# How to add laziness to a strict language without even being odd


Philip Wadler
Bell Laboratories, Lucent Technologies
wadler@research.bell-labs.com

Walid Taha
Oregon Graduate Institute
walidt@cse.ogi.edu

David MacQueen
Bell Laboratories, Lucent Technologies
dbm@research.bell-labs.com



**Abstract**

This note explicates two styles of lazy programming, showing how one (called "odd") is easy to encode in the traditional '`delay`' and '`force`' syntax, but forces too much evaluation, while the other (called "even") delays evaluation as in traditional lazy languages, but is harder to encode. It presents a new '`lazy`' syntax, and shows how the even style is easy to encode in this syntax, while the odd style is harder to encode. The new '`lazy`' syntax is defined by translation into the '`delay-force`' syntax. Comparisons are drawn with two other syntaxes, one used in CAML and one proposed by Chris Okasaki, and a prototype implementation in SML/NJ is described.


## 1 Introduction

It's easy to add laziness to a strict language. A number of schemes have been proposed, all variants of the following idea: add a type constructor for suspensions, and language constructs to delay and force evaluation.

$$\frac{\texttt{e : t}}{\texttt{delay e : t susp}} \qquad \frac{\texttt{e : t susp}}{\texttt{force e : t}}$$

The delayed expression is not evaluated unless the corresponding suspension is forced, and if this happens the suspension is overwritten with a value, so the expression is not evaluated again. The '`delay`' operation must be a new language construct, not a function, because it does not evaluate its argument. For symmetry, '`force`' is also taken to be a language construct.

However, although easy to understand and implement, these schemes are not easy to use. Every scheme we've seen encourages a style of programming that forces more evaluation than traditional lazy languages (we call this style "odd"), and discourages a style of programming that delays evaluation as in traditional lazy languages (we call this style "even"). In essence, these schemes encourage the programmer to make an "off by one" error.

This note explicates the two styles, showing how the odd style is easy to encode in the traditional '`delay`' and '`force`' syntax, while the even style is harder to encode. It then presents a new '`lazy`' syntax, and shows how the even style is easy to encode in this syntax, while the odd style is harder to encode. The new '`lazy`' syntax is defined by translation into the '`delay-force`' syntax. Throughout, we take Standard ML as the strict language to be extended with lazy constructs (Milner et al., 1997). Comparisons are drawn with two other syntaxes, one used in CAML and one proposed by Chris Okasaki. Finally, we describe how this feature is implemented in the SML/NJ compiler.

## 2 Odd and even, with ease and difficulty

### 2.1 The odd style, with ease

Consider first a language simply augmented with '`delay`', '`force`', and '`susp`'. Streams in the odd style are defined as in Figure 1. One can find similar definitions in the texts of Abelson and Sussman (1985) and Paulson (1991).

In this style, here is the stream corresponding to the list `[1,2]`.

```
Cons (1, delay Cons (2, delay Nil))
```

This style is called odd because any finite stream will contain an odd number of constructors, where '`Cons`', '`Nil`', and '`delay`' are counted as constructors.

This style is also called odd because it exhibits behaviour that may seem peculiar to anyone familiar with streams. One would expect the expression

```
cutoff 5 (map (sqrt o real) (countdown 4))
```

to return the list of square roots of the integers from four to zero

```
[2.0, 1.73, 1.41, 1.0, 0.0]
```

but instead it returns

```
uncaught exception Sqrt.
```

The problem is that this definition of streams forces more evaluation than necessary. Even when the cutoff reaches zero, the stream must be evaluated either to '`Nil`' or '`Cons`', forcing one to compute the square root of minus one without need, and indeed with some harm.

### 2.2 The even style, with difficulty

Figure 2 presents the same program, rewritten to exhibit the traditional behaviour. The key change is to the definition of the type for streams, from which all else follows. We use a

```
datatype 'a stream           = Nil | Cons of 'a * 'a stream susp;

(*  map                      : ('a -> 'b) -> 'a stream -> 'b stream  *)
fun map f Nil                = Nil
  | map f (Cons(x,xs))       = Cons(f x, delay (map f (force xs)));

(*  countdown                : int -> int stream  *)
fun countdown n              = Cons(n, delay (countdown (n-1)));

(*  cutoff                   : int -> 'a stream -> 'a list  *)
fun cutoff 0 xs              = []
  | cutoff n Nil             = []
  | cutoff n (Cons(x,xs))    = x :: cutoff (n-1) (force xs);
```

