無制限の寿命を持つ単一呼出継続

小宮常康† 湯淺太一†

継続とはある時点以降の残りの計算を表したものである．Lispの一方言であるSchemeでは，継続を生成する関数が用意されており，継続をデータとして扱うことができる．これによって，非局所的脱出やコルーチン，マルチタスクなど様々な制御構造を実現することができる．スタックを用いた標準的なScheme処理系では，スタックの内容をヒープに退避することによって継続を生成し，呼び出し時にはそれをスタックに復元する．そのため，継続の生成と呼び出しの処理は重い．そこで本論文では，ほとんどの継続の使用法において，その呼び出しは一度に限られることに注目し，呼び出しは一度に限るが，いつどこからでも呼び出すことができる継続を提案する．この継続を用いることにより，継続を用いて実現されるほとんどのプログラムに対して，性能を改善することができる．

Indefinite One-time Continuation

TSUNEYASU KOMIYA† and TAIICHI YUASA†

A continuation represents the rest of computation from a given point. Scheme, a dialect of Lisp, provides a function to generate a continuation as a first-class object. Using Scheme continuations, we can express various control structures such as non-local exits, coroutines and multitasks. In stack-based Scheme implementations, continuations are implemented by saving the contents of the control stack into the heap when a continuation is captured and by restoring the control stack from the heap when the continuation is invoked. Therefore, the operation of capturing and invoking continuations is heavy. In this paper, based on the fact that each continuation is invoked only once in most applications, we propose indefinite one-time continuations which can be invoked only once but which can be invoked at any time. Using these continuations, we can improve performance of application programs of Scheme continuations.

1. はじめに

継続とはある時点以降の残りの計算を表したものである．Lispの一方言であるSchemeでは，継続をデータとして扱うことができる[1)~4)]．これによって，非局所的脱出やコルーチン[5)]，マルチタスク[6)]など様々な制御構造を実現することができる．

継続の生成は関数 `call-with-current-continuation`（以後，省略形の `call/cc` を用いる）によって

`(call/cc` 〈関数〉`)`

という形で行う．この式を評価すると，この式を評価した後の継続を生成し，その継続を引数として1引数の関数〈関数〉を呼び出す．そして〈関数〉からの返り値が `call/cc` 式の値となる．Schemeにおける継続は1引数の関数として実現されている．これを呼び出すと，与えられた引数の値を，その継続を生成した `call/cc` 式の値として `call/cc` 式以降の評価を続ける．〈関数〉の実行中に継続を呼び出さなかった場合は，〈関数〉からの返り値が `call/cc` 式の値となるが，これは〈関数〉の値を引数として継続を呼び出していると考えることができる．これを暗黙の継続呼び出しと呼ぶことにする．

`call/cc` による継続は何度でも呼び出すことができる．これを実現するために，スタックを用いた標準的なScheme処理系では，スタックの内容をヒープに退避することによって継続を生成し，呼び出し時にはそれをスタックに復元する．そのため，継続の生成と呼び出しの処理は重い．しかし，非局所的脱出に継続を使うときは，継続を生成した時点でのスタックポインタの値を記憶しておき，呼び出し時にスタックポインタを戻せばよい．このような非局所的脱出のための継続を生成する関数として，PaiLisp[7)]では，`call-with-escape-procedure`（以後，省略形の `call/ep` を用いる）を用意している．`call/ep` の基本的な使い方は `call/cc` と同じであるが，継続の呼び出

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

しは，その継続を生成した call/ep 式の引数である〈関数〉の実行中に制限される．したがって，call/ep の機能は本質的に Common Lisp[8] の catch と throw と同じである．call/ep による継続は，call/ep 式の評価が終了するとスタック上の継続情報が失われるので，呼び出すことができなくなる．一方，コルーチンやマルチタスクなどの非局所的脱出以外の継続の応用例では，スタックに継続情報がない状態で継続が呼び出される．したがって，これらを call/ep によって実現することはできない．

そこで本論文では，call/ep より使用法の制限を緩くした継続を提案する．この継続は，ほとんどの継続の使用法において，その呼び出しは一度に限られることに注目し，呼び出しは一度に限られるが，いつどこからでも呼び出すことができる．そのため，コルーチンなどの call/cc を使ったほとんどのプログラムに対して，提案する継続を用いることができ，性能を改善することができる．また，非局所的脱出にこの継続を用いると，多くの場合，call/ep と同程度の性能が得られる．

以下では，2 章で非局所的脱出以外の継続の応用例を示し，提案する継続について述べる．そして 3 章でその意味論を示し，4 章で実現方法について述べる．最後に，5 章で性能評価について述べる．

