末尾再帰の最適化と一級継続を実現するためのJVMの機能拡張

山本晃成[†] 湯淺太一[††]

プログラミング言語の中には末尾再帰の最適化と一級継続を必須としているものがある．たとえばScheme，Standard MLや他の関数型言語のほとんどがその機能を必要としている．これらの言語は，末尾再帰の効率によるところが大きく，また，継続が操作可能であることが言語の重要な特色でもある．しかし，Java仮想マシン（JVM）上で末尾再帰の最適化と一級継続を実現することは困難である．これはJVMの仕様がそれらを実現するための機構を十分に提供していないことが要因である．実際に，JVMのバイトコードを出力する様々なコンパイラが実装されているにもかかわらず，JVMの制限のために，言語が要求する完全な機能を実現できていないものが少なからず存在する．そこで，JVMで末尾再帰の最適化と一級継続を実現するために，いくつかのバイトコード命令とその実行を補うためのクラスを拡張することを検討する．様々な拡張方法や実現方法が考えられるが，JVMの基本設計は可能な限り尊重し，最低限の拡張でかつ効果的にこれらの機能を実現可能にすることを目標とする．

JVM Extentions to Realize Tail Recursion Optimization and First-class Continuations

Akishige Yamamoto[†] and Taiichi Yuasa[††]

There are several programming languages that require tail recursion optimization and first-class continuations. Scheme, Standard ML, and several other mostly functional languages require these features. These languages rely heavily on the efficiency of tail recursion, and the ability of controlling continuations is one of the important features. However, it is difficult to implement tail recursion optimization and first-class continuations on the Java Virtual Machine (JVM), because the JVM specification does not provide features to realize them. Although various compilers are implemented that produce JVM byte code, some of them cannot realize full language features because of the restriction of the JVM specification. In this research, we propose byte code extensions and classes to support their execution to realize tail recursion optimization and first-class continuations on the JVM. Although there are various ways to extend it and implement them, we aim at having respect for the basic design of the JVM as far as possible and realizing them efficiently with minimum extensions.

1. はじめに

JVMは抽象機械の1つであり，文献1)に仕様が定義されている．この仕様に沿って様々なJVMの実装が開発されている．

JVMの仕様はそれ自体で独立しているが，この抽象機械の背景にはJava言語がある．Java言語は文献2)に仕様が定義されており，近年様々な用途に利用されつつある．Java言語は一般的にJVMのバイトコードにコンパイルされてから実行される．

JVMにはコードの移送機能やセキュリティ機構などが備わっており，またその普及度から，他言語からJVMのバイトコードを生成するコンパイラを作成する試みも多数行われている．JVMとJava言語の仕様の役割は互いに独立しており，またJVMはJava言語で書かれたプログラムを実行するのに十分なアーキテクチャを持ち合わせているが，他言語からのコンパイル結果の実行を考えた場合，JVMのマシンアーキテクチャでは不十分である場合も存在する．実現できたとしても非効率である場合や事実上実現不可能である場合もある．

たとえば，Scheme言語[3]は，末尾再帰の最適化を言語仕様として規定している．また，継続の捕捉/実行という機能も必要である．Scheme言語に限らず，一

† 株式会社数理システム
Mathematical Systems, Inc.
†† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

般的に関数型言語と呼ばれるプログラミング言語をコンパイルすることを考えた場合，抽象機械上にこれらを実現する枠組みを用意しておくことが不可欠である．しかしながら，JVM の仕様はこれらの機能を実現するのに十分な機能を持ち合わせていない．

Scheme から JVM バイトコードへのコンパイラの 1 つに Kawa があるが，文献 4) によれば，多値，末尾再帰の最適化，一級継続の実装が簡単ではなかったとしている．多値に関しては本稿とは直接関係はないが，多値をラッピングするようなオブジェクトを用意することにより実現している．この方法はヒープを消費するという難点がある．末尾再帰の最適化については，トランポリンと呼ばれる手法を用いて実現しているが，効率の良い方法であるとはいえない．一級継続については，JVM の例外処理機能を利用して実現しているが，完全な一級継続の実現はできていない．

そこで，本稿では，

- 末尾再帰の最適化
- 一級継続

を実現するための JVM の拡張を提案する．拡張方法には様々な方法が考えられるが，効果的にかつ最低限の改造で実現する方法を検討する．

2 章では末尾再帰の最適化の実現方法を考察し，その実装方法の解説およびその評価を行う．3 章では一級継続の実現方法を考察し，その実装方法の解説およびそのインタフェースなどを記述する．

2. 末尾再帰の最適化の実現

2.1 実 現 手 法

関数型言語のコンパイルには継続渡し方式（CPS）と呼ばれる中間コードを用いたコンパイルがしばしば行われている[5]．その方式を用いた出力コードを実行するためには，関数を呼び出す際に引数とともに次に実行してほしい「継続」を渡す．関数から復帰する場合もじつは呼び出す場合と同様であり，継続へジャンプすればよい．この方式の場合，関数呼び出しとは引数を持った goto でありフレームを消費しない．つまりすべての関数呼び出しが末尾再帰であるといえる．

このような機構を従来の JVM で実現するのは容易ではない．しかし JVM に対してラベル（プログラム上のコードアドレス）を一級オブジェクトとして扱えるような仕組みを導入すれば実現可能になる．具体的にはラベルの取得およびラベルへのジャンプ命令を用意するといった方法が考えられる．

もう少し高レベル（継続渡し方式の中間コードレベル）な JVM の拡張を考えた場合，継続を表現する継続オブジェクトを用意し，継続オブジェクトの生成および継続へのジャンプ命令を用意するといった方法も考えられる．この場合，仮にジャンプ可能な飛先を JVM のメソッドのみとした場合，継続オブジェクトを JVM のオブジェクトであるとすれば，継続オブジェクトの生成とはつまり JVM のオブジェクト生成であり，継続へのジャンプとはつまりインスタンスメソッドへのジャンプであると解釈できる．したがって，従来の JVM にはなかったインスタンスメソッドへのジャンプ命令を用意さえすれば，CPS を実行することが可能になる．

なお，CPS を実行することを考えた場合，継続オブジェクトの指し示すメソッドへジャンプをしてほしい場合と，定まったメソッドへジャンプをしてほしい場合☆がある．ここで，定まったメソッドとはつまりクラスメソッドであると解釈でき，クラスメソッドへのジャンプ命令も用意しておけば，より効率の良い実行が可能になると考えられる．

以上，CPS を JVM で実行するための拡張方法を検討したが，より一般的には，以下のように JVM に末尾再帰機能を拡張すれば良い．

まず，JVM にはメソッド呼び出しを行う以下の 4 命令が用意されている．

- invokestatic
- invokevirtual
- invokespecial
- invokeinterface

これらに対し，適切な末尾再帰呼び出しを行う以下の 4 命令を用意する．

- tailinvokestatic
- tailinvokevirtual
- tailinvokespecial
- tailinvokeinterface

また自己に対する末尾再帰呼び出しに対し，実行効率を上げるために以下の 4 命令を用意する．

- selftailinvokestatic
- selftailinvokevirtual
- selftailinvokespecial
- selftailinvokeinterface

2.2 メソッド呼び出し命令

JVM ではメソッド呼び出しを行う命令が 4 つ用意されている．

invokestatic 命令は，クラスメソッドの呼び出しを

☆ そのように CPS 変換されるのは，元のプログラム上で本当に末尾再帰である場合である．

行う．オペランドにはコンスタントプール上のインデックスが格納される．このインデックスで示されるコンスタントプール上の要素はツリー構造をなしており，それをたどることにより，クラス名，メソッド名，型情報を解決することができる．

invokestatic 命令を呼び出す前には，あらかじめメソッド呼び出しに必要な引数がオペランドスタックに積み上げられており，invokestatic 命令は，指定されたメソッドの型情報を読み取り，メソッドが必要とする引数をオペランドスタックから pop する．次に新規フレームを実行中のスレッド上のフレームスタックに積み上げ，先ほど pop した引数を新規フレームのローカル変数領域に push する．最後に指定されたメソッドの先頭に pc を移し処理を続行する．加えて，呼び出し先のメソッド内の復帰命令により，フレームスタックが元に戻され，その状態のオペランドスタックに返り値が push され，invokestatic 命令の直後から実行が再開される．

