意味論的レゴ
(Semantic Lego)

デビッド エスピノーザ (著)
David Espinosa

2014 年 12 月 22 日

Columbia University Department of Computer Science New York, NY 10027 espinosa@cs.columbia.edu
Draft March 20, 1995
original: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.2885
figures including sample codes: https://github.com/iHdkz/semantic-lego

概要

表示的意味論（denotational semantics）[Sch86] は、プログラミング言語を記述するための強力な枠組みの一つである。しかしながら、表示的意味論による記述にはモジュール性の欠如、すなわち、概念的には独立した筈である言語のある特徴的機能がその言語の他の特徴的機能の意味論に影響を与えてしまうという問題がある。我々は、モジュール性を持った表示的意味論の理論を示していくことによって、この問題への対処を行なった。 Mosses[Mos92] に従い、我々は（言語の）一つの意味論を、計算 ADT（computation ADT）と言語 ADT（language ADT; ADT, abstract data type: 抽象データ型）の二つの部分に分ける。計算 ADT は、その言語にとっての基本的な意味論の構造を表現する。言語 ADT は、文法によって記述されるものとしての実用的な言語の構成を表現する。我々は計算 ADT を用いて言語 ADT を定義することとなるが、（言語 ADT の持つ性質はその組み立てられた言語 ADT にみられるものだけではなく）現実には、多くの異なる計算 ADT（の存在）によって、言語 ADT は多様な形態を持つものである。

目次

謝辞私の婚約者であるマリー Ng は、何年もの間わたってこのテーマの完成を根気強く待ってくれた。

1 導入

1.1 ADT（抽象データ型）としての言語

(compute '((lambda x (* x x)) 9)) =¿ 81
(compute (
=¿ 81

Listing 1 インタプリタ

```
(define (eval exp env)
  (cond ((number? exp) (eval-number exp env))
        ((variable? exp) (eval-variable exp env))
        ((lambda? exp) (eval-lambda exp env))
        ((if? exp) (eval-if exp env))
        ((+? exp) (eval-+ exp env))
        ((*? exp) (eval-* exp env))
        (else (eval-call exp env))))

```

```
10 (define (compute exp)
11   (eval exp (empty-env)))
12
13 (define (eval-number exp env)
14   exp)
15
16 (define (eval-variable exp env)
17   (env-lookup exp env))
18
19 (define (eval-lambda exp env)
20   (lambda (val)
21     (eval (lambda-body exp)
22           (extend-env env (lambda-variable exp) val))))
23
24 (define (eval-call exp env)
25   ((eval (call-operator exp) env)
26    (eval (call-operand exp) env)))
27
28 (define (eval-if exp env)
29   (if (eval (if-condition exp) env)
30     (eval (if-consequent exp) env)
31     (eval (if-alternative exp) env)))
32
33 (define (eval-+ exp env)
34   (+ (eval (op-arg1 exp) env)
35      (eval (op-arg2 exp) env)))
36
37 (define (empty-env) '())
38
39 (define (env-lookup var env)
40   (let ((entry (assq var env)))
41     (if entry
42       (error "Unbound variable: " var)
43       (right entry))))
44
45 (define (env-extend var val env)
46   (pair (pair var val) env))
```

```
1 (define %amb
2   (let ((unit (get-unit 'lists 'top))
```

```
      (bind (get-bind 'lists 'top)))
  (lambda (x y)
    (bind x
          (lambda (lx)
            (bind y
                  (lambda (ly)
                    (unit (append lx ly)))))))))
```

```
;; Computation ADT
(define computations
  (make-computations environments continuations nondeterminism))

;; Basic semantics

(-> Env (let A0 (List Ans) (-> (-> Val A0) A0)))

;; Simplified %amb

(lambda (x y)
  (lambda (env)
    (lambda (k)
      (reduce append ()
              (map k (append ((x env) list) ((y env) list)))))))

;; Example

(compute
  (%+ (%num 1)
      (%call/cc
        (%lambda 'k
                 (&* (%num 10)
                     (%amb (%num 3) (%call (%var 'k) (%num 4))))))))

;; => (31 51)
```

```
;; Computation ADT

(define computations
  (make-computations environments continuations2 nondeterminism))

