Implementing Lexical Scope: Closures and Environments

- How do we recover the original substitution behaviour?
- In the substitution evaluator,

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

returns

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

- Now we are "immune" to re-binding

2

```
{with {f {with {x 1} {fun {y} {+ x y}}}}
  {with {x 2}
    {call 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 the caching evaluator, the value of

3

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

is simply:

4

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

- there is no place where we save the 1 – **that** is the root of our problem.
- The returned expression contains a free identifier.
- we need to create an object that contains the body and the argument list, like the function syntax object
- There is no substitution, so we need to remember that we still need to substitute x by 1.

2019-02-12

- With the caching evaluator, the value of

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

is simply:

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

- there is no place where we save the 1 – **that** is the root of our problem.
- The returned expression contains a free identifier.
- we need to create an object that contains the body and the argument list, like the function syntax object
- There is no substitution, so we need to remember that we still need to substitute 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

- formal argument(s):

5 `{y}`

- body:

6 `{+ x y}`

- pending substitutions:

7 `[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

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

- there is no more need for the current substitution cache at this point
- we now continue with evaluating the body, with the new substitutions for the formal arguments and actual values given.

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

2019-02-12

(Eagerly) Evaluating calls

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

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

- there is no more need for the current substitution cache at this point
- we now continue with evaluating the body, with the new substitutions for the formal arguments and actual values given.

```
9    {+ 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

10
```
eval(N,sc)              = N
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))
```

- Except for evaluating a 'fun' form and a 'call' form:

11
```
eval({fun {x} E},sc)         = <{fun {x} E}, sc>
eval({call E1 E2},sc1)
          = eval(Ef,extend(x,eval(E2,sc1),sc2))
                                  if eval(E1,sc1) =
                                     <{fun {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:

12

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

Evaluation step by step

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

- f := evaluate E1 in env1
- if f is not a <`{fun ...}`,`...`> 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 `{call E1 E2}` in `env1`:

- f := evaluate E1 in env1
- if f is not a `<{fun ...},...>` 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 ⟨fun 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

13

```
(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:


```
(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 (eval expr env)
  (type-case FLANG expr
    [(Fun bound-id bound-body)
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (type-case VAL fval
         [(FunV bound-id bound-body f-env)
          (eval bound-body
                (Extend bound-id (eval arg-expr
                   env) f-env))]
         [else (error 'eval
                      (string-append "`call'
                         expects a function, got:
                         "
                            (to-string
                               fval)))]))]))
```

- 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
                        (string-append
                          "expects a number,
                          got: " (to-string
                          v)))]))]
    (NumV (op (NumV->number val1)
              (NumV->number val2)))))
```

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

17

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

Previously passing tests, new evaluator

18

```
(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)
```

Previously failing tests, new evaluator


```
(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 `{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x
                           y}}}}
                  123})
      124)
```

Fixing a Bug

- this version fixes a bug we had previously in the substitution version of FLANG.
- No change for correct code, but avoids name capture for code with free identifiers.

20

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

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

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