意味論的レゴ
(Semantic Lego)

デビッド エスピノーザ (著)
David Espinosa

岩城 秀和 (訳)

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

Translator's 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 Mary Ng は、数年間に渡ってこの論文のために忍耐強く待ってくれた。私は彼女なしでもそれを成すことができたでしょうが、それはずっと悪いものだったろうし、既に悪いものだった。Mary は大学院における苦もない最も嬉しい結果である。

私の母である Joanne Espinosa は２８年間に渡る偉大さを持っている。ありがとう、ママ。

ジェラルド J サスマン（Gerald J. Sussman)、私は彼を１０年前から知っていた、は常に一つの刺激であり続けた。彼の学生の間における彼の信用は決して落ちなかったし、彼と話をすることは、一瞬の間だけかもしれないが、あなたは何でもできるのだよと信じさせた。より物質的な面では、Jerry は私が去年一年間（またはそれ以上）彼の研究室に入り浸り状態になるのを許してくれた。

コロンビア大における私の指導教官である Sal Stolfo は、私の大学院キャリアの中で一人の極端に寛容な監視者であり続けた。私は指導教官として Sal を多少なりとも選択した、その理由は彼がいい人だったからだ。注目すべきこととして、彼は今もそうである。

私の防衛委員会、Gail Kaiser、Ken Ross、そして Mukesh Dalal は、コロンビア生活から私が抜け出すのを助けてくれた。Albert Greenberg は、AT & T での何回かの夏の間における職なしの状態から私を助けてくれた。『博士研究員を雇うために（出されている）論文の少ない仕事を取ってきた』から、彼はまず私を雇ってくれた。私たちは、並列フーリエ変換をうまくやり遂げることと通信ネットワークのモデルを解決することを楽しんでいた。あれとモナドの間のつながりは明白な、いや、現在においては、曖昧なようだ。

AT & T の博士奨学金は私を 5 年間支えてくれた、そしてそれらは 5 年間では無理だと私に懇願させることさえなかった。ただ、残念ながら、それらが私に与えたもののすべてはお金であった（Albert にも関わらず)。

Phil Chan、Mauricio Hernandez、Sushil Da Silva、Paul Michelman、そして Bulent Yener 達はコロンビアで付き合ってくれてありがとう。同様に Michael Blair、Koniaris、Natalya Cohen、Raj Surati、そして MIT の四階フロアにいるの他のすべての人たち。

また、私の音楽仲間、Joseph Briggs、Kerstin Kup、Brian と Karen Neal、Lois Winter、そして Johelen Carleton についても感謝する。Albert Meyer は、多くの場面で非常に愉快だった。意味論の内側と外側について知っている誰かと、その歴史の多くに沿って話をすることは素晴らしいことです。あなたはとてもじゃないが論文からあれを得ることはできない。

エウジニオ モッジ（Eugenio Moggi）は、（私のこの論文に）不可欠な彼の仕事に対して私の感謝を受けるに値する。私とモッジの関係が個人的というよりも科学的であるということから（特に私が彼に会ったことがないということから)、Albert はモッジをここに含めることに異議を唱えた。Albert は、論理学者として、彼が見つけることができる些細なことにはなんでもこだわる。

Jonathan Rees は私にモナドと圏論を紹介してくれた。私たちは後々一緒にもっと仕事ができるだろうと期待しています。今、彼は英国でバグの追いかけをしてしまっている。

Bill Rozas は私を多くの、多くの場面で助けて出してくれたそして意味論とアーキテクチャーについて議論することでいつも楽しませてくれた。Bill は信じられないほど気前がよく、そして私に、私たちがそうでないときでさえ、私たちは対等だと思わせてくれた。私はこの特性をもつ彼が妬ましい。

Carl Gunter は助言と援助の大きな源であった。彼と最初に会ったのは 1992 年の LFP の時で、彼は穏やかな話し方をする男だった、激論となった意味論に関する議論が終わった後に、「実は、現実的な答えは・・・」と言うような。彼の説明と彼の本 [Gun92] はクリスタルのように明瞭だ。

Charles Leiserson は、一人の見習いとして MIT に通うための納得する証拠を提出してくれた、私が Brown を訪れた時にそこで一番興味深い人物であるとしてくれることによって。彼は私にアルゴリズムを教えるという偉大な仕事をしたが、思ったとおりその分野はなんにせよ非常に簡単だった。Charles はまさに何についてでも形式化するという驚くべき技能を持っている。

