PL: Lecture #7

Tuesday, October 1st

Lazy vs Eager Evaluation

As we have previously seen, there are two basic approaches for evaluation: either eager or lazy. In lazy evaluation, bindings are used for sort of textual references — it is only for avoiding writing an expression twice, but the associated computation is done twice anyway. In eager evaluation, we eliminate not only the textual redundancy, but also the computation.

Which evaluation method did our evaluator use? The relevant piece of formalism is the treatment of `with`:

```
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
```

And the matching piece of code is:

```
[(With bound-id named-expr bound-body)
 (eval (subst bound-body
              bound-id
              (Num (eval named-expr))))]
```

How do we make this lazy?

In the formal equation:

```
eval({with {x E1} E2}) = eval(E2[E1/x])
```

and in the code:

```
(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  named-expr))] ;*** no eval and no Num wrapping
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```

We can verify the way this works by tracing `eval` (compare the trace you get for the two versions):

```
> (trace eval) ; (put this in the definitions window)
> (run "{with {x {+ 1 2}} {* x x}}")
```

Ignoring the traces for now, the modified WAE interpreter works as before, specifically, all tests pass. So the question is whether the language we get is actually different than the one we had before. One difference is in execution speed, but we can't really notice a difference, and we care more about meaning. Is there any program that will run differently in the two languages?

The main feature of the lazy evaluator is that it is not evaluating the named expression until it is actually needed. As we have seen, this leads to duplicating computations if the bound identifier is used more than once — meaning that it does not eliminate the dynamic redundancy. But what if the bound identifier is not used at all? In that case the named expression simply evaporates. This is a good hint at an expression that

behaves differently in the two languages — if we add division to both languages, we get a different result when we try running:

```
{with {x {/ 8 0}} 7}
```

The eager evaluator stops with an error when it tries evaluating the division — and the lazy evaluator simply ignores it.

Even without division, we get a similar behavior for

```
{with {x y} 7}
```

but it is questionable whether the fact that this evaluates to 7 is correct behavior — we really want to forbid program that use free variable.

Furthermore, there is an issue with name capturing — we don't want to substitute an expression into a context that captures some of its free variables. But our substitution allows just that, which is usually not a problem because by the time we do the substitution, the named expression should not have free variables that need to be replaced. However, consider evaluating this program:

```
{with {y x}
  {with {x 2}
    {+ x y}}}
```

under the two evaluation regimens: the eager version stops with an error, and the lazy version succeed. This points at a bug in our substitution, or rather not dealing with an issue that we do not encounter.

So the summary is: as long as the initial program is correct, both evaluation regimens produce the same results. If a program contains free variables, they might get captured in a naive lazy evaluator implementation (but this is a bug that should be fixed). Also, there are some cases where eager evaluation runs into a run-time problem which does not happen in a lazy evaluator because the expression is not used. It is possible to prove that when you evaluate an expression, if there is an error that can be avoided, lazy evaluation will always avoid it, whereas an eager evaluator will always run into it. On the other hand, lazy evaluators are usually slower than eager evaluator, so it's a speed vs. robustness trade-off.

Note that with lazy evaluation we say that an identifier is bound to an expression rather than a value. (Again, this is why the eager version needed to wrap `eval`'s result in a `Num` and this one doesn't.)

(It is possible to change things and get a more well behaved substitution, we basically will need to find if a capture might happen, and rename things to avoid it. For example,

```
{with {y E1} E2}[v/x]
  if `x' and `y' are equal
    = {with {y E1[v/x]} E2} = {with {x E1[v/x]} E2}
  if `y' has a free occurrence in `v'
    = {with {y1 E1[v/x]} E2[y1/y][v/x]} ; `y1' is "fresh"
  otherwise
    = {with {y E1[v/x]} E2[v/x]}
```

With this, we might have gone through this path in evaluating the above:

```
{with {y x} {with {x 2} {+ x y}}}
{with {x₁ 2} {+ x₁ x}} ; note that x₁ is a fresh name, not x
{+ 2 x}
error: free `x`
```

But you can see that this is much more complicated (more code: requires a `free-in` predicate, being able to invent new *fresh* names, etc). And it's not even the end of that story…)

de Bruijn Indexes

This whole story revolves around names, specifically, name capture is a problem that should always be avoided (it is one major source of PL headaches).

But are names the only way we can use bindings?

There is a least one alternative way: note that the only thing we used names for are for references. We don't really care what the name is, which is pretty obvious when we consider the two WAE expressions:

```
{with {x 5} {+ x x}}
{with {y 5} {+ y y}}
```

or the two Racket function definitions:

```
(define (foo x) (list x x))
(define (foo y) (list y y))
```

Both of these show a pair of expressions that we should consider as equal in some sense (this is called "alpha-equality"). The only thing we care about is what variable points where: the binding structure is the only thing that matters. In other words, as long as DrRacket produces the same arrows when we use Check Syntax, we consider the program to be the same, regardless of name choices (for argument names and local names, not for global names like `foo` in the above).

The alternative idea uses this principle: if all we care about is where the arrows go, then simply get rid of the names… Instead of referencing a binding through its name, just specify which of the surrounding scopes we want to refer to. For example, instead of:

```
{with {x 5} {with {y 6} {+ x y}}}
```

we can use a new "reference" syntax — `[N]` — and use this instead of the above:

```
{with 5 {with 6 {+ [1] [0]}}}
```

So the rules for `[N]` are — `[0]` is the value bound in the current scope, `[1]` is the value from the next one up etc.

Of course, to do this translation, we have to know the precise scope rules. Two more complicated examples:

```
{with {x 5} {+ x {with {y 6} {+ x y}}}}
```

is translated to:

```
{with 5 {+ [0] {with 6 {+ [1] [0]}}}}
```

(note how `x` appears as a different reference based on where it appeared in the original code.) Even more subtle:

```
{with {x 5} {with {y {+ x 1}} {+ x y}}}
```

is translated to:

```
{with 5 {with {+ [0] 1} {+ [1] [0]}}}
```

because the inner `with` does not have its own named expression in its scope, so the named expression is immediately in the scope of the outer `with`.

This is called "de Bruijn Indexes": instead of referencing identifiers by their name, we use an index into the surrounding binding context. The major disadvantage, as can be seen in the above examples, is that it is not convenient for humans to work with. Specifically, the same identifier is referenced using different numbers, which makes it hard to understand what some code is doing. After all, *abstractions* are the main thing we deal with when we write programs, and having labels make the bindings structure much easier to understand than scope counts.

However, practically all compilers use this for compiled code (think about stack pointers). For example, GCC compiles this code:

```
{
  int x = 5;
  {
    int y = x + 1;
    return x + y;
  }
}
```

to:

```
subl $8, %esp
movl $5, -4(%ebp)   ; int x = 5
movl -4(%ebp), %eax
incl %eax
movl %eax, -8(%ebp) ; int y = %eax
movl -8(%ebp), %eax
addl -4(%ebp), %eax
```

Functions & Function Values

> 👉 ***PLAI §4***

Now that we have a form for local bindings, which forced us to deal with proper substitutions and everything that is related, we can get to functions. The concept of a function is itself very close to substitution, and to our `with` form. For example, when we write:

```
{with {x 5}
  {* x x}}
```

then the `{* x x}` body is itself parametrized over some value for `x`. If we take this expression and take out the `5`, we're left with something that has all of the necessary ingredients of a function — a bunch of code that is parameterized over some input identifier:

```
{with {x}
  {* x x}}
```

We only need to replace `with` and use a proper name that indicates that it's a function:

```
{fun {x}
  {* x x}}
```

Now we have a new form in our language, one that should have a function as its meaning. As we have seen in the case of `with` expressions, we also need a new form to *use* these functions. We will use `call` for this, so that

```
{call {fun {x} {* x x}}
      5}
```

will be the same as the original `with` expression that we started with — the `fun` expression is like the `with` expression with no value, and applying it on `5` is providing that value back:

```
{with {x 5}
  {* x x}}
```

Of course, this does not help much — all we get is a way to use local bindings that is more verbose from what we started with. What we're really missing is a way to *name* these functions. If we get the right evaluation rules, we can evaluate a `fun` expression to some value — which will allow us to bind it to a variable using `with`. Something like this:

```
{with {sqr {fun {x} {* x x}}}
  {+ {call sqr 5}
     {call sqr 6}}}
```

In this expression, we say that `x` is the formal parameter (or argument), and the `5` and `6` are actual parameters (sometimes abbreviated as formals and actuals). Note that naming functions often helps, but many times there are small functions that are fine to specify without a name — for example, consider a two-stage addition function, where there is no apparent good name for the returned function:

