一般教養としてのGarbage Collection (コンパイラ演習番外編)

遠藤敏夫 (endo@logos.t.u-tokyo.ac.jp)

第 6 版: Jan 27, 2005

1 Garbage Collection とは?

世の中のプログラミング言語をメモリ管理の方法に注目して大きく 2 分すると、(1) 手動のメモリ解放を必要とする言語、(2) 自動的なメモリ解放を行なってくれる言語、に分けられる (表 1)。C 言語や C++, Pascal などは手動であり、ML や Java, Perl などは自動である。

C 言語のプログラムでは malloc 関数によってメモリ領域を確保 (allocate) したら、使い終った後に free 関数によって解放しなければならない (以下本稿では、malloc で確保される領域や、Java/C++におけるオブジェクト、ML における tuple や record などを、まとめて オブジェクト と呼ぶ)。このため C 言語のプログラマは、どのオブジェクトが使い終わったのかを一々気にする必要があり、たちの悪い、つまり後々まで気がつきにくいバグを引き起こしやすいのである。最近、サーバプログラムのバグをついた不法侵入やシステム破壊が報道されることが多いが、その原因の一つはメモリ解放ミス (早すぎる/遅すぎる free) である[1][2]。

一方、ML プログラム等において作成された tuple や record は、プログラマがほうっておいても、使い終った後自動的に解放されるため、プログラマの苦労は大きく軽減される。

本稿のトピックである、garbage collection(GC) は、このような自動メモリ管理方式のうち最も広く使われているものである。ところが、この便利な GC は世間では嫌われていることが多い。もはや昔話であるが、以下のような話がある。

- 筆者が情報科学科に進学して UNIX をはじめて使ったころ (1995 年ころ) の emacs は、キーボードを打っていると突然、数秒間固まり反応しなくなることがしばしばあった。これは GC のせいである。
- Java が広まりはじめたころ、web ブラウザ上の Java アプレットのゲームが広まりかけたことがあった。しかしリアルタイムゲームで 0.5 秒固まると、ゲーム性が落ちる。

本稿では、一般ユーザ/プログラマからは (えてして迷惑な)black box として扱われがちな GC の中身について解説する。GC といっても様々種類があるので (図 1)、代表的なアルゴリズムを解説する。

最近の言語処理系では、図中のアルゴリズムを単独で用いることは少なく、性能向上のためにさまざまなアルゴリズムを組み合わせている場合が多い。例えば、最も有名な Java 処理系である、Sun の HotSpot VM (Ver. 1.4.2) の GC は、以下のようにアルゴリズムを組み合わせている[3] 。基本は generational GC (6.2 節) であり、新世代ヒープ (後述) については copying GC (4.2 節) を採用、旧世代ヒープについては mark-compact GC (4.3 節) を採用している。さらに、-Xincgc オプションをつけて実行した場合は incremental GC (6.1 節) が用いられる。

大まかな話の流れは成書 [7] やサーベイ論文 [12] に沿っているので、興味のある人はそちらを参考にしてもらいたい。

[1] 2002 年 3 月に、zlib という LZW 圧縮ライブラリに二重解放バグが見つかった。最悪の場合 root 権限が奪われる

[2] セキュリティホールの重要な原因として、メモリ解放ミスの他にもバッファオーバーフローなどが挙げられるが、それらは本稿の対象外である

[3] http://java.sun.com/docs/hotspot/

言語	自動／手動	確保方法	解放方法
C	手動	malloc	free
C++	手動	new, malloc	delete, free
Java	自動	new, 文字列演算など	(GC)
ML	自動	tuple 作成、record 作成、文字列演算など	(GC)
Scheme	自動	pair 作成、vector 作成、文字列演算など	(GC)
Perl	自動	配列作成、文字列演算など	(GC)

