修 士 論 文

自己反映的逐次プログラミング言語の効率的なコンパイル手法について

指導教官　渡部 卓雄

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

佐伯豊

平成9年 2月 14日

要 旨

自己反映的 (reflective) な言語システムとは、その言語システム自身が、自己の構造や、計算過程に関する計算をおこなうことを可能とするような構造をもった言語処理系である。その利点は、システムが実行時の状況に応じて柔軟に自分自身の構造を変化させることが可能であるという点、そしてユーザーによる意図的な拡張が可能であるという点である。現在までに多様な自己反映的な言語処理系が提案されてきたが、実行効率と高い拡張性を兼ね備えたシステムは、いまだに得られていない。また、拡張部分のモジュール化のための枠組みが与えられておらず、プログラミングの容易さからいってもいまだ実用的な処理系の実装はなされていない。本研究では、言語の意味を二つのオブジェクトによって実現し、言語の拡張を、そのオブジェクトの性質を変化させるようなモジュールの実行時の組み込みとして提供することで自己反映的な言語処理系を実現した。その際、特に各モジュールの結合の安全性に注意し、設計をおこなった。これによってプログラムの実行時における部分的な言語拡張を提供することを可能とし、計算システムに対する拡張の容易な記述方法を示した。また処理系のコンパイラとしての実装をおこない、プログラムの効率的な実行を試みた。

目 次

図 目 次

第 1 章

序論

計算システムが自分自身の構成や、計算過程に関する計算をおこなうことを**自己反映計算 (リフレクション)** という。自己反映計算の機能をもつ言語システムを自己反映的な言語とよび、このような言語では優れた拡張性と動的な環境の変化に応じた自己改変を提供することができる。本研究では、自己反映的な言語の設計・実装において、特に拡張に関する記述のしやすさに着目して、いくつか問題点を挙げ、その解決の手段として、拡張部分のモジュール的な記述方法を提供する。さらにその拡張手法に基づいた自己反映的な言語のコンパイラの設計・実装を試みる。

1.1 背景

自己反映的なシステムは、高度な拡張性をもち、動的な環境の変化にたいして柔軟にシステム自身を変更することで適応することが可能であるという利点がある。そのため特に言語処理系や、OS といった広義の記述系にたいして自己反映的な枠組みを導入することによって、移動計算システムや分散システムなどの高度で複雑なシステムをより秩序立った構成で構築することが可能になることが示されている。

特に自己反映的な言語については、様々なシステムが実装されており、その有効性が実証されている。しかしこうした自己反映的な言語を実現する場合、多くの処理系が meta-circular なインタプリタによる、拡張部分のメタレベル実行という方法をとるため、プログラムの解釈実行のためのオーバーヘッドによる実行効率の低下が生じるという問題がある。

さらに、プログラミングのしやすさの観点からすると、自己改変の影響がプログラム全

体におよぶことと、メタレベルで扱う構造の抽象度が低いことにより、記述の際、常にそれまでになされたすべての拡張事項に関して留意しておく必要があるため、メタレベルに関する記述の再利性が低く、プログラミングが困難であるという問題があげられる。このように、実用性において自己反映的な言語はいまだに多くの問題点を残している。

1.2 目的

本研究の目的は、特に以下の項目に示した要求を満足するような自己反映的な言語の設計および実装をおこなうことである。

1. 自己反映計算に関する記述の理解の容易さ。
2. 拡張部分に関する記述の再利用性。
3. 効率的な実行。
4. 高い拡張性。

こうした要求を満たすためのアプローチとして、まず言語拡張に関する記述をベースプログラム自身とは独立したモジュールとして与え、それらをユーザーがプログラムに必要に応じて組み込むというかたちの整理された手段を提供した。また、各種のモジュールを組み合わせて用いるための枠組みを提供することによって容易な再拡張の方法を提供し、言語対して高い拡張性を与えることを可能にした。さらに、言語の意味に対する拡張部分の構造をプログラムからできるだけ切り離すことで、十分な拡張性を保ったまま、言語処理系をコンパイラとして実装することを可能にした。

1.3 本稿の構成

２章では自己反映計算のモデルに関する基本的な説明と、いくつかの実装例について述べ、考察をおこなう。３章では、monadic な言語設計に関する説明をおこない、今回言語を設計する際に参考にした Sobel らの研究??について述べ、本研究との比較をおこなう。４章では、実際の設計と各構文の適用例について述べ、６章で実装に関して解説をおこなう。最後に７章で結論を述べる。

第 2 章

自己反映計算

本研究は自己反映的な言語システムのコンパイラの設計・実装を目的としている。本章では、自己反映計算の概念について解説をおこない、具体的な自己反映的言語の実装例についていくつかのべ、その設計に関して考察をおこなう。

2.1 自己反映的システムについて

自己反映的 (reflective) なシステムとは、その計算システム自身が、自己の構造や、計算過程に関する計算をおこなうことが可能なシステムである。本節では自己反映計算に関するより詳細な解説をおこない、本研究における言語の設計に関係する基本的な概念を示す。

2.1.1 自己反映計算の仕組み

まず、自己反映計算における自己改変の仕組みについて解説をおこなう。ある計算システムにおいて、そのシステムが計算対象としている実体 (entity) と、そのシステムにおける内部表現との間で、どちらか一方に対して変更が生じた場合、もう一方がその変更に応じて影響をうけるという関係が成り立つ場合、両者は**因果的結合**をもつという [15]。自己反映的システムとは、システム自身に関する表現を内包していて、その表現とシステム自身とが因果的結合の関係にあるような計算システムをいう。このことによって、システムが内部に含んでいる自分自身のモデルに対して何らかの操作をおこなうことによる自己改変を実現することが可能になる。

2.1.2　手続き的リフレクション

自己反映的なシステムを実現するばあい、自己のモデルをどのような構造で表現するか、そして因果的結合をどのように提供するのかという問題がある。自己反映的システムの一般的なモデルとしては**手続き的リフレクション**とよばれる概念が与えられている [20]。ここでは一般的な自己反映計算の設計の基本的な枠組みを示す意味で、手続き的リフレクションについて解説をおこなう。

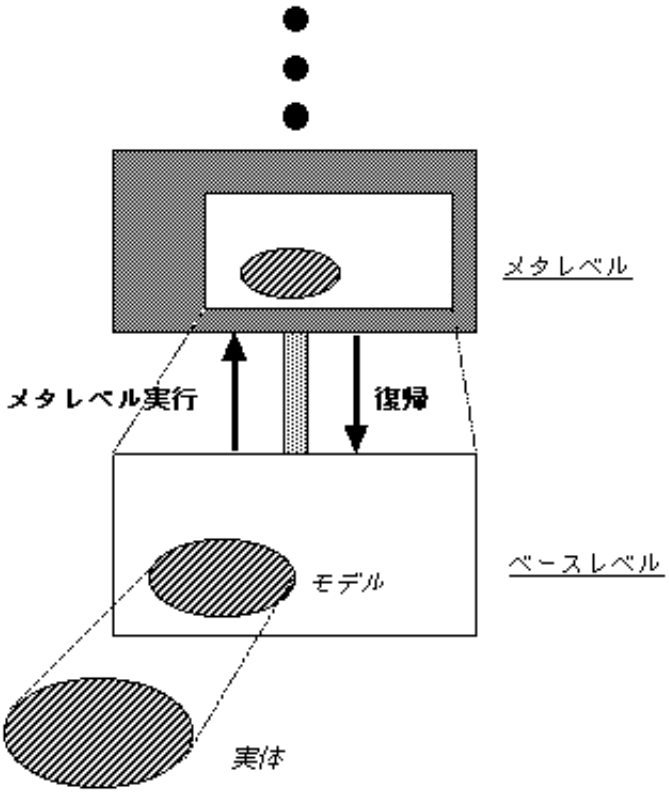


図 2.1: 手続き的リフレクション

図 2. 1に示すように、直接現実世界の実体のモデルを扱う本来の計算に相当する部分システムを**ベースレベル**のシステムとよぶ。また、あるシステムのモデルを内包していて、その振る舞いをシミュレートするシステムを**メタレベル**のシステムとよぶ。手続き的リフレクションのモデルは、ベースレベルから、そのメタレベルを順次階層的に構成していく形式で与えられる。このような階層構造を**リフレクティブタワー**と呼ぶ。

メタレベルにおいてシミュレートされるベースレベルシステムの表現は、メタレベルにおいて自由に参照・変更が可能であり、その影響は間接的にシミュレートされるシステムに

反映される。手続き的リフレクションでは、自己改変を**メタレベル実行**と呼ばれる仕組によって実現する。メタレベル実行とは、シミュレートされるレベルの計算の一部をシミュレーションを介さずにメタレベルで直接実行するような仕組である。このことでメタレベルの構造に対する間接的なアクセス手段を提供することができ、因果的結合が実現される。

ここで見たように、自己反映的なシステムが実現する計算の構造とは、本来の目的としている計算と、その計算の内容を実現している意味を定義した記述とが、言語の提供するアーキテクチャにもとづいて、異なるレベルとして分離された構造である

システムの機能をその抽象度によって分類し、モジュール化することで、従来の計算システムではアドホックに導入されてきた機能 (効率化や例外的な処理、インターフェイスなど) をあつかうための構造を、言語の提供するアーキテクチャに基づいた整合的な枠組みのもとで共有することが可能になるのである [26]。

2.1.3 オブジェクト指向言語における自己反映計算

自己反映計算モデルは、オブジェクト指向言語において実現されることが多い。オブジェクト指向言語における自己反映計算は、**メタオブジェクト**とよばれる、オブジェクトそのものに関する概念 (クラス、メッセージ、メソッドなど) を統一的に表現するためのオブジェクトによって実現される。さらに、メタオブジェクト自身もまた、オブジェクトであるので、それ自身もメタオブジェクトをもつ。これは、リフレクティブタワーと同等な構造である。すなわちオブジェクト指向の枠組みそのものが、自己反映計算の概念にうまく合致することがいえる。実際 3-KRS[14][16] をはじめとして、様々な自己反映的なオブジェクト指向言語が実現されている。

実用的な自己反映的な言語が、オブジェクト指向言語において数多く生まれたのは、オブジェクト指向モデルそのものが、優れたモジュール化手法であり、メタレベルの再利用が容易であることが挙げられる。

2.2 自己反映的言語の実装

本節では、従来の自己反映的な言語の実装について解説をおこない、それらの問題点について考察をおこなう。

2.2.1 Brown

Brown[25] は、Scheme 上に実現された自己反映的な言語システムである。Br own では、式、環境、継続という三つの計算状態によって、ベースレベルのモデル化を行っている。メタレベル実行によって制御をメタレベルに移したあと、ベースレベルに復帰する際には、メタレベルのシステムの状態 (復帰時の継続) を保存しておかなければならない。これを**メタ継続**とよび、Br own では、メタ継続の集合によってリフレクティブタワーを実現している。

Br own は、表示的な意味に近いモデルを言語自身に与えており、その性質が明確に示されているという利点がある。しかし、実装においては、メタレベル実行が、システムの状態を Br own の扱うシンボリックなデータ構造にマッピングする関数の呼びだしを通じておこなわれるため、実行時にシステムの状態がソースコード の形式であたえられている必要がある。そのため、実行時のソースコード とその解釈系の存在が必要となり、実行効率の問題が生じる。

また、言語のモデルとして、式、環境、継続を渡す形態の固定的なインタープリタを採用しているため、関数呼びだしの方法や変数の参照方法といった、より言語の基本的な機能に関する拡張ができない。

他にも様々なインタープリタベースの自己反映的言語システム [6][11] が提案されているが、総じて解釈実行によるオーバーヘッド の問題が存在する。

2.2.2 CRML

CRML は、関数型言語 ML において、コンパイル時の自己反映的な計算を実現した言語システムである。CRML の設計は、すべてのプログラムの記述を、メタレベルでおこなうようになっている。実際の記述は、システムが提供する、ベースレベルの表現をメタレベルの表現に変換するための構文と、直接記述したメタレベルの表現を、組み込むための機能を利用しておこなう。こうした変換と埋め込みはコンパイル前にすべておこなわれるので、そのまま直接 ML の処理系によって扱うことが可能であり、解釈実行をおこなう必要がないため実行効率が高い点である。また、ML の型チェック機構を利用することが可能で、実行時の型の安全性を保証することができる。

問題点としてはプログラマが常にメタレベルへの変換を考慮にいれて、プログラムを記述しなければならないこと、またメタレベルとベースレベルとの境が明確ないので、記述が困難であり、メタレベルの再利用が困難であるという問題がある。さらに、拡張可能な

部分が、上述のようにコンパイル時に組み込むことができるような記述に限られるため、単なる構文の拡張にとどまってしまい、動的な改変を直接実現するようなことはできない。

コンパイル時の、自己反映計算をおこなうシステムのなかには、ユーザーがコンパイラの内部構造に対して変更を加えるような機能を提供するものがある。しかし、コンパイラ自身は非常に複雑なシステムであり、言語の意味の変更を、コンパイラのコード生成過程に反映させることが困難であるため、ユーザーにとっての負担が大きくなる。

2.2.3 ABCL/R3

ABCL/R3[17]は、並列計算に関する、負荷分散・スケジューリングといった、メタな制御をあつかうために設計された自己反映的な並列オブジェクト指向言語である。

ABCL/R3では、オブジェクトのメソッド実行をメタレベルで操作的に定義するような評価器(evaluator)が存在し、それを委譲形式(deligation)によって拡張することが可能になっている。この評価器オブジェクトをプログラムに対して与えることで、前述の制御を実現する。

ABCL/R3では拡張された評価器を持つようなプログラムは、部分計算を適用することでコンパイルされる。しかし部分計算手法は、副作用や並列計算などに関する制限があるため、それを解決するために、実行効率を多少犠牲にしている。また、部分計算は、基本的にソースコードレベルでのプログラム変換であり、常にメタレベルの評価器がソースコードで提供される必要があるため、メタレベル評価器のモジュール性が低いことや、評価器の動的な変更に関しては、あらかじめいくつかのバリエーションを用意しておいて、実行時に選択するという手段をとる必要がある。

しかし部分計算による自己反映計算の実現の有効性は実証されており、将来的には有望なアプローチであると思われる。

2.2.4 その他の言語

また、OpenC++[2], Black[1]やRbOC[10]、AL-1/D[19]など、自己反映的な言語処理系が多数存在するが、完全にコンパイルされたコードにおいて動的な改変を実現するような処理系はいまだ存在しない。

2.3 自己反映的言語のコンパイラの実現方法について

自己反映的な言語のコンパイラの設計は、一般的に困難であるといわれてきた。コンパイルとは、言語の意味を保存したままプログラムのソースコードをバイナリコードに変換する操作であるといえる。本節では、自己反映的な言語のコンパイルに関する考察をおこなう。

2.3.1 コンパイルの過程について

ソースコードに記述された構造のなかには、静的に決定できる部分と、実行時でなければ確定しない部分が存在する。通常のコンパイルの過程では、プログラムの意味における静的に決定が可能であるような計算状態、例えば継続や、式の型などを、コンパイル (プログラムの変換) 過程における計算状態として利用することになる。そのため静的に決定できる計算状態は、コードの生成過程に反映され、動的な部分は生成コード中にあらわれる。

自己反映的な言語のコンパイルが困難である理由の一つは、言語の意味そのものが流動的であるため、言語の変換過程に対して、固定的に利用可能な計算状態というものがあらかじめ特定できないという点である。もし、すべての状態を実行時にあつかうとしたら、結果的に完全なインタープリタが生成されることになる。

本来、計算対象であるデータ自身は、コードの生成過程に関して影響を与えることはない。いくつかのデータに依存するような最適化手法においては、コード生成の段階で特定のパターンを検出し、その意味を保存したままで、より効率的なパターンへ変換をおこなう。しかし結果的には定数のような、コンパイル時に識別できるものとして、あらかじめ認識されているような構造にたいして、**ad-hock な適用**がなされるだけであり、データによってコード生成を変化させるような統一的な枠組みは与えられていない。しかしながら、自己反映的な言語においては、コードを生成しようとする時には特定されていない計算システムの状態自身がデータなのである。

2. 3. 2 従来のアプローチ

理想的には、すべての改変がコード生成の過程において解決されているものが望ましいが、現実には無限のバリエーションが存在する。

そのためまず考えられるのは、自己反映的な言語のコンパイラの実現に際して、拡張機

能にある程度の制限を加え、通常のコンパイラが扱うことが可能な構造をもったコードを、独立した構造として提供するというアプローチである。これは、ある目的に特化した言語システムとしては実行効率や抽象性の点で有効であるが、逆にすべての要求を満たすような拡張機能の設計は困難であり、ユーザーにも負担がある。

別の解決手段としては、プログラム中からコンパイラそのものに変更を加えることで、改変を実現する手法がある。つまり、ユーザープログラムがコンパイラの状態そのものに影響を与えるような記述をおこなうことになる。これは、コンパイラの構造の複雑さからいって得策ではない。もしこの手法を扱うとしたら、ユーザーが安全かつ容易に言語拡張をおこなうために自己改変の機能を大きく制限する必要がある。アプローチとしては、コンパイラ自身をオブジェクト指向のモデルに基づいて構築し、ユーザーによる変更を容易にする手法が提案されているが、コンパイラの構造に関する、ユーザーによる拡張を簡潔にあつかうようなモデルはいまだ与えられていない。

