Schemeインタプリタにおける仮想マシンアーキテクチャの最適化

前田敦司[†] 山口喜教[†]

スタックアーキテクチャの仮想マシン命令を解釈実行するSchemeインタプリタの効率的な実装法について述べる．単純な命令セット定義から出発し，ベンチマークプログラムにおける命令列の実行頻度に応じて命令セットを最適化することで解釈実行のオーバヘッドを最小化する体系的な手法を示す．本手法を用いて構成したインタプリタの性能評価を行い，インタプリタの移植性，対話性，デバッグの容易さを損なうことなく，ネイティブコードの20〜40%程度の，比較的高い性能が得られていることを示す．

Optimization of Virtual Machine Architecture for a Scheme Interpreter

Atusi Maeda[†] and Yoshinori Yamaguchi[†]

An efficient implementation methodology for Scheme interpreters that interpretes stack-oriented virtual machine instructions is presented. Starting from simple instruction set definition, we show how our systematic optimization techniques reduces the interpretation overhead based on dynamic frequency of instruction sequence. Evaluation shows that our interpreter can achieve 20 to 40 percent of the performance of native code, without losing portability, interactivity, and ease of debugging.

1. はじめに

Webコンピューティングの隆盛にともない，各種のスクリプティング言語を用いたソフトウェア開発がさかんになってきている．インタプリタを用いたソフトウェア開発は，デバッグのサイクルが短く，高い生産性を得られる可能性がある．また，特定のマシンのネィティブコードを出力するコンパイラと異なり，インタプリタはCなどの高級言語で処理系をすべて記述できるため，処理系の移植性を高めるのが容易である．

一方で，ネイティブコードと比較した実行性能の低さが，依然としてその応用の範囲を制限していることもまた事実である．

インタプリタが実行のためのプログラムを保持する内部形式としては，文字列をほぼそのまま保持するものから，構文木あるいはそれに準ずる木構造で保持するもの，仮想マシンの機械語へコンパイルするものなどがあげられるが，実行効率の観点からは仮想マシンの機械語へコンパイルするものが有利である．

内部形式に仮想マシンの機械語を用いるインタプリタでは，入力ソースプログラムをコンパイルした後，C言語などで記述された仮想マシンがそのコンパイル結果を実行する．このような構成のインタプリタでは，命令セットの選択が実行効率に大きく影響すると思われるが，その指針を与えるような文献は少ない．

ソフトウェアで記述する仮想マシンにおいては，命令のフェッチとデコード（実行ルーチンへのディスパッチ処理）がそのまま実行のオーバヘッドとなるため，実行命令数を削減することが重要である．ソフトウェアで記述する仮想マシンでは，ハードウェアのCPUで実装が困難な複雑な操作まで含むような巨大な命令セットを用いることも可能である．このようなアプローチは，実行命令数を削減するという観点からは有利であるが，無制限に命令セットを拡張できるわけではなく，一定のトレードオフが存在する．命令セットを大きくしすぎると，平均の命令長が大きくなり，1命令あたりのフェッチやデコードの処理に時間がかかるおそれがある．また，ほとんど実行されることのない多数の実行ルーチンのためにインタプリタが巨大化し，起動に時間がかかるようになる可能性がある．スクリプト言語としての用途には，インタプリタの起動がなるべく軽いことが望ましい．

本稿では，最小限の命令セットを持つ仮想機械から

† 筑波大学電子・情報工学系
Institute of Information Science and Electronics, University of Tsukuba

出発して，なるべく恣意的でない合理的な基準によって命令セットを拡張し，効率の良い処理系を作成する手法について述べる．

この手法を用いる前提条件として，性能の基準として理想的なベンチマークプログラム（の集合）が存在することを仮定する．すなわち，そのプログラムでの性能が高ければ，一般のプログラムにおいても性能が高いといえるようなプログラムがあるものとする．

プログラミング言語としては Scheme[9] を対象とする．また，効率化の目的としては，仮想マシンの機械語プログラムの実行速度を第一義的な最適化の目標とする．インタプリタは C 言語で記述するものとし，なるべく多くの環境でそのまま適用できる手法を目標にするが，C 言語の規格準拠性には特にこだわらないこととする．

2. 背景と関連研究

2.1 ディスパッチ処理の重要性

インタプリタにおいても，最終的にはネイティブコードによる実行と同等の操作が実行される．1 章で触れたように，仮想マシン命令のフェッチと，実行ルーチンへのディスパッチの処理は，ネイティブコードでは必要のないインタプリタに固有の処理であり，たとえ実行コードの部分をネイティブコードとまったく等しくできたとしても純粋なオーバヘッドとして残る．このオーバヘッドの削減が，インタプリタの処理の効率化のためには決定的に重要である．

削減の方法としては，

- 1 回のディスパッチ処理を高速化する，
- 命令をフェッチする回数を削減する，

という 2 つのアプローチがある．

2.1.1 ディスパッチ処理の高速化

まず，ディスパッチ処理の高速化技法について述べる．C 言語のような高級言語でディスパッチ処理を実現する主な手法には以下のようなものがある（用語法は文献 3) による）．

(1) switch threading
C 言語の switch 文のような多分岐構文により，仮想マシン命令語をキーとして分岐する．

(2) call threading
各命令ごとの実行ルーチンを 1 つずつ別の関数としておき，仮想マシンの機械語として関数の入り口番地を用いる．C 言語では，関数へのポインタを配列の形で並べておき，インタプリタのメインループではポインタを 1 つ取り出して，その関数を呼び出すことでディスパッチ処理を行う．
この手法は，次の threaded code のポータブルな代用品といえる．

(3) (direct) threaded code[1]
この手法は，call threading と同様に実行ルーチンの入口番地を機械語のかわりに用いるが，サブルーチンコール命令で呼び出し，リターン命令で戻ってループの先頭に分岐するかわりに，実行ルーチンの末尾で次の処理ルーチンへ間接ジャンプを行う．他の手法のように，インタプリタの中心となるループは存在しない．

表 1 ディスパッチのコスト
Table 1 Dispatch cost.

