（翻訳途中）意味論的レゴ
(Semantic Lego)

デビッド エスピノーザ (著)
David Espinosa

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

iHdkz(訳)
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。

第 1 章

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枠組みである。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。理解することも、特にその力を考慮するにあたっては、難しい理論ではない。

ただ残念ながら、それら記述にはモジュール性が欠如しているため、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。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（store）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ある。この論文は、いくつかの重要な貢献をする：

- 我々は、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。
- 我々は、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。
- 我々は、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。
- 我々は、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、Semantic Lego、を提示し、幾つかの例を与える。

[*1] レゴ（LEGO) は登録商標です。

この論文は幾つかの重要な結果を持つ。

- 我々は、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。
- 我々は、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。

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。

純粋な関数型言語のための単純なインタプリタは図１．１-１．３で表されている。その典型的な使用は、

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

である。

このインタプリタはリストの形式の具体的な表現を構文解析する。つまり、インタプリタは妥当なプログラムであるリストの部分集合を認識する。構文解析は意味論を必要とすることが少しある、そのため、図１．４で示されるような型構築子を用いつつ、我々は抽象的な構文を渡す。そうすると、同様のプログラムは以下

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


=> 81
```

のようになり、より構文が明示的になる（可読性は下がるが）。

compute 手続きと協力はするものの、これら型構築子は抽象データ型（ADT）としてインタプリタを記述する。なお、その用法表記は図１．５に示されている。もちろん、その用法表記はインタプリタの振舞いをただ部分的に特徴づけているだけである。より完全な振舞いを記述する最も容易な方法は、図１．1-１．４で与えられているような、実装の「モデル」を提供することである。

もう我々はインタプリタのインターフェースを特徴づけたのだから、我々はより単純な実装が存在するのかどうかについて尋ねることができる。現実に、図１．６はそれが存在することを示している。この図においては、我々はカリー化された関数を定義するために Scheme の構文を用いている、そのため、

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

は

```
(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)))
```

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))
```

図 1.2 環境 ADT

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

に展開される。

すると、構文的な型構築子と選択子は完全に消えてしまうことがわかる。その新しい実装は今までのものより短く、またオリジナルの意味論的内容を残している。我々は図 1.6 を言語 ADT の表示的実装と呼ぶ。なぜなら、構文よりもむしろ意味によって表現を記述しているからである。以下の等式

$$Den = Env \to Val$$

を言語の基本的意味と呼ぶ。視点が変更したことを反映させるため Exp の代わりに Den と書く。しかし ADT の意味論は以前のものと同じままである。

これら優位性にもかかわらず、このスタイルでインタプリタを記述する著者はほとんどいない。ひょっとしたら多くの言語は（抽象データ型よりも）具体的なデータ型を使用することを推進しているからかもしれない。例えば、Scheme はリストを強調する一方で、Miranda や Haskell は自由代数的なデータ型（和 (sum) や積 (product)）を強調する。ADT のより良い支援を受けている言語のプログラマーはより楽にこのスタイルに到達するかもしれない、第一級の関数（first class functions）もまた必要となるが。

表示的スタイルは意味論とは合成的（compositional）であると明確に示している。ここで合成的であるとはつまり表現の意味とはその表現の直接の部分表現の意味から組み合わされたものであるということである。そのオリジナルの実装は

```
(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)
```

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))
```

図 1.4 構築子（constructor；コンストラクタ

`compute`	$: Exp$	$\rightarrow Val$
`%num`	$: Val$	$\rightarrow Exp$
`%var`	$: Name$	$\rightarrow Exp$
`%lambda`	$: Name \times Exp$	$\rightarrow Exp$
`%call`	$: Exp \times Exp$	$\rightarrow Exp$
`%if`	$: Exp \times Exp \times Exp$	$\rightarrow Exp$
`%+`	$: Exp \times Exp$	$\rightarrow Exp$
`%*`	$: Exp \times Exp$	$\rightarrow Exp$

図 1.5 インタプリタのインターフェース

```
;; 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)))
```

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))))))
```

図 1.7 構文から意味論への写像

表現の意味がその部分表現の構文に依存できる可能性を排除するものではない。図 1.1-14 と図 1.6 は同様のインターフェース（図 1.5）の、全く異なる基本の型を用いた実装である。前者は表現を用いる、一方で後者は表示を用いる。既に図 1.7 で示したように、我々は表現から表示への写像、つまり構文から意味論への写像を定義することができる。例えば、`(* 2 3)` は `(%* (%num 2) (%num 3))` となる。

インタプリタの表示的アプローチは [GTWW77] から始まる。この論文は、表現の実装は ADT インターフェースの実装の圏の始対象であることを示した（2.1.4 節参照）。一つ重大なことはすべての構文は同型であり、そしてすなわち、数学的な視点からは構文は問題ではないとわかった点である。

ADT としての言語の表現は、[ASS85] または [Wad92] にさえ反して、「メタ言語的」抽象化とデータ抽象化との間に実質的な違いは存在しないということを示した。新しい構文（抽象的な構文でさえ）は新しい言語にとっては必要ではない。本質的に、すべての ADT は新しい言語を形成するし、逆もまた同様である。もちろん我々は構文なしの言語を手に入れることはできない、実際には我々は Scheme の構文を再利用する。例えば、