Franklyn Turbak と私はここ 2 年間インタプリタと言語いじりを非常に楽しんでいた。Lyn に会うまでずっと、私は形式意味論は、読者を実際の内容を持った領域について考えることをわざと混乱させるための無意味なごちゃ混ぜのギリシャ文字だと思っていた。私の現在の見解は、この論文を読むことによってあなたが見つけ出さなくてはならない。

1 はじめに

表示的意味論はプログラミング言語を定義するための強力なフレームワークである。それを使用することで、我々は簡潔かつ明確な言語を記述し、実際のプログラムを実行するインタプリタを構築することができる。特にその力を考慮するにあたって、理解することが難しい理論ではない。

ただ残念ながら、表示的な記述を主に読んだり書いたりすることは困難である、なぜならそれら記述にはモジュール性が欠如しているためである。言語のそれぞれの構成概念は、言語の基礎を形成している意味論的に組み立てられたブロックのすべてと相互作用をする。例えば、もし我々がストアを用いて割り当て（assignment）を実現するならば、すべての言語の構成概念は、割り当てそのものではなくそのストアと情報やりとりをしなければならない。この相互作用の複雑さは表示的な記述をより難解にする。

この論文では表示的記述のモジュラーな書き方を提示するが、そのモジュラーな書き方は、構成要素からインタプリタを組み立てる Scheme プログラムである Semantic Lego[*1](SL) として自動化した。本質的に、SL は言語を記述するための言語です。この論文は、いくつかの重要な貢献をする：

- 私たちは、抽象データ型としてのプログラミング言語のアイディアを再導入する。このスタイルで書かれたインタプリタは通常よりも短く明確である。
- 私たちは、より多くの人たちがアクセスすることができるように、単純な言葉で持ち上げの Moggi の理論を言い直した。
- 私たちは持ち上げよりもより強力でより単純な階層（stratification）の新理論を説明するこの理論は意味論的代数についての Mosses の仕事に構造とモジュラー性を追加することで拡張したものである。
- 我々はモジュラーインタプリタの二つの書き方を示す、それらはそれぞれ持ち上げ (lifting) と階層 (stratification) に基づいている。
- 私たちは Semantic Lego、モジュラーインタプリタの構文の集まりで階層に基づいたものを提示し、幾つかの例を与える。

この論文は幾つかの重要な結果を持つ。

- 私たちは理解し、議論し、そして言語を分解することによってより良い言語教えることができる。我々はいくつかの単純な機能の組み合わせとして、それを見るまで例えば、並列処理の resumptions モデルは、複雑な表示されます。
- 私たちは、新しい言語を試すことができる。SL は、設計者に高次元の問題を自由に考えさせるようにしたまま、表示的記述に関連した管理作業を処理する。SL の基礎となる理論は、新しい言語の構成概念を提案する助けにもなる。

[*1] レゴ（LEGO) は登録商標です。

以下の話は、SL の力を示すものである。MIT の大学院のプログラミング言語コースのための三人のティーチングアシスタント（TA）は、状態の存在する場合における洗練された制御構文（shift）の意味を記述する必要があった。彼らは制御と状態を独立して理解してはいるものの、問題の集まりを区分する以前に、適合するこれら特徴の間の相互作用を見つけ出すことができなかった。

SL を用いることで、1 分足らずで私は二つの解法を作り出した。事実、SL は完全なインタプリタを形成しており、構文を要求される単なる一つのものではない。もう一つの意味論的完全なものである、例外 (error) を追加することもまた簡単である。SL は徹底的にテストされているので、もしそれらがうまく形式付けられているかどうかを確認するための定義を試してみる必要はない。

この論文は次のように構成されている。3 章は持ち上げ (lifting) について議論し、第 4 章では階層 (stratification) について議論する。第 5 章はそれら二つのアプローチを比較し、以前のものを再検討する。付録 A においてはこの論文に接線方向に関連する話題を取り上げる。

私たちは表示的意味論と関数型プログラミングについての初歩的な理解を前提としている、さらなる背景については [Wad92] を参照。すべての例とコードの断片は Scheme [CR91] で書かれている。

この章の残りでは、抽象データ型としての言語を提示し、通常のインタプリタの書き方はモジュラーではないということを実演し、モジュラーインタプリタの二つの書き方について示し、そして例題の集まりとともに SL を練習していく。

