PL: Lecture #9

Tuesday, October 8th

Substitution Caches

> 👉 ***PLAI §5*** *(called "deferred substitutions" there)*

Evaluating using substitutions is very inefficient — at each scope, we copy a piece of the program AST. This includes all function calls which implies an impractical cost (function calls should be *cheap*!).

To get over this, we want to use a cache of substitutions.

Basic idea: we begin evaluating with no cached substitutions, then collect them as we encounter bindings.

[Implies another change for our evaluator: we don't really substitute identifiers until we get to them; when we reach an identifier, it is no longer an error — we must consult the substitution cache.]

Implementation of Cache Functionality

First, we need a type for a substitution cache. For this we will use a list of lists of two elements each — a name and its value FLANG:

```
;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))
```

We need to have an empty substitution cache, a way to extend it, and a way to look things up:

```
(: empty-subst : SubstCache)
(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)
;; extend a given substitution cache with a new mapping
(define (extend id expr sc)
  (cons (list id expr) sc))

(: lookup : Symbol SubstCache -> FLANG)
;; lookup a symbol in a substitution cache, return the value it is
;; bound to (or throw an error if it isn't bound)
(define (lookup name sc)
  (cond [(null? sc) (error 'lookup "no binding for ~s" name)]
        [(eq? name (first (first sc))) (second (first sc))]
        [else (lookup name (rest sc))]))
```

Actually, the reason to use such list of lists is that Racket has a built-in function called `assq` that will do this kind of search (`assq` is a search in an association list using `eq?` for the key comparison). This is a version of `lookup` that uses `assq`:

```
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))
```

Formal Rules for Cached Substitutions

The formal evaluation rules are now different. Evaluation carries along a *substitution cache* that begins its life as empty: so `eval` needs an extra argument. We begin by writing the rules that deal with the cache, and use the above function names for simplicity — the behavior of the three definitions can be summed up in a single rule for `lookup`:

```
lookup(x,empty-subst)     = error!
lookup(x,extend(x,E,sc))  = E
lookup(x,extend(y,E,sc))  = lookup(x,sc)  if `x` is not `y`
```

And now we can write the new rules for `eval`

```
eval(N,sc)                = N
eval({+ E1 E2},sc)        = eval(E1,sc) + eval(E2,sc)
eval({- E1 E2},sc)        = eval(E1,sc) - eval(E2,sc)
eval({* E1 E2},sc)        = eval(E1,sc) * eval(E2,sc)
eval({/ E1 E2},sc)        = eval(E1,sc) / eval(E2,sc)
eval(x,sc)                = lookup(x,sc)
eval({with {x E1} E2},sc) = eval(E2,extend(x,eval(E1,sc),sc))
eval({fun {x} E},sc)      = {fun {x} E}
eval({call E1 E2},sc)     = eval(B,extend(x,eval(E2,sc),sc))
                              if eval(E1,sc) = {fun {x} B}
                          = error!  otherwise
```

Note that there is no mention of `subst` — the whole point is that we don't really do substitution, but use the cache instead. The `lookup` rules, and the places where `extend` is used replaces `subst`, and therefore specifies our scoping rules.

Also note that the rule for `call` is still very similar to the rule for `with`, but it looks like we have lost something — the interesting bit with substituting into `fun` expressions.

Evaluating with Substitution Caches

Implementing the new `eval` is easy now — it is extended in the same way that the formal `eval` rule is extended:

```
(: eval : FLANG SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (eval expr sc)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l sc) (eval r sc))]
    [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
    [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
    [(Div l r) (arith-op / (eval l sc) (eval r sc))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (extend bound-id (eval named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr sc))
     (cases fval
       [(Fun bound-id bound-body)
        (eval bound-body (extend bound-id (eval arg-expr sc) sc))]
```

```
      [else (error 'eval "`call' expects a function, got: ~s"
                   fval)])]))
```

Again, note that we don't need `subst` anymore, but the rest of the code (the data type definition, parsing, and `arith-op`) is exactly the same.

