Comp 411
Principles of Programming Languages
Lecture 13
The Semantics of Recursive Let

Corky Cartwright

February 22, 2021

The Semantics of Recursive Binding

- Let's add the recursive binding construct `letrec` (akin to `let`) to LC where we restrict right-hand sides to λ-expressions (which is the only useful case for unary `letrec`).
- The Scheme code for the AST is:

```
(define-struct rec-let (lhs   ; variable
                        rhs   ; a λ-expression
                        body))
```

where `lhs` is the new local variable, `rhs` is the lambda-expression defining the value of the new variable, and `body` is an expression thatcan use the new local variable. The new variable `lhs` is visible in both `rhs` and `body` in both `rhs` and `body`. The code for it in the interpreter might look like:

```
((rec-let? M) (MEval (rec-let-body M)
                     (extend env
                             (rec-let-lhs M)
                             (make-closure (rec-let-rhs M) <E>)))
```

- Problem: how should `<E>` expand into code? The environment should be the enclosing `(extend ...)` expression.

How Can We Construct This Circular Environment?

Let's treat environments abstractly.

We need to build an environment **E** such that

```
E = (extend env
            (rec-let-lhs M)
            (make-closure (rec-let-rhs M) E)))
```

What is wrong with the following Scheme code?

```
(define E (extend env
                  (rec-let-lhs M)
                  (make-closure (rec-let-rhs M) E)))
```

Can We Find a Representation That Works?

- Slogan: functions are the ultimate lazy data structures. But they are completely opaque; the only primitive operation on functions is application. (Recall the convention in Java regarding interfaces corresponding to lambda-notation.)
- Unfortunately, even the function representation of environments cannot salvage the preceding environment definition because a call-by-value language always evaluates the right-hand-side of **`define`** and the arguments of function calls. We need to tweak our code so that the circular reference to the new environment is embedded inside a **`lambda`**. The following revision of our **`eval`** clause works:

```
((rec-let? M) (MEval (rec-let-body M)
                     (rec-extend env (rec-let-lhs M) (rec-let-rhs M)))
```

where

```
(define rec-extend
  (lambda (env var rhs)
    (letrec [(new-env (lambda (v)
                        (if (equal? v var) (make-closure rhs new-env)
                          (env v))))])
      new-env)))
```

OO Representations for Environments

- If an environment is represented by an OO class or interface, it can include whatever methods are appropriate. Methods such as printing, equality testing (not an issue in our interpreters) and iteration (not currently an issue in our interpreters since mutation is forbidden) can easily be included. Moreover, deferred evaluation can be incorporated (if desired) in the interface. For example, a **`Binding`** interface might have both eager (call-by-value) and lazy (call-by-name) subclasses or a single concrete subclass with constructors corresponding to eager and lazy evaluation.

- On the other hand, poorly designed OO interfaces can be just as opaque as functions. Consider the standard command pattern interface which has only one method (command invocation).

Question to Ponder

- Can we eliminate λ if we include the right functional constants (combinators) in our language?

- Haskell Curry preferred combinators to explicit λ-notation. The former are algebraic while the latter is not. Schoenfinkel and (later) Curry independently discovered combinators before Church invented the λ–calculus.