1.1 ADT（抽象データ型）としての言語

我々は [ASS85] のスタイルの単純なインタプリタから始め、できるだけ多くの構文の問題を除き、そのインタプリタを本質的なものへと単純化する。このアプローチは、(アベルソン (Abelson) とサスマン (Sussman) の意味での)『メタ言語的抽象化 (metalinguistic abstraction)』は通常の抽象化とは違いがないことを示している。言い換えれば、新しい言語を形成するために言語の『外に出る』必要はない。それはまたインタプリタの記述を短くし、構文と意味の違いをよりはっきりさせる合理化されたインタプリタの文体を提供する。

図で表されているのは、純粋関数型言語のための単純なインタプリです。

1.1 1.3 その典型的な使用は

```
(compute '((lambda x (* x x)) 9))
=> 81

(compute (%call (%lambda 'x (%* (%var 'x) (%var 'x))) (%num 9)))

=> 81

(define ((f a) b) ...)
```

```
(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))))

(define (compute exp)
  (eval exp (empty-env)))

(define (eval-number exp env)
  exp)

(define (eval-variable exp env)
  (env-lookup exp env))

(define (eval-lambda exp env)
  (lambda (val)
    (eval (lambda-body exp)
    (extend-env env (lambda-variable exp) val))))

(define (eval-call exp env)
  ((eval (call-operator exp) env)
   (eval (call-operand exp) env)))

(define (eval-if exp env)
  (if (eval (if-condition exp) env)
    (eval (if-consequent exp) env)
    (eval (if-alternative exp) env)))

(define (eval-+ exp env)
  (+ (eval (op-arg1 exp) env)
     (eval (op-arg2 exp) env)))

(define (empty-env) '())

(define (env-lookup var env)
  (let ((entry (assq var env)))
```

```
    (if entry
      (error "Unbound␣variable:␣" var)
      (right entry))))

(define (env-extend var val env)
  (pair (pair var val) env))
```

1zw 図 1.1 インタプリタ

```
(define (empty-env) '())

(define (env-lookup var env)
  (let ((entry (assq var env)))
    (if entry
      (error "Unbound␣variable:␣" var)
      (right entry))))

(define (env-extend var val env)
  (pair (pair var val) env))
```

1zw 図 1.2 環境 ADT（環境抽象データ型）

```
(define f (lambda (a) (lambda (b) ...)))
```

$$Den = Env \to Val$$

```
(define variable? symbol?)

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

(define lambda-variable second)
(define lambda-body third)

(define call-operator first)
(define call-operand second)

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

(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)))

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

1zw 図 1.6 表示的（意味論的）な実装

```
(define (D exp)
  (cond ((number? exp) (%num exp))
  ((variable? exp) (%var exp))
  ((lambda? exp)
   (%lambda (lambda-variable exp)
      (D (lambda-body exp))))
  ((if? exp)
   (%if (D (if-condition exp))
        (D (if-consequent exp))
        (D (if-alternative exp))))
  ((+? exp)
   (%* (D (op-arg1 exp))
       (D (op-arg2 exp))))
  ((*? exp)
   (%* (D (op-arg1 exp))
       (D (op-arg2 exp))))
  (else
    (%call (D (call-operator exp))
     (D (call-operand exp))))))
```

1zw 図 1.7 構文から意味への写像

```
(%+ (%num 1) (%num 2))
```

1.2 単層（monolithic）インタプリタ

$$Den = Env \to Sto \to Val \times Sto$$

```
(define (((%num n) env) sto)
  (pair n sto))

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

1.3 部品的（modular）インタプリタ

$$Den = Env \to Sto \to Val \times Sto$$

$$E = Env \to Sto \to Val \times Sto$$
$$S = Sto \to Val \times Sto$$
$$V = Val$$

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

(define (((%num n) env) sto)
  (pair n sto))

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

(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

```
(define (unit a)
  (list a))

(define (bind tb f)
  (flatten (map f tb)))

(define (square-list l)
  (bind l (lambda (n) (unit (square n)))))
```

1.3.1 インタプリタの持ち上げ（lifting）

$$f : X \times A \times A \to A$$

$$f' : X \times B \times B \to B$$

```
(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))

(define (((bindVE t f) env) sto)
  (let ((p ((t env) sto)))
    (let ((v (left p))
    (s (right p)))
      (((f v ) env) s))))
```

