# 継続の生成におけるスタックコピーの遅延

鵜川始陽[†] 皆川宜久[†] 小宮常康[†]
八杉昌宏[†] 湯淺太一[†]

実行にスタックを利用している処理系でファーストクラスの継続を生成する場合，スタックの内容をヒープにコピーするのが一般的である．スタックの内容をヒープにコピーする戦略には様々なものが提案されているが，実際の処理系の多くはスタック法を採用している．スタック法では継続の生成のたびに処理系のスタック全体の内容をヒープにコピーする．このように，スタック法は動作が単純なので簡単に実装でき，また他言語呼び出しをサポートした処理系でも利用できるという長所がある．しかし，継続の生成に時間がかかり，また，メモリ効率が悪いという欠点があり，利用の妨げとなる場合もある．本発表では，スタック法による継続の生成に対する効率化手法として，スタックのコピーの遅延を提案する．継続の生成によりコピーされるスタックの内容は，継続を生成した関数からリターンするまでスタック上にも残っている．したがって，スタックのコピーを継続を生成した関数からリターンするまで遅らせてもよい．もし，生成した継続が，スタックをコピーする前にごみになれば，スタックのコピーを省略できる．本発表では，さらに，スタックコピーの遅延により可能となる，継続オブジェクトの部分的共有も提案する．これらの効率化手法を Scheme 処理系に組み込んで性能を評価したところ，継続を使ったプログラムの実行効率が改善された．

# Lazy Stack Copying for First-class Continuations

Tomoharu Ugawa,[†] Nobuhisa Minagawa,[†] Tsuneyasu Komiya,[†]
Masahiro Yasugi[†] and Taiichi Yuasa[†]

In order to capture first-class continuations, most stack-based implementations copy contents of the stack to the heap. While various implementation strategies for copying are proposed, most implementations employ the stack strategy. With this strategy, the entire stack contents are copied to the heap whenever a continuation is captured. This simple strategy is easy to implement and can be used for implementations with foreign language interface. However, continuations with this strategy spend a lot of time and memory for their creations and invocations, and this week point often discourages programmers from using first-class continuations. We propose a lazy stack copying technique. The contents of the stack to copy will be preserved on the stack until returning from the function that is capturing the continuation. So we delay stack copying for a continuation until the function returns. We can avoid stack copying if it is detected that the continuation became a garbage before the function returns. In addition, we propose another technique, partial sharing of the copied stack, which is realized by using the lazy stack copying technique. We applied these techniques to Scheme interpreters and found that the proposed techniques improve runtime and memory efficiency of programs that use continuations.

## 1. はじめに

継続とは，計算のある時点での残りの計算を表したものである．Scheme[1)]などの言語では，継続をファーストクラスオブジェクトとして取り出す（これを継続のキャプチャという）ことができる．これを用いると，例外処理などの非局所的脱出やマルチタスクなど，様々な制御構造をユーザレベルで実現することができる．

Scheme では，

`(call/cc` ⟨*proc*⟩`)`

を使って ⟨*proc*⟩ の呼び出しの継続をキャプチャすることができる．ここで ⟨*proc*⟩ は 1 引数の関数である．キャプチャされた継続は，⟨*proc*⟩ の引数に渡される．継続に 1 つの引数を与えて呼び出すことで，継続をキャプチャした `call/cc` 式以降の計算に制御が移る．このとき，継続の呼び出しにおける引数が `call/cc` からの返り値となる．継続は，⟨*proc*⟩ の実行中に限らず，プログラムの実行中いつでも呼び出すことができる．

† 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

ところで，⟨*proc*⟩ の呼び出しの継続を実行するためには，`call/cc` が呼び出されたときにアクティブであったすべての関数フレーム（呼び出されていてまだリターンしていない関数の関数フレーム）中に格納されている環境や制御情報が必要になる．多くの処理系では，関数フレームをスタック上に確保している．したがって，リターンすると関数フレームは解放され，その領域は次に呼び出される関数の関数フレームのために再利用されてしまう．このような処理系では，`call/cc` が呼び出されたときにスタックの内容をコピーする方式が一般的である．スタックは一般のセルなどに比べ非常に大きいため，`call/cc` の処理は非常に時間のかかる処理となっている．

本発表では，1）継続が「ごみ」となると（つまり，Scheme レベルで参照がなくなると）それ以降決して呼び出されない，2）`call/cc` からリターンするまでは，継続の実行に必要な関数フレームは破壊されないという点に着目して，`call/cc` を含むプログラムの実行効率を高める手法を示す．

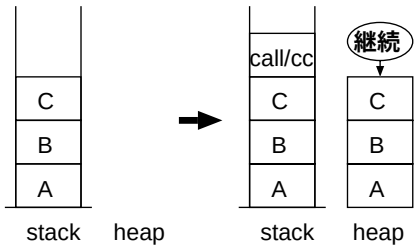


**図 1** スタック法による継続のキャプチャ
Fig. 1 Continuation capturing in the stack strategy.

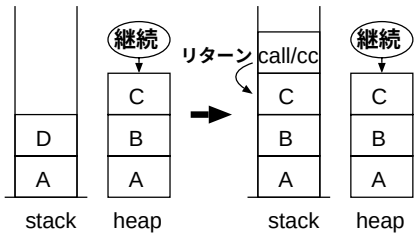


**図 2** スタック法による継続の呼び出し
Fig. 2 Continuation invocation in the stack strategy.

## 2. 従来の継続の実装

まず従来の継続のキャプチャの方式を説明する．多くの処理系は，実行のためにスタックを用いている．スタックにはリターンアドレス（関数の実行が終了したときに制御を戻す番地）などの制御情報や，実行中の関数の環境などが積まれている．このような処理系では，継続の実行のためにスタックの内容が必要になる．したがって，継続オブジェクトには，キャプチャ時にスタック上にあった関数フレームを再構成するための情報が必要になる．一般には継続をキャプチャするときに，スタックの内容をヒープにコピーして保存する．このとき，先に行われた継続のキャプチャなどにより，すでにコピーされた部分を共有するなど，様々なスタックのコピーの戦略が提案されている．ここでは，本発表が対象とするスタック法[2]について説明する．

スタック法は，最も単純な戦略である．継続のキャプチャ時や呼び出し時には，スタックの構造を無視して内容全体を一括して扱う．

(`call/cc` ⟨*proc*⟩)

