# **意味論的レゴ**©（Semantic Lego）

デビッド エスピノーザ（David Espinosa）著

岩城 秀和（訳）

第2022/10/06 10:54:00版

## 凡例

Semantic Lego

David Espinosa

Columbia University

Department of Computer Science

New York, NY 10027

[~~espinosa@cs.columbia.edu~~](mailto:espinosa@cs.columbia.edu)

Draft - March 20, 1995

# 概要

**意味論レゴ**

デビッド A. エスピノーザ

　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。ただ残念なことに、その記述にはモジュール性が欠如している。つまり、概念的に独立した言語の機能的特徴が、他の機能的特徴の意味論に影響してしまっている。そこで、モジュラーな表示的意味論の理論を提案することで、我々はこの問題に取り組む。

　モーゼス（Mosses）[Mos92]に従い、我々は一つの意味論を、計算ADT（computation ADT）と言語ADT（language ADT; ADT, abstract data type: 抽象データ型）の二つの部分に分割する。計算ADTは、言語の基本的な意味論的構造を表現する。言語ADTは、文法によって記述されるような現実の言語構築関数（language constructs）を表現する。我々は計算ADTを利用しつつ言語ADTを定義する。実のところ、言語構築関数は多くの異なる計算ADT上で多相的である。

　モッジ（Moggi）[Mog89a]に従い、モナド（monad）とモナド変換子（monad transformer）を利用しながら、我々は組み合わせ可能な部分から計算ADTを組み立てる。これら技法は多くの異なる計算ADTを組み立てることができるようにし、そして、我々の言語構築関数が多相的であることから、多くの異なる言語の意味論も組み立てることもできるようにする。

　我々はSemantic Lego（SL）−schemeで記述されたモジュラーな言語構築関数の集まり− という形でこれらアイディアを自動化する。SLは組み合わせ可能な部分から自動的にインタプリタを生成する。そして、SLはプログラミング言語の設計において有用なツールである。

意味論的レゴ©（Semantic Lego） 1

概要 4

謝辞 7

第1章　はじめに 8

1.1　ADT（抽象データ型）としての言語 9

1.2　モノリシックインタプリタ（monolithic interpreter） 14

1.3　モジュラーインタプリタ（modular interpreter） 17

1.3.1　インタプリタの持ち上げ（lifting interpreter） 18

1.3.2　階層化（stratified）インタプリタ 23

1.4　例示 26

1.4.1　Schemeのような言語 26

1.4.2　非決定性と継続 28

1.4.3　パラメータ化された統合システム（unified system of parametrization） 31

1.4.4　再開機能（Resumption） 32

第２章　モナド 34

2.1　基本的な圏論 34

2.1.1　圏（category） 34

2.1.2　関手（functor） 34

2.1.3　自然変換（natural transformation） 35

2.1.4　始対象の性質（initiality） 35

2.1.5　双対性（duality） 36

2.1.6　圏論と関数型プログラミング 36

2.1.7　参考文献 36

2.2　モナド（monads） 36

2.2.1　一つ目の定式化 36

2.2.2　二つ目の定式化 37

2.2.3　解釈 41

2.3　モナド射（monad morphism） 43

2.4　組み合わせないモナド 43

2.5　組み合せたモナド 44

2.6　モナド変換子（monad transformers） 46

2.6.1　動機 46

2.6.2　形式化 48

2.6.3　モナド変換子のクラス 49

2.6.4　モナド変換子の合成 51

第３章　持ち上げ（lifting） 52

3.1　持ち上げ（lifting） 52

3.1.1　形式的な持ち上げ 52

3.1.2　モナドと持ち上げ 53

3.2　語用論（pragmatics） 54

3.2.1　ボトムアップ 54

3.2.2　トップダウン 55

第4章　階層性（stratification） 56

4.1　階層化モナド（stratified monads） 56

4.2　階層化モナド変換子（stratified monad transformers） 57

4.2.1　頂変換子（top transformers） 58

4.2.2　底変換子（bottom transformers） 58

4.2.3　周辺変換子（around transformers） 58

4.2.4　継続変換子（continuation transformers） 58

4.3　計算ADT 59

4.4　言語ADT 60

第5章　結論 62

5.1　持ち上げ 対 階層性 62

5.2　極限 62

5.3　関連事項 62

5.4　将来的事項 64

5.5　結論 64

付録A　雑記 65

A.1　なぜSchemeか 65

A.2　型についての重要な点 65

A.3　型付きの値 対 型無しの値 66

A.4　拡張可能な和と積 67

付録B　コード 68

B.1　モナド変換子の定義 68

参考文献 75

著者参考文献一覧 75

訳者あとがき 77

# 図の一覧

* 1. インタプリタ
  2. 環境ADT
  3. 述語とセレクタの表現
  4. 構築子の表現
  5. インタプリタのインターフェース
  6. 表示的〔意味論的〕実装
  7. 構文から意味論への写像
  8. モノリシックインタプリタ、パート１
  9. モノリシックインタプリタ、パート２
  10. ストアADT
  11. 持ち上げ演算子
  12. 値レベル
  13. ストアレベル
  14. 環境レベル
  15. レベル交渉演算子
  16. モジュラーインタプリタ、パート１
  17. モジュラーインタプリタ、パート２
  18. 仕様と表現の例
  19. %let ソースによる定義
  20. %let 定義の単純化
  21. %amb ソースによる定義
  22. %amb バージョン１
  23. %amb バージョン２
  24. %amb バージョン３
  25. パラメータ化に関して統一化されたシステム
  26. 統一化されたパラメータ化の例
  27. 再開機能を用いた平行言語

1. モナドの例、パート１
2. モナドの例、パート２
3. モナドは組み合わせしない
4. 環境モナド変換子
   1. 第一回答変換子
   2. 第二回答変換子
5. %end-of-input?の奇妙な定義
6. 環境変換子
7. 例外変換子
8. 継続変換子
9. ストア変換子
10. 第一持ち上げ変換子
11. 第二持ち上げ変換子
12. リスト変換子
13. モノイド変換子
14. 再開機能変換子
15. Amb
16. Reset
17. Stores
18. Output
19. While
20. Begin
21. エラー例外
22. エラー値
23. バッチ I/O
24. ブール値
25. 現在の継続呼び出し（Call with current continuation）
26. 動的スコープ手続き
27. 環境
28. 不動点
29. 不動点を用いたLetrec
30. 数字
31. 静的スコープ手続き
32. 再開機能
33. 積
34. 和
35. シフト

# 表の一覧

1. モナド型構築子
2. 存在の意味
3. モナド変換子
4. モナド変換子の分類
5. 分類の例
   1. 各々の変換子に関連付けられた名前
   2. 複雑な言語に対応したレベルと名前
   3. 手続き型
   4. モジュールと言語を構築する関数
      1. 非局所言語構築関数

# 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ろうが、それはずっと悪いものになっただろうし、とっくに悪いものだった。Maryは大学院における疑いもなく最も嬉しい結果である。

私の母であるJoanne Espinosaは２８年間に渡る偉大さがある。母に感謝する。

私は彼を１０年前から知っていたが、ジェラルド J. サスマン（Gerald J. Sussman）は常に私の刺激の一つであり続けた。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。より題材的な面では、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Salを指導教官として多少なりとも選んだ、その理由は彼がナイスガイだったからだ。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無傷で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』ということで、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はすべてお金だけであった（Albertがいたにも関わらず）。

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の四階フロアにいた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であり続けた。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。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（sine qua non）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もあり）、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、彼は英国で枝葉末節のバグの追いかけをしてしまっている。

Bill Rozas は私を本当に多くの、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んでくれた。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]は水晶のように明瞭だ。

Charles Leiserson は、私がBrownを訪れた時にそこで一番興味深い人物がいるとしてくれたことによって、一人の大学生としてMITに参加するために納得させる証明を出してくれた。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。表示的意味論を用いることで、我々は簡潔かつ曖昧さのない言語を記述し、現実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。表示的意味論は、特にその力を考察するには、理解するにあたって難しい理論ではない。

　ただ残念ながら、表示的な記述を読んだり書いたりすることは難しい、その主原因としてはモジュール性の欠如である。各言語構築関数（language construct）は、言語の基礎を形作る意味論的に組み立てられたブロックすべてとやりとりをする。例えば、もし我々がストア（store）を用いて代入（assignment）をモデル化するならば、*すべての*言語構築関数は、代入の機能そのものではなくストアとやりとりをしなければならない。このやりとりの複雑さは表示的な記述をより入り組んだ形にする。

　この論文では表示的な記述のモジュラーな書き方を提示する、なお、その書き方はSEMANTIC LEGO[[1]](#footnote-1) (SL)、Schemeで記述されたコンポーネントから組み立てられたインタプリタのプログラム、として自動化している。本質的に、SLは言語を記述するための言語である。これは、以下に示す重要な貢献を果たす。

* 抽象データ型としてのプログラミング言語の考え方の再導入をする。このスタイルで書かれたインタプリタは従来のものよりも短く明確である。
* 持ち上げ（lifting）に関するモッジの理論を簡単な用語で述べなおした。これは、今までよりももっと広くの聴衆に対してこの理論に近づくことができるようにしている。
* 依然として強力である持ち上げ（lifting）よりも単純化した階層化（stratification）に関する新しい理論を記述している。この理論はモーゼス（Mosses）の意味論代数の理論に構造とモジュール性を追加して拡張したものである。
* 我々は、持ち上げと階層化それぞれに基づいた、２つのモジュラーインタプリタの書き方を示す。
* 階層化に基づいたモジュラーな言語構築関数の集合であるSEMANTIC LEGOを提示し、さらにいくつかの例を与えている。

この論文はいくつかの重要な結果を持つ。

* 我々は言語をパーツごとに分解することで言語をより良く理解し議論し、そして教えることができる。例えば、並行主義での再開機能（resumption）のモデル化は、いくつかの単純な特徴の組み合わせとして見るまでは複雑に見える。
* 我々は新しい言語で実験することができる。SLは、高度な問題を考えるにあたっての設計者の自由を残しつつ、表示的記述に関連した記述の管理をすることができる。SLの基礎となる理論は、新しい言語の構築関数を思いつく助けにもなる。

　以下の話は、SLの力を説明するものである。三人のMITの大学院のプログラミング言語コースの教諭助手（TA）は、洗練された制御の構築関数（shift）の意味を、状態（state）があるところで、記述する必要があった。彼らは、個々に独立したときの制御と状態は理解しているけれども、その（直面している）問題の集まりを区分けする以前に、制御と状態の特徴の間で適合するやりとりを見つけることができなかった。

SLを用いることで、私は1分足らずで二つの解法を作り出した。事実、SLは、要求される構築関数ただ一つからなるものではない完全なインタプリタを作成した。また、例外処理（error）を追加すること、すなわち他の意味論的な複雑化、も容易である。SLは徹底的にテストされているので、その定義がうまく作られているかどうか試す必要はない。

　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章では階層化（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、前章までのものを評価する。付録Aにおいてはこの論文に関連する話題を取り上げる。

我々は、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、より深い背景については[Wad92]を参照せよ。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証し、モジュラーインタプリタの二つの書き方について示し、一連の例題でSLを練習する。

## 表示的意味論

この節では、SLの基盤となっている理論である表示的意味論を議論する。素晴らしい参考文献は[Gun92, Sch86]である。一般に、意味論は意味付けの研究といえる。これのゴールは、例えば、 は『x に１を足す』というように決定することである。

プログラミング言語の意味論は実際のところ数理論理学の一種である、その主たる違いは、論理学は推論（reasoning）であるのに対し、プログラミング言語は計算（computation）であるという点である。（命題かプログラムどちらかの）意味論は相互に関連する２つの部分に分割されている。証明論では、そのより構文的な部分では、推論も計算もさせる。モデル論では、そのより意味論的部分においては、実際に推論的でもあり計算的でもあることについて記述させる。

