PL: Lecture #15

Tuesday, October 29th

Implementing `rec` Using Cyclic Structures

> 👉 ***PLAI §10***

Looking at the arrows in the environment diagrams, what we're really looking for is a closure that has an environment pointer which is the same environment in which it was defined. This will make it possible for `fact` to be bound to a closure that can refer to *itself* since its environment is the same one in which it is defined. However, so far we have no tools that makes it possible to do this.

What we need is to create a "cycle of pointers", and so far we do not have a way of achieving that: when we create a closure, we begin with an environment which is saved in the slot's environment slot, but we want that closure to be the value of a binding in that same environment.

Boxes and Mutation

To actually implement a circular structure, we will now use *side-effects*, using a new kind of Racket value which supports mutation: a box. A box value is built with the `box` constructor:

```
(define my-thing (box 7))
```

the value is retrieved with the `unbox' function,

```
(* 6 (unbox my-thing))
```

and finally, the value can be changed with the `set-box!` function.

```
(set-box! my-thing 17)
(* 6 (unbox my-thing))
```

An important thing to note is that `set-box!` is much like `display` etc, it returns a value that is not printed in the Racket REPL, because there is no point in using the result of a `set-box!`, it is called for the side-effect it generates. (Languages like C blur this distinction between returning a value and a side-effect with its assignment statement.)

As a side note, we now have side effects of two kinds: mutation of state, and I/O (at least the O part). (Actually, there is also infinite looping that can be viewed as another form of a side effect.) This means that we're now in a completely different world, and lots of new things can make sense now. A few things that you should know about:

- We never used more than one expression in a function body because there was no point in it, but now there is. To evaluate a sequence of Racket expressions, you wrap them in a `begin` expression.
- In most places you don't actually need to use `begin` — these are places that are said to have an *implicit* `begin`: the body of a function (or any lambda expression), the body of a `let` (and `let`-relatives), the consequence positions in `cond`, `match`, and `cases` clauses and more. One of the common places where a `begin` is used is in an `if` expression (and some people prefer using `cond` instead when there is more than a single expression).
- `cond` without an `else` in the end can make sense, if all you're using it it for is side-effects.
- `if` could get a single expression which is executed when the condition is true (and an unspecified value is used otherwise), but our language (as well as the default Racket language) always forbids this —

there are convenient special forms for a one-sided `if`s: `when` & `unless`, and they can have any number of expressions (they have an implicit `begin`). They have an advantage of saying “this code does some side-effects here” more explicit.

- There is a function called `for-each` which is just like `map`, except that it doesn’t collect the list of results, it is used only for performing side effects.
- Aliasing and the concept of “object equality”: `equal?` vs `eq?`. For example:

```
(: foo : (Boxof ...) (Boxof ...) -> ...)
(define (foo a b)
  (set-box! a 1)) ;*** this might change b, can check `eq?`
```

When any one of these things is used (in Racket or other languages), you can tell that side-effects are involved, because there is no point in any of them otherwise. In addition, any name that ends with a `!` (“bang”) is used to mark a function that changes state (usually a function that only changes state).

So how do we create a cycle? Simple, boxes can have any value, and they can be put in other values like lists, so we can do this:

```
#lang pl untyped
(define foo (list 1 (box 3)))
(set-box! (second foo) foo)
```

and we get a circular value. (Note how it is printed.) And with types:

```
#lang pl
(: foo : (List Number (Boxof Any)))
(define foo (list 1 (box 3)))
(set-box! (second foo) foo)
```

Types for Boxes

Obviously, `Any` is not too great — it is the most generic type, so it provides the least information. For example, notice that

```
(unbox (second foo))
```

returns the right list, which is equal to `foo` itself — but if we try to grab some part of the resulting list:

```
(second (unbox (second foo)))
```

we get a type error, because the result of the `unbox` is `Any`, so Typed Racket knows nothing about it, and won’t allow you to treat it as a list. It is not too surprising that the type constructor that can help in this case is `Rec` which we have already seen — it allows a type that can refer to itself:

```
#lang pl
(: foo : (Rec this (List Number (Boxof (U #f this)))))
(define foo (list 1 (box #f)))
(set-box! (second foo) foo)
```

Note that either `foo` or the value in the box are both printed with a `Rec` type — the value in the box can’t just have a `(U #f this)` type, since `this` doesn’t mean anything in there, so the whole type needs to still be present.

