Comp 411
Principles of Programming Languages
Lecture 12
The Semantics of Recursive Let

Corky Cartwright

September 21, 2012

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 Scala code for the letrec class is:

```
case class LetRec(lhs: Symbol, rhs: Exp, body: Exp) extends Exp
```

where **`lhs`** is the new local variable, **`rhs`** is the 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:

```
case LetRec(lhs, rhs, body) =>
  val rhsVal = eval(rhs, <E?>)
  val newEnv = env.bind(lhs, rhsVal)
  eval(body, newEnv)
```

Problem: how should **`<E?>`** expand into code? The environment should be **`newEnv`** above. In Scala, we cannot forward reference . Moreover, we need to defer the evaluation of the binding expression for **`rhsVal`** until after **`newEnv`** is bound.

How Can We Construct This Circular Environment?

Let's treat environments abstractly.

We need to build an environment `E` such that

```
E = env.bind(lhs, Closure(rhs, E)))
```

What is wrong with using the preceding equatin as Scala code?

```
val E = env.bind(lhs, Closure(rhs, E)))
```

What does `E` mean on the rhs of the Scala binding?

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 in a call-by-value language always evaluates the right-hand-side of bindings (Scala **`val`**) and the arguments of function calls. In a call-by-value language, we need to tweak our code so that the circular reference to the new environment is embedded inside a **`lambda`**. Assuming we use functions to represent environments, he following revision of our **`eval`** clause works:

```
case LetRec(lhs, rhs, body) =>
 val newEnv = recBind(env, lhs, rhs)
 eval(body, newEnv)
```

where (recall **`type Env = Symbol => Value`**)

```
def recBind(env: Env, newS: Symbol, rhs: AST) {
  val newEnv:Env = /* Scala val/var binding is single-variable letrec */
    (s: Symbol) =>
      if (s == newS) Closure((v:Value) => eval(rhs, newEnv)) else env(v)
  newEnv
}
```

OO Representations for Environments

OO traits/interfaces can be used to specify whatever structure is appropriate for an environment. Hence, additional methods such as printing, equality testing (not used in our interpreters) and iteration (non currently used in our interpreters) can easily be included. Moreover, deferred evaluation can be hidden (if desired) by the interface. For example, a BoundVal interface might have eager (call-by-value) and lazy (call-by-need/name) subclasses or even a single implementation class with constructors corresponding to eager and lazy evaluation.

On the other hand, OO interfaces can be just as opaque as functions. Consider the standard command pattern interface which has only one method (command invocation). The OO model enables the programmer to reveal the appropriate amount of detail.

Question to Ponder

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