JVM にはほかにもメソッド呼び出し命令が用意されているが，それぞれ，使用条件や，メソッドアドレスの検索ルールが違う．

invokevirtual 命令は，インスタンスメソッドの呼び出しを行うもので，オペランドスタックに積まれたオブジェクトの型から動的なメソッド検索を行う．

invokeinterface 命令はインタフェースメソッドの呼び出しを行うもので，オペランドスタックに積まれたオブジェクトの型から動的なメソッド検索を行う．

invokespecial 命令は，

(1) インスタンス初期化メソッド (<init>)
(2) 実行中のクラスの private インスタンスメソッド
(3) スーパークラスのインスタンスメソッド

の呼び出しを行う特殊な命令である．なお，(1) と (2) は呼び出しメソッドを静的に決定できるが，(3) は invokevirtual と同様に動的なメソッド検索が必要である．

JVM にはメソッド呼び出しのために，以上の 4 命令が用意されているが，末尾再帰の最適化を考えた場合，メソッド検索ルールの違いは重要ではなく，フレームの状態遷移に着目すれば十分である．

2.3 フレーム構造

メソッド呼び出し命令のフレーム操作は前述のように定められているが，JVM の仕様上ではフレームの具体的な構造までは規定していない．しかし，以下のようなフレーム構造が最も一般的である．議論を進めるうえで，本稿ではこのようなフレーム構造を前提とする．

$$
\begin{array}{cl}
 & operand_{curstack} \\
 & \cdots \\
 & operand_1 \\
Frame & (prev, return, \ldots) \\
 & local_{maxlocals} \\
 & \cdots \\
 & local_1
\end{array}
$$

$operand_1$〜$operand_{curstack}$ はオペランドスタックであり，コードの実行過程で利用する領域で，上下に伸長する．$prev$ は親フレームへのポインタで復帰作業やスタックトレースを行う際に利用し，実際に復帰するアドレスは $return$ に格納される．ここでは $prev$ と $return$ しか書いていないが，この領域にはその他様々なフレーム固有の情報が格納される．$local_1$〜$local_{maxlocals}$ は局所変数領域でありメソッド固有の変数を割り当てる場所である．これらをひとまとまりにしてフレームと呼び，この構造をメモリの連続領域におき，再帰呼び出しされた場合はこの図では上の方向にフレームを積んでいくようなスタックを形成することができる．この図全体をフレームと呼ぶが，フレームを一意に示すには上図で $Frame$ とマークされたアドレスを利用する．このアドレスが一意に定まればフレーム上に格納された任意の要素をオフセットを利用して簡単にアクセスできる．C 言語などで実装する場合は上図の括弧で囲まれた部分を構造体として定義しておけばよい．

なお，JDK1.2.2[6] の JVM 実装に限っていえば $Frame$ から復帰する際のアドレスは $return$ ではなく，$prev$ の指すフレーム上（つまり親フレーム上）の $return$ の位置に格納されている．

ここで，連続的なスタックフレーム領域にフレームが n 個積まれている状態でメソッド呼び出しを行うことを考える．このときトップフレーム上のオペランドスタックには呼び出すべきメソッドが必要とする arg_1〜arg_m が引数として積まれているので，**図 1** のようになっているはずである．

この状態でメソッド呼び出しが発生した場合，仕様上は $Frame_n$ 上に積まれている引数 arg_1〜arg_m を pop し，新規フレームの局所変数領域に push し直す必要があるが，上記のようなフレーム構造になっている場合，$Frame_n$ 上の引数 arg_1〜arg_m をそのまま新規フレーム上の局所変数と見なすことができる．その結果は**図 2** のようになる．

2.4 末尾再帰命令

メソッド f の内部に，メソッド g を呼び出す命令

	arg_m
	...
	arg_1
	$operand_{n,curstack(n)-m}$
	...
	$operand_{n,1}$
$Frame_n$	$(prev_n, return_n, \ldots)$
	$local_{n,maxlocals(n)}$
	...
	$local_{n,1}$
	$operand_{n-1,curstack(n-1)}$
	...
	$operand_{n-1,1}$
$Frame_{n-1}$	$(prev_{n-1}, return_{n-1}, \ldots)$
	$local_{n-1,maxlocals(n-1)}$
	...
	$local_{n-1,1}$
	...

図 1 メソッド呼び出し前のフレーム状態
Fig. 1 Frame state before method invocation.

$Frame_{n+1}$	$(prev_{n+1}, return_{n+1}, \ldots)$
	$local_{n+1,maxlocals(n+1)}$
	...
	$local_{n+1,m+1}$
	arg_m
	...
	arg_1
	$operand_{n,curstack(n)-m}$
	...
	$operand_{n,1}$
$Frame_n$	$(prev_n, return'_n, \ldots)$
	$local_{n,maxlocals(n)}$
	...
	$local_{n,1}$
	...

図 2 メソッド呼び出し後のフレーム状態
Fig. 2 Frame state after method invocation.

があり，g の実行結果が f 自身の結果になる場合，g の呼び出しを末尾呼び出しと呼ぶ．末尾再帰命令は末尾呼び出しであるという条件の下に，通常のメソッド呼び出し命令より最適なフレームの状態遷移を与えるものである．

通常のメソッド呼び出しはメソッド復帰後に実行を継続する必要があるため，現在実行中のメソッドに固有の環境を蓄えたフレーム構造を保存してから，指定されたメソッドに処理を移す必要があった．しかし，末尾呼び出しであることがあらかじめ分かっている場合，フレームを保存する必要はない．末尾再帰の場合，現在実行中のメソッドが返すべき値は，これから呼び出そうとしているメソッドの戻り値であるが，そのメソッドを呼び出す時点で現在トップにあるフレーム構造はすでに不必要になってしまっているからである．

$Frame'_n$	$(prev_n, return'_n, \ldots)$
	$local'_{n,maxlocals'(n)}$
	...
	$local'_{n,m+1}$
	arg_m
	...
	arg_1
	$operand_{n-1,curstack(n-1)}$
	...
	$operand_{n-1,1}$
$Frame_{n-1}$	$(prev_{n-1}, return_{n-1}, \ldots)$
	$local_{n-1,maxlocals(n-1)}$
	...
	$local_{n-1,1}$
	...

図 3 末尾再帰後のフレーム状態
Fig. 3 Frame state after tail recursion.

図 1 のフレーム状態のときに末尾再帰が発生した場合，まず，$Frame_{n-1}$ 上のオペランドスタックに arg_1〜arg_m をコピーする．このとき $Frame_n$ が破壊されてしまうが，前述のようにこの時点で $Frame_n$ は不要になっているので構わない．そして，その上に $Frame'_n$ を積み上げればよい．その結果は**図 3** のようになる．

こうすることにより，$Frame_n$ 分のスタック消費量を削減することができ，$Frame'_n$ から 1 回の復帰で $Frame_{n-1}$ に戻ることができるようになる．

2.5 自己末尾再帰命令

自己末尾再帰命令は，末尾再帰命令の条件をさらに強めたものであり，呼び出し場所が末尾であり，かつ，自己への再帰呼び出しである場合のみ利用できる命令である．フレームの状態遷移という面では末尾再帰命令と同様である．

しかし，前述のようなフレーム構造になっている場合，この条件を利用して実行性能の向上が期待できる．つまり，末尾再帰なので，旧トップフレームは破壊可能であり，新規のフレームをその領域に上書きするこ

とができるのは末尾再帰と同様であるが，引数を含めたローカル変数領域のサイズは旧フレームと同じであるため，$Frame'_n$ の位置は変わらない．したがってフレームアドレスの再計算が必要なくなり，また，旧フレーム構造内の情報を再利用できる可能性があり，フレームの構築にかかる手間を削減することができる．

