Comp 411
Principles of Programming Languages
Lecture 7
Meta-interpreters

Corky Cartwright

January 23, 2023

Denotational Semantics

- The primary alternative to *syntactic* semantics is *denotational* semantics. A denotational semantics maps abstract syntax trees into a set of *denotations* (mathematical values like numbers, lists, and functions).
- The denotations of simple data values 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-abstraction 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* or algebra.
- In the realm of programming languages, a purely functional interpreter (defined by recursion on the structure of ASTs) implicitly constitutes a denotational definition of the language.
  - Syntactic interpreters do not have this property. The defect is that the output of a syntactic interpreter is restricted to values that can be characterized syntactically. (How do you output a function?)
  - On the other hand, efficient interpreters implicitly introduce a simple form of functional abstraction. An efficient 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 to ASTs that are values rather than mapping them to abstract mathematical meanings. 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. We can concretely define denotational meaning using purely functional meta-interpreters.

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, informal 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, most elegantly as projections). This material is covered in graduate level courses 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

- A program is an inductively defined AST. We have thoroughly discussed this topic. In most functional languages, a program is simply a *closed* expression, where an expression has a simple inductive definition as an AST.
- What about denotations? Our meta-interpreter will return ordinary values like numbers, lists, tuples, etc., but each such value depends on the environment, so the meaning of a sub-expression is really a function from environments to values.
- How do we formalize the set of environments? Mathematicians like to use functions. An environment is a function from variables (syntax) 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. How do we represent them? For now we will use (partial) functions from environments to values (since they require no further specification).

Critique of Deferred Substitution Interpreter from Lecture 6

- How did we represent the denotations of `lambda`-abstractions (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)

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

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

⇒ (((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ 5 y)))
     0)

⇒ ((lambda (x) ((lambda (y) (+ 5 y)) ((lambda (y) (+ 5 y)) x))
     0)

⇒ ((lambda (y) (+ 5 y)) ((lambda (y) (+ 5 y)) 0))
⇒ ((lambda (y) (+ 5 y)) (+ 5 0))
⇒ ((lambda (y) (+ 5 y)) 5) ⇒ (+ 5 5) ⇒ 10
```

Evaluate (using our bad 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 trace appears on the preceding slide!
- The interpreter uses the wrong binding for the free variable `x` in
  `(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".

Corrected Semantic Interpreter

```
(define-struct (closure proc env))      ; closure is name of type
;; V = Const | Closure                ; revises our former definition of V
;; Binding = (make-Binding Sym V)      ; Note: Sym not Var
;; Env = (listOf Binding)              ; Lists are built-in to Scheme
;; Closure = (make-closure Proc Env)
;; Type contract: 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)
         (const-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))))))

;; Closure representing V → V assumed to be <code,environment> pair
(define apply
  (lambda (cl v)                        ; assume cl is a closure
    (eval (proc-body (closure-proc cl))  ; closure-proc may blow-up
          (cons (make-binding (proc-param (closure-proc cl)) v)
                (closure-env cl)))
```

A Meta-Interpreter for CBN

- **Recall** the syntactic semantics for the CBN version of LC. What is different from our standard CBV (call-by-value) syntactic semantics? What is our rule in CBV for reducing applications of program-defined functions (lambda-abstractions) **`(lambda` *`x M`*`)`**? Are there any restrictions on β-reduction?
- How do we implement CBN (unrestricted) β-reduction in a meta-interpreter. Recall that we must defer substitutions for parameters in the lambda-abstractions. How can we get the right answer even when we defer evaluation? What did we do in our CBV interpreter when we passed functions (lambda-abstractions) as argument values?
- What problem do closures eliminate? Finding the correct values for free variables in the bundled lambda-abstraction. In CBN we need to bind variables to unevaluated expressions. How can we avoid getting incorrect values for free variables in such expressions, just like we did for lambda-abstractions as values? You must answer this question to do Project 2. Hint: free variables in CBN and free variables in function definitions can be addressed in essentially the same way.