There is another issue to be aware of with `Boxof` types. For most type constructors (like `Listof`), if `T1` is a subtype of `T2`, then we also know that `(Listof T1)` is a subtype of `(Listof T2)`. This makes the following code typecheck:

```
#lang pl
(: foo : (Listof Number) -> Number)
```

```
(define (foo l)
  (first l))
(: bar : Integer -> Number)
(define (bar x)
  (foo (list x)))
```

Since the `(Listof Integer)` is a subtype of the `(Listof Number)` input for `foo`, the application typechecks. But this is *not* the same for the output type, for example — if we change the `bar` type to:

```
(: bar : Integer -> Integer)
```

we get a type error since `Number` is not a subtype of `Integer`. So subtypes are required to “go higher” on the input side and “lower” on the other. So, in a sense, the fact that boxes are mutable means that their contents can be considered to be on the other side of the arrow, which is why for such `T1` subtype of `T2`, it is `(Boxof T2)` that is a subtype of `(Boxof T1)`, instead of the usual. For example, this doesn’t work:

```
#lang pl
(: foo : (Boxof Number) -> Number)
(define (foo b)
  (unbox b))
(: bar : Integer -> Number)
(define (bar x)
  (: b : (Boxof Integer))
  (define b (box x))
  (foo b))                    ;***
```

And you can see why this is the case — the marked line is fine given a `Number` contents, so if the type checker allows passing in a box holding an integer, then that expression would mutate the contents and make it an invalid value.

However, boxes are not only mutable, they hold a value that can be read too, which means that they’re on *both* sides of the arrow, and this means that `(Boxof T1)` is a subtype of `(Boxof T2)` if `T2` is a subtype of `T1` *and* `T1` is a subtype of `T2` — in other words, this happens only when `T1` and `T2` are the same type. (See below for an extended demonstration of all of this.)

Note also that this demonstration requires that extra `b` definition, if it’s skipped:

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

then this will typecheck again — Typed Racket will just consider the context that requires a box holding a `Number`, and it is still fine to initialize such a box with an `Integer` value.

> *As a side comment, this didn’t always work. Earlier in its existence, Typed Racket would always choose a specific type for values, which would lead to confusing errors with boxes. For example, the above would need to be written as*
>
> ```
> (define (bar x)
>   (foo (box (ann x : Number))))
> ```
>
> *to prevent Typed Racket from inferring a specific type. This is no longer the case, but there can still be some surprises. A similar annotation was needed in the case of a list holding a self-referential box, to avoid the initial `#f` from getting a specific-but-wrong type.*

> *Another way to see the problem these days is to enter the following expressions and see what types Typed Racket guesses for them:*
>
> ```
> > (define a 0)
> > (define b (box 0))
> > a
> - : Integer [more precisely: Zero]        ;***
> 0
> > b
> ```

```
- : (Boxof Integer)                    ;***
'#&0
```

Note that for `a`, the assigned type is very specific, because Typed Racket assumes that it will not change. But with a boxed value, using a type of `(Boxof Zero)` would lead to a useless box, since it'll only allow using `set-box!` with `0`, and therefore can never change. This shows that this is exactly that: a guess given the lack or explicit user-specified type, so there's no canonical guess that can be inferred here.

Boxof's Lack of Subtyping

The lack of any subtype relations between `(Boxof T)` and `(Boxof S)` regardless of `S` and `T` can roughly be explained as follows.

First, a box is a container that you can pull a value out of — which makes it similar to lists. In the case of lists, we have:

```
if:           S   subtype-of          T
then: (Listof S)  subtype-of  (Listof T)
```

This is true for all such containers that you can pull a value out of: if you expect to pull a `T` but you're given a container of a subtype `S`, then things are still fine (you'll get an `S` which is also a `T`). Such "containers" include functions that produce a value — for example:

```
if:         S  subtype-of       T
then:  Q -> S  subtype-of  Q -> T
```

However, functions also have the other side, where things are different — instead of a side of some *produced* value, it's the side of the *consumed* value. We get the opposite rule there:

```
if:    T       subtype-of  S
then:  S -> Q  subtype-of  T -> Q
```

To see why this is right, use `Number` and `Integer` for `S` and `T`:

```
if:    Integer      subtype-of  Number
then:  Number -> Q  subtype-of  Integer -> Q
```

so — if you expect a function that takes an integer, a valid *subtype* value that I can give you is a function that takes a number. In other words, every function that takes a number is also a function that takes an integer, but not the other way.

To summarize all of this, when you make the output type of a function "smaller" (more constrained), the resulting type is smaller (a subset), but on the input side things are flipped — a bigger input type means a more constrained function.