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

という表現は（直接的に）Scheme において意味を持つし、（`compute` 命令を用いる）解釈される言語においても意味を持つ。拡張可能な構文解析機能があるならば、我々は解釈される言語をより可読しやすいものにすることができるかもしれない。最後に、我々は表示的スタイルが Scheme 以外の言語（例えば、C 言語）においてもうまくいくということを見る。

1.2 モノリシックインタプリタ

この節では、普通のモノリシックなインタプリタの書き方を調査し、それが部品として扱えない（モジュラー (modular) ではないということを示す。モノリシック (monolithic) とは、プログラムがもとのコードの形ではモジュールの形に分割されていないということを意味する。非モジュラー (non-modular) とは、局所的な概念変更が大域的なコードの変更を要求するものを言う。すなわち、モノリシックは構文的な特徴であり、一方で非モジュラーとは意味論的な特徴である。

非モジュラーな例を見るために、我々は上で示した言語をストア (Store) 機能を追加するために拡張する。我々は三つの新しい演算子を加える。

$$\begin{array}{ll} \texttt{\%begin} & : Exp \times Exp \quad \rightarrow Exp \\ \texttt{\%fetch} & : Loc \qquad\quad \rightarrow Exp \\ \texttt{\%store} & : Loc \times Exp \quad \rightarrow Exp \end{array}$$

これら演算子の直感的な意味は `%begin` は順序付けられた二つの表現をストアに結びつけ、`%fetch` はそのストアから値を読み出し、そして `%store` はストアの中に値を書き込む。我々は `%let` を用いて `%begin` を定義できるかもしれない、しかしそれらは同じ ADT に属する演算子であるので、それらに同一の状態を与えるものである方が良い。

図 1.8 と 1.9 は拡張された言語のモノリシックで表示的な実装を示している。図 1.10 で示されているストア ADT は環境 ADT とほとんど同一のものである。その基本的意味論の直感的な意味

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

とは、表現は環境とストアに関連した解釈をされるということである。例えば、`(%fetch 'a)` を評価するためには、記憶場所 `a` にストアされているものを知る必要がある。値を返すことに加えて、表示内容は更新されたストアも返す。

我々は新しい言語の構成を三つだけ付け加えたが、それ以外の構成に関する実装は抜本的に変化する。例えば、数字はストアするものを何も持たないが、無理やり

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

と、

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

の代わりに書くことができる。

このように、我々は非モジュラーな例を持つ。つまり、概念的な局所的な変更はそのコードに従った大域的な変更を要請する。

1.3 モジュラーインタプリタ

モジュラーなプログラムはモノリシックなプログラムよりも以下のようにいくつか優位な点がある。

- 理解することが容易
- 理由をつけて説明することが容易
- 拡張と修正が容易

この節では、二つのモジュラーインタプリタを記述する。我々はこの前の節で構成されたインタプリタを調査することから始める。基本的意味論は

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

である。この型において、我々は以下のように三つの全く異なった「レベル」をはっきりと分ける。

$$\begin{aligned} E &= Env \to Sto \to Val \times Sto \\ S &= Sto \to Val \times Sto \\ V &= Val \end{aligned}$$

```
;; Den  = Env → Sto → Val × Sto
;; Proc = Val → Sto → Val × 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)))))
```

図 1.8 モノリシックインタプリタ 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)))
```

図 1.9 モノリシックインタプリタ 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))
```

図 1.10 ストア ADT

モジュラリティはあり得るものである、なぜならば多くの言語の構成は単一のレベル上で主に操作をしているから（レベル分けをすれば実現可能なもの）である。例えば、`%var` は環境（environment）を操作し、`%+` は値を操作し、そして `%store` はストアを操作する。

モジュラーインタプリタを組み立てるための二つの手法が存在し、我々は家の建築のアナロジーによってその手法を記述する。両方の手法ともに、一つのフロアを一度に組み立て、それを底として組み立て始める。しかしながら、第一の方法は、我々はそれぞれのフロアが完成した後に自分たちの所有物（じゅうたん、家具、陶磁器、絵画、本）を引越しする。第二の方法は、引越しをする前に家が完成するのを待つ。第二の方法がうまくいくということは驚くべきことではない。

第一の方法において、我々は値のレベル（values level）と `%+` のような構成概念で始める。次に我々はストアのレベル（stores level）と `%fetch` のようなより多くの構成概念を追加する。さらに我々は値のレベルの構成概念をストアのレベルに持ち上げる（これは興味深い部分である）。それから我々は環境レベル（environments level）と `%call` のような構成概念を加える。そして我々は値とストアの構成概念も環境レベルに持ち上げる。

第二の方法において、我々は値と関数をすべてのレベルのペアの間で持ち上げるために演算子を定義する。例えば、`unitVE` は値を環境へ持ち上げる。我々はこのような演算子を段階的に定義することができるが、一斉にそれらを定義するのは容易である。それから我々は各々の言語の構成を一度に定義するために、いくつかのレベルを通じて持ち上げることなく、それら演算子を用いる。

どちらのインタプリタにおいても、我々はレベルのペアを関連付けるためにモナドを用いる。モナドとは、型構築子と二つの多相演算子からなる三つ組（T, `unit`, `bind`）である。

