インクリメンタルなメジャーコレクションを行なう世代別GC

前田敦司 † 山口喜教 †

マークスイープアルゴリズムに基づくインクリメンタルガーベジコレクタは、停止時間をごく短時間に抑えることが可能であるが、効率が悪く、CPU 時間が増加する。この性能低下の原因は、細かすぎる粒度で GC を起動することによるコンテキスト切り替えのオーバーヘッドと、アロケーション処理がインライン展開できないこと等による。

一方、世代別ガーベジコレクタは高い CPU 効率が得られ、また平均の停止時間は比較的短いが、旧世代領域の GC(メジャーコレクション) の際には長い時間にわたって計算処理が停止してしまい、リアルタイム応用には適さない。

我々は、新世代領域の GC(マイナーコレクション) のたびに、インクリメンタルに旧世代領域のガーベジコレクションを行なう新しい GC アルゴリズムを提案する。インクリメンタル化のために必要となるライトバリアは、世代別ガーベジコレクタの実装にいずれにせよ必要なライトバリアと同じ仕組みを利用する。

このアルゴリズムにより、通常の世代別ガーベジコレクタに近い効率を保ちながら、停止時間を短く保ち、ガーベジコレクタのリアルタイム性を向上させることができる。

Generational Garbage Collector with Incremental Major Collection

Atusi Maeda † and Yoshinori Yamaguchi †

Incremental garbage collectors based on mark-sweep algorithms can minimize the pause caused by garbage collection at the expense of increased CPU time. The performance degradation is due to context switch overhead of too-fine-granularity GC invocation and not-inlineable allocator.

Generational collectors, on the other hand, can achieve high efficiency and relatively short average pause time, while occasional long pause for collection of old generation space (major collection) makes these collectors unsuitable for real-time applications.

We propose a new GC algorithm which incrementally reclaims old generation space every time a minor collection is performed. Write barrier for generational collector is also used for incremental collection.

With this algorithm, we can improve real-time response of garbage collector by keeping pause time short, with little overhead added to ordinary generational collectors.

1. はじめに

動的に割り付けたメモリ領域を自動的に回収し再利用するガーベジコレクション（GC）技術は、メモリ管理に関するエラーからプログラマを解放し、プログラムの生産性を向上させることができる。しかし、GC に伴う処理停止時間のため、GC を利用できる応用はリアルタイム性を必要としない分野に限られてきた。ハードリアルタイムの応用だけでなく、アニメーションなどのソフトリアルタイム応用においても、GC による停止時間は品質の低下となって現れる。また、より一般的な対話的処理においても処理の停止はユーザにとって不快であり、GC における停止時間の短縮は大きな課題であった。

† 筑波大学 電子・情報工学系
Institute of Information Science and Electronics, University of Tsukuba

1.1 インクリメンタル GC とその問題点

インクリメンタル GC[2),9)] は、GC の回収サイクルを細かなステップに分割して、わずかなステップずつ処理を行うことにより、停止時間をごく短い時間に制限する技術である。インクリメンタル GC は実質的に、ユーザプログラムの処理を行うスレッド（ミューテータ）と別に動作するガーベジコレクションスレッド（コレクタ）を設け、これらを並行に動作させるものとみなせる。最も一般的な実装では、メモリ割り付け要求のたびにあるステップ数だけ GC 処理を行なう。

インクリメンタル GC を用いることにより、ガーベジコレクションを利用したプログラムでもリアルタイム処理を実現できる可能性があるが、インクリメンタ

ルでない（一括型の）ガーベジコレクタを用いた場合と比較して CPU 時間が増加するという欠点がある。その原因としては、(1) 高い頻度のコンテキスト切り替え、(2) ライトバリアのオーバーヘッド、(3) 割り付け要求のインライン展開ができないことの 3 点が挙げられる。

1.1.1 コンテキスト切替の頻度

汎用のスレッドライブラリを用いて完全に別のスレッドとして動作させる場合に比較すれば軽量ではあるもののインクリメンタルな GC のステップを実行するたびに、（例えば）どこまでマーキング処理を進めたか等、ある程度のコンテキスト情報を退避・復帰する必要がある。

