記号処理 41－4
（1987. 6. 26）

汎用計算機に適した実時間ごみ集め

湯淺 太一
京都大学数理解析研究所

汎用計算機，すなわち単一プロセッサのフォン・ノイマン型逐次実行計算機上のリスト処理システムに適した実時間ごみ集めのアルゴリズムを提案する。本アルゴリズムは，使用頻度の高い，Lispにおける CARや CDR，スタック操作などに実行時オーバーヘッドをかけないために，汎用計算機上でも効率よく実現できる。本報告では，アルゴリズムの紹介とともに，その正当性を議論し，動的な挙動を解析する。

Realtime Garbage Collection on General-purpose Machines

Taiichi YUASA
Research Institute for Mathematical Sciences
Kyoto University

An algorithm of realtime garbage collection is presented, which is intended for list processing systems on general-purpose machines, i.e., Von Neumann style serial computers with a single processor. By avoiding execution overhead on frequently used operations such as pointer dereferences (Lisp CAR and CDR) and stack manipulations, this algorithm can be efficiently implemented on such machines. The correctness of the algorithm is discussed and the dynamic behavior is analyzed.

1. はじめに

ごみ集め(garbage collection)は、Lispをはじめとするリスト処理システムにおいて、不用になったリスト・セルを回収するために広く利用されている。現在実用化されているごみ集めの方法は、基本的には次のようなものである。使用可能なセル(フリーセル)をフリーリストとして保持しておき、リスト処理プログラムが新しいセルを要求するたびに、フリーリストからセルを取り出す。フリーリストが空になると、リスト処理プログラムの実行を一時的に中断し、ごみ集め処理を開始する。ごみ集め処理は2つのフェイズからなり、まずマーク(mark)フェイズで現在使用中のすべてのセルをマークする。次のスイープ(sweep)フェイズで全セルを走査し、マークされなかったセルをフリーリストに回収する。これらの一連の処理が終ると、中断されていたプログラムの実行を再開する。

ごみ集めの最大の難点は、プログラムの実行を一時的に中断することにある。通常は数秒から数十秒間中断されるが、高度なリスト処理プログラムになるほど、この時間は長くなる。ごみ集めのためのこのようなプログラム実行の中断は、実時間処理を必要とするリスト処理プログラムにとっては大問題である。リスト処理は従来人工知能分野の応用に広く利用されており、その多くの応用プログラムが実時間処理を必要としていることから、プログラム実行の中断はリスト処理実用化への大きな障害となっている。

このため、ごみ集め処理をリスト処理プログラムの実行と並行して行う、いわゆる実時間ごみ集めの方法がいくつか提案されてきた。しかしこれらの方法は、ごみ集め専用のプロセッサを前提としたり、あるいはマイクロコード・レベルの並列性を利用してはじめて実用的な実行速度を得るものであり、現実に多くのリスト処理システムが稼働している汎用の計算機には適さない。ここでいう汎用計算機とは、VAXやMC68020といった、単一プロセッサによるフォン・ノイマン型計算機を指す。これまで提案されてきた実時間ごみ集めの方法は、汎用計算機上でも原理的には実現可能であるが、その際のプログラム実行効率の低下が大きいことから、実用化されていない。

以下では、汎用計算機におけるリスト処理に適した実時間ごみ集めのアルゴリズムを提案する。このアルゴリズムは、(1)汎用計算機上でもリスト処理プログラムの実行効率の低下が少ない、(2)メモリ効率の低下も少ない、(3)従来の一括方式のごみ集めを採用しているシステムへの応用が容易である、などの特長を備えている。また、汎用計算機のみならず、リスト処理専用の計算機への応用も可能である。