呼び出し先が自己であるか否かは，呼び出し命令の種類によっては実行時まで判定できない場合がある．invokestatic 命令の場合，自己判定は呼び出し先のメソッドの「名前」が自己と同一であるかを調べれば十分である．しかし動的なメソッド検索が発生するような呼び出し命令の場合，実際に呼び出される先は実行時まで確定できず，たとえ名前が同一であっても自己への再帰呼び出しであるという保証はできない．実行前に自己末尾再帰であることを確定できるのは，以下の条件のいずれかを満たす末尾再帰である．

- invokestatic 命令．
- invokevirtual 命令で呼び出し先が private インスタンスメソッドである場合（通常は invokespecial 命令にコンパイルされる）．
- invokespecial 命令で呼び出し先がコンストラクタもしくは private インスタンスメソッドである場合．

これらの条件を満たす末尾再帰は確定的な自己末尾再帰である．このような自己末尾再帰の場合，無条件に前述の自己末尾再帰処理を行うことができる．

上の条件を満たさないが，呼び出し先の名前が自己と同一であるような末尾再帰は不確定的な自己末尾再帰である．このような末尾再帰は，実行時に呼び出し先の自己判定を行い，呼び出し先が自己であると判定された場合のみ，前述の自己末尾再帰処理を行うことができる．一方，呼び出し先が自己でないと判定された場合は，自己末尾再帰処理は行えないが，(非自己) 末尾再帰処理を行うことができる．

2.6 quick 命令

JVM の仕様では，クラスファイルを読み込んだ後に，コードの構造的な検証（verification）を行い，クラスフィールドの初期化などの準備（preparation）を行い，クラスファイル上のシンボリックな参照を解決（resolution）してから処理を進めるとされている．なお，解決作業をリンク時に行うか，実行時に行うかは実装依存とされている．

実際に JDK1.2.2 の JVM 実装では，メソッド呼び出し命令のオペランドの解決は実行時に行われるようになっている．メソッド呼び出し命令に対し，対応する quick 命令が用意されており，初めてメソッド呼び出し命令が実行されると，そのバイトコード自体を対応する quick 命令に置き換える．

quick 命令には，

- invokestatic_quick
- invokevirtual_quick
- invokenonvirtual_quick
- invokevirtualobject_quick
- invokesuper_quick
- invokeinterface_quick

が用意されており，それぞれ別々のオペコードが割り当てられている．それぞれ，2 回目以降の実行を高速に実行できるように，オペランドには呼び出されるべきメソッドの情報が格納されている．また，実行時のメソッド検索を最低限に抑えることができるようにこの時点で場合分けを行っている．補足すると，これらの quick 命令はあくまでも実装内部での使用に限られており，バイトコードに埋め込むことは許されていない．

末尾再帰命令に関しても，通常のメソッド呼び出し命令と同様に quick 命令を用意することができる．通常のメソッド呼び出し命令に対する quick 化は，末尾再帰命令に関してもまったく同様に行える．

2.7 利用方法

末尾再帰の最適化のために拡張したバイトコードの利用方法には，

(1) コンパイラが前述の末尾再帰のための拡張命令を出力する方法，
(2) クラスファイルロード時にバイトコードから末尾再帰を検出し，自動変換する方法，

が考えられる．

(1) の方法は，ロード時の自動変換の手間を省くことができる．また，コンパイラがコード生成時に適切に末尾再帰すべきことを明示的に指示できる．これは，コンパイル時に末尾再帰の振舞いを保証できるという利点があり，Scheme 言語のように言語仕様で末尾再帰の最適化を規定している言語をコンパイルする場合には便利な機能である．

一方，(1) の方法は，実際は末尾再帰でないのに，末尾再帰命令もしくは自己末尾再帰命令が利用されていないことを，検証フェーズで確認する必要が出てくる．また，拡張命令が含まれるバイトコードでは，通常の JVM 実装では実行不可能になってしまうという重大な欠点がある．

これらをふまえたうえで，(2) の方法は，自動変換の手間が生じるが，そのコストはそれほど高くならないと考えられ（仮に (1) の方法を採用したとしても検

```
static public int sum(int n, int r) {
  if (n > 0) return sum(n - 1, r + n);
  else return r;
}
static public int sumTail(int n, int r) {
  if (n > 0) return sumTail2(n - 1, r + n);
  else return r;
}
static public int sumTail2(int n, int r) {
  if (n > 0) return sumTail(n - 1, r + n);
  else return r;
}
static public int sumLoop(int n, int r) {
  while (n > 0) {
      r += n--;
  }
  return r;
}
static public int sumExc(int n, int r) {
  try {
    if (n > 0) return sumExc(n - 1, r + n);
    else if (n == 0) return r;
    else throw new Exception("negative");
  } catch (Exception e) {
    return -1;
  }
}
```

図 4 ベンチマークコード
Fig. 4 Benchmark code.

証フェーズで同様の検査を行う必要が出てくる），従来の JVM 実装でも実行が可能であることは重要だと考えられるので，(2) の方法を採用すべきだと判断した．

2.8 末尾再帰命令への自動変換

末尾再帰命令への自動変換は，自己末尾再帰を対応する自己末尾再帰命令に，そうでない末尾再帰を対応する末尾再帰命令に置き換えるものである．

単純に再帰命令の直後に復帰命令がある場合，末尾再帰と判定することはできない．**図 4** の sumExc のような場合がある．このコード上の return sumExc (n − 1, r + n) は，

```
invokestatic sumExc
ireturn
```

と JDK1.2.2 ではコンパイルされるが，invokestatic 命令の継続は ireturn とは限らない．なぜなら invokestatic 命令が例外を受け取った場合，次に実行される命令は，例外ハンドラの先頭コードである．つまり，invokestatic 命令の返り値が，sum メソッド自身の返り値とならない可能性があるので，これは末尾再帰ではない．

また，次のような例もある．

```
    invokestatic foo
    goto label
label:
    ireturn
```

このコードで foo が int を返すメソッドである場合，invokestatic 命令の直後は復帰命令ではないが，invokestatic 命令の返り値が，この関数自身の返り値になるため，これは末尾再帰である．

ほかには，以下のような例で，

```
    iconst_1
    invokestatic foo
    ireturn
```

foo が void メソッドである場合は末尾再帰にならない．なぜなら ireturn が返すのは invokestatic 命令の返り値ではなく，その前に積まれた値 1 であるからである．

以上をまとめると，メソッド呼び出し命令が，

(1) メソッド呼び出し命令の直後に任意個の pc 以外を変化させない命令を狭んで復帰命令がある，
(2) 呼び出すメソッドの戻り値の型と復帰命令の型が一致する，
(3) メソッド呼び出し命令から復帰命令の間に例外ハンドラが設定されていない，

の条件を満たす場合，そのメソッド呼び出しは末尾再帰であり，末尾再帰命令もしくは自己末尾再帰命令に置き換えることが可能である．なお，JVM の命令セットの中で pc 以外を変化させない命令は nop と goto と goto_w の 3 つである．また，上でいう型とはオブジェクトの型ではなく，JVM がバイトコード上で区別している型（int，long，float，double，Object，void）である．

自動変換処理はローダの準備フェーズへ組み込む方法を選択した．なぜなら検証フェーズには手を入れずに済むからである．また，この変換処理は一度バイトコードを上から下まで走査するだけで実現可能であり，JVM の検証フェーズにかかる時間に比べれば誤差の範囲になる．

なお，前述の quick 化のように呼び出し先メソッドの解決を実行時に行っている場合，自動変換処理も実行時に行う方法も考えられる．その場合，末尾再帰命令や自己末尾再帰命令を用意する必要はなく，初めて

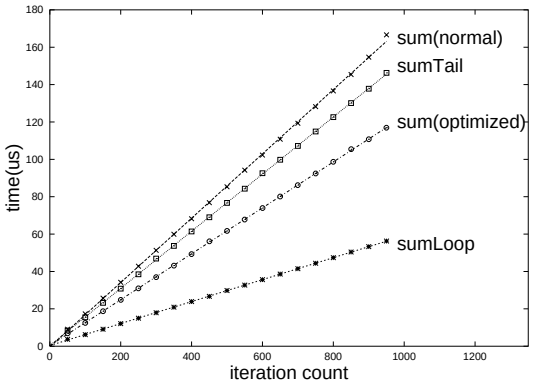

図 5 再帰回数と処理時間の関係
Fig. 5 Relation between iteration count and execution time.