インクリメンタルな GC 処理を独立したスレッドと見なした場合、コンテキスト切り替えの頻度は通常のマルチスレッドプログラムと比較してはるかに高くなりうる。GC 機能を持つ言語では、数十命令に 1 回程度のメモリ割り付けを行うプログラムも稀ではないが、これは最近の CPU では 100ns に 1 回以上のコンテキスト切り替えに相当することになり、通常の OS のタイムスライスである 10ms 程度と比較して著しく高いといえる。

1.1.2 ライトバリアのオーバーヘッド

インクリメンタルなガーベジコレクタでは、ユーザプログラムのスレッド（ミューテータ）がコレクタスレッドと並行して動作するため、両者の協調処理が必要となる。代表的なインクリメンタル GC アルゴリズムであるインクリメンタルアップデート型[2] 以下 IU 型）アルゴリズムおよびスナップショット型[9]（以下 SB 型）アルゴリズムでは、ミューテータがヒープ領域の書き換えを行った際にもコレクタがポインタ追跡を正しく行えるよう、ミューテータによるヒープへの書き込みの際に、書き換え結果をコレクタに通知するライトバリア処理が必要となる。このため、ヒープへの書き込みは一括型 GC に比較して遅くなる。

1.1.3 インライン展開の困難さ

ガーベジコレクションのアルゴリズムは、ヒープのレイアウトやメモリ割り付けの方法と相互に密接な関連がある。コピー方式のガーベジコレクタにおいては、メモリ領域の割り付けが、割り付け済みの領域の終端を示すポインタの移動（および境界チェック☆）のみの、機械語にして数命令の処理ですむ。さらに、割り付け処理をインライン展開することができるため、非常に高速な処理が可能となる。マークスイープ方式のガーベジコレクタにおいても、Lisp における cons セルのように、頻繁に割り付けられる特定のデータ型のみならばコピー方式に近い高速な処理が可能であり、同様にインライン展開することができる。

しかし、メモリ割り付けのたびにガーベジコレクション処理を行なうインクリメンタルガーベジコレクタにおいて割り付け処理をインライン展開して速度向上を得るには、ガーベジコレクタ自体のかなりの部分のプログラムの複製を数多く作ることになり、現実的でない。

1.2 世代別 GC と停止時間

一方、消費する CPU 時間を削減できる GC アルゴリズムとして世代別 GC[5),6)] がある。この方式は、「動的に割り付けられたデータのうち、寿命が短いものに含まれるゴミの割合は、全体の平均より高い」という経験的事実（generational hypothesis）を利用して、最近割り付けられたデータオブジェクト（短寿命データ）のみを対象にしたガーベジコレクションを行うことによって、高い効率で空きエリアを回収するものである。

短寿命データのみを対象とした GC（マイナーコレクションと呼ぶ）の際には、プログラムのルート集合（値スタック、CPU のレジスタ、ポインタを含む大域変数）および長寿命のデータから短寿命のデータへのポインタの集合を出発点として、短寿命のデータのみに対してポインタ追跡を行い、到達不能であったデータを回収する。このマイナーコレクションに要する時間は、世代別管理を行わないガーベジコレクタの停止時間に比較して桁違いに短く、処理系のコンフィギュレーションやシステム性能に依存するが通常は数 ms 程度である。この停止時間は、それまでのガーベジコレクションアルゴリズムで問題となっていた応用分野のうちのかなりの範囲に対しては十分許容可能な程度に短いものである。

しかしながら、プログラムを実行し続ければいずれ必要となる、長寿命のデータに対する GC（メジャーコレクション）の際の停止時間は、以前の GC と同様に長いものとなり、結果として世代型でない一括型ガーベジコレクションが適さない分野については、世代別 GC もやはり利用できない。

2. インクリメンタル世代別 GC

本稿では、細分化したメジャーコレクションのステップをマイナーコレクションのたびにインクリメンタルに行うことによって、世代別 GC の停止時間をマイ

☆ ただし、ページ書き込み保護を用いて境界チェックを行えば、比較命令や分岐命令は不要になる。[3)]