2. 基本的アイデア

従来の方法では、ごみ集めを一括して行うためにプログラム実行の中断をおこすのであった。そこで、ごみ集めの一連の処理を、小さな部分処理に分割し、プログラム実行の間に少しずつごみ集め処理を行うことで実時間化が可能となる。しかし、ごみ集め射の途中でもプログラム実行が継続することから、一括方式のごみ集め処理を単に分割するだけでは不十分で、ごみ集め中のリスト構造の変化に対応できるような方法が必要となる。特に、マークフェイズにおいて、マークすべきすべてのセルを確実にマークしなければならない。例えば、図1(a)で、セルBまでマークされたとする。(図中の太い矢印は、セルBの先の他のセルをマークできるように、セルBがごみ集め用のスタックから指されていることを示す。)この時点でプログラム実行に制御が移り、その影響で図1(b)に示す状態になったとする。セルCはセルAから指されれているのでマークする必要があるが、セルBはもうセルCを指していないので、ごみ集め処理が再開されてもセルCはマークされ

ずに終る。セルCが何らかの方法でマークされるようなメカニズムが実時間化に際しては必要である。

リスト処理の基本操作には

cons、car、cdr、rplaca、rplacd の5つがある。実際の経験から、carとcdrの使用頻度は他の3つに比べてはるかに高い。後述のように、従来提案されてきた実時間ごみ集めのアルゴリズムは、上の例におけるセルCをマークするためのメカニズムをcarとcdrに加えるものであった。最も使用頻度の高い操作に実時間化のための負担がかかるために、専用ハードウェアによる並列・高速処理なくしては実用化できなかったのである。汎用計算機上でも効率のよい実時間ごみ集めを導入するには、carとcdrに負担をかけず、むしろ使用頻度の低いrplacaとrplacdに何らかの実時間化のためのメカニズムを加えるべきである。

さて、図1(a)の状態から図1(b)の状態に至るには、図2に示すような基本操作が使われる。ここでx、y、zはプログラム変数のつもりである。この3つの基本操作のうちのいずれかが、セルCをマークすると同時にCへのポインタをごみ集め用スタックに積んでおかねばならない。(セルのマーキングとそのセルへのポインタをスタックに積む操作とは同時に行われるので、以下では、「マークする」といえばこの両方の操作を意味することにする。)まずcarとcdrが、その値となるセルにマークする方法が考えられる。Bakerのアルゴリズム[1]はこの方法を採用している。しかし、この方法は、使用頻度の高いcarとcdrの実行速度を低下させるために、汎用計算機には適さない。第2の方法としては、rplacaとrplacdが、その第2引数であるセルをマークすることが考えられる。これはDijkstraのon-the-flyアルゴリズム[2]に採用されている。この方法は一見carとcdrに負担をかけないようにみえるが、rplacaとrplacdにおける処理だけでは実時間化に対応できない。これは、図3のような場合を考えれば明らかで、結局carとcdrにも負担をかけざるを得ず、やはり汎用計算機には適さない。そこで、第3の方法によって、carとcdrに全く負担をかけない実時間化を行おうというのが、ここで紹介するアルゴリズムの最も基本となるアイデアである。すなわち、rplacdが、その第1引数のcdr部の指していたセルにマークする方法である。rplacaにも同様の操作が必要で、rplacaの場合は第1引数のcar部の指すセルにマークをつけることになる。つまり、セルの内容が書きかえられる際に、書きかえる以前に指していたセルをマークし、これによって、一括方式のごみ集めでポインタをたどることによってマークできたすべてのセルをマークすることができる。この方法が図3の例に対しても有効であることは明らかである。

実時間化のためにはこの他にも、解決すべき問題がいくつかある。まず、ごみ集めをいつ開始するかという問題がある。一括方式ではフリーリストが空になってからごみ集めを開始してもよかったが、実時間方式では、ごみ集めの最中にもリスト処理プログラムからセルの要求があり、それらの要求に即座に対応できるようにフリーリストにいくらかのセルが残っている状態でごみ集めを開始しなければならない。このため、ここで提案するアルゴリズムでは、フリーセルの現在の長さを記憶しておくためのカウンタを設け、その値がある数M以下になった時にごみ集めを開始することにする。Mの適当な値については後で言及する。

もう1つの問題は、セルの二重回収をいかに防ぐかということである。一括方式ではフリーリストが空の状態でスイープフェイズを開始し、マークのついていないセルはすべてフリーリストに回収すればよかった。しかし実時間方式では、スイープフェイズの途中でもセルの要求があるので、スイープフェイズのはじめにフリーリストを空にするわけにはいかない。この問題を解決するには、フリーセルをその他のセルと区別できればよい。フリーリストは、通常は各フリーセルのcar部のみ(cdr部でもよいが、ここではcar部としておく)を使って構成され、フリーセルのcdr部は全く利用されていない。そこで、この使用されていないcdr

