情報処理学会論文誌　プログラミング　Vol. 3　No. 5　1–17 (Dec. 2010)

L-Closureを用いた真に末尾再帰的なSchemeインタプリタ

八杉昌宏[†1]　小島啓史[†2]　小宮常康[†3]
平石拓[†4]　馬谷誠二[†1]　湯淺太一[†1]

Scheme 処理系は真に末尾再帰的であることが要求されており，アクティブな末尾呼び出しの数に制限がない場合もサポートしなくてはならない．Clinger は真の末尾再帰の形式的定義の 1 つを空間効率の点から与えており，その定義に従えば，末尾呼び出しの最適化（末尾呼び出しをトランポリンなどによりジャンプに置き換えて実装する方法）だけでなく，Baker の CPS（継続渡しスタイル）変換を用いた C 言語における Scheme の実装手法も，真に末尾再帰的と分類できる．Baker の実装手法は，CPS 変換された末尾呼び出しにおいて新たな継続を生成せず，C 言語の実行スタックに対してもごみ集めを行うため，空間効率が良い．本論文では拡張 C 言語による真に末尾再帰的な Scheme インタプリタの実装手法を提案する．本手法は CPS 変換を用いず，C の実行スタックがあふれそうになれば，残りの計算に必要な “Frame” オブジェクトのみを含むリストとして表現された空間効率の良い一級継続を生成し，すぐさまその継続を呼び出すというアイディアに基づく．ごみ集めや継続のキャプチャにおいては，実行スタックに合法的にアクセスできる，つまりデータ構造や変数の値としてアクセスできる L-closure という言語機構を用いている．ベースとなる Scheme インタプリタは，Java アプリケーション組み込み用 Lisp ドライバである JAKLD をもとに C 言語で再実装されたものとした．

A Properly Tail-recursive Scheme Interpreter Using L-closures

Masahiro Yasugi,[†1] Hirofumi Kojima,[†2]
Tsuneyasu Komiya,[†3] Tasuku Hiraishi,[†4]
Seiji Umatani[†1] and Taiichi Yuasa[†1]

Implementations of Scheme are required to be properly tail-recursive and to support an unbounded number of active tail calls. Clinger proposed a formal definition of proper tail recursion based on space efficiency. The definition covers systematic tail call optimization, where every tail call is converted to a jump (with an optional trampoline), as well as Baker's implementation of Scheme in the C language with CPS (continuation-passing style) conversion. Baker's implementation is space-efficient, since no new continuation is created on a CPS-converted tail call and garbage is collected even on C's execution stack. We propose techniques to implement a properly tail-recursive Scheme interpreter in an extended C language. Our approach does not convert a program into CPS. The key idea is to avoid stack overflow by creating a space-efficient first-class continuation represented as a list containing only the "Frame" objects necessary for the rest of computation and immediately invoking the continuation. We use a language mechanism called "L-closures" to access the contents of the execution stack as values of legal data structures and variables for implementing garbage collection and capturing continuations. This research is based on a Scheme interpreter which is developed in the C language by referring to an existing Lisp driver called JAKLD that is intended to be embedded in Java applications.

1. はじめに

関数型言語では反復計算を副作用によらず再帰により表現できる．このとき，末尾再帰の形に書けば制御（control）のための空間（space）を一定以上必要とせず計算が進められると期待されている．たとえば，Lisp 言語の方言である Scheme 言語[16]) では，

```
(define (my-gcd a b) (if (= b 0) a (my-gcd b (remainder a b))))
```

のようにして，ユークリッドの互除法で最大公約数を求める手続き（関数）my-gcd を末尾再帰の形ですっきりと定義できる．

Scheme 処理系は**真に末尾再帰的**（properly tail-recursive）であることが要求されており，アクティブな（リターン前の）**末尾呼び出し**（tail calls）の数に制限がない場合もサポートしなくてはならない．これには，一見しては再帰的とは分からない末尾呼び出しも含む点に注意が必要である．たとえば，CPS（continuation-passing style；継続渡しスタイ

†1 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University
†2 レッドハット株式会社グローバルサービス本部
Red Hat K.K., Global Services, Japan
†3 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, the University of Electro-Communications
†4 京都大学学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University

© 2010 Information Processing Society of Japan

ル）に変換された Scheme プログラムではすべての呼び出しが末尾呼び出しとなり，その大部分は実際には（相互）再帰的であったとしても，一見してはそう分からない．つまり，**真の末尾再帰**（proper tail recursion；以下，“PTR” と略記することがある）とは，末尾再帰呼び出しを関数内部のジャンプ（goto 命令）に置き換えるような命令型言語のコンパイラなどにおける最適化（**末尾再帰の除去**）と似ているが別のものである．

Clinger は Scheme 言語のサブセットを対象に，真の末尾再帰の形式的定義の 1 つを**空間効率**の点から与えている[4]．本研究は，この漸近的空間計算量を用いた定義を満たすような実装手法を扱うものとする．その定義に従えば，末尾呼び出しの最適化を必須とした実装手法だけでなく，Baker の CPS 変換を用いた C 言語における Scheme の実装手法[2] も，真に末尾再帰的と分類できる．ここでの末尾呼び出しの（空間上の）最適化とは，末尾呼び出しの実装手法として広く用いられているものであり，末尾呼び出しを呼び出し先（別の手続きでもよい）への（トランポリン[18] などによる実質的な）ジャンプに置き換えることとする．この場合のジャンプとは，上記の goto 命令とは違い現在の呼び出しをリターン前に終わらせる（現在の環境を捨てる）とともに，「共通のリターンの責任と評価済みの新たな引数」を渡した別の呼び出しを開始することとする．

本論文では拡張 C 言語による真に末尾再帰的な Scheme インタプリタの実装手法を提案する．本手法は CPS 変換を用いず，C の実行スタックがあふれそうになれば，残りの計算に必要な “Frame” オブジェクトのみを含むリストとして表現された空間効率の良い一級継続を生成し，すぐさまその継続を呼び出すというアイディアに基づく．本研究においてリストとして表現される継続オブジェクトはそのまま一級継続として利用できる．ただし，本研究の主な対象は真の末尾再帰の実現法である点に注意してほしい．

ごみ集めや継続のキャプチャにおいては，実行スタックに合法的にアクセスできる，つまりデータ構造や変数の値としてアクセスできる L-closure という言語機構[20),23] を用いている．本研究は，我々が提案している L-closure の実用化の研究という側面も持ち，L-closure によるごみ集めの実装，一級継続の実装，またそれを利用しての PTR の実現とも考えられる．

Scheme インタプリタのベースには，Java 言語で開発された JAKLD[25] を C 言語で再実装したもの[21]（以下，JAKLD/C と呼ぶ）を用いた．JAKLD は Java アプリケーション組み込み用 Lisp ドライバとして設計・開発されたが，通常の Lisp 処理系として単独でも利用可能である．JAKLD では，コンパイラ実装とはせず，組み込み関数や特殊フォームの直接的な実装が可能であり，保守・拡張が容易となっている．今回行った実装においても，コード量がある程度増えるものの直接的な実装が可能となっている．また JAKLD/C は，ごみ集め（GC）の研究に使える点が特徴的である．

本論文の主要な貢献は，次の 4 点にまとめられる．

- 真の末尾再帰の実現のため，「空間効率の良い一級継続を生成してすぐに呼び出す」というアイディアと，これを何らかの基準で実行スタックがあふれそうになったら行い，実行スタックの利用をリセットすることを提案するとともに，より一般化した「とりあえずのものを定数サイズで利用」というアイディアを述べた．また，抽象的なアイディアを具体化した．
- 真の末尾再帰の一般的な定義，形式的な定義を様々な角度から取り上げるとととともに，インフォーマルではあるが，既存の実装方法や提案手法が形式的な定義（定数倍，定数差の違いを無視する定義）を満たすのか議論した．
- 提案する実装手法の詳細を示した．GC とキャプチャの組合せなどでの注意点とその対策などを示すとともに，性能評価を行った．提案方式は CPS 変換などはともなわないため，直接的な実装が容易に行え，保守性などが良い．また同時に，スタックあふれの防止策や一級継続の実装にもなっている．
- L-closure の応用として，インタプリタにおいても GC を実現できること，また，ある程度の規模の処理系に実際に適用できたことを示した．同時に，L-closure によって一級継続が実現できることや，それを利用して真の末尾再帰が実現できることも示した．

以下，2 章で継続や一級継続について述べる．3 章では真の末尾再帰とそのための実装手法について述べるとともに Clinger による形式的定義のエッセンスを紹介する．4 章では Java 言語による Lisp インタプリタである JAKLD とその C 言語による再実装について述べる．5 章で，提案手法の基本アイディアを述べるとともに，その具体化の方針を示す．6 章で，今回提案手法の実装に用いた L-closure を備えた拡張 C 言語 XC-cube について紹介したのち，7 章で提案手法の実装の詳細を示す．8 章では，提案手法などが Clinger による真の末尾再帰の形式的定義を満たすかに関してインフォーマルな考察を行う．9 章で性能評価を行い，10 章で関連研究について述べ，11 章でまとめと今後の課題について述べる．

2. 継続と一級継続

継続（continuations）とは残りの計算のことである．これは多くのプログラミング言語において，プログラムの実行について考察するときに自然に用いられている概念といえる．継続は有用な概念であり，いくつかの使われ方がある．

