意味論的レゴ
(Semantic Lego)

デビッド エスピノーザ (著)　　岩城 秀和（訳）
David Espinosa

2014 年 12 月 28 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

```
1  (define (eval exp env)
2    (cond ((number? exp) (eval-number exp env))
3    ((variable? exp) (eval-variable exp env))
4    ((lambda? exp) (eval-lambda exp env))
5    ((if? exp) (eval-if exp env))
6    ((+? exp) (eval-+ exp env))
7    ((*? exp) (eval-* exp env))
8    (else (eval-call exp env))))
9
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))
```

1zw 図 1.1 インタプリタ

```
1   (define (empty-env) '())
2
3   (define (env-lookup var env)
4     (let ((entry (assq var env)))
5       (if entry
6         (error "Unbound␣variable:␣" var)
7         (right entry))))
8
9   (define (env-extend var val env)
10    (pair (pair var val) env))
```

1zw 図 1.2 環境 ADT（環境抽象データ型）

```
1   (define variable? symbol?)
2
3   (define (lambda? exp)
4     (eq? 'lambda (first exp)))
5
6   (define lambda-variable second)
7   (define lambda-body third)
8
9   (define call-operator first)
10  (define call-operand second)
11
12  (define (if? exp)
13    (eq? 'if (first exp)))
14
15  (define if-condition second)
```

```
(define if-consequent third)
(define if-alternative fourth)

(define (+? exp)
  (eq? '+ (first exp)))

(define (*? exp)
  (eq? '* (first exp)))

(define op-arg1 second)
(define op-arg2 third)
```

1zw 図 1.3 述語と選択子（selector）の表現

```
(define (%num x) x)
(define (%var name) name)
(define (%lambda name exp) (list 'lambda var exp))
(define (%if e1 e2 e3) (list 'if e1 e2 e3))
(define (%+ e1 e2) (list '+ e1 e2))
(define (%* e1 e2) (list '* e1 e2))
```

1zw 図 1.4 構築子（constructor；コンストラクタ

```
;; Den  = Env -> Val
;; Proc = Val -> Val

(define ((%num n) env)
  n)

(define ((%var name) env)
  (env-lookup name env))

(define ((%lambda name den) env)
  (lambda (val)
    (den (env-extend env var val))))

(define ((%call d1 d2) env)
  ((d1 env) (d2 env)))

(define ((%if d1 d2 d3) env)
  (if (d1 env) (d2 env) (d3 env)))

(define ((%+ d1 d2) env)
  (+ (d1 env) (d2 env)))
```

```
22
23 (define ((%* d1 d2) env)
24   (* (d1 env) (d2 env)))
```

1zw 図 1.6 表示的（意味論的）な実装

```
 1 (define (D exp)
 2   (cond ((number? exp) (%num exp))
 3   ((variable? exp) (%var exp))
 4   ((lambda? exp)
 5    (%lambda (lambda-variable exp)
 6       (D (lambda-body exp))))
 7   ((if? exp)
 8    (%if (D (if-condition exp))
 9         (D (if-consequent exp))
10         (D (if-alternative exp))))
11   ((+? exp)
12    (%* (D (op-arg1 exp))
13        (D (op-arg2 exp))))
14   ((*? exp)
15    (%* (D (op-arg1 exp))
16        (D (op-arg2 exp))))
17   (else
18     (%call (D (call-operator exp))
19      (D (call-operand exp))))))
```

1zw 図 1.7 構文から意味への写像

1.2 単層（monolithic）インタプリタ

1.3 部品的（modular）インタプリタ

```
 1 ;; Den  = Env -> Sto -> Val x Sto
 2 ;; Proc = Val -> Sto -> Val x Sto
 3
 4 (define (((%num n) env) sto)
 5   (pair n sto))
 6
 7 (define (((%var name) env) sto)
 8   (pair (env-lookup name env) sto))
 9
10 (define (((%lambda name den) env) sto)
```

```
  (pair (lambda (val) (den (env-extend env name val)))
  sto))

(define (((%call d1 d2) env) sto)
  (with-pair ((d1 env) sto)
       (lambda (v1 s1)
         (with-pair ((d2 env) s1)
          (lambda (v2 s2)
            ((v1 v2) s2))))))

(define (((%if d1 d2 d3) env) sto)
  (with-pair ((d1 env) sto)
       (lambda (v1 s1)
         (if v1
     ((d2 env) s1)
     ((d3 env) s1)))))
```