### 領域/型

第一歩はモデルを組み立てるにあたっての生の題材を記述することである。これの込み入った主題は領域理論（domain theory）と呼ばれる。しかしながら、その多くは、言語の特徴をモデリングするという我々の興味関心からは関係無いものである。筆者は大抵、領域を型と呼ぶが、その概念は全く同一のものではない。

まず、Num、Bool,そしてStringといった基本型から始める。これら基本型を型構築子（type constructor） (関数空間)、（デカルト積）、そして（分離和）を用いて組み合わせる。 はＡからＢへの関数の集合である。 は右結合（associates to the right）であるので、 は を意味する。は全ての組 の集合である。はすべての と の集合である。すなわち、常にタグを調べることでどちらの元であるか言い当てられるということを除いて、 の要素はＡのものかBのものどちらかとなる。従って、 は Ａ とは異なる。

二つの型ＡとＢが（集合論的）同型であるとは、関数 と が存在し、 となることをいう。同型を と記述することにすると、

従って、 , そして は加法、乗法、そして指数に類似している。このような構造から、これら構築子を用いて組み立てられた型は代数型と呼ばれている。また、ただ一つの元を持つ集合は全て同型（そしてほかの濃度についても同様）である。もし各々の濃度の集合を区別したならば、 と というように同一性を書くこともできる。

### 論理

論理は言語、モデルのクラス、推論規則の集合、そして各々のモデルに対する表示するものまたは意味関数によって与えられる。意味関数は言語の項をモデルの要素に写し、言語をモデルに関して存在する物に解釈する。推論規則は推論もしくは計算に用いられる。簡単な算術言語を考えよう。

Exp = Num

| Exp ＋ Exp

| Exp × Exp

(Exp)

Num = Dig | DigNum

Dig = 0 … 9

例題として を考える。モデルとしては、自然数を通常持つ。推論規則としては、同一性、結合規則、commutative、分配測を通常持つ。意味関数は表現に実際の数を送る。例えば、意味関数は に という数字を送る。推論規則は*健全（sound）*でなくてはならない。すなわち、意味を保存しなくてはならない。例えば、分配測を に適用し、 を得たならば、この全体として全く異なる表現の意味もまた にならなくてはならない。注意として、ここではただ一つのモデルだけを指定している、一方で我々の定義はモデルのクラスも許している。[Mes89]を参照せよ。これはより徹底した論理の取り扱いをしている。

興味深いことに、表現の評価には２つの方法がある。推論規則を用いた言語内部からの方法と、意味関数を用いた言語外部からの方法である。意味関数を用いる場合、意味関数を記述した言語（メタ言語）の内部で計算するための方法が必要となる。時によっては、この意味は手に入れることができる。例えば、Ｃ言語を用いて上記言語の評価機を実装したならば、その時はＣ言語の算術系を使用できる。しかしながら、時にはそういうわけにいかないこともある。手作業で算術評価をする場合、ふつう加法と乗法のアルゴリズムの構文的操作を説明するために推論規則を使用することになる。この論文においては、意味関数を用いてインタプリタを組み立てるとともに推論規則よりもモデルに着目する。

ほとんどの意味関数の定義は合成的である。つまり表現の意味が、正確な構文よりもむしろ、部分表現の意味だけに依存している。例えば、算術言語においては、 の意味は と の意味の和となる。

### 環境（environments）

より複雑なプログラミング言語の意味論は単純に上記の例をより詳細にしたものである。この節の残りでは、基本的な技術について議論する。例として、変数と束縛の構築関数 let を言語に加えることを考えよう。さて、それは（Schemeに似た表記に変えれば）次のように書くことができるだろう。

(let ((a 1) (b 2))

(+ a (\* b 3)))

プログラム全体の意味は、相変わらず数字としてもよいであろうが、ではa といった表現の意味はどうすればいいだろうか？表現の意味は環境から数字への関数として割り当てられる。ここでいう環境とは変数に値を割り当てるものである。環境は変数から値への関数または変数/値の組のリストとしてみることができる。これはどちらで考えてもよい。

従って、意味（または表示するもの）は、存在している数字の代わりに、関数

の要素である。

　変数に対する主たる推論規則は*代入（substitution）*である。これは注意深く定義されなければならず、いくつかのバリエーションをもつ、しかし、一般に、

(let ((a 1) (b 2))

(+ a (\* b 3)))

という表現は、 と同じ意味を持つと推論できる。一見単純なようだが、この推論規則が環境モデルに対して*健全（soundness）*であることを証明することでさえ、とっくに簡単なことではない。

### ストア（Stores）

文（state）をモデル化する際には次のような複雑な問題が発生する。代入文（assignment statement）と順次実行（sequencing）を持つ言語を考える。すると、次のようなプログラムを書くことができる。

a = 1

b = 2

a = a + b

return a

文の意味はストア（store）からストアへの関数である。ここで、ストア（store）というのは環境（environment）の単なる別名である。すなわち、文は束縛している変数の集合を受け取り新しい集合を返すものである。これを次のように書く。

例えば、a = a + 1の意味は、a に3を束縛しているストアをaに4を束縛しているストアに送ることである。正確には、プログラム、文、式に別々の意味関数が必要である。つまり以下を持つ

さらに、なんとかして未束縛の変数も取り扱う必要がある。それにはいくつか実現可能なアプローチが存在する。〔未束縛な変数の評価に際して〕監視可能なエラー値（テストにおけるもの）、監視可能でないエラー値（未変更のすべての演算子を通る〔評価の〕進行に際して出るだろう）を返すという形にしてもよいし、もしくは計算を直ちに中止（abort）させるという形にすることもできるだろう。

### 継続（Continuations）

計算中止（abort；アボート）演算子をモデル化する一つの方法は*継続（continuation）*を用いることである。継続は計算の残り、現在のストアを最後のストアに写像するもの、を表現する。文はストアと継続の両方を受け入れる。多くの文は（大抵の場合）新しいストアを形成し、その結果に対して継続を適用する、従って、計算を継続する。アボートするならば、文はそのストアを直接返し、継続は無視する。従って、文のモデルとしては次のようなものになる

catchやthrowのような継続を操作する演算子を持つ言語において、バックトラッキングもしくはコルーチンに適した非局所的な制御構造を組み立てることができる。この例が示すように、意味論的概念は新しく有用な言語の特徴を時々提案することができる。

### 型の重要性

大抵の場合、いったん言語とモデルを特定してしまうと、項（terms）を表示するもの（denotations）に写像するほかの方法はほとんどなくなってしまう。いったん

であると知ってしまうと（ここで、環境は変数を数字に写像するものである）、どのように変数参照を定義すればいいか明確である。従って、経験豊富な意味論者は、言語とモデル、そして言語がその意味論を再構築するためにどのように振る舞うべきかという一般的な指示だけを必要とする。モデルだけで、興味深い情報のほとんどが伝わる。これが型は意味論を記述するための良いツールであることの理由である。

　モデルはなぜ言語構築関数の意味論をそれほどまでに完璧に決定するのであろうか？操作する値に関して、言語構築関数は一般的（general）かつ統一的（uniform）である。例えば、関数呼び出しは全ての型に対して同じように動作する。この概念はReynoldsによって形式化され、パラメトリックポリモルフィズム（parametric polymorphism）と呼ばれる。

　上記意味論において、変数についての唯一実用的な意味は、環境から変数の値を調べて見つけ出すこと、ということであることは明らかである。それ以外の方法で数字を得ることは可能であろうか？やるとすれば、数字を作り出さなければならない、ただしそれは一般的ではないだろう。この題材のよい参考文献は[Wad89]である。そこでは多相関数がその型からだけ導かれる恒等式に従うというReynoldsの結果を記述している。例えば、関数 は、任意の と に対して恒等式

(f (map g l)) = (map g (f l))

に従わなければならない。

モデルは言語の意味を強く制約することから、我々は適合するモデルを探すということから常に始める。さらに、インタプリタのモジュール理論がモデルのモジュール理論から始まることは、驚くことではない。

## 1.2　ADT（抽象データ型）としての言語

我々はアベルソン（Abelson）とサスマン（Sussman）の[ASS85]のスタイルの単純なインタプリタから始めて、できる限り多くの構文上の問題を除きながら、その本質的な部分からなるものに簡約化する。このアプローチの仕方は、『メタ言語的抽象化（metalinguistic abstraction）』が普通の抽象化とは違いがないことを示す。言い換えれば、目的言語でプログラムするために、実装言語から『外に出る（go outside）』必要はない。このやり方はまたインタプリタを短くする合理化された書き方を提供し、よりはっきりと構文と意味を分離させる。

図1.1-1.3は、純粋な関数型言語のための単純なインタプリタを表している。その代表的な使い方は、

(compute '((lambda x (\* x x)) 9))

=> 81

である。

このインタプリタはリスト形式の具体的な構文を分節（parse）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のである。分節は意味論を伴って行わなければならないことがほとんどないようになっているので、図1.4で表す構築子（constructor）を用いて、抽象的な構文に渡す形になっている。そうすると、同じプログラムは、（可読性は悪くなるが）より明確になった

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

=> 81

として読める。

compute手続き従って評価されるこれら構築子は、抽象データ型（ADT）としてインタプリタを記述するものである。なお、その抽象データ型のインターフェースの表記（signature）は図1.5に示している。もちろん、その表記はインタプリタの振る舞いを部分的に特定（specifies）しただけである。もっと完全な振る舞いを記述する最も簡単な方法は、図1.1-図1.4で示すような〔インタプリタの〕「モデル」の実装を提供することである。

いまやインタプリタのインターフェースの特記したのだから、よりシンプルな実装があるかどうかについて問うことができる。実際、図1.6はそういったものが存在することを示している。この図においては、カリー化された関数の定義に関するSchemeの構文を用いるので、

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

は

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

に展開される。

(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)

(env-extend 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 (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

　そこで、構文的な構築子とセレクター（selector）は全部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、依然としてオリジナルの意味論的内容を保存している。我々は図1.6を言語ADTの*表示的*実装（*denotational* implementation）と呼ぶ。なぜならば、構文よりもむしろ表示するもの（denotation）によって表現を表しているからである。我々は、

という等式を言語の*基本的意味（basic semantics）*と呼ぶ。視点が変更したことを反映させるため の代わりに と書くが、ADTの意味論は以前と同じである。

これら優位性にもかかわらず、このスタイル〔まず抽象データ型で記述しその後に表示的記述を与えるスタイル〕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勧め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は代数的データ型（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、第一級関数（first class functions）もまた必要となるが、より楽にこのスタイルに到達するかもしれない。

　表示的スタイルは明示的に、意味論は*合成的（compositional）*なものであるということを示す。ここで、合成的であるとは、いうなれば、ある表現の意味とは、その表現の直接の部分表現の意味を組み合わせたものであるということである。ただこれは、もともとの実装が、部分表現の構文に依存している表現の意味というものがありうる可能性を排除するものではない。

(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：述語とセレクターの表現

(define (%num x) x)

(define (%var name) name)

(define (%lambda name exp) (list 'lambda var exp))

(define (%call

(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）の表現

compute

%num

%var

%lambda

%call

%if

%+

%\*

図1.5：インタプリタのインターフェース

図1.1−14と図1.6は、（図1.5で示される）同じインターフェースをかなり違う基本型を用いて実装している。前者は表現（expression）を用いており、もう一方は表示するもの（denotation）を用いている。図1.7で示されているように、我々は表現から表示するもの、すなわち、構文から意味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の創案者は[GTWW77]である。この論文は、表現の実装とは、ADTインターフェースの実装の圏の始対象（initial）であることを示した（2.1.4節参照）。一つの結論は、すべての構文は同型（isomorphic）であり、それはすなわち、数学的な観点からは構文は問題とはならないということである。