Figure 1: Streams in odd style, with ease

```
datatype 'a stream_          = Nil_ | Cons_ of 'a * 'a stream
withtype 'a stream           = 'a stream_ susp;

(*  map                      : ('a -> 'b) -> 'a stream  -> 'b stream   *)
(*  map_                     : ('a -> 'b) -> 'a stream_ -> 'b stream_  *)
fun map f ws                 = delay map_ f (force ws)
and map_ f Nil_              = Nil_
  | map_ f (Cons_(x,xs))     = Cons_(f x, map f xs)

(*  countdown                : int -> int stream  *)
fun countdown n              = delay Cons_(n, countdown (n-1));

(*  cutoff                   : int -> 'a stream  -> 'a list  *)
(*  cutoff_                  : int -> 'a stream_ -> 'a list  *)
fun cutoff 0 ws              = []
  | cutoff n ws              = cutoff_ n (force ws)
and cutoff_ n Nil_           = []
  | cutoff_ n (Cons_(x,xs))  = x :: cutoff (n-1) xs;
```

Figure 2: Streams in even style, with difficulty

convention of appending an underbar to name the auxiliary type and the operations upon it. Note that the '`stream_`' of this definition is equivalent to the '`stream`' of the previous definition.

In this style, here is the stream corresponding to the list `[1,2]`.

```
delay Cons (1, delay Cons (2, delay Nil))
```

This style is called even because any finite stream will contain an even number of constructors, where '`Cons`', '`Nil`', and '`delay`' are counted as constructors.

The new program gives the expected result for the previous expression, returning a list of square roots rather than an exception.

Alas, our definition has nearly doubled in size, and halved in perspicuity. Paulson wrote to us that "My book originally used the second definition you present, but that extra bit of laziness cost too much in legibility." He reverted to the odd style, relegating even style to an exercise.

### 2.3 The even style, with ease

Figure 3 show how the definition looks under the new proposal. Intuitively, the effect of writing '`lazy`' in a datatype definition is to make each constructor return a suspension, and the effect of writing '`lazy`' in a function definition is to delay evaluation of the right-hand side. The exact meaning of these constructs will be given later, by translation into '`delay`', '`force`', and '`susp`'. Under that translation, this program is equivalent to the previous program.

A semantic subtlety: all functions and constructors here are strict, in that their arguments are always evaluated. Delayed evaluation is always indicated by adding the keyword '`lazy`' to a function or value definition. We will return to this point after discussing the translation of programs.

A syntactic subtlety: it may seem more natural to say '`lazy fun`' than '`fun lazy`', but the latter works better when there are mutually recursive types or functions, only some of which are lazy. We'll see examples in the next sec-

```
datatype lazy  'a stream      = Nil | Cons of 'a * 'a stream;

(*  map                       : ('a -> 'b) -> 'a stream -> 'b stream  *)
fun lazy map f Nil            = Nil
       | map f (Cons(x,xs))   = Cons(f x, map f xs);

(*  countdown                 : int -> int stream  *)
fun lazy countdown n          = Cons (n, countdown (n-1));

(*  cutoff                    : int -> 'a stream -> 'a list  *)
fun cutoff 0 xs               = []
  | cutoff n Nil              = []
  | cutoff n (Cons(x,xs))     = x :: cutoff (n-1) xs;
```

Figure 3: Streams in even style, with ease

```
datatype 'a stream                = Nil | Cons of 'a * 'a stream_
and lazy 'a stream_               = Delay of 'a stream;

(*  map                           : ('a -> 'b) -> 'a stream  -> 'b stream  *)
(*  map_                          : ('a -> 'b) -> 'a stream_ -> 'b stream_ *)
fun map f Nil                     =  Nil
  | map f (Cons(x,xs))            =  Cons(f x, map_ f xs)
and lazy map_ f (Delay xs)        =  Delay (map f xs);

(*  countdown                     : int -> int stream   *)
(*  countdown_                    : int -> int stream_  *)
fun countdown n                   = Cons (n, countdown_ n)
and lazy countdown_ n             = Delay (countdown (n-1));

(*  cutoff                        : int -> 'a stream -> 'a list  *)
fun cutoff 0 xs                   = []
  | cutoff n Nil                  = []
  | cutoff n (Cons(x,Delay(xs)))  = x :: cutoff (n-1) xs;
```

Figure 4: Streams in odd style, with difficulty

tion.

### 2.4 The odd style, with difficulty

If the odd style is truly what you intend, it is still possible. Figure 4 shows a program for that purpose in the new notation. Again, the types dictate the structure of the program. Under the translation rules given below, this program has exactly the same semantics as the previous program for odd style. The situation is pleasantly dual: the odd style in the new notation is almost exactly as hard to express as the even style was in the old.