2. Indefinite one-time continuation

継続によって実現される制御構造のほとんどは，生成された継続を暗黙の継続呼び出しを含めてそれぞれ一度しか呼び出さない．例えば，継続で実現される制御構造のひとつにコルーチンがある（マルチタスクもコルーチンの一種である）．コルーチンを実現するには，コルーチンの実行中断時に継続を生成し，実行を再開するときにはその継続を呼び出せばよい．以下に，call/cc によるコルーチンの例を示す．

```
(define (f k)
  (do () (#f)
    (produce)
    (set! k (call/cc k))))
(define (g k)
  (do () (#f)
    (consume)
    (set! k (call/cc k))))
```

この例は，コルーチン f が生成した何らかのデータをコルーチン g が処理するものである．f はデータを生成すると制御を g に渡し，g はそのデータに対して何らかの処理を行う．処理が終わると f に制御を戻し，

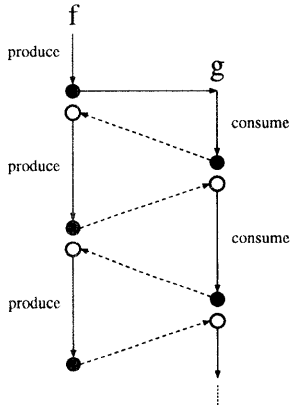

図 1 コルーチン
Fig. 1 Coroutine.

f は再びデータを生成するということを繰り返す．このプログラムを (f g) として起動した場合の動作は図 1 のようになる．図において実線は制御の流れ，破線は継続呼び出しによる制御の移行，黒丸は継続を生成する点，白丸は継続を生成した call/cc 式の直後の点を表す．図からわかるように，コルーチンは実行を中断するたびに継続を生成し，実行再開時にそれらを一度だけ呼び出す．非局所的脱出以外の継続の応用例のほとんどは，この例のように個々の継続を一度しか呼び出さない．

また，非局所的脱出以外の応用例においては，スタックに継続情報がない状態で継続が呼び出される．例えば図 1 では，g から継続呼び出しによって f の実行を再開した時点で，g の継続情報は失われる．したがって，f から継続呼び出しによって g の実行を再開するときには，スタック上に継続情報は存在しない．そのため，非局所的脱出以外の応用に対しては call/ep は使用できない．

そこで本論文では，暗黙の継続呼び出しを含めた呼び出しは一度に限るが，いつどこからでも呼び出すことができる継続を提案する．call/ep による継続は，one-time continuation と呼ばれることがあるが，提案する継続はいつどこからでも呼び出すことができるので，区別するために indefinite one-time continuation（以後 IOC と略す）と名付ける．IOC の生成は関数 call-with-indefinite-one-time-continuation（以後，省略形の call/ioc を用いる）によって

(call/ioc 〈関数〉)

という形で行う．すると，call/cc の場合と同様に現在の IOC を生成し，それを引数として〈関数〉を呼び出す．そして〈関数〉からの返り値が call/ioc 式の値となる．

実際上，継続の応用のほとんどは IOC によって置き換えることができるが，IOC では対応できない応用も存在する．そこで，従来の `call/cc` を残し，`call/ioc` を新たに追加することを提案する．これによって，`call/cc` を使ったプログラムを，少しずつ `call/ioc` に置き換えて効率を向上させるといったことが可能となる．

IOC は `call/ep` による継続と同様，その呼び出しは一度に制限される．複数回 IOC が呼び出された場合はエラーとなる．ただし，`call/cc` による継続が呼び出された場合は，IOC が呼び出されたかどうかの状態は `call/cc` の実行時の状態に戻るものとし，複数回 IOC を呼び出すことができるものとする．IOC 呼び出しの影響を `call/cc` の継続に与えないのは，これら 2 種類の継続を併用する場合，それらは異なった役割で用いられることが多く，ユーザはそれぞれの動作を分離して考えることができる方が望ましいからである．

3. 意 味 論

本章では文献 3) の記法に従って IOC の表示的意味論を示す．本論文で用いる記法，抽象構文，意味領域は次のとおりである．

記法

$\langle \ldots \rangle$	列
$s \downarrow k$	列 s の k 番目の要素
$\#s$	列 s の長さ
$s \S t$	列 s と列 t の連結
$s \dagger k$	列 s の先頭の k 個の要素を除いた列
$t \rightarrow a, b$	McCarthy の条件式 "if t then a else b"
$\rho[x/i]$	ρ の中の i を x で置き換える
x in D	領域 D への x の injection
$x \mid$ D	領域 D への x の projection

抽象構文

K ∈ Con	constants, including quotations
I ∈ Ide	identifiers (variables)
E ∈ Exp	expressions
Γ ∈ Com = Exp	commands

```
Exp ⟶ K | I | (E0 E*)
      | (lambda (I*) Γ* E0)
      | (if E0 E1 E2)
      | (set! I E)
```

