Purely Functional Data Structures

Chris Okasaki

Book DOI: http://dx.doi.org/10.1017/CBO9780511530104

Online ISBN: 9780511530104

Hardback ISBN: 9780521631242

Paperback ISBN: 9780521663502

Chapter

4 - Lazy Evaluation pp. 31-38

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.005

Cambridge University Press

4

Lazy Evaluation

Lazy evaluation is the default evaluation strategy of many functional programming languages (although not of Standard ML). This strategy has two essential properties. First, the evaluation of a given expression is delayed, or *suspended*, until its result is needed. Second, the first time a suspended expression is evaluated, the result is *memoized* (i.e., cached) so that, if it is ever needed again, it can be looked up rather than recomputed. Both aspects of lazy evaluation are algorithmically useful.

In this chapter, we introduce a convenient notation for lazy evaluation and illustrate this notation by developing a simple streams package. We will use both lazy evaluation and streams extensively in later chapters.

4.1 $-notation

Unfortunately, the definition of Standard ML [MTHM97] does not include support for lazy evaluation, so each compiler is free to provide its own set of lazy evaluation primitives. We present here one such set of primitives, called $-notation. Translating programs written with $-notation into other notations for lazy evaluation should be straightforward.

In $-notation, we introduce a new type α susp to represent suspensions. This type has a single, unary constructor called $. To a first approximation, α susp and $ behave as if defined by the ordinary datatype declaration

datatype α susp = $ **of** α

We create a new suspension of type τ susp by writing $$e$, where e is an expression of type τ. Similarly, we extract the contents of an existing suspension by matching against the pattern $$p$. If the pattern p matches values of type τ, then $$p$ matches suspensions of type τ susp.

The main difference between $ and ordinary constructors is that $ does not

immediately evaluate its argument. Instead, it saves whatever information it will need to resume the evaluation of the argument expression at a later time. (Typically, this information consists of a code pointer together with the values of the free variables in the expression.) The argument expression is not evaluated until and unless the suspension is matched against a pattern of the form $\$p$. At that time, the argument expression is evaluated and the result is memoized. Then the result is matched against the pattern p. If the suspension is later matched against another pattern of the form $\$p'$, the memoized value of the suspension is looked up and matched against p'.

The \$ constructor is also parsed differently from ordinary constructors. First, the scope of the \$ constructor extends as far to the right as possible. Thus, for example, the expression \$*f x* parses as \$(*f x*) rather than (\$*f*) *x*, and the pattern \$CONS (*x, xs*) parses as \$(CONS (*x, xs*)) rather than (\$CONS) (*x, xs*). Second, \$ does not constitute a valid expression by itself—it must always be accompanied by an argument.

As an example of \$-notation, consider the following program fragment:

```
val s = $primes 1000000      (* fast *)
...
val $x = s                   (* slow *)
...
val $y = s                   (* fast *)
...
```

This program computes the one millionth prime. The first line, which simply creates a new suspension, runs very quickly. The second line actually computes the prime by evaluating the suspension. Depending on the algorithm for computing primes, it might take a long time. The third line looks up the memoized value and also runs very quickly.

As a second example, consider the fragment

```
let val s = $primes 1000000
in 15 end
```

This program never demands the contents of the suspension, and so never evaluates primes 1000000.

Although we can program all the examples of lazy evaluation in this book using only \$-expressions and \$-patterns, two forms of syntactic sugar will be convenient. The first is the force operator, defined as

```
fun force ($x) = x
```

This is useful for extracting the contents of a suspension in the middle of an expression, where it might be awkward to introduce a pattern matching construct.

