Chapter 11

Implementing Laziness

Now that we've seen Haskell and shell scripts at work, we're ready to study the implementation of laziness. That is, we will keep the syntax of our language unchanged, but alter the semantics of function application to be lazy.

11.1 Implementing Laziness

Consider the following expression:

{with {x {+ 4 5}}
  {with {y {+ x x}}
    {with {z y}
      {with {x 4}
        z}}}}

Recall that in a lazy language, the argument to a function—which includes the named expression of a with—does not get evaluated until use. Therefore, we can naively think of the expression above reducing as follows:

{with {x {+ 4 5}}
  {with {y {+ x x}}
    {with {z y}
      {with {x 4}
        z}}}} [x -> {+ 4 5}]

= {with {y {+ x x}}
  {with {x 4}
    {with {z y}
      z}}} [x -> {+ 4 5}]

= {with {x 4}
  {with {z y}
    z}} [x -> {+ 4 5}, y -> {+ x x}]
= {with {z y} 
  z}                 [x -> 4, y -> {+ x x}]
= z                     [x -> 4, y -> {+ x x}, z -> y]
= y                     [x -> 4, y -> {+ x x}, z -> y]
= {+ x x}               [x -> 4, y -> {+ x x}, z -> y]
= {+ 4 4}               
= 8

In contrast, suppose we used substitution instead of environments:

{with {x {+ 4 5}}
 {with {y {+ x x}}
  {with {z y}
    {with {x 4}
      z}}}}
= {with {y {+ {+ 4 5} {+ 4 5})){
  {with {z y}
    {with {x 4}
      z}}}
= {with {z {+ {+ 4 5} {+ 4 5}}}}
  {with {x 4}
    z}}
= {with {x 4}
    {+ {+ 4 5} {+ 4 5}}}
= {+ {+ 4 5} {+ 4 5}}
= {+ 9 9}
= 18

We perform substitution, which means we replace identifiers whenever we encounter bindings for them,
but we don’t replace them only with values: sometimes we replace them with entire expressions. Those
expressions have themselves already had all identifiers substituted.

This situation should look highly familiar by now: this is exactly the same problem we encountered
when trying to correctly define functions as values. Substitution produces the answer that we should take
to be the definition of what a program should produce. Using environments to defer substitutions, however,
sometimes (inadvertently) changes the behavior of the program.

The way we addressed this problem before was to use closures. That is, the text of a function was closed
over (i.e., wrapped in a structure containing) its environment at the point of definition, which was then used
when evaluating the function’s body. The difference here is that we must create closures for all expressions
that are not immediately reduced to values, so their environments can be used when the reduction to a value
actually happens.

We shall refer to these new kinds of values as expression closures. Since they can be the result of
evaluating an expression (as we will soon see), it makes sense to extend the set of values with this new kind of
value. We will also assume that our language has conditionals (since they help illustrate an some interesting
11.1. IMPLEMENTING LAZINESS

points about laziness). Thus we will define the language CFAL (where the L will denote “laziness”) with the following grammar:

\[ \text{<CFAL>} ::= \text{<num> } \]
\[ | \{+ \text{<CFAL>} \text{ <CFAL>}\} \]
\[ | \{- \text{<CFAL>} \text{ <CFAL>}\} \]
\[ | \text{id} \]
\[ | \{\text{fun} \{\text{id}\} \text{ <CFAL>}\} \]
\[ | \{\text{<CFAL>} \text{ <CFAL>}\} \]
\[ | \{\text{if0} \text{ <CFAL>} \text{ <CFAL>} \text{ <CFAL>}\} \]

Observe that the eager counterpart of this language would have the same syntax. The difference lies entirely in its interpretation. As before, we will continue to assume that \texttt{with} expressions are converted into immediate function applications by the parser or by a pre-processor.