意味領域

$\alpha \in \mathtt{L}$	locations
$\mathtt{T} = \{false, true\}$	booleans
$\phi \in \mathtt{F} = \mathtt{L} \times (\mathtt{E}^* \rightarrow \mathtt{G} \rightarrow \mathtt{C})$	procedure values
$\epsilon \in \mathtt{E}$	expressed values
$\sigma \in \mathtt{S} = \mathtt{L} \rightarrow (\mathtt{E} \times \mathtt{T})$	stores
$\rho \in \mathtt{U} = \mathtt{Ide} \rightarrow \mathtt{L}$	environments
$\theta \in \mathtt{C} = \mathtt{S} \rightarrow \mathtt{A}$	command continuations
$\kappa \in \mathtt{K} = \mathtt{E}^* \rightarrow \mathtt{C}$	expression continuations
A	answers
X	errors

IOC は暗黙の継続呼び出しを含めて一度しか呼び出すことができないので，IOC が呼び出されたかどうかを意味領域上で表現できなければならない．したがって，`call/cc` の継続のように expression continuaion によって IOC の意味を表現することはできない．そこで，部分継続[9)~13)]（partial continuation）の表示的意味論を表現するために導入されたコンテクスト[13)]（context）の概念を用いて IOC の意味を表すことにする．

コンテクストは部分コンテクスト（partial context）を連結したものある．それらの領域を次のように表す．

$\pi \in \mathtt{P} = (\mathtt{E}^* \rightarrow \mathtt{G} \rightarrow \mathtt{C}) \times \mathtt{L}$ partial contexts

$\gamma \in \mathtt{G} = \mathtt{P} \times \mathtt{G}$ contexts

部分コンテクストは言語処理系における制御スタック上のフレームに相当する．部分コンテクストの location は，その部分コンテクストがすでに呼び出されたかどうかを表すためのもので，まだ呼び出されていなければその値は *true* であり，すでに呼び出されたものであれば値は *false* である．部分コンテクストと真偽値の対応はストア（store）σ で表す．通常，ストアは継続の一部として保存されないが，部分コンテクストと真偽値の対応は保存する必要がある．そこでストアを

$\sigma \in \mathtt{S} = (\mathtt{L} \rightarrow (\mathtt{E} \times \mathtt{T})) \times (\mathtt{L} \rightarrow (\mathtt{E} \times \mathtt{T}))$ stores

と二つの関数の組として定義し直し，ストアを更新する関数 *update1* と *update2* を

$update1 : \mathtt{L} \rightarrow \mathtt{E} \rightarrow \mathtt{S} \rightarrow \mathtt{S}$

$update1 = \lambda\alpha\epsilon\sigma\,.\,(\sigma \downarrow 1)[(\epsilon, true)/\alpha]$

$update2 : \mathtt{L} \rightarrow \mathtt{E} \rightarrow \mathtt{S} \rightarrow \mathtt{S}$

$update2 = \lambda\alpha\epsilon\sigma\,.\,(\sigma \downarrow 2)[(\epsilon, true)/\alpha]$

と定義する．*update1* は記憶場所（location）と値の対応を更新する関数であり，*update2* は部分コンテクストと真偽値の対応を更新する関数である．

コンテクストを操作する関数には次のものがある．

```
addframe : G → P → G
addframe = λγπ . ⟨π⟩ § γ
send : E → G → C
send =
  λϵγ . λσ . (σ ↓ 2)((γ ↓ 1) ↓ 2) = true →
    ((γ ↓ 1) ↓ 1) ⟨ϵ⟩ (γ † 1)
                (update2 ((γ ↓ 1) ↓ 2) false σ),
    wrong "invalid invocation"
```

addframe は部分コンテクスト π をコンテクスト γ の先頭に連結する．*send* は，コンテクスト γ の先頭の部分コンテクストがまだ呼び出されていないかどうかを調べ，もしそうであれば値 ϵ と残りのコンテクストを引数としてその部分コンテクストを呼び出す．その際，呼び出す部分コンテクストのストア σ を *update2* によって更新する．コンテクストに基づく Scheme のプリミティブな式の意味論を付録に示す．ただし付録には，文献 3) で記述されているものと異なるものだけを載せた．

コンテクストを用いて call/ioc の意味を表すと次のようになる．

```
cwioc : E* → G → C
cwioc =
  onearg(λϵγ . ϵ ∈ F →
          (λσ . new σ ∈ L →
            applicate ϵ
              ⟨⟨new σ | L, λϵ*γ' . send ϵ*γ⟩in E⟩
              γ
              (update1 (new σ | L)
                       unspecified
                       σ),
            wrong "out of memory" σ),
          wrong "bad procedure argument")
```

