修 士 論 文

コンパイル時自己反映計算を用いた
カスタマイズ可能な言語処理系に関する研究

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

山田 聖

2000 年 2 月 15 日

要 旨

Scheme のような動的な型づけに基づく言語は、実行時に型検査が行われるため効率が悪く、型誤りの検出が難しいという問題がある。このような言語のための型推論機構は、コンパイル時に型推論を行い静的な型検査と実行時コードの効率化を目指しているが、言語の動的な側面のために十分な型情報を得るのは難しい。

一方、動的な型づけに基づく言語を用いる場合であっても、適切な型の値に対してのみ手続きが適用されるように注意するといったように、型を考慮しつつ記述することになる。このような、プログラマが持つ型に関する考えを明示するためのインターフェイスを用意すれば、それを用いることにより、コンパイル時により多くの型情報を得ることができると考えられる。静的な型づけに基づく言語における型宣言の形式を導入することも考えられるが、プログラマが持つ型に関する考えは、プログラムの構文的な構造とは異なる論理的な構造に基づいているために、この、型宣言の形で表現することが難しいことがある。

以上で述べたことから、本研究では動的な型づけに基づく言語のための型推論機構が型を解析するためのヒントとなる情報を、プログラマが明示できるようにするための機構を提案する。具体的には、自己反映計算機構を用いて、Scheme のための型推論機構の型を解析するフェーズをカスタマイズするインターフェイスを設計する。プログラマは、このインターフェイスを使用して、型推論機構の推論規則を拡張することができるようになる。

更に、この設計をもとに実装を行い、処理系の有効性の検証を行う。

目 次

第 1 章

序論

1.1 背景

Scheme[3] は動的な型づけに基づく言語である。このような言語では、実行時の値に対して型が決まる。変数は任意の型の値を束縛することができ、型宣言が不要であることから、比較的自由にプログラムを記述することができ、シンボリックな計算やプロトタイピングといった用途に適している。

C 言語や Standard ML のような静的に型づけされる言語では、全ての型検査を静的に行うことにより、実行時プログラムを型安全で効率的なものにするのに対し、Scheme や Smalltalk のような動的に型づけされる言語では、本来の計算に加えて動的な型検査が実行時に行われるため、効率が悪く、また、型に関する不適切な操作の検出が難しい。これらの問題点を改善するために、Soft Typing[6] や Type Recovery[5] といった、動的に型づけされる言語のための型推論機構が提案されている。これらは、定数の変数への束縛や原始関数の適用を元に、変数に束縛される可能性のある値の集合としての型を推論するが、言語の持つ動的な側面のために保守的な結果しか得られない。記述性を損なうこと無く、実行時コードの効率化, 型誤りの早期検出を実現するためには、動的に型づけされる言語のための型推論機構が、コンパイル時により多くの型情報を得るための手法が必要となる。

1.2 目的

本研究では、動的な型づけに基づく言語のための型推論機構が、型を解析するためのヒントとなる情報を、プログラマが明示できるようにするための機構を提案する。具体的には、自己反映計算機構を用いて、Scheme のための型推論機構の型を解析するフェーズをカスタマイズするためのインターフェイスを設計する。更に、この設計をもとに実装を行い、処理系の有効性の検証を行う。

プログラマは、このインターフェイスを使用することにより、プログラムの記述性を損なうこと無く、コンパイル時により多くの型情報を得ることができるように、型推論機構をカスタマイズすることができる。

1.3 構成

本稿の構成は、以下の通りである。

第 1 章 本章。

第 2 章 既存の言語処理系を比較し、問題点を明らかにする。

第 3 章 処理系に求められている機能を示し、それを実現するための処理系の構成について述べる。

第 4 章 動的型づけに基づく言語のための型推論機構の意味を示し、それに対するカスタマイズのためのインターフェイスを定義する。

第 5 章 カスタマイズの記述例を示す。

第 6 章 まとめと、今後の課題について述べる。

第 2 章

言語処理系

本章では、本研究の要となる概念である型と自己反映計算について簡単に解説し、それぞれについての代表的な言語, および言語処理系を比較する。更に、既存の言語における型に関する問題点を明らかにする。

2.1 型

2.1.1 型とは

ある式の取り得る値を規定したり、その値に対する操作の適用性を示すものが**型**である。大抵の言語処理系では、与えられたプログラムが、例えば真偽値どうしの加算のような不適切な操作が行われることが無いように、**型検査**が行われる。このように、型に関して不適切な操作が行われることの無いプログラムは**型安全**であると言われる。

言語は、型検査を行うタイミングにより大きく二種類に分類できる。一方は、プログラムの実行中に本来の計算を行うとともに型検査も行うものであり、Smalltalk や Lisp 系の言語がこれにあたる。このような言語は**動的な型づけ**に基づく言語と呼ばれ、型が変数ではなく値と関連づけられている。もう一方は、型検査をコンパイル時、つまり実行時以前に行うものであり、Haskell, Standard ML, C 系の言語等がこれに含まれる。このような言語を**静的な型づけ**に基づく言語と呼ぶ。

静的に型づけされる言語のなかには、プログラムを構成する式やデータ型等に対して

明示的に型宣言を行わなくても、構文や文脈から系統的な推論によって型情報を得るものがある。このような推論を**型推論**と言い、型推論を行う言語として Standard ML がある。この言語では、変数や手続きの引数に静的に束縛されている値や、予め引数や処理の結果の型がわかっている原始的手続きの適用などから推測できる情報をもとに、推論規則を繰り返し適用して行くことにより、式の型を導く。

2.1.2 既存の言語処理系

Scheme

Scheme は、レキシカルスコープを持つ一級手続きを扱うことができる Lisp の方言の一つであり、比較的小規模な関数型言語である。

この言語では、型は式ではなく値と関連づけられており、型誤りの検出は、実行時の原始手続きの適用時に引数の値の型を検査することによって実現されている。このように、本来の計算に加え型検査が実行時に行われることが、実行時の効率を悪くする原因となっており、また実行時に問題が発生するまで型誤りが検出されないことから、型に関する不適切な操作を早期に発見することができないといった問題点がある。

しかし一方では、式と型が関連づけられないので、複雑なデータ構造を容易に取り扱うことができるといった、型にとらわれることのない自由度の高い記述を行うことができる。このため、Scheme はシンボリックな計算やプロトタイピングの用途に向いている言語であると言える。

Common Lisp

Common Lisp[2] は、Scheme や Standard ML のルーツとなった Lisp の方言の一つあり、ISO により標準化されている。この言語は、オブジェクト指向やモジュール等、様々な機能が組み込まれている比較的規模の大きい関数型言語である。この言語の型に関する特徴は Scheme と同様であるが、宣言のための構文 `declare` と型指定子 `type`, `ftype`, `the` 等を持ち、これらを用いて変数や式の値の型を宣言することができる。

Common Lisp における型宣言の目的は、型宣言された変数への値の束縛時や、型宣言された手続きへの値の適用時に、その値の型検査を行うことによる、型に関する不適切

な操作の早期検出を行うことである。また、Common Lisp のコンパイラが、この型情報に基づいて、効率的な実行時コードを生成できるようにすることも目的の一つである。

Standard ML

Standard ML は Scheme と同じくレキシカルスコープを持つ一級手続きを扱うことができ、更にパターンマッチングを使用することのできる関数型言語である。

Scheme と大きく異なる点の一つは、静的な型づけに基づく言語であることである。コンパイル時に全ての式の型検査が行われ、型安全な実行プログラムが作成される。また、実行時に型検査が行われないことに加え、コンパイル時に型情報に基づいた最適化を行うことができるため、プログラムの実行効率が良い。

もう一つの相違点は、Standard ML が型推論機構を持つことである。プログラマが明示的に型宣言を行わずとも、型推論機構によって式の構造から体系的な推論に基づき型情報を導き出すことができるため、静的型づけに基づく言語に見られる型宣言の煩わしさが取り除かれる。