1zw 図 1.8 単層インタプリタ 一部

```
(define ((((make-op op) d1 d2) env) sto)
  (with-pair ((d1 env) sto)
       (lambda (v1 s1)
         (with-pair ((d2 env) s1)
          (lambda (v2 s2)
            (pair (op v1 v2) s2))))))
(define %+ (make-op +))
(define %* (make-op *))

(define (((%begin d1 d2) env) sto)
  ((d2 env) (right ((d1 env) sto))))

(define (((%fetch loc) env) sto)
  (pair (store-fetch loc sto) sto))

(define (((%store loc den) env) sto)
  (with-pair ((den env) sto)
       (lambda (val sto)
         (pair 'unit (store-store loc val sto)))))

(define (with-pair p k)
  (k (left p) (right p)))
```

1zw 図 1.9 単層インタプリタ 二部

```
(define (empty-store) '())

(define (store-fetch loc sto)
  (let ((entry (assq loc sto)))
    (if entry
      (error "Empty location: " loc)
      (right entry))))

(define (store-store loc val sto)
  (pair (pair loc val) sto))
```

1zw 図 1.10 格納（Store）ADT

1.3.1 インタプリタの持ち上げ（lifting）

```
(define ((lift-p1-a0 unit bind op) p1)
  (unit (op p1)))

(define ((lift-p1-a1 unit bind op) d1)
  (bind d1
  (lambda (v1)
    (unit (op v1)))))

(define ((lift-p0-a2 unit bind op) d1 d2)
  (bind d1
  (lambda (v1)
    (bind d2
    (lambda (v2)
      (unit (op v1 v2)))))))

(define ((lift-p1-a1 unit bind op) p1 d1)
  (bind d1
  (lambda (v1)
    (unit (op p1 v1)))))

(define ((lift-if unit bind op) d1 d2 d3)
  (bind d1
  (lambda (v1)
    (op v1 d2 d3))))
```

1zw 図 1.11 持ち上げ演算子

```
;;; V = Val

(define computeV id)

(define %numV id)
(define %+V +)
(define %*V *)

(define (%ifV d1 d2 d3)
  (if d1 d2 d3))
```

1zw 図 1.12 値レベル

1.3.2 多階層（stratified）インタプリタ

```
;;; S = Sto -> V x Sto

;; Store monad

(define (unitS v)
  (lambda (sto)
    (pair v sto)))

(define (bindS s f)
  (lambda (sto)
    (let ((v*sto (s sto)))
      (let ((v (left v*sto))
      (sto (right v*sto)))
  ((f v) sto)))))

;; Lifted operators

(define (computeS den)
  (computeV (left (den (empty-store)))))

(define %numS (lift-p1-a0 unitS bindS %numV))
(define %+S (lift-p0-a2 unitS bindS %+V))
(define %*S (lift-p0-a2 unitS bindS %*V))

(define %ifS  (lift-if unitS bindS %ifV))

;; New operators

```

```
(define ((%fetchS loc) sto)
  (pair (store-fetch loc sto) sto))

(define ((%storeS loc den) sto)
  (let ((v*s (den sto)))
    (let ((v (left v*s))
    (s (right v*s)))
      (pair 'unit
      (store-store loc v s)))))

(define ((%beginS d1 d2) sto)
  (d2 (right (d1 sto))))
```