Table 1: 様々なプログラム言語のメモリ確保／解放方法

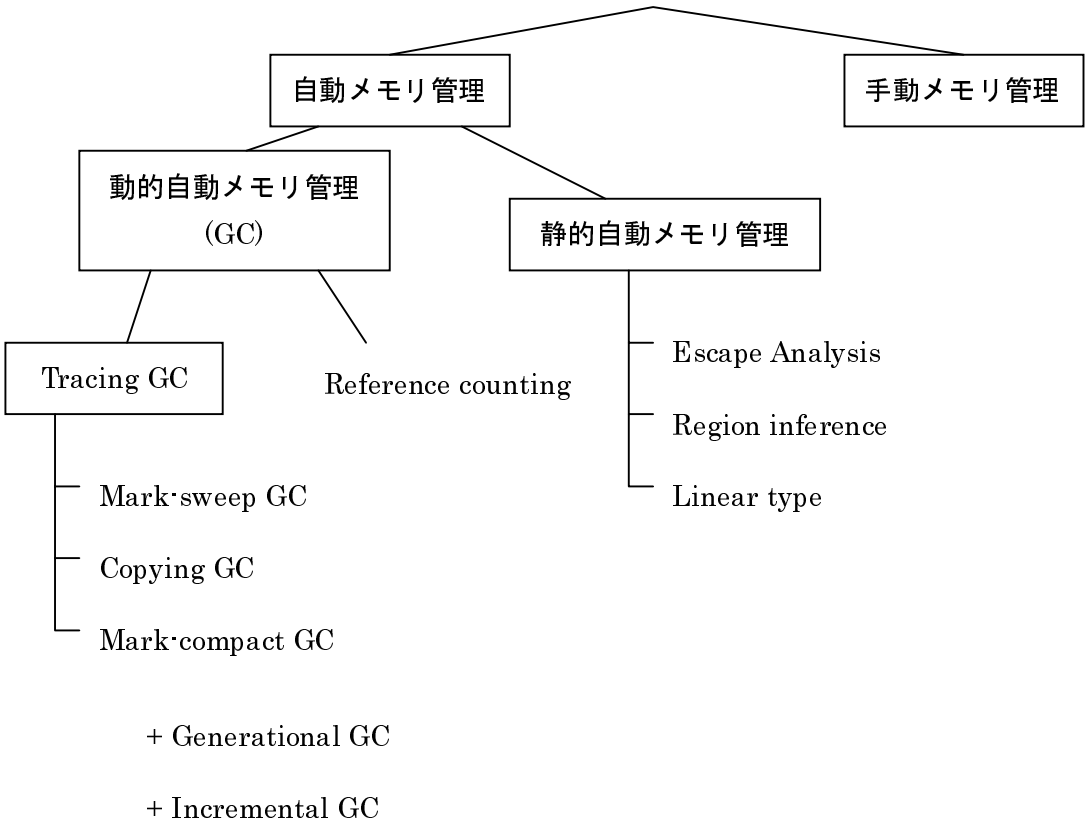

Figure 1: メモリ管理方式の種別。Tracing GC を指して単に「GC」と呼ぶこともある。

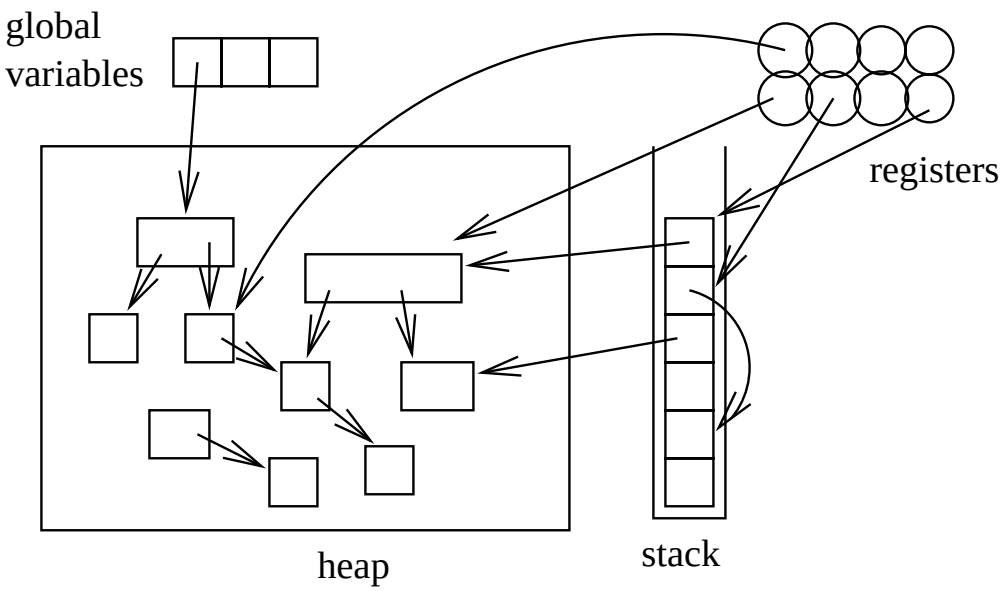

Figure 2: プログラム動作中のメモリ構造の例

2 GC を知る上での予備知識

2.1 実行時メモリ構造

これからの物事を理解するために、プログラムが動いている時のメモリ領域の構造を図 2 に示す。大まかには、Java, ML, C 言語も含めほとんどの言語は似たような構造になる。

ヒープ内の小さい四角はオブジェクトを表す。また、あるオブジェクト A の中に含まれるポインタが、別のオブジェクト B を指していることがある (record の中に record がある場合や、オブジェクトのフィールドに別のオブジェクトを含む場合など)。図中では参照関係を、A から B への矢印で表す。