しかし、Srandard ML でプログラムを記述する場合、常に型を考慮せねばならず、プログラムの作成に先立って式やデータ構造の型を設計しておく必要がある。

Soft Scheme

Soft Scheme は、Scheme に型推論機構を組み込んだ処理系である。言語を変更すること無く型推論機構を導入することにより、Scheme の利点を損なうこと無く型検査をコンパイル時に行い、Scheme の問題点の原因である実行時の型検査の頻度を少なくしている。Scheme の動的な型づけに基づく言語たる側面から、静的に型を推論することが困難である場合もあり、そのような式に対する実行時のコードは、本来の Scheme 処理系での場合と同様に、型検査が行われることになる。

動的型づけに基づく言語のための型推論機構として、Set-based Analysis[1] や Type Recovery[5] といった方法も提案されているが、これらも同様の問題点を持つ。

```
(lambda (exp)
  (if (pair?  exp)
      (case (car exp)
        ((define) ⟨define 式の場合の処理⟩*1)
        ((let)  ⟨let 式の場合の処理⟩*2)
        ⋮
```

図 2.1: scheme の構文を処理するプログラムの一部

2.1.3 問題点

図 2.1 は、Scheme の構文に従う式に対し、その構文に応じて処理を振り分ける。この式に対し、Soft Scheme によって型推論を行うと、変数 exp は (*1), (*2) のどちらの箇所でも型 (symbol . (not nil)) を持つという結果が得られる。ここで、(not nil) は nil でない任意値を示す型である。この結果は、制御が (*1), (*2) へ至る間に行われる、if 式と cond 式による変数 exp の値に対する型の検査から推論される。しかし、式が常に Scheme の構文に従うことが保証されている場合には、define, let のそれぞれの構文が、

(define ⟨ 変数 ⟩ ⟨ 式 ⟩)
(let ((⟨ 変数 $_1$ ⟩ ⟨ 式 $_1$ ⟩)...) ⟨ 本体 $_1$ ⟩ ...)

であることから、変数 exp の型は、(*1) の箇所では、

```
(symbol symbol (not nil))
```
(2.1)

であり、また、(*2) の箇所では、

```
(symbol (list (symbol (not nil))) .  (not nil))
```
(2.2)

であってほしい。しかし、Soft Scheme には、こういった「exp の値は Scheme の構文に従った構造である」という情報を処理系に伝える為のインターフェイスが用意されていないため、期待される結果を得ることができない。

```
datatype symbol = sym of string;
datatype sexp = symbol
              | set of symbol * sexp
              | let of (symbol * sexp) list * sexp list
              ⋮
```

図 2.2: Scheme の構文を取り扱うデータ構造の Standard ML での定義例の一部

この問題点を改善するためには、処理系に対して型情報を教えるためのインターフェイスを用意すれば良い。単純には、静的型づけに基づく言語にみられるように型宣言を行うための構文を用意すれば良く、実際に Common Lisp にはそのような構文が用意されている。

ここで問題になるのが、この型宣言の記述法である。Scheme の構文の例では、変数 `exp` に対して、「リストの先頭要素が `define` というシンボルである場合、その型は (2.1)、リストの先頭要素が `let` というシンボルである場合、その型は (2.2)、...」といった形で型宣言を行いたくなるが、Common Lisp ではこの形の型宣言は難しい。また、「頭文字が `i` である変数は、整数型である」といった形の型宣言も、それを行うのは困難である。

一方、standard ml でこのようなデータ構造を取り扱おうとした場合を考えると、プログラム本体を記述する前にまず、図 2.2 に示すようなデータ構造を表現する型を設計しなければならない。このため、プロトタイピングのような実験的なプログラムを素早く作成するといった用途に Standard ML を用いるのは適切であるとは言えない。

2.2 自己反映計算

2.2.1 自己反映計算とは

計算機システムが、それ自身の構成や計算過程といったモデルを持ち、計算機システムとそのモデルが因果的に結合されている時、そのモデルに関して行う計算を**自己反映計算**という [8]。自己反映計算の能力を持つ計算機システムは、自分自身のモデルを参照

し (レイフィケイション)、必要に応じてモデルを変更する (狭義の自己反映計算) ことによって、計算機システム自身を変更することができるといった特徴をもつ。

自己反映計算の能力を持つ言語では、一般的に次のような利点がある。

拡張性 言語処理系の構成や動作といった、従来の処理系では隠蔽されている部分を陽に取り扱うことができる。このことにより、ある言語が提供する機能がその利用者の要求を満たさない場合に、処理系を別のものに変更するこなく、その言語の範囲内で処理系をカスタマイズすることができるようになる。

モジュール性 本来の計算 (ベースレベル) とその計算に関する制御やカスタマイズ (メタレベル) を分離する、新たなモジュール化法を提供する。ベースレベルとメタレベルを明確に分離することにより、それぞれのモジュールの独立性が増すため、それらの再利用が容易になる。

一般的に、メタレベルはベースレベルを解釈実行するインタプリタとして抽象化されるため、自己反映計算は効率が悪い。これを改善するための方法の 1 つが**コンパイル時自己反映計算**であり、コンパイル時に自己反映計算を行ってしまうことにより、実行時の効率の低下を防ぐというものである。

コンパイル時自己反映計算機構を持つ処理系として、open C++や CRML[4] などが提案されている。これらはどちらも抽象構文木を扱うフェーズをカスタマイズの対象としてる。

2.2.2 既存の言語処理系におけるコンパイル時自己反映計算

CRML

Crml は、Standard ML のためのコンパイル時自己反映計算機構である。この書理系は Standard ML のプリプロセッサとして構成され、入力されるプログラムは、トップレベルに宣言された、オブジェクトプログラムと呼ばれるベースレベルのプログラムと、メタプログラムから構成される。これらは、どちらも拡張された Standard ML の構文に従う。コンパイル時にはメタレベルプログラムが実行され、オブジェクトプログラムの作成, 変換が行われ、その結果、メタプログラムを含まない Standard ML のためのプログ

ラムが出力される。

メタレベルでは、lisp 系の言語における s-式のようにオブジェクトプログラムをデータとして取り扱うことができ、その記述自体が Standard ML の拡張構文に従うため型検査が行われる点がこの処理系の特徴である。

OpenC++

OpenC++は、C++のためのプリプロセッサとして構成されたコンパイル時自己反映計算機構である [7]。

C++は、構文や意味が複雑であるため、従来のマクロのトークンの置き換えによるプログラムの変換が役に立たない場合が多い。これに対し、OpenC++ではプログラムの意味構造に基づく変換を行う。意味構造とは、具体的にはコンパイラが生成するパース木に対応する。変換が構文木ではなく意味構造に基づいていることにより、ベースレベルのクラスの中に、お互いに関連があるものの記述が分散されている要素を 1 つの要素として操作することができる。

OpenC++では、従来のクラスをベースレベルとし、それに対応するメタレベルとして、ベースレベルのクラスの構文の意味に応じて呼び出されるメソッド群により構成されるメタクラスが導入される。メタレベルがクラスとして記述できることから、継承による再利用性が向上し、メタオブジェクトの記述が容易に行える。

マクロ

マクロは特定の言語ではなく、C 言語のプリプロセッサや Scheme 等、様々な言語処理系で広く使用される技法である。

プログラミング言語のためのマクロは一般的に、パターンと対応する置換テキストから構成されるマクロ定義 (これもマクロと呼ぶ) に基づき、あるプログラムテキスト中のパターンにマッチする部分を対応する置換テキストで置き換えるといった形で動作する。

この時、処理の対象となるプログラムテキストをベースレベル、マクロを処理するマクロプロセッサをメタレベルとする関係を考えると、(1) マクロプロセッサ内に存在する、プログラムテキストに対応する構文木やトークン列がプログラムテキスト自身のモデル

