Chapter 15

Lazy Data Structures

In ML all variables are bound *by value*, which means that the bindings of variables are *fully evaluated* expressions, or *values*. This general principle has several consequences:

1. The right-hand side of a `val` binding is evaluated before the binding is effected. If the right-hand side has no value, the `val` binding does not take effect.

2. In a function application the argument is evaluated before being passed to the function by binding that value to the parameter of the function. If the argument does not have a value, then neither does the application.

3. The arguments to value constructors are evaluated before the constructed value is created.

According to the by-value discipline, the bindings of variables are evaluated, regardless of whether that variable is ever needed to complete execution. For example, to compute the result of applying the function `fn x => 1` to an argument, we never actually need to evaluate the argument, but we do anyway. For this reason ML is sometimes said to be an *eager* language.

An alternative is to bind variables *by name*,[1] which means that the binding of a variable is an *unevaluated* expression, known as a *computation* or a

[1]The terminology is historical, and not well-motivated. It is, however, firmly established.

suspension or a *thunk*.[2] This principle has several consequences:

1. The right-hand side of a `val` binding is not evaluated before the binding is effected. The variable is bound to a computation (unevaluated expression), not a value.

2. In a function application the argument is passed to the function in unevaluated form by binding it directly to the parameter of the function. This holds regardless of whether the argument has a value or not.

3. The arguments to value constructor are left unevaluated when the constructed value is created.

According to the by-name discipline, the bindings of variables are only evaluated (if ever) when their values are required by a primitive operation. For example, to evaluate the expression `x+x`, it is necessary to evaluate the binding of `x` in order to perform the addition. Languages that adopt the by-name discipline are, for this reason, said to be *lazy*.

This discussion glosses over another important aspect of lazy evaluation, called *memoization*. In actual fact laziness is based on a refinement of the *by-name* principle, called the *by-need* principle. According to the by-name principle, variables are bound to unevaluated computations, and are evaluated only as often as the value of that variable's binding is required to complete the computation. In particular, to evaluate the expression `x+x` the value of the binding of `x` is needed *twice*, and hence it is evaluated twice. According to the by-need principle, the binding of a variable is evaluated *at most once* — not at all, if it is never needed, and exactly once if it ever needed at all. Re-evaluation of the same computation is avoided by *memoization*. Once a computation is evaluated, its value is saved for future reference should that computation ever be needed again.

The advantages and disadvantages of lazy *vs.* eager languages have been hotly debated. We will not enter into this debate here, but rather content ourselves with the observation that *laziness is a special case of eagerness*. (Recent versions of) ML have *lazy data types* that allow us to treat unevaluated computations as values of such types, allowing us to incorporate laziness into the language without disrupting its fundamental character

[2] For reasons that are lost in the mists of time.

on which so much else depends. This affords the benefits of laziness, but on a controlled basis — we can use it when it is appropriate, and ignore it when it is not.

The main benefit of laziness is that it supports *demand-driven* computation. This is useful for representing *on-line* data structures that are created only insofar as we examine them. *Infinite* data structures, such as the sequence of *all* prime numbers in order of magnitude, are one example of an on-line data structure. Clearly we cannot ever "finish" creating the sequence of all prime numbers, but we can create as much of this sequence as we need for a given run of a program. *Interactive* data structures, such as the sequence of inputs provided by the user of an interactive system, are another example of on-line data structures. In such a system the user's inputs are not pre-determined at the start of execution, but rather are created "on demand" in response to the progress of computation up to that point. The demand-driven nature of on-line data structures is precisely what is needed to model this behavior.

Note: Lazy evaluation is a non-standard feature of ML that is supported only by the SML/NJ compiler. The lazy evaluation features must be enabled by executing the following at top level:

```
Compiler.Control.lazysml := true;
open Lazy;
```

15.1 Lazy Data Types

SML/NJ provides a general mechanism for introducing lazy data types by simply attaching the keyword `lazy` to an ordinary `datatype` declaration. The ideas are best illustrated by example. We will focus attention on the type of *infinite streams*, which may be declared as follows:

```
datatype lazy 'a stream = Cons of 'a * 'a stream
```

Notice that this type definition has no "base case"! Had we omitted the keyword `lazy`, such a `datatype` would not be very useful, since there would be no way to create a value of that type!

Adding the keyword `lazy` makes all the difference. Doing so specifies that the values of type *typ* `stream` are *computations* of values of the form

`Cons (`*val*`,` *val*′`)`,

where *val* is of type *typ*, and *val′* is another such computation. Notice how this description captures the "incremental" nature of lazy data structures. The computation is not evaluated until we examine it. When we do, its structure is revealed as consisting of an element *val* together with another suspended computation of the same type. Should we inspect that computation, it will again have this form, and so on *ad infinitum*.

Values of type *typ* `stream` are created using a `val rec lazy` declaration that provides a means for building a "circular" data structure. Here is a declaration of the infinite stream of `1`'s as a value of type `int stream`:

```
val rec lazy ones = Cons (1, ones)
```

The keyword `lazy` indicates that we are binding `ones` to a computation, rather than a value. The keyword `rec` indicates that the computation is *recursive* (or *self-referential* or *circular*). It is the computation whose underlying value is constructed using `Cons` (the only possibility) from the integer `1` and *the very same computation itself*.

We can inspect the underlying value of a computation by pattern matching. For example, the binding

```
val Cons (h, t) = ones
```

extracts the "head" and "tail" of the stream `ones`. This is performed by evaluating the computation bound to `ones`, yielding `Cons (1, ones)`, then performing ordinary pattern matching to bind `h` to `1` and `t` to `ones`.

