リターン・バリア

湯 淺 太 一† 中川 雄一郎†
小 宮 常 康† 八 杉 昌 宏†

Lisp などの大多数のリスト処理システムでは，不用セルを回収するためにごみ集め（GC）が行われる．一般的に採用されている GC は，ごみ集めの間プログラムの実行が中断されるので実時間処理には適さない．この問題を解決するために，ごみ集めの一連の処理を小さな部分処理に細分化し，プログラムの実行と並行してごみ集め処理を少しずつ進行させる実時間方式の GC が提案されている．代表的な実時間 GC であるスナップショット GC は，スタックなどのルート領域から直接指されているセルを GC 開始時にすべてマークしておかなければならない．この間の実行停止時間は，ルート領域の大きさによっては，無視できなくなる．そこで，関数からのリターン時にマーク漏れがないようにチェックすることで，スタックから直接指されているセルを関数フレーム単位でマークする方法を提案する．スタック上のルート領域をフレーム単位でマークしていき，ある関数からリターンする際に次の関数フレームがマークされているかどうかをチェックし，マークされていなければその関数フレームをマークしてからリターンする．これをリターン・バリアと呼ぶことにする．ルート領域のマークが終了したら，従来のスナップショット GC と同様に残りのセルをマークする．本論文では，Common Lisp 処理系 KCL（Kyoto Common Lisp）上でリターン・バリアを実装し，GC による実行停止時間について，従来のスナップショット GC と比較評価および検討を行った．

Return Barrier

Taiichi Yuasa,† Yuichiro Nakagawa,† Tsuneyasu Komiya†
and Masahiro Yasugi†

Garbage collection (GC) is the most popular method in list processing systems such as Lisp to reclaim discarded cells. GC periodically suspends the execution of the main list processing program. In order to avoid this problem, realtime GC which runs in parallel with the main program so that the time for each list processing primitive is bounded by some small constant has been proposed. The snapshot GC, which is one of the most popular realtime GC methods, has to mark all cells directly pointed to from the root area at the beginning of a GC process. The suspension of the main program by this root marking cannot be ignored when the root area is large. This paper proposes “return barrier” in order to divide the process of root marking into small chunks and to reduce the suspension time of the main program. The root area on the stack is marked frame by frame each time a new cell is requierd. When a function returns, the garbage collector checks if cells pointed to from the frame of the caller function have been already marked, and marks them if not. After marking all cells directly pointed to from the root area, other cells are marked as in the original snapshot GC. In this paper, we implemented the snapshot GC equipped with the return barrier in KCL (Kyoto Common Lisp). We compare and discuss the suspension times of this GC and the original snapshot GC.

1. 序　　論

Lisp などの大多数のリスト処理システムでは，リストや数値，記号などのデータはセルと呼ばれる記憶単位で表現される．このようなシステムでは，新しいデータが生じるたびにそのデータを表現するために新たにセルを用意するが，一度使用されたセルのうち，もはやどこからも参照されていないものが現れることがある．これを不用セルと呼ぶ．セルの総数には限りがあるので，利用可能なセル（フリー・セル）が不足してきた場合，不用セルを回収して再利用する必要がある．通常，不用セルを回収するためにはごみ集め（garbage collection）と呼ばれる手法が用いられる．レジスタやスタックなど，システムがいつでも参照す

† 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

ることのできるデータ領域（ルート集合）から，ポインタで直接あるいは間接に指されているセルはプログラムの実行中に使用される可能性がある．そこで，ごみ集めでは，ルート集合から参照されているすべてのセルをたどり，参照されていないセルをフリー・セルとして回収する．

現在最も一般的に採用されているごみ集めは，実行中のプログラムを一時中断してその間にごみ集め処理を行い，その後にプログラムの実行を再開する一括方式のごみ集めである．大規模なシステムの場合，ごみ集め処理には数十ミリ秒から数秒を要する．そのため一括方式のごみ集めを行うリスト処理システムは実時間処理には適さない．一方，人工知能やロボット制御などの分野では実用的なアプリケーションを実行するために実時間リスト処理システムを必要としている．実時間リスト処理システムを実現するためにはプログラムの中断をともなわないようなごみ集め，すなわち実時間ごみ集めの手法が不可欠となる．

実時間ごみ集めはごみ集めの一連の処理を小さな部分処理に細分化し，プログラムの実行と並行してごみ集め処理を少しずつ進行させる手法である．ただし実時間ごみ集めではごみ集め中にもプログラムの実行が継続するため，ごみ集め処理中のリスト構造の変化に対応する必要がある．

これまでに提案された実時間ごみ集めアルゴリズムは，リスト構造の変化への対処方法によって特徴付けられる．その中でも湯淺の考案した実時間ごみ集めであるスナップショットごみ集めアルゴリズム[3),7),8),10)]は，セルからの参照が変更される場合にのみ変更前の参照を保存する方式（この方式はライトバリアと呼ばれている）をとっているので，他の実時間ごみ集めアルゴリズムに比べて

- 専用のハードウェアを用いなくてもプログラムの実行効率の低下が少ない，
- 従来の一括方式のごみ集めを採用している処理系への応用が容易である，

という利点を持っており，汎用計算機上のリスト処理システムに適している．

スナップショットごみ集めでは，ごみ集め開始時にルート集合から直接または間接に参照できるセルは，たとえ不用になってもそのごみ集め中には回収されない．また，ごみ集め中に新しく割り当てられたセルも，たとえ不用セルになっても，そのごみ集め中には回収されない．すなわち，ごみ集め開始時にすでに不用であったセルのみを回収する．あたかも，ごみ集め開始時にデータ領域の写真（スナップショット）を撮っておき，その写真のとおりに不用セルを回収するのでその名が付けられた．

スナップショットごみ集めは，マーク&スイープ法を基礎としている．マーク&スイープ法とは，ごみ集め処理をマーク操作とスイープ操作に分けて行う方法で，スナップショットごみ集めでは，各操作を部分処理に細分化し，少しずつ処理することによって実時間性を保証する．各操作は以下のとおりである．

- マーク操作
  ルート集合から直接あるいは間接に参照されているすべてのセルに印を付ける．
- スイープ操作
  印の付いていないセルをフリー・セルとして回収する．

しかし，ルート集合は頻繁に書き換えられるので，スナップショットごみ集めでは，マーク操作を行う前にルート集合のスナップショットを撮る．すなわち，ルート集合から直接指されているすべてのセルに対して一括して印を付ける．この処理をルート挿入と呼ぶ．ルート集合の大部分を占めるスタックの大きさは実行時に変動するため，ルート挿入によるプログラムの停止時間に上限はなく，実時間性が損なわれる可能性がある．