For this language, we can define the extended set of values:

\[
\begin{align*}
\text{(define-type CFAL-Value} \\
& \quad \text{[numV (n number?)]} \\
& \quad \text{[closureV (param symbol?)} \\
& \quad \quad \text{(body CFAL?)} \\
& \quad \quad \text{(sc SubCache?)]} \\
& \quad \text{[exprV (expr CFAL?)} \\
& \quad \quad \text{(sc SubCache?)])}
\end{align*}
\]

That is, a \texttt{exprV} is just a wrapper that holds an expression and the environment of its definition.

What needs to change in the interpreter? Obviously, procedure application must change. By definition, we should not evaluate the argument expression; furthermore, to preserve static scope, we should close it over its environment:

\[
\begin{align*}
\text{[app (fun-exp arg-exp]} \\
& \quad \text{(local ([define fun-val (interp fun-exp sc)]} \\
& \quad \quad \text{[define arg-val [(exprV arg-exp sc)]]} \text{])} \\
& \quad \text{(interp (closureV-body fun-val)} \\
& \quad \quad \text{(aSub (closureV-param fun-val)} \\
& \quad \quad \quad \text{arg-val} \\
& \quad \quad \quad \text{(closureV-sc fun-val))])}
\end{align*}
\]

As a consequence, an expression such as

\[
\{\text{with } \{x \ 3\} \\
\quad x\}
\]

will evaluate to some expression closure value, such as

\[
\text{#(|struct:expV| #(struct:num 3) #(struct:mtSub))}
\]

\[\text{\[1\]}\text{The argument expression results in an expression closure, which we then bind to the function’s formal parameter. Since parameters are bound to values, it becomes natural to regard the expression closure as a kind of value.}\]
CHAPTER 11. IMPLEMENTING LAZINESS

This says that the representation of the 3 is closed over the empty environment. We will return to the problem of presenting this more usefully later.

That may be an acceptable output for a particularly simple program, but what happens when we evaluate this one?

```
{with {x 3}
  (+ x x)}
```

The interpreter evaluates each x in the body to an expression closure (because that’s what’s bound to x in the environment), but the addition procedure cannot handle these. Indeed, the addition procedure (and similarly any other arithmetic primitive) needs to know exactly which number the expression closure corresponds to. The interpreter must therefore “force” the expression closure to reduce to an actual value. Indeed, we must do so in other positions as well: the function position of an application, for instance, needs to know which procedure to invoke. If we do not force evaluation at these points, then even a simple expression such as

```
{with {double {fun {x} {+ x x}}}
  (+ {double 5}
    {double 10}})
```

cannot be evaluated (since at the points of application, double is bound to an expression closure, not a procedural closure with an identifiable parameter name and body).

Because we need to force expression closures to values in several places in the interpreter, it makes sense to write the code to do this only once:

```
;; strict : CFAL-Value → CFAL-Value [excluding exprV]
(define strict e)
  (type-case CFAL-Value e
    [exprV (expr sc)
      (strict (interp expr sc))]
    [else e]))
```

Now we can use this for numbers,

```
(define (num+ n1 n2)
  (numV (+ (numV-n (strict n1)) (numV-n (strict n2))))
```

and similarly in other arithmetic primitives, and also for applications:

```
[app (fun-expr arg-expr)
  (local ([define fun-val (strict (interp fun-expr sc))])
    [define arg-val (exprV arg-expr sc)])
    (interp (closureV-body fun-val)
      (aSub (closureV-param fun-val)
        arg-val
        (closureV-sc fun-val))))]
```
The points where the implementation of a lazy language forces an expression to reduce to a value (if any) are called the *strictness* points of the language; hence the perhaps odd name, *strict*, for the procedure that annotates these points of the interpreter.

We can exercise the lazy portion of our interpreter in several ways. Consider the following simple example:

\[
{\text{with } \{ \text{f } \{ \text{undef x} \} \} \quad 4}
\]

Had the language been strict, it would have evaluated the named expression, halting with an error (that \texttt{undef} is not defined). In contrast, out interpreter yields the value 4.