ナーコレクションによる停止時間程度のオーダーにとどめるガーベジコレクション方式を提案する。

この方式を用いることで、マイナーコレクション程度の停止時間を許容できる応用に関してはガーベジコレクションを利用したプログラム開発が可能となる。また、提案する方式はインクリメンタル GC の一種ではあるが、1.1 節で述べた問題点の多くを解決できる。

(1) コンテキスト切り替えの頻度
本方式では、メジャーコレクションの処理を細分化し、マイナーコレクションごとに少しずつ処理を進める。メモリ割り付け要求ごとにコンテキストを切り替える通常のインクリメンタル GC と比べると、コンテキスト切り替えの頻度は何桁も低いため、オーバーヘッドは問題にならない。

(2) ライトバリアのオーバーヘッド
世代別 GC では長寿命のデータから短寿命のデータを指し示すポインタをすべて把握する必要があるため、ヒープ領域への代入の際にライトバリアを用いてそのようなポインタが生じたことを検出する。すなわち、メジャーコレクションをインクリメンタル化する以前から、世代別 GC の処理にはもともとライトバリアのオー バーヘッドが含まれており、それでも世代型でない GC と比較した場合には、多くのプログラムに対してそれを補って余りあるだけの速度向上が得られている。

(3) インライン展開の可能性
メモリ割り付け要求の結果、データが短寿命データとして最初に割り付けられる際の処理は世代別 GC と全く同じである。したがって、通常の世代別 GC と同様に、マイナーコレクションの方式としてコピー GC を用いた場合にはメモリ割り付け処理を容易にインライン展開できる。

長寿命データに対するメジャーコレクションにはライトバリアを用いたインクリメンタル GC を用いるため、メジャーコレクションのアルゴリズムはコピー方式以外のアルゴリズムをベースとしたものとなる。このアルゴリズムをインクリメンタル化する方法としては IU 型、SB 型いずれを用いることも考えられる。

以下のアルゴリズムの説明では、次のような構成を仮定する。

- 簡単のため、ヒープに割り付けられるデータ型はポインタを 2 個含むデータ（Lisp の cons セル）のみとする。GC の対象にならないデータ（即値やコードへのポインタ等）は、ポインタワードの中にエンコードされており、ポインタ値から判別できるものとする。
- ヒープは、短寿命データ領域と長寿命データ領域の 2 つの世代からなる。
- マイナーコレクションの際、短寿命データ領域内のデータは全て長寿命データ領域にコピー (tenuring) される。
- 長寿命領域内のセルは、プログラマから見えるデータとは別の双方向ポインタにより接続されて、1 つの環状リストを成している。（すなわち、メジャーコレクションのアルゴリズムは Treadmill 型 GC[1] を用いる。）

```
struct Cell {
  struct Cell *car, *cdr;
};

Set<Cell> white, black, gray;
Set<Cell*> root;

void gc() {
  foreach (i in root) {
    scan(&i);
  }
  while (gray != {}) {
    Cell* g = gray から要素を一つ取り出す;
    black = black + {g};
    scan(&g->car);
    scan(&g->cdr);
  }
}

void scan(Cell* c) {
  if (*c in white) {
    white = white - {*c};
    gray = gray + {*c};
  }
}
```

図 1 ポインタ追跡型 GC の基本アルゴリズム
Fig. 1 Basic algorithm of tracing GC

2.1 基本アルゴリズム

コピー型・マークスイープ型・Treadmill 型など、ポインタ追跡を行う GC アルゴリズムに共通するアルゴリズムの概略[8] を、C++ に似た疑似言語を用いて図

1 に示す。

ここで white, black, gray はセルの集合であり、GC の開始時点で、ヒープ内の全てのセルは white に属しているものとする。まだ追跡していないポインタを含むセルはいったん gray 集合に入れられ、ポインタ追跡がすんだ時点で black となる。gray 集合が空になった時点でアルゴリズムは終了し、その時 white に属するセルがルート集合から到達不能なもの（ゴミ）である。

