Comp 311
Principles of Programming Languages
Lecture 13
The Semantics of Recursive Let

Corky Cartwright

Swarat Chaudhuri

September 21, 2011

The Semantics of Recursive Binding

- Let's add a recursive binding mechanism (akin to **`let`**) to LC where we restrict right-hand sides to lambda expressions.
- The Scheme code for the AST is:

```
(define-struct rec-let (lhs   ; variable
                        rhs   ; required to be a lambda-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 that can use the new local variable. The new variable **`lhs`** is visible 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 **`(extend ...)`** above.

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 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.

Unfortunately, even the function representation of environments cannot salvage the preceding environment definition because it 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)
     (local
       [(define new-env
          (lambda (v) (if (equal? v var) (make-closure rhs new-env) (env v))))]
          new-env)))
```

OO Representations for Environments

OO interfaces can be used to add whatever structure is appropriate. Hence, additional methods such as printing, equality testing (not an issue in our interpreters) and iteration (non currently an issue in our interpreters) can easily be included. Moreover, deferred evaluation can be hidden (if desired) by the interface. For example, a Binding interface might have eager (call-by-value) and lazy (call-by-name) implementing subclasses or even a single implementation class 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 lambda if we include the right functional constants (combinators) in our language?