Chapter 5

A taste of ML

In the last few chapters, we have started with pure lambda calculus and systematically extended it with new primitive features. For example we added the 'let' construct as primitive for the sake of making the polymorphic typing more useful, and a recursion operation in order to restore the computational power that was lost when adding types. By travelling further along this path, we will eventually arrive at ML, although it's still valuable to retain the simple worldview of typed lambda calculus.

The next stage is to abandon the encodings of datatypes like booleans and natural numbers via lambda terms, and instead make them primitive. That is, we have new primitive types like `bool` and `int` (actually positive and negative integers) and new type constructors like $\times$, a step we have in many ways anticipated in the last chapter. Associated with these are new constants and new conversion rules. For example, the expression $2 + 2$ is evaluated using machine arithmetic, rather than by translation into Church numerals and performing β-conversions. These additional conversions, regarded as augmenting the usual lambda operations, are often referred to as 'δ-conversions'. We will see in due course how the ML language is extended in other ways in comparison with pure lambda calculus. First, we must address the fundamental question of ML's evaluation strategy.

5.1 Eager evaluation

We have said that from a theoretical point of view, normal order (top down left-to-right) reduction of expressions is to be preferred, since if any strategy terminates, this one will.[1] However it has some practical defects. For example, consider the following:

$$(\lambda x.\ x + x + x)\ (10 + 5)$$

[1]This strategy is familiar from a few traditional languages like Algol 60 where it is referred to as *call by name*.

A normal order reduction results in $(10 + 5) + (10 + 5) + (10 + 5)$, and the following reduction steps need to evaluate three separate instances of the same expression. This is unacceptable in practice. There are two main solutions to this problem, and these solutions divide the world of functional programming languages into two camps.

The first alternative is to stick with normal order reduction, but attempt to optimize the implementation so multiple subexpressions arising in this way are shared, and never evaluated more than once. Internally, expressions are represented as directed acyclic graphs rather than as trees. This is known as *lazy* or *call-by-need* evaluation, since expressions are only evaluated when absolutely necessary.

The second approach is to turn the theoretical considerations about reduction strategy on their head and evaluate arguments to functions before passing the values to the function. This is known as *applicative order* or *eager* evaluation. The latter name arises because function arguments are evaluated even when they may be unnecessary, e.g. t in $(\lambda x.\ y)\ t$. Of course, eager evaluation means that some expressions may loop that would terminate in a lazy regime. But this is considered acceptable, since these instances are generally easy to avoid in practice. In any case, an eager evaluation strategy is the standard in many programming languages like C, where it is referred to as *call by value*.

ML adopts eager evaluation, for two main reasons. Choreographing the reductions and sharings that occur in lazy evaluation is quite tricky, and implementations tend to be relatively inefficient and complicated. Unless the programmer is very careful, memory can fill up with pending unevaluated expressions, and in general it is hard to understand the space behaviour of programs. In fact many implementations of lazy evaluation try to optimize it to eager evaluation in cases where there is no semantic difference.[2] By contrast, in ML, we always first evaluate the arguments to functions and only then perform the β-reduction — this is simple and efficient, and is easy to implement using standard compiler technology.

The second reason for preferring applicative order evaluation is that ML is not a *pure* functional language, but includes imperative features (variables, assignments etc.). Therefore the order of evaluation of subexpressions can make a *semantic* difference, rather than merely affecting efficiency. If lazy evaluation is used, it seems to become practically impossible for the programmer to visualize, in a nontrivial program, exactly when each subexpression gets evaluated. In the eager ML system, one just needs to remember the simple evaluation rule.

It is important to realize, however, that the ML evaluation strategy is *not* simply bottom-up, the exact opposite of normal order reduction. In fact, ML *never evaluates underneath a lambda*. (In particular, it never reduces η-redexes,

[2]They try to perform *strictness analysis*, a form of static analysis that can often discover that arguments are necessarily going to be evaluated (Mycroft 1981).

only β-redexes.) In evaluating $(\lambda x.\ s[x])\ t$, t is evaluated first. However $s[x]$ is not touched since it is underneath a lambda, and any subexpressions of t that happen to be in the scope of a lambda are also left alone. The precise evaluation rules are as follows:

- Constants evaluate to themselves.
- Evaluation stops immediately at λ-abstractions and does not look inside them. In particular, there is no η-conversion.
- When evaluating a combination $s\ t$, then *first* both s and t are evaluated. Then, assuming that the evaluated form of s is a lambda abstraction, a toplevel β-conversion is performed and the process is repeated.