Had the pattern been "deeper", further evaluation would be required, as in the following binding:

```
val Cons (h, (Cons (h', t')) = ones
```

To evaluate this binding, we evaluate `ones` to `Cons (1, ones)`, binding `h` to `1` in the process, then evaluate `ones` again to `Cons (1, ones)`, binding `h'` to `1` and `t'` to `ones`. The general rule is *pattern matching forces evaluation of a computation to the extent required by the pattern*. This is the means by which lazy data structures are evaluated only insofar as required.

15.2 Lazy Function Definitions

The combination of (recursive) `lazy` function definitions and decomposition by pattern matching are the core mechanisms required to support lazy

evaluation. However, there is a subtlety about function definitions that requires careful consideration, and a third new mechanism, the *lazy function* declaration.

Using pattern matching we may easily define functions over lazy data structures in a familiar manner. For example, we may define two functions to extract the head and tail of a stream as follows:

```
fun shd (Cons (h, _)) = h
fun stl (Cons (_, s)) = s
```

These are functions that, when applied to a stream, evaluate it, and match it against the given patterns to extract the head and tail, respectively.

While these functions are surely very natural, there is a subtle issue that deserves careful discussion. The issue is whether these functions are "lazy enough". From one point of view, what we are doing is decomposing a computation by evaluating it and retrieving its components. In the case of the `shd` function there is no other interpretation — we are extracting a value of type *typ* from a value of type *typ* `stream`, which is a computation of a value of the form `Cons (`exp_h`,` exp_t`)`. We can adopt a similar viewpoint about `stl`, namely that it is simply extracting a component value from a computation of a value of the form `Cons (`exp_h`,` exp_t`)`.

However, in the case of `stl`, another point of view is also possible. Rather than think of `stl` as *extracting* a value from a stream, we may instead think of it as *creating* a stream out of another stream. Since streams are computations, the stream created by `stl` (according to this view) should also be suspended until its value is required. Under this interpretation the argument to `stl` should not be evaluated until its result is required, rather than at the time `stl` is applied. This leads to a variant notion of "tail" that may be defined as follows:

```
fun lazy lstl (Cons (_, s)) = s
```

The keyword `lazy` indicates that an application of `lstl` to a stream does *not* immediately perform pattern matching on its argument, but rather *sets up* a stream computation that, when forced, forces the argument and extracts the tail of the stream.

The behavior of the two forms of tail function can be distinguished using `print` statements as follows:

```
val rec lazy s = (print "."; Cons (1, s))
val _ = stl s    (* prints "." *)
val _ = stl s    (* silent *)
val rec lazy s = (print "."; Cons (1, s));
val _ = lstl s   (* silent *)
val _ = stl s    (* prints "." *)
```

Since `stl` evaluates its argument when applied, the "`.`" is printed when it is first called, but not if it is called again. However, since `lstl` only sets up a computation, its argument is not evaluated when it is called, but only when its result is evaluated.

15.3 Programming with Streams

Let's define a function `smap` that applies a function to every element of a stream, yielding another stream. The type of `smap` should be `('a -> 'b) -> 'a stream -> 'b stream`. The thing to keep in mind is that the application of `smap` to a function and a stream should set up (but not compute) another stream that, when forced, forces the argument stream to obtain the head element, applies the given function to it, and yields this as the head of the result.

Here's the code:

```
fun smap f =
    let
        fun lazy loop (Cons (x, s)) =
            Cons (f x, loop s)
    in
        loop
    end
```

We have "staged" the computation so that the partial application of `smap` to a function yields a function that loops over a given stream, applying the given function to each element. This loop is a lazy function to ensure that it merely sets up a stream computation, rather than evaluating its argument when it is called. Had we dropped the keyword `lazy` from the definition of the loop, then an application of `smap` to a function and a stream would immediately force the computation of the head element of

the stream, rather than merely set up a future computation of the same result.

To illustrate the use of smap, here's a definition of the infinite stream of natural numbers:

```
val one_plus = smap (fn n => n+1)
val rec lazy nats = Cons (0, one_plus nats)
```

Now let's define a function `sfilter` of type

```
('a -> bool) -> 'a stream -> 'a stream
```

that filters out all elements of a stream that do not satisfy a given predicate.

```
fun sfilter pred =
    let
        fun lazy loop (Cons (x, s)) =
            if pred x then
               Cons (x, loop s)
            else
               loop s
    in
        loop
    end
```

We can use `sfilter` to define a function `sieve` that, when applied to a stream of numbers, retains only those numbers that are not divisible by a preceding number in the stream:

```
fun m mod n = m - n * (m div n)
fun divides m n = n mod m = 0
fun lazy sieve (Cons (x, s)) =
    Cons (x, sieve (sfilter (not o (divides x)) s))
```

(This example uses `o` for function composition.)

We may now define the infinite stream of primes by applying `sieve` to the natural numbers greater than or equal to 2:

```
val nats2 = stl (stl nats)
val primes = sieve nats2
```

To inspect the values of a stream it is often useful to use the following function that takes $n \geq 0$ elements from a stream and builds a list of those n values:

```
fun take 0 _ = nil
  | take n (Cons (x, s)) = x :: take (n-1) s
```

Here's an example to illustrate the effects of memoization:

```
val rec lazy s = Cons ((print "."; 1), s)
val Cons (h, _) = s;
(* prints ".", binds h to 1 *)
val Cons (h, _) = s;
(* silent, binds h to 1 *)
```

Replace `print ".";1` by a time-consuming operation yielding `1` as result, and you will see that the second time we force `s` the result is returned instantly, taking advantage of the effort expended on the time-consuming operation induced by the first force of `s`.

15.4 Sample Code

Here is the code for this chapter.