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

それら演算子は 2.2 節で議論するようにいくつかの恒等式に従うことを要請される。二つの型 A と B は、$B = T(A)$ であるならばモナド（T, `unit`, `bind`）によって関係づけられる。`unit` は値を A から B へ持ち上げ、`bind` は $A \to B$ の型の関数を型 $B \to B$ の型の関数へ持ち上げる。

持ち上げが一体何を意味するのか調べてみることにしよう。我々は関数 `square` : $Num \to Num$ を持っているとし、これから、リストの各々の数字を平方する `square-list` : $List(Num) \to List(Num)$

関数を定義したいものとする。リストモナド

```
;;; T(A) = List(A)
(define (unit a)
  (list a))

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

が与えられたとすると、我々は square-list を

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

と定義することができる。我々は標準的な Scheme の map 関数を用いてこの持ち上げを実現することができる。しかし、2.2.3 節で示されるようにモナドは関数を持ち上げることができるが、map やその一般化したものは持ち上げることができない。我々は節 3.1 において持ち上げの形式的な定義を示す。

1.3.1 インタプリタを持ち上げる（lifting）

この節では持ち上げを用いたモジュラーなインタプリタの組み立てたもの提示する。階層化 (stratification) はより単純でより強力であるので、初めて読む際は次の節はスキップする方が良いかもしれない。

そのインタプリタは図 1.11-1.14 で示されたものである。最初の図は持ち上げ演算子の集まりを示している。これら演算子はレベル A と B を関連付けるモナドと A 上で定義された関数を受けつける。それらは B 上で定義された関数を返す。関数は持ち上げのプロセスには関わらないパラメータ型（X で表す）も受け付けるかもしれない。それらパラメータは常に実際の引数の前に来る。lift-pN-aM 演算子は N パラメータと M 引数の関数を持ち上げる。例えば、lift-p1-a2 は関数

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

を関数

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

に持ち上げる。持ち上げ演算子は引数として取る関数がすべて型 A の値を返すことを想定している。

二つ目の図は値のレベルとそのレベルで定義された構成概念を示している。三つ目の図では持ち上げ演算子とストアモナドをストアのレベルにこれら演算子を持ち上げるために使用している。四つ目の図では環境に対して同様のことを行っている。Scheme プログラム（本質的には、値呼び出しのラムダ計算）に関して解釈し直した適切な法則を用いることで、我々はその最後の構成概念は図 1.8 のモノリシックな定義と操作的に等価であることを示すことができる。

このインタプリタのためのコードはやや長いにもかかわらず、かなりモジュラーである。例えば、それぞれ以下のようなことを意味することとなる

- %num、%+、そして %*は環境とストアを必要としない。
- %fetch と %store は環境を必要としない、そして
- %var、%lambda、そして %call はストアを必要としない。

我々は unit と bind を用いた標準的（canonical）な方法おける持ち上げ演算子を用いることでモジュラリティを得る。標準的（canonical）とは、恒等的な型を持つ演算子は恒等的な持ち上げを持つということである。例外は %if であり、これは特別な扱いを必要とする。この場合においてさえ、%if の持ち上げはすべてのレベルに対して同じ形である。

より深刻なモジュラリティの欠如は、%var、%lambda そして %call を定義するときに発生する。ここで我々は unitS と bindS を用いる、これらはストアレベルのためだけに設計されたものである。また、我々は環境は値レベルからの値を含むと仮定する。環境の構成概念が多数のレベルと相互作用することから、我々はそれらを非局所的（non-local）と呼ぶ。

```
(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))))
```

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))
```

図 1.12 値レベル

1.3.2 階層化（stratified）インタプリタ

第二のインタプリタは最初のものよりもよりシンプルである。我々は言語のすべての構成概念を五つの演算子を用いて定義する。それら５つの演算子は次のようにペアとなっているレベルに関係している。

$$\begin{aligned}
&\texttt{unitSE} : S \to E \\
&\texttt{unitVS} : V \to S \\
&\texttt{unitVE} : V \to E \\
&\texttt{bindSE} : E \times (S \to E) \to E \\
&\texttt{bindVE} : E \times (V \to E) \to E
\end{aligned}$$

なお、bindVS は必要ないので除いた。これら演算子は「計算」の抽象データ型を形作る。その抽象データ型からは普通の言語 ADT を組み立てることができる。我々は他にもそれを「表示」の ADT とも呼ぶが、モッジ（Moggi）のモナドに関する仕事は、我々が彼の意図するものをやや変えはしたものの、「計算」についての先例となっている。

ピーター・モーゼス（Peter Mosses）は、一つの言語の基本的意味論を抽象化している ADT を記述した最初の著者である [Mos92]。これの何が新しいのかといえば、それは階層化（stratification）であり、階層化は次に示すいくつかの優位な点を持つ。

- 我々はもっと自然に非局所的な言語の構成概念を定義することができる。

```
;;; 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))))
```

図 1.13 ストアレベル

```
;;; 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))))))
```

図 1.14 環境レベル

- 我々は計算と言語の構成概念を階層化が提供する構造を通して理解することができる。
- 我々は階層化された計算 ADT を自動的にコンポーネントモジュールから組み立てることができる。