に対応し、(2) モデルに対する置換操作は、ベースレベルであるプログラムテキストの変換につながることから、これらの間には因果的結合が成り立つ。

従って、プログラムテキスト中の、マクロ定義に基づくマクロ展開の処理を、自己反映計算ととらえることができる。更に、構文木やトークン列から成るメタレベルは、マクロプロセッサの処理の終了とともに消滅してしまうことから、このフェーズをコンパイルとみなすことにより、マクロ展開はコンパイル時自己反映計算であると考えることができる。

第 3 章

本研究のアプローチ

第 2.1.3 節では、既存の言語処理系において、型を思い通りに扱うことができない例を示した。それを受けて、本章では言語に期待されている能力を洗い出し、それを満たすための処理系の構成を示す。

3.1 言語の満たすべき条件

3.1.1 1つの言語で対処できること

ソフトウエア開発の初期段階では、要求の分析や実現可能性の評価を行う為の、試作ソフトウエアを単期間のうちに作成する必要がある。このような状況では、Scheme のような動的に型づけされる言語の型を深く考慮せずに複雑なデータ構造や式を記述できるという特徴が役に立つ (図 3.1 参照)。試作ソフトウエア作成過程で試行錯誤を繰り返し、徐々にプログラムコードが安定すると、それをもとに実用的なソフトウエアを作成する

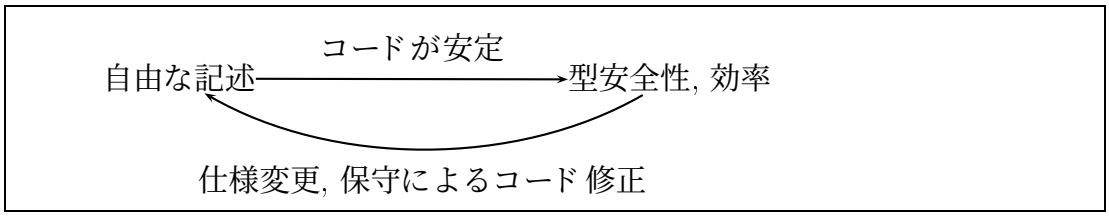

図 3.1: ソフトウエア開発の過程と処理系への要求の関係

ことになる。この、実用的なソフトウエアの作成には、型に関する不適切な操作を取り除いたり実行時の効率を向上させるために、静的型づけに基づく言語を用いたい。しかし、動的に型づけされる言語と静的に型づけされる言語はその構文が異なるため、試作ソフトウエアをもとに新たな言語のためのソフトウエアを書き直さなければならず、この作業は手間がかかり、また、誤りが混入する恐れがある。新たな言語へ機械的に変換することも考えられるが、変換前と後のコード間の構文上, 意味上の対応が明確ではないことが多く、変換されたコードをもとに開発を続けることを難しくする。これらから、試作ソフトウエアの作成と実用的なソフトウエアの作成は、共通の言語で行いたいという要求が生ずる。

では、言語はどのような要求を満たすべきかを具体的に考えると次のようになる。

- プロトタイプを容易に作成できる

  試作ソフトウエアを単期間のうちに開発し、要求の分析、実現可能性などの評価を行う。

  型を深く考慮すること無く、複雑なデータ構造やアルゴリズムを自由に記述したい。

- コードが固まった部分に対し、型情報を与えるという作業をインクリメンタルに行える

  静的な型検査により、コンパイル時に型に関する不適切な操作を検出する。

  型情報をもとに、実行時コードの効率化を行いたい。

- 仕様変更などの理由からコードを修正する必要がある場合、型情報の記述を容易に取り除くことができる

3.1.2 型に関する情報を柔軟に記述できること

ソフトウエア中の型に関する不適切な操作を予め検出し、また実行時コードの効率化を促すためには、コンパイル時に十分な型情報が必要となる。動的な型づけに基づく言語のための型推論機構では、十分な型情報を推論から導くことが困難であるため、コンパイラに対し型情報を伝えるためのインターフェイスが必要であると考えられる。静的に型づけされる言語では型の取り扱いに厳密性を必要とするため、複雑なデータ構造や

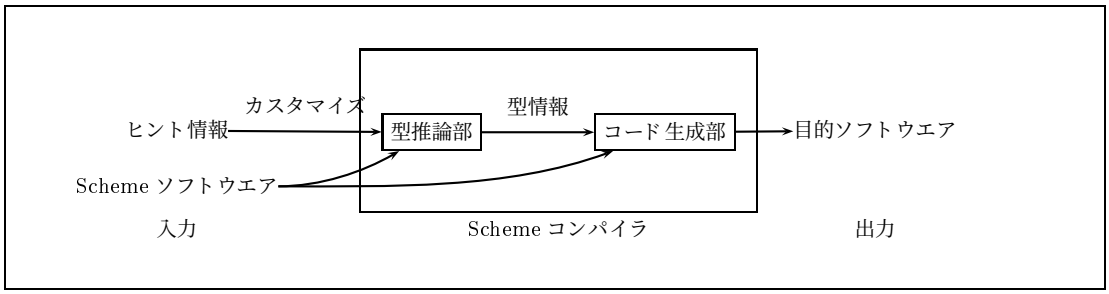

図 3.2: 処理系の構成

式を記述する場合、型を考慮してそれを設計しなければならず、この作業は手間のかかるものとなる。以上から、言語は処理系に対し型情報を伝えるためのインターフェイスをもつべきであると考えられるが、従来の型宣言という形式ではなく、2 章 2.1.3 節で述べたようなプログラマが持つ型に関する考えを直接的に表現できる、より柔軟な形式に基づく方法が重要となる。

3.2 処理系の構成

3.1 節で述べた要求を満たすような言語を実現するために、本研究では、型推論機構を持つ動的な型づけに基づく言語を基礎とし、その型推論機構を自己反映計算機構を用いてカスタマイズ可能にするものとする。

処理系は、図 3.2 に示されるような構成となる。

- 型推論機構をもつ Scheme とする

  これは、Scheme が型を考慮せず自由な記述がゆるされるという特徴を持つ動的に型づけされる言語であることと、型推論機構を持つことにより、型情報を記述することなく推論から式の型を推論し、不適切な操作を実行時より前に検出したり効率的なコードを生成することができるためである。また、Scheme は規模の小さな関数型言語であるため、この言語自体を実現することが難しくないためである。型推論機構は、Scheme の式を抽象実行することにより、変数に束縛され得る値の集合を調べあげるという、素朴な形式となっている。型推論機構の詳細は、次章で詳しく解説を行う。

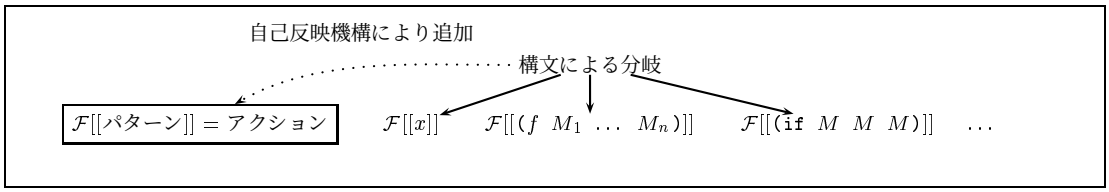

図 3.3: 型推論機構のカスタマイズ

- 型情報を伝えるための構文を持つ

  Scheme 言語に対して型情報を伝えるための構文を用意することにより、プロトタイピングによって作成されたソフトウエア中の、コードが固まって来た分部から順に型宣言を行い、記述性重視から不適切な操作の検出重視への、滑らかな移り代わりを可能にする。

  この型情報を伝えるための構文は、型推論機構に対するコンパイル時自己反映計算機構として実現される。自己反映計算機構を用い型推論機構をカスタマイズすることにより、従来の型宣言の形式だけでなく、前述のような直観的な形式によって処理系に対し型情報を伝えることが可能になる。また、処理系に型に関する情報を伝えるだでなく、型推論機構の型づけ過程のトレースを表示したり、ブレークポイントを設定したりといった、型推論機構の動作を制御することも可能であり、型に基づいたデバッグを支援する。

