JAIST Repository
https://dspace.jaist.ac.jp/

Title	再利用可能な拡張機構を備えた言語処理系
Author(s)	佐伯，豊
Citation	
Issue Date	2001-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/908
Rights	
Description	Supervisor:渡部 卓雄，情報科学研究科，博士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

博 士 論 文

再利用可能な拡張機構を備えた言語処理系

指導教官　渡部 卓雄 助教授

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 13 年 2 月 3 日

Copyright © 2001 by Yutaka Saeki

要 旨

並列・分散処理や，適応可能なソフトウェアなど，複雑なアプリケーションを構築する際，固定された記述体系 (言語) では，直接表現することができない問題が生じる．そのような場合，アプリケーションの実際の動作に関する記述に対して，それを与えられた言語の枠組みで解釈実行するための記述をプログラム中に埋め込まなければならない．こうした煩雑で不自然な記述が，コードの各所に分散して存在すれば，メンテナンスやデバッグなどの解析のさまたげになり，検出が困難なプログラムのミスを生じやすくなる．また，こうしたアプリケーションに特化した記述が遍在することはコードの再利用を困難にする．

そのような問題点を解決するための方法として，プログラミング言語をアプリケーションプログラムに応じて拡張するアプローチが有効であることが知られている．特にメタレベルアーキテクチャに基づく言語拡張においては，本来目的とする計算 (**ベースレベル**) と，その実行方式を与える計算 (**メタレベル**) とを自然な形で分離して記述することができ，こうしたメタレベルの定義を他の同様なソフトウェア設計上の要求に対して適用することで，コード全体の再利用性を向上することができる．しかし，このようなメタレベル記述の再利用を試みた場合に，別の人間によって，互いに組み合わせて用いることを想定しないで定義された言語拡張の記述同士を，組み合わせて適用することが起こりうる．そのため言語拡張の記述の意味的な衝突が生じる可能性がある．これは，拡張部分の定義者のもつ拡張を施す対象 (言語) に関する認識と，利用者が実際に拡張を適用する時点における拡張される対象との違いから生じる問題である．

そこで本研究では，プログラマが容易に拡張の衝突の可能性を検出し，修正を容易にするためのフレームワークを提案する．

ソフトウェアの拡張時における安全性の保証は，発展的なソフトウェア構築の立場においても重要な課題であり，本研究で提案するモジュール化と拡張の枠組みの一般的なソフトウェアへの適用によって，改変が容易なソフトウェアの構築が可能になることが期待される．

目 次

1 序論 2

1.1 拡張可能なプログラミング言語 2

1.2 言語システムの構成 3

1.3 問題点と本研究のアプローチ 4

1.4 論文の構成 6

2 研究の背景と準備 7

2.1 背景 7

2.1.1 自己反映計算における拡張 8

2.1.2 手続き的リフレクション 8

2.1.3 メタオブジェクト 9

2.1.4 自己反映的言語 9

2.2 準備 11

2.2.1 Monad 11

2.2.2 例.(Monadic Interpreter) 12

2.2.3 monad transformer 14

2.2.4 Lifting 14

3 言語処理系のモジュール化 18

3.1 言語システムの動作の概要 18

3.1.1 仮想機械 19

3.1.2 基盤言語の定義 20

3.1.3 言語モジュールのレイヤー構造 20

3.2 コンパイラのモデル 26

3.2.1 内部状態へのアクセス 27

3.3 モジュールのメタ表現 29
3.3.1 例:(算術式) 29
3.3.2 例:(条件式) 30

4 コンパイラ記述言語 32
4.1 言語部品の定義 32
4.1.1 コンパイラ記述言語 33
4.1.2 メタモジュールの定義 35
4.1.3 メタモジュールの定義関数 36
4.1.4 単純な例 (算術式) 38
4.1.5 より複雑な例 (ベクトル) 38
4.2 変換規則 41

5 言語処理系の拡張 48
5.1 言語拡張の適用方法 48
5.2 各部品の拡張 49
5.3 言語拡張の例 51
5.3.1 算術演算の拡張例 51
5.3.2 関数呼び出しの拡張 51
5.3.3 数値演算の拡張 (2) 56
5.3.4 数値演算の拡張 (3) 56

6 拡張における衝突回避 59
6.1 衝突の可能性のある拡張記述の組み合わせ 59
6.1.1 変数参照の拡張例 59
6.1.2 拡張モジュールの記述 59
6.1.3 優先度の原理 62
6.1.4 衝突を起す事例 66
6.1.5 衝突原因の考察 67
6.1.6 衝突回避のためのアプローチ 68
6.1.7 参照制御の流れ 69
6.1.8 拡張の組み合わせ 71

6.2 参照制限の実現 73
6.3 衝突回避の事例 74
6.3.1 関数呼び出しのログ 74
6.4 依存関係に関する考察 77

7 まとめ 78
7.1 考察 78
7.2 関連研究 80
7.2.1 Domain-specific Languages 80
7.2.2 コンパイル時自己反映計算 81
7.2.3 属性文法の利用 82
7.2.4 安全な拡張記述のための方法論 82
7.3 今後の課題 83

謝辞 84

Appendix 89
A 関数適用の定義 89
B スタックフレームの定義 90
C クロージャの定義 91

本研究に関する発表論文 94

第 1章

序論

1.1　拡張可能なプログラミング言語

並列・分散処理や，適応可能なソフトウェアなどの，複雑なアプリケーションを構築する際，固定された記述体系 (言語) では，アプリケーションの実際の動作に関する記述に対して，それを与えられた言語の枠組みで解釈実行するための記述をプログラム中に埋め込まなければならない．そのような煩雑で不自然な記述が，コードの各所に分散して存在すれば，メンテナンスやデバッグなどの解析のさまたげになり，検出が困難なプログラムのミスを生じやすくなる．また，こうしたアプリケーションに特化した記述が遍在することはコードの再利用を困難にする．

そのような問題点に対して，自己反映計算 [21] など，メタレベルアーキテクチャに基づく言語拡張を利用することで，本来目的とする計算 (**ベースレベル**) と，その実行方式を与える計算 (**メタレベル**) とを自然な形で分離して記述することができ，またこうしたメタレベル記述の定義を他の同様なソフトウェア設計上の要求に対して利用することで，コード全体の再利用性を向上することができることが知られている．

しかし，現実に拡張の再利用をおこなおうとした場合，別々に定義された言語拡張の定義同士を，複合して適用する自体が想定できる．そのような場合，それぞれの拡張記述は，他のすべての拡張記述の存在を認識することは困難であることから拡張を施されていない言語を前提として定義されたものであることが予想できるため，言語拡張の定義間での衝突が生じる可能性がある．そうした場合，プログラムの実行時にユーザーの予想できない動作が起こりうるうえに，その原因をつきとめることは困難である．

そこで本研究では，言語の設計者が拡張モジュールを適用する際，システムが衝突のおこらないような拡張記述の選択をおこなう，あるいは拡張記述の間の衝突の可能性を拡張の適用時に検出し，事前に警告をおこなうためのフレームワークを提案する．

1.2 言語システムの構成

本研究ではコンパイラのコード生成過程の定義を部品化し，その各部品の動作を抽象度の高い言語によって表現するという手法でその内部の構造を言語設計者に示す．そしてそれらの各部品ごとに提供される関数の再定義を拡張機能の設計者に対して許すことによって，コンパイラのある部品が実現する言語の意味を変更するという手法で言語拡張を実現する．

本稿ではまず，基本的な関数型言語 (**基盤言語**) のコンパイラを，その部分機能ごとに複数のモジュールに分解するための基本的な仕組みを与える．本稿で想定する言語システムにおける基盤言語の内部構造は言語の特定の機能を実装したモジュールの階層構造として定義されている．各モジュールはそれぞれがコンパイラの部品であるとみなすことが可能であり，言語の構文要素，共有概念，およびメモリ操作のような低レベル機能などに対応した仮想機械のバイトコード生成の手順を実装したものとして与えられる．これらの各部品は各々が実現する概念の抽象度に応じて階層化されており下位の層が上位の層に対してサービスを提供する構造をもっている．

各モジュールはその機能を実現するための関数の集合として定義されている．各関数はコンパイラのコード生成過程を抽象度の高いレベルで記述することが可能な言語 (コンパイラ記述言語) によって定義されている．この言語は基盤言語に近い構文をもち，その構文の基盤言語における意味から自然に導き出すことが可能なコンパイラ実装レベルのコードを生成するように設計されている．

基盤言語に対してある特定の拡張機能を追加する場合，具体的な言語拡張の記述は，モジュール内の関数の置き換えとして与えられモジュール単位でまとめて適用される (**拡張モジュール**)．すなわち新たな言語機能を実現するように特定の部品を再定義することによって言語拡張をおこなうという手法を用いる．その際拡張をおこなう対象となる既存の言語機能を実現しているモジュールに対して，その動作を定義した内部の関数のいくつかをユーザーが定義したものに置き変える，あるいは新たな関数を追加するというような差分的な

関数定義の集合を適用することによって言語の拡張が実現される．

ここであるモジュールに対してその各内部関数はコンパイラ記述言語に対するメタな宣言群 (メタ宣言部) と実際のコード生成の過程をそのコンパイラ記述言語で定義した関数群 (操作定義部) によって定義される．メタ宣言部では階層構造のより低位で提供されたサービスの利用に関する記述をコンパイラ記述言語の拡張機能として定義することによってこのコンパイラ記述言語自身の拡張を提供する．すなわち記述言語によるコンパイル過程の記述をこの言語のメタレベルとして考えるならば低位層へのアクセスの宣言はコンパイラのメタ–メタレベルとみなすことが可能である．あるモジュールにとって低レベルであるような記述を記述言語のメタレベルとして抽象化することによって実行コード生成の過程のような操作の定義を宣言的に与えることが可能になる．この宣言性によってコンパイラの機能単位での拡張が容易になる．

実際の言語拡張部品の配布方法としてはある特定の言語の拡張機能を複数の拡張モジュールの集まり (パッケージ) として提供することを想定しており，配布される部品を利用して必要な言語を構築する立場の言語設計者すなわち言語部品のエンドユーザーは必要とする機能を提供するような複数のパッケージを同時に適用することによって新たな言語を実現することが可能である．

1.3 問題点と本研究のアプローチ

拡張記述の再利用性を考慮すると拡張モジュールの適用対象としてすでになんらかの拡張がなされたあとの言語モジュールを選ぶことによる差分的拡張が可能である必要性がある．

しかしモジュール化された言語部品の書き換えによる拡張において差分的な拡張を許す場合問題となるのは複数の拡張機能が混在する状況においては言語設計者が想定する機能を備えた言語が構築可能であるか否かが設計者にとって明確でないことである．例えば一つのモジュールに対して複数の拡張が適用された場合に，ある内部関数が同一モジュール内の他の関数の動作に依存していた場合，その関数が将来的に変更される可能性があると想定した際には望むような動作を保証することはできない．

他にも以下のような問題点が挙げられる．

- 一般的なコンパイラで適用されている多くの最適化は，その言語特有の性質を利用したものである．しかし柔軟な言語拡張によってそのような前提は保証されないために

最適化は効果は期待できないか，誤まった結果を返す可能性がある．

- 相反する性質をもつ二つの機能拡張が，一時に同居する場合，どちらの機能が優先されるのかはその内部的な設計方針によるため，部品として利用する立場では必ずしも組み合わせの効果が自明ではない．

- コンパイラには言語処理系特有の低レベルな処理が複雑に関連しているため，拡張を重ねた結果として部品間で明示的にはあらわれない相互の依存関係が発生する可能性がある．そのため拡張の影響が予測できない．

これらをまとめると拡張の衝突は明示的に与えられない暗黙の共通認識の間の誤解から生じるものであるといえる．このような共通認識の誤解はどのような要因で発生するのであろうか．我々は以下の項目を主な要因ととらえ，その認識に基づいて安全に拡張が可能なプログラミング言語の設計をおこなう．

- 異なる抽象度に属する記述が一つの記述単位に混在することで言語機能ごとのモジュール化がうまくできないため，部品の分類や関連付けが明確でない．

- 通常の単純なモジュール間の参照関係では拡張によるモジュール間の関連性の変化が反映されない．

- 拡張間の優先度が明確でない

- リソース使用の衝突を事前に検出できない

本研究では以下のアプローチをとり安全にモジュラーな言語拡張が可能なシステムの構築をおこなう．

まず拡張モジュール間の優先度を言語部品のエンドユーザーが設定するための優先度決定のための前提を設定する．本研究が想定するシステムでは適用順位がよりおそい拡張モジュールを優先することにする．すなわちあとから適用された拡張の性質が強くなるという前提である．

次にシステム側である程度拡張に関する制限を加える部分がある．これは言語部品関の階層構造の設定と拡張可能なグループの限定，およびメタ–メタ言語による下位ルーチンの抽象化によって実現する．

最後に拡張モジュールの設計者側で，その拡張によって発生する言語部品間の関係の変化に関する情報を明示的に設定することによって拡張機能の選択を制御する．

1.4 論文の構成

第 2 章では研究の背景の説明をおこない後半では準備として Monad の解説をおこなう．第 3 章では，実際にコンパイラと実行時のシステムを合わせたインタラクティブなプログラミング環境の構造を示す．第 4 章では高級なコンパイラ記述言語を設定し実際のコンパイラモジュールの記述例を示す．第 5 章では言語拡張の過程を実際の例を示して解説し，第 6 章ではその衝突の検出方法について述べる．第 7 章で関連する研究を示し本研究のアプローチについて評価および考察をおこなう．

第 2章

研究の背景と準備

2.1　背景

近年，DSL (Domain-Specific Language) など，特定の用途に特化した機能を言語機構として組み込むことで，アプリケーションの実装やメンテナンスにかかるコストを軽減するアプローチが提案されている．しかし，ある特定の分野に精通した設計者が，必ずしもプログラミング言語を設計するだけの知識を有することは期待できない．そのため，特定の分野で真に求められる機能を完備した言語が提供されているとはいえないのが現状である．

そこでそれほどプログラミング言語に精通していない設計者が，必要とする機能を組み合わせて，実際のアプリケーション開発に利用可能な言語を構築するための枠組が必要とされる．このような要求から従来のモジュール化手法を利用した言語部品の提供手法に基づくさまざまな試みがなされてきた．

例えばDSLをいくつかの選択肢においてその問題領域に関する選択をすることによって言語を構築するシステムである [20]．ただしこのような枠組では，設計者が必要とするような機能を不足なく提供できるために選択肢を注意深く用意する必要がある．

他のアプローチとしては言語構築の基盤として拡張可能な言語を提供しその拡張として部品化された言語の機能を選択的に取入れることでアプリケーションのドメインに応じた差分的な言語の変更を提供する手法が試みられている．この場合，言語の差分的な拡張において要求される知識は，基盤となる言語の知識と，適用される拡張機能の性質であり，設計者はその両者に関して十分に習熟していると想定できるため，より柔軟な言語の構築手段を容易に提供できる．

2.1.1 自己反映計算における拡張

本研究ではあらかじめ基盤となるモジュール化されたコンパイラを用意し，モジュール単位の差分的な拡張によって要求を満すというアプーチをとった．同様な拡張機能を実現するための枠組として自己反映計算の概念がある．

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 [12]．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 [22]．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．

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**のシステムとよび，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．

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 [29].

2.1.3 メタオブジェクト

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といえる．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[11][13] 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．

2.1.4 自己反映的言語

Brown[26] は，Scheme 上に実現された自己反映的な言語システムである．Brown 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，Brown 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．

Brown 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 Brown 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 [3][8] 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．

ABCL/R3[14] 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．

また，Black[1] や RbCl[7]，AL-1/D[19] など，自己反映的な言語処理系が多数存在する．

2.2 準備

2.2.1 Monad

表示的意味論 (Denotational Semantics) は，プログラミング言語の意味を記述するための数学的な手段として用いられている．しかし, そのモジュール性の欠如から, 拡張の困難さなどの問題が指摘されてきた [16].

Moggi [15] は，monad を用いたモジュラな表示的意味論を示し, Wadler [25] は実際に純粋な関数型言語における imperative な計算の monadic な実装を与えた.

