Implementing First-Class Functions with Environments

Lecture Notes for cs173, Fall 2001

`sk` and `rob`

September 21, 2001

> Save the environment. Create a closure today.
> —Cormac Flanagan
>
> It's kind of like a turtle. It carries its home around with it.
> —sk on closures
>
> Dynamic scoping is interesting as a historical mistake. But the important part is *not* that it's historic, it's that it was a mistake.
> –sk

1 Delaying Substitution

Calling *interp* on a Rip program requires traversing each expression of the code at least once. For each call to *subst* on a sub-expression, however, *interp* must traverse the sub-expression an additional time. Consider the Rip code

```
{{proc {x}
   {{proc {y}
      {{proc {z}
        {+ x {+ y z}}}
       5}}
    4}}
 3}
```

We have a series of nested procedure applications. Because we are using substitution, we traverse the expression `{+ x {+ y z}}` three extra times. In general, if there are n nested procedure applications, our interpreter will visit the inner-most expression n extra times due to substitution. These extra traversals seem wasteful. Let's get rid of *subst* entirely.

The problem is really that we are substituting eagerly. Let's try delaying substitution until we actually need the value of a variable. In this way we will walk over the code

only one time, looking up the values of variables as we go.

We will add *delayed substitutions* to our interpreter. A delayed substitution is just a data structure that keeps track of substitutions we will eventually want to perform. We provide the following abstract data type for delayed substitutions, and leave the implementation to you:

;; **empty-sub** : dsub
;; **extend-dsub** : dsub × var-name × value → dsub
;; **lookup-dsub** : dsub × var-name → value

Recall our interpreter. We would like to modify it to use delayed substitutions instead of *subst*. The first change is to the contract—*interp* needs to take a *dsub* as an extra argument.

;; **interp** : RP × dsub → value

```
(define (interp expr dsub)
  (cond
    [(numE? expr) ...]
    [(addE? expr) ...]
    [(varE? expr) ...]
    [(procE? expr) ...]
    [(appE? expr) ...]))
```

The *numE* clause is trivial. Since a *numE* expression cannot contain variables, we simply return *expr*.

The *addE* clause is also easy. It is exactly the same as before, but we must pass along the *dsub* with each recursive call to *interp*.

In our earlier interpreter, getting to a *varE* signaled an error. Since we are not performing substitution anymore, we need to look in the *dsub* to see what value to substitute for this variable. The code is (*dsub-lookup dsub* (*varE-name expr*)).

The *procE* clause will not change at all; we simply return the procedure.

Application is a little trickier. Let's start with our substitution version and convert it to use delayed substitutions. Our last interpreter's *appE* clause looked like

```
[(appE? expr) (do-subst-interp (interp (appE-proc expr))
                               (interp (appE-arg expr)))]
```

with *do-subst-interp* defined as

```
(define (do-subst-interp proc argV)
  (interp (subst (procE-body proc)
                 (procE-arg-name proc)
                 argV)))
```

What will change when we delay substitution? We still need to reduce the *appE-proc* to a *procE*, and the *appE-arg* to a value. We do this the same way—by calling *interp*. Once we have the *procE* and argument value, we can perform the application. Our code therefore looks like

```
[(appE? expr) (do-app (interp (appE-proc expr) dsub)
                      (interp (appE-arg expr) dsub))]
```

The function *do-app* will be very similar to *do-subst-interp*. Instead of doing a substitution, however, we will just extend our *dsub*.

```
;; do-app : procE × value → value

(define (do-app proc argV)
  (interp (procE-body proc)
          (extend-dsub ??
                       (procE-arg-name proc)
                       argV)))
```

Here we've created a new delayed substitution using *extend-subst*, analogous to calling *subst* in *do-subst-interp*. Which *dsub* do we pass into *extend-dsub*? We only have one *dsub* so it clearly must be that one. We modify the code slightly:

```
[(appE? expr) (do-app (interp (appE-proc expr) dsub)
                      (interp (appE-arg expr) dsub)
                      dsub)]

;; do-app : procE × value × dsub → value

(define (do-app proc argV dsub)
  (interp (procE-body proc)
          (extend-dsub dsub
                       (procE-arg-name proc)
                       argV)))
```