© 2010 Information Processing Society of Japan

式（expressions）から値を計算するような数学的な計算以外に，副作用を許すような命令型（imperative）プログラミング言語においては，保持する**値**（values）を変更できるような**場所**（locations）が許されているといえる．特に，**表示的意味論**を考える場合に，このような場所から値への対応を関数と考えて**ストア**（stores）と呼ぶ．命令（あるいは文，あるいはコマンド）は，ストアを変化させる，あるいは，数学的にはストアを受け取ってストアを返す関数を表すと考えられる．さらに，そのような言語では，ジャンプ（`goto` 命令など）により，制御（control）を離れた命令に渡す場合にもストアをそのまま引き渡せる．そのような制御渡し命令は，ストアを受け取ってストアを返す関数以上の何かを表さなくてはならない．よって，このような言語の表示的意味論においては，（制御を渡した先などでの）残りの計算を，ストアから最終的な**答え**（answers）を得る関数としてとらえ，これを継続と呼ぶ．つまり，ジャンプとは，ジャンプ先が表す残りの計算（継続）を開始することに相当する．表示的意味論では，プログラムをそのまま扱うので，変数から場所への対応を表す関数も用い，これを**環境**（environments）と呼ぶ．そして，命令の意味は，継続（命令の字面上の後のもの）と環境を追加の引数にとり，命令開始時点の継続を結果とする意味関数で与えられる．ここで，`goto` 命令では命令後の継続は無視することになるし，通常の命令なら「ストアを受け取り変更したストアで命令後の継続を開始して最終的な答え*1を得る」という継続を表すことになる．また，残りの計算において，ストア以外に値を必要とするような継続を考えることもできる．このような継続は式を評価するときに，（値が求まった場合に）その値から残りの計算をするための継続として利用できる．また，副作用のない純粋な関数型言語では，ストアは必要なく，継続は値から最終的な答えを得る関数とすればよい．

スモールステップの**操作的意味論**を抽象機械で示す場合にも，たとえば，CEKS 抽象機械[4] のようにプログラムあるいはコード（C），環境（E），継続（K），ストア（S）を用いることができる．ただし，抽象機械上なので，これらは構文的に（データ構造として）扱う必要がある．たとえば，操作的意味論での**継続**は，表示的意味論とは異なり，**抽象機械のデータ**として生成，破棄される点に注意してほしい．プログラム・コードはもちろん構文的であるが，環境も変数から場所への対応であれば構文的に扱える．「値」のうち，ラムダ式を評価した結果は，環境とラムダ式をペアにしたクロージャ（Scheme の場合は手続きと呼ぶ）として扱えばよい．これによりストアも構文的に扱える．継続については，言語の各コンストラクトについて残りの計算に必要な情報を保持するデータ構造を決めればよい．これは，典型的には，穴のある式に相当するデータ，保存した環境，それと親継続の組とすればよい．抽象機械において新しい継続のデータが必要なとき，その継続を**生成する**と呼ぶ．なお，副作用のない純粋な関数型言語ではストアは必要ないので，CEK 抽象機械[6] として扱うことがある．

CPS（continuation-passing style；継続渡しスタイル）に変換されたプログラムについて考える．たとえば，Scheme 言語で書かれたプログラムを継続渡しスタイルのプログラムに変換すると，継続を Scheme の手続きとして表し，呼び出しの際には必ず渡すようになる．呼び出し先では，変換後の Scheme としての呼び出しからのリターンはせず，渡された継続を呼び出すことで，変換前の Scheme プログラムのリターン相当の制御の移動を実現する．このスタイルでは，変換前の呼び出しもリターンも変換後はすべて末尾呼び出しとして実現される．

また，CPS 変換はコンパイラにおいて利用されることも多い．CPS 変換後は制御の流れなどが明示的になり，最適化などがやりやすくなるためである[12]．しかし，CPS 変換後のソースプログラム（Scheme プログラム）と，CPS 変換を利用したコンパイラ，との違いには注意が必要である．前者はプログラマが接するソースレベルの話であり，後者の変換結果はコンパイラ内部にとどまる限りプログラマが接することはない．

CPS 変換には様々な特徴や効果があるが，CPS 変換後のプログラムだけを対象とした CE 抽象機械，あるいは継続のパラメータを特別扱いした CEK 抽象機械[6] を考えると，継続を表すデータ構造は，クロージャただ 1 つで済む．一般のスタイルのプログラムに対しては継続のデータ構造として多くのバリエーションが必要であった．ただし，CPS 変換前に継続が持っていた情報の一部は，CPS 変換後には追加された事務的（administrative）ラムダ式のパラメータに関する環境に移動する．

また，Scheme 言語や一部の関数型言語では**一級継続**が扱えることが特徴的である．一級継続は，抽象機械レベルのデータである継続を一級のデータとして扱えるように，オブジェクトレベルに**具現化**したものである．一級継続は通常のデータと同じように渡したり，保存したりできる．このように具現化された一級継続を得ることを一級継続を**生成する**と呼ぶ．上で述べた抽象機械レベルのデータとしての継続についても**生成する**という用語を用いたが，それぞれ，オブジェクトレベルの一級継続データ，抽象機械レベルの継続データを対象としているということで明確に区別してほしい．

Scheme 言語の場合は，`call-with-current-continuation` あるいは `call/cc` により，現在の継続を具現化することができる．現在の継続とは，`call/cc` が値とストアをともなっ

*1 表示的意味論なので停止しない計算については表示 ⊥ が得られる．

© 2010 Information Processing Society of Japan

てリターンする先の継続である．Scheme の場合は，手続きとして具体化されており，値を渡して呼び出すことで，一級継続としてキャプチャしたときの継続に “ジャンプ” することができる．

様々な抽象機械の定め方が可能であり，Scheme の仕様を定めた，いわゆる R6RS[16] が定めている操作的意味論では，抽象機械は，そのデータとして式とストアのみを用いる（代入を用いて環境を不要としている）．このとき，式において次に評価すべき部分式を明確にするための**評価文脈**が定められる．評価文脈は継続を表現しており，`call/cc` は，これをキャプチャして Scheme の手続きとしてプログラム中から利用可能とする．R6RS[16] よりシンプルな評価文脈による操作的意味論は，たとえば Griffin の論文[8] などに見られる[*1]．

式	$N ::= x \mid N\,N \mid \lambda x.N \mid \mathcal{A}(N) \mid \mathcal{K}(N)$
値	$V ::= x \mid \lambda x.N$
評価文脈	$E ::= [\,] \mid E\,N \mid V\,E$
β 簡約	$E[(\lambda x.N)V] \rightarrow_\beta E[N[V/x]]$
abort	$E[\mathcal{A}(N)] \rightarrow_\mathcal{A} N$
call/cc	$E[\mathcal{K}(N)] \rightarrow_\mathcal{K} E[N\,\lambda x.\mathcal{A}(E[x])]$

ここで，E は $E[V]$ と V を渡されることで行う残りの計算（継続）を表現しており，操作的意味論における抽象機械のデータとしての（式の一部としての）「継続」にあたる．この場合に「継続を生成する」とは，評価文脈 E における評価を進めた結果，次に評価を進めるべき部分式 N' は $E'[N']$ といった新しい評価文脈 E' を必要とするときに E' が表現する継続を生成していると考える．一方，call/cc 規則で作られる $\lambda x.\mathcal{A}(E[x])$ が「一級継続」であり，このラムダ式（に該当するデータ）を得ることを指して「一級継続を生成する」という．以上のように定めれば「継続を生成する」と「一級継続を生成する」は明確に区別できる．

一級継続の扱い自体では，その空間効率を考える必要はないが，本研究では，次章で詳しく述べる真の末尾再帰の扱いのために，その空間効率を考えることになる．

3. 真の末尾再帰

1 章で述べたように，いくつかの関数型言語の仕様では，その処理系が真に末尾再帰的であることを要求している．たとえば，末尾呼び出しにおいて再帰的に書かれた反復的な計算が，（その制御のために）反復回数に比例するような空間を必要としてはならない．

呼び出し先においては新しい環境を用いるので，呼び出し元で（呼び出しの結果を使って）呼び出し後の計算を続けるためには，呼び出し前の環境などの情報を残しておく必要があると考えるのが自然である．すなわち，2 章で述べた抽象機械におけるデータとして「結果を受け取り環境を復元しつつ残りの計算を進める」という継続を生成し，呼び出し先に渡して「リターン」してもらう，つまりこの継続に結果とともにジャンプしてもらうと考えると分かりやすい．しかし，このままではまだリターンしていない呼び出しの回数分は継続が生成されており，反復計算の例においては，反復回数に比例した大きさの空間を継続のために必要とする．

末尾呼び出しとは，末尾文脈（tail contexts）に出現する呼び出しとされている．ラムダ式の本体における末尾式は末尾文脈に出現する．末尾式の値はそのラムダ式の手続きがリターンする値でもある．また末尾文脈に出現する `if` 式などで，それが `if` 式全体の値となるような末尾式も末尾文脈に出現するという．つまり，末尾文脈に出現する末尾式の値は現在の手続きがリターンする値となる．末尾文脈においては，実は結果が得られた後，リターンする（呼び出し元の継続にジャンプする）以外にやることはない．よって，末尾呼び出しを行う場合には，新たに継続を生成せずに同じ継続を末尾呼び出し先に渡して，直接そちらにリターンしてもらえれば十分である．この場合，新たに継続は生成されないため，アクティブな末尾呼び出しの個数が増えてもその制御のために個数に比例した大きさの空間を必要としないと期待され，そのような実装は真に末尾再帰的になると期待される．

実際には，3.3 節で紹介する形式的定義においては，末尾呼び出し以外の呼び出しにおいても新たな継続を生成しない**参照実装**を用いている．このことは 3.3 節で紹介するとともに 8 章で考察を行う．また，この Scheme についての形式的定義において，末尾文脈に現れない `let` 式における本体の末尾の式は末尾文脈に出現しているといえるのかに複数の解釈が可能である．形式的定義では `let` 式はマクロとして扱われているとも想定され，これをラムダ式に展開するとその本体の末尾式は末尾文脈に出現していることになるためである．

ここで，真の末尾再帰（proper tail recursion）という用語に対する本論文のスタンスを示しておく．まず「真の」は「適正な」とすべきかもしれないが慣習に従うことにした．一方，末尾呼び出しを対象とするのに，末尾再帰と呼ばれている理由には次の 3 つの説が考えられる．1 つは，末尾呼び出しからまた（別の手続きでよいので）末尾呼び出しが「プログラム上のラムダ式の数」を超えて繰り返される場合は，一見しては分からないとしても相互