の実行では，まず，スタックボトムからスタックポインタまでの範囲を格納するための領域をヒープ上に確保する．そして，スタックの内容をヒープ上にコピーする（**図 1**）．処理系によっては，制御スタックと値スタックというように，複数のスタックを使って実行している場合がある．このような場合は，継続を実行するのに必要なすべてのスタックがコピーの対象となる．スタックのコピーが終わると，スタックのコピーにスタックポインタの位置などのいくつかの情報を加えた継続オブジェクトを引数に，⟨*proc*⟩ を呼び出し，⟨*proc*⟩ の実行を開始する．継続が呼び出されたときは，ヒープにコピーしておいたスタックの内容をキャプチャしたときと同じメモリアドレスに一括して書き戻す．これにより，スタックは呼び出した継続をキャプチャした `call/cc` が呼び出されたときの状態に戻る（**図 2**）．この状態でリターンすることで，`call/cc` 式以降の実行が始まる．

スタック法は C 言語で記述された Scheme 処理系の多くで採用されている．C 言語で記述された Scheme 処理系では，C 言語の再帰呼び出しで Scheme 言語の再帰呼び出しを表現していることが多い．このような処理系では，継続をキャプチャするときに C 言語のスタックを保存する必要がある．C 言語のスタックは一般には構造が分からないので，このような処理系では，スタック全体をそのままコピーするスタック法が適している．また，他言語インタフェースを備えた処理系では，継続をキャプチャするときに他言語のスタックもコピーしなければならないので，スタック法が適している．このような理由から，スタック法は SCM[3] や MzScheme[4),5)] など多くの処理系で採用されている．

しかし，スタックは一般のセルなどに比べて非常に大きいため，スタック全体をコピーする処理は非常に時間がかかる．また，保存しておくために大量のメモ

リを消費するというデメリットがある．

## 3. 提案手法

### 3.1 スタックのコピーの遅延

ある関数 $F$ で，(call/cc ⟨*proc*⟩) を評価すると，call/cc は ⟨*proc*⟩ を呼び出す．関数フレームはスタックに積まれているので，⟨*proc*⟩ の実行中は関数 $F$ やそれを呼び出した関数の関数フレームが変更されることはない．この特徴を利用すると，継続をキャプチャするのと同時にスタックの内容をコピーする必要はない．継続をキャプチャした時点では，継続オブジェクトはスタックのコピーを参照する代わりに，スタックを直接参照しておけばよい．

提案手法では，継続をキャプチャするときは，スタックのコピーを持たない継続オブジェクトを生成する．この継続オブジェクトを「不完全な継続オブジェクト」と呼ぶことにする．不完全な継続オブジェクトは，キャプチャ時のスタックポインタだけを持っている．

⟨*proc*⟩ からリターンすると，すぐに call/cc からリターンし，関数 $F$ に制御が移る．その後 $F$ の実行にともなって $F$ の関数フレームの内容が変更され，call/cc でキャプチャした継続を実行するのに必要となる情報が失われる可能性がある．そのため，遅くとも ⟨*proc*⟩ からリターンするまでにスタックの内容のコピーを作って，継続オブジェクトがそれを参照するようにしておかなければならない．継続がスタックのコピーを参照するようにすることを，継続の「昇格」と呼ぶことにする．

提案手法では，⟨*proc*⟩ からリターンするときに継続を昇格させる．このように，特定のリターンのときに特別な処理を行うことを，文献 6) に倣って「リターンバリア」と呼ぶことにする．

**図 3** (a) で call/cc を呼び出すと，図 3 (b) のように不完全な継続オブジェクトが生成される．⟨*proc*⟩ の実行中は，call/cc よりも下の関数フレームの内容は変更されない．⟨*proc*⟩ からリターンすると，関数 $F$ の関数フレームの内容が変更されるようになるので，その前にスタックのコピーを行い，継続を昇格させる (図 3 (c))．昇格した継続は従来の継続と同様に扱うことができる．

call/cc がキャプチャした継続は，⟨*proc*⟩ の実行中のある時点で，それ以降決して呼び出されることがないと分かることがある．たとえば，継続オブジェクトへの参照がなくなる場合である．決して呼び出されることがなくなった継続を「ごみ」と呼ぶことにする．継続がごみになった場合は，⟨*proc*⟩ からリターンするときに昇格する必要はない．したがって，⟨*proc*⟩ の実行中にごみとなった継続のためのスタックのコピーを省くことができる．

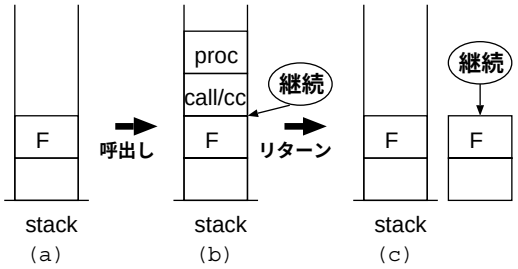


**図 3** call/cc の呼び出し
Fig. 3 Invocation of call/cc.

このように，スタックの内容のコピーを，⟨*proc*⟩ からリターンするまで遅らせることにより，不要なスタックのコピーを省く手法を「スタックのコピーの遅延」と呼ぶことにする．

提案手法では，不完全な継続オブジェクトを呼び出したときは，継続を実行するために必要な情報がスタックの底のほうに残っていることが保証されている．したがって，不完全な継続を呼び出したときは，その継続をキャプチャした call/cc より上に積まれた関数フレームを捨てて，call/cc からリターンするだけでよい．従来の方式では，継続の呼び出し時には継続の持つスタックのコピーをスタックに書き戻すという操作が必要であったが，この場合は，そのようなコピーは発生しない．

⟨*proc*⟩ の実行中に，⟨*proc*⟩ の呼び出し以前にキャプチャされた継続が呼び出されると，関数 $F$ やその呼び出し元の関数フレームが壊される．したがって，このような継続の呼び出しが行われたときも，⟨*proc*⟩ を呼び出すときにキャプチャした継続がごみになっていなければ，継続は昇格する必要がある．

継続の呼び出しにより，別の継続が昇格する例を**図 4** に示す．継続 1 は関数 $F$ が呼び出されるよりも先にキャプチャしてある継続である．関数 $F$ で (call/cc ⟨*proc*⟩) を評価すると，図 4 (a) のようになり，不完全な継続オブジェクトである継続 2 が生成される．ここで，継続 1 を呼び出すと，関数 $F$ の関数フレームが壊されるので，図 4 (b) のように継続 2 が昇格する．

⟨*proc*⟩ の実行中に，⟨*proc*⟩ へ受け渡された継続がまったく使われないで，⟨*proc*⟩ からリターンすることがある．その場合は，スタックのコピーは行われない．また，継続を非局所的脱出として利用する場合もスタックのコピーは行われない．非局所的脱出のための

