Vol. 40 No. SIG 4(PRO 7) 情報処理学会論文誌 Jun. 2000

浅い束縛による動的スコープ変数が存在する時の末尾再帰呼び出し

前田敦司† 曽和将容†

末尾再帰的な関数呼び出しをジャンプに変換する処理は，コンパイラの最適化として広く行なわれている．特に，Lisp の一方言である Scheme 言語においては末尾再帰呼び出しを空間計算量 $O(1)$ で実行することが言語仕様で要求されている．このように，空間計算量 $O(1)$ で実行することができる末尾再帰呼び出しを真の末尾再帰呼び出し（proper tail recursion）と呼ぶ．

動的スコープを持つ変数が存在する場合，通常の実装では，スコープを規定する構文の実行が終るまで変数束縛を保持しておく必要があるため，構文の末尾で再帰的な関数呼び出しがあってもそれを真の末尾再帰呼び出しとすることができない．しかしながら，リスト構造を用いて変数束縛を保持する，いわゆる深い束縛を用いる Lisp インタプリタについては，事実上定数空間計算量で末尾再帰呼び出しを処理することができる手法が知られている．

本論文では，動的スコープ変数の実装として現在一般的な，浅い束縛を用いた場合について，真の末尾再帰呼び出しを実現するための手法について述べる．

Proper Tail Recursion with Shallow-bound Dynamically Scoped Variables

Atusi Maeda† and Masahiro Sowa†

Conversion of tail recursive function call into simple jump is a technique widely used as optimization in compilers. Especially, Scheme, a dialect of Lisp, requires as part of language specification that tail recursion be performed in $O(1)$ space complexity. Tail recursion implemented in $O(1)$ space complexity is called “proper tail recursion”.

With existance of dynamically-scoped variables, ordinary implementation keeps variable bindings until binding construct exits. Thus, syntactic tail call cannot be implemented in a properly tail recursive fashion. For Lisp interpreters which keeps variable bindings in list structures (i.e. deep binding), there is a known way to achieve almost-proper tail recursion with dynamic scoping.

In this paper, we argue about an implementation method of proper tail recursion with shallow binding, which is a common way of implementing dynamically-scoped variables in current Lisp implementations.

1. はじめに

1.1 真の末尾再帰呼び出し

Lisp など，式の評価を計算の基本原理とする言語において，関数 `f` の定義中で，関数 `g` に対する呼び出し (`g` $arg_1, \ldots arg_n$) の値がそのまま `f` の値となる時，この `g` の呼び出しは末尾再帰的（tail recursive）であるという．また，このような関数呼び出しを末尾再帰呼び出し（tail recursion）という．

より形式的には，定数・変数・関数呼び出し・`defun` による関数定義・`if` で始まる条件式からなる Lisp 言語のサブセットにおいて，式の末尾再帰的な位置を以下のように定義する．

- 関数定義 (`defun f` ($v_1 \ldots v_m$) $E_1 \ldots E_n$) において，E_n は末尾再帰的な位置にあるという．
- 条件式 (`if` $E_1 E_2 E_3$) が末尾再帰的な位置にある時，E_2 および E_3 は末尾再帰的な位置にあるという．

以上に挙げた以外の全ての式は末尾再帰的な位置にないものとする．

この定義を用いて，末尾再帰的な位置にある関数呼び出し (`f` $E_1 \ldots E_n$) を末尾再帰呼び出しと定義する．(実際の Lisp 処理系では `progn` などの構文（special form）の各々について，それぞれ同様に末尾再帰的な位置を定義する必要があるが，本論文では簡単のため上で述べたサブセットのみを扱うこととする.)

関数呼び出しは，現在の実行コンテキストをスタッ

† 電気通信大学 大学院 情報システム学研究科
Graduate School of Information Science, University of Electro-Communications

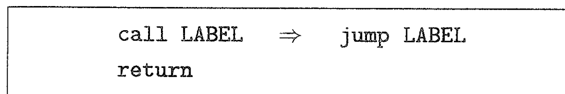


図 1 末尾再帰除去のアセンブリ言語レベルでの意味
Fig. 1 Semantics of tail recursion elimination in assembly language level.

クに退避してから目的とする関数の入口番地にジャンプするようにコンパイルされる．通常の計算機アーキテクチャでは，プログラムカウンタの退避とジャンプを同時に行なう命令（ここでは一般的に call 命令と呼ぶ）を用いることが多い．

関数 f の定義中で f 自身に対する末尾再帰呼び出しがあるとき，call 命令のかわりに単純なジャンプ命令を用いる最適化の技法は，古くから用いられてきた．この技法は末尾再帰の除去（tail recursion elimination）と呼ばれている．また，アセンブリ言語プログラマの間では，call 命令の直後にサブルーチンからの復帰命令（return 命令）が続いた時に，その 2 命令を 1 つの jump 命令に変換する技法が知られていた．この変換により，実行する命令数と消費するスタック空間のサイズの両者を節約することができる．この技法は，自分自身に対する再帰呼び出しに限らず，一般の末尾再帰呼び出しすべてに適用できる（図 1）．

