Comp 411
Principles of Programming Languages
Lecture 7
Meta-interpreters

Corky Cartwright
January 31, 2014
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 lambda-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 a 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 lambda-expressions (functions) in environments? By their ASTs. Is this implementation correct? No!

• Counterexample:

(let ([twice (lambda (f) (lambda (x) (f (f x))))])))
(let ([x 5])
 (twice (lambda (y) (+ x y))) 0)))
Evaluate (syntactically)

\[
(\text{let }[(\text{twice } (\lambda (f) (\lambda (x) (f (f x))))))]
\]

(\text{let }[(x 5)]

(\text{twice } (\lambda (y) (+ x y))) 0)

\Rightarrow

(\text{let }[(x 5)]

((\lambda (f) (\lambda (x) (f (f x))))

(\lambda (y) (+ x y)))

0))

\Rightarrow

((\lambda (f) (\lambda (x) (f (f x))))

(\lambda (y) (+ 5 y)))

0)

\Rightarrow

((\lambda (x) ((\lambda (y) (+ 5 y)) ((\lambda (y) (+ 5 y)) x))))

0) \Rightarrow

((\lambda (y) (+ 5 y)) ((\lambda (y) (+ 5 y)) 0)) \Rightarrow

((\lambda (y) (+ 5 y)) (+ 5 0)) \Rightarrow

((\lambda (y) (+ 5 y)) 5) \Rightarrow (+ 5 5) \Rightarrow 10
Evaluate (using our interpreter)

(let [[(twice (lambda (f) (lambda (x) (f (f x)))))]
    (let (x 5)]
        (twice (lambda (y) (+ x y))) 0)) ⇒

{ twice = (lambda (f) (lambda (x) (f (f x)))) } (let [(x 5)] ((twice (lambda (y) (+ x y))) 0)) ⇒

{ x = 5, twice = (lambda (f) (lambda (x) (f (f x)))) } ((twice (lambda (y) (+ x y))) 0) ⇒

{ x = 5, ... }

(((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ x y))) 0) ⇒

{ f = (lambda (y) (+ x y)), x = 5, ... } ((lambda (x) (f (f x))) 0) ⇒

{ x = 0, f = (lambda (y) (+ x y)), ... } (f (f x)) ⇒

{ x = 0, f = (lambda (y) (+ x y)), ... } ((lambda (y) (+ x y)) (f x)) ⇒

{ x = 0, ... } ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) x)) ⇒

{ x = 0, ... } ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) 0)) ⇒

{ y = 0, x = 0, ... } ((lambda (y) (+ x y)) (+ x y)) ⇒

{ y = 0, x = 0, ... } ((lambda (y) (+ x y)) (+ 0 y)) ⇒

{ y = 0, x = 0, ... } ((lambda (y) (+ x y)) (+ 0 0)) ⇒

{ y = 0, x = 0, ... } ((lambda (y) (+ x y)) 0) ⇒

{ y = 0, y = 0, x = 0, ... } (+ x y) ⇒ { y = 0, ... } (+ 0 y) ⇒

{ ... } (+ 0 0) ⇒ 0
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 \((\text{lambda}(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 it 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 literature. It is called “dynamic scoping” rather than a horrendous mistake. Watch out whenever you must program in a language with “dynamic scoping”.
Correct Semantic Interpretation

(define-struct (closure proc env))
;; V = Const | Closure ; revises our former definition of V
;; Binding = (make-Binding Sym V) ; Note: Sym not Var
;; Env = (listOf Binding)
;; Closure = (make-closure Proc Env)
;; R Env → V
(define eval
  (lambda (M env)
    (cond
     ((var? M) (lookup (var-name M) env))
     ((const? M) M)
     ((proc? M)) (make-closure M env))
     ((add? M) ; M has form (+ l r)
      (add (eval (add-left M) env) (eval (add-right M) env)))
     (else ; M has form (N1 N2)
      (apply (eval (app-rator M) env) (eval (app-rand M) env))))))

(define apply
  (lambda (cl v) ; assume cl is a closure
    (eval (proc-body (closure-proc cl))
          (cons (make-binding (proc-param (closure-proc cl)) v)
                (closure-env cl))))