我々は第 4 章においてこのアプローチに戻る。

図 1.15 はこの意味論における計算 ADT を示しており、図 1.16 と 1.17 は言語 ADT をそれから組み立てているさまを表している。もう一度、このインタプリタはオリジナルのモノリシックなインタプリタと観察の上では等価である。それはまたやや非モジュラーである。具体的には、すべての構成概念は以下を仮定する

- E の上にはレベルは存在しない。
- レベル S は E のすぐ下にある。
- レベル V は S のすぐ下にある。

節 4.3 は、自動的に生成されるインタプリタにおけるこれらモジュラリティ問題をそれぞれのレベルにいくつかの名前を与えることで解決する。

1.4 例

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

- 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）を用いてモデル化された一つの並列言語

を検討する。

```
;; 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))))
```

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))))
```

図 1.16 部品的インタプリタ パート 1

```
(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)))))))
```

図 1.17 部品的インタプリタ パート 2

1.4.1 scheme のようなある言語

我々は環境、値呼び出し手続き、ストア、継続、非決定性、そして例外処理を備えた言語のインタプリタを構築する。図 1.18 は完全な言語の仕様、基本的意味論、そして二つの例を示している。SL は自動的に基本的な意味論の記述を接頭形式で生成する。

我々は二つの段階を踏んでインタプリタを組み立てる。本質的に、SL はちょうど今しがた示した階層化インタプリタを組み立てるために用いた手動の方法を自動化する。まず初めに、意味論的モジュールのリストを引数に取る `make-computations` を用いて計算 ADT を定義する。結果を表す ADT は適切に命名された `unit` と `bind` 演算子の集まりでしかない。

次に、いくつかの言語のコンストラクトのファイルを積み上げる。これらは計算 ADT から抜き出した演算子を用いて言語 ADT を定義する。これら定義はこの前の節におけるものと似ている。構成概念は任意の適切な意味論モジュールを含んだ計算 ADT 上で定義されるかもしれない。例えば、`%amb` コンストラクトは `nondeterminism` モジュールを必要とする。普通、同一のコンストラクトの定義は、異なる計算 ADT 上で定義されたとき、異なる意味論をもたらす。

典型的なコンストラクトは `%let` である。これの（`environments` ファイルからの）ソースコードによる定義は、図 1.19 に示されている。我々は詳細にこの定義を説明するにはまだ十分に SL を記述していない、しかしその形式は明快であるはずだ。付録？？は SL の間もなく得られる各々のコンストラクトの定義が示されている。

Scheme の手続きは大抵わかりにくいものだが、MIT Scheme は抽象的な構文であるコンストラクトを具現化することを可能にする。我々は次にインライン化と β 及び η リダクションを実行することでプログラムを簡約化する。計算 ADT の演算子をインライン化することと簡約化することによって、言語のコンストラクトの表示的スタイルによる定義が自動的に生成される。

図 1.20 に示されているように具体的に明記された計算 ADT の文脈における `%let` の簡約化の結果は、全く我々が手で書いたようなものである。SL の核心は `%let` のソースコードによる定義はストアや継続について言及していないということである。けれどもそれらストアや継続はきちんと自動的に導入された。

1.4.2 非決定性と継続

この節では、非決定性と継続の間の相互作用を探し出すために SL を用いる。我々は 3 つの異なった計算 ADT を用いるが、

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

```
;; Language ADT

(load "error-exceptions" "numbers" "booleans" "numeric-predicates"
      "amb" "procedures" "environments" "stores" "while" "callcc")

;; Basic Semantics

(show-computations)

=> (-> Env
       (-> Sto
       (let A0 (* Val Sto)
         (let A1 (+ (list A0) Err)
           (-> (-> A0 A1) A1)))))

;; Sample expressions

(compute
  (%call (%lambda 'x (%+ (%var 'x) (%var 'x)))
     (%amb (%num 1) (%num 2))))

=> (2 4)     ; would be (2 3 3 4) in call-by-name

(compute
  (%begin
    (%store 'n (%amb (%num 4) (%num 5)))
    (%store 'r (%num 1))
    (%call/cc
      (%lambda 'exit
           (%while (%true)
               (%begin
             (%if (%zero? (%fetch 'n))
                  (%call (%var 'exit) (%fetch 'r))
                  (%unit))
             (%store 'r (%* (%fetch 'r) (%fetch 'n)))
             (%store 'n (%- (%fetch 'n) (%num 1)))))))))

=> (24 120)
```

図 1.18 仕様と表現の例

```
(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)))))))))))
```

図 1.19 %let ソースの定義

```
(lambda (name c1 c2)
  (lambda (env)
    (lambda (sto)
      (lambda (k)
    (((c1 env) sto)
     (lambda (a)     ; Val x Sto
       (((c2 (env-extend env name (left a))) (right a)) k)))))))
```

図 1.20 簡約化した %let 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)))))))))
```

図 1.21 %amb ソースの定義

すべての言語コンストラクトの定義は変更しないままとする。参考までに、図 1.21 は %amb のソースコードによる定義を与えている。それぞれの意味論について、我々は計算 ADT を形成するモジュール、基本的意味論、簡約版の %amb、そして評価したプログラム例を示す。