2 章で述べた call/ioc と共存することのできる call/cc の意味は以下のようになる．

```
cwcc : E* → G → C
cwcc =
  onearg(λϵγ . ϵ ∈ F →
          (λσ . new σ ∈ L →
            applicate ϵ
              ⟨⟨new σ | L,
                λϵ*γ' . λσ' .
                  send ϵ*γ((σ' ↓ 1)§(σ ↓ 2)))⟩ in E⟩
              γ
              (update1 (new σ | L)
                       unspecified
                       σ),
            wrong "out of memory" σ),
          wrong "bad procedure argument")
```

call/cc の call/ioc との意味の違いは，call/cc による継続では，継続生成時の部分コンテクストと真偽値の対応を表す関数 $(\sigma \downarrow 2)$ を保存することである．

4. 実　　現

この章では call/ioc の実現について述べる．まず準備段階として call/ep の実現方法の概要について述べ，それから call/ioc の実現方法について述べる．

4.1 call/ep の実現方法

call/ep の実現[7]は，継続の生成時に現在のスタックポインタの値を記憶しておき，呼び出し時にスタックポインタを生成時の値に戻せばよい．しかし，この方法では継続生成時のスタックの内容がスタック上にまだ残っているかどうかの判定が難しい．そこで継続の生成時に，スタックポインタの値を記憶する代わりにスタックポインタの位置を表すマーカをスタックに積み，そのマーカを記憶する．そして呼び出し時には，そのマーカがスタック上に残っているかどうかでエラー判定を行い，もし残っていればそのマーカの位置にスタックポインタを戻す．

4.2 call/ioc の実現方法

生成時にマーカをスタックに積み，呼び出し時にマーカを探し，もし見つかればその位置にスタックポインタを戻すという点では，call/ioc の実現は call/ep と同じである．異なるのは，IOC ではいつどこからでも呼び出せるようにしなければならないことである．call/ep による継続の呼び出しが call/ep の引数である〈関数〉の実行中に限られるのは，継続の呼び出しによって〈関数〉の実行を終了すると，スタックが巻き戻され，スタックに蓄えられていた継続情報が失われるからである．

このため，call/ioc では継続情報が失われないように，スタックを巻き戻すときには継続情報を退避しなければならない．退避する場所には，ヒープを用いることもできるが，call/cc の実現と同様，IOC の呼び出し時にヒープからスタックへの復元の操作が必要となる．そこで，スタックを複数用いて call/ioc を実現する．複数あるスタックのうち，現在，制御スタックとして使用しているものを特にカレント・スタックと呼ぶ．カレント・スタック以外のスタックは，スタックを巻き戻すときに，継続情報の退避場所として用いる．すなわち，各スタックはコルーチンまたはタ

スクの実行コンテクストの役割を果たす．この方法では，退避した継続情報の復元はカレント・スタックを切り替えるだけで済む．これは，IOCがたかだか一度しか呼び出されないからである．call/ccによる継続の場合は，何度でも呼び出すことができるため，退避した継続情報はそれを参照する継続が存在する間，保存しなければならず，もし上記の方法でcall/ccを実現すると，呼び出しのたびにスタックの確保とフレームのコピーがともなってしまう．

スタックは必要に応じて，ヒープなどから割り当てる．しかし，IOCオブジェクトがどこからも参照されなくなり，ごみになると，対応するスタックもごみとなる．そこで，ごみとなったスタックを再利用できるようにスタック用のフリーリストを用意する．

継続情報を退避するタイミングには，IOCの生成時と呼び出し時の2通りが考えられる．生成時に退避する方法は，生成時に確保した新しいスタックをカレント・スタックとし，それまでのスタックはIOCの呼び出しに備えて保存する．しかし，この方法はIOC生成のたびにスタックを確保するのでメモリ効率の点で問題がある．また，スタックをまたがったリターンの処理が繁雑になる．

一方，IOC呼び出し時に退避する方法では，IOCの生成時にはcall/epのときと同様に継続情報をカレント・スタックに積んでおく．そしてIOC呼び出しの処理によってスタックを巻き戻すときに，スタック上の継続情報を新たに確保したスタックに退避する．この方法では，スタックの数をコルーチン（またはタスク）の数に抑えることができ，コルーチン間の制御の切り替えをスタックの切り替えで行うことができる．この方法によってIOCを呼び出すときの動作を図**2**と図**3**に示す．図においてスタック上のa～dは個々のフレームを表し，グレーのフレームはマーカが積まれていることを意味する．

