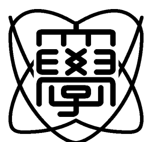

平成23年度 修士論文

メモ化の適用範囲を拡張するプログラム変換器の設計と実装

電気通信大学 大学院情報システム学研究科

情報システム基盤学専攻

1053009 康 娜丹

指導教員 小宮 常康 准教授
多田 好克 教授
大森 匡 教授

提出日 平成24年1月26日

目 次

第 1 章　序論 1

第 2 章　メモ化と自動メモ化 3

2.1 メモ化 3

2.2 自動メモ化 5

第 3 章　研究の動機 8

第 4 章　メモ化の大域変数問題 10

第 5 章　提案方法 13

第 6 章　設計 18

6.1 フロー解析 18

6.1.1 大域変数情報の収集 18

6.1.2 不動点を求める 22

6.2 プログラム変換 24

第 7 章　実装 28

7.1 準備 28

7.1.1 Scheme の構文 30

7.1.2 環境構造 30

7.2 プログラム解析 33

7.2.1 変数情報 34

7.2.2 処理関数 36

7.2.3 解析例 42

7.3 プログラム変換 45

第 8 章 実験 **47**

第 9 章 関連研究 **53**

9.1 STM によるメモ化適用範囲の拡張 53

9.2 チェックポインティングによるデバッギング 54

第 10 章 今後の課題 **56**

第 11 章 結論 **58**

図 目 次

2.1 メモ化 4

3.1 Mugshot を応用したプロセス移送 9

7.1 変換器処理のフローチャート 29
7.2 変換器の環境モデル 31
7.3 単純な環境構造 32
7.4 lambda 式を解析する際の環境 39
7.5 解析過程 43
7.6 変換処理 46
7.7 変換例 46

8.1 ○×ゲームのゲーム木 50

表 目 次

5.1 関数 f と g のためのテーブル 17

6.1 大域変数情報の収集 23

7.1 変数情報 34

8.1 実験環境 47
8.2 tak の実行結果 48
8.3 評価器の実験結果 49
8.4 ゲーム木の実験結果 51

第 1 章

序論

メモ化は重複した計算を省略することでプログラムを高速化する手法である。この手法では、関数呼び出しの引数と返値のペアを記憶しておき、同一の関数呼び出しを再度行う際に、記憶した返値を再利用する。プログラムにメモ化を適用するために自動メモ化という技術が利用される。自動メモ化はメモ化のためのコードをプログラムに自動挿入するので、プログラマはソースコードを書き直す作業から解放される。しかし、メモ化可能な関数は参照透過性を備えるものに限られる。関数が大域変数へアクセスする場合、大域変数の値がこの関数の計算結果、あるいは次に呼び出される関数の計算結果に影響する。プログラムに対して自動メモ化を行う際、このような関数はメモ化できない。本研究では、メモ化適用可能な関数の範囲を拡張するために、大域変数への参照や代入がある関数に対してプログラム変換を行う。具体的には、大域変数の値を伝えるためのパラメータを関数へ追加する。また、大域変数へ代入された値を本来の返値と組み合わせたペアやリストで返すようにする。変換後の各関数には大域変数に相当する局所変数を導入し、大域変数の代わりに、局所変数にアクセスすることによって、間接的に大域変数の参照や代入を行うようにする。関数にメモ化を適用する場合、参照した大域変数に相当する局所変数の値が引数として関数に渡されるので、キーの一部としてテーブルに記憶される。また、大域変数に相当する局所変数に代入された値も関数の返値になるので、それもテーブルに記憶される必要がある。その後に、同じ関数を呼び出す場合、本来の引数の値と大域変数の値から成るキーを辿り、過去の計

算結果を検索することができる。このような変換によって、大域変数へアクセスする関数がメモ化可能な関数になり、プログラムのセマンティクスを守りながら、自動メモ化を利用することができる。

本論文は、以下のように構成されている。第 2 章では、メモ化と自動メモ化技術について説明する。第 3 章では、メモ化を研究する動機について述べる。続いて第 4 章では、大域変数へアクセスする関数に対してメモ化を適用できない原因を説明する。第 5 章では、メモ化の大域変数問題を解決するためのプログラム変換手法を提案する。第 6、7 章はプログラム変換器の設計と実装である。第 8 章では、変換後のプログラムに対してメモ化を適用した実験結果を考察する。第 9 章では、関連研究について述べる。第 10、11 章は今後の課題と結論である。

第 2 章

メモ化と自動メモ化

2.1 メモ化

メモ化はプログラムの高速化のための最適化手法の一種である。メモ化された関数は、過去の呼び出しの結果を当時の引数とともにテーブルに記憶しておき、後で同じ引数で呼び出される際、計算せずに記憶されている結果を返す。メモ化によって、重複した計算が省略できる。しかし、関数の実行時間コストが削減される反面、メモリコストが増えることがある。

例えば、図 2.1 に示すように、関数 f が引数 c1、c2 で 1000 回呼出されるとする。メモ化しない場合、関数 f が引数 c1、c2 を受け取り、同じ実行を 1000 回繰り返し実行する。一回の実行時間が t であれば、上記のループ実行のために 1000t の時間コストがかかる。この関数にメモ化を適用して実行する場合、関数 f が引数 c1、c2 で初めて呼び出される際、引数 c1、c2 の値及び実行した結果がテーブルに記憶される。その後の 999 回の呼出しでは、テーブルをルックアップして引数 c1、c2 というキーを見つけるので、そのキーと対応する値を返す。一方、関数 f が c1、c2 以外の引数で呼出される場合は、テーブルの中に対応するキーがないので、新しいエントリがテーブルに追加される。関数 f を引数 c1、c2 で 1000 回呼出すプログラムは実質上、関数 f の計算を一回しか実行しない。実行コストは t ＋検索時間× 999 になる。関数 f の実行時間がキーの検索時間より大きければ、メモ化によって、このプログラムの実行時間が削減できる。

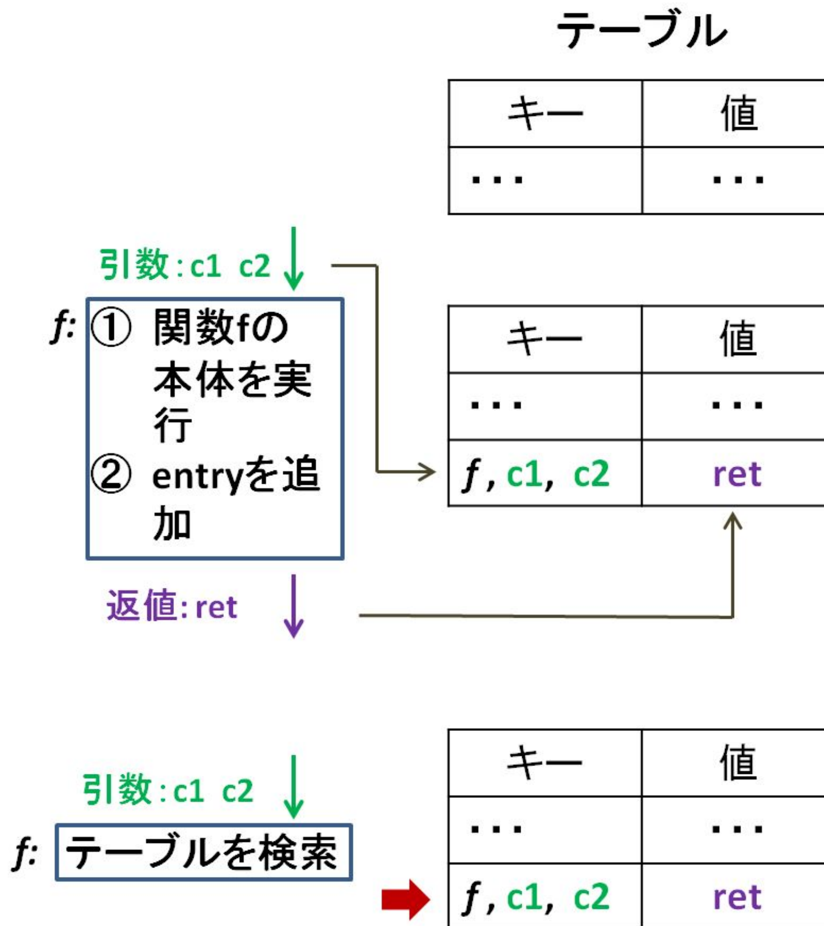

図 2.1: メモ化

メモ化された関数が呼び出される直前に、まず、テーブルを検索し、もし引数と同じキーを見つければ、関数を実行せずに、テーブルに記憶された返値を返す。そうでなければ、関数を実行し、計算結果を返す前にテーブルに格納する。もし再び同じ引数でこの関数を呼び出せば、計算は省略される。メモ化を関数に適用するために、ソースコードを書き直す必要がある。既存の関数定義にテーブルを構築するコード、テーブルを検索、結果を返すコード及びテーブルにエントリを追加するコード等が挿入される。しかし、この作業はプログラマにとって煩わしい。関数が異なると、メモ化ためのコードは多少変わる。複雑な関数を書き直す時に、バグを導入してしまう可能性もある。これは、メモ化を利用し、ソフトウェア性能を改

善することを期待している開発者にとって、非常に厄介である。このような問題を解決するために、自動メモ化が登場し、プログラマを煩わしい仕事から解放する。

2.2 自動メモ化

自動メモ化によって、メモ化するためのコードを自動的にプログラムに挿入する。プログラマはソースコードを書き直す必要がない。

フィボナッチ関数を例として挙げる。この関数は引数 n を受け取り、n 番目のフィボナッチ数を返す。n が 0 または 1 であれば、フィボナッチ数は 1 である。そうでなければ、フィボナッチ数は n-1 番目と n-2 番目の数の和になる。

```
1  function  fib (n)
2    if n=0 or n=1
3     then return 1
4     else return fib(n-1)+fib(n-2)
5     end if
6  end function
```

フィボナッチ関数の実行時間は引数 n に関わっている。n が大きければ大きいほど、実行時間が急激に増加する。一方、n 番目のフィボナッチ数を求める前に、n-1 番目と n-2 番目のフィボッナチ数は既に得た。従って、メモ化によって、以前に実行した結果を再利用し、実行効率を上げられる。自動メモ化の一手法としては各関数をメモ化させる汎用メモ化関数を利用する手法がある。この汎用メモ化関数を以下に擬似コードで示す。

```
1  function  memo (F)
2    make an associative array called table
3    if F.table[arg] is empty
4      then F.table[arg]=F(arg)
5    end if
6    return F.table[arg]
```

```
7    end function
```

関数をデータとして扱える言語では、このような自動メモ化手法が利用できる。本来の関数を呼出す際、この汎用メモ化関数が呼び出される。例えば、上の fib 関数を自動メモ化する場合、汎用メモ化関数は以下のように利用される。

```
1  function  fib (n)
2     if n=0 or n=1
3      then return 1
4      else return fib(n-1)+fib(n-2)
5      end if
6   end function
7
8   var fib = memo (fib)
```

以上のコードは関数 fib を再定義し、関数オブジェクトを引数として関数 memo に渡す。fib が引数 10 で呼び出される際、引数 10 が再定義された fib に渡される。それから、新しい fib はテーブルを検索したり、エントリを追加したりする。関数 fib が再帰的に呼び出される際、同じ引数で再び呼び出される場合は過去の計算結果を直接取り出すだけで済む。実行時間は依然として n の大きさと関わっている。しかし、メモ化される前のように指数関数的に増加することはなく、実行時間が大幅に削減される。一方、メモリの消費とテーブルの検索時間を考慮する必要がある。メモ化された関数が初めてある引数で呼び出される際、引数と返値からなるエントリをテーブルに追加する必要がある。関数の実行効率が良くなっても、メモリが本来の関数より多く消費されるかもしれない。よって、メモリ使用の制限がある場合、メモ化を利用できない時もある。また、テーブルに記憶されたエントリ数が多い場合、関数の引数と対応するキーをテーブルから辿る時間がかかる。もしテーブルの検索時間が通常の実行時間より大きければ、プログラムの実行効率は上げられない。従って、検索時間を短縮する方法を探す必要がある。

