A Pure Language with Default Strict Evaluation Order and Explicit Laziness

Tim Sheard
OGI School of Science & Engineering
Oregon Health & Science University
20000 N.W. Walker Road
Beaverton, OR 97006-8921
sheard@cse.ogi.edu

ABSTRACT

We propose a pure language with a default strict evaluation order and explicit laziness annotations. These annotations do not make a distinction between the types of values and computations as is often done in lazy additions to strict languages. This supplies all the expressivity available in lazy languages with much of the speed available in strict languages, for the modest cost of programmer supplied annotations. We explore the design space of mixed evaluation strategies, and compare our proposal with other work in this area. A simple implementation suggests that it is relatively straight-forward for the programmer to supply the correct annotations.

1. INTRODUCTION

In lazy languages, computations are deferred until needed. Strictness analysis tries to infer which computations will always be needed in order to avoid building thunks for those computations. A lot of recent work has suggested that the vast majority of computations not identified by strictness analysis will either always be needed anyway, or at least relatively cheap to compute (even if they're never needed). This has suggested evaluation strategies which are "mostly eager", evaluating every computation optimistically, and "backing off" only if it appears that such computation are diverging [3, 11]. In a lazy framework, deferring computation that appears to be diverging is safe, since deferred computation can be restarted, if and when it is eventually needed.

Measurements of such systems show that they can speed up computation significantly, since most computation proceeds without the overhead required to implement laziness. The cost of this "win" is increased complexity in the implementation which must instrument computation to discover when it appears to be diverging. Such instrumentation schemes have been tied to time and space allocation, and require significant additional complexity in both the compiler and the run-time system.

This complexity suggests an alternative approach: Can a language be designed and implemented where evaluation is by default strict, except where it is explicitly annotated by the programmer to be lazy? Such a language would not even need strictness analysis (much less the "backing off" machinery) and would trade complexity of the compiler and run-time system, for increased burden on the programmer (and the possibility if the programmer is wrong, that the program might really diverge). After all the programmer often knows where his program needs to be lazy, and should be able to communicate this to the compiler.

We want to be clear here, that what we are proposing is an investigation into a previously unexplored region of the mixed-evaluation strategy landscape. (See Figure 2 for details). We differ from lazy additions to strict languages like Scheme and ML by requiring only a single *lazy* annotation rather than a pair of *force* and *delay* operations [2, 9]. The second (most observable) difference is that there is no distinction between the types of a lazy computation and a regular one. We differ from the *optimistic evaluation* variants of lazy languages by requiring the programmer to be explicit about where laziness arises, thus allowing a far simpler implementation. What we are considering is a language where denoting lazy evaluation is explicit, but pulling on a lazy computation implicitly forces the computation with no further explicit direction from the programmer. The language implementation expects that any result could indeed turn out to be a lazy computation, but it *never creates* a lazy computation without explicit direction from the programmer.

The language is pure (with no effects other than possible non- termination), and employs a default strict evaluation order. This makes reasoning about space and time straightforward in those programs that don't use lazy evaluation, and the programmer has complete control over which programs do.

In this paper we report the results of a simple thought experiment. Can such a language be designed? What annotations are necessary? What do the annotations mean? Can the language be used by programmers to code up the classic programs where laziness is essential without undue

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Haskell Workshop August 28, 2003. Uppsala, Sweden
Copyright 2003 ACM ??? ...$5.00.

burden? Can we pay for laziness only when it is needed?

To answer these questions we modified a small Haskell interpreter we had built for other reasons. We were surprised how easy it was to give semantics to such a language and how easy it was to program with. While our implementation is only a toy, the ease of use of such a language, and the facility with which it was given semantics, suggests that the functional language community ought to further explore the idea of a pure language with default strict evaluation order and explicit laziness.

2. EXAMPLES

In this section we illustrate by example how to write programs with explicit laziness annotations. We write a few functions (which should look exactly like Haskell, except for the annotations) and then evaluate some small expressions inside our interpreter's read-eval-print loop.

In order to clearly describe what's going on, the *print* part of the read-eval-print loop is non-standard. Rather than use a strict print function that demands the whole object being printed, we use a special print mechanism that can distinguish lazy computations from values. This lets us display where laziness occurs in the system. To do this, the system distinguishes thunks (computations) from other values, and displays them as "`...`". The system also displays lists in the ordinary notation (i.e. `1:2:[]` displays as `[1,2]`) with the additional syntactic sugar that a list ending in a thunk, like `(1 : 2 : ...)`, displays as `[1,2 ...]`. Note the missing comma after the `2`, a list whose last element is a thunk (as opposed to having a tail as a thunk), such as `(1 : 2 : ... : [])` displays with the separating comma as `[1,2, ...]` .