最初の意味論（図 1.22）において、%amb の部分表現は継続としての list とともに評価された。その結果は付け加えられた上で返される。その例において、list 継続は 1 を足し合わせる継続に置き換えられた、したがって、結果は 5 1 となる。

二つ目の意味論（図 1.23) において、我々は continuations を continuations2 で置き換える。これらモジュールは継続のアンサーについての演算子の取り扱いについてだけ異なる。continuations 変換子は恒等継続を渡し、その渡した演算子を結果に適用し、それから適切な方法でオリジナルな継続に適用する。continuations2 はオリジナルの継続を直接渡しその渡した演算子を結果に適用する。この意味論における例の評価は明快である。

三つ目の意味論（図 1.24）において、我々は continuations と nondeterminism モジュールを逆の順序で組み合わせる。ここで、継続は値そのものではなく値のリストを受け取る。%amb は二つのリストをとり、それらをつなぎ合わせ、その結果を継続させる。例において、捕まえた継続の呼び出しはこのプロセスを終了させ、4 を直接返す。したがって、表現は他の二つの意味論と対比してただ一つの値を持つ。ここで意味論が提供するものに関して、これはスティールのシステムが生成するものだけが該当

する [Ste94]。ついでながら、continuations を continuations2 で置き換えることは %amb を変更しないままにする。

```
;; 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)
```

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)
```

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)
```

図 1.24 %amb バージョン 3

1.4.3 パラメータ化された統合システム

この節では、John Lamping の『パラメータ化された統合システム（Unified System of Parametrization)』[Lam88] を実現するために SL を用いる。Lamping は、表現を（再帰的に）表示する変数上でパ

ラメータ化された表現を取りうる意味論を記述している。この再帰は置き換えられた項が変数を含むこととなる置き換えをモデル化している。その言語はまた変化しない環境の名前呼び出しを含んでいる。すなわち、基本的意味論は

$$Den = Env \to EEnv \to Val$$

ここで Env と $EEnv$ は $EEnv \to Val$ を含んでいる。図 1.25 は SL 言語の仕様と %evar と %elet の意味論を示している。なお、%evar と %elet の意味論は表現を作るために用いられた。***で表される線の部分は特に興味深い。図 1.26 はいくつかの例を示している。

1.4.4 再開機能（Resumption）

```
;; 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))))))
```

図 1.25 パラメータ化された統合システム

```
(compute
  (%let 'f (%* (%evar 'x) (%evar 'x))
    (%+ (%elet 'x (%num 3) (%var 'f))
        (%elet 'x (%num 4) (%var 'f)))))

;; => 25

(compute
  (%let 'g (%+ (%evar 'a) (%evar 'a))
    (%let 'f (%elet 'a (%* (%evar 'x) (%evar 'x))
               (%var 'g))
          (%elet 'x (%num 3) (%var 'f)))))
```

```
;; => 18
```

図 1.26 統合されたパラメータ化の例

```
;; Computation and language ADTs

(define computations
  (make-computations resumptions stores lists))

(load "error-values" "numbers" "booleans" "begin" "while"
      "products" "numeric-predicates" "amb" "stores" "resumptions")

;; Examples

(compute
  (%par (%num 1) (%num 2) (%num 3)))

;; => (1 2 1 3 2 3)

(compute
  (%seq
    (%store 'x (%unit))
    (%par
      (%store 'x (%pair (%num 3) (%fetch 'x)))
      (%store 'x (%pair (%num 2) (%fetch 'x)))
      (%store 'x (%pair (%num 1) (%fetch 'x))))
    (%fetch 'x)))

;; =>
;; ((pair 3 (pair 2 (pair 1 unit)))
;;  (pair 2 (pair 3 (pair 1 unit)))
;;  (pair 3 (pair 1 (pair 2 unit)))
;;  (pair 1 (pair 3 (pair 2 unit)))
;;  (pair 2 (pair 1 (pair 3 unit)))
;;  (pair 1 (pair 2 (pair 3 unit))))

(compute
  (%seq
    (%store 'x (%num 1))
    (%store 'go (%true))
    (%par
      (%store 'go (%false))
      (%while (%and (%fetch 'go)
                    (%< (%fetch 'x) (%num 7)))
              (%pause (%store 'x (%1+ (%fetch 'x))))))
    (%fetch 'x)))

;; => (2 3 4 5 6 7 7 1)
```

図 1.27 再開機能を用いた平行言語

第 2 章

モナド

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 $f : A \to B$ かつ $g : B \to C$ が二つの射であるならば、$g \cdot f : A \to C$ 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

関手 S から T への自然変換（natural transformation）とは、多相関数（polymorphic function）

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

で、以下の条件

全ての $\texttt{f} : A \to B$ に対して
$\texttt{(o sigma (mapS f)) = (o (mapT f) sigma)} : S(A) \to T(B)$

が成り立つものを言う。「sigma は map で可換である」とこの性質を覚えておくと簡単である。他の例として以下がある

$$\begin{array}{ll} \texttt{reverse} & : List(A) \to List(A) \\ \texttt{flatten} & : List(list(A)) \to List(A) \\ \texttt{list} & : A \to List(A) \\ \texttt{left} & : A \times A \to A \\ \texttt{diag} & : A \to A \times A \end{array}$$