メモ化を利用するために、一つ重要な前提条件がある。メモ化可能な関数は参照透過性を備えるものに限られている。プログラムに対して自動メモ化を行う場合、プログラマはソースコード中の関数に対して、メモ化するかしないかを表す印を付けるなどして区別をつけておく必要がある。例えば、副作用を含む関数に「メモ化しない」ことを表す印を付ける。自動メモ化はソースコードを読み込んで、その印が付いていない関数についてのみメモ化できるように変換する。

第 3 章

研究の動機

前章で、自動メモ化について説明した。メモ化を適用できない関数が多いプログラムに対して自動メモ化を行う場合、メモ化の最適化効果が制限される。本研究では、自動メモ化をより広い範囲のプログラムに適用できるようにするためのプログラム変換器を開発した。

動機となった自動メモ化の活用例について述べる。自動メモ化を利用し、プロセスの移送効率を良くすることを考えている。James Mickens らの Mugshot システム [1] は JavaScript プログラムのイベントをキャプチャし、デバッギングのために、過去の実行を再演するシステムである。あるデバッギング手法はバグが出た際のスナップショットを開発者に送る。しかし、この手法では、バグを起こる原因が調べにくい。Mugshot は再演法を利用し、非決定的に起こったイベントを記録することによって、プログラムの実行を決定的に再現する。このように、バグの原因となるイベントを調べられる。Mugshot の JavaScript ライブラリを組み込んだアプリケーションをユーザが利用すると、マウスクリックなどの非決定関数呼出しのログが記録される。もしそのアプリケーションがバグによって実行の不具合が生じると、イベントのログが関係者に送られる仕組みになっている。開発者は送られたログファイルに基づいて、不具合の生じた実行を再演し、バグの原因を調べる。この手法をプロセス移送のために流用することが可能であると考えている。図 3.1 のように、移送先でイベントログファイルに記録されたイベントリストによって、移送元の計算過程を丸ごと再演し、中断した時点の状態へ復元することができる。

JavaScript はノンプリエンプティブな単一スレッドとイベント駆動の特性を持っているので、コンテンツ及び順序づけたイベントのリストさえあれば、再演に十分である。イベントしか記録しないので、中断する時点の実行状態を保存することと比べ、コストが低い。しかし、元の実行を再現するために、中断した時点までのプログラムを再実行しなければならないので、移送先での再演時間が長い。

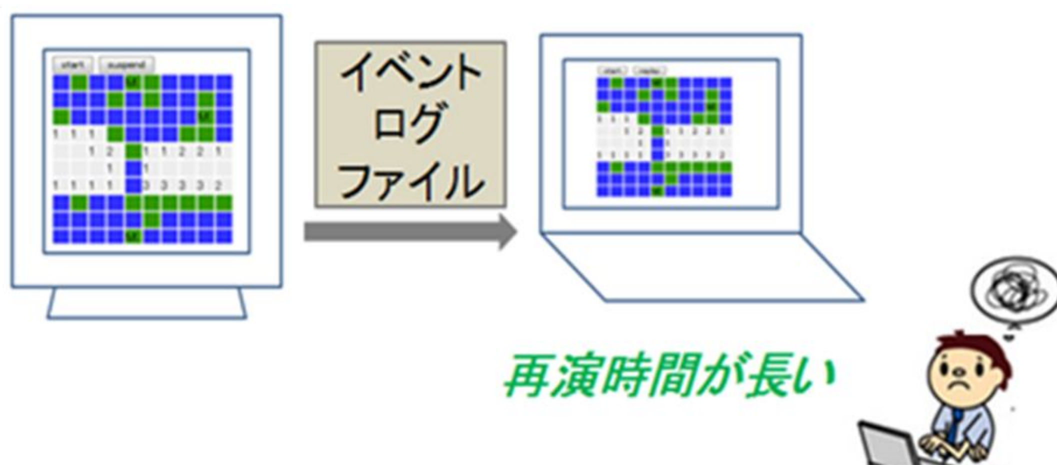

図 3.1: Mugshot を応用したプロセス移送

そこで、自動メモ化技術と併用することで、再演時間を短縮することを考えている。しかし、現実のプログラムではメモ化可能な関数は極めて少ないので、任意のプログラムを対象とするには前章で述べたメモ化の適用性の問題を解決する必要がある。

本研究では、自動メモ化を行いやすくするために、メモ化の適用範囲を拡張する手法を提案した。プログラム変換によって、大域変数への参照や代入がある関数もメモ化適用可能な関数になる。これによって、プログラマは大域変数があるかどうかを区別し印を付ける必要がなく、メモ化するための作業を安心して自動メモ化器に任せられる。また、この手法で、大域変数へアクセスする関数が多いアプリケーションに対して自動メモ化を行う場合、メモ化可能な関数が増える。

第 4 章

メモ化の大域変数問題

大域変数へアクセスする関数は一般に参照透過性を備えておらず、そのままではメモ化が適用できない。引数以外に大域変数の値も関数の計算結果に影響するので、関数が同じ引数を受け取っても異なる計算結果を返す可能性があるからである。以下に例を示す。

```
1  function f(n)
2    return n+g_var
3  end function
```

例のように、関数 f は引数 n を受け取り、n に大域変数 g_var を足して、その結果を返す。関数 f の計算結果は引数 n と大域変数 g_var の両方の値に依存する。しかし、従来のメモ化は引数だけをキーとしてテーブルに格納し、それによって計算結果を検索するので、関数がアクセスした大域変数の値がテーブルに記憶されない。このような関数にメモ化を適用すると、問題が起こる。例えば、関数 f が何回も引数 6 で呼び出されるとする。関数 f が初めて引数 6 で呼び出される際の大域変数 g_var の値が 2 ならば、 f の計算結果は 8 である。従って、引数 6 と返値 8 の組み合わせのエントリがテーブルに追加される。その後に、大域変数 g_var の値が 4 に更新されるとする。関数 f が再び引数 6 で呼び出される際、テーブルから 6 というキーを見つけるので、8 を結果として返すが、この計算結果は正しくない。正しい計算結果は 10 である。

また、大域変数への代入がある関数にメモ化を適用することも安全ではない。

```
1  function  g(n)
2    g_var=n
3    return n
4  end function
```

擬似コードのように、関数 g は引数 n を受け取り、大域変数 g_var へ n の値を代入し、それから n の値を返す。一見したところ、関数 g の計算結果は引数 n だけに依存しているので、メモ化が適用できると思われるかもしれない。しかし、関数 g を含むプログラムの全体を考えると、そうではない。関数が大域変数を参照しない場合、関数の計算結果は引数によって決められる。しかし、大域変数は全ての関数に共有されるので、次に呼び出される関数の計算結果はその大域変数の値と関わる可能性がある。例えば、関数 g が引数 5 で呼び出される時に、大域変数 g_var に 5 を代入する。次に、g_var を参照する関数 f が引数 6 で呼び出されれば、 計算結果が 11 になる。この結果は関数 f のテーブルに記憶される。その後に、関数 g が引数 10 で呼び出され、g_var に 10 を代入する。関数 f が再び引数 6 で呼出されれば、テーブルを検索し、11 を結果として返す。これは前述した大域変数の参照問題と同じになる。関数 g は関数 f の計算結果に影響する大域変数の値を書き換えるので、関数 g が呼び出された後に、関数 f が再び過去の引数で呼び出される場合、f の返値は疑わしい。

従って、大域変数への参照や代入がある関数は一般にメモ化できない関数と考える。メモ化はあくまでも最適化手段の一種である。プログラムの性能を改善することより、プログラムが正しく実行されることの方がより重要なことである。プログラムの実行速度がいくら速くなっても、もしプログラムのセマンティクスが変わってしまうと、最適化の意味がなくなる。安全にメモ化を利用するために、通常、プログラマはメモ化できる関数とメモ化できない関数に印を付け、自動メモ化器にメモ化するかどうかという指示を与える。大域変数へアクセスする関数はメモ化できない関数と見なされるので、「メモ化しない」を表す印を関数につける。

自動メモ化はその印を見つけると、関数に対するメモ化変換処理を省く。このような方法でプログラムのセマンティクスは守られる。しかし、メモ化できる関数の範囲が制限されるので、大域変数へアクセスする関数が多いプログラムに対してメモ化を適用すれば、最適化効果が予想通りにならない。

第 5 章

提案方法

従来、関数にメモ化を適用するための前提条件は関数が参照透過性を備えることであった。関数が大域変数へアクセスする場合、計算結果が大域変数にも依存しているので、メモ化が適用できない。本研究では、大域変数への参照や代入がある関数に対してプログラム変換を行う。変換後の関数は参照透過性を備え、メモ化適用可能な関数になる。

大域変数を参照する関数にメモ化を適用できない原因は、メモ化手法が関数の引数だけをキーとして記憶しておくことである。関数が呼び出される際、テーブルを検索し、引数と対応する計算結果を取り出す。もし引数のほかに計算結果に影響する要素があれば、メモ化される関数の返値は疑わしい。提案手法の基本アイデアは、大域変数の値も引数とともにテーブルに記憶しておくことである。この手法では、関数が呼び出される際、テーブルのキーを検索し、もし引数と参照された大域変数の値から成るキーを見つければ、対応する値を返せば良い。実際の提案手法では、大域変数を直接扱うことはせず、大域変数の値を受け取るパラメータを関数のパラメータリストに追加するようにして、大域変数を局所変数として扱う。

前章の関数 f を例として挙げる。

```
1  function  f(n)
2    return n+g_var
3  end function
```

関数 f を以下のように変換する。

```
1  function  f(n, g_var_l)
2    return n+g_var_l
3  end function
```

関数 f のパラメータリストには大域変数 g_var の値を受け取るためのパラメータ g_var_l が追加される。関数 f は n に g_var_l の値を加えた結果を返す。これはオリジナルの関数 f と同じセマンティクスを持っている。このように、変換後の関数 f はメモ化を適用できる関数になる。関数 f が異なる時点で引数 6 で呼び出される場合を考える。最初の呼び出しで、g_var_l の値が 2 だとすると、f の返値は 8 になる。よって、引数 6 で関数 f を呼び出すと、(6,2) をキー、8 を値としてテーブルにエントリを追加する。その後に、大域変数 g_var の値を 4 に更新したとする。関数 f が 2 回目に呼び出される際、テーブルの中に (6,4) というキーは見つからないので、テーブルに新しいエントリを追加し、10 を返す。

変換は、関数呼び出しについても行う必要がある。関数定義のパラメータ数が増えたので、関数呼び出しはそれと対応して、同じ個数の引数を受けなければいけない。関数 f が呼び出される際、大域変数 g_var を引数リストに追加する。

大域変数をパラメータとして表す考え方はラムダリフティング [7,10] を流用した。ラムダリフティングはネストした関数定義を平らにするための変換である。ラムダリフティングは自由変数を内部関数定義から取り除き、局所変数として関数のパラメータに追加し、内部関数定義をトップレベルにする手法である。このような最適化によって、内部定義を実行する時に、クロージャを生成する必要がなくなり、実行時のコストが減らせる。本研究ではメモ化の適用範囲を拡張するために、このような変換方法を流用した。

一方、関数が大域変数へ代入する場合、そのままではメモ化を適用することができない。このような関数にメモ化を適用する場合、同じ引数のキーを見つけると、過去の実行結果を返す。しかし、関数の実行は省略されているので、大域変数への代入は実際には行われない。これは次にこの大域変数を参照する関数の実行結果に

影響する可能性がある。従って、大域変数への代入を含む関数を以下のように変換する。大域変数に代入される値と本来の返値と組み合わせたペアを返す。そして、関数がリターンした後に、大域変数への代入をやり直す。前章の関数 g のコードを例として挙げる。

```
1  function  g(n)
2    g_var=n
3    return n
4  end function
```

変換後の関数 g は以下のコードになる。

```
1  function  g(n)
2    var g_var_l
3    g_var_l=n
4    return (n, g_var_l)
5  end function
```

変換後の関数 g には新しい局所変数 g_var_l が導入される。この局所変数は大域変数 g_var に相当する変数である。関数 g が大域変数 g_var の代わりに、g_var_l に n の値を代入し、それから、g_var_l と n から成るペアを返す。関数にメモ化を適用する時に、大域変数の値はテーブルに格納される。例えば、関数 g が初めて引数 5 で呼び出される場合、大域変数 g_var に 5 を代入する。関数のテーブルに引数と返値 (5,5) と組み合わせたエントリを追加する。その後に関数 g が同じ引数で呼出されれば、大域変数 g_var に代入される値が取り出せる。