There is actually one more strictness point in our language: the evaluation of the conditional. It needs to know the precise value that the test expression evaluates to so it can determine which branch to proceed evaluating. This highlights a benefit of studying languages through interpreters: failing this strictness test (but assuming good test cases!), we would quickly determine this problem. (In practice, we might bury the strictness requirement in a helper function such as \texttt{num-zero?}, just as the arithmetic primitives’ strictness is buried in procedures such as \texttt{num+}. We therefore need to trace which expression evaluations invoke such strictness-forcing primitives to truly understand the language’s strictness positions.)

Figure 11.1 and Figure 11.2 present the heart of the interpreter.

**Exercise 11.1.1** Did we need to add conditionals as a primitive construct in the language? A conditional (such as \texttt{if0}) serves two purposes: to make a decision about a value, and to avoid evaluating an unnecessary expression. Which of these does laziness encompass? Explain your thoughts with a modified interpreter.

**Exercise 11.1.2** Interactive Haskell environments usually have one other, extra-lingual strictness point: the top-level of the Interaction window. (How) Is this reflected in your interpreter?

### 11.2 Caching Computation

Evaluating an expression like

\[
{\text{with } \{ x \{ + 4 5 \} \} \quad {\text{with } \{ y \{ + x x\} \} \quad {\text{with } \{ z y \} \quad {\text{with } \{ x 4 \} \quad z}}}}\]

can be rather wasteful: we see in the hand-evaluation, for instance, that we reduce the same expression, \{+ 4 5\}, to 9 two times. The waste arises because we bind identifiers to expressions, rather than to their values. So whereas one of our justifications for laziness was that it helped us avoid evaluating unnecessary expressions, laziness has had a very unfortunate (and unforeseen) effect: it has has forced the re-evaluation of necessary expressions.
Let’s make a small change to the interpreter to study the effect of repeated evaluation. Concretely, we should modify `strict` to notify us every time it reduces an expression closure to a value:

```scheme
(define (strict e)
  (type-case CFAL-Value e
    [exprV (expr sc)
      (local ((define the-value (strict (interp expr sc)))))
      (begin
        (printf "Forcing exprV to \"a\"n\" the-value)
        the-value)])
    [else e]))
```

This will let us track the amount of computation being performed by the interpreter on account of laziness. (How many times for our running example? Determine the answer by hand, then modify `strict` in the interpreter to check your answer!)

Can we do better? Naturally: once we have computed the value of an identifier, instead of only using it, we can also cache it for future use. Where should we store it? The expression closure is a natural container: that way we can easily access it the next time we need to evaluate that closure.

To implement caching, we modify the interpreter as follows. First, we have to create a field for the value of the expression closure. What’s the value of this field? Initially it needs to hold a dummy value, to eventually be replaced by the actual one. “Replaced” means its value needs to change; therefore, it needs to be a box. Concretely, we’ll use the boolean value `false` as the initial value.

```scheme
(define-type CFAL-Value
  [numV (n number?)
    [closureV (param symbol?)
      (body CFAL?)
      (sc SubCache?)
    ]
    [exprV (expr CFAL?)
      (sc SubCache?)
      (value boxed-boolean/CFAL-Value?)])
```

We define the cache’s field predicate as follows:

```scheme
(define (boxed-boolean/CFAL-Value? v)
  (and (box? v)
    (or (boolean? (unbox v))
      (numV? (unbox v))
      (closureV? (unbox v))))
)
```

Notice that we carefully exclude `exprV` values from residing in the box. The box is meant to cache the result of strictness, which by definition and construction cannot result in an `exprV`. Therefore, this exclusion should never result in an error (and an indication to the contrary should be investigated).