そこで，本論文ではルート集合，特にスタックを分割して挿入するためのリターン・バリアと呼ばれる手法を提案する．リターン・バリアは，

- プログラムは関数の呼び出しが入れ子状態になって実行される，
- 関数が呼び出されると，その関数の実行に必要なデータ領域（関数フレーム）がスタックに積まれ，関数からリターンするとそのフレームは破棄される，
- 関数の実行中は，その関数のフレーム内にしか読み書きを行わない，

というスタックの性質に注目し，スタックをフレーム単位に分割してルート挿入する手法である．この手法を KCL 上に実装し，ごみ集め処理による実行停止時間について，リターン・バリア付きスナップショットごみ集めと従来のスナップショットごみ集めで比較評価および検討を行う．

2. ルート挿入の分割処理

従来のスナップショットごみ集めアルゴリズムでは，ルート挿入時には mutator が必ず停止される．このルート挿入時の停止時間は，ルート集合の大きさに依存する．ルート集合とは，mutator の実行時に用いら

れるスタックとそれ以外の固定領域を含めたものである．固定領域には，レジスタや，システムの大域変数（たとえば，シンボル表へのポインタを保持する変数）などが含まれる．固定領域の大きさは小さいので，これらを一括してルート挿入してもプログラムの実時間性を損なうことはない．一方，ルート集合の大部分を占めるスタックの大きさは実行時に変動するので，スタックのルート挿入による停止時間に対する上限は定まらず，実時間性が損なわれる可能性がある．

しかし，スタックは LIFO（Last-In, First-Out）に従った読み書きがなされるために全領域がランダムに読み書きされることはない．すなわち，mutator がリスト処理中にスタックを読み書きする前にその部分のルート挿入を行えば，スタックを分割してルート挿入することが可能となり，ルート挿入のための 1 回のプログラム中断時間を減少させプログラムの実時間性を保証することができる．

2.1　分割処理における問題点

ルート挿入を分割して行うと，分割されたルート挿入処理の間にプログラムが実行されるので，使用中のセルをマークし損なう可能性がある．**図 1** にその例を示す．

ルート挿入が分割して行われ，図 1 (a) の状態になったとする．セル A はマークされているが，セル B はまだ挿入されていないルートから指されているためマークされていない．この状態の後に，プログラムの実行が再開され，mutator によって図 1 (b) のようにポインタが書き換えられたと仮定する．セル B はスタックから指されているが，図 1 (a) でセル B を指していたルートが別のセル C を指すように変化している．この後に，ルート挿入を再開しても，セル B は挿入済みのルートからしか指されていないのでマークされない．したがって，使用中のセルであるにもかかわらず，ごみと見なされて回収されてしまう．

1 章で述べたライトバリアを用いれば，この問題は解決できる．しかし，頻繁に書き換えられるスタックに対してライトバリアを用いて書き込み保護をかけると，オーバヘッドが大きくなる．そこで，セルが書き換えられるたびにバリアをかけるのではなく，スタックに積まれる関数フレームの特性を利用して，フレーム単位で書き込み保護をかけることによりオーバヘッドを軽減し，プログラムの実時間性を保証する手法を提案する．

2.2　フ レ ー ム

プログラムは一般に関数の呼び出しが入れ子状態になって実行される．関数が呼び出されると，スタック

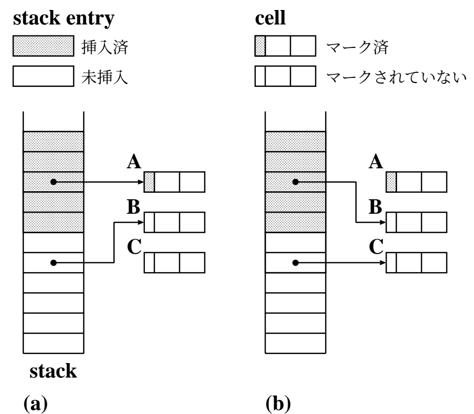

図 1　ルート挿入の分割処理における問題点
Fig. 1　Problem of incremental root insertion.

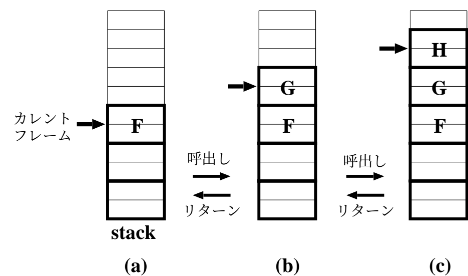

図 2　関数フレーム
Fig. 2　Function frames.

にはその関数のフレームが積まれる．フレームには，実引数，返り値，局所変数，リターン・アドレス（関数実行が終了したときに制御を戻す番地）など，その関数を実行するのに必要な情報が格納されている．実行中の関数のフレームをカレントフレームと呼ぶ．関数の実行が終了する（関数からリターンする）と，カレントフレームはスタックからポップされる．

図 2 にその様子を示す．図 2 (a) は関数 F が実行中であり，F のフレームがカレントフレームであることを表している．この F から別の関数 G が呼び出されると，G を実行するためのフレームがスタックに積まれた状態（図 2 (b)）になる．G の実行が終了すれば，G のフレームはポップされて呼び出し前の状態（図 2 (a)）に戻る．また，G の実行中にさらに他の関数 H が呼び出されたときは，H のためのフレームが積まれ（図 2 (c)），H の実行が終われば H のフレームはポップされる（図 2 (b)）．

このことより，

- 通常，スタックには複数個のフレームが積まれている，
- カレントフレームは，スタックの最上位に積まれている，

- mutator のスタックに対する読み書きは，カレントフレーム内に限られる，
- ルート挿入はごみ集め開始時のルート集合に対して行われるので，ごみ集め開始後にスタックに積まれたフレームは考慮しなくてよい，

ということが分かる．

これらの性質に注目し，リターン・バリア（return barrier）と呼ばれる手法を提案する．従来のスナップショットごみ集めが，ごみ集め処理が起動されると mutator を停止させ，スタックの要素すべてを一括でルート挿入するのに対して，リターン・バリアを使用すれば，ルート挿入をフレーム単位で分割して行い，mutator の停止時間を抑えることが可能である．

2.3 リターン・バリア

リターン・バリアとは，関数からのリターンが生じた際に，スタック内のマークされた部分とまだマークされていない部分の境目（以後，この境目をバリアと呼ぶ）に注目し，カレントフレームがバリアを越えて下位（ボトムの方向）に移ることを防ぐことで，まだ挿入されていないルートに対して書き込み保護をかける手法である．もし，カレントフレームがバリアを越えて下位に移動しようとした場合，mutator の実行を中断し，ルート挿入を進めることによって，バリアを下位に移動する．