関数呼び出しはいくつかの文に変換される。関数がリターンする際は、大域変数へ代入された値及び本来の返値から成るペアやリストを返すようにする。関数がリターンした後に、まず本来の計算結果を取り出す。それから、返されたペアやリストから大域変数に代入された値を取り出し、その値で大域変数を更新する。その後に呼び出される関数は大域変数の新しい値を得られる。

また、提案した変換手法は、大域変数へ直接アクセスしない関数に対しても影

響するので注意する必要がある。 例えば, 以下のプログラムに対して変換を行う場合を考える。

```
1  function g(y)
2    h_var=y
3    return g_var+y
4  end function
5
6  function f(x)
7    var res=g(x)
8    return res
9  end function
```

関数 f は関数 g を呼出す。 関数 f は直接大域変数へアクセスしない。しかし、 関数 g が大域変数 g_var と h_var へアクセスするので、関数 f は間接的に g_var と h_var へアクセスすることになる。 そのために、関数 f に対して、 以下の変換を行う。

```
1  function g(y, g_var_l)
2    var h_var_l=y
3    return (g_var_l+y, h_var_l)
4  end function
5
6  function f(x, g_var_l)
7    var res=g(x, g_var_l)
8    var h_var_l=get value  of h_var from res
9    return (get original result from res, h_var_l)
10 end function
```

関数 f のパラメータリストに局所変数 g_var_l を追加し、 関数 g の返値から大域変数 h_var に代入しようとする値を取り出す。その値を関数 f の局所変数 h_var_l に代入し、 それから、 h_var_l と本来の返値 res によって作られたペアを返す。このような変換で、 大域変数 g_var と h_var の値は各関数の間に伝わる。 変換後の関

表 5.1: 関数 f と g のためのテーブル

キー	値
g, y, g_var_l	g_var_l+y, h_var_l
f, x, g_var_l	res, h_var_l

数に対してメモ化を適用する時に、大域変数 g_var と h_var の値は引数として関数に渡されるので、テーブルに記憶される。また、大域変数 h_var に代入された値が返されるので、それもテーブルに記憶する。このプログラムによるテーブルの内容を表 5.1 で示す。

第 6 章

設計

本研究で提案した手法は、大域変数への参照や代入がある関数を、プログラム変換器によって、参照透過性を備える関数へ変換する。このような変換によって、メモ化の適用範囲が広がり、自動メモ化が行いやすくなる。

前章で大域変数へアクセスする関数がメモ化できない原因について説明した。そして、その問題を解決するためのプログラム変換方法を提案した。実際、プログラム変換は前章で挙げた例のように簡単ではない。プログラムには様々なパターンがある。プログラム変換はそれに応じて行われる。変換器はプログラム解析とプログラム変換の二つの段階を含む。プログラム解析の段階でプログラムを安全に変換するための大域変数情報を集める。それから、集めた情報に基づいて、プログラム変換を行う。

6.1 フロー解析

6.1.1 大域変数情報の収集

プログラム変換はプログラムのセマンティクスを守らなければならない。本研究では、大域変数へアクセスする関数を安全にメモ化が適用できる関数へ変換するために、変換対象とされる関数の大域変数情報を集める必要がある。大域変数情報は関数が参照する大域変数の集合と代入される大域変数の集合を含んでいる。以後では、大域変数情報を（参照した大域変数の集合　代入した大域変数の集合）

という形で表す。最も保守的なやり方はプログラムに含まれる大域変数全体を関数の大域変数情報として見なすことである。しかし、このような方法で集めた大域変数集合は関数が実際にアクセスした大域変数集合より広い。プログラムの大域変数が非常に膨大であれば、変換後の関数にメモ化を適用すると、テーブルにエントリを追加するコストとキーを検索するコストが高くなる。また、メモリが多く消費される。従って、本研究では、このような手法を採用しない。集めた関数の大域変数情報がプログラムのセマンティクスにとって安全であり、また、余分な大域変数を含まないことが望ましい。

具体的に情報を集める方法について説明するために、以下のプログラムを例としてあげる。

```
1  function test (n)
2   var i = 0
3   while i<5000000 do
4     g_var = i + n * h_var
5     i = i + 1
6   end while
7   return g_var
8  end function
```

関数 test には大域変数への参照と代入が両方ある。変換器は test の関数定義を読み取ると、順に式を解析する。g_var = i + n * h_var 式に到達する時点で初めて関数 test の大域変数を見つける。test の関数定義を全部解析すると、参照した大域変数が h_var、代入した大域変数が g_var という情報が得られる。この簡単なプログラムに対して、((h_var)(g_var))という情報に基づき、安全に変換することができる。

また、次のプログラムを考える。

```
1  function g2 (y)
2    return g_var + h_var + y
```

```
3   end function
4
5   function  f2 (x)
6     g2 (3)
7   end function
```

このプログラムでは、関数 f2 の中で関数 g2 が呼出される。この二つの関数に対して解析する。関数 g2 は大域変数 g_var と h_var を参照する。関数 g2 に対する解析結果は（(g_var h_var)（　)）である。二つ目の集合が空になる原因は関数 g2 が大域変数へ代入しないことからである。次に、関数 f2 に対して解析する。関数は大域変数へ直接アクセスしないので、最後に得る大域変数情報は（(　)（　)）になる。一見して、この解析は正確だと思われる。しかし、この情報はプログラム変換にとって安全ではない。この情報に基づき、関数 f2 と g2 を変換すれば、関数 g2 のパラメータに大域変数 g_var と h_var の値を伝えるためのパラメータが追加される。関数定義に応じて、関数 f2 の本体にある関数 g2 の呼び出しは二つの大域変数を引数として追加する。一方、関数 f2 が大域変数を参照していないと思われるので、プログラム変換を行わない。このような変換は正しくない。g_var と h_var の値が関数 f2 と g2 に両方伝わる必要がある。セマンティクスに対して安全な情報を求めるために、以下の等式を満たす必要がある。

{ 関数が参照する大域変数 }={ 関数内で直接参照する大域変数 } ∪{ 呼び出される関数が参照する大域変数 }

{ 関数が代入する大域変数 } = { 関数内で直接代入される大域変数 } ∪ { 呼び出される関数が代入する大域変数 }

即ち、解析によって直接アクセスする大域変数と間接的にアクセスする大域変数の両方を集める必要がある。この二つの等式に従って、関数 f2 の正しい大域変数情報は関数 g2 の呼び出しによって間接に参照した大域変数 h_var を含む。変換後の関数 f2 は以下のようになる。

```
1 function  f2 (x, g_var_l , h_var_l)
2   g2 (3 g_var_l h_var_l)
3 end function
```

関数 f2 のパラメータリストに大域変数 g_var と h_var の値を伝えるための局所変数を追加する。このような変換は安全であり、変換後の関数 f2 がメモ化可能な関数になる。

また、以下の相互再帰呼び出し関数の大域変数情報について考える。

```
1  function  f (x)
2    return g(x) + f_var
3  end function
4
5  function  g (x)
6    return h(x) + g_var
7  end function
8
9  function  h (x)
10   return w(x) + h_var
11 end function
12
13 function  w (x)
14   return f(x) + w_var
15 end function
```

この四つの関数同士は再帰的な呼出しを行っている。関数がアクセスする大域変数の集合を FUN-INFO で表せば、大域変数情報は以下の関係を満たす。

$\text{FUN-INFO_f} = \text{FUN-INFO_g} \cup \{\text{f_var}\}$

$\text{FUN-INFO_g} = \text{FUN-INFO_h} \cup \{\text{g_var}\}$

$\text{FUN-INFO_h} = \text{FUN-INFO_w} \cup \{\text{h_var}\}$

FUN-INFO_w = FUN-INFO_f ∪{w_var}

関数 f の大域変数情報は関数 g の情報に依存する。関数 g が関数 h の情報に依存し、同様に関数 h は関数 w の情報 に依存し、関数 w は関数 f の情報に依存する。結局、各関数に対して大域変数情報を集めることは実際無限のループになるので、各関数の大域変数集合は決められない。そこで次節のようにして求める。

6.1.2 不動点を求める

再帰呼び出し関数の大域変数情報を得ようとすると、プログラム解析が無限ループになるので、解析を止めるタイミングを決める必要がある。前の例を考えてみる。一回目の解析では、関数 f が大域変数 f_var を参照したので、FUN-INFO_f に f_var を追加する。各関数の大域変数情報は本体で直接アクセスした大域変数以外に、呼び出した関数がアクセスした大域変数も含む。しかし、 関数 g の大域変数情報はまだ分からないので、関数 f の情報集めは不完全である。次の関数 g、h、w も同じように解析される。関数 g については、FUN-INFO_g に g_var が追加され、関数 h については、FUN-INFO_h に h_var が追加される。関数 w についてはFUN-INFO_w に w_var が追加される。この時点で関数 f が大域変数 f_var を参照することが分かったので、関数 w の FUN-INFO_w に f_var も追加する。しかし、集めた情報はプログラム変換にとってまだ不完全であるので、2 回目の解析が必要である。2 回目の解析では、関数 g が大域変数 g_var を参照することは既に分かっているので、FUN-INFO_f に g_var が追加される。関数 h が h_var を参照するので、FUN-INFO_g に h_var が追加される。同じように、FUN-INFO_h に w_var と f_var が追加される。そして、FUN-INFO_wには g_var が追加される。2 回目で集めた大域変数情報は一回目で集めた情報より増えた。

解析はまだ止められない。続いて 3 回目と 4 回目の解析を行う。4 回目で四つの関数の大域変数情報はどれでも {f_var g_var h_var w_var} となる。その後に行った解析の結果は 4 回目と完全に同じ情報を得る。この時点で集めた大域変数情報は完

表 6.1: 大域変数情報の収集

	解析一回目	解析二回目	解析三回目	解析四回目
f	{f_var}	{f_var g_var}	{f_var g_var h_var }	{f_var g_var h_var w_var }
g	{g_var}	{g_var h_var}	{g_var h_var w_var f_var }	{g_var h_var w_var f_var }
h	{h_var}	{h_var w_var f_var }	{h_var w_var f_var g_var }	{h_var w_var f_var g_var }
w	{w_var f_var }	{w_var f_var g_var }	{w_var f_var g_var h_var }	{w_var f_var g_var h_var }

全であり、プログラム変換にとって安全であるため、フロー解析を止める。最後の解析結果は、関数 f、g、h、w の大域変数情報が {f_var g_var h_var w_var} になる。

上記の例では、解析と入力プログラムの不動点 [5] を見つけた。プログラムの大域変数集合は有限である。毎回の解析で集めた大域変数情報は前回より増えるが、必ず何回目かの解析で集めた大域変数情報が前回と同じになる。この時点が解析の不動点である。

大域変数を集めるアルゴリズムは以下通りである。

入力 ： 解析対象となるプログラム

出力 ： 変換するための大域変数情報

メソッド ：

STEP1 :初期化する

```
FUN-INFO := null
```

STEP2 :プログラムを解析

```
while FUN-INFO_after not equal FUN-INFO_before
do FUN-INFO_after := FUN-INFO_before
STEP2 に飛ぶ
```

アルゴリズムの入力は解析対象となるプログラムである。前の例でいえば、関数 f、g、h、w の定義を入力として受け取り、この四つの関数に対してフロー解析を行う。ステップ1は FUN-INFO の初期化処理である。各関数の大域変数情報 FUN-INFO の初期値は空である。ステップ2は繰り返し処理である。もし解析で集めた大域変数集合が解析前の FUN-INFO と異なれば、FUN-INFO を新しい大域変数集合で更新する。それから、もう一度各関数を解析する。もし、解析前後の FUN-INFO が変化しなければ、このアルゴリズムと入力プログラムの不動点は見つかったことになる。次に、FUN-INFO を出力すれば済む。

6.2 プログラム変換

大域変数情報を集めた後に、プログラム変換が行われる。本手法ではソースコードレベルにおいて、プログラム変換を行う。前節で集めた各関数の大域変数情報 FUN-INFO に基づき、異なるパターンの関数に応じて変換する。