The point of this exercise is to illustrate that the need for laziness is almost always known by the programmer, and this need is easy to make explicit. Our first example is the `from` function that generates an infinite list of successive integers starting from some initial element. Because this list is meant to be infinite, laziness is required, the programmer should know this, and it can be easily expressed using the laziness annotation `lazy`.

```
from n =  n : (lazy (from (n+1)))

take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs
```

The tail of the list created by `from` will always be a thunk because of the annotation. Other expressions are evaluated in a strict, call-by- value manner.

```
prompt> from 3
[3 ...] : [Int]

prompt> take 5 (from 1)
[1,2,3,4,5] : [Int]
```

Evaluating `(from 3)` returns the infinite list whose tail is a thunk. But, applying `(take 5)` to `(from 1)` forces evaluation of the thunked tail to produce a list of length 5. While creating a thunk is explicit, forcing a thunk is implicit, and is performed whenever computation pulls on the thunk. This is completely analogous to pulling on a thunk in Haskell.

Unfortunately things are not quite as simple as this. Consider the two functions `map1` and `map2`.

```
map1 f [] = []
map1 f (x:xs) = f x : map1 f xs

map2 f [] = []
map2 f (x:xs) = f x : (lazy (map2 f xs))
```

Applying these functions to a few lists, we observe:

```
prompt> map2 (+1) [1,2,3]
[2 ...] : [Int]

prompt> map2 (+1) (from 1)
[2 ...] : [Int]

prompt> map1 (+1) [1,2,3]
[2,3,4] : [Int]

prompt> map1 (+1) (from 1)
-- program diverges
```

First, `map2` is too lazy. When applied to a list without any embedded thunks, it inserts unnecessary ones. But `map1` is equally bad, because when applied to a thunk guarding an infinite computation, it repeatedly forces each of the embedded thunks, and hence diverges. What is needed is some behavior in between.

This motivates a need for a second laziness annotation. This annotation is implemented by the first class (but primitive) function `mimic ::  (a -> b) -> a -> b`. The application `(mimic f x)` mimics the *thunkiness* of the argument `x`. If `x` is a thunk it creates a new thunk which will apply `f` to `x` only when pulled on. If `x` is just an ordinary value, it applies `f` to `x` and returns the resulting value. Using `mimic` we can define a reasonable `map` function.

```
map f [] = []
map f (x:xs) = f x : mimic (map f) xs

prompt> map (+1) [1,2,3]
[2,3,4] : [Int]

prompt> map (+1) (from 1)
[2 ...] : [Int]
```

Using only these two annotations, it is possible to program a wide variety of functions that depend crucially on laziness, and to indicate exactly where laziness is required, thus allowing the compiler to use a strict evaluation order everywhere else. We illustrate with a few classical examples. Defining infinite lists by using recursive definitions is possible, just be sure and guard the recursion by a `lazy` annotation.

```
ones = 1 : (lazy ones)

expose 0 xs = xs
expose n [] = []
expose n (x:xs) = x : expose (n-1) xs

prompt> expose 3 ones
[1,1,1 ...] : [Int]
```

The function `expose` exposes an arbitrary prefix of an infinite list, but preserves the thunk of the tail. Computing the infinite list of Fibonacci numbers can use the same pattern as `ones`: exhibit some finite prefix `(0 : 1 : _)` and guard the recursion by lazy.

```
fibs = 0 : 1 : (lazy (zipWith (+) fibs (tail fibs)))
```

```
zipWith f (x:xs) (y:ys) =
  f x y : mimic (mimic (zipWith f) xs) ys

prompt> expose 5 fibs
[0,1,1,2,3 ...] : [Int]
```

Note the use of `mimic` twice in `zipWith` since it recurses down two lists simultaneously, and either list may have embedded thunks.

Computing the infinite list of primes using the sieve of Eratosthenes is also straightforward. Just be sure that `iterate` and `filter` are correctly annotated.

```
iterate f a = a : (lazy (iterate f (f a)))

filter p [] = []
filter p (x:xs) = if p x
                    then x : mimic (filter p) xs
                    else mimic (filter p) xs

sieve (p:xs) = filter pred xs
  where pred y = rem y p /= 0

primes = map head (iter sieve (from 2))

prompt> take 10 primes
[2,3,5,7,11,13,17,19,23,29] : [Int]
```

A comment about `filter` is in order. One could imagine writing the test part of the `if` expression as (`mimic p x`) in anticipation that the elements of the list, as well as its tail, could be a lazy computation. We discuss this further in Section 4.