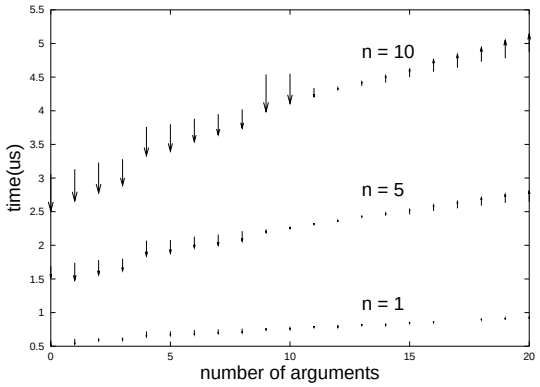

図 6 自己末尾再帰の最適化効果（n = 1, 5, 10）
Fig. 6 Effect of self tail recursion optimization (n=1, 5, 10).

メソッド呼び出しが発生したタイミングで末尾再帰検査を行い，末尾再帰である場合はその命令自身を末尾再帰を行う quick 命令に変換し，そうでない場合は従来の quick 化を行えばよい．

2.9 性能測定

性能測定は以下の環境で行った．

CPU	AMD Athlon 800 MHz
メモリ	256 Mbytes
OS	Debian-2.2 (Linux-2.2.16)
JVM	JDK1.2.2（JIT なし）

2.9.1 再帰回数と処理時間

図 4 のコードで，再帰呼び出しの深さ n を変化させて処理時間を計測すると，**図 5** の結果が得られた．

図 5 は上から通常呼び出し,（非自己）末尾再帰，自己末尾再帰，ループの処理時間を示している．この計測は JVM の種別（従来版，改造版）とベンチマークコードの組合せで行った．組合せは下記のとおりである．sum のコードを自動変換にかけると自己末尾再帰になり，sumTail のコードを自動変換にかけると（非自己）末尾再帰になることを前提としている．結果は 10,000 回の試行における 1 回あたりの平均時間（μs）である．

	JVM	コード
通常呼び出し	従来版	sum
（非自己）末尾再帰	改造版	sumTail
自己末尾再帰	改造版	sum
ループ	従来版	sumLoop

2.9.2 引数の個数と処理時間

引数の個数 $0 \leq m \leq 20$ を変化させて処理時間を計測した．

図 6，**図 7** は引数の個数 m と自己末尾再帰の処理時間の関係を示し，**図 8**，**図 9** は引数の個数 m と（非自己）末尾再帰の処理時間の関係を示す．結果は 100,000 回の試行における 1 回あたりの平均時間（μs）である．

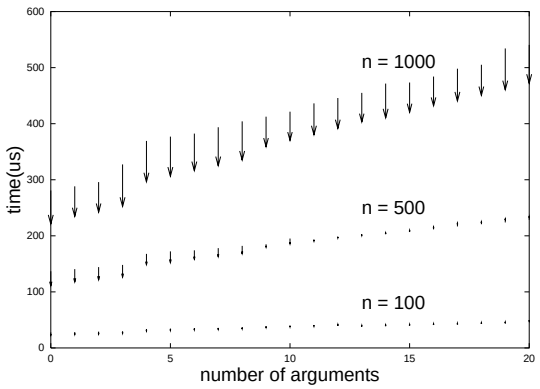

図 7 自己末尾再帰の最適化効果（n = 100, 500, 1000）
Fig. 7 Effect of self tail recursion optimization (n=100, 500, 1000).

この図上の，矢印の元が従来の JVM での処理時間であり，矢印の先が末尾再帰の最適化を行う JVM での処理時間である．つまり，下向きの矢印は性能向上を表し，上向きの矢印は性能低下を表している．

2.10 性能分析

深さ n の m 引数の再帰呼び出しを実行した場合，

$$
\begin{aligned}
&T_{\mathrm{nomral}}(n,m)\\
&= n(C + mC_{\mathrm{push}} + C_{\mathrm{call}} + C_{\mathrm{ret}})\\
&= mnC_{\mathrm{push}} + n(C + C_{\mathrm{call}} + C_{\mathrm{ret}})
\end{aligned}
$$

程度の処理時間がかかると概算できる．ここで，C はメソッド内部で固定的にかかる時間，C_{push} は引数の積上げにかかる時間，C_{call} は呼び出し処理にかかる時間，C_{ret} は復帰作業にかかる時間である．

一方，本手法で，深さ n の m 引数の再帰呼び出し

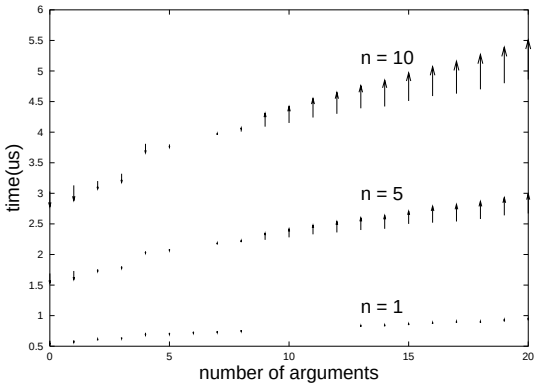

図 8 （非自己）末尾再帰の最適化効果（n = 1, 5, 10）
Fig. 8 Effect of (non-self) tail recursion optimization (n=1, 5, 10).

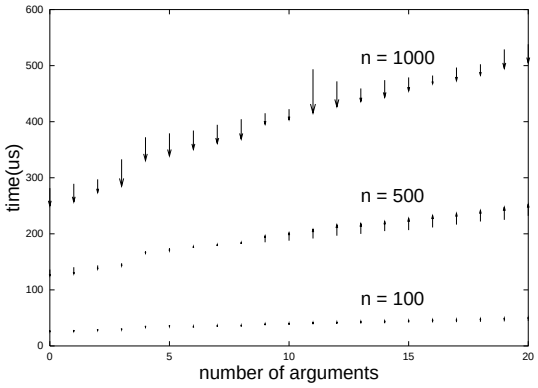

図 9 （非自己）末尾再帰の最適化効果（n = 100, 500, 1000）
Fig. 9 Effect of (non-self) tail recursion optimization (n=100, 500, 1000).

を実行した場合，

$$
\begin{aligned}
&T_{\mathrm{tail}}(n,m)\\
&= n(C + mC_{\mathrm{push}} + T_{\mathrm{call}}(m)) + C_{\mathrm{ret}}\\
&= n(C + mC_{\mathrm{push}} + C'_{\mathrm{call}} + mC_{\mathrm{copy}}) + C_{\mathrm{ret}}\\
&= mn(C_{push} + C_{\mathrm{copy}}) + n(C + C'_{\mathrm{call}}) + C_{\mathrm{ret}}
\end{aligned}
$$

程度の処理時間がかかると概算できる．ここで，$T_{\mathrm{call}}(m)$ は m 引数の末尾呼び出し処理にかかる時間である．T_{call} は m の関数であり，固定時間 C'_{call} と m 回のメモリ転送 mC_{copy} に分割することができる．

つまり，処理時間の差は，

$$
\begin{aligned}
&T_{\mathrm{normal}}(n,m) - T_{\mathrm{tail}}(n,m)\\
&= n(C_{\mathrm{ret}} + C_{\mathrm{call}} - C'_{\mathrm{call}} - mC_{\mathrm{copy}}) + C_{\mathrm{ret}}
\end{aligned}
$$

であるので，本手法を利用した場合，

(1) 処理時間に関する最適化効果は，再帰の深さ（n）にほぼ比例する．つまり，再帰が深いほど，最適化効果（高速化もしくは低速化）が顕著化する，

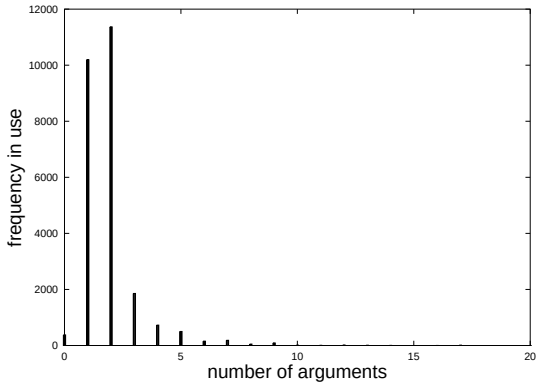

図 10 引数の個数の頻度分布
Fig. 10 Frequency distribution for number of arguments.

