Comp 411

Principles of Programming Languages

Lecture 6

Implementing Syntactic Interpreters

Corky Cartwright

January 23, 2022

A Syntactic Evaluator

Can we translate our syntactic reduction rules into a program?

```
;; AST → V ⊆ AST        ; an illegal program can return a non-value in AST\V
(define eval
  (lambda (M)    ; M is an AST
    (cond                ; case split on form of M
      ((var? M)  M)                    ; M is a free var (stuck!)
      ((or (const? M) (proc? M)) M)    ; M is a value
      ((add? M)                        ; M has form (+ l r)
        (const-add (eval (add-left M)) (eval (add-right M))))
      (else                            ; M has form (N1 N2)
        (apply (eval (app-rator M)) (eval (app-rand M)))))))

;; A⇒B A → B
(define apply (lambda (a-proc a-value)
  (cond
    ((not (proc? A-proc))          ; ill-formed app
      (make-app a-proc a-value))   ; return stuck state [ERROR!]
    (else                          ; return reduced, substituted body
      (eval
        (subst a-value (proc-param a-proc)(proc-body a-proc)))))))
```

Coding Substitution

```
;; V Sym R → R   Blindly substitutes v for x in M (ignoring capture)
(define subst
  (lambda (v x M)
    (cond
      [(var? M) (cond [(equal? (var-name M) x) v] [else M])]
      [(const? M) M]
      [(proc? M))
        (cond [(equal? x (proc-param M)) M]
              [else (make-proc (proc-param M) (subst v x (proc-body M)))])]
      [(add? M) (make-add (subst v x (add-left M))
                          (subst v x (add-right M)))]
      [else    ;; M is (N1 N2)
        (make-app (subst v x (app-rator M))
                  (subst v x (app-rand M)))])))
```

Is **`subst`** safe? No! It is oblivious to free variables in **`M`**. Does it work in the context of our syntactic interpreter? Provisionally: it fails in some cases for illegal programs unless a distinction is made between top level *constants* and variables. All top level bindings (**`define`** in Scheme) must be classified as *constants*. So the set of *constants* is program dependent. In the absence of such a distinction, some programs with free variables fail to generate run-time errors when free variables are evaluated because they have been captured

Exercise: Revise **`subst`** so that it is safe. Note that blind substitution is safe as long as our top-level expression/program **`M`** is well-formed and contains no free variables. Why?

Comments on our Syntactic Interpreter

We still need to define `const-add`. What does `const-add` do on non-`const` values? 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. Counter-example:

```
((lambda (x) (+ x y)) 7)
```

If `const-add` mirrored syntactic evaluation, then it would return the abstract syntax tree for `(+ 7 y)` which is an irreducible "stuck" state—not a value—and the correct choice if we are strictly implementing syntactic evaluation. A more attractive alternative that is an elaboration of syntactic interpretation is to generate a run-time error because `y` is not a value. In a context where `y` is bound to (the abstract syntax tree for) `5`, it returns (the abstract syntax tree for) `12`; which is not (the abstract syntax tree for) `(+ 7 y)` or a run-time error. From a mathematical perspective, The meaning of sub-expressions should be defined so that meaning ⟦...⟧ is compositional, *i.e.*

$$⟦ \texttt{(c } \texttt{M}_1 \ldots \texttt{M}_k \texttt{)} ⟧ = ⟦ \texttt{c} ⟧ (⟦ \texttt{M}_1 ⟧, \ldots , ⟦ \texttt{M}_k ⟧)$$

Syntactic interpretation utterly fails in this regard because it cannot cope with free variables.

Can We Make Syntactic Evaluation Compositional?

The short answer is “no”.

Since syntactic evaluation does not directly assign meaning to components of abstract syntax trees, it technically does not satisfy the compositionality criterion. If we apply the definition syntactic evaluation to the subtrees of a program, then we must address the fact that the syntactic evaluation of many subtrees will “stick”. Some of these stuck results correspond to actual run-time errors in the syntactic evaluation of the entire program. Hence, the compositional meaning (based on syntactic evaluation) of a program must be error if any subtree means error. But this definition of meaning is clearly wrong and inconsistent with the results of syntactic evaluation since every program containing a variable reference will be assigned some form of error as its meaning.

To assign a compositional meaning to abstract syntax trees, we need a more sophisticated definition of meaning for program expressions than syntactic evaluation. We will address this issue in the next lecture.

Toward Semantic Interpretation

From a software engineering perspective, what is wrong with our syntactic interpreter? How fast is **`subst`**? How can we do better? We can avoid unnecessary and costly substitution operations by keeping a table of bindings, which we will call an environment. This same "refactoring" also corresponds to a compositional semantics.

```
;; Binding = (make-Binding Sym V)   ; Note: Sym not Var [coding detail]
;; Env = (listOf Binding)
;; R Env → V
(define eval
 (lambda (M env)
   (cond
     ((var? M) (lookup (var-name M) env))
     ((or (const? M)(proc? M)) M)    ;; Do procs really evaluate to themselves
     ((add? M) ; M has form '(+ l r)' in LC syntax
       (const-add (eval (add-left M) env) (eval (add-right M) env)))
     (else      ; M has form '(N1 N2)' in LC syntax
       (apply (eval (app-rator M) env) (eval (app-rand M) env) env)))))

;; Proc V Env → V
(define apply
  (lambda (a-proc a-value env)
    (eval (proc-body a-proc) (cons ((proc-param a-proc) a-value) env)))
```

Aside: More Readable Notation for Lambda Expressions (which is used on next slide)

- In essentially all functional programming languages, there is alternate special notation for

  ```
  ((lambda x M) N)
  ```

  namely

  ```
  (let [(x N)] M)
  ```  Scheme

  or

  ```
  let x := N; in M
  ```  Jam

- This alternate notation is literally an abbreviation for the explicit `lambda` form

- For this alternate notation, the beta-reduction rule has the form

  ```
  (let [(x V)] M) ⇒ M[x := V]
  ```  Call-by-value (mainstream languages)

  ```
  (let [(x N)] M) ⇒ M[x := N]
  ```  Call-by-name (Haskell only)

# Gotcha's in Naive Semantic Interpretation

- What if `a-proc` in our putative semantic interpreter contains free variables (which can happen in legal programs)? Do we always get the right answer (as defined by syntactic interpretation)? Illustration:

- ```
  (let [(a 5)]
    (let [(app-to-a (lambda (f) (f a)))]
      (let [(a 10)]
        (+ a (app-to-a (lambda (x) x))))))
  ```

- What goes **wrong** ? Should a `lambda`-expression really evaluate to itself? **This is the most serious and most common blunder in writing interpreters.**

- Think about how you might fix the problem. Hint: what information is missing in `env` when `a-proc` is evaluated? Remember, you want the same result as if you were performing syntactic interpretation.


  ```