また別のアプローチとしてその有効性が示されているのが、部分計算による実現方法である [5]。部分計算とは、部分的な入力に基づくプログラム変換の手法である [4]。変数のバインド情報や、継続といった計算状態を部分的な入力に応じて変換時に計算することで、プログラムを入力データに特化したものに変換することが可能である。インタープリタそのものもまた、プログラムの一種であり、それに対して、ユーザープログラムはあらかじめ与えられた部分的なデータであると考えることが可能である。ユーザープログラムに対して変換を施されたインタープリタは、ユーザープログラムに対して特化されたものであり、改変に関する記述に注意して変換を施すことにより、静的に決定可能なデータは変換後のインタープリタに組み込まれる。こうして得られた変換コードは、メタレベルのインタープリタにとっての入力データと考えることができる。こうして、インタープリタの階層構造を、動的な構造を継承したまま単独の層にまで変換し、直接コンパイルが可能なプログラムを自動的に生成することが可能になる。この手法には、meta-circular なインタープリタの構造をそのままユーザーに提供することが可能であり、記述性が高い上に、プログラム変換によって効率的な実行が提供可能であるという利点がある。しかし今のところ部分計算自身に関する制限や、コンパイル後のコードの再利用性などの問題があり、それが言語の設計に関する足かせとなっていることは否めない。

2.3.3 考察

ここで述べた自己反映的な言語では、すでに存在している計算システムの状態を、ユーザーが参照・変更することが可能な枠組みを提供することで自己反映計算を実現していたため、実装にさいして、ユーザーが拡張をおこなうために必要とするであろう情報を網羅しようとすると、言語システム自身のあらゆる状態が必要となること、さらに従来の処理系では言語システムの状態をユーザーが利用可能なデータ構造として表現する必要があるということによって、実行時の解釈実行系の存在が必要であった。

ならば、視点を変えてユーザーが拡張されるべき言語の意味構造そのものを容易に定義し、実行時に組込むことが可能な枠組みを提供すれば、ユーザーは、アクセス可能な処理系の状態の構造そのものを定義することが可能なことになり、効率と拡張性の両面において有利な設計を実現できると言えるのではないだろうか。これは、プログラマがシステムのモデルとその表現そのものを記述することによって、拡張に際してある程度の融通がきく、つまり拡張に都合のいいようにシステムの構造そのものを決めてしまえるという利点を生む。また、改変の対象となる構造は、本来そのシステムが備えていない言語の機能を**直接**処理系に組み込んだものであるため、言語システムの機能を別のシステムでシミュレートするという概念を必要としない。つまり階層構造を生まないため、解釈実行のモデルを本質的に含まない構造をもっている。

このように容易に言語処理系の意味を、モジュール的に拡張していくような枠組みとして、**monad** をもちいた言語の構成方法がある。第 3 章では、この monad について解説をおこない、それをふまえて今回実装をおこなった言語の設計方針について述べる。

第 3 章

準備

第 2 章で述べたように、言語の意味を、モジュール的に拡張していくような枠組みとして、monad をもちいた言語の構成方法がある。第 3 章では、この monad について解説をおこない、それから今回言語の設計に際して参考にした Sobel の研究に関して述べ、最後に今回実装した処理系の設計方針を示し比較をおこなう。

3.1 monad

今回実装をおこなった言語の設計は、カテゴリ理論において monad と呼ばれている数学的な構造を利用した言語の構成方法にもとづいている。

それまで表示的意味論の欠点は、その拡張性とモジュール性の欠如であった。Moggi は、表示的意味の枠組みにおいて言語の意味をモジュール的に段階をおって拡張していくことで構築していくための枠組みとして monad を用いることを提案した [18]。

Wadler らは、それを純粋な関数型言語に適用し、例外処理や入出力といった imperative な機能を、monad を利用したモジュール的な手法を用いて実現するための枠組みを提供した [23][24]。それを受けて、monad は純粋な関数型言語の拡張手法として頻繁に用いられ、並列計算やグラフィックス処理などの様々な応用例が提案された。

今回設計をおこなった言語の拡張モデルは、monad による言語の意味のモジュール的な定義方法に基づいている。本章では、第 4 章で与える言語の設計の準備として monad に関する基本的な解説をおこない、その自己反映計算との関連について述べる。

3.1.1 monad とは

一般的なインタープリタの動作は、式 (Exp)、環境 (Env)、継続 (Cont)、記憶領域 (Store) といった計算の状態をしめす値の組から、結果 (Value) への対応づけととらえることができ、次のような型で示される。ここで'::'は、左辺が右辺の型をもつことを示しているとする。

$$Eval :: Exp \to Env \to Cont \to Store \to Value$$

これに対して monad 的な構成方法をとるインタープリタでは、計算の意味をある式から **computation** (環境や継続などの扱いに関する詳細を隠した構造) への対応づけとみなす。

$$Eval :: Exp \to M\ Value$$

ここで示された型構成子 M が monad と呼ばれる構造である。M は次のふたつの多相関数 unit と bind によってその性質が決まる。

$$unit :: a \to M\ a$$
$$bind :: M\ a \to (a \to M\ b) \to M\ b$$

monad 形式への変換の基本的な考え方は、**型 $a \to b$ の関数は、型 $a \to M\ b$ の関数に変換される**というものである。

概念的には、$M\ a$ は**型 a の値を返すような computation** とみなす。また unit は通常の同値関数に相当するもので、値を computation に置き換える関数と考える。例えば値 $v :: a$ について、$unit\ v :: M\ a$ は、**値 v を返すだけの computation** を表わす。

そして bind は関数の結合に相当し、ある computation と、その返す値に依存するような computation の二つを結合する操作の定義をしめす。例えば computation $m :: M\ a$ と、値から computation への関数 $k :: a \to M\ b$ について、$M\ b\ bind : M\ b$ は、m の返す値を、関数 k が受け取って、その結果として computation を返すような操作を示す。

3.1.2 monad の適用例

ここでは具体的な適用例として、実際にインタープリタを拡張していく過程を示す。まず、非常に単純なインタープリタを定義する。

$$
\begin{aligned}
data\ Term &= Con\ Int \mid Div\ Term\ Term \\
eval &:: Term \to Int \\
eval\ (Con\ a) &= a \\
eval\ (Div\ t\ u) &= eval\ t\ /\ eval\ u
\end{aligned}
$$

図 3.1: インタープリタ 1

これに対して、monadic なインタープリタの定義は、以下のようになる。

$$
\begin{aligned}
eval &:: Term \to M\ Int \\
eval\ (Con\ a) &= unit\ a \\
eval\ (Div\ t\ u) &= (eval\ t)\ `bind`\ \lambda a.(eval\ u)\ `bind`\ \lambda b.\ unit\ (a/b)
\end{aligned}
$$

図 32: インタープリタ 2

ここに示したように monadic なインタープリタは、コンストラクタ M と unit そして bind [1]の定義をおきかえるだけで簡単に変更が可能な構造になっている。具体例としてもっとも基本的な monad の定義をあげる。

$$
\begin{aligned}
type\ M\ a &= a \\
unit &:: a \to M\ a \\
unit &= a \\
bind &:: M\ a \to (a \to M\ b) \to M\ b \\
bind\ a\ k &= k\ a
\end{aligned}
$$

図 3.3基本

図 3.3において定義した基本的な monad をそのまま図 3.2のインタープリタに組み込むと 3.2が得られる。これを利用して、言語に様々な機能を加えることが可能になる。たとえば例外処理機構は図 3.4に示した monad によって組み込まれる。

[1]ここでは理解のために左結合の infix 記法で示している。

$$
\begin{aligned}
data\ M\ a &= Raise\ String \mid Return\ a \\
unit &:: a \to M\ a \\
unit &= Return\ a \\
bind &:: M\ a \to (a \to M\ b) \to M\ b \\
bind\ m\ k &= Raise\ e \to Raise\ e \mid Return\ a \to k\ a \\
raise &:: Exception \to M\ a \\
raise\ e &= Raise\ e
\end{aligned}
$$

図 3.4: 例外処理

このように、monad を定義してインタープリタに組み込むだけで、容易に言語の拡張が可能になる。この枠組みは、インタープリタだけでなく、コンパイラに対しても適用することができることが知られている [13]。monad が実現するものは、計算自身を直接計算対象として扱うことが可能であるという点である。つまり、computation とは、そのシステムに関するメタレベルであるといえよう。

3.2 Sobel らの研究

monad によって reflective な言語を説明することは、いくつかの研究において試みられている。例えば [8] では Brown において reflection の意味を与えるために提案されたメタ継続が monad をもちいて実現されている。

しかし実際に monad に基づく自己反映的な言語の設計があたえられたのはごく最近のことである。Sobel は monad をオブジェクトの継承を利用して定義するための枠組みと、reify や reflect によってユーザーが言語の拡張をおこなう方法を示し、それに基づく変換ベースの言語の設計と実装をおこなった [21]。

ただしプログラムの記述のしやすさや、実行効率の面で以下のような問題があった。

- 言語の拡張の影響がプログラム全体におよぶこと。
- 変換をおこなう対象となる言語にオブジェクト指向プログラミングのための機能を前提としていること。

- 言語自身は、基本的に純粋な関数型言語であり、制限があること。
- 言語の拡張部分が複雑に関連しあっており、互いの有効な結合方法がしめされていないこと。
- 拡張機能ごとに複数の実行スレッドを生成することで実行効率が悪いこと。

このような問題点を含みながらも、Sobel が示した手法には、拡張部分の記述の優れたモジュール性や、変換ベースの実装など本研究の目的から望ましい性質が多い。

3.3 本研究のアプローチ

そこで本研究では Sobel らの提案をもとにして自己反映的な言語 Flect(Fluid Reflective Language) のコンパイラの設計・実装をおこなった。Flect の実装に際しては、以下に示す方針に基づく設計をおこなった。

- Scheme 言語をベースとした言語処理系であり、Scheme の強力な記述力をそののまま利用できる。
- 既存の言語処理系によって直接扱うことが可能な、プログラム変換ベースの実装。
- コンパイルしたプログラムにおいて、自己反映的な計算をおこなう。
- monad に基づくモデルによって言語の拡張部分の記述を、モジュール的におこなうことが可能。
- プログラム中の特定の範囲を指定して、言語の拡張をおこなうための手法を提案し、実装する。
- それぞれ独立した役割をもつ二つのオブジェクトによる簡潔なモデル化をおこない、拡張モジュールの役割を明確化することで、各モジュール間の安全な結合を実現する。

第 4 章では、この方針にもとづいた言語 Flect の設計とその実装について解説をおこなう。

第 4 章

自己反映的言語 Flect の設計

第 3 章において、monad に基づいた言語拡張の枠組みについて述べ、今回実装をおこなった言語 (Flect) の設計方針を示した。本章ではこの方針に基づいた言語の設計と実装について述べていく。

4.1 システムの実行モデル

Flectでは、言語の意味を決定する部分を、処理系から切り離して提供する。それは図 4.1 のように二つのオブジェクト (reflective-object と control-object) によって与えられる。また、それぞれのオブジェクトの性質は、複数のモジュールによって実現される。ユーザーが、オブジェクトに対して、実行時におこなうことが可能な操作は、以下に示すものである、

- あらかじめ定義しておいたモジュールを組み込むこと。
- プログラム中からモジュールを定義し、組み込むこと。
- オブジェクトの状態にアクセスすること。

このような操作によって、ユーザーは間接的に言語の意味そのものに変更を加えることが可能になる。つまり、オブジェクトが自分自身のモデルであり、モジュールの集合がその表現であるといえる。モジュールの集合に対する操作はそのままオブジェクトのふるまいを変化させ、その結果が即座にシステム自身に反映されることから、因果的結合が実現されているものとみなせる。

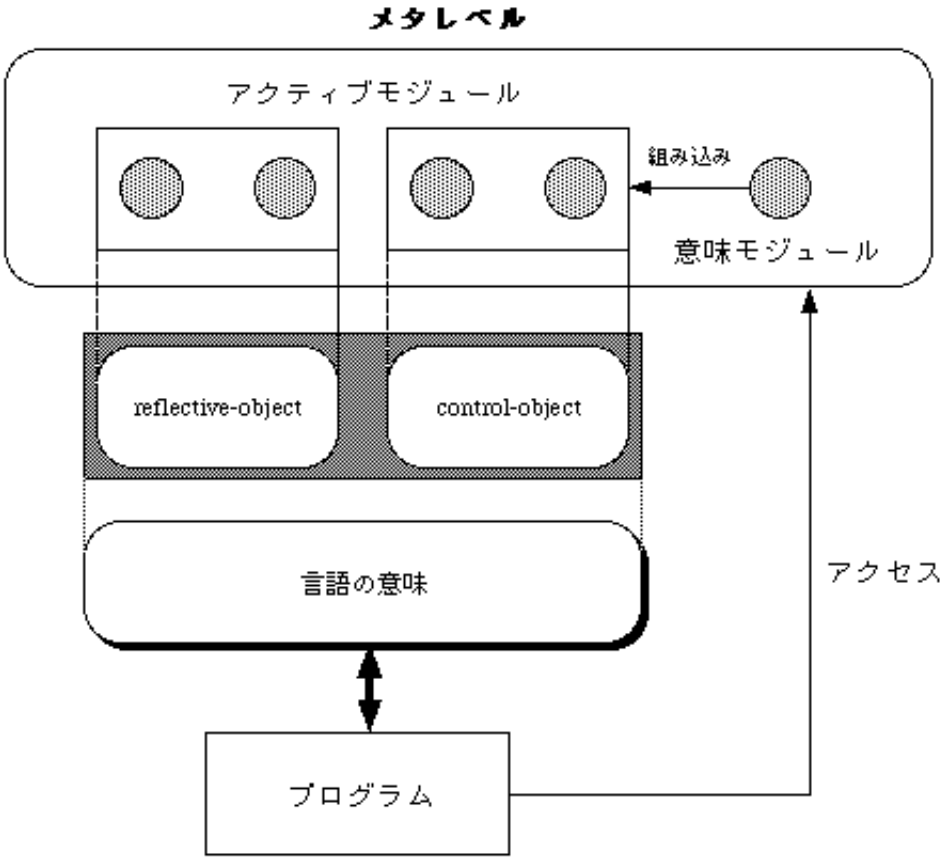


図 4.1: Flect システムのモデル

4.1.1 モジュールによる言語の拡張

Flect のソースコードは、システムによって実行コードへコンパイルされる。しかし、通常のコンパイラと異なり、Flec の実行コードにおいては、言語の意味を反映している部分が分離され、独立したモジュールデータの集合として与えられている。Flec の実行時のシステムは、そのモジュール群によって実現された言語の意味にしたがって、実際の計算を進めていく。

Flec の特徴は、ユーザーが言語の意味を独立したモジュールによって抽象的な形で定義し、プログラムのなかで、言語システムに組み込むための仕組みを提供することにより、言語の部分的な拡張を実現した点にある。その利点を、以下に示す。

- 言語拡張の定義があらかじめコンパイル可能なデータ構造であること。
- 拡張部分の定義の抽象度が高く、アプリケーションに依存しない構造であるため、再利用が可能であること。
- それぞれのモジュールが、それ自身で完結しているため言語拡張の組み合わせが容易であること。

従来の関数型言語のコンパイラでは通常、式の評価過程における静的に決定可能な部分をそのまま実行コードの生成過程に適用することで、言語の意味を生成コードに反映させるため、自己反映計算における言語の意味の実行時における改変を、いったん生成されてしまった実行コードに対して適用することは困難であった。

それに対して Flec では、言語の意味そのものを抽象的な構造でモジュール化したものを実行コードにおいて分離した形で提供している。また式の型や、環境、特殊形式の扱いなど静的に決定可能な構造は、Scheme の semantics をそのまま利用し、直接コンパイルされるため、コンパイルされたコードに対して、モジュール的な枠組みで、整理された言語の意味の拡張が可能であり、また、生成コードにおける改変の影響を受ける部分の特定が容易である。以上のことにより、Flec においてコンパイルされたコードにおける自己反映計算を実現することが可能であることが言える。

4.1.2 言語の意味の構造

本節では各オブジェクトが具体的に、Flect においてどのような意味構造を決定しているのかを述べる。

Flect のプログラムは、Scheme と同様に S 式によって構成されている。しかし Scheme では、プログラムの各構造に対する意味があらかじめ与えられている。具体的にいえば、式と環境と継続という決まった計算状態を受け渡していく規則が明確な意味として定められている。いわば、式自身が計算そのものの表現であるといってもよい。

例えば、リテラルは、対応するデータ構造そのものを示し、変数は環境に束縛されている値を示し、プロシジャ呼びだしは、引き数を評価した結果をローカルな環境として、また残りの計算を継続として渡すという一連の動作そのものを示すというふうに、式の表現する計算は、固定的である。

もし、こういった計算状態をユーザーが言語の拡張機能として組み込むことを考え、直接記述しようとすれば、複雑な処理をソースコード中に埋め込まなければならず、プログラムの理解が困難になり、エラーを生じやすくなり、またプログラムの修正も容易ではなくなる。特に、これらの構造を組み合わせて適用するには、様々な関連を考慮しなければならない。

Flect システムは、こうした計算状態に関する操作を抽象化し、モジュールとして定義する機構をもっている。ただし、モジュールの記述対象となるような計算の構造は 2 種類に限定されている。すなわち環境のように計算過程によって変化し、計算の値として利用するために参照される計算状態と、継続のように実行制御に影響をあたえるような計算状態である。この 2 つにクラス分けされたモジュールがあつまって、それぞれのオブジェクトを構成するわけである。

二つのオブジェクトは、どちらも計算状態に関する操作を定義したものであるが、制御構造を独立したオブジェクトとして実現した理由は、制御構造を分離することで、それぞれの構造が明確になり、モジュール同士の結合が容易になることによる。

4.1.3 atomic な計算と compound な計算

では具体的に前述のような計算の意味はどのように抽象化されているのだろうか。ここで第 3 章で解説をおこなった monad による言語の構成手法が利用できる。monad とは、計算という概念そのものを、first-class な値として扱うという手法である。では計算そのもの