(2) フレームの再利用による高速化（$C_{\mathrm{call}} - C'_{\mathrm{call}}$）および復帰回数が削減した分の高速化が，メモリ転送によるオーバヘッド（mC_{copy}）を上回っている場合，実行効率は良くなる．逆転した場合は，実行効率は悪くなる，

と考えられる．

実際に計測してみると，他の要因も加わるため，正確にこのようにはならないが，図 5 の結果は (1) を示しており，図 6，図 7，図 8，図 9 の結果は (2) を示している．

しかし，再帰呼び出しの深さ n が 1000 程度になると，従来版による呼び出しの性能が著しく低下している（引数の個数にかかわらず末尾再帰の最適化を行った方が速くなっている）．これはスタックが肥大化するので，ページングが発生しているためと思われる．スタックの消費量を削減することは実行効率にも影響していることが分かる．

自己末尾再帰呼び出しの場合は，フレームの再利用による高速化は十分大きく，今回の計測では，最適化効果の臨界点になる引数の個数 (m) は 11〜12 程度になっている．

（非自己）末尾再帰の場合は，$C_{\mathrm{call}} - C'_{\mathrm{call}}$ がそれほど大きくならないため，最適化効果の臨界点になる引数の個数 (m) は 6〜7 程度にとどまっている．

なお，Java API 上の java，javax パッケージの 25579 メソッドの引数の個数の分布は**図 10** ようになっている（double/long は 2 ワード，インスタンスメソッドである場合は引数のサイズ +1 ワードと計算している）．

この結果によれば 1 メソッドあたりの平均の引数の個数は 1.89 個となる．引数の個数が 6 以上であるのは全体の 2.17%であり，11 以上は全体の 0.23%である．

表 1 Gabriel ベンチマーク
Table 1 Gabriel benchmark.

JVM	Kawa	tak	takl	takr	cpstak	deriv	fft	div-iter	div-rec
off	on	985.9	1745.0	533.6	695.7	967.5	3568.1	542.7	1119.9
on	on	480.9	1679.7	508.2	672.6	917.7	3438.1	529.1	1100.9
off	off	166.7	467.0	170.8	error	574.2	3602.4	529.5	534.7
on	off	159.4	424.2	163.4	350.9	551.0	3472.3	524.5	528.2
scm		58.0	456.2	60.5	161.4	193.4	503.2	131.0	133.1
guile		142.4	1117.1	155.3	221.8	296.1	808.8	244.8	255.2
SUN JIT	on	205.5	156.1	228.5	229.5	107.1	1152.9	36.2	99.9
SUN JIT	off	14.9	53.3	17.9	error	66.4	1318.8	36.6	40.5
IBM JIT	on	74.5	264.2	126.4	154.7	240.9	501.9	56.0	177.0
IBM JIT	off	9.7	29.5	14.6	error	123.8	474.4	54.2	63.4

Scheme 関数をコンパイルしたときの引数の個数を，後述の Gabriel ベンチマークを Kawa でコンパイルして調査した．結果は 0 引数のものが全体の 7.29%で，1 引数のものが 76.56%であり，平均は 1.09 個であった．必ずしも一般性があるとはいえないが，Gabriel ベンチマークを見る限り，引数の個数は，Java クラスライブラリよりも少なくなる傾向があるようだ．

2.11 Scheme 処理系としての性能

Scheme 言語から JVM のバイトコードを出力するコンパイラの 1 つに Kawa[4] がある．Kawa の出力に対しベンチマークを行った．

Kawa は独自の末尾再帰の最適化機能を持っている．今まで述べてきたように，JVM のアーキテクチャでは効率の良い末尾再帰処理を行うことができない．しかし，JVM のアーキテクチャの範囲でもトランポリンと呼ばれる手法を用いれば末尾再帰を適切に処理することが可能であり，Kawa でもその手法を採用している．同様の手法はほかには，Standard ML から C へのトランスレータである SML2C[7] でも用いられている．

この手法を用いれば，スタックをまったく消費せずに末尾再帰が行えるので，たとえ無限ループであっても末尾再帰を用いて表現することができる．しかし，すべての関数呼び出しにオーバヘッドがかかるので，十分な性能を出すことは期待できない．

Scheme のベンチマークとしてよく利用される Gabriel ベンチマーク[8] の中のいくつかについてベンチマークを行い，**表 1** の結果を得た．

上の 4 行は，JVM の末尾最適化機能の on/off と Kawa の末尾最適化機能の on/off の組合せで行ったものである．次の 2 行は代表的な Scheme インタプリタ（scm–5d2[9]，guile–1.3[10]）での結果である．次の 2 行は，Kawa（kawa–1.6.70）によるコンパイル結果を blackdown–1.3.0–FCS[11] による JIT コンパイラを用いて実行した結果である．また，最後の 2 行は，同様に IBM による JIT コンパイラ[12]（IBM Developer Kit for Linux，Version 1.3）を用いて実行した結果である．

なお，blackdown の JIT は Sun Microsystems の HotSpot 技術を用いたものであるが，末尾再帰の最適化は行っていない．一方 IBM の JIT は Method inlining の特殊な場合として，static な自己末尾再帰のループ展開を行っている．今回の末尾再帰の最適化手法は JIT 機能を実装していないので単純な比較はできないが，参考までに計測している．

tak は再帰を大量に発生するベンチマークである．takl は tak 上の比較命令を cons セルの長さ比較に置き換えており，その比較自体は自己末尾再帰になっている．takr は tak を 100 の関数に分割することでキャッシュによる影響を減少させるものであり，結果的に tak 上の自己末尾再帰が（非自己）末尾再帰になっている．Kawa の末尾再帰の最適化機能を利用しない場合，tak は 5.2%，takr は 3.7%，takl は 9.3%の末尾再帰の最適化効果が現れている．

cpstak は tak を CPS 変換したもので，すべての呼び出しが末尾再帰になっている．tak では末尾再帰でなかった呼び出しも，継続を引数とともに渡すことですべての呼び出しが末尾再帰になっているので tak よりは遅くなる．このとき，末尾再帰の最適化を行わない場合は再帰呼び出しがかなり深くなるため，JVM もしくは Kawa のいずれの最適機能も用いない場合は，スタックオーバフローが発生してしまう．

deriv は多項式の導関数を求めるもので，cons セルの生成の速度に依存する．fft は高速フーリエ変換を行うもので，浮動小数点演算，配列アクセスの速度に依存する．今回の最適化とは関係がないものと思っていたが，内部でループが発生するので，結果的にはそれぞれ 4.0%，3.6%の最適化効果が現れている．

div–iter，div–rec はリストの要素を 1 つ飛びに集めるものである．div–iter は do 文を利用して繰返し

を行い，div–rec は再帰（末尾再帰ではない）によって繰返しを行っている．Kawa では do 文はループ展開されるので div–iter に最適化効果は現れていない．また通常の再帰呼び出しを利用している div–rec も最適化効果は現れていない．

全体的に Kawa の末尾再帰の最適化機能に対して JVM の末尾再帰の最適化機能は著しい性能向上が見られる．そもそも，Kawa の末尾再帰の最適化機能は Scheme の仕様を満たすために，速度面の不利益を黙認して実現しているものである．Kawa の末尾再帰の最適化が速度低下を引き起こしていることは，表 1 の 1 行目と 3 行目を比較すればあきらかである．一方 JVM の末尾再帰の最適化機能は，速度向上の可能性もあることを前述したが，表 1 の 3 行目と 4 行目の結果はそれを実証している．

