1 Semantics engineering

1.1 Theory of programming languages

Calculus is a logic for calculating with the terms of the language. For example,

e = x | λ x.e | (e e)

Extensions with primitive data:

e = x | λ x.e | (e e) | tt | ff | (if e e e)

External interpretation functions (δ):

(if tt e e') if-tt e

(if ff e e') if-ff e'

Semantics is a system for determining the value of a program.

Reduction is a relation on terms:

((λ x.e) e') beta e[x = x'] (e with x replaced by e')

((λ x.e) e') beta [e'/x]e (substitution e' for x in e)

Equational system is defined with three properties:

For any relation R,

reflexivity $\frac{\text{e R e'}}{\text{e e'' R e' e''}}$ *symmetry* $\frac{\text{e R e'}}{\text{e'' e R e'' e'}}$ *transitivity* $\frac{\text{e R e'}}{\lambda \text{ x.e R } \lambda \text{ x.e'}}$

With an equational system, we can prove such facts as

e (Y e) = (Y e)

meaning every single term has a *fixpoint*.

*

* *

In Plotkin's theory of programming languages, a semantic is a relation *eval* from programs to values:

eval : Program × Value

def e eval v iff e = v

We get a *specification* of an interpreter after proving that eval is a function.

eval : Program → Value

eval(e) = v

Prove that the calculus satisfies a standard reduction property. This gives us a second semantic.

eval-standard : Program → Value

def eval-standard(e) = v iff e standard reduces to v

Curry-Feys's *standard reduction* is a strategy for the lambda calculus, that is, a function that picks the next reducible expression (called *redex*) to reduce. Plotkin specifically uses the leftmost-outermost strategy.

Plotkin adds the *truth* to the specification.

def e ∼ e' iff placing e and e' into any context yields programs that
produce the same observable behavior according to eval

1.2 Syntax

```
; syntax trees
(define-language Lambda
  (e ::= x
         (lambda (x ...) e)
         (e e ...))
  (x ::= variable-not-otherwise-mentioned))

; instances
(define e1 (term y))
(define e2 (term (lambda (y) y)))
(define e3 (term (lambda (x y) y)))
```

```
(define e4 (term (,e2 ,e3)))

; a predicate that tests membership
(define lambda? (redex-match? Lambda e))

; language tests formulations
(test-equal (lambda? e1) #true)
(test-equal (lambda? e2) #true)
(test-equal (lambda? e3) #true)
(test-equal (lambda? e4) #true)

(define eb1 (term (lambda (x x) y)))
(define eb2 (term (lambda (x y) 3)))

(test-equal (lambda? eb1) #true)
(test-equal (lambda? eb2) #false)
```

1.3 Metafunction

A metafunction is a function on terms of a specific language.

```
; are the identifiers in the given sequence unique?
; extended Kleene patterns: (lambda (x_!_ ...) e)

(module+ test
  (test-equal (term (unique-vars x y)) #true)
  (test-equal (term (unique-vars x y x)) #false))

(define-metafunction Lambda
  ; a Redex contract with patterns
  unique-vars : x ... -> boolean
  [(unique-vars) #true]
  [(unique-vars x x_1 ... x x_2 ...) #false]
  [(unique-vars x x_1 ...) (unique-vars x_1 ...)])

(module+ test
  (test-results))

; (subtract (x ...) x_1 ...) removes x_1 ... from (x ...)

(module+ test
  (test-equal (term (subtract (x y z x) x z)) (term (y))))

(define-metafunction Lambda
  subtract : (x ...) x ... -> (x ...)
  [(subtract (x ...)) (x ...)]
  [(subtract (x ...) x_1 x_2 ...)
   (subtract (subtract1 (x ...) x_1) x_2 ...)])

(module+ test
  (test-results))

; (subtract1 (x ...) x_1) removes x_1  from (x ...)

(module+ test
  (test-equal (term (subtract1 (x y z x) x)) (term (y z))))

(define-metafunction Lambda
  subtract1 : (x ...) x -> (x ...)
  [(subtract1 (x_1 ... x x_2 ...) x)
   (x_1 ... x_2new ...)
   (where (x_2new ...) (subtract1 (x_2 ...) x))
   (where #false (in x (x_1 ...)))]
```

```
  [(subtract1 (x ...) x_1) (x ...)])

(define-metafunction Lambda
  in : x (x ...) -> boolean
  [(in x (x_1 ... x x_2 ...)) #true]
  [(in x (x_1 ...)) #false])

(module+ test
  (test-results))
```

1.4 Scope

To specify the scope, a free-variables function specifies which language constructs bind and which one don't.

```
; (fv e) computes the sequence of free variables of e
; a variable occurrence of x is free in e
; if no (lambda (... x ...) ...) dominates its occurrence

(module+ test
  (test-equal (term (fv x)) (term (x)))
  (test-equal (term (fv (lambda (x) x))) (term ()))
  (test-equal (term (fv (lambda (x) (y z x)))) (term (y z))))

(define-metafunction Lambda
  fv : e -> (x ...)
  [(fv x) (x)]
  [(fv (lambda (x ...) e))
   (subtract (x_e ...) x ...)
   (where (x_e ...) (fv e))]
  [(fv (e_f e_a ...))
   (x_f ... x_a ... ...)
   (where (x_f ...) (fv e_f))
   (where ((x_a ...) ...) ((fv e_a) ...))])
```