The order of evaluation of s and t varies between versions of ML. In the version we will use, t is always evaluated first. Strictly, we should also specify a rule for `let` expressions, since as we have said they are by now accepted as a primitive. However from the point of view of evaluation they can be understood in the original way, as a lambda applied to an argument, with the understanding that the rand will be evaluated first. To make this explicit, the rule for

$$\text{let } x = s \text{ in } t$$

is that first of all s is evaluated, the resulting value is substituted for x in t, and finally the new version of t is evaluated. Let us see some examples of evaluating expressions:

$$\begin{aligned}
(\lambda x.\ (\lambda y.\ y + y)\ x)(2 + 2) &\longrightarrow (\lambda x.\ (\lambda y.\ y + y)\ x)4 \\
&\longrightarrow (\lambda y.\ y + y)4 \\
&\longrightarrow 4 + 4 \\
&\longrightarrow 8
\end{aligned}$$

Note that the subterm $(\lambda y.\ y + y)\ x$ is *not reduced*, since it is within the scope of a lambda. However, terms that are reducible and *not* enclosed by a lambda in both function and argument get reduced before the function application itself is evaluated, e.g. the second step in the following:

$$\begin{aligned}
((\lambda f\ x.\ f\ x)\ (\lambda y.\ y + y))\ (2 + 2) &\longrightarrow ((\lambda f\ x.\ f\ x)\ (\lambda y.\ y + y))\ 4 \\
&\longrightarrow (\lambda x.\ (\lambda y.\ y + y)\ x)\ 4 \\
&\longrightarrow (\lambda y.\ y + y)\ 4 \\
&\longrightarrow 4 + 4
\end{aligned}$$

The fact that ML does not evaluate under lambdas is of crucial importance to advanced ML programmers. It gives precise control over the evaluation of expressions, and can be used to mimic many of the helpful cases of lazy evaluation. We shall see a very simple instance in the next section.

5.2 Consequences of eager evaluation

The use of eager evaluation forces us to make additional features primitive, with their own special reduction procedures, rather than implementing them directly in terms of lambda calculus. In particular, we can no longer regard the conditional construct:

$$\text{if } b \text{ then } e_1 \text{ else } e_2$$

as the application of an ordinary ternary operator:

$$\text{COND } b\ e_1\ e_2$$

The reason is that in any particular instance, because of eager evaluation, we evaluate all the expressions b, e_1 and e_2 first, before evaluating the body of the COND. Usually, this is disastrous. For example, recall our definition of the factorial function:

$$\text{let rec fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

If the conditional evaluates all its arguments, then in order to evaluate fact(0), the 'else' arm must be evaluated, in its turn causing fact(PRE 0) to be evaluated. This in its turn causes the evaluation of fact(PRE (PRE 0)), and so on. Consequently the computation will loop indefinitely.

Accordingly, we make the conditional a new primitive construct, and modify the usual reduction strategy so that *first* the boolean expression is evaluated, and only then is *exactly one* of the two arms evaluated, as appropriate.

What about the process of recursion itself? We have suggested understanding recursive definitions in terms of a recursion operator *Rec* with its own reduction rule:

$$\mathit{Rec}\ f \longrightarrow f(\mathit{Rec}\ f)$$

This would also loop using an eager evaluation strategy:

$$\mathit{Rec}\ f \longrightarrow f(\mathit{Rec}\ f) \longrightarrow f(f(\mathit{Rec}\ f)) \longrightarrow f(f(f(\mathit{Rec}\ f))) \longrightarrow \cdots$$

However we only need a very simple modification to the reduction rule to make things work correctly:

$$Rec\ f \longrightarrow f(\lambda x.\ Rec\ f\ x)$$

Now the lambda on the right means that $\lambda x.\ Rec\ f\ x$ evaluates to itself, and so only after the expression has been further reduced following substitution in the body of f will evaluation continue.