プログラム変換の仕方について説明する。関数 F の解析結果は、大域変数情報 FUN-INFO_F が $\{\{G_1...G_m\}\ \{G'_1...G'_n\}\}$ となる。$\{G_1...G_m\}$ は関数 F が直接や間接的に参照した大域変数集合である。$\{G'_1...G'_n\}$ は関数 F が直接や間接的に代入した大域変数集合である。

大域変数へ参照する時の変換規則は以下の通りである。変換器は関数定義と関数呼出しを両方変換する必要がある。

[関数定義の変換]

パラメータリスト $(P_1, ..., P_t) \ \triangleright (P_1, ..., P_t, L_1, ..., L_m)$

参照される大域変数 $G_i \triangleright L_i\ \{i \mid i \in \{1, ..., m\}\}$

[関数呼び出しの変換]

引数リスト $(\mathrm{Arg}_1, ..., \mathrm{Arg}_t)$ $\triangleright$ $(\mathrm{Arg}_1, ..., \mathrm{Arg}_t, \mathrm{G}_1, ..., \mathrm{G}_m)$

関数定義の変換部分は2か所がある。一つは関数定義のパラメータリストに大域変数の値を伝えるためのパラメータを追加することである。元の関数Fのパラメータリストが $(\mathrm{P}_1, ..., \mathrm{P}_t)$ である。変換後のパラメータリストは $(\mathrm{P}_1, ..., \mathrm{P}_t, \mathrm{L}_1, ..., \mathrm{L}_m)$ になる。また、関数が参照する大域変数Giはそれと対応する局所変数 L_i で置き換える。関数呼出しは関数定義のパラメータリストに応じて、引数リストに大域変数 G_i を追加する。

大域変数への代入がある関数は以下のような変換規則を持っている。

[関数定義の変換]

局所変数 $\mathrm{L'}_1 ... \mathrm{L'}_n$ を宣言

代入される大域変数 $\mathrm{G'}_j \triangleright \mathrm{L'}_j \ \{ j \mid j \in \{1, ..., n\}\}$

関数の返値 Ret $\triangleright$ $(\mathrm{Ret}, \mathrm{L'}_1, ..., \mathrm{L'}_n)$

[関数呼び出しの変換]

$$\mathrm{F}(\mathrm{Arg}_1, ..., \mathrm{Arg}_t) \quad \triangleright \begin{cases} \text{get result part of } \mathrm{F}(\mathrm{Arg}_1, ..., \mathrm{Arg}_t) \\ \mathrm{G}'_\mathrm{j} := \text{ get value of } \mathrm{L'}_j \ \{\mathrm{j} \mid \mathrm{j} \in \{1, ..., n\}\} \end{cases}$$

大域変数を参照する関数に対する変換と同じように、関数定義と関数呼出しを両方変換する必要がある。関数定義では、局所変数 $\mathrm{L'}_1 ... \mathrm{L'}_n$ の宣言が追加される。大域変数 G_j の代わりに、局所変数 $\mathrm{L'}_j$ へ代入を行う。また、大域変数に代入される値を伝えるために、関数の返値を $\mathrm{Ret}, \mathrm{L'}_1, ..., \mathrm{L'}_n$ というリストにする。一方、関数の呼び出し $\mathrm{F}(\mathrm{Arg}_1, ..., \mathrm{Arg}_t)$ はいくつの文に変換される。まず、本来の計算結果を関数Fの返値から取り出す。そして、関数Fの返値から $\mathrm{L'}_1 \cdots \mathrm{L'}_n$ の値を取り出し、その値を対応する大域変数 $\mathrm{G'}_1 ... \mathrm{G'}_n$ に代入する。このような変換によって、大域変数の値は関数の間に伝わる。

関数に大域変数への参照と代入の両方がある場合が多いので、変換器は二つの変換規則を併用して、プログラムを変換する。関数 test に対する変換を例として

挙げる。

```
1  function  test (n)
2    var i = 0
3    while i<5000000 do
4      g_var = i + n * h_var
5      i = i + 1
6    end while
7    return i
8   end function
```

[関数 test の定義に対する変換]

パラメータリスト (n) $\triangleright$ $(n, \text{h_var_l})$

局所変数 g_var_l を宣言

g_var = i + n * h_var $\triangleright$ g_var_l = i + n * h_var_l

返値 i $\triangleright$ (i, g_var_l)

[呼出し test(6) に対する変換]

引数リスト（6） $\triangleright$ (6, h_var)

get original result part from test(6, h_var)

g_var = get h_var's value from test(6, h_var)

関数 test は以下通りに変換される。

```
1  function  test (n, h_var_l)
2    var g_var_l
3    var i = 0
4    while i<5000000 do
5      g_var_l = i + n * h_var_l
6      i = i + 1
7    end while
8    return (i, h_var_l)
9   end function
```

変換後の関数は直接、大域変数への参照や代入をせず、その代わりに、各大域変数に相当する局所変数に対して参照や代入の処理を行う。このような変換によって、変換後の関数は参照透過性を備え、メモ化を適用できるようになる。

第 7 章

実装

本研究では、前章で述べたプログラム変換器を実装した。実装言語は Scheme[2] である。Scheme の式は前置記法のリストとして書かれる。つまり、最初の要素はオペレータであり、リストの残りは引数である。このような記述方法はプログラミング言語の構文木と同じである。Scheme が他の言語と比べ、ソースプログラムが構文木そのものとも言える。従って、Scheme プログラムの構文解析は容易である。また、プログラムをデータとして扱えるので、変換器は比較的に使い易い。本研究の変換器は source–to–source の変換であり、DrScheme 言語処理系 [8] で実装した。

変換器はプログラム解析処理と変換処理を両方行うプログラムであり、図 7.1 で示すフローチャートのように処理する。この図には自動メモ化を行う処理は含まれていない。まず、Scheme のソースコードを読み取り、変換器の大域変数 CONTENTS に格納する。入力が完成した後に、CONTENTS に格納される各関数定義と関数呼出しを順に解析する。大域変数情報を全部集めた後に、次のパスで CONTENTS に格納されるソースコードに対して変換を行う。

7.1 準備

本研究のプログラム変換はプログラム中の文字列を、コンテキストを無視して、あらかじめ定義された規則に従って機械的に置換するものではない。変換器は関数の大域変数の情報を集めるために、プログラムの構文を解析し、変数の解析時

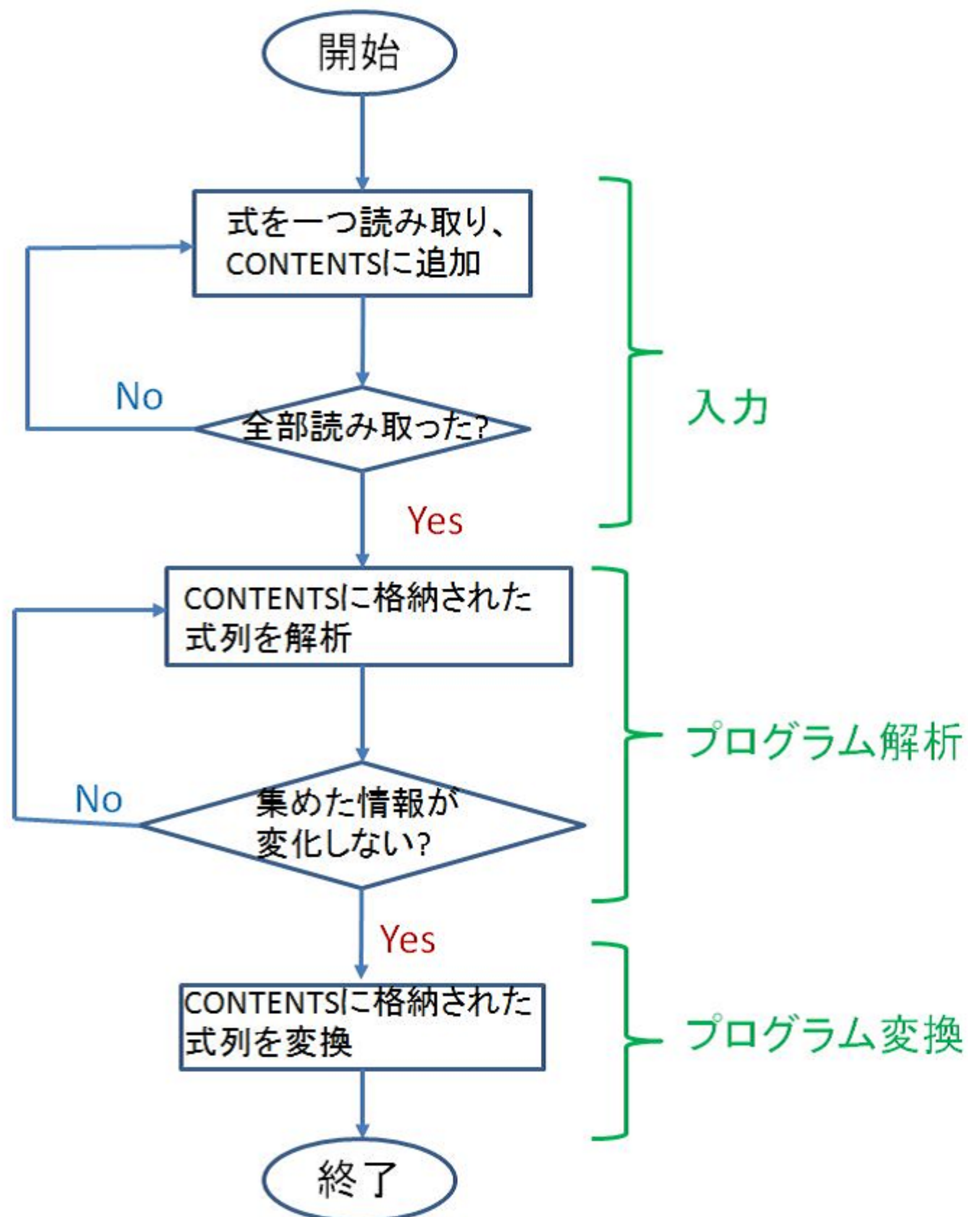

図 7.1: 変換器処理のフローチャート

環境を構築する必要がある。まず、Scheme 言語の構文について紹介する。

7.1.1 Schemeの構文

変換器に入力されるSchemeプログラムはデータとして解析される。Schemeプログラムは式、定義、及び構文定義の列から成る。各式はカッコで括られた前置記法を使用し、式の型は以下のように分類される。

＜式＞ ::= ＜変数＞
｜＜リテラル＞
｜<代入式>
｜＜条件式＞
｜＜ラムダ式＞
｜＜派生式＞
｜＜手続き呼び出し＞
｜＜マクロ使用＞
｜＜マクロ・ブロック＞

派生式は原始的ではなく、マクロとして定義される式である。例えば、let式、cond式とbegin式等は派生式である。本実装では、Scheme変換器を単純にするために、Scheme言語から重要な機能の一部を取り除いた。マクロの使用、内部定義及び高階関数の使用は本稿で議論しない。プログラム変換器はSchemeの手続きとして実装され、プログラムと環境を引数として受け取る。式の種類に応じて処理を振り分ける。

7.1.2 環境構造

プログラムを解析するために、環境を構築する必要がある。変換器の環境はSchemeの超循環評価器[6](インタプリタ)の環境と異なり、変数と値の束縛の並びではなく、変数情報の並びで構築されるデータ構造である。それによって、変換器は関数の中に出現した変数が大域変数であるかどうか等を判断する。また、環

境に格納された関数の大域変数情報に基づき、プログラム変換を行う。本章では、変数名と変数情報の対応付けを束縛と呼ぶ。変換器が lambda 式や let 式を解析する際、環境を拡張する。パラメータや局所変数の束縛の並びからなるフレームを環境に追加する。環境のフレームを遡って調べることによって、変換器は関数の本体に現れた変数が大域変数と局所変数とどちらかが分かる (図 7.2 を参照)。もし lambda 式や let 式が設けるフレームの中に、 変数の束縛を見つければ、この変数は局所変数と見なされる。変数の束縛を見つけることができない場合、外側の環境フレームに辿り、束縛を探す。トップレベルの環境で変数の束縛を見つける場合、この変数は大域変数と見なされる。もし環境の中に束縛がなければ、この変数は束縛されないと認められる。本稿では、述べている環境は大域関数の環境であり、内部関数定義及び let 式による定義される局所関数の環境について議論しない。環境モデルは図 7.2 で示す。