- スタック には、局所変数や関数呼び出しの履歴などが格納される。(話をごく簡単にすると) 関数呼び出しが起こると伸びて、return が起こると縮む。
- 大域変数 を格納する領域が用意される。定数が格納される領域も独立に用意される場合が多い。
- レジスタは、関数の引数、計算の途中結果など様々な値をとる。オブジェクトを指すポインタの可能性もある。
- ヒープ は自由な順番でメモリ領域を確保/解放できる領域である。オブジェクト作成 (C では malloc) を行なうと、ヒープの中に必要な大きさの領域が確保される。
- 図には示していないが、プログラムの コード領域 もどこかにある。

2.2 ヒープの必要性

ユーザプログラムがオブジェクト (または tuple, record...) を作成するたびに、必要な領域がヒープ中に確保され、ヒープはどんどん食い潰される。オブジェクト作成を行なう関数 *allocate(n)*(言語処理系作成者が用意する) は、ヒープから連続した n バイトの領域を取り出し、そのポインタをユーザプログラムに渡す。*allocate* は、新しい領域、つまり他の用途に使われていない領域を確保しなければならない。プログラムはそのポインタを用いて、演算 (オブジェクト内容の読み書きなど) を行なう。

さて、この n バイトの領域をわざわざヒープに確保するのはなぜだろうか？スタックに n バイト push してはいけないのだろうか？以下のようなプログラムがあるため、ダメである。

```
f() {
  r = new Object(....); // オブジェクト作成
  s = new Object(....); // オブジェクト作成
  (r, s を用いた処理)
  return r; // r を返す
}
```

関数 f はオブジェクト r と s を作り、そのうち r を呼出元へ返すとする。f が終了した後も r を残す必要があるので、r をスタックに置くわけにはいかない。そのためヒープに r を確保するのである。

さて、ヒープを食い潰す一方ではいつしかメモリ不足になってしまう。そのため、GC のない言語 (C/C++ など) ではプログラマが手動でいらないオブジェクトを解放してやる必要がある。一方 ML や Java では、GC が自動的に「将来使われる可能性のあるオブジェクト」と「もう使われないオブジェクト (=garbage)」を分類し、後者を解放してくれる。たとえば上のプログラム例において、GC が f() の終了直後に発生したとする (通常は GC 発生タイミングにプログラマは関与しない)。すると自動的に r は生き残り、s は解放される。

この不思議な GC の正体は、言語処理系作成者が提供するランタイムライブラリの一つである。つまり、入出力ルーチン/実数計算ルーチンなどと同等であり、ユーザプログラム実行中 にメモリに置かれ、呼ばれるのを待つプログラムたちである。予言者ならぬただのプログラムである GC が、どうやっていらないオブジェクトを区別するのか？GC には大きく分けて (1) reference counting (2) tracing GC という方法があり、それぞれ次章以降で説明する[4]。

2.3 手動メモリ管理

ここでは、allocate+手動 free の実装について軽く触れておくことにする。実は多くの GC アルゴリズムにおいても、ここで述べる処理と同じようなことが起こっている[5]。

関数 *free(p)* が行なうべき仕事は、指定されたオブジェクト p を空き領域 (=将来の allocate のために利用できる領域) にすることである。プログラムは飛び飛びに存在するオブジェクトを free するかもしれないため、一般に空き領域はヒープ上に通常は散らばってしまっている。これらを将来再利用するために、リストにつないで管理 (フリーリスト) するのが一般的である。結局、free が行なう仕事は、p から始まる領域をフリーリストへつなぐということである。

一方 *allocate(n)* が行なうべき仕事は、様々な空き領域がつまったフリーリストから、要求された大きさ以上の箇所を見つける、ということになる。

ここで速度に関して補足しておく。Allocate/free にかかる時間はたいがいの場合一瞬なのだが、必ずしも定数時間で終わるとは限らない。fragmentation(後述) がひどいときに allocate を行なうと充分な大きさの空き領域を見つけるためにフリーリストの終りの方までたどらないといけないかもしれない。また free において、coalescing(後述) にかかる時間を短縮するには工夫が必要である。

用語の説明をいくつか示す。

- *fragmentation:* 小さい空き領域が飛び飛びにあることを fragmentation と呼ぶ。たとえば 50 バイトの空き領域が 2 つ別にあるとき、80 バイトの allocate 要求に答えることはできない。
- *coalescing:* 連続した空き領域ができたとき、それらを一つの空き領域にまとめることを coalescing と呼ぶ。
- *header:* ユーザが使うメモリ領域／空き領域の一つ一つに対して管理情報が必要である。その管理情報を、オブジェクトの最初のワードの直前に配置するシステムが多い。その管理領域を header と呼ぶ。例えばオブジェクトサイズ、次の空き領域のポインタなどが含まれる。