> *The technical names for these properties are: a "covariant" type is one that preserves the subtype relationship, and a "contravairant" type is one that reverses it. (Which is similar to how these terms are used in math.)*

(Side note: this is related to the fact that in logic, `P => Q` is roughly equivalent to `not(P) or Q` — the left side, `P`, is inside negation. It also explains why in `((S -> T) -> Q)` the `S` obeys the first rule, as if it was on the right side — because it's negated twice.)

Now, a `(Boxof T)` is a producer of `T` when you pull a value out of the box, but it's also a consumer of `T` when you put such a value in it. This means that — using the above analogy — the `T` is on both sides of the arrow. This means that

```
if:    S subtype-of T  *and*  T subtype-of S
then:  (Boxof S) subtype-of (Boxof T)
```

which is actually:

```
if:           S   is-the-same-type-as         T
then:  (Boxof S)  is-the-same-type-as  (Boxof T)
```

A different way to look at this conclusion is to consider the function type of `(A -> A)`: when is it a subtype of some other `(B -> B)`? Only when `A` is a subtype of `B` and `B` is a subtype of `A`, which means that this happens only when `A` and `B` are the same type.

> *The term for this is "nonvariant" (or "invariant"): `(A -> A)` is unrelated to `(B -> B)` regardless of how `A` and `B` are related. The only exception is, of course, when they are the same type. The Wikipedia entry about these puts the terms together nicely in the face of mutation:*
>
> > *Read-only data types (sources) can be covariant; write-only data types (sinks) can be contravariant. Mutable data types which act as both sources and sinks should be invariant.*

The following piece of code makes the analogy to function types more formally. Boxes behave as if their contents is on both sides of a function arrow — on the right because they're readable, and on the left because they're writable, which the conclusion that a `(Boxof A)` type is a subtype of itself and no other `(Boxof B)`.

```
#lang pl

;; a type for a "read-only" box
(define-type (Boxof/R A) = (-> A))
;; Boxof/R constructor
(: box/r : (All (A) A -> (Boxof/R A)))
(define (box/r x) (lambda () x))
;; we can see that (Boxof/R T1) is a subtype of (Boxof/R T2)
;; if T1 is a subtype of T2 (this is not surprising, since
;; these boxes are similar to any other container, like lists):
(: foo1 : Integer -> (Boxof/R Integer))
(define (foo1 b) (box/r b))
(: bar1 : (Boxof/R Number) -> Number)
(define (bar1 b) (b))
(test (bar1 (foo1 123)) => 123)

;; a type for a "write-only" box
(define-type (Boxof/W A) = (A -> Void))
;; Boxof/W constructor
(: box/w : (All (A) A -> (Boxof/W A)))
(define (box/w x) (lambda (new) (set! x new)))
;; in contrast to the above, (Boxof/W T1) is a subtype of
;; (Boxof/W T2) if T2 is a subtype of T1, *not* the other way
;; (and note how this is related to A being on the *left* side
;; of the arrow in the `Boxof/W' type):
(: foo2 : Number -> (Boxof/W Number))
(define (foo2 b) (box/w b))
(: bar2 : (Boxof/W Integer) Integer -> Void)
(define (bar2 b new) (b new))
(test (bar2 (foo2 123) 456))

;; combining the above two into a type for a "read/write" box
(define-type (Boxof/RW A) = (A -> A))
;; Boxof/RW constructor
(: box/rw : (All (A) A -> (Boxof/RW A)))
(define (box/rw x) (lambda (new) (let ([old x]) (set! x new) old)))
;; this combines the above two: `A' appears on both sides of the
;; arrow, so (Boxof/RW T1) is a subtype of (Boxof/RW T2) if T1
;; is a subtype of T2 (because there's an A on the right) *and*
```

```
;; if T2 is a subtype of T1 (because there's another A on the
;; left) -- and that can happen only when T1 and T2 are the same
;; type. So this is a type error:
;;   (: foo3 : Integer -> (Boxof/RW Integer))
;;   (define (foo3 b) (box/rw b))
;;   (: bar3 : (Boxof/RW Number) Number -> Number)
;;   (define (bar3 b new) (b new))
;;   (test (bar3 (foo3 123) 456) => 123)
;;   ** Expected (Number -> Number), but got (Integer -> Integer)
;; And this a type error too:
;;   (: foo3 : Number -> (Boxof/RW Number))
;;   (define (foo3 b) (box/rw b))
;;   (: bar3 : (Boxof/RW Integer) Integer -> Integer)
;;   (define (bar3 b new) (b new))
;;   (test (bar3 (foo3 123) 456) => 123)
;;   ** Expected (Integer -> Integer), but got (Number -> Number)
;; The two types must be the same for this to work:
(: foo3 : Integer -> (Boxof/RW Integer))
(define (foo3 b) (box/rw b))
(: bar3 : (Boxof/RW Integer) Integer -> Integer)
(define (bar3 b new) (b new))
(test (bar3 (foo3 123) 456) => 123)
```