1zw 図 1.13 格納（Store）レベル

```
;;; E  = Env -> S
;;; Proc = V -> S

;; Environment monad

(define (unitE s)
  (lambda (env) s))

(define (bindE e f)
  (lambda (env)
    ((f (e env)) env)))

;; Lifted operators

(define (compute den)
  (comuteS (den (empty-env))))

(define %num (lift-p1-a0 unitE bindE %numS))

(define %+ (lift-p0-a2 unitE bindE %+S))
(define %* (lift-p0-a2 unitE bindE %*S))

(define %if (lift-if unitE bindE %ifS))

(define %fetch (lift-p1-a0 unitE bindE %fetchS))
(define %store (lift-p1-a1 unitE bindE %storeS))
(define %begin (lift-p0-a2 unitE bindE %beginS))

```

```
;; New operators

(define ((%var name) env)
  (unitS (env-lookup name env)))

(define ((%lambda name den) env)
  (unitS
    (lambda (val)
      (den (env-extend name val env)))))

(define ((%call d1 d2) env)
  (bindS (d1 env)
   (lambda (v1)
     (bindS (d2 env)
      (lambda (v2)
        (v1 v2))))))
```

1zw 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。

```
;; E = Env -> S
;; S = Sto -> V x Sto
;; V = Val

(define ((unitSE s) env)
  s)

(define ((unitVS v) sto)
  (pair v sto))

(define (((unitVE v) env) sto)
  (pair v sto))

(define ((bindSE t f) env)
  ((f (t env)) env))
```

```
16
17 (define (((bindVE t f) env) sto)
18   (let ((p ((t env) sto)))
19     (let ((v (left p))
20     (s (right p)))
21       (((f v ) env) s))))
```

1zw 図 1.15 レベル交渉演算子

```
1  ;; E  = Env -> S
2  ;; S  = Sto -> V x Sto
3  ;; V  = Val
4  ;; Proc = V -> S
5
6  (define (%num v)
7    (unitVE v))
8
9  (define ((%var name) env)
10   (unitVS (env-lookup env name)))
11
12 (define ((%lambda name den) env)
13   (unitVS
14     (lambda (val)
15       (den (env-extend env name val)))))
16
17 (define (%call d1 d2)
18   (bindVE d1
19     (lambda (v1)
20       (bindVE d2
21         (lambda (v2)
22           (unitSE (v1 v2)))))))
23
24 (define (%if d1 d2 d3)
25   (bindVE d1
26     (lambda (v1)
27       (if v1 d2 d3))))
```

1zw 図 1.16 部品的インタプリタ 第一部

```
1  (define ((make-op op) d1 d2)
2    (bindVE d1
3      (lambda (v1)
4        (bindVE d2
5          (lambda (v2)
```

```
6           (unitVE (op v1 v2)))))))
7
8  (define %+ (make-op +))
9  (define %* (make-op *))
10
11 (define (%begin d1 d2)
12   (beindVE d1
13     (lambda (v1)
14       d2)))
15
16 (define (%fetch loc)
17   (unitSE
18     (lambda (sto)
19       (pair (store-fetch loc sto) sto))))
20
21 (define (%store loc den)
22   (bindVE den
23     (lambda (val)
24       (unitSE
25         (lambda (sto)
26     (pair 'unit (store-store loc val sto)))))))
```

1zw 図 1.17 部品的インタプリタ 第二部

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）

```
1  (define %let
2    (let ((unitE (get-unit 'envs 'top))
3    (bindE (get-bind 'envs 'top))
4    (bindV (get-bind 'env-values 'top)))
5      (lambda (name c1 c2)
6        (bindV c1
7         (lambda (v1)
8           (bindE c2
```

```
9             (lambda (e2)
10          (unitE
11            (lambda (env)
12              (e2 (env-extend env name v1)))))))))))
```

1zw 図 1.19 %let ソースの定義

```
1  (define %amb
2    (let ((unit (get-unit 'lists 'top))
3    (bind (get-bind 'lists 'top)))
4      (lambda (x y)
5        (bind x
6        (lambda (lx)
7          (bind y
8          (lambda (ly)
9            (unit (append lx ly)))))))))
```

