平成20年度 修士論文

ポータブルなJITコンパイラを開発するための使いやすいバックエンド

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 06M-3708-8

栗田 洋輔

指導教員

千葉 滋 准教授

平成19年1月25日

概要

近年、柔軟性が高い点や簡潔な表現が可能な点が評価され、Ruby や Python などの動的言語が注目を浴びている。しかし、動的言語は一般にインタープリタを用いるため、実行速度が遅いという問題がある。また、事前にコンパイルする手法もあるが、eval 関数を用いた動的評価など、実行時まで決まらない性質も多く、そのような手法には限界がある。

このような問題は、実行時に動的にコードをコンパイルする JIT コンパイラを用いることで解決できる。しかし、JIT コンパイラの開発には二つ困難な点がある。一つは、アセンブリや機械語に関する専門知識が必要であるという点である。もう一つは、既存の JIT コンパイラはマシン依存であり移植が困難であるという点である。また、JIT コンパイラ付き VM に言語を移植するという手法もあるが、言語仕様と VM の仕様との間のミスマッチを解消するのが困難である。

そこで我々は、ポータブルな JIT コンパイラを開発するための使いやすいバックエンドを提案した。このバックエンドを利用して開発された JIT コンパイラを動的言語などのインタープリタに組み込むことでプログラムの実行速度を向上することができる。JIT コンパイラの開発者は、バイトコードまたは抽象構文木を中間表現に変換するフロントエンドと、中間表現に意味を与えるテンプレートを作成し、このバックエンドと組み合わせればよい。

フロントエンドとテンプレートのどちらもインタープリタのソースコードの多くを再利用できるため、開発者は容易に JIT コンパイラを実装できる。また、この方法による JIT コンパイラの開発には機械語の知識を必要としない。さらに、作成される JIT コンパイラはマシンに依存しない。

我々はこのバックエンドを評価するため、Forth を解釈するバイトコードインタープリタ Gforth と、Lisp のサブセットを解釈する抽象構文木インタープリタに関して予備的な実験を行った。その結果、どちらも少ないコード量で JIT コンパイラを開発することが可能であり、それによって実行速度を大きく向上させることができることを確認した。

謝辞

本稿は以下の方々なくして、存在しえなかったでしょう。本システムの提案および本稿の編集になにかと心を砕いていただいた千葉滋准教授、光来健一助手、そして研究室のみなさん。心より感謝しています。

目次

第1章　はじめに **7**
- 1.1 研究の背景 7
- 1.2 本論文の構成 8

第2章　典型的なインタープリタのオーバーヘッド **9**
- 2.1 典型的なバイトコードインタープリター 9
- 2.2 典型的な抽象構文木インタープリター 10
- 2.3 JIT Compiler の有用性 11

第3章　本システムの機能と構成 **14**
- 3.1 本システムの構成 14
- 3.2 本システムで作成できる JIT コンパイラの例 16
  - 3.2.1 本システムの利用法 16
- 3.3 JIT コンパイルの利用方法 18
- 3.4 本システムで用いる中間表現 18
- 3.5 フロントエンドによる最適化 19

第4章　実装 **25**
- 4.1 テンプレートの実装 25
- 4.2 最適化コードの生成方法 26
  - 4.2.1 コンパイル済みテンプレートの解析 26
  - 4.2.2 中間表現の生成 28
  - 4.2.3 中間表現の分析から測値命令への定数埋め込みまで 31
- 4.3 テンプレートを用いる利点 31
- 4.4 実現されたマシンの非依存性 32

第5章　実験 **35**
- 5.1 抽象構文木インタープリタでの実験 35
  - 5.1.1 典型的な抽象構文木インタープリタの最適化 35

5.1.2 実験結果 35
5.2 バイトコードインタープリタでの実験 36
5.2.1 Threaded code interpreter 37
5.2.2 Gforth 用 JIT コンパイラの開発 38
5.2.3 実験結果 38

第 6 章 関連研究 **45**
6.1 基本的なアイデア 45
6.2 部分計算と動的特化技術 45
6.3 命令ポインタを除去する手法 46
6.4 ポータビリティを確保する他の手法 46
6.5 バイトコードインタプリタを自動生成する研究 47

第 7 章 まとめ **48**

図目次

2.1 典型的なバイトコードインタープリター 12
2.2 典型的な抽象構文木インタープリター 13

3.1 本システムのアーキテクチャ 14
3.2 x + 3 を実行する最適化されたコード 16
3.3 テンプレート 17
3.4 フロントエンドの実装 21
3.5 バイトコードに対応する中間表現 22
3.6 抽象構文木に対応する中間表現 22
3.7 分岐命令の中間表現 23
3.8 フロントエンドによる最適化 24

4.1 objdump によるコンパイル済みテンプレートの解析例 . . . 27
4.2 コンパイル済みテンプレートの解析情報 29
4.3 中間表現の構造 30

5.1 抽象構文木インタープリターのフロントエンド 40
5.2 抽象構文木インタープリター用テンプレート 41
5.3 Direct threaded code 42
5.4 Gforth インタプリタからの抜粋 43
5.5 Gforth における add 演算のテンプレート 44

表 目 次

4.1 機械語の解析を行う API 26
4.2 中間表現の仕様 33
4.3 中間表現生成のための API 34

5.1 (+ x 3) の実行結果 36
5.2 (if (- x 1) (+ x y) (* x y)) の実行結果 36
5.3 Gforth のベンチマーク 39
5.4 Gforth 用 JIT コンパイラの開発にかかったコスト 39

第1章　はじめに

1.1　研究の背景

近年、柔軟性が高い点や簡潔な表現が可能な点が評価され、Ruby や Python などの動的言語が注目を浴びている。しかし、動的言語は一般にインタープリタを用いるため、実行速度が遅いという問題がある。また、事前にコンパイルする手法もあるが、eval 関数を用いた動的評価など、実行時まで決まらない性質も多く、そのような手法には限界がある。

このような問題は、実行時に動的にコードをコンパイルする JIT コンパイラを用いることで解決できる。しかし、JIT コンパイラの開発には二つ困難な点がある。一つは、アセンブリや機械語に関する専門知識が必要であるという点である。もう一つは、既存の JIT コンパイラはマシン依存であり移植が困難であるという点である。

そこで我々は、ポータブルな JIT コンパイラを開発するための使いやすいバックエンドを提案する。このバックエンドは上記の問題を解決し、次のような特徴を持つ。

- 移植性が高い
  JIT コンパイラの開発者はマシン依存のコードを書く必要がない。具体的には C 言語とマシン非依存のマクロのみを用いるだけでよい。本システムはマシンによる差異を吸収するアーキテクチャになっている。

- JIT コンパイラの開発が容易
  JIT コンパイラの開発者はフロントエンドとコード生成テンプレートを記述するだけでよい。一般的に、コンパイラフレームワークを用いてコンパイラを開発する場合、中間表現のセマンティクスは固定されている。一方、本システムでは、中間表現の構造は固定されているが、セマンティクスは自由に与えることができる。対象言語に合わせた中間表現を用いることができるので、フロントエンドを

開発するコストは小さい。また、コード生成テンプレートに関しても、元のインタープリタのコードの多くを流用できるので、記述は容易である。

1.2 本論文の構成

本稿の残りは、次のような構成からなっている。第 2 章では、典型的なインタープリタのオーバーヘッドとその解決法について述べる。第 3 章では、本システムのアーキテクチャーと機能を述べる。第 4 章では、本システムのこれまでの実装を述べる。第 5 章では、本システムを用いた予備的な実験について述べる。第 6 章では、関連研究について述べる。最後に第 7 章では、まとめを述べる。

第2章　典型的なインタープリタのオーバーヘッド

コンパイラに比べ、インタープリターには様々な実行時オーバーヘッドが存在する。この実行時オーバーヘッドは、特定のインタープリターに固有のオーバーヘッドと、典型的な実装であればどのようなインタープリターにも存在する本質的オーバーヘッドがある。この章では、バイトコードを解釈するバイトコードインタープリターと、抽象構文木をそのまま巡回する非バイトコードインタープリターのそれぞれにおいて、本質的オーバーヘッドを説明する。

2.1　典型的なバイトコードインタープリター

典型的なバイトコードインタープリターの核となるループは、非常に単純である。次のバイトコードをフェッチし、switch 文を用いて適切な実装にディスパッチが行われる。図 2.1 は、典型的なバイトコードインタープリターの核となるループと、「x + 3」を計算するバイトコードの配列である。