図 3 にその様子を示す．図 3 (a.1) は，カレントフレームがルート挿入済みの領域に含まれている状態を表している．プログラムの実行によってスタックの状態が変化しても影響を受けるのはカレントフレーム内だけなので，2.1 節で述べたようなマーク漏れは生じない．しかし図 3 (a.2) のように，実行中の関数がリターンし，まだ挿入されていないルート領域にカレントフレームが含まれてしまうと，マーク漏れが生じる可能性がある．そこで，図 3 (b.1) のように，実行中の関数からリターンすると次に実行される関数のフレームがバリアを越えてしまう場合，図 3 (b.2) のように，プログラムの実行を一時中断してルート挿入を少しだけ実行し，挿入した分だけバリアを進める．この後でカレントフレームをポップしても，図 3 (b.3) のように，その後のカレントフレームはルート挿入済みの領域に含まれるので，マーク漏れが生じることはない．

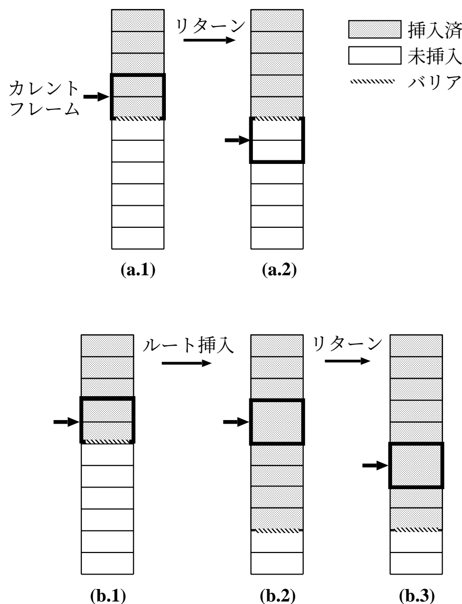

図 3 リターン・バリア
Fig. 3 Return barrier.

実際には，バリアがスタックの最下位に来るまで，collector は以下のようにスタックのルート挿入処理を行う．

- 前処理
  ごみ集めが起動されると，
  (1) その時点のカレントフレーム分だけスタックをルート挿入する．
  (2) そのフレーム内のリターン・アドレスをスタック以外のどこかに保存して（以後，「退避する」と記す），代わりにその場所へ，ルート挿入を少しだけ実行するためのコード（Return Barrier code，RBcode と略す）の番地を格納する．つまり，このコードがリターン・バリアの役割を果たす．
- 繰返し処理
  セルが要求されるたびに，一定個のフレーム分だけスタックをルート挿入して
  (1) 退避させていた古いリターン・アドレスを元の場所，すなわち RBcode の番地が格納されている場所に戻す．
  (2) 最後にルート挿入したフレームのリターン・アドレスを退避し，代わりにその場所へ，RBcode の番地を格納する．

もし，mutator の実行によってカレントフレームがバリアを越えてしまう場合は，自動的に RBcode に制御が移行する．RBcode は次の処理を行う．

(1) 退避させていたリターン・アドレス X を元の場所に戻す．
(2) 一定個のフレーム分だけスタックをルート挿入する．
(3) 最後にルート挿入したフレーム内のリターン・アドレスを退避させて，代わりにその場所へ，

RBcode の番地を格納する．

(4)　X にジャンプする．

今回提案するリターン・バリアは，関数を実行する際には，自動的に，つまり mutator の処理に変更を加えることなく，ルート挿入を実行するコード（RBcode）に制御を移行することができるため，プログラムの実行性能に負担をかけることなく実装することが可能である．

2.4　実装上の留意点

2.2 節で述べた関数フレームの用法は，最も基本的なものであり，次の 2 点を前提としている．

(1)　現在実行中の関数は，カレントフレームのみを参照する．

(2)　関数は必ず呼び出し元へ正常にリターンする．

実際のシステムでは，この 2 点が必ず満たされるとは限らない．前者は，局所関数をサポートする言語システムの場合は成り立たず，後者は，非局所的脱出をサポートする言語システムの場合は成り立たない．本節では，リターン・バリアの有効性を損なわずに，これらの言語機能を実装するための手法を述べる．

局所関数を含んだ次の簡単な Common Lisp [6] コードを考える．

```
(defun F (x)
  (labels ((G (y z) ... (G z x) ...))
    (G x x)))
```

関数 F の中で，局所関数 G が定義されており，G からは G 自身が再帰的に呼び出されている．再帰呼び出しの式に現れる変数 x は，F へのパラメータであり，F のフレーム中に存在する．したがって，G の実行中は，G のフレームだけでなく，F のフレームも参照することになる．この参照を可能にするために，G のフレームには F のフレームへの静的リンク（static link）を格納しておくのが一般的である．

局所関数の実行中は，静的リンクでたどることのできるフレームが，リターン・バリアより下位に位置する可能性がある（**図 4** 参照）．そこで，システムが局所関数のフレームを挿入するときには，そこから静的リンクでたどることのできるすべてのフレームを挿入しておく必要がある．そうしなければ，未挿入のフレームが参照される可能性が生じるからである．

局所関数フレームに対するこの特別な処理は，実時間性を損なうものではない．ある局所関数から静的リンクによってたどることのできるフレームの個数は，その局所関数定義の構文上の深さ（上の Common Lisp の例では，G の深さは 1，もし G の中でさらに局所関数が定義されていれば，その深さは 2）を超えないからである．

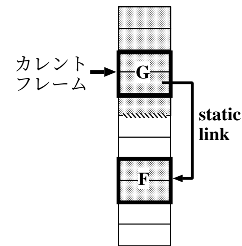

図 4　局所関数フレームの処理
Fig. 4　Processing local function frames.

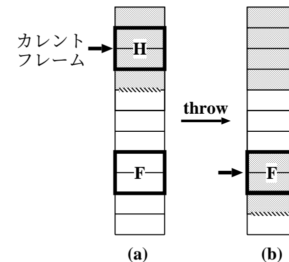

図 5　非局所的脱出の処理
Fig. 5　Processing non-local exit.

