3L-1

CPS を用いた実行方式におけるスタックごみ集めの実時間化手法

石中 貴† 小宮常康‡

豊橋技術科学大学大学院情報工学専攻† 電気通信大学大学院情報システム学研究科‡

1 はじめに

Scheme 言語から C 言語へコンパイルする Scheme 処理系において，スタックのごみ集めを行うことで末尾再帰の最適化を実現する方法がある．本稿では，このスタックのごみ集めを実時間化する手法を提案する．

ごみ集めの処理中はプログラムの実行が中断されるため，一括方式のごみ集めではプログラムのレスポンス性能が低下する．実時間ごみ集めは，ごみ集めの処理を細分化し少しずつ実行することで，これを解決するものである．しかし，この細分化されたごみ集め処理が頻繁に実行されてしまうと，細分化の意味がなくなり，実時間性が失われてしまうため，細分化されたごみ集めの処理同士の間隔を保証する必要がある．

2 CPS を用いた実行方式

CPS（Continuation Passing Style）はコンパイラの中間言語としてよく用いられる表現形式である．CPS における関数呼び出しでは，その時点における残りの計算を表す「継続」を引数として明示的に渡す．そして，関数からのリターンの代わりにこの継続を明示的に呼び出すことでプログラムの続きを実行する．

Baker による CPS を用いた実行方式 [1] は，スタックのごみ集めを行うことで末尾再帰の最適化を行う方式の一つであり，ソース言語を Scheme 言語，ターゲット言語を CPS 表現の C 言語とし，CPS 表現のプログラムをそのまま実行する．また環境は C 言語の配列等で管理し，継続は C 言語の関数と環境によって実現する．以降，この継続を継続クロージャと呼ぶ．ただし，継続クロージャは 1 つのオブジェクトとして表現されているわけではない（関数呼び出しの際にコード部分と環境部分をそれぞれ引数として渡せば済む）．CPS を用いた実行方式ではリターンを継続クロージャの呼び出しで実現しているため，スタックは伸び続け，いずれオーバーフローを引き起こすが，スタックのごみ集めを行うことでこれを回避する．また、これによって末尾再帰の最適化が実現される．

我々が現在開発中の Scheme-to-C コンパイラは，この実行方式をベースとしている．スタックとヒープ両方のごみ集めを行なうが，ヒープごみ集めは湯淺らのリターンバリア付きスナップショットごみ集めアルゴリズム [2, 3] により実時間化されている．そこで，このヒープの実時間ごみ集めと同等の実時間性を持つスタックの実時間ごみ集めの手法を開発した．

3 スタックの実時間ごみ集め

Scheme-to-C コンパイラのスタックのごみ集めは，コピー方式のごみ集めを用いている．コピー方式のごみ集めは同じ大きさの二つの領域，Tospace と Fromspace を交互に使用する．今まで使用していた Fromspace が一杯になると，ごみ集めは使用中のオブジェクトだけを Tospace にコピーする．一般的なコピー方式のごみ集めは処理を一括して行うため，ごみ集め処理中はプログラムの実行が停止する．

提案する実時間ごみ集めは，このコピー方式のごみ集めの処理を細分化し，プログラムの実行により関数が呼び出されるたびに少しずつフレーム単位でコピーする．Scheme-to-C コンパイラのスタックのごみ集めでコピーされるのは C 言語の配列で管理される環境だけであり，この環境配列は単方向リスト構造をしている．

図 1 に示すように，ごみ集め中は環境配列が Tospace と Fromspace の両方に存在する．プログラムの実行により継続クロージャが呼び出されると，Fromspace に置かれた環境を参照することがあるため，環境配列のコピー漏れが生じる恐れがある．そこで Tospace と Fromspace の環境配列の境界にリターンバリア [3] を設ける．リターンバリアは，継続クロージャの呼び出しによって Fromspace へ置かれた環境を参照することを防ぐ．もし，そのような参照をしようとした際は，参照先をコピーしてバリアを下位に移動する．

Real-time Stack Garbage Collection Scheme for CPS Execution Model

† Takashi ISHINAKA. Graduate School of Information and Computer Sciences, Toyohashi University of Technology

‡ Tsuneyasu KOMIYA. Graduate School of Information Systems, University of Electro-Communications