Liang らは，Haskell の constructor class を利用して monad transformer を実現し，そのもとで実際にインタープリタの定義を与えた [10]．また Espinoasa は，stratified monad を提案することによって言語のモジュール化のための，より構造的なアプローチを与え, またそのフレームワークにもとづいた Scheme による実装を与えた [5].

また，モジュラーなコンパイラの構築のためのアプローチとしては, [9] がある．この研究では，環境 monad のオペレータ同士について成り立つ公理 (environment axiom) に基づいて，効率的な ML コードへの変換を行うためのフレームワークが与えられた.

本研究ではコンパイラを複数の内部状態をもち，入力式に応じてそれらを更新していくようなモデルにもとづいて構築する． 本研究では以下の観点から monad を採用した.

- 言語システムのモジュール化をおこなうことを容易にするための手段
- 拡張に対する制限を与えるための，モジュールの性格付け

Monad と拡張可能なプログラミング言語との関連としては，monad を適用してメタ継続 [26] を実現した例として [6] があり，また monad そのものをメタレベルとみなした自己反映計算の提案として [23] がある.

本節では, これまでに monad や monad transformer に関してなされてきた研究について解説し，コンパイラの構築に必要な monad transformer を示す.

monad は, 型構成子 M および二つの操作 (多相関数) からなる三つ組み $(M, unit(M), bind(M))$ で定義される．$unit$ と $bind$ は，次にしめす型をもつ:

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

これらの直感的な意味としては, $M\ a$ は，a 型の値を返す *computation* を示し，操作 $unit\, x$ は，単に値 x を返すような *computation* を生成する操作，$bind\, c1\ \lambda v.c2$ は *computation* の結合を与えるような操作である． *bind* の実際の働きとしては，まず *computation* $c1$ を実行し, その結果を変数 v に束縛した状態で，*computation* $c2$ を呼び出す流れを示している．

また *monad* の操作 (*unit* と *bind*) は次の条件をみたす.

$$
\begin{aligned}
(unit\, a) * k &= ka \\
c * unit &= c \\
a * (\lambda x.(b * \lambda y.c)) &= (a * \lambda x.b) * \lambda y.c
\end{aligned}
$$

読みやすさのため, 以降では次の表記を使うことがある.

$$
\begin{aligned}
unit\, a &\Rightarrow [a] \\
bind\, a\ \lambda v.b &\Rightarrow a \gg_v b
\end{aligned}
$$

通常，関数型言語では，計算をある型をもった値を，別の値に対応させる関数として実現する．monadic な計算のモデルでは，関数は値を *computation* に対応させる．すなわち，型 $a \to b$ の関数を，型 $a \to M\, a$ の関数に変換することで, 入出力や変数への代入に代表されるような副作用や，例外処理を提供するための付帯的な操作を monad によって隠蔽し, *computation* 上の計算として抽象化する.

具体的に示すと，例えば恒等関数 $I : a \to a$ は，*computation unit* (型は $a \to M\, a$) に対応し, 関数 $f : a \to b$ と関数 $g : b \to c$ の結合 $fog : a \to c$ は, *computation* $f' : a \to M\ b$ と *computation* $g' : b \to M\ c$ の結合を示す *computation*:

$$\lambda a.f'\, a \gg_b h'\, b$$

に対応する.

2.2.2 例.(Monadic Interpreter)

本節では，monad を実際に適用した具体例として monadic なインタープリタの構築方法を示す．一般的なインタープリタの評価器 (evaluator) は，項 (term) を値に対応させる関数として与えられる (図.2.1).

例えば例外処理をおこなうインタープリタは，図.2.2 のように定義される．この例のように，言語を拡張する際には従来の方法ではうまくモジュール化することができない状況が

$$
\begin{array}{lcl}
\mathrm{eval} & : & Term \to Int \\
\mathrm{eval}\,(Num\,n) & = & n \\
\mathrm{eval}\,(Div\,t\,u) & = & (\mathrm{eval}\,t) \div (\mathrm{eval}\,u)
\end{array}
$$

図 2.1: 通常のインタープリタ

$$
\begin{array}{lcl}
\mathrm{data}\ \mathit{Value} & = & \mathit{Raise\ String} \mid \mathit{Return\ Int} \\
\mathrm{eval} & : & \mathit{Term} \to \mathit{Value} \\
\mathrm{eval}\ (\mathrm{Num}\ n) & = & \mathit{Return}\ n \\
\mathrm{eval}\ (\mathrm{Div}\ t\ u) & = & \mathrm{case}\ (\mathrm{eval}\ \mathtt{t}) \\
 & & \quad \mathit{Raise}\ e \to \mathit{Raise}\ e \\
 & \mid & \quad \mathit{Return}\ a \to \\
 & & \qquad \mathrm{case}\ (\mathrm{eval}\ \mathtt{u}) \\
 & & \qquad \mathit{Raise}\ e \to \mathit{Raise}\ e \\
 & & \qquad \mid \mathit{Return}\ b \to \\
 & & \qquad\quad \mathrm{if}\ \mathtt{b} = \mathtt{0} \\
 & & \qquad\quad \mathrm{then}\ \mathit{Raise}\ \text{"}\mathit{Zero\ Divide}\text{"} \\
 & & \qquad\quad \mathrm{else}\ \mathit{Return}\ (a \div b)
\end{array}
$$

図 2.2: 例外処理をおこなうインタープリタ

ありうる．このような言語の構造は，インタープリタを *Computation* 上で定義された式として表現することによって有効にモジュール化することが可能であることが知られている．

そこでまず，図.2.1 で定義されたインタープリタを，monad をもちいて *computation* に関する式として定義しなおした例を示す．図.2.3 は，monadic なインタープリタを示しており, term を $M\ Int$ 型の computation に対応させる．ここで与えられた monad は最も単純なもので，*computation* $M\ a$ は値 a そのものである．

まず図.2.3 のインタープリタを state を扱うように拡張する．*Computation* は，型構成子 M と二種類の操作 (*unit* と *bind*) によって決まる．図.2.4 の state monad と，図.2.3 の monad を置き換えることでインタープリタを state が扱えるように拡張できる．このように，monadic な言語の拡張は非常に容易である．*Computation update* は state に引き数 f

$$
\begin{array}{lcl}
\text{type}\ M\ a & = & a \\
unit & : & a \to M\,a \\
unit\ a & = & a \\
bind & : & M\,a \to (a \to M\,b) \to M\,b \\
bind\ a\ k & = & ka \\
\\
\text{eval} & : & Term \to M\,Int \\
\text{eval}\,(Num\,n) & = & [n] \\
\text{eval}\,(Div\,t\,u) & = & (\text{eval}\,t) \gg_{v1} \\
 & & (\text{eval}\,u) \gg_{v2} \\
 & & [v1\ \div\ v2]
\end{array}
$$

図 2.3: monadic なインタープリタ

を適用することで更新をおこなう.

2.2.3 monad transformer

実際の言語を実装する場合には，state 以外にも様々な monad (継続，環境，例外等) を必要とする．しかしこれら複数の monad を組み合わせて用いる場合，互いにどう影響しあうかを考慮しなければならない.

state monad transformer (図.2.5) は，与えられた monad に対して state monad の性質を追加する働きを持つ.

Environment monad transformer (図.2.6) は環境に関する monad transformer である.

Exception monad transformer (図.2.7) は例外処理に関する monad transformer である.

2.2.4 Lifting

monad transformer によって，言語をモジュール化することができるが, monad transformer を適用された側の monad に関する操作を示す *Computation* (update や rdEnv inEnv など) が, 新たな monad 上で再定義されていない．そこで以前の定義を新たに積み重ねら

$$
\begin{array}{lcl}
type\, M\, a & = & State \to (a, State) \\
\\
unit & : & a \to M\, a \\
unit\, a & = & \lambda x.(a, x) \\
\\
bind & : & M\, a \to (a \to M\, b) \to M\, b \\
bind\, m\, k & = & \lambda x.let\, (a, y) = m\, x \\
 & & in\ let\, (b, z) = k\, a\, y \\
 & & in\ (b, z) \\
\\
update & : & (s \to s) \to M\ s \\
update\ f & = & \lambda s.(f\, s, s)
\end{array}
$$

図 2.4: State monad

$$
\begin{array}{lcl}
type\, StateT\, s\, m\, a & = & s \to m\ (s, a) \\
\\
unit\ s\, x & = & \lambda s.[(s, x)]^m \\
bind\ c\, k & = & \lambda s_0.c\, s_0 \gg^m_{(s_1, a)} k\, a\, s_1 \\
\\
update\ f & = & \lambda s.[(f\, s, s)]^m
\end{array}
$$

図 2.5: State Monad Transformer

$$
\begin{array}{lcl}
\mathit{type}\ \mathit{EnvT}\ r\ m\ a & = & r \to ma \\
\\
\mathit{unit}\ a & = & \lambda r.[a]^m \\
\mathit{bind}\ c\ k & = & \lambda r.m\ r\ \gg_a^m k\ a\ r \\
\\
\mathit{inEnv}\ r\ m & = & \lambda r_0.m\ r \\
\mathit{rdEnv} & = & \lambda r.[r]^m
\end{array}
$$

図 2.6: Environment Monad Transformer

$$
\begin{array}{lcl}
\text{type}\ M\ a & = & \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a \\
\mathit{unit}\ a & = & \mathit{Return}\ a \\
\mathit{bind}\ a\ k & = & \text{case}\ \mathtt{a} \\
 & & \mathit{Raise}\ e \to \mathit{Raise}\ e \\
 & \mid & \mathit{Return}\ b \to k\ b \\
\mathit{err}\ \mathit{str} & = & (\mathit{Raise}\ \mathit{str}) \\
\mathit{try}\ c & = & c \gg^m_{\mathit{Raise}\ e} [\mathit{Return}\ e]^m \\
 & \mid & c \gg^m_{\mathit{Return}\ a} [\mathit{Return}\ a]^m
\end{array}
$$

図 2.7: Exception Monad Transformer

$$
\begin{array}{lll}
update & \Longrightarrow^{\text{lift}}_{state} & \lambda s.update \gg_x [(s, x)] \\
rdEnv & \Longrightarrow^{\text{lift}}_{state} & \lambda s.rdEnv \gg_x [(s, x)] \\
inEnv\ r\ c & \Longrightarrow^{\text{lift}}_{state} & \lambda s.inEnv\ r\ (c\ s) \\
err\ e & \Longrightarrow^{\text{lift}}_{state} & \lambda s.err\ (e, s) \\
try\ x & \Longrightarrow^{\text{lift}}_{state} & \lambda s.try\ (x\ s) \\
 & & \\
update & \Longrightarrow^{\text{lift}}_{env} & \lambda r.update \\
rdEnv & \Longrightarrow^{\text{lift}}_{env} & \lambda r.rdEnv \\
inEnv\ r\ c & \Longrightarrow^{\text{lift}}_{env} & \lambda r'.inEnv\ r\ (c\ r') \\
err\ e & \Longrightarrow^{\text{lift}}_{env} & \lambda r.err\ e \\
try\ x & \Longrightarrow^{\text{lift}}_{env} & \lambda r.try\ (x\ r)
\end{array}
$$

図 2.8: Lifting Operations

れた monad transformer のレイヤーから呼び出すことを可能とする操作 (lifting) をおこなう必要がある (図.2.8 を参照).

第 3章

言語処理系のモジュール化

今後は, 実際にコンパイラと実行時のシステムを合わせたインタラクティブなプログラミング環境の構造を示し，その拡張に関する議論をおこなう．対象とする言語は関数型で ML に近い文法をもったプログラミング言語である．

3.1 言語システムの動作の概要

想定する言語システムの形態としては一般的な関数型言語に見られるようなインタラクティブにユーザーの入力式を受取り，そのコンパイル結果を仮想機械が実行したあとで結果を出力して，またユーザーの入力を待つという流れをもつものとする．

図.3.1 に示すように，ユーザーの入力した式は，構文解析をおこなって構文木を生成し，ソースコードレベルの解析をおこなったあとでコード生成器に渡され仮想機械の実行コードとストアを生成するために用いられる．

構文解析部とコード変換部およびコード生成部など各コンポーネントが拡張された場合の関連づけは重要な課題であるが，議論を簡潔にするため，構文の拡張は新しい構文が追加する場合に限定し，本研究では特に実行コード生成部において拡張がなされた場合を対象とする．それ以外のコンポーネントの変更は，適時行われるものとして考察の対象としない．

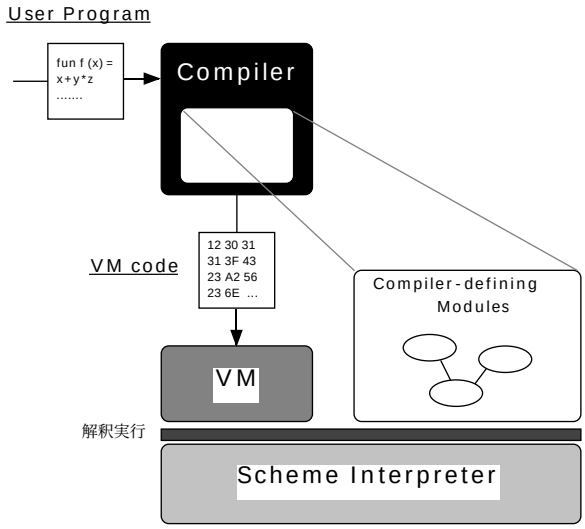

図 3.1: プログラム実行の流れ

3.1.1 仮想機械

ユーザーの入力式は，仮想機械によって実行される命令列に変換される．仮想機械はレジスタベースでロード・ストア方式の命令セットを持つ．基盤言語におけるメモリ使用については 図.3.2 に示すような構成をとる．メモリの利用形態としては，基本的には以下の方式に分類される．

- ストアは定数や配列など一般的なデータの格納に利用される．特殊なレジスタ GP からの相対番地で参照される．
- スタックは関数呼び出し時の引数の格納やレジスタの退避など一時的な記憶に利用される．
- ヒープ領域は実行時に割り当てられる動的なオブジェクトを提供するために利用される．
- コード領域は実行コードの列が格納してある．

また， 実行時システムとして， メモリセルの実行時の割り当てと，ガベージコレクションをサポートしている．

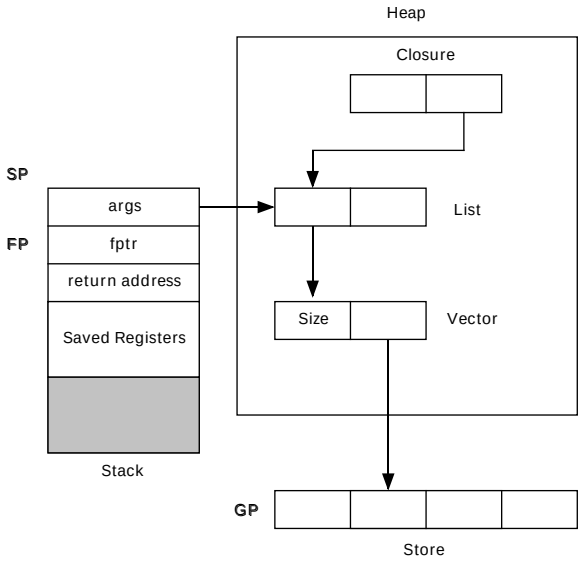

図 3.2: Memory Management

3.1.2 基盤言語の定義

コンパイラの拡張に関する議論をおこなう前に，まずは，拡張の対象となる基盤言語を設定する．我々が想定した言語は Dynamic-Typing の関数型言語で，ML に近い構文 (図.3.3 および 3.4) をもつ．

3.1.3 言語モジュールのレイヤー構造

コンパイラの機能全体においてコード生成部が担当する段階は，前処理によって生成された入力プログラムの内部表現である項から仮想機械が実行可能な命令コードを生成する一連の手続きである．

図.?? における，コード生成器を示すコンポーネントの内部構造はいくつかのグループに分類された複数の部品の集合体 (Compiler-defining Modules) として実現されている．個々の部品はそれぞれコンパイラの特定の機能を実現しており，各々の部品が言語拡張をおこなう際の基本単位 (**意味単位**) となる．これらの部品をユーザーが再定義したものに置き変えることによって言語拡張を実現する (図.3.5)．