非局所的脱出とは，実行中の関数から，その呼び出し元にリターンするのではなく，ある条件を満たす関数へ制御を戻すものであり，その代表が，catch & throw 機構である．**図 5** にその例を示す．図 5 (a) において，関数 F が catcher を設定した後，関数 H の実行中に，その catcher に対応する throw が実行される場合を考える．これによって，通常の関数呼び出しとリターンのメカニズムはスキップされ，関数 H から一挙に関数 F へ制御が戻る（図 5 (b)）．

非局所的脱出によってリターン・バリアを越えてスタックが巻き戻された場合，新しいカレントフレーム（例では，F のフレーム）が挿入済みでなければならない．一般に，非局所的脱出による戻り先のフレーム（新しいカレントフレーム）は，実行時に検索される．catch & throw の場合であれば，throw に対して指定したものと同じタグを持つ catcher を設定したフレームを，スタックのトップからボトム方向へ検索する．この検索の際に，システムは，リターン・バリアを越えてスタックが巻き戻されるかどうかを判断できる．もしリターン・バリアを越える場合は，新しいカレントフレームに対して挿入を行えばよい．ここで重要なことは，throw の直前のカレントフレームと，戻り先の新しいカレントフレームとの間に位置するフレームは，挿入する必要がない点である．これらのフレームの個数は不定であり，もしこれらすべてを挿入すると，

実時間性を損なう可能性がある．しかし，これらのフレームはまったく参照されないので，挿入をしなくても問題は生じない．ただしこれによって，厳密にはスナップショット型とはいえなくなる．スキップされたフレームからのみ指されていたセルは，現在のごみ集め中に回収されてしまうからである．しかしこれはごみの回収時期が早まるだけのことであり，むしろ歓迎すべき特性である．

3. ごみ集め開始条件

実時間ごみ集めではごみ集め中にもセルの要求があり，フリー・セルが消費されていく．しかし不用セルの回収が始まるのは，処理がスイープ・フェイズに入ってからである．そのため，もしごみ集め開始時点で不用セル回収開始までのセル要求に十分応じられるだけのフリー・セルを確保していなければ，不用セルの回収が始まる前にフリー・セルが尽きてしまう．この状態を飢餓状態（starvation）と呼ぶ．システムがごみ集め中に飢餓状態に陥った場合，復旧措置としてはセル領域を拡張するか，あるいはごみ集め処理の未完了分を一括して行うしかないが，一般にはこのような措置はある程度の処理時間を必要とするため，プログラムの中断は避けられない．よって，ごみ集め処理は十分な数のフリー・セルが残されている状態で開始する必要がある．本章では，飢餓状態を回避するための，ごみ集め開始条件を検討する．次のシステムパラメータを使用する．

K_0：ルート挿入中のセル要求ごとにマークするスタック上のルート数

K_1：マーク・フェイズ中のセル要求ごとにマークするセル数

K_2：スイープ・フェイズ中のセル要求ごとにスイープするセル数

図 6 に型 q のフリー・セル数 $F_q(t)$ の変化を示す．t はプログラム実行開始時点からの通算セル要求回数を表す．以下，「プログラム実行開始から t 個目のセルが要求された時点」を単に「t の時点」と表現する．a の時点において型 q のフリー・セル数がある水準 M_q に達し，ごみ集めが開始され，b の時点においてマーク・フェイズからスイープ・フェイズへ移行し，c の時点で終了したとする．

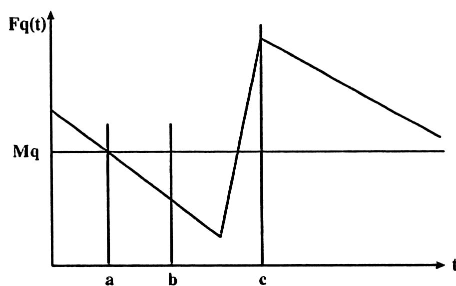

図 6　フリー・セル数の変化
Fig. 6　Number of free cells.

ごみ集め処理はまずルート挿入を分割して行う．ルート挿入中は不用セルの回収は行われないので，セルの消費にともない，フリー・セル数は減少し続ける．ごみ集め開始時点におけるスタックの大きさを $R(a)$ とすると，ルート挿入中のセル要求回数は

$$\lceil R(a)/K_0 \rceil$$

となる．

$t = a + \lceil R(a)/K_0 \rceil$ の時点でルート挿入は終了し，ごみ集め処理はマーク・フェイズに移行する．マーク・フェイズ中も不用セルの回収は行われないので，セルの消費にともない，フリー・セル数は減少し続ける．マーキング対象となるセルの数はごみ集め開始時点における総使用セル数 $A(a)$ なので，マーク・フェイズ中のセル要求回数は

$$\lceil A(a)/K_1 \rceil$$

となる．

$t = b = a + \lceil R(a)/K_0 \rceil + \lceil A(a)/K_1 \rceil$ の時点でマーク・フェイズは終了し，ごみ集め処理はスイープ・フェイズに移行する．このときスイープされるセルの数は全セル数 N である．しかし，移行してもすぐに型 q の不用セルが見つかるとは限らないので，スイープ・フェイズに移行してからもしばらくはフリー・セル数は減少し続ける．ごみ集め中に不用になったセルは次回のごみ集めで回収されるので，スイープ・フェイズで回収されるのはごみ集め開始時点の不用セルである．よってごみ集めによって回収される型 q の不用セルの個数は，型 q のセル数 N_q から $t = a$ における型 q の使用セル数 $A_q(a)$ およびフリー・セル数 $F_q(a)$ を引いた残り，すなわち

$$N_q - A_q(a) - F_q(a)$$

である．最悪の場合，これらの不用セルがセル用の領域の最後に集中し，他のすべてのセルがスイープされた後に回収されることになる．よって，1 回のセル要求ごとに K_2 個のセルをスイープするとすれば，スイープ・フェイズに移行してから型 q の不用セルが実際に回収され始めるまでに最大

$$\lceil \{N - (N_q - A_q(a) - F_q(a))\}/K_2 \rceil$$

回のセル要求があることになる．

ごみ集めを開始してから終了するまでの間に型 q のセルが要求される頻度を C_q で表すと，ごみ集めを開

始してから型 q の不用セルが回収され始めるまでの型 q の最大消費セル数は

$$C_q(\lceil R(a)/K_0\rceil + \lceil A(a)/K_1\rceil + \lceil\{N-(N_q-A_q(a)-F_q(a))\}/K_2\rceil)$$

となる．ここで，便宜上分数は割り切れるものと仮定すると，ごみ集め中に飢餓状態に陥らないためには

$$\begin{aligned}F_q(a) &\geq C_q(R(a)/K_0 + A(a)/K_1 + \{N-(N_q-A_q(a)-F_q(a))\}/K_2)\\ &\geq C_q(R(a)/K_0 + A(a)/K_1 + \{N-N_q+A_q(a)\}/K_2)(1-C_q/K_2)^{-1}\end{aligned}$$

