Comp 411
Principles of Programming Languages
Lecture 7
Meta-interpreters

Corky Cartwright

September 14, 2011

Denotational Semantics

- The primary alternative to *syntactic* semantics is *denotational* semantics. A denotational semantics maps abstract syntax trees to a set of *denotations* (mathematical values like numbers, lists, and functions).
- Simple denotations like numbers and lists are essentially the same mathematical objects as syntactic values: they have simple inductive definitions with exactly the same structure as the corresponding abstract syntax trees.
- But denotations can also be complex mathematical objects like *functions* or *sets*. For example, the denotation for a λ-expression in "pure" (functional) Scheme is a function mapping denotations to denotations--*not* some syntax tree as in a syntactic semantics.

Meta-interpreters

- Denotational semantics is rooted in mathematical logic: the semantics of terms (expressions) in the predicate calculus is defined denotationally by *recursion* on the syntactic structure of terms. The meaning of each term is a value in an mathematical *structure* (as used in first-order logic).
- In the realm of programming languages, purely functional interpreters (defined by pure recursion on the structure of ASTs) constitute a restricted form of denotational definition.
  - The defect is that the output of an actual interpreter is restricted to values that can be characterized syntactically. (How do you output a function?)
  - On the other hand, interpreters naturally introduce a simple form of functional abstraction. A recursive interpreter accepts an extra input, an environment mapping free variables to values, thus defining the meaning of a program expression as a function from environments to values.
  - Syntactic interpreters are *not denotational* because they transform ASTs. A denotational interpreter uses pure structural recursion. To handle the bindings to variables, it cannot perform substitutions; it must maintain an environment of bindings instead.

Meta-interpreters cont.

- Interpreters written in a denotational style are often called *meta*-interpreters because they are defined in a meta-mathematical framework where programming language expressions and denotations are objects in the framework. The definition of the interpreter is a level above definitions of functions in the language being defined.
- In mathematical logic, meta-level definitions are expressed informally as definitions of mathematical functions.
- In program semantics, meta-level definitions are expressed in a convenient functional framework with a semantics that is easily defined and understood using informal mathematics. *Formal* denotational definitions are written in a mathematical meta-language corresponding to some formulation of a *Universal Domain* (a mathematical domain in which all relevant programming language domains can be simply embedded, usually as projections). This material is subject of a graduate level course on domain theory.
- A functional interpreter for language L written in a functional subset of L is called a *meta-circular* interpreter. It really isn't circular because it reduces the meaning of all programs to a single purely functional program which can be understood independently using simple mathematical machinery (inductive definitions over familiar mathematical domains).

Denotational Building Blocks

- Inductively defined ASTs for program syntax. We have thoroughly discussed this topic.
- What about denotations? For now, we will only use simple inductively defined values (without functional abstraction) like numbers, lists, tuples, etc.
- What about environments? Mathematicians like to use functions. An environment is a function from variables to denotations. But environment functions are special because they are *finite*. Software engineers prefer to represent them as lists of pairs binding variables to denotations.
- In “higher-order” languages, functions are data objects. How do we represent them? For now we will use ASTs possibly supplemented by simple denotations (as described above).

Critique of Deferred Substitution Interpreter from Lecture 6

- How did we represent the denotations of λ-expressions (functions) in environments? By their ASTs. Is this implementation correct? No!
- Counterexample: twice

```
(let
  ([twice (λ (f) (λ (x) (f (f x))))]
   [x 5])
  ((twice (λ (y) (+ x y))) 0))
```

Evaluate (syntactically)

```
(let
  [(twice (λ (f) (λ (x) (f (f x)))))
   (x 5)]
  ((twice (λ (y) (+ x y))) 0))

=>
(((λ (f) (λ (x) (f (f x))))
   (λ (y) (+ 5 y))) 0)

=>
((λ (x)
   ((λ (y) (+ 5 y)) ((λ (y) (+ 5 y)) x)))
   0)

=>
((λ (y) (+ 5 y)) ((λ (y) (+ 5 y)) 0)) =>
((λ (y) (+ 5 y)) (+ 5 0)) =>
((λ (y) (+ 5 y)) 5) => (+ 5  5) => 10
```

Closures Are Essential

- **Exercise**: evaluate the same expression using our broken interpreter.
- The computed "answer" is 0!
- The interpreter uses the wrong binding for the free variable `x` in `(λ (y) (+ x y))`.
- The environment records deferred substitutions. When we pass a function as an argument, we need to pass a "package" including the deferred substitutions. Why? The function will be applied in a *different* environment which may associate the *wrong* bindings with free variables. In the PL (programming languages) literature, these packages (code representation, environment) are called *closures*.
- Note the similarity between this mistake and the "capture of bound variables".
- Unfortunately, this mistake has been labeled as a feature rather than a bug in much of the PL (programming languages) literature. It is called "dynamic scoping" rather than a horrendous mistake.

Correct Semantic Interpretation

```
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 = Map[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 = env + (s -> v)
}
```