Having changed the number of fields, we must modify all uses of the constructor. There’s only one: in function application.

```scheme
[app (fun-expr arg-expr)
```
That leaves only the definition of \textit{strict}. By now, it should be clear how to implement it using the box:

\begin{verbatim}
(define (strict e)
  (type-case CFAL-Value e
    [exprV (expr sc value)
      (if (boolean? (unbox value))
        (local [[(define the-value (strict (interp expr sc)))]
          (begin
            (printf "Forcing exprV \~a to \~a\n" expr the-value)
            (set-box! value the-value)
            the-value))
          (begin
            (printf "Using cached value\n")
            (unbox value)))]
      [else e]))

With these changes, we see that interpreting the running example needs to force an expression closure fewer times (how many?). The other instances reuse the value of a prior reduction. Figure 11.3 and Figure 11.4 present the heart of the interpreter. Haskell uses the value cache we have just studied, so it combines the benefit of laziness (not evaluating unnecessary arguments) with reasonable performance (evaluating the necessary ones only once).

\begin{exercise}
An expression closure is awfully similar to a regular (function) closure. Indeed, if should be possible to replace the former with the latter. When doing so, however, we don’t really need all the pieces of function closures: there are no arguments, so only the body and environment matter. Such a closure is called a thunk, a name borrowed from the technique used to implement a form of laziness in Algol 60. Implement laziness entirely using thunks, getting rid of expression closures.
\end{exercise}

\begin{exercise}
We could have achieved the same effect as using thunks (see Exercise 11.2.1) by simply using one-argument procedures with a dummy argument value. Why didn’t we propose this? Put otherwise, what benefit do we derive by keeping expression closures as a different kind of value?
\end{exercise}

\begin{exercise}
Extend this language with recursion and list primitives so you can run the equivalent of the programs we saw in Section 10.1. In this extended language, code up the Fibonacci example, run it with and without value caching, and arrive at a conclusion about the time complexity of the Haskell definition.
\end{exercise}
CHAPTER 11. IMPLEMENTING LAZINESS

11.3 Caching Computations Safely

Any language that caches computation (whether in an eager or lazy regime) is making a very strong tacit assumption: that an expression computes the same value every time it evaluates. If an expression can yield a different value in a later evaluation, then the value in the cache is corrupt, and using it in place of the correct value can cause the computation to go awry. So we must examine this evaluation decision of Haskell.

This assumption cannot be applied to most programs written in traditional languages, because of the use of side-effects. A method invocation in Java can, for instance, depend on the values of fields (directly, or indirectly via method accesses) in numerous other objects, any one of which may later change, which will almost certainly invalidate a cache of the method invocation’s computed value. To avoid having to track this complex web of dependencies, languages like Java avoid caching values altogether in the general case (though an optimizing compiler may introduce a cache under various circumstances).

Haskell implementations can cache values because Haskell does not provide explicit mutation operations. Haskell instead forces programmers to perform all computations by composing functions. While this may seem an onerous style to those unaccustomed to it, the resulting programs are in fact extremely elegant, and Haskell provides a powerful collection of primitives to enable their construction; we caught a glimpse of both the style and the primitives in Section 10.1. Furthermore, the lack of side-effects makes it possible for Haskell compilers to perform some very powerful optimizations not available to traditional language compilers, so what seems like an inefficient style on the surface (such as the creation of numerous intermediate tuples, lists and other data structures) often has little run-time impact.

Of course, no useful Haskell program is an island; programs must eventually interact with the world, which itself has true side-effects (at least in practice). Haskell therefore provides a set of “unsafe” operators that conduct input-output and other operations. Computations that depend on the results of unsafe operations cannot be cached. Haskell does, however, have a sophisticated type system (featuring quite a bit more, in fact, than we saw in Section 10.1) that makes it possible to distinguish between the unsafe and “safe” operations, thereby restoring the benefits of caching to at least portions of a program’s computation. In practice, Haskell programmers exploit this by limiting unsafe computations to a small portion of a program, leaving the remainder in the pure style espoused by the language.

The absence of side-effects benefits not only the compiler but, for related reasons, the programmer also. It greatly simplifies reasoning about programs, because to understand what a particular function is doing a programmer doesn’t need to be aware of the global flow of control of the program. In particular, programmers can study a program through equational reasoning, using the process of reduction we have studied in high-school algebra. The extent to which we can apply equational reasoning depends on the number of expressions we can reasonably substitute with other, equivalent expressions (including answers).
Referential Transparency

People sometimes refer to the lack of mutation as “referential transparency”, as in, “Haskell is referentially transparent” (and, by extension, languages like Java, C++, Scheme and ML are not). What do they really mean?