手法	コスト（clock/回）
switch	8.01
call	10.01
direct	5.80

最近の CPU では，いずれの手法でも，1 回の分岐に 10 クロック程度の CPU サイクルを要するという報告がある[3]．Pentium III 933 MHz のマシンで実測したところ，**表 1** のような結果が得られた．

ここにあげた手法のうち，direct threaded code は標準 C の機能では実現できないが，さまざまなプラットフォームに移植されている C 言語処理系である GCC[6] では `void *`型の変数に `goto` ラベルを代入することができ，また変数の値を用いて間接ジャンプができるように言語仕様が拡張されている．現在のところ，この方法が最も広い環境で効率の良いディスパッチを実現できる手法と思われる．

2.1.2 命令数の削減

利用可能な最も高速な手法である direct threade code を用いてもなお 1 命令のディスパッチあたりに 6 クロック程度を要するということは，仮想マシンのアーキテクチャなどの工夫により実行ルーチンを極限まで最適化できたと仮定して，ネイティブコードプログラムで 1 命令にあたる処理を実行ルーチンでまったく同じ実マシンの 1 命令で処理できたとしても，インタプリタの性能の上限がネイティブコードの 1/7 程度ということになってしまうということを意味する．スーパスカラ CPU が一般的なものとなり，ネイティブコードの 1 クロックあたりの実行命令数が 1 を超えている現在では，問題はさらに深刻である．なぜなら，インタプリタの 1 つの命令の実行ルーチンと次の命令の実行ルーチンの間には，分岐予測が成功する可能性の低い多方向分岐があるため，インタプリタの実行ルーチンが各々1 命令からなっていたとすると，そ

れらが並列実行される可能性はきわめて低いと考えられるためである．

ディスパッチのオーバヘッドの差を超えてインタプリタを高速化するためには，実行される命令数を削減する必要がある．このためには，実行時によく現れる命令の並びをまとめて，より高機能な命令を設けることが考えられる．これにより，ディスパッチのオーバヘッドを削減するとともに，1つの実行ルーチンに含まれる命令数を増やすことによってコンパイラやCPUが命令のスケジューリングや最適化を行う機会を増やすことができる．

この方法は instruction merging[13]，instruction compression[11]，superinstruction[5] など，さまざまな名前で呼ばれている．

本稿では，複数の命令の頻出する列を新たな単一の命令とすることを**命令の融合**と呼ぶことにする．

2.2 その他の性能決定要因

スタックマシンアーキテクチャでは，式の演算に対してスタックトップのメモリアクセスの回数が増える傾向がある．実マシンの場合はスタックトップをCPU上にキャッシュしたり[11]，複数の命令を動的にまとめて無駄なロード・セーブを省く[12] などの手法がとられる．ソフトウェアで実現した仮想マシンの場合には，スタックレベルごとに別の命令実行ルーチンを用意して，スタックトップを実マシンのレジスタにキャッシュする手法がある．[4]

先に述べた命令の融合により複数の命令をまとめた場合，スタックトップへのロードやセーブを削除できる可能性がある．

また，仮想マシン命令がオペランドを持つ場合，そのフェッチ回数を削減できれば性能を向上できる．これについては3.2節で議論する．

3. 命令セットの最適化

仮想マシンに基づいたインタプリタを設計する場合に，命令セットにはさまざまな選択肢がありうる．スタックマシンアーキテクチャに限った場合でも，純粋なゼロオペランド命令セット以外に，データのロードとオペレーションを融合した命令をあらかじめ用意しておくことも考えられる．

また，条件分岐命令についても，条件のテスト命令がスタックに積んだ真偽値によって分岐する2つの分岐命令 br-true，br-falseだけを用意する最小限のもののほか，条件のテストと分岐をまとめた多数の分岐命令を用意することも考えられる．

そのほか，オペランドのうちで頻繁に現れる値や範囲に対して特別に速い命令を用意することも考えられる．たとえば，定数0をロードする命令が頻繁に現れるならば，汎用の定数ロード命令に加えて0だけをロードする命令を別に用意すると，オペランドのワードを読み出す手間が省けるぶんだけ実行を高速化できる可能性がある．

表 3 レジスタセット
Table 3 Register set.

sp	値スタックポインタ
esp	制御スタックポインタ
pc	プログラムカウンタ

先に述べたとおり，効率の点からは可能な限り多くの命令を融合した命令を用意しておき，ディスパッチ回数やスタックアクセス回数の削減を図ることが望ましい．しかし，インタプリタが用意できる命令数には限界があり，また命令セットの巨大化にはドローバックもあることから，どの命令を融合しておくかの選択が性能を左右することになる．