1zw 図 1.21 %amb ソースの定義

```
1  ;; Computation ADT
2  (define computations
3    (make-computations environments continuations nondeterminism))
4
5  ;; Basic semantics
6
7  (-> Env (let A0 (List Ans) (-> (-> Val A0) A0)))
8
9  ;; Simplified %amb
10
11 (lambda (x y)
12   (lambda (env)
13     (lambda (k)
14       (reduce append ()
15         (map k (append ((x env) list) ((y env) list)))))))
16
17 ;; Example
18
19 (compute
20   (%+ (%num 1)
21       (%call/cc
22   (%lambda 'k
23      (&* (%num 10)
24          (%amb (%num 3) (%call (%var 'k) (%num 4)))))))
25
```

```
;; => (31 51)
```

1zw 図 1.22 %amb バージョン 1

```
;; Computation ADT

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

;; Basic semantics

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

;; 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)
```

1zw 図 1.23 %amb バージョン 2

```
;; Computation ADT

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

;; Basic semantics

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

;; Simplified %amb

```

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

;; Example

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

;; => (5)
```

1zw 図 1.24 %amb バージョン 3

```
;; Computation and language ADTs

(define computations
  (make-computations cbn-environments exp-environments))

(load "error-values" "numbers" "booleans" "numeric-predicates"
      "environmens" "exp-environments")

;; Simplified %evar and %elet

(lambda (name)
  (lambda (env)
    (lambda (eenv)
      (if (env-lookup eenv name)
  ((right (env-lookup eenv name)) eenv) ; ***
  (in 'errors (unbound-error name))))))

(lambda (name c1 c2)
  (lambda (env)
    (lambda (eenv)
      ((c2 env) (env-extend eenv name (c1 env))))))
```

1zw 図 1.25 単一化された、パラメータによって指定されるシステム

1.5.2 **単一化された、パラメータによって指定されるシステム（**Unified system of parametrization**）**

1.5.3 **再開機能（**Resumption**）**

2 モナド（Monads）

2.1 基本的な圏論

2.1.1 圏

2.1.2 関手

2.1.3 自然変換

2.1.4 始対象の性質（initiality）

2.1.5 双対性

2.1.6 圏論と関数型プログラミング

2.1.7 参照

2.2 モナド

2.2.1 初段階目の形式化

2.2.2 二段階目の形式化

2.2.3 解釈

```
;; Identity: T(A) = A

(define (unit a)
  a)

(define (bind ta f)
  (f ta))

;; Lists: T(A) = List(A)

(define (unit a)
  (list a))

(define (bind ta f)
  (reduce append '() (map f ta)))

;; Environments: T(A) = Env -> A

(define (unit a)
  (lambda (env) a))

(define (bind ta f)
  (lambda (env)
```

```
24      ((f (ta env)) env)))
25
26  ;; Stores: T(A) = Sto -> A x Sto
27
28  (define (unit a)
29    (lambda (sto) (pair a sto)))
30
31  (define (bind ta f)
32    (lambda (sto)
33      (let ((a*s (ta sto)))
34        (let ((a (left a*s))
35        (s (right a*s)))
36    ((f a) s)))))
```

1zw 図 2.1 モナドの例 パート 1

```
1   ;; Exceptions: T(A) = A + X
2
3   (define (unit a)
4     (in-left a))
5
6   (define (bind ta f)
7     (sum-case ta
8         (lambda (a) (f a))
9         (lambda (x) (in-right x))))
10
11  ;; Monoids: T(A) = A x M
12
13  (define (unit a)
14    (pair a monoid-unit))
15
16  (define (bind ta f)
17    (let ((a1 (left ta))
18    (m1 (right ta)))
19      (let ((a*m (f a1)))
20        (let ((a2 (left a*m))
21        (m2 (right a*m)))
22    (pair a2 (monoid-product m1 m2))))))
23
24  ;; Continuations: T(A) = (A -> Ans) -> Ans
25
26  (define (unit a)
27    (lambda (k) (k a)))
```