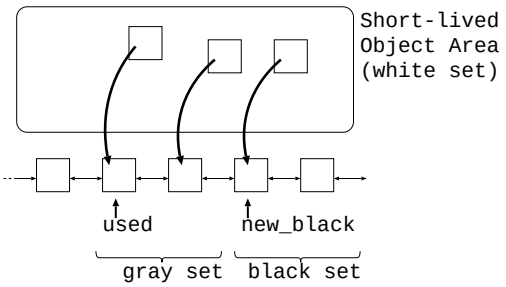

図 2 マイナーコレクションの集合の表現
Fig. 2 Representation of sets in minor collection

```
struct NewNode {
  boolean is_forwarded;
  union {
    struct Cell cell;
    struct Cell *new_address;
  };
};

struct OldNode {
  struct OldNode *prev, *next;
  struct Cell cell;
  boolean color;
};

struct Node *black, *gray, *used;
Set<Cell*> root, remembered;
boolean current_color;
```

図 3 ヒープのデータ構造
Fig. 3 Data structures of the heap

以後のアルゴリズムは、この基本アルゴリズムにおいて、white, black, gray の各集合をどのように表現するかのバリエーションという観点から説明する。本稿で提案するヒープ構成では、短寿命領域から長寿命領域へのマイナーコレクションにおける各集合の表現は図 2 に示すようになる。white 集合は短寿命領域に相当する。ルート集合から到達可能で、まだコピーされていないセルの内容は長寿命領域の used ポインタが指すセルにコピーされ、used ポインタが 1 つ進む。new_black ポインタを used ポインタを追うように進めながら、セルの中のポインタを走査していく。used ポインタと new_black ポインタの間の部分が gray 集合に相当する。new_black をポインタが通過した後のセルは black 集合に相当する。ヒープのデータ構造を図 3 に示す。

このデータ構造を用いた、インクリメンタル化する前の（すなわち単なる世代別 GC の）マイナーコレクションおよびメジャーコレクションのアルゴリズムをそれぞれ図 4 および図 5 に示す。長寿命領域への書き込みが起きた場合には、新たに書き込まれるのが短寿命領域へのポインタであるかどうかを検査し、もしそうならばその短寿命領域へのポインタを remembered 集合に加える。remembered 集合はマイナーコレクションの際のポインタ追跡の出発点となる。

メジャーコレクションは、環状リストの全てのセルをちょうど使いきった時点で呼び出され、その時点で短寿命領域は空で、全てのセルが current_color と同じ color フィールドの値を持つと仮定している。（このようにぴったり使いきらない場合の処理は省略する。）到達可能と判ったセルは、color を反転し、双方向リストを操作して gray ポインタの指すセルの次に挿入され、次いで gray ポインタが進む。セルの内容を走査する位置を示す black ポインタと gray ポインタの間が gray 集合となる（図 6）。

2.2 IU 型インクリメンタル世代別 GC

IU 型のアルゴリズムを用いてメジャーコレクションをインクリメンタル化した GC のアルゴリズムを図 7 に示す。マイナーコレクションの際にすべての処理を行い、used ポインタを用いた tenuring と gray および black ポインタを用いた走査が少しずつ並行して行われるので、長寿命領域のレイアウトは図 8 に示すように、オリジナルの Treadmill GC と同じになる。

この他、ライトバリアによる書き込みの検査の際、(1) 代入の左辺が color として current_color を持つ長寿命領域内のセルであり、かつ (2) 代入の右辺が color として current_color でない色を持つ長寿命領域内のセルへのポインタならば、右辺のポインタが指し示すセルを gray にする処理が必要となる。

