Laziness and Pairs
Shriram Krishnamurthi
2001-09-28

1 Prelude

Our discussion of recursive procedures concluded with an admonishment: always place syntactic values on the RHSes of recursive declarations. That is, if you write

\[
\text{rec}\ \{f\ f\}
\]

you’re likely to run afoul of the evaluator: this could result in an infinite loop, an unusable value, an error, or any number of other things. The problem, recall, arises from trying to evaluate the RHS expression prematurely.

This suggests that the problem may not be the program so much as the act of evaluating prematurely. What if we were to wait and evaluate the RHS only when we use it in the body? This harkens back to a lazy language. We already know that \(\text{let}\ \{x\ 3\}\ x\) is just an abbreviation for \(\{\text{proc}\ \{x\}\ x\}\ 3\), where the RHS expression of the \text{let} (3, in this case) is essentially the argument to a procedure. In a lazy language, we would evaluate the 3 only when we refer to \(x\) in the body. By analogy, perhaps we should treat the RHS of a \text{rec} the same way.

In the example above, this immediately buys us some predictability. By the time we evaluate the body, the \textbf{f} on the RHS is bound to itself. Asking for its value in the body should send the evaluator into an infinite loop (of constantly looking up the value of \textbf{f}). Perhaps it’s not the most useful of programs (to put it mildly), but at least we know what it does.

Before we go too far, we should make sure we understand how lazy evaluation works. Let’s do that first. Of course, we’ll do it through interpreters.

2 Implementing Laziness

2.1 Procedures

Let’s first examine a sample program. Consider:

\[
\text{let}\ \{x\ 3\}
\]
\{let \{f \{proc \{y\} \{+ x y\}\}\}\{let \{x 5\}\{f 10\}\}\}\}

This program should evaluate to 13. Note that eagerness and laziness have nothing to do with the outcome. Static scoping dictates that \(x\) inside the procedure refer to the binding that associates \(x\) with 3. When we evaluate the argument, 10, makes no difference to its value. Hence the answer.

Consider this slightly modified program:
\{let \{x 3\}\{let \{f \{proc \{y\} \{+ x y\}\}\}\{let \{x 5\}\{f x\}\}\}\}

This is more interesting. There are three possible answers: 6, 8 and 10. The last one is clearly unacceptable, because it would mean violating static scoping. The other two both seem like distinct possibilities. An eager interpreter would reduce this to 8, but what should a lazy interpreter do? If it evaluates the argument \(x\) in the environment of the procedure’s body, then \(x\) is 3, not 5. Which result should it produce?

The answer ought to be: the same result as the eager interpreter. Just because we evaluate the argument \(x\) later doesn’t mean its value should change! Otherwise this would be just like dynamic binding, except in reverse. It would make it impossible for the user to maintain any predictability over the function’s behavior, since this would depend entirely on the variables that just happened to be bound at the procedure’s declaration.

The simple fix appears to be to modify the interpretation of \(appE\):
\[(appE? expr) (do-app (interp (appE-proc expr) dsub) (appE-arg expr))]\]

This, after all, seems to be the essence of lazy evaluation: to not evaluate the argument at applications! Unfortunately, there are two things wrong with this simplistic modification. The first, which points us in the direction of the second, is that this violates contracts. By making this change, we give \(do-app\) an \(RP\) as a second argument, rather than a value.

The second problem is as follows. Suppose we were to modify \(do-app\) and \(extend-dsub\) to accept \(RPs\). Then \(y\) would be bound to \((make-varE \ ‘x’\)\). Unfortunately, this records no mention of the fact that \(x\)’s value must really be from the dynamic environment active at the time of the procedure invocation, not the static one.

When we ran into the opposite problem when implementing statically scoped procedures, we addressed it by introducing \(closures\). This suggests we should do the same, except in reverse. We therefore introduce a new type of value, the \(expression\ closure\) (we could also have called it a \(dynamic\ closure\):\footnote{The technical name for an expression closure is a \textit{thunk}. That name obviously makes a lot more sense.}
(define-struct exprclos (expr dsub))
;; expr : RP
;; dsub : dsub
Whenever we pass an expression around unevaluated, we first wrap it in an expression closure, which carries along the environment at the time the expression would have been evaluated in an eager interpreter. (Do you see the analogy to closures?) Specifically,

\[
[(\text{appE}\? \text{expr}) (\text{do-app} (\text{interp} (\text{appE-proc} \text{expr}) \text{dsub})
(\text{make-exprclos} (\text{appE-arg} \text{expr}) \text{dsub}))]
\]
which captures the dynamic environment.