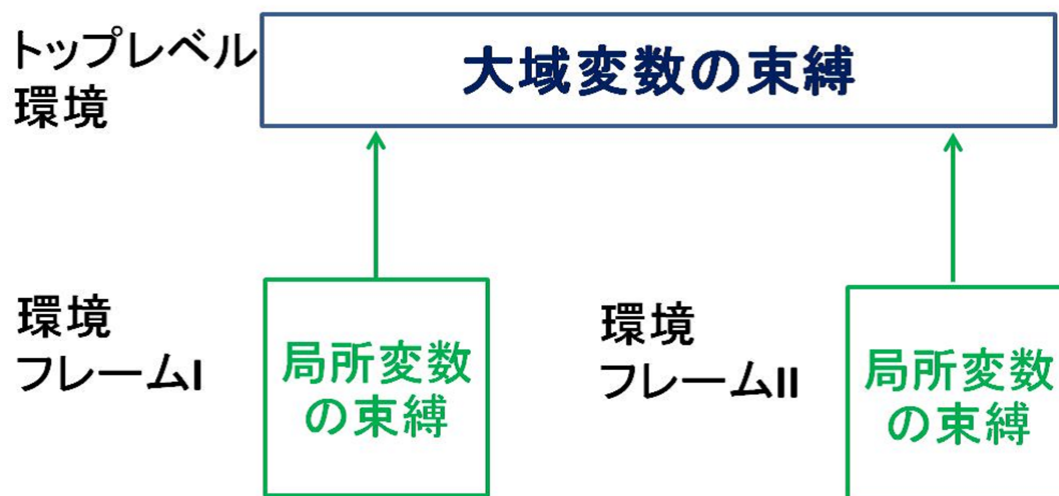

図 7.2: 変換器の環境モデル

変換器の環境構造について、以下の例で説明する。

```
1  function g (y)
2    h_var = y
```

```
3     return g_var + y
4   end function
5
6   function f (x)
7     var res = g (x)
8     return res
9   end function
```

このプログラムの環境構造を図 7.3 に示す。

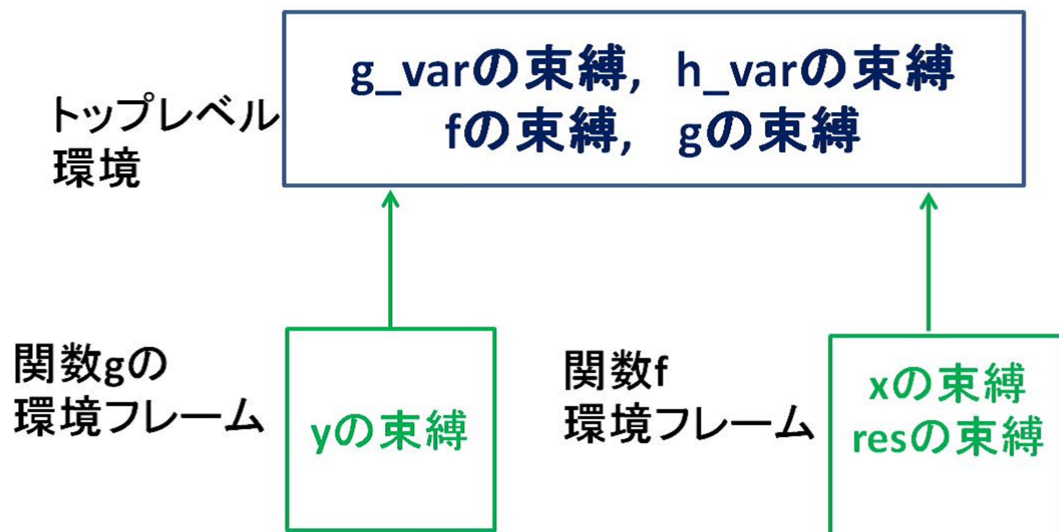

図 7.3: 単純な環境構造

トップレベルの環境は大域変数 g_var、h_var、f 及び g の束縛を含む。矢印は環境へのポインタである。関数 g を解析する際、まずパラメータ y の束縛から成るフレームを環境に追加する。このフレームはトップレベル環境へのポイントを持っている。関数 f を解析する際、パラメータ x と局所変数 res の束縛から成るフレームを環境に追加する。このフレームもトップレベル環境へのポインタを持っている。関数 f と g の本体を解析する時に、各フレームで y と x の束縛が見つかるので、変数 y と x は関数の局所変数と見なされる。g_var と h_var は関数 f と g のフレームで

見つからないので、フレームのポインタを辿り、外側の環境へ進む。トップレベル環境の中に g_var と h_var の束縛が見つかるので、この二つの変数は大域変数と見なされる。

関数に対する解析が終了すれば、関数のフレームを環境から外す。外すべき時は外さないと、間違った環境を作って、スコープに存在しない変数が見えてしまうことになる。

7.2 プログラム解析

本研究では、変換器はプログラム解析とプログラム変換に分離した。入力されたプログラムを解析し、集めた各関数の大域変数情報によって、プログラム変換を行う。以後、入力として変換器に読み込まれたソースコードと区別するために、変換器のプログラムをイタリックで表示する。

translator はトップレベルの関数であり、本体で振り分け処理を行う関数 *tran_exp* を呼出す。*tran_exp* は以下のコードである。

```
1  (define (tran_exp exp env)
2     (cond ((self-evaluating? exp) (t_self_evaluating exp))
3           ((var? exp) (t_lookup_var exp env))
4           ((let? exp) (t_let exp env))
5           ((set? exp) (t_set! exp env))
6           ((define? exp) (t_define exp env))
7           ((if? exp) (t_if exp env))
8           ((cond? exp) (t_if (cond->if exp) env))
9           ((lambda? exp) (t_lambda exp env))
10          ((begin? exp) (t_begin exp env))
11          ((primary_fun? exp) (t_primary_fun exp env))
12          ((funcall? exp) (t_funcall exp env))))
```

tran_exp は *cond* 式を使い、式の構文型につき、場合分けて解析する。式の各型

表 7.1: 変数情報

< VAR_NAME >	変数名
IF_REF	変数が参照されたかどうか
IF_ASSIGN	変数が代入されたかどうか
IF_CLOSED	変数が関数に閉じ込められたかどうか
IF_LIFT	関数がパラメータに変数を追加する必要があるか
LIFT_VARS	パラメータに追加される変数集合
IF_RET	関数が大域変数の値を返す必要があるか
RET_VARS	返値として返される変数集合

に応じて、対応する処理関数を呼出す。“？”が最後に付けられる関数は式の型をテストする述語である。“ t_ ”が関数名の先頭に付けられるのは解析と変換処理を行う関数である。例えば、*primary_fun?* は式が基本関数呼出しであるかどうかを判定する。真であれば、*t_primary_fun* を呼び出す。この関数は基本関数呼出しの式に対して、プログラム解析と変換を行う。そうでなければ、続いて他の種類の式であるかテストを行う。このような実装方式は変換器が認識可能な新しい種類の式を追加することが容易にできる。

7.2.1 変数情報

変換器は、変数の情報を以下のようなリストで表し、これと変数名からなる束縛を環境に登録する。

(< VAR_NAME > IF_REF IF_ASSIGN IF_CLOSED IF_LIFT LIFT_VARS IF_RET RET_VARS)

それぞれの情報がどんな意味を表しているかを表 7.1 に示す。

変数情報を説明するために、以下の例を挙げる。

```
1  (define (g y)
2    (set! h_var y)
3    (+ g_var y))
```

このプログラムを解析する際、変数 y、h_var、g_var 及び g に情報を付ける。IF_REF 、IF_ASSIGN 、IF_CLOSED は関数の中に出現した変数ための専用情報である。例えば、変数 y の情報は（y #t #f #f #f ( ) #f ( )）である。ここで#t と#f はそれぞれ真と偽を表す。変数 y は関数 g の中で参照されるので、２番目の IF_REF を#t にする。変数 h_var の情報は (h_var #f #t #t #f ( ) #f ( )) である。変数 h_var は set!式によって、y の値が代入されるので、３番目の IF_ASSIGN を #t にする。また、h_var は関数 g に閉じ込められた大域変数であるので、4 番目の IF_CLOSED を＃ｔにする。変数 g_var は参照される大域変数であるので、情報は (g_var #t #f #t #f ( ) #f ( )) である。また、IF_LIFT、LIFT_VARS、IF_RET、RET_VARS は関数オブジェクトにバインドされる変数ための専用情報である。例えば、関数オブジェクトにバインドされた変数 g の情報は（g #f #f #f #t (g_var) #t (h_var)）である。関数 g は大域変数 g_var を参照し、大域変数 h_var へ代入するので、IF_LIFT と IF_RET は#t にする。即ち、関数 g に対して、参照された大域変数をパラメータに追加し、代入された大域変数を返値として返す必要があると示す。また、関数 g の LIFT_VARS と RET_VARS には参照された大域変数の集合と代入された大域変数の集合を入れる。そして、関数に対して、LIFT_VARS と RET_VARS に含まれる大域変数集合に基づき、変換を行う。

変換器が lambda 式や let 式を解析する際、関数 *make-var* を呼び出し、上記の情報を含む変数束縛を作る。*make-var* は以下のコードである。

```
1  (define (make-var name)
2       (if (symbol? name)
3           (if (member name  '(if begin lambda let set!
               define ......))
```

```
4                 (write  "this is not a variable ")
5                 (list name #f #f #f #f '() #f '())))))
```

make-var は記号を受け取り、もし記号が if、begin、lambda 等でなければ、変数構造が構築され、デフォルトの値 (＜VAR_NAME＞ #f #f #f #f ( ) #f ( )) で初期化された後に、環境に追加される。その後に、変換器は関数の本体を解析する。もし変数が参照されれば、IF_REF を#t に変更する。変数が代入されば IF_ASSIGN を#t に変更する。もし、変数が大域変数であれば、IF_CLOSED を#t に変更する。関数に対する解析が終了したところで、関数を表す変数に大域変数情報を付け加え、6.1 章で述べたフロー解析による集めた大域変数集合で LIFT_VARS や RET_VARS を更新する。

7.2.2 処理関数

変換器は式の種類に応じた処理関数を呼び出す。処理関数は環境に変数の束縛を追加したり、環境を辿って変数情報を調べたりする等の解析処理、及び次の節で述べる変換処理を行う。まず、いくつかの処理関数の実装で利用した fluid-let マクロについて説明する。

fluid-let **マクロ**

Scheme は静的スコープを持つプログラミング言語である。動的スコープの変数をエミュレートするには fluid-let マクロを利用する。fluid-let の使い方を説明するために、通常の局所変数を定義するための let 式と比べる。

```
(let ((<変数 1> <初期値 1>) ... (<変数 m> <初期値 m>))
      <式 1>...<式 n>)
(fluid-let ((<変数 1> <初期値 1>) ... (<変数 m> <初期値 m>))
      <式 1>...<式 n>)
```

let 式と fluid-let 式の構文は以上の通りである。let 式では、変数 1 から変数 m までが局所変数の名前である。評価されると、まず＜初期値 1＞ から＜初期値 m＞ までが評価され、それらの値に、対応する変数がバインドされる。その後で let 式の本体である＜式 1＞ から＜式 n＞ までが順に評価される。let 式の本体が＜変数 1＞... ＜変数 m＞ のスコープである。let 式は静的スコープを持つので、本体から呼び出された関数の本体はこれらの変数のスコープに含まれない。fluid-let 式は let 式と同じ形式を持っている。異なるのは、fluid-let では、新しいバイディングを作っておらず、既存のバイディングを一時的に新しい値で書き換えることである。fluid-let の本体から呼び出された関数の本体も＜変数 1＞... ＜変数 m＞ のスコープに含まれる。

let 式と fluid-let 式を利用した例を挙げる。

```
(define time 1)

(define (f x)
   (let ((time 16))
      (g x)))

 (define (g y)
     (set! time y)
      time)

 > (f 6)
 6
 > time
 6
```

関数 f は一般的な let 式を利用した。この let 式では、大域変数 time と同名の局所変数を定義する。let 式の本体では関数 g を呼出し、関数 g は変数 time の値を変える。関数 g の中に time という名前の局所変数が定義されないので、この time は大