  カスタマイズは、具体的には型推論機構が構文に応じて処理を振り分ける部分に対し、新たな処理を追加するといった形式となる (図 3.3)。個々のカスタマイズはパターンとアクションから構成され、パターン部分はソフトウエアの抽象構文木とのマッチングに使われ、マッチングに成功した場合にアクション部分が実行される。このアクション部分では、抽象構文木の部分木を変更するだけでなく、型推論機構が取り扱う型に関する情報を操作することができるため、型に関する柔軟な制御が可能となる。

- 型情報を伝えるための構文は、式の中に埋め込まれない

  型情報は、ソフトウエア中に埋め込まれるのではなく、ソフトウエア本体とは別の場所に記述する形式とする。ソフトウエア本体と型情報を別々に記述することによ

り、仕様変更などの理由からコードを修正する必要がある場合、型情報の記述を用意に取り除くことができる。

第 4 章

カスタマイズ可能な型推論機構

4.1 文法と値

解析の対象となる言語は Scheme のサブセットとし、その文法は図 4.1 に示されるものとする。この文法では、値の外部表現として論理値, 整数値を、変数としてシンボルを扱うことができる。`letrec` は相互再帰的な定義を行うための束縛構成手続きであり、`lambda` は手続き, (Exp Exp) は手続き呼び出し, `if` は条件式を表す。また、外部表現をそれ自身に評価する `quote` がある。これらの構文に対する意味は、Scheme のものと同じであるとする。

一方、式の値として取り扱えるものは、図 4.2 に示されるように、整数, シンボル, 論理値, 空のリスト, 組、手続きである。

4.2 型推論機構

プログラムの実行中、ある変数に束縛され得る値の集合を推論するために、制御フロー解析を行う。

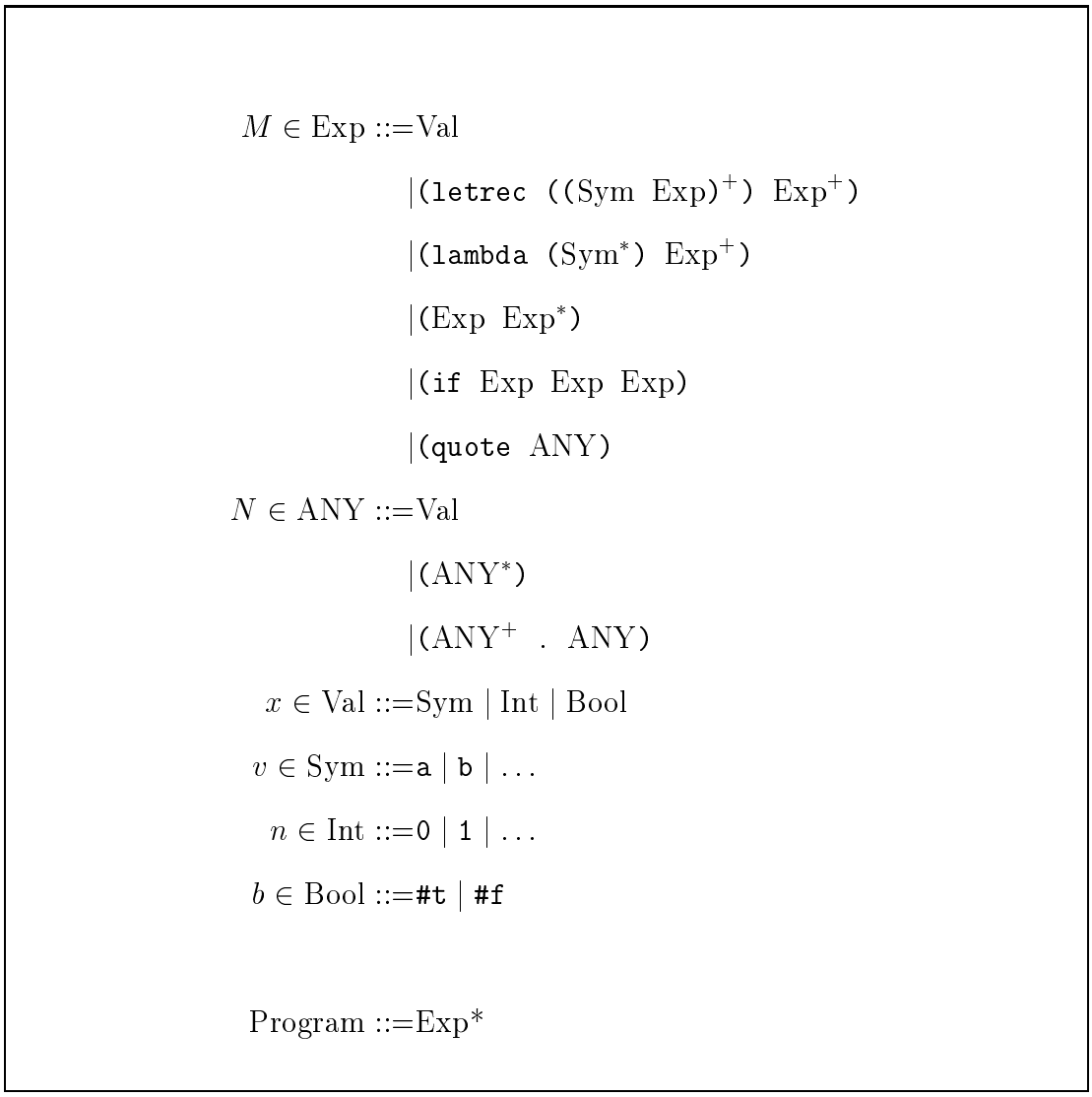
$M \in$ Exp ::=Val
|(letrec ((Sym Exp)$^+$) Exp$^+$)
|(lambda (Sym*) Exp$^+$)
|(Exp Exp*)
|(if Exp Exp Exp)
|(quote ANY)
$N \in$ ANY ::=Val
|(ANY*)
|(ANY$^+$. ANY)
$x \in$ Val ::=Sym | Int | Bool
$v \in$ Sym ::=a | b | ...
$n \in$ Int ::=0 | 1 | ...
$b \in$ Bool ::=#t | #f
Program ::=Exp*

図 4.1: 文法

$$
\begin{aligned}
a_0 \in \mathrm{Value}_0 &= \mathrm{Integer}_0 \cup \mathrm{Symobl}_0 \cup \mathrm{Boolean}_0 \cup \mathrm{Nil}_0 \cup \mathrm{Pair}_0 \cup \mathrm{Closure}_0 \\
\mathrm{Integer}_0 &= \{0, 1, 2, \ldots\} \\
\mathrm{Symbol}_0 &= \{a, b, c, \ldots\} \\
\mathrm{Boolean}_0 &= \{\mathrm{true}, \mathrm{false}\} \\
\mathrm{Nil}_0 &= \{\mathrm{nil}\} \\
\mathrm{Pair}_0 &= \langle v_0, v_0 \rangle \\
\mathrm{Closure}_0 &= \langle (\lambda v_0^*.M^+), r_0 \rangle \\
\\
r_0 \in \mathrm{Environment}_0 &= \mathrm{Sym} \to \mathrm{Value}_0
\end{aligned}
$$

図 4.2: 値

4.2.1 型, 環境

型推論機構が抽象実行時に取り扱う値は、図 4.3 に示される通りである。本来の Scheme が取り扱う値と比較すると、整数値が TYPE/INTEGER に、シンボルが TYPE/SYMBOL に集約されていることと、型推論機構の扱う値が本来の Scheme が扱う値を抽象化した値の集合体の部分集合となっている点が異なる。この集合が、本来の Scheme における型に対応したものである。

型推論機構は、抽象実行を行う際に Env, TEnv の二つの環境を必要とする。Env は変数から値の集合への写像であり、従来の Scheme における環境に対応するものである。