このインタープリターは、次のバイトコードにディスパッチを行う switch 文を含んだ無限ループである。switch 文の内部の各々の case は、一つのバイトコードを実装している。また、各々の case では、次のバイトコードを実行するために、switch 文を break し、無限ループの始めに戻るようになっている。

このインタープリターには、以下のようなオーバーヘッドが存在する。

- instructionPointer のインクリメント
- メモリからの次のバイトコードのロード
- オペコードによる switch

- case 文から戻るジャンプ
- 定数のロード

ネイティブコードの実行では、CPU によって、命令レジスタが自動的にインクリメントされ、次のマシン命令がフェッチされる。バイトコードインタープリターの場合は、そのような処理はソフトウェアによって明示的に行われるため、オーバーヘッドが発生してしまう。また、典型的な実装では、バイトコードを実行する度にオペコードのチェックを行うので、それもオーバーヘッドとなる。最後に挙げた「定数のロード」は、図 2.1 におけるバイトコードの中に埋め込まれた「3」や「x」をロードする処理のことである。

典型的なバイトコードインタープリターは作成や理解が容易で移植性が高いが、このようなオーバーヘッドのために処理は遅い。特にバイトコードが、(図 2.1 における PUSH_CONST のような) 単純な処理を行う場合は、実行時間の大部分が、バイトコードに対応する本質的な処理ではなく、ディスパッチのオーバーヘッドに消費されてしまう。

2.2　典型的な抽象構文木インタープリター

抽象構文木を巡回する典型的な抽象構文木インタープリターによる処理は、バイトコードインタープリターによる処理に比べて実行時間やプログラムのサイズという点で不利である。しかし、バイトコードコンパイラを作成する手間が省けるという実装の容易性から、現在でもプログラミング言語 Ruby 等でこのような非バイトコードインタープリターは用いられている。

図 2.2 は、「x + 3」を計算するために必要となる、典型的な抽象構文木インタープリターの実装の一部である。eval 関数は式 expr をとり、expr の型に応じて異なる処理を行う。

このインタープリターには、以下のようなオーバーヘッドが存在する。

- 式の型による switch
- 部分式を eval するための call/return
- 定数のロード

典型的な非バイトコードインタープリターは、移植性が高く、バイトコードインタープリター以上に作成や理解が容易だが、call/return のオーバーヘッドが大きいので処理は非常に遅い。また、バイトコードインタープリターと同様に、式のオペランドに対応する処理が単純であるほど、全体の計算時間に占めるオーバーヘッドは大きくなる。

2.3　JIT Compiler の有用性

インタープリターには以上で述べたように、多くのオーバーヘッドが存在する。そのため、計算処理の速度は遅い。この問題を解決するために、多くのインタープリターには、評価される式 (またはバイトコード) を実行時にネイティブコードに置き換える JIT コンパイラが組み込まれている。

JIT コンパイラの実装方法は様々であるが、一般的にはアセンブリや機械語に関する深い知識を要求する。そのような知識は、インタープリターの実装に用いる知識とは異なるため、JIT コンパイラの作成は一般的には困難である。また、アセンブリや機械語を直接用いた実装はマシン依存であるため、移植性が低いという問題もある。

```
compiled code:
unsigned char code[] = { ...,
  PUSH_GLOBAL_VAR,
  x,
  PUSH_CONST,
  3
  ADD, ...};

bytecode implementations:
int *stackPointer;
unsigned char *instructionPointer = code - 1;
for (;;) {
  unsigned char bytecode = *++instructionPointer;
  switch (bytecode) {
  /* ... */
  case PUSH_GLOBAL_VAR:
    *++stackPointer =
        getVarFromSymbol(*++instructionPointer);
    break;
  case PUSH_CONST:
    *++stackPointer = *++instructionPointer;
    break;
  case ADD:
    --stackPointer;
    *stackPointer += stackPointer[1];
    break;
  /* ... */
  }
}
```

図 2.1: 典型的なバイトコードインタープリター

```
expr: (+ x 3)

int eval(List* expr) {
  switch (testElementType(expr)) {
  /* ... */
  case NUMBER :
    return getIntegerElement(expr);
  case SYMBOL :
    return getVarFromSymbol(expr);
  case PLUS :
    return eval(getSecond(expr)) + eval(getThird(expr));
  /* ... */
  }
}
```

図 2.2: 典型的な抽象構文木インタープリター

第3章　本システムの機能と構成

2章で述べたように、典型的なインタープリターには本質的なオーバーヘッドが存在する。このオーバーヘッドは、基本的に、どのような言語を実装するインタープリターにも存在している。そこで、我々はそのような本質的オーバーヘッドを除去することを目的とした単純な JIT コンパイラを作成するためのシステムを提案する。このシステムの特徴は、C 言語のみを用いて JIT コンパイラを作成できる点と、JIT コンパイラが (制限はあるが) マシン非依存である点である。

3.1　本システムの構成

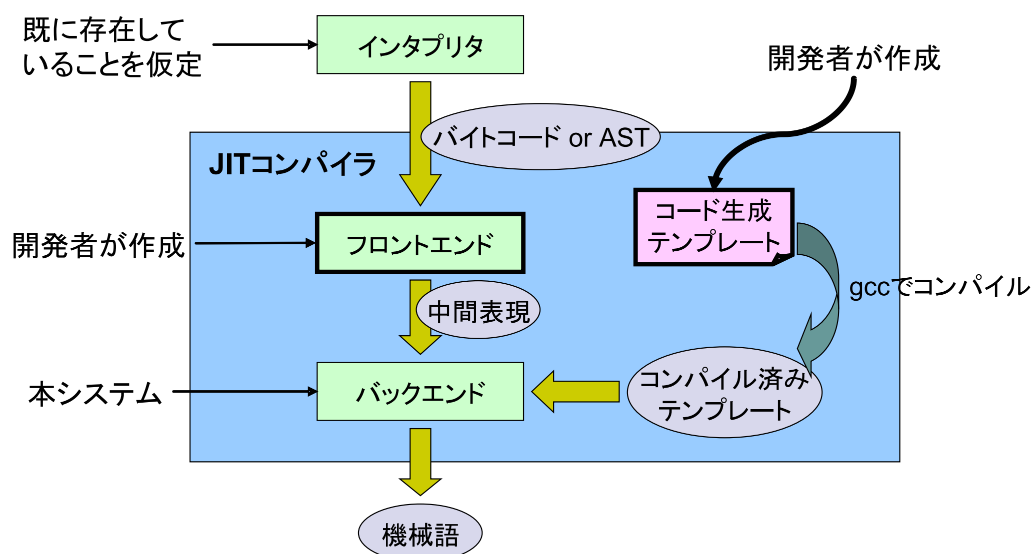

図 3.1: 本システムのアーキテクチャ

本システムの構成と入出力は図 3.1 のようになっている。これらの構成要素をここで簡単に説明する。

- インタプリタ
  JIT コンパイラを組み込むターゲットとなるインタプリタである。本システムを用いる時点で既に存在することを前提とする。実装は C 言語で行われていなければならない。

- JIT コンパイラ
  バイトコードまたは抽象構文木 (AST) を入力すると、最適化された機械語を出力する。フロントエンドおよびバックエンドから構成される。この JIT コンパイラの作成を容易にすることが本システムの目的である。

- 中間表現
  フロントエンドとバックエンドの間で受け渡しされるデータ。バイトコードまたは抽象構文木の構造を保持している。

- コード生成テンプレート
  C 言語で書かれたコードである。しかし、C プログラム上でこのコードが呼び出されることは仮定しない。コード生成テンプレートを gcc でコンパイルした結果できる機械語列のコンパイル済みテンプレートが最適化された機械語列の材料となる。このコードは本システムを用いて JIT コンパイラを開発する開発者が記述する必要がある。

- コンパイル済みテンプレート
  コード生成テンプレートを gcc でコンパイルすることで得られる機械語列である。JIT コンパイラのバックエンドの入力となる。

- フロントエンド
  JIT コンパイラを構成するモジュールである。バイトコードまたは抽象構文木を入力とし、中間表現を出力とする。フロントエンドは本システムを用いて JIT コンパイラを開発する開発者が記述する必要がある。

- バックエンド
  JIT コンパイラを構成するモジュールである。中間表現およびコンパイル済みテンプレートを入力とし、最適化された機械語列を出力

とする。この部分は本システムによって提供されており、JIT コンパイラの開発者はそれを利用するだけでよい。

3.2　本システムで作成できるJIT コンパイラの例

本システムを用いれば、インタープリタの本質的オーバーヘッドを除去する JIT コンパイラを作成することができる。本システムで作成した JIT コンパイラを用いて、図 2.1 のインタープリターを「x + 3」に対応するバイトコードに対して特化し、本質的なオーバーヘッドを除去したものが図 3.2 である。(JIT コンパイラによって生成される実際のコードはネイティブコードであるが、わかりやすく C 言語で表記した)

```
*++stackPointer = getVarFromSymbol(x);
*++stackPointer = 3;
--stackPointer;
*stackPointer += stackPointer[1];
```

