Functional Programming

Substitution Model, Higher-Order Programming

Luis Diogo Couto

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

25 April 2013

Substitution Model

Higher-Order Programming

Substitution Model

Higher-Order Programming

The Substitution Model

- Already mentioned
- One of two models for evaluating expressions in Scheme
- Suitable for a very large portion of Scheme

How it works

To evaluate an expression E, continuously apply rewrite rules starting with E. Each rule substitutes an expression with a new one. This process continues until no rule is applicable.

The Substitution Model

- Rewrite rules only
  - values for identifiers
  - calculate expressions
  - ...
- Symbolic computation
- No separate data structures, no side effects

Substitution: Lambda

- λ-calculus

$$
\begin{aligned}
& (\lambda x \cdot (x+1))2 \\
\longrightarrow\ & (\lambda x \cdot (x+1))[2/x] \\
\longrightarrow\ & 2+1 \\
\longrightarrow\ & 3
\end{aligned}
$$

- Scheme

```
  ((lambda (x) (+ x 1)) 2)
→ (+ x 1) : [2/x]
→ (+ 2 1)
→ 3
```

Substitution: Let

syntax

```
(let ((⟨variable₁⟩ ⟨expression₁⟩) ...)
  ⟨body⟩)
```

substitution

```
((lambda ( ⟨variable₁⟩ ...)
    ⟨body⟩)
 ⟨expression₁⟩ ...)
```

- let is not a primitive. It can be defined in terms of `lambda`

Example: let

```
(let ((x 1)
(y 2)
(z 3))
(< x y z))
```

$\longrightarrow$ `((lambda (x y z) (< x y z)) 1 2 3)`

$\longrightarrow$ `(< x y z)` : $[1, 2, 3/x, y, z]$

$\longrightarrow$ `(< 1 2 3)`

$\longrightarrow$ `#t`

Substitution: Let*

syntax

```
(let* ((⟨variable₁⟩ ⟨expression₁⟩) ...)
  ⟨body⟩)
```

substitution

```
(let (( ⟨variable₁⟩ ⟨expression₁⟩))
  (let* (( ⟨variable₂⟩ ⟨expression₂⟩) ...)
    ⟨body⟩))
```

- `let*` is rewritten as `let`

Example: let*

```
   (let* ((x 1) (y (+ x x)))
     (+ x y))
→ (let ((x 1))
     (let* ((y (+ x x)))
       (+ x y))
→ ((lambda (x)
      (let* ((y (+ x x))) (+ x y)))
    1)
→ (let* ((y (+ x x))) (+ x y)) : [1/x]
→ (let* ((y (+ 1 1))) (+ 1 y))
→ ((lambda (y) (+ 1 y)) (+ 1 1))       let* + let rewrites
→ ((lambda (y) (+ 1 y)) 2)
→ ((lambda (y) (+ 1 y)) 2) : [2/y]
→ (+ 1 2)
→ 3
```

The letrec expression

syntax

(letrec (($\langle$variable$_1\rangle$ $\langle$expression$_1\rangle$) ...)
  $\langle$body$\rangle$)

The letrec expression

syntax

(letrec (($\langle$variable$_1\rangle$ $\langle$expression$_1\rangle$) ...)
  $\langle$body$\rangle$)

- Used for recursive definitions
- Every $\langle$expression$_i\rangle$ may use every $\langle$variable$_i\rangle$
- But every expression must be evaluated without use of the values of the $\langle$variable$_i\rangle$
  - Usually done by having all references to $\langle$variable$_i$ $\rangle$in $\langle$expression$_i\rangle$ inside lambdas.
- Can be defined in substitution model (but we will not go into it)

Factorials Again

Traditional Recursion

```
  (fact 4)
→ ((lambda (n) (if (= n 1))
    1 (* n (fact (- n 1))))) 4)  : [4/n]
→ (if (= 4 1) 1 (* n (fact (- 4 1)))) 4)
→ (* 4 (fact (- 4 1)))
→ (* 4 (fact 3))
→ (* 4 (* 3 (fact 2)))
→ (* 4 (* 3 (* 2 (fact 1))))
→ (* 4 (* 3 (* 2 1)))
→ 24
```

Factorials Again

Tail Recursion

```
  (fact-tl 4)
→ (fact-tl 4 1)
→ (if (= n 1) r (fact-tl (- n 1) (* r n))) : [4, 1/n, r]
→ (if (= 4 1) r (fact-tl (- 4 1) (* 1 4)))
→ (fact-tl (- 4 1) (* 1 4))
→ (fact-tl 3 4)
→ (fact-tl 2 12)
→ 24
```

Limitations of Substitution Model

- Does not cover entire language
- Some substitutions (`letrec`) hard to comprehend
- Some constructs are problematic: `define`, `set!`
- IO also cannot be expressed
- SICP section 1.1.5

Substitution Model

Higher-Order Programming

Higher-Order Programming

- Treat procedures as any other data type
  - Create when needed
  - Use as parameters
  - Return as values
  - Manipulate procedures
- A defining characteristic of functional programming
- More general approach
- Enable reuse
- Promote abstraction

Advantages

- Abstraction is a core skill of successful Software Engineers
- Reason about code (and algorithms) at a higher level
- Once understood, very powerful and expressive
- Not just theoretical
  - Available in Python, Ruby, Scala...
- “Write beautiful, elegant code.”

Examples

- Already saw `map`
- Another classical example: `fold`
  - "folds" a list with an operator
  - `(fold + 0 '(1 2 3))`$\rightarrow$`...`$\rightarrow$6
  - `(define sum (lambda (l) (fold + 0 l)))`
  - Implement it!