CS4613 Lecture 8: Dynamic and Lexical Scope

David Bremner

March 21, 2023

Environments

- PLAI: Evaluating Functions (this part is closer to the book)
- Evaluating using substitutions is very inefficient
- To work around this, we want to use a cache of substitutions.
- We begin evaluating with no cached substitutions, then collect them as we encounter bindings.
- When we reach an identifier it is no longer an error – we must consult the substitution cache at that point.

Environments

- PLAI: Evaluating Functions (this part is closer to the book)
- Evaluating using substitutions is very inefficient
- To work around this, we want to use a cache of substitutions.
- We begin evaluating with no cached substitutions, then collect them as we encounter bindings.
- When we reach an identifier it is no longer an error – we must consult the substitution cache at that point.

1. When evaluating with substitutions, 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**!).

Formal Rules for (Naive) Cached Substitutions

- The formal evaluation rules are now different.
- Evaluation carries along a “substitution cache”

Our substitution rules are replaced by:

```
lookup(x,empty-subst)      = error!
lookup(x,extend(x,E,sc))   = E
lookup(x,extend(y,E,sc))   = lookup(x,sc)
                             if `x' != `y'
```

Now we can write the new rules for ‘eval’...

The change to arithmetic and `lam` is small, just pass an extra parameter.

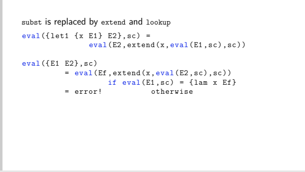
```
eval(N,sc)                      = N
eval({+ E1 E2},sc)              = eval(E1,sc) + eval(E2,sc)
eval({lam x E},sc)            = {lam x E}
```

Identifiers need to be looked up:

```
eval(x,sc)                        = lookup(x,sc)
```

`subst` is replaced by `extend` and `lookup`

```
eval({let1 {x E1} E2},sc) =
              eval(E2,extend(x,eval(E1,sc),sc))

eval({E1 E2},sc)
         = eval(Ef,extend(x,eval(E2,sc),sc))
                 if eval(E1,sc) = {lam x Ef}
         = error!          otherwise
```



`subst` is replaced by `extend` and `lookup`

```
eval({let1 {x E1} E2},sc) =
               eval(E2,extend(x,eval(E1,sc),sc))

eval({E1 E2},sc)
          = eval(Ef,extend(x,eval(E2,sc),sc))
                    if eval(E1,sc) = {lam x Ef}
          = error!             otherwise
```

1. 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.
2. 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 calls with substitution caches

```
[(Call fun-expr arg-expr)
 (let ([fval (interp fun-expr sc)]
       [aval (interp arg-expr sc)])
   (type-case FLANG fval
     [(Lam bound-id bound-body)
      (interp bound-body (extend bound-id aval sc))]
     [else (error 'eval
                  (string-append "non-function: "
                                 (to-string fval)))]))]
```

Testing the new evaluator

First, some fairly fancy looking things work:

```
(test (run `{{lam x {+ x 1}} 4}) 5)

(test (run `{let1 {add3 {lam x {+ x 3}}}
             {add3 1}})
      4)

(test (run `{let1 {add3 {lam x {+ x 3}}}
             {let1 {add1 {lam x {+ x 1}}}
               {let1 {x 3}
                 {add1 {add3 x}}}}})
      7)
```

By tracing, we see the substution cache acts like a stack

12
```
(trace lookup)
(test (run `{let1 {identity {lam x x}}
              {let1 {foo {lam x {+ x 1}}}
                {{identity foo}
                      123}}})
      124)
```

OTOH, we have three failing tests; the reasons look different, but turn out to be related.

```
(test (run `{let1 {x 3}
              {let1 {f {lam y {+ x y}}}
                {let1 {x 5}
                  {f 4}}}})    7)
(test (run `{{let1 {x 3}
                {lam y {+ x y}}}
                  4})    7)
(test (run `{{{lam x {x 1}}
                        {lam x {lam y {+ x y}}}}
                  123})
      124)
```

Dynamic and Lexical Scope

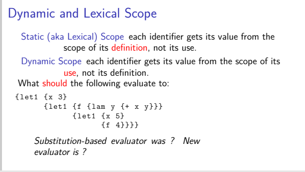
Static (aka Lexical) Scope each identifier gets its value from the scope of its definition, not its use.

Dynamic Scope each identifier gets its value from the scope of its use, not its definition.

What should the following evaluate to:

```
{let1 {x 3}
      {let1 {f {lam y {+ x y}}}
            {let1 {x 5}
                  {f 4}}}}
```

Substitution-based evaluator was ? New evaluator is ?



Dynamic and Lexical Scope

Static (aka Lexical) Scope each identifier gets its value from the scope of its definition, not its use.

Dynamic Scope each identifier gets its value from the scope of its use, not its definition.

What should the following evaluate to:

```
{let1 {x 3}
      {let1 {f {lam y {+ x y}}}
            {let1 {x 5}
                  {f 4}}}}
```

Substitution-based evaluator was ? New evaluator is ?