Note that an expression closure takes the place of a value. In an eager language, we would have boldly interpreted the argument, leaving no need for expression closures. We’re deferring interpretation now, which means whatever takes its place should be considered a kind of value (ripe for later interpretation, and hence reduction to a value). Therefore, a value is either a \text{numE}, a \text{closure} or a \text{exprclos}.

What happens to expression closures? They hang around in the environment, waiting to be used. We don’t see them again until we try to extract the value of a variable. When we do, the environment returns an \text{exprclos} value. We’re finally ready to use its value, but first we must reduce it to an answer. Therefore,

\[
[(\text{varE}\? \text{expr}) (\text{reduce} (\text{lookup-dsub} \text{expr} \text{dsub}))]
\]
where

\[
;; \text{reduce : exprclos} \rightarrow \text{value}
\]
\[
\textbf{(define} \text{reduce} \text{v})
\quad (\text{interp} (\text{exprclos-expr} \text{v}) (\text{exprclos-dsub} \text{v}))
\]
That is, we extract an expression and an environment. The expression is the one passed as an argument to the procedure. The environment gives values for any names in this expression. We then interpret the former in the latter—exactly as we do the body of a closure in its environment.

These are the only changes our interpreter requires. Everything else, including \text{do-app} and the other parts of the interpreter, remain untouched.

### 2.2 Relationship Between Eagerness and Laziness

In the preceding text, we’ve hinted that perhaps an eager and lazy interpreter should return the same value always. This has some practical implications. For instance, it suggests we can’t tell whether our implementation is using a lazy or eager semantics. This seems rather unsettling.

Well, it might \textit{seem} like we said they should produce the same value, but we didn’t quite say that. Read it again more carefully. What we actually said (implicitly) is that they should return the same value \textit{if they return a value at all}. 
That seems like an odd phrase. What could it mean? Well, remember: the essence of laziness is that we defer the execution of phrases. How long can we possibly defer it? How about indefinitely? In this example program, we never evaluate the argument, 10:

\{\text{proc \{x\} 5\} 10\}

Big deal, you might think. These two programs still produce the same answer in both a lazy and an eager discipline. Perhaps you write something more elaborate as the argument:

\{\text{proc \{x\} 5\} (+ 5 (* (+ 2 1) 3))\}

Okay, they still return the same value. But in the eager case, it takes a lot longer to produce that same answer, because it must first reduce the argument to a value. The lazy evaluator dispenses with this formality, and returns 5 right away.

Naturally, we should ask, exactly how long can an argument take to evaluate? Could it take forever? If it could, then we’d really be on to something. Suppose \text{INF} were a special instruction we add to Rip; evaluating it goes into an infinite loop. Then,

\{\text{proc \{x\} 5\} INF\}

would produce 5 in the lazy discipline, but no answer at all in the eager one. That looks pretty interesting!

**Puzzle.** Can you write an infinite loop using just functions and variables in Rip? (Obviously, you can once you also have \text{rec: \{ref \{f \{proc \{x\} \{f x\}\} \{f 5\}\} \{f 5\}\} does the trick just fine.)

### 2.3 Recursion

Given the modifications we had to make to accommodate procedures, it seems natural that we must change something to support recursion in a lazy language: recall the analogy between \text{let} and \text{rec}. We might initially be tempted to leave \text{circularly-bind} unchanged. Since we don’t want to evaluate the RHS of the \text{rec} until necessary, we might instead write

\[(\text{recE? expr}) (\text{interp (recE-body expr)})\]

\[(\text{circularly-bind dsub})\]

\[(\text{(recE-var-name expr)})\]

\[(\text{(make-exprclos (recE-val expr) dsub))})]\]

This appears to be the right analog to our treatment of application.

Trying this on an example such as \{\text{ref \{f \{f \{f \}\}\}}\} immediately demonstrates that this doesn’t seem to do the trick: we’re told that \text{f} isn’t bound! How could that be? In the eager implementation, we don’t determine the environment for the RHS until we’re inside \text{circularly-bind}, at which point we use the new,
extended (and partially-correct) environment we’ve just built. Here, however, we’re prematurely closing the RHS over the \texttt{dsub} active outside the \texttt{rec} expression, so when we try to look up \texttt{f}, we don’t find it.