域変数 time を指す。よって、大域変数 time が更新されて、返されるので、引数 6 で関数 f を呼出すと、大域変数 time の値が 6 で更新される。その後に大域変数 time への参照があれば、6 という値が得られる。次に fluid-let を利用した例を示す。

```
1   (define time 1)
2
3   (define (f x)
4      (fluid-let ((time 16))
5         (g x)))
6
7    (define (g y)
8        (set! time y)
9         time)
10
11   > (f 6)
12   6
13   > time
14   1
```

このプログラムでは、関数 f は let の代わりに fluid-let を利用した。この例では、関数 f は大域変数 time のバインディングを一時的に変更し、それから、関数 g を呼出す。g は大域変数 time の値を更新する。しかし、fluid-let の実行を終えると、time のバインディングは元に戻るので、大域変数 time の値は依然として 1 である。これは本来の Scheme が備えない機能である。fluid-let マクロの実装によって、動的スコープをエミュレートする。変換器の *t_lambda* 関数と *t_let* 関数は fluid-let マクロを利用した。次の節から各処理関数について説明する。

t_lambda 関数

この関数の引数は lambda 式を表すリストである。lambda 式は次の形をしている。

```
(lambda (<パラメータリスト>) <式 1> ... <式 n>)
```

変換器の大域変数*env*はソースコードに含まれた大域変数の束縛のリストである。*t_lambda* 関数は *fluid-let* マクロを利用し、大域変数 *env* に一時的に環境フレームを追加する。まず、*cons* で *env* の先頭に *CB* という、大域関数の本体の環境と外側の環境の境界を示す印を付ける。この記号によって大域変数と関数の局所変数は区別される。本研究では、内部関数定義及び let 式で定義する局所関数を含む関数が変換器の変換対象から除かれるので、*CB* は局所関数の本体の環境と外側の環境の境界を示さない。次に、7.2.1 節の項で述べた *make-var* 関数で＜パラメータリスト＞に含まれる変数の束縛を作り、それを環境に追加する。lambda 式を解析する時の環境を図 7.4 に示す。

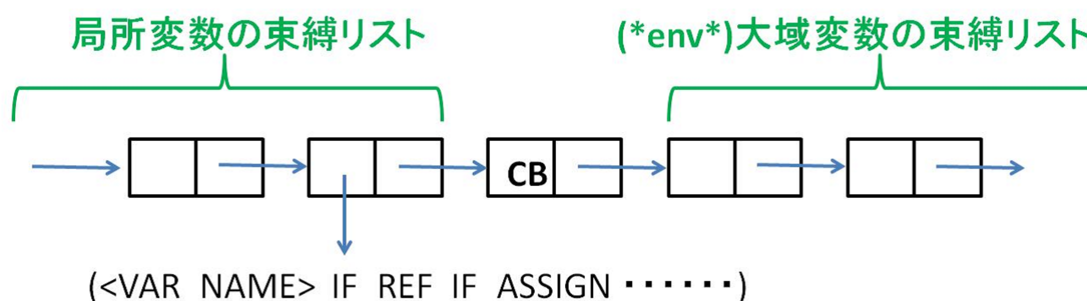

図 7.4: lambda 式を解析する際の環境

CB の左側は関数の局所変数の束縛リストであり、右側は大域変数の束縛リストである。lambda 式の本体に対する解析はこの環境の下で行われる。lambda 式の解析が終わり、*fluid-let* のスコープを抜けると、*env*が lambda 式に入る前の環境に戻る。lambda 式の本体を解析する時には、各式に対応する処理関数を呼び出す。もし式が変数、set!式あるいは let 式であれば、次の *t_lookup_var*、*t_set!* あるいは *t_let* 関数が呼び出される。

t_lookup_var 関数

変換器は lambda 式の本体に現れる変数を解析する時に、この関数を呼出す。*t_lookup_var* は環境を辿り、変数の束縛を探す。フレームの先頭から変数の束縛リストを走査し、もし *CB* に辿る前に変数を見つければ、この変数は関数の局所変数と見なされる。*CB* を越えた後に変数を見つければ、この変数が参照された大域変数だと判断する。

*t_set!*関数

この関数が処理する set!式の構文は以下の通りである。

```
(set! <変数名> <式>)
```

set!式の解析では、まず、変数名と式が解析される。*t_lookup_var* 関数と同じように環境の先頭のフレームから変数の束縛リストを走査する。*CB* に辿り着く前に代入される変数の束縛が見つかれば、この変数は局所変数である。そうでなければ、この変数が大域変数であることが分かる。

t_let 関数

この関数は let 式を表すリストを処理する。let 式は以下の構文を持つ。

```
(let ((<変数 1> <初期値 1>) ... (<変数 m> <初期値 m>))
      <式 1>...<式 n>)
```

t_let 関数の処理は *t_lambda* と似ている。まず、let 式の各初期値の式と対応する処理関数が呼び出され、初期値の式を解析した後に、環境を let 式の局所変数の束縛で拡張する。そして、拡張された環境の下で let 式の本体の各式を解析する。

他の処理関数

上の四つの関数以外に、*t_self_evaluating*、*t_define*、*t_if*、 *t_cond*、 *t_begin*、*t_primary_fun*、 *t_funcall* 等の関数を実装した。これらの関数は自己評価式、定義式、条件式、逐次式、関数呼出しを解析する時に呼び出される。

- 自己評価式

  本実装では、*t_self_evaluating* 関数は整数、文字列及びリテラルに対して処理を行う。これらの式は変換せずにそのまま出力する。

- 定義式

  Scheme 言語の定義式の構文は以下通りである。

```
(define <変数名> <式>)
(define (<関数名>  <変数 1> ... <変数 m>)
        <式 1> ... <式 n>)
(define <関数名>
        (lambda (<パラメータリスト>)  <式 1> ... <式 n>))
```

  define 式で変数及び関数を定義できる。変数定義の define 式を解析する際、変数に束縛される式を再帰的に解析する必要がある。関数定義の define 式は 2 種類がある。lambda 式を使わない形と使う形がある。*t_define* はまず lambda 式を使わない define 式を lambda 式に変換し、そして、*t_lambda* 関数を呼び出して、lambda 式を解析する。

- 条件式

  実装した変換器は if と cond の二種類の条件式を処理できる。Scheme の if 式と cond 式の構文は以下の通りである。

```
(if <条件部> <帰結部> <代替部>)
(if <条件部> <帰結部>)
(cond (<条件1> <式> ... <式>)
      (<条件2> <式> ... <式>)
      ...
      (<条件n> <式> ... <式>))
```

cond 式は if 式の派生式であるので、まず、変換器の *t_cond* 関数は cond 式を if 式に変換する。そして、変換器は *t_if* 及び if 式の条件部、帰結部及び代替部に対応する処理関数を呼び出し、if 式の各部分を解析する。

- 逐次式

  逐次式の構文は (begin ＜式 1 ＞ ＜式 2 ＞　...) である。*t_begin* は begin 式の各式を解析する。

- 関数呼出し

  関数呼出しの式は一般に (＜関数名＞ ＜式 1 ＞ ... ＜式 n ＞) の形をしている。本来は (＜式＞ ＜式 1 ＞ ... ＜式 n ＞) であるが、本研究では大域関数しか想定しないので、＜関数名＞と同じ名前の変数の値を取り出し、その値である関数を呼び出すということとする。*t_funcall* は呼び出されると、＜式 1 ＞から＜式 n ＞までに対して、対応する処理関数を呼び出して、各式を解析する。

7.2.3 解析例

解析過程を説明するために、以下の例で説明する。

```
1  (define g_var 10)
```

```
2   (define h_var 20)
3
4   (define test
5    (lambda (n)
6     (let ((i 2))
7        (if (>= n 10)
8            g_var
9            (begin
10              (set! h_var i)
11              h_var)))))
```

このプログラムの式に対する解析は図 7.5 で示す。

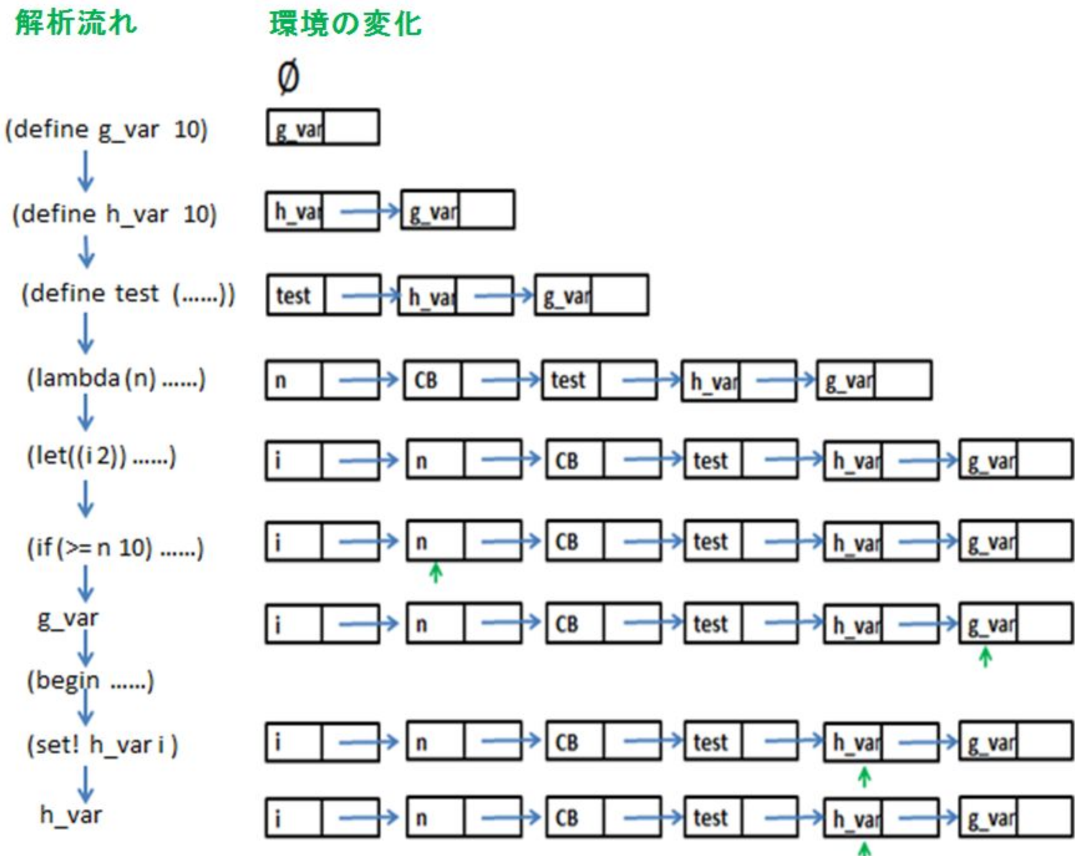

図 7.5: 解析過程

図の左側は変換器がプログラムを再帰的に解析する過程を示している。右側は解析に伴う環境の変化である。これは図 7.4 の環境構造と同じである。図を簡潔にするために、情報を持つ変数束縛の形を一つの記号で表す。最初、環境*env*は空である。大域変数 g_var と h_var の定義を解析する際、環境*env*に変数 g_var と h_var の束縛を追加する。次に、関数 test の定義を解析する。まず、環境に変数 test の束縛を追加する。その後に、*t_lambda* 関数が呼出され、関数 test の本体を解析する。*t_lambda* 関数は *fluid-let* マクロによって、環境にクロージャバウンダリー *CB* を付け加える。それから、パラメータ n を一時的に環境*env*に追加する。続いて、*t_let* 関数が呼び出され、環境*env*に局所変数 i の束縛を追加する。ここまでで、関数 test の環境は構築された。次に、let 式の本体の各式を順に解析する。すると、if 式にたどり着くので、*t_if* 関数を呼び出す。if 式のテスト部、帰結部と代替部は再帰的に解析される。テスト部 (＞＝ n 10) を解析する時に、*t_lookup_var* が呼出され、環境の中に変数 n の束縛を探す。*CB* の左側に見つかるので、変数 n は局所変数だと分かり、test の束縛の LIFT_VARS リストに追加する必要がない。if 式の帰結部には g_var への参照がある。*t_lookup_var* 関数によって、環境を走査し、*CB* の右側のリストで g_var の束縛が見つかる。従って、g_var は大域変数だと分かり、test の束縛の LIFT_VARS リストに g_var を追加する。次に、代替部の begin 式の各式を順に解析する。（set! h_var i）式の解析時、h_var が *CB* の右側で見つかるので、代入されるのは大域変数であることが分かる。よって、test の RET_VARS リストに h_var を追加する。最後の式は h_var への参照である。以上から、test の LIFT_VARS リストに h_var を追加する。関数 test を解析した結果、((g_var h_var) (h_var)) という大域変数情報を得た。

