PL: Lecture #10

Tuesday, October 8th

Dynamic versus Lexical Scope

And back to the discussion of whether we should use dynamic or lexical scope:

- The most important fact is that we want to view programs as executed by the normal substituting evaluator. Our original motivation was to optimize evaluation only — not to *change* the semantics! It follows that we want the result of this optimization to behave in the same way. All we need is to evaluate:

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

  in the original evaluator to get convinced that `7` should be the correct result (note also that the same code, when translated into Racket, evaluates to `7`).

  (Yet, this is a very important optimization, which without it lots of programs become too slow to be feasible, so you might claim that you're fine with the modified semantics…)

- It does not allow using functions as objects, for example, we have seen that we have a functional representation for pairs:

  ```
  (define (kons x y)
    (lambda (n)
      (match n
        ['first  x]
        ['second y]
        [else (error ...)])))

  (define my-pair (kons 1 2))
  ```

  If this is evaluated in a dynamically-scoped language, we do get a function as a result, but the values bound to `x` and `y` are now gone! Using the substitution model we substituted these values in, but now they were only held in a cache which no has no entries for them…

  In the same way, currying would not work, our nice `deriv` function would not work etc etc etc.

- Makes reasoning impossible, because any piece of code behaves in a way that *cannot* be predicted until run-time. For example, if dynamic scoping was used in Racket, then you wouldn't be able to know what this function is doing:

  ```
  (define (foo)
    x)
  ```

  As it is, it will cause a run-time error, but if you call it like this:

  ```
  (let ([x 1])
    (foo))
  ```

  then it will return `1`, and if you later do this:

  ```
  (define (bar x)
    (foo))
  ```

```
(let ([x 1])
  (bar 2))
```

then you would get `2`!

These problems can be demonstrated in Emacs Lisp too, but Racket goes one step further — it uses the same rule for evaluating a function as well as its values (Lisp uses a different name-space for functions). Because of this, you cannot even rely on the following function:

```
(define (add x y)
  (+ x y))
```

to always add `x` and `y`! — A similar example to the above:

```
(let ([+ -])
  (add 1 2))
```

would return `-1`!

- Many so-called “scripting” languages begin their lives with dynamic scoping. The main reason, as we’ve seen, is that implementing it is extremely simple (no, *nobody* does substitution in the real world! (Well, *almost* nobody…)).

  Another reason is that these problems make life impossible if you want to use functions as object like you do in Racket, so you notice them very fast — but in a `normal` language without first-class functions, problems are not as obvious.

- For example, bash has `local` variables, but they have dynamic scope:

  ```
  x="the global x"
  print_x() { echo "The current value of x is \"$x\""; }
  foo() { local x="x from foo"; print_x; }
  print_x; foo; print_x
  ```

  Perl began its life with dynamic scope for variables that are declared `local`:

  ```
  $x="the global x";
  sub print_x { print "The current value of x is \"$x\"\n"; }
  sub foo { local($x); $x="x from foo"; print_x; }
  print_x; foo; print_x;
  ```

  When faced with this problem, “the Perl way” was, obviously, not to remove or fix features, but to pile them up — so `local` *still* behaves in this way, and now there is a `my` declaration which achieves proper lexical scope (and every serious Perl programmer knows that you should always use `my`)…

  There are other examples of languages that changed, and languages that want to change (e.g, nobody likes dynamic scope in Emacs Lisp, but there’s just too much code now).

- This is still a tricky issue, like any other issue with bindings. For example, googling got me quickly to **a Python blog post** which is confused about what “dynamic scoping” is… It claims that Python uses dynamic scope (Search for “Python uses dynamic as opposed to lexical scoping”), yet python always used lexical scope rules, as can be seen by translating their code to Racket (ignore side-effects in this computation):

  ```
  (define (orange-juice)
    (* x 2))
  (define x 3)
  (define y (orange-juice)) ; y is now 6
  (define x 1)
  (define y (orange-juice)) ; y is now 2
  ```

  or by trying this in Python:

```
def orange_juice():
  return x*2
def foo(x):
  return orange_juice()
foo(2)
```

The real problem of python (pre 2.1, and pre 2.2 without the funny

```
from __future__ import nested_scope
```

line) is that it didn't create closures, which we will talk about shortly.