Scheme インタプリタである scm-5d2，guile-1.3 は，ほとんどの場合 Kawa の結果より速い結果が得られている．しかし JIT 機能を持つ JVM と Kawa を組み合わせると，さらに速い結果が得られる．Kawa による末尾再帰の最適化機能はかなりの速度低下を引き起こしているが，その機能を利用しない限り前述したとおり cpstak はスタックオーバフローを発生してしまう．

IBM の JIT は自己末尾再帰の最適化を行うが，Scheme の仕様に厳密に従うためには，自己でも非自己であっても末尾再帰にスタックを消費してはならない．また，JIT の実装によって末尾再帰を最適化の挙動が違っては，Scheme 処理系としては不十分である．

IBM の JIT における末尾再帰の最適化は static な自己末尾再帰に限定される．これらの条件を満たす末尾再帰については，ループに展開した方がフレーム操作がまったく必要なくなる分有利である．一方，本手法はこれらの条件を満たしているようなメソッド呼び出しの場合でもフレーム操作が必要になるが，その処理が最低限になるよう可能な限り最適化を行っている．また，本手法は virtual なメソッド呼び出しの場合や自己以外への末尾再帰も最適化可能である．

3. 一級継続の実現

3.1 実現手法

一級継続とは動的環境を含む実行状態を表現するものである．インナークラスを用いた擬似的なクロージャ渡しでは，静的環境を取り込んだ関数閉包は実現できるが，外側の関数がリターンしたあとの継続を受け渡すことはできない．一級継続を実現するにはスタックを直接アクセスする必要がある．

一級継続の実現手法には様々なものが提案されており，文献 13) が詳しい．

JVM に継続を導入することを考えた場合，incremental stack/heap 戦略が最適である．JVM の実装ではフレームをスタック上に割り当てている場合が多い．また，継続の捕捉/実行がいっさい発生しないコードの実行速度低下を引き起こすことは望ましくない．速度低下を引き起こさずに，継続の高速な捕捉/実行が行える，incremental stack/heap 戦略が良いと思われる．

どのような手法を用いるにせよ，JVM のアーキテクチャにはスタックを操作する仕組みが用意されていないので，一級継続を実現するために，以下の 2 命令を用意することにする．

- capturecont
- throwcont

capturecont 命令は継続の捕捉を行うもので，throwcont は継続の実行を行うものである．また，これらの命令が扱う継続を表現するオブジェクトを用意する必要がある．

3.1.1 継続の捕捉命令

capturecont 命令は 3 バイト命令であり，オペランドには 2 バイトのオフセットが格納されている．

capturecont 命令を実行すると，継続を捕捉する．捕捉した継続は継続オブジェクトという形でヒープ上に退避しておき，その参照をオペランドスタックに積み上げる．継続オブジェクトには，オペランドに指定されたオフセットも格納しておく．

3.1.2 継続の実行命令

throwcont 命令は 1 バイト命令である．

throwcont 命令を実行すると，オペランドスタックから継続オブジェクトへの参照と，結果オブジェクトへの参照を取得する．次に継続オブジェクトに退避されていたスタックフレームを復元し，結果オブジェクトへの参照を復元されたスタックフレーム上のスタックトップに積み上げる．最後に継続オブジェクトに保存されているオフセット分 pc を進める．

3.2 incremental stack/heap 戦略の実現

incremental stack/heap 戦略は，フレームをスタック表現とリスト表現に分けて実現する方法である．通常の実行はスタックを用いて行い，継続が捕捉されるとリスト表現に変換され，スタックは空になる．そのとき，スタックボトムからリストへのポインタを憶えておき，スタックアンダフローが発生したときは，リストからフレームを 1 つスタックに書き戻し，実行を継続する．継続の実行もスタックアンダフローが発生したときと同様の処理を行えばよい．

	$result_2$
	$result_1$
	$parent$
$Frame_1$	$(prev_1, return_1, \ldots)$
	$local_m$
	$\ldots$
	$local_1$

図 11 監視フレーム
Fig. 11 Sentinel frame.

このような構造にするのは，継続の捕捉は密集して発生する場合が多いからである．一度捕捉された継続は再捕捉されたり，近い位置（前後を含む）で継続が捕捉されたりする可能性が高い．その場合，リスト表現にすればフレームの共有が行え，継続の捕捉はコピーする量が減るので高速になり，継続の捕捉/実行が繰り返して発生する場合は一般的に高速化が期待できる．一方，継続の捕捉がいっさい発生しない従来の JVM コードに対しては速度低下を引き起すことはない．

スタックアンダフローの検査は，スタックボトムに監視フレームを用意すれば，復帰ごとの検査を省くことができる．JVM でそれを行うためには，特殊なトラップ命令を用意しておき，監視フレームに復帰した際には必ずその命令を実行する仕組みを作り上げておけば，それがトラップの働きをする．

3.3 監視フレームの構造

監視フレームは**図 11** のような構造をしており，必ずスタックの最下部に配置される．

$local_1$〜$local_m$ は局所変数領域だが通常は何も格納されない．監視フレーム上の $prev_1$，$return_1$ はここでは意味をなさないが，他のフレーム構造とあわせるために用意している．$parent$ はヒープ上に退避された親フレームへの参照であり，親フレームがない場合は null が設定される．$result_1$ と $result_2$ は戻り値を格納する領域である．JVM のオブジェクトサイズは最大 2 ワード（long もしくは double の場合）であるので，2 ワード分の領域を空けておく．通常のフレームのオペランドスタックは可変長であるが，監視フレームの場合は固定サイズである．

3.4 継続の捕捉命令の動作

capturecont 命令はオペランドにオフセット δ を持ち，

- 現在の pc をオフセット δ 分移動した位置 pc' を求める，
- pc' にジャンプしたときのスタックフレームを仮定し，そのスタックフレームをヒープ上のリスト表現に変換し，そのリストへの参照 r を憶えておく，
- 監視フレームだけを残しスタックフレームをいったん消去し，r のトップフレームを書き戻す．監視フレーム上の $parent$ は r の次のフレームへの参照に書き換えておく，
- 先ほど憶えておいたリストへの参照 r をオペランドスタックにプッシュする，
- pc を進める，

という動作を行えばよい．

3.5 継続の実行命令の動作

throwcont 命令は，

- オペランドスタックから継続オブジェクトへの参照 r と，結果オブジェクトへの参照 r' を取得する，
- 監視フレームだけを残しスタックフレームをいったん消去し，r のトップフレームを書き戻す．監視フレーム上の $parent$ には r の次のフレームへの参照に書き換えておく，
- r' をオペランドスタックにプッシュする，

という動作を行えばよい．

3.6 トラップ命令の動作

復帰ごとの検査を省くためにトラップ命令を用意することを前述したが，その命令（framesentinel 命令）は，

- オペランドスタックから継続オブジェクトへの参照 r と，戻り値 $result_i$ を取得する，
- スタックフレームに r のトップフレームを書き戻す，
- 監視フレーム上の $parent$ には r の次のフレームへの参照に書き換えておく，
- $result_i$ をオペランドスタックにをプッシュする，

という動作を行えばよい．

3.7 JVM 起動時の動作

JVM の実行の開始は指定されたクラスの public かつ static かつ void であると宣言された main メソッドから行われる．そして，main メソッドからの復帰後（復帰命令による場合と，例外が発生した場合がある）にスタックフレームが空になった時点で実行を終了する．

監視フレームを導入するには JVM の開始処理部分を変更する必要がある．開始処理は，まず，あらかじめ，以下のようなバイトコードを用意しておく．

```
invokestatic foo
framesentinel
```

そして，最初に起動すべき static メソッドが実行さ

れるように，このコード上の foo の部分やその他の環境をうまく調整をしておき，スタックの最下部には監視フレームを用意し，そのフレーム上の *parent* には null を設定しておく．そのうえで，このバイトコードの先頭から実行を開始すればよい．

こうすれば，監視フレームに達した時点でオペランドスタックに継続オブジェクトのリストへの参照が設定されている場合，framesentinel 命令が前述のような処理を行い，実行を継続する．また，継続オブジェクトのリストへの参照が null である場合は，本当のスタックボトムに達したことを意味するので，実行を終了する．JNI によるネイティブメソッドからの Java バイトコード呼び出しも，同様に監視フレームを用意してから実行すればよい．返り値のあるメソッド呼び出しの場合は，その値が監視フレームの $result_i$ に蓄えられている．