コード生成の基本的な原理は式の構文上の分類に応じて対応するコード生成関数のディスパッチをおこなう単純な形式である．本システムでは個々のコード生成関数とそれを独立に定義するための補助的な定義をまとめたものをそれぞれモジュールとして実現する．各

```
<id> ∈ [a-zA-Z][a-zA-Z0-9_]*
<number> ∈ [0-9]+

<mod_name> = <sig_name> = <var> = <type> = <id>

<arg> ::= <var>
       | <type>:(<arg>,...,<arg>)
       | <var> as <type>:(<arg>,...,<arg>)
```

関数宣言：

```
<fdcl> ::= fun <var> (<arg>,...,<arg>) = <exp>
```

変数宣言：

```
<vdcl> ::= val <arg> = <exp>
```

宣言文：

```
<dcl> ::= <fdcl> | <vdcl>
```

モジュール宣言：

```
<moddec> ::= module <mod_name> : <sig_name> (<var>,...,<var>)
             in <dcl>;...;<dcl> end
```

シグニチャ宣言：

```
<sigdec> ::= signature <sig_name>
             in <dsc>;...;<dsc> end
```

演算子：

```
<op>   ::= +  | -  | *  | /  | % | < | <= | > | >= | ==
```

図 3.3: 構文定義 (1)

```
<exp>  ::= <number>                                  整数
         |  true                                    真
         |  false                                   偽
         |  <var>                                   変数
         |  <type>:(<exp>,...,<exp>)                 構造体
         |  <var>.<var>                             構造体の参照
         |  <exp> :: <exp>                          ペア
         |  []                                      空リスト
         |  [<exp>,...,<exp>]                       リスト
         |  <<exp>,...,<exp>>                       ベクトル
         |  vref <exp> <exp>                        ベクトル参照
         |  <exp> ... <exp>                         関数適用
         |  <exp> ()                                0 引数の関数適用
         |  let <dcl>;...;<dcl> in <exp> end        局所変数の宣言
         |  letrec <dcl>;...;<dcl> in <exp> end     局所変数の再帰的宣言
         |  if <exp> then <exp> else <exp>          条件式
         |  <exp> <op> <exp>                       演算
         |  begin <exp>;...;<exp> end               式の列
         |  fn (<arg>,...,<arg>) => <exp>           関数抽象
         | (<exp>)
```

図 3.4: 構文定義 (2)

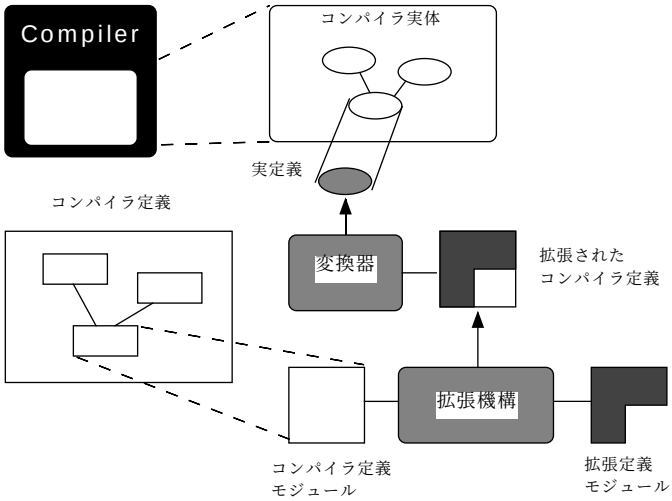

図 3.5: 言語拡張の概要

モジュール内の関数は特定の機能 (その言語で提供される構文や，コンパイラを構成する概念) に対応するコード生成を実現するための *computation* を定義するものである．特に各構文に対応した部品には，結果を返すレジスタをとるような *computation* が存在し，それらは， ディスパッチ時にその式に応じて呼ばれる．

また上記のようにモジュール化をおこなった場合に複数のモジュール間で共有する必要のある概念が存在する．例えば関数呼び出しと関数抽象および変数参照をそれぞれ実現したモジュールに対してスタックフレームを実現したモジュールは共通して必要となる概念である．こうした概念はある特定の構文に関するモジュールに属するような性格をもっていない．

モジュール間の依存関係 (あるモジュールの変更が他者に影響をおよぼすような関係) はモジュール同士の参照構造から導きだす方法が簡潔である．しかし単純に一部の機能を利用するだけのために構文に直結するようなモジュール間に依存性を設定することは望ましくない．例えば関数抽象と関数適用とは密接な関連があるが，それらと変数参照は基本的にスタックフレームを介した関係でありそれに拘らないような変更については直接影響をうけるものではない．すなわち拡張がされていない段階では前者の 2 つのモジュール同士の関連性と後者のモジュールに対する関連性とは性質が異なるもの，すなわちスタックフレームを介した間接的な関係である．

参照関係と関連性とを一致させるために，言語の意味を決定する部品 (言語モジュール) のうちで構文に直結した性質を決定するモジュールを一つのグループとして設定しそのグ

ループ内での橋渡し的な役割をもつモジュールを下位レベルのグループとして分類することによって，グループにまたがるような参照関係に別の意味をもたせることは自然である．またグループの階層化は各部品が実現するコンパイラの機能の抽象度に応じたものとなっており，参照関係の単純化に寄与している．(図.3.6)

次にさらに下位のレイヤを構成するものとしてメモリ操作やコード生成に関連した基本的な概念を実現するモジュールの階層を設定する．この階層が特殊な理由は基本的に拡張による置き換えができない固定された機能であるために，グループ内の参照関係の変化にともなう上位の階層に属するモジュールとの間の特別な依存関係を考慮する必要がないすなわち上位のレイヤから自由に参照可能であるような前提を設定可能である点である．またすべての拡張に対して固定された機能を実現しているため，言語拡張を記述する場合の基礎となる部分ともいえる．システムがコンパイラの実際の動作を生成する際に暗黙的かつ安全に利用することが可能な道具でもある．

最後に最下層のレイヤとして Monad による *computation* の定義とその関連する関数定義を与える．このレイヤの存在によって実際のコード生成の過程で必要となる低レベルな記述を隠蔽することができ，各部品の記述が容易になる上に再利用性が向上するという利点がある．

このレイヤは特殊な構造をもっており拡張をおこなう立場においては各モジュールの実体は前章で示した Monad Transformer である．これはこのレイヤのモジュールのアクセスに対して特殊な意味をもたせることで，システムが上位で定義された関数の挙動を制御することが可能になるからである．ただし拡張を実際に記述する立場にとっては上位から見えるこの階層の実体は特定の順番で Monad Transformer を適用することによって得られた Monad のオペレータを提供するような単一のモジュールであり，どのモジュールにアクセスするかを意識する必要はない．またこのレイヤの拡張に関する安全性は State および Environment Monad Transformer の追加適用と，特定のあらかじめ安全性を考慮して提供された Monad Transformer の選択肢を選ぶような拡張に制限することによって容易に保証できる．

ここで図.3.6 は，モジュールの階層構造を図にまとめたもので，上位にあるモジュールは下位のサービス (公開された関数) を利用することが可能であることを示しており，各々の階層が以下の性質を持ったモジュールの集合 (グループ) を示している．

- Syntactic Constructs：言語の各構文に対応したコード生成モジュール群で

式の構文に応じてコード生成関数が呼ばれる．

  - 変数参照 (Id)
  - 関数抽象 (Closure)
  - 関数適用 (Application)
  - 数値演算 (Arithmetic)
  - 数値定数 (Number)
  - 条件分岐 (Condition)
  - リスト (List)
  - ベクトル (Vector)

- Share：複数の構文要素の間で共有されるような概念を実装したモジュール群で Syntactic Constructs に属するモジュール間で共有される．
  - スタックフレームの操作 (Frame)
  - 値の判定 (Check)
- Base：メモリ操作やコード生成に関連した基本的な概念を実現するモジュール群で Share 以上のレイヤで利用される．システムがコード生成時に暗黙的に利用することがあり，基本的に拡張はおこなえない．
  - ストアの操作 (Store)
  - レジスタの割り当て (Register)
  - スタックの操作 (Stack)
  - メモリセルの操作 (Cell)
  - 命令の生成 (Instructions)
  - 命令生成のカウント (Count)
  - 生成された命令列の操作 (Code Fragment)
- Monad：Monad を実装したもの

この階層構造は下位レイヤの変更が上位のモジュール間の関連性になるべく影響を与えないように注意深く設計したものである．Syntactic Constructs および Share に属する各モジュールは自由に置き換えや追加が可能となっている．それを可能とするために，これ

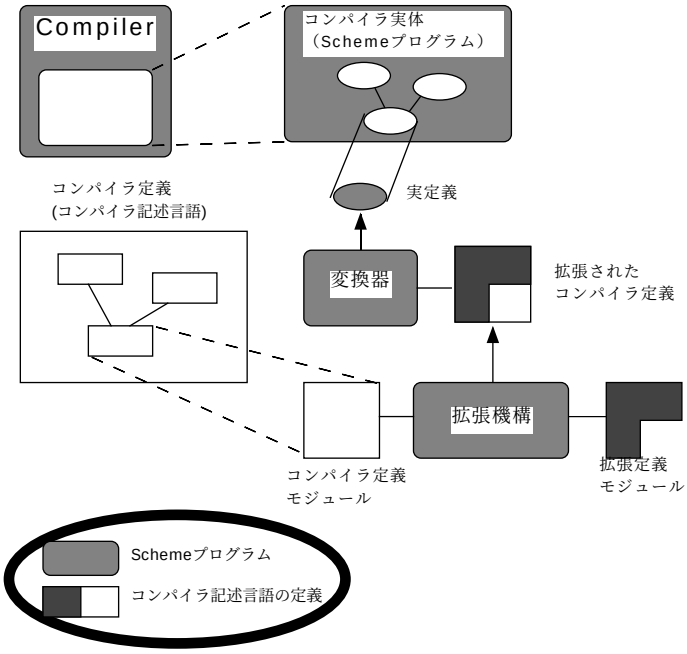

図 3.6: レイヤ間の関係

らのモジュールに対しては高級かつ変更可能なデータ表現で与えられたそれぞれの実装に等価な定義が用意されており，図.3.1 の右の部分のように，ユーザーの置き換えや拡張に対応することになる．以降はこれらの置き換え可能なレイヤに限って議論を進める．

3.2 コンパイラのモデル

本節では， monad transformer を用いたコンパイラの構築方法について示す．なお， 以降でなんらかのデータ構造の定義を行う際には，関数型言語 Haskel の表記法をもちいる．

まずコンパイラを，ソースコードから仮想機械の実行コードを生成する *computation* として定義する．そのために， いくつかの monad transformer を組み合わせた monad を定義する．ここでコンパイラの計算状態は monad によって与えられる次の値によって表現される．

図 3.7 はコンパイラを含むプログラミングシステム全体の状態を示したもので，それぞれ以下の意味をもつ.

$$
\begin{array}{llcl}
\text{type} & \mathit{Register} & = & \mathit{gpr}_1 \mid \mathit{gpr}_2 \mid \ldots \mid \mathit{gpr}_n \\
\text{type} & \mathit{Memval} & = & \mathit{Inst} \mid \mathit{Int} \mid \mathit{Char} \\
\\
\text{type} & \mathit{global} & = & [(\mathit{label}, \mathit{address})] \\
\text{type} & \mathit{local} & = & [[\mathit{name}]] \\
\text{type} & \mathit{mem} & = & [\mathit{memval}] \\
\text{type} & \mathit{reg} & = & ([\mathit{register}], [\mathit{register}]) \\
\text{type} & \mathit{code} & = & [(\mathit{label}, [\mathit{Inst}]] \\
\text{type} & \mathit{counter} & = & \mathit{Int} \\
\text{type} & \mathit{inst} & = & [\mathit{Inst}] \\
\text{type} & \mathit{store} & = & [\mathit{Int}]
\end{array}
$$

図 3.7: システムの状態

local ローカル変数の lexical address

reg レジスタの利用状況

code ラベル付けされた命令列の集まり

counter 局所ジャンプのためのカウンタ

inst 生成中の命令列

store グローバルなストア

global 大域変数の束縛状態

mem コンパイルの結果を格納するメモリ

このような内部状態をもつコンパイラを定義する *computation* は結果として図.3.8 のように monad transformer を適用していくことで得られる monad によって決定する．

3.2.1 内部状態へのアクセス

基盤言語のコンパイラを定義する際に利用される内部状態のアクセス関数 (*computation*) を，表.3.1 に示す．コンパイラの実際の動作の記述は，これらの関数の呼び出しを組み合わせて定義される．

$$
\begin{aligned}
&\mathit{type}\ M\ a = \\
&\quad (\mathit{EnvT}\ \mathit{global} \\
&\quad\quad (\mathit{EnvT}\ \mathit{local} \\
&\quad\quad\quad (\mathit{StaT}\ \mathit{mem} \\
&\quad\quad\quad\quad (\mathit{StaT}\ \mathit{reg} \\
&\quad\quad\quad\quad\quad (\mathit{StaT}\ \mathit{heap} \\
&\quad\quad\quad\quad\quad\quad (\mathit{StaT}\ \mathit{counter} \\
&\quad\quad\quad\quad\quad\quad\quad (\mathit{StaT}\ \mathit{inst} \\
&\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{StaT}\ \mathit{store}\ (\mathit{ErrT}\ \mathit{Id}))))))))\ a
\end{aligned}
$$

図 3.8: Monad Transformer の適用

表 3.1: コンパイラ定義のためのインターフェイス

名前	引き数	state	戻り値
compile	式		結果を渡すレジスタ
alloc-reg		reg	利用可能なレジスタ
free-reg	レジスタ	reg	
rdEnv			呼び出された時点の変数の状態
inEnv	環境,computation	local	
switch	コードセグメント	heap	更新される前のコード生成環境
store-inst	命令	inst	結果を受け取るレジスタ
inc-counter		count	更新後のカウンター
store-value	格納する値	store	値を格納した番地

3.3 モジュールのメタ表現

3.3.1 例:(算術式)

まず最初の例として簡単なコンパイラの定義を示す．図 3.9 の定義は，単純な整数演算のみを扱うコンパイラの動作をあらわしている．ここで示された各関数は，それぞれが整定数および数値演算を実装するモジュールの代表的な関数の定義を示したものである．

$$
\begin{array}{lll}
\text{compile} & :: & \mathit{Term} \rightarrow M\ \mathit{Register} \\
\text{compile (Num } n\text{)} & = & \text{(alloc-reg)} \\
 & \gg_{r1} & \langle n \rangle \\
 & \gg_{x} & \text{(alloc-store } x\text{)} \\
 & \gg_{a1} & \text{(store-inst (ld } \texttt{r1}\ \text{GP } \texttt{a1}\text{) } \texttt{r1}\text{)} \\
 & & \\
\text{compile (Add } t\ u\text{)} & = & \text{(compile } t\text{)} \\
 & \gg_{r1} & \text{(compile } u\text{)} \\
 & \gg_{r2} & \text{(store-inst (add } \texttt{r1}\ \texttt{r2}\text{) } \texttt{r1}\text{)} \\
 & \gg_{r1} & \text{(free-reg } r2\text{)} \\
 & \gg_{_} & \langle r1 \rangle
\end{array}
$$

図 3.9: A Simple Compiler

関数 **compile** は，型 $Term \rightarrow M\ Register$ をもち， 実際のコード生成の過程は，*bind* によって結合された，対応する *computation* の並びとして定義されている．

たとえば， 数値を表す項 $(Num\ n)$ に対するコンパイル手順は，まず新しいレジスタをアロケートし，数値 n をアドレス a に格納する．そして得られたレジスタにアドレス a の値を読み込む命令を生成する．

また，二項演算に関しては， 第一式と第二式を各々コンパイルし，その結果としてえられるレジスタ (結果を格納) 同士の演算をおこなう命令を生成する．コンパイル結果は図 3.10 に示されている．

$1 + 2 + 3$

(store) GP → [0 1 1 2 3]

(code)

```
L1 :  ld    reg0  GP  2
      ld    reg1  GP  3
      add   reg0  reg1
      ld    reg1  GP  4
      add   reg0  reg1
      halt  reg0
```

図 3.10: コンパイル結果

表 3.2: 条件式のコンパイルのための *computation* 一覧

名前	引き数	結果
set-address	レジスタ	番地
compile-count	式, 値を格納する番地	レジスタ
current-counter		カウンタの値

3.3.2 例:(条件式)

より複雑な例として， 条件式のコンパイルをおこなう computation を追加する．表 3.2 に， 必要な *computation* の一覧を示す.