---

*1 Griffin の論文では実際には `call/cc` を表す $\mathcal{K}$ とは別のオペレータに着目して一級継続を備えた言語における Curry/Howard 同型を論じている．

© 2010 Information Processing Society of Japan

再帰があるはずである．この場合は，そのような相互末尾再帰が無制限に繰り返されたとしても，真に末尾再帰的な実装であれば対応できるという意味が考えられる．他には，末尾呼び出しにおいて新たに継続を生成せずに末尾呼び出し先に環境の変更をともなうジャンプをするという考え方と，その実装において空間を節約する点が，命令型言語における末尾再帰の除去（環境の変更をともなわない `goto` 命令へ置き換え）という最適化技法と似ていることから，末尾再帰という用語が使われたのかもしれない．最後に，末尾呼び出しからまた（別の手続きでよいので）末尾呼び出しが繰り返される場合を考えると，新たに継続を生成しない実装がなされれば，そのとき継続としては同じ継続が繰り返し現れることになる．つまり継続側に着目すると同じ継続が再帰的に現れることになり，用語と一致する．

3.1　トランポリン

末尾呼び出しにおいて，新たに継続を生成せずに呼び出し先に制御を移すのは，実装の枠組みによっては難しいことがある．特に C 言語のような実行スタックを使った呼び出しを行う言語上に実装を行い，Scheme の手続き呼び出しや特殊フォーム（言語コンストラクト）の評価における制御の移動を C の関数呼び出しに直接対応させると，C の関数呼び出しでは必ず実行スタックを消費することから，何か対処しない限り，呼び出し数に比例する空間計算量となってしまう．Java 言語上でも同様のことがいえる．

また，実行スタックを消費せずに制御を渡すために 1 つの大きな関数にジャンプ先をすべて入れてしまうという実装は，コンパイル時間やその他の制限や保守性の問題から避けたいことも多い．

その対策としてトランポリンを用いる手法[18] が知られている．たとえば，関数 A1 から関数 A2 へ “ジャンプ” したいときは，A2 のアドレスをリターンする．A1 の呼び出し元はリターンされた A2 を呼び出す．つまりこの呼び出し元を B1 とすると，B1 はトランポリンのようにリターンされた関数を繰り返し呼び出すものとする．これにより，A2 からさらに別の関数 A3 への “ジャンプ” なども B1 を経由することにより実現できる．これを繰り返しても毎度リターンしているため，実行スタックの使用量は定数で抑えられる．もちろん，Scheme の手続き呼び出しに関する “ジャンプ” を実現するには，手続き以外に引数の情報も必要となるので，実際の処理系でリターンするのは次に呼び出したい関数を含む情報となる．

末尾呼び出しの（空間上の）最適化を必須とした実装手法に，このトランポリンを用いた手法が利用できる．つまり，末尾呼び出しを呼び出し先（別の手続きでもよい）へのトランポリン[18] などによる実質的なジャンプに置き換える．この場合のジャンプとは，現在の呼び出しをリターン前に終わらせる（現在の環境を捨てる）とともに，「同じ継続と新たな引数」を渡した別の呼び出しを開始することとする．このようなジャンプは，真の末尾再帰のための末尾呼び出しの実装手法として広く用いられているものであり，真の末尾再帰と同一と見なされていることも多い．トランポリンを用いない場合も，新たな引数などによる環境で現在の環境を上書きするといった実装がなされることが多い．

3.2　Baker の実装手法

Baker の実装手法[2] は，CPS 変換された末尾呼び出しにおいて新たな継続を生成せず，C 言語の実行スタックに対してもごみ集めを行うため，空間効率が良い．これはコンパイル方式を想定している．

Baker の手法では，Scheme のプログラムを継続渡しスタイルの C 言語プログラムへコンパイルする．継続のクロージャは，環境を引数で受け取る C 言語の関数ポインタと C 言語のスタックに置かれる環境からなる C 言語データとして表現する．スタックのごみ集めは，現在の継続を表すクロージャのみをトレースすることで実現できる（Baker の手法では，継続のクロージャ以外に Scheme のオブジェクトもスタックに割り当てる）．スタックはつねに延びる方向にのみ利用し，決して C の上ではリターンしないので，オブジェクトの連続割当てが可能となる．

スタックはごみ集めできるため，あらかじめ継続のクロージャをヒープに割り当てる方式（SML/NJ [1] など）と同様といえ，末尾呼び出し先に，新たに継続を生成しないで渡すというのは「そのまま」であり，この点においては，PTR が実現できることは明らかである．

スタックを含めた GC はスタックの利用サイズがある一定の基準を超えそうになれば行う．継続のクロージャなどは，ポインタを介していつでもアクセスできるようにデータとしてメモリに配置されているので，コピー GC の対象としてスタックの外に確保されたヒープ空間に退避させることができる．このとき，C のフレームの詳細は使わないし，邪魔にはならない．実行スタックをリセットするには `longjmp` などを用いる．

この Baker の方式をもとに開発された Scheme コンパイラに，Chicken Scheme コンパイラ[3] がある．

3.3　真の末尾再帰の形式的定義

文献 4) で述べられている Clinger による真の末尾再帰の形式的定義を簡単に紹介する．ポイントとなるキーワードは，**参照実装**，**空間消費関数**，**漸近的空間計算量**である．また，GC が重要な役割を果たす．詳細については，文献 4) を参照いただきたい．

プログラム P を入力 D により実行したときに実装 X が消費する可能性のある最大の空

© 2010 Information Processing Society of Japan

間消費量を，$S_X(P,D)$ の値が実数か無限大となるような関数 S_X で表す．S_X を実装 X の空間消費関数と呼ぶ．2 つの実装を，それぞれの空間消費関数の漸近的振舞いにより比較する．特に参照実装 tail を CEKS 抽象機械として与え，その空間消費関数 S_{tail} を与える．その与え方は，$S_{\mathrm{tail}}(P,D)$ の値が，可能な（非決定的な）実行列すべてについての「各実行列における最大の空間消費量」の最大値となるようにする．ここで，実行列は GC を可能な限り行うものとし，場合によっては可算無限長である．そして，吟味したい実装の空間消費関数 S が与えられたとき，

$$\forall(P,D).\, S(P,D) \leq c_1 S_{\mathrm{tail}}(P,D) + c_0$$

を満たすような実定数 c_1（> 0）と実定数 c_0 が存在することを，S は S_{tail} の漸近的空間計算量クラス $O(S_{\mathrm{tail}})$ に属するという．そして，空間消費関数 S を持つ実装が真に末尾再帰的であるとは，S が集合 $O(S_{\mathrm{tail}})$ に属することであると定義する．

以上のことをごくおおざっぱにいえば，PTR の基準となる CEKS 抽象機械として与えられる参照実装 tail に対して，定数倍（係数），定数差（定数項）の違いを無視すれば，どんなプログラムや入力に対しても空間消費関数の値が悪くはないならば PTR に合格ということになる．

参照実装 tail の CEKS 抽象機械は，基本的には愚直であるが，（末尾呼び出しだけでなく）手続き呼び出し時には新たに継続を生成せず，環境（E）レジスタの変更をともなう呼び出し先のコードへのジャンプを行う点を特徴とする．継続は，呼び出しのためではなく部分式を評価するために生成される[17]．これらの継続にはすでに環境が保存されているため，部分式として手続き呼び出しが現れたとしても，改めてリターン後の環境を保存する必要はない．

またこの CEKS 抽象機械はごみ集め規則を持つ．ストア（S）は場所から値への対応をとるが，今後アクセスされることがない場所に関してはストアから取り除きたい．そのためには，場所の集合を，ごみの候補として考えたとき，それらを一括して取り除いても，ストアだけでなく，現在のコード，現在の環境，現在の継続という残ったところにも取り除いた場所が現れないということが条件となる．

実際の処理系では GC までに時間的猶予があるため，各瞬間の空間消費量だけを考えたのでは大きさの違いは何倍にもなってしまい，定数係数で抑えられない恐れがあるが，Clinger の形式的定義では，実行列の時間方向の最大値をとっているため大丈夫である．

ただ，Clinger の形式的定義での空間消費量は，単純な足し算であるのが，実際のメモリ管理においてフラグメンテーションが起こるような方式だと，その未使用部分を考慮すると最悪何倍にでも空間を使っていることになってしまう．この点は形式的定義では明示的に扱われていないが，単純な話ではない．ただ，フラグメンテーションが起こる空間の大きさを一定（定数）以内に限定すれば漸近的空間計算量により扱えると考えられる．

なお，Clinger による真の末尾再帰の形式的定義では，一級継続は扱っていない．

4. JAKLD

JAKLD[25] とは，「Java アプリケーション組み込み用 Lisp ドライバ」のことであると考えられている．

4.1 Java 言語による Scheme インタプリタ

JAKLD 処理系は，本来，Java 言語によって開発されるアプリケーションに組み込んで使用することを主要目的として設計・開発された Lisp ドライバであるが，通常の Lisp 処理系として単独でも利用可能である．

JAKLD の設計には次のような項目が重視されている．

(1) Lisp 処理系の実装ノウハウを持たない Java プログラマにも機能の追加・削除・変更が容易に行えること．
(2) Java で開発したソフトウェア部品を扱うための機能を容易に組み込めること．
(3) コンパクトな実装であること．
(4) 高度な Lisp プログラム開発支援ツールを備える必要はないが，デバッグのために最低限必要な機能は備えること．
(5) 高性能である必要はないが，性能が極端に悪くないこと．

特に，機能の追加は分かりやすく，コンパイラではないため，関数や特殊フォームを直接的に書くことができる．部分式の評価は単純に環境などとともに `eval` を呼び出すだけでよい．