図2はフレームdにおいてフレームcに対応するIOCを呼び出すときの図である．この場合，フレームcの上位方向（図ではフレームcの下位方向）には，マーカの積まれたフレームは存在しない．つまり，保存する必要のある継続情報は残っていない．したがって，この場合のIOC呼び出しはスタックポインタをフレームcの位置に戻すだけでよい．

図3は，フレームdにおいてフレームaに対応するIOCを呼び出すときの動作である．この場合は，フレームaの上位方向にマーカの積まれたフレームcが残っているので，スタック用のフリーリストから新たにスタックを確保し，フレームbとフレームcをそ

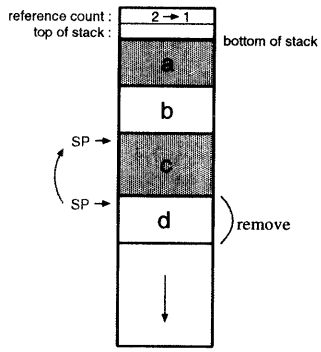

図2 IOCの呼び出し(1)
Fig. 2 Invocation of IOC(1).

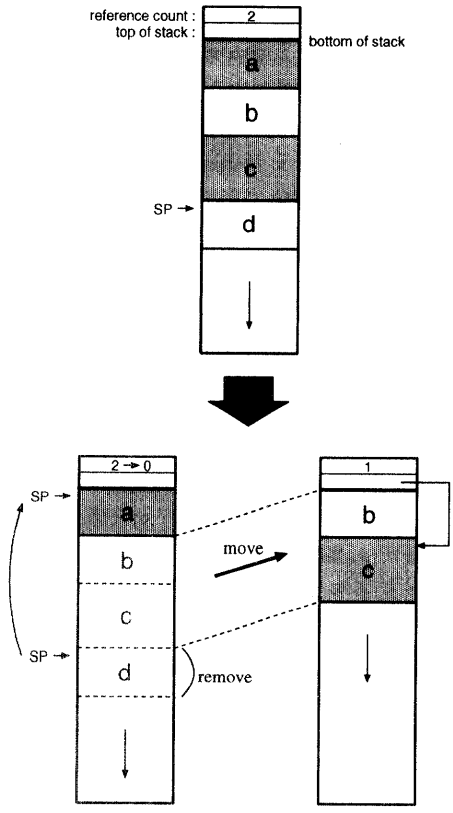

図3 IOCの呼び出し(2)
Fig. 3 Invocation of IOC(2).

こへコピーする．フレームdは，もはや参照されることはないのでコピーしない．また，フレームbからフレームaにはもはや戻ることは許されないので新たに確保したスタックの底にはエラー処理ルーチンへのアドレスをリターンアドレスとして積んでおく．最後に，スタックポインタをカレント・スタックのフレームaの位置に戻す．

各スタックにはリファレンス・カウントと，スタッ

ク上の最も上に置かれたフレームの位置を記憶しておくためのトップ・フレームスロットを用意する．リファレンス・カウントスロットはスタック上のマーカの積まれたフレーム数を記憶し，ごみとなったスタックを回収するときの情報として用いる（カレント・スタック以外のスタックがマーカの積まれたフレームを一つも持たないときは，そのスタックはごみである）．トップ・フレームスロットは，スタックのどこまでフレームが積まれているかを覚えておくためのものであり，マーカの探索やごみ集め（GC）時のスタックのスキャンはこの位置から行われる．図2では，継続の呼び出しによってリファレンス・カウントは2から1になる．一方，トップ・フレームスロットは，図2の場合，フレーム位置はスタックポインタが記憶しているので使用しない．

この方法によるIOCの呼び出しでは，カレント・スタック以外のスタックに置かれるIOCを呼び出すことがある．そこで，IOCオブジェクトに記憶されるマーカに，IOC生成時のスタックを記憶しておき，そのスタックの中からマーカの位置を探す．この探索は，スタックのトップ・フレームスロットの指す位置からフレームリンクをたどって行う．もし，マーカが見つかればカレント・スタックのトップ・フレームスロットにカレント・スタックのトップ・フレームの位置を保存しておく．ただし，カレント・スタックのリファレンス・カウントが0であれば，このスタックを直ちに回収し，スタック用のフリーリストにつなぐ．そして，マーカの見つかったスタックをカレント・スタックとし，マーカの位置にスタックポインタを戻す．なお，図3のように，マーカの積まれたフレームを別のスタックへコピーするときには，マーカが記憶するスタックをコピー先のスタックに更新する必要がある．

`call/cc` と `call/ioc` を共存させるためには，`call/cc` で継続を生成するときにスタック上のフレームだけでなくスタックのリファレンス・カウントも保存するようにする．そして呼び出し時には，ヒープに退避されたフレームのほかにリファレンスカウントも復元すればよい．