条件式を扱うための *computation* の定義を 図 3.11 に示す．コード生成の流れを直感的に説明すると，まず最初に条件部のコンパイルをおこなう．次に *computation* “set-address” が分岐先を格納するための領域を確保し，分岐命令を生成．最後に “compile-count” を使って分岐先の計算をおこなう.

compile	::	$Term \rightarrow M\ Register$
compile (If $e1$ $e2$ $e3$)	=	(compile $e1$)
	$\gg_{r1}$	(set-address $r1$ $jump$)
	$\gg_{a1}$	(compile-count $e2$ $a1$ jump-then)
	$\gg_{r2}$	(set-address $r2$ brc-neq)
	$\gg_{a2}$	(compile-count $e3$ $a2$ jump-else)
set-address	::	$Term \rightarrow Instruction \rightarrow M\ Address$
set-address $reg1$ $inst$	=	(alloc-store $'dummy'$)
	$\gg_{addr}$	(store-inst (cmpi-eq `r1` `0`))
	$\gg_{r1}$	(alloc-reg)
	$\gg_{r2}$	(store-inst (ld `r2` `gptr` `addr`))
	$\gg_{r2}$	(store-inst ($inst$ $r2$))
	$\gg_{r2}$	(free-reg $r1$)
	$\gg_{\text{-}}$	(free-reg $r2$)
	$\gg_{\text{-}}$	$\langle addr \rangle$
compile-count	::	$Term \rightarrow Address \rightarrow M\ Register$
compile-count exp $addr$ x	=	(current-counter)
	$\gg_{c1}$	(compile exp)
	$\gg_{reg}$	(current-counter)
	$\gg_{c2}$	($store$ $addr$ $((c2 - c1) + x)$)
	$\gg_{\text{-}}$	$\langle reg \rangle$

図 3.11: 条件式のコンパイラ

第 4章

コンパイラ記述言語

我々はコンパイラの動作をユーザーが拡張可能な表現で提供する．本章ではそのためのメタ言語を示す．この言語で記述された部品は変換規則に基づいてコンパイラの実装レベルと同じレベルで動作するプログラムのソースコードに変換される．その記述形式は前章で示したコンパイラの動作を定義した関数定義と同等のものであり，今回我々が実装したシステムではコンパイラの実装言語はScheme であるため，変換結果としては Scheme で記述されたコンパイラのプログラム部品が生成され，コンパイル時に呼び出されることになる．(図.4.1).

4.1 言語部品の定義

前章では各構文についてコンパイラの各状態に対する直接参照操作の列を与えることによってコード生成動作の定義をおこなう方法を示した．しかしこのようなコード生成の過程を直接記述することには以下のような問題点がある．

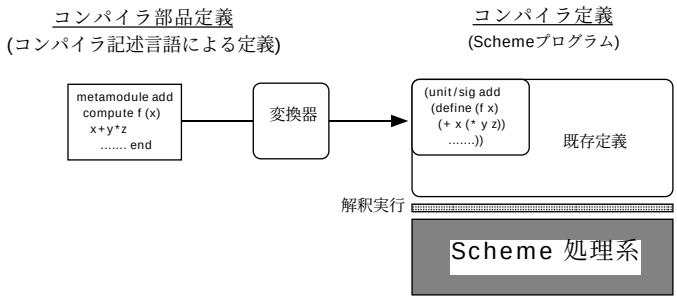

図 4.1: プログラム実行の流れ

- 部品の定義をおこなうための記述が複雑になる.
- 拡張モジュールの設計者が対象となる部品の全体的な構造を把握することが困難である
- 拡張をおこなう際に置き換えの対象となる記述部分を明確に指定することができない
- 他のモジュールの拡張による影響がうまく隠蔽されていないために再利用性が低下する
- モジュールが実現しようとしている機能とは直接は関係しない低レベルな記述の組み合わせによってある一つの部分的な機能を実現しているため，複数の抽象度をもつ記述が混在してしまう

これらの問題はモジュール定義のための記述の抽象度が最も低いレベルに設定されていたために生じるものでありコンパイラの動作記述の抽象化とそのモジュール化をいかに提供するかの問題に帰着する.

我々の方針として一つのモジュールの中では一貫した抽象度による記述をおこなうことができ，さらに一定の安全な拡張機構を提供するような隠蔽手法を提供する必要がある．すなわち前章で示した抽象度に応じたグループの分類と階層化を自然な形で提供できるようなモジュール化機構の提案が必須である.

そこで我々はコンパイラの動作を記述するためのより抽象度の高いインターフェイスを提供する．本節ではコンパイラのモジュラーな定義を補助するための高級なコンパイラ記述言語を定義し，その言語によって記述された定義を前章で示した直接実行可能な定義言語 (Defining-Language) による表現 (メタ表現) への変換規則を示す.

4.1.1 コンパイラ記述言語

まず各モジュールの定義に，被定義言語の構文を用いることで抽象的な定義手法を提供し可読性を向上させる．そこで基盤言語に対して図.4.2 の構文を追加定義する.

```
<pexp>    ::= <pexp> <op> <pexp> | <number> | <var>
           | <pexp> <pexp>
           | <pexp> ()
           | <pexp> ≫_var <pexp>         (unit operator)
           | ⟨ <pexp> ⟩        (bind operator)

<mtpt>    ::= val <var_name> = <pexp>
           |  fun <fun_name> (<arg>,...,<arg>) -> <pexp>
           |  val <var_name> (<var>) -> <exp>
           |  val <name>:(<var>,...,<var>) -> <pexp>
           |  val <name> (<arg>,...,<arg>) ->
                          (<var>,...,<var>)@{<pexp>,...,<pexp>}

<mdcl>    ::= <dcl>
           |  compute <name> (<arg>,...,<arg>)
              where meta
                  <mtpt>;...;<mtpt>
              in
                  <fdcl>;...;<fdcl>
              end

<metamod> ::= metamodule <mod_name> : <sig_name> (<var>,...,<var>)
              meta
                  <mtpt>;...;<mtpt>
              in
                  <mdcl>;...;<mdcl>
              end
```

図 4.2: コンパイラ記述のための追加構文

4.1.2 メタモジュールの定義

構文 <metamod> はコンパイラの部品を定義するための特殊なモジュール (メタモジュール) 定義文である．前章で示した置き換え可能なグループのコンポーネントを定義するために用いる．<mod_name> はモジュールの名前，<sig_name> はシグニチャ名，つづく変数名の並び (<var>,...,<var>) は参照されるモジュールのシグニチャ名である．ただし下位のレイヤにあるモジュールは特別指定する必要はない．これは同一グループ内でのモジュールの参照関係のみによって意味単位の間の関連性を設定するためである．

我々は，あるレベルの機能を実現するモジュールの定義に際して，より低レベルな記述を外部のモジュールとして設計する．そこで他のコンポーネントの拡張による影響が直接発生する可能性のある定義すなわち低位のモジュール内の定義を参照するような記述はそのモジュールにおけるコンパイラ記述言語のメタレベル (メタ宣言部) として分離することが自然である．すなわち各意味単位について特化されたコンパイラ記述言語が存在すると考えることができる．

次に meta から in で囲まれた部分 (共有メタ宣言部) の定義はモジュール内で共有されるメタ宣言部の定義であり，唯一外部のモジュールに公開することが可能なメタ宣言部である．共有メタ宣言部の定義は同一のレイヤに属し，このモジュールを参照するモジュールにとってもメタレベルであると考えることは容易にできる．基本的にこの部分で定義されるのはそのモジュールが実現するデータオブジェクトの構造定義である．いいかえるならば，このモジュールのためのコンパイラ記述言語特有のデータ構造を定義していると考えてもよい．

実は同一グループ内のモジュール間で明示的に参照可能な定義はこの共有メタ宣言部のみである．なぜならモジュール全体で共有されるメタな定義すなわちこのモジュールに特化したコンパイラ記述言語のベースとなる部分 (データ構造の定義) こそがこのモジュールの特徴を決定するものであるということができるからである．すなわち同一レイヤに存在するモジュールはそのコンパイラ記述言語において特定のデータ構造の定義を共有することによって依存関係をもつということになる．

なお以降で示すモジュール本体で定義される関数は基本的により上位のレイヤからメタレベルとして参照されるか，もしくはシステムによって暗黙的に呼び出されることになる．

4.1.3 メタモジュールの定義関数

図.4.2 の宣言 <mdcl> はコンパイラが実際に実行コードを生成する過程で呼ばれる関数を定義するためのものであり，以下のように与えられる．

- **メタ宣言部** (meta から in で囲まれた部分) の定義は，値の初期化や，他のモジュールで定義された関数の呼び出しを提供するためのメタ定義である
  - 変数定義 val <var_name> = <pexp> は，宣言部のすべての定義において参照可能な値の定義を示している．例えばメモリ領域の確保などコンパイルの初期段階でおこなう値の初期化などを定義する．
  - 変数パターン定義 val <name>:(<var>,...,<var>) -> <exp> はデータ構造 <name> の生成操作の定義をおこなっている．
  - 変数パターン定義
    val <name> (<arg>) -> (<var_0>,...,<var_n>)@{<exp_0>,...,<exp_n>}はデータ構造 <name> の i 番目の要素を参照する操作 exp_i の定義をおこなう．上の定義とあわせてコンパイラ記述言語のデータ構造の定義に用いられる．
  - 関数パターン定義 fun <name> (...) -> <exp> は，動作定義部で呼ばれるメタ関数の定義である．<exp> の定義内で輸入された外部のモジュールの変数や関数を参照することができる．
  - 関数定義 fun <name> (...) = <exp> は，動作定義の際に補助的に呼ばれる関数を定義している．
- **動作定義部** (in から end で囲まれた部分) はコード生成動作の実際の定義であり，メタ宣言部の定義に従ってその意味が決定される．まずその時点で compute 文で定義している関数と同名で引数の形式が同じ関数が必ず存在する必要がある．各定義関数の引数はすべてコンパイル時に決定されるものとする．ここで動作定義部で与えられたの関数の部分式はメタ定義式とベース定義式とに分類される．メタ定義式は実行時の仮想機械の動作を定義しており，ベース定義式はコンパイル時に評価される式を表している．以下がメタ定義式とベース定義式の定義である (判定のための規則を図.4.3 に示す)．
  - ある動作定義関数は全体としてはメタ定義式である．

$$
\begin{array}{lcl}
\mathcal{K} :: \mathit{Expr} \to \mathit{benv} \to \mathit{menv} \to \{\mathit{Base}, \mathit{Meta}\} & & \\
\mathcal{K}[\![\mathit{exp}_1 \ldots \mathit{exp}_n]\!]\ \rho\ \delta & = & \mathcal{K}[\![\mathit{exp}_1]\!]\ \rho\ \delta \\
\mathcal{K}[\![\text{let } \mathit{dcls} \text{ in } \mathit{exp} \text{ end}]\!]\ \rho\ \delta & = & \mathcal{K}[\![\mathit{exp}]\!]\ \rho \cup \{\mathit{var}_i \mid \mathcal{K}[\![\mathit{exp}_i]\!] = \mathit{Base}\} \\
 & & \qquad \delta \cup \{\mathit{var}_j \mid \mathcal{K}[\![\mathit{exp}_j]\!] = \mathit{Meta}\} \\
 & & \mathit{dcls} ::= \text{val } \mathit{var}_1{=}\mathit{exp}_1\ ; \ldots ;\ \text{val } \mathit{var}_n{=}\mathit{exp}_n \\
\mathcal{K}[\![\mathit{number}]\!]\ \rho\ \delta & = & \mathit{Base} \\
\mathcal{K}[\![\mathit{var}]\!]\ \rho\ \delta & = & \mathit{Base} \quad (\mathit{if}\ \mathit{var} \in \rho) \\
 & & \mathit{Meta} \quad (\mathit{otherwise}\ \mathit{if}\ \mathit{var} \in \delta) \\
\mathcal{K}[\![\text{if } \mathit{exp}_1 \text{ then } \mathit{exp}_2 \text{ else } \mathit{exp}_3]\!]\ \rho\ \delta & = & \mathcal{K}[\![\mathit{exp}_1]\!]\ \rho\ \delta \\
\mathcal{K}[\![\mathit{exp}_1 \text{ op } \mathit{exp}_2]\!]\ \rho\ \delta & = & \mathcal{K}[\![\mathit{exp}_1]\!]\ \rho\ \delta \wedge \mathcal{K}[\![\mathit{exp}_2]\!]\ \rho\ \delta
\end{array}
$$

benv は局所変数の環境，menv はメタ定義部で宣言された名前を示す環境．

$(\mathit{Meta} \wedge \mathit{Meta} = \mathit{Meta}, \mathit{Base} \wedge \mathit{Meta} = \mathit{Meta}, \mathit{Meta} \wedge \mathit{Base} = \mathit{Meta}, \mathit{Base} \wedge \mathit{Base} = \mathit{Base})$

図 4.3: Level 判定関数 $\mathcal{K}$

- 関数 `compile` の呼び出しはメタ定義式である．
- メタ宣言部で定義された関数パターンの呼び出しは，すべてメタ定義式である．
- メタ定義式を部分式に持つような算術および比較演算は，すべてメタ定義式である．
- メタ定義式を条件判定部にもつような条件式，はすべてメタ定義式である．
- メタ定義式に束縛された変数および構造体の宣言は，すべてメタ定義式である．
- 式 `begin .... end` は，最後に評価される式がメタ定義式であるならばメタ定義式である．
- 上記以外の式はすべてベース定義式である．

ここで `compile` はコンパイラの再帰的呼び出しを示す特殊な関数．

このような記法を採用することで，実行コード生成部の抽象度の高い定義をおこなうことができる．以降ではいくつかの例をあげてこのコンパイラ記述言語の用法を解説する．

4.1.4 単純な例 (算術式)

まず前章でも示した単純な算術演算のコンパイル方法を記述したモジュール群の定義を，図.4.4 および 図.4.5 に示す．まず図.4.4 の算術式についてであるが，これは単純に二つの部分式を評価した結果を足し合せる操作を記述したものである．また図.4.5 の整数値のコンパイル方法については，あらかじめ値をあるメモリ領域に格納しておいて実行時にそのアドレスからロードするという操作を記述している．

4.1.5 より複雑な例 (ベクトル)

図.4.7，4.8 の定義は，ベクトルのコンパイルのためのモジュール定義である．まず共有メタ宣言部の最初の二つの定義

```
val vect:(x,y) -> make_cell x y;
val vect (x)   -> (y,z)@{cell_ref y x 0,cell_ref z x 1};
```

はベクトルのデータ構造の実行時のアロケートおよび参照操作を定義したもので外部に公開される (図.4.6).

`compile_vector` は，式がベクトルの定義 (例えば `<1,2,3>`) であった時に呼ばれる関数であり外部に公開される (図.4.6)．`compile_vector` は，ベクトル式の要素を先頭から順番に評価して，その結果を連続領域に格納するという動作を示す.