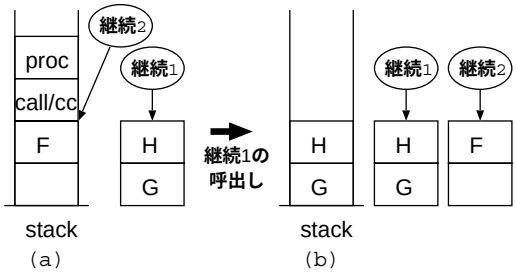


**図 4** 継続の呼び出し
Fig. 4 Invocation of a continuation.

継続は，⟨*proc*⟩ の実行中にのみ呼び出される．⟨*proc*⟩ の実行中に呼び出されずにリターンすると，その継続はごみになるので，スタックのコピーは行われない．⟨*proc*⟩ の実行中に呼び出されたときも，その呼び出し以降に再度呼び出されることがないので，やはり，スタックのコピーは行われない．さらに，継続が呼び出されたときのスタックのコピーの書き戻しも行われない．このように，非局所的脱出のための継続では，スタックのコピーがまったく行われない．

### 3.2 スタックコピーの共有

提案手法では，継続のキャプチャによるスタックのコピーを，継続をキャプチャした関数にリターンするまで遅らせる．したがって，`call/cc` が入れ子状態になって呼び出された場合，**図 5** (a) のように，不完全な継続が複数ある状態になる．矢印で示す ⟨*proc*⟩ からリターンするときに，継続 1 が昇格する必要があれば図 5 (b) のようになる．このとき，継続 1 が持つスタックのコピーには，将来継続 2 が昇格するときにコピーするスタックの内容（図 5 (b) の斜線部分）が含まれている．したがって，継続 2 が継続 1 のスタックのコピーの一部を共有することにすれば（図 5 (c)），非常に短い時間で継続 2 が昇格することができ，また，継続 2 のためのスタックのコピーを作るメモリ領域を確保する必要がなくなる．

提案手法では，ある継続 $k$ が昇格したとき，不完全な継続オブジェクトは $k$ のためにコピーしたスタックの一部を共有することで昇格することにする．これを「スタックのコピーの共有」と呼ぶことにする．

## 4. 実　　装

### 4.1 ごみの判定

処理系は，`call/cc` からリターンするとき，キャプチャした継続オブジェクトがすでにごみになったかどうかを判定する必要がある．継続オブジェクトがごみになったかどうかの判定は，メモリ管理における「ごみ」と同様に，スタックや大域変数から継続オブジェクトへの到達性で近似すると，小さなオーバヘッドで判定できる．

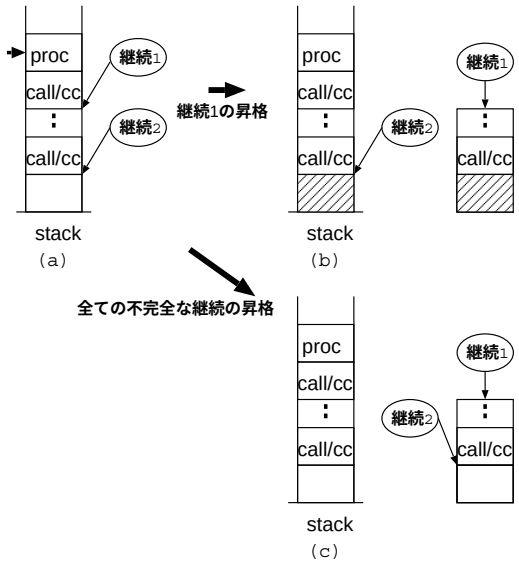


**図 5** スタックのコピーの共有
Fig. 5 Sharing a stack copy.

`(call/cc` ⟨*proc*⟩`)` を評価すると，不完全な継続オブジェクトが作られ，それを引数として ⟨*proc*⟩ が呼び出される．この継続を $k$ とする．⟨*proc*⟩ は，一般に $k$ への参照を，局所変数（スタック上），大域変数，ヒープ上のオブジェクトのスロットにコピーしながら実行する．⟨*proc*⟩ からリターンするとき，$k$ がごみになっているかどうかは，次のようにして近似できる．

- ⟨*proc*⟩ の実行中に $k$ の参照が局所変数だけにしか作られなかったときは，⟨*proc*⟩ からのリターンと同時に $k$ はごみになる．
- ⟨*proc*⟩ の実行中に $k$ の参照が大域変数かヒープ上のオブジェクトのスロットに作られたときは，⟨*proc*⟩ からリターンしても $k$ への参照が残り，ごみにならない可能性がある．
- ⟨*proc*⟩ の実行中に $k$ の参照が局所変数だけにしか作られなかったときでも，⟨*proc*⟩ の返り値が $k$ であれば，$k$ はごみにならない．

**図 6** (a) では，継続オブジェクトへの参照がスタック上にしかないので，⟨*proc*⟩ からリターンするときに継続オブジェクトがごみになる．しかし図 6 (b) ではヒープ上のオブジェクトから継続オブジェクトへの参照があるので，⟨*proc*⟩ からリターンしても継続オブジェクトがごみにならない可能性がある．

この性質を利用して，継続をキャプチャした関数か

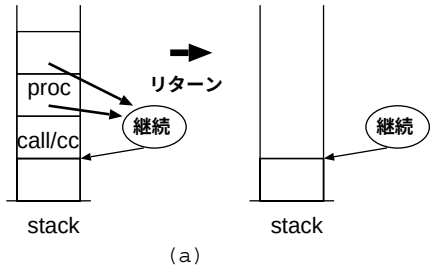


**図 6** 継続オブジェクトへの参照
Fig. 6 Reference to a continuation object.

らリターンするときに，大域変数かヒープ上のオブジェクトから継続への参照が作られた場合か，継続が返り値になっている場合だけ継続が昇格することにする．

まず，継続オブジェクトに「昇格予約フラク」というフラグを用意する．大域変数やヒープ上のオブジェクトから継続への参照が作られると，参照される継続の昇格予約フラグをセットする．これは，ライトバリアと，オブジェクトの生成時のチェックにより実現できる．継続をキャプチャした関数からリターンするとき，