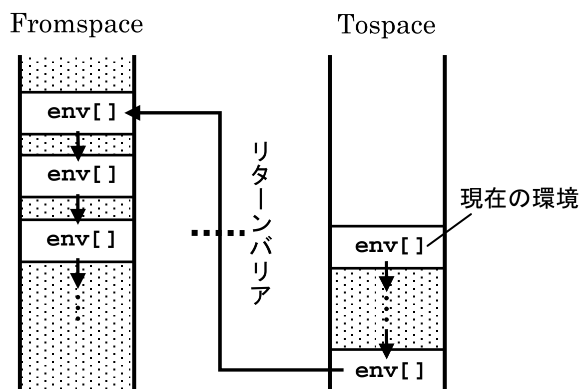

図 1　環境配列の構造とリターンバリア

リターンバリアの実装は，CPS を用いた実行方式の利点を生かして行える．Tospace から直接指されている Fromspace の環境配列（１つしかない）を環境として持つ継続クロージャのコード部分（関数へのポインタ）を上記で述べた処理を行う関数ポインタへ置き換えることで実装可能である．継続クロージャのコード部分を置き換え可能にするために，関数呼び出しの際にコード部分と環境部分をそれぞれ引数として渡すことはせずに，コード部分を環境配列の中へ埋め込んでいる．

4　細分化されたごみ集め処理の間隔

細分化されたスタックのごみ集め処理は，ターゲット言語である CPS 表現の C 言語の全ての関数の先頭で行えばよい．しかし，これらの関数は小さく，頻繁に呼び出しを行うため，細分化されたごみ集めの処理同士の間隔は非常に狭くなる．このため，ある時間に注目するとごみ集めばかり行っていることになり，実時間性が失われてしまう．

そこで本手法では，プログラムの制御フロー解析と関数ごとのスタックの消費量の見積りを行い，一定量のスタックの消費ごとに細分化されたごみ集め処理を行うコードを挿入することで，処理の間隔を調整した．また，小さなループや制御の合流点など，コードを挿入する間隔が調整不可能な場合でも，実行される細分化されたごみ集め処理の間隔が保証できるように，田中らの Punctual Polling[4] におけるポーリング間隔の動的なチェック手法を導入した．Punctual Polling は実行した命令数を動的にチェックすることで任意のポーリングの間隔を保証する．この命令数をスタック消費量に置き換え，前回の細分化されたごみ集め処理からのスタック消費量を動的にチェックすることで，処理同士の間隔を保証した．

このようにスタックのごみ集めの間隔を保証できたが，一方でヒープのごみ集めはヒープが消費されるタイミングで実行されるため，スタックとヒープの細分化されたごみ集め処理が**図 2** のような状態になり，実時間性が損なわれる可能性がある．そこで本手法では**図 3** のようにスタックのごみ集め中は，スタックのごみ集めのタイミングでヒープのごみ集めも行うようにし，ヒープが消費されるタイミングではごみ集め処理を行わないものとした．

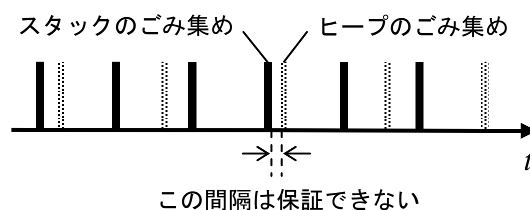

図 2　ヒープのごみ集めによる実時間性の喪失

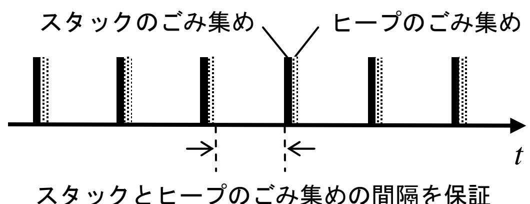

図 3　スタックのごみ集め中の処理間隔の保証

5　まとめ

本稿では，CPS を用いた実行方式をベースとした処理系における，スタックのごみ集めの実時間化手法の提案を行った．

参考文献

1) H. Baker: CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A., *ACM Sigplan Notices*, Vol.30, No.9, pp.17－20 (1995).
2) T. Yuasa: Real-time garbage collection on general-purpose machines, *The Journal of Systems and Software*, Vol.11, No.3, pp. 181－198 (1990).
3) 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), pp87－99 (2000).
4) 田中義純，田浦健次朗，米沢明憲: 定期的なポーリングを保証するアルゴリズム，並列処理シンポジウム JSPP2001, pp.229－236, (2001).