なお、ここで list は Id 関手から $List$ 関手へ自然であり、left はペアを作る関手から Id 関手へ自然である、そして diag は Id 関手からペアを作る関手へ自然である。

圏論における用語を用いると、自然変換は対象から射への写像である。ある対象 A が与えられたとき、我々は一つの射 $\texttt{sigmaA} : S(A) \to T(A)$ を得る。言い換えれば、型を添え字とする関数の族を得るのである。上の自然性の条件は我々の射の選び方を体系付けており、我々は任意に射を選ぶことができない。これはアドホックなポリモルフィズム（多相性）というよりもむしろパラメータ付けられたポリモルフィズムをもたらしている。より詳しい情報については [Wadb] 参照。

2.1.4 始対象の性質（initiality）

圏における一つの対象は、その対象から圏の各対象へただ一つの射が存在するとき、始対象（initial）である。一つの対象が終対象（terminal）であるとは、圏の対象からその対象へただ一つの射が存在するものをいう。始対象と終対象はもしそれらが存在すれば同型を除いて一意的（unique up to isomorphism）に定まる。圏の二つの対象 A, B が同型であるとは、$g \circ f = Id_B$ かつ $f \circ g = Id_A$ となる射 $f : A \to B$ と $g : B \to A$ が存在することを言う。

例えば、集合と全域関数の圏において、空集合は始対象であり任意の一点要素からなる集合は終対象である。ここで、多くの一点要素からなる集合が存在し、それらはすべて同型であることに気づかされる。始対象の性質はこの論文において多用するところを見ることはない、ひょっとしたら圏論での**その論文の**基礎を成す概念であるかもしれないが。

2.1.5 双対性

圏 C が与えられたとき、圏の各々の射の向きと合成の順序を反対にすることによって、我々はその双対（dual） $C^o p$ を得ることができる。いうまででもないが、この操作は普通の関数型プログラミングにおけるものとは全く異なる。もし対象が圏 C において始対象であるならば、双対 $C^o p$ では終対象となり、逆もまた同様である。したがって、我々は始対象と終対象は双対概念であると呼ぶ。他のよく知られる双対概念としては積（product）/和（sum）と単射（injective）/全射（surjective）がある。

彼の素晴らしい修士論文 [Fil89] において、Filinski は、値（value）と継続（continuation）は双対であることを示した。彼の言語から直感を高めることが難しいにもかかわらず、彼の論文は多くの驚くべき洞察を多く含んでいる。

2.1.6 圏論と関数型プログラミング

数学とプログラミングは二つの異なる活動であるということを少なくとも当面の間、思い出すことは重要である。その主な問題は、現在の言語はプログラムの性質を表現したり確かめたりする自動化されたサポートを提供していないところである。

この論文においては、ある特定の方法、すなわち対象は型、射は関数というように関数型プログラミングに圏論を埋め込む。他の埋め込み方もあり得る。例えば、[RB90] を参照、そこでは対象を値として表現する。それらアプローチは他のものよりも簡単でわかりやすいものではないが、より柔軟である。

我々の選んだ埋め込み方には幾つかの問題がある。

- 現在の言語は、弱いか存在しないかまたは暗黙の型システムを持つ（節 A.2 参照）。圏論においては、ともかく任意の種類の対象からなる圏を作ることができる。
- 圏論的合成を関数の合成として表現することは簡単ではないかもしれない（もしくはそもそも可能なものか、しかし私はこれについては確認していない）。我々はまた計算可能ではない合成を表現することもできない。

この埋め込みにおける、圏論と関数型プログラミングについての明白で最も包括的な取り扱いは [Spi93] を見よ。願わくば、Spivey がこれら手書きのメモを電子的または本の形式ですぐに刊行してくれればいいのだが。多くの省略されたバージョンは [Spi89] で見られる。

2.1.7 参考文献

計算機科学のための圏論についての一般的な参考文献は [Pie91] と [BW90] である。後者は多くの例と応用を含みそしてその長さにもかかわらずわかりやすい。圏論は学ぶにあたって恐ろしく難しくはない、なぜならばその豊かで記述的な内容は読者に概念を一つづつ得られるように導いて行き、すでに理

解している他の領域における概念をそれぞれ関連づけていくからである。

圏論は抽象代数の一部であると思われているかもしれない。マックレーン（MacLane）とバーコフ（Birkhoff）の大きめの本 [MB88] は代数への素晴らしい入門である、なぜならば終わりごろに圏論が紹介されているだけでなく、圏論的洞察が初めから終わりまで用いられているからである。

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[*1] (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。

[*1] モナドはまたトリプル（triples）とも呼ばれる。

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

```
;; 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)))))
```

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))))
```

図 2.2 モナドの例 パート 2

2.3 モナド射（monad morphism）

対象の間の射はそれら対象と同じくらい重要であるという『圏論的規則』に合わせて、我々はモナドの間の射を定義する。すなわち、我々はモナドの圏とモナドの射を作る。

クライスリ圏はモナド則の開発に助けとなったことから、我々はモナド S と T の間の射はそれらクライスリ圏の間の関手 K であるとする。K は対象の恒等射として振舞う。射について、我々は

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

が以下の関手的性質