```
28
29 (define (bind ta f)
30   (lambda (k) (ta (lambda (a) ((f a) k)))))
31
32 ;; Resumptions: T(A) = fix(X)(A + X)
33
34 (define (unit a)
35   (in-left a))
36
37 (define (bind ta f)
38   (sum-case ta
39       (lambda (a) (f a))
40       (lambda (ta) (bind ta f))))
```

1zw 図 2.2 モナドの例 パート 2

2.3 モナド射（monad morphism）

2.4 組み合わせないモナド

```
1  ;; S(A) = EnvS -> A
2  ;; T(A) = EnvT -> A
3  ;; ST(A) = EnvS -> Env T -> A
4
5  (define ((joinS ssa) envS)
6    ((ssa envS) envS))
7
8  (define ((joinT tta) envT)
9    ((tta envT) envT))
10
11 (define (((joinST ststa) envS) envT)
12   ((((ststa envS) envT) envS) envT))
```

1zw 図 2.3 組み合わさないモナド

2.5 組み合せたモナド

2.6 モナド変換子（monad transformers）

2.6.1 動機

```
1 ;; F(T)(A) = Env -> T(A)
```

```
2
3  (define (environment-transformer m)
4    (let ((unitT (monad-unit m))
5    (mapT  (monad-map m))
6    (joinT (monad-join m)))
7
8      (define (unit a)
9        (lambda (env) (unitT a)))
10
11     (define ((map f) fta)
12       (lambda (env)
13   ((mapT f) (fta env))))
14
15     (define (join ftfta)
16       (lambda (env)
17   (joinT
18     ((mapT (lambda (fta) (fta env)))
19      (ftfta env)))))
20     (make-monad unit map join)))
```

1zw 図 2.4 環境モナド変換子

2.6.2 形式化

2.6.3 モナド変換子のクラス

2.6.4 モナド変換子の組み合わせ

3 持ち上げ（lifting）

3.1 持ち上げ（lifting）

3.1.1 形式的持ち上げ

3.1.2 モナドと持ち上げ

3.2 語用論（pragmatics）

3.2.1 上昇型（bottom-up）

3.2.2 下降型（top-down）

4 多階層性（stratification）

4.1 多階層モナド（stratified monads）

4.2 多階層モナド変換子（stratified monad transformers）

4.2.1 頂変換子（top transformers）

4.2.2 底変換子（bottom transformers）

4.2.3 周辺変換子（around transformers）

4.2.4 継続変換子（continuation transformers）

4.3 計算 ADT

4.4 言語 ADT

5 結論

5.1 持ち上げ 対 多階層性

5.2 極限

5.3 関連事項

5.4 将来的事項

5.5 結論

A 雑録

A-1 **なぜ** scheme **か**

A-2 **型についての重要な点**

A-3 **型付きの値 対 型無しの値**

A-4 **拡張可能な和と積**

B　コード

B-1　モナド変換子の定義

```
;; Environments: F(T)(A) = Env -> T(A)

(define (env-trans t)
  (with-monad t
        (lambda (unit bind compute)
    (make-monad

      (lambda (a)
        (lambda (env) (unit a)))

      (lambda (c f)
        (lambda (env)
          (bind (c env)
          (lambda (a)
            ((f a) env)))))

      (lambda (c f)
        (compute (c empty-env) f))

      ))))
```

1zw 図 B.1 環境変換子

```
;;; Exceptions: F(T)(A) = T(A + X)

(define (exception-trans t)
  (with-monad t
        (lambda(unit bind compute)
    (make-monad

      (lambda (a) (unit (in-left a)))

      (lambda (c f)
        (bind c (sum-function f (lambda (x) (unit (in-right x))))))
      (lambda (c f)
        (compute c (sum-function f compute-x)))