Allocate/free を高速にする研究も 40 年間以上にわたり数多くなされている。興味のある人は Wilson らのサーベイ論文 [13] を参考にしてほしい。

[4]残念ながら「完璧に」区別するのは不可能であり、どこかで近似することになる。どんな GC でも、必要なものを捨ててしまうよりは、不要かもしれないものを保持しておく方がましだ、という方針をとる

[5]例外は copying GC などで、ヒープポインタと呼ばれる値を増やすだけで allocate することができる

一つだけ segregated free lists という手法を紹介する。単純なフリーリスト方式だと、allocate 時に最悪フリーリストの長さに比例した時間がかかる。これを改善するため、フリーリストを空き領域サイズに応じて複数用意することができる。一つのフリーリスト中に同一サイズの領域しか入っていないので、allocate はほとんど定数時間でできる。

3 Reference Counting

本章では reference counting という GC 方式を説明する。この方式の基本的な考え方は、「どこかからポインタで指されているオブジェクトは、将来使い道がある」というものである。この方式では、それぞれのオブジェクトに対して reference count(参照数) を常に数えておく。オブジェクト a の参照数とは、世の中のどこか (ヒープ中の他のオブジェクト/スタック/大域変数など) から a を指しているポインタの本数である。この参照数はプログラムの実行につれて増減する。

Reference counting の基本方針は以下の通りである。あるオブジェクト a の参照数が正のときは、a は将来使われるかもしれないのでとっておく。やがて a の参照数が 0 になったら、そのオブジェクト a は今後の使い道がないと見なし、それを解放する。

この解放処理は連鎖する可能性がある。たとえば、a が別のオブジェクト b を参照していたら b の参照数を 1 減らす。その結果 b の参照数が 0 になるかも知れず、その時 b を解放する...

この参照数を管理するために、コンパイラが協力するのが主流である。ポインタの増減が起りうる個所全てで、参照数をいじるコードを余分に出力するのである。なお、参照数の置き場所については、オブジェクトのヘッダに記録する処理系が多い。

Reference counting 方式の欠点は、以下のようになる。

- オブジェクトへのポインタが増減するたびに余計な処理をするので、ユーザプログラムの実行時間が遅くなりがちである。
- オブジェクトのサイクルができると解放できなくなる。たとえば a が b を指し、b が a を指すとする。プログラムの他の部分が a, b へのポインタを一切持っていない場合でも、a も b も参照数が 0 にならずにいつまでも解放できない。

これらの欠点ため、ML や Java 処理系ではあまり使われていない。ただし、tracing GC に比べて停止時間が比較的短いため、トータルでの実行速度よりもリアルタイム性の方が重要な、スクリプト言語などでは使われる場合もあるようだ (perl など)。

なお、上述の問題を軽減する技法も多く提案されている。1 番目の問題に対しては、defered reference counting アルゴリズム [4] が提案されている。ルート (スタック/レジスタ/大域変数) からオブジェクトへのポインタと、オブジェクトからオブジェクトへのポインタを区別し、後者のみ記録する (ルートは書き換え頻度が非常に高いので)。ある期間おきにルートをスキャンし、ルートから直接参照されておらずかつ参照数 0 のオブジェクトのみを解放する。

2 番目の問題に対しては、主に 2 通りの解決法が考えられる。(1) 普段は reference counting を用い、ヒープが満杯になってきたら後述の tracing GC を起動する。(2) ゴミサイクルではないか？と思われる候補を見つけたら、それを trace して他所から参照されていないことを確かめ、解放する (cycle detection アルゴリズム)。

一方、これらの技法を用いると停止時間が短いという利点は失われる傾向にある。「全て良い」アルゴリズムはなかなか無いものである。

4 Tracing GC

Reference counting ではオブジェクトの生死判定をいわば「局所的に」行なっていた。オブジェクト o の生死判定をするには、o へのポインタ増減を見張っていれば良かった。この方式にはサイクルがあると解放できないので困るという欠点がある。

Tracing GC は、より「大域的に」生死判定を行なう手法である。ヒープ全体を見渡して、全オブジェクトの生死判定を一気に行なうのである。このため tracing GC は、reference counting と動作タイミング

が異なる。ヒープが満杯になるまでは allocate を続け、満杯になったときにヒープ中のゴミオブジェクト (死んだオブジェクト) を一気に検出して解放する... というのが典型的である。

その生死判定は、以下のように行なわれる。

- ユーザプログラムが現在直接さわれるメモリ領域 (レジスタ、スタック、大域変数などで、これらを GC のルートと呼ぶ) からポインタによって指されるオブジェクトは生きている。
- 生きたオブジェクトからポインタによって指されるオブジェクトは生きている。
- それ以外のオブジェクトはゴミであり、解放されるべきである。