JAKLD は非常にコンパクトな処理系であるが，IEEE Scheme のほぼフルセットを採用している．IEEE Scheme に準拠していない点として，`call/cc` により生成された Scheme 手続きとしての継続は，`call/cc` がリターンした後は呼び出せない．つまり，escape procedure として機能し，Common Lisp における catch & throw のような非局所的脱出には利用できるが，コルーチンを実現することはできない．

最近，トランポリンを追加することで，PTR に対応している．

4.2 C 言語による再実装

我々の研究室では，JAKLD を C 言語で再実装した処理系がある[21]．ここでは，JAKLD/C

© 2010 Information Processing Society of Japan

と呼ぶことにする．JAKLD/C は，いわゆる bignum のサポートを取り除いた言語仕様となっている．

ごみ集め（GC）のアルゴリズムを実験するために開発されたインタプリタである，という特徴がある．そのため GC の実装を簡単に行えるように，次の 3 点の特徴がある．

- オブジェクトに対する読み出し，書き込みの部分はマクロで書かれており，複製方式 GC[13] やスナップショット GC などの実装に必要なリードバリア，ライトバリアの実装が容易である．
- メモリ管理部と解釈実行部分とが完全に分離されており，解釈実行部に手を加えることなく GC の実装が可能である．
- どの GC アルゴリズムにおいても必要なルートスキャンやオブジェクトスキャンは，共通部分として実装済みである．

これらの特徴により，処理系内部の実装とは独立して GC の実装が可能となる．

ただし，ルートとなるパラメータや局所変数の管理は，ルートのアドレスのスタックを用いており，そのままではこれらの変数のレジスタ割当てが阻害されてしまう．本研究では，別の研究で開発した L-closure を適用して GC の性能改善も目指す．

次章で述べる基本アイディアに基づく提案手法との比較を行うために，JAKLD に追加されたトランポリンを参考に，JAKLD/C にトランポリンを追加した処理系も作成した．

5. 基本アイディア

真の末尾再帰を実装するための提案手法の基本アイディアは，「空間効率の良い一級継続を生成しすぐに呼び出す」であり，これを何らかの基準で実行スタックがあふれそうになったら行い，実行スタックの利用をリセットすることである．空間効率が良い一級継続を生成することで，空間効率が改善され，また実行スタックのための空間の大きさを一定（適当な定数）以下にとどめるという基準を設けておくことで，定数差以上には実行スタックのために空間を消費しない．Clinger の形式的定義で漸近的な空間計算量を考えるときには，空間消費量関数における実行スタックに関する定数項は考慮する必要はないため，真の末尾再帰と分類できる．

このアイディアの背景にある，さらに一般化したアイディアは，「とりあえずのものを定数サイズで利用」というアイディアである．ここで，「とりあえず」とはまずは利用してみればよく，必要なら利用するのを止めればよいものとする．「一時的」ともいえるが，そのまま利用し続けてもよいので，文字どおりの「一時的」とは限らない．もちろん，利用するからにはその性能が良いなどの長所を持つことが期待される．いわゆる「キャッシュ」などはこの一般化したアイディアに含まれるであろう．提案手法のように大きさが一定（適当な定数）以下の「実行スタック」を使うというアイディアに限っても，他に「実時間ごみ集め」に役立てるということも考えられる．つまり，本論文ではこれ以上扱わないが，実行スタックの大きさが一定以下で，そのスキャンの時間が限定でき，それが最大停止時間の要求と比較して十分小さいなら合格とするといった実装手法が考えられる．

実際の処理系にこのような抽象的なアイディアを適用していくには，アイディアの具体化が必要である．特に「とりあえず」利用したものの利用をどのようにして止めるのかがポイントになる．実行スタックの場合，その中で管理している計算を続けるためのデータを救い出す必要がある．これには

- ポインタを介していつでもアクセスできるようにデータをメモリに配置する方式
- 局所変数で（callee セーブレジスタなどに）レジスタ割当てがなされているようなデータであっても関数をリターンするなどしてアクセスする方式

が考えられる．どちらもデータの救い出しに対する「備え」をすることには変わりがないが，「とりあえず」の（といってもほとんどそれだけで済ませられるかもしれない）ものを利用している間の性能の良さでは後者のほうが有利である．次章で述べる L-closure[20),23)] は，後者の性能の良さで「備え」を提供するものであり，関数をリターンしなくても，アクセスを可能とするものとして開発された．他の方式を用いることも可能であるが今回は，主にこの L-closure と，それと同時に開発した少しマイルドな “closure” で実装を進めた．

6. L-Closure を備えた拡張 C 言語 XC-cube

「L-closure」と名付けた安全な計算状態操作機構の提案，設計とその実装研究，応用研究を行っている．その実装研究のうち C コンパイラ拡張方式[20),23)] を今回は用いている．

L-closure は C 言語のような手続き型プログラミング言語を拡張する言語機構として提案している．そのような拡張 C 言語は，直接のアプリケーション作成よりは，高水準言語コンパイラなどにおける中間言語に用いることを意図している．たとえばごみ集めを備えた高水準言語の実装では，メモリ不足を避けるためのごみ集めの際に「ルート（「参照」を保持する変数)」をスキャンする必要があるが，翻訳（コンパイル）後の中間コードにおいて提案機構を用いれば，スタックにとられる呼び出し元に眠る変数の値に操作を加えられる．ただし今回，本論文においては，中間コード生成を用いずに Scheme インタプリタのプログラムを直接，人が書くために拡張 C 言語を用いており，そのような使い方も可能である．

© 2010 Information Processing Society of Japan

L-closure が実現するのは「備え（あれば憂いなし）」である．しかも平常時の実行効率の低下を巧妙に避ける．非平常時の特別処理は裏方の仕事であり，軽視されがちであるが，抜本的再構成やメンテナンスによりソフトウェアの信頼性と柔軟性を高めるのに重要である．

L-closure は，いわゆる「例外処理」と対比できる．例外処理機構を備えたプログラミング言語では，例外発生時に例外ハンドラに制御を移して特別処理を行う．一般にはその際，例外ハンドラ設定点までのコンテキストを捨て非局所脱出する．一方，入れ子の関数定義から生成される L-closure の，非局所的な「呼び出し」により特別処理を行う提案方式では，呼び出し後に平常時の計算に戻るし，複数の呼び出しを組み合わせられる．よって提案機構のプログラミング言語への導入は，例外処理以上の力強さを持つともいえる．

実装手法としては，L-closure の初期化をその実際の呼び出しまで遅延させる，L-closure からアクセスされる変数へのレジスタ割当てを可能とする（メモリと疑似レジスタの両方に変数の場所を準備し，その間の一貫性維持を遅延させる）といった手法を用いた．

7. 実装の詳細

まず，ごみ集め（GC）の実装について述べる．L-closure を用いた GC の実装では，実行スタック中のルートに L-closure を用いてアクセスする．その方法は文献 20)，23) のとおりであるが，ある程度の規模に適用したもの，あるいは，インタプリタに適用したものを発表するのは，今回が最初となる．**図 1** と**図 2** に具体的なコード例を示す．ここで，図 1 ではルートのアドレス用スタックに，ルートのアドレスを push したり pop したりしているが，図 2 ではその代わりに L-closure `scan1` 内でルートとなる変数（`e1` など）をスキャンするような「備え」を準備している．そして `cond` について `eval` を呼び出す際に，この L-closure `scan1` を渡して必要なら呼び出してもらう．L-closure `scan1` が呼び出されると，関数 `l_if` に渡された L-closure `scan0` を呼び出すことによって実行スタックのさらに深い部分のスキャンを行う．`scan0` でも同様により深い部分のスキャンを行うことで，実行スタック全体のルートをスキャンできる．このとき，L-closure の存在のせいで変数 `e1` がレジスタ割当ての対象からはずれるといったことにならないようにしている．

さらに L-closure を用いて一級継続を実装したものが**図 3** である．L-closure `scan1` を，GC のためにも，キャプチャのためにも利用している．また，変数 `pc` を追加し，関数の呼び出し箇所を整数値で表現している．図 3 の場合，`cond` の `eval` の呼び出しにおいて `pc` を 1 にセットしている．`scan1` をキャプチャのために呼び出す場合はパラメータ `w` に `ForCapture` を渡す．マクロ `SAVE_FRAME` は，新しい “`Frame`” オブジェクトを生成し，`Frame` オブジェクトのリストの先頭を保持する `Frame` 型の大域変数 `x_frame` にアクセスし，その先頭に追加する．`Frame` オブジェクトのリストの次の要素は `Frame` オブジェクト自身の `next` フィールドでたどれるようにしている．生成した `Frame` オブジェクトには，この `l_if` 関数が行っている計算の続きをするのに必要な情報を保存する．まず，関数 `l_if_c`（へのポインタ）と `pc` の値を，それぞれ `fp` フィールドと `pc` フィールドに保存する．関数 `l_if` の代わりに関数 `l_if_c` とする理由は後述する．次に，マクロ `SAVE_PARAM4` で `l_if` 関数の 4 つの

```
#define DDl_if def_sp(l_if, "if", "OO", "O", false)
static void *l_if(Object cond, Object e1, Object e2, Env env)
{
  void *result;

  new_push3(e1, e2, env); //3
  result = eval(cond, env);
  new_pop(3); //0

  if (!EQP(result, F))
    return eval(e1, env);
  else
    return (e2 == NULL) ? Nil : eval(e2, env);
}
```

図 1　JAKLD/C における “`if`” の実装
Fig. 1　Implementing `if` for JAKLD/C.

```
static void *l_if(scanL scan0, Object cond, Object e1, Object e2, Env env)
{
  void *result;

  void scan1 lightweight (move_f mv){
    scan0(mv); e1 = mv(e1); e2 = mv(e2); env = mv(env);
  }

  result = eval(scan1, cond, env);

  if (!EQP(result, F))
    return eval(scan0, e1, env);
  else
    return (e2 == NULL) ? Nil : eval(scan0, e2, env);
}
```

図 2　L-closure を用いたコピー GC の実装
Fig. 2　Implementing copying GC with L-closures.