をどのようにモデル化しているのだろうか。

monad では基本的に計算を、その性質によって 2 つのカテゴリに分け、それぞれに対するアクションを与えることによって、意味づけをおこなっている。その性質とは以下に示したものである。

atomic プログラムの単独のステップに関する計算（変数・定数・closure の生成など）。

compound 式の評価のために、その部分式を評価する必要がある複合的な計算 (関数呼び出し・条件式など) であり、さらに以下のように細分化される。

- **init** は部分式のうち、最初に評価されるものについての計算。
- **rest** はそれ以外の残りの式に関する計算。

つまり、atomic 計算とは、式に対して値の表現を与えるための操作であり、compound 計算とは、その表現を複合的な計算の間で関連づけるための操作であるといえる。この両者によって言語の意味が定義されることを直観的に理解するために、次の例題を用いて解説をおこなう。

```
(if                    ;;          |
    x                  ;; init     |
    (write x)          ;;         rest
    (write "false")    ;;          |
```

例題における init 計算は変数参照、すなわち atomic 計算であり、x という変数に関する計算、つまり環境において x に束縛された値を返す一連の操作が意味的に含まれている。

rest 計算は、init 計算の返す値 (すなわち変数参照の結果) を受けて実行される、残りの計算 (条件分岐の結果として呼ばれる計算) を示している。ここで、条件が真であった場合、変数 x の参照をおこなっている。この x は、通常の Scheme の semantics では init 計算で参照されたものと同一の環境から参照される。この環境の受け渡しを定義するのが compound 計算の意味である。要するに、compound 計算とは、**計算同士の組み合わせにおける計算状態の受け渡し**に関する操作を意味的に含んでいる。Flect は、この二つの意味を言語処理系自身から分離して扱う。

視点を変えると、ここで分類をおこなった計算の単位とは、インタープリタにおける式評価 (eval) の適用される単位であると考えることもできる。compound 計算にあたる式の評価においては、インタープリタはさらに eval を呼び出すことをおこなう。つまり compound 計算の意味とは、この eval 呼び出しにあたって渡される環境や、部分式といった計算状態の受け渡し過程そのものであるといってよい。いいかえると、各モジュールは、それぞれが特定の計算状態に特化した構造をもつ独立した、しかも互いに結合可能なインタープリタの**部品**なのである。

ここで興味深いのは実行コードの効率がこのインタープリタの部品の一度に組み込まれる数とそれぞれの質によって決まるということである。もし効率を重視するなら、なるべく各部品の構造を単純にしてその組み合わせを工夫するべきである。

Fleet においては、前述の二つのオブジェクトによってすべての計算状態を管理している。このオブジェクトは、計算の間で受け渡していくことによって呼び出し (状態の更新) がなされる。このふたつのオブジェクトによって表現されるものは、いわばモジュールの集合によって構成された**総合的な**インタープリタそのものであると言ってよい。つまり、拡張可能なインタープリタを受け渡していくような計算モデルをイメージすることが可能である。こうした枠組みの、より直接的なアプローチとしては [22] がある。

4.1.4 言語の意味の反映

Flec における言語の意味を決定する要素とは、**計算状態**と、その計算過程に応じた**変化規則**である。計算状態は変数として与えられ、変化規則は、その変数への更新アクションとして与えられる。Flec では、上述の計算単位において Scheme 言語の semantics によるアクションの直前に、ユーザーが定義した操作を hook する。ここで hook される操作が、すなわちユーザー定義の状態変数に対する更新操作であり、atomic、compound の各計算に関する操作と、対象となる状態変数との組によって、ひとつの意味モジュールが構成される。

Flec では、プログラムの各構文に対してこのモジュールの実現する意味を反映させるために、プログラムの変換をおこない、プログラムを評価される式という立場から、計算そのものの表現という立場に置き換える。

簡単な変換

例として、非常に単純な式の変換を示す。

```
(+ 1 2)
```

簡単な加算であるが、これは以下のように変換される。ただしここで(alpha x)とは、x を扱うという atomic な計算示し、(gamma x (lambda (g) y))とは x という計算と、その結果を受け取って続きの計算 y をおこなうという計算を示している。[1]詳細な変換規則に関しては、第 5 章において示す。

```
(gamma (alpha +)
  (lambda (g0)
    (gamma (alpha 1)
      (lambda (g1)
        (gamma (alpha 2)
          (lambda (g2) (g0 g1 g2)))))))
```

図 4.2: 変換後

計算の意味は alpha と gamma の内容による。つまり、alpha と gamma を変更することでいくらでもこの計算の意味は変化するのである。図 4.1 にしめした、2 つのオブジェクトとは、この alpha と gamma の内容を決定するオブジェクトと考えることができる。このことから、モジュールの組み込みが言語自身の構造を変化させることはあきらかである。

いわば Flect システムとは、式を計算の表現へ変換し、実行時に適切なアクションを呼び出すためのサポートをおこなうシステムであるといえる。

[1]最終的な値を返すぶ部分 (図 4.2の(g0 g1 g2)) は、計算として扱われてはいない。これはプロシジャ呼びだしと、変数参照、条件分岐、副作用に関する意味は、Flect のオブジェクトでは扱われていない構造であり、Scheme の semantics をそのまま利用するため、計算として表現されないためである。

4.2 システムの概要

ここでは言語システムの実現する拡張機能に関して具体的な解説をおこなう。まず、なんら拡張をおこなわない場合、Flect のふるまいは単なる Scheme のサブセットとしての意味だけをもつコンパイラである。しかし Flect システムは、 ユーザーが言語の意味構造を決定するオブジェクトの機能を、構文 **defrmod** および **defcmod** によって宣言されたモジュールによって定義し、それを言語システムに組み込む手段を提供することによって言語の意味の拡張を実現している。

しかもその拡張は、プログラムの実行の流れのなかのある区間を指定しておこなうことが可能である。その結果としてユーザーは、プログラム中の、必要とする部分において、自身が定義し、組み込んだモジュールによって与えられる言語の意味構造にたいしてのみ考慮して言語拡張に関係したプログラミングをおこなえばよく、それ以外はたとえば通常の Scheme 言語のコードを記述するだけでよいため、プログラムの見通しがよくなる。

Flect における各モジュールは、計算中にユーザープログラムおよび言語システムによって参照される、計算の状態 (**メタ情報**) を言語に組み込むための構造である。まず、各モジュールは、おのおのが表現する言語の意味の構造によって、２つに分類される。

4.2.1 reflective-module

一つは、計算過程を通じて変化する状態をあらわす、特殊な値をシステムに組み込むためのモジュール (**reflective-module**) で、reflective-object に関する機能をあつかう。いいかえると、reflective-object とは reflective-module の列によって与えられた抽象的な構造であるといえる。

ユーザーは、atom、init、rest の各計算に関して、メタ情報に対する操作を記述する。これによってたとえばある時点までの計算コストや、変数の束縛情報を保持しておく環境などを定義することができる。また、その時点において計算対象となっている値そのものも、状態のひとつに含まれ、特殊なシステム組み込みのメタ情報 (**base**) として与えられる。これを利用することで実行中に計算対象となる値に関する計算の意味を変更することができる。

reflective-module は、プログラム中から構文 **import-reflective** によってシステムに組み込まれる。その際、その時点ですでに組み込まれている有効なモジュール (**アクティブなモジュール**) の影響はそのまま持ち越されることになり、新たなモジュールは、それま

でに定義されたモジュールのメタ情報にアクセスすることが可能である (同一の名前が存在した際は、プログラムの実行の流れにおいて最も新しく組み込まれた方が優先される)。このことで、例えば実行コストのカウントを利用するようなモジュールといったものが定義可能になる。

またシステムは、計算過程の全体を通じて有効な、基本となる reflective-module をあらかじめ組み込んでおり、前述の base は、そのモジュールのメタ情報として与えられている。そのためプログラム中に組み込まれるあらゆる reflective-module からの base へのアクセスが可能になる。

4.2.2 contrd-module

もう一つは、計算の流れを決定する新たな構造をユーザーがシステムに組み込むモジュールであり、プログラム中から構文 **import-control** によって control-object に組み込まれる (**control-module**)。

このモジュールにおいても、制御状態に関する情報を保存するための変数を用意することができる。例えば CPS における current continuation に相当するデータと考えてよい。control-module は、実行制御を決定する規則の他に、その時点でアクティブな reflective-module を含む計算に関するすべての情報を保持しており、本システムにおけるプログラムの実行は、アクティブな control-module を、計算自身を示す関数の間で受け渡していくことによって進められていく。前述の制御構造を規定する規則とはこの関数の呼びだしの流れを規定するものである。また reflective-module と異なり、control-module ではアクティブなモジュールとは最後に組み込まれたモジュールをさす。コントロールの流れは、常に最も新しく組み込まれた control-module に従う。つまり、ある時点における実行制御の形態は常に一種類しか存在しないことになる。

4.2.3 rest-caller

前述の二つのモジュールが実行コードに与える影響は、互いに独立しており両者が互いに影響を与えあうようなことはない。これはコードの安全性や、プログラミングのしやすさの面からいって良い性質であろうがこの両者の中間的な性格をもつような計算によって、より豊かな言語の拡張が可能なことはまちがいない。これはデータの表現形式と実行の制御形式にまたがるような計算、例えば非決定的な選択を許すような計算や、例外処理など

を実現する際に必要となる。

この種の拡張によって、reflective-module と object-module による拡張だけでは不可能なより広い意味での拡張機能を提供することができる。

そこで本システムでは、reflective-object に対して一つだけ、その計算状態に応じた実行制御をおこなうようなメカニズムを組み込む手段として rest-caller を提供した。

```
(inc x)                ;; 変換前

-> (gamma
     (alpha inc)         ;; init 部
     (lambda (g) ...))   ;; rest 部
```

図 4.3:

compound な計算において、その rest 部分では内部的に init 部分の計算結果を受け取って、残りの計算の呼びだしをおこなっている。この呼びだしの方法は基本的には固定的であるが、Flect ではこの操作 (**rest-caller**) を定義し、組み込むための操作を提供した。rest-caller からは、reflect-object の状態が参照可能であり、また引き数として rest 部の式そのものと、それに渡されるべき値が与えられる。ユーザーは、その時点の計算状態に応じた呼びだし方法を定義することが可能である。また、計算対象である値そのものを変更するような reflective-module と組み合わせてもちいることで、実行可能なデータそのものを拡張することが可能である。(例えば、計算の値として木構造を渡し、何らかの探索をおこなって選択的な呼びだしを提供するなど)

注意深く設計された reflective-module は、rest-caller を組み替えることによって多様な振る舞いを示す。また状況に応じて、組み込む rest-caller を変更することも可能である。このことで実行コードの再利用や、動的な言語の改変をおこなうことも可能である。

4.2.4 reify と reflect

ここでただ様々なモジュールを組み込むだけではユーザーができることは限定されている。モジュールの組み込みによって実現されるのは、言語に対する計算状態の追加のみであり、その状態とは純粋に実際の計算過程の背景として存在する情報である。たとえば、変数の束縛情報を保存していたとして、新たな変数束縛を追加したり、ある変数名に束縛されている値を得たりする手段がなければ何の意味もない。すなわち実際の計算過程において、ユーザーが意図的に計算状態に対するアクセスをおこなう手段が必要となる。

そこでシステムは、ユーザープログラムからのメタ情報への参照と変更を許すため、**reify**、**reflect** という構文を用意している。これは、現在アクティブな状態にあるモジュールのメタ情報を、直接計算対象として得るための仕組みである。この操作によってアクセス可能な構造はメタ情報のみであるが、あらかじめ適切なモジュールの設計を与えておけば非常に簡潔な操作によって自己改変が可能になる。

4.3 オブジェクトの表現

ここまでに示したことがらをまとめると、reflect-object と control-object とは以下のような性格をもったオブジェクトであると言うことができる。

- プログラム実行時のモジュールの組み込みに応じて、状態そのものが追加・削除される。
- 状態の追加・削除と同時に状態に対する操作の内容そのものが変化する。
- オブジェクトの状態は、ユーザープログラムから reify、reflect の呼びだしによってアクセスできる。

また、ある時点における各オブジェクトの性質を決定づけるアクティブなモジュールの定義は図 4.4に示したようになる。すなわちオブジェクトの表現とは、組み込み順に並んだモジュールの列であり、reflective-object とは、ある時点で組み込まれているすべての reflective-module の複合したものであり、また control-object は、列をスタック構造と考えた場合に、その時点でそのトップにある control-module そのものである。

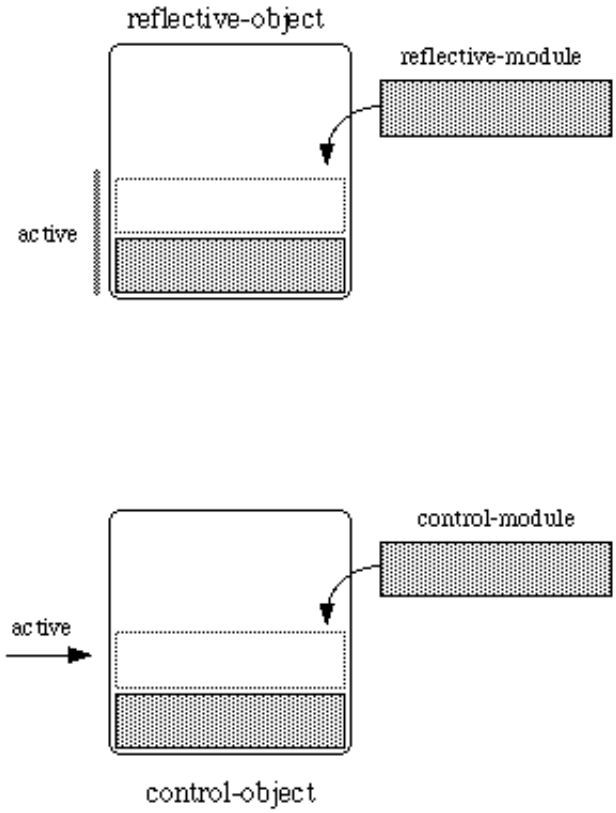


図 4.4: object model

4.4 構文

Flect は、Scheme を拡張した形式の言語であり Scheme の構文 [3] をそのまま継承している。Flec特有の拡張機能は、Scheme の構文にたいして次の構文を追加することによって実現している。

$$
\begin{array}{rcl}
exp & \rightarrow & (\texttt{import-reflective}\ variable\ binding\ body) \\
 & | & (\texttt{import-control}\ variable\ binding\ body) \\
 & | & (\texttt{embed-reflective}\ (binding\ dyn\ dyn\ dyn)\ bofy) \\
 & | & (\texttt{reify}\ (variable^*)\ exp) \\
 & | & (\texttt{reflect}\ (variable^*)\ exp) \\
 & | & (\texttt{reify-c}\ (variable^*)\ exp) \\
 & | & (\texttt{reflect-c}\ (variable^*)\ exp) \\
 & | & (\texttt{rest-caller}\ variable\ exp) \\
 & | & (\texttt{rmod-with}\ (variable\ binding\ variable)^*\ exp) \\
dyn & \rightarrow & (\texttt{dlambda}\ (variable^*)\ (variable^*)\ body) \\
rdef & \rightarrow & (\texttt{defrmod}\ variable\ (variable^*)\ dyn\ dyn\ dyn) \\
cdef & \rightarrow & (\texttt{defcmod}\ variable\ (variable^*)\ dyn\ dyn) \\
mdef & \rightarrow & (\texttt{defmetafunc}\ variable\ exp) \\
comp & \rightarrow & (\texttt{def-restcaller}\ variable\ dyn)
\end{array}
$$

4.5 各構文の意味

以下に先に示した Flect の構文の詳細な説明を与える。

4.5.1 reflective module

Flectシステムでは、言語の拡張を reflective-object と control-object の変更としてモデル化している。reflective-object を拡張する手段として、ユーザーは reflective-module という

構造を宣言し、実行中に組み込むための記述をおこなう。まず、モジュール宣言の記述について述べる。reflective-module は、そのモジュールによってシステムに組み込まれる状態をあらわす変数 (**メタ情報**) と、各計算過程において自動的に呼び出される、メタ情報に対する操作 (**reflective-operation**) を保持するような構造として表現される。以下にその記述方法を挙げる。

```
(defrmod mod-name (var_0 ... varm)       ;; メタ変数の宣言
  (dlambda (v) (x_0 ... x_l) operation)  ;; atom
  (dlambda () (y_0 ... y_m) operation)   ;; init
  (dlambda () (z_0 ... z_n) operation))  ;; rest
```

reflective-module の各 reflective-operation を定義する dlambda 式の宣言には以下に示す条件がある。

- ここで記述される式は、Flect のメタレベルを記述するもので、純粋な Schem の式と同様の評価がおこなわれる。
- 基本的に、各 reflective-operation では、メタ変数に対する副作用のみが重要になり、システムは meta-eorpat の返り値を無視する。
- 上から順に atomic な計算、compound な計算のはじめ (init) に計算する部分、そしてその結果を受けておこなわれる残りの計算 (rest) が呼ばれた直後における状態の変化を定義する。
- dlambda 式の第一引き数宣言部で与えられる引き数はあらかじめ決められている。上の atomic な計算が受け取る値は、atomic な計算に渡される値そのものを示す。
- 第二引き数宣言部において、その reflective-operation でアクセスするメタ情報の宣言をおこなう。
- アクセス可能なメタ情報は、現在定義中のモジュール内のメタ情報と、モジュールを組み込む時点でアクティブ (すでに組み込まれている) であると仮定した reflective-module のメタ変数。