まずメタ宣言部の定義

```
val addr  = alloc_stores len;
val addr2 = alloc_store addr;
val addr3 = alloc_store len;
```

は，ベクトル要素格納のための連続領域 (`addr`) とその番地を直値として格納する領域 (`addr2`)，およびベクトルの長さを格納する領域のアロケートを行なっている．つづく二つのパターン定義

```
fun set_vect (idx,v) -> st:(v,gptr,(addr + idx));
fun alloc_vect ()    -> vect:(addr2,addr3);
```

```
metamodule arith : arith_sig (const_sig)
in
    compute compile_add (exp as add:(e1,e2))
    where meta
        fun add (e1,e2) -> add:(e1,e2)
    in
        fun compile_add (exp as add:(e1,e2))
          = add (compile e1) (compile e2);
    end
end;
```

図 4.4: 数値演算の定義

```
metamodule const : const_sig ()
in
    compute compile_num (exp as num:(n))
    where meta
      val addr = alloc_store n;
      val x    = alloc_reg ();
      fun get_num () -> ld:(x,gptr,addr)
    in
      fun compile_num (exp as num:(n)) = get_num ()
    end
end;
```

図 4.5: 整数定数の定義

は，それぞれ実行時の，ベクトル領域の idx 番目に値 v を格納する操作 (store 命令) およびベクトルを指すセルを一つ確保する操作を与えている．

これらのメタ定義に基づいて動作定義部は実際のコード生成の流れを与えている．

```
fun comp_vect (lst,n,max) =
if (lst == []) then alloc_vect ();
else begin
    set_vect (max - n) (compile (hd lst));
    comp_vect (tl lst) (n - 1) max
end
```

要約すると，式のリストを先頭からコンパイルしていってリストが空になるまで順番にメモリ上に格納していく操作をあらわしている．メタ定義式の条件によって部分式のコンパイルおよびそれをリストが空になるまで再帰的におこなう操作はコンパイル時に行なわれ，実行時には部分式の評価とその結果のメモリへの格納とメモリセルの割り当てを行なう．実際には部分式の評価結果を特定のレジスタに格納する必要があるがその部分は自動化されている．この部分を拡張可能にするためにさらにメタレベルの拡張機構が必要であるため課題として残される．

また同様に図.4.8 の compile_vref の定義は，ベクトル要素の参照 (例えば vref <1,2,3> 2) の定義を示している．まずメタ宣言部の定義

```
fun ref_vect (y,x) -> ld:(y,gptr,x)
```

は，y に x 番目のストアに格納された値を読み込む実行時の動作を定義している．また動作定義部

```
fun compile_vref (exp as vrf:(vect,index)) =
let val y = compile index
in
  begin
    let val vect:(sto,len) = compile vect
    in
      if (len < y) then err ()
```

```
signature vect_sig
meta
    val vect:(adr,len);
    val vect (x)
in
    compute compile_vector;
    compute compile_vref
end
```

図 4.6: The signature of the Vector Module

```
      else ref_vect y (sto + y)
    end
  end
end
```

を要約すると，まずコンパイル時には参照する場所を示す式をコンパイルし，次にベクトル本体をコンパイルするその結果は，メタ定義式なのでベクトル構造の実行時の定義方法に従ってアロケーションのための命令を生成する．

実行時には参照する場所がベクトルの長さを超えていないかどうかを検査し (条件式の部分) 問題がなければその要素を返す．また問題があれば例外を発生する．

4.2 変換規則

前節で示したように実行時の動作とコンパイラの動作を分離することで簡潔な記述が可能となる．本節ではこのようにして与えられた定義から実際に動作するコンパイラの定義を導き出すための変換規則を与える．変換の基本的な目的はメタ定義式を同等な実行時のコードに，ベース定義式をコンパイル時に直接実行される式に変換する作業である．

その詳細としては

- 式 `if e1 then e2 else e3` において部分式 `e1` がベース定義式であるなら，コンパイラの定義言語における条件式にそのまま対応させる．`e1` がメタ定義式であるなら，条件式をコンパイルしたコードに各部分式の変換結果を埋め込んだものを対応させる．

```
metamodule vector : vect_sig ()
meta
    val vect:(x,y) -> make_cell x y;
    val vect (x)   -> (y,z)@{cell_ref y x 0,cell_ref z x 1};
in
    compute compile_vector (vect:(len,lst))
    where meta
        val addr  = alloc_stores len;
        val addr2 = alloc_store addr;
        val addr3 = alloc_store len;
        fun set_vect (idx,v) -> st:(v,gptr,(addr + idx));
        fun alloc_vect ()    -> vect:(addr2,addr3);
    in
        fun compile_vector (exp as vect:(len,lst))
            = comp_vect lst len len;
        fun comp_vect (lst,n,max) =
        if (lst == []) then alloc_vect ();
        else begin
               set_vect (max - n) (compile (hd lst));
               comp_vect (tl lst) (n - 1) max
             end
        end
    end
    ....
end
```

図 4.7: ベクトル式に対するコンパイラ定義 (前半)

```
metamodule vector : vect_sig ()
meta
    val vect:(x,y) -> make_cell x y;
    val vect (x)   -> (y,z)@{cell_ref y x 0,cell_ref z x 1};
in
    ....

    compute compile_vref (exp as vrf:(vect,index))
    where meta
        fun ref_vect (y,x) -> ld:(y,gptr,x)
    in
        fun compile_vref (exp as vrf:(vect,index)) =
        let val y = compile index
        in
          begin
            let val vect:(sto,len) = compile vect
            in
             if (len < y) then err ()
              else (ref_vect y (sto + y))
            end
          end
        end
    end
end;
```

図 4.8: ベクトル式に対するコンパイラ定義 (後半)

- ある算術式がベース定義式であるなら，コンパイラの定義言語における算術式にそのまま対応させる．メタ定義式であるなら，算術式をコンパイルしたコードに各部分式の変換結果を埋め込んだものを対応させる．
- 定数は上位のメタ定義式の部分式として埋め込まれる場合には定数式をコンパイルしたコードを埋め込み，それ以外は対応する定数そのものを与える．
- 式 `let dcl_1;...;dcl_n in e end` は各宣言 `dcl_i` を *computation* の列として変換したものと `e` を変換したものの結合とする．
- 式 `begin .... end` は，各部分式を *computation* の列として変換したものであり，各部分式の変換の扱いはこの式全体の分類によるものとする．

具体的な変換規則は図.4.9 に与えてある (コンパイラの定義言語として Scheme をもちいている)．ここで型 *Mbind* はメタ定義部で定義された関数パターンの名前の束縛関係である．また型 *Bbind* は通常の変数の束縛関係を示す．

まず，メタ宣言部において

$$\text{val } name = exp$$

の形式で定義されているものを探し，定義された順番に結合をおこなう．例えば `function1` を定義している場合には以下のような初期化関数を生成する (一部構文を読み易く変更してある)．ここで `exp_i'` は `exp_i` に対応する Scheme の式である．

```
val name_0 = exp_0
      ...
val name_n = exp_n
    ⇓
(define (init_function1 x0 ... xm)
  (exp'_0 ≫_name_0
   ... ≫_name_n-1
   exp'_n ≫_name_n
    (inMetaEnv (vector name_0 ... name_n)
        (call_function1 x0 ... xm)))
```

次にメタ宣言部において関数パターンの定義

$$\text{fun } name\ (arg_0, \ldots, arg_n)\ \Rightarrow \text{exp}$$

を取り出し，次のように変換する．(ただし exp' は exp に対応する Scheme の式である).

```
fun fname (x1,...,xn) = exp
    ⇓
(lambda (x1 ... xn)
  (rdMetaEnv
    ≫r
    (letrec ((name0 (vector-ref r 0))
               ...
             (namen (vector-ref r n))) exp')))
```

この変換結果を関数名 fname に束縛したものを環境 Mbind に追加する同様の変換を関数パターンに対して適用した状態で動作定義部の各関数に対して図.4.9 の変換を適用すれば Scheme 処理系上で直接実行可能なコンパイラのコードが生成される．

データ構造定義の変換についても同様の手法が適用されるが変換規則が複雑になるためここでは割愛した．

図.4.9 の変換規則中に出現する式

$$(\text{compile}\ '(\text{embed}\ exp_1)\ +\ (\text{embed}\ exp_2)'\ )$$

において，その部分式 (embed <exp>) は式 <exp> の評価結果がコンパイル時にそのまま埋め込まれることを示している．この機構によってより抽象度の高い記述を提供することができる．

$\mathcal{T} :: Expr \rightarrow Bbind \rightarrow Mbind \rightarrow MetaRep$

$$
\begin{array}{lcl}
\mathcal{T}[\![exp_1 \ldots exp_n]\!]\ \rho\ \delta & = & \mathcal{T}[\![exp_{m_1}]\!]\ \rho\ \delta \gg_{a_1} \ldots \gg_{a_{m_{(k-1)}}} \mathcal{T}[\![exp_{m_k}]\!]\ \rho\ \delta \\
 & & \gg_{a_k} (b_1 \ldots b_n) \\
 & & \gg_{res} (\text{free-regs}\ (a_1 \ldots a_k)) \\
 & & \gg_{_} (\text{unit}\ res) \\
 & & (b_i = a_i \text{ if } (\mathcal{K}[\![exp_i]\!] = Meta) \text{ else } b_i = \mathcal{T}[\![exp_i]\!]\ \rho\ \delta) \\
 & & (1 \leq m_i \leq n\ s.t.\ \mathcal{K}[\![exp_{m_i}]\!] = Meta) \\
\mathcal{T}[\![\text{let } dcls \text{ in } exp \text{ end}]\!]\ \rho\ \delta & = & \mathcal{T}[\![exp_{m_1}]\!]\ \rho\ \delta \gg_{v_{m_1}} \ldots \gg_{v_{m_{(k-1)}}} \mathcal{T}[\![exp_{m_k}]\!]\ \rho\ \delta \\
 & & \gg_{v_{m_k}} \mathcal{T}[\![exp]\!] \\
 & & \rho\{(var_{b_1}, \ldots, var_{b_k}) \mapsto (\mathcal{T}[\![exp_{b_1}]\!]\ \rho\ \delta, \ldots, \mathcal{T}[\![exp_{b_k}]\!]\ \rho\ \delta)\} \\
 & & \delta\{(var_{m_1}, \ldots, var_{m_k}) \mapsto (v_{m_1}, \ldots, v_{m_k})\} \\
 & & \gg_{res} (\text{free-regs}\ (v_{m_1} \ldots v_{m_k})) \gg_{_} (\text{unit}\ res) \\
 & & (dcls ::= \text{val}\ va\rho{=}exp_1\ ; \ldots ;\ \text{val}\ var_n{=}exp_n) \\
 & & (1 \leq m_i \leq n\ s.t.\ \mathcal{K}[\![exp_{m_i}]\!] = Meta) \\
 & & (1 \leq b_i \leq n\ s.t.\ \mathcal{K}[\![exp_{b_i}]\!] = Base) \\
\mathcal{T}[\![number]\!]\ \rho\ \delta & = & number \\
\mathcal{T}[\![var]\!]\ \rho\ \delta & = & var \qquad (\textit{if var is bound in } \rho) \\
 & | & (\delta\ var) \qquad (\textit{if var is bound in } \delta) \\
\mathcal{T}[\![\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3]\!]\ \rho\ \delta & = & (\text{if}\ \mathcal{T}[\![exp_1]\!]\ \rho\ \delta\ \mathcal{T}[\![exp_2]\!]\ \rho\ \delta\ \mathcal{T}[\![exp_3]\!]\ \rho\ \delta) \\
 & & \qquad \textit{(when the expression is Base)} \\
 & | & (\text{compile}\ '\text{if (embed}\ \mathcal{T}'[\![exp_1]\!]\ \rho\ \delta) \\
 & & \qquad \text{then (embed}\ \mathcal{T}[\![exp_2]\!]\ \rho\ \delta) \\
 & & \qquad \text{else (embed}\ \mathcal{T}[\![exp_3]\!]\ \rho\ \delta)') \\
 & & \qquad \textit{(when the expression is Meta)} \\
\mathcal{T}[\![exp_1\ op\ exp_2]\!]\ \rho\ \delta & = & (op\ \mathcal{T}[\![exp_1]\!]\ \rho\ \delta\ \mathcal{T}[\![exp_2]\!]\ \rho\ \delta) \\
 & & \qquad \textit{(when the expression is Base)} \\
 & | & (\text{compile}\ '(\text{embed}\ \mathcal{T}'[\![exp_1]\!]\ \rho\ \delta)\ op\ (\text{embed}\ \mathcal{T}'[\![exp_2]\!]\ \rho\ \delta)'\ ) \\
 & & \qquad \textit{(when the expression is Meta)}
\end{array}
$$

図 4.9: 変換規則 $\mathcal{T}[\![exp]\!]$

$$
\begin{array}{lcll}
\mathcal{T}' :: \mathit{expression} \rightarrow \mathit{lbind} \rightarrow \mathit{gbind} \rightarrow \mathit{MetaRep} & & & \\
\mathcal{T}'[\![\mathit{number}]\!]\ \rho\ \delta & = & (\mathrm{compile}\ '\mathit{number}') & \\
\mathcal{T}'[\![\mathit{var}]\!]\ \rho\ \delta & = & \langle \mathit{var} \rangle & (\mathit{if\ var\ is\ bound\ in}\ \rho) \\
 & \mid & (\delta\ \mathit{var}) & (\mathit{if\ var\ is\ bound\ in}\ \delta)
\end{array}
$$

For the other expressions, $\mathcal{T}'$ is the same as $\mathcal{T}$ except for its sub-translation $\mathcal{T}$ is replaced as $\mathcal{T}'$

図 4.10: 変換規則 $\mathcal{T}'[\![\mathit{exp}]\!]$

第 5章

言語処理系の拡張

5.1 言語拡張の適用方法

本研究の枠組では，言語の拡張の最小単位は構文レイヤーおよび共有レイヤーに存在する言語の機能毎に提供された各部品であるが，実際に意味のある言語拡張は，複数の部品に対する拡張との連携によって実現されることが多い．またコード生成部品以外の拡張も同時に適用されると考えることが自然である．また，コード生成部分の拡張以外で必要な操作もいくつか存在する．エンドユーザーレベルで実際に利用する拡張部品は以下の操作のうち必要な要素を列挙したファイルによって与えられるものである．

```
# 各言語部品の読み込み
  load <file-name>

# 言語部品の追加
  create <module-name> <layer>

# 既存の言語部品に対する拡張記述の複数の選択肢
# リストで与えられたものから衝突を起さない定義を選択
  extend <module-name> [<ext_1>,...,<ext_n>]

# コンパイラの内部状態（Monad Transformer）の追加
  extend EnvMonad <StateName> <updater-name>
```

```
  extend StateMonad <EnvName> <reader-name> <updater>

# パーサの拡張
  NewSyntax <datatype> <pattern>

# 新しい構文のディスパッチ部への登録
  Entry <datatype> <module-name>.<function-name>
```

こうした単一の拡張のために必要な道具をまとめたものをパッケージと呼ぶことにする．各パッケージは名前をもち，そのパッケージに属するものの名前はパッケージ名によって修飾される．そこで同一名のパッケージは同じものと考えることによって同じ名前で異なる意味をもった言語部品が存在することを避けることができる．すなわち名前の衝突を回避することが可能となる．

5.2 各部品の拡張

本節では前章の高級な記法によって提供されたコンパイラの各部品を拡張するための手法について述べる．まずモジュール拡張のために，以下の記法を提供する．

```
extension <ext-name> : <ext-sig> (<arg>,...,<arg>)
where meta
    <mtpt>;...;<mtpt>
in
    <mdcl>;...;<mdcl>
end;
```

ここで (<arg>,...,<arg>) は追加して参照するモジュールのシグニチャ．本体の定義は被拡張モジュール内の関数およびメタ宣言の追加もしくは再定義である．拡張モジュールで記述される操作の詳細とその意味は以下のものである (典型的な例を図.5.1 に示す).