- 昇格予約フラグがセットされている，または，
- 返り値が継続である，

場合に継続を昇格する．

関数クロージャは生成されたときの環境を持つヒープ上のオブジェクトである．したがって，クロージャを生成するときは，環境を調べて，継続への参照が環境に含まれていれば，その継続オブジェクトの昇格予約フラグをセットする必要がある．たとえば次のようなプログラムでは，(*) 印の行で生成されるクロージャの環境に継続 $k$ が含まれる．

```
(define (f) (call/cc
              (lambda (k)
                (lambda () k))))  ; (*)
```

しかし，たとえば次のようなプログラムでは，本来 (*) 印の行で生成されるクロージャの環境に継続 $k$ が含まれているが，クロージャを実行しても使われることがないので，昇格予約フラグをセットしなくてよい．

```
(define (f) (call/cc
              (lambda (k)
                (lambda () 1))))  ; (*)
```

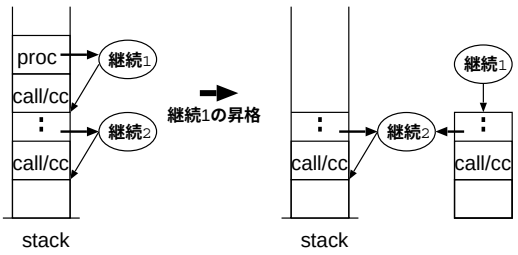


**図 7** 昇格した継続オブジェクトからの参照
Fig. 7 Reference from a promoted continuation object.

S 式を実行前にコンパイルする処理系では，プログラムを解析して，クロージャ内で使わない変数はクロージャの環境に含めないようにしていることが多い．このような処理系では，容易に使われないクロージャの環境からの参照により昇格予約フラグがセットされるのを防ぐことができる．

継続オブジェクト自身もヒープ上のオブジェクトである．昇格した継続はスタックのコピーを持つため，スタックに不完全な継続への参照が含まれている場合はその不完全な継続の昇格予約フラグをセットする必要がある．**図 7** では，継続 2 が継続 1 から参照されているので，昇格予約フラグをセットする必要がある．しかし，スタックのコピーの共有を行うと，ある継続が昇格するとき，他の不完全な継続も同時に昇格する．したがって，昇格した継続の持つスタックのコピーから参照されている継続は，すべて昇格している．

### 4.2 不完全な継続オブジェクトの管理

提案方式では，継続が呼び出されたとき，その呼び出しで壊される関数フレームを必要とする継続は昇格する．また，1 つ継続が昇格するときに他の不完全な継続も昇格する．このように，スタックトップにある関数がキャプチャした継続以外の継続も昇格することがある．このようなとき，処理系はごみでない不完全な継続をすべて見つけなければならない．

そこで，不完全な継続オブジェクトを溜めるスタックを用意する．これを「継続スタック」と呼ぶことにする．継続がキャプチャされると，不完全な継続オブジェクトが作られて継続スタックに積まれる．そして，継続をキャプチャした関数からリターンすると，継続スタックから継続をポップして，必要であれば昇格させる．また，これ以外でも継続が昇格するときは，継続スタックから継続をすべてポップする．継続の呼び出しを行っても，継続スタックは継続をキャプチャしたときの状態には戻さない．これは，一度昇格した継

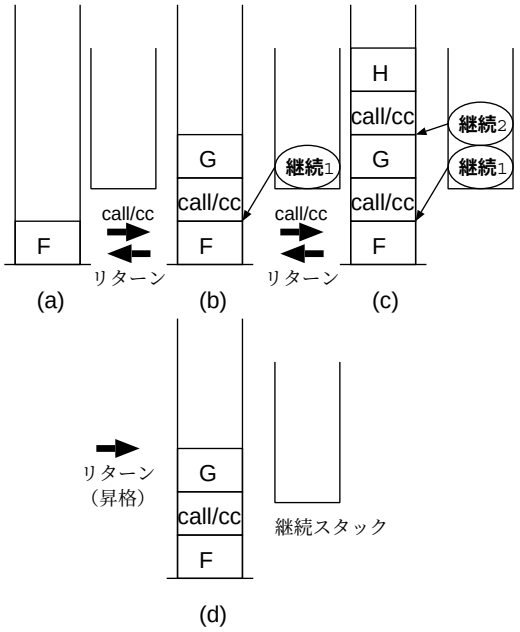


**図 8** 継続スタック
Fig. 8 Continuation stack.

続が不完全な継続に戻ることはないからである．こうすることで，まだごみになっていないすべての不完全な継続が継続スタックに積まれているという状態を保つことができる．

**図 8** に制御スタックと継続スタックの対応を示す．図 8 (a) の状態で `call/cc` を呼び出すと，継続オブジェクトを生成して継続スタックに積む（図 8 (b)）．さらに，関数 $G$ から `call/cc` を呼び出すと，さらに継続オブジェクトを生成して，継続スタックに積む．（図 8 (c)）．`call/cc` からリターンするとき，継続オブジェクトに昇格予約フラグがセットされていなければ，継続スタックの先頭から継続をポップするだけなので，図 8 (b)，図 8 (a) と戻る．

継続が昇格したときは，スタックのコピーを共有するので，継続スタックに積まれている残りの継続もすべて昇格させ，継続スタックを空にする．また，継続の呼び出しにより，呼び出されたのとは別の継続をキャプチャした関数からまだリターンしていないスタックの状態が復元されても，継続スタックは空のままである．このように，継続をキャプチャした関数からリターンするときには，継続スタックに継続が積まれていないことがある．この場合，継続スタックは必ず空になっている．継続スタックが空になっていれば，継続の昇格は終わっているので何もせずにリターンすればよい．

図 8 (c) で関数 $H$ からリターンするときに継続が昇格したとすると，図 8 (d) のようになる．この後関数 $G$ からリターンするときは，継続スタックが空になっているので，何もせずにリターンする．また，継続 2 が呼び出された後も図 8 (d) のようになっている．

継続スタックを使えば，次に昇格させるべき継続が分かるので，1 つの継続が昇格するときに，すべての継続を同時に昇格させなくても，スタックのコピーを共有することができる．昇格しようとする継続と，継続スタックでその 1 つ下にある継続（$k$ とする）だけを昇格させ，$k$ は継続スタックに戻す．$k$ をキャプチャした関数からリターンするときに $k$ を継続スタックからとり出す．$k$ はすでに昇格している．さらに，$k$ の 1 つ下の継続を昇格させ継続スタックに戻す．このとき最初の昇格で作ったスタックのコピーのアドレスは $k$ が持っているので，それを利用してスタックのコピーの共有を行うことができる．