以下では，最小限の命令セットから出発して，Schemeにおける命令の実行頻度に最適化されたインタプリタを構成する手順について述べる．

3.1 基本命令セット

Scheme仮想マシン構築の出発点となる基本命令セットを**表 2** に示す．このマシンは制御スタックと値スタックの2つのスタックを持つスタックアーキテクチャの仮想マシンで，**表 3** に示す内部レジスタ（C言語の変数）を持つ．2章で述べた direct threaded code を使うことを前提とし，各命令語およびオペランドはそれぞれ1ポインタワードを占めるものとする．

関数呼び出しの際，引数は第1引数から順にスタックに積まれる．制御スタックには，関数呼び出しから戻った際の sp と pc の値が積まれる．局所変数の値は sp からのオフセットで参照される☆．

call 命令は，スタックトップの関数オブジェクトの仮引数の数と *nargs* オペランドが示す実引数の数が一致するかのチェックを行った後，引数を除いた残りの sp の値と pc を制御スタックに積み，関数の入り口へ制御を移す．return 命令は単に pc と sp の値を制御スタックから取り出すことによって呼び出し元へ復帰する．call-tail 命令はレジスタを保存しない点を，また call-local 命令は関数の入り口番地をオペランドから直接得て引数のチェックを行わない点をそれぞれ除けば，動作は call 命令と同じである．

☆ 実際の実装では，スタックトップの値のみをメモリでなく acc というレジスタに保持している．したがって，(vref 0) は acc の内容を参照することを意味する．

表 2 基本命令セット
Table 2 Baseline instruction set.

命令	意味
(vref *n*)	スタックトップから n 番目の値をスタックに積む.
(quote *const*)	即値定数 *const* をスタックに積む.
(qref *n*)	定数表の *n* 番目の値をスタックに積む.
(br *label*)	無条件で *label* に分岐する.
(br-false *label*)	スタックトップの値をポップし，偽 (#f) なら *label* に分岐する.
(br-true *label*)	スタックトップの値をポップし，偽 (#f) でないなら *label* に分岐する.
(return)	関数呼び出しから復帰する.
(check-stack *n*)	スタックに *n* ワードの空きがあるかどうかチェックする.
(call *nargs nsave*)	スタックトップの関数オブジェクトを *nargs* 引数で呼び出す.
(call-tail *nargs*)	スタックトップの関数オブジェクトを *nargs* 引数で末尾再帰的に呼び出す.
(call-local *label nargs nsave*)	*label* からはじまるローカル関数を *nargs* 引数で呼び出す.
(make-procedure *label nvars nargs has_rest?*)	*label* からはじまりスタックに積まれた *nvars* 個の値を閉じ込めた *nargs* 引数の関数オブジェクトを作る.
(pop)	スタックトップの値をポップして捨てる.
(discard *n*)	まず値をスタックトップからポップし，次にスタックの先頭の *n* ワードを捨てた後，最初にポップした値を積む.
(gref *sym*)	シンボル *sym* のトップレベル環境での値を積む.

Scheme の組み込み関数は C 言語で記述し，関数へのポインタを保持した関数オブジェクトを変数名シンボルの大域変数フィールドに入れておく．すなわち，関数オブジェクトには threaded code を指すものと C 言語の関数を指すものの 2 種が存在する．この判別も call（call-tail）命令が行う．

例として，cons 関数を呼び出す命令列は，

```
<第 1 引数のコード>
<第 2 引数のコード>
(gref cons) ; シンボル cons の値
(call 2 x)
```

のようになる．

call 命令および call-local 命令の最後の引数は，現在スタック上で生きている変数の数を意味し，命令実行の際には用いない．後述するスタック GC ルーチンで用いられる．

コンパイラは Clinger らの TWOBIT コンパイラ[2] を簡略化したもので，Scheme のソースコードを 3 パスで仮想マシンコードに変換した後，主として制御の最適化のためにピープホール最適化を行う．ソースレベルでの最適化は，使用されていない変数束縛の削除を除いて現時点ではほとんど何も行っていない．

ピープホール最適化は文献 15) にあげられている condition reversal，cross jumping，branch chaining などの変換をテーブル駆動によって行う．特に，Scheme 言語ではスタックを消費せずに末尾再帰呼び出しを行うことが必須であるが，これはピープホール最適化処理で call 命令と return 命令の並びを call-tail 命令に変換することによって行う．return 命令への無条件分岐は return 命令自身に変換され，また return 命令の直前のスタックレベルを合わせる discard 命令も除去されるので，すべての末尾再帰呼び出しを正しく検出することができる．末尾再帰的な call-local 命令は br 命令に変換される．

コンパイラの行う代入変換（assignment elimination[2]）処理によって，代入される局所変数は，ヒープ上に置かれる cell に実際の値を保持し，変数への参照と代入はそれぞれ cell-ref および cell-set!という関数への呼び出しに置き換えられる．すなわち，

```
(let ((v value))
  (set! v new-value)
  ... v ...)
```

という式は，

```
(let ((v (make-cell value)))
  (set-cell! v new-value)
  ... (cell-ref v) ...)
```

という式に変換して処理される．

また，大域変数への代入も同様に global-set! 関数の呼び出しに変換される．このため，基本命令セットには代入のための命令を含まない．

代入変換により，束縛が共有されているかどうかにかかわらず変数の値を自由にコピーできるようになる．make-procedure 命令が関数オブジェクトを生成する際には，現在の環境で束縛されている変数の値をヒープにコピーする．関数オブジェクトが呼び出されたときには，関数オブジェクト内に保持されている値をスタックにコピーし，make-procedure 命令の実行時と

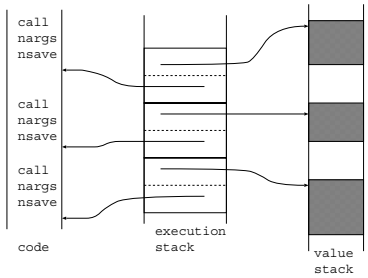

図 1 スタックのレイアウト
Fig. 1 Stack layout.