TEnv は、変数とラベルの組から値の集合への写像である。これは、`letrec` や `lambda` において、変数に値が束縛されたことを記憶しておくための環境であり、異なった時点において変数に値が束縛される毎に、それ以前の環境の値と変数に束縛される値の和集合が新たな環境の値となるように拡張される。 ある変数がプログラム中の異なった場所で現れる可能性を考慮して、TEnv 環境の型は、プログラム中の位置を表す型 Label を用

$$
\begin{aligned}
a \in \text{Value} &= pow(\text{Integer} \cup \text{Symbol} \cup \text{Boolean} \cup \text{Nil} \cup \text{Pair} \cup \text{Closure}) \\
\text{Integer} &= \{\text{TYPE/INTEGER}\} \\
\text{Symbol} &= \{\text{TYPE/SYMBOL}\} \\
\text{Boolean} &= \{\text{true}, \text{false}\} \\
\text{Nil} &= \{\text{nil}\} \\
\text{Pair} &= \langle v, v \rangle \\
\text{Closure} &= \langle (\lambda v^*.M^+), r, l \rangle \\
\\
r \in \text{Environment} &= \text{Sym} \to \text{Value} \\
l \in \text{Label} &
\end{aligned}
$$

図 4.3: 型推論機構が取り扱う抽象的な値

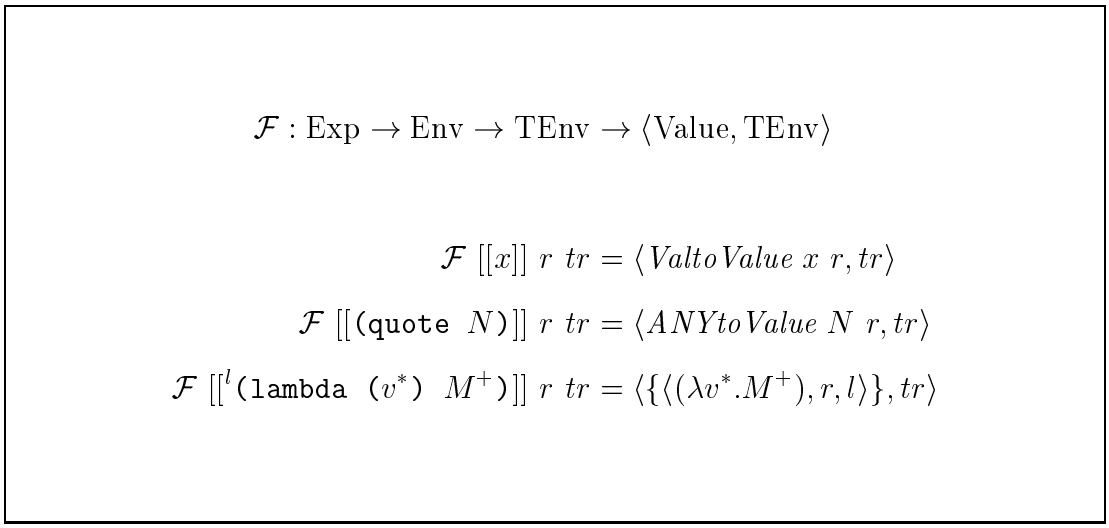

図 4.4: 変数参照, リテラル式, 手続きの意味

いて、

$$\text{TEnv} : (\text{Sym} * \text{Label}) \rightarrow \text{Value}$$

と表すことができる。

4.2.2 変数参照, リテラル式, 手続き

変数参照およびリテラル式の意味は、図 4.4 に示される意味関数 $\mathcal{F}$ によって定義される。意味関数 $\mathcal{F}$ は、Scheme の式と一連の環境を取り、結果として、値と TEnv の組を返す。

式がシンボル, または整数値, 真偽値の外部表現である場合、補助関数 *ValToValue* によって値が求められる (図 4.5)。補助関数 *ValToValue* は、式がシンボルである場合、環境を適用して値を求め、それ以外の場合それぞれの内部表現を値とする。ここで、整数の内部表現は、整数値を抽象化した {TYPE/INTEGER} という値で表現されることに注意すべきである。

一方、式が `quote` である場合は、補助関数 *AnyToValue* によって値を求めている。補助関数 *AnyToValue* は、式が組の外部表現である場合は、再帰的にその要素に対し、補助関数 *AnyToValue* を適用して値へ変換し、空リストの外部表現である場合は、その内部表

現である {nil} を値とする。式がシンボルである場合は、環境を適用せず抽象的な値である {TYPE/SYMBOL} を値とし、式が整数値または真偽値の外部表現である場合は、補助関数 $ValToValue$ により内部表現に変換される。

更に、式が `lambda` である場合、引数と複数の式からからなる手続きと、そのときの環境の組を値とする。これは、クロージャを表現するものである。

4.2.3 条件式

条件式は図 4.6 に示されているように定義される。

条件式は、まずテスト式 (M_c) を評価する。結果として得られた値が、{false} 以外の何かであった場合は帰結式 (M_t) を評価し、{false} である場合は代わりの帰結式 (M_e) を評価する。これは、従来の Scheme と同様のふるまいである。しかし、型推論機構の抽象実行の過程では評価の結果が重ね合わされるため、これらの条件に加え以下のふるまいが追加される。

条件部分の評価の結果が false とそれ以外の値の両方を含む集合であった場合、つまり、抽象実行時にテスト式が真, 偽のどちらにも評価される可能性がある場合、帰結式と代わりの帰結式の両方が評価され、それぞれの値の集合の和集合を結果とする。

更に、条件部分の評価の結果が空集合であった場合は、帰結式と代わりの帰結式の双方とも評価せず、空集合を値とする。これは、抽象実行の過程で異常が発生した場合に対しての処理であり、重要な意味を持つわけではない。

条件式の意味の中での環境 TEnv は、どちらも評価の順に受け渡されて行く形式となっている。

4.2.4 関数適用

関数適用の意味は、図 4.8 のように定義される。

まず、オペレータ式 (f) とオペランド式 (M_i) を評価する。次に、オペレータ式の評価結果をオペランド式の評価結果に適応するのだが、オペレータ式が複数の手続きや、手続き以外の値に評価される可能性があるため、ここでは、手続き一つ一つに対し、関数適応の処理を行いそれぞれの結果の和集合を結果としている。関数適応の処理は、まず、

$$ValtoValue : \text{Exp} \to \text{Env} \to \text{Value}$$

$$\begin{aligned}
ValtoValue\ [[v]]\ r &= rv \\
ValtoValue\ [[n]]\ r &= \{\text{TYPE/INTEGER}\} \\
ValToValue\ [[\texttt{\#t}]]\ r &= \{\text{true}\} \\
ValToValue\ [[\texttt{\#f}]]\ r &= \{\text{false}\}
\end{aligned}$$

$$ANYtoValue : \text{Exp} \to \text{Env} \to \text{Value}$$

$$\begin{aligned}
ANYtoValue\ [[\texttt{()}]]\ r &= \{\text{nil}\} \\
ANYtoValue\ [[\texttt{(}N_1\ \ N^+\texttt{.}\ \ N_2\texttt{)}]]\ r &= (ANYtoValue\ N_1\ r\ .\ ANYtoValue\ \texttt{(}N^+\ \ .\ \ N_2\texttt{)}\ r) \\
ANYtoValue\ [[\texttt{(}N_1\ \ .\ \ N_2\texttt{)}]]\ r &= (ANYtoValue\ N_1\ r\ .\ ANYtoValue\ N_2\ r) \\
ANYtoValue\ [[x]]\ r &= \begin{cases} \{\text{TYPE/SYMBOL}\} & x \in \text{Sym} \\ ValtoValue\ x\ r & \text{otherwize} \end{cases}
\end{aligned}$$