高級言語において，このように一般化された末尾再帰呼び出しの最適化を行なう処理系は少ないが，Scheme 言語[19] では末尾再帰呼び出しを定数空間計算量で実行することが言語仕様で要求されている[9]．空間計算量 $O(1)$ で実行される末尾再帰呼び出しを真の末尾再帰呼び出し（**proper tail recursion**）といい，任意の末尾再帰呼び出しを定数空間計算量で実行できる時，その言語処理系は真に末尾再帰的（**properly tail recursive**）であるという[4]．

Scheme のように静的スコープルールを採用した言語では，関数呼び出しを call 命令のように（プログラムカウンタなど）なんらかのコンテキストを必ず退避する操作ととらえるかわりに，『引数を渡すジャンプ』とみなし，コンテキストの退避／回復は関数呼び出しを囲む式が必要なら行なうと解釈することによって，真の末尾再帰呼び出しを自然に実現できることが知られている[15),16]．

1.2 動的スコープ変数と末尾再帰

最初の実用 Lisp 処理系である LISP 1.5[10] 以来，Lisp の変数のスコープルールは，実行時にある変数が生存している間はプログラム中のどこからでも参照できるという，いわゆる動的スコープルールが主流であった．それに対して，Scheme およびそれに影響を受けた Common Lisp[14] は，他の多くのプログラミング言語と同様に，変数の参照はその変数が定義された構文の中だけに限られるという，静的スコープルールを採用している☆．現在広く用いられている Lisp 処理系の中では，GNU Emacs Lisp[12] が動的スコープルールを用いている．

動的スコープの実現法を大別すると，線形リストやスタックに変数名シンボルへのポインタと値の対を保持し，変数名シンボルへのポインタをキーとして線形探索することによって変数を参照する深い束縛（図 2）と，変数名シンボルの構造体の中に値を保持するスロットを設け，変数名シンボルへのポインタからのオフセットで変数を参照する浅い束縛（図 3）の 2 つの技法がある．浅い束縛では，変数を定義する構文の入口で，スタックに変数名シンボルと束縛前の古い値の対を退避し，構文から出る際にスタックから元の値を取り出してスロットの値を回復する．この処理を変数の unbind 処理と呼ぶことにする．

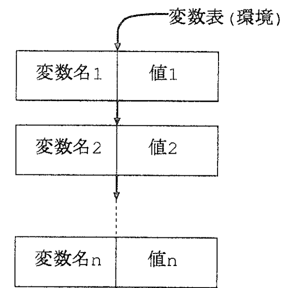


図 2 深い束縛
Fig. 2 Deep binding.

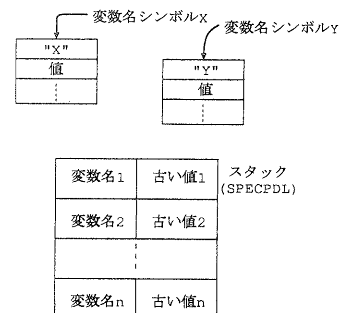


図 3 浅い束縛
Fig. 3 Shallow binding.

☆ ただし，Common Lisp においては，変数に宣言を加えることにより，その変数のスコープを動的スコープとすることもできる．

Steeleは文献15)において，真の末尾再帰呼び出しを実現するためには静的スコープルールが必須であると述べている（pp.12–13）．Steeleが挙げている理由の概略は以下の通りである．

```
(defun foo (x)
  (bar x))

(defun bar (y)
  (foo y))
```

図4 相互に末尾再帰を行なう関数
Fig. 4 Mutually tail recursive functions.

図4は，相互に末尾再帰的な呼び出しを行なう2つの関数の例である．図4のプログラムは，真に末尾再帰的な処理系では停止せず，有限のメモリのみを用いて永久に実行し続けるはずである．このプログラムを動的なスコープルールを用いた処理系で実行することを考えると，関数barの実行中は変数xが参照される可能性があるので，呼び出し側の関数fooではbarから戻ってくるまでxの束縛を捨て去ることができない．

```
foo: bind x     ; 引数をpopして束縛
     push x     ;xの値を引数スタックへpush
     call bar   ;barの呼び出し
     unbind     ;xの束縛を解除
     return
```

図5 関数fooのコード（浅い束縛）
Fig. 5 Code for function foo (shallow binding).

関数fooをコンパイルした結果を疑似コードで示すと図5のようになる．簡単のため，プログラムカウンタを退避する制御スタックと，引数の受渡しを行なう引数スタックは別であると仮定している．また，変数の束縛には浅い束縛を用いることとし，先に述べた2つのスタックとは別に，変数名と古い値の対を保持するスタック（歴史的にSpecial Variable Push Down ListまたはSPECPDLと呼ぶ）があり，bind命令で引数を引数スタックから取り出して実引数に束縛し，unbind命令で束縛を解除するものとする．

図5のコードから分かる通り，barの呼び出しは末尾再帰の位置にある関数呼び出しであるにもかかわらず，barから返ってきたのちにxのunbind処理を行なってからでないとfooからリターンすることができない．このため，barへの関数呼び出しはjump命令に変換することができない．barの中からのfooの呼び出しも同様である．したがって図4のプログラムを動的なスコープを持つLisp処理系で実行すると，SPECPDLが無制限に伸びてしまい，実行を続けることができなくなる．（barの中でxを参照していないことをコンパイラが認識すれば最適化が可能であるが，それは一般にはできない．）

しかしながら，深い束縛を行なうLisp処理系においては，以下に示す通り文献8)，11)で述べられている技法を用いて真の末尾再帰呼び出しを実現することが可能である．☆

```
foo: bind x          ; 引数をpopして束縛
     push x          ;xの値を引数スタックへpush
     save ENV        ;ENVレジスタを退避
     call bar        ;barの呼び出し
     restore ENV     ;ENVレジスタを回復
     return
```

図6 関数fooのコード（深い束縛）
Fig. 6 Code for function foo (deep binding).