こうして宣言したモジュールは、以下の構文で、プログラム中のある領域に組み込むことができる。ここでモジュールの組み込みは、計算の一部としては扱わない。

```
(import-reflective mod-name          ;;  定義済みのモジュールの読み込み
  ((var_0 val_0) ... (var_m val_m))  ;;  メタ変数の初期化
    body)                            ;;  影響を受ける部分
```

reflective-object の初期型としてあらかじめ基本的な reflective-module が組み込まれている。システムは、常に図 5.5に示した base に束縛された値を計算中の値を必要とする時点 (具体的には、compound な計算において計算結果を残りの計算に渡す際) で参照する。以下にシステムにあらかじめ組み込まれている、計算中の値に関する状態を定義する最も基本的なモジュールの具体的な内容を示す。

```
(defrmod basic (base)
  (dlambda (v) (base) (set! base v))
  (dlambda () () 'noop)
  (dlambda () () 'noop))
```

ここではシステムが現在計算対象としている値として参照するように決められている特殊なメタ変数 baseを宣言し、atomic な計算において、うけとった値を baseに代入している。前にのべた、メタ変数の優先順位に関する性質によって、あとからシステムに組み込まれたモジュールからも、baseはアクセス可能であり、重複した変更が加えられる可能性があるが、実際の計算にもちいられる値としてシステムがみなすものは、最終的な baseの値である。

また、直接プログラムコード中にモジュール定義を埋め込むことも可能である。ここで、各 reflective-operation の定義規則は、あらかじめ定義する場合と同様である。こういった組み込み手段をもちいることの利点とは、実行時の値を reflective-operation の**定義**に反映できるということである。実際の適用例は、図 4.6の実行例で示してある。

```
(embed-reflective
```

```
(((var_0 val_0) ... (var_m val_m))     ;; 宣言と初期化
(dlambda (v) (x_0 ... x_l) operation)  ;; atom
(dlambda () (y_0 ... y_m) operation)   ;; init
(dlambda () (z_0 ... z_n) operation))  ;; rest
```

こうして組み込まれたシステムの状態は、実行時に **reify**、**reflect** というオペレーションを呼ぶことによって、ユーザープログラムからのアクセスが可能である。

```
(reify (var_0 ... var_m) body)
```

reify は、現在の計算の状態を参照するための操作で、メタ情報(var_0 ... var_m) を同名の変数にバインドし、そのスコープ内で、body を評価する。ここで同名のメタ情報がアクティブな reflective-module 内に複数存在する場合は、最も新しいものが優先される。

```
(reflect (var_0 ... var_m) body)
```

reflect は、計算の状態に対して変更を加えるための操作で、変数(var_0 ... var_m) の内容を同名のメタ情報にバインドし、その影響下で、body を評価する。

例:実行コストの計算

以下に reflective-object に関する拡張の例を示す。

```
(defrmod cost-counter
  (ticks)
  (dlambda (v) (ticks) (set! ticks (+ ticks 1)))
  (dlambda () (ticks) (set! ticks (+ ticks 1)))
  (dlambda () () 'noop))

(define get-ticks
  (lambda ()
    (reify (ticks) ticks)))
```

図 4.5: モジュールの定義

図 4.5の最初の定義式は、reflective-object に組み込む module の定義であり、atomic な計算および initial な計算に際してカウンタticks を 1 加えることで、計算コストを得る。2番目の定義式は、ticks をプログラム中から参照するプロシジャを宣言している。

```
(define test0
  (lambda ()
    (import-reflective cost-counter ((ticks 0))
      (+ 3 (+ 1 2))
      (get-ticks))))

>(test0)
16

(define test1
  (lambda ()
    (import-reflective cost-counter ((ticks 0))
      (+ 3 (reify (ticks)
             (let ((res (+ 1 2)))
               (reflect (ticks) res))))
      (get-ticks))))

>(test1)
11

(define test-a
  (lambda (x)
```

```
    (embed-reflective (((ticks 0))
      (dlambda (v) (ticks) (set! ticks (+ ticks x)))
      (dlambda () (ticks) (set! ticks (+ ticks x)))
      (dlambda () () 'noop))
    (+ 1 2)
    (get-ticks))))

>(test-a 3)
33
```

図 4.6: 実行例

図 4.6は実際の適用例であり、test 0では単純なコストの計算を、test1 では (+ 1 2) の計算にかかるコストを除外している。その際、除外操作自身についてカウントされる計算コストは最後の res の評価にかかったコストのみである。これは、reflect された値が、reflect の返り値の評価部分以降有効であることによる。また、test-a ではカウンタの増加する値をユーザーの入力によって決定している。

例:値の意味の変更

以下では、計算する値そのものに変更を加えている。このモジュールが組み込まれた範囲では、すべての数値は、その値を 2 で割ったあまりとして扱われるため、2 値演算の範囲を指定することが可能である。

```
(defrmod binary-calc
  ()
  (dlambda (v) (base)
    (if (number? v)
        (set! base (remainder v 2))
```

```
        'noop))
  (dlambda () () 'noop)
  (dlambda () (base)
    (if (number? base)
        (set! base (remainder base 2))
        'noop)))
```

図 4.7: モジュールの定義

図 4.7における定義では atomic な計算と rest な計算に際して、値の変更をおこなっている。これは、リテラルとしての値そのものと、途中計算の結果にたいする操作とみなすことができる。

```
(define test2
  (lambda ()
    (+ 4 (import-reflective binary-calc ()
           (+ 2 3)))))

>(test2)
5
```

図 4.8: 実行例

図 4.8の例では、計算の途中で reflective-module の組み込みをおこなっている。つまり部分的な改変をおこなった例である。

4.5.2 contrd-mo dule

reflective-object と同様に、control-object を拡張する手段として、ユーザーは control-module という構造を宣言する。プログラム中、以下のような宣言をおこなうことで特定の範囲の実行制御の構造を規定することができる。このモジュールによって定義することができるのは、atomic 、init 、rest に相当する各計算が呼び出されるタイミングである。まず、モジュール宣言の記述について述べる。

```
(defcmod mod-name (var_0 ... varm)
  (dlambda (f) (x_0 ... x_l) operation)    ;; atom
  (dlambda (f g) (y_0 ... y_m) operation)) ;; compound
```

control-module の各 reflective-operation を定義する dlambda 式の宣言には以下のような条件がある。

- reflective-mo dule と同様に、返り値は必要としない。
- 上から順に atomic な計算、comp ound な計算に際しての制御に関する状態の変化、および、計算そのものを表現した関数の呼びだしによる実行の流れを定義している。
- dl ambda 式の第一引き数宣言部で与えられる引き数はあらかじめ決められている。
  - 上で atomic な計算のための操作を定義した dl ambda 式の f は、at omi cな計算自身を示す関数。
  - comp ound な計算のための操作における f は、initial な計算自身を示す関数。
  - comp ound な計算ぼ定義における g はその残りの計算を示す関数。

  つまり、各 reflective-operation は、atomic な計算と、comp ound な計算の流れそのものを、ユーザーに対して提供しているとみなすことができる。
- 各関数は、引き数として object-module を受け取り、結果としてアクティブな object-mo dul e を返す。
- 第二引き数宣言部でアクセスするメタ情報の宣言をおこなう。

- アクセス可能なメタ情報は、現在定義中モジュールのメタ情報と、特殊な変数 **self** そして、アクティブな reflective-module のリスト **objs** である。self は control-object そのものであり、objs は reflective-object そのものである。それぞれが意味するものは、現在アクティブなモジュールの情報である。
- 各計算を示す関数の呼び出しは、self を渡すことによっておこなわれる。
- 基本的に可能な操作は、渡された関数をメタ情報として保存しておくことで、呼び出しのタイミングを変化させることである。

宣言したモジュールが表現する制御構造は、以下の構文でプログラム中のある領域に組み込むことができる。

```
(import-control mod-name
  ((var_0 val_0) ... (var_m val_m))  ;; メタ情報の初期化
    body)
```

以下にシステムにあらかじめ組み込まれている、計算の制御の進めかたを定義する最も基本的なモジュールを示す。

```
(defcmod main ()
  (dlambda (f) (self) (f self))
  (dlambda (f g) (self) (g (f self))))
```

図 4.9: 基本的な制御構造

図 4.9示したように、基本となる計算の進め方は単に決まった順番どうりの各関数の呼びだしである。

例:continuation-passing

control-object の変更の例として CPS(継続渡し形式) の制御構造を組み込む例を挙げる。CPS に関しては、[9] を参考にした。以下にそのための control-module の定義を与える。

```
(defcmod cps
  (cc)
  (dlambda (f) (cc self)
    (f self)
    (letrec ((loop (lambda ()
                     (if (pair? cc)
                         (let ((x (car cc)))
                           (set! cc (cdr cc))
                           (x self)
                           (loop))))))
      (loop)))
  (dlambda (f g) (cc self)
    (set! cc (cons g cc))
    (f self)))
```

図 4.10: モジュールの定義

図 4.11に示すように、定義 4.10で示した制御構造は、木構造の最外探索と等しい。つまり、atomic な計算に行き当たるまで、計算を cc へ保存しておくことで遅らせておいて、atomic な計算に行き当たった時点で、呼び出すような制御構造を実現している。つまり、常にその時点での残りの計算を cc に保存している状態にあるわけである。

以下では、図 4.7の CPS モジュールを利用して call/cc(call-with-current-continuation) を定義する例を与える。そのためには、次の二つの構文について言及する必要がある。control オブジェクトに関しても reify-c、reflect-c の呼びだしによって制御情報へのアクセスを提供している。

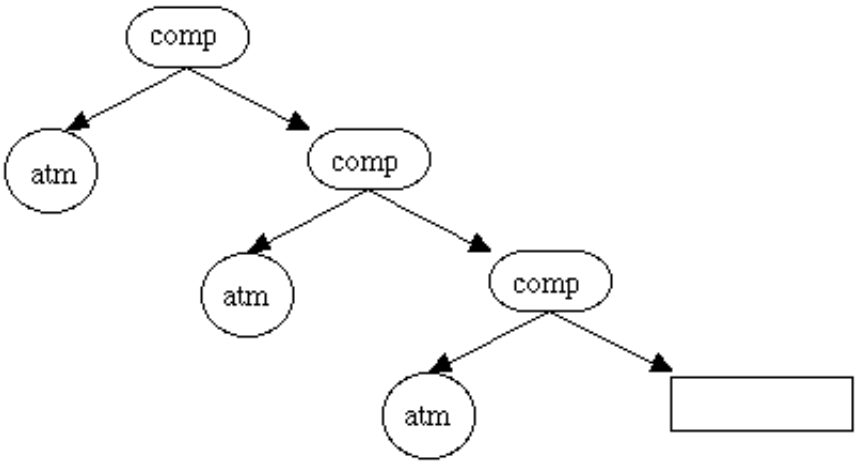


図 4.11: CPS control

```
(reify-c (var_0 ... var_m) body)
```

reify-cは、メタ情報(var_0 ... var_m) を同名の変数にバインドし、そのスコープ内で、body を評価する。

```
(reflect-c (var_0 ... var_m) body)
```

reflect-c は、変数(var_0 ... var_m) を同名のメタ情報にバインドし、その影響下で、body を評価する。

```
(define call/cc
  (lambda (f)
    (reify-c (cc objs)
      (f (lambda (v)
           (reflect-c (cc objs) v))))))
```

図 4.12: call/ccの定義

call/ccの実現は、図 4.12に示したように、call/ccが呼ばれた時点の cc を reflect する関数を、Scheme の current-continuation のように与えている。たとえば、cc に変数や定数と

いった値が渡された場合、制御の流れが atomic な計算に行き着いたことになり、そのまま保存されていた残りの計算が呼び出されることになる。以下がその実行例である。

```
(define test3
  (lambda ()
    (import-control cps ((cc '()))
      (let ((r (lambda (k) (k 'c))))
        (cons 'a (cons 'b (call/cc r)))))))

>(test3)
(a b . c)
```

例:簡単なモジュールの組み合わせ

reflective-object と control-object は、互いに独立した言語機能を実現しており、互いに関連しあわないため、容易に組み合わせることが可能である。例えば以下のように CPS の制御構造のなかで、2 値演算をおこなうことが可能になる。

```
(define test4
  (lambda ()
    (import-reflective binary-calc ()
      (import-control cps ((cc '()))
        (+ 1 (double 3))))))

>(test4)
0
```

4.5.3 reflective-module と制御との関連づけ

プログラムの流れによって決まるような、システムの状態に応じた実行制御をおこないたい場合 (例えば、例外的な処理が発生したという状態に応じた実行制御をおこないたい

場合など) に対応するために、本システムでは rest-caller という操作の組み込みを提案した。これは現在アクティブな reflective-module に対して、特別な制御構造を関連づけるために、あらかじめ rest-caller と呼ぶオペレーションを定義しておき、実行時にシステムに組み込むことで、システムの状態と制御構造にまたがるような意味構造を追加することを可能とする。

```
(def-restcaller name
  (dlambda (f v) (x_0 ... x_l) operation))
```

rest-caller を定義する dlambda 式の宣言には以下のような条件がある。

- dlambda 式の第一引き数宣言部で与えられる引き数はあらかじめ決められている。
  - 第一引き数の f は、control-module において関数 g で示されている残りの計算の実体であり、gamma の第二引き数であたえられたラムダ式そのものである。
  - 値 v は、その式に渡されるべき値である。
- 第二引き数宣言部でアクセスするメタ情報の宣言をおこなう。
- アクセス可能なメタ情報は、現在アクティブな reflective-module のメタ変数。

システム初期状態ではあらかじめ以下の rest-caller が与えられている。

```
(def-restcaller org (dlambda (f v) (f v)))
```

set-caller 文によって、現在組み込まれている rest-caller を任意のユーザー定義の rest-caller に組み替えることが可能であり、その影響は reflective-module に準ずるが、control-module と同様に有効な rest-caller は最後に組み込まれたもののみである。

```
(set-caller rest-caller-name body)
```

rest-caller は特定の reflective-module と関連させて用いられることが多い。そこで特別に両者を組み合わせて宣言する以下の構文が与えられている。

```
(rmod-with
  (rmod-name ((var_0 val_0) ... (var_m val_m)) rest-caller)
    body)
```

例:nondeterministic choice

rest-caller を利用した例として非決定的な選択を許すような構造を、計算システムに組み込むための記述例をしめす。

```
(defrmod nondeterminal-r
  ()
  (dlambda (v) (base)
    (set! base (list 'nondet v)))
  (dlambda () () 'noop)
  (dlambda () () 'noop))

;; 特殊なリスト (nondet によってタグ付けされたもの) の平坦化

(def-metafunc concat
  (lambda (lst)
    (if (null? lst)
        '()
        (let ((elm (car lst)))
          (if (and (pair? elm) (eq? (car elm) 'nondet))
              (append (cdr elm) (concat (cdr lst)))
              (cons elm (concat (cdr lst))))))))

;; リストで表現された複数の選択に対して、計算をマップする。
```

```
(def-restcaller nondet-caller
  (dlambda (f v) ()
    (cons 'nondet (concat (map f (cdr v))))))

;; ユーザーへのインターフェイスの定義

(define amb (lambda (a b) (list 'nondet a b)))
```

まず、reflective-module において、計算の対象とする値を変更している。すべての値は、シンボル nondet によってタグを付けられたリストとして表現することになり、そのリストの要素が非決定的選択の選択肢になる。rest-caller は、この値を実際の計算に渡すための規則を定義しており、値 $(nondet x_0 \ \ldots \ x_n)$ の各 x_i に対して、残りの計算を示す関数 f を適用している。concat は、今回定義したデータ表現のための特殊なもので、結果の値を再構成するために利用している。concat の定義にもちいた構文(def-metafunc ...) は、reflective-operation の内部から呼ばれる補助的な関数を定義するために用いるための完全に Scheme の semantics に従うような、変換を施されない関数を定義する。

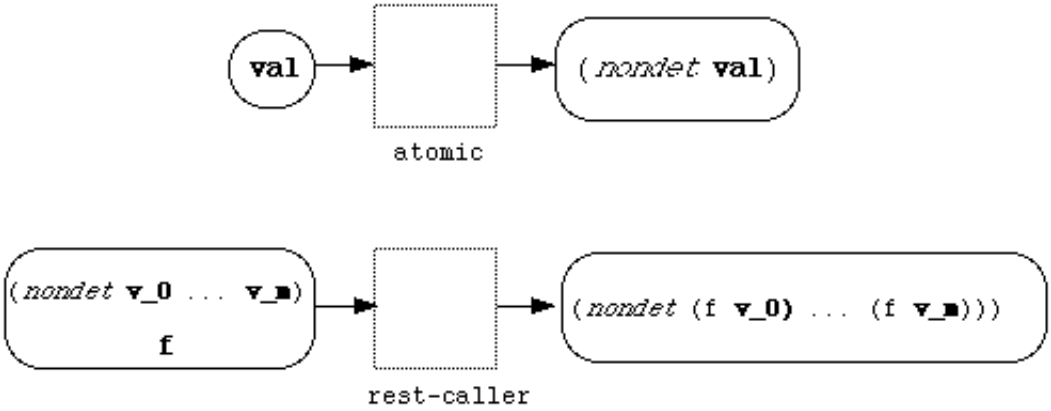


図 4.13: Nondeterministic

```
(define test5
  (lambda ()
```

```
      (rmod-with (nondeterminal-r () nondet-caller)
        ((amb + -) (amb 3 4) (amb 1 2)))))

>(test5)
(nondet 4 5 5 6 2 1 3 2)
```