3.8 継続オブジェクト

継続の捕捉時にフレームイメージを退避する継続オブジェクトはリスト構造をなしており，トップフレームからボトムフレームへの 1 方向リストにしておけばよい．

各フレームオブジェクトは，スタック領域で用いられているフレームイメージをそのまま蓄えておけばよい．しかし，フレーム上のオペランドスタックと局所変数領域上にあるオブジェクトの参照値は GC から保護されるようにしておく必要がある．また，捕捉された継続オブジェクトへの参照も GC の走査対象にする必要がある．

継続オブジェクトへの参照は前述のように，監視フレームのオペランドスタックに蓄えられるので走査対象になっている．ただし，継続の捕捉作業，つまりスタック表現からリスト表現への変換中に GC が発生する可能性があるので，注意が必要である．たとえば，リストを cons するたびに監視フレーム上の *parent* を更新するようにすれば，これを防ぐことができる．

次にオペランドスタックと局所変数領域の退避方法であるが，単純に byte の配列として退避した場合はフレーム上にあるオブジェクトへの参照値が GC から保護されないので問題である．しかしオブジェクトの配列として退避した場合，退避するイメージの中には違法なオブジェクトが入っている可能性がある．JDK1.2.2 の JVM 実装では int，long，float，double などのプリミティブデータもそのままオペランドスタックに積まれ，それらの型情報は実行時には分からない．そのため，GC のスタック走査は，保守的（conservative）に行うことでその問題を回避している．一方，オブジェクトの配列の走査は要素がすべて合法なオブジェクトへの参照であると仮定しているため，保守的な走査を行っていない．そのため，フレームのイメージをオブジェクトの配列として退避した場合，JVM システムが GC 時に落ちてしまう．

これを回避する方法はいくつか考えられる．

(1) フレームのイメージをオブジェクトの配列として退避しておき，オブジェクトの配列の走査もスタックの操作と同様に保守的に行うようにする．
(2) フレームのイメージを byte の配列として退避しておく．さらに継続の捕捉時に，フレーム上のオペランドスタックおよび局所変数領域の内容を調べ，オブジェクトである可能性があるものだけを，オブジェクトの配列として別に退避しておく．
(3) 見かけはオブジェクトの配列であるが，GC 時は保守的に走査を行うような新しい型を用意し，フレームのイメージをその型として退避しておく．

(1) の方法はすべてのオブジェクトの配列の走査が低速化してしまう．(2) の方法は継続の捕捉時に余計な手間が増え，継続オブジェクトのサイズも大きくなってしまう．(3) の方法は効率面ではまったく問題ない．

しかし，今回の継続オブジェクトの実装には，JDK1.2.2 の JVM 実装でネイティブオブジェクトを作る際に利用されている仕組みを利用した．javah ツールに-old オプションを指定することで JDK1.0 形式のヘッダファイルを生成する方法で，汎用性はないが JNI1.1 形式よりは高速にフィールドアクセスなどが可能である．java.lang.String などの基本クラスも，この方法で作成されている．このような方法を用いた場合，(3) の方法は簡単ではない．(3) の方法を実現するには JNI を利用しないでオブジェクトを作り込む必要がある．また，継続をいっさい利用しないコードの速度低下が発生することも望ましくないので，現在は (2) の方法を採用している．

3.9 Java 言語での使用例

図 12 のような，Scheme による call/cc を利用した大域脱出の例を考える．この Scheme 関数と同等の機能を持った Java 言語のメソッドは**図 13** のように定義可能である．

Java 言語には例外処理機能が用意されており，大域脱出は try/catch で実現できるので，一級継続のサンプルとしてはふさわしくないかもしれないが，簡単なためこの例を選んだ．Continuation.callCc は引数に ContinautionReceiver オブジェクトを受け取る．

```
(define (first-negative lst)
  (call-with-current-continuation
   (lambda (exit)
     (for-each (lambda (x)
                 (if (negative? x)
                     (exit x)))
               lst)
     #t)))
```

図 12 大域脱出（Scheme）
Fig. 12 Non-local exit (Scheme).

```
static public void firstNegative(
                   final int flist[]) {
 Object result =
  Continuation.callCc(
    new ContinuationReceiver() {
      public Object receiveContinuation(
                    Continuation cont) {
        for (int i = 0;
             i < flist.length;
             i++) {
          if (flist[i] < 0)
            throw cont.throwContinuation(
                  new Integer(flist[i]));
        }
        throw cont.throwContinuation(null);
      }
    });
 System.out.println(result);
}
```

図 13 大域脱出（Java）
Fig. 13 Non-local exit (Java).

callCc メソッドは継続を捕捉し，そのオブジェクトを，引数として受け取った ContinuationReceiver オブジェクトの receiveContinuation メソッドに渡す．つまり，ContinuationReceiver オブジェクトとは，Scheme 言語の call/cc に指定するラムダ式に相当するものである．ラムダ式の内部で処理を記述する場合，外部参照するケースが非常に多い．Scheme の場合はレキシカルなスコープで外部参照ができる．一方 Java 言語にはクロージャという概念はないがそれに近いものに，インナークラスがある．インナークラス内のメソッド定義からは外部参照が可能である．そのため，少々複雑だがこのようなインタフェースを採用した．なお，Java 言語のインナークラスのメソッド定義内からの外部参照は多少制限がある．そのため，上記の例では final int flist[] と宣言している．この final がないとコンパイルすることができない．

3.10 問題点および解決策

静的にバイトコードの安全性を確認するためには，ベリファイアで継続の捕捉/実行命令に対する適切なフロー解析を行う必要があるが，現在は行っていない．

フロー解析処理は以下のような処理を加えればよい．

- フロー解析中に「capturecont label」コードがあった場合，そのスタック状態に初期化済みオブジェクトをプッシュした状態で「goto label」を実行しても安全か否かを検査する．
- フロー解析中に「throwcont」コードがあった場合は，オペランドスタックに適切な引数が積まれているか否かを検査する．

ほかには，オペコード検査，スタックの上限値の検査なども，他の命令コードと同様に行う必要がある．

上のフロー解析が通ったコードであれば，継続の実行先は捕捉時と同一のメソッド内であることが保証されており，実行先に処理を移してもデータフローとしては安全であることが保証されている．また，ジャンプ先が同一メソッド内であるので，捕捉前のスレッドのアクセス権を実行後に復元すれば，アクセス権の面でも継続の捕捉/実行処理は安全である．

また，以下のような問題がある．

(1) 現在の実装では try – catch – finally 節の try 部分で，継続の実行が発生した場合，finally 節を実行しない．

(2) synchronized されたオブジェクトがある状態で，継続の捕捉が発生し，synchronized ブロックを抜けた後に，継続の実行が発生し，synchronized ブロック内に処理が戻るような場合，同期ロックに不整合が生じる．

Java のこの種の機能は，Scheme の仕様と相性が悪い．Scheme の言語内でも，たとえばファイルを閉じる前に継続を実行したら，同じような問題が生じてしまうが，この問題を避けるのはプログラマの責任になっている．

どうしても，処理系で対応する必要があれば，次のような解決策がある，

(1) の問題は，継続の実行処理前にスタックをたどりながらすべての finally 節を実行し（通常の例外処理と同様に行えばよい），その処理後に継続の実行処理を行うようにすれば解決できる．また，このような仕組みを用意しておけば，Scheme の dynamic–wind と同様の処理を Java で記述することが容易になる．

(2) の問題を防ぐには，継続の実行時に同期ロック

を再取得するようにする方法がある．その際，複数の同期ロックがある場合にはその取得順序にも気をつける必要がある（逆に復元するとデッドロックが発生する可能性がある）．具体的には，各スレッドごとに同期ロックの取得履歴スタックを用意しておき，monitorenter が発生したタイミングで，そのスタックにプッシュするようにしておく．継続の実行処理は，まずそのスタックを参照し順番どおりに同期ロックを再取得してから，処理を継続すればよい．