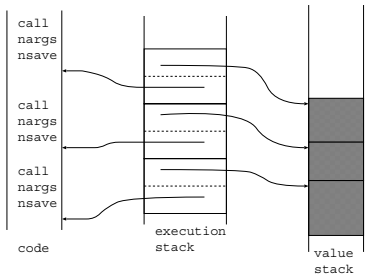

図 2 スタック GC 後
Fig. 2 Stack after stack-gc.

同じ状態をスタックトップに作り出してから関数本体のコードを実行する．

call-tail 命令および br 命令で関数を末尾再帰的に呼び出すとき，スタックに積まれた引数は return 命令が実行されるまで捨てられることがない．末尾再帰を繰り返すプログラムでは，スタックが無制限に伸びることになる．check-stack 命令がスタックのあふれを検出すると，スタック GC[10] を行い，スタック中の不要な値を取り除く．**図 1** は実行時のスタックの様子を示している．制御スタックには，call 命令の次の命令ワードのアドレスが入っており，その 1 ワード前には，値スタック上で生きている変数のワード数が call 命令の nsave オペランドとして記録されている．スタック GC はこの情報を用いて，生きている変数のみが残るように値スタックを詰める（**図 2**）．

表 2 に示した 15 種の基本命令セットだけでは，それほど実行効率の良いインタプリタにはならない．**図 4** に，TAKL ベンチマーク中の shorterp 関数（**図 3**）をコンパイルした結果を示す．空リストのテストを行うためにさえ，いちいち関数オブジェクトをスタックに積んで関数呼び出しを行っており，その際には引数の個数チェックなども行われる．

この命令セットを持つインタプリタ（basic）において TAK ベンチマークを行った実行時間を他の処理系と比較した結果を**表 4** に示す．basic のコンパイルに

```
(define (shorterp x y)
  (and (not (null? y))
       (or (null? x)
           (shorterp (cdr x)
                     (cdr y)))))
```

図 3 shorterp プログラム
Fig. 3 shorterp program.

```
(
        (qref 54)       ;shorterp
        (make-procedure L774lambda 0 2 #f)
        (gref global-set!)
        (call-tail 2)   ;tail-merge
(label L774lambda)
        (check-stack 3)
        (vref 1)        ;x
        (vref 1)        ;y
(label L772lambda)
        (check-stack 5)
        (vref 0)        ;y
        (gref null?)
        (call 1 3)
        (br-false L776)
        (quote #f)
        (return)
(label L776)
        (vref 1)        ;x
        (gref null?)
        (call 1 3)
        (vref 0)        ;v
        (br-false L778)
        (return)
(label L778)
        (vref 2)        ;x
        (gref cdr)
        (call 1 4)
        (vref 2)        ;y
        (gref cdr)
        (call 1 5)
        (br L772lambda)
)
```

図 4 shorterp のコンパイル結果（基本命令セット）
Fig. 4 shorterp compilation result (baseline instruction set).

表 4 TAK 実行時間
Table 4 TAK execution time.

処理系	時間 (ms)
basic	13.1
petite	22.7
scm	66.1
C	1.3

は gcc 2.96 を用いた．petite は Petite Chez Scheme 6.0a で，scm は scm5d2 で，同じプログラムをそれぞれ実行した結果である．ドキュメントによれば Petite Chez Scheme は threaded code を用いているとの記述があるが，詳細は不明である．また scm は switch threading を用いてバイトコードを解釈実行する処理系である[8]．C は，同等のプログラムを C で記述し，gcc 2.96 に`-O4 -fomit-frame-pointer` オプションを与えてコンパイルした結果を実行したものである．petite と scm の実行結果は処理系に内蔵の計時機能により，プログラムを 100 回実行した結果の平均値である．C および basic の計時は，Intel CPU の TSC レジスタを読み出して得たサイクル数をクロック周波数で割って得た．実行したマシンは Pentium III 933 MHz（SDRAM 1 GB），OS は Linux 2.4.9 である．

basic インタプリタの性能はすでに他の Scheme 処理系を上回っているが，C コンパイラが生成したネイティィブコードとは 1 桁の速度差がある．

3.2 命令セットの改良

基本命令セットを持つインタプリタに対して，ベンチマークテストの結果に基づいて命令セットに改良を加えることによってインタプリタの効率化をはかる．

すでに述べたとおり，命令の融合や特定のオペランドだけを別命令にする操作を，あるベンチマークの結果に対して繰り返していく．すなわち，

- **命令の融合**
  頻度の高い (op1 ...) (op2 ...) という命令列を (op1_op2_...) という 1 命令にする．
- **オペランドの融合**
  頻度の高い (op *operand* ...) 命令を (op_*operand* ...) というオペランドの少ない命令にする．

という手順を繰り返す．本稿では，これら 2 つの操作をまとめて融合操作と呼ぶことにする．

さらにここでは，Scheme の組み込み関数を命令セットに加えることも考慮する．このための準備として，gref 命令のうち，組み込み関数名シンボルをスタックに積むものを gref&，その直後に続く call，call-tail 命令をそれぞれ call& および call-tail&と呼び替える．呼び替えただけで命令の動作は同じであり，実行ルーチンは同じものを用いる．

表 5 basic インタプリタの実行クロック数
Table 5 Clock cycles of basic interpreter.

