In this lecture, we will study how to implement laziness in our interpreter. That is, we will keep the syntax of our language unchanged, but alter the semantics of function application to be lazy.

To do this, we will first elide inessential parts of the language, primarily with and recursion. Therefore, in the rest of this lecture, we will write interpreters for the language CFAE: one with conditionals, functions and arithmetic expressions. However, for our examples we will continue to use with for brevity. As we have seen before, it’s easy for the parser to convert all instances of with into instances of (immediate) function application; we can assume that our parser does exactly this.

1 Implementing Laziness

Consider the following expression:

\[
\{\text{with} \ (x \ 3)\\ 
\ (\text{with} \ (y \ (+ \ x \ x))\\ 
\ (\text{with} \ (z \ y)\\ 
\ (\text{with} \ (x \ 4)\\ 
\ z))\}\}
\]

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:

\[
\{\text{with} \ (x \ 3)\\ 
\ (\text{with} \ (y \ (+ \ x \ x))\\ 
\ (\text{with} \ (z \ y)\\ 
\ (\text{with} \ (x \ 4)\\ 
\ z))\}\] \ [x \rightarrow 3]
\]

\[
\{\text{with} \ (x \ 4)\\ 
\ (\text{with} \ (z \ y)\\ 
\ z)\} \ [x \rightarrow 3, \ y \rightarrow (+ \ x \ x)]
\]

\[
\{\text{with} \ (z \ y)\\ 
\ z\} \ [x \rightarrow 4, \ y \rightarrow (+ \ x \ x)]
\]

\[
z \ [x \rightarrow 4, \ y \rightarrow (+ \ x \ x), \ z \rightarrow y]
\]

\[
y \ [x \rightarrow 4, \ y \rightarrow (+ \ x \ x), \ z \rightarrow y]
\]

\[
(+ \ x \ x) \ [x \rightarrow 4, \ y \rightarrow (+ \ x \ x), \ z \rightarrow y]
\]

\[
(+ \ 4 \ 4)
\]

\[
8
\]

In contrast, suppose we used substitution instead of environments:

\[
\{\text{with} \ (x \ 3)\\ 
\}
\]
Just in case this looks confusing: don’t worry, it’s still lazy, it’s not eager. 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, however, already had all identifiers substituted.

This 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 causes incorrect substitution to happen.

The way we addressed this problem before was to use closures. That is, the text of a function was wrapped in a structure with (“closed over”, in the parlance) 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 can define these new values, which we’ll call expression closures, easily enough:

```
(define-datatype CFA-value CFA-value?
  [numV (n number?)]
  [closureV (param symbol?)
    (body CFAE?)
    (env Env?)]
  [exprV (expr CFAE?)
    (env Env?)])
```

That is, an `exprV` is just a wrapper that holds an expression and the environment of its definition. It’s reasonable to make this a new kind of value because the environment is sometimes going to map identifiers to `exprV` values.

What needs to change in the interpreter? Obviously, in a procedure application, we should simply wrap the procedure’s actual argument in an expression closure:

```
(app (fun-expr arg-expr)
  (local ([define fun-val (interp fun-expr env)])
    [define arg-val (exprV arg-expr env)])
  (cases CFA-value fun-val
    [closureV (cl-param cl-body cl-env) (interp cl-body
      (aSub cl-param
        arg-val
        cl-env))]
    [else (error "interp "can only apply functions")])]
```

Note, in particular, what changed: the line

```
[define arg-val (interp arg-expr env)]
```
became

(define arg-val (exprV arg-expr env))

Great: so we defer the evaluation of named expressions. This means, for instance, that an expression such as

{with {x 3}
  x}

will evaluate to some expression closure value, such as

#(struct:exprv #(struct:num 3) #(struct:mtsub))

which a post-processor could turn into a more attractive representation.

That may be an acceptable output for a particularly simple program, but how about this one?

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

The interpreter evaluates each x in the body to an expression closure (what’s bound in the environment), but the addition procedure cannot handle these. One solution is to alter the definition of num-n so it reads

(define (numV-n value)
  (cases RCFWA-value value
    [numV (n) n]
    [exprV (expr env) (interp expr env)]
    [else (error 'numV-n "not a number")]))

Unfortunately, this isn’t enough: (in a program without type errors) the result of interpreting returns a numV value, while numV-n needs to return a number. Therefore, we must alter it slightly, though we could just be lazy and write

(define (numV-n value)
  (cases RCFWA-value value
    [numV (n) n]
    [exprV (expr env) (numV-n (interp expr env))]
    [else (error 'numV-n "not a number")]))

This works fine until we get to an example of this form:

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

The problem here is that, when evaluated, the application position of a function application evaluates to an expression closure. So long as procedure definitions were immediate, this didn’t create a problem. But looking up an identifier such as double in the environment returns the expression closure to which it’s bound. The application rule of the interpreter doesn’t know what to do with this kind of CFA-value, so it halts with an error.

It’s clear that, before we can continue with function application, we need to know exactly what kind of value—specifically, which function—the function position’s application closure contains. Without this knowledge, we cannot begin to evaluate that function’s body! Therefore, we must convert the expression closure into a value.