でなくてはならない．したがって M_q として最適なセル数は

$$M_q = C_q(R(a)/K_0 + A(a)/K_1 + \{N-N_q+A_q(a)\}/K_2)(1-C_q/K_2)^{-1}$$

である．ここで $C_qA(a)$ は型 q の使用中セル数 $A_q(a)$ と等しいと仮定すると，

$$M_q = (C_qR(a)/K_0 + A_q(a)/K_1 + C_q(N-N_q+A_q(a))/K_2)(1-C_q/K_2)^{-1}$$

となる．

ごみ集め開始のタイミングを決定するためには，この条件式中の未決定の定数その他を具体化しなければならない．以下，その妥当な値を検討する．

まずセルの要求頻度 C_q を推定する．ここで，C_q は型 q の最大使用中セル数 A_{qmax} の最大全使用中セル数 A_{max} に対する割合で近似できると仮定する．各型のセルに対し適切な領域配分が行われれば，A_{max} および A_{qmax} はそれぞれ N および N_q とほぼ等しくなるので，C_q の値は

$$C_q = N_q/N$$

と推定される．

次に使用中のセル数 $A_q(a)$ を推定する．$A_q(a)$ はごみ集め開始時期などに依存するので実際には推定が困難である．そこで $A_q(a)$ の代わりにフリー・セル以外のセル数

$$U_q(a) = N_q - F_q(a)$$

を用いることにする．

スタックの大きさ $R(a)$ はプログラムに依存するので，推定することはまず不可能である．そこで，「ごみ集め開始時点における使用中のセルがすべてスタックに積まれている」という最悪の場合を想定し，

$$R(a) = A(a)$$

と仮定する．

これらの推定値を用いると，M_q は

$$M_q = N_q(1/K_0 + 1/K_1 + 1/K_2)(1+1/K_0+1/K_1)^{-1}$$

となり，N_q のみに依存する．たとえば，$K_0 = K_1 = K_2 = 20$ の場合では，

$$M_q = 0.1364N_q$$

すなわち，残りフリー・セル数がその型の総セル数の約14%になったところでごみ集めを開始すればよいことになる．これに対して，従来方式のスナップショットごみ集めでは，ルート挿入を一度に行うので，$K_0 = \infty$ とすると，$M_q = 0.0952N_q$ となる．ルート挿入を分割するために，この差の分だけ，ごみ集めを早く開始することになる．上記の計算の課程では，$R(a)$ を $A(a)$ で近似したために，ごみ集めの開始が必要以上に早くなる．それでも従来方式と 4 ポイント程度の差であり，許容範囲と考えてよいであろう．

4. KCL におけるリターン・バリアの実装

リターン・バリアを，KCL（Kyoto Common Lisp）[9] に実装した．まず KCL におけるルート集合について説明し，次に実装およびその際に生じた問題点とその解決方法について述べる．

4.1 KCL におけるルート集合

KCL におけるルート集合は，スタック（動的領域）とそれ以外の固定領域に分類できる．

固定領域は，セルへのポインタを 160 個格納した配列である．その要素の大半は，システム起動時に初期化された後は値が変わらない．これらの要素は，システムが常時必要とするデータを，GC によって回収されないように保護するために用いられている．動的に値が変わる要素は，18 個だけであり，システム内部における一時変数として使われている．GC 起動時に一括して挿入する必要がある要素は，この 18 個だけである．残りの要素については，他のルート挿入が終了した後に適宜分割してルート挿入を行っても問題は生じない．

KCL は次に示す複数のスタックを使用している．

- value スタック
  コンパイルされた関数の局所変数や一時変数を割り当てたり，引数や返り値の受渡しに使用する．
- bind スタック
  動的変数に対する浅い束縛（shallow binding）を実現するために使われるスタックであり，動的変数の名前と古い値のペアが格納されている．
- frame スタック
  `throw` や `return-from` などによる非局所的脱出を実現するための “catcher” を格納したスタック

である．各 catcher には，`catch` や `block` などによって脱出先が設定された時点における value スタックや bind スタックの先頭アドレスなどが記憶されており，これらの情報によって，脱出後の状態を復元する．

- invocation history スタック
  デバッグ用のスタックであり，関数呼び出しの履歴を格納している．
- C スタック
  KCL コンパイラは，Lisp で書かれたソースコードを C 言語に変換し，C コンパイラを用いてオブジェクトコードに変換する．KCL のカーネルも C 言語で書かれている．したがって，KCL の実行は，C スタック（C 言語の制御スタック）を使って制御されている．処理系の移植性を保証するために，C スタックを KCL が直接操作することはない．

以上の 5 本のスタックのうち，C スタックはルート挿入の対象とはならない．セルへのポインタが C スタックに積まれることはあるが，その場合は必ず value スタックにもポインタを積むことによって，C スタックを直接アクセスしなくてもごみ集めが行えるようになっている．

4.2 リターン・バリアの実装

KCL では，リターン・アドレスは C スタックに積まれるので，それを直接操作することはできない．そのため，2.3 節で述べたような，「リターン・アドレスを RBcode に書き換えることによって，まだマークされていない部分への書き込み保護をかける」という手法は実現できない．そこで，今回の実装では，各スタックに対して次のアルゴリズムを採用した．

- ごみ集めが起動されると，固定領域のうち，値が動的に変化するものだけをルート挿入し，次に，その時点のカレントフレームをルート挿入する．
- セルが要求されるたびに，各スタックを一定個ずつルート挿入していく．
- 実行中の関数からリターンしたら，次に実行される関数のフレームがすべてルート挿入されているかどうかをチェックし，まだルート挿入されていなければ，ルート挿入を行う．
- `catch` や `throw` などによる非局所的脱出が起きた場合は，脱出後のフレームがすべてルート挿入されているように，関数リターンと同様の処理を行う．

このアルゴリズムでは，実行中の関数からリターンする際に，リターン後のカレントフレームがルート挿入されているかどうかをチェックする．すなわち，ルート挿入時ではなく，関数の実行時にオーバヘッドがかかるので，2.3 節で述べたアルゴリズムより実行効率が悪くなる．

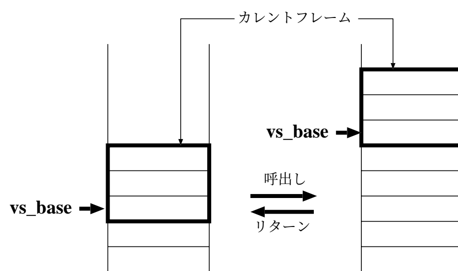

図 7 value スタック
Fig. 7 Value stack.