- 新たな compute 宣言された関数を extension 宣言の内部で，通常のモジュール定義と同様に宣言することで追加する．

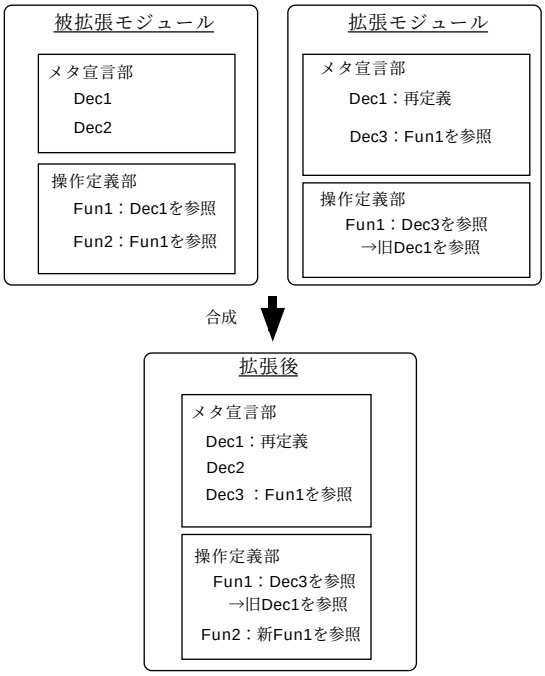

図 5.1: モジュールの拡張

- 特定の compute 宣言された関数について，その内部宣言の変更を行う．
- メタ宣言部の宣言を拡張する方法は以下のいずれか
  - モジュール宣言の記法に従って新たなメタ宣言を追加する．
  - 宣言をそのまま使う (継承する) 場合は，何も記述しない．
  - 被拡張関数の動作定義部の関数を明示的に参照する場合はメタ定義部における右辺で関数呼び出しの形式をとる．
  - 同一名の宣言を定義することで既存の宣言を再定義 (上書き) する．
  - 宣言を除外する場合は，() を代入する．
- 動作定義部の関数を拡張する方法は以下のいずれか
  - 動作定義部の関数呼び出しは基本的に拡張後のものを参照する．拡張前の関数を明示的に呼び出す場合はメタ宣言を経由する．
  - 関数を再定義 (上書き) する場合は，拡張モジュール内で新たに宣言する．
  - 新たな関数を定義する．

通常の継承関係と比較した場合の特徴として，動作定義部の拡張は関数の再定義および追加定義に限定されるということ，および他のモジュール (被拡張モジュールを含む) の関数を参照する部分をメタレベルとして分離したことで，各関数の性質を論ずることが容易になったことが挙げられる．結果としてメタ宣言部を変更することで動作定義関数の性質を変化させることも可能となる．またそのような関数の性質を利用することで，衝突回避のためのより柔軟な継承の制御をおこなうことが可能になる．

5.3 言語拡張の例

本節では前節で示した拡張機構をもちいた簡単な言語拡張の例を示す.

5.3.1 算術演算の拡張例

図.4.4 および図.4.5 から得られる整数演算の定義は, 扱うことが可能な値が整数のみであるような言語のものであった. しかし, 実際の言語では他にベクトルやリストのように, より複雑な値を扱うことができる. そこで, 整数以外のデータを動的にアロケートされるオブジェクトとし, 整数と他のオブジェクトの区別のために, データにタグを付けることにする. そこで, 数値演算と, 定数に関するモジュールを変更する

図.5.3 および図.5.4 は一組で整数演算モジュールの拡張例である．この拡張の意味は, 部分式を評価した結果が整数であるか判定して整数であればタグを除去する．整数でなければ例外を発生する．というものである．この結果動的な型検査を実現することができた．この拡張記述はメタ表現では図.3.9 の定義から図.5.2 定義を生成する操作である．結果として図.5.5 に示すように動的型判定のためのコードが図.3.10 の生成コードに対して埋め込まれている.

5.3.2 関数呼び出しの拡張

以下は巻末で定義されている関数適用およびクロージャの定義モジュールを拡張するものである．これらの定義は常に一組で適用されるものであり，同一のパッケージにあるものとする.

```
extension cls_ext : cls_with_id_sig ()
```

$$
\begin{array}{lll}
\text{datatype } \mathit{value} & = & \mathit{Int} \mid \mathit{Clos} \mid \mathit{Vect} \mid \mathit{List} \\
\text{compile} & :: & \mathit{Term} \rightarrow M\ \mathit{value} \\
\text{compile (Num } n\text{)} & = & \text{(alloc-reg)} \\
 & \gg_{r1} & \text{(allow-tag } n\text{)} \\
 & \gg_{x} & \text{(store-value } x\text{)} \\
 & \gg_{a1} & \text{(store-inst (ld } \texttt{r1} \text{ GP } \texttt{a1}\text{))} \\
 & & \\
\text{compile (Add } t\ u\text{)} & = & \text{(compile } t\text{)} \\
 & \gg_{r1} & \text{(check-val } r1\text{)} \\
 & \gg_{r1} & \text{(store-inst (shr } \texttt{r1}\text{))} \\
 & \gg_{r1} & \text{(compile } u\text{)} \\
 & \gg_{r2} & \text{(check-val } r2\text{)} \\
 & \gg_{r2} & \text{(store-inst (shr } \texttt{r2}\text{))} \\
 & \gg_{r2} & \text{(store-inst (add } \texttt{r1}\ \texttt{r2}\text{))} \\
 & \gg_{r1} & \text{(store-inst (shl } \texttt{r1}\text{))}
\end{array}
$$

図 5.2: 数値演算および整定数の拡張

```
extension object_add : add_sig (check_sig) in
  compute compile_add (vect:(len,lst))
  where meta
      fun add (e1,e2)   -> add:(e1,e2) ≫v shl:(v);
      fun compile (exp) -> compile exp ≫v check_val v ≫_ shr:(v)
  end
end;
```

図 5.3: 数値演算の拡張モジュール

```
extension taged_const : const_sig () in
  compute compile_num (exp as num:(n))
  where meta
      fun get_num () -> ld:(x,gptr,addr) ≫x shl:(x)
  end
end;
```

図 5.4: 整定数の拡張モジュール

$1 + 2 + 3$

(store)		GP $\rightarrow$	[0 2 2 4 6]
(code)	$L1:$	ld	reg_0 GP 2
		ld	reg_1 GP 3
		shr	reg_0
		excp	$crreg_0$
		shr	reg_1
		excp	$crreg_1$
		add	reg_0 reg_1
		shl	reg_0
		ld	reg_1 GP 4
		shr	reg_0
		excp	$crreg_0$
		shr	reg_1
		excp	$crreg_1$
		add	reg_0 reg_1
		shl	reg_0
		halt	reg_0

図 5.5: 拡張後のコンパイル結果

```
meta
  val cls:(code,parm,id) -> make_4cell code parm id;
  val cls (x) -> (y,z,w,v)@{ld:(y,x,0),
                            ld:(z,x,1),
                            ld:(w,x,2)}
in
  compute compile_closure (exp as cls:(vars,body))
  where meta
    fun get_closure (x,y) -> cls:(x,y,unique_id ());
  end
end
```

上記の拡張定義は実行時にクロージャをメモリ上で確保する際に一意の番号を付ける操作である．クロージャのデータ構造の定義を変更し，番号を付ける操作を追加している．

```
extension app_ext : app_with_id_sig (cache_sig)
in
    compute compile_application (exp as app:(rator,rands))
    where meta
      fun call_cls (code,lst) -> call_closure code lst
    in
      fun call_closure (rator as cls:(code,parm,id),lst) =
      let val res = ref_cache id lst
      in
        if res then res else (call_cls rator lst)
      end
    end
end
```

上記の拡張定義はクロージャの適用時に，クロージャの識別番号と引数をキーとしてキャッシュを参照している．ここではキャッシュは外部で定義されているものとしている拡張の結果として頻繁に呼ばれるような関数は計算結果が即時に返ることが期待できる．

5.3.3 数値演算の拡張 (2)

数値演算の拡張をおこなうことで簡単な効率化をおこなうことも可能である．以下は整定数同士の演算を判定し，コンパイル時に計算するように変更した例である．

```
metamodule arith : arith_sig ()
in
    compute compile_add (exp as add:(e1,e2))
    where meta
        fun add (e1,e2) -> add:(e1,e2)
    in
        fun compile_add (exp as add:(e1,e2)) =
        if (num? e1) and (num? e2)
        then (e1.n + e2.n)
        else add (compile e1) (compile e2);
    end
end;
```

5.3.4 数値演算の拡張 (3)

数値演算の拡張をおこなう別の例として任意長の整定数をリストとして表現する例を示す (図.5.7, 図.5.6)．この場合，本来は整定数上の計算をおこなう全ての演算に対して拡張記述が定義されなくてはならない．

```
metamodule arith : arith_sig (list_sig)
in
    compute compile_add (exp as add:(e1,e2))
    where meta
        fun add (e1,e2)  -> add:(e1,e2);
        fun length (lst) -> get_length lst;
        fun reverse (lst) -> get_reverse lst;
        fun normal (lst) -> allow_normal lst
    in
        fun compile_add (exp as add:(e1,e2)) =
        let val v1 = reverse (compile e1);
            val v2 = reverse (compile e2)
        in
          if (length v1) > (length v2)
          then
            normal (add_vnum v1 v2)
          else
            normal (add_vnum v2 v1)
        end;

        fun add_vnum (lst1,lst2) =
        if lst2 == [] then lst1
        else
         add (compile (hd lst1)) (compile (hd lst2))
          :: add_vnum (tl lst1) (tl lst2)
    end
end;
```

図 5.6: 任意長の整定数の演算

```
metamodule const : const_sig (list_sig)
in
    compute compile_num (exp as num:(lst))
    where meta
      fun comp (n) -> compile_num num:(n)
    in
      fun compile_num (exp as num:(n)) =
      if (lst == []) then []
      else comp (hd n) :: compile_num (tl n)
    end
end;
```

図 5.7: 任意長の整定数とその演算

第 6章

拡張における衝突回避

6.1　衝突の可能性のある拡張記述の組み合わせ

本節ではモジュラーなコンパイラの拡張における衝突の具体例を示し衝突回避のための考察をおこなう.

6.1.1　変数参照の拡張例

ここで二つの拡張機能を同時に適用した場合に想定される問題の例を示す．まず大域変数の参照制限のための新たな構文 `restriction` を付加する．図.6.1 で示すようにこの宣言文の内部で参照可能な大域変数は許可されたもの (例では `x`) のみである.

次に最小値をもった変数宣言のための新たな構文 `minval` を付加する．図.6.2 で示すようにこの宣言文で定義された名前は最小値が設定され，その名前の変数の束縛された値が最小値以下の場合は最小値が与えられる．この際宣言された名前のスコープは大域変数と同じものとする.

6.1.2　拡張モジュールの記述

ここでは二つの拡張のためのモジュールおよび被拡張モジュールである `identifier` の定義を実際に示し，発生しうる問題点を示す.

図.6.5 は変数参照に関するコンパイル方法の定義である．局所変数の環境からレキシカルアドレスを取得し，その結果に応じて局所変数もしくは大域変数の参照のためのコード

```
val x = 1;
val y = 2;
restriction (x)
in
  print x;
  print y
end;
--> 1
y : unbound variable
```

図 6.1: 大域変数の参照制限

```
minval limit = 100;
let val limit = 10 in
  print limit
end;
--> 100
```

図 6.2: 変数の最小値の設定

```
minval limit1 = 100;
minval limit2 = 100;
val x = 1;
restriction (limit2)
in let val limit1 = 10;
       val limit2 = 10 in
     print limit1;
     print limit2;
     print x
   end
end;
-->10
-->100
x : unbound variable
```

図 6.3: 拡張機能の組み合わせ (1)

```
minval limit = 100;
val x = 1;
restriction ()
in let val limit = 10
   in
     print limit;
     print x
   end
end;
-->100
x : unbound variable
```

図 6.4: 拡張機能の組み合わせ (2)

を生成する操作を呼び出すという手続きが定義されている

図.6.6 は変数名に最小値を設定するための拡張パッケージのうち特に変数参照に対する拡張モジュールを示したものである．これは実行時に与えられた変数名に束縛された値を比較する操作をおこなうように拡張している．より詳細な定義は，以下のものである．

1. `minval` 文で宣言された変数を特別なデータ構造 `minval:(num)` として大域的な環境に登録
2. 変数参照時に与えられた変数名とその名前で登録された最小値とを比較し，より大きな値をもつものを返り値とする．

また図.6.7 は大域変数の参照制限のための拡張パッケージのうち特に変数参照に対する拡張モジュールを示したものである．これは参照可能な大域変数名を保持した環境から与えられた変数名を探して見付かった場合に限って参照を許可するという拡張である．より詳細な定義は，以下のものである．

1. `restriction` 文で許可された変数を専用の環境に保存
2. 大域変数の参照で環境に存在するものは許可
3. なんらかの値をもつ局所変数でも参照を許可された大域変数でもない変数参照は実行時に未定義変数の参照であるものとして例外処理となる

6.1.3 優先度の原理

ここで我々が想定する拡張モデルではある拡張単位においては，適用順位がより遅い拡張モジュールの性質が優先される，すなわちあとから適用された最新の拡張の性質が強くなるという前提を与える．そのための指針として以下の優先度の原理を与え，拡張の組み合わせにおいてこの原理に従わないようなものは他の拡張との間で衝突をおこしているものとする．

1. 同一の内部状態へのアクセスが競合する場合，優先度の高いものが基本的に採用される．

```
metamodule identifier : id_sig ()
in
    compute compile_id (exp as id:(name))
    where meta
        val env  = rdEnv;
        val ladr = search name env;
        val addr -> alloc_lst name;

        fun depth ()              -> get_depth ();
        fun ref_store (n,dep,off) -> ref_fvar n dep off;
        fun search_global (n)     -> search:(adr,length n)
    in
        fun compile_id (exp as id:(name)) =
        if ladr then ref_local ladr
        else
        let val res = ref_global name
        in
          if res then res
          else error ()
        end;

        fun ref_local (pair:(dep,off)) = ref_store (depth ()) dep off;
        fun ref_global (name) = search_global name
    end;
end;
```

図 6.5: 変数参照の定義モジュール

```
extension ext1 : id_sig (min_sig) in
compute compile_id (exp as id:(name))
where meta
  fun ref_lcl (res) -> ref_local res;
  fun min? (res)  -> check_minval res
in
  fun ref_local (res) =
  let val res1 = ref_lcl res
      val res2 = ref_global name
  in
    if res2 and (min? res2) then
      if (res1 < res2.num) then res2.num else res1
    else res1
  end
end
end;
```

図 6.6: 拡張モジュール (最小値設定)

```
extension ext2 : id_sig () in
compute compile_id (exp as id:(name))
where meta
  val genv = rdGEnv;
  fun ref_glb (name) -> ref_global name
in
  fun ref_global (name) =
  let val res = search_name name genv
  in
    if res then ref_glb name
    else false
  end
end
end;
```

図 6.7: 拡張モジュール (参照の制限 1)

```
extension ext1' : id_sig () in
compute compile_id (exp as id:(name))
  fun ref_lcl (res) -> ref_local res;
  fun min? (res) -> check_minval res
in
  fun compile_id (exp as id:(name)) =
  let val y = ref_global name in
    if ladr then
    let val x = ref_lcl ladr in
      if y and (min? y) then
        if (x < y.num) then y.num else x
      else x
    end
    else
    if y then
      if (min? y) then y.num else y
    else error ()
  end
end
```

図 6.8: 拡張モジュール (最小値設定 2)

2. 拡張同士でその影響が有効なスコープが重なる場合には，重なった部分のみに原理が適用される．

3. 優先度の高い拡張による影響の外では優先度の低いものであっても有効である．

6.1.4 衝突を起す事例

ここで二つの拡張 (ext1，ext2) を何の制約も課さず同時に適用した場合を想定したい．すると図.6.3 のような結果を返すようなコンパイラが構築される．この結果は参照制限に関する拡張が優先されたものである．

ext2，ext1 の順番で拡張を適用した場合に本来期待する結果としては，最初の limit1 を参照した部分で設定された最小値を返すという動作を優先したもの (すなわち 100 の値をとるという結果) が要求される．しかしながらこの二つの拡張の適用順位では原理に沿わない結果が生じる．すなわちこれら二つの拡張は上記の順番で拡張を適用した場合に衝突を起こすとみなすことができる．同様に ext1'(図.6.8) は ext1(図.6.6) に対する別の実装であるが，この場合でも同じ問題が発生する．

6.1.5 衝突原因の考察

以降では適用順位が変数の最小値設定を優先する (図.6.4 の結果を返す) ような順番 (ext2 のあとに ext1 を適用) であった場合，その特性を反映するような拡張記述をどのようなものにすればよいのかを示す．

優先度の原理を実際に与えられた拡張に対して反映させるためには拡張の適用順位とその呼ばれる状況によって参照する関数を自動的に変化させるための機構および必要な操作が実現されているかを判定する機構が必要となる．ただし各拡張モジュールの設計者は他のモジュールの存在を知ることは基本的に不可能であるため以下の理由から一般のオブジェクト指向言語で採用されているような参照制御の手法を適用することは不可能である．