      ))))
```

1zw 図 B.2 例外変換子

```
;;; Continuations: F(T)(A) = (A -> T(Ans)) -> T(Ans)

(define (continuation-trans t)
  (with-monad t
        (lambda (unit bind compute)
    (make-monad

      (lambda (a)
        (lambda (k) (k a)))

      (lambda (c f)
        (lambda (k)
          (c (lambda (a) ((f a) k)))))

      (lambda (c f)
        (compute (c (compose1 unit value->answer)) f))

      ))))
```

1zw 図 B.3 継続変換子

```
;;; Stores: F(T)(A) = Sto -> T(A * Sto)

(define (store-trans t)
  (with-monad t
        (lambda (unit bind compute)
    (make-monad

      (lambda (a)
        (lambda (sto)
          (unit (pair a sto))))

      (lambda (c f)
        (lambda (sto)
          (bind (c sto)
          (lambda (as)
            ((f (left as)) (right as))))))

      (lambda (c f)
        (compute (c (initial-store))
           (lambda (a*s)
             (compute-store (f (left a*s)) (right a*s)))))

      ))))
```

1zw 図 B.4 ストア変換子

```
;;; Lifting 1: F(T)(A) = 1 -> T(A)

(define (lift1-trans t)
  (with-monad t
        (lambda (unit bind compute)
    (make-monad

      (lambda (a)
        (lambda () (unit a)))

      (lambda (c f)
        (lambda ()
          (bind (c) (lambda (a) ((f a))))))

      (lambda (c f)
        (compute (c) f))

      ))))
```

1zw 図 B.5 第一持ち上げ変換子

```
;;; Lifting 2: F(T)(A) = T(1 -> A)

(define (lift2-trans t)
  (with-monad t
        (lambda (unit bind compute)
    (make-monad

      (lambda (a)
        (unit (lambda () a)))

      (lambda (c f)
        (bind c (lambda (l) (f (l)))))

      (lambda (c f)
        (compute c (lambda (l) (f (l)))))

      ))))
```

1zw 図 B.6 第二持ち上げ変換子

```
;;; Lists: F(T)(A) = T(List(A))

(define (list-trans t)
  (with-monad t
        (lambda (unit bind compute)

    (define (amb x y)
      (bind x
      (lambda (x)
        (bind y
        (lambda (y)
          (unit (append x y)))))))

    (make-monad

      (lambda (a)
        (unit (list a)))

      (lambda (c f)
        (bind c
        (lambda (l)
          (reduce amb (unit '()) (map f l)))))

      (lambda (c f)
        (compute c (lambda (l) (map f l))))

      ))))
```

1zw 図 B.7 リスト変換子

```
;;; Monoids: F(T)(A) = T(A * M)

(define (monoid-trans t)
  (with-monad t
        (lambda (unit bind compute)
    (make-monad

      (lambda (a) (unit (pair a (monoid-unit))))

      (lambda (c f)
        (bind c
        (lambda (a*m)
          (let ((c2 (f (left a*m))))
            (bind c2
            (lambda (a*m2)
              (unit
          (pair (left a*m2
                (monoid-product
                  (right a*m) (right a*m2)))))))))))

      (lambda (c f)
        (compute
          c (lambda (a*m)
        (compute-m (f (left a*m)) (right a*m)))))

      ))))
```

1zw 図 B.8 モノイド変換子

```
;;; Resumptions: F(T)(A) = fix(X) T(A + X)

(define (resumption-trans t)
  (with-monad t
        (lambda (unit bind compute)
    (make-monad

      (lambda (a) (unit (in-left a)))

      (lambda (c f)
        (let loop ((c c))
          (bind c
          (sum-function
            f (lambda (c)
          (unit (in-right (loop c))))))))

      (lambda (c f)
        (compute
          (let loop ((c c))
      (bind c
            (sum-function
        (compose1 unit f)
        loop)))
          id))

      ))))
```

1zw 図 B.9 再開機能変換子

C 参考文献