### 4.3 末尾再帰呼び出し

Scheme 処理系は末尾再帰呼び出しの最適化を要求されている．つまり，末尾呼び出しが任意の回数連続する場合でも，スタックを定数量しか消費してはいけない．そのため，普通 Scheme 処理系は，末尾呼び出しが行われると，呼び出した関数の関数フレームを呼び出された関数の関数フレームのために再利用する．(`call/cc` ⟨*proc*⟩) では，`call/cc` が ⟨*proc*⟩ を末尾呼び出しするため，このような処理系では，`call/cc` の関数フレームは ⟨*proc*⟩ の関数フレームとして再利用される．C 言語の関数の再帰呼び出しで Scheme の関数の再帰呼び出しを表現している処理系では，普通の関数呼び出しのときは新しく Scheme の関数を実行する C 言語の関数（これを `eval` と呼ぶことにする）を呼び出すが，末尾呼び出しでは，呼び出した側の Scheme の関数を実行していた `eval` が，呼び出された側の Scheme の関数も実行する．

**図 9** のようなプログラムを実行すると，**図 10** (b) のように，$F$ から `call/cc` が呼び出される．これは末尾呼び出しではないので，通常の呼び出しになる．`call/cc` は関数 $G$ を呼び出すが，これは末尾呼び出しであるので，関数 $G$ は `call/cc` を実行していた `eval` が実行する（図 10 (c)）．次に関数 $G$ は関数 $H$ を呼び出すが，これは末尾呼び出しであるため，新しい `eval` は呼び出されず，関数 $G$ を実行していた `eval` が関数 $H$ を実行する（図 10 (d)）．

このように，実際の処理系では，`call/cc` もそれ以外の関数も同じ `eval` という関数で実行されている．提案方式では，`call/cc` からリターンするときにキャ

```
(define (G) (H))
(define (F) (call/cc G) ...)
```

**図 9** 末尾再帰呼び出しのプログラム例
Fig. 9 Example of a program using tail recursive calls.

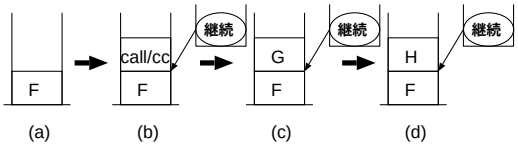


**図 10** 末尾再帰呼び出し
Fig. 10 Tail recursive call.

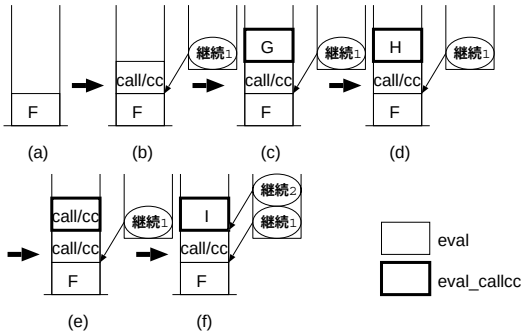


**図 11** eval_callcc を使った末尾再帰呼び出し
Fig. 11 Tail recursive call using eval_callcc.

プチャした継続を昇格させる処理が必要である．この処理は call/cc からのリターンのときだけ必要になる．しかし，call/cc もそれ以外の関数も同じ eval という関数で実行される処理系では，call/cc 以外の関数からのリターンにオーバヘッドをかけずにリターンバリアを実現するのは難しい．

そこで，リターンバリアの入った Scheme の関数を実行する C 言語の関数 eval_callcc を用意する．eval_callcc は，リターンの直前に特別な処理が入っていることと，call/cc の処理のしかた以外は eval と同じである．(call/cc ⟨*proc*⟩) が eval で評価されると，call/cc の関数フレームを再利用せずに eval_callcc を呼び出して ⟨*proc*⟩ を実行するようにする．図 9 のプログラムでは，関数 $G$ の呼び出しのときに eval_callcc が呼び出される（**図 11** (c)）．

eval_callcc は eval と違って，call/cc を実行しても，新しく eval_callcc は呼び出さず，同じ eval_callcc で call/cc の引数を実行する．図 9 の関数 $H$ が call/cc を末尾呼び出ししたとすると，図 11 (e) のようになるが，この後新しく eval_callcc を呼び出すのではなく，図 11 (f) のように同じ eval_callcc が call/cc の引数である関数 $I$ を実行する．

こうすることで，call/cc の引数と，そこから末尾呼び出しされた関数だけをリターンバリアの入った eval_callcc で評価し，ふだんはリターンバリアの入っていない eval を使って式を評価するようになる．eval_callcc からのリターンは 1 回なので，本当にバリアが必要なリターンだけにオーバヘッドがかかるようになっている．

さらに，ある関数 $F$ が関数 $G$ を末尾呼び出しするとき，$G$ の呼び出しの継続は $F$ の呼び出しの継続と同じであることに着目すると，call/cc が末尾再帰的に呼び出されたときに，最初の call/cc の呼び出し以外での継続の生成を避けることができる．つまり，eval_callcc が実行した call/cc は，継続オブジェクトを生成する代わりに，その呼び出し元で生成した継続オブジェクトを共有すればよい．この継続オブジェクトは継続スタックの先頭に積まれている．図 11 では，(f) で継続 2 を生成する代わりに継続 1 を使えばよい．しかし，eval_callcc で call/cc を呼び出したときに継続スタックが空の場合がある．このような場合は，継続オブジェクトの共有をあきらめて，新しく継続オブジェクトを生成しなければならない．

## 5. 性能評価

提案方式の評価のために，スタック法を使っている Scheme 処理系 SCM と MzScheme に提案方式を組み込んで，従来方式と実行時間を比較した．

SCM [3] も MzScheme [4),5] も C 言語で記述された処理系である．どちらの処理系も制御スタックは C 言語のスタックを使っている．

SCM は制御スタック以外に環境を保存するための環境スタックを持っている．環境スタックは一定の大きさの配列からなるリスト構造でできており，継続のキャプチャでは，先頭の配列のみコピーして，残りの配列は共有するようになっている．そこで，SCM では継続のキャプチャにおける制御スタックと環境スタックのコピーの遅延と，複数の継続オブジェクトからの制御スタックのコピーの共有を行うようにした．