The solution is to actually leave the interpreter unchanged:

\[
\left( \texttt{recE? expr} \right) \left( \texttt{interp \ (recE-body expr)} \right)
\left( \texttt{circularly-bind \ dsub} \right)
\left( \texttt{recE-var-name expr} \right)
\left( \texttt{recE-val expr} \right) \right)
\]

and change \texttt{circularly-bind} instead:

\[
\begin{array}{l}
\texttt{(define \ (circularly-bind \ env \ var-name \ valE)} \\
\texttt{\ (let \ ((\text{new-env} \ \text{cons} \ \text{make-binding} \ \text{var-name} \ \texttt{’undefined} \ \text{env}))} \\
\texttt{\ (begin} \\
\texttt{\ \ (set-binding-val! \ (first \ new-env) \ \text{make-exprclo} \ \text{valE} \ new-env) } \\
\texttt{\ new-env))})
\end{array}
\]

Note that we’re closing the RHS expression in a \texttt{exprclo} over the \texttt{new} environment. By the time we get around to using it, the environment will have been mutated to have the correct value bound to the new variable.

Using this interpreter to evaluate \texttt{\{rec \ \{f \ f\} \ f\}} results in an infinite loop.

## 3 Adding Pairs as Data

Adding pairs is unexciting in an eager language and interesting in a lazy one.

### 3.1 Preliminaries

First, we extend the data definition for Rip programs:

A Rip Program (\texttt{RP}) is either

- \texttt{(make-numE \ (\text{num})} \\
- \texttt{(make-addE \ RP \ RP)} \\
- \texttt{(make-procE \ var-name \ RP)} \\
- \texttt{(make-appE \ RP \ RP)} \\
- \texttt{(make-recE \ var-name \ RP \ RP)} \\
- \texttt{(make-pairE \ RP \ RP)} \\
- \texttt{(make-firstE \ RP)} \\
- \texttt{(make-secondE \ RP)}

(We’ll continue to assume \texttt{let} gets expanded away.)
3.2 Eager Semantics

In the eager version, \texttt{make-pairE} evaluates both its arguments, and creates a value of type
\begin{verbatim}
(define-struct rippair (first second))
;; first : value
;; second : value
\end{verbatim}
where we assume that \texttt{rippair} enters the set of Rip values. \texttt{first} and \texttt{second} extract the first and second projections of \texttt{rippair} pairs.

3.3 Lazy Semantics

What happens when we evaluate a \texttt{pair} expression? The pair constructor is a procedure, so it presumably should delay the evaluation of its arguments. This means the interpreter should read
\begin{verbatim}
[(pairE? expr) (make-rippair (make-exprclos (pairE-first expr) dsub) (make-exprclos (pairE-second expr) dsub))]
\end{verbatim}
(recall that an \texttt{exprclos}, which is merely a surrogate for a value, is a value, too, so this doesn’t violate any contract).

How about the accessors? In a legal program, \texttt{first} (and, similarly, \texttt{second}) consumes only pairs. Nothing else consumes pairs. As a result, the argument to \texttt{first}, after interpretation, is still a \texttt{exprclos}. We can’t extract the first value in a pair without first knowing what the pair contains. We must therefore force the \texttt{exprclos} to divulge its value. We already have a procedure that does this: \texttt{reduce}. Once we have reduced them to values, we are free to project their fields. Therefore,
\begin{verbatim}
[(firstE? expr) (rippair-first (reduce (interp (firstE-arg expr) dsub)))]
[(secondE? expr) (rippair-second (reduce (interp (secondE-arg expr) dsub)))]
\end{verbatim}

Actually, we need to modify the definition of \texttt{reduce} slightly. Interpreting a pair construction evaluates to a \texttt{rippair}; it’s the individual fields of a \texttt{rippair} that \texttt{reduce} must handle (consider the program \{\texttt{first \{pair 1 2\}}\}: the \texttt{exprclos}es wrap the numbers). In general, \texttt{reduce} needs to consume any value and rid it of enough \texttt{exprclos}es for interpretation to continue.\footnote{We have written a very liberal \texttt{reduce} that eliminates all \texttt{exprclos}. Some language designers prefer to eliminate them only as necessary, one “level” at a time.}

\begin{verbatim}
;; reduce : value \rightarrow value
;; reduce reduces all \texttt{exprclos} to values of the other three kinds
(define (reduce v) (cond
\end{verbatim}
Again, these are the only changes we need to make to the interpreter to support pairs in a lazy language.