## 3 The translation

While the new syntax looks pleasant, it is useless unless there is a clear, rigorous, and uniform way to assign it a meaning. Fortunately, this can be done in a fairly straightforward way, by specifying a translation from the new language, with the '`lazy`' keyword, to the old language, augmented with '`delay`', '`force`', and '`susp`'. There is one rewrite rule for each form of '`lazy`' construct added to the language.

- (datatype)

```
  datatype lazy t = c1 of t1 | ... | cn of tn
===>
  datatype t_ = c1_ of t1 | ... | cn_ of tn
  withtype t  = t_ susp
```

  where `t` is a type variable list followed by a type identifier, `t_` is the same with a fresh type identifier name, and `c1_`, ..., `cn_` are fresh constructor names.

- (fun)

```
  fun lazy f p11 ... p1n = e1
         | ...
```

```
  val rec lazy x1 = e1 and ... and lazy xm = em and y1 = f1 and ... and yn = fn
===>
  val r1 = ref (delay (raise BlackHole)) and ... and rm = ref (delay (raise BlackHole))
  val x1 = delay (force (!r1)) and ... and xm = delay (force (!rm))
  val rec y1 = f1 and ... and yn = fn
  val _ = (r1 := delay (force e1); ...; rm := delay (force em))
```

Figure 5: Translation for mutual recursion

```
        | f pm1 ... pmn = em
===>
  fun f x1 ... xn     = delay (f_ x1 ... xn)
  and f_ p11 ... p1n = force e1
    | ...
    | f_ pm1 ... pmn = force em
```

where `f_` is a fresh variable name.

- (constructor)

```
  c
===>
  fn x => delay (c_ x)
```

where `c` is declared in a lazy datatype, and `c_` is the corresponding fresh constructor.

- (destructor)

[We assume that before this step is applied that all uses of destructors are translated to case expressions, in the usual way.]

```
  case e0 of
    c1 x1 => e1
  | ...
  | cn xn => en
===>
  case (force e0) of
    c1_ x1 => e1
  | ...
  | cn_ xn => en
```

where `c1`,...,`cn` are declared in a lazy datatype, and `c1_`,...,`cn_` are the corresponding fresh constructors.

- (val)

```
  val lazy x = e
==>
  val x = delay (force e)
```

- (val rec)

```
  val rec lazy x = e
===>
  val r = ref (delay (raise BlackHole))
  val x = delay (force (!r))
  val _ = (r := delay (force e))
```

This exploits the usual trick of creating a circularity via references. The exception `BlackHole` is raised only if the expression bound to `x` depends immediately on the value of `x`, as in the ill-defined expression `let val rec lazy x = x in x end`.

This translation encounters a problem with typing: the value restriction on polymorphism prohibits proper generalisation of type variables in the types of `r` and `x`. Fortunately, it is sound to lift the restriction in this case, since the references are assigned to only once when tying the recursive knot.

The translation extends to mutual recursion, as shown in Figure 5. In the general case, there are $m$ lazy bindings to suspensions, and $n$ strict bindings to functional values.

Returning to the semantic subtlety mentioned previously, note that we have the translation

```
  cutoff 0 (Cons(sqrt ~1.0, Nil))
===>
  cutoff 0
    ((fn x => delay (Cons_ x)) (sqrt ~1.0, Nil))
```

and so evaluating this expression raises an error. On the other hand, we also have the translation

```
  let val lazy x = Cons(sqrt ~1.0, Nil)
  in cutoff 0 x end
===>
  let val x = delay (force (Cons(sqrt ~1.0, Nil)))
  in cutoff 0 x end
```

and so evaluating this expression returns `[]`. All functions and constructors evaluate their arguments; evaluation is suspended only where the keyword '`lazy`' appears.

## 4 Previous work

### 4.1 CAML

Some versions of CAML allow summands of datatypes or fields of records to be declared lazy (Weiss 1990, Mauny 1991). Again, this notation makes it easy to use the odd style, and harder to use the even style. Furthermore, one needs to know whether a constructor contains '`lazy`' in its declaration before one knows whether its arguments will be evaluated, unlike the notation described above.

For instance, in the presence of the CAML declaration

```
  type 'a susp = lazy Delay of 'a
```

then the CAML term 'Delay e' is like 'delay e' in our old notation, and the CAML term 'case e of Delay x => x' is like 'force e' in our old notation.