5.3 The ML family

We have been talking about 'ML' as if it were a single language. In fact there are many variants of ML, even including 'Lazy ML', an implementation from Chalmers University in Sweden that is based on lazy evaluation. The most popular version of ML in education is 'Standard ML', but we will use another version, called CAML ('camel') Light.[3] We chose CAML Light for several reasons:

- The implementation is small and highly portable, so can be run effectively on Unix machines, PCs, Macs etc.
- The system is simple, syntactically and semantically, making it relatively easy to approach.
- The system is well suited to practical programming, e.g. it can easily be interfaced to C and supports standard `make`-compatible separate compilation.

However, we will be teaching quite general techniques, and any code we write can be run with a few minor syntactic changes in any version of ML, and indeed often in other functional languages.

5.4 Starting up ML

ML has already been installed on Thor. In order to use it, you need to put the CAML binary directory on your `PATH`. This can be done (assuming you use the 'bash' shell or one of its family), as follows:

```
PATH="$PATH:/home/jrh13/caml/bin"
export PATH
```

To avoid doing this every time you log on, you can insert these lines near the end of your `.bash_profile`, or the equivalent for your favourite shell. Now to use CAML in its normal interactive and interpretive mode, you simply need to type `camllight`, and the system should fire up and present its prompt ('`#`'):

[3] The name stands for 'Categorical Abstract Machine', the underlying implementation method.

```
$ camllight
>       Caml Light version 0.73

#
```

In order to exit the system, simply type `ctrl/d` or `quit();;` at the prompt. If you are interested in installing CAML Light on your own machine, you should consult the following Web page for detailed information:

```
http://pauillac.inria.fr/caml/
```

5.5 Interacting with ML

When ML presents you with its prompt, you can type in expressions, terminated by two successive semicolons, and it will evaluate them and print the result. In computing jargon, the ML system sits in a read-eval-print loop: it repeatedly reads an expression, evaluates it, and prints the result. For example, ML can be used as a simple calculator:

```
#10 + 5;;
- : int = 15
```

The system not only returns the answer, but also the *type* of the expression, which it has inferred automatically. It can do this because it knows the type of the built-in addition operator `+`. On the other hand, if an expression is not typable, the system will reject it, and try to give some idea about how the types fail to match up. In complicated cases, the error messages can be quite tricky to understand.

```
#1 + true;;
Toplevel input:
>let it = 1 + true;;
>             ^^^^
This expression has type bool,
but is used with type int.
```

Since ML is a functional language, expressions are allowed to have function type. The ML syntax for a lambda abstraction $\lambda x.\ t[x]$ is `fun x -> t[x]`. For example we can define the successor function:

```
#fun x -> x + 1;;
- : int -> int = <fun>
```

Again, the type of the expression, this time `int -> int`, is inferred and displayed. However the function itself is not printed; the system merely writes `<fun>`. This is because, in general, the internal representations of functions are not very readable.[4] Functions are applied to arguments just as in lambda calculus, by juxtaposition. For example:

```
#(fun x -> x + 1) 4;;
- : int = 5
```

Again as in lambda calculus, function application associates to the left, and you can write curried functions using the same convention of eliding multiple λ's (i.e. `fun`'s). For example, the following are all equivalent:

```
#((fun x -> (fun y -> x + y)) 1) 2;;
- : int = 3
#(fun x -> fun y -> x + y) 1 2;;
- : int = 3
#(fun x y -> x + y) 1 2;;
- : int = 3
```

5.6 Bindings and declarations

It is not convenient, of course, to evaluate the whole expression in one go; rather, we want to use `let` to bind useful subexpressions to names. This can be done as follows:

```
#let successor = fun x -> x + 1 in
 successor(successor(successor 0));;
- : int = 3
```

Function bindings may use the more elegant sugaring:

```
#let successor x = x + 1 in
successor(successor(successor 0));;
- : int = 3
```

and can be made recursive just by adding the `rec` keyword:

```
#let rec fact n = if n = 0 then 1
                  else n * fact(n - 1) in
 fact 6;;
- : int = 720
```

By using `and`, we can make several binding simultaneously, and define mutually recursive functions. For example, here are two simple, though highly inefficient, functions to decide whether or not a natural number is odd or even:

[4] CAML does not store them simply as syntax trees, but compiles them into bytecode.

```
#let rec even n = if n = 0 then true else odd (n - 1)
     and odd n  = if n = 0 then false else even (n - 1);;
even : int -> bool = <fun>
odd : int -> bool = <fun>
#even 12;;
- : bool = true
#odd 14;;
- : bool = false
```

