Comp 411
Principles of Programming Languages
Lecture 6
Implementing Syntactic Interpreters

Corky Cartwright
September 10-12, 2011
Comments on Syntactic Interpreter

• The key property of this evaluator is that it only manipulates (abstract) syntax. It specifies the meaning of LC by mechanically transforming the syntactic representation of a program.

• This approach only assigns a satisfactory meaning to complete LC programs, not to subtrees of complete programs.

Counterexample: \( \lambda x . 7 + y \)

\[
\text{App(Lambda('x, Sum(Var('x), Var('y))), Num(7))}
\]

In a context where \( y \) is bound to \( 5 \), it returns \( 12 \); not \( (+ 7 y) \) or a run-time error. The meaning of sub-expressions should be defined so that meaning \( \llbracket \cdot \rrbracket \) is compositional, \( i.e., \)

\[
\llbracket (c \ M_1 \ldots \ M_k) \rrbracket = \llbracket c \rrbracket \square(\llbracket M_1 \rrbracket, \ldots, \llbracket M_k \rrbracket)
\]

• Syntactic interpretation utterly fails in this regard.
Toward Semantic Interpretation I

From a software engineering perspective, what is wrong with our syntactic interpreter?

- How fast is subst? How can we do better?
- Avoid unnecessary substitutions by keeping a table of bindings.

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
case class Sum(left: Exp, right: Exp) extends Exp

object Exp {
  type Env = Map[Symbol, Value]
  def eval(e: Exp, env: Env): Value = {
    e match {
      case v:Num => v
      case Var(s) => env(s) // env.get(s) returns Option[Value]
      case App(rator, rand) => apply(eval(rator, env), eval(rand, env), env)
      case l:Lambda => l
      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, env: Env) = {
    fn match {
        case Lambda(s, b) => eval(b, bind(s, arg, env))
        case _ => throw new IllegalArgumentException("Attempted to apply non-function " + fn + " in an application")
    }
}

def bind(s: Symbol, v: Value, env: Env): Env = env + (s -> v)
Gotcha's in Semantic Interpretation

What if the argument \texttt{fn} passed to \texttt{apply} contains free variables? Do we always get the right answer (as defined by syntactic interpretation)?

- Illustration in Jam (can be translated into LC)
  
  \[
  \begin{aligned}
  \text{let } a := 5; \\
  \text{in let } \text{lazy-a} := \text{map } x \text{ to } a; \\
  \text{In let } a := 10 \\
  \text{in } \text{lazy-a}(0)
  \end{aligned}
  \]

- What goes \texttt{wrong}? (Hint: what is the value of \texttt{a} when \texttt{lazy-a} is evaluated in the body of the inner \texttt{let}. Note that the preceding Jam program is equivalent to the following LC program:
  
  \[
  \begin{aligned}
  (\lambda a . (\lambda \text{lazy-a} . \\
  (\lambda a . \text{lazy-a } 0) 10) \\
  (\lambda x . a)
  \\
  5)
  \end{aligned}
  \]

- Think about how you might fix the problem. Our first attempt at writing a semantic interpreter is BROKEN.
object aCell {

    val a = 5

    def lazy_a(f: Int => Int):Int = a

    def test():Int {
        val a = 10
        lazy_a((x: Int) => x)   // "(x: Int)" not "x:Int"
    }
}

What does \texttt{a} mean inside the definition of \texttt{lazy_a}?
Scheme Binding (Scoping) Constructs

• In Scheme, 
  \((let \[(v1 M1) \ldots (vn Mn)] N)\)
  abbreviates
  \(((\lambda (v1 \ldots vn) N) M1 \ldots Mn)\)

• Similarly, 
  \((let* \[(v1 M1) \ldots (vn Mn)] N)\)
  abbreviates
  \((let \[(v1 M1)] (let \ldots (let \[(vn Mn)] N) \ldots ))\)

• And
  \((letrec \[(v1 M1) \ldots (vn Mn)] N)\)
  means \(v1 \ldots vn\) are bound recursively, \(i.e.,\) \(v1 \ldots vn\) are in scope in
  \(M1 \ldots Mn\) as well as in \(N\).