ベンチマーク	仮想命令数	実クロック数 ($\times 10^6$)
browse	4,660,927	19.5
cpstak	1,399,447	69.6
dderiv	4,318,391	108.2
deriv	3,528,067	90.5
destruct	5,415,249	114.3
div	4,490,356	95.1
tak	858,768	13.0
takl	9,843,490	130.1
boyer	19,913,293	432.5
takr	1,177,302	20.4
計	55,605,290	1,093.5

ベンチマークテストには Gabriel ベンチマーク[7]の William Clinger による Scheme 版のうちの browse, cpstak, dderiv, deriv, destruct, div, tak, takl, boyer, takr の 10 個のプログラムを用い，これらのプログラムの総実行時間を最小化することを目的として最適化を行う．

basic インタプリタでこのベンチマークで実行した場合に実行される仮想マシン命令の数および実マシン（Pentium III）のクロックサイクル数を**表 5** に示す．ただし，数値には関数定義や定数の準備のための実行サイクルを含む．また，実マシンクロック数は 5 回の平均値で，ガーベジコレクションのためのサイクル数を含む．

このベンチマークの実行の際の，すべての基本ブロックの実行回数をカウントし，それに基づいて任意の長さの命令列の動的な出現頻度の統計をとる．ここでいう基本ブロックとは，ラベル・call 命令・call-local 命令の後から，最初に現れるラベル・関数呼び出し・分岐命令・return 命令まで（これらを含む）命令の並びとする（call& および call-tail& 命令は，基本ブロックの区切りと見なさない）．

具体的には，各基本ブロックのすべての部分命令列について実行回数を重みとして与え，プログラムの複数個所で同じ形の命令列が現れている場合には，すべての個所における重みの合計をその命令列の重みとする．

たとえば，

```
(vref 0)
(quote 1)
(vref 0)
```

という命令の並びを持つ基本ブロックが 100 回実行された場合，

- (vref 0) (quote 1) (vref 0) という列が 100 回
- (vref 0) (quote 1) という列が 100 回
- (quote 1) (vref 0) という列が 100 回
- (vref 0) が 200 回
- (quote 1) が 100 回

それぞれ実行されたものとカウントする．

このように重みをつけた長さ 1 以上のすべての命令列のうち，最も頻度の高いものを融合の対象とする．オペランドについては，ラベル・即値でない定数・gref 命令に現れるユーザ定義のシンボルを除いた第 1 オペランドを融合の対象とする（gref& 命令のオペランドに現れる組み込み関数名シンボルは対象になる）．

こうして得られた最も頻度の高い命令列を新たな命令として命令セットに加える．その結果の命令列が新たな命令に置き換えられると以下の順位は変動する可能性があるので，ベンチマークプログラム全体に対して置き換えを行ったのち，あらためて頻度を数え直す．これを繰り返すことによって重要な命令から順に命令セットを拡張していくことができる．

このアルゴリズムによって重要と判断された命令を**表 6** に示す．691 回この操作を繰り返すことによって，1 回でも実行されたすべての命令列を融合し終えた．

融合を繰り返したとき，実行される命令数とフェッチされるオペランドのワード数が減っていく様子を**図 5** に示す．

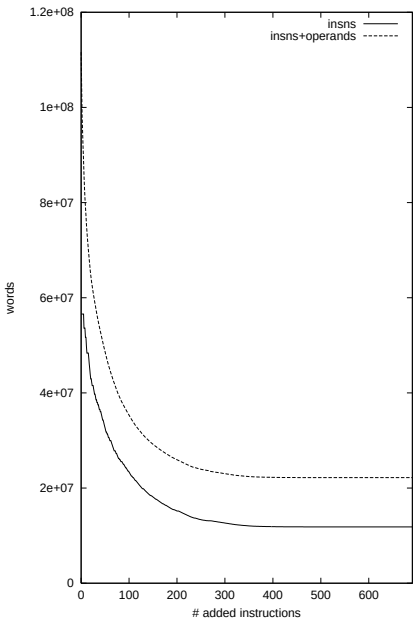

図 5 融合回数と命令・オペランドの数
Fig. 5 Number of fusions vs. instruction and operand words.