© 2010 Information Processing Society of Japan

```
/* globals */
scanL scan_in_capture;
void *last_val;
Frame x_frame;

static void *l_if(scanL scan0, Object cond, Object e1, Object e2, Env env) {
  void *result;
  int pc = 0;
  void scan1 lightweight (enum why_scan w, move_f mv) {
    switch(w){
    case ForGC:
      scan0(w, mv); e1 = mv(e1); e2 = mv(e2); env = mv(env);
      break;
    case ForCapture:
      scan0(w, mv);
      SAVE_FRAME(x_frame, scan_in_capture, l_if_c, pc);
      SAVE_PARAM4(x_frame, scan_in_capture, NULL, e1, e2, env);
      SAVE_LOCAL0(x_frame, scan_in_capture);
    }
  }

  if (last_val) {
    pc = REF(x_frame, pc); RESTORE_LOCAL0(x_frame);
    x_frame = 0;
    switch(pc) {
    case 1: result = last_val; last_val = 0; goto L1;
    }
  }

  pc = 1;
  result = eval(scan1, cond, env);
 L1:;

  if (!EQP(result, F))
    return eval(scan0, e1, env); // tail call
  else
    return (e2 == NULL) ? Nil : eval(scan0, e2, env);  // tail call
}

static void l_if_c(scanL scan0) {
  Vector params = REF(x_frame, params);
  last_val = l_if(scan0, REF(params, value[0]), REF(params, value[1]),
                         REF(params, value[2]), REF(params, value[3]));
}
```

図 3　L-closure を用いたコピー GC と一級継続の実装
Fig. 3　Implementing copying GC and first-class continuations with L-closures.

パラメータである cond，e1，e2，env の値を保存する．これにはベクタオブジェクトを借りて使っている．ベクタオブジェクトは Frame オブジェクトの params フィールドに保存する．ただし，ここでは，cond はもはや生きていないため，参照先のないことを示す特別な値 NULL を保存している．次に，マクロ SAVE_LOCAL0 で l_if 関数の局所変数を保存する．これには同様にベクタオブジェクトを使い，Frame オブジェクトの locals フィールドに保存する．ただし，この例ではたまたま保存すべき局所変数は別扱いの pc の他にはなく 0 個であるので，このマクロですべき処理はない．

キャプチャを行っている間にも GC が発生する可能性がある点に注意すべきである．たとえば，Frame オブジェクトの生成があるし，パラメータや局所変数のためのベクタオブジェクトの生成もあるためである．その対策として，大域変数 scan_in_capture を準備し，継続のキャプチャを開始するときにはこの大域変数に L-closure を設定し，キャプチャ中であっても設定された L-closure を用いればルートスキャンができるようにしておくこととした．また，キャプチャ中にルートが増加するのも問題である．そこで，大域変数 x_frame を用いることとし，コレクタがスキャンできない部分を作らないようにしている．

詳細は後述するが，基本的には以上のようにして，現在の継続をキャプチャし，Frame オブジェクトのリストとして**一級継続**を生成することができる．

このように Frame オブジェクトとして保存された「一級継続（の一部）」を呼び出すには，(1) リターンされた値を last_val にセットし，(2) Frame オブジェクトを大域変数 x_frame にセットし，(3) Frame オブジェクトの fp フィールドに保存した関数ポインタ（現在の例では l_if_c）を呼び出せばよい．ここで，大域変数 x_frame は，関数間での Frame オブジェクトの授受に用いることとし，キャプチャ時に作りかけの一級継続を表す Frame オブジェクトの先頭を保持するのとは別の使い道となっている．図 3 の例では，関数 l_if_c はパラメータらを取り出し，それらを渡して l_if を呼び出す．このようにすることで，(3) の呼び出しは，どんな Frame オブジェクトについても（引数の数は一定であるため）共通にできる．呼び出された l_if は図 3 の中央付近にあるように，マクロ RESTORE_LOCAL0 で局所変数を復元し（ただし，この例では 0 個），リターンされた値を last_val から得て result に設定し，pc の値に対応する関数呼び出しの後ろから処理を再開する．なお，その前に，関数間でリターン値や Frame オブジェクトを授受するのに用いた last_val と x_frame の値をクリアしてある．

図 3 の例では，関数 l_if_c は関数 l_if がリターンした値を last_val にセットしている．この値は「次の」Frame オブジェクトにより利用される．すなわち，保存した Frame

© 2010 Information Processing Society of Japan

```
next_frame = x_frame;                    // 実行すべき継続
while (next_frame) {
  x_frame = next_frame;                  // 実行すべき継続の 1 フレーム分
  next_frame = REF(next_frame, next);    // 1 フレームを除く残りのフレーム
  REF(x_frame, fp)(scan1);               // 1 フレーム分実行
}
```

図 4　継続実行スケジューラ
Fig. 4　Continuation scheduler.

オブジェクトのリスト（一級継続）を呼び出すには，**図 4** に示す継続実行用スケジューラを用いればよい．このループは main 関数内にあり，next_frame はその局所変数である．継続のキャプチャ時にもこのスケジューラは働いていることがあり，next_frame には，前回以前のキャプチャにおいて生成済みの Frame オブジェクトのリストの先頭が（一級継続が表す残りの計算の一部として）保存されていることになる．

以上をふまえて，キャプチャ時の動作をより詳細に説明すると，図 3 の例で L-closure scan1 を呼び出したときには，まず l_if に呼び出し元から渡されている L-closure scan0 を呼び出して，より深い側の Frame オブジェクトを生成する．scan1 ではその「残りの計算」を表す Frame オブジェクトのリストの先頭に，l_if 自身の部分的な継続を表す Frame オブジェクトを追加する．scan0 でも同様により深い側の Frame オブジェクトを生成するというネストが呼び出し元にさかのぼって続けられるが，main 関数が持つ L-closure（図 4 のスケジューラのコードで渡している scan1）までさかのぼったときには，この L-closure 内では

```
x_frame = next_frame;
```

を実行する．これにより，すでに生成済みの一級継続のための Frame オブジェクトのリストの適当な位置から tail 側を共有しつつ，別の先頭となる Frame オブジェクトを追加していく形で，現在の継続のキャプチャが可能となる．

次に，一級継続の呼び出し時の動作をより詳細に説明する．一級継続を呼び出すには，(1) リターンしたい値を last_val にセットし，(2) その一級継続を表現する Frame オブジェクトを大域変数 x_frame にセットし，図 4 の継続実行スケジューラに実行を委ねればよい．図 4 は main 関数内にあるため，C ライブラリ関数である longjmp を用いれば（かつ，そうできるように setjmp しておけば），現在の継続を捨てて継続実行スケジューラに実行を委ねることができる．

なお，call/cc で扱うような一級継続については，スタックからの継続のキャプチャ後に，スタックをリセットしてキャプチャした一級継続で使うことで，フレームの共有が進むことが知られている[5]．本処理系では，(1) 実行スタックを残してより高速な実行ができるであろうが続く call/cc で継続の共通部分も再キャプチャすることになる実装と，(2) スタックをリセットして共有を促進する実装を行った．一級継続の効率良い実現のためには実行スタックに関しても共有を行うといった研究が考えられるが，今回の研究の主題は PTR にあり，一級継続の効率良い実現は今後の課題とする．ここで，call/cc を用いない場合，PTR のために提案手法が生成する一級継続は 1 つしかないので，再キャプチャすることになる部分継続は実行スタックには存在しない．

提案する PTR の実装手法は，実行スタック利用を限定して，基準を超えるようなら「空間効率の良い一級継続を生成しすぐに呼び出す」ことであった．これは，基準を超えないかを主要な関数，たとえば eval 関数でチェックしつつ，超えた場合は，一級継続を生成し，すぐにこれを呼び出すことで実現できる．もちろん，実際に空間効率の良い一級継続となるように実装する必要がある．つまり重要な点の 1 つは，従来手法でいえば末尾呼び出しにおいて新たな継続を生成しないことである．提案手法に言い換えると，とりあえず実行スタックにおいては新たな継続を用いてもよいが，キャプチャを行い一級継続を生成するときには，末尾呼び出しにおいて新たな継続を生成していなかった形にする必要がある．実際に，図 3 での 2 つの（eval の）末尾呼び出しにおいては，l_if 自身の持つ scan1 ではなく，呼び出し元から渡された scan0 をそのまま渡している．つまり，キャプチャ時には l_if 自身の処理内容のない Frame オブジェクトは生成されず，l_if の呼び出し元が l_if に渡した継続に値がリターンできるように，より短い Frame オブジェクトのリストの形で一級継続が生成されることになる．

8. 真に末尾再帰であるかに関するインフォーマルな考察

本章では，提案手法や既存の実装手法が PTR の形式的定義を満たすのかについて，インフォーマルな考察を行う．より形式的な考察は今後の課題とする．

まず，形式的定義における参照実装（抽象機械）では，末尾に限らずすべての呼び出しにおいて新たな継続を生成しない点と，実際の実装では「末尾呼び出しでは新たな継続を生成しないが，それ以外の呼び出しでは環境を保存するための新たな継続を生成する」ことがあるという点とのギャップについて考察する．これは，参照実装においてはあらかじめ部分式を評価するときには環境を保存した継続を生成しており，あらためて保存する必要はないためである．実際の実装においては，部分式の評価においては継続を生成せず（環境を保存せ

© 2010 Information Processing Society of Japan

ず)，部分式が（したがって末尾でない）呼び出しのときのみ継続を生成する（環境を保存する）方式が考えられる．しかし，これは，単純により lazy に生成（保存）を行っているだけであり，空間効率が悪くなることはない．また，実際の実装において，部分式の評価と末尾でない呼び出しの両方で継続を生成（環境を保存）する方法も考えられるが，定数倍の違いなので問題ない．以上により，末尾呼び出しのときには新たな継続を生成しないという手法に特に問題ない．