ADTとしての言語の表現方法は、[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。新しい構文は（抽象的な構文でさえ）新しい言語には必要ではない。本質的に、どのADTも新しい言語を作り出すし、逆もまた同様なのである。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、事実として、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Schemeにおいて（直接的に）意味を持つし、（compute 命令を用いて）解釈される言語においても意味を持つ。拡張できる構文解析機（parser）があるならば、我々はより可読のしやすい解釈される言語を作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以外の言語で使うことができることを見る。もう一頑張りすれば、C言語で言語ADTを実装することができるだろう、その結果は同様に使用できるというものになるであろう。

;; 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構文から意味への写像

## 1.3　モノリシックインタプリタ（monolithic interpreter）

この節では、我々は普通のモノリシックなインタプリタの書き方を調べて、それがモジュラーではないことを示す。*モノリシック（monolithic）*とは、プログラムがテキストとしてモジュール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、局所的な概念の変更が大域的なコードの変更を要請するものを言う。すなわち、モノリシックは構文的な特徴であり、他方、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、上で示した言語を拡張し、ストア（Store）機能を含むものにする。ここで、以下の三つの新しい演算子を加える。

これら三つの演算子の直観的な意味としては、 %begin は順序付けられた二つの表現をストアに結びつける、%fetch はそのストアから値を読み出す、そして %store はストアの中に値を書き込む、というものである。%let を用いて %begin を定義できるかもしれないが、二つは同じADTの演算子であるので、筆者は、どちらも同じ状態を与えるものとしたい。

　図1.8と1.9は拡張された言語のモノリシックな表示的実装を示している。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。以下の基本的意味論

の直観的な意味とは、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、表示するものは更新されたストアも返す。

たとえ新しい言語構築関数を三つ付け加えただけでも、それ以外の構成についての実装は抜本的に変化する。具体的には、数字はストアを伴うことを何も持たないが、無理やり

(define ((%num n) env)

n)

を置き換えて

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

(pair n sto))

と書くことになる。

このように、非モジュラーな実例を手に入れた。つまり、概念的な局所的な変更は、テキスト的に大域的な変更を要請することになるのである。

;; 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（その１）

(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（その２）

(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

## 1.4　モジュラーインタプリタ（modular interpreter）

モノリシックなプログラムよりもモジュラーなプログラムは以下のいくつか優位な点を持つ。

* 理解することが容易
* 理論付けが容易
* 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。まず前節で構築したインタプリタを調べることから始める。その基本的意味論は

であった。この型について、以下のように三つの全く異なった「レベル」を区別する。

ほとんどの言語構築関数は主として単一レベル上で演算を行うことになるので、モジュラリティは可能である。例えば、 %var は環境〔レベル〕上で演算を行い、%+ は値〔レベル〕上で演算を行い、そして %store はストア〔レベル〕上で演算を行う。

　モジュラーインタプリタを組み立てる２つの方法が存在する。ここで、我々はそれを家の組み立て〔建築〕に喩える。どちらも手法においても、まずいっぺんに床を組み立てしまい、それを底にして組み立てをはじめる。しかしながら、第一の手法は、それぞれの階の床が完成した後に自分たちの所有物（絨毯、家具、陶磁器、絵画、本）の引越しをする。第二の方法は、引越しをする前に家が完成するのを待つ。第二の手法がうまくいくことに驚くことはないだろう。

　第一の手法において、値のレベルと %+ のような構築関数を用いるところから始める。次に、ストアレベルと %fetch のようなより多くの構築関数を追加する。さらに、値のレベルの構築関数をストアのレベルに持ち上げる（ここは興味深い箇所である）。それから、環境レベルと %call のような構築関数を加える。そして、値とストアの構築関数も環境レベルに持ち上げる。

　第二の手法において、すべてのレベルの組の間で値と関数を持ち上げるための演算子を定義する。例えば、unitVEは値を環境へ持ち上げる。このような演算子は段階的に定義することができるが、一度にそれらを定義する方が容易である。次に、いくつかのレベルを通して持ち上げるということが無いのであれば、ある一つの段階でそれぞれの言語構築関数を定義するためにそれら演算子を用いる。

　どちらのインタプリタについても、レベルの組を関係付けるためにモナドを用いる。ここで、*モナド（monad）*とは、型構築子と二つの多相演算子からなる三つ組 (, unit, bind) のことである。

unit

bind

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が要求される。二つの型 と は、 であれば、モナド (, unit, bind) によって関係付けられる。unit は から へ値を持ち上げ、bind は の型の関数を型 の型の関数へ持ち上げる。bind は他の型の関数を持ち上げることにも用いる。

　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。関数 が定義されており、リストの要素であるそれぞれ数を平方する 関数を定義したいものとする。ここで、以下のようにリストモナド（list monad）が与えられているとき、

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

(define (unit a)

(list a))

(define (bind tb f)

(flatten (map f tb)))

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。標準的なSchemeの map 関数を用いればこの持ち上げを行うことができるが、しかし、2.2.3節で示されるように、モナドは、map（やその一般化したもの）では持ち上げることができない関数を持ち上げることができる。持ち上げの形式的な定義については節3.1において示す。

### 1.4.1　インタプリタの持ち上げ（lifting interpreter）

この節では持ち上げを用いたモジュラーインタプリタの組み立てを提示する。階層化（stratification）はより単純かつ強力であるので、次の節は初めて読む場合は飛ばした方がいいかもしれない。

　図1.11−1.14はそのインタプリタを示している。最初の図〔図1.11〕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 と を関係付けるモナドと 上で定義された関数を〔引数として〕受け取る。返り値は 上で定義された関数である。関数は（と書かれた）パラメータ型も受け取ってもよいし、このパラメータ型は持ち上げプロセスには影響されない。パラメータ値は常に実際の引数よりも前の箇所に来るようにする。lift-pN-aM 演算子は 個のパラメータ（**p**arameters）と 個の引数（**a**rguments）からなる関数の持ち上げ（**lift**）を行う。例えば、lift-p1-a2 は関数

を関数

に持ち上げる。持ち上げ演算子は、引数として受け取る関数はすべて型 の値を返す関数であることを前提としている。

　二つ目の図〔図1.12〕は、値レベルと値レベル上で定義された言語構築関数を示している。三つ目の図〔図1.13〕は、これら〔図1.12で示される値レベルの〕演算子をストアのレベルに持ち上げるために、持ち上げ演算子とストアモナドを用いている。そこではまた、いくつかの新しい演算子も定義している。四つ目の図〔図1.14〕は、同様のことを環境に対して行っている。Schemeプログラムを理論付ける適切な法則（本質的には、値呼び出しのラムダ計算）を用いることで、その図の最後の言語構築関数は図1.8のモノリシックな定義と操作的には等価であることを示すことができる。

(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

　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、相当モジュラーである。例えば、以下は定義である

* %num 、%+ 、そして %\* は環境またはストアを必要としない、
* %fetch と %store は環境を必要とせず、そして
* %var 、%lambda 、さらに %call はストアを必要としない。

　unit と bind を用いた標準的（canonical）な方法で持ち上げ演算を行うことで、我々はモジュール性を得ている。標準的（canonical）とは、一意的（identity）な型を持つ演算子は一意的な持ち上げを持つという意味である。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。我々は unitS と bindS を用いることとする。これらはストアレベルだけに用いる予定のものである。さらに、環境は値レベル由来の値を含むと仮定する。環境の言語構築関数は多数のレベルとやりとりをすることになるから、それらは非局所的（non-local）であると言うことにする。

;;; 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

### 1.4.2　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もっと単純である。すべての言語構築関数は、レベルのペアを関係付ける以下の五つの演算子を用いて定義される。

なお、bindVS は必要ないので省いた。これら演算子は*計算（computation）*の抽象データ型〔計算ADT〕を作る。大抵の言語ADTはこの計算の抽象データ型から組み立てることができる。計算の抽象データ型は、他方で*表示するもの（denotation）*のADTとも呼ばれることもあるが、モッジ（Moggi）のモナドに関する仕事は「計算（computation）」の一つの先例となっている。ただ、我々は彼の意味しているものをいくらか変えてはいる。

　ピーター・モーゼス（Peter Mosses）は、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だった [Mos92]。ここで何が新しいのかといえば、それは階層化（stratification）であり、階層化は以下に示すいくつかの優位な点を持つ。

* 非局所的な言語構築関数をもっと自然に定義できる
* 階層化が提供する構造を通して計算概念と言語構築関数を理解できる
* 部品モジュールから自動的に階層化された計算ADTを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、そして図1.16と1.17はその計算ADTから組み立てられた言語ADTを示している。もう一度繰り返せば、このインタプリタは、注意深く観察すれば、オリジナルのモノリシックなインタプリタと等価である。しかもやや非モジュラーであり、特記すれば、すべての言語構築関数は以下を仮定している。

* Eより上のレベルは存在しない
* レベルSはEの直下にある
* レベルVはSの直下にある

節4.3では、これら自動的に生成されたインタプリタの内容におけるモジュラー性の問題を、それぞれのレベルにいくつかの名前を与えることによって解決をする。

;; 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：レベルの取り決め演算子

;; 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：モジュラーインタプリタ（その１）

## 1.5　例示

この節における例示は、SEMANTIC LEGOの入力/出力の振る舞いがどういったものかを示すものである。さらに、次の二つの章では、その背後のメカニズムの説明を行う。また次について

* 十分に特徴を持たせたSchemeらしい言語、
* 非決定性（nondeterminism）と継続（continuation）の間の三つのやりとり、
* Lampingのパラメータ化された統合システム（unified system of parametrization）、
* 再開機能（resumption）を用いてモデル化された平行言語

も考える。

### 1.5.1　Schemeのような言語

　環境（environment）、値呼び出し手続き（call-by-value procedure）、ストア（store）、継続（continuation）、非決定性（nondeterminism）、そして例外処理（error）を備えた言語のインタプリタを構築する。図1.18は言語の完全な仕様、基本的な意味論、そして二つの例示の表現を表している。SLは接頭形式の形の基本的な意味論の記述を自動的に生成する。

　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は次に示される〔図1.17〕階層化インタプリタの組み立てに使う手動の方法を自動化する。まず初めにmake-computations を用いて計算ADTを定義する。Make-computationは意味論的モジュールのリストを引数として受け取るものである。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、いくつかの言語構築関数のファイルをロードする。これらは計算ADTから抜き出した演算子を用いて言語ADTを定義している。これら定義は前節におけるものと似ている。言語構築関数は適切な意味論モジュールを含んだ任意の計算ADT上で定義されるかもしれない。例えば、%amb 構築関数は nondeterminism モジュールを要求する。一般に、異なる計算ADT上で定義されたとき、同じ言語構築関数の定義であっても異なる意味論をもたらす。

典型的な言語構築関数は %let である。これの（environments ファイルからの）大元の定義は、図1.19に示されている。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確なはずである。付録B.2はSLにおいて今得られる各々の言語構築関数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はそれらを抽象的構文として具現化することを可能にしている。次に、プログラムの簡約化を適用し、インライン化とβ及びηリダクションを実行する。計算ADTの演算子のインライン化と簡約化によって、言語構築関数の表示的スタイルによる定義が自動的に生成される。

具体的に明記された計算ADTの中身における %let の簡約化の結果は、図1.20に示されているように、正確に我々が手で書いたようなものである。SLの肝心なところは %let の大元の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ることになる。

(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：モジュラーインタプリタ（その２）

;; 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の大元の定義

(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：簡略化された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の大元の定義

### 1.5.2　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。３つの異なった計算ADTを用いるが、すべての言語構築関数の定義は変更しないものとする。参考までに、図1.21は %amb の大元の定義を与えている。それぞれの意味論について、計算ADTを形成するモジュール、基本的意味論、簡易版の %amb 、そして例示プログラムの評価したものを示す。