部に、そのセルがフリーリストであることを示す特別な識別子(leave_meと呼ぼう)を必ず格納しておくことにする。そして、スイープフェイズでは、マークのついていないセルで、しかもそのcdr部がleave_meでないセルのみを回収すればよい。

3. アルゴリズム

この節では、前節で紹介した方法による実時間ごみ集めのアルゴリズムを示す。説明を簡単にするため、次のような単純なリスト処理システムのモデルを想定する。このモデルは、図4に示すようなデータ領域を使用する。レジスタR[1]～R[NR]は、プログラムの使用するデータを保存するための領域である。ここでレジスタの数NRは十分小さいものとする。これらのレジスタはマーキングの際のルートである。つまり、いずれかのレジスタから、ポインタをたどることによって到達できるセルのみがその時点で使用中のセルとする。ヒープはセルを割当てる領域である。このモデルではセル以外のアトム(数値や記号)も取り扱えるものとするが、それらはここでは興味の対象外であり、ヒープとは別のどこかに割当てられるものと考える。gcsはごみ集め用のスタックであり、free_listはフリーリストの先頭を記憶しておくためのシステム変数である。このモデルは非常に単純なもので、実際のリスト処理システムに不可欠ないくつかの機能が欠けている。例えば、リストセルという1種類のセルしか取り扱えないし、実行時スタックもない。しかし、後述のように、本アルゴリズムをこれらの機能をもつシステムに拡張するのは容易である。

比較のために、図5に一括方式のごみ集めアルゴリズムを示す。ここで使われているgcs_pushは、セルを「マーク」する関数で、実際のマーキングと同時に、そのセルへのポインタをgcsにプッシュする。consは、フリーリストが空かどうかを調べ、もし空ならごみ集め処理gcを起動する。ごみ集めを行ってもフリーセルが見付からなければ、飢餓(starvation)状態になる。実際のシステムでは、ここでデータ領域を拡張するなどして実行を継続するのが普通だが、ここでは簡単のために飢餓状態になればシステムは死んでしまうことにする。

この一括方式のアルゴリズムを実時間化する。まずgcのマークフェイズとスイープフェイズの各ループを細分化した markとsweepを定義する(図6参照)。markはマークフェイズのループ本体をある回数K1だけ実行し、sweepはスイープフェイズのループ本体をある回数K2だけ実行する。K1とK2の実際の値についてはあとで触れる。これらの使って、cons、rplaca、rplacdを実時間化に対応するように書きかえたものが図7である。carとcdrには全く変更はない。変数phaseは現在のごみ集めの状態を保持し、その値によって3つの基本関数が実時間ごみ集めのための必要な処理を行う。

4. 正当性

一般に、ごみ集めアルゴリズムが「正しい」ためには、少なくとも次の性質を満たさねばならない。

(a)使用中のセルは回収しない。
(b)未使用のセルはいつかは回収する。

本アルゴリズムについては、以下のことがいえる。

(1)markは、ごみ集め開始時に使用中のセルをすべて唯一度だけマークし、そ

れ以外のセルはマークしない。
(2)sweepは、ごみ集め開始時に未使用で、かつフリーセルでないセルをすべて回収し、それ以外のセルは回収しない。
(3)consは、マークフェイズ中はその返すセルすべてをマークし、スイープフェイズ中はスイープされていないセルのみをマークする。

この3点から、本アルゴリズムは、ごみ集め開始時に未使用でかつフリーセルでないセルのすべて、しかもそのようなセルのみを1回のごみ集めで回収することがいえる。これから、上の(a)と(b)を本アルゴリズムが満たしていることが証明できる。詳しい証明は文献[3]にある。