マイナーコレクションごとにルート集合の変更を反映させるため、図 4 で用いた scanNew のかわりに

```
void minor_gc() {
  new_black = used;
  foreach (i in root + remembered) {
    scanNew(&i);
  }
  remembered = {};
  while (new_black != used) {
    Cell* g = &new_black->cell;
    new_black = new_black->next;
    scanNew(&g->car);
    scanNew(&g->cdr);
  }
}

void scanNew(Cell** c) {
  if (*c が短寿命領域内の Cell を指すポインタ) {
    if ((*c)->is_forwarded) {
      *c = (*c)->new_address;
    } else {
      copyAndForward(c);
    }
  }
}

void copyAndForward(Cell **c) {
  used->cell.car = (*c)->car;
  used->cell.cdr = (*c)->cdr;
  used->color = current_color;
  (*c)->is_forwarded = true;
  (*c)->new_address = used;
  used = used->next;
}
```

図 4 インクリメンタルでないマイナーコレクションのアルゴリズム
Fig. 4 non-incremental minor collection algorithm

scanBoth を用いて、ルート集合内の長寿命領域へのポインタを毎回スキャンしている。

通常の IU 型インクリメンタルコレクタにおいて、新たに割り付けられるセルは white に属し、ルートからの到達性が判明しなければ GC サイクルの終わりにゴミとして回収される。図 7 のアルゴリズムは新たに tenure されるセルを black として扱っており、これらのセルは現在の GC サイクルの次のサイクルの終りまで決して回収されない。これは通常の IU 型コレクタに比較して効率が悪いように思われるが、実際に tenure される瞬間にはルート集合から指し示されているので、black とみなすのが適切と思われる。

```
void major_gc() {
  gray = black = used;
  current_color = ! current_color;
  foreach (i in root) {
    scanOld(i);
  }
  while (black != gray) {
    black = black->prev;
    Cell* g = &black->cell;
    scanOld(g->car);
    scanOld(g->cdr);
  }
}

void scanOld(Cell *c) {
  if (c が長寿命領域内の Cell を指すポインタ) {
    if (c->color != current_color) {
      delete(c);
      c->color = current_color;
      insert_before(gray, c);
      gray = c;
    }
  }
}
```

図 5 インクリメンタルでないメジャーコレクションのアルゴリズム
Fig. 5 non-incremental major collection algorithm

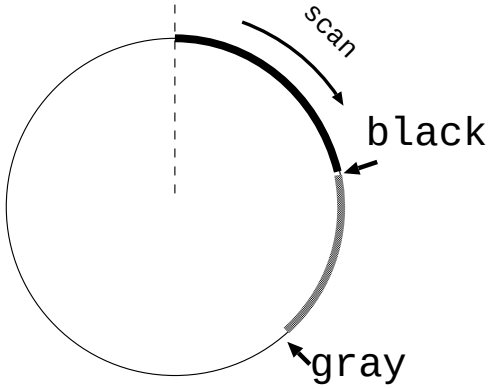

図 6 メジャーコレクションの集合の表現
Fig. 6 Representation of sets in major collection

2.3 SB 型インクリメンタル世代別 GC

SB 型のアルゴリズムを用いてメジャーコレクションをインクリメンタル化した GC のアルゴリズムを図 9 に示す。マイナーコレクションごとにルート集合の

変更を検知する必要はなく、また、新たに tenure されたセルから長寿命領域のセルへのポインタを追跡する必要もないため、scanBoth は必要なくなる。

この他、ライトバリアによる書き込みの検査の際、(1) 代入の左辺が持つ長寿命領域内のセルであり、かつ (2) 上書きされるポインタが color として current_color でない色を持つ長寿命領域内のセルへのポインタならば、上書きされる前にそのポインタの指すセルを gray にする処理が必要となる。

3. 議　　論

提案した 2 つのインクリメンタル世代別 GC のうち、SB 型を用いたアルゴリズムは図 5 および図 4 に示したインクリメンタルでないアルゴリズムとの差異が小さく、同じ数のセルを tenure する際のコストは scanBoth を用いる必要のある IU 型より小さいものと思われる。ただし、長寿命領域への書き込みが起きたときの処理は、通常のインクリメンタル GC の場合と同様に SB 型の方がより保守的である（実際はゴミである場合にも gray にする可能性がある）。