Finally, we need to make sure that `eval` is initially called with an empty cache. This is easy to change in our main `run` entry point:

```
(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) empty-subst)])
    (cases result
      [(Num n) n]
      [else (error 'run "evaluation returned a non-number: ~s"
                   result)])))
```

The full code (including the same tests, but not including formal rules for now) follows. Note that one test does not pass.

```
#lang pl

(define-type FLANG
  [Num  Number]
  [Add  FLANG FLANG]
  [Sub  FLANG FLANG]
  [Mul  FLANG FLANG]
  [Div  FLANG FLANG]
  [Id   Symbol]
  [With Symbol FLANG FLANG]
  [Fun  Symbol FLANG]
  [Call FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)
;; parses s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        (Fun name (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg)
                       (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> FLANG)
```

```
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))

(: empty-subst : SubstCache)
(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)
;; extend a given substitution cache with a new mapping
(define (extend name val sc)
  (cons (list name val) sc))

(: lookup : Symbol SubstCache -> FLANG)
;; lookup a symbol in a substitution cache, return the value it is
;; bound to (or throw an error if it isn't bound)
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))

(: Num->number : FLANG -> Number)
;; convert a FLANG number to a Racket one
(define (Num->number e)
  (cases e
    [(Num n) n]
    [else (error 'arith-op "expected a number, got: ~s" e)]))

(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op val1 val2)
  (Num (op (Num->number val1) (Num->number val2))))

(: eval : FLANG SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (eval expr sc)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l sc) (eval r sc))]
    [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
    [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
    [(Div l r) (arith-op / (eval l sc) (eval r sc))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (extend bound-id (eval named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr sc))
     (cases fval
       [(Fun bound-id bound-body)
        (eval bound-body (extend bound-id (eval arg-expr sc) sc))]
       [else (error 'eval "`call' expects a function, got: ~s"
                    fval)])]))
```

```
(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) empty-subst)])
    (cases result
      [(Num n) n]
      [else (error 'run "evaluation returned a non-number: ~s"
                   result)])))

;; tests
(test (run "{call {fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {with {add1 {fun {x} {+ x 1}}}
                {with {x 3}
                  {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
              {with {foo {fun {x} {+ x 1}}}
                {call {call identity foo} 123}}}")
      => 124)
(test (run "{call {with {x 3}
                    {fun {y} {+ x y}}}
                  4}")
      => 7)
(test (run "{with {f {with {x 3} {fun {y} {+ x y}}}}
              {with {x 100}
                {call f 4}}}")
      => 7)
(test (run "{with {x 3}
              {with {f {fun {y} {+ x y}}}
                {with {x 5}
                  {call f 4}}}}")
      => "???")
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```

Dynamic and Lexical Scopes

This seems like it should work, and it even worked on a few examples, except for one which was hard to follow. Seems like we have a bug…

Now we get to a tricky issue that managed to be a problem for *lots* of language implementors, including the first version of Lisp. Lets try to run the following expression — try to figure out what it will evaluate to:

```
(run "{with {x 3}
        {with {f {fun {y} {+ x y}}}
          {with {x 5}
            {call f 4}}}}")
```

We expect it to return `7` (at least I do!), but we get `9` instead… The question is — *should* it return `9`?

What we have arrived to is called *dynamic scope*. Scope is determined by the dynamic run-time environment (which is represented by our substitution cache). This is *almost always* undesirable, as I hope to convince you.

Before we start, we define two scope options for a programming language:

- Static Scope (also called Lexical Scope): In a language with static scope, each identifier gets its value from the scope of its definition, not its use.
- Dynamic Scope: In a language with dynamic scope, each identifier gets its value from the scope of its use, not its definition.

Racket uses lexical scope, our new evaluator uses dynamic, the old substitution-based evaluator was static etc.

As a side-remark, Lisp began its life as a dynamically-scoped language. The artifacts of this were (sort-of) dismissed as an implementation bug. When Scheme was introduced, it was the first Lisp dialect that used strictly lexical scoping, and Racket is obviously doing the same. (Some Lisp implementations used dynamic scope for interpreted code and lexical scope for compiled code!) In fact, Emacs Lisp is the only *live* dialects of Lisp that is still dynamically scoped by default. To see this, compare a version of the above code in Racket:

```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (f 4))))
```

and the Emacs Lisp version (which looks almost the same):

```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (funcall f 4))))
```

which also happens when we use another function on the way:

```
(defun blah (func val)
  (funcall func val))