このアルゴリズムを実現するために，各スタックに対して，ルート挿入されている部分とされていない部分の境目を指すための変数 `barrier` を用意した．以下に各スタックに対するリターン・バリアの実装方法を述べる．

まず，value スタックに対する実装方法を述べる．value スタック上のカレントフレームは，ベースポインタ`vs_base` の指す位置からスタックのトップまでの間である（**図 7** 参照）．関数からリターンする際には，`vs_base` が下位方向（ボトムの方向）に移動する．その際に，`vs_base` が`barrier` を越えて移動しないかどうかをチェックする．もし `vs_base` が `barrier` よりも下位に移動した場合は，mutator の実行を中断して，次の処理をする．

- `vs_base` と `barrier` の間に存在するスタック要素が K_0 個以上の場合は，それらの要素をすべてルート挿入し，`barrier` を `vs_base` に移動する．
- K_0 個未満の場合は，K_0 個だけスタック要素をルート挿入し，`barrier` を K_0 個だけ下位に移動させる．

`vs_base` と `barrier` の間に存在するスタック要素が K_0 個より多い場合，ルート挿入しなければならないスタック要素の個数はプログラムに依存する．このため，実時間性を損なう可能性がある．しかし，この個数は，プログラム実行のある時点において参照可能な局所変数（関数のパラメータを含む）の個数を超えることはなく，極端なプログラムを除けば，その個数は比較的少ないと考えてよい．

bind スタックについては，実際には，リターン・バリアは必要でない．bind スタックがポップされるときに，動的変数の古い値はその変数を表す記号の値スロットに格納される．したがって，ライトバリアによっ

て，必ずマークされることが保証される．

invocation history スタックについても，リターン・バリアは必要でない．このスタックには，呼び出された関数へのポインタが積まれているが，そのポインタが別のポインタで置き換えられたり，あるいは他のルートに代入されることがないからである．

frame スタック上の catcher には，インタプリタが関数を実行する際の静的環境（lexical environment）が格納されている．非局所的脱出が起きた場合は，この静的環境をインタプリタが取り出して，実行を継続する．KCL の静的環境は cons セルで表現されているので，マーキングの対象となる．そこで，frame スタックを巻き戻す際に，戻り先の catcher が `barrier` より下に位置するかどうかをチェックする．もし下に位置すれば，その catcher をルート挿入すればよい．

5. 性能評価

リターン・バリアの性能評価のために，リターン・バリアを実装した KCL に対していくつかの評価実験を行い，挿入を一括して行う従来のスナップショット方式と比較検討した．ベンチマーク・プログラムとして，次の 2 つを用いた．

- fib
  再帰的に n 番目のフィボナッチ数を求める関数である．この関数を KCL のインタプリタで実行すると，インタプリタが環境を連想リストにするので，fib の呼び出しごとに 2 個の cons セルが消費され，これが不用セルになる．以下の測定結果では，$n = 27$ の場合を示す．$n > 27$ の場合は KCL がヒープを自動拡張するために，性能評価には適さない．$n < 27$ の場合の測定結果は，$n = 27$ の場合と同様の傾向だったので省略する．
- boyer [1]
  与えられた論理式に対して論理証明を行うプログラムである．大量のセルを生成し，それらに繰り返しアクセスするため，スタックの深さ，変動量ともに大きく，ごみ集めが頻繁に起動される．

テストに使用した計算機は，200 MHz の Pentium Pro を搭載した Solaris マシンで，メモリは 192 MB である．テスト実行中にヒープが自動拡張しないように，十分な大きさのヒープ領域として 1.5 MB を割り当てた．また，各テストの実行前にはごみ集めを強制的に行い，不用セルがすべて回収済みの状態でテストを行った．時間計測には，Intel 社の推奨する RDTSC（read time-stamp counter）命令を使った方法[4]を採用した．

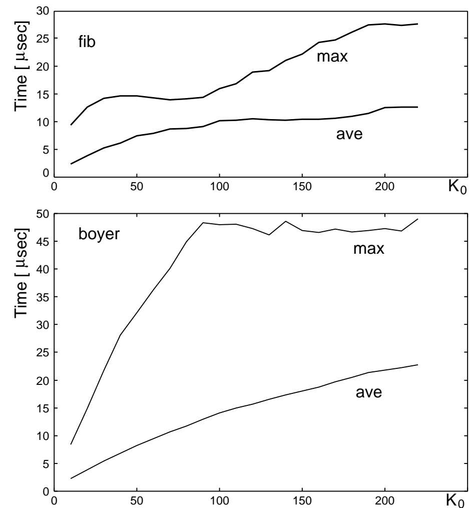

図 8 ルート挿入時の停止時間
Fig. 8 Suspension times during root insertion.

5.1 停止時間

リターン・バリア方式では，ルート集合を分割して挿入することにより，ごみ集め処理によるプログラムの停止時間を，一定時間に抑えることが可能である．K_0（ルート挿入中のセル要求ごとにマークするスタック上のルート数）を小さくすれば，この停止時間は短くなるはずである．そこで，K_0 の値によって，ルート挿入時の停止時間がどのように変動するかを計測した．**図 8** に，$K_0 = 10, 20, \ldots, 220$ とした場合の最大停止時間と平均停止時間を示す．

図から，平均停止時間は，K_0 の減少にともなって順調に減少することが分かる．最大停止時間も同じ傾向があるが，K_0 との相関は，平均停止時間ほど強くはない．また，各 K_0 に対する最大停止時間と平均停止時間には，大きな差がある．同じ K_0 に対しては，1 回の停止時間中に行うルート挿入処理のステップ数はほぼ同じである．したがってこの差は，キャッシュ効果が影響しているものと考えられる．

比較のために，従来のスナップショットごみ集めにおける，ルート挿入時間（つまり，一括ルート挿入にともなう停止時間）を計測した．**図 9** は，ごみ集め起動時点におけるスタックサイズと，ルート挿入時間の関係を表したものである．1 個の点が，1 回のごみ集めに相当する．ここで，スタックサイズとは，ルート挿入の対象となる KCL の 4 つのスタックのエントリ数の総和である．図から，ごみ集め開始時点におけるスタックサイズには，かなりの変動があることが分かる．boyer では，最大のスタックサイズは，最小のスタックサイズの 4 倍にもなる．

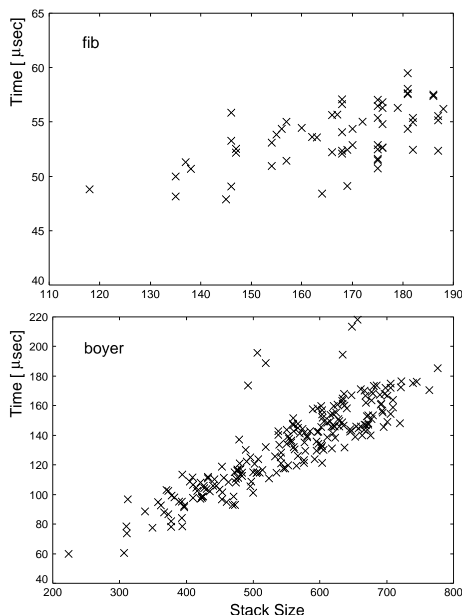

図 9　スタックサイズとルート挿入時間
Fig. 9　Stack sizes and root insersion times.