1. Scope has been problem for *many* language implementors, including the first version of Lisp.
2. 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 one of the only **live** dialects of Lisp that is still dynamically scoped by default. That default is changing as 2021, although there still exists a large body of dynamically scoped elisp.

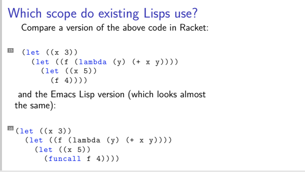
Which scope do existing Lisps use?

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 scope do existing Lisps use?

Compare a version of the above code in Racket:

15
```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (f 4))))
```

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

16
```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (funcall f 4))))
```

In Emacs 24, emacs-lisp acquires optional lexical scope, on a file by file basis. In Emacs 27

Racket `plai-dynamic`

16a
```
#lang plai-dynamic
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (f 4))))
```

- demonstrates maleability of Racket
- internally uses *parameters*
- is a quick and dirty hack

What happens when we use another function on the way?:

17
```
(define (blah func val) (func val))

(let ([x 3])
  (let ([f (λ (y) (+ x y))])
    (let ([x 5])
      (blah f 4))))
```

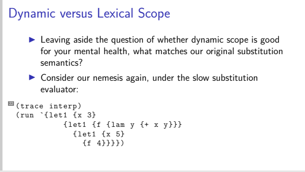
Note that renaming identifiers can lead to different results

18
```
(define  (blah func x) (func x))

(let ([x 3])
  (let ([f (λ (y) (+ x y))])
    (let ([x 5])
      (blah f 4))))
```

Dynamic versus Lexical Scope

- Leaving aside the question of whether dynamic scope is good for your mental health, what matches our original substitution semantics?
- Consider our nemesis again, under the slow substitution evaluator:

22
```
(trace interp)
(run `{let1 {x 3}
           {let1 {f {lam y {+ x y}}}
             {let1 {x 5}
               {f 4}}}})
```



Dynamic versus Lexical Scope

- Leaving aside the question of whether dynamic scope is good for your mental health, what matches our original substitution semantics?
- Consider our nemesis again, under the slow substitution evaluator:

```
(trace interp)
(run `{let1 {x 3}
          {let1 {f {lam y {+ x y}}}
            {let1 {x 5}
              {f 4}}}})
```

1. Subsitution caching 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...

- Dynamic scope means no closures, no partial evaluation.
- In a dynamic scoped language, you don't even know if this is valid code until run time:

23
```
 (define (foo) x)
```

Racket uses the same rule for evaluating a function as well as its values. This makes dynamic scope more powerful/dangerous:

24
```
(define (add x y)
      (+ x y))

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

CS4613 Lecture 8: Dynamic and Lexical Scope
└Scope

1. Lisp-2's uses a different name-space for functions

- Dynamic scope means no closures, no partial evaluation.
- In a dynamic scoped language, you don't even know if this is valid code until run time:

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

Racket uses the same rule for evaluating a function as well as its values. This makes dynamic scope more powerful/dangerous:

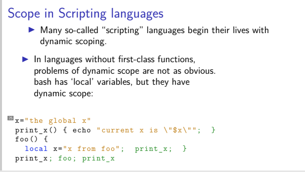
```
(define (add x y)
        (+ x y))

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

Scope in Scripting languages

- Many so-called “scripting” languages begin their lives with dynamic scoping.

- In languages without first-class functions, problems of dynamic scope are not as obvious. bash has ‘local’ variables, but they have dynamic scope:

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



Scope in Scripting languages

- Many so-called “scripting” languages begin their lives with dynamic scoping.
- In languages without first-class functions, problems of dynamic scope are not as obvious. bash has ‘local’ variables, but they have dynamic scope:

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

1. The main reason for starting with dynamic scope, as we’ve seen, is that implementing it is extremely simple (no, **nobody** does substitution in the real world! (Well, **almost** nobody...)).

Function scope in JavaScript

Pre-ES2015, JavaScript only had function scope.

```
function f1() {
    var x = 1;
    {
        var x = 2;
    }
    console.log(x);
}
f1();
```

- Python had something similar pre-2.1
- These days JavaScript has `let`, which behaves sanely.

There are some advantages for dynamic scope.

  - Dynamic scope makes it easy to have a "configuration variable" easily change for the extent of a calling piece of code.
  - Optional dynamically scoped variables are useful

- the problem of dynamic scoping is that **all** variables are modifiable.
- It is sometimes desirable to change a function dynamically (for example, see "Aspect Oriented Programming"), but if all functions can change, no code can be reliable.

- Dynamic scoping makes recursion immediately available – for example, dynamic scope gives us easy loops:

```
{let1 {f
       {lam x
            {f x}}}
      {f 0}}
```

- We'll see later that adding recursion with lexical scope is less trivial.

Controlled Dynamic Scope

racket provides “parameters” for dynamic scope, internally used by plai-dynamic

34
```
(define location (make-parameter "here"))
(define (foo)
  (location))
(parameterize ([location "there"])
  (foo))
(foo)
(parameterize ([location "in a house"])
  (list (foo)
        (parameterize
            ([location "with a mouse"])
          (foo))
        (foo)))
(location)
```