```
;; f : A →S(B)
;; g : B →S(C)


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

を満たすようにする。我々は unit、bind そして自然変換 $K : S(A) \to T(A)$ によってこの定義を再定式化し、その場合

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

となる。

モナドの間の射の一つの例はリストモナドからそれ自身への reverse 関数である。

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

リストモナドからそれ自身への自然変換であってモナドの間の射ではないものの一つの例は (lambda (l) '()) であり、これは最初の法則を満たさない。

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))
```

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)))
```

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 2.1 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 モナド変換子の合成

```
(compose
 environments
 stores
```

```
continuations
nondeterminism
exceptions))
```

$$F(T)(A) = Env \rightarrow Sto \rightarrow (A \times Sto \rightarrow List(Ans + Err)) \rightarrow List(Ans + Err))$$

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。

第 3 章

持ち上げ（lifting）

この章では重要な概念である持ち上げ（lifting）を通してインタプリタの組み立てるためにモナド変換子をどのように使えばいいかということを示す。はじめの節では持ち上げの一般的な定義を提示し、次の節では幾つかのインタプリタの組み立ての方法について記述する。

3.1 持ち上げ（lifting）

この節では、我々は持ち上げを形式化し、モナドはどのようにして単純な記号で演算を持ち上げることができるかということについて示す。

3.1.1 形式的持ち上げ

我々は関手 S によってパラメータ化された型 $t(S)$ の言語を次のように定義する。

$$\begin{aligned} t(S) = S &\quad \text{(定数)} \\ |V &\quad \text{(変数)} \\ |t \times t &\quad \text{(組)} \\ |t \to t &\quad \text{(関数)} \\ |S(t) &\quad \text{(関手)} \end{aligned}$$

この定義の形式は [LJH95] による。若干やや複雑なバージョンは [Mog89a] に見られる。それはパラメータ性についての Reynold の研究における基礎的な定義とほぼ同じでもある [Wadb]。

特定の S について $t(S)$ は、型変数を許しているので、まだ多相的（polymorphic）である。二つの関手 S , S' と自然変換 `sigma` : $S \to S'$ が与えられたとすると、`sigma` を用いた型 t の持ち上げとは、写像

$$L : t(S) \to t(S')$$

であり、以下

```
(L a)            = a                    (定数)
(L v)            = v                    (変数)
(L (pairt x y))  = (pair (L x) (L y))   (組)
((L f) (L x))    = (L (f x))            (関数)
(L s)            = (sigma (mapS L s)    (関手)
```

を満たすものである。

ここで、sigma が自然（natural）であることから

```
(sigma (mapS L s)) = (mapS' L (sigma s))
```

を満たすことに気づく。この定義は持ち上げを、一つの関数ではなく、一つの関係として特徴付ける。実際、 t と sigma が与えられれば、一つないし 0 個の多くの持ち上げが得られるかもしれない。例えば、以下の記号、関手、そして関数を固定したとしよう。すなわち、

$$\begin{array}{lcl} t(S) & = & S(A) \to A \\ S(A) & = & A \\ \texttt{(mapS f a)} & = & \texttt{(f a)} \\ \texttt{id} & : & t(S) \end{array}$$

であるする。このとき、もし関手 S' と S から　S' への自然変換 sigma を特徴付けるとすると、sigma に沿った id の持ち上げを列挙することができる。まず、

;; S'(A) = A × A

```
(define ((mapS' f) p)
  (pair (f (left p)) (f (right p))))


(define (sigma a) (pair a a))|

(f (pair a a)) = a

;; S'(A) = List(A)


(define (mapS' f l) (map f l))


(define (sigma a) (list a))

(f (list a)) = a

T(A) = List(A)
append : T(A) x T(A) → T(A)
```

```
;; F(T)(A) = Env -> T(A)
(define (unitF ta)
  (lambda (env) ta))

;; lifted-append : F(T)(A) x F(T)(A) -> F(T)(A)
(define (lifted-append fta1 fta2)
  (lambda (env)
    (append (fta1 env) (fta2 env))))

;; %var : Name -> Env -> T(A)
;; Env = Name -> A

(define (%var name)
  (lambda (env) (unitT (env-lookup env name))))
```

3.1.2 モナドと持ち上げ

そろそろ、モナドは持ち上げを定義できることを明らかにするべきだろう。例えば、二項演算 f : $A \times B \to C$ をモナド T の unit に沿って F へ持ち上げてみよう。我々は F を

```
(define (F ta tb)
  (bind ta
        (lambda (a)
          (bind tb
                (lambda (b)
                  (unit (f a b)))))))
```

と書く。

モナド則を用いることで、F は f の一つの前節に従った持ち上げであることを示すことができる。我々は、

```
(F (unit a) (unit b)) = (unit (f a b))
```

という性質を必要とする。

置き換えと第一モナド則を二度用いることで、

```
(F (unit a) (unit b))
```

```
= (bind (unit a)
        (lambda (a)
          (bind (unit b)
                (lambda (b)
                  (unit (f a b))))))

= (bind (unit b)
        (lambda (b)
          (unit (f a b))))

= (unit (f a b))
```