次に，トランポリンを用いた手法について考察する．これは継続を生成せずにジャンプするという考え方を素直に実現しているものであり，普通に考えれば特に問題はない．問題が生ずるとしたら，たとえば，トランポリンのところに除去し忘れた参照が残り余計なオブジェクトが生き残るといった場合である．しかし，これはどのような処理系実装でも共通して注意すべき点である．

提案方式が形式的定義を満たしているかについて考察する．一定（定数）以下の大きさしか実行スタックが使われないこと，継続をキャプチャして一級継続の形として呼び出したときに実行スタックはリセットされ実行スタック中に除去し忘れの参照は残らないことから，この部分には定数差の議論をあてはめることができる．また，`Frame` オブジェクトについては固定サイズであり問題ない，C のパラメータや局所変数の個数もある定数以下であるため問題ない．本当に可変長なのは Scheme の引数などであるが，これは C の引数ではなく C の引数に含まれる環境オブジェクトとして扱われており，そのサイズに比例した空間を利用するものである．あとは GC が無駄なく行えるように実装されていればよい．つまり，Scheme としての実行を考えると，もう使われなくなる変数についてはその参照をクリアしたり，特殊な値（NULL）を代わりに保存したりするなどして GC に気を配る必要がある．

Baker 方式は，CPS 変換を用いて継続はすべてクロージャとして実現される．Baker 方式では，その割当て（それ以外のオブジェクトを含め）をヒープ扱い可能な（ごみ集め可能な空間としての）実行スタック上に行う．この空間はヒープでもあるので，その点だけでいえば，最も自然に Clinger の定義を満たせる．むしろ，スタックには連続割当てしかできないことの制約が，空間計算量に与える影響を考えるべきではあるが，この点，Clinger の定義では考えられていない．ただし，スタックの大きさを提案手法と同様に一定（定数）以下とし，この部分は「配置が不自由な空間」と考えるならば，Clinger の定義において無視可能な定数差の違いとなり問題ない．

2 章で，CPS 変換を行うと，変換前は継続の持っていた情報の一部は変換後は環境が持つようになるという点を指摘した．たとえば「継続」の中で表現されている「評価済みの引数」は，CPS 変換すると一時変数の値として「環境」のほうに移される．Baker 方式も含めてこの点を考察する必要がある．つまり環境に移動しているため，環境の扱いによっては参照先が増え，空間効率が悪化する恐れがある．同様の問題は A 正規変換[6] にもある．実は，「継続」のためのクロージャを作るときに，変数として環境に移されたもののうち，自由変数（コードとして使う変数）のみについてコピーする形で flat でコンパクトな環境を閉じ込めれば問題ない．逆に，事務管理上導入された CPS 変換前には存在しなかった変数をそのまま環境に残して継続のためのクロージャを生成すると問題が生じる．以下が限定しないと問題となる例である：

```
(h (g 1 (f 1 1) ... (f 1 n))
   (g 2 (f 2 1) ... (f 2 n))
    ...
   (g n (f n 1) ... (f n n)))
```

ここで (`f` x y) が $x+y$ の大きさのオブジェクトを返すとすると，CPS 変換後の $O(n^2)$ 個の `f` の呼び出し結果を保持する一時変数（変換により生成された変数）をそのまま含む環境では空間計算量が $O(n^3)$ であるのに対し，$O(n)$ 個の自由変数のみを含む環境では空間計算量が $O(n^2)$ と異なる．もちろん「Clinger の参照実装」では $O(n^2)$ である．Clinger の漸近的空間計算量の定義には「どんなプログラムでも」を含むため，このようにプログラムを変化させる方向に空間消費関数への引数を変化させた場合も対処できる必要がある．

この点，Baker の実装手法をもとにした Chicken Scheme コンパイラ[3] などでは，クロージャ変換の際にコードで使われる自由変数についてのみの環境を用いているため問題ない．一方で，論文上の抽象機械[6),12] などでは，環境がそのまま継続のクロージャにも使われているため，そのままでは Clinger の形式的定義を満たさないことになる．なお，文献 6) の CPS 変換あるいは A 正規変換後のための抽象機械では，本章の最初で議論した「継続を生成するのは末尾でない呼び出しのときのみ」というものになっていて興味深い．

そのほかにもコンパイラの場合は，最適化[12] などが，Clinger の形式的定義とどのような関係にあるのかさらに検討が必要である．これは今後の課題としておく．場合によっては，CPS 変換の環境サイズ増加が許されるように形式的定義を変更したほうがよいといった結論も考えられる．

なお，抽象機械から環境（E）をなくしてしまえば，つまり（コード中に）代入してしまえば（たとえば文献 8)，16)，24)）シンプルになるだけでなく，代入先の変数にしか値が残らず，環境を限定したのと同じ効果があるといえる．ただし，この場合は，コードにプロ

© 2010 Information Processing Society of Japan

グラム（のほぼコピー）が含まれてしまうため，空間消費量として考えるときに，環境のみ考慮するよりも大きくなる場合がある．

9. 性能評価

本章では，実行性能の面から提案手法の評価を行う．空間性能は特に評価していないが，PTR が実装されていない場合にスタックあふれにより動作しないプログラムがあることは確認できる．また，実装が困難ではないという点の評価は図 3 の例などにより読者に委ねたい．

表 1 に評価環境をまとめた．それぞれの環境において次の Gabriel ベンチマーク[7] により測定を行った．

- boyer：Bob Boyer によって作られた定理証明チェッカを使ったベンチマークプログラム．
- fft：高速フーリエ変換を行う．多くの代入を行うプログラムである．
- tak：竹内により作られた再帰呼び出しを繰り返すプログラムである．
- traverse：木構造を作り，それをトラバースするプログラムである．
- cpstak：プログラム tak を，CPS で書いたプログラムである．
- puzzle：全探索でパズルを解くプログラムである．各手から次の手を探すループからの脱出に `call/cc` の一級継続を用いた非局所脱出を用いている．

また，次のプログラムも用いている．

- fib：フィボナッチ数列の n 番目の数を求めるプログラム．フィボナッチ数列の定義を再帰呼び出しで書いたものを使用しているため，n の値が大きくなると計算時間が急速に増大する．本研究では 30 番目のフィボナッチ数を求めるプログラムを使用した．
- fib-cps：フィボナッチ数列の n 番目の数を求めるプログラム．上のプログラム fib を CPS で書いている．プログラム fib と同様に n の値を 30 としている．

表 1　評価環境
Table 1　Specifications of evaluation platforms.

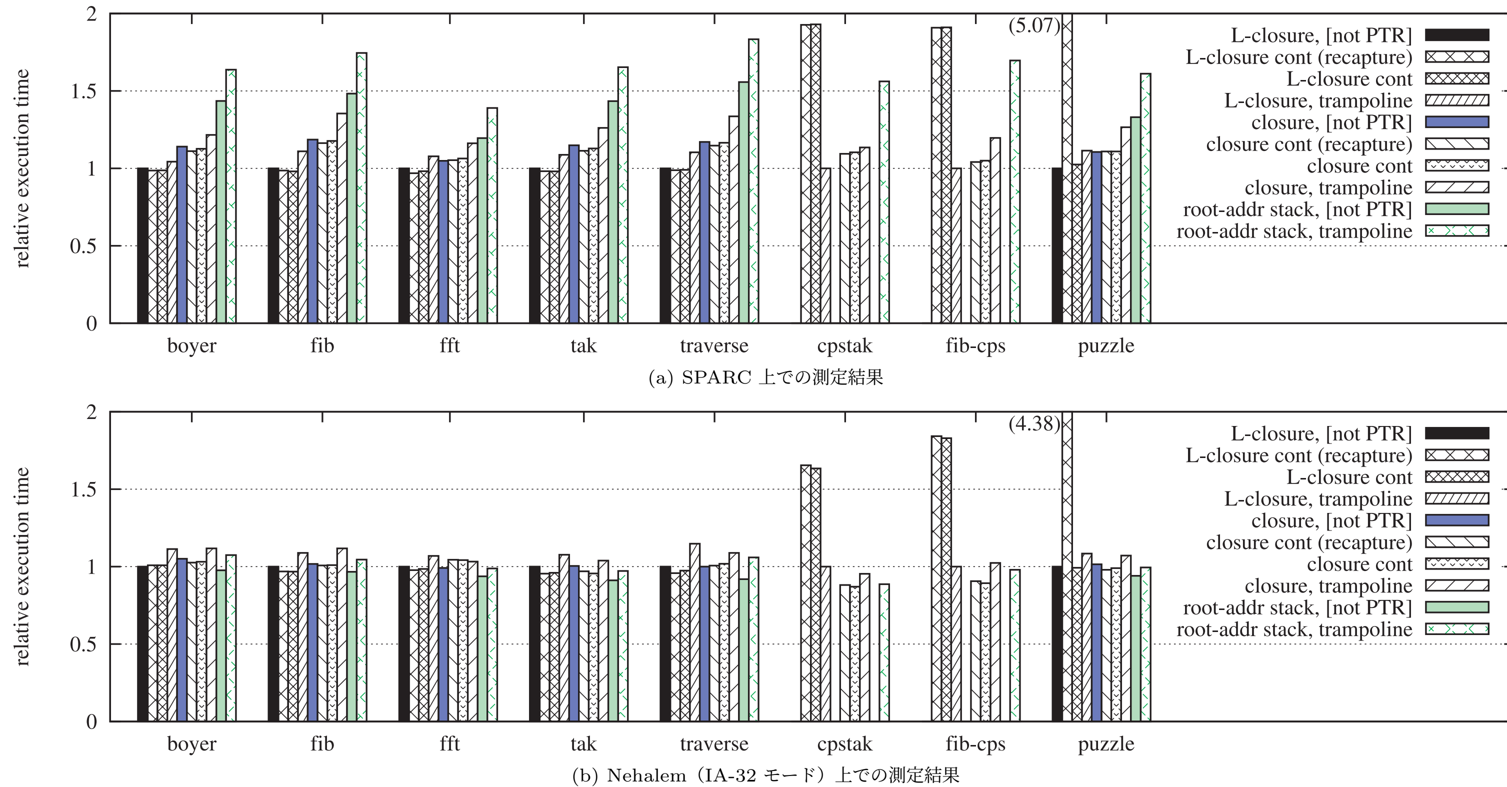
	Nehalem サーバ（Intel）	UltraSPARC T2 Plus サーバ
プロセッサ	Xeon X5570 2.93 GHz Quad-Core × 2 Hyper-threading 無効（計 8 コア）	UltraSPARC T2 Plus 1.4 GHz 8-core × 2 コアあたり 8 スレッド（計 128 スレッド）
キャッシュ	L1D: 32 KB 64B-line 8way, L2: 256 KB, L3: 8,192 KB 4 KB pages, 4-way associative, 64-entry DTLB	L1D: 8 KB/core, L2: 4 MB shared 16-way 64 B-line 128FA DTLB
メモリ	24 GB	24 GB
OS	Linux 2.6.9 (64 bit)	SunOS 5.10 (64 bit)
コンパイラ	XC-cube（IA-32 GCC 3.4.6 ベース）-O2	XC-cube（SPARC 32 bit GCC 3.4.6 ベース）-O2