最初の意味論（図1.22）において、%amb の部分表現は、継続としての変数listとともに評価される。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、継続である変数list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５１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、次に適切な方法でオリジナルな継続を適用する。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確である。

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中止（abort）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は対照的にただ一つの値を持つ。ここで意味論が提供するものについて、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 の変更にはつながらない。

;; Computation ADT

(define computations

(make-computations environments continuations nondeterminism))

;; Basic semantics

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

;; 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第一版

;; Computation ADT

(define computations

(make-computations environments continuations2 nondeterminism))

;; Basic semantics

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

;; Simplified %amb

(lambda (x y)

(lambda (env)

(lambda (k)

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

;; Example

(compute

(%+ (%num 1)

(%call/cc

(%lambda 'k

(%\* (%num 10)

(%amb (%num 3) (%call (%var 'k) (%num 4))))))))

;; => (31 5)

図1.23：amb第二版

;; Computation ADT

(define computations

(make-computations environments nondeterminism continuations))

;; Basic semantics

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

;; Simplified %amb

(lambda (x y)

(lambda (env)

(lambda (k)

((x env)

(lambda (a)

((y env)

(lambda (a0)

(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第三版

### 1.5.3　パラメータ化された統合システム（unified system of parametrization）

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る表現の意味論を記述した。この再帰は置き換えられた項が変数を含むことができる置き換えをモデル化している。その言語はまた静的な環境の名前呼び出しを含んでいる。したがって、基本的意味論は

ここで と は 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 の意味論は表現を形作るために用いられるものである。\*\*\*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。

;; 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1.25unified system of parametrization

(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：unified parametrizationの例

### 1.5.4　再開機能（Resumption）

　再開機能（Resumption）は、中断可能な実行列の表示的モデルである。再開機能の意味論の基本意味論構造は、

である。ここで、Tは言語における他の特徴を提供しているものを記述する型構築子である。この型構築子は、計算が値を生成することを終了させるか、将来に継続することを仮定して計算を生成することを一時停止するかどちらかを意味している。典型的な再開機能の使い方は、ある計算を一時停止するまで実行し、それから次の計算を実行する、といった具合に複数の計算を並行して行うことである。

　[Sch86]で記述された標準的な平行意味論は

を持つ。そのため、計算はストアを受け取り、ストアを返し、複数の計算に分岐することができる。したがって、表示する物の完全な型は

となる。図1.27はこの言語についてのSLの仕様を示しており、いくつかの例に沿っている。表現は、実行可能な順序ごとに1つの値のリストとして評価される。%parの定義は付録B.2の図B.27に記してある。その拡張も示すことができるだろうが、それは特に明確なものではない。

;; 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

この章においては、まずいくつかの基本的な圏論を提示し、次にモナド、モナド間の射、モナド合成そしてモナド変換子を議論する。これは「あなたがモナドについて常に知りたいと思っていたことのすべて」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だけである。もっと詳しい情報については[BW85,Mog89a]を参照せよ。

モナドは１９９０年代の関数型プログラミングの業界で「ホットな話題」だったかもしれないが、現実の「モナドの急沸騰」は１９６０年代の圏論／代数的トポロジーの業界、そのとき彼らが初めてモナドを開発した、で起こった。私は自分自身を１９９０年代の基準で相当良い「モナドハッカー」だと考えるが、１９６０年代のモナドハッカーの一覧には載りさえしない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「モナド−って状態とかああいうものじゃないの？」という計算機科学者の質問を聞くことがほとんど無いということに気づいた。それは「代数−１＋１＝２とかああいうものじゃないの？」と質問するようなものであるのに。

## 2.1　基本的な圏論

この節においては、圏論の基本概念を定義し、圏論と関数型プログラミングの間の関係を議論し、そしていくつかの参考文献のことについて述べる。

### 2.1.1　圏（category）

圏は型付き関数の合成を抽象化する。 圏（category）は、対象（型）の集合、射（関数）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）への方向を指し示す。 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。各々の対象からそれ自身への他とは区別された恒等射が存在する。 合成は、左恒等的かつ右恒等的である恒等射について結合的でなくてはならない。

我々が使用する基本的な圏は、意味論をするか関数プログラミングをするかどうかに依ることになる。 意味論においては、適合する領域理論（domain theory）を使用する（[Gun92]を参照）。 関数型プログラミングにおいては、言語の型と関数を使用する、この論文で使用する言語はScheme[CR91]である。 Schemeは明示的には型を持っていない、そのため自分で型を想像しなくてはならない。

圏では、合成は関数適用より重要である。 関数型プログラミングでは、組み合わせ言語（combinator language）でプログラムしない限り、その重要度は逆になる。 この観点における変更は、実際上ほとんど問題にならない。なので、最も便利なものどれでも使用する。

### 2.1.2　関手（functor）

圏論では、対象のクラスを定義するときはいつでも、それらの間の適切な写像も定義する、すると、それらは圏になる。 この理由から、我々は今から圏の間の射を考える。

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（endofunction）とは、圏からその圏自身への関数である。 この論文では、自己関数とは型構築子であるとする。型構築子は、なにか他のものから型を作りだすものである。 例えば、T(A) = List(A) は好きな任意の型のリストを作る。 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。 *関手* T : C → D を関数 T 、これもまた T と呼ぶ、に C の射から D の射に移す関数 mapT で以下を満たすものを加えて定義する。

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

(mapT id) = id

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

*自己関手（endofunctor）*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、リストの普通の map 関数は T(A) = List(A) を自己関手になる。 関手とは、圏間の写像の適切なクラスとなる、なぜかといえば、それは写像の適切なクラスが恒等射と合成の構造を遵守するからである。 他の関手は組関手（pairing functor）

;; T(A) = A × A

(define ((map f) ta)

(pair (f (left ta)) (f (right ta))))

と環境関手（environment functor）、これは環境で型をパラメータ化する関手である、がある。

;; T(A) = Env -> A

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

(f (ta env)))

### 2.1.3　自然変換（natural transformation）

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

であって、

すべての に対して

(o sigma (mapS f)) = (o (mapT f) sigma)

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

ここで list は Id から List へ自然であり、left は組（pairing）を作る関手から Id へ自然である、そして diag は Id から組を作る関手への自然である。

圏論の用語では、自然変換は対象から射への写像である。対象 A が与えられたとき、一つの射 sigmaA : S(A) → T(A) を得ることになる。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。上記の自然性（naturality）の条件は、射の選び方を構造化している。つまり、我々が勝手に射を選ぶことはできない。これは、アドホックな多相性というよりもむしろパラメータ化された多相性を作り出している。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）

圏の対象が始（initial、始対象）であるとは、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を言う。対象が終（terminal、終対象）であるとは、圏の各対象からその対象へただ一つの射が存在するときをいう。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、 かつ となる射 と が存在することを言う。

例として、集合と全域関数の圏においては、空集合は始対象であり任意の一点集合は終対象となる。注意としては、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である。始対象の性質（initiality）はこの論文において多く用いているところを見ることはないだろう、もしかしたら圏論の欠かせないような基礎概念であるかもしれないが。

### 2.1.5　双対性（duality）

圏 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、その双対（dual） を作ることができる〔逆圏（opposite category）ともいうほうが普通〕。言うまでもなく、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 において始対象であるならば、双対 では終対象となり、逆もまた同様である。 このため、始対象と終対象は双対概念であると言う。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。一般に、圏論の用語だけで定式化された任意の概念の双対を作ることができる。

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。 彼の言語の直観を発展させることは難しいが、彼の論文は多くの驚くべき洞察を含んでいる。

### 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思い出すことは、少なくとも当面の間、重要である。 その主な問題としては、現在の言語がプログラムの性質を表現することや検証するための自動化支援を提供していないというところがある。

この論文では、ある特定の方法、対象は型であり射は関数であるというような対応付け、を行うことで圏論を関数型プログラミングに埋め込む。 なお、他の埋込方法も可能である、例えば[RB90]を参照せよ、そこでは対象は値として表現されている。それらアプローチは他のものよりもわかりやすいものではないが、その代わりもっと柔軟である。

我々の選択した埋込方法にはいくつかの問題がある。

* 現在の言語は、弱い、存在しない、または暗黙の型システムを持つものである（節A.2参照）。圏論においては、とに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。また計算可能ではない合成を表現することもできない。

この埋込方法おける、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。望むところとしては、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。圏論は学習するにあたっておそろしく難しいということはない、なぜならその豊かで記述的な内容が読者に概念を一つず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、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（monads）

この節では、モナドの二つの定式化を提示し、それらの背後の直観について議論する。モナドは付加的な構造を伴った関手であり、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、三つ組 (T, unit, join) であり、自己関手 T と二つの自然変換

からなる。ここで、unit は恒等関手から T へ自然であり、値は へ写像される。 例として、リストモナドのための unit は list である。unit は入射的（injective）であることを要請されない、ただし、多くの応用の場面では実用的にはそうである。join は から へ自然であり、多重の を一重の へ平らにする。リストモナドのための join は flatten である。

環境モナド 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

(o join (map unit)) = id

(o join (map join)) = (o join join)

この定式化は修正されたモノイドとしてのモナド[Mac71]を提示している（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・右単位則と結合法則である。

表2.1は意味論で用いられるいくつかの共通したモナドの型構築子を示している。次の節でそれらモナドの unit と join 演算子を（二つ目の定式化を通して）記述する。



### 2.2.2　二つ目の定式化

モナドは三つ組 (, unit, bind) としても記述できる。ここで、 は*自己関数（endofunction）*、unit は必ずしも自然性（natural）を要求されていない射の族、そして bind は射の集合の間の写像である。

unit 関数は全く前のものと同じである。map が 型の関数を受け取り 型の関数を返すのと同様、bind はより一般的に 型の関数を受け取り 型の関数を返す。unit と bind は以下に示すようにいくつかの規則に従う。

;; f

;; g

(bind unit) = id

(o (bind f) unit) = f

(o (bind g) (bind f)) = (bind (o (bind g) f))

二つのモナドの定式化（bind 対 map と join）の等価性を示すため、以下のように書く。

(define ((map f) ta) (bind ta (o unit f)))

(define (join tta) (bind tta id))

(define (bind ta f) (join ((map f) ta)))

二つの規則が等価であるということを証明するのは簡単である。

　第二の定式化は、 型の関数空間についての*クライスリ合成（Kleisli composition）* oT という用語で言い換えるならば、理解することがより簡単となる。

;;

(define ((oT g f) a)

(bind (f a) g))

規則は以下のようになる。

(oT unit f) = f

(oT f unit) = f

(oT h (oT g f)) = (oT (oT h g) f)

言い換えるならば、oT は結合的（associative）でかつ左・右単位（identity）である unit を持つ。すなわち、対象は型、射は 型の関数であり、合成として oT を持つ*クライスリ圏（Kleisli category）*を作ることができる。全ての関数型のコンビネータのシステムのように、クライスリ合成は法則を表明したり導出したりするのには有用だが、プログラムを書くには使いにくい。

　図2.1と図2.2は、表2.1に示される型構築子について unit と bind を定義している。これら図は、以下のように逆カリー化（uncurrying）と引数を逆順に受け取るようにした（合成的というよりはむしろ）適用的なバージョンのbindを用いている。

;; 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)))))

図2.1：モナドの例（その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（その２）

### 2.2.3　解釈

#### この節では、モナドのいくつかの解釈を与える。ここの最後に議論するモッジのモデルは、この論文に最も関連するものである。

#### モノイドに似たモナド

マックレーン[Mac71]はモナドを、単位元としての unit とモノイド積としての join を持つモノイドの一種として記述した。（型構築子上における）*モノイド*積は次の型を持つ。