Our last annotation is the `strict` annotation. It is the identity function which forces its argument if it is a thunk.

```
prompt> tail ones
 ... : [Int]

prompt> id (tail ones)
 ... : [Int]

prompt> strict (tail ones)
[1 ...] : [Int]
```

Note how the tail of `ones` is a thunk. Also note that applying the `id` function to `(tail ones)` does not force the tail since id is not strict in its argument. By the use of `strict` we can force this computation (one step at least). In this sense `strict` is like Haskell's `seq` forcing its argument in to some form of head normal form (i.e. the top level part of the returned result appears as a value, though there may be thunks deeper inside).

For most types, a source level function could be written to have the same effect since, source level functions have a call by value evaluation order. A thunk can be forced by a gratuitous inspection (or pattern match). For example, at the types `Int` and `[a]`, we could write:

```
strictInt x = x+0

strictList (x:xs) = x:xs
```

But since a single function, that works at all types, seems preferable, we supply it as a primitive.

Finally, we exploit laziness on trees, rather than lists, to show that the technique can be generalized to any kind of data. The example illustrated (which is a modified version of depth-first search from [7]) produces a depth-first tree that spans that portion of a graph reachable from a given root. The algorithm proceeds by creating an (possibly) infinite tree from the graph and root, and then pruning away the redundant parts of the tree.

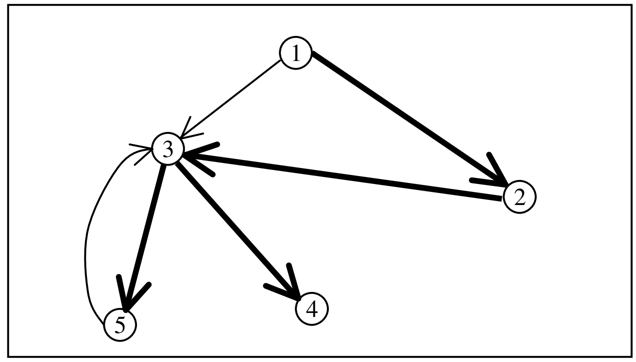

Figure 1: Graph with Superimposed Depth-first Spanning Tree.

```
data Tree = Node Int [Tree]
data Graph  = G [(Int,[Int])]

expand :: Graph -> Int -> Tree
expand (G xs) root =
       case lookup root xs of
         Just children ->
            Node root
                 (map (\ w -> lazy (expand (G xs) w))
                      children)
```

A `Graph` is simply a list of node identifiers (`Int`) paired with a list of successor node identifiers. A `Tree` consists of a `Node` with an identifier and a (possibly empty) list of children (sub-trees). We can `expand` a `Graph` into a `Tree` by building a tree Node for the root, and incorporating the recursively constructed trees for each successor of the root as the children of the `Node`.

Note that if the graph is cyclic then the tree will be infinite. To build the tree on demand, we apply the lazy annotation to each recursive call to `expand`.

In Figure 1 we illustrate a small cyclic graph with a superimposed depth-first spanning tree. We can represent this graph by using the `Graph` datatype constructor `G`. We can expand it into a tree, and observe the structure of the tree.

```
g1 = G [ (1,[2,3]),(2,[3]),(3,[4,5]),(4,[]),(5,[3]) ]

push 0 g = g
push n (Node x ys) = Node x (map (push (n-1)) ys)

prompt> expand g1 1
(Node 1 [ ..., ...]) : Tree

prompt> push 2 (expand g1 1)
(Node 1 [(Node 2 [ ...]),(Node 3 [ ..., ...])]) : Tree
```

Note how the expanded tree has thunks for each child. The function `push` which demands `n` levels of a tree, forces some of those thunks, but leaves undisturbed those thunks below the `n` threshold.

The key to the algorithm is the function `prune`. It is completely oblivious to the needed laziness, which is confined to just the definition of `expand`. The function `prune` behaves like a state transformer. Taking as input a list of nodes previously visited, a list of pending tree nodes yet to be explored, and returns a pair of the trees built, and another (augmented) list of visited nodes.

```
prune :: [Int] -> [Tree] -> ([Tree],[Int])
prune visited [] = ([],visited)
prune visited ((Node n cs) : ts) =
  if member n visited
     then prune visited ts
     else let (xs,vs) = prune (n : visited) cs
              (ys,us) = prune vs ts
          in ((Node n xs) : ys, us)
```