図 3.2: x + 3 を実行する最適化されたコード

バイトコードの評価を行う実装が直列化されたことで、instructionPointer のインクリメント、メモリからの次のバイトコードのロード、オペコードによる switch、case 文から戻るジャンプを省略している。また、2 行目では、stackPointer が指すメモリに定数を直接ストアしている。このような定数のストアは、ネイティブコードでは即値命令で表現される。図 2.1 のインタープリターでは、メモリから「3」をロードし、それをストアするという一連の処理を行う必要があったが、図 3.2 では即値命令でストアするだけである。これによって、定数のロードが省略された。

図 2.2 の非バイトコードインタープリターも、グローバル変数 stack-Pointer を追加し、本システムで JIT コンパイラを作成することで、図 3.2 のようなコードを生成することができる。

3.2.1　本システムの利用法

本システムのユーザー (JIT コンパイラの開発者) は、コード生成テンプレートと中間表現を共に C 言語で記述することによって JIT コンパイ

ラを作成する。インタープリターは JIT コンパイル時に、バイトコードあるいは抽象構文木をフロントエンドに渡し、中間表現を生成する。そして、その中間表現をバックエンドに渡すことで最適化されたコードを取得・実行することができる。バックエンドは、中間表現に登録されている情報を元に、コンパイル済みテンプレートからマシン非依存な最適化コードを生成する。

ここでは、本システムの利用法を説明する例として、図 2.1 のインタープリタに組み込む JIT コンパイラの作成を考える。「x + 3」に対応するバイトコードが入力として与えられた場合に、図 3.2 のような最適化コードを生成できればよい。

```
int* stackPointer;

void code_NUMBER() {
  int  ret_val;
  HOLE(code_NUMBER_hole, ret_val);
  *++stackPointer = ret_val;
}

void code_SYMBOL() {
  char* symbol;
  HOLE(code_SYMBOL_hole, symbol);
  *++stackPointer =
   getVarFromSymbol(symbol);
}

void code_PLUS() {
  LABEL(code_Plus_section_1);
  LABEL(code_Plus_section_2);
  stackPointer--;
  *stackPointer = stackPointer[1];
}
```

図 3.3: テンプレート

図 3.2 は、図 2.1 の各々の case 内の実装を結合したものといえる。そこで、まずユーザーは、case 内の実装に対応するテンプレート (図 3.3) を記述する必要がある。図 2.1 の case PUSH_GLOBAL_VAR は、図 3.3 の

code_SYMBOL に、case PUSH_PUSH_CONST は、図3.3 の code_NUMBER に、case ADD は、図3.3 の code_PLUS に対応している。

テンプレートの記述が完成したら、ユーザーは次にコンパイル済みテンプレートを結合するためのフロントエンド (図3.4) を記述する (詳細は後述)。フロントエンドは、図2.1 のインタープリタの制御構造をそのまま借用すればよい。制御が case PUSH_GLOBAL_VAR の中に入った場合は、最適化コードの末尾に code_SYMBOL の実装を付加すればよい。この際、code_SYMBOL 内の symbol には、バイトコードから取得した変数名を埋め込む。制御が case PUSH_CONST の中に入った場合は、最適化コードの末尾に code_NUMBER の実装を付加すればよい。この際、code_NUMBER 内の ret_val には、バイトコードから取得した数値を埋め込む。制御が case ADD の中に入ったときは、最適化コードの末尾に code_PLUS の実装を付加すればよい。

図3.4 のフロントエンドを、バイトコードへの参照を引数にして呼び出せば、中間表現を取得できる。最後に、バックエンドに中間表現を渡せば、最適化された機械語を取得することができる。

3.3 JIT コンパイルの利用方法

JIT コンパイラを実装する際は、まず、「どのタイミングで、どのコードを最適化」するかを考える。例としては、関数が呼び出される度に増加するカウンタを用意し、カウンタが一定の値を超えると JIT コンパイルを行うという方法が挙げられる。JIT コンパイルを行うタイミングと、最適化するコードが決定したら、インタープリター内の適切な場所で中間表現を生成し、generateCode() を呼ぶようにすればよい。最適化されたコードは、再利用するためにインタープリター内の適切なデータ構造に保存するとよい。関数単位で最適化を行う場合は関数テーブルに最適化された機械語のアドレスを格納する。

3.4 本システムで用いる中間表現

本システムにおける中間表現はグラフ構造になっており、それぞれのノードがバイトコード命令や、抽象構文木のノードに対応している。図3.5はバイトコードインタープリタ用JITコンパイラの内部で用いる中間構造の例であり、図3.6は抽象構文木インタープリタ用JITコンパイラの内部で用いる中間構造の例である。どちらの場合も、元となるバイトコードや抽象構文木の構造が維持されている。一般的なコンパイラのフロントエンドとは異なり、構造を変える必要がないので、フロントエンドの実装は容易である。

図3.5では、中間表現にlabelという属性とholesという属性がある。label属性は、最適化された機械語を生成する際にどのテンプレートを用いるかを指定する。holes属性はテンプレート中にどのような定数を埋め込むかを指定する。

また、図3.6では、テンプレートが入れ子状になっている。ここではinnnertemplate属性が用いられている。この属性で、テンプレート中のどこの部分に他のどのテンプレートを挿入するかを指定する。

また本システムでは命令ポインタを除去するため、バイトコードレベルの分岐命令を機械語レベルの分岐命令に変換する。この場合、JITコンパイラの開発者は、分岐命令用のテンプレートを作成するのではなく、本システムが用意した分岐命令用の特殊な中間表現ノードを用いる。このノードはバイトコードレベルでの分岐先アドレスを情報として持つ。本システムのバックエンドはこの中間表現を、適切な機械語の分岐命令に変換する。

3.5　フロントエンドによる最適化

本システムを用いることで、インタープリタに共通する本質的なオーバーヘッドを除去し、最適化を行うことができる。しかし、最適化の可能性はそれだけではない。フロントエンドで生成する中間表現を工夫することで、バイトコードレベル、抽象構文木レベルの最適化を行うことができる。

図3.8はフロントエンドによる最適化の例である。上部の中間表現では、引き算と掛け算の中間表現を融合している。引き算と掛け算を合わせたテンプレートを記述し、それを利用することで最適化を図っている。

融合命令を作成すると、コンパイラ (本システムの場合は gcc) が複数の命令間で最適化を行うので、それぞれの命令を別々にコンパイルして連結したものよりも高性能なコードを生み出すことができる。

また、下部の中間表現では、2 と 3 をスタックにプッシュし、それを足し合わせるという中間表現を、5 をスタックにプッシュする中間表現に変換している。これはバイトコードレベルでの定数の畳み込みである。バイトコードコンパイラによって生成されるバイトコードがナイーブであれば、このような最適化によって JIT コンパイラの性能向上を実現できる。

```
compiled code:
unsigned char code[] = { ...,
  PUSH_GLOBAL_VAR,
  x,
  PUSH_CONST,
  3
  ADD, ...};

bytecode implementations:
BlockCompnt* analyze(unsigned char* inst_pter) {
 BlockCmpnt* cmpnt;
 for (;;) {
  unsigned char bytecode = *++inst_pter;
  switch (bytecode) {
  /* ... */
  case PUSH_GLOBAL_VAR:
    appendBlock(cmpnt,
                initByTemplate(v_code_SYMBOL));
    addHoleInfo(cmpnt,
                v_code_SYMBOL_hole,
                *++inst_pter);
    break;
  case PUSH_CONST:
    appendBlock(cmpnt,
                initByTemplate(v_code_NUMBER));
    addHoleInfo(cmpnt,
                v_code_NUMBER_hole,
                *++inst_pter);
    break;
  case ADD:
    appendBlock(cmpnt,
                initByTemplate(v_code_Plus);
    break;
  /* ... */
  }
}
```