例として、 に関する append を考えよ。*モナド*の積は次の型を持つ。

例として、flatten を考えよ。これら概念が、共通する一般化を持つということはやや奇妙である。

#### モナドは置き換え（substitution）をモデル化する

　T(A)は、以下のように変数Aの集合上の算術表現の型とする。

T(A) = A | T(A) + T(A) | T(A) \* T(A)

次に、unit は変数を表現に変換し、bind は置き換えを行う。ここで、置き換えとは、A の各々の変数に対して B 上の式 T(B) を与える写像 A → T(B) のことである。bind は、A 上の式を取り、置き換え、B 上の式を返す。

join はまた「表現上の表現」を平らにすることによって置き換えを行う。 bind は join が行うことの全てができるが、T の応用は一つより多く見る必要は決してない。

#### モナドは持ち上げをモデル化する

bind は map の一般化としてみることができる。unit の自然さ（naturality）とは次を意味する。

(unit (f a)) = ((map f) (unit a))

言い換えれば、f の関数適用は unit を適用する前でも (map f) の後でも問題は無い。(map f) を f の unit を経由した*持ち上げ（lifting）*と呼ぶ。非常に一般的な持ち上げの定義は第3章において与える。

　map は 型の関数を持ち上げるのと同様、bind はより一般的に 型の関数を持ち上げ、そして最初の二つのモナド則は上記に似た性質を保証しているということができる。bind はまた積についての関数も持ち上げることができる。例えば、リストモナドを用いて + 演算子を以下のように数字のリスト上で働くように持ち上げることができる。

(define (list+ l1 l2)

(bind l1

(lambda (n1)

(bind l2

(lambda (n2)

(unit (+ n1 n2)))))))

この定義は map を用いただけでは可能ではない。しかしながら、写像

があれば、以下のように書くことができる。

(define (list+ l1 l2)

(map (product l1 l2)

(lambda (n1\*n2)

(+ (left n1\*n2) (right n1\*n2)))))

モナド上のラムダ計算の純粋な圏論的モデル（[Mog89b]参照）は、実際は「テンソル強（tensor strength）」の性質に類似したものを、bind を使用するときでさえもproductに要求するものである。したがって、（map と比較したときの）bind の強力さのいくつかは純粋な圏論からではなくむしろSchemeから来ているということである。



#### モナドは計算をモデル化する

モッジの洞察とは、型 T(A) は 型 A の値の計算結果（a computation of a value of type A）を表現するということであった。例えば、非決定計算はただ一つの値ではなく取り得る値の集合を生成する。unit は値を、ある値を生成する（だけで他には何もしない）計算に持ち上げる。join は計算の計算を単一の計算に平ら（flatten）にする。bind は値から計算への関数を合成する。

モッジはまた次の定義を行った。計算（computation）*とは*、値であるか、またはunitの像の中にある場合、つまり、ある値vについて (unit v) と等しい場合の存在である。したがって、非決定計算がただ一つ単一の値を生成するとき、非決定計算は存在すると言うことができる。表2.2は他のモナドについての存在の意味を示している。

unit がモナドに入れる方法であるとき、外に出る方法も必要となる。ただ、計算から値への写像を持つことはできない、なぜなら１つよりも多くの値を参照したいかもしれないし、またはひょっとしたら最後にストアされている生成された計算が何なのか知りたいかもしれないからである。したがって、我々のすべてのモナドを写像

compute : T(A) × (A → Rep) → Rep

で拡張する。なお、ここで Rep はユーザーが読解するためにデザインされた普遍的な表現の型である。多くの言語において、この型は String であろう、だが Scheme ではリストや数値などを使用する。一つの代案は

compute : T(Rep) → Rep

と、map で A → Rep 型の関数を map し T(A) に渡してから、次にcompute を適用するというものである。このアプローチは直接的であるものの、計算のアナロジー（computational analogy）とは矛盾があり、それは T(Rep) は値の計算（a computation of value）というよりはむしろ表現の計算（a computation of a representation）であるからである。

この論文の残りの部分で compute については議論しない、ただし、B.1 節では我々が使用する各々のモナド変換子についてその定義を示している。

#### なぜモナドか？

モッジの「値と計算」の直観は、map と unit に加えて join がなぜ必要なのかということを全く説明していない。

現実には、我々は関数で計算全体を抽象化するために join を必要とする。

　非決定計算の演算子 amb を持つ言語において、amb は次の意味論でモデル化される。

ここで、以下のプログラムについて考える。

(define (f n)

(g (amb n (+ n 1))))

(define (g n)

(amb n (\* n 2)))

関数 g は、値を引数として受け取るものの、amb を用いているため、計算を返さなくてはならない。関数 f は g に値ではなく計算を適用させなければならない。すなわち、我々は関数

を必要とする。この関数は本質的に bind である。

## 2.3　モナド射（monad morphism）

対象間の射は対象自身と同じくらい重要であるという『圏論的な責務』を保ちつつ、我々はモナドの間の射を定義する。つまり、モナドの圏とモナド射を作る。

　クライスリ圏はモナド則の開発において助けとなったことから、モナド S と T の間の射とは、それらのクライスリ圏の間の関手 K であると考える。K は対象の恒等射として働く。射に関する、

が以下の関手的性質を満たすものを考える。

;; f : A →S(B)

;; g : B →S(C)

(mapK idS) = idT

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

この定義は unit 、bind そして次の性質を満たす自然変換 K : S(A) → T(A) によって再定式化することができる。

(K (unitS a)) = (unitT a)

(K (bindS sa f)) = (bindT (K sa) (o K f))

モナド間の射の例としてはリストモナドからそれ自身への reverse 関数がある。

(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例としては (lambda (l) '()) がある。これは最初の法則を満たさない。

## 2.4　モナドは合成しない（Monads don’t compose）

モナドが提供する多様な意味論的特徴が与えられれば、あらゆる種類の言語を構築することに問題はないように思われる。ただ、不幸なことに、モナドは合成しない。関手は合成するので、これは奇妙に思われるかもしれない。unit もまた合成する、しかし join も bind も合成はしない。

一つの例として、二つの環境モナド S, T を合成することを試みてみよう。図2.3 は個々のモナドとそれらの合成についての join を表している。たとえunitとmapを使用しても、後者は前者から定義され得ないことが調査したところ明らかになった。

ジョーンズ（Jones）とデュポンチール（Duponcheel）[JD93]は、joinST の型は含意的論理では他の演算子の型から証明することができないことを示すことで、型としての命題（the propositions as types）のアナロジーに基づいて厳格な証明を与えた。しかしながら、もしモナドを一般化されたモノイドとみなすならば、すなわち、S や T の演算子に従うものとするならば、unit が S または T を導入し、join は SS または TT を平らにし、map は配列の中であればどこでも機能することを許すことを悟る。これら演算子を用いることで STST を ST に簡約することができないのは、S/T の境界の数が決して減らないため、であることは明らかである。

;; 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合成を行わないモナド

## 2.5　モナドは合成する（Monads do compose）

前節では、モナド S, T が与えられたとき、S と T の演算子だけでは ST のモナドを作る方法が無いということを示した。この困難さの周辺には、分配則（distributive laws）、持ち上げ（lifting）そして互換性（compatibility）を通じた、三つの等価な方法がある。

分配則は、写像

は に を分配させつつ、いくつかの副条件に従う。整理すれば、S と T の任意の配列を単一の組（single pair）に簡約することができるということである。T 上の S の持ち上げとは、T-代数のアイレンバーグ=ムーア圏（Eilenberg-Moore category）の中のモナドであるが、以後議論しない。T のクライスリ圏の中で S を持ち上げることもできるかもしれない、ただし、その構成方法はあまり直接的ではないだろう。この形式の持ち上げとしてのモナド変換子（節2.6参照）を提供することもまたできるかもしれない。

最後に、ST を S と T と*両立させる*（compatible）ための条件を記述することができる。これら条件には二つの定式化が存在する。第一の定式化は、バー（Barr）によるもの[BW55]（315ページ）で、

map = mapS o mapT

(C1) unitST = unitS o unit = mapS(unit) o unitS

(C2) joinST o mapST(unitS) = mapS(joint)

(C3) joinST o mapS(unit) = joinS

(C4) joinS o mapS(joinS) = joinST o joinS

(C5) joinST o mapST(mapS(joinT)) = mapS(joinT) o joinST

図式として記述するとき、これら法則はまさに三角形と正方形であるし、次の型によって十分に記述されるものである。

(C1) : Id → ST

(C2) : STT → ST

(C3) : SST → ST

(C4) : SSTST → ST

(C5) : STSTT → ST

第二の定式化は、ベック（Beck）によるもの[Bec69]で、二つの写像

unitS :T → ST

mapS(unit) :S → ST

はモナド射（節2.3参照）とし、かつ中間単位法則（the middle unitary law）

joinST o mapS(unit o unitS) = id : ST → ST

を持つことを要請する。

分配則と持ち上げはベックによって発見された[Beck69]。ベック[Beck69]とバー[Barr85]の両者は、持ち上げ、分配則、そして互換性は等価であることを証明した。バーがなぜ異なる互換性の定式化を用いたのかははっきりしない[[2]](#footnote-2)。バーは分配則をベックと同時期に研究したが（ベックの論文参照）、ひょっとしたら二つの集合は独立して導出されるのかもしれない。ここでは条件が実際に等価であることを確認したことにとどまる。

ジョーンズとデュポンチールの論文[JD93]は全体的に分配則に関するものであるが、初期参考文献を理解し損なっている。この不作為は驚くことではない、なぜなら[BW85]の関連する節でも本の終わり頃に発生しており、そのタイトルも『モナド合成』ではなく『分配則』となっているからである。

互換性を用いると、関数というよりもむしろ関係としてモナド合成を発展させることができる。二つのモナドの合成とは、このようにそれらと互換的な全てのモナドの*集合（set）*のことである。互換性則（the compatibility laws）は、モナド T と恒等射を合成は、正確に T だけを作り出し他のモナドは作り出さないことを導き出す。私は結合法則を示すことがまだできない – さらなる条件として必要になるかもしれない。

結合法則が有効であれば、射がモナドである関係圏（relational category）を作ることができる。関係圏の概念は明白だが以前に研究されていたことは無いようである（ひょっとしたら興味をもたらすに十分な構造が欠けているのかもしれない）。圏論学者の大きなメーリングリストへの参考意見の要望に対してはほとんど返答がなかった。ある人は直接関連する返答をしてくれた。マルティン・バルジング（Martin Wirsing）（ミュンヘン）は、彼の（無名の）学生が非決定計算に関連するそのアイディアを研究しており、彼/彼女の論文が1995年夏にでも出てくるはずだと述べた。

言うまでもなく、互換性法則はモナド合成のための手法を作り出しはしない。三つのモナドが互換的であることを計算的に検証することさえできない。なぜなら関数の等価性の確認ができないためである。同様の理由から、関数の一つが何か他のものの持ち上げであることを確認することもできない（節3.1）。それにもかかわらず、互換性法則は可能な合成を制限し、それらの理由付けを可能にする。

## 2.6　モナド変換子（monad transformers）

モナドの合成は構成的にすることに失敗したことから、モナド変換を試みる。圏論的に言えば、もしモナドが射でなければ、それらを対象とする。すなわち、他のモナドからモナドを組み立てる。その基本概念の動機付けをした後、我々の構成方法を形式化する。

### 2.6.1　動機

例えば、環境モナド変換子（これは図2.4に示されている）は、任意のモナドに環境を付け加える。我々は環境モナド変換子を次のように書く。

これはモナド を引数として取り、新しいモナド を返す変換子であることを示している。型 についての の作用は上記のとおりである。

　 と のモナド変換子を恒等モナドに適用することは、図2.3で示される joinST があれば、両方の環境についてのモナドを厳密に生成する。したがって、以前作ることができなかった構成方法の例をすぐさま見ることができる。

join の変換は非常に複雑である。引数

を受け取り、joinT を使用するために型 T(T(A)) の値を作り出す。 したがって、T(Env → T(A)) の内側でmapT を用いることで Env → T(A) を T(A) に簡約することになる。