To build a depth-first tree that spans that part of a graph rooted at some node, first create an infinite tree, and then walk the tree in depth first order, pruning away any subtrees, which correspond to graph nodes visited earlier in the walk.

```
dff :: Graph -> Int -> [Tree]
dff g root = fst( prune [] [expand g root] )

prompt> dff g1 1
[(Node 1 [(Node 2 [(Node 3 [(Node 4 []),
                            (Node 5 [])])])])] : [Tree]
```

3. SEMANTICS

In order to think abstractly about such a system, we first devise a simple small step semantics which describes the behavior we wish to use. Then later, we write a monadic interpreter which implements this behavior. In our small step semantics we have the normal β_v and *case* rules

$$
\begin{array}{rcl}
(\lambda\, x \,.\, e)\; v & \rightarrow & e[v/x] \\
\texttt{case}\; (C\; v)\; \texttt{of}\; C\; x\; \texttt{->}\; e & \rightarrow & e[v/x]
\end{array}
$$

Plus the following rules which describe the interaction with the laziness annotations.

$$
\begin{array}{rcl}
(\texttt{lazy}\; e_1)\; e_2 & \rightarrow & e_1\; e_2 \\
\texttt{case}\; (\texttt{lazy}\; e)\; \texttt{of}\ldots & \rightarrow & \texttt{case}\; e\; \texttt{of}\ldots \\
\texttt{strict}\; (\texttt{lazy}\; e) & \rightarrow & e \\
(\texttt{mimic}\; e_1)\; (\texttt{lazy}\; e_2) & \rightarrow & (\texttt{lazy}\; (e_1\; e_2)) \\
(\texttt{mimic}\; e)\; v & \rightarrow & e\; v
\end{array}
$$

Thus we see that `lazy` computations are forced only when appearing in one of three contexts: the function part of an application, the scrutinee of a case, or the argument of a `strict` annotation. To give a big step semantics we use a simple monadic interpreter: `eval :: Env -> Exp -> M V`. Where `V` is the domain of values, `Env` maps program variables to (possibly lazy) values, `Exp` represents programs, and `M` is the monad. In the current implementation `M` is the IO monad with exceptions.

Values include: integers, pairs, data constructed objects, functions, and thunks. Thunks are the key to the explicit laziness and are implemented as `IORef`s so that they can be overwritten once evaluated.

```
type Thunk a = (IORef (Either (M a) a))

data V
  = Vlit Int
  | Vprod V V
  | Vfun (V -> M V)
  | Vcon String [ V ]
  | Vthunk (Thunk V)
```

A thunk (implemented by `Vthunk`) is either a suspended computation (`M V`), or if previously forced, a value (`V`). We use an IORef to model sharing, and to ensure that a computation is only evaluated at most once (the hallmark of true laziness). This works by initializing the reference to (`Left` *computation*), and overwriting the reference with (`Right` *value*) after it has been forced.

In the interpreter all access to thunks is mediated by the two functions: `thunk :: M V -> M V` for creating thunks from computations, and `pullWith :: (V -> M a) -> Thunk V -> M a` for forcing and overwriting thunks.

```
thunk :: M V -> M V
thunk c = do { r <- newRef (Left c); return(Vthunk r) }
```

Given a computation that should be suspended, `thunk` allocates a new reference cell, stores the computation inside the reference cell, and returns a new thunk value.

```
pullWith :: (V -> M a) -> Thunk V -> M a
pullWith f ref = do { v <- down ref; f v }

down :: Ref V -> FIO V
down ref =
  do { x <- readRef ref
     ; case x of
         Left m ->
           do { v <- m
              ; u <- case v of
                      Vthunk r2 -> down r2
                      u -> return u
              ; writeRef ref (Right u)
              ; return u }
         Right v -> return v
     }
```

The expression `(pullWith f ref)` applies `f` to the value stored in the reference `ref`. Forcing and overwriting it with its value if it is still latent, or directly applying `f` if it has been previously forced. The normal pattern of use of `pullWith` is as follows:

```
f: V -> M a
f (Vthunk ref) = pullWith f ref
f pattern = ...
```

Some function `f`, from value to computation, is applied to a value. `f` has several clauses, one for thunks, and others for its normal operation. Normal operation of `f` requires inspecting the value (often by some sort of pattern match). To perform this inspection `pullWith` forces the thunk inside `ref`, overwrites the reference and (recursively) applies `f` to the resulting value. The forcing, and overwriting are captured in the function `down`.

The interpreter is a standard interpreter for a call by value language. It can be found in its entirety in Appendix A. We will only discuss the pertinent parts that relate to explicit laziness, as everything else is completely standard.