5. 動的特性

図8に、本アルゴリズムを使った場合のフリーセルの個数の変化を示す。図で、横軸はconsの呼び出し回数、縦軸はセル数を表わす。F(t)とA(t)は、t回目のconsの呼び出し時点におけるフリーセル数と使用中のセル数をそれぞれ表わす。Nはシステムが利用できる全セル数、Mはごみ集めを開始するときのフリーリストの長さ、K1とK2はそれぞれmarkとsweepのループ本体の繰り返し回数である。最初、フリーセルはN個あり、consの呼び出しごとに1個ずつ減少する。その数がMになると最初のごみ集めが始まるが、マークフェイズの間はセルの回収は行われないので consの呼び出しごとにフリーセルの個数は1ずつ減少する。スイープフェイズになるとフリーセルの回収がconsの呼び出しごとに行われるために、フリーセルの個数は増減を繰り返す。このためこの間のF(t)のグラフは一般にはギザギザとなる。ごみ集めが終ると、F(t)は再び傾き-1で減少し、F(t)=Mになったところで2回目のごみ集めが始まる。

いま、ごみ集めがt=aに始まり、t=bにマークフェイズからスイープフェイズにかわり、t=cにごみ集めが終ったとする。consは、マークフェイズ中は、ごみ集め開始時に使用中のセルのみをマークし、しかも同じセルは1度しかマークしない。そこで、合計A(a)個のセルをマークすることになる。一方、consの呼び出しごとにK1個のセルがマークされるので、A(a)/K1回のconsの呼び出しでマークフェイズが完了する。スイープフェイズでは、全セルを走査し、1回のcons呼び出しではK2個のセルを走査する。したがってスイープフェイズはN/K2回のcons呼び出しで終了する。ごみ集め開始時にはM個のフリーセルが残っており、ごみ集め中に、consの呼び出しによってc-a=A(a)/K1+N/K2個のセルが新たに消費される。そして、ごみ集め開始時に使用中か、あるいはごみ集め開始時にフリーリストにはいっていたセルを除くすべてのセルが回収される。つまり、N-A(a)-F(a)個のセルが回収される。その結果、ごみ集め終了時のフリーセルの個数は

$$M + (N - A(a) - F(a)) - A(a)/K1 - N/K2$$

となる。

ここで、飢餓状態をおこさないための十分条件を考える。飢餓状態が起るということは、いいかえれば、ある時点においてF(t)<0となることである。そこで、すべての時点においてF(t)≧0となるための十分条件を示せばよい。スイープフェイズ中のF(t)の値は、ヒープ上にどのように回収可能なセルが分散しているかに依存する。回収可能なセル数が同じであれば、ヒープの最初の方に回収可能なセルが全くなく、ある位置から後のセルがすべての回収可能な場合が最悪のケースである。回収可能なセルが見付かるまで、フリーセルの個数はconsの呼び出しごとに減少し、それ以後フリーセルの個数は線形に増加する。この減少から増加

に移行する時点のフリーリストの個数が、ごみ集め中にF(t)の取りうる最小値となる。回収不可能なセル数はA(a)+Mだから、この最悪のケースでは、t=(A(a)+M)/K2においてF(t)が減少から増加に移行し、その時のF(t)の値は

$$M - A(a)/K1 - (A(a) + M)/K2$$

である。もしこの値が非負であればこのごみ集めの間には飢餓状態は有り得ない。この条件を整理すると、

$$M \geqq A(a) (1/K1 + 1/K2)/(1 - 1/K2)$$

となる。ここでA(a)はごみ集め開始時に使用中のセル数であるが、これはごみ集めをいつ開始するかに依存し、推定するのが困難である。あるリスト処理プログラムの実行中に、同時に使用されるセル数の最大値なら、推定が比較的容易である。その値をAmaxとすると、上の条件が成り立つための次の十分条件を得る。

$$(1) \quad M \geqq Amax (1/K1 + 1/K2)/(1 - 1/K2)$$

上の議論は、ごみ集め開始時にM個のフリーセルが残っていることを前提としている。しかし、状況によっては、ごみ集め終了時のフリーセルの個数がM以下となり、ただちに次のごみ集めが始まることも考えられる。そこで上式が飢餓状態を起こさないための十分条件となるには、さらに、F(c)≧Mがすべてのごみ集め終了時t=cについてなりたつ必要がある。この条件が成り立つための十分条件は、上と同様の計算によって、