今，変数名と値の対をENVレジスタが指すリストに保持する処理系を考える．関数呼び出しの前後では，このENVレジスタの退避/回復を呼び出し側で行なうものとする．この処理系で図4の関数fooを素朴にコンパイルした結果は図6のようになる．ここで，saveとrestoreは，レジスタをプログラムカウンタと同じ制御スタックに退避/回復する命令とする．

図6のコードを見ると，barから復帰した以後はENVを用いた変数参照を行なわないのでbarの実行中にENVの値が変わってしまってもかまわない．したがってENVを退避する必要はないことが分かる．すなわち，図7のように，call命令をjump命令に変換することができる．

```
foo: bind x     ; 引数をpopして束縛
     push x     ;xの値を引数スタックへpush
     jump bar   ;barの末尾再帰呼び出し
```

図7 関数fooのコード（深い束縛; スタックの抑制）
Fig. 7 Code for function foo (deep binding; stack suppressed).

しかし，このままでは変数束縛を保持するリストが

☆ 文献8)，11)ではインタプリタを対象としているが，ここで述べたコンパイラに対する手法と本質的には同じである．

barの呼び出しの際に伸びてしまい，やはり有限のメモリで実行し続けることはできない．

ここで，ENVレジスタから指されているリストの実行中の様子を図示すると図8のように，同じ名前の変数が繰り返し現われたリストになる．

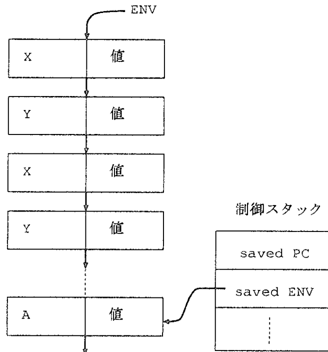


図8　変数束縛リスト
Fig. 8 Variable binding list.

ENVレジスタからこのリストをたどって線形探索を行なう際に同じ名前の変数が複数回現われていたとすると，先頭から2回目以降の出現は最初の出現によって隠されてしまい，けっして参照されることがない．このことを利用して，bind命令では変数名と値の対をリストの先頭に単につけくわえるのではなく，ENVからたどってスタックにもっとも最近退避された箇所までの間に同じ名前の変数名が現われていたら，その対の値の部分を上書きしてしまうようにする．すなわち，同じ名前の変数はたかだか1つしかリストの先頭部分（ENVからもっとも最近退避された箇所までの間）に現われないようにできる．この処理によって，リストが無制限に伸びることは無くなり，図4のプログラムを有限のメモリで実行し続けることが可能になる．リスト構造でなく，スタックを用いた深い束縛でも，同様の方式で真の末尾再帰呼び出しが実現できる．

1.3 浅い束縛と末尾再帰

上に述べた動的スコープのLispにおける真の末尾再帰呼び出しの実現法は，深い束縛を用いると変数ごとに陽に束縛を解除する必要がなく，単に変数表リスト（またはスタック）の先頭部分を捨て去るだけで必要な全ての束縛を一気に解除できることを利用している．

しかしながら，変数1つずつについて陽に束縛を解除する必要のある浅い束縛の技法を用いた場合（図5）については，Steeleの指摘が依然として有効であるように見える．変数の参照が定数時間で行なえることから，1970年ごろ以降の多くのLisp処理系では動的スコープ変数の実現に浅い束縛を使っている．そのため，浅い束縛を用いた処理系において真の末尾再帰呼び出しを実現することができれば，その実用的な価値は大きい．

本論文では，浅い束縛を用いて動的スコープルールを実現するLisp処理系において真の末尾再帰呼び出しを実装する手法を示す．またその手法の効率を改善する方法について述べ，Common Lispのように静的スコープと動的スコープが混在する場合に生じる問題点とその解決策についても議論する．

さらに，本論文で提案する手法を実際のLisp処理系（Emacs Lisp）に適用し，その効果について評価を行なう．

2. 浅い束縛における真の末尾再帰呼び出しの実現法

浅い束縛を用いたLisp処理系におけるSPECPDLは，深い束縛のスタックまたはリストと同じタイミングで伸縮する．ここで，深い束縛におけるENVレジスタと同様にSPECPDLの深さのレベルを保持するレジスタLEVELを用意する．必要ならば関数呼び出しの前後でLEVELレジスタを呼び出し側が退避/回復するものとする．束縛を解く際には変数の個数だけunbind命令を実行するのではなく，unbind_all命令によって一度にLEVELの深さまでunbind処理を行なうものとする．

このように変更した命令セットを用いて図4の関数fooをコンパイルすると，結果のコードは図9のようになる．ただし，SPECPDL_PTRはSPECPDLの先頭を指すスタックポインタとする．

```
foo: bind x          ; 引数を pop して束縛
     push x          ;x の値を引数スタックへ push
     LEVEL := SPECPDL_PTR
     save LEVEL      ;LEVEL レジスタを退避
     call bar        ;bar の呼び出し
     restore LEVEL   ;LEVEL レジスタを回復
     unbind_all      ; 深さ LEVEL まで束縛を解除
     return
```

図9　関数fooのコード（浅い束縛 + LEVEL）
Fig. 9 Code for function foo (shallow binding + LEVEL).