他のモナド変換子は表2.3に一覧されている、各々のモナドに対するものは表2.1と表2.2にある。より正確に言うと、Xモナド変換子を恒等モナドに適用することは、Xモナドを生成することである。ここで Xは環境、ストアなどである。付録 B.1 は各々の変換子の定義を示している。大抵の場合、変換子の合成は可換ではない、後でこの事実の創造的な使用方法を作れるだろう。

さらにモナド変換子の必要性を描写するために、非決定計算を持つ言語と、表示の型

の使い方を述べる。

この型に適合する unit は

(define (unit a)

(lambda (sto) (list (pair a sto))))

であるが、これは次

(define (unitS a)

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

(define (unitL a)

(list a))

から定義することはできないことは明らかである。

実際、合成によって Den(A) を構築する方法は無い。型構築子 T(A) = A × Sto を持つただ一つのモナドは、使い道のない

(define (unitT a)

(pair a (empty-store)))

を持つ。一方で、ストアと非決定計算モナド変換子を合成することによってこの型を容易に構築することができる。

;; 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



### 2.6.2　形式化

すでに我々はモナドの圏を構築してきたので、モナド変換子の定義の仕方についてはいくつかの選択肢がある。それは関数（対象に関するもの）や、関手（射についての map 作用を持つもの）や、前モナド（unit を伴う関手のこと）や、または単なるモナドであるかもしれない。 このアイディアを順に発展させ、モナドの圏の中のモナドはその響きよりも複雑ではないことを示す。形式的には、*モナド変換子（monad transformer）*をモナドの圏の中の前モナドとなるように定義するが、それは少なくとも高次圏のモナドでは全く無い一つ有用なモナド変換子（ストア）のケースが存在するためである。

モナド変換子の対象についての作用は、モナド T からモナド F(T) を作ることである。射についての作用は、モナド間の射 K : S → T を、関手性を満たしたような射 (mapF K) : F(S) → F(T) へ移すことである。例えば、射に関するリストモナド変換子の作用は、

;;

(define ((mapF K) fta)

(map K fta))

対象（モナド）に関する作用は、より複雑であり付録B.1に与えられている。モナド変換子 F の各々について、恒等関手から F への自然変換

unitF : T(A) → F(T)(A)

を要請する。unitF は F(T)A において T(A) から値を持ち上げることを許すものである。

T から F(T) へ関数を持ち上げるために、mapF を使うか、もしくは、可能であれば、たいていの規則に従う写像

bindF : F(T)(A) × (T(A) → F(T)(B)) → F(T)(B)

を定義することができる。 bindF は F を元にしてモナドの圏の中のモナドにする。

このような眩暈を起こすような抽象性の高所で酸素欠乏による失神をする前に、例として環境モナド変換子を考えよう。 まず、対象（モナド）についての作用を検討する。

;;

;;

;;

(define (unitFT a)

(lambda (env) (unitT a)))

(define (bindFT fta f)

(lambda (env)

(bindT (fta env)

(lambda (a)

((f a) env)))))

次に（モナド間の）射についての作用を検討する。

;;

;;

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

(K (fsa env)))

最後に、unitF と bindF を検討する。

;;

;;

;;

(define (unitF ta)

(lambda (env) ta))

(define (bindF fta f)

(lambda (env)

((f (fta env)) env)))

この最後の定義は、現実には普通の環境モナドのものと同一である（図2.1参照）。 unitF と bindF はこのように unitFT と bindFT よりもより単純である。 抽象性のより高い段階を避けたので、この定義はより簡単になったが型はより複雑になる。 本質的には、関数がより多相的であれば、つまりその引数について知っていることが少なければ、できることも少ない（節A.2参照）。 以前に述べた通り、unitF と bindF があれば、mapF は不必要である。

### 2.6.3　モナド変換子のクラス

F が型構築子を変換すると考える。モナドに対する F の作用を拡張することはできないかもしれない。例えば、我々は

を拡張することができるが、

はできない。我々は厳密な証明を与えないが、挑戦してみて何が起こるか見てほしい。



(define (bindFT fta f)

(bindT fta

(lambda (env->a)

...)))

(define (bindFT fta f)

(unitT

(lambda (env)

(bindT fta ; \*\*\*

(lambda (env->a)

(env->a env))))))

最初の試みは完全に失敗した。２つ目の試みはもっと望みがあるようにみえるが、星印がつけた行では悪い型付けがされている。 類似の引数は

と定義できるが

とはできないように指示する。

これらの観察によって自然に、*top*, *bottom*, もしくは *around* といった、表2.4に示される、モナド変換子の分類が導かれる。継続と再開機能はこの分類には合致しないが、表2.5はいままで議論した他のモナド変換子を分類している。持ち上げは二度現れている、なぜなら二つの異なる持ち上げ変換子があるからである。

bottom と top のモナド変換子における S と T はモナディックな構造を持つということには注意して欲しい、それはその変換子を恒等モナドに作用させることができるからである。around モナド変換子において、S◦T はモナディック構造を持つ。 topとaround モナド変換子のそれぞれについて我々は一つだけしか良い例をもたないが、その分類が有用であることはのちに、節4.2で、証明されるだろう。

非決定計算モナド変換子は、現実的にはリストではなくむしろ集合を使用すべきである。実際、リストモナド変換子から創出された「モナド」は、大元のモナドが可換でない限り結合法則に従わない（[JD93]参照）。しかしながら、我々のインタプリタにおいては、リストとして集合を表現しており、読者に順序と積の区別を取り壊すよう要請することになる。



### 2.6.4　モナド変換子の合成

我々は非常に複雑な型を組み立てるために変換子を使用する。例えば、環境、ストア、継続、非決定計算、そして例外処理の機能を持つ言語を検討しよう。我々は以下のように変換子を合成し

(compose

environments

stores

continuations

nondeterminism

exceptions)

これによって次の型を得る。

F(T)(A) = Env →

Sto →

(A × Sto → List( Ans + Err )) →

List( Ans + Err ))

# 第３章　持ち上げ（lifting）

この章では重要概念である持ち上げ（lifting）を通してインタプリタを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、次の節ではいく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を持つ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　形式的な持ち上げ

我々は関手 によってパラメータ化された型 の言語を次のように定義する。

t(S) ＝ S （定数）

| V （変数）

| t × t （組、ペア）

| t → t （関数）

| S(t) （関手）

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]。

全ての特定の について 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 , と自然変換 が与えられたとすると、sigma を通じた型 の持ち上げとは、写像

であり、以下

(L a) = a （定数）

(L v) = v （変数）

(L (pair x y)) = (pair (L x) (L y)) （組、ペア）

((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))

を満たすことに注意せよ。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、多くの持ち上げか、１個もしくは０個、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、

であるとする。このとき、もし関手 と から 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))

ということを試みる。このとき、leftとrightというidの二つの持ち上げがあるとする。持ち上げたfの制約は

(f (pair a a)) = a

とleftもrightも両方満たすべしということである。S’とsigmaの他の選択は

;; S’(A) = List(A)

(define (mapS’ f1) (map f1))

(define (sigma a) (list a))

というものである。この場合における持ち上げは型List(A) → Aを持つ。しかし、この型の関数は存在しない、なぜなら空リストが送られるものというものを知らないだろうからだ。間の抜けたように、それらの全ては持ち上げの制約

(f (list a)) = a

を受ける。

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) × 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 をモナド の unit に沿って へ持ち上げてみよう。我々は 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、 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 のようないく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

を考えよう。

ボトムアップでは、我々はモナドの列 を作る。トップダウンでは、モナド変換子の列 を作る。自然に、我々はいく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

ボトムアップのアプローチにおいて、我々は基本のモナド、普通の恒等モナド、そして恒等モナドへのモナド変換子の適用から始める。変換子に適用を行うと、我々は

* 変換子を通して存在する演算子を持ち上げ
* 新しい演算子を変換されたモナドの結果に追加する

ということを行う。すなわち、存在する演算子だけでなくおまけのいくつかの新しい演算子を持つモナドを得る。この場合において、モナド変換子である unitF 演算に従って持ち上げを行う。我々はモナドの圏のモナドを用いて技術的に働くが、Schemeの暗黙の多相性は持ち上げのために普通のモナドを使うことを許す。例えば、我々は普通の環境モナドを用いることで、 から へ演算子を持ち上げることができる。簡単ではあるものの、 上の演算子は持ち上げる必要もなく*既に* 上の演算子である。これらの近道は、多相的なラムダ計算ではできるものではないだろう。なおここで、我々は明示的に型を把握していなくてはならないだろう。

　我々は、モナド変換子を通して常に演算子を持ち上げることは明らかだろう。ここでモナドよりもモナド変換子を用いることはすべてのケースで当てはまるものではない。モナド変換子上で演算子を定義するために、しばしば変換するモナドを通して値と関数を持ち上げなくてはならない。例えば、 上で参照を行う値呼び出しの変数を定義するために、我々は以下のとおり unitT を用いて値を持ち上げなくてはならない。

;;

;;

(define (%var name)

(lambda (env) (unitT (env-lookup env name))))

類似したものとして、 上で %amb を定義するために、我々は を通して append を持ち上げる。ここで考察しているModulo、持ち上げが土台となっている組み立てシステム、は簡単である。

### 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、 において、我々は 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 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 とモナド変換子 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 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）

この章では、階層化モナドとその変換子を形式的に定義し、SLは実際にどのように動くかということについて記述する。

## 4.1　階層化モナド（stratified monads）

互換モナド（2.5節参照）を用いることで、第１章と第３章で議論した「モナドによって関係付けられたレベル」の概念を形式化することができる。

*レベル（level）*とは単純に型構築子（圏の自己関手）のことである。そして、モナド は、 であるとき、 を に*関係付ける*（A monad *relates* to ）ものである。ここで、圏の条件と合成の適合性に合致するように射の合成を好みに応じた任意の方法で定義することで、対象は*レベル（level）*であり射はモナドである圏を構成することができる。

　例えば、レベルと という意味論のモナドからなる圏を作ってみることにしよう。我々は以下のレベルを持つ。

これらレベルは以下のモナドによって関係付けられている。

ここで、 は環境モナド、 はストアモナド、 は「二重環境（double environment）」モナド、 は「環境／ストア（environment / store）」モナドである。また、我々はそれぞれのレベルからそのレベル自身へ向かう恒等モナド（identity monad）も持つ（ここには示していない）。 から へ向かうモナドが無いことに気づいて欲しい。合成は以下のようになる

なお、両方とも合成規則を満たす。

*階層化モナド（stratified monad）*とは、レベルといくつかの付加的性質を満たすモナドからなる圏のことである。ここで、その付加的性質とは、その圏の構造だけからは導かれない性質であり、以下のものが要請される。

* すべての図式は可換である。
* 区別されたレベル と がある。
* は恒等型構築子（identity type constructor）である。
* と はモナド によって関係付けられる。我々はまた階層化モナド自体のことも と呼ぶ。
* は極小（minimal）でなければならず、 は極大（maximal）でなくてはならない、これはすなわち、なにか を または をなにか に関係付けるモナドは存在しないことを意味する。

　すべての図式は可換であるという要請は、平行な射があれば図式を作れることから、任意の二つのレベルを関係付けるモナドが大抵の場合（無いこともあるだろうが）存在するということを意味している。また、射の構造を忘れることによって、我々は、もし を に関係付けるモナドが存在すれば、かつその場合に限り成り立つ関係である半順序 を得る。

　必要に応じて、我々は「すべての図式は可換」の条件をやめることができる。しかしながら、レベルを関係付ける多数の方法は普通は無いことから、ほとんどの意味論はその条件に従う。実際、ある言語の構築関数を定義するためにレベルの間に多数のモナドを要求するならば、その言語は「非単一的（non-uniform）」と呼ばれることもある。この制限は、我々が欲しいのは*どの*モナドであるか具体的に挙げる必要が無くなるので、SLの実装をより簡単にする。

　 と に（極小と極大よりも強い条件の）始対象と終対象であることを要請しない、それはいくつかの意味論で と に関係付けないレベルを含んでいるからである。具体的には、上の例において、 は始対象とすることはできない、 モナドが存在しないからである。多くの意味論においては、しかしながら、実際にはそれぞれ始対象と終対象となっている。

　あるレベル は、 をその に関係付けるモナドが存在するとき、（単なる自己関手というよりは）モナドである。すなわち、 はすべてのレベルがモナドであるとき、かつそのときに限り始対象である。それゆえに、その前の例において、 はモナドではない。私は未だ が終対象とすることができない何か実際の例を見つけてはいない、しかし、我々は対称性からこの条件を弱いものとした。

4.2　階層化モナド変換子（stratified monad transformers）

「圏論的な義務」に従えば、階層化モナドの圏をいま作るべきである。しかしながら、この応用にあたっては、この圏論的な構造は必要としない。

　だから、*階層化モナド変換子（stratified monad transformer）*とは、階層化モナドの*集合（set）*上の自己関数（endofunction）である。We can verify these directly for each transformer that it respects the stratified monad structure.