$$(2) \quad N \geqq (M + (1 + 1/K1) Amax) / (1 - 1/K2)$$

となる。したがって、式(1)と(2)の両方を満たせば飢餓状態は起りえない。例えば、K1=K2=20であれば、この両式を満たすNの最小値は1.216Amaxである。この値はK1とK2を大きくすればAmaxに近づく。一方、一括方式であれば、明らかにN≧Amaxであれば飢餓状態は起らない。そこで、K1=K2=20の場合を例にとれば、最悪の場合でも、本アルゴリズムを採用した場合に、一括方式の1.216倍のセルがあれば飢餓状態におちいらないことが保証される。なお、この場合、(1)と(2)を満足するMの最小値は 0.105Amaxである。つまり、フリーセルの個数が全セル数のおよそ10%になったときにごみ集めを始めればよい。

次に、ごみ集めの回数を調べる。式(2)を仮定すると、あるごみ集めがt=aで始まってから次のごみ集めが始まるまでに、consは

$$A(a)/K1 + N/K2 + (N - A(a) - F(a)) - A(a)/K1 - N/K2$$
$$= N - A(a) - M$$

回呼び出される。簡単のためにA(t)を定数Ameanとすると、consをT回呼び出す間にごみ集めが起る回数はT/(N-Amean-M)で与えられる。上のK1=K2=20の場合のMとNの最小値 M=0.105Amax、N=1.216Amaxを使い、Amax=2Ameanとして計算すれば、0.82T/Amean回のごみ集めがおこることになる。一方、一括方式ではすぐにわかるように、T/(N-Amean)回のごみ集めが起る。上のようにNとしてAmaxをとれば、この値はT/Ameanである。つまり、この場合は本アルゴリズムの方がごみ集めの回数は少なくなることがわかる。もちろん、同じNの値に対しては、Mの分だけ本アルゴリズムの方がごみ集めの回数は多くなる。

6. 拡張

これまでは、簡単なリスト処理システムのモデルを使って、そのごみ集めを実時間化する方法を述べてきた。これを実際のリスト処理システムに応用するための方法を以下で考える。

実際のシステムでは、少数のレジスタ群ではなく、スタックがマーキングのルートとなる。ごみ集め開始時に、スタックから指されている全セルを一度にマークすると、その間のプログラム中断は決して短くない。スタックから指されるセルを少しずつマークする必要がある。このためには、ごみ集め開始時に、スタックの内容をすべて退避しておき、consが呼び出されるごとに少しずつマークしていけばよい。スタック内容の退避は、ほとんどの汎用計算機が提供しているブロック転送の機能を使うことによって短時間に行える。

また、実際のシステムでは、リストセル1種類だけではなく、記号セルなどを含めた多種類のセルを取り扱う。このような場合には、セルの種類ごとにフリーリストを用意し、セルの種類ごとにこれまでリストセルに対して述べた方法を適用することによって、本アルゴリズムを拡張することは容易である。

本アルゴリズムは、セル領域も含め全メモリ領域のcompactionを行うシステムに応用することは困難と思われる。しかし、ベクタなどの可変長データの本体のみをrelocateするシステムには応用できる。マークフェイズで、可変長データのヘッダセルがマークされるごとに、その本体可変長データ用領域の一方に移動する。そして、移動されたすべての本体を、マークフェイズの間に少しずつ走査し、これら本体から指されているセルをマークしていけばよい。本体の移動にもブロック転送が利用できる。

7. まとめ

汎用計算機上においても効率のよい実時間ごみ集めのアルゴリズムを提案し、その正しさを示すとともに、動的特性を解析した。本アルゴリズムの主要な特長は、はじめに記したとおりである。現在、本アルゴリズムをKCL[4]上に実現することを計画している。KCLはカーネルがC言語で書かれた移植性の非常に高いシステムであり、さまざまな汎用計算機上で稼働している。本アルゴリズムにはハードウェアやOSに依存した操作を必要としないことから、KCLの高い移植性を保持しながらごみ集めの実時間化がはかれるものと思われる。また、rplacaなどの破壊操作には、簡単なコードを挿入するだけで実時間化に対応できることから、システムのインストール時にユーザが一括方式か実時間方式かのどちらかを選択できるようにもできると思われる。

参考文献

[1] H.G.Baker "List Processing in Real Time on a Serial Computer", CACM Vol.21 No.4, 1978.