3.1 Interaction with Thunks

Only minor accommodations were made to handle explicit laziness. In the function position of an application, a function value (`Vfun f`) is expected, but a thunk may occur.

```
eval env (App f x) =
  do { g <- eval env f
     ; y <- eval env x
     ; applyV g y }
```

The `applyV` function `applies` the function value in its first argument, to its second argument. If the first argument is an unexpected thunk, the `pullWith` function handles this complication using the generic scheme described above.

```
applyV :: V -> V -> M V
applyV (Vthunk m) x = pullWith (\ v -> applyV v x) m
applyV (Vfun h) x =  h x
applyV z y = fail ("Non function in application")
```

The only other accommodation for thunks is in pattern matching. If a pattern is to be matched against a thunk, the thunk must be forced before matching can proceed. The meta-function `mPat` implements pattern matching in the interpreter. It performs a simultaneous analysis over a pattern and a value, adding bindings to an environment fragment, which consist of pairs of variables appearing in the pattern and values. If during this analysis a thunk value is matched against, the function `pullWith` handles this as described above.

```
mPat :: Pat -> V -> EnvFrag -> M (Maybe EnvFrag)
mPat (Pvar s) v es = return(Just((s,v):es))
mPat p (Vthunk r) es = pullWith (\ v -> mPat p v es) r
mPat p ... -- additional cases
```

Note that the clause for matching a pattern variable (`mPat (Pvar s) v es = ...`) precedes the clause for thunks (`mPat p (Vthunk r) es = ...`) as a variable can be bound to a thunk, which need only be "pulled on" when the value of the variable is demanded.

3.2 Implementing the Lazy Annotations

There are three laziness annotations. One language construct `lazy`, and two primitive first class functions: `mimic` and `strict`. The language construct `lazy` annotates those parts of the program that should be evaluated lazily. The annotation is implemented as the case in the `eval` function for the `Lazy` syntax constructor.

```
eval env (Lazy x) = thunk (eval env x)
```

If a term `x` is to be evaluated lazily, wrap the computation that will compute it (`eval env x`) up in a thunk and return this thunk. Here we use the meta-function `thunk` described in Section 3.

The other two annotations can be implemented as functions which are present in the initial environment.

```
mimic :: V  -- prim mimic :: (a -> b) -> a -> b
mimic = Vfun(\ f -> return(Vfun (g f)))
  where g f (Vthunk m) = thunk(pullWith (g f) m)
        g f v = applyV f v
```

Recall that `mimic` takes two arguments, hence the structure `Vfun(\ f -> return(Vfun (g f)))`. But if the second argument of `mimic` is a thunk, we want the result of mimic to be a thunk as well. If the second argument is not a thunk, than just apply the first argument to the second.

The last annotation `strict` forces its argument if its a thunk.

```
strict :: V
strict = Vfun f1 -- prim strict :: a -> a
  where f1 (Vthunk m) = pullWith f1 m
        f1 v = return v
```

4. DISCUSSION

This paper is about a rather simple idea: that the user can use explicit laziness annotations to indicate where laziness is necessary. This allows the implementation to use a simple and efficient strict call-by-value default evaluation order on the rest of the program.

While our implementation is just a toy, this strategy should be easy to implement on top of the runtime system of existing lazy languages. Such languages must already distinguish thunks from values, and force them when necessary. Using explicit laziness requires the same run-time system as an ordinary lazy language, but needs a considerably simpler compiler, since strictness analysis is not required.

The biggest surprise to the author was how simple it was to write programs where laziness must be explicitly annotated. The key to the effective use of explicit annotation was the function `mimic` which maintains existing laziness. We believe `mimic` is original to this work.

Lastly we want to say a word about libraries. One could imagine that library writers might rely heavily on the use `mimic` because they do not know which arguments might be computations rather than values. For example, in Section 2 we suggested that the `filter` function might be written as:

```
filter p [] = []
filter p (x:xs) = if mimic p x
                    then x : mimic (filter p) xs
                    else mimic (filter p) xs
```

Anticipating that the elements of the list might also be lazy. While this might be the case in other library functions, it is not the case in `filter`. Note that the element must be fully evaluated if `p` is to return a `Bool`, so no `mimic` in the test of the `if` is needed. Of course other library functions might not be so straight-forward so we expect liberal use of `mimic` in library functions. The cost of this depends upon the cost of using the mimic function. In our interpreter the cost of using mimic is relatively cheap, only slightly more costly than the cost of using actual application.

5. RELATED WORK

A summary of the related work can be found in Figure 2 we compare a number of systems along two dimensions. The First dimension is whether or not the system requires a type distinction between lazy and eagerly evaluated computations. The second dimension is what type of annotations are required by the user.