Referential transparency is commonly translated as the ability to “replace equals with equals”. For example, we can always replace $1 + 2$ with $3$. Now think about that (very loose) definition for a moment: when can you not replace something with something else that the original thing is equal to? Never, of course—you always can. So by that definition, every language is “referentially transparent” (in this rather informal sense). That informal definition therefore renders the term vacuous.

Referential transparency really describes a relation: it relates pairs of terms exactly when they can be considered equivalent in all contexts. Thus, in most languages, $1 + 2$ is referentially transparent to $3$ (assuming no overflow), and $\sqrt{4}$ (written in the appropriate notation) is referentially transparent to $2$ (assuming the square root function returns only the positive root).

Given this understanding, we can now ask the following question: what is the size of the referential transparency relation for a program in a given language? While even a language like C subscribes a referential transparency relation, and some C programs have larger relations (because they minimize side-effects), the size of this relation is inherently larger for programs written in a language without mutation.

This larger relation enables a much greater use of equational reasoning.

As a programmer, you should strive to make this relation as large as possible, no matter what language you program in: this has a positive impact on long-term program maintenance (for instance, when other programmers need to modify your code). As a student of programming languages, however, please use this term with care; in particular, always remember that it describes a relation between phrases in a program, and is rarely meaningful when applied to languages as a whole.

We have argued that caching computation is safe in the absence of side-effects. But the eager version of our interpreted language doesn’t have side-effects either! We didn’t need to cache computation in the same way we have just studied, because by definition an eager language associates identifiers with values in the environment, eliminating the possibility of re-computation on use. There is, however, a slightly different notion of caching that applies in an eager language.
CHAPTER 11. IMPLEMENTING LAZINESS

Memoization

Memoization associates a cache with each function. The cache tracks actual argument tuples and their corresponding return values. When the program invokes a “memoized” function, the evaluator first tries to find the function’s value in the cache, and only invokes the function proper if that argument tuple hadn’t been cached before. If the function is recursive, the recursive calls might also go through the memoized version. Memoization in this instance reduces the exponential number of calls in computing Fibonacci numbers to a linear number, without altering the natural recursive definition.

It’s important to note that what we have implemented for lazy languages is not memoization. While we do cache the value of each expression closure, this is different from caching the value of all expression closures that contain the same expression closed over the same environment. In our implementation, if a program contains the same source expression (such as a function invocation) twice, each use of that expression results in a separate evaluation.

Of course, to use memoization safely, the programmer or implementation would have to establish that the function’s body does not depend on side-effects—or invalidate the cache when a relevant effect happens. Memoization is sometimes introduced automatically as a compiler optimization.

Exercise 11.3.1 There are no lazy languages that permit mutation. Why not? Is there a deeper reason beyond the invalidation of several compiler optimizations?

Exercise 11.3.2 Why do you think there are no lazy languages without type systems?

Hint: This is related to Exercise 11.3.1

11.4 Scope and Evaluation Regimes

Students of programming languages often confuse the notions of scope (static versus dynamic) and evaluation regimes (eager versus lazy). In particular, readers often engage in the following fallacious reasoning:

Because lazy evaluation substitutes expressions, not values, and because substituting expressions (naively) results in variables getting their values from the point of use rather than the point of definition, therefore lazy evaluation must result in dynamic scope.