7.3 プログラム変換

変換器は入力された Scheme プログラムをリストとして処理する。cons、car、cdr、set-car!と set-cdr ! 等の関数を利用し、プログラムを変換する。例えば、図 7.6 に示すように、(lambda (n) 式 1…) はリストとして処理される。パラメータリストに変数 x を追加する場合、まず、*(cadr '(lambda (n) 式 1…))* でパラメータ部分 (n) を取り出す。*(set-cdr! (cadr '(lambda (n) 式 1…)) '(x))* で、(n) 部分に変数 x を追加する。変換後の式は (lambda (n x) 式 1…) になる。関数定義と呼出しに対する変換はすべてリスト処理である。

式に対する解析及び変換は同じ処理関数に任せる。解析であるか、変換であるか、変換器の大域変数 *IF_TRAN* によって区別する。もし *IF_TRAN* が *#f* であれば、*t_lambda* 等の処理関数は式を解析し、変換するための情報を集める。大域変数情報を備えると、変換器は変数 *IF_TRAN* を *#t* に更新する。その後に、プログラム変換のモードに入る。これらの処理関数は集めた情報に基づき、引数として受け取った式を変換する。大域変数 *IF_TRAN* は変換器の処理モードを制御するスイッチである。

また、大域変数に相当する局所変数の名前を元の大域変数と区別できるようにする必要がある。新しい名前を生成するために *make-label* 関数を利用する。*make-label* 関数は変数の名前の最後に *label-counter* の値を付け加え、新しい名前を返す関数である。*label-counter* は変換器の大域変数であり、現在のレキシカル・レベルを示している。*label-counter* の初期値は 0 である。*t_lambda* 関数は *fluid-let* マクロによって、一時的に *label-counter* の値を 1 増やす。関数に閉じ込められた大域変数は *make-label* 関数と *label-counter* によって、新しい局所変数の名前に変換される。図 7.7 で示すように、関数 f に入ると、*label-counter* が 1 になる。大域変数 g_var と h_var 及び関数 f のパラメータリストに追加される変数は g_var1 と h_var1 となる。

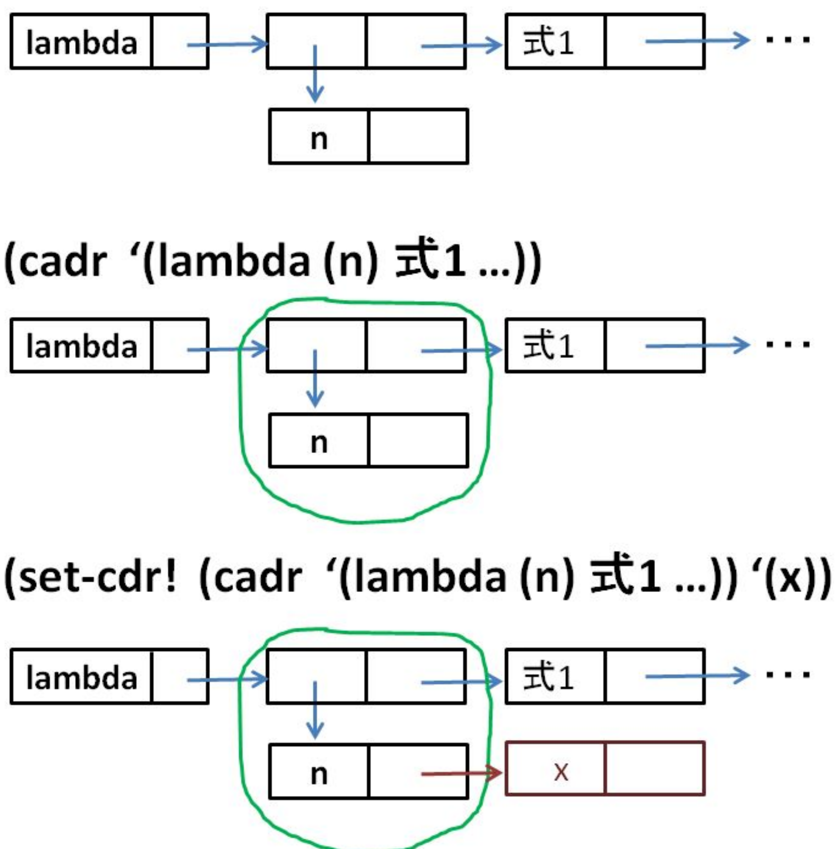

図 7.6: 変換処理

```
(define f
  (lambda (x g_var1 h_var1)
    (+ g_var1 h_var1 x)
  ))
```

label-counter = *0*

label-counter = *1*

label-counter = *0*

図 7.7: 変換例

第 8 章

実験

本研究では、関数が大域変数へアクセスするプログラムに対して、変換の実験を行った。そのために、既存の関数定義に大域変数の使用を導入した。高階関数と内部定義を含むプログラムを変換対象から除いて、それ以外の複数のプログラムを実装した変換器によって、安全にプログラム変換を行った。変換後のプログラムは変換前のプログラムと同じセマンティクスを持ち、自動メモ化によってメモ化できる関数に変換された。また、変換後のプログラムに対してメモ化を適用する実験も行った。実験環境は表 8.1 で示す。

大域変数を含む関数にメモ化を適用した効果を測定するために、フィボナッチ関数、竹内関数等のプログラムに対して、実験を行った。実験は四つの場合に分ける：1)　関数が大域変数を含まない場合、2) 関数が大域変数への参照のみある場合、3) 関数が大域変数への代入のみある場合及び 4) 関数が大域変数への参照と代入が両方ある場合である。 それぞれに対して、メモ化をしない版とメモ化をした版の実行時間を測定した。tak 関数に対する実験を例として説明する。Scheme で

表 8.1: 実験環境

プロセッサ	Intel(R) Core(TM) 2 CPU 2.66GHz
OS	Windows Vista 32 ビット
メモリ	4.00GB

表 8.2: tak の実行結果

	実行時間 (ms) (メモ化なし)	実行時間 (ms) (メモ化あり)	エントリ数
大域変数なし	2142	1 ミリ秒未満	255
大域変数へ参照	2327	1 ミリ秒未満	255
大域変数へ代入	2426	1 ミリ秒未満	255
参照と代入両方	3667	メモリ使い切り	12604861

書く tak 関数のコードは以下の通りである。

```
1  (define (tak x y z)
2    (if (> x y)
3        (tak (tak (- x 1) y z)
4             (tak (- y 1) z x)
5             (tak (- z 1) x y))
6        y))
```

この関数は同じ引数の再帰呼び出しを大量に実行するので、メモ化によって重複する計算が省略できる。大域変数へアクセスする場合の実験を行うために、関数 tak に一つの大域変数を導入した。参照のみの場合は関数の中に一つの定数 1 を大域変数で置き換えた。代入のみの場合、または、参照と代入両方ある場合は代入式とインクリメント式を強引に導入した。表 8.2 で (tak 12 6 0)）の実験結果を示す。各時間データは十回測定した時間の平均値である。

大域変数がない場合、及び参照または代入のどちらかのみ場合のメモ化効果は明らかである。大域変数なし、及び参照や代入のみの場合、メモ化を利用した tak 関数の実行時間は 1 ミリ秒未満であり、メモ化をしない場合より大幅に削減した。

表 8.3: 評価器の実験結果

	実行時間 (ms) (通常)	実行時間 (ms) (メモ化なしの評価器で)	実行時間 (ms) (メモ化ありの評価器で)
(tak 10 3 1)	4.8	1304.4 (15.8)	82.6 (1)
(fib 20)	4.7	915.7 (37.1)	24.7 (1)
(ack 3 6)	48.3	8814.0 (8.0)	1106.0 (1)

それは tak 関数の大域変数が参照のみ、あるいは代入のみだからである。変換後の tak 関数が再帰的に呼び出される際、テーブルに記憶されたキーのパターンが少ない。一方、関数が大域変数のインクリメント式を含む場合、メモリを使い切ったので、実行が途中で強制的に終了させられた。その原因は、関数が呼出すたびに大域変数の値を更新するためである。引数としての大域変数の影響で、テーブルに記憶されるエントリ数は呼出しの回数と同じである。結局、過去の計算結果を再利用できない。関数の実行時間は本来の実行時間とエントリ作成時間とテーブル検索時間の総和になる。従って、再帰呼出し回数が多ければ多いほど、大域変数のインクリメント式を含む関数のオーバヘッドが大きい。このような関数はメモ化によって最適化することができない。

比較的大きなプログラムに対する変換が正しく行えるかどうかを実験するために、350 ステップの超循環評価器 [6] を対象として変換を行った。この評価器プログラムは基本手続き名の束縛を含む大域的環境、入力及び出力の際のプロンプトを表す大域変数を含む。変換後、これらの大域変数は評価器の処理関数のパラメータに追加される。変換後の評価器にメモ化を適用した際、実験結果を表 8.3 に示す。

本実験でテストした超循環評価器は単純であり、効率が良くない。評価器はプロ

グラムをリストとして受け取り、構文を解析し、式の型によって場合分けて処理する。式の評価は再帰的に行われる。竹内関数やフィボナッチ関数のような再帰呼び出しばかりを行うプログラムを評価する際、同じ処理を繰り返し行うことが多い。評価器にメモ化を適用すれば、重複した部分の評価は省略できる。(tak 10 3 1) を評価する際、メモ化する場合はメモ化しない場合より約 16 倍速くなった。(fib 20) の評価は約 37 倍速くなった。(ack 3 6) の評価は約 8 倍速くなった。

また、○×ゲームのゲーム木ソースコードに対してプログラム変換及びメモ化適用の実験を行った。○×ゲームのゲーム木は図 8.1 で示す。

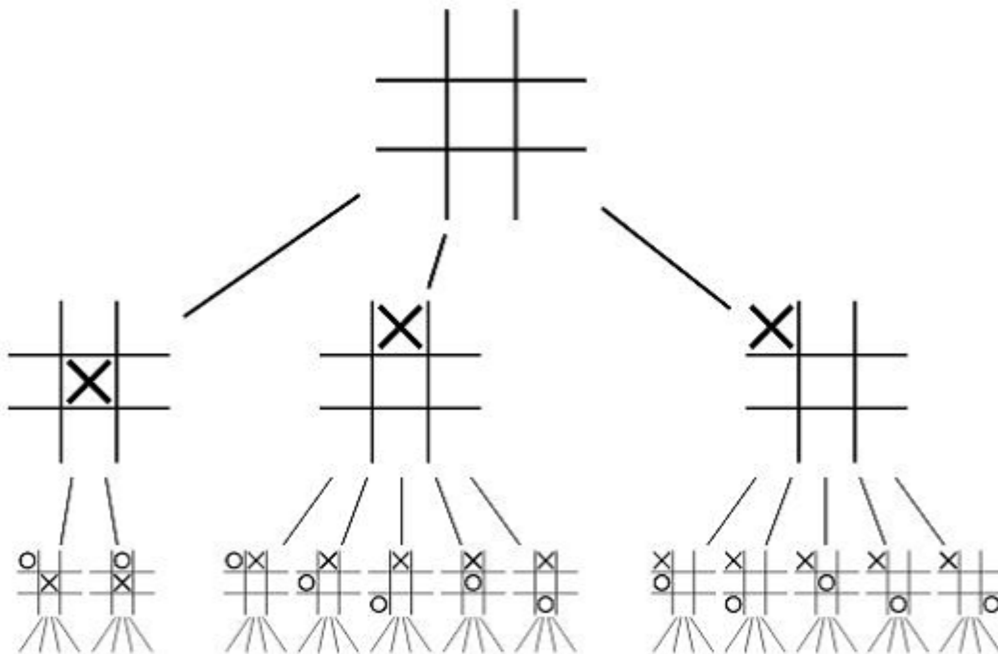

図 8.1: ○×ゲームのゲーム木 [9]