図 3.4: フロントエンドの実装

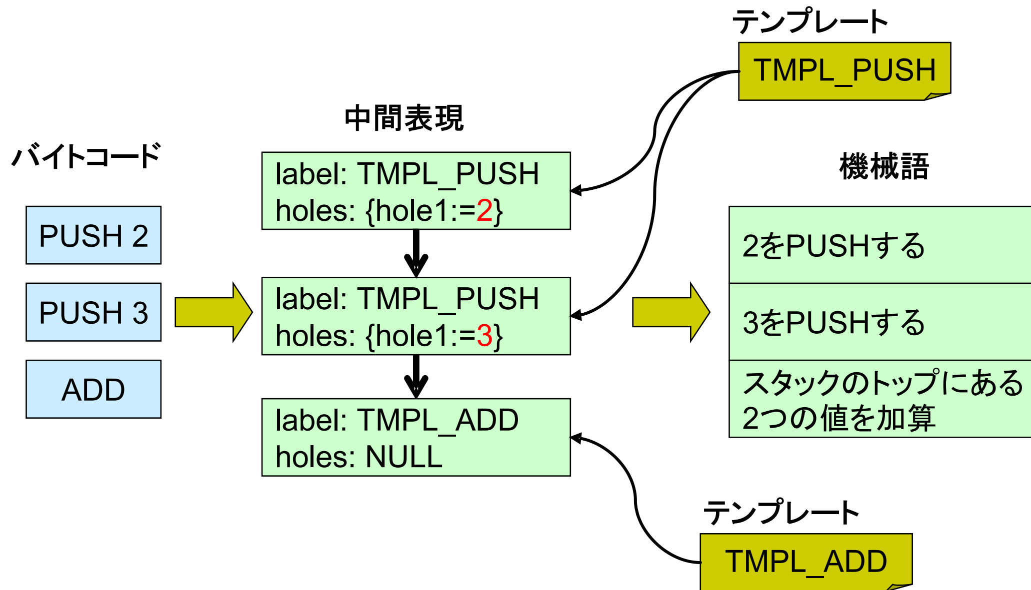

図 3.5: バイトコードに対応する中間表現

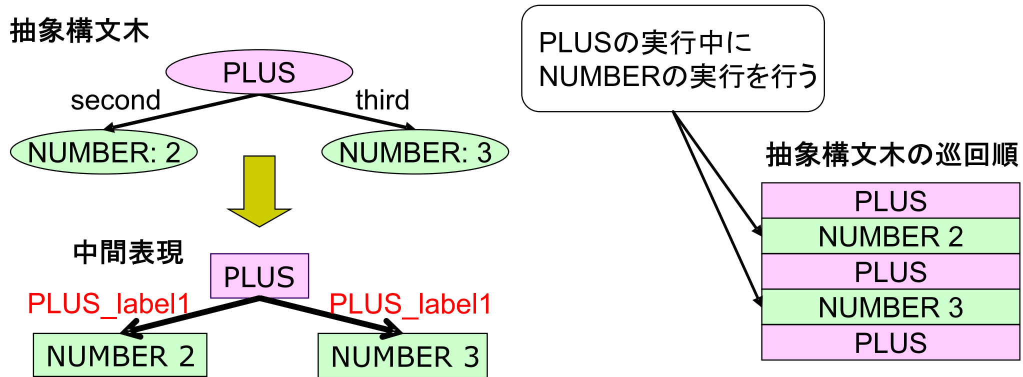

図 3.6: 抽象構文木に対応する中間表現

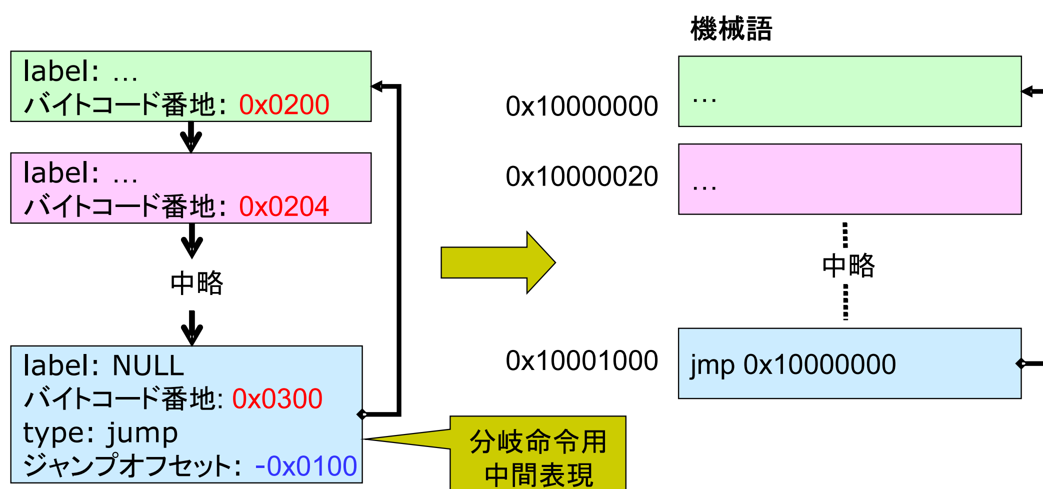

図 3.7: 分岐命令の中間表現

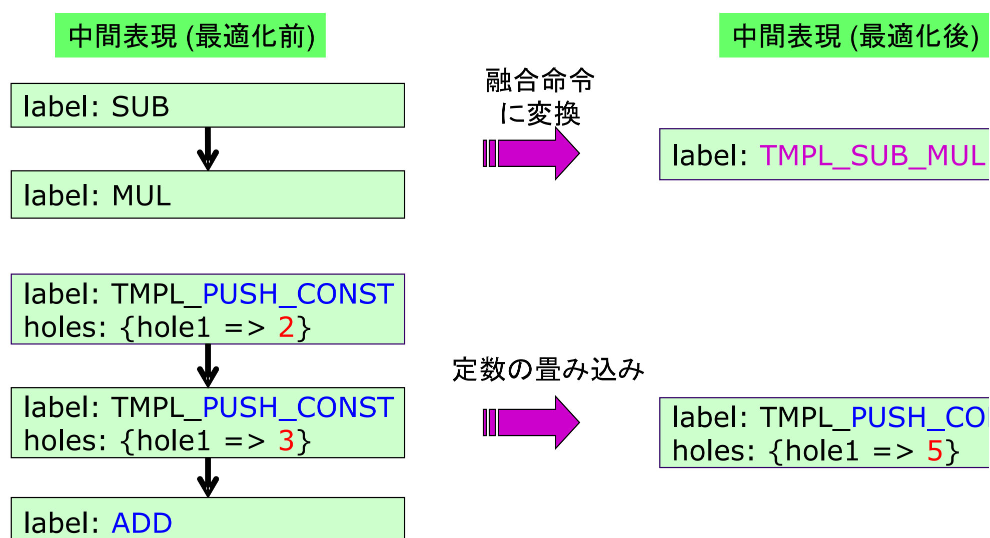

図 3.8: フロントエンドによる最適化

第4章　実装

4.1　テンプレートの実装

テンプレートは、C 言語で書かれたコードの断片である。抽象構文木インタープリタの場合は無引数の void 関数内に記述する。バイトコードインタープリタの場合はインタープリタのメインループまたは (threading 技術を用いている場合は) メインループを元に変換されている部分に記述する。この void 関数は、機械語列を作成するための材料であり、呼び出されることはない。またメインループ内に記述したテンプレートも実行されることはない。また、関数名も JIT コンパイル時に識別するために必要なだけであるので、任意でよい。識別する際は、関数名の前に v_を付けた変数を用いる。なお、複数のテンプレートを横断する値にはグローバル変数を使用する。

これらのテンプレート内では、マクロ LABEL と HOLE が使われている。これらは gcc のプリプロセッサによってインラインアセンブリに変換される。LABEL は、アセンブリレベルのラベルに変換される。このラベル情報は実行ファイル内に残り、JIT コンパイル時に参照することができる。本システムはこの位置情報を利用してテンプレートを操作する。また、HOLE に関しては、第一引数がアセンブリレベルのラベルに変換され、第二引数はその名前の変数に任意の値を代入する即値命令に変換される。即値の部分には、テンプレートを組み立てる際に適切な値が代入される。

本システムが提供するバックエンドは、構造体 BlockInfo をノードとする中間用言を辿ることで最適化コードを生成する。BlockInfo は、テンプレートへの参照、テンプレート内に埋め込む定数と埋め込む位置、テンプレート内に挟み込む他の BlockInfo の情報と挟み込む位置などを保持する。