MzScheme は制御スタックのほかに Scheme の局所変数を積む値スタックを持っている．値スタックも配列のリストとして実現されているが，SCM の環境スタックと違って継続をキャプチャするときに，値スタック全体をコピーしている．そこで，MzScheme では継続のキャプチャにおける制御スタックと値スタックのコピーを遅延し，また，制御スタックのコピーも値スタックのコピーも複数の継続から共有するように

```
(define (ctak x y z) (call/cc (lambda (k) (ctak-aux k x y z))))
(define (ctak-aux k x y z)
  (cond ((not (< y x)) (k z))
        (else (call/cc
               (ctak-aux k (call/cc (lambda (k) (ctak-aux k (- x 1) y z)))
                           (call/cc (lambda (k) (ctak-aux k (- y 1) z x)))
                           (call/cc (lambda (k) (ctak-aux k (- z 1) x y)))))))))
```

**図 12** ctak
Fig. 12 The ctak program.

した．

また，MzScheme は SCM と違って，S 式を実行前にコンパイルする．クロージャの生成では，コンパイル時に解析を行っており，クロージャの実行に使われない変数はクロージャの環境にとり込まれないようになっている．したがって，MzScheme では，クロージャの生成時に余分に継続の昇格予約フラグがセットされることはない．

ごみ集めの方式は，SCM も MzScheme も保守的なマークアンドスイープごみ集めを使っている．ごみ集めでかかる時間は，ごみ集めで生き残るスタックのコピーの合計の大きさが大きくなると長くなる．

性能評価のためのベンチマークプログラムには，以下のプログラムを用いた．same-fringe 以外のプログラムは Gabriel ベンチマーク[7]のものを利用した．

**tak** 本体の小さな関数の呼び出しを何回も行うため，リターンバリアのオーバヘッドが発生する．オブジェクトの生成や書き込みはしないためライトバリアのオーバヘッドはない．また，`call/cc` は使わないので，速度向上の要素はない．

**ctak** tak のリターンを継続を使った非局所的脱出に置き替えたプログラムである．リターンバリアのオーバヘッドはない．代わりに `call/cc` を使うので，速度向上の要素がある．ただし，**図 12** に示すように，ctak の再帰部分である `ctak-aux` は，`ctak-aux` の呼び出し元で生成された継続を引数で受けとっており，さらに再帰呼び出しをするときにはクロージャを生成している．したがって，コンパイル時にクロージャの環境に必要な変数しか含めない解析をしていない SCM では，多くの継続が昇格してしまう．この場合，速度向上の要素は，再帰呼び出しの最後で生成される継続のためのスタックのコピーが省けることと，スタックのコピーが共有されることになる．

**boyer** cons セルを大量に生成するため，ライトバリアによるオーバヘッドがある．`call/cc` は使わないので，速度向上の要素はない．

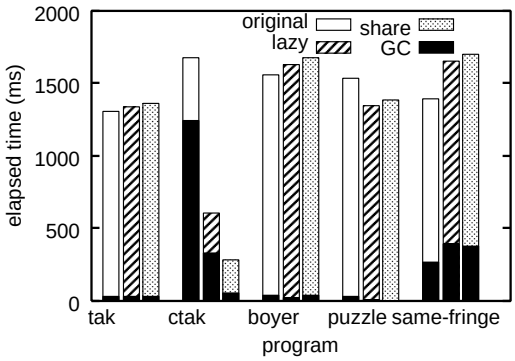


**図 13** SCM の実行時間（Pentium 4）
Fig. 13 Elapsed time on SCM (Pentium 4).

**puzzle** 全探索でパズルを解くプログラムで，オブジェクトへの書き込みを頻繁に行う．また，各手から次の手を探すループを継続による非局所的脱出により抜けるので，速度向上の要素もある．

**same-fringe** コルーチンのプログラム例である．継続を使うが，キャプチャした継続はすべて昇格する．コルーチン間の制御の移動には継続の呼び出しを使っているため，このプログラムでは制御が移るたびに継続スタックがクリアされる．そのため，スタックのコピーの共有も行われない．したがって，スタックのコピーを遅らせたり，共有するための構造を作ったりすることによる，継続の操作にかかるオーバヘッドだけがかかる．

SCM で行ったベンチマークテストの結果について，Pentium 4 での結果を**図 13** に，Ultra SPARC IIe での結果を**図 14** に示す．また，MzScheme で行ったベンチマークテストの結果を，Pentium 4 での結果を**図 15** に，Ultra SPARC IIe での結果を**図 16** に示す．図では，original が従来の処理系での実行時間，lazy が継続キャプチャ時のスタックのコピーを遅延させた処理系での実行時間，share がスタックのコピーの遅延に加えて，スタックのコピーのを共有も行う処理系

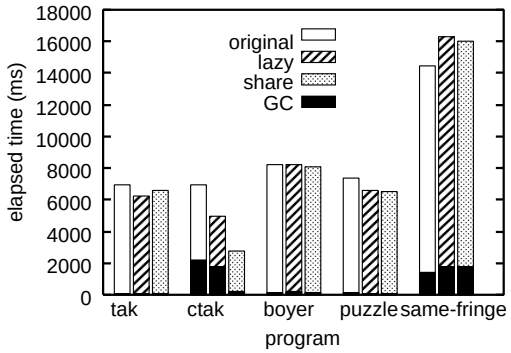


**図 14** SCM の実行時間（Ultra SPARC IIe）
Fig. 14 Elapsed time on SCM (Ultra SPARC IIe).

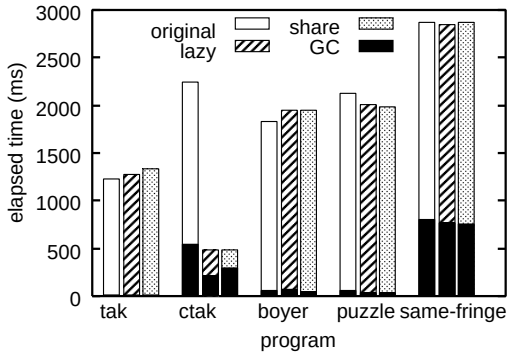


**図 15** MzScheme の実行時間（Pentium 4）
Fig. 15 Elapsed time on MzScheme (Pentium 4).

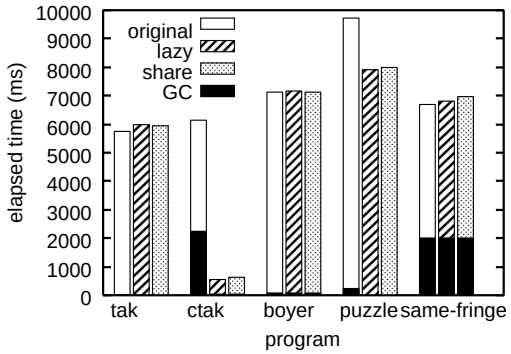


**図 16** MzScheme の実行時間（Ultra SPARC IIe）
Fig. 16 Elapsed time on MzScheme (Ultra SPARC IIe).

での実行時間である．また，実行時間の内訳のうち，ごみ集めにかかった時間を GC で表している．

全体に ctak は非常に高速化されている．特にごみ集めの時間の短縮が大きい．ctak では，スタックのコピーの遅延やスタックのコピーの共有により，スタックのコピーを持つ継続オブジェクトを作る数が大幅に減っている．これにより，ごみ集めの回数が減っていると考えられる．また，毎回のごみ集めのマークフェーズで，参照を探すためにスキャンするメモリの量も減っていると考えられる．ctak のように `call/cc` を使って末尾再帰呼び出し的に書いたプログラムは，非常に効率良く実行されると考えられる．