ここで注意すべき点は，LEVELレジスタおよびSPECPDL_PTRを退避するのは呼び出し側の責任であり，return命令を実行した時点ではまだ元の値に戻

さなくても良いという点である．例えば図 9 のコードでは， unbind_all を実行した時点でも x の束縛は解除されておらず， foo から return した時点でも x が束縛されたままになる．その後，呼び出し側が LEVEL の回復と unbind_all を行なってはじめて foo の呼び出し前の束縛状態に戻る．

SPECPDL_PTR を（すなわち変数の束縛状態を）保存しておいて，正しく関数呼び出し前の値に戻す必要があるのは，その関数呼び出しの後に変数の参照を行なう場合だけである．図 9 では， bar から復帰した後に変数の参照を行なわないので， LEVEL および SPECPDL_PTR の値は失われてもかまわない．

このことから，深い束縛を用いた場合と同様に不要な退避/回復を省くと call 命令の直後に return 命令が実行されることになり，この 2 つを jump 命令に変換することが可能となる（図 **10**）．

```
foo: bind x        ; 引数を pop して束縛
     push x        ;x の値を引数スタックへ push
     jump bar      ;bar の呼び出し
```

図 10 関数 foo のコード（浅い束縛; スタックの抑制）
Fig. 10 Code for function foo (shallow binding; stack suppressed).

このままではやはり bind 命令によって SPECPDL が伸びてしまう．そこでさらに bind 命令の意味を，変数名と古い値の対を常に SPECPDL にプッシュしてから変数名シンボルの値スロットを書き換えるのでなく， SPECPDL_PTR から LEVEL までの間に bind するシンボルがあれば，古い値の退避を行なわずに値スロットを上書きするように変更する．

以上の処理を行なうことによって，Emacs Lisp のように浅い束縛を用いる動的スコープの Lisp 処理系で真の末尾再帰呼び出しを実現することができる．

3. 効率の改善

従来の浅い束縛の実装では定数時間で行なわれていた bind 命令の処理に SPECPDL 内の線形探索の処理が加わることによって処理速度が低下することが 2 節で述べた手法の潜在的な欠点として挙げられる．例えば n 個の引数を持つ関数が末尾再帰呼び出しを行なった場合，個々の引数を束縛するたびに n に比例する計算時間がかかることになり，1 回の関数呼び出しの計算量は $O(n^2)$ となってしまう．(unbind 命令を unbind_all 命令に変えたことによって計算量のオーダーが悪化することはない．)

ここでは，真に末尾再帰的な性質を保ちながら bind 命令の処理を定数時間で行なうための改良について述べる．

2 節の方法で線形探索を行なう理由は変数の値を求めることではなく，変数が SPECPDL の最上部（ LEVEL までの間）に退避されているか否かを知ることであった．そのためには，変数が最後に退避された SPECPDL 内の位置（変数の束縛レベルと呼ぶことにする）が分かれば十分である．

浅い束縛の基本となっている，『最新の値を変数名シンボル中のスロットに置く』という考え方を流用して，変数の束縛レベルを変数名シンボルのスロットに置くことを考える．変数を束縛する際には，変数の古い値に加えて，変数の束縛レベルも SPECPDL 中に退避する．束縛されていない変数名シンボルの束縛レベルスロットの値は， LEVEL レジスタより必ず大きな値（SPECPDL が番地の若い方向に伸びる場合; 逆の場合は小さな値）を初期値として与えておく．

このようにすると線形探索の必要はなく，変数を SPECPDL に退避すべきか，そのまま上書きしてよいかの判別を定数時間で行なうことができる．ただし，

(1) SPECPDL の使用量が増大する．
(2) シンボルの大きさが増加する．
(3) 束縛レベルも退避/回復する必要があり， bind の速度は向上する可能性があるが，unbind_all の速度は低下する．

という問題点がある．このうち，問題点 2 についてまず述べる．例えば GNU Emacs では 1 万個から 5 万個程度のシンボルがメモリ上に存在する．このうち実際に変数として束縛されるものはごく一部であるにもかかわらず，メモリの使用量が 100kB 以上増大することになってしまう．

これを避けるためには，束縛レベルを記録するスロットを必要になるまで割り当てないようにすればよい．具体的には，シンボルのアドレスをキーとするハッシュ表を設けて，そこに束縛レベルを格納する．ハッシュ表の大きさは同時に束縛される変数の数に応じて確保する必要があるが，関数呼び出しのネスティングの深さがせいぜい 1000 程度とすれば，通常はその倍の 2000 程度のバケット数で十分と考えられる．(GNU Emacs の初期値では関数呼び出しの深さは 300 まで，SPECPDL のサイズは 600 エントリまでに制限されている．ただしユーザが実行時に値を増やすことも可能である．)