4. 評　　価

融合によって得られた新たな命令を加えたインタプリタを作成し，ベンチマークの速度変化を調べた結果を**図 6** に示す．ここでは最終的な命令数が 64 になるまで融合を続け，プログラマが手動でインタプリタの実行ルーチンを追加し，実行ルーチンから無駄なスタックアクセスを省くなどの最適化を行った．

また，32 命令（17 命令追加）の場合と，64 命令（49 命令追加）の場合の各ベンチマークの実クロックサイクル数を**表 7** に示す．

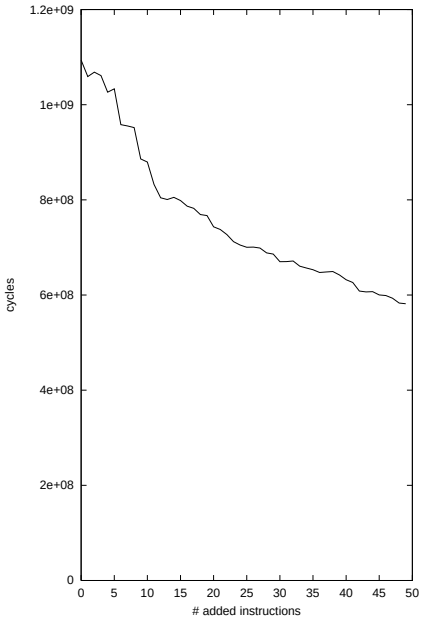

図 6 融合回数と実クロックサイクル数
Fig. 6 Added instructions vs. real clock cycles.

5. 考　　察

表 7 に示したとおり，すべてのベンチマークにおいて速度の向上が得られた．特に takl ベンチマークでは性能向上が顕著で，basic インタプリタと比較して 64 命令の場合で 6.9 倍もの速度向上を得ている．32 命令および 64 命令の命令セットを用いて takl ベンチマーク中の shorterp をコンパイルした結果をそれぞれ**図 7** および**図 8** に示す．64 命令の場合では，ほとんどすべての基本ブロックが 1 命令にまで融合されており，本手法における限界近くまで性能が向上していることが分かる．

TAK ベンチマークではクロック数が 53%にまで減少しており，これは C 言語の 20%程度の性能に相当する．出力コードを調べると，shorterp ほど極端な融

表 6 重要な命令（上位 20）
Table 6 Top 20 important instructions.

順位	元の命令列	頻度	新命令
1	(call& 1)	7343981	(call&_1)
2	(vref 1)	5939664	(vref_1)
3	(vref 0)	3454368	(vref_0)
4	(call& 2)	3063918	(call&_2)
5	(gref& cdr)	2978806	(gref&_cdr)
6	(gref&_cdr) (call&_1)	2969294	(gref&_cdr_call&_1)
7	(vref 2)	2834495	(vref_2)
8	(gref& null?)	1990856	(gref&_null?)
9	(gref&_null?) (call&_1)	1941856	(gref&_null?_call&_1)
10	(gref& car)	1789743	(gref&_car)
11	(gref&_car) (call&_1)	1789743	(gref&_car_call&_1)
12	(gref&_null?_call&_1) (br-false _)	1506104	(gref&_null?_call&_1_br-false _)
13	(gref& cons)	1368036	(gref&_cons)
14	(quote #f)	1212885	(quote_f)
15	(check-stack 5)	1206091	(check-stack_5)
16	(vref_0) (gref&_null?_call&_1_br-false _)	1143299	(vref_0_gref&_null?_call&_1_br-false _)
17	(quote_f) (return)	1138380	(quote_f_return)
18	(vref_1) (gref&_cdr_call&_1)	1087793	(vref_1_gref&_cdr_call&_1)
19	(vref_2 gref&_cdr_call&_1)	1045962	(vref_2_gref&_cdr_call&_1)
20	(gref&_cons call&_2)	996442	(gref&_cons_call&_2)

表 7 インタプリタの実行クロック数（$\times 10^6$）
Table 7 Clock cycles of interpreters.

ベンチマーク	基本 15 命令	32 命令	64 命令
browse	19.5	13,0	11.3
cpstak	69.6	68.5	63.9
dderiv	108.2	86.7	68.3
deriv	90.5	70.5	56.7
destruct	114.3	96.7	50.8
div	95.1	50.1	35.2
tak	13.0	10.9	6.9
takl	130.1	42.9	18.8
boyer	432.5	322.6	255.1
takr	20.4	19.0	13.9
計	1093.5	781.5	581.0

合は行われておらず，まだ速度向上の余地は残されていた．さらに融合した場合と同等の threaded code を手で書いて計測した場合，C 言語の 45%程度の速度（2.9 ms）が得られている．

図 8 の結果を見ると，あまりに特定のプログラムに依存しすぎているのではないかという疑問が生じる．本手法の最終的な目的は，特定のプログラムに特化したインタプリタを得ることではなく，Scheme 言語一般について高い処理性能を発揮するインタプリタを得ることであるが，ベンチマークスイートはまさにそういった基準となることを目的として作られていることを考慮すれば，適切なプログラムを用いることによりこの最適化手順によって Scheme プログラム一般について性能向上が得られるはずである☆．

融合の行われる順序がベンチマークプログラムに

```
(
        (qref 54)
        (make-procedure l774lambda 0 2 #f)
        (gref& global-set!)
        (call-tail& 2)
(label l774lambda)
        (check-stack 3)
        (vref_1)
        (vref_1)
(label l772lambda)
        (check-stack_5)
        (vref_0_gref&_null?_call&_1_br-false l776)
        (quote_f_return)
(label l776)
        (vref_1)
        (gref&_null?_call&_1)
        (vref_0)
        (br-false l778)
        (return)
(label l778)
        (vref_2)
        (gref&_cdr_call&_1)
        (vref_2)
        (gref&_cdr_call&_1)
        (br l772lambda)
)
```

図 7 shorterp のコンパイル結果（32 命令）
Fig. 7 shorterp compilation result (with 32 instructions).