図 4.5: 変数参照とリテラル式のための補助関数

$$\mathcal{F}\ [[\texttt{(if}\ \ M_c\ \ M_t\ \ M_e\texttt{)}]]\ r\ tr = val$$

where

$$\langle cv, ctr\rangle = \mathcal{F}\ [[M_c]]\ r\ tr$$

$$val = \begin{cases} \langle \emptyset, ctr\rangle & cv = \emptyset \\ \mathcal{F}\ [[M_t]]\ r\ ctr & cond \not\ni \text{false} \wedge cond - \{\text{false}\} \neq \emptyset \\ \mathcal{F}\ [[M_e]]\ r\ ctr & cond = \{\text{false}\} \\ \langle tv \cup ev, etr\rangle & \text{otherwize} \end{cases}$$

where

$$\langle tv, ttr\rangle = \mathcal{F}\ M_t\ r\ ctr$$

$$\langle ev, etr\rangle = \mathcal{F}\ M_e\ r\ ttr$$

図 4.6: 条件式の意味

$$\mathcal{F}^* : \mathrm{Exp}^* \to \mathrm{Env} \to \mathrm{TEnv}$$

$$\mathcal{F}^*\ [[\langle M \rangle]]\ r\ tr = \mathcal{F}\ [[M]]\ r\ tr$$

$$\mathcal{F}^*\ [[\langle M_1, M_2, \ldots, M_n \rangle]]\ r\ tr = \mathcal{F}\ [[\langle M_2, \ldots, M_n \rangle]]\ r\ tr'$$

$$\text{where } \langle v, tr' \rangle = \mathcal{F}\ [[M]]\ r\ tr$$

$$\mathit{ExtendTEnv} : \mathrm{TEnv} \to \mathrm{Symbol}^* \to \mathrm{Value}^* \to \mathrm{Label} \to \mathrm{TEnv}$$

$\mathit{ExtendTEnv}\ tr\ v\ av\ l = tr_n$

where

$tr_0 = tr$

$$tr_i = \begin{cases} tr_{i-1}[\langle v_i, l \rangle \to av_i] & tr_{i-1}\langle v_i, l \rangle = \bot \\ tr_{i-1}[\langle v_i, l \rangle \to (av_i \cup tr\langle v_i, l \rangle)] & \text{otherwise} \end{cases}$$

図 4.7: 関数適用, バインディング構成手続きのための補助関数

$$App : \mathrm{Value} \to \mathrm{Value}^* \to \mathrm{TEnv} \to \mathrm{Label} \to \langle \mathrm{Value}, \mathrm{TEnv} \rangle$$

$$App\ f\ av\ tr\ l = \begin{cases} \langle \emptyset, tr \rangle & f = \emptyset \\ App\ (f - v)\ av_i\ tr' & v \in f \wedge v \neq \langle \lambda v_1 \ldots v_n.M_1 \ldots M_n, r, l \rangle \\ \langle v \cup v', tr'' \rangle & \langle \lambda v_1 \ldots v_n.M_1 \ldots M_n, r, l \rangle \in f \end{cases}$$

where

$$\langle v, tr' \rangle = \mathcal{F}^* \langle M_1, \ldots, M_n \rangle\ (r[v_i \to av_i])\ (ExtendTEnv\ tr\ v\ av\ l)$$

$$\langle v', tr'' \rangle = App\ (f - \langle \lambda v_1 \ldots v_n.M_1 \ldots M_n, r, l \rangle)\ av_i\ tr'$$

$$\mathcal{F}\ [[(f\ \ M_1\ \ \ldots\ \ M_n)]]\ r\ tr = App\ f'\ av\ tr_n\ l$$

where

$$\langle f', tr_0 \rangle = \mathcal{F}\ [[f]]\ r\ tr$$

$$\langle av_i, tr_i \rangle = \mathcal{F}\ [[M_i]]\ r\ tr_{i-1}$$

図 4.8: 関数適用の意味

組によって表現されるクロージャに格納されている手続きが作られたときの環境を、同じくクロージャに格納されている手続きの、仮引数名であるシンボル v_i それぞれに、オペレータ式の評価結果の値それぞれを束縛するように拡張する。次に、拡張された環境の元で手続きの本体部分を 1 つずつ評価して行き、最後に評価した式の値を結果とする。

4.2.5 バインディング構成手続き

バインディング手続きの意味は、図 4.9 のように定義される。

まず、バインディング部分の変数 (v_i) それぞれが未定義の値、つまりここでは空集合を束縛するように環境が拡張される。同様に、型環境も未定義の値と式の位置を示すラベルを元に拡張される。これらの環境と、バインディング部分の初期値式 (M_i) と、ラベルに対し、$LetrecAssign$ 補助関数が適用される。$LetrecAssign$ 補助関数は、与えられた初期値式を 1 つずつ評価し、その値を元に、環境と型環境を更新する。

$LetrecAssign$ 補助関数の適応の後、更新された環境においてボディ部を構成する式それぞれが評価され、最後に評価された式の値を結果とする。

ここでの重要な点は、バインディング部の初期化式を評価する時点で、既に環境が仮の値によって拡張されていることから、相互再帰的な手続きの記述ができることである。

4.3 ヒント情報

型推論機構に対するヒント情報は、図 4.10 に示されるようにパターン部と式から構成され、これを記述するために `match` 構文が導入される。`match` 構文は 2 つの式をとり、1 つ目をパターン部, 2 つ目をアクション部と呼ぶ。パターン部は、抽象構文木の構造との比較のために使われ、比較に成功した場合にアクション部分が評価される。つまり、ヒント情報とは、型推論機構の意味関数 $\mathcal{F}$ に対応するものであり、`match` 構文を用いると、意味関数 $\mathcal{F}$ を拡張することができるわけである。

パターン部は図 4.10 に示すように、Scheme の式と同じ構造を記述することができ、それは記述したものと同一の式にマッチする。ただし、下線 (_) だけは特殊で、これは任意の式 (Exp) にマッチする。

アクション部は、抽象構文木の中のある部分の構造が、パターン部分に記述された構

$$LetrecAssign : \text{Symbol}^* \to \text{Exp}^* \to \text{Env} \to \text{TEnv} \to \text{Label} \to \text{TEnv}$$

$$LetrecAssign\ \langle\rangle\ \langle\rangle\ r\ tr\ l = tr$$

$$LetrecAssign\ \langle v_1, v_2, \ldots v_n\rangle\ \langle M_1, M_2 \ldots, M_n\rangle\ r\ tr\ l =$$

$$LetrecAssign\ \langle v_2, \ldots v_n\rangle\ \langle M_2 \ldots, M_n\rangle\ rtr'\ l$$

where

$$\langle a, tr'\rangle = \mathcal{F}\ [[M_1]]\ r\ tr$$

$$r := r[v_1 \to a]$$

$$tr' := ExtendTEnv\ tr\ v_1\ a\ l$$

$$\mathcal{F}\ [[{}^{l}\texttt{(letrec ((}v_1 M_1\texttt{)) ...(}v_n M_n\texttt{))}\ M'_1 \ldots M'_m\texttt{)}]]\ r\ tr_0 = \mathcal{F}^*\ \langle M'_1 \ldots M'_n\rangle\ r'\ tr'$$

where

$$r' = r[v_i \to \emptyset]$$

$$tr_i = ExtendTEnv\ tr_{i-1}\ v_i\ \emptyset\ l$$

$$tr' = LetrecAssign\ \langle v_i \ldots v_n\rangle\ \langle M_1, \ldots, M_n\rangle\ r\ tr_n\ l$$

図 4.9: バインディング構成手続きの意味

$$\text{Program'} ::= (\text{Exp} \mid \texttt{(match}\ \text{Pattern}\ \text{Exp}\texttt{)})^*$$

$$\begin{aligned} p \in \text{Pattern} ::= {} & \_ \\ \mid {} & \text{Val} \\ \mid {} & \texttt{(}\text{Pattern}^*\texttt{)} \\ \mid {} & \texttt{(}\text{Pattern}^+\ \texttt{.}\ \text{Pattern}\texttt{)} \end{aligned}$$