```

```
;; Basic semantics

(-> Env (let AO (List Ans) (-> (-> Val AO) AO)))

;; Simplified %amb

(lambda (x y)
  (lambda (env)
    (lambda (k)
      (append ((x env) k) ((y env) k)))))

;; Example

(compute
  (%+ (%num 1)
      (%call/cc
        (%lambda 'k
                 (%* (%num 10)
                     (%amb (%num 3) (%call (%var 'k) (%num 4))))))))

;; => (31 5)
```

```
;; Computation ADT

(define computations
  (make-computations environments nondeterminism continuations))

;; Basic semantics

(-> Env (let AO (List Ans) (-> (-> (List Val) AO) AO)))

;; Simplified %amb

(lambda (x y)
  (lambda (env)
    (lambda (k)
      ((x env)
       (lambda (a)
         ((y env)
          (lambda (a0)
```

```
19                (k (append a a0)))))))))
20
21  ;; Example
22
23  (compute
24    (%+ (%num 1)
25        (%call/cc
26          (%lambda 'k
27                   (%* (%num 10)
28                       (%amb (%num 3) (%call (%var 'k) (%num 4))))))))
29
30  ;; => (5)
```

```
1   ;; Computation and language ADTs
2
3   (define computations
4     (make-computations cbn-environments exp-environments))
5
6   (load "error-values" "numbers" "booleans" "numeric-predicates"
7         "environmens" "exp-environments")
8
9   ;; Simplified %evar and %elet
10
11  (lambda (name)
12    (lambda (env)
13      (lambda (eenv)
14        (if (env-lookup eenv name)
15          ((right (env-lookup eenv name)) eenv) ; ***
16          (in 'errors (unbound-error name))))))
17
18  (lambda (name c1 c2)
19    (lambda (env)
20      (lambda (eenv)
21        ((c2 env) (env-extend eenv name (c1 env))))))
```

1.2 単層（monolithic）インタプリタ

1.3 部品的（modular）インタプリタ

図 1.8 単層インタプリタ 一部 ,

図 1.9 単層インタプリタ 二部,
図 1.10 格納（Store）ADT

1.3.1 インタプリタの持ち上げ（lifting）

図 1.11 持ち上げ演算子,
図 1.12 値レベル

1.3.2 多階層（stratified）インタプリタ

図 1.13 格納（Store）レベル,
図 1.14 環境レベル

1.4 例示

この節における例示は SEMANTIC LEGO（以後 SL と略記）の入力/出力の振る舞いを示しており、次の二つの章はその背後の仕組みを説明する。我々はそこで、

フル装備の、scheme に似たある言語、非決定性と継続の間の三つの相互作用、Lamping の単一化された、パラメータで指定されるシステム（unified system of parametrization）、再開機能（resumption）を用いて模型化された一つの並列言語を考えることになる。図 1.15 レベル交渉演算子
図 1.16 部品的インタプリタ 一部
図 1.17 部品的インタプリタ 二部

1.5 ある scheme に似た言語

我々は、環境（environment）、手続きの値呼び出し（call by value procedure）、格納（store）、継続（continuation）、非決定性基盤（nondeterminism）及び例外処理（error）の各機能を持つ言語のインタプリタを構成する。図 1.18 は完全なその言語の仕様、基本的な意味論及び例としての二つの式を示している。SL は、接頭式（prefix form）内において、自動的に基本的な意味論の記述が生成される。

我々は、二つの段階を経てインタプリタを建設する。要点としては、

1.5.1 非決定性（nondeterministic）と継続（continuation）

図 1.19 図 1.21