Implementing a Circular Environment

We now use this to implement `rec` in the following way:

1. Change environments so that instead of values they hold boxes of values: `(Boxof VAL)` instead of `VAL`, and whenever `lookup` is used, the resulting boxed value is unboxed,
2. In the `WRec` case, create the new environment with some temporary binding for the identifier — any value will do since it should not be used (when named expressions are always `fun` expressions),
3. Evaluate the expression in the new environment,
4. Change the binding of the identifier (the box) to the result of this evaluation.

The resulting definition is:

```
(: extend-rec : Symbol FLANG ENV -> ENV)
;; extend an environment with a new binding that is the result of
;; evaluating an expression in the same environment as the extended
;; result
(define (extend-rec id expr rest-env)
  (let ([new-cell (box (NumV 42))])
    (let ([new-env (Extend id new-cell rest-env)])
      (let ([value (eval expr new-env)])
        (set-box! new-cell value)
        new-env))))
```

Racket has another `let` relative for such cases of multiple-nested `let`s — `let*`. This form is a derived form — it is defined as a shorthand for using nested `let`s. The above is therefore exactly the same as this code:

```
(: extend-rec : Symbol FLANG ENV -> ENV)
;; extend an environment with a new binding that is the result of
;; evaluating an expression in the same environment as the extended
;; result
(define (extend-rec id expr rest-env)
```

```
  (let* ([new-cell (box (NumV 42))]
         [new-env  (Extend id new-cell rest-env)]
         [value    (eval expr new-env)])
    (set-box! new-cell value)
    new-env))
```

This `let*` form can be read almost as a C/Java-ish kind of code:

```
fun extend_rec(id, expr, rest_env) {
  new_cell  = new NumV(42);
  new_env   = Extend(id, new_cell, rest_env);
  value     = eval(expr, new_env);
  *new_cell = value;
  return new_env;
}
```

The code can be simpler if we fold the evaluation into the `set-box!` (since `value` is used just there), and if use `lookup` to do the mutation — since this way there is no need to hold onto the box. This is a bit more expensive, but since the binding is guaranteed to be the first one in the environment, the addition is just one quick step. The only binding that we need is the one for the new environment, which we can do as an internal definition, leaving us with:

```
(: extend-rec : Symbol FLANG ENV -> ENV)
(define (extend-rec id expr rest-env)
  (define new-env (Extend id (box (NumV 42)) rest-env))
  (set-box! (lookup id new-env) (eval expr new-env))
  new-env)
```

A complete rehacked version of FLANG with a `rec` binding follows. We can't test `rec` easily since we have no conditionals, but you can at least verify that

```
(run "{rec {f {fun {x} {call f x}}} {call f 0}}")
```

is an infinite loop.

```
#lang pl

(define-type FLANG
  [Num  Number]
  [Add  FLANG FLANG]
  [Sub  FLANG FLANG]
  [Mul  FLANG FLANG]
  [Div  FLANG FLANG]
  [Id   Symbol]
  [With Symbol FLANG FLANG]
  [WRec 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 (or 'with 'rec) more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [(list 'rec (list (symbol: name) named) body)
```

```
         (WRec name (parse-sexpr named) (parse-sexpr body))]
        [(cons x more)
         (error 'parse-sexpr "bad `~s' syntax in ~s" x 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 (Boxof VAL) ENV])

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

(: lookup : Symbol ENV -> (Boxof 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 boxed-val rest-env)
     (if (eq? id name) boxed-val (lookup name rest-env))]))

(: extend-rec : Symbol FLANG ENV -> ENV)
;; extend an environment with a new binding that is the result of
;; evaluating an expression in the same environment as the extended
;; result
(define (extend-rec id expr rest-env)
  (define new-env (Extend id (box (NumV 42)) rest-env))
  (set-box! (lookup id new-env) (eval expr new-env))
  new-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 (box (eval named-expr env)) env))]
    [(WRec bound-id named-expr bound-body)
     (eval bound-body
           (extend-rec bound-id named-expr env))]
    [(Id name) (unbox (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 (box (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)
```