図 4.10: ヒント情報の形式

$$\mathcal{M} : \text{Exp}^* \to (\text{Exp} \to \text{Env} \to \text{TEnv} \to \langle \text{Value}, \text{TEnv} \rangle) \to \text{Env} \to \text{TEnv} \to \text{TEnv}$$

$$\mathcal{M}\ [[\ ]]\ f\ r\ tr = tr$$

$$\mathcal{M}\ [[M_1\ M_2 \ldots M_n]]\ f\ r\ tr = \mathcal{M}\ [[M_2 \ldots M_n]]\ f\ r\ tr'$$

where $\langle a, tr' \rangle = f\ [[M_1]]\ r\ tr$

$$\mathcal{M}\ [[\texttt{(match}\ p\ M\texttt{)}\ M_2 \ldots M_n]]\ f\ r\ tr = \mathcal{M}\ [[M_2 \ldots M_n]]\ f'\ r\ tr$$

where $f' = f\ [GenFunc\ p\ M\ f]$

図 4.11: 型推論機構のドライバ

造にマッチした場合に実行される式である。この式には、型推論機構が扱うデータが受け渡されることになるため、lambda 構文を用いて手続きを記述しなければならない。具体的には、

$$\texttt{(lambda (exp env tenv self super)}\ M^{+}\texttt{)}$$

を記述することになる。引数は exp, env, tenv self super の 4 つであり、これらはそれぞれ、パターンにマッチした部分式, 抽象実行中の環境, 推論中の型環境, 抽象実行中の意味関数, 定義時の意味関数が渡される。これらの値をもとに処理を行った結果は (env' tenv') のようにリストで返す。ここからわかるように、アクション部で行うことができるのは、環境と型環境に対する参照および変更ということになる。

型推論機構の実行は、図 4.11 の関数 $\mathcal{M}$ で示されるように行われる。$\mathcal{M}$ はプログラムと意味関数 (f), 環境 (e), それに型環境 tr を取り、型環境を返す。プログラムは式か match 構文の並んだものであるが、関数 $\mathcal{M}$ はそれを先頭から順に処理して行く。

先頭が式である場合、その式および環境と型環境に対しに対し意味関数 f を適用して値と新しい型環境を得た後、プログラムの残りの部分を関数 $\mathcal{M}$ により再帰的に処理する。このとき、得られた値は捨てられ、型環境は受け渡される。

先頭が match 構文である場合、そのパターン部とアクション部をもとに、補助関数 $GenFunc$ を用いて意味関数のための部分関数を作成し、それを用いて意味関数を更新する。プログラムの残りの部分の処理は、関数 $\mathcal{M}$ により再帰的に実行されるが、それらに使われる意味関数は、ここで更新されたものとなる。

第 5 章

記述例

5.1 素朴な型宣言

素朴な型宣言の例として、次のような式を考える。

```
(letrec ((x (f a b))) (g x))
```

これは、変数 a, b の値に対して手続き f を適用した結果を変数 x で束縛し、その変数 x に対して手続き g を適用する式である。この式に現れる変数 x が整数型であると宣言したい。この場合、いくつかの方法が考えられるが、次のように記述すると簡潔である。

```
(match (letrec ((x (f a b))) (g x)) (lambda (exp env tenv self super)
  (let* ((a-and-tr (super exp env tenv))
         (newtenv (cadr a-and-tr)))
    (or (Type::euqal? (Type::integer)
                      (Env::lookup newtenv `(x ,(Exp->Label exp))))
        (write (list 'type-missmatch (car a-and-tr) (abstract exp))))
    a-and-tr)))
```

この例では、型宣言を行いたい変数を含む式に完全にマッチするように、パターンが記述されている。一方、アクション部は 5 つの引数を取る手続きがあり、その引数はそれぞれ抽象構文木の中のパターンにマッチした部分 (exp), 抽象実行中の環境 (env), 型環境 (tenv), 抽象実行時の意味関数 (self), このヒント情報が定義される直前の意味関数

(super)である。ここでは、まずsuperを用いて拡張する前の意味関数を用いて計算を行い、その結果をa-and-trで束縛し、また、a-and-trのcadr部分、つまり、意味関数によって計算された型環境がnewtenvで束縛する。

つぎに、シンボルxと構文木expのラベルの組にnewtenvを適用し、その値が{TYPE/INTEGER}であるかどうかを調べる。この調査が、アクション部で指定したletrec式において、変数xが整数型であることを確認している。型が異なる場合には、整数値以外が束縛される可能性があるのだから、型エラーとしてメッセージを表示する。そして、a-and-trを結果とすることにより、処理を終える。

5.2 変数の名前に応じた型宣言

変数名の頭文字がiであるものは整数型である、のように、変数の名前に応じてその型が決まるというケースは、次のように書くことで対処できる。

```
(match (letrec _ _) (lambda (exp env tenv self super)
  (let* ((a-and-tr (super exp env tenv))
         (newenv (cadr a-and-tr)))
    (for-each (lambda (binding)
      (and (eq? #\i (string-ref (symbol->string (car binding)) 0))
           (not (Type::equal? (Type::integer)
                              (Env::lookup newenv
                                           `(,(car binding)
                                            (Exp->Label exp)))))
           (write `(type-missmatch ,(car a-and-tr)
                                   ,(car binding)
                                   ,(abstract exp)))))
      (cadr exp))
    a-and-tr)))
```

この例では、パターン部に(letrec _ _)と記述してるため、全てのletrec式にマッチする。マッチした場合のアクションは、まず、前の例と同様に、superを用いて拡張する

前の意味関数において、この構文に対する処理を行う。そして、その結果を a-and-tr で束縛し、意味関数によって計算された型環境が newtenv で束縛する。

次に、マッチした letrec 構文のバインディング部 1 つ 1 つに対して、変数名の頭文字が i かどうかを検査する。もし、頭文字が i であった場合は、型環境 tenv を用いてそれに束縛され得る値の候補を得て、その値が {TYPE/INTEGER} で無い場合はメッセージを表示するというものである。最後に、a-and-tr を値として作業を終える。

変数束縛は letrec 式だけでなく、関数適用時の手続きでも行われるため、こちらも検査する必要がある。関数適用部分に関する検査は、次のように書けば良い。

```
(match (_ _) (lambda (exp env tenv self super)
  (if (memq (car exp) '(letrec lambda if quote))
      (super exp env tenv)
      (letrec ((a-and-tr (super exp env tenv))
               (newenv (cadr a-and-tr)))
        (for-each (lambda (f)
          (and (Type::function? f)
               (for-each (lambda (v)
                 (and (eq? #\i (string-ref (symbol->string v) 0))
                      (not (Type::equal? (Type::integer)
                           (Env::lookup newenv
                                        `(,v
                                        (Type::function-label f)))))
                      (write `(type-missmatch ,(car a-and-tr)
                                              ,v
                                              ,(abstract exp)))))
                 (Type::function-args f))))
          (car (f (self (car exp) env tenv))))
        a-and-tr)))))
```

この場合も、letrec の場合と同様で、まず本来の解析を行いその結果に対して型検査を行っている。具体的には、まず、パターンとして関数適用の形式が指定されており、全

ての関数適応に対してこれはマッチする。更に、letrec や lambda といった特殊形式にもマッチしてしまうため、まず、それらを排除する。

次に、exp 部分の解析を行い、その結果と結果に含まれる型環境をそれぞれ、a-and-tr, newtenv で束縛しておき、関数適用の関数部分の式とと一連の環境に対し self を適用して適用される手続きの候補を求める。この 1 つ 1 つに対し、それが関数である場合に、その引数の頭文字が i であるかどうかを検査し、そうであれば、型環境 tenv を用いてそれに束縛され得る値の候補を得て、その値が {TYPE/INTEGER} で無い場合にメッセージを表示する。そして、最後に a-and-tr を値として作業を終える。

第 6 章

まとめ

本研究では、Scheme の為の型推論機構に対する自己反映計算を用いたカスタマイズの方法を定義し、この方法に従いカスタマイズ可能な Scheme の型推論機構を実装した。

