Comp 411
Principles of Programming Languages
Lecture 8
Meta-interpreters II

Corky Cartwright

September 24, 2011

Representation Tricks

- We described the meaning of program λ-expressions as closures, *i.e.*, <code, env> pairs.
  - Are other representations possible/defensible? Yes, particularly in a functional language.
  - Closures can be represented as (Scheme or Scala) functions. Idea: wrap **(s: Symbol) =>** around code applying the pair closure in our meta-interpreter to the symbol **s**.
- What about environment representations?
  - A functional representation mapping symbols to values is elegant if not good software engineering.

Environments as Functions

- Mathematically elegant
- Questionable from software engineering perspective. Why? Functions are generally not finite and cannot be treated as tables. Environments, in contrast, are finite functions. One consequence of th fact that functions are infinite objects:,functions are opaque in output while structure closures are not.
- Not literally possible in languages like Java that support inner classes rather than closures. But there is a Java equivalent: return a class implementing an interface **Lambda<Sym,V>**, the *strategy/command* design pattern. Java formulation has essentially the same advantages and disadvantages as he Scheme formulation.
- Exercise: revise our previous correct meta-interpreters to use functions to represent environments. What changes?

Meta-interpreter with Environments as Functions

```
sealed trait Exp
sealed trait Value
case class Num(n: Int) extends Exp with Value
case class Var(s: Symbol) extends Exp
case class App(rator: Exp, rand: Exp) extends Exp
case class Lambda(s: Symbol, b: Exp) extends Exp with Value // s is a Symbol, not a Var
case class Sum(left: Exp, right: Exp) extends Exp
object Exp {
  type Env = Symbol => Value
  case class Closure(code: Exp, env: Env) extends Value
  def eval(e: Exp, env: Env): Value = {
    e match {
      case v:Num => v
      case Var(s) => env(s)
      case App(rator, rand) => apply(eval(rator, env), eval(rand, env))
      case l:Lambda => Closure(l, env)
      case Sum(left, right) => Num(toInt(eval(left, env).asInstanceOf[Num]) +
        toInt(eval(right,env).asInstanceOf[Num]))
    }
  }
  def toInt(n: Num):Int = n match { case Num(i) => i  }
  def apply(fun: Value, arg: Value) = {
    fun match {
      case Closure(Lambda(s, b), env) => eval(b, bind(s, arg, env))
      case _ => throw new IllegalArgumentException("Attempted to apply non-function " +
        fun + " in an application")
    }
  }
  def bind(s: Symbol, v: Value, env: Env): Env = (s1: Symbol) => if (s == s1) v else env(s1)
```

This code does not encapsulate the representation of closures. How would the code change if we encapsulated it?

Closures as Functions

- Mathematically elegant
- Questionable from software engineering perspective. Why? Functions are opaque. Their internal form cannot be examined. (Why?) Closures as structures, in contrast, are open to inspection. Since functions are built-in, we cannot insert debugging code (print statements).
- The implementation is directly supported by Scala and Scheme and indirectly supported by Java (an anonymous inner class implementing an interface **Lambda<V,V>** is conceptually a function (modulo crufty syntax).

Meta-interpreter Using Functions for Closures and Environments

```
sealed trait Exp
sealed trait Value
case class Num(n: Int) extends Exp with Value
case class Var(s: Symbol) extends Exp
case class App(rator: Exp, rand: Exp) extends Exp
case class Lambda(s: Symbol, b: Exp) extends Exp with Value // s is a Symbol, not a Var
case class Sum(left: Exp, right: Exp) extends Exp
object Exp {
  type Env = Symbol => Value
  case class Closure(fn: Value => Value) extends Value
  def eval(e: Exp, env: Env): Value = {
    e match {
      case v:Num => v
      case Var(s) => env(s)
      case App(rator, rand) => apply(eval(rator, env), eval(rand, env))
      case Lambda(s, b) => Closure((v:Value) => eval(b, bind(s,v, env))
      case Sum(left, right) => Num(toInt(eval(left, env).asInstanceOf[Num]) +
        toInt(eval(right,env).asInstanceOf[Num]))
    }
  }
  def toInt(n: Num):Int = n match { case Num(i) => i  }
  def apply(fn: Value, arg: Value) = {
    fun match {
      case Closure(fn) => fn(arg)
      case _ => throw new IllegalArgumentException("Attempted to apply non-function " +
        fn + " in an application")
    }
  }
  def bind(s: Symbol, v: Value, env: Env): Env = (s1: Symbol) => if (s == s1) v else env(s1)
```

Important Variations on Our Meta-interpreter

- *Call-by-name* (CBN) beta-reduction. Recall that in our syntactic intepreter for LC that we chose to restrict beta-reduction to *values*. In practice, this restriction is very important in languages with *mutable* data. But LC does not (yet) support *mutation*.
- *Call-by-need* evaluation of arguments. There is no syntactic equivalent since this evaluation policy is a meta-interpreter based optimization of *Call-by-name*. In the presence of mutation, *call-by-need* is not equivalent to *call-by-name*.

Call-by-name Discussion

- In *Call-by-name* syntactic interpretation, no argument is evaluated until its value is demanded by a primitive operation (only **+** in LC). If a parameter is never evaluated, the corresponding argument is never evaluated.
- Disadvantage: if a parameter is evaluated multiple times, so is the corresponding argument!
- Thought exercise: how can we defer the evaluation of an argument expression (Hint: think about treating argument values as closures of no arguments)?