1zw 図 1.15 レベル交渉演算子

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

(define (%num v)
  (unitVE v))

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

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

(define (%call d1 d2)
  (bindVE d1
    (lambda (v1)
      (bindVE d2
        (lambda (v2)
          (unitSE (v1 v2)))))))

(define (%if d1 d2 d3)
  (bindVE d1
    (lambda (v1)
      (if v1 d2 d3))))
```

1zw 図 1.16 部品的インタプリタ 第一部

```
(define ((make-op op) d1 d2)
  (bindVE d1
    (lambda (v1)
      (bindVE d2
        (lambda (v2)
          (unitVE (op v1 v2)))))))

(define %+ (make-op +))
(define %* (make-op *))

(define (%begin d1 d2)
  (beindVE d1
     (lambda (v1)
       d2)))

(define (%fetch loc)
  (unitSE
    (lambda (sto)
      (pair (store-fetch loc sto) sto))))

(define (%store loc den)
  (bindVE den
    (lambda (val)
      (unitSE
        (lambda (sto)
    (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）

```
(define %let
  (let ((unitE (get-unit 'envs 'top))
  (bindE (get-bind 'envs 'top))
  (bindV (get-bind 'env-values 'top)))
    (lambda (name c1 c2)
      (bindV c1
       (lambda (v1)
         (bindE c2
          (lambda (e2)
      (unitE
        (lambda (env)
          (e2 (env-extend env name v1)))))))))))
```

1zw 図 1.19 %let ソースの定義

```
(define %amb
  (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)))))))))
```

1zw 図 1.21 %amb ソースの定義

```
;; 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)
```

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）

この章では、我々はまず幾つかの基本的な圏論を提示し、モナド、モナドの間の射、モナドの組み合わせそしてモナド変換子について議論する。これはあなたがモナドについて常に知りたがったことすべてのように聞こえるかもしれないが、現実にはその表面をかろうじてひっかいているものだ。より詳細な情報については [BW85,Mog89a] を参照のこと。

モナドは１９９０年代の関数型プログラミングコミュニティで「熱い話題」だったかもしれないが、現実の「モナドの激増」は、それらがまず考え出された１９６０年代の間に圏論と代数的トポロジーのコミュニティの中で発生した。私は自分自身１９９０年代基準でかなりよい「モナドハッカー」だと考えるが、１９６０年代の一覧表に乗ってさえいないということを認めなくてはならない。たとえそうであっても、私は計算機科学者が、「モナド、、これらは状態についてのものじゃないのかい？」と尋ねることを聞くことがほとんど無いことに気づいた。それは、「代数、、それは１＋１＝２についてのものなんじゃないの？」と聞くようなものだ。

2.1　基本的な圏論

この節では、我々は圏論における基本的な概念を定義し、圏論と関数型プログラミングとの間の関係性、そして幾つかの参考文献について述べる。

2.1.1　圏

圏は型付き関数の合成を抽象化する。一つの圏は、対象（これらは型である）の集合、射（これらは関数である）の集合、そして射の合成演算子からなる。各々の射は、一つの対象（ドメイン）からもう一つの対象（余ドメイン）への方向を指す。もし f : A → B かつ g : B → C が二つの射であるならば、g・f : A → C はそれらの合成である。他とは区別された各々の対象からそれ自身への恒等射が存在する。合成は左単位則、右単位則として恒等射に関して結合的でなくてはならない。

我々が用いる基本的な圏は、我々が意味論や関数プログラミングをしているかどうかに左右されるものだ。意味論において、我々は適合する領域理論（domain theory）を用いる（[Gun92] を参照）。関数型プログラミングにおいては、我々は、この場合においては、Scheme[CR91] の型と関数を使用する。Scheme は明示的には型を持っていないため、我々はそれらを我々自身で想像しなくてはならない。圏においては、合成は、適用よりも主たるものである。関数型プログラミングにおいては、組み合わせ言語でプログラムを行わない限りは、その逆である。この視点の変化は実際におけるいくつかの問題をもたらす。我々は最も便利な方を使用する。

