```

Since the components of a record are identified by name, not position, the order in which they occur in a record value or record pattern is not important. However, in a record *expression* (in which the components may not be fully evaluated), the fields are evaluated from left to right in the order written, just as for tuple expressions.

Here are some examples to help clarify the use of record types. First, let us define the record type `hyperlink` as follows:

```
type hyperlink =
     { protocol : string,
       address : string,
       display : string }
```

The record binding

```
val mailto_rwh : hyperlink =
    { protocol="mailto",
      address="rwh@cs.cmu.edu",
      display="Robert Harper" }
```

defines a variable of type `hyperlink`. The record binding

```
val { protocol=prot, display=disp, address=addr } = mailto_rwh
```

decomposes into the three variable bindings

```
val prot = "mailto"
val addr = "rwh@cs.cmu.edu"
val disp = "Robert Harper"
```

which extract the values of the fields of `mailto_rwh`.

Using wild cards we can extract selected fields from a record. For example, we may write

```
val {protocol=prot, address=_, display=_} = mailto_rwh
```

to bind the variable `prot` to the `protocol` field of the record value `mailto_rwh`.

It is quite common to encounter record types with tens of fields. In such cases even the wild card notation doesn't help much when it comes to selecting one or two fields from such a record. For this we often use *ellipsis patterns* in records, as illustrated by the following example.

```
val {protocol=prot,...} = intro_home
```

The pattern `{protocol=prot,...}` stands for the expanded pattern

```
{protocol=prot, address=_, display=_}
```

in which the elided fields are implicitly bound to wildcard patterns.

In general the ellipsis is replaced by as many wildcard bindings as are necessary to fill out the pattern to be consistent with its type. In order for this to occur *the compiler must be able to determine unambiguously the type of the record pattern*. Here the right-hand side of the value binding determines the type of the pattern, which then determines which additional fields to fill in. In some situations the context does not disambiguate, in which case you must supply additional type information, or avoid the use of ellipsis notation.

Finally, ML provides a convenient abbreviated form of record pattern

$\{lab_1, \ldots, lab_n\}$

which stands for the pattern

$\{lab_1{=}var_1, \ldots, lab_n{=}var_n\}$

where the variables var_i are variables with the same name as the corresponding label lab_i. For example, the binding

```
val { protocol, address, display } = mailto_rwh
```

decomposes into the sequence of atomic bindings

```
val protocol = "mailto"
val address = "rwh@cs.cmu.edu"
val display = "Robert Harper"
```

This avoids the need to think up a variable name for each field; we can just make the label do "double duty" as a variable.

5.3 Multiple Arguments and Multiple Results

A function may bind more than one argument by using a pattern, rather than a variable, in the argument position. Function expressions are generalized to have the form

`fn` *pat* `=>` *exp*

where *pat* is a pattern and *exp* is an expression. Application of such a function proceeds much as before, except that the argument value is matched against the parameter pattern to determine the bindings of zero or more variables, which are then used during the evaluation of the body of the function.

For example, we may make the following definition of the Euclidean distance function:

```
val dist
    : real * real -> real
    = fn (x:real, y:real) => sqrt (x*x + y*y)
```

This function may then be applied to a pair (a two-tuple!) of arguments to yield the distance between them. For example, `dist (2.0,3.0)` evaluates to (approximately) `4.0`.

Using `fun` notation, the distance function may be defined more concisely as follows:

```
fun dist (x:real, y:real):real = sqrt (x*x + y*y)
```

The meaning is the same as the more verbose `val` binding given earlier.

Keyword parameter passing is supported through the use of record patterns. For example, we may define the distance function using keyword parameters as follows:

```
fun dist' {x=x:real, y=y:real} = sqrt (x*x + y*y)
```

The expression `dist' {x=2.0,y=3.0}` invokes this function with the indicated x and y values.

Functions with multiple results may be thought of as functions yielding tuples (or records). For example, we might compute two different notions of distance between two points at once as follows:

```
fun dist2 (x:real, y:real):real*real
    = (sqrt (x*x+y*y), abs(x-y))
```

Notice that the result type is a pair, which may be thought of as two results.

These examples illustrate a pleasing regularity in the design of ML. Rather than introduce *ad hoc* notions such as multiple arguments, multiple results, or keyword parameters, we make use of the general mechanisms of tuples, records, and pattern matching.

It is sometimes useful to have a function to select a particular component from a tuple or record (*e.g.*, the third component or the component with a given label). Such functions may be easily defined using pattern matching. But since they arise so frequently, they are pre-defined in ML using *sharp notation*. For any tuple type

$typ_1 * \cdots * typ_n$,

and each $1 \leq i \leq n$, there is a function `#`i of type

$typ_1 * \cdots * typ_n$`->`typ_i

defined as follows:

```
fun #i (_, ..., _, x, _, ..., _) = x
```

where `x` occurs in the ith position of the tuple (and there are underscores in the other $n-1$ positions).

Thus we may refer to the second field of a three-tuple *val* by writing `#2(`*val*`)`. It is bad style, however, to over-use the sharp notation; code is generally clearer and easier to maintain if you use patterns wherever possible. Compare, for example, the following definition of the Euclidean distance function written using sharp notation with the original.

```
fun dist (p:real*real):real
    = sqrt((#1 p)*(#1 p)+(#2 p)*(#2 p))
```

You can easily see that this gets out of hand very quickly, leading to unreadable code. *Use of the sharp notation is strongly discouraged!*

A similar notation is provided for record field selection. The following function `#lab` selects the component of a record with label lab.

```
fun #lab {lab=x,...} = x
```

Notice the use of ellipsis! Bear in mind the disambiguation requirement: any use of `#lab` must be in a context sufficient to determine the full record type of its argument.

5.4 Sample Code

Here is the complete code for this chapter.