4.3 末尾再帰呼び出しの最適化

Schemeでは，末尾再帰呼び出しは繰り返しとして実現される．この最適化（tail-recursion optimization）は，末尾再帰呼び出しの際に，新しいフレームをスタックに積まずに，スタック上の現在のフレームを上書きすることで実現される．しかし，3章で述べたようにフレームは部分コンテクストに相当し，継続情報に加えて，呼び出されたかどうかのフラグが存在する．そのため，末尾再帰呼び出しによってフレームが上書きされると，フラグまで書き換えられてしまうことになる．このため，`call/ioc` が末尾再帰的に繰り返し呼び出された場合，複数のIOCが生成されるが，フレームの上書きによって実際には一つのIOCしか呼び出すことができない．しかし，実際には一つのIOCだけを呼び出すことができれば十分である．以下ではその理由を示す．

図4は，`call/ioc` が末尾再帰的に繰り返し呼び出された場合のスタックの様子である．グレーのフレームが `call/ioc` によってマーカの積まれたフレームであり，繰り返し呼び出されたためにそれらは連続している．なお，このスタックの図は実際に実現されるスタックの様子ではなく，末尾再帰呼び出しの最適化をしない場合のスタック（本質的に3章で述べたコンテクストと同じものである）の様子である．いま，フレームeにおいてIOCを呼び出し，フレームdに制御が移行したとする．フレームb～フレームdは末尾再帰的に呼び出されているために，各フレームは単純にリターンするだけである．したがって，フレームdに制御が移行した後は，フレームbとフレームcは暗黙のIOC呼び出し以外に呼び出されることはない．

次に**図5**のようにフレームeからIOC呼び出しによってフレームbに移行し，フレームaにリターンした後，IOC呼び出しによってフレームdに移行した場合を考える．この場合，まずフレームcへ，次にフ

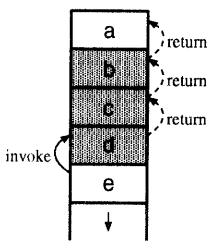

図4 `call/ioc` の末尾再帰呼び出し(1)
Fig. 4 Tail-recursive calls of `call/ioc`(1).

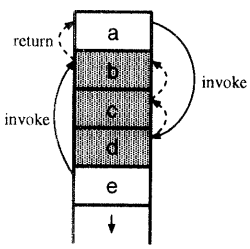

図5 `call/ioc` の末尾再帰呼び出し(2)
Fig. 5 Tail-recursive calls of `call/ioc`(2).

表1 ベンチマークの実行時間
Table 1 Execution time of benchmarks.

ベンチマーク名	実行時間 [sec]				GC に要した時間 [sec]		速度向上比
	call/cc	call/cc′	call/ioc	call/ep	call/cc	call/ioc	
ctak	4.26	4.31	3.26	3.15	1.67	1.08	1.31
coroutine	9.17	9.56	4.32	N/A	3.38	0.58	2.12
same fringe	483.90	494.82	64.45	N/A	276.07	34.90	7.51
mfib	13.10	13.9	7.54	N/A	7.80	2.30	1.74

レームbへと暗黙のIOC呼び出しによってリターンする．しかし，フレームbはすでに呼び出されているので，ここでエラーとなる．つまり，フレームdへのIOC呼び出しは意味のない継続呼び出しである．

以上のことから，call/iocが末尾再帰的に繰り返し呼び出されて生成されたIOCは，結局それらのうちの一つだけしか呼び出せないことがわかる．したがって，末尾再帰呼び出しの最適化を行うことができる．その際，マーカの積まれたフレームを複数のIOCオブジェクトから共有すればよい．

5. 性能評価

本論文で述べた call/ioc は Scheme 処理系 TUTScheme[14]に実現されている．この章では，ベンチマーク・プログラムを用いて TUTScheme 上の call/ioc の性能評価を行う．用意したベンチマーク・プログラムは次の4つである．

ctak tak 関数の継続版[15]

coroutine 2つのコルーチン間で制御の移行だけを行う[1]

same fringe 2つの木の縁（leaf）をたどり，現れる節の順序が同じかどうかを調べる[11]

mfib マルチタスク・シミュレータ上の future[16]を用いてフィボナッチ数列の n 番目を求める[11]