This is getting tedious—we did just this for numbers just a short while ago—so we’ll abstract this code into its own separate procedure:

;; strict : exprV → CFAE [not including exprV]
(define (strict e)
  (cases CFA-value e
    [exprV (expr env) (strict (interp expr env))]
    [else e]))
Now we can use this for numbers,

```
(define (numV-n value)
  (cases CFA-value value
    [numV (n) n]
    [exprV (expr env) (numV-n (strict value))]
    [else (error 'numV-n "not a number")]]
)
```

and also in the interpreter:

```
(app (fun-expr arg-expr)
  (local ([define fun-val (strict (interp fun-expr env))])
    [define arg-val (exprV arg-expr env)]
    (cases CFA-value fun-val
      [closureV (cl-param cl-body cl-env) (interp cl-body
        (aSub cl-param arg-val
          cl-env))]
      [else (error 'interp "can only apply functions")]]
)
```

Specifically, the definition of `fun-val` changed from

```
(local ([define fun-val (interp fun-expr env)])
```

to

```
(local ([define fun-val (strict (interp fun-expr env)])
```

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 our perhaps odd name, strict, for the procedure that accomplishes this task.

We see that this lazy interpreter also correctly evaluates a term like

```
{with {f {undef x}}
  4}
```

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

There is actually one more strictness point in our language, except it’s hidden in the bowels of the interpreter. This is the conditional statement. If we had a generalized `if` construct, we would need to force the conditional expression to a value to determine which branch to follow. In our interpreter, however, the strictness is induced subtly: the conditional clause invokes `numV-zero?`, which in turn invokes `numV-n`, which eventually invokes `strict`.

## 2 Caching Computation

Evaluating an expression like

```
{with {x 3}
  (with {y (+ x x)}
    {with {x 4}
      y})}
```

can be extremely wasteful. A small change to the interpreter

```
(define (strict e)
  (cases CFA-value e
    [exprV (expr env)
      (local ([define the-value (strict (interp expr env))])
        (begin
          (printf "Forcing exprV to "a\n" the-value)
          the-value)]
      [else e]))
)
```

```
lets us see how many times expression closures are evaluated, and we see that this one forces evaluation three times.\(^1\)

Can we do better? Naturally: once we have computed the value of an identifier, instead of only using it, we can also keep track of it for future use. Where should we store it? The expression closure seems a natural spot, since that’s where we’re likely to look the next time we need that value.

To implement this, 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.

\[
\text{(define-datatype CFA-value CFA-value?}
\begin{align*}
\quad &\text{[numV (n number?)]} \\
\quad &\text{[closureV (param symbol?) (body CFAE?) (env Env?)]} \\
\quad &\text{[exprV (expr CFAE?) (env Env?) (value-box (lambda (x) (and (box? x) (or (boolean? (unbox x)) (CFA-value? (unbox x))))))]} \\
\end{align*}
\]

Having changed the number of fields, we must modify all uses of the constructor. There’s only one. What initial value do we use? Naturally, we have to use false:

\[
\text{[app (fun-expr arg-expr)} \\
\qquad \text{(local [(define fun-val (strict (interp fun-expr env)))]}} \\
\qquad \text{[define arg-val (exprV arg-expr env (box false))]} \\
\qquad \text{(cases CFA-value fun-val \ldots)]} \\
\]

At this point, we still can’t execute our program, because the cases statements that dispatch on CFAE values must also change. There are two such instances: one in numV-n and one more in strict. In fact, we might as well simplify numV-n to always use strict, because if strict is given a value that isn’t an expression closure, it simply returns that value. Thus, numV-n becomes:

\[
\text{(define (numV-n value)} \\
\qquad \text{(cases CFA-value (strict value)} \\
\qquad \text{[numV (n) n]} \\
\qquad \text{[else (error ’numV-n "not a number")])}} \\
\]

That leaves strict. By now, it should be pretty clear how to implement it using the box:

\[
\text{(define (strict e)} \\
\qquad \text{(cases CFA-value e)} \\
\qquad \text{[exprV (expr env value-box)} \\
\qquad \qquad \text{(if (boolean? (unbox value-box))}} \\
\qquad \qquad \qquad \text{(local [(define the-value (strict (interp expr env)))]}} \\
\qquad \qquad \qquad \text{(begin}} \\
\qquad \qquad \qquad \qquad \text{(set-box! value-box the-value)} \\
\qquad \qquad \qquad \qquad \text{the-value))} \\
\qquad \qquad \text{(unbox value-box))]} \\
\qquad \text{[else e])]} \\
\]

With this change, and a modification to the interpreter to print the number of evaluations of expression closures, we see that applying the interpreter to the example above needs to reduce an expression closure to a value only two times; the third instance can reuse a reduction.

\(^{1}\)Why three?
Puzzles

1. It’s clear that an expression closure is awfully similar to a regular closure representing a function. Indeed, if should be possible to replace the former with the latter. If we do so, we don’t really need the full power of closures: the body and environment are all that matter, so the closure can have no arguments. Such a closure is called a thunk, a name borrowed from the technique used to implement laziness in Algol 60. (Thunks are a poor man’s way of obtaining laziness in an eager language like Scheme.) Using your multi-arity functions, implement laziness entirely using thunks, getting rid of expression closures.

2. We could have achieved the same effect as using thunks by simply using one-argument procedures with a dummy argument value. Why didn’t we do this? Put otherwise, what benefit do we derive by keeping expression closures as a different kind of value?

3. Why do you think there are no lazy languages without type systems?

4. Why do you think there are no lazy languages that also allow programmers to change the values bound to identifiers? Can you write a program that illustrates your reason especially dramatically?