その他の問題点については 5 節で考察する．

4. 静的スコープ変数の混在

Emacs Lisp など，動的スコープ変数しか存在しない言語の処理系では，すべての関数呼び出しについて前節までに述べた変更を加えることによって真の末尾再帰呼び出しが実現できる．しかし Common Lisp のように動的スコープを持つ変数と静的スコープを持つ変数が混在する場合には，このままではむだが大きい．通常は静的スコープを持つ変数のほうがはるかに多く使われ，動的スコープを持つ変数を参照する頻度は比較的少ないにもかかわらず，末尾再帰的でないすべての関数呼び出しの前後で LEVEL の退避/回復と unbind_all の処理を行なうオーバーヘッドが加わってしまうからである．

このオーバーヘッドを避けるためには，LEVEL の退避/回復は呼び出された側で，LEVEL を変更する場合のみ行なうようにすればよいが，そうすると真に末尾再帰的ではなくなってしまう．

```
fun1:  save LEVEL
       ...
       bind var
       ...
       push arg
       call fun2
       restore LEVEL
       unbind_all
       return
```

図 11 LEVEL を callee で退避するコード
Fig. 11 Code with callee-saved LEVEL.

図 11 において，LEVEL の値の回復と変数束縛の解除は関数 fun1 の中で行なわなければならない．このため fun2 の呼び出しが末尾再帰呼び出しであるにもかかわらず，これを jump に変換することはできない．

ここで，末尾再帰呼び出しから返ってきた後のコードが常に同じ形（restore LEVEL，unbind_all，return）をしていることに注目し，これをサブルーチンの形にまとめ，fixup と名付ける．すると，図 11 のコードは図 12 のように書き直せる．

fun1 から fun2 を末尾再帰的に呼び出す際には，call 命令を用いず，fixup の番地を戻り番地として制御スタックに置いて jump 命令で fun2 に制御を移す．このようにして回復処理を共用することにすれば，制御スタックの先頭が fixup かどうかを調べることで，現在の関数から戻ると回復処理が行なわれるかどうか

```
fun1:  save LEVEL
       ...
       bind var
       ...
       push arg
       save fixup
       jump fun2

fixup: restore LEVEL
       unbind_all
       return
```

図 12 変数束縛の回復処理を fixup にまとめたコード
Fig. 12 Code using centralized restoration handler fixup.

を知ることができる．

末尾再帰呼び出しを行なう際に，制御スタックの先頭が fixup ならばこの関数で変数束縛の回復処理を行なう必要がない（呼び出し元で回復がなされる）ので，退避してあった LEVEL を捨ててしまい，真の末尾再帰呼び出しを行なうことができる（図 13，図 14）．

```
fun1:  save LEVEL
       ...
       bind var
       ...
       push arg
       if top == fixup then jump tail
       save fixup
       jump fun2       ; 真の末尾再帰でない
tail:
       restore LEVEL  ;LEVEL を捨てる
       jump fun2       ; 真の末尾再帰呼び出し

fixup: restore LEVEL
       unbind_all
       return
```

図 13 動的に真の末尾再帰を検出するコード
Fig. 13 Code which dynamically detect proper tail recursion.

このように fixup 処理の共通化と戻り先の動的な判定を行なう必要があるのは，動的スコープ変数を束縛した関数の中から末尾再帰呼び出しを行なう場合のみである．その他の場合（末尾再帰呼び出しでない関数呼び出しを行なう場合や，動的スコープ変数を束縛していない関数から末尾再帰呼び出しを行なう場合）に

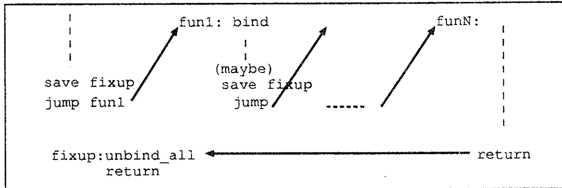


図 14 真の末尾再帰の動的な検出
Fig. 14 Dynamic detection of proper tail recursion.

は処理を変更する必要は無く，オーバーヘッドは生じない．

5. 実装と評価

動的スコープを持つ Lisp 処理系である GNU Emacs Lisp に対して変更を加え，先の 2 節および 3 節で提案した手法を実装して評価を行なった．

実験に用いた GNU Emacs のバージョンは 20.4, GNU Emacs をコンパイルする際に用いた C コンパイラは GNU C version 2.7.2.3 である．クロック 166MHz の Pentium プロセッサを備えたパーソナルコンピュータ上で計測を行なった．主記憶容量は 48MB, OS は Linux 2.0.36 である．コンパイル時のオプションはすべて Emacs のデフォルトのまま（`-g -O`）である．

比較の対象とする処理系は，オリジナルの Emacs 20.4 (orig)，2 節で述べた技法を用いて真の末尾再帰呼び出しを可能としたもの（search），線形探索を避けるために束縛レベルを保持するスロットをシンボルに加えたもの（slot），およびスロットを加えるかわりにハッシュ表を用いてメモリ容量の削減を図ったもの（hash）の 4 つである．

加えた改造はいずれの処理系も `defun` で定義される変数のみを対象としており，`let` など他の構文で宣言される変数は扱っていない．

表 1 実行形式のサイズ (バイト)
Table 1 Executable size (bytes).