Here's a CAML type declaration for streams in odd style.

```
type 'a stream
  = Nil
  | Cons of {head: 'a, lazy tail: 'a stream};
```

Here's one for streams in even style.

```
type 'a stream_ = Nil | Cons of 'a * 'a stream
and  'a stream  = lazy Delay of 'a stream_;
```

And here's one that corresponds to none of the previous examples.

```
type 'a stream
  = Nil
  | lazy Cons of 'a * 'a stream;
```

This last declaration suspends evaluation of both the head and the tail of the stream.

### 4.2 Okasaki

Okasaki (1996) proposes that one abbreviate 'delay e' by writing $e, and, dually, that the equivalent of 'force e' be written using pattern matching, as (case e of $x => x). The '$' sign may be nested with other patterns.

With this proposal it is even easier to implement streams in odd style. Figure 6 shows a program equivalent to Figure 1. It looks elegant: too bad it is not the program we desire!

### 4.3 Extended '$' notation

In the course of pursuing this topic, we discovered a simple and logical addition to Okasaki's notation that makes it relatively easy to express even style, in at least some situations. The addition is to treat the declaration

```
fun $f p11 ... p1n  =  e1
  | ...
  | $f pm1 ... pmn  =  em
```

as equivalent to

```
fun f p11 ... p1n  =  force e1
  | ...
  | f pm1 ... pmn  =  force em.
```

This is a natural extension of the other abbreviations.

Figure 7 shows streams in even style in the extended $ notation. The code for the functions can be derived from the corresponding code in 'lazy' notation by uniformly replacing 'Nil' with '$Nil', 'Cons(x,xs)' with '$Cons(x,xs)', 'map f xs' with '$map f xs', and 'countdown n' with '$countdown n'. Thus, the test expression becomes

```
cutoff 5 ($map (sqrt o real) ($countdown 4)).
```

The meaning of this program is identical to that in Figure 3 although a little extra laziness has been introduced. In the 'lazy' notation, the decrement in 'countdown (n-1)' is performed immediately, while here the decrement in '$countdown (n-1)' is performed only if the result of the suspension is demanded.

Compared to the 'lazy' notation, the '$' notation has three drawbacks. First, the program may become buried under dollar signs. Second, in the 'lazy' notation one can switch between lazy and strict representations by adding a keyword at the definition site, while in the '$' notation one must change the definition site and add a symbol at each call site as well. Third, the '$' notation works best only when all function and constructor applications are fully saturated. For instance, in the 'lazy' notation one may square each element in a stream of streams by writing 'map (map sqr) xss' while in the '$' notation one must write '$map (fn xs => $map sqr xs) xss'.

On the other hand, the '$' notation has the advantage of fine control over laziness. This will be useful in those few situations where the odd style is actually desired, because one is willing to pay an extra price in complexity to eliminate every unnecessary vestige of lazy evaluation.

*Postscript:* After the first part of this note was written, Chris Okasaki observed that the 'lazy' syntax combines perfectly well with the '$' syntax. Neither syntactic sugar interferes with the other, and the sum appears to combine the advantages of both.

## 5 Implementation

Prototype implementations of the lazy declarations described in the last section have been implemented in SML/NJ. For some years SML/NJ has provided a type of suspensions with delay and force operations, similar to that described in the introduction.

```
type 'a susp
val delay : (unit -> 'a) -> 'a susp
val force : 'a susp -> 'a
```

For the implementation of lazy declarations, this type was repackaged as a special datatype (rather like the ref type), similar to that described in Section 4.2.

```
datatype 'a susp = $ of 'a
```

where '$' is a suspending data constructor. When used in an expression, '$' has the effect of delaying the evaluation of the expression to which it is applied, and it can be defined in terms of the delay primitive as follows.

```
  ($ e)
===>
  (delay (fn () => e))
```

When used in a pattern, '$' has the dual effect, namely, that of forcing the evaluation of its argument; during match-compilation, the following translation is used.

```
  (case a of ($p) => e)
===>
  (let val p = force a in e)
```

In the SML/NJ front end, top-level declarations are processed in three major steps: parse, elaborate, and translate. These phases have roughly the following types (ignoring extra parameters like environments).

```
token stream -[parse]->     Ast.dec
             -[elaborate]-> Absyn.dec
             -[translate]-> FLINT.prog
```

```
datatype 'a stream           = Nil | Cons of 'a * 'a stream susp;

(*  map                      : ('a -> 'b) -> 'a stream -> 'b stream  *)
fun map f Nil                = Nil
  | map f (Cons(x,$xs))      = Cons(f x, $map f xs);

(*  countdown                : int -> int stream  *)
fun countdown n              = Cons (n, $countdown (n-1));

(*  cutoff                   : int -> 'a stream -> 'a list  *)
fun cutoff 0 xs              = []
  | cutoff n Nil             = []
  | cutoff n (Cons(x,$xs))   = x :: cutoff (n-1) xs;
```