　実践的には、我々は階層化モナド上で働くようにするために、普通のモナド変換子の「持ち上げ」によって階層化モナド変換子を組み立てる。

我々は恒等階層化モナドから始める。恒等階層化モナドは以下の単一のレベルと単一のモナドからなる。

我々は非決定性階層化モナド変換子 に適用をする。それぞれのレベルでの恒等モナドについては省いた上で、我々は以下を得る。

は型構築子であるだけではなく、モナドすべてであることを思い出そう。ところで、我々がストア階層化モナド変換子 に適用を行うと、以下が得られる。

最後に、我々が環境階層化モナド変換子 に適用を行うと、以下が得られる。

次の二、三節で、2.6.3節で議論したモナド変換子のクラスから組み立てられた階層化モナド変換子の作用を詳しく述べる。

### 4.2.1　頂変換子（top transformers）

頂変換子（top transformer）は という形を持つ。 は新しい頂レベル を追加することによって階層化モナドのレベル上で作用する。 は に関係付けるモナドすべてに を適用することによってそのモナド上で作用する。我々はオリジナルのモナドを無くすことなしに新しいモナドを追加する。

　我々は環境変換子に関する上記の例においてこの作用を確認することができる。なお、環境変換子は を持つ。我々は環境を必要とするそれぞれのモナドを、いくつかのレベルを に関連付けるモナドを変換することで作る。逆に言えば、そのようなモナドのすべては変換されたものである。

### 4.2.2　底変換子（bottom transformers）

底変換子（bottom transformer）は という形を持つ。 は を伴うそれぞれのレベルを合成し底（bottom）に新しい恒等（identity）を追加することによって、レベル上で作用する。前と同様に、オリジナルのモナドを無くすことなしに新しいモナドを追加する。

　我々は非決定性変換子に関する上記の例においてこの作用を確認することができる。なお、非決定性変換子は を持つ。我々はリストを必要とするそれぞれのモナドを、いくつかのレベルを に関連付けるモナドを変換することで作る。逆に言えば、そのようなモナドのすべては変換されたものである。

### 4.2.3　周辺変換子（around transformers）

周辺変換子は底および頂変換子よりもいくぶんやや複雑である。異なるレベルを関係付けるモナドを出来うる限りすべて構築するために、ただ一つだけではなく*三つの*よくある普通のモナド変換子を要求する。周辺変換子が

であるならば、我々は

であることも要求する。レベル上の作用は、新しい頂レベル を追加すること、 を伴うレベル以下それぞれを合成すること、そして として新しい恒等変換（identity）を追加することである。モナド上の作用は、 を用いてモナド を変換すること、 を用いて に関係付けをするすべてのモナドを変換すること、そして を用いて に関係付けをするすべてのモナドを変換することである。前と同じく、我々は新しいモナドをその結果に、古いものは適当な所に残しつつ、追加する。

　我々は以下のように受け取ることによってストアモナド変換子を得る。

ここで、 変換子は用いていない、 変換子は以下のようにならなくてはならないだろうし、

この選択は意味をなさないことを見てきたからである。

### 4.2.4　継続変換子（continuation transformers）

継続変換子は

である。ここで、 は回答（answer）の固定された域（domain）である。 は、 としてのレベル上で働く。つまり、いったん は回答に適用され、我々がそれを適用させようとする何か他のものすべては無視する。我々は、単一の新しいレベル を追加する、ここで、 は古い頂（top）レベルである。

　 は以下のとおりモナド上で働く。 は継続変換子を用いて を変換し、 を新しい に関係付けるモナドをもたらす。 は、また、特別な「回答変換子」 を通して に関係付けをするモナド も変換し、 を新しい に関係付けるモナドをもたらす。これらのモナドは我々に回答型 のレベルにアクセスをさせてくれる。

　 の選び方は二つ存在し、図4.1と図4.2で示されている。我々が意味論

において、これら選び方のそれぞれを用いて amb を定義すると、1.4.2節で議論したものとして以下が得られる。

(define ((amb1 d1 d2) k)

(reduce append ()

(map k (append (d1 list) (d2 list)))))

(define ((amb2 d1 d2) k)

(append (d1 k) (d2 k)))

両方の定義共に納得できるものである。

(define ((unit1 a) k)

(bindT (unitM a) k))

(define ((bind1 c f) k)

(bindM (c unitT)

(lambda (a)

((f a) k))))

図4.1：最初の回答変換子

(define ((unit2 a) k)

(unitM a))

(define ((bind2 c f) k)

(bindM (c k)

(lambda (a)

((f a) k))))

図4.2：第二の回答変換子

## 4.3　計算ADT



(define computations

(make-computations

cbv-environments

stores

continuations

nondeterminism

errors))

(define %amb

(let ((unit (get-unit 'Lists 'Top))

(bind (get-bind 'Lists 'Top)))

(lambda (x y)

(bind x

(lambda (x)

(bind y

(lambda (y)

(unit (append x y)))))))))

## 4.4　言語ADT

一般的に、%amb （図1.21）や %let （図1.19）のように主に単一のレベルで働く演算子は、標準的なイディオムを用いて書くことが簡単である。%call/cc といったより複雑な演算子は、例示した意味論において例示の中の定義から抽象化したものを用いて一番よく書かれたものである。十分に複雑な意味論を用いることは、概念的に全く異なるレベルは混同されないということを保証する。表4.4は得られるモジュールと値そしてそれらが定義する言語構築関数の一覧である。名前から導かれるパーセント指標は省いた。

　SLは手続きの四つの型を含んできる。表4.3はそれら型の域（domain）と余域（codomain）のレベルを示している。我々は同じ環境モナド変換し上での手続きのすべての四つの型を、異なるバージョンの %lambda と %call を書くことで定義する。整理すると、他の手続きの型も我々は同じくらい上手に定義することができる。



# 第5章　まとめ

この章では持ち上げと階層性の比較し、これらの考えが乗り越えた障壁を記述し、この仕事と以前の調査と関係付け、そして将来的事項の提案を行う。

## 5.1　持ち上げ 対 階層性

持ち上げすることに伴う問題とは、言語構築関数とモナド変換子を直接つなぐところにある。例えば、Env → T(A) に関する参照変数を定義すれば、未束縛変数（unbound variable）の例外発生をさせる簡単な方法はない。一応、この問題を迂回するため一時しのぎ（ad-hoc）の低レベル操作のセットを定義しようと思えばできる（[LJH95]）。しかしながら、階層化は原理的な（principled）低レベル操作のセットをどのように定義すればよいか明らかにしてくれる。

違った言い方では、言語構築関数は複数の意味レベルで相互のやりとりはできないということである。例えば、＋ は数値以外の引数に対して例外を発生させつつ、値レベルと例外レベルで相互のやりとりを行う。call/cc は継続、環境そして値のレベルで相互のやりとりを行う。表5.1 は、もっと複雑な言語構築関数によって参照される意味論のレベルを示している。

持ち上げを行うことはまた古いものの持ち上げに新しい操作の創設も差し込む。これらの作用は概念的に分離しているだけではなく、変換子を使用するとき、

もう一度思い出せば、階層化は基本的な意味論から言語構築関数を分離することによる一つの解決をもたらす。それでもなお、持ち上げのアプローチはより局所的に定義された演算子を持つ抽象データ型

図PIC2 の非形式的な可換図式を用いることで二つのアプローチを比較することもできる。粗い用語で言えば、持ち上げは下方の路線に沿っている一方、階層化は上方の路線に沿っている。演算子 ai、bi と ci は言語ADTを形成する。演算子 ui と bi は計算ADTを形成する。配列の各々の演算子は、持ち上げる前のものから作られたものである。持ち上げのアプローチにおいては、計算ADTから言語ADTを構築することは簡単である一方、言語ADTを持ち上げることは

困難である。階層化アプローチにおいては、計算ADTを持ち上げることは簡単である一方、計算ADTから言語ADTを形成することは困難である。

この図式はおおよそ、持ち上げのアプローチにおいては、計算ADTの持ち上げも行うということだけを表しているものである。新しい言語構築関数を定義することは必要である。その理由は変換子が高階、すなわち、型よりも変換子を変換するからである。例として節3.2を参照せよ。



## 5.2　限界

SLは次のいくつかの意図された限界がある。

* 表示的モデルを構築するために想定されたものであることから、SLは型と構文の課題に取り組むことはない。型と構文は大抵の言語の仕様の大部分を占めているにもかかわらず。

## 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築関数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、新しい構築関数である、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

T(A+X)

ということは実現できるが、

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

**実装 （implementation）**

**拡張性（extensibility）**

**モデル（model）**

**論理（logic）**

## 5.5　結論

この論文を命令調に要約すると以下の四つとなる。

* 型で考えよ、抽象的にも具体的にもだ。
* 表示するもの（denotation）で計算せよ、表現（expression）ではするな。
* 複雑な言語は計算ADTと言語ADTに分割せよ。
* モナドとモナド変換子を用いて計算ADTを構造化せよ。

　このうち最初の２つが最も重要である。型は構造化された方法を依然としてシンプルに我々に考えさせてくれるものであり、表示するものは容易かつ直接的にインタプリタを実装させるものだからである。これら二つの考えは、異なる領域の誰かが、関数プログラミングを理解する誰かあたりに話しかけることができるようにする。

# 付録A　雑記

この付録では多様な観点からこの理論の本筋に深く関わる問題について議論する。

## A.1　なぜSchemeか

この論文をScheme[[3]](#footnote-3)[1]で書いた現実的な理由は、私が使い慣れているからである。Schemeは多くの問題（とりわけ、モジュールと抽象データ型の欠如）を抱えているものの、Schemeでプログラムを書くことには依然としてかなりの楽しさがある。しかし、この論文にとっては、Schemeには以下のいくつかの不利な点がある。

* 数学での型付き構造を表現することに失敗している
* 暗黙の多相性（polymorphism）が、多相関数と多様型（various type）の実例化（インスタンス化）とで違いが明確でない。すなわち、構造の全体のレベルが無い。
* 型付けが機械的に検証されないため、我々が正しいと考える型付けが間違っていることがある

その一方で、Schemeが有利であるのは、表現できるものに制限が無いところにある。例えば、普通のHindley-Milner型システムでは、多相的な値（polymorphic value）は第一級（first-class）ではない、なぜならば、種々の型では多層的な値というものを実例化することができないからである。 は第一級の多相性を持つが、型として型構築子を扱うことはできない、そのため、モナドは第一級となることができないのである。