4.2　最適化コードの生成方法

最適化コード生成は以下の 6 つのステップから構成される。

- コンパイル済みテンプレートの解析
- 中間表現の生成
- 中間表現の分析
- テンプレートのコピー
- 相対アドレスの修正
- 即値命令への定数埋め込み

一つ目の機械語の解析はインタープリタの起動時に一回だけ行われ、他のステップは JIT コンパイルを行う度に実行される。以下、それぞれのステップについて解説する。

4.2.1　コンパイル済みテンプレートの解析

解析のための API

コンパイル済みテンプレートに含まれる情報は静的なので、その解析は一回だけ行えばよい。JIT コンパイラの開発者は、表 4.1 の関数 objdump_parse を、最初の JIT コンパイルが行われる前の任意の時点で一回呼び出すようにする。

表 4.1: 機械語の解析を行う API

関数名	objdump_parse
戻り値	void
第一引数	char* begin_label
第二引数	char* end_label
処理	インタープリタを構成する機械語のうち、begin_label から end_label までの範囲を解析し、データ構造を生成する

このステップでは、範囲を指定することで、コンパイル済みテンプレート以外の部分を解析しないことが重要である。コンパイル済みテンプレート以外の部分を解析しても JIT コンパイラは正常に動作するが、解析に不必要な時間を消費してしまうし、無駄な解析情報を作成することでメモリも消費してしまう。なお、コンパイル済みテンプレートの全てがメモリ中の連続した領域に配置されない場合は、objdump_parse を分断された範囲ごとに呼び出せばよい。

解析の内容

コンパイル済みテンプレートの解析は、現時点ではディスアセンブラである objdump の出力テキストを分析することで対応している。図 4.1 は-d オプションを付けて objdump を呼び出し、コンパイル済みテンプレートを解析した例である。機械語のアドレスやニーモニックに加え、Elf フォーマットのバイナリに含まれているラベル情報も出力される。

```
0804cd2c <code_EvalIf>:
 804cd2c: 55                    push   %ebp
 804cd2d: 89 e5                 mov    %esp,%ebp

0804cd2f <code_EvalIf_section1>:
 804cd2f: a1 b0 16 05 08        mov    0x80516b0,%eax
 804cd34: 83 e8 01              sub    $0x1,%eax
 804cd37: a3 b0 16 05 08        mov    %eax,0x80516b0
 804cd3c: a1 b0 16 05 08        mov    0x80516b0,%eax
 804cd41: 8b 04 85 20 17 05 08  mov    0x8051720(,%eax,4),%eax
 804cd48: 85 c0                 test   %eax,%eax
 804cd4a: 74 02                 je     804cd4e <code_EvalIf__e>
```

図 4.1: objdump によるコンパイル済みテンプレートの解析例

コンパイル済みテンプレートを解析するための API である関数 objdump_parse を呼び出すと、objdump -d によるこのような出力を取得した後、そのテキストを解析してデータ構造を構成する。本システムでは、ラベルによって挟まれている部分毎に構造体 LabelInfo のインスタンスを生成し、それを連結リストにしている。図 4.1 の場合は、code_EvalIf に対

応する LabelInfo 構造体のインスタンスと code_EvalIf_section1 に対応する LabelInfo 構造体のインスタンスがポインタによって連結される。また、このステップでは、命令ごとに InstructinInfo 構造体のインスタンスを作成する。この構造体は、命令が配置されているアドレスと、命令の種類、そしてデバッグのためのニーモニックを情報として保持する。このように、命令ごとに多くの情報を保存することが、解析の範囲を不必要に大きくしないことが重要である理由である。なお、InstructionInfo も LabelInfo と同様に連結リストを構成する。LabelInfo および InstructionInfo を連結リストで実装するのは、通常、これらがシーケンシャルにアクセスされるからである。また、LabelInfo は、そのラベルによって指定されるアドレスにある命令に対応する InstructionInfo へのポインタを保持している。図 4.1 を解析して生成されるデータ構造は図 4.2 のようになる。図中の abs はその命令が絶対アドレスだけを用いていることを示している。

このステップで生成されるコンパイル済みテンプレートの解析情報は、JIT コンパイルを行う度に利用される。そのため、JIT コンパイルの時間を短くするためには、この情報を短時間で取得することが重要になる。テンプレートが検索できれば、そこから先はシーケンシャルアクセスを行うので連結リストが効率的だが、初めにテンプレートを検索するにはこの方法は非効率的である。そのため、本システムでは LabelInfo の検索にハッシュを利用している。具体的には、キーをラベル名、エントリを LabelInfo のインスタンスとしている。