4 Lazy Evaluation and Cyclic Data

4.1 Construction

We’ve already established, several times over, that \( \{\text{rec } \{f \ f\} \ f\} \) is a pretty useless expression in both a lazy and eager semantics. Similarly, the program

\[
\{\text{rec } \{\text{ones} \ \{\text{pair} \ 1 \ \text{ones}\}\} \\
\quad \text{ones}\}
\]

is equally useless in an eager semantics: it either loops forever or tries to pair a 1 with some junk value, which we detect when we try to do anything with the second projection of the pair.

In a lazy semantics, however, this definition takes on a new life. Recall that we don’t examine the value of the RHS until we’re in the body. We now know exactly what this means: we create an \textit{exprclos} to hold on to the extended, cyclic environment, and we evaluate the RHS in this \textit{exprclos}'s stored environment when we look up the bound name. In the above example, this means we bind \textit{ones} to

\[
(\text{make-exprclos} \ (\text{make-pairE} \ (\text{make-numE} \ 1) \ (\text{make-varE} \ \text{'ones})) \\
\quad \text{Env})
\]

where \textit{Env} is an environment that binds \textit{ones} (representing \textit{ones}) to the very \textit{exprclos} above.

What happens when we try to project the \textit{first} element of this pair? This interprets the body of the \textit{exprclos} in its stored environment to produce a pair, whose first value is obviously 1. Asking for the second projection results in evaluating \textit{ones} in the stored environment, \textit{Env}. Recall that this just gives us the same \textit{exprclos} that we began with a moment ago. Therefore, asking for \{\textit{first} \ \textit{second} \ \textit{ones}\} also yields 1, as indeed does nesting any number of calls to \textit{second}. In short,

\textit{ones} is a truly \textit{infinite} sequence of pairs.

By not evaluating the arguments to \textit{pair}, we have deferred the construction to the point where the actual value we’ve built is infinitely large. We can store and manipulate this in a computer because at any point in time, we can only have examined a finite prefix of it. Nevertheless, by building the infinite object, we’ve avoided having to commit a priori to any finite size bound on it.
In short, cyclic data structures generate something more than just trees of values: they generate graphs. We call an infinite list of value a stream.

5 Applications

There are several extremely elegant uses for infinite data streams and graphs (of which infinite streams are just a special case). We’ll present just a few of them.

1. A digital signal is most easily represented using these data structures: it’s a stream of electrical levels. A simple digital circuit is a tree of components (as you recall from evaluating simple boolean circuits), each of which consumes and produces a stream, but when the circuit contains feedback, it becomes a graph.

2. In general, of course, graphs arise all over computer science. Lazy data structures help us build graphs directly and expressively, without mucking with their underlying representation.3

3. Unix pipes (1) rely fundamentally on infinite streams (of layout-ordained strings). The producer end of a pipe blocks on writing (in theory, anyway) until the consumer end of the pipe reads its past output. That is, the last element in a pipe drives all elements behind it, until the first element. If the last one wants to read nothing, the previous elements would produce nothing. To see a stream in action, type yes at a Unix prompt.

4. Finally, we can return to where we began. Our exploration of recursion had to fit a very specific bill. We tried to construct recursive functions. This reduced to requiring cyclic environments. To build those, we jumped through various hoops, involving either a meta-circular solution with procedural representations (very subtle and not very informative) or a mutation-based solution with explicit structures (inelegant).

Here’s a third solution. Remember, we’re trying to build a cyclic data structure (the environment). In a hypothetical lazy language, we could simply write

\[
\text{rec \{E \{extend-env E0
\text{\{make-closure <procE>
E\}\}}\}}
\]

and make the programming language do all the heavy lifting of constructing the environment correctly! In other words, if we use a lazy language as the meta-language for an eager (or lazy) interpreted language, defining 3This is all well-and-good provided your graphs aren’t 2¹⁰⁰⁰ states large.
recursion becomes extremely easy. (Of course, if you didn’t understand how lazy data structures worked, you still wouldn’t understand what was happening to the environment . . . but you do understand them now!)

6 Coda

Lazy functional languages have been around for over two decades. About a decade ago, the implementors of the various lazy languages decided to band forces, producing a single, unified effort called Haskell (the first name of Curry). Haskell is elegant and charming, and sees use for everything from graphics to robotics. We’ve only seen a fraction of its appeal. Learn more about it from

http://www.haskell.org/