それぞれのベンチマーク・プログラムについて従来の call/cc，call/ioc と共存させるために call/cc に変更を施した call/cc′，call/ioc，call/ep の4つを用いて SPARCstation 10 model 30 上で実行したときの実行時間と，実行時間のうち GC に要した時間を**表1**に示す．それぞれの継続で実行させるための各プログラムの変更は，call/cc を call/ioc または call/ep に置き換えるだけである．表中の速度向上比は call/cc による実行時間に対する call/ioc による実行時間の向上比である．TUTScheme の call/cc の実現方法は，インクリメンタル・スタック/ヒープ法[15]である．表中 call/ep の ctak 以外の結果が得られていないのは，ctak 以外のベンチマーク・プログラムに対しては call/ep を使用できないからである．なお，TUTScheme における call/ioc の実現では，通常のヒープとは別のメモリ領域にスタックを割り当てている．そこで，処理系へのメモリ割り当て量を公平にするため，call/cc によるプログラムを実行するときには，call/ioc によるプログラムで新たに割り当てられるスタックの分だけあらかじめヒープを拡大しておいた．

call/ioc を用いた場合，すべてのベンチマークにおいて call/cc を用いたときより実行時間が短い．特に GC に要する時間が大幅に短縮されていることがわかる．これは，同じスタックを使い続け，継続情報はスタックに残っているからである．

call/ioc による ctak は call/ep のときとほぼ同じ性能が得られている．これは，call/ioc による ctak では，最後に生成された IOC が必ず最初に呼び出されるため，スタックのコピーがまったく起こらず，call/ep と同じ動作をしているためである．

また，call/cc に対する call/cc′ のオーバヘッドはいずれも数%程度であり小さい．

6. おわりに

本論文では，呼び出しは一度に限るが，いつどこからでも呼び出すことができる継続 indefinite one-time continuation を提案した．この継続は，従来の継続を用いて実現されるほとんどのプログラムに対して適用でき，プログラムの性能を改善することができる．

また，この継続は部分継続[9)~13)]を実現するのにも有効である．部分継続とはある時点以降の残りの計算の一部を表したものである．継続とは異なり，呼び出された部分継続の実行が終了すると，部分継続の呼び出し元に制御が戻る．したがって，部分継続は通常の関数のような振舞いをする．部分継続は call/cc を用いて実現することができるが[11]，call/cc による継続は残りの計算すべてを含むために無駄が多い．そこで，呼び出しを一度に限った部分継続であれば，call/ioc を用いて効率よく実現することができる．実際，部分継続の応用例のほとんどは，部分継続を一度しか呼び出さない．

参考文献

1) 湯淺太一：Scheme 入門，岩波書店 (1991).
2) IEEE Standard for the Scheme Programming Language, IEEE (1991).
3) Clinger, W. and Rees, J. (Eds.): Revised^4 Report on the Algorithmic Language Scheme, *MIT AI Memo 848b*, MIT (1991).
4) 小宮常康，湯淺太一：Future ベースの並列 Scheme における継続の拡張，情報処理学会論文誌，Vol.35, No.11, pp.2382–2391 (1994).
5) Haynes, C.T., Friedman, D.P. and Wand, M.: Continuations and Coroutines, *Proc. 1984 ACM Conference on Lisp and Functional Programming*, pp.293–298 (1984).
6) Wand, M.: Continuation-based Multiprocessing, *Conference Record of the 1980 Lisp Conference*, ACM, pp.19–28 (1980).
7) 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).
8) Steele, G.L., et al.: *Common Lisp: The Language*, Digital Press (1984).
9) Felleisen, M., Friedman, D.P., Duba, B. and Merrill, J.: Beyond Continuations, Technical Report No.216, Indiana University (1987).
10) Johnson, G.F.: GL – A Denotational Testbed with Continuations and Partial Continuations, *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pp.165–176 (1987).
11) Queinnec, C. and Serpette, B.: A Dynamic Extent Control Operator for Partial Continuations, *Proc. Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp.174–184 (1991).
12) Moreau, L. and Queinnec, C.: Partial Continuations as the Difference of Continuations – A Duumvirate of Control Operators, *Lecture Notes in Computer Science*, Vol.844, Springer-Verlag, pp.182–197 (1994).
13) Felleisen, M., Wand, M., Friedman, D.P. and Duba, B.F.: Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps, *Proc. 1988 ACM Conference on Lisp and Functional Programming*, pp.52–62 (1988).
14) 湯淺太一他：TUTScheme のマニュアル，豊橋技術科学大学 湯淺研究室 (1994).
15) Clinger, W., Hartheimer, A.H. and Ost, E.M.: Implementation Strategies for Continuations, *Proc. 1988 ACM Conference on Lisp and Functional Programming*, pp.124–131 (1988).
16) Halstead, R.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol.7, No.4, pp.501–538 (1985).