2.1.2 関手

圏論において、我々は対象のクラスを定義するときはいつでも、我々はそれらの間の適切な写像も定義する、それ故に、それらは一つの圏となる。このような理由から、これから我々は圏の間の写像を考える。圏 C と D の間の関数 T は C の対象から D の対象への写像である。自己関数とは、圏からその圏自身への関数である。我々の場合、一つの自己関数は型構築子である。それは他の型から一つの型を構築する。例えば、T(A) = List(A) は我々の好きな任意の型のリストを構築する。我々の用いる他の型構築子は、関数空間（→)、積（×)、和（+）である。関数は圏の間の写像として不十分である、なぜならば射についての作用がないからである。我々は関手 T : C → D を、これもまた T と呼ばれる以下のような C の射から D の射に移す関数 mapT と同様となるように定義する。

```
;; mapT : (A -> B) -> (T(A) -> T(B))

(mapT id)         = id
(mapT (oC g f)    = (oD (mapT g) (mapT f))
```

自己関手は圏からその圏自身への関手である、だから我々はただ一つの合成演算子しか必要としない。例えば、リストに関する普通の map 関数は T(A) = List(A) を自己関手にする。

```
;; T(A) = A × A

(define ((map f) ta)
  (pair (f (left ta)) (f (right ta))))

;; T(A) = Env -> A

(define (((map f) ta) env)
  (f (ta env)))
```

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 参考文献

2.2 モナド

この節においては、我々はモナドについて二つの定式化を提示し、それらの背後の洞察について議論する。モナドは付加的な構造を伴った関手である、同様にして、関手は付加的な構造を持った関数である。

2.2.1 一つ目の定式化

モナドは、自己関手と二つの自然変換

$$\texttt{unit} : A \to T(A)$$

$$\texttt{join} : T(T(A)) \to T(A)$$

からなる三つ組[*2] (T, `unit`, `join`) である。ここで、`unit` は恒等関手から T へ自然であり、値は T へ写される。例えば、リストモナド用の `unit` は `list` である。`unit` は入射的（injective）であることを要請されないが、その代わり、it actually is in most applications. `join` は $T \circ T$ から T へ自然であり、多重の T を一重の T へ平らにする。リストモナド用の `join` は `flatten` である。

環境モナド $T(A) = Env \to A$ 用の `unit` と `join` は、

```
(define ((unit a) env)
  a)

(define ((join tta) env)
  ((tta env) env))
```

`unit` と `join` は（以下の）付加的な性質を満たさなくてはならない

```
(o join unit)          = id              : T(A) → T(A)
(o join (map unit))    = id              : T(A) → T(A)
(o join (map join))    = (o join join)   : T(T(T(A))) → T(A)
```

この定式化は修正されたモノイドとしてのモナド [Mac71] を示している[訳注 1)]（そのため、そのような名称となっている）なお、ここで `unit` は恒等元（identity）であり `join` はモノイド演算子である。上記の法則は、左・右単位則と結合規則である。

表 2.1 は意味論において使用される幾つかの共通したモナドの型構築子を示している。我々は次の節でそれらの `unit` と `join` 演算子を記述する（二つ目の定式化を経過した上で）。

2.2.2 二つ目の定式化

2.2.3 解釈

[*2] モナドはまたトリプル（triples）とも呼ばれる。

```
;; 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)
    ((f (ta env)) env)))

;; Stores: T(A) = Sto -> A x Sto

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

(define (bind ta f)
  (lambda (sto)
    (let ((a*s (ta sto)))
      (let ((a (left a*s))
      (s (right a*s)))
  ((f a) s)))))
```

1zw 図 2.1 モナドの例 パート 1

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

(define (unit a)
  (in-left a))

(define (bind ta f)
  (sum-case ta
      (lambda (a) (f a))
      (lambda (x) (in-right x))))

;; Monoids: T(A) = A x M

(define (unit a)
  (pair a monoid-unit))

(define (bind ta f)
  (let ((a1 (left ta))
  (m1 (right ta)))
    (let ((a*m (f a1)))
      (let ((a2 (left a*m))
      (m2 (right a*m)))
  (pair a2 (monoid-product m1 m2))))))

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

(define (unit a)
  (lambda (k) (k a)))

(define (bind ta f)
  (lambda (k) (ta (lambda (a) ((f a) k)))))

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

(define (unit a)
  (in-left a))

(define (bind ta f)
  (sum-case ta
      (lambda (a) (f a))
```

```
(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）

$$\texttt{mapK} : (A \to S(B)) \to (A \to T(B))$$

```
;; f : A →S(B)
;; g : B →S(C)

(mapK idS)       = idT
(mapK (oS g f))  = (oT (mapK g) (mapK f))

(K (unitS a))    = (unitT a)
(K (bindS sa f)) = (bindT (K sa) (o K f))

(reverse (list a))         = (list a)
(reverse (append-map f l)) = (append-map (o reverse f) (reverse l))
```