4.2.2 中間表現の生成

JIT コンパイルを行う際、まず初めにフロントエンドにより中間表現の生成を行う。図 4.2 は中間表現の仕様、図 4.3 は中間表現の構造を示している。

中間表現を生成するには図 4.3 を用いる。

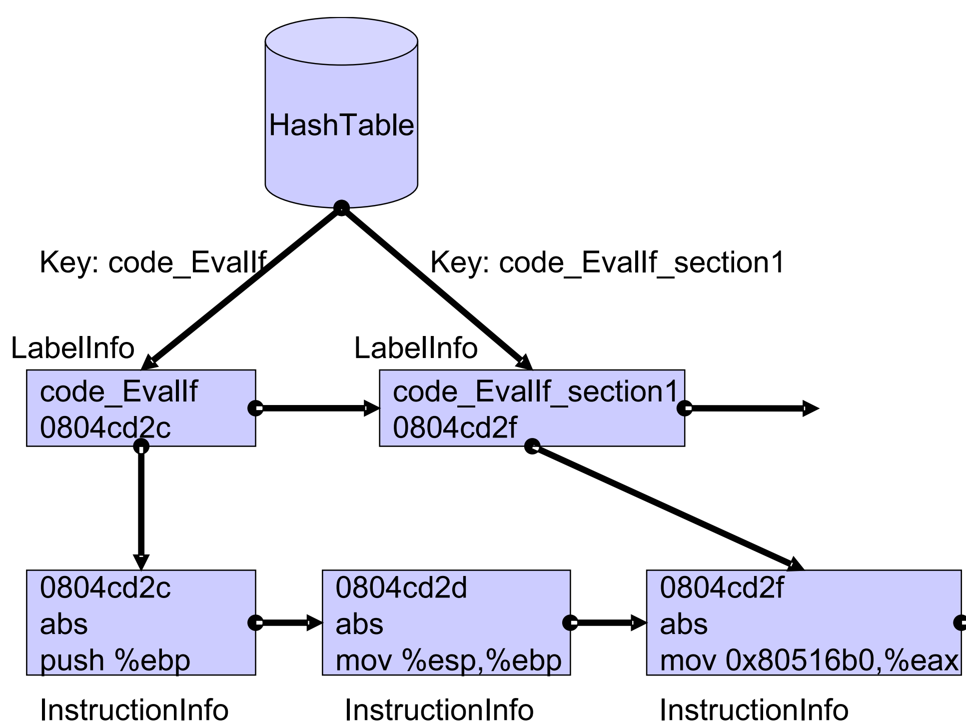

図 4.2: コンパイル済みテンプレートの解析情報

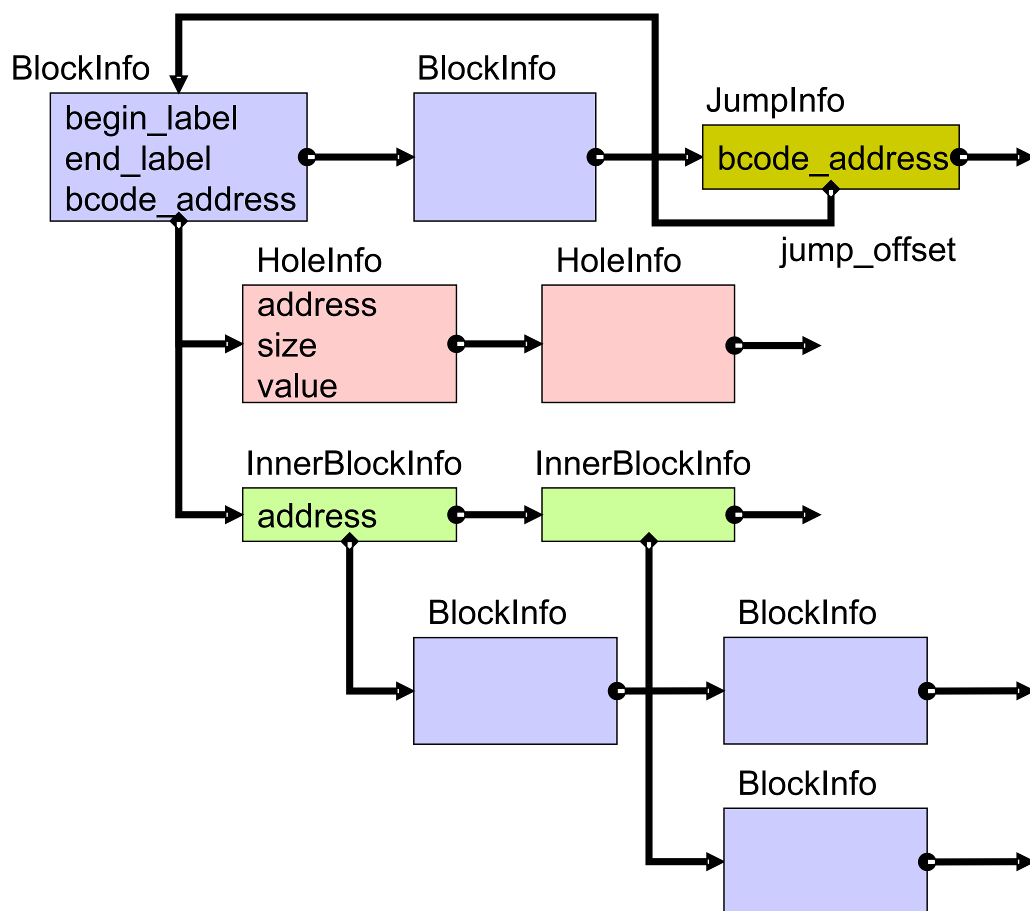

図 4.3: 中間表現の構造

4.2.3　中間表現の分析から測値命令への定数埋め込みまで

フロントエンドによって生成された中間表現はバックエンドにより分析される。バックエンドは、コンパイルされたテンプレートをコピーして並べるのであるが、この際に、定数を埋めたり、call 命令や jump 命令の相対オフセットを調節したりといった低レベルな操作を自動的に行う。テンプレート内に他のテンプレートが挿入される場合は、jump 命令の相対オフセットを、挿入されたテンプレートのサイズだけ調整する。jump 命令が short jump であるときは相対オフセットを増加させるだけでは対応できない場合もあるので、そのようなときはオペコードも自動的に変更を行う。

テンプレートを通常通りにコピーしてからオペコードを変更すると、その後、テンプレートをずらすためにまたコピーを行う必要性が生じてしまう。そのため、本システムのバックエンドは、テンプレートをコピーする前にオペコードを変更する必要性を認識する。

4.3　テンプレートを用いる利点

最適化コードの生成において、C 言語で書かれたテンプレートを用いることには様々な利点がある。

まず、アセンブリや機械語を直接記述する必要がないことから、それに関する知識がない開発者でも JIT コンパイラを作成することができる。また、JIT コンパイラの実装は制限はあるながらも、マシン非依存となる。

テンプレートは C 言語で記述されているので可読性が高く、gcc によって静的にコンパイルされるので、gcc の手続き内最適化がそのまま利用可能である。JIT コンパイラが JIT コンパイル時に行う処理は、テンプレートを結合し、一部の内容を書き換えるだけであるので、Register Tranfer Language(RTL) や 3 番地コードなどの中間表現を用いた実装よりも、コード生成のコストが小さい。

テンプレートを用いた場合の欠点としては、ソースコードがバイナリーに変換された際に情報を失ってしまうことで、高度な最適化を行いにくくなることが考えられる。しかし、テンプレートを用いた実装でも peephole optimazion に関しては行うことが可能である。

4.4 実現されたマシンの非依存性

本システムは、マシンアーキテクチャの実装を隠蔽した API を提供しているので、それを用いて JIT コンパイラを作成するユーザーは、アーキテクチャを意識する必要がない。しかし、本システム自体はアーキテクチャ固有の実装に対応する必要がある。本システム内部のアーキテクチャによって可変の部分には、ディスアセンブラ、HOLE マクロ、即値命令の中に定数を埋め込む方法、jump や call の destination の調整方法がある。また、本システムの実装は gcc インラインアセンブリに依存しているため、gcc がそのアーキテクチャに対応している必要がある。

表 4.2: 中間表現の仕様

構造体 BlockInfo	コンパイル済みテンプレートの使用部分を示す
メンバ begin_label	コンパイル済みテンプレートの使用部分の開始位置
メンバ end_label	コンパイル済みテンプレートの使用部分の終了位置
メンバ bcode_address	バイトコードのアドレス (抽象構文木インタープリタでは使われない)
メンバ next	次に配置するコンパイル済みテンプレートに対応する BlockInfo
メンバ hole_info	そのテンプレート内で使われる hole の情報
メンバ innet_template_info	そのテンプレートの内部に挿入される他のテンプレートの情報

構造体 HoleInfo	コンパイル済みテンプレート内の hole の情報
メンバ address	hole のアドレス
メンバ size	hole のサイズ
メンバ value	hole に埋め込まれる値
メンバ next	次の hole の情報を持つ HoleInfo

構造体 InnerBlockInfo	テンプレートの内部に挿入されるテンプレートの情報
メンバ address	テンプレートが挿入されるアドレス
メンバ block_info	挿入されるテンプレートの情報を持つ BlockInfo
メンバ next	テンプレート内部の他の部分に挿入される InnerBlock-Info

構造体 JumpInfo	ジャンプ命令の情報
メンバ bcode_address	バイトコードのアドレス (抽象構文木インタープリタでは使われない)
メンバ next	次に配置するコンパイル済みテンプレートに対応する BlockInfo
メンバ jump_to	そのテンプレートの内部に挿入される他のテンプレートの情報

表 4.3: 中間表現生成のための API

関数名	init_by_template
戻り値	BlockInfo*
第一引数	Address* begin_label
第二引数	Address* end_label
第三引数	BCodeAddress* bcode_address
処理	インタープリタを構成する機械語のうち、begin_label から end_label までの範囲に対応する BlockInfo 構造体のインスタンスを生成し返す。bcode_address でバイトコードのアドレスを結びつける。

関数名	appendBlock
戻り値	void
第一引数	BlockInfo* block_info_prev
第二引数	BlockInfo* block_info_next
処理	block_info_prev の次に block_info_next を連結する。

関数名	addHoleInfo
戻り値	void
第一引数	BlockInfo* block_info
第二引数	Address* insertion_label
第三引数	long value
処理	block_info で指定されたブロックの insertion_label の部分に value を挿入するという情報を付加する。

関数名	addInnerBlock
戻り値	void
第一引数	BlockInfo* outer_block_info
第二引数	BlockInfo* inner_block_info
第三引数	Address* insertion_label
処理	outer_block_info に inner_block_info と insertion_label の情報を付加する。外側のブロック中の insertion_label で指定された部分に内側のブロックが挿入される。

第5章　実験

我々は、本システムを評価するため、プログラミング言語 Forth[2] を解釈するバイトコードインタープリタ Gforth[1] と、Lisp のサブセットを解釈する抽象構文木インタープリタに関して予備的な実験を行った。その結果、どちらも少ないコード量で JIT コンパイラを開発してインタープリタに組み込む可能であり、それによって実行速度を向上させることができることを確認した。

5.1　抽象構文木インタープリタでの実験

抽象構文木インタープリタに組み込む JIT コンパイラを本システムで作成し実験を行った。

5.1.1　典型的な抽象構文木インタープリタの最適化

図 5.1 は、図 2.2 の典型的な抽象構文木インタープリターに対して作成したフロントエンドである。テンプレートに関しては、図 5.2 を利用する。関数 addInnerCmpnt は、第一引数で指定した BlockCmpnt に、「第二引数で指定した位置に、第三引数のテンプレートを挿入する」いう情報を付加する。この例では code_PLUS_section に二つのテンプレートが挿入されているが、このような場合は、最初に挿入されたテンプレートが前に来る仕様にしている。

フロントエンドは、インタープリターの構造をそのまま流用することができ、記述するコードも少ないので実装は容易である。

5.1.2　実験結果

Intel Xeon CPU 3.06GHz x 2 (Memory 2GB)、Linux 2.6.17、gcc4.1.1、glibc 2.4 という実験環境で、前節で示したコードを元に実験を行った。時

間を測定するに当たっては、1000回実行して平均を計算した。また、gccでインタープリターをコンパイルする際に最適化オプションO1、O2、O3を付けた場合も測定している。表5.1は(+ x 3)を計算した実験結果である。なお、実行前にxは3に束縛している。

実験から、単純な式において、インタープリターのオーバーヘッドの除去によって最適化が可能なことを確認した。また、コードは省略するが、制御構造を含む複雑な式でも最適化は効果を示した(表5.2)。