2 Testing our New Interpreter

`3` ⇒ `3`

{`+ 1 2`} ⇒ `3`

{{`proc {x} {+ x 1}} 3`} ⇒ `4`

Now consider the Rip program

```
{let {x 3}
  {let {f {proc {y} {+ x y}}}
    {let {x 5}
      {f 10}}}}
```

We will annotate this program with the current *dsub*. Preceding each expression is a box which represents *dsub* at the time that expression is interpreted:

$\square$`{let {x `$\square$`3}`

$\boxed{x \mapsto 3}$`{let {f `$\boxed{x \mapsto 3}$`{proc {y} {+ x y}}}`

$\boxed{x \mapsto 3, f \mapsto procE...}$`{let {x `$\boxed{x \mapsto 3, f \mapsto procE...}$`5}`

$\boxed{x \mapsto 5, f \mapsto procE...}$`{f `$\boxed{x \mapsto 5, f \mapsto procE...}$`10}`

When we call interp on `{f 10}`, the *dsub* is $[x \mapsto 5, f \mapsto procE...]$. We call *do-app*, which in turn calls *interp* on the body of the procedure, `{+ x y}`, with the *dsub* $[y \mapsto 10, x \mapsto 5, f \mapsto procE...]$. Thus the return value of the Rip program is `15`.

But is this correct? No. Actually, it's entirely absurd. We created a procedure `f` in an *environment* (another word for delayed substitution, which we will use from now on), in which `x` was bound to `3`. When we invoke the procedure, the environment happens to have `x` bound to 5, and the semantics of `f` is now completely different.

Hopefully, in programming, one of your goals is to write useful procedures—which means that people other than yourself will actually use them. (Some of us actually do suffer this misfortune.) Given the semantics of the interpreter we just defined, you will have no way of knowing what your procedure will actually do, since you have no way of predicting in what kinds of environments your users will invoke your procedure.

Procedures need to be invoked in the environment in which the procedure was created (which would give the answer `13`), not the *calling* environment (the answer `15`). John McCarthy, the creator of Lisp, misunderstood this point. The original versions of Lisp would all return 15. (Common Lisp, in contrast, provides *statically-scoped* procedures, taking its cue from Scheme, which was really just created to understand this point about procedures!)

Definition 1 (static scoping) *A language with static scoping uses the environment in which a procedure was created to evaluate its body during function application.*

Definition 2 (dynamic scoping) *A language with dynamic scoping uses the environment in which a procedure was invoked to evaluate its body during function application.*

We use the term static scoping because the semantics of a function doesn't change based on where you use it. Dynamic scoping is so named because the semantics of a function depend on how the whole program runs.

3 A Statically-Scoped Interpreter

The problem with our last interpreter is that we weren't actually *delaying* substitution. In fact, the interpreter forgets certain substitutions that exist when procedures are created. Let's delay substitution correctly. We define something called a *closure* which wraps the procedure along with its environment:

```
(define-struct closure (proc dsub))

;; proc : procE
;; dsub : dsub
```

To interpret a *procE*, we make a closure. To evaluate an *appE*, we will do exactly as before, except that we use the *dsub* from inside the closure. Thus, our interpreter is:

```
(define (interp expr dsub)
  (cond
    [(numE? expr) expr]
    [(addE? expr) (+ (interp (addE-left expr) dsub)
                     (interp (addE-right expr) dsub))]
    [(varE? expr) (lookup-dsub dsub (varE-var-name expr))]
    [(procE? expr) (make-closure expr dsub)]
    [(appE? expr) (do-app (interp (appE-proc expr) dsub)
                          (interp (appE-arg expr) dsub))]))
```

The function *do-app* is now

```
(define (do-app clos argV)
  (interp (procE-body (procE-body (closure-proc clos))
          (extend-dsub (closure-dsub clos)
                       (procE-arg-name (closure-proc clos))
                       argV))))
```

This new interpreter implements a language with

- static scoping
- first-class procedures (procedures are values in their own right)
- eager application semantics (we interpret the argument at the time of application)
- delayed substitutions (environments), instead of *subst*