- 拡張の適用順位によって参照されるべき関数の決定方法が変化するため，拡張モジュール単位で，各関数の参照制限を一意に決定することができない．
- 他の拡張の性質を想定できないため，明示的に他のモジュールへの参照制御を記述させることはできない．

以上のことからそれぞれの拡張モジュールの性質のみに立脚した間接的な参照制御をおこなう必要がある．すなわち拡張記述自体のモジュール性が重要である．

そこで我々は拡張モジュールの設計者が，その責任において同一の対象への再拡張に対して制限を加えるという方法をとる．これは最新の拡張記述は尊重するが，最低限の制約を守らせることによって拡張の影響を継承するという方針を示している

6.1.6 衝突回避のためのアプローチ

では前節ような選択のための条件をどのように設定したらいいのだろうか．我々の採用したコンパイラのモデルは，入力式を解析しながら内部状態を更新していくことで最終的な結果を得るというものであった．すなわちコンパイラの内部状態に対してどのような操作を行っているかによって関数の性質を知ることができる．そこで本節では前節の例題における内部状態へのアクセス関数 rdGEnv に着目し，拡張モジュールの動作定義と，拡張の適用順位との間で対応付けをおこなう．

まず ext2 を定義する段階で設計者が知っている状況は以下のものである．

- ext2 は大域変数の参照関数 (ref_global) に対して，その参照方法をコンパイラの内部状態 GEnv に依存して決定するように拡張する拡張モジュールである．
- 局所変数を参照する関数 (ref_local) を，GEnv に依存する関数 (ref_local:(rdGEnv)) で上書きされる可能性があるということは，局所変数の参照時にグローバル変数に関する参照制限が適用される可能性がある．すなわち ext2 が あとから適用された拡張に優先される可能性があるということになる．
- compile_id が拡張によって上書きされる際，結果的に GEnv が参照されないということは，すなわち上書きされた結果としてスコープ外での有効性が保証されないということになる．

以上の前提から ext2 の設計者は以下の条件を設定することで再拡張に対して制約を与える．

1. **ref_local の拡張時に GEnv の値を用いてはならない**
   - この条件のもとでは, ref_local 内の ref_global は拡張後のものを利用することができないため，優先度の原理が守られる．
2. **compile_id の拡張時に GEnv の値を用いてはならない**
   - この条件のもとでは, ext1’ の compile_id での ref_global は拡張後のものを利用することができないため，優先度の原理が守られる．
3. compile_id は最終的に必ず GEnv の値を用いる

- この条件によってスコープルールの違反を除外することができる．

ここで前者のようにコンパイラの状態の参照に関する禁止項目を与えるような制約を否定的制約と呼び，後者のように必ず行なわれることを期待するような制約を肯定的制約と呼ぶことにする．これらの制約は ext2 によって与えられた拡張に対する再拡張に関して与えられた制約であり適用順位が逆の場合には前述の制約条件は作用しないために ref_global は拡張後のものが参照され優先度の原理が守られる．これらの制約に対して拡張機構は次の方針で参照関数の決定をおこなう

- 基本的には拡張の結果として最終的に与えられた関数を呼び出す
- 参照関数が否定的制約に反する場合
  - 制約に違反する操作を直接参照している場合は拡張の結合に失敗する
  - 制約に違反する操作を間接的に呼び出している場合は制約を満すものまで上書きの過程を逆のぼって探索をおこなう
- 基盤言語まで逆のぼった結果，否定的制約を満さない場合は拡張の結合に失敗する
- 最終的な結果が，肯定的制約を満さない場合は拡張の結合に失敗する

この規則から，結果的に新たなモジュール定義が生成される．以降では ext2 のあとで ext1 を適用する場合を例にとって参照制御がおこなわれる過程を示す．

6.1.7 参照制御の流れ

1. ext1 では，ref_global の置き換えは行われないので ref_global の最終的な結果の候補は ext2 で再定義された ref_global:(rdGEnv) となる．

```
fun ref_global:(rdGEnv) (name) =
let val res = search_name name genv
in
  if res then (ref_glb name) else false
end;
```

2. 最新の定義が優先されるので ext1 で再定義された ref_local が最終結果の候補となる．

3. ext1 で再定義された ref_local 内では，ref_global が参照されている．このとき，ref_global の最終的な結果の候補は ref_global:(rdGEnv) である．

```
fun ref_local (res) =
let val res1 = ref_lcl res
    val res2 = ref_global name
in
  if res2 and (min? res2) then
    if (res1 < res2.num) then res2.num else res1
  else res1
end
```

4. ext2 で与えられた制約を満たすように ref_global の名前で参照される関数を逆のぼると，拡張以前の ref_global:(prv) が対応する．結果として拡張後の ref_local の最終的な結果は ref_local2 となる．

5. ref_global:(rdGEnv) の再定義にはなんら制約がないので ref_global の最終的な結果は ref_global:(rdGEnv) となる．

```
fun ref_local2 (res) =
let val res1 = ref_lcl res
    val res2 = ref_global:(prv) name
in
  if res2 and (min? res2) then
    if (res1 < res2.num) then res2.num else res1
  else res1
```

6. 結果として得られた compile_id は肯定的制約を満たすので拡張の結果は衝突を起していないものとみなされる．

6.1.8 拡張の組み合わせ

前節で示したような参照制御をおこなった結果，我々は表.6.1 の結果を得ることができ，ext2 のあとで ext1' を適用する場合は肯定的制約を満たさないので排除される．ここで各関数名は以下の動作をしめしている．

- compile_id はまず局所変数の環境を検索し，その結果に応じて最新の ref_local もしくは最新の ref_global を呼び出す.
- ref_local は変数に対応するメモリからその値を参照する命令を生成する.
- ref_global は実行時の大域的環境から変数名に対応する値を参照する命令を生成する.
- ref_global:(rdGEnv) は大域変数で参照可能なものを環境 genv で検索し結果が真であれば実行時にその値を参照するような命令を生成する.
- ref_local1 はまず局所的な変数の値を参照するための命令列を生成し，最新の大域変数の参照結果 (ref_global) と比較をおこなう.
- ref_local2 はまず局所的な変数の値を参照するための命令を生成し，大域変数の参照結果 (ref_global) と比較をおこなう.
- compile_id2 はまず与えられた名前をもつ大域変数を参照 (最新の ref_global) し，その結果に応じて局所変数の値 (ref_local) との比較をおこなうか，得られた値そのものを返す．v
- compile_id:(rdGEnv) はまず局所変数の環境を検索し，その結果に応じて ref_local2 もしくは ref_global:(rdGEnv) を呼び出す.
- ref_local:(rdGEnv) はまず局所的な変数の値を参照するための命令列を生成し，大域変数の参照結果 (ref_global:(rdGEnv)) と比較をおこなう.

このようにどの関数がどの内部状態にアクセスするのか，あるいはしないのかという情報は，おおもとの関数が拡張の結果どのように動作するのか (ある条件を満すのか) を決定する上で重要な要素であることが言える．本研究ではこの方法を一般化し Monad によって実現されたコンパイラの内部状態に対する参照に関する制約を与えることによって拡張モジュール同士の衝突を回避するというアプローチをとり，そのための機構を提供する.

表 6.1: 拡張の適用順序と参照される関数

id	compile_id	ref_local	ref_global
ext2			ref_global:(rdGEnv)
ext1		ref_local2	
結合	compile_id	ref_local2	ref_global:(rdGEnv)
ext2			ref_global:(rdGEnv)
ext1'	compile_id2	ref_local	
結合	compile_id2	ref_local	呼ばれない
ext1		ref_local1	
ext2		ref_local:(rdGEnv)	ref_global:(rdGEnv)
結合	compile_id	ref_local:(rdGEnv)	ref_global:(rdGEnv)
ext1'	compile_id2	ref_local	
ext2			ref_global:(rdGEnv)
結合	compile_id2	ref_local	ref_global:(rdGEnv)

6.2 参照制限の実現

本節では各拡張モジュールがどのようにして他の拡張モジュールによる上書き拡張を制限するのかを示す.

まず参照制限の情報が必要になるのは制限拡張がおこなわれる段階であり，その時点は前に適用された拡張がどの内部関数の上書きを制限しているのかという情報だけが必要となる．そこで我々は拡張モジュールのシグニチャを宣言する際に制限を加えるために以下のようにシグニチャ定義を拡張する.

```
signature id_sig2
in
    compute compile_id
    in
      fun compile_id   : (rdGEnv) (rdGEnv);
      fun ref_variable : (rdGEnv) ();
      fun ref_global
    end
end
```

ここで

```
fun ref_variable : (rdGEnv) ();
```

とは `ref_variable` を再定義する時に `rdGEnv` の値を参照してはならないという制約と参照しなくてはならないコンパイラの内部状態は存在しないという制約を与えるもので，ここで指定可能なプリミティブは基盤言語と同じ状態をもつ言語を対象とするならば代表的なものとしては表6.2 に示したものがある.

これでモジュール拡張のための機構に参照制限に関する情報を伝えることが可能となった.

次は動作定義部の各関数がどのプリミティブを呼び出しているかを知る必要がある．そこである関数が参照するプリミティブの集合を返すような関数を図6.9 に示す．ここで *Prims* はメタ宣言部の関数名からその内部で参照されるプリミティブの集合を返す環境である.

表 6.2: 基盤言語における言語定義プリミティブ

alloc-reg	レジスタ確保
free-reg	レジスタ解放
rdEnv	環境取得
inEnv	環境更新
switch	コード生成領域の変更
store-inst	コードの格納
inc-counter	カウンタの更新
store-value	ストアの確保と値の格納
put-io	IO-出力
get-io	IO-入力
search	グローバル環境の検索

6.3 衝突回避の事例

本節では拡張の衝突回避に関する他の適用例を示す.

6.3.1 関数呼び出しのログ

図 6.10 は関数を呼び出すたびにどの関数が呼ばれたのかを出力するための拡張である. (応用として，特殊な宣言をおこなった関数のみログを取るようにすることも容易である). また図 6.11 は同様の動作を記述した別の実装である.

これらが適用された関数呼び出しの定義に対して前章で示した関数呼び出しのキャッシングと組み合せてみると `app_log1` ではキャッシュされた関数であっても呼び出しログが書き出されるのであるが，`app_log2` では出力されない．この場合はどちらの拡張もスコープをもたないが，競合するものではないので適用順序によらずログを書き出される結果が望ましい．ここで有効なのは，`applog_sig` で与える以下の制約である.

まず `app_log1` の定義者が想定するシグネチャとしては以下のものが与えられる．これは必ずログの出力をおこなうという制約と `call_closure` を上書きする場合には出力を行うような操作は呼ばないという制約である.

$$
\begin{array}{lcl}
\mathcal{P} :: \mathit{Expr} \rightarrow \mathit{Bounded} \rightarrow \mathit{Prims} \rightarrow \mathit{Primitives} & & \\
\mathcal{P}[\![\mathit{exp}_1 \ldots \mathit{exp}_n]\!]\ \rho\ \delta & = & \mathcal{P}[\![\mathit{exp}_1]\!] \wedge \ldots \wedge \mathcal{P}[\![\mathit{exp}_n]\!] \\
\mathcal{P}[\![\text{let } \mathit{dcls} \text{ in } \mathit{exp} \text{ end}]\!]\ \rho\ \delta & = & \mathcal{P}[\![\mathit{exp}]\!]\ \rho \wedge \{\mathit{var}_i \mid \mathcal{K}[\![\mathit{exp}_i]\!] = \mathit{Base}\} \\
 & & \qquad \delta \wedge \{\mathit{var}_j \mid \mathcal{K}[\![\mathit{exp}_j]\!] = \mathit{Meta}\} \\
 & & \mathit{dcls} ::= \text{val } \mathit{var}_1{=}\mathit{exp}_1\ ; \ldots ;\ \text{val } \mathit{var}_n{=}\mathit{exp}_n \\
\mathcal{P}[\![\mathit{number}]\!]\ \rho\ \delta & = & \phi \\
\mathcal{P}[\![\mathit{var}]\!]\ \rho\ \delta & = & \phi \quad (\mathit{if}\ \mathit{var} \in \rho) \\
 & & \{(\delta\ \mathit{var})\} \quad (\mathit{otherwise\ if\ var\ is\ bounded in}\ \delta) \\
\mathcal{P}[\![\text{if } \mathit{exp}_1 \text{ then } \mathit{exp}_2 \text{ else } \mathit{exp}_3]\!]\ \rho\ \delta & = & \mathcal{P}[\![\mathit{exp}_1]\!]\ \rho\ \delta \wedge \mathcal{P}[\![\mathit{exp}_2]\!]\ \rho\ \delta \wedge \mathcal{P}[\![\mathit{exp}_3]\!]\ \rho\ \delta \\
\mathcal{P}[\![\mathit{exp}_1 \text{ op } \mathit{exp}_2]\!]\ \rho\ \delta & = & \mathcal{P}[\![\mathit{exp}_1]\!]\ \rho\ \delta \wedge \mathcal{P}[\![\mathit{exp}_2]\!]\ \rho\ \delta
\end{array}
$$

図 6.9: 参照プリミティブ判定関数

```
signature applog_sig1
in
    compile_application
    in
      fun compile_application () (put_io);
      fun call_closure : (put_io) ()
    end
end
```

次に app_log2 の定義者が想定するシグネチャとしては以下のものが与えられる．これは必ずログの出力をおこなうという制約と compile_application を上書きする場合に出力を行うような操作は呼ばないという制約である．

```
signature applog_sig2
in
    compile_application
    in
      fun compile_application : (put_io) (put_io);
      fun call_closure : () ()
```

```
extension app_log1 : applog_sig1 (cls_sig)
in
    compute compile_application (exp as app:(rator,rands))
    where meta
      fun write (num)      -> write_num num;
    in
        fun compile_application (exp as app:(rator,rands)) =
        let val cls = compile rator;
            val lst = compile_rands rands
        in
          write cls.code;
          call_closure cls lst
        end;
    end
end;
```

図 6.10: 拡張モジュール (関数呼び出しのログ 1)

```
extension app_log2 : applog_sig2 (cls_sig)
in
    compute compile_application (exp as app:(rator,rands))
    where meta
      fun write (num)        -> write_num num;
      fun call_cls (cls,lst) -> call_closure cls lst
    in
        fun call_closure (cls,lst) =
        begin
          write code;
          call_cls cls lst
        end
    end
end;
```


図 6.11: 拡張モジュール (関数呼び出しのログ 2)

```
    end
end
```

これらの制約によって結果的に順序にかかわらずログが生成される．

6.4 依存関係に関する考察

我々が設定したモジュール間の依存関係を示すものは唯一共有メタ宣言部を参照する目的でのモジュールの輸入であった．そこで拡張の記述をおこなうものが，拡張の結果として共有メタ宣言部を参照する必要がなくなると判断した場合には必ずそのことを明示する．すなわち件のモジュールの輸入をおこなわないことを宣言することで依存関係の変更を示すわけである．ただしこれは, 弱い意味での宣言であって基盤言語の定義をのぞくそれ以前の拡張記述においてその必要性が宣言されている場合にはその除外は有効ではない．

これを実現するために拡張モジュールの輸入モジュールの宣言のもつ意味を変更する必要がある．そこで以下の規則を適用する．

参照モジュールの決定規則

> 基盤言語のモジュールが参照しているモジュールの集合を S，同一レイヤで参照可能なすべてのモジュールの集合を あ $M(S \subset M)$ とする．そのとき拡張モジュール E_1 において輸入を宣言したモジュールの集合を $S'(S' \subset M)$ ，さらに拡張モジュール E_2 を適用した時に輸入を宣言したモジュールの集合を $S''(S'' \subset M)$ とする．このとき実際に参照するモジュールの集合は $S' \cup S''$ である．

このように参照関係の変化を各拡張を適用する段階で明確にすることで常に各モジュール同士の関連性を追跡することができるため，拡張の組み合わせによって製作者が想定していなかったモジュール間の依存関係が存在している場合や，想定した依存関係が存在しないために発生しうるトラブルにかんしてエンドユーザーに警告を与えることができる．

第 7章

まとめ

本研究では以下のアプローチをとり安全にモジュラーな言語拡張が可能なシステムの構築をおこなった.