比較した Scheme インタプリタでの共通事項として，ごみ集めには，標準的な Cheney のコピー GC を，2 つの 50 MB の空間からなるヒープについて用いた．また，提案手法では一定サイズ以下の C スタックを利用するとしているが，Scheme 手続きまたは特殊フォームの解釈実行を開始する C の関数呼び出しのネストを 1,500 回に制限し，それ以上でスタックあふれの防止を働かせた．これらの関数を呼び出さずに C の関数呼び出しが数回以上ネストすることはなく，`alloca` などは用いないため，スタックサイズはある定数以下に抑えられる．

図 5 に測定結果を示す．縦軸は L-closure により GC を実装した処理系の性能を 1 とした相対実行時間を示す．ただし，CPS で書かれたプログラムは PTR でない処理系では，スタックあふれにより動作しないため，cpstak，fib-cps についてはトランポリンで PTR を実現した処理系を 1 とした．図 5 で比較している処理系は以下のとおりである．

- L-closure, [not PTR]：L-closure をごみ集めの実装に用いているが，一級継続を実装せず PTR を実装していないもの．cpstak，fib-cps を除き，実行時間の基準としている．PTR を実装していないため，cpstak，fib-cps はスタックあふれにより動作しない．
- L-closure cont (recapture)：L-closure による一級継続を実装し，PTR を実装したもの．`call/cc` の際に実行スタックを空にしないため，実行スタック中の同じ部分継続を再キャプチャする可能性があるが，実行スタックで動作する分，高速に動作する可能性もある．
- L-closure cont：L-closure による一級継続を実装し，PTR を実装したもの．`call/cc` の際に実行スタックを空にし，共有を促進する．
- L-closure, trampoline：L-closure をごみ集めの実装に用いているが，一級継続を実装せず，トランポリンにより PTR を実装したもの．cpstak，fib-cps では実行時間の基準とした．
- closure ⋯：拡張 C 言語 XC-cube において，L-closure の代わりに，呼び出しコストもそれ以外のコストと同様に考慮した，よりマイルドな “closure” を用いたもの．
- root-addr stack, [not PTR]：ルートのアドレスのスタックを用いて GC のルートスキャンを実現する JAKLD/C のもとの処理系．PTR は実装されていないため，cpstak，fib-cps はスタックあふれにより動作しない．

© 2010 Information Processing Society of Japan



(a) SPARC 上での測定結果

(b) Nehalem（IA-32 モード）上での測定結果

図 5　真の末尾呼び出し未実装で GC 実装済み（L-closure）の処理系に対する相対実行時間．ただし，cpstak，fib-cps についてはトランポリンで真の末尾呼び出し実装したものを基準とした

Fig. 5　Execution times relative to the implementation supporting GC with L-closures but not supporting proper tail recursion, except for cpstak and fib-cps where times are normalized to those of the implementation supporting proper tail recursion with trampolines.

- root-addr stack, trampoline：JAKLD/C のもとの処理系にトランポリンにより PTR を実装したもの．

このうち提案手法を最も代表するものは L-closure cont である．後述するように，この処理系が多くの環境，ベンチマークで良い実行性能を達成している．ただし，CPS で書かれたプログラムに対しては真の末尾再帰は実現されているものの実行性能に問題があった．以下では，先に GC のルートスキャンの実装に L-closure を使った点について考察した後，PTR に関する考察を続ける．

ルートアドレスのスタックを用いた JAKLD/C のもとの処理系では，L-closure をごみ集めのルートスキャンに適用した場合と比較し，SPARC において実行時間がかなり長くなる傾向が見られた．これは変数のアドレスをとることによってレジスタ割当てが阻害された影響と考えられる．つまり，SPARC については，L-closure の利用により期待された効果が得られている．一方，Intel においては L-closure や closure による GC をしたときの性

© 2010 Information Processing Society of Japan

能は，ルートアドレスのスタックを用いた JAKLD/C のもとの処理系よりも悪い場合が多い．これは，IA-32 モードではレジスタが少なく，レジスタ割当ての効果が得られにくい点が理由として考えられるが，それ以外に JAKLD では小さな関数が多いため callee-save レジスタを使った場合にわざわざ callee で保存しただけの効果が得られず逆効果になる点が考えられる．

PTR 実現手法としての提案手法の有効性を特に示しているのは，boyer，fib，fft，tak，traverse のプログラムを Intel と SPARC で実行した結果である．ここでは，トランポリンによる実装手法よりも，一級継続を用いた提案手法のほうが実行時間が短縮されている傾向がある．これらのプログラムでは，実行スタックが基準以上に延びることはなく，提案手法において実際には一級継続は生成されない．トランポリンを用いた場合，トランポリンに渡す `Call` オブジェクトの生成の時間的オーバヘッドと空間的オーバヘッド（つまり GC 時間の増大），ならびにトランポリン処理のオーバヘッドが存在することになる．また，XC-cube の L-closure を用いたほうが XC-cube の closure を用いた場合よりも性能が良い．

cpstak，fib-cps といった CPS 変換を用いるプログラムは，Intel，SPARC ともに L-closure による一級継続を用いた実装手法による実行時間が，トランポリンを用いた実装手法による実行時間より増加している．これは，CPS で書かれたプログラムは末尾呼び出しのみを行うことになり，リターンすることなく C のスタックが延び続けるため，実際に L-closure が呼び出され，一級継続が生成されたためである．特に，L-closure は実行スタックに散らばった callee-save レジスタの値を集めてきて，L-closure 内での変数アクセスを行うため，呼び出し時には実行スタックの走査をともなう．このため，大きなコストを払うことになった．一方，XC-cube の L-closure の代わりに（呼び出しコストにも配慮した）closure を用いた場合は，提案手法のほうがトランポリンよりも低いコストで済んでいる．

以上の点からは，末尾呼び出しをスタックの消費が著しいほど多用する継続渡しスタイルなどで書かれたプログラムに対しては，一級継続を利用した提案手法を closure で用いるのが望ましく，末尾呼び出しがあっても一定サイズ以下の実行スタックで済ませられるような（通常，そうであることが多い）プログラムに対しては一級継続を利用した提案手法を L-closure で用いるのが望ましいことが分かる．あらかじめプログラムの性質が分からない場合や，実行中にフェーズによって動的に性質が変わる場合は，どちらのモードでも実行できる処理系としておき，PTR のための空間効率の良い一級継続の生成が発生してしまった後にはしばらくの間は closure を用いて実行するといったフィードバック制御も考えられる．

puzzle のプログラムは Intel，SPARC ともに `call/cc` 時に実行スタック中の部分継続を再キャプチャする実装手法による実行時間が突出している．実際，puzzle は全探索中に継続呼び出しによる非局所脱出を可能とするため `call/cc` により継続を多数キャプチャする．効率の良い一級継続の実装は今後の課題とするが，この結果からは，ひとまず文献 5) で述べられているとおり，`call/cc` によるキャプチャ時にはスタックをリセットし，一級継続の共有を促進しておくのがよいといえる．

10. 関連研究

10.1 スタックのごみ集めによる真の末尾再帰

真の末尾再帰をスタックのごみ集めによって実現する手法が Baker や Kelsey によって提案されている[2),11)]．CPS 変換以外での両者の違いとしては，Baker の方式では，C スタックを用いることができ，これは C スタックを主にヒープとしても利用していることがあげられる．

Kelsey の手法は，我々の手法と同様にスタックはヒープとは別扱いであり，CPS 変換をせずスタック上の継続を直接扱う．そのため，スタックを C 言語のデータとして実装するバイトコードインタプリタなどの処理系が対象であり，我々の手法が対象とするような C 言語の実行スタックを Scheme の実行スタックとして利用するインタプリタには不正当な方法を使用しない限り適用できない手法である．

10.2 一級継続の実装手法

スタックベースの処理系における一級継続の実装には様々な手法が提案されている[5),9),14),19)]．インクリメンタル・スタック/ヒープ法[5)] は，効率の良い手法として知られている．この手法では，ヒープへ退避させた継続を呼び出す時点では，フレームを 1 つだけコピーしてスタックへ戻す．その先のフレームは必要に応じて戻す．一方，継続のキャプチャの際は，スタック上のフレームすべてをヒープへ退避させた後，スタックを空にする．以上の動作により，フレームの共有が可能となり，フレームのコピー量を削減できる．ただし，この手法を実装するには，フレームへのアクセスが許されており，また，フレームをリロケータブルに配置できなければならない．したがって，C 言語のスタックを実行スタックとして利用する処理系には適用できない．インクリメンタル・スタック/ヒープ法のバリエーションとして複数のスタックを用いる手法もある[9)]．

Scheme の継続が C 言語の継続である処理系においては，フレームを認識することなくスタックの内容全体をヒープへ退避させることでキャプチャを実装し，退避させた内容全体をスタックの同じ位置へ戻すことで呼び出しを実装する方法が一般的である．この方法は