The second form of syntactic sugar is useful for writing certain kinds of lazy functions. For example, consider the following function for addition of suspended integers:

fun plus ($m, $n) = $m+n

Although this function seems perfectly reasonable, it is in fact not the function that we probably intended. The problem is that it forces both of its arguments too early. In particular, it forces its arguments when plus is applied, rather than when the suspension created by plus is forced. One way to get the desired behavior is to explicitly delay the pattern matching, as in

fun plus (x, y) = **$case** ($x$, y) **of** ($m, $n) ⇒ $m+n$

However, this idiom is common enough that we provide syntactic sugar for it, writing

fun lazy $f\ p = e$

instead of

fun $f\ x$ = **$case** x **of** p ⇒ force e

The extra force ensures that the **lazy** keyword has no effect on the type of a function (assuming that the result was a susp type to begin with), so we can add or remove the annotation without changing the function text in any other way. Now we can write the desired function for addition of suspended integers simply as

fun lazy plus ($m, $n) = $m+n

Expanding this syntactic sugar yields

fun plus (x, y) = **$case** ($x$, y) **of** ($m, $n) ⇒ force ($m+n)

which is exactly same as the hand-written version above except for the extra force and $ around the $m+n$. This force and $ would be optimized away by a good compiler since force ($e) is equivalent to e for any e.

The plus function uses the **lazy** annotation to delay pattern matching, so that $-patterns are not matched prematurely. However, the **lazy** annotation is also useful when the right-hand side of the function is an expression that returns a suspension as the result of a possibly long and involved computation. Using the **lazy** annotation in this situation delays the execution of the expensive computation from the time the function is applied until the time that its resulting suspension is forced. We will see several functions in the next section that use the **lazy** annotation in this fashion.

The syntax and semantics of $-notation are formally defined in [Oka96a].

4.2 Streams

As an extended example of lazy evaluation and $-notation in Standard ML, we next develop a small streams package. These streams will be used in several of the data structures in subsequent chapters.

Streams (also known as lazy lists) are similar to ordinary lists, except that every cell is systematically suspended. The type of streams is

```
datatype α StreamCell = NIL | CONS of α × α Stream
withtype α Stream = α StreamCell susp
```

A simple stream containing the elements 1, 2, and 3 could be written

```
$CONS (1, $CONS (2, $CONS (3, $NIL)))
```

It is illuminating to contrast streams with suspended lists of type α list susp. The computations represented by the latter type are inherently *monolithic*—once begun by forcing the suspended list, they run to completion. The computations represented by streams, on the other hand, are often *incremental*—forcing a stream executes only enough of the computation to produce the outermost cell and suspends the rest. This behavior is common among datatypes such as streams that contain nested suspensions.

To see this difference in behavior more clearly, consider the append function, written $s +\!\!+ t$. On suspended lists, this function might be written

```
fun s ++ t = $(force s @ force t)
```

or, equivalently,

```
fun lazy ($xs) ++ ($ys) = $(xs @ ys)
```

The suspension produced by this function forces both arguments and then appends the two lists, producing the entire result. Hence, this suspension is monolithic. We also say that the function is monolithic. On streams, the append function is written

```
fun lazy ($NIL) ++ t = t
       | ($CONS (x, s)) ++ t = $CONS (x, s ++ t)
```

This function immediately returns a suspension, which, when forced, demands the first cell of the left stream by matching against a $-pattern. If this cell is a CONS, then we construct the result from x and $s +\!\!+ t$. Because of the **lazy** annotation, the recursive call simply creates another suspension, without doing any extra work. Hence, the computation described by this function is

incremental; it produces the first cell of the result and delays the rest. We also say that the function is incremental.

Another incremental function is take, which extracts the first n elements of a stream.

```
fun lazy take (0, s) = $NIL
        | take (n, $NIL) = $NIL
        | take (n, $CONS (x, s)) = $CONS (x, take (n−1, s))
```

As with ++, the recursive call to take $(n-1, s)$ immediately returns a suspension, rather than executing the rest of the function.

However, consider the function to delete the first n elements of a stream, which could be written

```
fun lazy drop (0, s) = s
        | drop (n, $NIL) = $NIL
        | drop (n, $CONS (x, s)) = drop (n−1, s)
```

or more efficiently as

```
fun lazy drop (n, s) = let fun drop' (0, s) = s
                           | drop' (n, $NIL) = $NIL
                           | drop' (n, $CONS (x, s)) = drop' (n−1, s)
                       in drop' (n, s) end
```

This function is monolithic because the recursive calls to drop′ are never delayed—calculating the first cell of the result requires executing the entire function. Here we use the **lazy** annotation to delay the initial call to drop′ rather than to delay pattern matching.