(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (blah f 4))))
```

and note that renaming identifiers can lead to different code — change that `val` to `x`:

```
(defun blah (func x)
  (funcall func x))

(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (blah f 4))))
```

and you get `8` because the argument name changed the `x` that the internal function sees!

Consider also this Emacs Lisp function:

```
(defun return-x ()
  x)
```

which has no meaning by itself (`x` is unbound),

```
(return-x)
```

but can be given a dynamic meaning using a `let`:

```
(let ((x 5)) (return-x))
```

or a function application:

```
(defun foo (x)
  (return-x))

(foo 5)
```

There is also a dynamically-scoped language in the course languages:

```
#lang pl dynamic

(define x 123)

(define (getx) x)

(define (bar1 x) (getx))
(define (bar2 y) (getx))

(test (getx) => 123)
(test (let ([x 456]) (getx)) => 456)
(test (getx) => 123)
(test (bar1 999) => 999)
(test (bar2 999) => 123)

(define (foo x) (define (helper) (+ x 1)) helper)
(test ((foo 0)) => 124)

;; and *much* worse:
(define (add x y) (+ x y))
(test (let ([+ *]) (add 6 7)) => 42)
```

Note how bad the last example gets: you basically cannot call any function and know in advance what it will do.

There are some cases where dynamic scope can be useful in that it allows you to “remotely” customize any piece of code. A good example of where this is taken to an extreme is Emacs: originally, it was based on an ancient Lisp dialect that was still dynamically scoped, but it retained this feature even when practically all Lisp dialects moved on to having lexical scope by default. The reason for this is that the danger of dynamic scope is also a way to make a very open system where almost anything can be customized by changing it “remotely”. Here’s a concrete example for a similar kind of dynamic scope usage that makes a very hackable and open system:

```
#lang pl dynamic

(define tax% 6.25)
(define (with-tax n)
  (+ n (* n (/ tax% 100))))

(with-tax 10) ; how much do we pay?
(let ([tax% 17.0]) (with-tax 10)) ; how much would we pay in Israel?

;; make that into a function
(define il-tax% 17.0)
(define (ma-over-il-saving n)
  (- (let ([tax% il-tax%]) (with-tax n))
     (with-tax n)))
```

```
(ma-over-il-saving 10)
;; can even control that: how much would we save if
;; the tax in israel went down one percent?
(let ([il-tax% (- il-tax% 1)]) (ma-over-il-saving 10))

;; or change both: how much savings in NH instead of MA?
(let ((tax% 0.0) (il-tax% tax%))
  (ma-over-il-saving 1000))
```

Obviously, this power to customize everything is also the main source of problems with getting no guarantees for code. A common way to get the best of both worlds is to have *controllable* dynamic scope. For example, Common Lisp also has lexical scope everywhere by default, but some variables can be declared as *special*, which means that they are dynamically scoped. The main problem with that is that you can't tell when a variable is special by just looking at the code that uses it, so a more popular approach is the one that is used in Racket: all bindings are always lexically scoped, but there are *parameters* which are a kind of dynamically scoped value containers — but they are bound to plain (lexically scoped) identifiers. Here's the same code as above, translated to Racket with parameters:

```
#lang racket

(define tax% (make-parameter 6.5))  ; create the dynamic container
(define (with-tax n)
  (+ n (* n (/ (tax%) 100))))       ; note how its value is accessed

(with-tax 10) ; how much do we pay?
(parameterize ([tax% 17.0]) (with-tax 10)) ; not a `let'

;; make that into a function
(define il-tax% (make-parameter 17.0))
(define (ma-over-il-saving n)
  (- (parameterize ([tax% (il-tax%)]) (with-tax n))
     (with-tax n)))

(ma-over-il-saving 10)
(parameterize ([il-tax% (- (il-tax%) 1)]) (ma-over-il-saving 10))
```

The main point here is that the points where a dynamically scoped value is used are under the programmer's control — you cannot "customize" what `-` is doing, for example. This gives us back the guarantees that we like to have (= that code works), but of course these points are pre-determined, unlike an environment where everything can be customized including things that are unexpectedly useful.

> *As a side-note, after many decades of debating this, Emacs has finally added lexical scope in its core language, but this is still determined by a flag — a global `lexical-binding` variable.*