☆ ただし，今回用いたこれらのプログラムがこのような理想的な性質を備えていると主張するものではない．

```
(
        (qref 54)
        (make-procedure l774lambda 0 2 #f)
        (gref& global-set!)
        (call-tail&_2)
(label l774lambda)
        (check-stack_3_vref_1_vref_1)
(label l772lambda)
        (check-stack_5_vref_0_gref&_null?_call&_1_br-false l776)
        (quote_f_return)
(label l776)
        (vref_1_gref&_null?_call&_1_vref_0_br-false l778)
        (return)
(label l778)
        (vref_2_gref&_cdr_call&_1_vref_2_gref&_cdr_call&_1)
        (br l772lambda)
)
```

図 8 shorterp のコンパイル結果（64 命令）
Fig. 8 shorterp compilation result (with 64 instructions).

表 8 重みを変えたときの順位変動
Table 8 Change of importance with different weight.

順位	等しい重みの場合	boyer の重みを半分にした場合
14	(quote_f)	(check-stack_5)
15	(check-stack_5)	(vref_2_gref&_cdr_call&_1)
16	(vref_0_gref&_null?_call&_1_br-false _)	(quote_f)
17	(quote_f_return)	(vref_0_gref&_null?_call&_1_br-false _)
18	(vref_1_gref&_cdr_call&_1)	(vref_1_gref&_cdr_call&_1)
19	(vref_2_gref&_cdr_call&_1)	(quote_f_return)
20	(gref&_cons_call&_2)	(gref&_cons_call&_2)

よってどのくらい敏感に変わるかを調べるため，ベンチマークスイート中で実行される仮想命令数が最も多い boyer の重み（ベンチマーク全体の 35.8%）を半分にしたときに，融合の順序がどのように変わるかを調べた．その結果，本手法で重要と判定した順位の 1 位から 13 位までは表 6 とまったく同じであったが，14 位以降に多少の順位の変化が見られた．等しい重みで計算した場合と，boyer の重みを半分にした場合の，14 位から 20 位までの順位の変化を**表 8** に示す．

この程度のベンチマークスイートの変化では，順位が上下に 2 つ程度ずれることはあるものの，融合する命令数が同じならば，ほぼ同じ命令列が融合対象として選ばれることが分かる．実際，49 位まで融合を進めた場合でも，重みの変化によって融合されなくなった命令列は，等しい重みの際の順位で 46 位および 47 位の命令列のみであり，かわりに融合された命令列は 50 位および 51 位の命令列であった．

さらに，ベンチマークプログラムの偏りをできるだけ排除するためには，たとえば (vref n) という命令のオペランドを融合する際に，n 以下のオペランドはすべて融合するというように，戦略を一般化することも考えられる．

次に，いくつかの改良すべき点について述べる．

5.1 基本ブロックの定義

図 8 の結果を見ると，return 命令で終わる基本ブロックについてはあまり融合がなされない短い状態のままとなっている．これは実は，3.2 節であげた基本ブロックの定義に問題があったことによる．融合した命令の途中に飛び込むことはできないため，ラベルや呼び出し命令を越えて融合を行うことはできないが，条件分岐命令やリターン命令を融合の区切りにする必要はなかったのである．条件分岐やリターンも融合の対象にしたとすると，最終的には shorterp のループ全体が単一の命令になるまで融合を進めることが可能となる．これは結局，shoreterp の Scheme プログラム全体が C 言語で記述されて，インタプリタの中に取り込まれることを意味する．

5.2 命令とオペランドの重み

今回用いたアルゴリズムでは，命令のディスパッチ回数とフェッチするオペランドのワード数を同じ重みと見なしているが，厳密にはこれは正しくない．実マシンでの計測を行って正しい重みを与えれば，より有効にオーバヘッドを削減することが可能となると思われる．

5.3 命令順位の逆転

表 6 の 10 位に現れている

```
(gref& car)
```

と 11 位の

```
(gref&_car) (call&_1)
```

は頻度がまったく同じであることから，前者は実は後者の一部としてのみ現れるということが分かる．このような場合，11 番目の新命令 (gref&_car&_call_1) を加えた後では，10 番目に加えた (gref&_car&) はまったく使われなくなる．

この単純な場合については，2 段階の命令追加を一度にすませることが容易にできるが，より一般に，当初の重みづけで上位に来た命令列が，後に他の命令をいくつか加えることでほとんど使われなくなることがありうる．たとえば 5 位の cdr と 6 位や，8 位の null? と 9 位もそうである（これらで頻度がまったく同じでないのは，call& でなく call-tail& が後に付く場合があるからである）．

このように，順位が後になって逆転する場合まで考慮した最適な命令セットの選択については今後の課題としたい．

また，これらの計測プロセスからインタプリタの生成までをさらに自動化する仕組みについて検討を行っている．その際には，Scheme 言語のみならず，できるだけ広い範囲に適用できるよう，仮想マシンのアーキテクチャに柔軟性を持たせる必要がある．