これでなぜうまくいくかを少し説明する。GC の終了後、将来ユーザプログラムが何をするかもしれないか考えよう。例えばユーザプログラムは今レジスタに入っているポインタが指すレコードのフィールド読み込みをするかもしれない。それがまたポインタだったらその内容を読むかもしれない。そのときにおかしくならないためには、ユーザプログラムが将来アクセスし得るオブジェクトを全部、保持する必要がある。

この辺りの考え方は reference counting と共通だが、reference counting と異なり、サイクルでも解放できることを確認してほしい。「どこかから指されているか否か」ではなく「ルートから到達可能か否か」という大域的な情報を用いるので、これが可能なのである。

そういうわけで tracing GC の主な仕事は、ルートからポインタによって到達可能なオブジェクトを再帰的に探索し、たどられたオブジェクトを何らかの方法で「区別」することである。探索には単純な allocate/free 処理よりもずっと長い時間が必要なため、図 6 のようにユーザプログラムを止めてしまう[6]。これが世の中で「GC は遅い」と言われるゆえんである。

なお、8.2 節で述べる条件 (ルートの識別、ポインタか否かの判定、オブジェクトサイズの知識など) は、この再帰探索を行なうために必要となってくる。

Tracing GC の中にもいくつか種類があり、代表的なものは mark-sweep GC (4.1 節)、copying GC (4.2 節)、mark-compact GC (4.3 節) である。

それぞれの説明に入る前に、Tracing GC アルゴリズムを理解する上で役に立つ、オブジェクトの「色」モデルを紹介する。GC 中のオブジェクトには、白、灰色、黒の 3 色のいずれかが塗られていると考える。

- 白 ··· GC にまだ発見されていない
- 灰色 ··· GC に発見された。ただし白いオブジェクトを直接指しているかもしれない
- 黒 ··· GC に発見されたし、直接指すオブジェクトもすでに GC に発見されている (黒か灰色)

探索開始時にはヒープ中オブジェクトは全て白である。白いオブジェクトたちの世界に対して、黒組が勢力を拡大していく。各瞬間の黒と白の波打ち際に灰色が位置する。世の中が全て黒か白になったら探索は終了、というのが直観的な説明である。

4.1 Mark-Sweep GC

これまでに述べた tracing GC の基本アイデアを最も素直に実装する、最古の tracing GC アルゴリズムが mark-sweep GC である。最古と言っても現在でも広く使われており、Java の実装の一つである Kaffe や、C 言語のための GC である Boehm GC などが採用している。

各オブジェクトに、1 ビットの マークビット を割り当てる。オブジェクトのヘッダに含めるのが簡単である[7]。

一回の GC 処理は、探索を行なう マークフェイズ と、ゴミ領域を解放する スイープフェイズ からなりたつ。

マークフェイズ　GC 開始時には全オブジェクトのマークビットは 0(つまり白) である。そしてルートオブジェクトから到達可能なオブジェクトのマークビットを、次々に 1 にしていく。GC 処理の wave front(つまり灰色オブジェクトたち) を覚えるために、マークスタック というデータ構造を用いるのが一般的である[8]。

[6]通常、allocate 試行 → メモリ不足を発見 → GC 起動というタイミングで GC が起こる。このためユーザプログラムから見ると、allocate にかかる時間が時々とても長くなるように見える。例えば、400MHz Ultra SPARC のマシンで 20MB の live オブジェクトを探索するのに約 700ms かかる (mark-sweep の場合)。これが GUI プログラムの最中に起るのは良くない

[7]他の方法として、GC 中のメモリアクセス局所化のために、ヒープ外にマークビット専用の配列を用意する処理系もある

[8]関数の再帰呼びだしなどで記述するよりメモリ効率がよい

```
;;; mark-phase
push all roots into mark-stack
while (mark-stack is not empty)
    o = pop(mark-stack)
    for i = 0 to sizeof(o)
        c = ith field of o
        if (c is pointer) and (mark-bit(c) == 0)
            mark-bit(c) = 1
            push(c, mark-stack)
;;; sweep-phase
p = heap-bottom
while (p < heap-top)
    if (mark-bit(p) == 0)
        free(p)
    p = p + sizeof(p)
```

Figure 3: 単純な mark-sweep GC のアルゴリズム

スイープフェーズ　　ヒープ中の全オブジェクトのマークビットを調べ、0 であるオブジェクト (白) を free する。

より詳細なアルゴリズムは図 3 のようになる。途中、マークビットが 0 であるか調べてから 1 にしている。これによって、複数箇所から参照されているオブジェクトであっても、正しく一度だけマークされる。

