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

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

山本 晃成 [1)]　　湯淺 太一 [2)]
Akishige YAMAMOTO　Taiichi YUASA

(1) 株式会社数理システム システム技術部（〒 160–0022 東京都新宿区新宿 2–4–3 フォーシーズンビル 10 階　E-mail: yamamoto@msi.co.jp）

(2) 京都大学大学院情報学研究科通信情報システム専攻（〒 606–8501 京都府京都市左京区吉田本町　E-mail: yuasa@kuis.kyoto–u.ac.jp）

ABSTRACT. 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. 背　　景

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．

2. 目　　的

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．

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．

3. 末尾再帰の最適化の実現

（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 [4]．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及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，より効率のよ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よい．

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 4 命令を用意し、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更に 4 命令を用意する．

（2）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．

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

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.

	$operand_{curstack}$
	$\ldots$
	$operand_1$
$Frame$	$(prev, return, \ldots)$
	$local_{maxlocals}$
	$\ldots$
	$local_1$

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．

☆ 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．

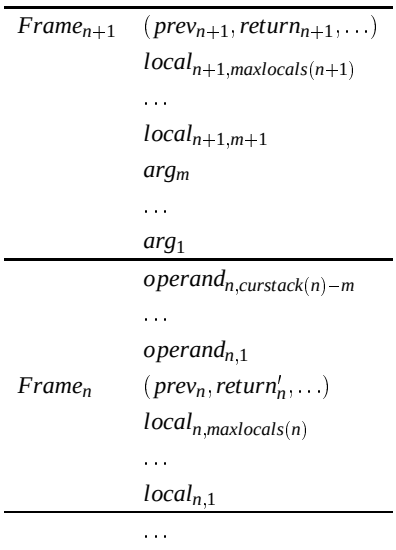

図 2 メソッド呼び出し後のフレーム状態
Fig. 2 Frame state after method invocation.

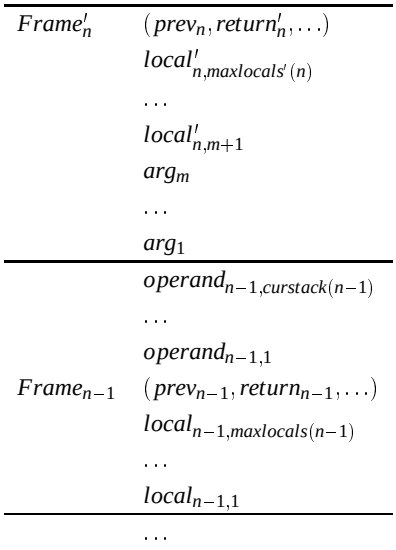

図 3 末尾再帰後のフレーム状態
Fig. 3 Frame state after tail recursion.

（3）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．

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．

（4）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．

（5）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．

(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，上で言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．