更に、定義に基づいたシステムを作成し、例題を与えることで動作の確認を行った。

以下では、考察を行い、今後の課題を述べる。

6.1 考察

6.1.1 評価

本研究で提案した処理系は、Scheme のための型推論機構に対し、コンパイル時自己反映計算の形式に従い、型に関する情報を与えられるようにするものである。この手法には、次のような利点があることが、処理系を実装し、それにに対していくつかの記述例を与えることによって示すことができる。

- 本研究で提案した処理系のための言語は、Scheme に `match` 構文を追加したものであるため、Scheme の持つ特徴を継承する
- 型推論機構を用いて、プログラマが明示しなくても、ある程度の型情報を獲得することができる
- 型推論機構が推論する型情報よりも多くの型情報を、ヒント情報を記述することに

よりコンパイル時に得られるようにすることができる

- メタ記述としてのヒント情報は、ベースレベルであるプログラムから分離して記述することができる。

一方、問題点として、次にあげるようなものが得られた。

- 型推論機構が停止しないことがある
- ヒント情報の適用範囲の制御ができない
- ヒント情報の記述量が多い

型推論機構は Scheme の式を抽象実行することにより型情報を獲得している。これに、無限ループを行うようなプログラムを型推論機構に与えると、型推論機構も無限ループに陥り、型情報を得ることができなくなってしまう。これは、本研究で提案する処理系が、停止性関して考慮をしていないことが原因である。これに対処するためには、型推論機構の推論規則を注意深く構成する、または、コールフローグラフに基づいて無限ループに陥るのを避けるといった方法が考えられる。

ヒント情報の適用範囲が制御できない問題に関しては、ヒント情報のパターン部に、構文木との比較を行うためのパターンに対し、有効範囲を制御するための拡張を導入すれば良いと考えられる。また、パターン部に述語を記述できるようにするという方法も、適用範囲をきめ細かく制御することを可能にすると考えられる。

ヒント情報の記述量に関しては、これが型を扱うという目的のためのプログラムであるということから、より短い記述を導くことは難しい。これに関しては、ヒント情報の記述を一般化しライブラリ化することにより対処すればよいと考える。

6.1.2 他言語との比較

SoftScheme

本研究で提案した処理系では、メタ記述が用いられない場合は型推論機構をもつ Scheme の処理系としてふるまい、型推論のアルゴリズムが異なるために得られる型情報の制度に違いが見られるものの、SoftScheme と同程度の情報が得られる。本研究で提案した処

理系では、メタ記述を用いることにより、更に多くの型情報をコンパイラに与えることができる点において SoftScheme に勝っている。

Standard ML

本研究で提案した処理系の型推論機構は、動的に型づけされる言語のための型推論機構であるため、Standard ML に見られるような完全な型づけを実現するのは難しい。しかし、本研究で提案した処理系は、動的に型づけされる言語の特徴である自由度の高い記述性を失うことなく、必要に応じてメタ記述を通して型情報を付け加えて行くことができ、動的な型づけに基づく言語としての側面と、静的に型づけされる言語としての側面の双方の利点をある程度両立することができている。この点において、本研究で提案した処理系は Standard ML よりも適用範囲が広いと考えられる。

Common Lisp

Common Lisp では、`declare` と型指定子 `type`, `ftype`, `the` 等を用いて変数や式の値の型を宣言できる。Common Lisp が動的に型づけされる言語である点や、Common Lisp における型宣言の目的が、Common Lisp コンパイラが、コンパイル時に型誤りを検出したり、効率的な実行プログラムを生成したりすることができるようにする点等、本研究と共通する部分が多い。

これらの間の大きな違いは型情報の捉え方にある。Common Lisp では、型宣言をその他の言語機能と同一のレベルの概念として捉えているのに対し、本研究では、型に関する操作を本来の計算に対するメタレベルの概念として捉えている。Common Lisp では、本来の計算を行う式の中に型宣言のための式が挿入されてしまうのに対し、本研究では、自己反映計算の概念に基づき、本来の計算とその型に関する操作を分離することに成功している。これらの分離から得られる利点は、それぞれのレベルの記述をそれぞれ別々に再利用できる点にある。具体的には、多相的な手続きに対し用途に応じたメタレベルで解釈することにより、その式を単相的に特化させるといったベースレベルの再利用と、ある式のパターンに対する型の解釈をスタイル化するといったメタレベルの再利用が実現される。

第 2.2.2 節において述べたように、マクロはコンパイル時自己反映計算として捉えることができる。これと、本研究で提案した処理系のヒント情報は非常に類似している。具体的には、どちらもパターン部と本体部 (本研究ではアクション部と呼ぶ部分) からなり、このパターンにマッチする式やトークン等を、本体部分に記述した通りに解釈するように指定するメタ記述として捉えることができることが類似点である。

一方、相違点は、一般的なマクロの操作がプログラムテキストのトークン列や構文木の書き換えであるのに対し、本研究で提案したシステムは、Scheme の型推論のための意味の変更を行う点である。このため、マクロの本体部分にテキストしか書けないのに対し、本研究で提案する処理系では対応するアクション部に式が書け、パターンにマッチした時の情報をもとに計算を行うことができるといった違いがある。

OpenC++

OpenC++では、カスタマイズの対象が構文木ではなくパース木である。このため、プログラムの構文ではなく、論理的な構造に基づいた自己反映的な処理を行うことができる。これは、C++のような構文が複雑な処理系に対するマクロの実現に、極めて有用な方法であると考えられる。

一方、本研究で提案した処理系は、対象としている言語が Scheme であるため、構文は極めて単純であることから、構文に基づいたマッチングを採用した。しかし、`letrec` 式や `lambda`, 関数適用など、意味が複雑な構文に対するヒント情報を記述しようとした場合、既に定義されている意味の粒度が高いため、うまくそれらを再利用できないといった問題点がある。

本研究で提案したシステムを、OpenC++のように、構文とは異なった単位に基づくヒント情報を与えられるように変更することにより、この問題点に対処できるのではないかと考えられる。

6.2 今後の課題

今後の課題として、次にあげるものが考えられる。

- 型推論機構の改善

本研究で定義した型推論機構には、Scheme のサブセットを対象としているために扱うことのできる構文が少ないことや、推論の停止性を考慮していないことなどといった問題点がある。構文が少ないことに関しては、未定義の構文を追加することにより解決することができるが、停止性に関しては現在の型推論機構を修正することにより、実現することは難しい。

- ヒント情報の形式の改善

  本論文に示した定義に基づくヒント情報の有効範囲は、それを記述した箇所より後全体となってしまい、1 つのプログラムの中の特定の範囲にのみ適用させることができない。また、ヒント情報のパターン部に対し、マッチして欲しい式と、それに類似しているマッチして欲しくない式を区別できるように記述するのが難しい場合がある。ヒント情報の有効範囲の制御やその指定方法に関しての、より柔軟な記法を考える必要がある。

参考文献

[1] Cormac Flanagan and Matthias Felleisen. Set-based analysis for full scheme and its use in soft-typing. Technical Report TR95-254, Rice University, October, 1995.

[2] GUY L. STEELE JR. *COMMON LISP THE LANGUAGE.* Digital Equipment Corporation, 2nd edition, 1990.

[3] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[4] Tim Sheard and James Hook. *Meta-Programming tools for ML.* Pacific Software Research Center, February 1994. `http://www.cse.ogi.edu/~sheard/crml.html`.

[5] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 3, pages 47–88. MIT Press, 1991.

[6] Andrew K. Wright and Robert Cartwright. A practival soft type system for Scheme. Technical Report COMP TR93-218, Depertment of Computer Science, Rice University, December 1993.

[7] 千葉 滋. *OpenC++ 2.5 Reference Manual.* Institute of Information Science and Electronics, University of Tsukuba, 1999. `http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html`.

[8] 渡部卓雄. チュートリアル リフレクション. **コンピュータソフトウエア**, 11(3):5–14, November 1994.