The use of a type distinction often allows more efficient implementation as dynamic tests to distinguish the two do not have to be performed. The drawback of this distinction is that it is harder to use. The programmer must track these types, and multiple versions of some functions must be written to accommodate varying degrees of laziness in their arguments. We now discuss some individual systems.

Speculative Evaluation. Some of the earliest work on adding strictness to a lazy language was done by Karl-Filip Faxen [4, 5]. He used a *whole-program* flow analysis to identify which computations would always be needed, or be cheap to compute. The compiler then arranges to speculatively evaluate such computations eagerly. The draw back with this approach is that it can only be applied if the whole program is available for analysis, and speculative evaluation must be decided statically.

	Type distinction	No type distinction
No Annotations		A. Haskell – Lazy evaluation only B. Faxen – Requires whole program analysis. C. Maessen – Uses dynamic information, significant slowdowns in some programs. D. Ennals & Peyton Jones – Uses better dynamic information. Asymptotic improvements over A in speed and size of space leaks. Complicated to implement.
Just delay		E. This paper – Convenient to use, possible asymptotic improvements over A., plugs arbitrary large space leaks, Simpler to implement than B, C, or D. Requires programmer annotations.
Delay & force	G. Wadler, MaQueen, and Taha – efficient, no eval check, no need for boxed types, but at most a constant improvement over E, but less convenient than E.	F. Scheme – No types
Just Force		H. Haskell – with use of "seq"

Figure 2: Comparison of systems with Mixed evaluation Strategies.

Optimistic Evaluation. Recent work by Jan-Willem Maessen [10, 11] and Robert Ennals and Simon Peyton Jones [3] also uses a hybrid lazy-eager evaluation strategy that relaxes the whole program restriction by using run-time information to trigger optimistic evaluation.

The basic difference is that all evaluation is eager – periodic suspends are inserted which return evaluation to a previous context. With eager evaluation great care must be taken not to abort too often, or too infrequently, and run-time information is used to make these decisions. Additionally, special care must be taken with handling computations which raise errors (e.g., the standard Haskell function `error`): a new kind of thunk must be introduced that can be forced only on resumption of computation.

Maessen [10, 11] describes important speedups on some programs, and significant slow downs on others. Robert Ennals and Simon Peyton Jones, describe similar results by evaluating most computations eagerly and backing off only when computations appear to be diverging [3]. The system, built on top of the GHC compiler, is an engineering marvel, combining a number of dynamic techniques to measure and detect potential divergence. Measurements on the system measure considerable speedups on several Haskell benchmarks, and no large slow downs on any program in the benchmarks tested. They attribute this improvement over Maessen to more accurate dynamic information being used when "backing-off". The cost of this accurate information is large and complicated additions to the compiler and run-time systems, which itself adds significant overhead (though of a different kind) to the system. They remain optimistic that the situation can be further improved.

Using explicit laziness annotations we would expect a compiled implementation to attain similar (perhaps better) speedups as the overhead to detect divergence is not necessary. The solution we propose here is much simpler. It requires neither complicated additions to the runtime system, nor expensive compile-time analysis such as strictness or flow analysis. Instead, we rely on the programmer's knowledge of her own program to specify explicitly where laziness is essential.

Suspensions in Strict Languages. Adding laziness to a strict language is not new [6]. Most approaches involve adding an explicit *force* and *delay* constructs (e.g., [13, 8, 2]). The expression `delay e` delays the evaluation of expression e, usually converting it into an explicit thunk type:

```
val e = delay (2+3)
val e = - : int suspension
```

The expression *force* forces the evaluation of the thunk obtaining the value. The key difference between this approach and the current paper is the type constructor *suspension* which partitions lazy computations from strict ones, and the need to explicitly coerce lazy computations into ordinary ones by use of the *force* function.

While the type distinction is inconvenient for the programmer, requiring the insertion of explicit `force`, and possible the duplication of library functions with different laziness distinctions in their parameters, it does allow the compiler a big win. Boxed types are never needed. This allows the compiler to take advantage of representation issues, and could translate into a considerable advantage in speed over the current proposal, for a modest loss in convenience and expressivity.

In some languages, such as Standard ML, it is possible to define the suspension type in the language itself, using dummy functions that do not examine their arguments for delaying computation and references to avoid repeated computation of delayed values. The technique is fairly standard [15], but can be rather awkward to use, as the user must manage all the low-level machinery needed for call-by-need evaluation explicitly, with no linguistic support, for each lazy data-type. Chris Okasaki proposed an intuitive syntax for suspended computations in strict languages [12, 13].