In fact, any bindings can be performed separately from the final application. ML remembers a set of variable bindings, and the user can add to that set interactively. Simply omit the `in` and terminate the phrase with a double semicolon:

```
#let successor = fun x -> x + 1;;
successor : int -> int = <fun>
```

After this declaration, any later phrase may use the function `successor`, e.g:

```
#successor 11;;
- : int = 12
```

Note that we are not making *assignments* to *variables*. Each binding is only done once when the system analyses the input; it cannot be repeated or modified. It can be overwritten by a new definition using the same name, but this is not assignment in the usual sense, since the sequence of events is only connected with the *compilation* process, not with the dynamics of program *execution*. Indeed, apart from the more interactive feedback from the system, we could equally replace all the double semicolons after the declarations by `in` and evaluate everything at once. On this view we can see that the overwriting of a declaration really corresponds to the definition of a new local variable that hides the outer one, according to the usual lambda calculus rules. For example:

```
#let x = 1;;
x : int = 1
#let y = 2;;
y : int = 2
#let x = 3;;
x : int = 3
#x + y;;
- : int = 5
```

is the same as:

```
#let x = 1 in
 let y = 2 in
 let x = 3 in
 x + y;;
- : int = 5
```

Note carefully that, also following lambda calculus, variable binding is *static*, i.e. the first binding of `x` is still used until an inner binding occurs, and any uses of it until that point are not affected by the inner binding. For example:

```
#let x = 1;;
x : int = 1
#let f w = w + x;;
f : int -> int = <fun>
#let x = 2;;
x : int = 2
#f 0;;
- : int = 1
```

The first version of LISP, however, used *dynamic* binding, where a rebinding of a variable propagated to earlier uses of the variable, so that the analogous sequence to the above would return 2. This was in fact originally regarded as a bug, but soon programmers started to appreciate its convenience. It meant that when some low-level function was modified, the change propagated automatically to all applications of it in higher level functions, without the need for recompilation. The feature survived for a long time in many LISP dialects, but eventually the view that static binding is better prevailed. In Common LISP, static binding is the default, but dynamic binding is available if desired via the keyword `special`.

5.7 Polymorphic functions

We can define polymorphic functions, like the identity operator:

```
#let I = fun x -> x;;
I : 'a -> 'a = <fun>
```

ML prints type variables as `'a`, `'b` etc. These are supposed to be ASCII representations of α, β and so on. We can now use the polymorphic function several times with different types:

```
#I true;;
- : bool = true
#I 1;;
- : int = 1
#I I I I 12;;
- : int = 12
```

Each instance of `I` in the last expression has a different type, and intuitively corresponds to a different function. In fact, let's define all the combinators:

```
#let I x = x;;
I : 'a -> 'a = <fun>
#let K x y = x;;
K : 'a -> 'b -> 'a = <fun>
#let S f g x = (f x) (g x);;
S : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Note that the system keeps track of the types for us, even though in the last case they were quite complicated. Now, recall that $I = S\ K\ K$; let us try this out in ML:[5]

```
#let I' = S K K;;
I' : '_a -> '_a = <fun>
```

It has the right type,[6] and it may easily be checked in all concrete cases, e.g:

```
#I' 3 = 3;;
- : bool = true
```

In the above examples of polymorphic functions, the system very quickly infers a most general type for each expression, and the type it infers is simple. This usually happens in practice, but there are pathological cases, e.g. the following example due to Mairson (1990). The type of this expression takes about 10 seconds to calculate, and occupies over 4000 lines on an 80-column terminal.