通常のインクリメンタル GC において、IU 型と SB 型の間の実用上最も大きな違いは、GC サイクル中に割り付けられて、GC サイクルが終るまでにゴミとなるセルを、その GC サイクルの終りで回収できるか否かにあると思われるが、今回提案したインクリメンタル世代別コレクタでは、どちらも新たに tenure したセルを black として割り付けており、この点の違いはなくなっている。

このように違いがなくなった（IU 型が、通常より保守的になった）原因は、マイナーコレクションのたびにすべてのルートポインタを走査している点にある。GC サイクルの開始時には、ルート集合のうち比較的値の変化しにくいシンボル表やプログラムコードのみを走査し、それらから到達可能なセルを全て追跡し終えた後にスタックやレジスタなどの変化の激しいルートの部分を走査することも考えられる。

2 節で示したアルゴリズムはいずれも、長寿命領域に対するコレクションが常に動作している状態にある。実際には、GC サイクルが終了した時点でなるべくぴったり長寿命領域を使いきるように、マイナーコレクションごとに行うポインタ追跡のステップ数を調節することが望ましい。ヒープに余裕がある状態で GC サイクルが終了するということは、必要以上の CPU サイクル数をコレクタが消費していることを意味するからである。

この基準に照らすと、通常のインクリメンタル GC では、メモリ割り付け要求 1 回あたりに追跡すべきポインタ数が数個以下となり、場合によっては 1 を下回る。このため、ある程度の割合までヒープを消費し終えるまでコレクタを動作させないようにする制御が行われる。本稿の方式では、1 回のステップで処理する粒度がはるかに荒いため、GC を停止させる場合に比べて、より均一な停止時間が期待できる。

4. 他の研究との関連

インクリメンタル GC と世代別 GC を組み合わせる試みとして 10) がある。これは細粒度のインクリメンタル GC の効率を改善ずるため、細粒度のまま世代を別けたものである。

世代別 GC のバリエーション内で、停止時間がある限度を越えない範囲で tenured garbage をなるべく減らす（すなわち tenuring をなるべく遅らせる）研究に 7) がある。tenuring を遅らせることによって長寿命 GC をある程度は先延ばしにできるが、いずれは長寿命 GC による停止が訪れる。また、不必要に tenuring を遅らせることは、コピー回数を増やし、コレクタが消費する CPU 時間を増加させる。

世代別管理の考え方を用いて、より大きな粒度でインクリメンタルに処理を行うコレクションアルゴリズムに Train Algorithm[4] がある。この方式は、長寿命領域を限られたサイズの小領域（car と呼ぶ）に分割し、1 度のコレクションで 1 つの car に含まれるデータだけを他の car にコピーする処理を行うことで、停止時間に制限を設けている。また、強連結成分の要素をすべて、1 つの train と呼ぶ car のグループに集め、train 外からの参照がなくなったら train 全体を回収することによって、大きなサイクルの回収を可能としている。数多くの小領域の間の remembered set を維持管理するコストや、train を増やすポリシーなど、問題となる点は未だに多いと思われる。

5. おわりに

メジャーコレクションをインクリメンタルに行うことにより、細粒度のインクリメンタル GC より高い効率で、停止時間をそれほど長くない程度に抑えることを目指した新たな GC アルゴリズムを提案した。

本稿で提案した 2 つのアルゴリズムは、インクリメンタル GC における効率の問題点の多くを解決すると思われる。また、停止時間のオーダーは、世代別 GC のマイナーコレクションと比較してそれほど劣らない範囲に収まると思われ、この程度の停止時間であれば許容できる応用も多いと思われる。