Exercise 4.1 Use the fact that force ($\$e$) is equivalent to e to show that these two definitions of drop are equivalent. ◇

Another common monolithic stream function is reverse.

```
fun lazy reverse s =
    let fun reverse' ($NIL, r) = r
          | reverse' ($CONS (x, s), r) = reverse' (s, $CONS (x, r))
    in reverse' (s, $NIL) end
```

Here the recursive calls to reverse′ are never delayed, but note that each recursive call creates a new suspension of the form $\$\text{CONS}\ (x, r)$. It might seem then that reverse does not in fact do all of its work at once. However, suspensions such as these, whose bodies contain only a few constructors and variables, with no function applications, are called *trivial*. Trivial suspensions are delayed, not for any algorithmic purpose, but rather to make the types work out. We can consider the body of a trivial suspension to be executed at the time the

```
signature STREAM =
sig
  datatype α StreamCell = NIL | CONS of α × α Stream
  withtype α Stream = α StreamCell susp

  val ++      : α Stream × α Stream → α Stream   (* stream append *)
  val take    : int × α Stream → α Stream
  val drop    : int × α Stream → α Stream
  val reverse : α Stream → α Stream
end

structure Stream : STREAM =
struct
  datatype α StreamCell = NIL | CONS of α × α Stream
  withtype α Stream = α StreamCell susp

  fun lazy ($NIL) ++ t = t
         | ($CONS (x, s)) ++ t = $CONS (x, s ++ t)
  fun lazy take (0, s) = $NIL
         | take (n, $NIL) = $NIL
         | take (n, $CONS (x, s)) = $CONS (x, take (n−1, s))
  fun lazy drop (n, s) =
       let fun drop' (0, s) = s
             | drop' (n, $NIL) = $NIL
             | drop' (n, $CONS (x, s)) = drop' (n−1, s)
       in drop' (n, s) end
  fun lazy reverse s =
       let fun reverse' ($NIL, r) = r
             | reverse' ($CONS (x, s), r) = reverse' (s, $CONS (x, r))
       in reverse' (s, $NIL) end
end
```

Figure 4.1. A small streams package.

suspension is created. In fact, a reasonable compiler optimization is to create such suspensions in already-memoized form. Either way, forcing a trivial suspension never takes more than $O(1)$ time.

Although monolithic stream functions such as drop and reverse are common, incremental functions such as ++ and take are the *raison d'être* of streams. Each suspension carries a small but significant overhead, so for maximum efficiency laziness should be used only when there is a good reason to do so. If the only uses of lazy lists in a given application are monolithic, then that application should use simple suspended lists rather than streams.

Figure 4.1 summarizes these stream functions as a Standard ML module. Note that this module does not export functions such as isEmpty and cons, as one might expect. Instead, we deliberately expose the internal representation in order to support pattern matching on streams.

Exercise 4.2 Implement insertion sort on streams. Show that extracting the first k elements of sort *xs* takes only $O(n \cdot k)$ time, where n is the length of *xs*, rather than $O(n^2)$ time, as might be expected of insertion sort.

4.3 Chapter Notes

Lazy Evaluation Wadsworth [Wad71] introduced lazy evaluation as an optimization of normal-order reduction in the lambda calculus. Vuillemin [Vui74] later showed that, under certain restricted conditions, lazy evaluation is an optimal evaluation strategy. The formal semantics of lazy evaluation has been studied extensively [Jos89, Lau93, OLT94, AFM+95].

Streams Landin introduced streams in [Lan65], but without memoization. Friedman and Wise [FW76] and Henderson and Morris [HM76] extended Landin's streams with memoization.

Memoization Michie [Mic68] coined the term memoization to denote the augmentation of functions with a cache of argument–result pairs. The argument field is dropped when memoizing suspensions by regarding suspensions as nullary functions—that is, functions with zero arguments. Hughes [Hug85] later applied memoization, in the original sense of Michie, to functional programs.

Algorithmics Both components of lazy evaluation—delaying computations and memoizing the results—have a long history in algorithm design, although not always in combination. The idea of delaying the execution of potentially expensive computations (often deletions) is used to good effect in hash tables [WV86], priority queues [ST86b, FT87], and search trees [DSST89]. Memoization, on the other hand, is the basic principle of such techniques as dynamic programming [Bel57] and path compression [HU73, TvL84].