© 2010 Information Processing Society of Japan

コピー量が多く，またフレームが共有されないのでメモリ効率が非常に悪い．そこで，フレームのコピーを遅延させせることでフレームが共有できる機会を増やす手法[19] が提案されている．

本論文で提案する手法においては，継続は C 言語のデータとして取り出すことができるため，インクリメンタル・スタック/ヒープ法のようなフレームに対する操作の必要要件が厳しい実装手法であっても適用することができる．ただし，真の末尾再帰を実現するためには，適用する一級継続の空間効率について慎重に考える必要がある．

10.3　言語処理系内における継続の利用

継続は言語処理系内部の処理に活用されることがある．Gauche では，スタックオーバフロー時に一級継続をキャプチャしてスタックの内容をヒープへ退避させることで（事実上の）スタックを拡張する．同様の手法をマルチスレッドのスタック管理に用いる考えもある[9]．

10.4　L-closure 以外の方法

L-closure を使わなくても，C 関数からリターンして変数にアクセスする方法がある．実際，L-closure そのものも，その変換ベースの実装では，リターンしてアクセスする方式を用いている[10]．リターンして継続を生成しているといえる方式には，文献 22) などがある．トランポリンもリターン方式の一種といえるかもしれない．また，例外ハンドラを用いるものには文献 14)，15) がある．

11．まとめと今後の課題

本論文では，真の末尾再帰の実現のため，「空間効率の良い一級継続を生成してすぐに呼び出す」というアイディアと，これを何らかの基準で実行スタックがあふれそうになったら行い，実行スタックの利用をリセットすることを提案した．実行スタックを一定の大きさ以下で用いるため，定数差の違いを無視できる，Clinger の真の末尾再帰の形式的な定義を満たしている．また，提案する実装手法の詳細を示すとともに，性能評価を行った．トランポリンを用いた実装手法と比較すると，プログラムごとに拡張 C 言語 XC-cube の L-closure を用いるか，closure を用いるかでより高い性能が得られる．ただし，それぞれの実装は得意とするプログラムが異なるため，動的に切り替えるにはどうすべきかあるいはどう評価すべきかという課題が得られた．

提案方式は CPS 変換などはともなわないため，直接的な実装が容易に行え，保守性などが良い．また同時に，スタックあふれの防止策や一級継続の実装にもなっている．L-closure の応用としてみた場合は，インタプリタにおいてもごみ集めを実現できること，また，ある程度の規模の処理系に実際に適用できたことを示した．同時に，L-closure によって一級継続が実現できることや，それを利用して真の末尾再帰が実現できることも示した．

今後の課題として，コンパイラ版の扱い（比較などを含む），形式的定義のより精密な扱い，その最適化との関係，空間効率の良い一級継続生成の別実現法，実行スタックを残しての一級継続の共有手法，他の処理系との比較などを進めていきたい．

謝辞　JAKLD の C 言語実装を行った加藤秀敏氏に深く感謝する．本研究の一部は，科学研究費基盤研究（B)「安全な計算状態操作機構の実用化」(21300008）の助成を得て行った．

参　考　文　献

1) Appel, A.W.: *Compiling with Continuations*, Cambridge University Press (1992).
2) Baker, H.G.: CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A., *ACM SIGPLAN Notices*, Vol.30, pp.17–20 (1995).
3) Chicken-hackers: Chicken Scheme. http://www.call-with-current-continuation.org/
4) Clinger, W.D.: Proper Tail Recursion and Space Efficiency, *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (*PLDI '98*), pp.174–185 (1998).
5) Clinger, W.D., Hartheimer, A.H. and Ost, E.M.: Implementation Strategies for First-Class Continuations, *Higher-Order and Symbolic Computation*, Vol.12, pp.7–45 (1999).
6) Flanagan, C., Sabry, A., Duba, B.F. and Felleisen, M.: The Essence of Compiling with Continuations, *Proc. ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (*PLDI '93*), pp.237–247 (1993).
7) Gabriel, R.P.: *Performance and Evaluation of Lisp Systems*, Massachusetts Institute of Technology, Cambridge, MA, USA (1985).
8) Griffin, T.G.: A Formulae-as-Types Notion of Control, *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL'90*), pp.47–58 (1990).
9) Hieb, R., Dybvig, R.K. and Bruggeman, C.: Representing Control in the Presence of First-Class Continuations, *Proc. SIGPLAN 1990 Conference on Programming Language Design and Implementation* (*PLDI '90*), pp.66–77 (1990).
10) Hiraishi, T., Yasugi, M. and Yuasa, T.: A Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol.2, pp.262–279 (2006). (*IPSJ Trans. Programming*, Vol.47, No.SIG 6(PRO 29), pp.50–67).
11) Kelsey, R.A.: Tail-Recursive Stack Disciplines for an Interpreter, Technical Report NU-CCS-93-03, College of Computer Science, Northeastern University (1993).

© 2010 Information Processing Society of Japan

12) Kennedy, A.: Compiling with Continuations, Continued, *Proc. 12th ACM SIGPLAN International Conference on Functional Programming* (*ICFP 2007*), pp.177–190 (2007).
13) Nettles, S. and O'Toole, J.: Real-Time Replication Garbage Collection, *Proc. ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (*PLDI '93*), pp.217–226 (1993).
14) Pettyjohn, G., Clements, J., Marshall, J., Krishnamurthi, S. and Felleisen, M.: Continuations from Generalized Stack Inspection, *Proc. 10th ACM SIGPLAN International Conference on Functional Programming* (*ICFP 2005*), pp.216–227 (2005).
15) Sekiguchi, T., Sakamoto, T. and Yonezawa, A.: Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling, *Advances in Exception Handling Techniques* (*the book grow out of a ECOOP 2000 workshop*), *LNCS* 2022, pp.217–233, Springer (2001).
16) Sperber, M., Dybvig, R.K., Flatt, M., Straaten, A.V., Findler, R. and Matthews, J.: Revised^6 Report on the Algorithmic Language Scheme, *Journal of Functional Programming*, Vol.19, pp.1–301 (2009).
17) Steel Jr., G.L.: RABBIT: A Compiler for SCHEME, Technical Report AITR-474, Massachusetts Institute of Technology, Cambridge, MA, USA (1978).
18) Tarditi, D., Lee, P. and Acharya, A.: No Assembly Required: Compiling Standard ML to C, *ACM Letters on Programming Languages and Systems*, Vol.1, pp.161–177 (1992).
19) Ugawa, T., Minagawa, N., Komiya, T., Yasugi, M. and Yuasa, T.: Lazy Stack Copying and Stack Copy Sharing for the Efficient Implementation of Continuations, *The 1st Asian Symposium on Programming Languages and Systems* (*APLAS'03*), *LNCS* 2895, pp.410–426 (2003).
20) Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proc. 15th International Conference on Compiler Construction* (*CC2006*), LNCS 3923, pp.170–184 (2006).
21) 加藤秀敏：ごみ集めアルゴリズム実験用のコンパクトな Lisp 処理系，京都大学工学部情報学科卒業論文 (2007).
22) 八杉昌宏，馬谷誠二，鎌田十三郎，田畑悠介，伊藤智一，小宮常康，湯淺太一：オブジェクト指向並列言語 OPA のためのコード生成手法，情報処理学会論文誌：プログラミング，Vol.42, No.SIG11 (PRO12), pp.1–13 (2001).
23) 八杉昌宏，平石　拓，篠原丈成，湯淺太一：L-Closure：高性能・高信頼プログラミング言語の実装向け言語機構，情報処理学会論文誌：プログラミング，Vol.49, No.SIG1 (PRO35), pp.63–83 (2008).
24) 西崎真也，植田友幸：ファーストクラス継続をもつ仮想機械，日本ソフトウェア科学会第 20 回大会論文集 (2003).
25) 湯淺太一：Java アプリケーション組込み用の Lisp ドライバ，情報処理学会論文誌：プログラミング，Vol.44, No.SIG4 (PRO17), pp.1–16 (2003).

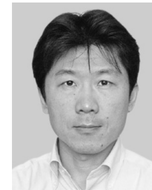
(平成 22 年 5 月 14 日受付)
(平成 22 年 8 月 27 日採録)

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．1993～1995 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 年同大学助教授．2007 年より同大学准教授．博士（理学）．1998～2001 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．平成 21 年度情報処理学会論文誌プログラミング優秀論文賞受賞．

小島　啓史

1984 年生．2008 年大阪大学基礎工学部情報科学科卒業．2010 年京都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修了．同年レッドハット株式会社入社．プログラミング言語，言語処理系等に興味を持つ．



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 年同大学院情報学研究科通信情報システム専攻助手．2003 年豊橋技術科学大学情報工学系講師．2007 年 1 月電気通信大学大学院情報システム学研究科助教授．2007 年 4 月より同研究科准教授．博士（工学）．プログラミング言語と言語処理系に興味を持つ．平成 8 年度情報処理学会論文賞受賞．

© 2010 Information Processing Society of Japan

平石　　拓（正会員）

1981 年生．2003 年京都大学工学部情報学科卒業．2005 年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修了．2008 年同専攻博士課程修了．2007〜2008 年日本学術振興会特別研究員．同年より京都大学学術情報メディアセンタースーパーコンピューティング研究分野助教．博士（情報学）．プログラミング言語，並列計算に興味を持つ．日本ソフトウェア科学会会員．

馬谷　誠二（正会員）

1974 年生．1999 年京都大学工学部情報学科卒業．2001 年同大学大学院情報学研究科修士課程修了．2004 年同大学院情報学研究科博士後期課程修了．同年同大学院情報学研究科産学官連携研究員．2005 年同研究科助手．2007 年より同研究科助教．博士（情報学）．プログラミング言語，並列/分散処理に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

湯淺　太一（フェロー）

1952 年神戸生れ．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．記号処理およびプログラミング言語処理系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．

© 2010 Information Processing Society of Japan