例:例外処理

計算状態と、制御構造の組み合わせに関する別の例として、例外処理機構の実現について解説をおこなう。まず、計算の状態としてメタ情報 state を用意する。計算中、state は通常の式の評価中の状態 (Return) か、もしくは例外的な状態から復帰をおこなおうとする段階 (Raise) のどちらかにある。

rest-caller は計算を進めるために state を参照し、状態が Return であれば通常の計算をおこなうが、Raise 状態に変わると exception-caller の範囲から出るまで残りの計算を呼ばず、計算からの脱出を試みる。

```
(def-restcaller exception-caller
  (dlambda (f v) (state)
    (if (eq? state 'Return)
        (f v)
        v)))

(defrmod exception
  (state)
  (dlambda (v) () 'noop)
  (dlambda () () 'noop)
  (dlambda () () 'noop))

(define raise
  (lambda (mess)
```

```
      (let ((state 'Raise))
        (reflect (state) mess))))
```

state へのアクセスには、reflect をもちいる。以下に 0 による割り算についての例外処理をあつかった実行例を示す。

```
(define div
  (lambda (x y)
    (if (eq? y 0)
        (raise "zero divide")
        (/ x y))))

(define test6
  (lambda (x y)
    (rmod-with (exception ((state 'Return)) exception-caller)
      (+ 1 (div x y)))))

>(test6 1 2)
1/2
>(test6 1 0)
"zero divide"
```

図 4.14: 例外処理の実装

図 4. 14の例で示した例外処理は、ある指定し範囲において有効な処理である。つまり、特定の範囲からの脱出手段を提供していることになる。[2]これは、Flect の部分的な自己改変の特徴的な例であるといえる。

[2] ここで例外処理の返り値は、指定した領域の外からみればただのデータである

4.6 実行過程の概要

まず、各々のオブジェクトが実際の計算についてどのように参照されるのかを示す。

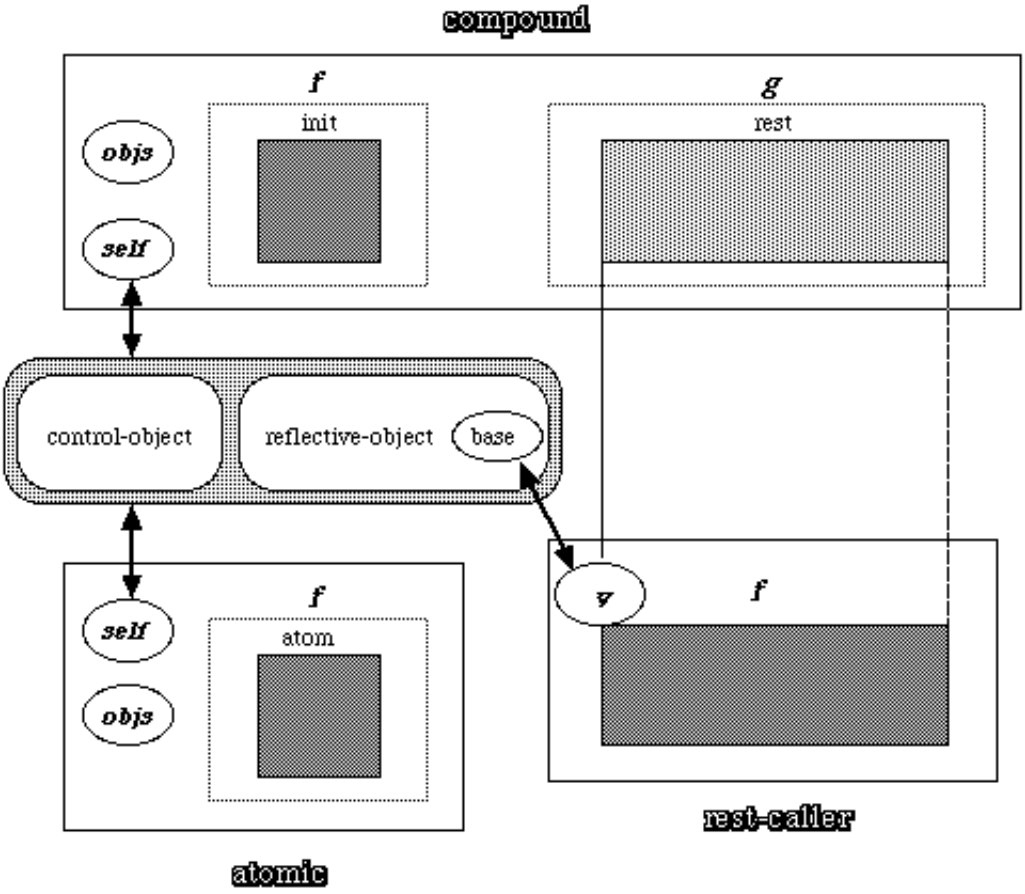


図 4.15: オブジェクトの各計算における参照

図4.15で示すように、各オブジェクトの参照(reflective-operationの呼びだし)は、atomicな計算とcompoundな計算に際しておこなわれる。

reflective-objectに関しては、図4.15における点線で囲まれた部分の実行に際してその計算の種類に応じたreflective-operationがアクティブなモジュールすべてについて組み込み順に呼ばれる。reflective-moduleにおける操作の本質は、メタ情報の更新であるので、返り値は無視して単純に操作を呼びだすだけである。

control-objectの参照がなされるのは、図4.15の白抜き文字で示したatomicとcompoundである。control-objectにおけるそれぞれの操作(atomとcomp)は、そのまま計算の流れそのもの(ここではfやgの呼びだし)を表現している。Flectにおける計算は、control-object

と reflective-object の組 (すなわち、計算に関するすべての状態を含んだ構造) を、各計算の間で受け渡していくことですすめられるようになっている。制御に関する操作の記述とは、この構造 (メタ変数 self で与えられる) を計算のあいだで受け渡すタイミングの定義に他ならない。control-object のアクティブなモジュールは、今回の設計では最新のものに限定されているが、すべてのモジュールの影響を有効にして、複数の計算の流れを生成することも可能である。しかし、実行効率の面からも、そのような適用をおこなう必然性は現在のところ考えられない。

4.6.1 前準備

これまでに、Flet 言語の意味のオブジェクトによる表現と、そのモジュールによる拡張について述べた。本節では、それらのモジュールが実際いかにしてプログラムの実行に影響をあたえるのかについて例をあげて解説をおこなう。

まず前提として control-object と rest-caller は、デフォルトのものをそのまま利用し、あらかじめ次のモジュールが定義され、組み込まれているものとする。(control-object や rest-caller の適用については第 5 章で詳しくのべることにし、ここでは実行過程の概要を示すのみとする。)

```
(defrmod big-num (save)     ;; メタ情報の宣言
  (dlambda (v) (save)       ;; atom 計算に関する更新操作
    (if (number? v)         ;; 値が数値ならば、いままでの最大値を更新
        (set! save (max save v))))
  (dlambda () () 'noop)     ;; init 計算に関する更新操作
  (dlambda () () 'noop))    ;; rest 計算に関する更新操作
```

save の初期値を 0 として、次のプログラムを実行する際の流れを追っていく。まず、次のように変換がなされる。

```
(+ 6 x)


(gamma (alpha +)
```

```
        (lambda (g1)
          (gamma (alpha 6)
                 (lambda (g2)
                   (gamma (alpha x)
                          (lambda (g3) (g1 g2 g3)))))))
```

4.6.2 各関数の説明 (gamma)

alpha とは atom 計算そのものを、gamma とは comp ound 計算そのものを示す関数で、内部的にモジュール定義において与えられた各操作を呼び出す。

まず、gamma とは、二引き数の関数で、評価結果として、クロージャを返す。このクロージャは、control-object を受け取り、control-object を返すもので、我々はこの関数を **computation** と呼ぶことにする。control-object には、reflective-object を含むすべての計算状態が保存されているため、computation とは計算状態の更新過程をまとめたものと考えることができる。

特に、gamma を評価した結果として得られた computation は、**comp ound な computation** とよぶ。gamma の受け取る引き数とは、comp ound な計算の init 部と rest 部に対応するコードで、第一引き数として init 計算をあらわす computation を、そして第二引き数として、init 部で得られた結果 (baseの値) を受け取って、computationを生成するようなクロージャを示す。

comp ound な computation の概要を次に示す。

1. まずアクティブな reflective-module における init 操作を、組み込まれた順に呼び出す。
2. 更新後の control-object を、第一引き数であたえられた init計算に対応する computation に渡す。
3. さらにアクティブな reflective-module における rest 操作を、組み込まれた順に呼び出す。
4. メタ情報のうち、baseにバインドされた値を、第二引き数のクロージャに渡す。

5. クロージャの呼び出しの結果が computation ならば、control-object を渡し残りの計算をつづけ、そうでなければ、それを最終的な結果として base に保存する。

6. 最終的な control-object を返す。

4.6.3 各関数の説明 (alpha)

同様に、alpha は、一引き数の関数で、computation を返す **(atomic な computation)**。alpha の受け取る引き数とは、atomic な計算に対応する値で、定数やクロージャである。

atomic な computation の概要を以下にしめす。

1. アクティブな reflective-module における atom 操作を、組み込まれた順に呼び出す。その際、常にメタ情報 base に、値 v をバインドする操作が優先的に呼ばれる。

2. 最終的な control-object を返す。

4.6.4 実行の流れ

例題で示したプログラムの実行過程を順をおって解説する。全体的な構成は、図 4.16 を参照するとよい。

1. まず、gamma 式の二つの引数が評価される。第一引数の評価結果は、"+" に関する atomic な computation であり、第二引き数は、残りの計算そのものを含んだクロージャである。gamma の評価結果は、この二引数で与えられた computation に関する compound な computation である。

2. 前述の compound な computation に対して、初期状態 (save=0) を含んだ control-object が渡される。compound な computation の内部では、まず init 計算に関するすべての操作がよばれる。今回は、なんら更新はおこなわれないので、状態の変化は生じない。

3. control-object は、そのままあたえられた atomic な computation にわたされる。

4. atomic な computation の内部では、atom 計算に関する操作が、古い順に呼ばれる。まず、システムがあらかじめ用意しているモジュールの操作呼びだしがおこなわれ、特殊なメタ情報 base に、プロシジャ "+" がバインドされる。

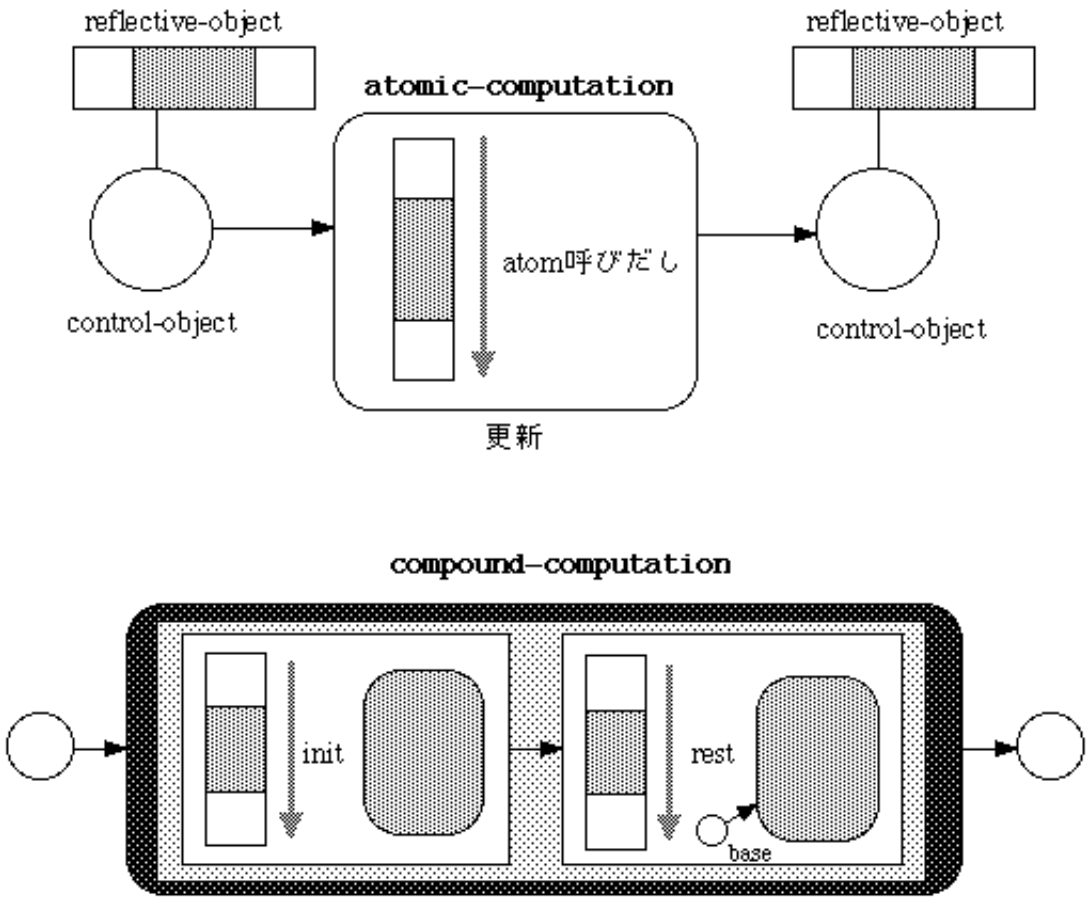

reflective-object
reflective-object
atomic-computation
atom呼びだし
control-object
control-object
更新
compound-computation
init
rest
base

図 4.16: 計算の流れ

5. そののち、今回組み込んだモジュール big-num についての呼びだしがおこなわれる。今回は、条件式(num? v) が偽なので、なにも起こらない。

6. 次に、残りの計算に入るまえに、rest 操作の呼びだしをおこなう。ここにおいても何ら影響はない。

7. 6 の結果として得られた base を、残りの計算を示すクロージャに渡す。その結果は、この例題では compound な computation であり、こうして得られた computation に対して現在の control-object を渡す。

8. 続きの計算は、ここまで述べてきた操作の繰り返しで、最終的に、残りの計算が computation を返さなくなった時点(g1 g2 g3) の評価結果を base にバインドして、その control-object を変えす。その際、save には、計算に使われた値 (途中経過は含まない) の最大値が保存されている。

4.7 モジュール結合

本章では、各モジュールの言語への組み込みによる意味の拡張について、述べてきた。本研究の特徴は、これらのモジュールの安全な結合の提供である。そこで、これまでに述べてきた各モジュールの結合に関することがらについて整理する。

Flect では、計算状態を表現する構造を、制御に関するオブジェクトと、そのほかの計算状態に関するオブジェクトという、互いに完全に独立した意味を実現するオブジェクトによって提供する。両者を完全に分離した理由は、それぞれのオブジェクトの拡張を記述するためのモジュールの機能を明確にし、安全な結合を実現するためである。

このような設計によって、以下に示すモジュール結合の枠組みを提供した。

- reflective-object を実現するモジュールは、自由に結合 (同時期に言語へ組み込む) 可能である。これは、reflective-module の操作対象が、control-object とは関係しない計算状態であるからである。

- 任意の control-object と、reflective-module との結合が可能。

- 任意の control-object と、reflective-object、および rest-caller の結合。control-object が表現するのは、制御の流れであり、rest-caller が定義するのは、複合的 (compound)

な計算における残りの計算 (rest) の呼びだし方法である。rest-caller は、完全に control-object の rest のなかに完全に組み込まれており、control-object はその内容に関しては何ら考慮する必要はない。rest-caller は reflective-module との関連のみを扱う。

このように、各モジュールの安全な結合は、各モジュールの実現する構造を、それぞれ独立したものに制限することによって実現されている (本章で示した各モジュールの関係は章末の図 4.17に示す)。さらに、Flect の有利な点は、モジュールの組み込みを、プログラム実行のある特定の範囲を指定しておこなうことが可能という点である。すなわち、ユーザーはモジュール同士の相性を考慮して、同時に組み込まれるべきモジュールを使い分けることが可能である。

ここでモジュール間の結合のうち、特に困難な例が、制御構造と他の計算状態との関連である。そこで以降では、こうした結合が問題となる例を示し、Flect における解決方法について述べる。

4.7.1 contrd-module と reflective-module の結合

control-object は、常に制御に関する計算状態として現在アクティブな reflective-module の集合を保持している。これは、モジュールの有効範囲がプログラム実行時の時間的な範囲として与えられているため、その状態 (アクティブなモジュールの列) が実行の流れに依存して変化するからである。

たとえば、import-reflective によって形成された範囲から、call/cc を使って脱出する場合、組み込まれた reflective-module は、モジュール組み込みの影響範囲の定義から無効にされるべきである。たとえば以下のような例がある。

```
1: (define compose
2:   (lambda ()
3:     (import-reflective cost-counter ((ticks 0))
4:       (import-control cps ((cc '()))
5:         (fact 5)
6:         (call/cc (lambda (k)
7:                    (import-reflective cost-counter ((ticks 0))
```

```
8:                          (k (+ 2 3)))))
9:          (get-ticks)))))
```

ここでは、2重にcost-counterを組み込んでいる。Flectの設計では、このような適用は、それぞれの範囲について各モジュールが独立してコストの計算をおこなう。問題は、ふたつめのimport-reflective(7行目)の範囲からcall/ccをもちいた脱出をおこなっている部分である。この範囲から脱出する場合、2番目のモジュール組み込みは除去されてしかるべきである。

そのため、call/ccの実装においては、残りの計算を示す構造(cc)のみではなく、アクティブなreflective-module の集合もまた、復帰させる必要がある。

Flect では、こうした問題を解決するため、制御に関する状態の一つとしてアクティブなreflective-module の集合(objs)をユーザーから参照可能とした。ユーザーは、call/cc以前のアクティブなreflective-module を参照(reify)、保存しておき、継続が呼ばれた際に復帰(reflect)するような記述をあたえることができる。

このような構造は、reflective-module に対してなんらかの影響を与えているかのように見えるが、実際には純粋に制御に関する状態操作であり、reflective-module の内容については一切影響を与えない。例えば今回の例では、外側のcost-counterの内容は、計算の全過程において保護されており、get-ticksによってその最終的な値を得ることができる。[3]このことだけでも、control-moduleが制御に関する状態変化のみを実現していることがいえる。つまり安全な結合のための機構を提供しているといってよい。