```
{with {add {fun {x}
             {fun {y}
               {+ x y}}}}
  {call {call add 8} 9}}
```

Implementing First Class Functions

> 👉 ***PLAI §6*** *(uses some stuff from* 👉 ***PLAI §5****, which we do later)*

This is a simple plan, but it is directly related to how functions are going to be used in our language. We know that `{call {fun {x} E1} E2}` is equivalent to a `with` expression, but the new thing here is that we do allow writing just the `{fun ...}` expression by itself, and therefore we need to have some meaning for it. The meaning, or the value of this expression, should roughly be "an expression that needs a value to be plugged in for `x`". In other words, our language will have these new kinds of values that contain an expression to be evaluated later on.

There are three basic approaches that classify programming languages in relation to how the deal with functions:

1. First order: functions are not real values. They cannot be used or returned as values by other functions. This means that they cannot be stored in data structures. This is what most "conventional" languages used to have in the past. (You will be implementing such a language in homework 4.)

   An example of such a language is the Beginner Student language that is used in HtDP, where the language is intentionally first-order to help students write correct code (at the early stages where using a function as a value is usually an error). It's hard to find practical modern languages that fall in this category.

2. Higher order: functions can receive and return other functions as values. This is what you get with C and modern Fortran.

3. First class: functions are values with all the rights of other values. In particular, they can be supplied to other functions, returned from functions, stored in data structures, and new functions can be created at run-time. (And most modern languages have first class functions.)

The last category is the most interesting one. Back in the old days, complex expressions were not first-class in that they could not be freely composed. This is still the case in machine-code: as we've seen earlier, to compute an expression such as

```
(-b + sqrt(b^2 - 4*a*c)) / 2a
```

you have to do something like this:

```
x = b * b
y = 4 * a
y = y * c
x = x - y
x = sqrt(x)
y = -b
x = y + x
```

```
y = 2 * a
s = x / y
```

In other words, every intermediate value needs to have its own name. But with proper (“high-level”) programming languages (at least most of them…) you can just write the original expression, with no names for these values.

With first-class functions something similar happens — it is possible to have complex expressions that consume and return functions, and they do not need to be named.

What we get with our `fun` expression (if we can make it work) is exactly this: it generates a function, and you can choose to either bind it to a name, or not. The important thing is that the value exists independently of a name.

This has a major effect on the “personality” of a programming language as we will see. In fact, just adding this feature will make our language much more advanced than languages with just higher-order or first-order functions.

Quick Example: the following is working JavaScript code, that uses first class functions.

```
function foo(x) {
  function bar(y) { return x + y; }
  return bar;
}
function main() {
  var f = foo(1);
  var g = foo(10);
  return [f(2), g(2)];
}
```

Note that the above definition of `foo` does *not* use an anonymous “lambda expression” — in Racket terms, it’s translated to

```
(define (foo x)
  (define (bar y) (+ x y))
  bar)
```

The returned function is not anonymous, but it’s not really named either: the `bar` name is bound only inside the body of `foo`, and outside of it that name no longer exists since it’s not its scope. It gets used in the printed form if the function value is displayed, but this is merely a debugging aid. The anonymous `lambda` version that is common in Racket can be used in JavaScript too:

```
function foo(x) {
  return function(y) { return x + y; }
}
```

> *Side-note: GCC includes extensions that allow internal function definitions, but it still does not have first class functions — trying to do the above is broken:*
>
> ```
> #include <stdio.h>
> typedef int(*int2int)(int);
> int2int foo(int x) {
>   int bar(int y) { return x + y; }
>   return bar;
> }
> int main() {
>   int2int f = foo(1);
>   int2int g = foo(10);
>   printf(">> %d, %d\n", f(2), g(2));
> }
> ```

Side-note: how important is it to have *anonymous* functions?

You'll see many places where people refer to the feature of first-class functions as the ability to create *anonymous* functions, but this is a confusion and it's not accurate. Whether a function has a name or not is not the important question — instead, the important question is whether functions can exist with no *bindings* that refers to them.

As a quick example in Racket:

```
(define (foo x)
  (define (bar y) (+ x y))
  bar)
```

in Javascript:

```
function foo(x) {
  function bar(y) {
    return x + y;
  }
  return bar;
}
```

and in Python:

```
def foo(x):
  def bar(y):
    return x + y
  return bar
```

In all three of these, we have a `foo` function that returns a function *named* `bar` — but the `bar` name, is only available in the scope of `foo`. The fact that the name is displayed as part of the textual rendering of the function value is merely a debugging feature.