今後は、実験を通して提案したアルゴリズムの有効性についてさらに検証を進めていく。

参 考 文 献

1) Baker, H. G.: The Treadmill: Real-Time Garbage Collection Without Motion Sickness, *ACM SIGPLAN Notices*, Vol. 27, No. 3, pp. 66–70 (1992).
2) Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C.S. and Steffens, E. F.M.: On-the-fly garbage collection: An exercise in cooperation, *Communications of the ACM*, Vol. 21, No. 11, pp. 966–975 (1978).
3) Gabriel, R. P.: *Performance and evaluation of LISP systems*, MIT Press (1985).
4) Hudson, R. L. and Moss, J. E. B.: Incremental Collection of Mature Objects, *Proc. Int. Workshop on Memory Management*, No. 637, Saint-Malo (France), Springer-Verlag, pp. 388–403 (1992).
5) Lieberman, H. and Hewitt, C.: A real-time garbage collector based on the lifetimes of objects, *Communications of the ACM*, Vol. 26, No. 6, pp. 419–429 (1983).
6) Ungar, D.: Generation Scavenging: A non-disruptive high performance storage reclamation algorithm, *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pp. 157–167 (1984).
7) Ungar, D. and Jackson, F.: An adaptive tenuring policy for generation scavengers, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 14, No. 1, pp. 1–27 (1992).
8) Wilson, P. R.: Uniprocessor Garbage Collection Techniques, *Proc. Int. Workshop on Memory Management*, No. 637, Saint-Malo (France), Springer-Verlag (1992).
9) Yuasa, T.: Real-time garbage collection on general-purpose machines, *Journal of Software and Systems*, Vol. 11, No. 3, pp. 1–13 (1990).
10) 小池龍信, 岩井輝男, 中西正和: オブジェクトの世代を考慮に入れたインクリメンタルなごみ集め処理, 情報処理学会論文誌プログラミング, Vol. 40, No. SIG07-002 (1999).


```
boolean restart_major = true;

void minor_gc_iu() {
  if (restart_major) {
    restart_major = false;
    gray = black = used;
    current_color = ! current_color;
  }
  new_black = used;
  foreach (i in root + remembered) {
    scanBoth(&i);
  }
  remembered = {};
  while (new_black != used) {
    Cell* g = &new_black->cell;
    new_black = new_black->next;
    scanBoth(&g->car);
    scanBoth(&g->cdr);
  }
  for (int i = 0; black != gray && i < STEP;
    i++) {
    black = black->prev;
    Cell* g = &black->cell;
    scanOld(g->car);
    scanOld(g->cdr);
  }
  if (black == gray) {
    restart_major = true;
  }
}

void scanBoth(Cell** c) {
  if (*c が短寿命領域内の Cell を指すポインタ) {
    scanNew(c);
  } else {
    scanOld(*c);
  }
}
```

図 7 IU 型インクリメンタル世代別 GC アルゴリズム
Fig. 7 IU-type incremental generational collection algorithm

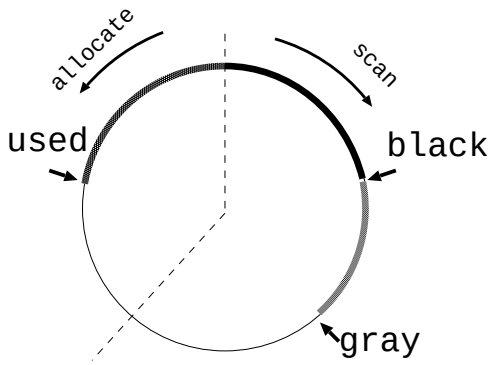

図 8 インクリメンタル GC の長寿命領域
Fig. 8 Long-lived object space of incremental algorithms

```
boolean restart_major = true;

void minor_gc_sb() {
  if (restart_major) {
    restart_major = false;
    gray = black = used;
    current_color = ! current_color;
    foreach (i in root) {
      scanOld(i);
    }
  }
  new_black = used;
  foreach (i in root + remembered) {
    scanNew(&i);
  }
  remembered = {};
  while (new_black != used) {
    Cell* g = &new_black->cell;
    new_black = new_black->next;
    scanNew(&g->car);
    scanNew(&g->cdr);
  }
  for (int i = 0; black != gray && i < STEP;
       i++) {
    black = black->prev;
    Cell* g = &black->cell;
    scanOld(g->car);
    scanOld(g->cdr);
  }
  if (black == gray) {
    restart_major = true;
  }
}
```

図 9 SB 型インクリメンタル世代別 GC アルゴリズム
Fig. 9 SB-type incremental generational collection algorithm