2.4 組み合わせないモナド

```
;; S(A) = EnvS -> A
;; T(A) = EnvT -> A
;; ST(A) = EnvS -> Env T -> A

(define ((joinS ssa) envS)
  ((ssa envS) envS))

(define ((joinT tta) envT)
  ((tta envT) envT))

(define (((joinST ststa) envS) envT)
  ((((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

$$\texttt{swap} : TS \to ST$$

$ST = S \circ T$
`map = mapS o mapT`

$(C1)$`unitST = unitS o unitT = mapS(unitT) o unitS`

$(C2)$`joinST o mapST(unitS) = mapS(joinT)`
$(C3)$`joinST o mapS(unitT) = joinS`

$(C4)$`joinS o mapS(joinS)          = joinST o joinS`
$(C5)$`joinST o mapST(mapS(joinT))  = mapS(joinT) o joinST`

2.6 モナド変換子（monad transformers）

2.6.1 動機

$$F(T)(A) = Env \to T(A)$$

$$\texttt{ftfta} : Env \to T(Env \to T(A))$$

$$Den(A) = Sto \to List(A \times Sto)$$

```
(define (unit a)
  (lambda (sto) (list (pair a sto))))
```

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

(define (environment-transformer m)
  (let ((unitT (monad-unit m))
  (mapT  (monad-map m))
  (joinT (monad-join m)))

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

    (define ((map f) fta)
      (lambda (env)
  ((mapT f) (fta env))))

    (define (join ftfta)
      (lambda (env)
  (joinT
    ((mapT (lambda (fta) (fta env)))
     (ftfta env)))))
    (make-monad unit map join)))
```

1zw 図 2.4 環境モナド変換子

```
(define (unitS a)
  (lambda (sto) (pair a sto)))

(define (unitL a)
  (list a))

(define (unitT a)
  (pair a (empty-store))
```

2.6.2 形式化

```
(define (unitT a)
  (pair a (empty-store))

;; F(T)(A) = List(T(A))

(define ((mapF K) fta)
  (map K fta))
```

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

;; unitFT : A -> F(T)(A)
;; bindFT : F(T)(A) x (A -> F(T)(B)) -> F(T)(B)

(define (unitFT a)
  (lambda (env) (unitT a)))

(define (bindFT fta f)
  (lambda (env)
    (bindT (fta env)
           (lambda (a)
             ((f a) env)))))

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

;; mapF : (S(A) -> T(A)) -> (F(S)(A) -> F(T)(A))

(define (((mapF K) fsa) env)
  (K (fsa env)))

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

;; unitF : T(A) -> F(T)(A)
;; bindF : F(T)(A) x (T(A) -> F(T)(B)) -> F(T)(B)

(define (unitF ta)
  (lambda (env) ta))

(define (bindF fta f)
  (lambda (env)
    ((f (fta env)) env)))
```

2.6.3 モナド変換子のクラス

$$F(T)(A) = Env \to T(A)$$

$$F(T)(A) = T(Env \to A)$$

```
(define (bindFT fta f)
```

```
  (bindT fta
         (lambda (env->a)
           ...)))

(define (bindFT fta f)
  (unitT
    (lambda (env)
      (bindT fta                    ; ***
             (lambda (env->a)
               (env->a env))))))
```

表 1　表 2.5 Classification の例

名称	型 $F(T)(A) =$	Classification
非決定性	$T(ListA)$	底
例外	$T(A + X)$	底
モノイド	$T(A \times X)$	底
持ち上げ 1	$T(1 \to A)$	底
持ち上げ 2	$1 \to T(A)$	頂
環境	$Env \to T(A)$	頂
ストア	$Sto \to T(A \times Sto)$	周辺

2.6.4　モナド変換子の組み合わせ

```
(compose
 environments
 stores
 continuations
 nondeterminism
 exceptions))
```

$$
\begin{aligned}
F(T)(A) = Env \quad & \to \\
Sto \quad & \to \\
& (A \times Sto \to List(Ans + Err)) \to \\
& List(Ans + Err))
\end{aligned}
$$

訳注

訳注 1) ここはおかしい。マックレーンの主張するモノイドのアナロジーとしてのモナドの話と著者の例示しているものはズレている。

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