[3]get-ticksは最も新しい、**アクティブな**cost-counterの計算状態(ticks)を参照するものである

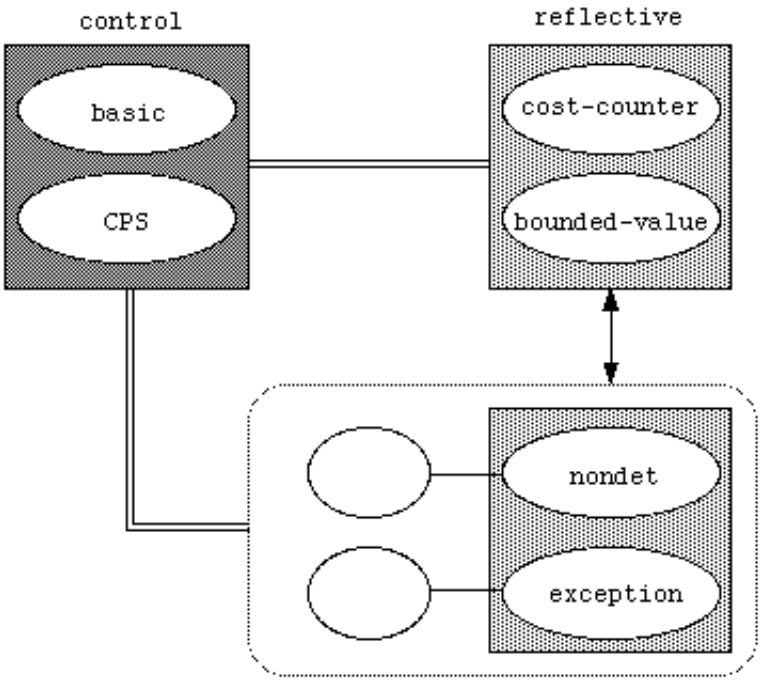


図 4.17: 各モジュールの結合

第 5 章

実装

本章では、Flect システムの実装に関する詳細な解説をおこない、具体的な実行過程について示す。

5.1 システムの構成

Flect システムは、基本的には Flect のソースコードを Scheme のコードに変換するトランスレータと、実行時に呼ばれる補助的なシステム、および変換されたコードをコンパイルし、実行するための処理系からなる。この処理系は、既存の Scheme のコンパイラが使われる。今回の実装では処理系として Macintosh 上で動作する Scheme コンパイラである Mac Gabimt 2.2 をもちいたが、処理系に依存する部分は、コードのコンパイルとその読み込みに関する部分だけなので、他のシステムに対する移植は容易である。

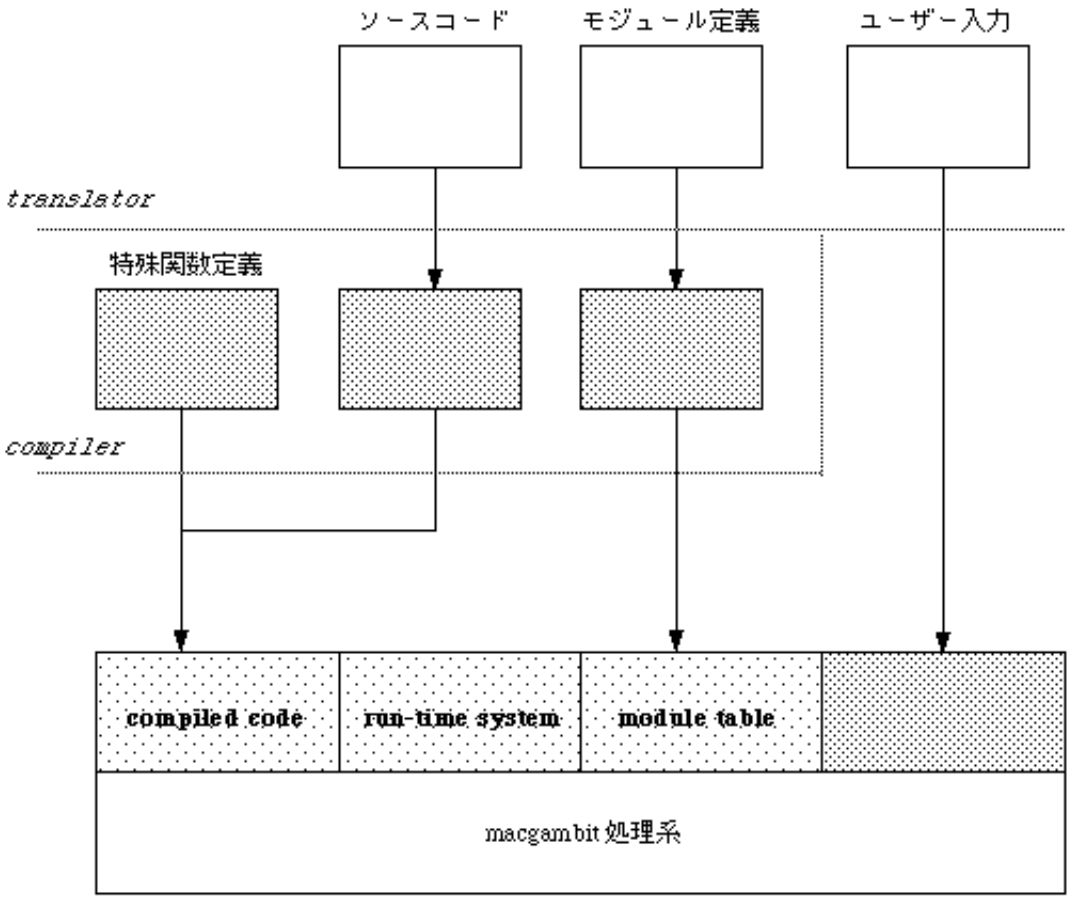


図 5.1: システムの構成

図 5. 1に示すように、Flect に与えられるコードの形態は 4 種類ある。

- 変換がほどこされたユーザーからの入力。
  これは、トップレベルでの通常のユーザー入力を示す。

- 変換がほどこされたコンパイル済みのファイル。
  これは、compile 文によってあらかじめコンパイルされたファイル中の通常の式である。

- 変換をおこなわないユーザーからの入力。
  これはトップレベルにおいて def-metafunc によって定義された式で、変換されずにベースとなるシステムによって直接評価される。

- 変換をおこなわないコンパイル済みのファイル。

これは compile されたファイル中で、def-metafunc によって定義された式。

5.2 プログラムの変換

Flect の処理系は、基本的に Fleco の拡張構文から、通常の Scheme 言語のコードへの変換によって実現されている。Fleco のソースコードおよび、ユーザーからの入力は、atomic な計算と compound な計算に応じて以下に示すような関数の組み合わせによる表現として変換を施される。

(alpha x) atomic な計算を表現する形式で、x はその値。

(gamma x y) compound な計算を表現する形式で、x は、部分式のうちではじめに評価される式を示し、y はその結果をうけて呼ばれる、残りの計算をあらわす関数を示す。

以降では Fleco の各構文に関する変換規則を示す。

補助的な宣言

$$
\begin{aligned}
c &: constant \\
v, x &: symbol \\
id &: number \\
e &: expression \\
m &: module\ \ e \\
cm &: rest - caller \\
dec &: ((v_0\ e_0)\ \ldots\ (v_n\ e_n)) \\
forms &: (x_0\ \ldots\ x_n) \\
body &: [e_0\ \ldots\ e_m] \\
T_s \to [e_0\ \ldots\ e_n] &: (gamma\ \ T(e_0)\ (lambda\ \ (g_0)\ (gamma\ \ \ldots\ (lambda\ \ (g_n)\ g_n)))) \\
T_ss \to [e_0\ \ldots\ e_n] &: (gamma\ \ T(e_0)\ (lambda\ \ (g_0)\ (gamma\ \ \ldots\ (lambda\ \ (g_n)\ (system\ g_n))))) \\
T_e \to dec &: ((cons\ \ v_0\ e_0)\ \ldots\ (cons\ \ v_n\ e_n)) \\
D : form\ body &\to (dlambda\ \ (ctr)\ form\ T_s(body))
\end{aligned}
$$

通常の式

$$
\begin{aligned}
T : c &\to (\texttt{alpha}\ c) \\
T : x &\to (\texttt{alpha}\ x) \\
T : (lambda\ \ forms\ \ body) &\to (\texttt{alpha}\ (lambda\ \ forms\ \ T_s(body))) \\
T : (if\ e_1\ e_2\ e_3) &\to (gamma\ \ T(e_1)\ (lambda\ \ (r)\ (if\ r\ T(e_2)\ T(e_3)))) \\
T : (set!\ v\ e) &\to (gamma\ \ T(e)\ (lambda\ \ (r)\ (set!\ v\ r))) \\
T : (begin\ e_0\ \ldots\ e_n) &\to T_s([e_0\ \ldots\ e_n]) \\
T : (let\ dec\ body) &\to T(((lambda\ \ (v_0\ \ldots\ v_n)\ body)\ e_0\ \ldots\ e_n))
\end{aligned}
$$

自己反映計算のための特殊な操作

$$T:(import-reflective\ m\ dec\ body) \rightarrow (local-reflective\ T_s(body)\ T_e(dec)\ atm\ ini\ res\ i$$

$$T:(impor\ t-control\ m\ dec\ body) \rightarrow (local-control\ T_s(body)\ T_e(dec)\ catm\ ccom\ id)$$

$$T:(embed\ -reflective\ (modul\ e-dec)\ body) \rightarrow (local-reflective\ T_s(body)\ T_e(dec)\ atm\ ini\ res\ i$$

$$T:(rest-caller\ cmbody\ ) \rightarrow (chang\ e-caller\ cm\ T_s(body))$$

$$T:(set-caller\ (m\ dec\ cm)\ body) \rightarrow (impor\ t-caller\ cm\ T_s(body)\ T_e(dec)\ atm\ ini\ res$$

$$T:(reify\ forms\ \ body) \rightarrow (lambda\ \ (c)\ (copy-retainf\ \ o\ c\ forms\ \ D(forms$$

$$T:(reflect\ forms\ \ body) \rightarrow (lambda\ \ (c)\ ((T(body))\ (assign-retainf\ \ o\ c\ for$$

$$T:(reify-c\ forms\ \ body) \rightarrow (lambda\ \ (c)\ (copy-control\ c\ forms\ \ body\ D(forr$$

$$T:(reflect-c\ forms\ \ body) \rightarrow (lambda\ \ (c)\ ((T(body))\ (assign-control\ c\ forms$$

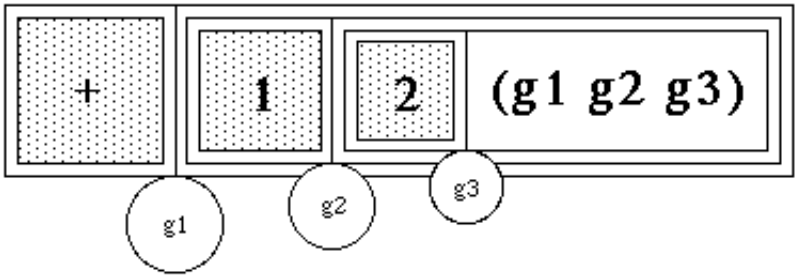


図 5.2: 変換

図 5.2に示すように、この変換によってソースコード中の式の、各構文要素が、alpha と gamma によって抽象化された計算 **computation** に置き換えられるのである。

atomic な計算と compound な計算の意味は、alpha関数と gamma 関数の振る舞いによって決まり、それぞれの関数の内部的な動作を決定するのが実行時に参照される reflective-object、control-objectおよび rest-callerである。いわば alpha 関数と、gamma 関数とは、それぞれ実行時に組み込まれるオブジェクトによって完成するような atomic な計算と compound な計算の雛型であるとかんがえることができる。

この変換によって拡張の影響がプログラムにたいして反映されるための準備がととのったことになる。以降では、この computation を表現するための関数 alpha と gamma の実装について解説をおこなう。

5.3 dlambda 式について

Flect の実装において重要な役割をもっている要素として **dlambda** 式がある。これは疑似的に動的なスコープをもつ変数を提供する lambda 式と考えるとよい。Fleaの処理系は、この式をプログラム変換によって実現することを可能にしたため、変換ベースでの処理系の容易な実装を可能にした。ここで示す実現方法は他の変換ベースの処理系においても適用可能であり、様々な応用が可能である。Fleaの実装に際してこの式をもちいた目的は、変換ベースの言語におけるオブジェクトの実装である。まずこの式が示す機能について解説をおこなう。

dlambda 式の働き

通常 lambda 式が評価された結果として得られる closure は、評価時の環境を保持していて、その body 部の評価はその環境について閉じている。これに対して dl ambda 式では、body 部の評価時に参照する環境を部分的に解放している。つまり、ある特定の変数に対する変更が dl ambda 式をまたいで有効になる。以下にその具体例を示す。

```
(define test
  (let ((y 0))
    (dlambda (x) (y) (+ x y))))
```

ここで test にバインドされるのは、引き数 x を受け取って closure を返すようなプロシジャである。この closure は、引き数として変数のバインド情報を示す a リスト (**D-環境**) を受け取る。

```
> (define env (list (cons 'y 2)))
(y . 2)
> ((test 1) env)
3
((test 1))
1
```

dl ambda 式の第二引き数宣言部の変数 **D-変数**の値は、D-環境から検索される。あとは通常の closure の lexical-scop e と同じで、第一引き数が参照され、外部での変数宣言が最後に参照される。さらに、dl ambda 式の変数への代入は、そのまま D-環境に反映される。これを利用してオブジェクトの state と metho d を疑似的に実現することも可能である。

ここで興味深いのは、ある環境を共有するような関数を呼び出す場合である。これによって変数の束縛情報を動的に扱うことが可能になる。例えば以下では第一宣言部の引き数として D-環境そのものを渡している。

```
> (define env '((y . 0)))
```

```
> (define test1
    (dlambda (e) (y)
      (set! y 2)
      ((test2 1) e)))

> (define test2
    (dlambda (x) (y)
      (+ x y)))

> ((test1 env) env)
3
```

ここで、もし呼び出された側で D-環境の内容を変更した場合、その結果は呼び出し側の D-変数にも反映される。

dlambda 式の実現

dlambda 式は、通常の Scheme のプログラムへの変換が可能である。以下にその変換例を示す。ここで変数名 gx は、他の変数と違う名前であるとする。

```
(dlambda (x y) (z) (+ x y z))

(lambda (x y)
  (lambda (g0)
    (let ((z (let ((g1 (assq 'z g0)))
               (if g1 g1 z))))          ;; D-環境から探す。見つからない
                                        ;;     場合は通常の scope に従う。

      (let ((g2 (+ x y z)))             ;; 計算をおこない、
        (let ((g1 (assq 'z g0)))        ;;       その結果を一時的に保存。
```

```
        (if g1 (set-cdr! g1 z)))      ;; 副作用のD-環境への反映
    g2))))                            ;; 復帰
```

ここでは示さなかったが、body のユーザー定義関数呼び出しの前後においても D-環境の更新をおこなっている。この実装方法については同名の lexical な変数と dynami cな変数が混在する場合に関して問題点がある。これは、内部で新たに変数を宣言する際に生じる。解決方法としては、変数宣言時に名前をチェックして、重複する場合にはその時点で更新をおこなって関数呼び出し前後の際の更新をおこなわない方法をとる。

また、変換に際して無駄な記述が多いという問題点がある。Flect においては dl ambda 式の呼び出しは頻繁におこなわれるため、効率的な変換コードの生成は必須である。

5.4 alpha と gamma の実装

al phaと gamma は、それぞれ atomi c な計算と comp ound な計算を抽象化して提供するための関数であり、その振る舞いは reflecti ve-object と control-objectによって決定される。そこでまず、reflective-object と control-objectがどのように実現されているかについて解説をおこなう。

5.4.1 reflective-object

まず、reflective-objectの実現方法について述べる。reflecti ve-ob ject とは、アクティブな reflecti ve-mo dule の列によって表現される構造である。

reflecti ve-mo dule は、図 5.3に示すように、そのモジュールによって reflecti ve-ob ject に組み込まれる reflecti ve-op eration と、アクセス可能なメタ情報を保持するための環境からなる。ここで reflecti ve-op eration は、前節でのべた dl ambda 式によって実現されており、、メタ情報を保持しているのが D-環境と考えればよい。

reflecti ve-ob ject は、図 5. 4に示すように、複数の reflecti ve-mo dul e の列によって決まる。システムは、常にアクティブな reflecti ve-mo dule の列を保持しており、プログラムの実行時に最も古く組み込まれたモジュールから順に変換時に得られたシステム関数 (al pha,gamma) の呼び出しに応じて適切な reflecti ve-op eration を呼び出す。

