More Closures (on both levels)

- Racket closures that encapsulate values can also encapsulate **behaviour**
- Currently we use a variant type to represent FLANG closures

1

```
[FunV (arg : Symbol) (body : FLANG) (env :
    ENV)])
```

- We can replace this by a function value, which will encapsulate the three values

2

```
(define-type VAL
  [NumV (n : number)]
  [FunV (f : (? -> ?))])
```

- We need this information for generating an FLANG closure in the ‘Fun’ case, and using it in the ‘Call’ case:

3

```
[(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))]
                       ⋮
```

- The trick will be to move a piece of the eval function into the FunV object.

4

```
(lambda (arg-val)
  (eval bound-body
        (Extend bound-id arg-val env)))
```

where the values of ‘bound-body’, ‘bound-id’, and ‘val’ are known at the time that the FunV is **constructed**.

- Doing this gives us the following code for Fun:

5

```
[(Fun bound-id bound-body)
 (FunV (lambda (arg-val)
         (eval bound-body
               (Extend bound-id arg-val
                  env))))]
```

- and Call

6

```
[(Call fun-expr arg-expr)
 (let ([fval (eval fun-expr env)])
   (type-case VAL fval
     [(FunV proc) (proc (eval arg-expr env))]
     [else (error 'eval "`call' expects a
        function")]))]
```

- And now the type of the function is clear:

7

```
(define-type VAL
  [NumV (n : number)]
  [FunV (f : (VAL -> VAL))])
```

As before, the rest of the code is unmodified; we can even choose closure implementation independently from environment implementation.

8

```
(define foo (eval (parse-sx `{fun {x} {+ x 1}})
                  (EmptyEnv)))
(define bar (eval (parse-sx `41) (EmptyEnv)))

(define (funcall fun-val arg)
  (type-case VAL fun-val
       [(FunV f)  (f arg)]
       [else (error 'funcall "oops")]))

(funcall foo bar)
```

Types of Evaluators

- We (just) saw implementing FLANG closures and environments using the corresponding host language features.
- Consider inheriting short circuit evaluation from C:

```
9 Racket_obj *And( int argc, Racket_obj *argv[]
    ) {
  Racket_obj *tmp;
  if ( argc != 2 )
    signal_racket_error("need 2 args");
  else if (racket_eval(argv[0]) !=
      racket_false &&
             (tmp = racket_eval(argv[1])) !=
                racket_false )
    return tmp;
  else
    return racket_false;
}
```

- We can also use `if` from the host language

10

```
// Disclaimer: not real Racket code
Racket_obj *eval_and( int argc, Racket_obj
    *argv[] )
{
  Racket_obj *tmp;
  if ( argc != 2 )
    signal_racket_error("bad number of
        arguments");
  else if ( racket_eval(argv[0]) !=
      racket_false )
    return racket_eval(argv[1]);
  else
    return racket_false;
}
```

How meta is your evaluator?

- A ⟨⟨syntactic evaluator⟩⟩ implements all target language behavior explicitly.
- A ⟨⟨meta evaluator⟩⟩ is an evaluator that uses language features of the host language to directly implement behavior of the evaluated language.
- our substitution-based FLANG evaluator was **close** to being a syntactic evaluator
- All of our evaluators rely on e.g. Racket arithmetic

- meta evaluators are easy **exactly** when there is a close match between host and target language.
- We can make our evaluator a meta evaluator by removing the encapsulation of FLANG values in a VAL type.
- This is so close to Racket, we can say something stronger.
- A ⟨⟨meta-circular evaluator⟩⟩ is a meta evaluator in which the implementation and the evaluated languages are the same.

2019-03-11

- meta evaluators are easy **exactly** when there is a close match between host and target language.
- We can make our evaluator a meta evaluator by removing the encapsulation of FLANG values in a VAL type.
- This is so close to Racket, we can say something stronger.
- A ⟨⟨meta-circular evaluator⟩⟩ is a meta evaluator in which the implementation and the evaluated languages are the same.

1. Put differently, the trivial nature of the evaluator clues us in to the deep connection between the two languages, whatever their syntactic differences may be.

Feature Embedding

- We saw that the difference between lazy evaluation and eager evaluation is in the evaluation rules for 'with' forms, function applications, etc:
- eager:

```
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
```

- lazy:

```
eval({with {x E1} E2}) = eval(E2[E1/x])
```

- the first rule is eager because of we understand the mathematical notation to be eager.
- Similarly, if Racket args were evaluated lazily, this would be lazy

```
(define (eval expr)
  (type-case FLANG expr
    ⋮
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
      ⋮
    ))
```

- A general phenomena where some of the semantic features of the host language/notation we use gets **embedded** into the language we implement.
- Consider the code that implements arithmetic:

14

```
;; evaluates FLANG expressions by reducing
   them to numbers
(define (eval expr)
  (type-case FLANG expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    ...))
```

- What if it was written like this, would it still implement unlimited integers and exact fractions?

15

```
FLANG eval(FLANG expr) {
  if (is_Num(expr))
    return num_of_Num(expr);
  else if (is_Add(expr))
    return eval(lhs_of_Add(expr)) +
           eval(rhs_of_Add(expr));
  else if ...
  ...
}
```

- The bottom line is that we should be aware of ”inherited” features (or lack thereof), and be very careful when we talk about semantics.
- Even the mathematical language that we use to communicate (semi-formal logic) can mean different things.

- The bottom line is that we should be aware of "inherited" features (or lack thereof), and be very careful when we talk about semantics.
- Even the mathematical language that we use to communicate (semi-formal logic) can mean different things.

1. Aside: read "Reflections on Trusting Trust" by Ken Thompson (You can skip to the "Stage II" part to get to the interesting stuff.)

Yet another FLANG evaluator

- It uses Racket values directly to implement values
- we now fall back more often on what Racket does.
- It is dynamically typed so we use `#lang plait #:untyped`
- The evaluation function is modified as follows

```
(define (eval expr env)
  (type-case FLANG expr
    [(Num n) n]                        ;*** racket
       number
    [(Add l r) (+ (eval l env) (eval r env))]
      ⋮
    [(Call fun-expr arg-expr)
     ((eval fun-expr env)              ;*** now
        trivial
      (eval arg-expr env))]))
```

- We can rely on racket to report type errors

17

```
(run `{+ {fun {x} 3} 1})
```

- Alternatively, we can take a bit more control of the error reporting

18

```
(define (evalN e)
  (let ([n (eval e env)])
    (if (number? n) n
        (error 'eval "got a non-number: ~s"
           n))))
(define (evalF e)
  (let ([f (eval e env)])
    (if (procedure? f) f
        (error 'eval "got a non-function: ~s"
           f))))
```

- Our new `eval` looks like:

```
(define (eval expr env)
  (type-case FLANG expr
    [(Num n) n]
    [(Add l r) (+ (evalN l) (evalN r))]
    [(Fun bound-id bound-body)
     (lambda (arg-val)
       (eval bound-body (Extend bound-id
          arg-val env)))]
    [(Call fun-expr arg-expr)
     ((evalF fun-expr)
      (eval arg-expr env))]))
```

- It now makes (some) sense to allow our run function to return procedures.

20

```
(test  (run `{with {x 1} x}) 1)
(define f (run `{fun {x} {+ x 1}}))
(test (procedure? f) #t)
(when (procedure? f)
  (f 41))
```