α *equivalence* is a realtion that virtually eliminates variables from phrases and replaces them with arrows to their declarations. In lambda calculus-based languages, this transformation is often a part of the compiler, called the *static-distance* phase.

```
; (sd e) computes the static distance version of e

(define-extended-language SD Lambda
  (e ::= ....
         (K n n)
         n)
  (n ::= natural))

(define sd1 (term (K 1 1)))
(define sd2 (term 1))

(define SD? (redex-match? SD e))

(module+ test
  (test-equal (SD? sd1) #true)
  (test-equal (SD? sd2) #true))


(define-metafunction SD
  sd : e -> e
  [(sd e_1) (sd/a e_1 ())])

(module+ test
  (test-equal (term (sd/a x ())) (term x))
  (test-equal (term (sd/a x ((y) (z) (x)))) (term (K 2 0)))
```

```
  (test-equal (term (sd/a ((lambda (x) x) (lambda (y) y)) ()))
              (term ((lambda () (K 0 0)) (lambda () (K 0 0)))))
  (test-equal (term (sd/a (lambda (x) (x (lambda (y) y))) ()))
              (term (lambda () ((K 0 0) (lambda () (K 0 0))))))
  (test-equal (term (sd/a (lambda (z x) (x (lambda (y) z))) ()))
              (term (lambda () ((K 0 1) (lambda () (K 1 0)))))))

(define-metafunction SD
  sd/a : e ((x ...) ...) -> e
  [(sd/a x ((x_1 ...) ... (x_0 ... x x_2 ...) (x_3 ...) ...))
   ; bound variable
   (K n_rib n_pos)
   (where n_rib ,(length (term ((x_1 ...) ...))))
   (where n_pos ,(length (term (x_0 ...))))
   (where #false (in x (x_1 ... ...)))]
  [(sd/a (lambda (x ...) e_1) (e_rest ...))
   (lambda () (sd/a e_1 ((x ...) e_rest ...)))]
  [(sd/a (e_fun e_arg ...) (e_rib ...))
   ((sd/a e_fun (e_rib ...)) (sd/a e_arg (e_rib ...)) ...)]
  [(sd/a e_1 any)
   ; a free variable is left alone
   e_1])
```

Steps of the last formulation:

```
   (sd/a (lambda (z x) (x (lambda (y) z))) ())
-> (lambda () (sd/a (x (lambda (y) z)) ((z x))))
-> (lambda () ((sd/a x ((z x))) (sd/a (lambda (y) z) ((z x)))))
-> (lambda () ((K 0 1) (lambda () (sd/a z ((y) (z x))))))
-> (lambda () ((K 0 1) (lambda () (K 1 0))))
```

α equivalence:

```
; (=α e_1 e_2) determines whether e_1 and e_2 are α equivalent

(module+ test
  (test-equal (term (=α (lambda (x) x) (lambda (y) y))) #true)
  (test-equal (term (=α (lambda (x) (x 1)) (lambda (y) (y 1)))) #true)
  (test-equal (term (=α (lambda (x) x) (lambda (y) z))) #false))

(define-metafunction SD
  =α : e e -> boolean
  [(=α e_1 e_2) ,(equal? (term (sd e_1)) (term (sd e_2)))])

(define (=α/racket x y) (term (=α ,x ,y)))

(module+ test
  (test-results))
```

1.5 Substitution

Substitution is the syntactic equivalent of function application.

```
; (subst ([e x] ...) e_*) substitutes e ... for x ... in e_* (hygienically)

(module+ test
  (test-equal (term (subst ([1 x][2 y]) x)) 1)
  (test-equal (term (subst ([1 x][2 y]) y)) 2)
  (test-equal (term (subst ([1 x][2 y]) z)) (term z))
  (test-equal (term (subst ([1 x][2 y]) (lambda (z w) (x y))))
              (term (lambda (z w) (1 2))))
  (test-equal (term (subst ([1 x][2 y]) (lambda (z w) (lambda (x) (x y)))))
              (term (lambda (z w) (lambda (x) (x 2))))
              #:equiv =/racket)
  (test-equal (term (subst ((2 x)) ((lambda (x) (1 x)) x)))
```

```
              (term ((lambda (x) (1 x)) 2))
              #:equiv =/racket))

(define-metafunction Lambda
  subst : ((any x) ...) any -> any
  [(subst [(any_1 x_1) ... (any_x x) (any_2 x_2) ...] x) any_x]
  [(subst [(any_1 x_1) ...]  x) x]
  [(subst [(any_1 x_1) ...]  (lambda (x ...) any_body))
   (lambda (x_new ...)
     (subst ((any_1 x_1) ...)
            (subst-raw ((x_new x) ...) any_body)))
   (where  (x_new ...)  ,(variables-not-in (term any_body) (term (x ...))))]
  [(subst [(any_1 x_1) ...]  (any ...)) ((subst [(any_1 x_1) ...]  any) ...)]
  [(subst [(any_1 x_1) ...]  any_*) any_*])

(define-metafunction Lambda
  subst-raw : ((x x) ...) any -> any
  [(subst-raw ((x_n1 x_o1) ... (x_new x) (x_n2 x_o2) ...) x) x_new]
  [(subst-raw ((x_n1 x_o1) ...)  x) x]
  [(subst-raw ((x_n1 x_o1) ...)  (lambda (x ...) any))
   (lambda (x ...) (subst-raw ((x_n1 x_o1) ...)  any))]
  [(subst-raw [(any_1 x_1) ...]  (any ...))
   ((subst-raw [(any_1 x_1) ...]  any) ...)]
  [(subst-raw [(any_1 x_1) ...]  any_*) any_*])

(module+ test
  (test-results))
```

References

[1] Robert Bruce Findler, Casey Klein, Burke Fetscher, and Matthias Felleisen. (2015) *Redex: Practical Semantics Engineering*, https://docs.racket-lang.org/redex/index.html.

Mingzhe Hu
humingzhework@163.com
Hefei, China

July 2023