スイープフェーズで行なう解放処理は、手動メモリ管理で説明した free 関数と同様に、フリーリストを用いる。プログラムによっては fragmentation が起る。

4.2 Copying GC

Copying GC アルゴリズムは、生きたオブジェクトをすきまをつめながら移動するというものである。これにより、mark sweep や reference counting で問題となる fragmentation が全く起らないという利点がある。多くの Scheme や ML 処理系で採用されている[9]。

ここでは最も単純な、ヒープを 2 等分する方式を紹介する。この方式では、一度に使えるヒープは 2 等分のうち片方のみである。使用中の片方が埋まった時点で GC を起動する (図 4)。そして到達可能なオブジェクトたちを、そっくりもう片方のヒープに隙間をつめながらコピーする。コピー元のヒープを from-space、コピー先のヒープを to-space と呼ぶ[10]。コピーが終った時点ではすでに from-space には生きたオブジェクトの残骸とゴミしか残っていないので用無しとなる。GC 終了後、ユーザプログラムは to-space を用いて動作を続ける。このように、GC の度に 2 つのヒープの役割を交替しながらシステムは動作する。

この方式を実装しようとするとき、いくつか注意が必要である。例えばコピー後のグラフは必ず to-space 内で完結し、from-space にポインタがのびていてはいけない。また、複数箇所から同オブジェクトが参照されている場合にも注意する。図では b と c が d を参照しているので、to space においても b' と c' は同オブジェクトを参照しなければならない。

アルゴリズムを 3 色モデルを用いて考えると以下のようになる[11]。

- 白 ··· まだコピーされていない (from-space にしかない) オブジェクト
- 灰色 ··· それ自身はコピーされたが、from-space のオブジェクト (白オブジェクトや他のオブジェクトのコピー前) を指している可能性がある。
- 黒 ··· それ自身はコピーされたし、from-space を指している可能性もない。