[2] E.W.Dijkstra他 "On-the-Fly Garbage Collection: An Excercise in Cooperation", CACM Vol.21 No.11, 1978.

[3] T.Yuasa "Realtime Garbage Collection on General-purpose Machines", RIMS Preprint No.535, 1986.

[4] T.Yuasa and M.Hagiya "KCL Report", 帝国印刷出版部, 1985.

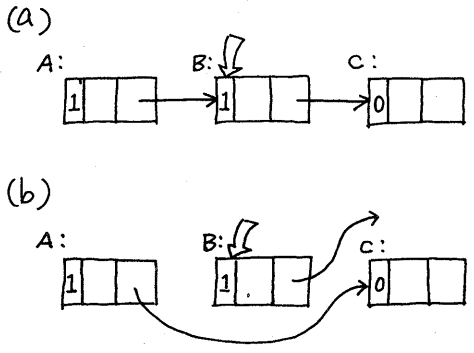

図1. ごみ集め中のリスト構造の変化

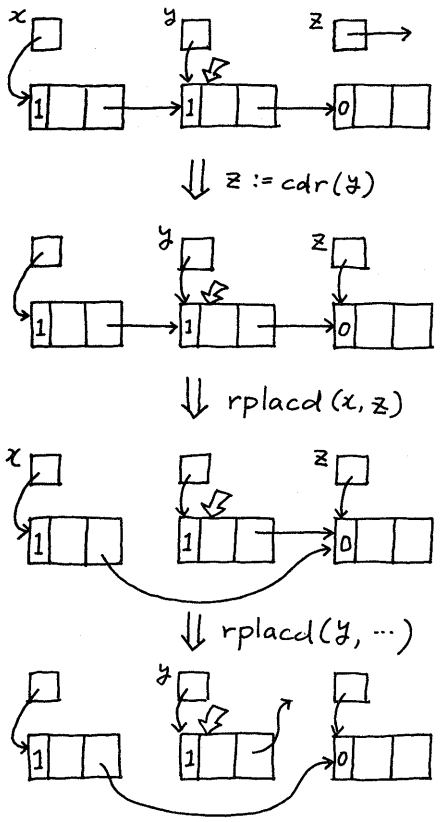

図2. リスト構造の変化と基本操作

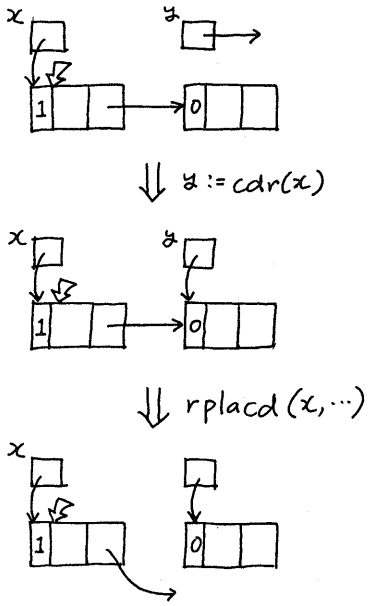

図3. もう1つの例

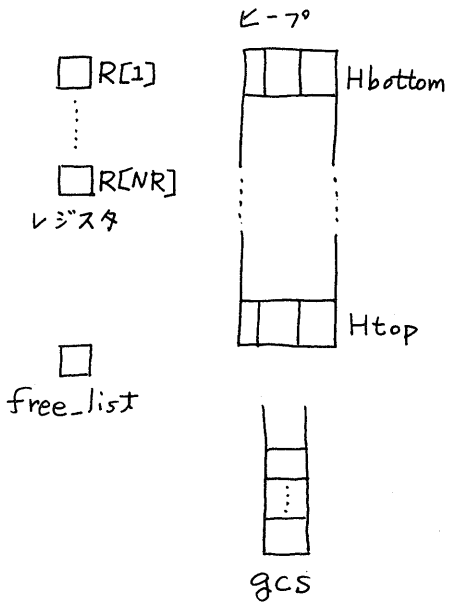

図4. モデルの使用するデータ領域

```
procedure Lcons(i,x,y);
  begin if free_list = nil then
          begin gc();
                if free_list = nil
                   then error("no storage")
          end;

        p := free_list;
        free_list := p^.car;

        p^.car := x;
        p^.cdr := y;

        R[i] := p
  end (of Lcons);

function Lcar(x) : pointer;  Lcar := x^.car;

function Lcdr(x) : pointer;  Lcdr := x^.cdr;

procedure Lrplaca(x,y);  x^.car := y;

procedure Lrplacd(x,y);  x^.cdr := y;
```

```
procedure gc();
  begin ( initialization )
        gcs_init();
        for i := 1 to NR do gcs_push(R[i]);

        ( mark phase )
        while not gcs_empty() do
          begin p := gcs_pop();
                gcs_push(p^.car);
                gcs_push(p^.cdr)
          end;

        ( sweep phase )
        for p := Hbtm to Htop do
          if p^.mark then
            p^.mark := false
          else begin p^.car := free_list;
                     free_list := p
               end
  end (of gc);
```

図5. 一括方式ごみ集めと基本操作

```
procedure mark()
  begin i := 1;
        while i =< K1 and (not gcs_empty()) do
          begin p := gcs_pop();
                gcs_push(p^.car);
                gcs_push(p^.cdr);