（6）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）

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．

```
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.

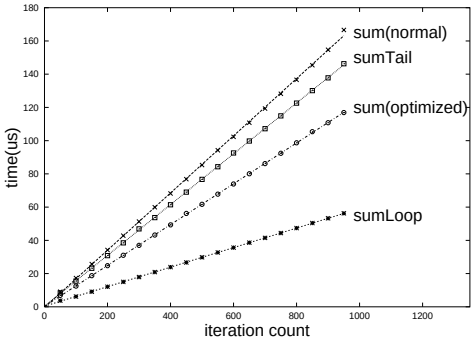

図 5 再帰回数と処理時間の関係
Fig. 5 Relation between iteration count and execution time.

	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

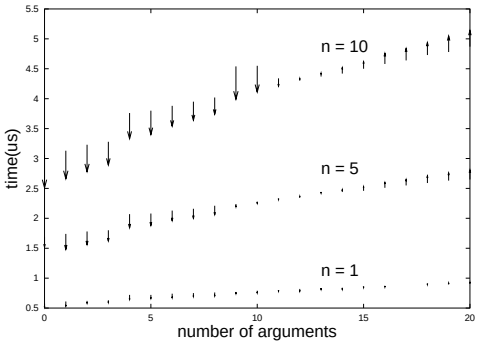

図 6　自己末尾再帰の最適化効果（n = 1, 5, 10）
Fig. 6　Effect of self tail recursion optimization (n=1, 5, 10).

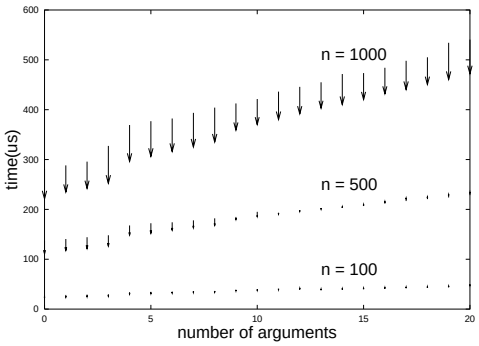

図 7　自己末尾再帰の最適化効果（n = 100, 500, 1000）
Fig. 7　Effect of self tail recursion optimization (n=100, 500, 1000).

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．

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．

（7）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，

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

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} は引数の積み上げにかかる時間，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．

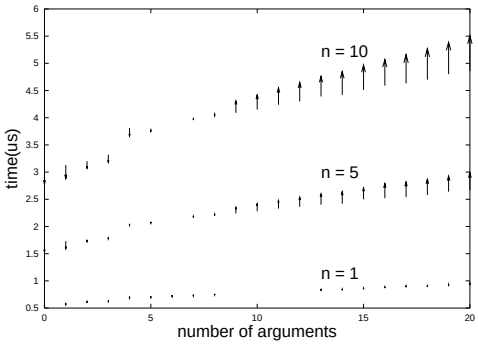

図 8　(非自己) 末尾再帰の最適化効果（n = 1, 5, 10）
Fig. 8　Effect of (non-self) tail recursion optimization (n=1, 5, 10).

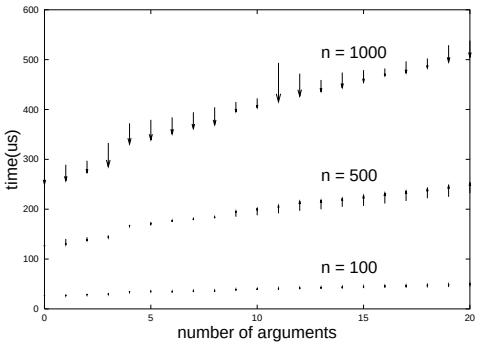

図 9　(非自己) 末尾再帰の最適化効果（n = 100, 500, 1000）
Fig. 9　Effect of (non-self) tail recursion optimization (n=100, 500, 1000).

一方，本手法で，深さ n の m 引数の再帰呼び出しを実行した場合，

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

程度の処理時間がかかると概算できる．ここで，$T_{\text{call}}(m)$ 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，

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

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，

(2)　フレームの再利用による高速化 ($C_{\text{call}} - C'_{\text{call}}$) および復帰回数が削減した分の高速化が，メモリ

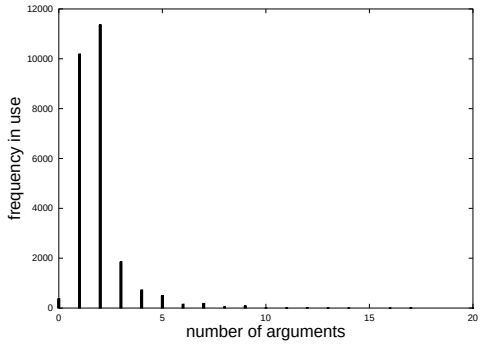

図 10 引数の個数の頻度分布
Fig. 10 Frequency distribution for number of arguments.

転送によるオーバヘッド (mC_{copy}) を上回っている場合，実行効率はよくなる．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．

4. 一級継続の実現

（1）実現手法

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 [5] 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 戦略がよいと思われる．

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．

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．

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．

（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．その時，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時は，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時と同様の処理を行えばよい．

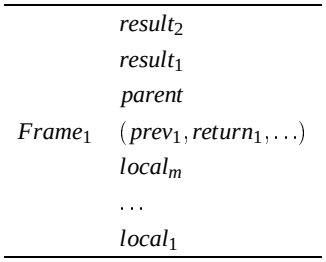

図 11 監視フレーム
Fig. 11 Sentinel frame.

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．

スタックアンダーフローの検査は，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）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．

（4）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．

```
(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).

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．

（5）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，簡単のた

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．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ー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．

5. 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，他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．

なお，本稿は文献 [6] の抜粋である．

6. 参加企業及び機関

なし．

契約件名　Java 仮想マシンへの継続渡し機能拡張

7. 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] Appel, A. W.: *Compiling with Continuations*, Cambridge University Press (1992).

[5] Clinger, W. D., Harhteimer, A. H. and Ost, E. M.: Implementation Strategies for Continuations, *Proceedings of the 1988 ACM conference on LISP and functional programming*, (1988).

[6] 山本晃成, 湯淺太一: 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, 情報処理学会論文誌:プログラミング, Vol. 42, No. SIG 11 (PRO 12) (2001).