表5.1: (+ x y)の実行結果

最適化オプション	O0	O1	O2	O3
実行時間(JITなし)[ns]	103	62	62	52
実行時間(JITあり)[ns]	31	27	27	27
JITコンパイル時間[μs]	139	142	142	106

表5.2: (if (- x 1) (+ x y) (* x y))の実行結果

最適化オプション	O0	O1	O2	O3
実行時間(JITなし)[ns]	642	363	335	264
実行時間(JITあり)[ns]	86	84	76	75
JITコンパイル時間[μs]	524	451	537	402

5.2　バイトコードインタープリタでの実験

バイトコードインタープリタとしてForth言語を解釈するインタープリタであるGforthを選び、実験を行った。forthは仮想スタックマシンの概念を用いた逐次型の手続き型言語である。逆ポーランド記法で演算を記述するので、構文解析が極めて単純であり、プログラムおよび処理系が小さくて済むという特徴がある。現在広く普及しているJava仮想マシンの仕様はforth仮想マシンの仕様をベースにしている。

我々は、以下の理由からgforthを実験対象として選んだ。

- forth言語はシンプルである

- Gforth は ANS Forth Standard に準じている
- オープンソースである
- 他の forth インタープリタよりも処理が効率的
- ポータビリティを考えて実装されており、Intel、PowerPC、Sparc、Alpha、Mips など多くのマシンで利用可能である

特に最後のポータビリティを考えて実装されており、マシン依存の部分と非依存の部分が明確に分離されているという点が、本研究の趣旨と合致している。

5.2.1 Threaded code interpreter

Threaded code [3] は Forth 言語で有名になったバイトコードインタプリタの最適化手法である。threaded code には複数のバリエーションがあるが、最も効率的とされているのは direct threading[1] である。

ナイーブなインタプリタにおけるディスパッチは以下の手順で行われる。図 2.1 がそのようなインタプリタの例である。

1. 次のバイトコード (オペコード) をフェッチ
2. フェッチしたコードに関連する実装を検索 (明示的にテーブルを検索することもあれば、switch 文によって暗黙のうちに検索されることもある)
3. 検索したアドレスに制御を移す

Direct threaded code は、このテーブルからの検索を除去することでディスパッチの速度を向上させる。具体的には、実行されるコードがオペコードの実装へのアドレスの列で表現される。そして、ディスパッチの際は、次のコードをフェッチして、そのアドレスに直接ジャンプする。

図 2.1 を、Direct threaded code で書き換えた実装が、図 5.3 である。

実行は、最初のオペコードの実装をコンパイルされたコードからフェッチし、そのアドレスにジャンプすることから始まる。それぞれのオペコードはそれが担当する処理を行った後、コンパイルされたコードで示される次のオペコードにディスパッチを行う。

threaded code を用いると、ナイーブなバイトコードインタプリタよりもオーバーヘッドが小さくなる。なぜならば、テーブルからオペコードの実装を検索をするコストと、ループの開始時点に制御を移すコストが省略されるからである。

5.2.2 Gforth 用 JIT コンパイラの開発

Gforth は、direct threaded code を用いて実装されている。Gforth のコアエンジンは、図 5.4(オペコード plus の実装) のようなコードが同じ関数内に並べられている。

命令ポインタを用いたディスパッチのためのコードが NEXT_P0、NEXT_P1、NEXT_P2 の 3 つの部分に分けられている。これは CPU のパイプラインを考慮して最適化を行った結果である。ベンチマークの結果を元に、これら 3 つのマクロに最適なコードが割り当てられる。

Gforth 用の JIT コンパイラに開発するには、この実装から不必要な部分を省略したテンプレートを記述すればよい。図 5.5 は、図 5.4 に示されたオペコードに対応するテンプレートの例である。デバッグ用の部分を省略し、さらに、ディスパッチのための部分も省略している。最適化コードではオペコードの実装が直接メモリ中に並べられるため、命令ポインタを用いたディスパッチを行う必要がない。なお、LABEL マクロの名前が本システムと衝突したため、ここでは JIT_LABEL という名前を用いた。
Gforth では、最適化のために特定のレジスタを指定している部分があり、そのレジスタが本システムのマクロが用いるレジスタと衝突する問題も発生した。そのため本システムの一部を書き換える必要があった。
forth ではルーチンのことをワードと呼び、ワードはディクショナリに登録される。我々が開発した JIT コンパイラでは、ルーチン単位の最適化を行い、最適化コードは拡張されたディクショナリの内部に登録するようにした。

5.2.3 実験結果

Intel Xeon CPU 3.06GHz x 2 (Memory 2GB)、Linux 2.6.17、gcc4.1.1、glibc 2.4 という実験環境で、Gforth 標準のベンチマークプログラムを用いて実験を行った。

表5.3が実験結果である。実行時間(JITなし)は、gforth-fast –no-dynamic -ss-number=0 で実行したものである。なお、Gforthインタプリタのコンパイルでは、我々の実験環境でconfigureを行った場合のデフォルトである -O2 -fomit-frame-pointer -force-addr -fforce-mem -march=pentium -fno-gcse -fno-strict-aliasing -fno-crossjumping -fno-defer-pop -fcaller-saves -fno-inline というオプションをそのまま利用している。これらの実験結果より、約2〜5倍の速度向上を確認した。

表 5.3: Gforthのベンチマーク

対象ベンチマーク	sieve	bubble	matrix	fib
実行時間 (JITなし)$[ms]$	476	630	773	710
実行時間 (JITあり)$[ms]$	229	317	165	187
JITコンパイル時間 $[ms]$	2.97	23.2	18.5	1.14

また、Gforth用JITコンパイラの開発にかかったコストは図5.4 のようになった。プリミティブ命令の数・行数に比べ、JITコンパイラ開発のために新たに書き加えた行数が少ないことから、少なくともコードの行数ベースでは、本システムによりJITコンパイラの開発が容易になることが確認できた。

表 5.4: Gforth用JITコンパイラの開発にかかったコスト

プリミティブ命令の数	431
インタープリタのプリミティブ命令の行数	17019
テンプレートのためにプリミティブ命令の実装からコピーした行数	4201
テンプレートのために新たに書き加えた行数	629
フロントエンドのためにインタープリタからコピーした行数	2355
フロントエンドのために新たに書き加えた行数	1348
インタープリタ本体に新たに書き加えた行数	306
JITコンパイラ開発のために新たに書き加えた総行数	2283(629+1348+306)

```
BlockCmpnt* analyze_eval(List* expr) {
  BlockCmpnt* cmpnt;
  switch (testElementType(expr)) {
  /* ... */
  case NUMBER :
    cmpnt = initByTemplate(v_code_NUMBER);
    addHoleInfo(cmpnt,
                v_code_NUMBER_hole,
                GetIntegerElement(expr));
    return cmpnt;
  case SYMBOL :
    cmpnt = initByTemplate(v_code_SYMBOL);
    addHoleInfo(cmpnt,
                v_code_SYMBOL_hole,
                GetSymbolElement(expr));
    return cmpnt;
 return cmpnt;
  case PLUS :
    cmpnt = initByTemplate(v_code_PLUS);
    addInnerBlock(cmpnt,
                  analyze_eval(GetSecond(expr)),
                  v_code_PLUS_section);
    addInnerBlock(cmpnt,
                  analyze_eval(GetThird(expr)),
                  v_code_PLUS_section);
    return cmpnt;
  /* ... */
  }
}
```

図 5.1: 抽象構文木インタープリターのフロントエンド

```
int* stackPointer;

void code_NUMBER() {
  int  ret_val;
  HOLE(code_NUMBER_hole, ret_val);
  *++stackPointer = ret_val;
}

void code_SYMBOL() {
  char* symbol;
  HOLE(code_SYMBOL_hole, symbol);
  *++stackPointer =
   getVarFromSymbol(symbol);
}

void code_PLUS() {
  LABEL(code_Plus_section);
  stackPointer--;
  *stackPointer = stackPointer[1];
}
```

図 5.2: 抽象構文木インタープリター用テンプレート

```
void *code[] = { ...,
  &&opcode_push3,
  &&opcode_push4,
  &&opcode_add, ... };


/* 次の命令をディスパッチ */
#define NEXT() goto **++instructionPointer

  void **instructionPointer = code - 1;
  /* 実行開始: 最初のオペコードをディスパッチ */
    NEXT();
  /* オペコードの実装 */
  opcode_push3:
    *++stackPointer = 3;
    NEXT();
  opcode_push4:
    *++stackPointer = 4;
    NEXT();
  opcode_add:
    --stackPointer;
    *stackPointer += stackPointer[1];
    NEXT();
  /* ... */
```