- Another example, which is an indicator of how easy it is to mess up your scope is the following Ruby bug — running in `irb`:

  ```
  % irb
  irb(main):001:0> x = 0
  => 0
  irb(main):002:0> lambda{|x| x}.call(5)
  => 5
  irb(main):003:0> x
  => 5
  ```

  (This is a bug due to weird scoping rules for variables, which was fixed in newer versions of Ruby. See **this Ruby rant** for details, or read about **Ruby and the principle of unwelcome surprise** for additional gems (the latter is gone, so you'll need the **web archive** to read it).)
- Another thing to consider is the fact that compilation is something that you do based only on the lexical structure of programs, since compilers never actually run code. This means that dynamic scope makes compilation close to impossible.
- There are some advantages for dynamic scope too. Two notable ones are:
  - Dynamic scope makes it easy to have a "configuration variable" easily change for the extent of a calling piece of code (this is used extensively in Emacs, for example). The thing is that usually we want to control which variables are "configurable" in this way, statically scoped languages like Racket often choose a separate facility for these. To rephrase the problem of dynamic scoping, it's that *all* variables are modifiable.

    The same can be said about functions: it is sometimes desirable to change a function dynamically (for example, see "Aspect Oriented Programming"), but if there is no control and all functions can change, we get a world where no code can every be reliable.
  - It makes recursion immediately available — for example,

    ```
    {with {f {fun {x} {call f x}}}
      {call f 0}}
    ```

    is an infinite loop with a dynamically scoped language. But in a lexically scoped language we will need to do some more work to get recursion going.

Implementing Lexical Scope: Closures and Environments

So how do we fix this?

Lets go back to the root of the problem: the new evaluator does not behave in the same way as the substituting evaluator. In the old evaluator, it was easy to see how functions can behave as objects that remember values. For example, when we do this:

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

the result was a function value, which actually was the syntax object for this:

```
{fun {y} {+ 1 y}}
```

Now if we call this function from someplace else like:

```
{with {f {with {x 1} {fun {y} {+ x y}}}}
  {with {x 2}
    {call f 3}}}
```

it is clear what the result will be: f is bound to a function that adds 1 to its input, so in the above the later binding for `x` has no effect at all.

But with the caching evaluator, the value of

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

is simply:

```
{fun {y} {+ x y}}
```

and there is no place where we save the 1 — *that's* the root of our problem. (That's also what makes people suspect that using `lambda` in Racket and any other functional language involves some inefficient code-recompiling magic.) In fact, we can verify that by inspecting the returned value, and see that it does contain a free identifier.

Clearly, we need to create an object that contains the body and the argument list, like the function syntax object — but we don't do any substitution, so in addition to the body an argument name(s) we need to remember that we still need to substitute `x` by `1` . This means that the pieces of information we need to know are:

```
- formal argument(s):    {y}
- body:                  {+ x y}
- pending substitutions: [1/x]
```

and that last bit has the missing `1`. The resulting object is called a `closure` because it closes the function body over the substitutions that are still pending (its environment).

So, the first change is in the value of functions which now need all these pieces, unlike the `Fun` case for the syntax object.

A second place that needs changing is the when functions are called. When we're done evaluating the `call` arguments (the function value and the argument value) but before we apply the function we have two *values* — there is no more use for the current substitution cache at this point: we have finished dealing with all substitutions that were necessary over the current expression — we now continue with evaluating the body of the function, with the new substitutions for the formal arguments and actual values given. But the body itself is the same one we had before — which is the previous body with its suspended substitutions that we *still* did not do.

Rewrite the evaluation rules — all are the same except for evaluating a `fun` form and a `call` form:

```
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}, sc>
eval({call E1 E2},sc1)    = eval(B,extend(x,eval(E2,sc1),sc2))
                              if eval(E1,sc1) = <{fun {x} B}, sc2>
                          = error!  otherwise
```

As a side note, these substitution caches are a little more than “just a cache” now — they actually hold an *environment* of substitutions in which expression should be evaluated. So we will switch to the common *environment* name now.:

```
eval(N,env)                = N
eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
eval({* E1 E2},env)        = eval(E1,env) * eval(E2,env)
eval({/ E1 E2},env)        = eval(E1,env) / eval(E2,env)
eval(x,env)                = lookup(x,env)
eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
eval({fun {x} E},env)      = <{fun {x} E}, env>
eval({call E1 E2},env1)    = eval(B,extend(x,eval(E2, env1),env2))
                              if eval(E1,env1) = <{fun {x} B}, env2>
                           = error!  otherwise
```

In case you find this easier to follow, the “flat algorithm” for evaluating a `call` is:

```
1. f := evaluate E1 in env1
2. if f is not a <{fun ...},...> closure then error!
3. x := evaluate E2 in env1
4. new_env := extend env_of(f) by mapping arg_of(f) to x
5. evaluate (and return) body_of(f) in new_env
```

Note how the scoping rules that are implied by this definition match the scoping rules that were implied by the substitution-based rules. (It should be possible to prove that they are the same.)