付録 コンテクストに基づく Scheme の意味論

A.1 意味関数

$$
\begin{aligned}
&\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] = \\
&\quad \lambda\rho\gamma\,.\,\lambda\sigma\,. \\
&\qquad new\,\sigma \in \mathrm{L} \rightarrow \\
&\qquad\quad \mathcal{E}^*(permute(\langle \mathrm{E}_0 \rangle \S \mathrm{E}^*)) \\
&\qquad\qquad \rho \\
&\qquad\qquad (addframe\,\gamma \\
&\qquad\qquad\quad \langle \lambda\epsilon^*\gamma\,.\,((\lambda\epsilon^*\,.\,applicate\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\gamma) \\
&\qquad\qquad\qquad\qquad (unpermute\,\epsilon^*)), \\
&\qquad\qquad\quad new\,\sigma\rangle) \\
&\qquad\qquad update2\,(new\,\sigma \mid \mathrm{L})\ true\,\sigma, \\
&\qquad\quad wrong\ \text{“out of memory”}\ \sigma \\
&\mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!] = \\
&\quad \lambda\rho\gamma\,.\,\lambda\sigma\,. \\
&\qquad new\,\sigma \in \mathrm{L} \rightarrow \\
&\qquad\quad \mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(addframe\,\gamma \\
&\qquad\qquad\qquad \langle single\,(\lambda\epsilon\gamma\,.\,truish\,\epsilon \rightarrow \\
&\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}_1]\!]\rho\gamma, \mathcal{E}[\![\mathrm{E}_2]\!]\rho\gamma), new\,\sigma\rangle) \\
&\qquad\qquad\quad (update2\,(new\,\sigma \mid \mathrm{L})\ true\,\sigma), \\
&\qquad\quad wrong\ \text{“out of memory”}\ \sigma \\
&\mathcal{E}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{E})]\!] = \\
&\quad \lambda\rho\gamma\,.\,\lambda\sigma\,. \\
&\qquad new\,\sigma \in \mathrm{L} \rightarrow \\
&\qquad\quad \mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(addframe\,\gamma \\
&\qquad\qquad\qquad \langle single(\lambda\epsilon\gamma\,.\,assign\,(lookup\,\rho\,\mathrm{I}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \epsilon \\
&\qquad\qquad\qquad\qquad\qquad\qquad (send\,unspecified\,\gamma)), \\
&\qquad\qquad\qquad new\,\sigma\rangle) \\
&\qquad\qquad\quad (update2\,(new\,\sigma \mid \mathrm{L})\ true\,\sigma), \\
&\qquad\quad wrong\ \text{“out of memory”}\ \sigma \\
&\mathcal{E}^*[\![\,]\!] = \lambda\rho\gamma\,.\,send\,\langle\,\rangle\,\gamma \\
&\mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] = \\
&\quad \lambda\rho\gamma\,.\,\lambda\sigma\,. \\
&\qquad new\,\sigma \in \mathrm{L} \rightarrow \\
&\qquad\quad (\lambda\sigma'\,.\,new\,\sigma' \in \mathrm{L} \rightarrow \\
&\qquad\qquad \mathcal{E}[\![\mathrm{E}_0]\!]\,\rho \\
&\qquad\qquad\quad (addframe\,\gamma \\
&\qquad\qquad\qquad \langle single(\lambda\epsilon_0\gamma\,. \\
&\qquad\qquad\qquad\qquad \mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho \\
&\qquad\qquad\qquad\qquad\quad (addframe\,\gamma \\
&\qquad\qquad\qquad\qquad\qquad \langle \lambda\epsilon^*\gamma\,.\,send\,(\langle\epsilon_0\rangle \S \epsilon^*)\gamma,
\end{aligned}
$$

$new\,\sigma'$))),

$new\,\sigma$))

($update2$ ($new\,\sigma'$ | L) $true\,\sigma'$),

$wrong$ "out of memory" σ')

($update2$ ($new\,\sigma$ | L) $true\,\sigma$),

$wrong$ "out of memory" σ)

A.2 補助関数

$hold : \mathtt{L} \to \mathtt{G} \to \mathtt{C}$

$hold = \lambda\alpha\gamma\sigma\,.\,send((\sigma \downarrow 1)\alpha \downarrow 1)\gamma\sigma$

(平成7年 7 月4日受付)
(平成7年10月5日採録)

小宮 常康（学生会員）

1969年生．1989年育英工業高等専門学校電気工学科卒業．1991年豊橋技術科学大学工学部情報工学課程卒業．1993年同大学院工学研究科情報工学専攻修士課程修了．現在，同大学院工学研究科システム情報工学専攻博士課程に在学中．記号処理言語と並列プログラミング言語に興味を持ち，現在は並列記号処理言語に関する研究に従事．

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．現在，同大学教授．理学博士．記号処理と超並列計算に興味を持っている．著書「Common Lisp 入門（共著）」他．電子情報通信学会，情報処理学会，ソフトウェア科学会，IEEE，ACM 各会員．