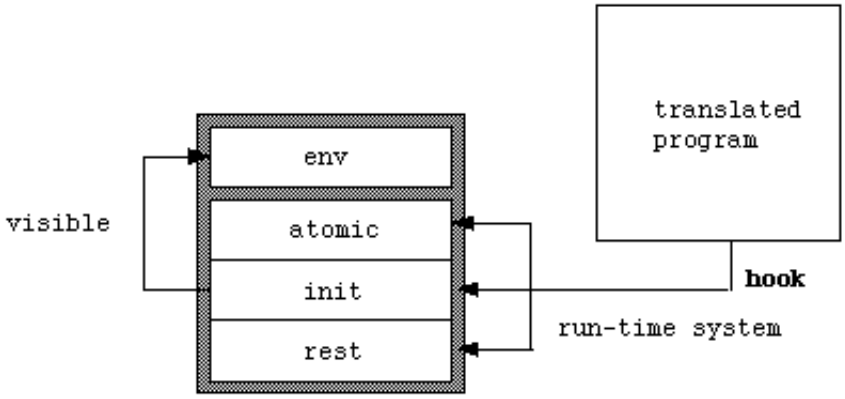


図 5.3: reflective module

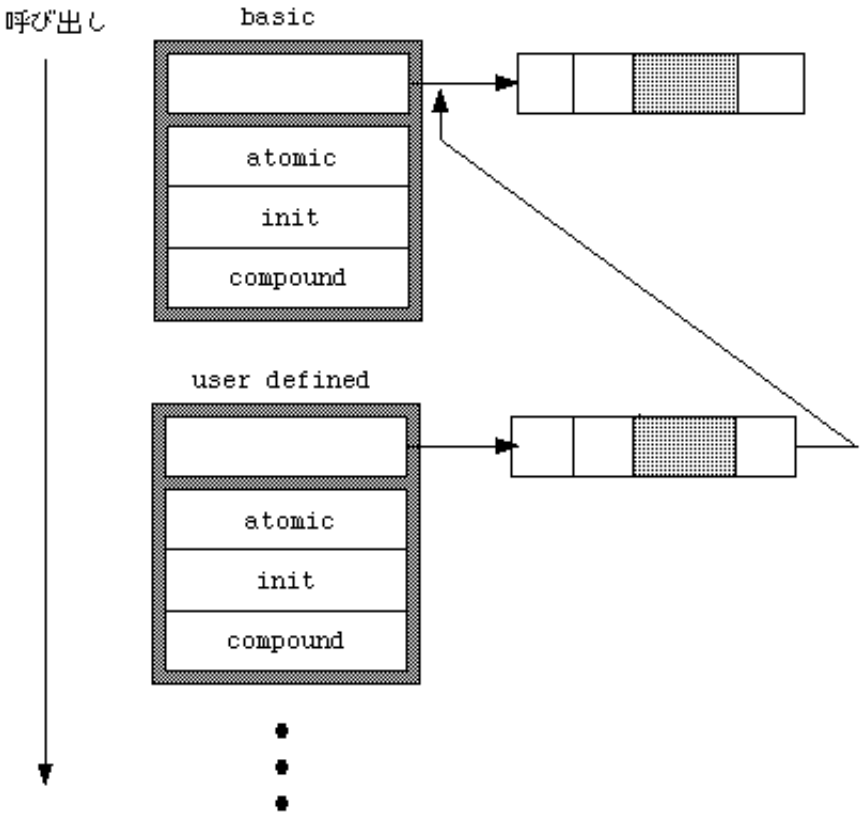


図 5.4: reflective-object

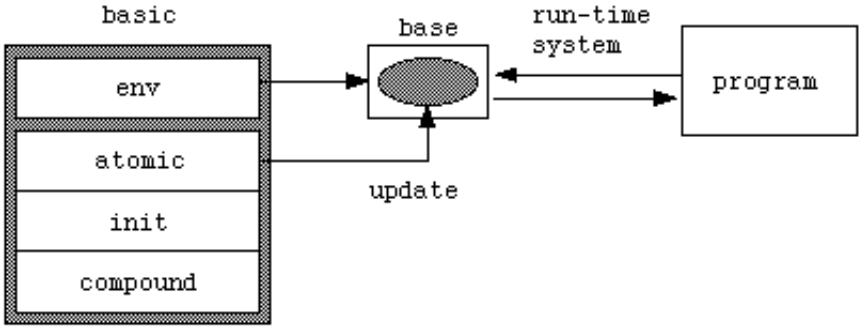


図 5.5: base

システムは、あらかじめ基本的な計算を実現するための特殊な reflective-module を、reflective-object の初期形態として与えている。これは、図 5.5に示すように、base と呼ばれる特殊なメタ変数を保持しており、通常の計算にもちいる値をここに保持している。baseがシステムによって変更されるのは、atomic な計算の最初の段階で、atomic な計算が扱う値を代入している。また、baseを参照するのは、compound な計算の rest 部に init 部の結果を渡す際である。ただしユーザーは、新たなモジュールによって baseに対する特殊なアクセスを記述することは可能である。

5.4.2 contrd-object

control-module の構造は基本的に reflective-object と同じである。ただし前節でのべたように reflective-operation の種類が異なっている。また control-object のふるまいは常に一つの module によって決まる。このとき、もっとも新しく組み込まれた module がアクティブな module になる。

アクティブな reflective-module の列は、control-module の状態の一部 (図 5.6の objsで示された部分) として保持されている。つまり、control-objectは、計算に関するすべての状態を保存している構造と考えてよい。

5.4.3 alphaと gamma の実装

alphaと gamma とは、プログラムの式の評価という過程のうち、特にモジュールによって与えられた拡張部分に関する計算を抽象化した関数である。

まず、今後 **computation** とはcontrol-object(その時点でアクティブな control-module) を

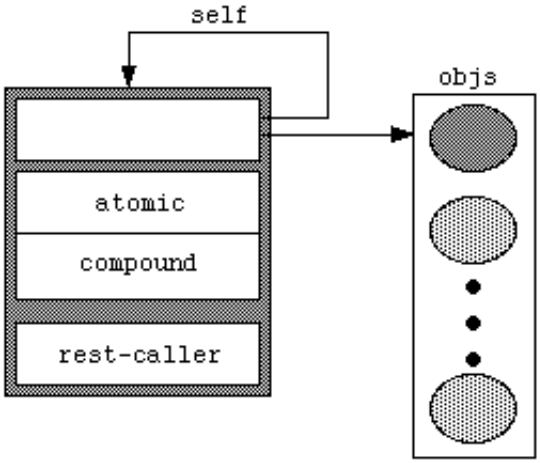


図 5.6: control-module

受け取り、control-object を返す関数であるとする。Flect の設計では、2 種類の computation が存在する。すなわち、atomic な計算と compound な計算に対応する computation である。

control-objectとはすべての計算状態を保持している構造であることを述べた。computation とは、この計算状態を変化させ、その結果を返す関数であると言える。

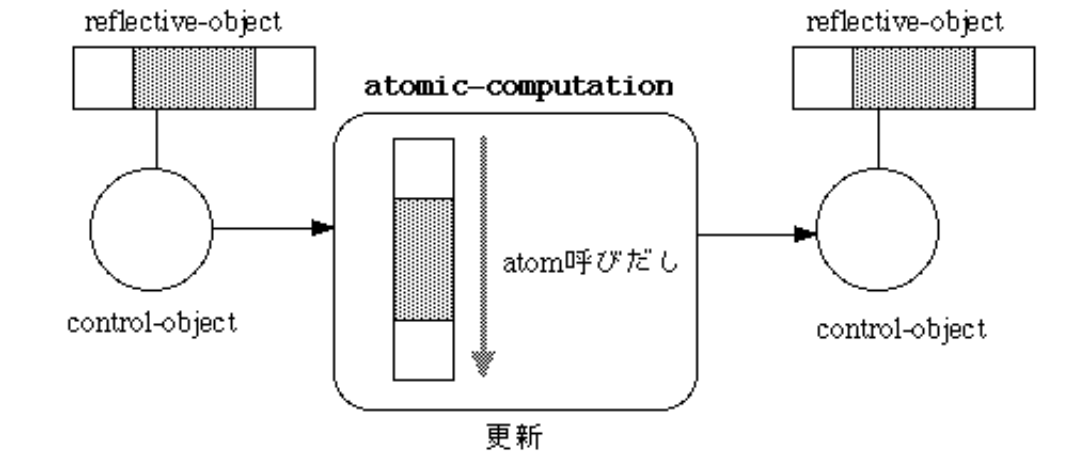
Flectにおける計算とは、この computation 間の control-object の受け渡しそのものである (図 5.7)。alpha と gamma の呼びだしは、この computation を生成するためのインターフェイスであると言える。

alpha の実装

alpha は 1 引数の関数であり、atomic な計算に対応する値 (定数、変数、クロージャなど) をうけとり、atomic な computation を返す。簡単に述べると、atomic な computation とは、その時点でアクティブな moduleにおける、atomic な計算に対応する操作の呼びだしである。alpha の呼びだしとは、その引き数によってパラメタライズされた atomic な computation の生成であるといってよい。以降で、atomic な computation の内部的な動作について解説する。

```
(alpha x)
```

図 5.8: 単純な alpha の例



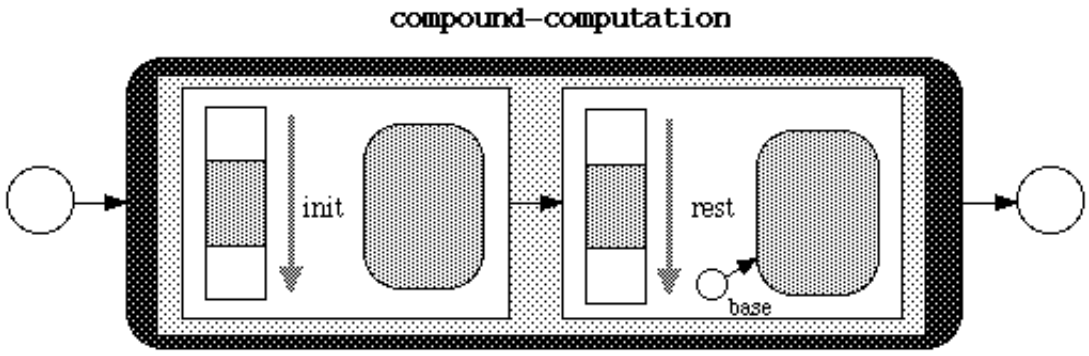


図 5.7: control-modul e

ここで参照される変数は、以下に示す値にバインドされているとする。

val alpha の呼びだし時に渡された値 (図 5.8における x)

ctr control-object

以降が、cintrol-objectを受け取った時点からの alphaの振る舞いである。

1. ctrから atomicな計算に関する操作 (dlambda 式) を参照する **(copr)**
2. ctr から制御に関するメタ情報を保持している環境 (D-環境) を参照する **(cenv)**
3. control-object を受け取るような以下に示す関数を定義する。**(f)**
   (i) control-object を受け取る **(c)**。
   (ii) control-object に保存されている、reflective-module の列を得る **(lst)**。
   (iii) control-object に保存されている、reflective-object の環境を得る **(renv)**。
   (iv) lst の各要素に関して、その atomic な計算に関する operationを列の最初から順に呼び出す。各 operationは dlambda 式であり、val を引き数として、renv を D-環境として渡す。
4. copr の第一引き数に f を、D-環境に cenv を渡して呼び出す
5. ctrを返す

gamma の実装

gamma は 2 引き数の関数である。おおまかな実行のながれは、第 4 章の図 4.15を参考にするとよい。

```
1: (gamma
2:   (alpha inc)
```

```
3:   (lambda (g0)
4:     (gamma
5:       (alpha x)
6:       (lambda (g1) (g0 g1))
7:     )
8:   )
9: )
```

図 5.9: 単純な gamma の例

第一引き数は、initial な計算に対応する computation、また第二引き数は、rest な計算に対応するラムダ式を受け取る。ここで rest な計算はあらたな computation(例えば図 5.9の 2 行目から 8 行目まで) か、または Scheme の意味に基づいた式 (例えば図 5.9の 6 行目) の呼び出しを含んでいる。そして、返り値として compound な computation を返す。compound な computation とは、その時点でアクティブな module における、compound な計算に対応する操作の呼びだしである。alpha と同様に、gamma 関数の呼びだしとは、二つの computation によってパラメタライズされた compound な computation の生成を意味する。以降で、compound な computation の内部的な動作について解説する。

ここで参照される変数は、以下に示す値にバインドされているとする。

ctr control-object

init gamma の第一引き数

rest gamma の第二引き数

以降が、control-object を受け取った時点からの gamma の振る舞いである。

1. ctr から compound な計算に関する操作 (dlambda 式) を参照する **(copr)**
2. ctrから制御に関するメタ情報を保持している環境 (D-環境) を参照する **(cenv)**

3. initial な計算に対応する関数 (**f**) と、rest 計算に対応する関数 (**g**) を宣言する **f の定義**

   (i) contrd-object を受け取る (**c**)。

   (ii) control-object に保存されている、reflective-module の列を得る (**lst**)。

   (iii) control-object に保存されている、reflective-object の環境を得る (**renv**)。

   (iv) lst の各要素に関して、その initial な計算に関する operation を列の最初から順に呼び出す。各 operation は dlambda 式であり、renv を D-環境として渡す。

   (v) init に c を渡して、initial な計算に移る。

   (vi) c を返す。

   g の定義

   (i) control-object を受け取る (**c**)。

   (ii) c に保存されている、reflective-module の列を得る (**lst**)。

   (iii) c に保存されている、reflective-object の環境を得る (**renv**)。

   (iv) lst の各要素に関して、その rest 計算に関する operation を列の最初から順に呼び出す。各 operation は dlambda 式であり、renv を D-環境として渡す。

   (v) reflective-object から base を参照する (**val**)。

   (vi) control-object から rest-caller(dlambda 式) を参照する (**rc**)。

   (vii) 関数 (**rcfunc**) を宣言する。その内容は、値を受け取って、その値を rest に渡し、その返り値が computation なら c を渡してその結果を返す。それ以外ならばその返り値自身をそのまま返す。これは、rest 計算の内容が 2 種類あることに対応する。

   (viii) rcfunc を、引き数として val を、D-環境として renv を渡して呼び出し、そのもどり値を reflective-object の base に格納し、その影響をうけた control-object を返す。

4. copr の第一引き数に f と g を、D-環境に cenv を渡して呼び出す。

5. ctr を返す

5.5 モジュールの組み込み

各モジュールの組み込みは非常に単純である。それぞれのオブジェクトに関する組み込みは、本章で示した、対応する構文に関する変換規則の右辺にあらわれた補助的な関数によってなされる。ここに示したすべての関数は、control-object を受け取るクロージャを生成する。以降では、control-object がすでに渡されたものとして考える。ここで注意したいのは、これらすべての関数が、何らかの計算としてはみなされていない点である。すべてのモジュール組み込み命令は、その body の計算に関して影響をあたえるモジュールの組み込みを定義している。すなわち、モジュールの組み込みは、その body 部の computation を呼び出す際に、control-object に対してフィルターをかけるような操作と考えればよい。

```
(local-reflective body dec atom init res id)
(local-control body dec atom comp id)
(change-caller opr body)
(import-caller opr body dec atom init res id)
```

モジュールを組み込む関数においてはまず、与えられた規則から、あらたなモジュールの実体化をおこなう。その際モジュールの実体を生成するための情報は、あらかじめ defrmod や defcmod によって宣言されているため、実行時には単にメタ情報を初期化するだけの操作をおこなう。生成したモジュールは、control-object やその内部の reflective-object に組み込まれる。ただし、モジュールの組み込みに関しては、変換時にユニークな ID が割り振られており、ある組み込み命令をループなどで何度も呼び出すことで、同一のモジュールの実体を重ねて組み込むことはおこなわないようにしてある。

control-object に関しては、現在の control-object を保存しておいて新しい module を実体化する。また、reflective-object に関しては、それまでのアクティブなモジュールの列に、新たなモジュールを付け加えることで、拡張をおこなう。こうして得られた新たな control-object が、body の computation に渡されるのである。body から制御がもどってくると、control-object を、モジュールが組み込まれるまえのものに復帰させ、それを最終的な返り値とする。

change-caller については、単に与えられた rest-caller を control-object に組み込み、body の実行の終了時に復帰させるだけである。また import-caller は、local-reflective と change-caller の操作を同時におこなう。

ここで問題となるのが、モジュールの結合によって生じるreflective-objectの状態(アクティブなreflective-moduleの列)と、制御構造との関わりである。control-objectは、制御構造を自由に扱うことが可能なことと、モジュールの組み込み操作が計算として扱われないためにcontrol-objectから見えないという理由から、モジュールの影響範囲からの復帰が正しくおこなわれない恐れがある。

```
(+ 1 (import-reflectve ... (+ 2 3)) 4)
     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

control-objectにとっては、(+ 2 3)と、その残りの計算は連続したものである。

そのため、変換時に正しく復帰をおこなうための関数(system)をbodyの返り値に対して適用させる。systemは、スタック上に組み込み前のreflective-moduleの列を保存しており、これを呼びだし時に復帰させる。この仕組みによって、正しいreflective-moduleからの復帰が保証される。

5.5.1 reifyとreflect

reifyとreflectの実現は非常に単純で、ある時点のメタ情報をユーザーにアクセス可能にする。図5.10に示したようにreifyでは、その時点のメタ情報をコピーしたものを実際の計

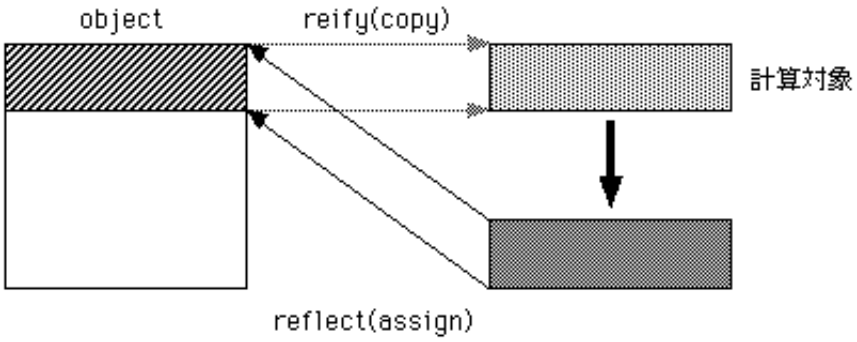


図5.10: reifyとreflect

算の値として参照可能なものにする。この際にも間接的ではあるが、dl ambda式をもちいている。つまりreifyされるオブジェクトから、メタ変数をコピーしてきて、それからD-環境を生成し、reify操作のbody部を、dl ambda式に変換したものに渡すことをおこなう。

これによって、メタ情報の参照シミュレートするのである。reflect の実現はさらに単純で、object 内部のメタ情報を副作用を使って更新するだけである。

5.6 アプリケーション

今回、Flect におけるプログラミングの例として、簡単な言語の構文解析器 (パーザ) の構築をおこなった。本節では、その実際のプログラミングの過程について解説をおこない、Flect の記述に関する能力の評価をおこなう。

5.6.1 パーザの設計

構文解析は、状態の変換のながれとして表現することができる。ここで扱う状態とは、解析される文字列である。たとえば、文字を一つだけ受け取るパーザ (**item**) を考えよう。パーザは、入力文字列を受け取って、その解析結果と、パーザというシステムの状態 (次にしらべる文字列) のリストを返す。ここでリストが返るのは、解析結果が一意的にはきまらないためである。つまり非決定的な選択が存在する (例えば、一文字か、または二文字をうけとるパーザなど)。