図 5.3: Direct threaded code

```
LABEL(plus) /* + ( n1 n2 -- n ) */
/*  */
NAME("+")
{
DEF_CA
Cell n1;
Cell n2;
Cell n;
NEXT_P0;              /* ディスパッチのためのコードその 1 */
vm_Cell2n(sp[1],n1);  /* スタックトップの次の値を n1 に代入 */
vm_Cell2n(spTOS,n2);  /* スタックトップの値を n2 に代入 */
#ifdef VM_DEBUG
if (vm_debug) {
fputs(" n1=", vm_out); printarg_n(n1);
fputs(" n2=", vm_out); printarg_n(n2);
}
#endif
sp += 1;
{
#line 681 "./prim"
n = n1+n2;
#line 2883 "prim.i"
}

#ifdef VM_DEBUG
if (vm_debug) {
fputs(" -- ", vm_out); fputs(" n=", vm_out); printarg_n(n);
fputc('\n', vm_out);
}
#endif
NEXT_P1;              /* ディスパッチのためのコードその 2 */
vm_n2Cell(n,spTOS);   /* n をスタックトップに代入 */
LABEL2(plus)
NEXT_P2;              /* ディスパッチのためのコードその 3 */
}
```

図 5.4: Gforth インタプリタからの抜粋

```
JIT_LABEL(plus) /* + ( n1 n2 -- n ) */
{
Cell n1;
Cell n2;
Cell n;
vm_Cell2n(sp[1],n1);  /* スタックトップの次の値を n1 に代入 */
vm_Cell2n(spTOS,n2);  /* スタックトップの値を n2 に代入 */
sp += 1;
n = n1+n2;
vm_n2Cell(n,spTOS);   /* n をスタックトップに代入 */
}
```

図 5.5: Gforth における add 演算のテンプレート

第6章　関連研究

Cコンパイラが生成した機械語を利用して最適化コードを生成するという技術は既に知られている。ここではそれらの関連研究との比較を行う。

6.1　基本的なアイデア

Rossi と Sivalingam は、[4] は、インタープリタの断片をコピーすることで機械語を動的に生成する方法を提案した。

direct threading with selective inlining[5] は、バイトコードインタープリタに対応するポータブルな JIT コンパイラを作成する方法を提案した。分岐命令を間に含まない複数のバイトコード命令の実装を結合することで、オーバーヘッドを削減する。複数の命令を包含する新たな命令を動的に生成するので、Dynamic Superinstruction と呼ばれている。インタープリタの本質的なオーバーヘッドを削減すること及びポータブルな JIT コンパイラを少ない労力で開発できるようにすることを主眼にしている点は本研究に近いが、定数のロードと制御フローに関する部分で効率のよいコードを生成できるという点で本システムの方が高度な最適化を行える。また、この手法は非バイトコードインタープリターには適用できない。

Vitale と Abdelrahman[6] は、インタープリタの実装を連結し、その穴を埋めることで最適化された機械語を生成するアイデアを、SPARC 上の Tcl 言語で実装した。本システムのように、命令ポインタを除去しないので、多くの命令キャッシュミスが発生してしまう。また、アセンブリコードに後処理を加えなければならない。

6.2　部分計算と動的特化技術

Tempo[7] と DyC[8] は、C 言語用の Run-time specializer である。Tempo は、テンプレートを動的に結合することで、実行時定数に特化したネイ

ティブコードを生成することができる。本研究よりも高い次元の記述でコードを生成することができるという長所はあるが、関数単位でしか特化できない。また、Tempo を用いた場合、JIT コンパイラはインタープリターから自動的に生成されるので、本フレームワークのように高度な最適化のためテンプレートレベルでの最適化を手動で組み込むということは困難である。DyC は、Tempo より低レベルなコードの操作を目的としている点で本研究に近いが、コンパイラをマシンアーキテクチャに応じて高度に拡張することを必要としているので、高いポータビリティを確保するのは困難である。

6.3　命令ポインタを除去する手法

Ertl らの JIT コンパイラの移植に関する研究 [9] は、命令ポインタを除去して複数のバイトコード命令を連結する手法を提案しており、本研究に最も近い。しかし、Ertl らはポータブルな JIT コンパイラの開発法を提案しているだけで、そのような JIT コンパイラを容易に開発するための方法は提案していない。本システムは JIT コンパイラのバックエンド、フロントエンド開発のための API、テンプレート作成のためのマクロを提供し、JIT コンパイラ全体を開発し、既存のインタープリタに組み込むことを容易にする。

6.4　ポータビリティを確保する他の手法

VCODE[10] と GNU Lightning は、インタープリタの実装をコピーするのではなく、独自のインターフェースを呼び出して動的にコードを生成する手法を採用している。コード生成が高速であるという長所があるが、本システムのように中間表現のセマンティクスを柔軟に定義ないので、フロントエンドの開発が複雑である。また、インタープリタの実装をコピーする方法に比べ、バックエンドでポータビリティを確保する労力が大きい。

6.5　バイトコードインタプリタを自動生成する研究

vmgen[13] は、簡便な記述からバイトコードインタープリタを自動する生成する方法を提案した。さらに、内山、緒方、脇田 [14] は、VM の仕様記述を厳密に定義することで、仕様記述上で命令合成することができるバイトコードインタプリタ生成系を提案した。仕様記述上の命令合成によって実装上の合成命令よりも効率的な合成命令を生成できるが、静的なコンパイルしか対応していない。

第7章　まとめ

本稿では、ポータブルな JIT コンパイラを開発するための使いやすいバックエンドについて提案した。このバックエンドを利用することで、インタープリタの開発者は、容易にポータブルな JIT コンパイラを開発することが可能である。具体的には、バイトコードまたは抽象構文木を中間表現に変換するフロントエンドと、その中間表現に意味を与えるテンプレートを作成すればよい。

フロントエンドに関しては、インタープリタの制御構造を再利用することが可能であり、本システムが提供する API を利用してオンメモリの中間表現を生成できるので、開発は容易である。また、テンプレートに関しても、インタープリタの実装を再利用することが可能であり、本システムが提供するマクロを利用することができるので開発は容易である。

参考文献

[1] M. Anton Ertl: A Portable Forth Engine. In *Proceegings of the euroFORTH '93*, pp. 253-257, (1993). http://www.complang.tuwien.ac.at/forth/threaded-code.html

[2] Charles H. Moore, Geoffrey C. Leach: FORTH - A Language for Interactive Computiong, Technical Report, Mohasco Industries, Inc., (1970). http://www.dnai.com/ jfox/F70POST.ZIP

[3] James R. Bell: Threaded Code, Communications of the ACM, 16(6) pp. 370-372, (1973).

[4] Markku Rossi, Kengatharan Sivalingam. A survey of instruction dispatch techniques for bytecode interpreters. Technical Report TKO-C79, faculty of Information Technology, Helsinki University of Technology, (1996).

[5] Ian Piumarta, Fabio Riccardi, Optimizing direct threaded code by selective inlining, *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 291-300 (1998).

[6] Benjamin Vitale, Tarek S. Abdelrahman: Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pp. 42—50, (2004).

[7] Francois Noel, Luke Hornof, Charles Consel, Julia L. Lawall: Automatic, template-based run-time specialization, Implementation and experimantal study. In *IEEE International Conference on Computer Languages (ICCL '98)*, pp. 123–142, (1998).

[8] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, Susan J. Eggers, DyC: an expressive annotation-directed dynamic compiler for C, Theoretical Computer Science, Vol. 248 No.1-2, pp. 147–199, (2000).

[9] M.Anton Ertl, David Gregg: Retargeting JIT compilers by using C-compiler generated executable code, *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT' 04)*, pp. 41–50, (2004).

[10] Dawson R. Engler. VCODE: A retargetable , extensible, very fast dynamic code generation system. In *SIGPLAN '96 Conference on Programming Language Design and Implementation* , pp. 160-170, (1996).

[11] M. Anton Ertl , David Gregg, Combining stack caching with dynamic superinstructions, *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators, June 07-07, 2004, Washington, D.C*, (2004).

[12] M. Anton Ertl, David Gregg: Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, (2003).

[13] M. Anton Ertl, David Gregg, Andreas Krall, Bernd Paysan : vmgen – A generator of efficient virtual machine interpreters. *Software–Practice and Experience*, vol. 32, No. 3, pp. 265-294 (2002).

[14] 内山 雄司, 緒方 大介, 脇田 建: 仮想機械の仕様記述に基づくバイトコードインタープリタ生成系. 情報処理学会 論文誌, Vol. 46, No. 6, pp. 1-17 (2005).