It is very important to not be trapped by this line of thought. The scoping rules of a language are determined a priori by the language designer. (For the reasons we have discussed in Section 6.5, this should almost always be static scope.) It is up to the language implementor to faithfully enforce them. Likewise, the language designer determines the reduction regime, perhaps based on some domain constraints. Again, the implementor must determine how to correctly implement the chosen regime. We have seen how the use of appropriate closure values can properly enforce static scope in both eager and lazy evaluation regimes.
(define-type CFAL
  [(num (n number?))]
  [(add (lhs CFAL?) (rhs CFAL?))]
  [(sub (lhs CFAL?) (rhs CFAL?))]
  [(id (name symbol?))]
  [(fun (param symbol?) (body CFAL?))]
  [(app (fun-expr CFAL?) (arg-expr CFAL?))]
  [(if0 (test CFAL?) (then CFAL?) (else CFAL?))])

(define-type CFAL-Value
  [(numV (n number?))]
  [(closureV (param symbol?)
               (body CFAL?)
               (sc SubCache?))]
  [(exprV (expr CFAL?)
           (sc SubCache?))])

(define-type SubCache
  [(mtSub)]
  [(aSub (name symbol?) (value CFAL-Value?) (sc SubCache?))])

;; num+ : CFAL-Value CFAL-Value → numV
(define (num+ n1 n2)
  (numV (+ (numV-n (strict n1)) (numV-n (strict n2)))))

;; num-zero? : CFAL-Value → boolean
(define (num-zero? n)
  (zero? (numV-n (strict n))))

;; strict : CFAL-Value → CFAL-Value [excluding exprV]
(define (strict e)
  (type-case CFAL-Value e
    [(exprV (expr env))
      (strict (interp expr env))]
    [else e]))

Figure 11.1: Implementation of Laziness: Support Code
;; interp : CFAL SubCache → CFAL-Value

(define (interp expr sc)
  (type-case CFAL expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l sc) (interp r sc))]
    [sub (l r) (num− (interp l sc) (interp r sc))]
    [id (v) (lookup v sc)]
    [fun (bound-id bound-body)
      (closureV bound-id bound-body sc)]
    [app (fun-exp expr arg-exp)
      (local ([define fun-val (strict (interp fun-exp sc))])
        [define arg-val (exprV arg-exp sc)])
      (interp (closureV-body fun-val)
        (aSub (closureV-param fun-val)
          arg-val
          (closureV-sc fun-val)))]
    [if0 (test pass fail)
      (if (num-zero? (interp test sc))
        (interp pass sc)
        (interp fail sc))]))

Figure 11.2: Implementation of Laziness: The Interpreter
11.4. SCOPE AND EVALUATION REGIMES

(define (boxed-boolean/CFAL-Value? v)
  (and (box? v)
       (or (boolean? (unbox v))
           (numV? (unbox v))
           (closureV? (unbox v)))))

(define-type CFAL-Value
  [numV (n number?)]
  [closureV (param symbol?)
    (body CFAL?)
    (sc SubCache?)]
  [exprV (expr CFAL?)
    (sc SubCache?)
    (value boxed-boolean/CFAL-Value?)])

;; strict : CFAL-Value → CFAL-Value [excluding exprV]

(define (strict e)
  (type-case CFAL-Value e
    [exprV (expr sc value)
      (if (boolean? (unbox value))
        (local [(define the-value (strict (interp expr sc)))
                  (begin
                    (set-box! value the-value)
                    the-value))
          (unbox value))]
    [else e]))

Figure 11.3: Implementation of Laziness with Caching: Support Code
;; interp : CFAL SubCache → CFAL-Value

(define (interp expr sc)
  (type-case CFAL expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l sc) (interp r sc))]
    [sub (l r) (num− (interp l sc) (interp r sc))]
    [id (v) (lookup v sc)]
    [fun (bound-id bound-body)
      (closureV bound-id bound-body sc)]
    [app (fun-expr arg-expr)
      (local ([define fun-val (strict (interp fun-expr sc))]
               [define arg-val (exprV arg-expr sc (box false))])
        (interp (closureV-body fun-val)
          (aSub (closureV-param fun-val)
            arg-val
            (closureV-sc fun-val)))]
    [if0 (test pass fail)
      (if (num-zero? (interp test sc))
        (interp pass sc)
        (interp fail sc))]))

Figure 11.4: Implementation of Laziness with Caching: The Interpreter