このプログラムが作るゲーム木は○×ゲームの最初からゲームが終わるまで全ての手を含む。一つのゲーム状態を入力として受け取り、ゲームが終わるまで全ての手を出力する。トップレベルの関数は gt である。ゲームの状態は 9 個の記号から成るリストで表す。ゲームの初期状態は (() () () () () () () () ()) である。○×

表 8.4: ゲーム木の実験結果

	実行時間 (ms) (大域変数なし)	実行時間 (ms) (大域変数 1 個)	実行時間 (ms) (大域変数 2 個)
メモ化なし	11088	11606	11332
メモ化あり (連想配列)	85805	91931	103159
メモ化あり (ハッシュテーブル)	507	552	564

を置く順序が異なっても、同じゲーム状態になる場合が多いので、このプログラムに対して、メモ化を適用すると、実行が速くなると考えられる。ゲーム木のプログラム中に大域変数の参照を導入する。一つの大域変数へ参照する場合は×手の記号表現を大域変数で置き換えた。二つの大域変数へ参照する場合は○手と×手の記号表現を大域変数で置き換えた。最初の手が×である場合の実験結果を表 8.4 に示す。

実験結果によると、一つ以上の大域変数を含む場合の実行時間は大域変数がない時より少し長くなった。これは、キーが長くなったためである。本実験の大域変数は定数の代わりなので、本来はこの二つの大域変数をメモ化の際のキーとして加える必要がない。しかし、開発した変換器では大域変数の値が変更さえるかどうかを解析していないのでキーとして加えている。

ゲーム木の呼び出し回数は 7916700 回である。メモ化を適用する場合、呼び出し回数は 53333 回に減り、テーブルに格納されるエントリ数は 53333 個となる。連想配列によってテーブルを実装する場合、メモ化を利用する際の実行時間はメモ化を利用しない場合より長い。その原因はテーブルに記憶されるエントリ数が多いので、検索するためにかかる時間が長いからだと推測する。一方、ハッシュテー

ブルによってテーブルを実装する場合、メモ化を利用する際の実行時間はメモ化を利用しない場合より短い。これはハッシュテーブルの検索と追加が格納されるエントリ数と関わらないからである。

実験の結果、プログラマは大域変数の使い方に注意する必要があることが分かる。場合によっては、大域変数は定数で置き換えることが可能である。あるいは、必要がない代入を関数の外側に移すことができる。余分な大域変数を使った関数に対してメモ化すれば、無駄なエントリ作成コストと検索コストがかかる。当然、どうしても必要な大域変数がある。例えば関数呼び出し回数を計測するために導入したインクリメント式は削除できない。このような関数にメモ化を適用する時に大きいオーバヘッドが出るので、自動メモ化を行う前に、プログラマは関数に「メモ化しない」という印を付ける必要がある。

第 9 章

関連研究

9.1 STM によるメモ化適用範囲の拡張

多くのオブジェクト指向プログラムの計算は内部状態に依存するので、メモ化を利用できない。Rito らはメモ化の適用範囲を拡張するために、ソフトウェア・トランザクショナル・メモリ (STM) を利用し、自動メモ化ツール ATOM システム [3] を実装した。STM はトランザクションに利用される技術であり、共有メモリへのアクセスを遮断し、読み込まれたロケーションと値を read-set に、書き込まれたロケーションと値を write-set に登録する。実行の最後に、read-set を調べ、登録された値が対応するロケーションの値と一致すれば、登録された書き込みをコミットする。ATOM は STM の特性を利用し、メソッドの結果に影響する関係状態を識別できる。メソッドの全ての読み込み操作が read-set に登録されるので、メソッドの結果の関係状態はキャプチャできる。これらの状態はテーブルに記憶され、その後に同じメモ化された関数が呼び出される時にキーとして検索される。もしキーが見つかれば、メソッドの実行は省かれる。そうでなければ、メソッドは再実行される。また、副作用があるメソッドに対しても、キーが見つかれば、実行せずに返値が得られる。しかし、本来の実行に存在する副作用を再実行する必要がある。メモリロケーション及び代入される値を登録した write-set はテーブルに記憶される。キーが見つかる場合は write-set を辿り、それによってロケーションに対する書き込みを再実行する。ATOM システムは STM で集めた情報に基づき、オブジェ

クト指向プログラムに対してメモ化する。このような方法を利用するために、ランタイムシステムがSTMをサポートする必要がある。

我々の研究も、Ritoらと同じ目的を持ち、メモ化適用の制限を除くことを目指している。大域変数へアクセスする関数をメモ化するために、プログラム変換を行う。関数の計算結果に影響する大域変数が関数のパラメータリストに追加されるので、大域変数の値は引数と一緒にテーブルに格納され、キーとして検索することができる。また、関数の中で行われる大域変数への代入が無くなる。その代わりに、関数は大域変数に相当する局所変数へ代入を行い、代入される値を本来の返値と一緒に返す。関数呼び出しは変換され、関数のリターン直後に、大域変数への代入がやり直される。これはRitoらの考え方と似ている。しかし、本研究の提案方法はATOMシステムと異なり、STMをサポートするランタイムシステムを要求しない。従って、より一般的なプログラムに対してメモ化を行うことが可能である。

9.2 チェックポインティングによるデバッギング

Mugshotシステム[1]のように、イベントログと再演を利用するデバッギング手法の他に、チェックポインティングによって、並列プログラムをデバッギングする手法もある。チェックポインティングはプログラムのある時点の実行状態を保存し、後にその状態を再現する手法である。プログラムを最初から実行する必要がなく、直接保存された状態を回復して、そこから実行を再開することができる。従って、チェックポイントによる手法はイベントの発生順序を保存し、再現する再演法と比べ、再開のための実行時間を節約することができる。チェックポイントは一定の間隔で行われる。並列プログラムの非決定性を対処するために、チェックポイントの回数は多いほうが良い。チェックポイントの間隔が短ければ短いほど、バグの発生場所とのずれが小さい。しかし、チェックポインティングのコストは大きくなる。このコストを減らすために、どれぐらいの間隔でチェックポイントを行うべきかを

考える必要がある。チェックポインティングはデバッギング以外に利用される場合もある。Stellner[4] はチェックポインティングを利用し、プロセスを移送する研究を行っている。このような移送方法は、移送先で実行を再開するのが速い。しかし、移送前に行う実行状態の保存コストは高くなる。一方、我々は Mugshot システムの手法を流用し、イベントログと再演によるプロセス移送を検討している。この方法で行うイベント順序の保存コストは小さい。しかし、プロセスの移送先でプログラムを初めから再実行する必要がある。本研究では、実行時間が長い再演法に対して、メモ化を適用し、移送後の再演時間を短縮することが可能であると考えている。

第 10 章

今後の課題

本研究では、変換対象となる Scheme プログラムが高階関数と内部定義関数を含んでないという前提がある。高階関数を含む場合、プログラムに対する解析はより複雑である。例えば関数オブジェクト a を引数として関数 b に渡すことができる。しかし、関数 b が実際にどんな関数オブジェクトを受け取るのかは実際にプログラムを実行しないと、確定しないことも多い。従って、関数 b に対する解析は極めて保守的に行う必要があり、任意の関数を受け取ると解析される可能性がある。高階関数を含む関数に対して、本研究で提案した静的解析より動的解析との併用のほうが適切だと考えている。

また、実験結果によると、変換後のプログラムに対してメモ化を適用する場合、プログラムパターンによって、メモ化の最適化効果が異なる。例えば、大域変数のインクリメント式を含む関数にメモ化を適用する場合、逆に大きいオーバヘッドが出る。このような関数はメモ化すべきではない関数であると考える。今後、自動メモ化を行う際、大域変数を含む関数をさらにメモ化適用できる関数とメモ化適用できない関数を区別する必要がある。そこで、関数にメモ化を適用する場合に最適化効果が出るかどうかを判断することを自動的に行えるツールがあれば、自動メモ化を利用する技術者にとって便利である。

本研究の活用例として、イベントのログによるプロセス再演を高速化するためにメモ化を利用することを考えている。Mugshot システムの手法を流用し、JavaScript プログラムが実行時に起こったイベントをログファイルに記録し、移送先でログ

ファイルによって再演する。しかし、再演時間が長いので、メモ化手法によって再演時間を短縮することを考えている。自動メモ化を行いやすくするために、関数にメモ化を適用する前に、大域変数へアクセスする関数に対してプログラム変換を行う必要がある。しかし、JavaScript プログラムは DOM 操作があるので、提案手法と同じようにプログラムを変換することができない。従って、大域変数へのアクセスと DOM 操作の部分を区別してプログラムを変換する必要がある。あるいは、DOM 操作がある関数を識別し、メモ化しないという印を自動的に付けられるツールを開発し、自動メモ化を行うことを考えている。

第 11 章

結論

自動メモ化を利用する時に、メモ化できる関数とメモ化できない関数を区別する必要がある。参照透過性を備えない関数はメモ化を適用できない。このような関数が多いプログラムに対して自動メモ化を行う場合、メモ化の最適化効果が制限される。本研究では、以上の問題を解決するために、プログラム変換によって、大域変数への参照や代入がある関数をメモ化できる関数に変換する手法を提案した。変換後の関数は大域変数へ直接アクセスしない。しかし、局所変数の導入によって、大域変数への参照と代入は実質上行われる。メモ化する際、関数呼び出しの引数と返値以外に、大域変数の値もテーブルに格納される。

変換後の関数に対してメモ化を適用した効果はプログラムのパターンによって異なる。参照あるいは代入のどちらかのみがある場合のメモ化効果は明らかである。しかし、同一の大域変数への参照と代入を両方含む場合はそうとは限らない。大域変数のインクリメント式を含む関数にメモ化を適用すると、逆にオーバヘッドが大きくなる。今後、メモ化の最適化効果を上げるために、大域変数へアクセスする関数をさらに区別してメモ化する必要があると考えている。また、メモ化をプロセス移送のために利用する場合、どのように本手法を活用し、イベントのログによるプロセス再演を高速化することを考えている。

謝辞

本論文の作成に於きましては, 多くの方々に有形無形の様々な援助と激励を賜わりました。特に指導教員の小宮常康先生に深く感謝します。ご多忙中にもかかわらず、論文を丁寧に読んで下さり、大変貴重な助言を頂きました。終始丁寧に指導して下さり、優しい言葉で私を励まして下さいました。有り難うございます。

多田好克先生、荒堀喜貴先生、佐藤喬先生には、研究を進める上の貴重なご助言をいただきました。ここに深く御礼申し上げます。そして、基盤ソフトウェア学講座の各位には、研究遂行にあたり、日ごろから有益なご討論を頂きました。大変感謝しております。最後に学生諸氏に論文の草稿を添削していただき、ここに感謝の意を表します。

参考文献

[1] James Mickens, Jeremy Elsom, and Jon Howell, Mugshot: Deterministic Capture and Replay for JavaScript Applications, NSDI' 10 Proceedings of the 7th USENIX conference on Networked systems design and implementation, pp.1–11, 2010.

[2] Richard Kelsey, William Clinger, and Jonathan Rees, translated by Hisao Suzuki, Revised5 Report on the Algorithmic Language Scheme, 1998.

[3] Hugo Rito et al., Memoization of Methods Using Software Transactional Memory to Track Internal State Dependencies, in Proc. 2010 ACM, pp.89–98, 2010.

[4] Georg Stellner, CoCheck: Checkpointing and process Migration for MPI, IPPS '96, pp.526–531, 1996.

[5] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, Principles of Program Analysis, 2nd printing 2005.

[6] ジェラルド・ジェイ・サスマン, ハロルド・エイブルソン, ジュリー・サスマン:「計算機プログラムの構造と解釈　第二版, ピアソンエデュケーション, 2000.

[7] William D Clinger, and Lars Thomas Hansen, Lambda, the Ultimate Label or A Simple Optimizing Compiler for Scheme, in Proc, ACM, pp.128–139, 1994.

[8] Robert Bruce Findler et al., DrScheme: A Programming Environment for Scheme, Journal of Functional Programming, 2002.

[9] wikipedia:「ゲーム木」, http://ja.wikipedia.org/wiki/ゲーム木.

[10] Thomas Johnsson, lambda lifting: transforming programs to recursive equations, in proc. of a conference on Functional programming languages and computer architecture, pp.190–203, 1985.