- コンパイラの特に実行コード生成の部分について，Monad を利用したコンパイラのモジュール化のための枠組みを示した.
- コンパイラ記述言語による高級な言語部品の記述方法を示した.
- 低レベルな記述をメタレベルとして分離することで，部品の共有化を可能にした.
- 各定義の上書きによる拡張のための仕組みを示した.
- 拡張時の内部状態へのアクセス制限によって再拡張の制限をおこなうことによる衝突回避の方法を提案した.

7.1 考察

本研究が想定するシステムではまず拡張モジュール間の優先度を言語部品のエンドユーザーが設定するための優先度決定のための前提として適用順位がよりおそい拡張モジュールを優先することにした．すなわちあとから適用された拡張の性質が強くなるという前提である．次にシステム側である程度拡張に関する制限を加ることで安全性を強化した.

- 実装する機能の抽象度に応じた階層構造を設計する．このことによってモジュールのグループ化をおこなうことで部分的にモジュール間の参照関係を固定化することがで

きる．

- 拡張にともなう参照関係の変化を限定的なものにするために拡張が可能な階層を選別する．
- コンパイル過程の共有化によって拡張の影響を正しく伝播させる．モジュールの変更はそれを利用するモジュールにおいても自然に反映される．

これらは拡張性と安全性とのトレードオフに基づくシステム側の選択であり拡張可能な部品を高位の階層に限定することで混乱を防ぐ効果がある．

さらに拡張モジュールの設計者側で設定する必要のある拡張に関する情報を以下のように設定した．

- モジュール単位で再拡張をおこなう際，より優先度の高い拡張モジュールに対してコンパイラの状態へのアクセスに関する明示的な制約を与える．
- 拡張モジュールを適用することでモジュール間の依存関係にどのような変化が起きるのかの明示的な記述をおこなう．

これらは拡張モジュールの設計者が与えるシステムに対する衝突の可能性に関するヒントである．さらに言うならばこれらの情報自身が衝突の概念の定義そのものであるとも言える．すなわち何が拡張間の衝突であるかは拡張モジュールの設計者の責任であるという立場をとるものである．

まとめると，システムで提供された拡張の枠組で定義された拡張モジュールのなかで以下の条件にあてはまるものを言語拡張の衝突とよび，言語を再構築する時点でこれらの衝突をおこさないような選択肢を選ぶか，あるいは選択肢が提供されていな場合は警告をおこなうことで言語拡張の衝突回避を実現した．

- 拡張モジュールの適用順位によってユーザーが想定している拡張間の優先度と，拡張モジュールの設計者が与えた制限に基づく優先度が一致しない場合．
- 依存関係の変化に対応するような拡張部品が提供されていない場合．

本アプローチはコンパイラを構成する部品を実現している各内部関数がどの Monadic Primitive を呼んいるかによってその性質が表現されるような拡張に有効な枠組である．ただ

し制限の方法が低レベルで間接的であるため有効な制限を与えることや，Monadic Primitive で各内部関数の性質を決定することの困難さといった問題がある．そのため同一の言語の意味単位に対する拡張がより自由に組み合わせられるような設計のための方法論の存在が求められる．

7.2 関連研究

7.2.1 Domain-specific Languages

特定の領域に特化したプログラミング言語を構築するための手法には大別して抽象度の高い形式的な仕様から自動的にプログラミング言語を生成する手法と，用意されたいくつかの部品を組み合わせて要求を満すようなプログラミング言語を構築するアプローチがある．

前者の例としてはコンパイラを言語仕様 (インタープリタ定義) から部分計算をもちいて自動生成する手法がよく知られている．この手法では言語仕様を与えるための記法が広く知られる表示的意味論に基づくため非常に一般的であり，また一度部分計算器を与えてしまえばしかし実際のコンパイラがおこなっている低レベルな操作を記述することが困難である点や言語仕様自体のモジュール性が低いために仕様の再利用性が低いという点，また部分計算器自身の提供が困難であるなどいくつかの問題点が存在する．

後者の例としては Domain-specific Languages (DSL) を言語機能に関するいくつかの選択肢から構築する手法が示されている [20]．この方法はあらかじめ用意された言語機能の組み合わせでしかなく，また新たな機能の選択肢を定義するための枠組が与えられておらず言語構築の自由度が高いとはいえない．

また Duggan [4] は言語処理系の部品を Java のクラスとして定義し，それらを結合することで様々な DSL を提供する試みをおこなっている．Duggan はインタープリタにおける環境やステートといったモジュール化を困難にする概念の実装のためのコード片の扱いを monad transformer を利用して簡潔に実現している．彼等は monad を *mixin* を用いて Java のクラスに導入することに成功している．ただし彼等の研究は現状では本稿で指摘しているような差分的な拡張に関する問題点が考慮されていない．

WansBrough 等は DSL を差分的に拡張するための枠組を提案している [27]．この枠組は Modular monadic action semantics と呼ばれる action semantics [17] と modular monadic semantics との融合である．彼等は action を定義，拡張するための手段として monadic

semantics を利用している．

Duggan [4] は，Java のクラスを部品のように組合せてインタープリタを構築する方法を提案した．このアプローチでは monad-transformer を *mixin* を利用して Java に導入することでインタープリタのモジュール化を実現した．我々のアプローチでは，コンパイラの定義をアクションの組み合わせではなく，より直接的なコードの生成過程の記述として表現する．その際 monadic semantics はコンパイラ記述言語のメタレベルとして隠蔽される．

7.2.2 コンパイル時自己反映計算

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 の目的は SML において Lisp のマクロに相当する機能を実現することである．すなわちプログラムの一部をユーザーが対応する構文木を表わす操作可能なデータ構造で表現しプログラム中の変更を可能にする．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でないので，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．

同様に OpenC++[2] は C++において構文木の操作およびその反映をおこなうことを可能にしたシステムであるが，差分的な拡張に関しては一般的な OOP の枠組にとどまる．

Java の拡張可能なプリプロセッサ EPP[28] は, ユーザーが拡張可能な, 柔軟なソースコード変換器を提供したフレームワークである．拡張機構として mixin を採用することで差分的な拡張を実現している．コード変換の正否によって言語拡張の結合の正否を決定することで言語拡張のモジュール化における安全な結合を達成しているが，本研究のように関数の参照関係によらないような依存関係による問題を判定することはできない．

本研究のアプローチはコンパイル時の自己反映計算をおこなうシステムで，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であると考えることが可能である．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は困難であり，ユーザーにとっての負担が大きくなる．そこで我々は容易にするような階層化と抽象化をおこなうことで負担の軽減を図っている．

また，より広範な問題領域に関する適用可能性を考えると，コード生成レベルにまでふみこんだ拡張可能性は重要な要素である．例えば実行効率の低下が致命的になるような計算を記述するような言語への要求や，並列計算機やゲームマシンのような特殊なマシンアーキテクチャに適用可能な言語拡張の枠組が今後必要とされると考えられ，複数のターゲットマシンへの適応に関しては重要な今後の課題である．

7.2.3 属性文法の利用

また拡張属性文法を用いたメタレベルの抽象化方式を与え, その形式的な定義を与えた研究として, [24] がある．これは言語システムに限らない, より一般的な記述系に対するメタレベルを対象としたフレームワークであり, 広範な適用が期待できる．

7.2.4 安全な拡張記述のための方法論

本研究のアプローチでは, 拡張の記述者が再拡張に関する制限を直接指定することで, 衝突の回避を実現する．すなわち拡張機構を制御する責任がモジュール設計者に属するという方針を採用している．そのためシステム側は, 基本的にモジュール設計者の意向に従って拡張モジュールを利用したプログラミングを支援するという立場に徹する．

そのため, モジュールの定義をおこなう立場のユーザーに対しては, より再拡張の容易な優れた方法論の提案が必要であると言える．このような結合可能な拡張に関する方法論的なアプローチとしては, Mulet らによるメタオブジェクトの合成に関する話題 [18] で述べられている．

7.3 今後の課題

今後の研究課題としては以下のものが挙げられる.

- コンパイラ記述言語の限界として変換過程で一つの関数定義がその引数のレベルによって複数の結果をとるような変換を許さないアルゴリズムを採用したために記述力が不足する可能性がある
- 拡張のためのインターフェイスを設計するための方法論を与える
- 本フレームワークを，コンパイラの他の局面にも適用することでシステム全体のモジュール化をおこない，拡張可能なシステムを設計する
  - トップレベルの拡張
  - 構文木の操作
  - 実行時システムの変更

謝辞

本研究を行なうに当たり, 終始御指導を賜わった渡部 卓雄助教授に深謝致します.

最後に, 本論文をまとめるに当たって御協力いただいた二木・渡部研究室の諸兄に厚く御礼申し上げます.

参考文献

[1] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation to implement reflective languages (extended abstract). In *OOPSLA '93 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1993.

[2] S. Chiba. A metaobject protocol for c++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pages 285–299, 1995.

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

[4] D. Duggan. A mixin-based,semantics-based approach to reusing domain-specific programming languages. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2000)*, number 1850 in Lecture Notes in Computer Science, pages 179–200, 2000.

[5] D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[6] A. Filinski. Representing monads. In *In Conference Record of POPL '94 : 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457. ACM Press, 1994.

[7] 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, pages 24–35, November 1992.

[8] 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*, pages 48–58, November 1992.

[9] S. Liang and P. Hudak. Modular denotational sematics for compiler construction. In H. R. Nielson, editor, *ESOP '96: 6th Europ Symposium on Programming*, pages 219–234, 1996.

[10] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the ACM Conference on the Principles of Programming Languages (POPL '95)*. ACM New York, NY, 1995.

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

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

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

[14] H. Masuhara, S. Matsuoka, T. Watanabe, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurent reflective language useing partial evaluation. In *in Proc of ACM Conf. on OOPSLA*, pages 300–315, 1995.

[15] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[16] P. Mosses. A basic abstract semantic algebra. In *Lecture Notes in Computer Science 173*. Springer-Verlag, New York, NY.

[17] Peter D. Mosses. Theory and practice of action semantics. In *Proceedings of 21st International Symposium on Mathematical Foundations of Computer Science*, volume 1113 of *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1996.

[18] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *OOPSLA 1995*, pages 316–330.

[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 Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 36–47, November 1992.

[20] P. Pfahler and U. Kastens. Language design and implementation by selection. In *Proceedings of the First ACM SIGPLAN Workshop on Domain-Specific Languages*, pages 97–108, 1997.

[21] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, 1984.

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

[23] J. M. Sobel and D. P. Friedman. An introduction to reflection-oriented programming. In *Reflection 96*.

[24] Akira Tanaka and Takuo Watanabe. An extensible LR parser generator — a case study of composable metalevel extensions —. In *Proceedings of International Workshop on Principles of Software Evolution*, pages 84–88, July 1999.

[25] P. L. Wadler. Monads and functional programming. In M. Broy, editor, *Marktoberdorf International Summer School on Program Design Calculi*. Springer-Verlag, New York, NY, 1993.

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

[27] K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Proceedings of the Conference on Domain-Specific Languages*, 1997.

[28] 一杉裕志. 高いモジュラリティと拡張性を持つ構文解析器. **情報処理学会論文誌*:*プログラミング**, 39(SIG 1(PRO1)), 1998.

[29] 渡部卓雄. 「リフレクション」(チュートリアル). In **コンピュータソフトウェア***, Vol. 11, No. 3*, pages 5–14. 日本ソフトウェア科学会, 1994.

Appendix

A 関数適用の定義

```
module application : app_sig (cls_sig)
in
    compute compile_application (exp as app:(rator,rands))
    where meta
        val regs = used_regs ();

        fun build_frame (x,y)   -> setup_frame:(regs,x,y);
        fun release_frame (x,y) -> reset_frame(x,y);
        fun call (x)            -> store_inst call:(x);
        fun compile_rands (lst) -> compile_vect vect:(lst)
    in
        fun compile_application (exp as app:(rator,rands)) =
        let val cls = compile rator;
            val lst = compile_rands rands
        in
          call_closure cls lst
        end;

        fun call_closure (rator as cls:(code,parm),lst) =
        begin
          build_frame parm lst;
          call code;
```

```
          release_frame (x,lst)
        end
    end
end
```

B　スタックフレームの定義

```
module frame : frm_sig (lst_sig)
meta
    val sfrm:(ret,old,args) -> push_list [ret,old,args];
    val sfrm (x) -> (ret,old,args)@{ld:(ret,fptr,1),
                                     ld:(old,fptr,2),
                                     ld:(args,fptr,3)}
in
    compute setup_frame (args,lst) =
    where meta
      fun old_link () -> store_inst newframe:()
    in
      setup_frame (args,lst) = sfrm:(0,old_link (),args::lst);
    end;

    compute release_frame (reg,lst) =
    where meta
      fun get_result () -> pop_val y
    in
      let val res = get_result ();
      in
        reload_regs lst;
        res
      end
```

```
    end
end
```

C　クロージャの定義

```
module closure : cls_sig ()
meta
    val cls:(code,parm) -> make_cell code parm;
    val cls (x) -> (y,z)@ld:(y,x,0),ld:(z,x,1);
in
    compute compile_closure (exp as cls:(vars,body))
    where meta
        val label = alloc_symbol (get_newlabel ());
        val depth = get-depth ();
        val env   = rdEnv;

        fun switch (c) -> switch_seg c label;
        fun save_regs () -> free_used_regs ();
        fun restore (lst) -> restore_regs (lst);
        fun get_closure (x,y) -> cls:(x,y);
        fun setup_frame (frm:(a,b,c)) -> frm:(mfspt:(a),b,c);
        fun reset_frame (frm:(a,b,c)) -> addi:(sptr,depth + 3)
                                        ≫_ frm:(a,ld:(x,b,0),c);
        fun ret (val) -> push val;
        fun set_avct (val) -> push val;
        fun extend_env (c,vars) -> inEnv (extend vars env) c
    in
        fun compile_closure (exp as cls:(vars,body)) =
        let val regs = save_regs ()
in
```

```
  begin
    let val code = switch (compile_body exp)
    in
      restore regs;
      let val frm:(ret,old,args) = get_current_frame ()
      in
        get_closure (code,args)
      end
    end
end;

fun compile_body (exp as cls:(vars,body)) =
let val frm:(ret,old,args) = get_current_frame ()
in
  let val frm:(ret,old,args) = setup_frame (frm)
  in
    begin
      expand-args args;
      return (extend_env (compile body) vars)
    end
  end
end;

fun return (val) =
let val frm:(ret,old,args) = get_current_frame ()
in
  let val frm:(ret,old,args) = reset_frame (frm)
  in
    ret val;
  end
```

```
        end;

        fun expand-args (lst) = callee_frame (depth+1,lst);

        fun callee_frame (n,lst) =
        if (n == []) then []
        else
          begin
            set_avct (hd lst);
            callee_frame (n - 1) (tl lst)
          end
        end
    end
end
```

本研究に関する発表論文

[1] 佐伯 豊, 渡部 卓雄, “自己反映的な言語のモジュール結合による実装とその効率化”, 第 24 回情報処理学会プログラミング研究会,1999,6.

[2] 佐伯 豊, 渡部 卓雄, “自己反映的な言語における言語拡張同士の安全な結合について”, 第 28 回情報処理学会プログラミング研究会,2000,3.

[3] 佐伯 豊, 渡部 卓雄, “自己反映的な言語システムのモジュール化”, 日本ソフトウェア科学会 第 16 回全国大会論文集, pp.265-268,1999,9.

[4] 佐伯 豊, 渡部 卓雄, “モジュール結合による自己反映的な言語拡張について”, 日本ソフトウェア科学会 第 14 回全国大会論文集, pp.209-212,1997,9.

[5] 佐伯 豊, 渡部 卓雄, “自己反映計算における再利用可能なメタレベルモジュールの設計”, 電子情報通信学会 ソフトウェアサイエンス研究会 技術研究報告, Vol.98,No.230,pp.33-40,SS98-20,1998,7.

[6] 佐伯 豊, 渡部 卓雄, “再利用可能な言語拡張部品の設計”, 日本ソフトウェア科学会論文誌 「コンピュータソフトウェア」, 投稿中.

[7] Yutaka Saeki , Takuo Watanabe, “Towards a Modular Construction Method of Extensible Compilers”, in Proceedings of International Workshop on Principles of Software Evolution, pp.1-18, 1999.7.