A more extensive integration of lazy features into a strict language was incorporated into some versions of the Standard ML of New Jersey compiler by Wadler, Taha, and MacQueen [9]. They first distinguish between two styles of lazy programming in a strict language, called *odd*, and *even*. The even style is closer to laziness as understood by a Haskell user: making a data-type `T` even-lazy means to define an auxiliary data-type `T'`, which is the same as `T`, except that the recursive the sub-structures are delayed (using a `suspension` type). Finally, the data-type corresponding to the original `T` is the suspension of `T'`. The odd-style, by contrast, works with the changed data-type definition `T'` directly, which may lead to too much strictness [9]. It's relative conciseness in programming lead to its wider adoption than the odd style [1, 14, 13].

The even style, while more intuitively lazy, leads to more elaborate and less clear definitions. To remedy this, a new modifier keyword `lazy` is added to SML. For each function defined with the keyword `lazy`, two functions (alternatively, data-types, `val` declarations and so on) are generated: one operating on a top-level suspended computation, and the other operating on a lazy computation "once unrolled," where the substructures being defined are again suspended.

The user need not see the two functions; the compiler uses types to determine which of the two versions to use. This approach, however, seems not to have gained a significant popularity among the users. One of the reasons is perhaps that the syntactic preprocessing of lazy programs does not interact very well with error reporting and introduces new names to the program of which the user may not be aware. By contrast, the interface to lazy values proposed in this paper is explicit in creating thunks, but implicitly forces them when demanded. It does not rely on any preprocessing or other source-to-source transformation of the code written by the programmer.

6. CONCLUSION

While only a simple thought experiment, tested with a simple interpreter, the author finds the idea of explicit laziness compelling enough to merit further review, and encourages others to investigate this exciting new approach.

7. ACKNOWLEDGMENTS

The work described here was supported by NSF Grant CCR-0098126, the M.J. Murdock Charitable Trust, and the Department of Defense.

8. REFERENCES

[1] H. Abelson and G. J. Sussman. *The Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] W. Clinger and J. R. (editors). Revised report on the algorithmic language scheme. Technical Report AIM-848B, MIT Artificial Intelligence Laboratory, Nov. 6 1991.

[3] R. Ennals and S. Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. 2003. Submitted to ICFP'03.

[4] K.-F. Faxen. Optimizing lazy functional programs using flow inference. *Lecture Notes in Computer Science*, 983:136–??, 1995.

[5] K.-F. Faxen. Cheap eagerness: speculative evaluation in a lazy functional language. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 150–161, N.Y., Sept. 18–21 2000. ACM Press.

[6] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, 1976.

[7] D. J. King and J. Launchbury. Structuring depth-first search algorithms in Haskell. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 344–354, New York, NY, USA, 1995. ACM Press.

[8] X. Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[9] D. B. MacQueen, P. Wadler, and W. Taha. How to add laziness to a strict language withouth even being odd. In *Proceedings of the 1998 ACM Workshop on ML*, pages 24–30, Sept. 1998. Baltimore, MD.

[10] J.-W. Maessen. Eager haskell: Resource-bounded execution yields efficient iteration. In *The Haskell Workshop, Pittsburg 2002*.

[11] J.-W. Maessen. *Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell.* PhD thesis, MIT, 2002.

[12] C. Okasaki. Functional data structures. *Lecture Notes in Computer Science*, 1129:131–??, 1996.

[13] C. Okasaki. *Purely Functional Data Structures.* PhD thesis, Carnegie Mellon University, 1996. Available as Technical Report CMU-CS-96-177.

[14] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[15] C. Reade. *Elements of Functional Programming.* International Computer Science Series. Addison-Wesley, 1989. Pages 265–316.