処理系	ファイル	相対値	text	data
orig	5219502	1.000	927472	1931300
search	5220952	1.000	928348	1930420
slot	5257972	1.007	928308	1967316
hash	5241726	1.004	928552	1950692

GNU Emacs は（Linux を含む多くの OS では）一部の基本的な Lisp ライブラリをあらかじめ読み込んだ状態でメモリイメージをファイルにダンプし，それを実行形式としている．読み込まれる Lisp ライブラリ中のシンボルの数はおよそ 7000 個である．各処理系の実行形式のファイルサイズと，orig との相対値，およびプログラムテキストと初期化済データのサイズを表 1 に示す．3 節で述べた通り，slot ではシンボルのサイズが増加するため初期化済データのサイズがそれに応じて増大している．しかし，元の処理系との差は 1%未満にすぎない．

3 節で挙げたその他の問題点のうち，SPECPDL の使用量の増加については，GNU Emacs ではもともと SPECPDL に退避する構造体の 4 つのフィールドのうち 3 つしか使用しておらず，今回は未使用のフィールドを用いたため，orig と他の処理系の使用量は変わらない．

表 2 プログラム実行時間 (秒)
Table 2 Program execution time (seconds).

プログラム	orig	search	slot	hash
recur1	11.54*	5.64	5.55	6.04
recur10	11.69*	5.90	5.55	6.02
recur100	11.87*	8.07	5.59	6.02
tak	4.76	4.87	5.02	5.76

関数呼び出しの性能を評価するために用いたプログラムを図 15 に示す．末尾再帰的な関数呼び出しを繰り返すプログラム `recur` と，同じプログラムの末尾再帰中に出現するシンボルの数をわざと 10 個および 100 個にそれぞれ増やしたプログラム `recur10` および `recur100` の 3 つに引数 1000000 を与えた場合の実行時間（CPU 時間）を表 2 に示す．この 3 つのプログラムは orig を除く処理系では `return` を 1 回しか実行しない．また，関数呼び出しのベンチマークである `tak` 関数 [7] に引数 21, 14, 7 を与えた結果も同じく示してある．実行時間は全て 10 回の測定を行なったうちの最良値である．

orig は真に末尾再帰的でないので，`recur`，`recur10`，`recur100` に大きな引数を与えることができない．表中で * をつけた値は，引数として 1000 を与えた実行を 1000 回繰り返した結果から，1000 回の空ループの実行時間（10 回の平均値）を差し引いた値である．

計測結果から，引数の数が極端に増えた場合には search の実行速度が大きく低下することがわかる．slot および hash では速度の低下は見られない．また，`unbind_all` および `return` の処理速度が全体の実行時間に大きく影響していることが分かる．`recur1`～`recur100` で orig の実行時間が遅いのは，orig だけが `unbind` と `return` を数多く行なうためである．search の `unbind_all` と `return` の処理速度は処理内容から見て orig とほぼ同じであり，引数束縛の漸近的計算量

```
(defun recur (n)
  (if (> n 0)
      (recur (1- n))))
(defun recur10 (n)
  (r10 n 0 0 0 0 0 0 0 0 0))
(defun r10 (n a b c d e f g h i)
  (recur n))
(defun recur100 (n)
  (r100 n
      0 0 0 0 0 0 0 0 0 0
      ;; (略)
      0 0 0 0 0 0 0 0 0 0))
(defun r100
  (n
   a0 a1 a2 a3 a4 a5 a6 a7 a8 a9
   ;; (略)
   a90 a91 a92 a93 a94 a95 a96 a97 a98 a99)
  (recur n))
(defun tak (x y z)
  (if (not (< y x))
      z
    (tak (tak (1- x) y z)
         (tak (1- y) z x)
         (tak (1- z) x y))))
```

図 15 計測に用いたプログラム
Fig. 15 Program for measurement.

が slot や hash に比べて大きいにもかかわらず tak を orig とほぼ同じ時間で実行している．slot および hash は unbind_all と return の処理が遅く（3 節の問題点 3），tak での実行速度が劣る結果となった．特に，hash の速度の低下は大きく，tak では orig と比較して約 21% 実行時間が増加している．search と slot に関しては tak においても速度の低下はわずかであり（slot で約 5%），このいずれかが実用的な選択肢と思われる．

表 3 N 引数関数呼び出し/復帰の処理時間
Table 3 Execution time of N-ary function call/return.