```
(single "test") -> [('t',"est")]

(one-or-two "test") -> [('t',"est") (('t','e'),"st")]

(single "") -> []

(one-or-two "a") -> [('a',"")]
```

5.6.2 パーザ・モジュール

解析器は、現在の状態を参照し、その解析結果を返すと同時に、解析内容に応じて状態を変化させる。Flect では、こうしたシステムの状態をそのまま言語自身に組み込むことを

おこなう。そのためにまず、reflective-module(parse-r)を設計する。これは、現在のパーザの状態のみをもつ構造である。計算の過程においては、なんら特別な操作はおこなわない。

```
(defrmod parse-r
  (inp)
  (dlambda (v) () 'noop)
  (dlambda () () 'noop)
  (dlambda () () 'noop))
```

5.6.3 単純なパーザ

まず、文字を一つだけ受け取るパーザを生成する。前提としてモジュール parse-r が組み込まれているものとする。また、前述のように計算に関して非決定性が存在するので、今後の計算は、前章で解説した nondeterminal モジュールの影響下にある。

このパーザがおこなうのは、メタ情報 inp を参照 (reify) し、その先頭を返すという操作である。また、状態の変化 (解析済みの文字の削除) が、そのままシステムに反映 (reflect) される。また、文字列が空の場合、計算の失敗を返す。

```
(define item
  (lambda ()
    (reify (inp)                        ;; 文字列の参照
      (if (null? inp)
          fail                          ;; 空列ならば失敗
          (let ((top (car inp)))        ;; 先頭要素を得る
            (set! inp (cdr inp))
            (reflect (inp) top))))))    ;; 状態を更新し、結果を返す
```

5.6.4 単純なパーザの組み合わせ

すでに、inp はシステムの状態として組み込まれており、計算のながれに応じて適切に受け渡される。実際の呼びだし例について次に示す。

```
(define parser ;; パーザの本体
  (lambda (func x)
    (import-reflective parse-r ((inp (string->list)))
      (rest-caller (nondeterminal-r () nondet-caller)
        (func)))))

(define single  ;; 一文字を受け取る
  (lambda () (item)))

>(parser single "test")
(nondet #\t)

>(parser single "")
(nondet)

(define double  ;; 二文字を受け取る
  (lambda () (list (item) (item))))

>(parser double "test")
(nondet (#\t #\e))

>(parser double "t")
(nondet)
```

5.6.5 alternation

ここでひとつ問題が生じる。これは、二つのモジュールの間の関連づけの問題である。Flect の設計では、計算状態の更新はメタ情報に対する副作用である。つまり、nondeterministic の影響が inp に反映されないという問題が生じる。そのため、inp を保存するような nondeterministic-choice のための特殊な操作を定義する必要がある。

```
(define alt
  (lambda (a b)
    (reify (inp)   ;; 状態の保存
      (amb a (lambda () (reflect (inp) (b)))))))
                   ;; 状態を復帰してから呼びだし
```

alt は二引き数の関数で、二つのプロシジャを受け取り、それらを非決定的に呼び出す。これを利用して、次のようなパーザを記述できる。

```
(define one-or-two  ;; 一文字あるいは二文字を受け取る
  (lambda () ((alt single double))))

>(parser one-or-two "test")
(nondet (#\t) (#\t #\e))

>(parser one-or-two "t")
(nondet (#\t))
```

5.6.6 フィルタの設計

読み込んだ文字の種類を判別するためのフィルタの設計をおこなう。フィルタは、入力を受け取って、条件を満たすかどうかを判別し、満たすならばそのままの値を返し、満たさないならば失敗を返す。

```
(define filter                ;; フィルタ
```

```
    (lambda (val p)
      (if (p val) val fail)))  ;; 条件 (p) を満たすか？

(define digit                  ;; 数値パーザ (数字 -> 数値)
  (lambda ()
    (char->digit (filter (item) char-numeric?))))

(define letter                 ;; 文字パーザ
  (lambda ()
    (filter (item) char-alphabetic?)))

(define lit                    ;; 特定の文字パーザ
  (lambda (c)
    (filter (item) (lambda (a) (char=? a c)))))

>(parser digit "1st")
(nondet 1)

>(parser digit "test")
(nondet)
```

5.6.7 繰り返しの定義

ここでは、文字の並びを受け取るパーザの定義をおこなう。たとえば、数値 (数の並び) や、識別子を解析する際に利用できる。繰り返し器 (iter) は、あるパーザを受け取り、それを失敗するまで繰り返し適用した結果の組み合わせを返す。

```
(define iter       ;; 繰り返し
  (lambda (opr)
    ((alt (lambda () (cons (opr) (iter opr)))
```

```
            (lambda () '())))))

(define numbers
  (lambda ()
    (numlist->number (cons (digit) (iter digit)) 0)))

>(parser numbers "123")
(nondet 123 12 1)  ;; 有り得る結果
```

しかし、実際には最も長い結果が必要となる場合が多い。上述の結果が得られるのは、iter 定義における選択の両方を実行しているためである。

そこで変わりに、最初の選択肢が失敗した場合のみ、次を実行するという方針 (biased choice) に切り替える。それには、rest-caller を変更する構文 (change-caller) を利用する。

```
(define reiter    ;; 繰り返し (最も長い結果のみ返す)
  (lambda (opr)
    (set-caller bias-caller (iter opr))))

(define number    ;; 値を一つ返す
  (lambda ()
    (numlist->number (cons (digit) (reiter digit)) 0)))

>(parser number "123a")
(nondet 123)
```

bias-caller の特徴は常に多くても二種類の選択しか存在しないことである。そのため、先頭の計算結果を見て次の計算をおこなうかどうかを決定するという方針で、以下のように記述する。

```
(def-restcaller bias-caller
```

```
(dlambda (f v) ()
  (if (fail? v)
      v
      (let ((val (f (cadr v))))
        (if (null? (cddr v))
            (cons 'nondet (concat (list val)))
            (if (fail? val)
                (cons 'nondet (concat (list (f (caddr v)))))
                (if (procedure? val)
                    (cons 'nondet (concat (cons val (cddr v))))
                    (cons 'nondet (concat (list val))))))))))
```

5.6.8 簡単な数式のパーザ

ここまでで定義してきたパーザを組み合わせることで、簡単な数式のパーザを定義することができる。まず、数式に対して括弧づけを必要とするパーザを記述する。具体的な文法は、次のようなものである。

```
term ::= number | '(' term '/' term ')'

この構文に該当する数式の例

123
(123/4)
((123/4)/5)
((1/2)/(3/4))
```

この構文に対するパーザの定義は、以下のようになる。なお、このパーザの実行には biased-choiœ を必要とする。なぜなら、最初の選択項目が成功した場合、もう片方が呼ば

れると、不要な状態の変更が生じるためである。つまりこのパーザは、複数の選択を許さないことになる。

こうした問題を解決するためには、rest-caller の実装に際して、reflective-module におけるメタ情報を保持する環境のコピーをおこなう手段を与えるような、何らかの考慮が必要であると思われる。

```
(define num (lambda (x) (list 'Con x)))  ;; 値の表現
(define skip (lambda (x) '()))           ;; 何も返さない
(define set (lambda (x) (list x)))       ;; term の生成

(define term
  (lambda ()
    (set-caller bias-caller
      ((alt (lambda () (num (number)))
            (lambda () (append
                        '(div)
                         (skip (lit #\())
                         (set (term))
                         (skip (lit #\/))
                         (set (term))
                         (skip (lit #\)))))))))))

>(parser term "(123/456)")
(nondet (div (Con 123) (Con 456)))

>(parser term "((123/456)/7)")
(nondet (div (div (Con 123) (Con 456)) (Con 7)))
```

次に、以下に示すような構文規則によって、括弧を含まない式のパーザの設計をおこなう。

```
term   ::= factor term2
```

```
term2  ::= '/' factor term2 | empty
factor ::= number | '(' term ')'
```

この構文に該当する数式の例

```
12/34/56
12/(34/56)
```

この構文をそのまま実装すると、次のようになる。

```
(define term
  (lambda () (term2 (factor))))

(define term2
  (lambda (t)
    ((alt (lambda () (term2 (append '(div)
                                    (list t)
                                    (skip (lit #\/))
                                    (set (factor)))))
          (lambda () t)))))

(define factor
  (lambda ()
    ((alt (lambda () (num (number)))
          (lambda () (append
                       (skip (lit #\())
                       (term)
                       (skip (lit #\)))))))))

>(parser term "1/2/3")
```

```
    (nondet (div (div (con 1) (con 2)) (con 3)))

    >(parser term "1/(2/3)")
    (nondet (div (con 1) (div (con 2) (con 3))))
```

5.6.9 記述性に関して

パーザの実装に関して、Flect を用いたことによる利点は、言語処理系自身をアプリケーションにあわせて拡張することによって、プログラムの見通しがよくなり、拡張性が向上したことがある。これは、単純なパーザを組み合わせるだけで、ある程度複雑な式の解析が可能になったことから言える。

しかし、alternative の実装の際に処理系の設計による制限から、統一的な記述がなされなかった部分があり、これは今後の設計の改善によって解消する必要がある。

5.6.10 実行効率に関して

Flectシステムでは、オブジェクトによって実現された言語の意味を頻繁によぶため、通常のコンパイルされたコードに対して、以下のような効率上の問題点が存在する。

- オブジェクト呼びだしに関するオーバーヘッド。
- reflective-operation の変換の冗長さ。

本節では、実際にテストプログラムを動かして見て、これらにかかるコストについて検討する。その際以下の項目に関してテストをおこなう。

1. Scheme インタープリタ上で動作するインタープリタ。
2. Flect のコンパイル後のコードを何ら拡張をおこなわないで 1 と同一のインタープリタ上で動作させた結果。
3. reflective-module を一つ加えた状態。
4. 同じ reflective-module を二つ組み込んだ状態。

5. control-module を CPS にした場合。

テストに用いたマシンは、Macintosh Powerbook 550c(MC68040　66MHz) で、ベースとなる Scheme 処理系には macgam bit2.2を用いた。

項目番号	(fact 500)	比率 (項目 2 の場合を 1 とする)
1	7.344 sec	1.199
2	6.123 sec	1.000
3	7.340 sec	1.199
4	8.120 sec	1.326
5	8.667 sec	1.415

今回実現した Flect システムの実行効率は、インタープリタ上で動作するインタープリタとほぼ同等である。

効率低下の原因の大部分は、オブジェクトの影響をコードに反映させるための computation の呼びだし時に生じるオーバーヘッドである。

モジュールの組み込みによる、効率の低下は、モジュールによって組み込まれる操作が、計算システムの状態をあつかうものであり、計算の全過程において参照されるような性質をもつ操作であることなどを考えると、妥当なものと考えられる。これは、モジュールの組み込まれる領域を限定することで軽減することが可能である。

第 6 章

結論

6.1 まとめ

本稿では、まず自己反映計算に関する基本的な概念と、いくつかのシステムの実装例について解説をおこない、計算状態に着目して考察をおこなった。そして、その考察をふまえて自己反映的な言語システム Flect の設計をおこない、そのコンパイラの実装をおこなった。その結果について以下にまとめる。

- monad に基づく自己反映的言語システムを、可変的なオブジェクトによる言語の意味の改変機構としてモデル化をおこない、実際にコンパイラとして実装した。
- 言語の拡張単位を、ひとつのモジュールとして与えることで、整理された記述が可能になった。
- 言語の意味を二つの独立したクラスに分類して提供することでモジュールの安全な結合のための枠組みを提供した。
- 言語の拡張を、ユーザーが指定した範囲において可能にしたことで、結合の自由度を高めた。
- first-class な値としてのメソッドの実現機構として dl ambda 式の機構を提案し、実際にシステムに適用した。
- 言語の適用例としてパーザの記述をおこない、アプリケーションへの適用のガイドラインを示した。

今後の課題としては、次に示すようなことがらを挙げることができる。

- 頻繁に computation の hook をおこなうために生じる、実行効率の低下を防ぐため、モジュール組み込みによる改変の影響範囲をより限定的なものにするような設計の見直しをおこなう。これは、変換時に影響を与えるようなモジュールの組み込みを意味するため、compile-time reflection と深い関係がある。
- メタ情報を副作用によって変更していくだけでは、制御構造との関連で実現可能な機能が限定され、そのために記述が複雑になる恐れがある。そこでメタ情報を保存している環境そのものへのアクセスを可能とするような設計を考案する必要がある。
- monad の適用例として、グラフィックシステムの構築や、並列計算の実現などがあげられる。それらを参考に、より広範な適用例について考察を行う必要がある。
- Fleet におけるシステムの拡張は、言語の構造をよりアプリケーションよりにカスタマイズすることに重点をおいていた。そこで今回、ベースである Scheme の意味に依存していた構造 (変数の参照や、プロシジャの呼びだしなど) を Fleet システム自身で用意し、ユーザーによるカスタマイズを可能にすることで、より言語寄りの拡張を可能にする。
- 各モジュールの組み合わせの手段は、reflective-module 同士のメタ情報を介するものと、reflective-object と rest-caller との連携に限定されている。しかし、monad の結合に関しては問題点が多い。そこでさらに monad の結合に関する調査をおこない、考察を深め、より充実した言語拡張の枠組みを与える必要がある。
- 今回示した言語の枠組みによるプログラミングの方法論について考察をおこなう。

6.2 関連研究

今回の言語の設計に際しては、[21] を参考にした。しかし、Sobel の研究では、プログラム中におけるモジュールの組み込みはサポートされておらず、また意味構造同士の組み合わせに関してはなんら考慮されていない。

また、メタレベルの再利用性に着目したアプローチとしては、言語の拡張手段として、インタープリタを first-class なオブジェクトとして渡す手法が提案されている [22] 。両者は、

記述に関する自由度の違い (Flect は限定的な拡張である) を別にすれば、言語拡張の手法に関するアプローチとしては、非常に関係が深いものである。

言語拡張に関する monad の利用例として、Scheme 上のシステムを用いた例としては、[7] があり、また monad によるコンパイラの構築例としては、[12] がある。これらの研究は、いずれも言語モジュール的な構成という立場が主体であり、実行時の自己改変という概念は考慮されていない。

謝辞

本研究を進めるにあたって終始ご指導くださった渡部先生に感謝致します。また有益な助言をいただいた二木先生に御礼を申し上げます。説明の仕方に関する議論につきあってくださった諸先輩方、そして有益な情報を与えてくれた友人たちに感謝の言葉をおくります。

参考文献

[1] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation — for a better understanding of reflective languages —. In *Lisp and Symbolic Computation*, Vol. 9, pp. 203–241. Kluwer Academic Publishers, 1996.

[2] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, 1993.

[3] W. Clinger and J. Rees. *revised*^{4} report on the algorithmic language scheme, 1991.

[4] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual ACM Symposium on POPL*, pp. 493–501. ACM, 1993.

[5] Olivier Danvy. Across the bridge between reflection and partial evaluation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pp. 83–116. North-Holland, 1988.

[6] Olivier Danvy and Karoline Malmkjær. Intentions and extensions in a reflective tower. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pp. 327–341. ACM, 1988.

[7] David Espinosa. Building interpreters by transforming stratified monads, December 1994. ftp from altdorf.ai.mit.edu:pub/dae.

[8] Andrzej Filinski. Representing monads. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on POPL*, pp. 446–457. ACM, 1994.

[9] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[10] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In Akinori Yonezawa and Brian C. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pp. 24–35, November 1992.

[11] Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. In Akinori Yonezawa and Brian C. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pp. 48–58, November 1992.

[12] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction, April 1996. http://www.swiss.ai.mit.edu/ftpdir/users/dae/home.html

[13] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction, esop, April 1996. http://www.swiss.ai.mit.edu/ftpdir/users/dae/home.html

[14] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 147–155, 1987.

[15] Pattie Maes. Issues in computational reflection. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pp. 21–35. North-Holland, 1988.

[16] Pattie Maes and Daniele Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[17] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proceedings of OOPSLA '95 (SIGPLAN Notices Vol.30, No.10)*, pp. 300–315, 1995.

[18] E. Moggi. Notions of computation and monads. *Information and Computation*, Vol. 93, pp. 55–92, 1991.

[19] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In Akinori Yonezawa and Brian C. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architeture (IMSA): Reflection and Meta-Level Architeture*, pp. 36–47, November 1992.

[20] Brian Cantwell Smith. Reflection and semantics in a procedural language (ph. d. thesis). Technical Report TR-272, Laboratory for Computer Science, MIT, 1982.

[21] Jonathan M. Sobel and Daniel P. Friedman. An introduction to reflection-oriented programming, April 1996. http://www.cs.indiana.edu/hyplan/dfried.html.

[22] Akira Tanaka and Takuo Watanabe. メタレベル記述の再利用を考慮した自己反映的言語, September 1995. http://ld-www.jaist.ac.jp:8080/~ak/research/paper/ipsj-51.ps.

[23] P. L. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, Vol. 2, pp. 461–493, 1992.

[24] P. L. Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, pp. 1–14, 1992.

[25] Mitchell Wand and David P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 298–307. ACM, 1986.

[26] Takuo Watanabe. リフレクション. コンピュータソフトウェア, 第 11 巻, pp. 5–14, 1994.