[9] 自作コンパイラ用に実装するのであれば、とりあえずこの方法をお勧めします。Mark-sweep GC はフリーリストなどの実装に手間がかかるので

[10] ヒープを 2 等分する代わりに、各ページごとに from-space/to-space の区別をする処理系もある

[11] コピー前とコピー後 (例えば A と A') のセットに対して、色が一つ決まると考える

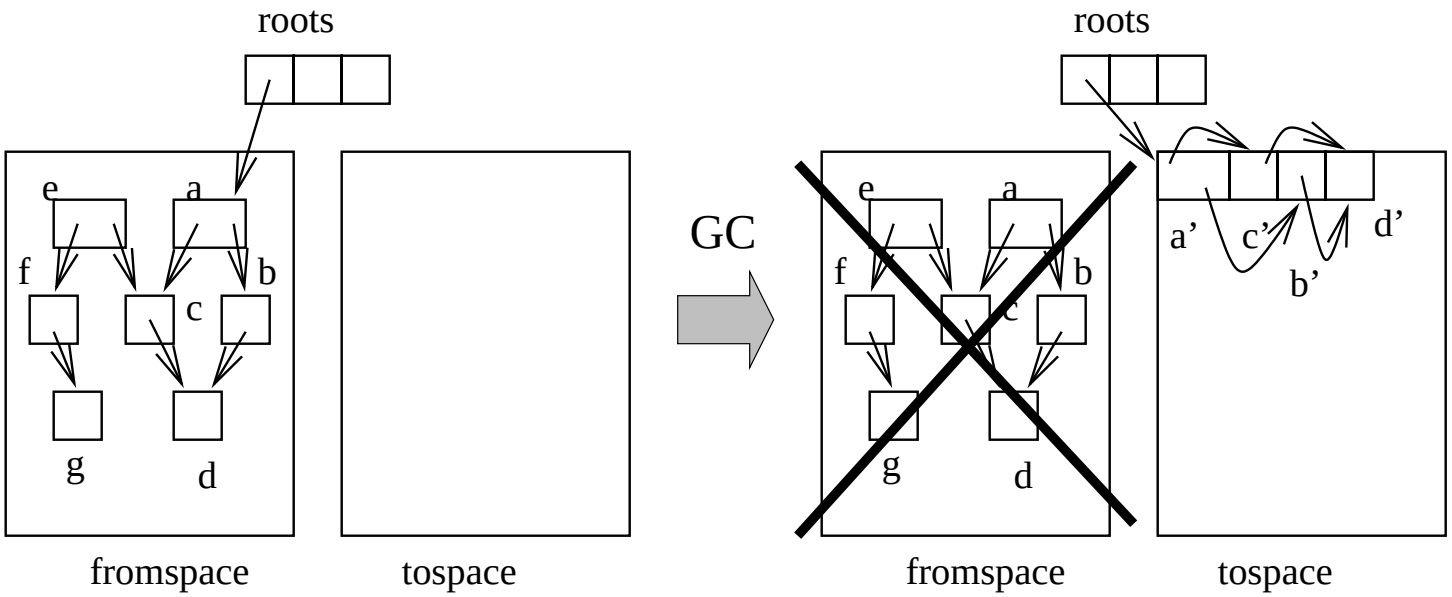

Figure 4: Copying GC の動作。From-space 中の到達可能なオブジェクトを to-space へコピーする

上のような条件を満たしていれば、深さ優先でも幅優先でも良い。ここでは、copying GC の代名詞になっている、Cheney の幅優先アルゴリズム [3] を紹介する。これは再帰呼び出しもマークスタックのような領域も使わない。ヒープ以外のメモリ使用量は O(1)!

GC 開始時に、ルートオブジェクトから直接参照されるオブジェクト (図では a のみ) を to space にコピーする。この時点で a は灰色と考えられる。このアルゴリズムは再帰探索のために 2 つのポインタを用いる。

- scanned··· 初期値は to-space の先頭。Scanned より手前は全て黒オブジェクトである。
- unscanned··· 初期値は、図の場合 (to-space の先頭) +sizeof(a)。Scanned と unscanned の間は全て灰色オブジェクトであり、unscanned 以降は空き領域。

GC は以下の処理を繰り返す。Scanned が指す先頭オブジェクト o を取り出し、o の子オブジェクト達のうち、未コピーであるものをアドレス unscanned にコピーする (o の子オブジェクト達を灰色にする)。その度に unscanned を進める。同時に、o 中のポインタがコピー先をさすように書き換え、scanned を進める (o を黒にする)。

Scanned が unscanned に追いついたら、GC を終了する。

さて、同じオブジェクトへの複数参照に対応するためには、from-space 中の d を見ただけで、「これはすでにコピー済みであり、コピー先は d' である」ということが分かる必要がある。そのために、灰色化の時点で d に *forwarding tag* (コピー済みであることを表す)、*forwarding pointer*(コピー先を教えてくれる) を書き込んでおく。

4.2.1 Copying GC の性質

Copying GC の興味深い性質は、コピーするときにオブジェクトの空きをつめるため、fragmentation が発生しないということである。よって手動 free を用いる場合 (2 章) や mark-sweep(4.1 章) のような、フリーリストを用いた複雑な allocation が必要ない。代わりに、空き領域の先頭と、ヒープ境界を表すポインタを管理しておけば充分である。allocate のたびに、巨大で唯一の空き領域を順番に食い潰していけばよい。これを linear allocation と呼ぶ。これはフリーリストを用いる確保処理よりも速い。

また、手動メモリ管理において free にかかる時間というのは、大まかにはゴミの量に比例する (100 個のオブジェクトを解放するには 100 回 free を呼ぶので当たり前)。一方、copy GC を一度行なうのにかかる時間は、ゴミの量ではなく、生きたオブジェクト量に比例する。この違いのため、プログラムの性質によっては手動メモリ管理よりも GC を使った方が速い、という場合すら出てくる (後に議論)。

	手動 free	reference counting	tracing GC
プログラム書きやすさ	×	○	○
停止時間	○	○	×
トータル実行時間	○	×	△ or ○
省メモリ性	○	△	×

Table 2: 各メモリ管理方式の利害得失

4.3 Mark-Compact GC

Mark-compact GC は、mark-sweep GC に似ているが、fragmentation が起らない方式である。Mark phase では mark-sweep GC と同じことを行う。その後の compact phase では、live object を全て、空きを詰めながらヒープの隅の方へ移動する。ポインタのつけ変えも必要である。

ヒープを 2 倍必要とする copy GC とは違って、ヒープは 1 つでも良い。このため、mark-sweep GC と copying GC の両者の良い所を取っているように見えるのだが、GC 中に全 live オブジェクトを複数回スキャンするのはやはり重く、mark-sweep/copying に比べ一度の GC 時間が 2 ～ 4 倍程度長くなるようだ。

5 GC の利点／欠点

手動メモリ管理、reference counting、tracing GC の利害得失を、表 2 に示す。Tracing GC の欠点は以下のようになる。

1. ユーザプログラムの停止時間が長くなる。この理由は、手動 free/reference counting と違い、ゴミの検出と解放を一気に行なうためである。これは GUI プログラムなどで特に問題となる。
2. トータルの実行時間も、多くの場合、長くなる。
3. メモリを無駄に消費しがち。これは、いらない領域をまとめて解放を行なうためである。