The changes to the code are almost trivial, except that we need a way to represent `<{fun {x} B}, env>` pairs.

The implication of this change is that we now cannot use the same type for function syntax and function values since function values have more than just syntax. There is a simple solution to this — we never do any substitutions now, so we don’t need to translate values into expressions — we can come up with a new type for values, separate from the type of abstract syntax trees.

When we do this, we will also fix our hack of using FLANG as the type of values: this was merely a convenience since the AST type had cases for all kinds of values that we needed. (In fact, you should have noticed that Racket does this too: numbers, strings, booleans, etc are all used by both programs and syntax representation (s-expressions) — but note that function values are *not* used in syntax.) We will now implement a separate `VAL` type for runtime values.

First, we need now a type for such environments — we can use `Listof` for this:

```
;; a type for environments:
(define-type ENV = (Listof (List Symbol VAL)))
```

but we can just as well define a new type for environment values:

```
(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])
```

Reimplementing `lookup` is now simple:

```
(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, return its value or throw an
;; error if it isn't bound
```

```
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))
```

… we don't need `extend` because we get `Extend` from the type definition, and we also get `(EmptyEnv)` instead of `empty-subst`.

We now use this with the new type for values — two variants of these:

```
(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV]) ; arg-name, body, scope
```

And now the new implementation of `eval` which uses the new type and implements lexical scope:

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

We also need to update `arith-op` to use `VAL` objects. The full code follows — it now passes all tests, including the example that we used to find the problem.

```
;; The Flang interpreter, using environments

#lang pl

#|
The grammar:
  <FLANG> ::= <num>
            | { + <FLANG> <FLANG> }
            | { - <FLANG> <FLANG> }
            | { * <FLANG> <FLANG> }
            | { / <FLANG> <FLANG> }
            | { with { <id> <FLANG> } <FLANG> }
            | <id>
            | { fun { <id> } <FLANG> }
            | { call <FLANG> <FLANG> }
```

```
Evaluation rules:
  eval(N,env)                = N
  eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
  eval({* E1 E2},env)        = eval(E1,env) * eval(E2,env)
  eval({/ E1 E2},env)        = eval(E1,env) / eval(E2,env)
  eval(x,env)                = lookup(x,env)
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env1)    = eval(B,extend(x,eval(E2,env1),env2))
                               if eval(E1,env1) = <{fun {x} B}, env2>
                             = error!  otherwise
|#

(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)))

;; Types for environments, values, and a lookup function

(define-type ENV
```

```
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV])

(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, return its value or throw an
;; error if it isn't bound
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))

(: NumV->number : VAL -> Number)
;; convert a FLANG runtime numeric value to a Racket one
(define (NumV->number val)
  (cases val
    [(NumV n) n]
    [else (error 'arith-op "expected a number, got: ~s" val)]))

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

(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add l r) (arith-op + (eval l env) (eval r env))]
    [(Sub l r) (arith-op - (eval l env) (eval r env))]
    [(Mul l r) (arith-op * (eval l env) (eval r env))]
    [(Div l r) (arith-op / (eval l env) (eval r env))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (eval named-expr env) env))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr env))
     (cases fval
       [(FunV bound-id bound-body f-env)
        (eval bound-body
              (Extend bound-id (eval arg-expr env) f-env))]
       [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) (EmptyEnv))])
    (cases result
```

```
      [(NumV 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 "{with {x 3}
              {with {f {fun {y} {+ x y}}}
                {with {x 5}
                  {call f 4}}}}")
      => 7)
(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 "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```

Fixing an Overlooked Bug

Incidentally, this version fixes a bug we had previously in the substitution version of FLANG:

```
(run "{with {f {fun {y} {+ x y}}}
        {with {x 7}
          {call f 1}}}")
```

This bug was due to our naive `subst`, which doesn't avoid capturing renames. But note that since that version of the evaluator makes its way from the outside in, there is no difference in semantics for *valid* programs — ones that don't have free identifiers.

(Reminder: This was *not* a dynamically scoped language, just a bug that happened when `x` wasn't substituted away before `f` was replaced with something that refers to `x`.)