キャッシュの影響のためか，かなりの幅があるものの，ルート挿入時間はスタックサイズにほぼ比例していることが分かる．ルート挿入のための停止時間は，次のとおりである．

fib：　最大 58.3 μsec，平均 53.0 μsec

boyer：最大 218.5 μsec，平均 145.3 μsec

リターン・バリア方式の場合の停止時間は，K_0 の値に依存するが，今回計測した最小の $K_0 = 10$ を指定すれば，どちらのベンチマーク・テストでも，最大停止時間 10 μsec 以下，平均停止時間 5 μsec 以下を達成でき，ルート挿入時の停止時間が大幅に短縮されることが分かる．

5.2　リターン・バリアによる停止時間

今回提案するリターン・バリア方式では，リターン・バリアを越えてスタックが巻き戻るとき，mutator が停止し，新しいカレントフレーム上のルートを挿入する．この停止回数および最大停止時間を測定した．計測の結果，fib では，停止はまったく起きなかった．boyer についての計測結果を，**図 10** に示す．

リターン・バリア方式では，セルを消費するたびにリターン・バリアがスタックの下位（ボトム方向）に移動する．したがって，関数が連続してリターンし，その間にセルの消費がないときに，リターン・バリアによる停止が発生する可能性がある．しかしその場合でも，ルート挿入が終了していたり，リターン・バリアがボトム近くにすでに移動したりしていれば，停止は生じない．停止するかどうかはプログラムに大きく依存するし，同じプログラムでも，実行時の状態に影響される．fib の実行中は一度もリターン・バリアによる停止が生じなかったが，これは偶然であろう．このような場合は，設定したリターン・バリアはまったく使われなかったことになるが，リターン・バリアによる停止が発生するかどうかは，実行してみないと分からない．したがって，一種の「安全保険」と理解すべきであろう．

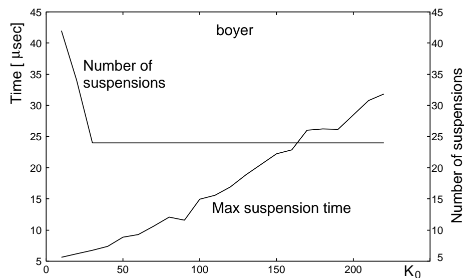

図 10　リターン・バリアによる停止回数と最大停止時間
Fig. 10　Suspension times caused by return barrier.

一方，boyer の場合は，リターン・バリアによる停止が生じている．$K_0 = 10$ のときが最も多く，42 回停止している．しかし，boyer 実行中に，250 回以上のごみ集めが起きている（後述の表 2 参照）．停止が生じるのは，ごみ集め 6 回に 1 回の割合であり，その可能性はきわめて小さい．また，停止時間はいずれも小さく抑えられていることが分かる．当然ながら，K_0 が小さくなるに従って最大停止時間も減少している．逆に停止回数は，K_0 が小さいほうが多くなる傾向にある．これは，リターン・バリアの降下する速度が，K_0 に比例するためである．

5.3　オーバヘッド

リターン・バリア方式によって生じる実行時オーバヘッドを調べるために，ベンチマーク・プログラムの実行に要した時間，実行中に生じたごみ集めの回数，および，ごみ集めごとに行うルート挿入の平均時間を求めた．結果を，**表 1** と**表 2** に示す．表には，リターン・バリア方式については，$K_0 = 10, 20, 40, 80, 160$ のときのデータを掲載する．ルート挿入を一括して行うスナップショット方式に関しては，従来方式の場合のデータに加え，オーバヘッドを考察するために「チェック追加」の場合のデータを掲載する．「チェック追加」については後で説明する．

リターン・バリア方式もスナップショット方式も，セルのマーキングとヒープのスイープは分割して行う．このために，実行結果は，K_0 以外の 2 つのシステム・

表 1 fib の実行結果
Table 1 Execution results of fib.

	リターン・バリア方式					スナップショット方式	
	$K_0 = 10$	$K_0 = 20$	$K_0 = 40$	$K_0 = 80$	$K_0 = 160$	チェック追加	従来方式
総実行時間 [sec]	12.270	12.110	12.030	11.980	11.980	11.920	10.830
ごみ集め回数	65	65	65	65	65	65	65
平均ルート挿入時間 [μsec]	81.127	72.745	67.063	64.483	63.908	52.975	52.975

表 2 boyer の実行結果
Table 2 Execution results of boyer.

	リターン・バリア方式					スナップショット方式	
	$K_0 = 10$	$K_0 = 20$	$K_0 = 40$	$K_0 = 80$	$K_0 = 160$	チェック追加	従来方式
総実行時間 [sec]	36.380	32.470	28.710	27.790	27.380	26.680	26.110
ごみ集め回数	304	268	226	216	212	208	208
平均ルート挿入時間 [μsec]	192.556	163.640	156.794	150.465	146.436	143.744	143.744

パラメータ K_1（マーク・フェイズにおいて，一度にマークするセル数）と K_2（スイープ・フェイズにおいて，一度にスイープするセル数）にも依存する．今回の実験では，ルート挿入時のオーバヘッドを調べることが目的なので，$K_1 = K_2 = 40$ に固定した．

リターン・バリアの実行時オーバヘッドとしては，次のものがあげられる．

(1) ごみ集めの開始をスナップショット方式より早める必要があり，そのためにごみ集めの回数が増加する．

(2) ルート挿入を分割して行うために，mutator の実行とルート挿入処理の実行との間で制御が頻繁に移行する．移行のための，文脈切替えの時間や，キャッシュ・ミスによる実行時オーバヘッドが生じる．

(3) 関数からリターンするときに，リターン・バリアを越えてしまう場合には，mutator の実行を中断して，ルート挿入処理を行う必要があり，制御移行のオーバヘッドが生じる．

(4) さらに，今回の KCL における実装では，関数からのリターン・アドレスを書き換える方法が採用できなかったので，関数からリターンするたびに，リターン・バリアを越えるかどうかをチェックするためのオーバヘッドが生じる．