MzScheme では，SCM と違い ctak ではスタックのコピーを共有しても高速化されなかった．SCM では，クロージャの生成時に，生成したときの環境に含まれる継続すべてに昇格予約フラグをセットしている．そのため，本来継続が昇格する必要がまったくない ctak でも，ある程度の数の継続は昇格してしまっている．したがって，スタックのコピーの共有によりスタックのコピーが作られる量を減らすことができ，SCM ではスタックのコピーの共有により ctak が高速化されている．一方，MzScheme は S 式をコンパイルするときに環境を解析している．そのため，スタックのコピーを遅延するだけで，すべての継続のためのスタックのコピーが省けている．そのため，スタックのコピーの共有をしても，さらに高速化はされていないと考えられる．

puzzle も継続を使ったベンチマークなので，ctak と同じような傾向が見られるが，ctak と比べて `call/cc` の量もプログラム全体に占める割合も少ないので，ctak ほど顕著な高速化はなされていない．puzzle では ctak と違ってクロージャの環境に継続が含まれないので，SCM でもスタックのコピーの遅延をしただけで，スタックのコピーがまったく行われなくなっている．スタックのコピーの共有をしても，実行時間がほとんど変わらないのはそのためと考えられる．

same-fringe では，大きいところで 2 割程度の速度低下が見られる．これはスタックのコピーを遅延させることで処理が複雑になったことが考えられる．また，`eval_callcc` を導入したことで，`call/cc` の呼び出しでスタックが余分に伸び，継続をキャプチャしたときに，それだけ大きなスタックコピーを作って，継続の生成や呼び出しとごみ集めに負担がかかるようになったのも原因と考えられる．

tak は継続を使わず，また，ライトバリアの入る処理も行わないので，本来実行時間は変わらないはずである．測定結果でも誤差程度しか実行時間が変わっていない．

boyer は継続は使っていないが，コンスセルを大量に作るので，そのときに不完全な継続の参照がコンスセルに書き込まれないか調べるオーバヘッドがかかる．一部の測定結果では，やや遅くなっているが，遅くなり方は小さい．

ところで，コルーチンのプログラムで，それぞれのコルーチンが脱出のための継続を使うことがある．脱出のための継続は，スタックのコピーの遅延により昇格される前にごみになる．しかし，継続がごみになる前に別のコルーチンに制御を移すと，そのときに昇格されてしまうため，スタックのコピーの遅延の効果はない．スタックのコピーの共有を行うと，コルーチンの制御のための継続が持つスタックのコピーと，脱出のための継続の持つスタックのコピーが共有され，高速化される．実際，same-fringe の各コルーチンが行うリターンを継続を使った脱出に置きかえたプログラムで（問題の規模は小さくしてある），Pentium 4 の環境で SCM を使って実行時間を計測したところ，従来の処理系で 960 ミリ秒，スタックのコピーの遅延を行う処理系で 1150 ミリ秒，スタックのコピーの共有を行う処理系で 810 ミリ秒であった．

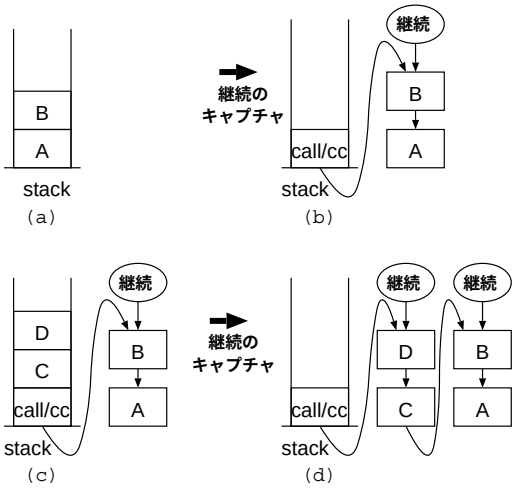


**図 17** インクリメンタルスタック/ヒープ法
Fig. 17 Incremental stack/heap strategy.

## 6. 関 連 研 究

### 6.1 非局所的脱出

`call/cc` を非局所的脱出に特化させる研究には `call/ep` がある[8]．また，SCM などの処理系では `call/cc` が `call/ep` のように振る舞うように処理系を作るコンパイルオプションが用意されている．`call/ep` は，必要になる関数フレームがスタック上に残っているときにのみ呼び出すことのできる継続を生成する．したがってスタックのコピーは発生せず，非局所的脱出の用途で効率良く働く．しかし，`call/ep` で生成した継続は必要になる関数フレームがスタック上に残っていなければ呼び出すことができないので，`call/cc` で作った継続より能力が低い．提案手法で `call/cc` が生成する継続は，呼び出されたときに必要になる関数フレームがスタック上からなくなる前にコピーを作るので，従来の継続の能力を保存している．

### 6.2 スタックのコピーの遅延

Baker の提案しているオブジェクトの領域割当ての手法[9]では，あらゆるオブジェクトを最初はスタック上に割り当てられる．この手法では，スタック上に確保したオブジェクトが大域変数や，ヒープ上のオブジェクト，スタックの底方向から参照されたときに，フォワーディングポインタを残して，オブジェクトをヒープ上に移動させる．この手法を継続オブジェクトに応用する例が示されている．この例では，関数フレームもオブジェクトとして扱う．継続オブジェクトがスタックからヒープに移動すると，スタック上の継続オブジェクトから関数フレームへの参照が，ヒープからの参照に変わるため，連鎖的に関数フレームもヒープに移動される．

