CS4613 Lecture 7 Environments

David Bremner

March 21, 2023

Implementing Lexical Scope: Closures and Environments

- How do we preserve the original substitution behaviour, while avoiding duplicate computation?
- In the substitution evaluator,

```
{let1 {x 1}
  {lam y
    {+ x y}}}
```

returns

```
{lam y {+ 1 y}}
```

- Now we are “immune” to re-binding

```
{let1 {f {let1 {x 1} {lam y {+ x y}}}}
  {let1 {x 2}
    {f 3}}}
```

- `f` is bound to a function that adds 1 to its input,
- `x` doesn't even appear, so rebinding it around the call does nothing.

With a naive caching evaluator, the value of

```
{let1 {x 1}
  {lam y
    {+ x y}}}
```

is simply:

```
{lam y {+ x y}}
```

root problem there is no place where we save the 1

- The returned expr. contains a free identifier.
- we need a value that contains the body and the argument list, like the function syntax object
- we need to remember that we still need to replace x by 1.

With a naive caching evaluator, the value of

```
{let1 {x 1}
  {lam y
    {+ x y}}}
```

is simply:

```
{lam y {+ x y}}
```

root problem there is no place where we save the 1

- The returned expr. contains a free identifier.
- we need a value that contains the body and the argument list, like the function syntax object
- we need to remember that we still need to replace x by 1.

1. That's also what makes people suspect that using 'lambda' in Racket and any other functional language involves some inefficient code-recompiling magic.

New Function Values

```
{let1 {x 1}
  {lam y
    {+ x y}}}
```

formal argument(s) `y`

body `{+ x y}`

pending substitutions `[1/x]`

Closures

- The resulting object is called a *closure* because it closes the function body over the substitutions that are still pending (its environment).
- FLANG functions will need to evaluate to some type representing a closure.

(Eagerly) Evaluating calls

- First we evaluate the function value and the argument value to yield two values

```
{f 3}, [] =>
        FunVal = < {lam y {+ x y}} , [x=1] >
        Arg =  < 3 >
```

- we now continue with evaluating the body, with the new substitutions for the formal arguments and actual values given.

```
{+ x y}, [y=3, x=1]
; look ma, no substitution
```

(Eagerly) Evaluating calls

- First we evaluate the function value and the argument value to yield two values

```
{f 3}, [] =>
       FunVal = < {lam y {+ x y}} , [x=1] >
       Arg =  < 3 >
```

- we now continue with evaluating the body, with the new substitutions for the formal arguments and actual values given.

```
{+ x y}, [y=3, x=1]
; look ma, no substitution
```

1. we have finished dealing with all substitutions that were necessary over the current expression

- Rewrite the evaluation rules – Most are the same

```
eval(N,sc)           = N
eval({+ E1 E2},sc)   = eval(E1,sc) + eval(E2,sc)
; ...
eval(x,sc)                 = lookup(x,sc)
eval({let1 {x E1} E2},sc) =
   eval(E2,extend(x,eval(E1,sc),sc))
```

- Except for evaluating a ‘lam’ form and a call

```
eval({lam x E},sc)        = <{lam x E}, sc>
eval({E1 E2},sc1)
          = eval(Ef,extend(x,eval(E2,sc1),sc2))
                           if eval(E1,sc1) =
                              <{lam x Ef}, sc2>
          = error!         otherwise
```

- These substitution caches are more than "just a cache" now – they hold an *environment* of definitions. So we will switch terminology...

Substitution Caches are Environments

```
eval({lam x E},env)     = <{lam x E}, env>
eval({E1 E2},env1)   =
     if eval(E1,env1) = <{lam x Ef}, env2> then
            eval(Ef,extend(x,eval(E2,env1),env2))
     else
            error!
```

Evaluation step by step

To evaluate `{E1 E2}` in `env1`:

- f := evaluate E1 in env1
- if f is not a `<{lam ...}, ...>` closure then error!
- a := evaluate E2 in env1
- `new_env` := extend `env_of(f)` by `[arg_of(f)= a]`
- evaluate (and return) `body_of(f)` in `new_env`

Evaluation step by step

To evaluate `{E1 E2}` in `env1`:

- f := evaluate E1 in env1
- if f is not a `<{lam ...}, ...>` closure then error!
- a := evaluate E2 in env1
- `new_env` := extend `env_of(f)` by `[arg_of(f)= a]`
- evaluate (and return) `body_of(f)` in `new_env`

1. Note how the implied scoping rules match substitution-based rules.
2. The changes to the code are almost trivial, except that we need a way to represent ⟨lam x Ef, env⟩ pairs.

- We need distinct types for function **syntax** and function **values**
- We never go back from values to syntax now, which simplifies things.
- We will now implement a separate 'VAL' type for runtime values.

- We need distinct types for function **syntax** and function **values**
- We never go back from values to syntax now, which simplifies things.
- We will now implement a separate 'VAL' type for runtime values.

1. 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.

- Thus, we need now a pair of types for our environments

```
(define-type ENV
  [EmptyEnv]
  [Extend (name : Symbol) (val : VAL)
          (rest : ENV)])

(define-type VAL
  [NumV (n : Number)]
  [FunV (arg : Symbol) (body : FLANG)
        (env : ENV)])
```

- we get ‘Extend’ from the type definition,
- we also get ‘(EmptyEnv)’ instead of ‘empty-subst’.

Reimplementing ‘lookup’ is now simple:

14

```
(define (lookup name env)
  (type-case ENV env
    [(EmptyEnv)  (error 'lookup "no binding")]
    [(Extend id val rest-env)
     (if (eq? id name)
         val
         (lookup name rest-env))]))
```

```
  ;; evaluates FLANGs by reducing them to VALs
  (define (interp expr env)
    (type-case FLANG expr
; ⋮
        [(Lam bound-id bound-body)
         (FunV bound-id bound-body env)]
        [(Call fun-expr arg-expr)
         (let ([fval (interp fun-expr env)])
           (type-case VAL fval
             [(FunV bound-id bound-body f-env)
              (interp
               bound-body
               (Extend
                bound-id
                (interp arg-expr env) f-env))]
             [else
              (error 'eval "not a function")]))]))
```

- We also need to update 'arith-op' to use VAL objects.

```
;; gets a Racket numeric binary operator,
;; uses it within a NumV wrapper
(define (arith-op op val1 val2)
  (local
      [(define (NumV->number v)
         (type-case VAL v
           [(NumV n) n]
           [else (error 'arith-op "not a number")]))]
    (NumV (op (NumV->number val1)
              (NumV->number val2)))))
```

- Finally we need to change `run` to use the new environment syntax

```
;; evaluate a FLANG program contained in an s-expression
(define (run s-exp)
  (let ([result (interp (parse-sx s-exp) (EmptyEnv))])
    (type-case VAL result
      [(NumV n) n]
      [else (error 'run "non-number")])))
```

Naively passing tests, new evaluator

```
(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)



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

Naively failing tests, new evaluator

19
```
(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)
```

Fixing a Bug

- this version fixes a bug we had previously in the substitution version of FLANG.
- bug is present for eager or lazy evaluator because of `lam`
- No change for correct code, but avoids name capture for code with free identifiers.

```
(run `{let1 {f {lam y {+ x y}}}
        {let1 {x 7}
          {f 1}}})
```

- compare with the substitution version (this highlights the connection between functions and laziness)

```
(run `{let1 {f {lam y {+ x y}}}
          {let1 {x 7}
            {f 1}}})
```