Figure 6: Streams in odd style, in '$' notation

```
datatype 'a stream_          = Nil | Cons of 'a * 'a stream
withtype 'a stream           = 'a stream_ susp;

(*  map                      : ('a -> 'b) -> 'a stream -> 'b stream_  *)
fun $map f ($Nil)            = $Nil
  | $map f ($Cons(x,xs))     = $Cons(f x, $map f xs)

(*  countdown                : int -> int stream_  *)
fun $countdown n             = $Cons(n, $countdown (n-1)));

(*  cutoff                   : int -> 'a stream  -> 'a list  *)
fun cutoff 0 xs              = []
  | cutoff n ($Nil)          = []
  | cutoff n ($Cons(x,xs))   = x :: cutoff (n-1) xs;
```

Figure 7: Streams in even style, in extended '$' notation

Here `Ast.dec` is a simple syntax tree with identifiers represented by uninterpreted strings, `Absyn.dec` is typed abstract syntax, and `FLINT.prog` is a higher-order typed, enriched lambda calulus. The parse phase does not deal with infix operators, so expressions are parsed as flat pre-expressions, which are reprocessed into their proper structure during the elaboration phase, where infix bindings of symbols are available from the static environment.

The lazy forms of declarations are implemented as "derived forms" (or macros) that are expanded as specified in Section 3. In the first prototype implementation (SML/NJ 110.1), this expansion was implemented in an additional pass:

```
Ast.dec -[lazycomp]-> Ast.dec
```

This translation is context sensitive because we need to distinguish between strict and lazy data constructors, so we have to do some elaboration, developing and propagating a special environment that is used to identify the lazy data constructors. This translation also required a separate fixity analysis prepass to elaborate pre-expressions into their proper structure. The implementation of '$' in terms of delay and force is done in the translate phase.

There were a couple of problems with this initial prototype. The expansion introduces "fragile" free identifiers ('$' in particular) that are subject to being rebound before the expansion takes place. And the fixity analysis phase and lazycomp phase duplicated work that was already being performed in the elaboration phase.

So a second generation prototype implementation (SML/NJ working version 110.8) moved the translation of lazy declarations into the elaboration phase. When elaborating lazy datatype declarations, the associated data constructors are marked as lazy in the static environment. Expansion then takes place after reparsing of infix expressions, and lazy data constructors are identified from their bindings in the normal static environment. This expansion is robust because it uses internal primitives instead of introducing free symbols that are vulnerable to rebinding (e.g. the internal representation of the dataconstructor '$').

The second generation prototype is also much more economical in terms of the code required, adding a total of less than 300 lines of additional code to the compiler.

### 5.1 Performance

Some benchmark results will be provided in the final paper. Preliminary experiments indicate performance is at least an order of magnitude faster than Hugs, but slower than GHC.

### 5.2 Future Work

The second generation implementation is still considered prototype implementation. Its drawbacks include

- cosmetic problems with type names (e.g. '`stream`' often prints as '`?.stream_ ?susp`').
- The cost of extra allocations and indirections for separate '`susp`' cell is significant (e.g., two heap allocated objects per lazy stream element).

These problems will be addressed by a third generation implementation that will make lazy datatypes and lazy data-constructors "first class". This means that lazy data constructors will be intrinsically suspending like '`$`', eliminating the need for them to be wrapped with '`$`'. This approach, however, will not support some hybrid algorithms (even/odd) that mix explicit use of '`$`' with '`lazy`' keywords.

### Acknowledgements

Chris Okasaki entered into a long correspondence on this topic, and his feedback greatly sharpened our thinking. For comments on earlier drafts, we also thank Lal George, Bob Harper, Michel Mauny, Larry Paulson, and Pierre Weis.

### References

Harold Abelson and Gerald Jay Sussman (1985), *The Structure and Interpretation of Computer Programs*, MIT Press.

Michel Mauny (1991), Integrating lazy evaluation in Strict ML, INRIA, Technical report 137.

Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen (1997), *The Definition of Standard ML (Revised)*, MIT Press.

Chris Okasaki (1996), Purely Functional Data Structures, PhD Thesis, Carnegie Mellon University, Technical report CMU-CS-96-177.

Pierre Weis et al. (1990), The CAML Reference Manual, version 2.6.1, INRIA, Technical report 121.

Larry Paulson (1991), *ML for the Working Programmer*, Cambridge University Press.