3.11 スタックオーバフロー回避への応用

本手法は，スタックオーバフローの回避手段として応用することができる．メソッド呼び出しを実行するタイミングでスタックオーバフローを検知した場合，capturecont 命令と同様に，スタックフレームのイメージをフレームリストの形にしてヒープ上に退避しておいてから，メソッド呼び出しを再度行うようにすればよい．こうすることにより，ヒープの許す限り，より複雑な（再帰が深い）計算を行うことが可能になる．速度的にはスタックを大きくとった方が有利だが，スタックを大量に消費するコードは全体としては稀であり，そのためだけに大きなスタック領域を用意するのは，（特にメモリの少ない環境では）無駄である．スタック領域をインクリメンタルに拡大していく方法も考えられるが，スタック領域は固定サイズとして，必要なときはそのイメージをヒープ上に退避するようにすれば，メモリ管理が単純になり，また効率の良いメモリ運用ができる．

スタックフレームのイメージをそのままヒープ上に退避する方法や，何個かのフレームをまとめて 1 つの塊として退避する方法も考えられるが，フレーム単位でリストを構成した方が効率が良い場合が多いと思われる．k 個のフレームをまとめて退避する方法を選択した場合，スタックオーバフローが発生した後に，従来の方法より余計にかかる時間 $T_{\mathrm{ret}}(k)$ は以下のように概算できる．

$$T_{\mathrm{ret}}(k) = nC_{\mathrm{save}} + \frac{n(C_{\mathrm{new}} + C_{\mathrm{under}})}{k}$$

ここで，スタックフレーム上には n 個のフレームがあるとしており，C_{save} はフレームのコピーにかかる時間で，C_{new} は継続オブジェクトの作成にかかる時間であり，C_{under} はスタックアンダフローの処理時間である．

単純に 1 回のスタックオーバフロー処理に関する効率を考えた場合，k が大きいほどオーバヘッドは小さくなるが，実際の処理時間を見積もるには退避処理後に，再度スタックオーバフローが発生する可能性を加味しなくてはならない．スタックオーバフローが発生する可能性は，k が小さければ残りのフレームが多いので低くなる．たとえば $k = n$ の場合，退避後に 1 回再帰呼び出しが発生しただけで，再度退避処理を行う必要がある．退避処理にかかるコストは大きいので，この処理回数はできる限り抑えることが望ましい．

継続の捕捉/実行を考えた場合，フレーム単位で退避する方法が明らかに有利である．なぜなら，捕捉された継続のフレーム共有が効率良く行えるからである．しかし，スタックオーバフロー回避手段としてのみに用いる場合，そもそもフレーム共有のことは考えなくてもよいので，最適な k は 1 ではないかもしれない．ただ，スタックを大量に消費するコードは，再帰レベルの変化も激しいと推測できるので，k はあまり大きくない方が望ましい．

本手法を実装する際には，退避作業自体が Java スタックを利用する可能性に注意する必要がある．実際今回は継続オブジェクトを JNI を用いて実装したため，退避作業にコンストラクタ呼び出しを行う必要があり，Java スタックを消費する．しかし，すでにスタックオーバフローが発生しているので，新規フレームを割り当てることができない．これを回避するには，緊急用のスタック領域をあらかじめ用意しておけばよい．実装次第であるが通常は 1 フレーム分，数十バイトを用意しておけば十分である．また，退避処理中の例外処理にも注意する必要がある．退避処理中にヒープメモリが足りなくなったとき，それは本当の（回避不可能な）スタックオーバフローが発生したことを意味する．

4. ま と め

本稿では，JVM で末尾再帰の最適化を実現する方法と，一級継続を実現する方法を提案した．

末尾再帰の最適化の実現方法は，従来のバイトコードを実行前に拡張命令に自動変換するものであった．本手法を用いれば，ほとんどの場合実行速度が向上することが分かった．逆に速度が低下することもありうるが，実際にはそのようなコードは稀である．

Java 言語で書かれた従来のプログラムには，末尾再帰は一般的にはそれほど多く出現しないかもしれないが，たとえばグラフ探索などの再帰的な構造を持つオブジェクトを扱う場合には末尾再帰が頻繁に出現しているはずであり，そのようなコードに対しては速度向上が期待できる．また，最適化のために今まで手作業で末尾再帰の展開を行っていたことも考えられるが，その必要性が減少することも期待できる．また，プロ

グラムの可読性が向上することにもつながる．

本手法は Java 言語で書かれたプログラムに対しても効果があるが，他言語で JVM を利用するアプリケーションを開発することが現実味を帯びてくる．これは従来 JVM を敬遠してきた分野にも JVM の適用範囲が広がる可能性を示唆している．今回 JIT への対応は行っていないが，対応はそれほど難しくないはずである．JIT への対応を行えば，関数型言語のインタプリタとしても十分使用に耐えられるようになるのではないかと思われる．

また，JIT を用いない軽い JVM インタプリタに対しても十分効果がある．末尾再帰でスタックを消費しないので，メモリの少ないマシンでも複雑な計算ができるようになる可能性がある．

一級継続の実現についてはバイトコードを拡張することにより実現しているため，あまり現実的ではないかもしれないが，JVM で関数型言語のコンパイル結果を実行することを考えた場合には必須の機能であり，この機能もまた JVM の可能性を広げるものである．

incremental stack/heap 戦略により一級継続を扱えるようにしたが，これはフレームを一級オブジェクトとして扱えるようになったことを意味しており，ほかにも様々な用途が考えられる．たとえばこの機能を利用してフレームのダンプを行うことや，デバッグのためのインタフェースなどとして利用することも簡単に実現できる．

また，応用例として上げたスタックオーバフローの回避策としての適用はメモリの効率利用や緊急回避手段として有効である．

謝辞 本研究開発は，情報処理振興事業協会（IPA）が実施している「未踏ソフトウェア創造事業」のもとで行われました．関係者ならびにお世話になった皆様に感謝いたします．

参 考 文 献

1) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Second Edition, Addison-Wesley (1996).
2) Joy, B., Steele, G., Gosling, J. and Bracha, G.: *The Java Language Specification*, Second Edition, Addison-Wesley (1996).
3) IEEE: *IEEE Standard for the Scheme Programming Language* (*IEEE P1178*), IEEE (1991).
4) Bothner, P.: Kawa: Compiling Scheme to Java. http://www.gnu.org/software/kawa/
5) Appel, A.W.: *Compiling with Continuations*, Cambridge University Press (1992).
6) Sun Microsystems, Inc.: Java (TM) 2 SDK, Standard Edition 1.2 (2000). http://java.sun.com/products/jdk/1.2/
7) Tarditi, D., Acharya, A. and Lee, P.: No Assembly Required: Compiling Standard ML to C, Technical Report, Carnegie Mellon University (1990).
8) Gabriel, R.P.: *Performance and Evaluation of Lisp Systems*, MIT Press (1985).
9) Jaffer, A., et al.: scm (1999). http://swissnet.ai.mit.edu/~jaffer/SCM.html
10) Jaffer, A., et al.: guile (2000). http://www.gnu.org/software/guile/
11) Java-Linux Porting Team: Java Linux (2000). http://www.blackdown.org
12) Suganuma, T., et al.: Overview of the IBM Java Just-in-Time Compiler, *IBM SYSTEMS JOURNAL*, Vol.39, No.1, pp.175–193 (2000).
13) Clinger, W.D., Harhteimer, A.H. and Ost, E.M.: Implementation Strategies for Continuations, *Proc. 1988 ACM conference on LISP and functional programming* (1988).

(平成 13 年 3 月 12 日受付)
(平成 13 年 6 月 21 日採録)

山本 晃成（正会員）

1971 年生．1994 年東京理科大学理学部応用数学科卒業．同年株式会社数理システム入社．Java や Common Lisp を用いたシステム開発に従事．プログラミング言語処理系，プログラミング環境に興味を持っている．

湯淺 太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，超並列計算に興味を持っている．著書「Common Lisp 入門」（共著），「Scheme 入門」，「C 言語によるプログラミング入門」ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．