APPENDIX

A. THE INTERPRETER

In this section we give the complete code of the interpreter.

```
---------------------------------
-- Syntax

data Pat
  = Plit Int            -- { 5 }
  | Pvar String         -- { x }
  | Pprod Pat Pat       -- { (p1,p2) }
  | Pcon String [Pat]   -- data constructors like:
                        -- (x:xs) or (Node x y)

data Exp
  = Var String          -- { x }
  | Lit Int             -- { 5 }
  | Prod Exp Exp        -- { (e1,e2) }
```

```
  | App Exp Exp        -- { f x }
  | Lam [Pat] Exp      -- { \ p1 p2 -> e }
  | Lazy Exp           -- { (lazy e)
  | Let String Exp Exp -- { let x = e in x+1 }
  | Case Exp [(Pat,Exp)]  -- { case e of m1; m2 }

---------------------------------------
-- Values

data V
  = Vlit Int           -- Int constants
  | Vprod V V          -- products
  | Vfun (V -> M V)    -- functions
  | Vcon String [ V ]  -- data like (Node x y)
  | Vthunk (Thunk V)     -- thunks

---------------------------------------
-- Thunks

thunk :: M V -> M V
thunk c = do { r <- newRef (Left c); return(Vthunk r) }

pullWith :: (V -> M a) -> Thunk V -> M a
pullWith f ref = do { v <- down ref; f v }

down :: Thunk V -> FIO V
down ref =
  do { x <- readRef ref
     ; case x of
         Left m ->
           do { v <- m
              ; u <- case v of
                       Vthunk r2 -> down r2
                       u -> return u
              ; writeRef ref (Right u)
              ; return u }
         Right v -> return v
     }

force :: (V -> M a) -> V -> M a
force f (Vthunk m) = pullWith f m
force f v = f v


---------------------------------
-- Environments

data Env = Env [(String,V)]
type EnvFrag = [(String,V)]

find :: Monad a => Env -> [Char] -> a V
find (Env xs) s =
       case lookup s xs of
        Just b -> return b
        Nothing -> fail ("Unknown var: "++s)

extendV :: EnvFrag -> Env -> Env
extendV xs (Env ys) = Env (xs++ys)

------------------------------------
-- The interpreter

eval :: Env -> Exp -> M V
eval env (Var s) = find env s
eval env (Lit x) = return(Vlit x)
eval env (Prod x y) =
  do { a <- eval env x
     ; b <- eval env y
     ; return(Vprod a b) }
eval env (Lazy x) = thunk (eval env x)
eval env (App f x) =
  do { g <- eval env f
     ; y <- eval env x
     ; applyV g y }
eval env (Lam ps e) = try [] ps   where
  try es [] = eval (extendV es env) e
  try es (p:ps) = return (Vfun h) where
      h v = do { z <- mPat p v es
               ; case z of
                  Just es2 -> try es2 ps
                  Nothing -> bad p v }
  bad p v = fail ("Pattern does not match value")
eval env (Let s e1 e2) =
  do { v <- fixM (\ u -> eval (extendV [(s,u)] env) e1)
     ; eval (extendV [(s,v)] env) e2 }
eval env (Case x ms) =
  do { v <- eval env x; caseV env v ms }
 where caseV env v [] = fail "value doesn't match any pattern"
       caseV env v ((p,body):ps) =
         do { z <- mPat p v []
            ; case z of
               Nothing -> caseV env v ps
               Just es -> eval (extendV es env) body }

----------------------------------------
-- Manipulating Values (which might contain thunks)

applyV :: V -> V -> M V
applyV (Vthunk m) x = pullWith (\ v -> applyV v x) m
applyV (Vfun h) x =  h x
applyV z y = fail ("Non function in application")

mPat :: Pat -> V -> EnvFrag -> M (Maybe EnvFrag)
mPat (Pvar s) v es = return(Just((s,v):es))
mPat p (Vthunk r) es = pullWith (\ v -> mPat p v es) r
mPat (Plit x) (Vlit y) es =
  if x==y then return(Just es) else return Nothing
mPat (Pprod x y) (Vprod u v) es =
  do { z <- mPat x u es
     ; case z of
        Just es2 -> mPat y v es2
        Nothing -> return Nothing}
mPat (Pcon n ps) (Vcon c vs) es =
  if n==c
     then mPatL ps vs es
     else return Nothing
mPat p v es = fail ("Pattern does not match value")

mPatL :: [Pat] -> [V] -> EnvFrag -> M (Maybe EnvFrag)
mPatL [] [] es = return(Just es)
mPatL (p:ps) (v:vs) es =
   do { z <- mPat p v es
      ; case z of
          Just es2 -> mPatL ps vs es2
          Nothing -> return Nothing}
mPatL _ _ es = return Nothing

------------------------------------------------
-- primitives

mimic :: V  -- prim mimic :: (a -> b) -> a -> b
mimic = Vfun(\ f -> return(Vfun (g f)))
  where g f (Vthunk m) = thunk(pullWith (g f) m)
        g f v = applyV f v

strict :: V
strict = Vfun f1 -- prim strict :: a -> a
  where f1 (Vthunk m) = pullWith f1 m
        f1 v = return v

plus :: V
plus = Vfun f1   -- prim plus :: Int -> Int -> Int
  where f1 (Vthunk ref) = pullWith f1 ref
        f1 (Vlit n) = return(Vfun (f2 n))
        f2 n (Vthunk ref) = pullWith (f2 n) ref
        f2 n (Vlit m)     = return(Vlit(n+m))
```