SLはで型付けすることができると思われる（ は型構築子を含む)。しかしながら、階層化モナドのレベルをどのように型付けすればよいのかははっきりしない。実例としては、

（節4.3を参照）の関係にあるEnvとEnv-Resultsを型付けする必要があるだろう。

Cardelliの探索言語（Quest language）でSLを書き換えることは良い練習問題となるだろう。Cardelliの探索言語はいくつかの高度な型付けの考えだけではなく高階の多相性をも含む[Car89]が、さしあたりはSchemeのままで続ける。

一般的には、プログラミング言語のコミュニティは現在の型システムは不十分であることをはっきり理解している。しかしながら、我々はさらに進んで、型付き言語についてその基本全体を問うべきである。

## A.2　型についての重要な点

表示的意味論においては、言語構築関数の定義はその意味論の基本型によって殆ど完全に決定される、 その部分的理由は、言語構築関数が大抵は計算過程で用いられる値からは独立しているためである。 すなわち、言語構築関数の定義はパラメトリックに多相的である。例えば、任意の型Aを固定している単一の定義

を書くことができる。ここで、Env(A)は型Aの値を含む環境の型である。もし我々が としてEnv(A)を具体的に実装するならば、 型 を持つただ一つの関数が

(lambda (name) (lambda (env) (env name)))

であるということは、現実に証明可能である。

この題材に対する良い参照は[Wadb]であり、これは、多相的関数はただそれらの型だけから導かれる個性に従うというReynoldsの結果を記述している。 例えば、関数 は、任意の と任意の に対して成り立つ等式

(f (map g l)) = (map g (f l))

に従わなければならない。

　型構造は堅く言語の意味論を制限することから、ベテランの意味論者は常に適合する基本型を定義することから始める。 さらに、インタプリタのモジュラー理論は型のモジュラー理論から始めることは驚くべきことではない。

　我々はまた型を仕様として見ることもできる。実例として、ある表現が型 を持つとき、積として評価しなければならない。型システムは、現在のところ使用においては、表現をしないし、プログラムの振る舞いについて多くを具体的に記述することができない。しかしながら、我々は、仕様で検証のためのユーザー補助を要求しているようなより表現出来るシステムを想像することができる。これら検討を展開していくことで、我々は、仕様としてのああいった型は、ただの実装であるものとしての、プログラムよりも重要であることを、我々は明らかにしてゆく。 例によって、『なに？』は『どのようにして？』に先立つのである。

## A.3　型付きの値 対 型無しの値

型付き言語は多値の領域を、型ごとに一つ、持っており、さらに静的な表現型を決定している。 型無しの言語はいくつか他の値の和である単一の値の領域をもっており、さらに動的な型を決定している。 型無しの言語はまた型付きの言語よりも型の特徴がより少ない、特に手続きに関してはそうである。 実例として、Schemeは組（pair）を返す手続きと値を返す手続きを区別しない。

全ての圏論的概念がそうであるように、モナドは型付けされている。すなわち、表示するものは値についての多相性である。 例えば、我々は言語構築関数 %zero? を次のように型付けすることができる。

しかしながら、SLにおいては、単一の型無しの値の、拡張可能な和として表現されている、領域上で計算を行う。 すなわち、 %zero? は次の型を持つ

ここで、我々は の代わりに と書く。値にそれらの型をタグ付けすることは、型の取り扱いと意味論的に等価であることをあらわにしており、さらにSchemeの現に存在する型からの抽出をしている。

大抵の型システムは、型付きのインタプリタを型付けするのに十分な力強さを持っていない[Wada]。 例えば、[Wad92]におけるインタプリタは、型付き言語であるHaskellで書かれたものであっても、型無しにされる。 問題は、 x が数字であるとき次のように書きたくなるというところである。

しかしながら、(%var ‘x) の型は表現が出てくる文脈に依存する。 一般的に、表現の型は名称とその自由変数の型を含んでいる必要があるが、それはまさに普通の順次発生に基づいた型付け規則においてそうであるからである。

わずかに異なる主題のものとして、スティール[Ste94]は、モナドを用いて値の領域を拡張することを試み、私はそれに追随した[Esp94]。 しかしながら、彼の指摘により、拡張可能和を組み立てるために例外モナドを用いることは次のような振る舞いを作り出す

(compute (%+ (%num 3) (%true)))

=> true

これは、モナドは拡張可能和を組み立てるための良いツールではないことを示している。

## A.4　拡張可能な和と積

この論文において拡張可能な和と積が果たす役割は殆ど無いものの、それらは拡張可能システムを組み立て（[Esp93b]を参照）、 圏論が言語の設計をどのように支援するかということを実証し、そしてオブジェクト指向プログラミングの本質を把握することができる。 我々は、静的もしくは動的に拡張することができる以下の型を考える

（静的か動的か）どちらかということは問題にしていない。これらの検討は基本的なアイディアとは直角の位置にある。 S と P に対応して以下の拡張可能な関数がある。

ここで、s と p は反対の型を持つことに気付く。我々は S または P を拡張するのであるから、s または p も拡張することになる。 例として、s はいろんな乗り物の最大の速さを計算するものと仮定する。そのとき、s は

であり、ここで、Vehicle は拡張可能な和である。拡張可能な和は、CLOS [KBdR91]で言うところの意味の単純な生成関数（generic function）である、その一方で拡張可能な積（product）はおなじみのようには使われていない。 s と p の型は和と積の圏論的な定義から直接的な由来を持つ。他に、オブジェクト指向プログラミングの理論的取り扱いについては、[GM94]を参照せよ。

# 付録B　コード

この付録では、階層化モナド変換子のプログラムツールとして用いられるSchemeコードの一覧を与える。ただし、わかりやすくするために型変換とunit演算子の逆操作を行うプログラムのコードはあえて省いた。

## B.1　モナド変換子の定義

この節では最も共通するモナド変換子のためのコードを示す。

;; 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：環境変換子

;;; 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.2：例外変換子

;;; 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.3：継続変換子

;;; 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.4：ストア変換子

;;; 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.5：最初の持ち上げ変換子

;;; 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.6：第二の持ち上げ変換子

;;; 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.7：リスト変換子

;;; 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.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：再開機能変換子

# 参考文献

[Abr91] Samson Abramsky. Domain theory in logical form. Annals of Pure and Applied Logic, 51:1-77, 1991.

[ASS85] Harold Abelson, Gerald J. Sussman, and Julie Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambrige, MA, 1985.

[Bec69] Jon Beck. Distributive laws. In Seminar on Triples and Categorical Homology Theory, volume 80 of Lecture Notes in Mathematics, pages 119-140. Springer Verlag, 1969.

[BW85] Michael Barr and Charles Wells. Toposes, Triples and Theories. Springer Verlag, New York, NY, 1985.

[BW90] Michael Barr and Charles Wells. Category Theory for Computing Science. Prentice-Hall, 1990.

[Car89] Luca Cardelli. Typefull programming. Technical Report 45, DEC Systems Research Center, Pal Alto, CA, May 1989.

[CF91] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In Theoretical Aspects of Computer Software, Sendai, Japan, April 1994.

[CR91] Will Clinger and Jonathan Rees. Revised4 Report on Scheme. Lisp Pointers, 4(3), 1991.

[Esp93a] David Espinosa. Language extensibility via first-class interpreters and constructive modules. Available via <http://www.cs.columbia.edu>, April 1993.

[Esp93b] David Espinosa. Language features for extensible programs. Available via <http://www.cs.columbia.edu>, October 1993.

[Esp94] David Espinosa. Semantic Lego. Available via <http://www.cs.columbia.edu>, January 1994.

[Fil89] Andrzej Filinski. Declarative continuations and categorical duality. Master’s thesis, University of Copenhagen, August 1989. Available via <http://www.cs.cmu.edu:8001>.

[Fil94] Andrzej Filinski. Representing monads. In Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages, Porland, OR, January 1994.

[GM94] Carl Gunter and John Mitchell, editors. Theoretical Aspects of Object-Oriented Programming. MIT Press, Cambridge, MA, 1994.

[GTWW77] J.A.Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. Journal of the ACM, 24:68-95, 1977.

[Gun92] Carl Gunter. Semantics of Programming Languages. MIT Press, Cambridge, MA, 1992.

[JD93] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU / DCS / RR-1004, Yale University, December 1993.

[KBdR91] Gregor Kiczales, Daniel G. Bobrow, and Jim des Rivieres. The Art of Metaobject Protocol. MIT Press, Cambridge, MA, 1991.

[KW92] David King and Philip Wadler. Combining monads. In Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming, Ayr, Scotland, 1992. Springer Verlag.

[Lam88] John Lamping. A unified system of parametrization for programming languages. In Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming, pages 316-326, Snowbird, Utah, July 1988.

[LJH95] Sheng Liang, Mark Jones, and Paul Hudak. Monad transformers and modular interpreters. In Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA, January 1995.

[Mac71] Saunders MacLane. Categories for Working Mathematician. Springer Verlag, New York, NY, 1971.

[MB88] Saunders MacLane, Garret Birkhoff. Algebra. Chelsea, New York, NY, 3rd edition, 1988.

[MC93] Eugenio Moggi and Pietro Cenciarelli. A syntactic approach to modularity in denotational semantics, In Category Theory and Computer Science, Lecture Notes in Computer Science. Springer Verlag, 1993.

[Mog89a] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1989. FTP from theory.doc.ic.ac.uk .

[Mog89b] Eugenio Moggi. Computational lambda calculus and monads. In IEEE Symposium on Logic in Computer Science, pages 14-23, Asilomac, CA, June 1989.

[Mog91] Eugenio Moggi. Notions of computational and monads. Information and Computation, 93:55-92, 1991.

[Mos92] Peter D. Mosses. Action Semantics, volume 26 of Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[Pie91] Benjamin C. Pierce, Basic Category Theory for Computer Scientist, MIT Press, Cambridge, MA, 1991.

[RB90] David Rydeheard and Rob Burstall. Computational Category Theory. Prentice-Hall, New York, NY, 1990.

[Sch86] David A. Schmidt. Denotational Semantics. Allyn and Bacon, New York, NY, 1986.

[Spi89] Michael Spivey. A categorical approach to the theory of lists. In Mathematics of Program Construction, volume 375 of Lecture Notes in Computer Science, pages 399-408. Springer Verlag, 1989.

[Spi90] Michael Spivey. A functional theory of exceptions. Science of Computer Programming, 14(1):25-42, June 1990.

[Spi93] Michael Spivey. Category theory and functional programming. Technical Report PRG TR 7-93, Oxford University, June 1993.

[Ste94] Guy L. Steele Jr. Building interpreters by composing monads. In Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages, Portland, OR, January 1994.

[Wada] Philip Wadler. ワドラーと著者との個人的やりとり（Personal Communication.）。

[Wadb] Philip Wadler. Theorems for free. In Proceedings of the ??th Annual ACM Symposium on Principles of Programming Languages, ???, ??, January ???.

[Wad92] Philip Wadler. The essence of functional programming. In Proceedings of the 19th Annual Symposium on Principles of Programming Languages, pages 1-14, Albuquerque, NM, January 1992.

# 訳者あとがき

1. レゴ（LEGO）の名称は登録商標されている。 [↑](#footnote-ref-1)
2. バーは、私の疑問に対して親切な返信をしてくれたが、なぜ彼の条件が異なるのかということについては思い出してくれなかった。 [↑](#footnote-ref-2)
3. [1] これの初期の理論案において、私がある例を十分説明することができないでいると、読者の一人は「君は自分の理論をSchemeで書けてないだろ！」と不平を述べていた。 [↑](#footnote-ref-3)