```
let pair x y = fun z -> z x y in
let x1 = fun y -> pair y y in
let x2 = fun y -> x1(x1 y) in
let x3 = fun y -> x2(x2 y) in
let x4 = fun y -> x3(x3 y) in
let x5 = fun y -> x4(x4 y) in
x5(fun z -> z);;
```

We have said that the ML programmer need never enter a type. This is true in the sense that ML will already allocate as general a type as possible to an expression. However it may sometimes be convenient to *restrict* the generality of a type. This cannot make code work that didn't work before, but it may serve as documentation regarding the intended purpose of the code; it is also possible to use shorter synonyms for complicated types. Type restriction can be achieved in ML by adding *type annotations* after some expression(s). These type annotations consist of a colon followed by a type. It usually doesn't matter exactly where these annotations are added, provided they enforce the appropriate constraints. For example, here are some alternative ways of constraining the identity function to type `int -> int`:

[5] We remarked that from the untyped point of view, $S\ K\ A = I$ for any A. However, the reader may try, for example, $S\ K\ S$ and see that the principal type is less general than expected.

[6] Ignore the underscores for now. This is connected with the typing of imperative features, and we will discuss it later.

```
#let I (x:int) = x;;
I : int -> int = <fun>
#let I x = (x:int);;
I : int -> int = <fun>
#let (I:int->int) = fun x -> x;;
I : int -> int = <fun>
#let I = fun (x:int) -> x;;
I : int -> int = <fun>
#let I = ((fun x -> x):int->int);;
I : int -> int = <fun>
```

5.8 Equality of functions

Instead of comparing the actions of I and I' on particular arguments like 3, it would seem that we can settle the matter definitively by comparing the functions themselves. However this doesn't work:

```
#I' = I;;
Uncaught exception: Invalid_argument "equal: functional value"
```

It is in general *forbidden* to compare functions for equality, though a few special instances, where the functions are obviously the same, yield `true`:

```
#let f x = x + 1;;
f : int -> int = <fun>
#let g x = x + 1;;
g : int -> int = <fun>
#f = f;;
- : bool = true
#f = g;;
Uncaught exception: Invalid_argument "equal: functional value"
#let h = g;;
h : int -> int = <fun>
#h = f;;
Uncaught exception: Invalid_argument "equal: functional value"
#h = g;;
- : bool = true
```

Why these restrictions? Aren't functions supposed to be first-class objects in ML? Yes, but unfortunately, (extensional) function equality is not computable. This follows from a number of classic theorems in recursion theory, such as the *unsolvability of the halting problem* and *Rice's theorem.*[7] Let us give a concrete illustration of why this might be so. It is still an open problem whether the

[7]You will see these results proved in the Computation Theory course. Rice's theorem is an extremely strong undecidability result which asserts that *any* nontrivial property of the function corresponding to a program is uncomputable from its text. An excellent computation theory textbook is Davis, Sigal, and Weyuker (1994).

following function terminates for all arguments, the assertion that it does being known as the *Collatz conjecture*:[8]

```
#let rec collatz n =
   if n <= 1 then 0
   else if even(n) then collatz(n / 2)
   else collatz(3 * n + 1);;
collatz : int -> int = <fun>
```

What is clear, though, is that if it does halt it returns 0. Now consider the following trivial function:

```
#let f (x:int) = 0;;
f : int -> int = <fun>
```

By deciding the equation `collatz = f`, the computer would settle the Collatz conjecture. It is easy to concoct other examples for open mathematical problems.

It is possible to trap out applications of the equality operator to functions and datatypes built up from them as part of typechecking, rather than at runtime. Types that do not involve functions in these ways are known as *equality types*, since it is always valid to test objects of such types for equality. On the negative side, this makes the type system much more complicated. However one might argue that static typechecking should be extended as far as feasibility allows.

Further reading

Many books on functional programming bear on the general issues we have discussed here, such as evaluation strategy. A good elementary introduction to CAML Light and functional programming is Mauny (1995). Paulson (1991) is another good textbook, though based on Standard ML.

Exercises

1. Suppose that a 'conditional' function defined by `ite(b,x,y) = if b then x else y` is the only function available that operates on arguments of type `bool`. Is there any way to write a factorial function?

2. Use the typing rules in the last chapter to write out a formal proof that the S combinator has the type indicated by the ML system.

[8] A good survey of this problem, and attempts to solve it, is given by Lagarias (1985). Strictly, we should use unlimited precision integers rather than machine arithmetic. We will see later how to do this.

3. Write a simple recursively defined function to perform exponentiation of integers, i.e. calculate x^n for $n \geq 0$. Write down two ML functions, the equality of which corresponds to the truth of Fermat's last theorem: there are no integers x, y, z and natural number $n > 2$ with $x^n + y^n = z^n$ except for the trivial case where $x = 0$ or $y = 0$.