実装	時間 (μsec)
orig	$1.98N + 4.65$
search	$0.002N^2 + 1.98N + 4.95$
slot	$2.02N + 4.96$
hash	$2.44N + 5.21$

関数呼び出しのオーバーヘッドをより詳細に調べるため，何もしないで復帰するだけの関数の実行時間を，さまざまな個数の引数について計測した結果を表 3 に示す．この値は実引数の push と bind，call，return，および unbind_all のすべての処理を総合した実行時間を比較したものとなる．例えば引数が 1 個の場合では，orig と比較して 1 組の関数呼び出し/復帰ごとにそれぞれ約 4%（search），5%（slot），16%（hash）のオーバーヘッドがある．hash を除いては N の項の係数は小さく，引数の数が増えるにしたがって orig に対する相対的なオーバーヘッドが減少する傾向にある．search の場合，引数が極端に多くなると N^2 の項が支配的になるが，引数が十個程度までの範囲ならオーバーヘッドの相対値は減少する．

前述の tak プログラムでは，バイトコードの静的な命令数は 26 命令で，うち 4 命令 (15.3%) が call 命令である．動的な命令カウントでは，実行された全命令（4155456 命令）の 9.5%（395756 命令）が call 命令であった．表 3 から，orig におけるこのプログラムの call および return 命令の実行時間の合計は

$$(1.98 \times 3 + 4.65) \times 395756 \times 10^{-6} = 4.19\,(\text{秒})$$

となり，全実行時間のほとんど（88%）を占めている．したがって，上で述べたオーバーヘッドはほとんどそのまま実行時間の増加に結びつくことになる．

ただし，search, slot, hash ではいずれも，call 命令のうち（静的/動的命令数とも）25%が真の末尾再帰に変換され，return および unbind_all 命令の実行回数がそれに応じて削減されているため，速度低下は表 3 から求めた値よりも少なくなる．

Emacs 20.4 に付属する Emacs Lisp ファイル (591 ファイル/490213 行) をコンパイルした結果では，静的なバイトコード命令数は 444265，うち call 命令の数は 93442 命令 (21.0%) であった．このうち，本論文の手法によって 13856 命令（call 命令の 14.8%, 全命令の 3.1%) を真の末尾再帰呼び出しに変換することができた．これらのプログラムでの call 命令の動的な頻度は不明であるが，静的な頻度が tak より高く，また真の末尾再帰に変換できる率も低いことからオーバーヘッドは相対的に tak の場合より増加すると予想できる．しかし上に述べた通り，たとえ実行時間のすべてを関数呼び出し/復帰関連の命令が占めたとしても，hash 以外の実装では実用上のオーバーヘッドは高々 4%〜5%に抑えられる．

6. 関連する研究

手続き型言語において，再帰呼び出しをソース言語レベルで GOTO 文に変換する手法についての研究は古くから行なわれており [2]，ソースプログラムがもと

もと末尾再帰呼び出しである場合については自動的にこの変換を行なうコンパイラもある[13),18)]．また，関数型言語では Scheme や ML[1)] など，真に末尾再帰的な処理を実現しているものが多い．これらはいずれも静的スコープルールを前提としている．

末尾再帰的でない再帰呼び出しを末尾再帰呼び出し（あるいは GOTO）に変換するプログラム変換の技法については，古くから研究が行なわれてきた[3),17)]．この流れに沿った最近の成果として例えば文献 6) がある．これらの技法は，プログラム中で使われている既知の関数の性質を利用して関数の適用順序を変え，再帰的呼び出しを可能な限り末尾再帰呼び出しに変換するものである．これらの技法を実装したコンパイラは筆者らの知る限りでは未だ存在しない．コンパイラに実装する上では，関数の適用順序を変更することによるエラー検出タイミングの変化などについて慎重に検討する必要があると思われる．本論文で述べた技法は関数の適用順序を変えない．また本論文の技法は，ソースコード上で末尾再帰呼び出しに変換された後でもそのままでは真の末尾再帰呼び出しにできない場合の処理を目的としている．

浅い束縛を高速化するためにネスティングのレベルを変数シンボルごとに記録する手法は文献 20) と類似しているが，文献 20) は末尾再帰呼び出しについてなんら考慮していない．ネスティングのレベルを記録する際にハッシュ表を用いて記憶領域を節約する技法は文献 5) でリファレンスカウントを記録する技法と本質的に同じである．

7. おわりに

浅い束縛を用いて動的スコープルールを実現する Lisp 処理系において，末尾再帰的呼び出しを定数空間計算量で実行するための手法を提案し，実際に GNU Emacs Lisp 上に実装することによって真の末尾再帰呼び出しが実現できていることを確認した．

これにより，従来は繰り返しのための専用の構文（`while`）を用いなければ記述できなかったプログラムも，より関数型プログラミングに近い形で記述することが可能となった．5 節で述べた通り Emacs 20.4 に付属する Emacs Lisp ファイルに現われる `call` 命令のうちの 14.8%を本論文の手法によって真の末尾再帰呼び出しに変換することができた．これらのプログラムが真の末尾再帰呼び出しのセマンティクスを前提とせずに書かれたものであることを考慮すれば，この変換できた命令の率は高いと言えるが，末尾再帰呼び出しを積極的に用いたプログラムではこの率はさらに上昇すると考えられる．

また，関数呼び出しに関して実行時間の計測を行ない，提案した手法のオーバーヘッドがどの程度であるかを評価した．引数が十個程度までの場合，オーバーヘッドは高々4%〜5%であり，この値は実用上許容できる範囲と思われるが，今回実装に用いた GNU Emacs Lisp は，最適化された高性能な処理系とは言えず，`return` や `unbind_all` など個々の操作が全体の実行時間に占める割合は今後のチューンアップにより変わりうる．