6. 他の手法との比較

本稿で述べた手法と同様に，プロファイル結果から命令の融合を行い，インタプリタを作成するためのシステムに vmgen [5] がある．基本命令の実行ルーチンの C 言語記述を処理系作成者が与えると，vmgen はその記述からマクロ展開と最適化を行うことによって，インタプリタの C プログラムを自動生成する．その際，与える記述中には単純な命令を組み合わせた命令（superinstruction）の定義を記述することができる．vmgen ではプロファイル機能が提供されており，最初の基本的なインタプリタの実行結果から，加えるべき superinstruction を人間が選択することができる．しかし，どの superinstruction を加えるかの選択は人間にまかされている．実験において著者らは，ある決められた長さ（2〜4）までの命令列について，一度でも実行された組合せをすべて superinstruction として追加している．本稿で述べた手法は長さを制限する必要がなく，また加えるべき命令について判断する基準を与えることができる．また，vmgen はオペランドの頻度については扱っていない．

Piumarta らは文献 14) で，direct threaded code を用いた仮想マシンの実行時に，頻出する仮想マシン命令の列を検出し，新たな threaded code 実行ルーチンを自動的に生成して仮想マシン命令を動的に追加するシステムについて述べている．新たな実行ルーチンの追加時には実マシンの機械語プログラムをコピーする必要があり，その際に相対ロード命令や絶対ジャンプ・絶対コール命令を含んでいないかなどの判別が必要となる．これはインタプリタの技法というよりも，特定のマシンに依存した JIT コンパイラの作成技法と考えるべきであると思われる．

7. おわりに

Direct threaded code を用いた Scheme 言語インタプリタの仮想マシンについて，ごく単純な命令セットから出発して，体系的に命令を追加することによってオーバヘッドを削減する手法について述べた．

実験を通じた評価により，この手法を用いて作成した処理系は，既存のインタプリタを上回り，ネィティブコードの数分の 1 程度の高い性能を発揮することを示した．

この命令追加の手順それ自体は，特に Scheme 言語に依存しておらず，仮想マシンを用いるインタプリタ一般に適用することが可能である．

参考文献

1) Bell, J.R.: Threaded code, *Comm. ACM*, Vol.16, No.6, pp.370–372 (1973).
2) Clinger, W. and Hansen, L.T.: Lambda, the ultimate label or a simple optimizing compiler for Scheme, *ACM Conference on Lisp and Functional Programming* (1994).
3) Ertl, M.A.: Threaded Code. http://www.complang.tuwien.ac.at/forth/threaded-code.html
4) Ertl, M.A.: Stack Caching for Interpreters, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.315–327 (1995).
5) Ertl, M.A., Gregg, D., Krall, A. and Paysan,

B.: Vmgen — a generator of efficient virtual machine interpreters, *Software — Practice and Experience*, Vol.32, No.3, pp.265–294 (2002).
6) Free Software Foundation: *Using and Porting the GNU Compiler Collection* (*GCC*), Cambridge, MA, for gcc ver. 2.96 edition (1999).
7) Gabriel, R.P.: *Performance and evaluation of LISP systems*, MIT Press (1985).
8) Jaffer, A.: SCM. http://swissnet.ai.mit.edu/~jaffer/SCM.html
9) Kelsey, R., Clinger, W. and Rees, J. (Eds.): Revised^5 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices*, Vol.33, No.9, pp.26–76 (1998).
10) Kelsey, R.A.: Tail-Recursive Stack Disciplines for an Interpreter, Technical Report NU-CCS-93-03, College of Computer Science, Northeastern University (1993).
11) Koopman, P.J.: *Stack Computers: The New Wave*, Ellis Horwood, West Sussex, England (1989).
12) McGhan, H. and O'Connor, M.: PicoJava: A Direct Execution Engine For Java Bytecode, *Computer*, Vol.31, No.10, pp.22–30 (1998).
13) Nässén, H., Carlsson, M. and Sagonas, K.: Instruction Merging and Specialization in the SICStus Prolog Virtual Machine, *ACM-SIGPLAN Conference on Principles and Practice of Declarative Programming* (*PPDP 2001*), pp.49–60, ACM (2001).
14) Piumarta, I. and Riccardi, F.: Optimizing Direct-threaded Code by Selective Inlining, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.291–300 (1998).
15) Wulf, W., Johnsson, R., Weinstock, C., Hobbs, S. and Geschke, C.: *The Design of an Optimizing Compiler*, North Holland (1975).

(平成 14 年 12 月 24 日受付)
(平成 15 年 7 月 1 日採録)

前田 敦司（正会員）

1994 年慶應義塾大学大学院理工学研究科数理科学専攻単位取得退学．博士（工学）（慶應義塾大学 1997 年）．同年電気通信大学大学院情報システム学研究科助手．2000 年筑波大学電子・情報工学系講師（現職）．並列/分散処理，コンピュータアーキテクチャ，プログラミング言語の実装，ガーベッジコレクション等に興味を持つ．日本ソフトウェア科学会，ACM 各会員

山口 喜教（正会員）

1972 年東京大学工学部電子工学科卒業．同年通商産業省工業技術院電子技術総合研究所入所．計算機方式研究室長等を経て，1999 年筑波大学電子・情報工学系教授，工学博士．高級言語計算機，並列計算機アーキテクチャ，並列実時間システム等の研究に従事．1991 年情報処理学会論文賞，1995 年市村学術賞受賞．著書「データ駆動型並列計算機（共著）」．IEEE Computer Society，ACM，電子情報通信学会各会員．