を得る。

モナドを用いて持ち上げ可能な記号の集まりを決めることは面白いであろう。`%callcc` のような幾つかの有用な演算子はどうやら持ち上げ可能ではないらしい。

3.2 語用論（pragmatics）

一つのモナドに適用するいくつかのモナド変換子の合成

$$(F_1 \circ \cdots \circ F_n)(T)$$

を考えよう。

ボトムアップでは、我々はモナドの列 $F_n(T), F_{n-1}(F_n(T)), ...$ を作る。トップダウンでは、モナド変換子の列 $F_1, F_1 \circ F_2, ...$ を作る。自然に、我々は幾つかのポイントで変換子の列を分割することでこれらアプローチを組み合わせることができる。すなわち左半分はトップダウン、右半分はボトムアップ、次に実際に適用するためにその二つの半分の列を組み合わせるのである。

3.2.1 ボトムアップ

3.2.2 トップダウン

トップダウンアプローチはワドラー [Wad92] を一般化した拡張可能なインタプリタのシステムを生成する。一つの列、$F_1, F_1 \circ F_2, ...,$ において、我々は F_1 をモナドによってパラメータ化された一つのインタプリタと見る。例えば、ワドラーの基本的なインタプリタは $F(T)(A) = Env \to T(A)$ である。しかしながら、我々はモナドを提供する代わりに、他のインタプリタを得るためにモナド変換子を提供する。言い換えれば、与えられたパラメータ化されたインタプリタ I とモナド変換子 F が与えられれば、

我々は他のパラメータ化されたインタプリタ $I \circ F$ を作る。もちろん、我々は演算子を正しく持ち上げることに気をつけなくてもならない。スティールはこのアプローチを探した [Ste94] が、高階型に渡すときにミスをした。

第 4 章

階層性（stratification）

第 5 章

結論

5.1 持ち上げ 対 階層性

5.2 極限

5.3 関連事項

Spivey [Spi90] は例外処理の取り扱いについて抽象化をするためにモナドを使用しているが、拡張性を伴うこれらアイディアについては結びついていない。

Moggi [Mog89b、Mog91] は『応用された（applied)』ラムダ計算を、一つのモナドとして表現された、核の部分（変数と環境）と拡張部分（他の特徴）に分離した。彼は多くの拡張を提示し、プログラムについて論証するために『計算的 (computational) ラムダ計算』を導き出した。

モッジはまた、モナド変換子は部品から複雑なモナドを組み立てることができるということも示した [Mog89a]。この重大な機能はこれまでにないものであった。しかしながら、彼の説明は難解で、わずかの研究者が彼が具体的進歩を成し得ていたことに気づいただけだった。

[Esp94] はモッジの手法を書き換えた。それら手法は、 %call/cc や %+ ようなものでさえ、多数の意味論的階層を呼び出している構成概念を簡単に扱っていないと私は理解した（非数値の例外を発生させるためである)。階層化モナド（stratified monad）はこの問題を解決し、計算 ADT と言語 ADT との間に抽象化の壁を挿入することによるモジュール性を加える。

Wadler [Wad92] は Haskell で書かれたモナディックなインタプリタを提案することによってモッジのアイディアを広めた。インタプリタのシングルトンなモナドによる拡張の限界が、この論文（を書くこと）の動機付けとなった。また、ワドラーとキング（King）は他のモナドと共に、継続（continuation）とリスト（list）を組み合わせる方法を示した [KW92]。モッジのモナド変換子の早くからの定式化にもかかわらず、彼らは『M から M L を構築すること』よりも『M と L を組み合わせること』について議論した。SL は一般的な形でモナド構築子を扱い、たった二つではない、多数のモジュールからインタプ

リタを組み立てるための完全なシステムを提示する。

Steele [Ste94] は、新しい構成概念である、擬モナド（pseudomonad）の組み合わせ方を示した、彼らは組み合わせはするものの、擬モナドはモナド変換子よりも複雑であり、より一般的なものではない。実際、擬モナドは本質的には底モナド変換子（bottom monad transformers）である。すなわちそれらは

$$F(T)(A) = T(ListA)$$
$$F(T)(A) = T(A + X)$$
$$F(T)(A) = T(A \times M)$$

ということは実現できるが、

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

ということはできない。

スティールの擬モナドは固定された合成演算子を提供することによってモナド変換子をより良いものにするという主張は、それらが同等の強力さを持っていないために、その適用に失敗している。しかしながら、スティールのモジュラーインタプリタの完全な実装はひらめきを与え続けていた、そしてここで記述される階層的アプローチは彼の擬モナドのタワーに基づいている。

Jones and Duponcheel [JD93] はモナドの組み合わせ問題に取り組んだ。彼らは

Mosses

Filinski

Cartwright and Felleisen

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))

          ))))
```

図 B.1 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)))

          ))))
```

図 B.2 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))

          ))))
```

図 B.3 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)))))

          ))))
```

図 B.4 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))

          ))))
```

図 B.5 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)))))

          ))))
```

図 B.6 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))))

          ))))
```

図 B.7 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)))))

          ))))
```

図 B.8 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))

          ))))
```

図 B.9 図 B.9 再開機能変換子

C

参考文献