真の末尾再帰呼び出しを実現できたことはガーベッジコレクション（GC）に対しても好ましい影響を与える．SPECPDL 内に退避される変数束縛の数が減少するため，GC においてルートを走査する時間は減少する．また，同じ名前の変数に隠蔽されて決して参照されない（すなわち事実上すでにゴミである）にもかかわらず SPECPDL 内に退避されていたデータを早期に解放することができるため，マーキング時間やヒープの占有量も減少すると考えられる．これらの点に対する定量的な評価も今後の課題である．

また，4 節で述べた静的スコープ変数と動的スコープ変数が混在する場合について，今後実装および性能の評価を行なっていく予定である．

参考文献

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

2) Auslander, M. A. and Strong, H. R.: Systematic Recursion Removal, *Communications of the ACM*, Vol. 21, No. 2, pp. 127–134 (1978).

3) Burstall, R.M. and Darlington, J.: A Transformation System for Developing Recursive Programs, *Journal of the ACM*, Vol. 24, No. 1, pp. 44–67 (1977).

4) Clinger, W. D.: Proper Tail Recursion and Space Efficiency, *Proc. SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pp. 174–185 (1998).

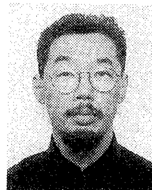
5) Deutsch, L. P. and Bobrow, D. G.: An Efficient Incremental Automatic Garbage Collector, *Communications of the ACM*, Vol. 19, No. 7, pp. 522–526 (1976).

6) 二村良彦, 大谷啓記: 線形再帰プログラムからの再帰除去とその実際的効果, コンピュータソフトウェア, Vol. 15, No. 3, pp. 38–49 (1998).

7) Gabriel, R. P.: *Performance and Evaluation of LISP Systems*, Stanford University Press, Stanford (1982).

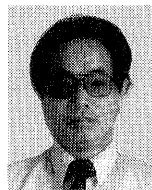
8) 菱沼千明, 山下堅治, 中西正和: LISP インタープリタにおけるスタック技法と a リストの抑制法, 情報処理, Vol. 17, No. 11, pp. 1002–1008 (1976).
9) Kelsey, R., Clinger, W. and Rees, J.(eds.): *Revised^5 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, Vol. 33, No. 9, ACM (1998).
10) McCarthy, J., Abrahams, P. W., Edwards, D.J., Hart, T.P. and Levin, M.I.: *LISP 1.5 Programmer's Manual*, Cambridge, MA (1965).
11) Nakanishi, M., Hishinuma, C. and Sakai, T.: A Design of LISP Interpreter for Minicomputers, *Proc. IFIP Conf. Software for Minicomputers*, North-Holland Pub. Co. (1976).
12) Stallman, R. M.: *The GNU Emacs Lisp Reference Manual*, Free Software Foundation, Cambridge, MA, 2.3 edition (1994).
13) Stallman, R. M.: *Using and Porting the GNU Compiler Collection (GCC)*, Free Software Foundation, Cambridge, MA, for gcc ver. 2.96 edition (1999).
14) Steele Jr., G. L.: *Common Lisp – the Language*, Digital Press, Woburn, MA, second edition (1990).
15) Steele Jr., G. L.: Lambda, the Ultimate Declarative, MIT AI Memo 379, Massachusetts Institute of Technology, Cambridge, Mass. (1976).
16) Steele Jr., G. L.: Debunking the "Expensive Procedure Call" Myth, or Procedure Call Implementations Considered Harmful, or LAMBDA, the Ultimate GOTO, *ACM Conference Proceedings*, pp. 153–162 (1977).
17) Strong, Jr., H. R.: Translating Recursion Equations into Flow Charts, *Journal of Computer and System Sciences*, Vol. 5, No. 3, pp. 254–285 (1971).
18) Sun Microsystems Inc.: *The C User's Guide*, Palo Alto, CA, 4.2 edition (1996).
19) Sussman, G. J. and Steele Jr., G. L.: Scheme: an Interpreter for Extended Lambda Calculus, MIT AI Memo 349, Massachusetts Institute of Technology, Cambridge, Mass. (1975).
20) White, J. L.: Constant Time Interpretation for Shallow-bound Variables in the Presence of Mixed SPECIAL/LOCAL Declarations, *Conf. Record of the ACM Symp. LISP and Functional Programming*, pp. 196–200 (1982).

(平成 11 年 10 月 15 日受付)
(平成 12 年 2 月 22 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）．現在 電気通信大学大学院情報システム学研究科助手．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などに興味を持つ．日本ソフトウェア科学会，ACM 各会員．



曽和　将容（正会員）

1978 年名古屋大学博士課程修了．群馬大学助教授，名古屋工業大学教授を経て現在電気通信大学大学院教授．1978 年からマルチマイクロコンピュータ，データフローコンピュータなど並列コンピュータと並列プロセッサの研究に従事．現在分散並列シリアルの統合されたコンピュータ環境構築に興味を持っている．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会各会員．