                i := i+1
          end
  end (of mark);

procedure sweep();
  begin i := 1;
        while i =< K2 and sweeper =< Htop do
          begin if sweeper^.mark then
                  sweeper^.mark := false;
                elseif not sweeper^.cdr = leave_me then
                  begin sweeper^.car := free_list;
                        sweeper^.cdr := leave_me;
                        free_list := sweeper;
                        free_count := free_count+1
                  end;
                sweeper := sweeper+1;
                i := i+1
          end
  end (of sweep);
```

図6. mark と sweep

```
procedure Lcons(i,x,y);
  begin ( garbage collection dispatcher )
        if phase = mark_phase then
          begin mark();
                if gcs_empty() then phase := sweep_phase
          end
        elseif phase = sweep_phase then
          begin sweep();
                if sweeper > Htop then phase := idling
          end
        elseif free_count =< M then
          begin phase := mark_phase;
                sweeper := Hbtm;
                for i := 1 to NR do gcs_push(R[i])
          end;

        if free_count =< 0 then error("no storage");

        p := free_list;
        free_list := p^.car;
        free_count := free_count-1;

        p^.car := x;
        p^.cdr := y;
        p^.mark := (p => sweeper);

        R[i] := p
  end (of Lcons);

procedure Lrplaca(x,y);
  begin if phase = mark_phase then gcs_push(x^.car);
        x^.car := y
  end;

procedure Lrplacd(x,y);
  begin if phase = mark_phase then gcs_push(x^.cdr);
        x^.cdr := y
  end;
```

図7. 実時間に対処する3つの基本操作

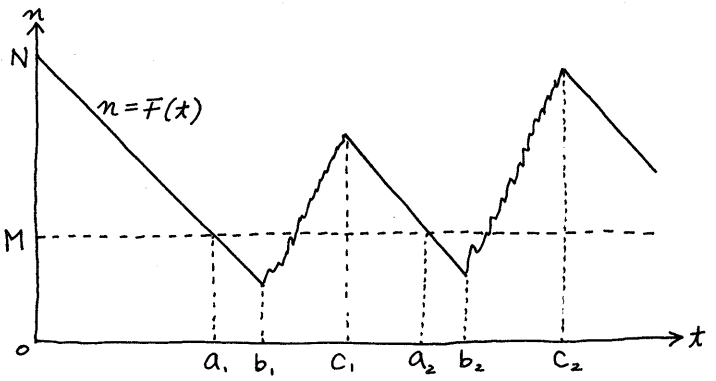

図8. フリーセル数の変化