(3) については，前節で議論したように，リターン・バリアを越える可能性が低いことから，さほど大きなオーバヘッドとはならない．(4) については，今回の実装に固有なので，このオーバヘッドを除外した性能評価も行いたかった．このために，スナップショット方式に手を加え，関数からリターンするたびに，リターン・バリアを越えるかどうかのチェックを追加した処理系を準備した．この処理系における実行結果を，表 1 と表 2 の「チェック追加」欄に掲載した．「従来方式」と比較すると，ごみ集め回数とルート挿入時間は同じであるが，チェックを追加した分だけ，総実行時間が増加する．この差が，(4) のオーバヘッドと考えてよい．実行時間の増加は，boyer では 2%にすぎないのに対して，fib では 10%と大きい．これは，1 回の関数呼び出しごとの作業量が少なく，リターン処理の占める割合が大きいためである．

fib に関しては，ごみ集め回数が比較的少ないために，ごみ集め開始のタイミングが異なっても，ごみ集め回数には影響がなかった．実行時間の差は，上記 (2) のオーバヘッドによるものである．「チェック追加」と比較すると，実行時間が $K_0 = 10$ のとき 3%と大きいが，$K_0 = 160$ のときは 0.5%にとどまっている．ルート挿入時間は K_0 によってかなりの差異があるが，ルート挿入全体の時間（平均時間にごみ集め回数を掛けた値）は $K_0 = 10$ のときでも 5 msec 程度であり，全体の実行時間に影響を与えるものではない．したがって，総実行時間の差は，制御の移行によるものがほとんどである．

セルの消費が激しい boyer では，ごみ集めが頻繁に起こり，ごみ集め開始のタイミングが異なることによって，ごみ集め回数に大きな差異が生じている．この差が総実行時間に大きな影響を与えている．$K_0 = 10$ の場合を $K_0 = 160$ の場合と比較すると，ごみ集めの回数が 1.5 倍であり，実行時間は約 33%増加している．リターン・バリア方式を実用的な応用に適用する場合には，K_0 を小さくすることによる 1 回の停止時間の短縮と，それにともなうオーバヘッドとのトレードオフを十分考慮して，K_0 を設定すべきである．一方，$K_0 = 160$ としたリターン・バリア方式とスナップショット方式では，ごみ集め回数，総実行時間とも，あ

まり大きな差異はない．この実験の場合は，$K_0 = 160$ のときの停止時間は 50 μsec（図 8 参照）であり，実時間性を必要とする大多数の応用プログラムにとっては，十分小さい値である．

6. 関連研究

関連研究として，インクリメンタルルート挿入[5]と呼ばれる手法がある．この手法でも，今回実装したリターン・バリア同様，スタックを一定個（この個数を K 個とする）ずつ挿入する．実際には，1 回のルート挿入処理で，スタックを下位（ボトム）から K 個挿入し，次の K 個に対して書き込み保護をかける．もし mutator が書き込み保護をかけている部分に書き込みを行ったら，mutator を停止させて，書き込み保護をかけている部分とルート集合の固定領域を挿入し，ルート挿入を終了する（マーク・フェイズに移行する）．この手法の問題点としては，以下のことが考えられる．

- 適切なごみ集め開始条件が保証できない．
  ルート挿入をスタックの下位から行うため，書き込み保護をかけている部分に mutator がいつ書き込みを行うかはプログラムに依存する．そのため，1 回に挿入する個数を小さくしすぎると，ルート挿入がなかなか終了せず，飢餓状態に陥る可能性がある．しかし，挿入する個数を大きくすると，その分停止時間が長くなってしまう．
- mutator にオーバヘッドがかかる．
  mutator は，書き込み操作を行うたびに，書き込む場所に書き込み保護がかけられているかどうか，チェックしなければならない．

インクリメンタルルート挿入を改善した手法として，top down 法[2]と呼ばれる手法も提案されている．この手法ではルート挿入をスタックの上位（トップ）から下位に向かって行う．インクリメンタルルート挿入では，mutator は，書き込み操作を行うたびに，書き込む場所に書き込み保護がかけられているかどうかチェックしなければならず，mutator の実行にオーバヘッドがかかった．そこで，top down 法では，このオーバヘッドを避けるために，OS のページング機能を使用して，1 ページ単位で書き込み保護をかける．この手法の問題点としては，以下のことが考えられる．

- OS に依存するために実時間性が保証されない．
- OS ごとに，ページング機能に対応した処理コードを用意する必要があるので，システムの可搬性が保てない．

7. おわりに

スナップショットごみ集めにリターン・バリアを導入することによって，ルート挿入を分割する方法を提案した．提案した方法を，KCL 上に実装し，評価を行った結果，ルート挿入時における mutator の停止時間を一定範囲内に抑えることが確認された．本論文では，KCL 上に実装したために，4.2 節で述べた理由により，2.3 節の効率の良いアルゴリズムを適用できなかった．しかし，このアルゴリズムを適用した場合の性能評価を擬似的に行い，リターン・バリア導入によるオーバヘッドがかなり低く抑えられることを示した．

提案した方法は，セル要求が発生するたびに，ルート挿入処理を少しずつ実行する．もし，セル要求がない状態で関数のリターンが連続すると，関数フレームが連続してスタックからポップされてしまい，リターン・バリアによる停止が生じる可能性がある．特異なプログラムの場合には，停止が連続して生じ，最悪の場合，ごみ集めの実時間性が損なわれる危険性がある．しかし実験の結果，バリアによる停止の可能性はきわめて小さいことが分かった．実用上は十分な実時間性を備えた方法が確立したといえるであろう．

参考文献

1) Gabriel, R.: *Performance Evaluation of Lisp Systems*, MIT Press (1985).
2) 岩井輝男，中西正和：Snapshot 並列型 GC におけるルート挿入時間の削減，情報処理学会論文誌，Vol.40, No.SIG4 (1999).
3) Jones, R. and Lins, R.: *Garbage Collection*, John Wiley & Sons (1996).
4) Intel Corporation: Using the RDTSC Instruction for Performance Monitoring. `http://www.intel.co.jp/drg/pentiumII/appnotes/RDTSCPM1.HTM`.
5) 近藤　豪，中西正和：実時間ごみ集めにおけるルート挿入の効率化，情報処理学会研究報告，No.16 (1997).
6) Steele, G.: *Common Lisp the Language, Second Edition*, Digital Press (1990).
7) 湯淺太一：汎用コンピューターでの実時間ごみ集め，日経サイエンス，pp.56–71 (Sep. 1988).
8) Yuasa, T.: Real-time garbage collection on general-purpose machines, *The Journal of Systems and Software*, Vol.11, No.3, pp.181–198 (1990).
9) Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Information*

Processing, Vol.13, No.3 (1990).
10) 湯淺太一：実時間ごみ集め，情報処理，Vol.35, No.11, pp.1006–1013 (1994).

(平成 12 年 3 月 7 日受付)
(平成 12 年 5 月 24 日採録)

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，超並列計算に興味を持っている．著書「Common Lisp 入門」（共著），「Scheme 入門」，「C 言語によるプログラミング入門」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．

中川雄一郎

1975 年生．1998 年京都大学工学部情報学科卒業．2000 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．同年日立製作所（株）入社．

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．記号処理言語と並列プログラミング言語に興味を持つ．

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 年より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．博士（理学）．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 会員．