提案手法もこの手法も，大域変数やヒープ上のオブジェクトに参照が作られない継続オブジェクトは昇格されない点で類似している．一方，昇格のタイミングは異なっており，この手法では大域変数やヒープ上のオブジェクトに参照が作られたときに昇格するのに対し，提案手法では昇格予約フラグをセットするだけで，昇格は継続を生成した関数からリターンするまで遅らされる．これにより，スタックのコピーが早くからヒープを圧迫するのを避けることができる．また，Baker の手法では，オブジェクトを移動させるため，フォワーディングポインタを用いる必要がある．逆に，提案手法では継続を昇格させるためにリターンバリアを用いる必要がある．しかし，実装を工夫することで，リターンバリアのオーバヘッドは非常に小さく抑えることができる．

### 6.3 スタックコピーの共有

インクリメンタルスタック/ヒープ法[2]などの戦略でもスタックのコピーの共有を行う．インクリメンタルスタック/ヒープ法では，継続をキャプチャするとき，スタック上の関数フレームをリスト構造にしてヒープにコピーする．そして，スタックは空にした後 `call/cc` の実行を始める．これにより，スタックの続きがヒープ上にある状態になる．**図 17** (a) で `call/cc` を呼び出すと，図 17 (b) のようになる．次に図 17 (c) の状態で継続をキャプチャするときは，図 17 (d) のように，スタックの底までコピーして，残りはすでにヒープにコピーされているリストにつなぐだけでよい．

このように，インクリメンタルスタック/ヒープ法

では，関数フレームのリストを作ることでスタックのコピーを共有することを可能にしている．そのため，次のプログラムのように，片方の継続が必要とするスタックのコピーが，他方が必要とするコピーに完全には含まれないような場合でも，スタックの呼び出し元のほうのコピーを共有することができる．

```
(define (f) (call/cc (lambda (k1) ...)
                     (call/cc (lambda (k2) ...))
```

しかし，C 言語のスタックのように，構造が完全に把握できないような場合には利用することができない．提案手法では，スタック法をもとにしているため，スタックがどのメモリアドレスにあるかが分かれば利用することができる．しかし，片方の継続が必要とするスタックのコピーが，他方が必要とするスタックのコピーに完全には含まれないような場合には，共有することができない．

## 7. 今後の課題

本発表では，スタックのコピーを遅延させることと，コピーしたスタックを共有することで，スタック法による継続の実装を改良する手法を示した．

提案手法では，ライトバリアが必要であるので，継続を使わないプログラムにもオーバヘッドがかかる．プログラムをあらかじめ解析することで，このオーバヘッドはある程度解消することができる．不完全な継続がバインドされることがない変数（これを「安全な変数」と呼ぶことにする）を実行前に見つけておけば，その変数を使ったオブジェクトの生成や，大域変数やオブジェクトのスロットへの代入はバリアなしで実行できる．そこで，オブジェクトの生成や，大域変数やオブジェクトのスロットへの代入をともなう Scheme のシステム関数やコンストラクトを，バリアの入ったバージョンと入っていないバージョンの両方用意する．安全な変数を使った操作はバリアの入っていないバージョンを使うようにすれば，ライトバリアによるオーバヘッドを小さくすることができる．

また，提案手法では，ヒープ上のオブジェクトから不完全な継続への参照が作られると，昇格予約フラグをセットしてしまっている．しかし，局所的に利用されるオブジェクトで，そのオブジェクトを生成した関数からリターンするとごみになるようなオブジェクトはよく使われる．このようなオブジェクトからの参照が作られたときには昇格予約フラグをセットしないようにする仕組みがあれば，昇格する継続の数を減らすことができる場合がある．局所的に利用されるオブジェクトを探すには，一般のオブジェクトにも昇格予約フラグのような働きをするフラグを設けて，このフラグがセットされていないオブジェクトからの参照では昇格予約フラグをセットしないようにするという方法が考えられる．これは文献 9) で提案されている stack allocation の一時的なオブジェクトの管理と，本質的に同じことをする．

今後，本手法にこのような改良を加えていく予定である．

**謝辞** 本研究の一部は，研究拠点形成補助金（課題番号：14213201）の支援を受けた．

## 参 考 文 献

1) Kelsey, R., Clinger, W. and Rees, J. (Eds.): Revised$^5$ Report on the Algorithmic Language Scheme, *Higher-Order and Symbolic Computation*, Vol.11, No.1, pp.7–105, Kluwer Academic Publishers (1998).
2) Clinger, W.D.: Implementation Strategies for First-Class Continuations, *Higher-Order and Symbolic Computation*, Vol.12, No.1, pp.7–45, Kluwer Academic Publishers (1999).
3) Jaffer, A.: SCM. http://www-swiss.ai.mit.edu/~jaffer/scm_toc.html
4) Rice University, University of Utah: *PLT MzScheme: Language Manual* (2000).
5) Rice University, University of Utah: *Inside PLT MzScheme* (2000).
6) 湯淺太一，中川雄一郎，小宮常康，八杉昌宏：リターン・バリア，情報処理学会論文誌：プログラミング，Vol.41, No.SIG 9（PRO 8），pp.87–99 (2000).
7) Gabriel, R.: *Performance and Evaluation of Lisp Systems*, MIT Press (1985).
8) 清野智弘，伊藤貴康：PaiLisp の並列構文の実現法と評価，情報処理学会論文誌，Vol.34, No.12, pp.2578–2591 (1993).
9) Baker, H.G.: CONS Should not CONS its Arguments, or, a Lazy Alloc is a Smart Alloc, *ACM SIGPLAN Notices*, Vol.27, No.3 (1992).

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

**鵜川 始陽**

1978 年生．2000 年京都大学工学部情報学科卒業．2002 年同大学大学院情報学研究科修士課程修了．同年より同大学院情報学研究科博士後期課程に在籍．言語処理系に興味を持つ．

**皆川 宣久**

2003 年京都大学工学部情報学科卒業．同大学大学院情報学研究科通信情報システム専攻修士課程在籍．

**小宮 常康**（正会員）

1969 年生．1991 年豊橋技術科学大学工学部情報工学課程卒業．1993 年同大学大学院工学研究科情報工学専攻修士課程修了．1996 年同大学院工学研究科システム情報工学専攻博士課程修了．同年京都大学大学院工学研究科情報工学専攻助手．1998 年より同大学院情報学研究科通信情報システム専攻助手．博士（工学）．記号処理言語と並列プログラミング言語に興味を持つ．1996 年度情報処理学会論文賞受賞．

**八杉 昌宏**（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993 年〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年京都大学大学院情報学研究科通信情報システム専攻講師．2003 年より同大学助教授．博士（理学）．1998 年〜2001 年科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

**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，並列処理に興味を持っている．著書「Common Lisp 入門」（共著),「C 言語によるプログラミング入門」,「コンパイラ」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．