ここで、1. の停止時間の問題と 2. のトータル時間の問題は明確に区別しておく必要がある。図 5、6 は、手動 free と Tracing GC のそれぞれにおいて、ユーザプログラム実行中の時間と消費メモリ量の関係のイメージ図である。図の灰色の部分はメモリ管理のために使われる時間を示す。

停止時間が重要なのか、トータル時間が重要なのか？はユーザプログラムの性質によって違う。停止時間はゲームや GUI プログラムでは重要だが、計算主体のプログラムではトータル時間の方が重要である。Incremental GC という (6.1 節)、停止時間を短縮できる手法もあるのだが、その一方でトータル時間が犠牲になってしまう。

5.1 Copying GC v.s. 手動 free

まじめな比較ではないが、copying GC と手動 free の実行時間の比較を行なってみる。遅いといわれる GC だが、プログラムによっては手動 free より速くなってしまう場合もある、ということを示す。

以下の議論で、live object の量を L, ヒープ全体の大きさを M とする。

ここでは、メモリ解放のコストパフォーマンスを、「解放した領域の大きさ / 解放のために費やした時間」として定義して比較する。この値は大きいほどトータル実行時間が短く、良い。手動 free の場合は、1 度の free の度にオブジェクトが一つ解放される。コストパフォーマンスは M, L に関係なく、定数 f であるとしておく[12]。

Copying GC の場合、生きたオブジェクトを全てコピーすれば終りなので、GC 時間は L に比例し、cL となる (c は定数)。cL の時間をかけて $M/2 - L$ の量を解放するのでコストパフォーマンスは $\frac{M/2-L}{cL}$ となる。

ここから、平均的に M が大きく L が小さいユーザプログラムであれば、手動 free よりも GC の方が早い (停止時間でなくトータル実行時間が) 場合は充分ありうると言える。L が小さいというのは、大量にオブ

[12] ところで、GC を嫌う人の中には「手動 free にかかる時間がゼロではない」ことを忘れている人もいる

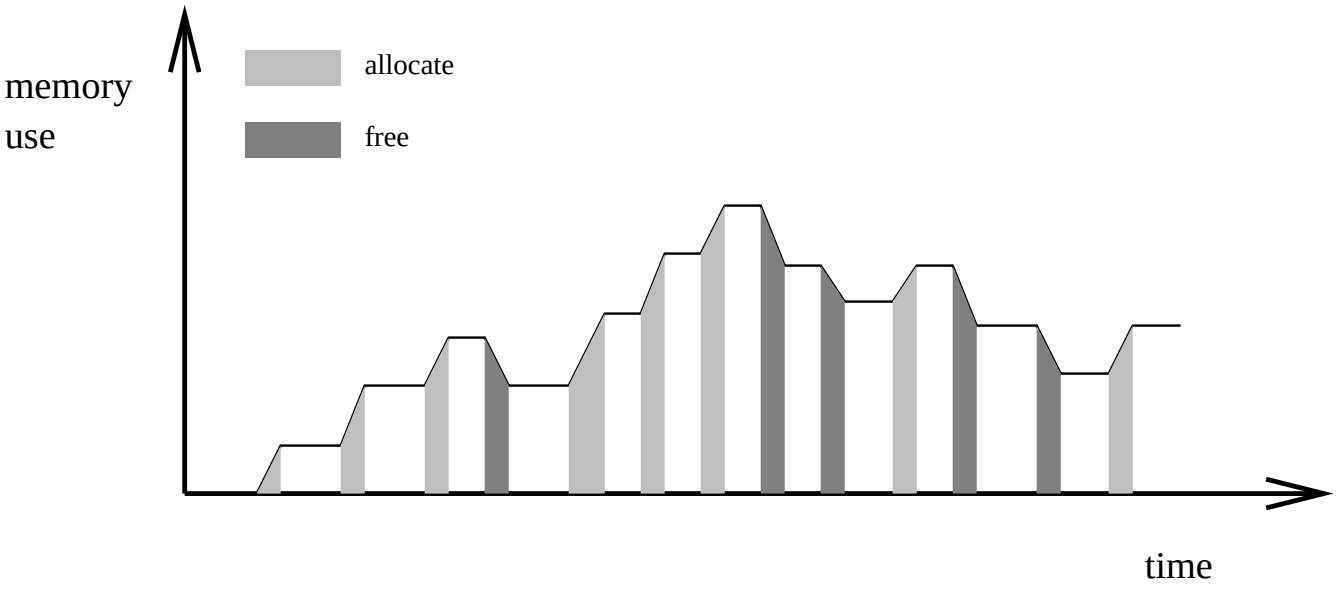

Figure 5: 手動 free の場合の時間とメモリ使用量の関係 (イメージ図)。Free が行なわれた時点ですぐにその領域は再利用可能になる

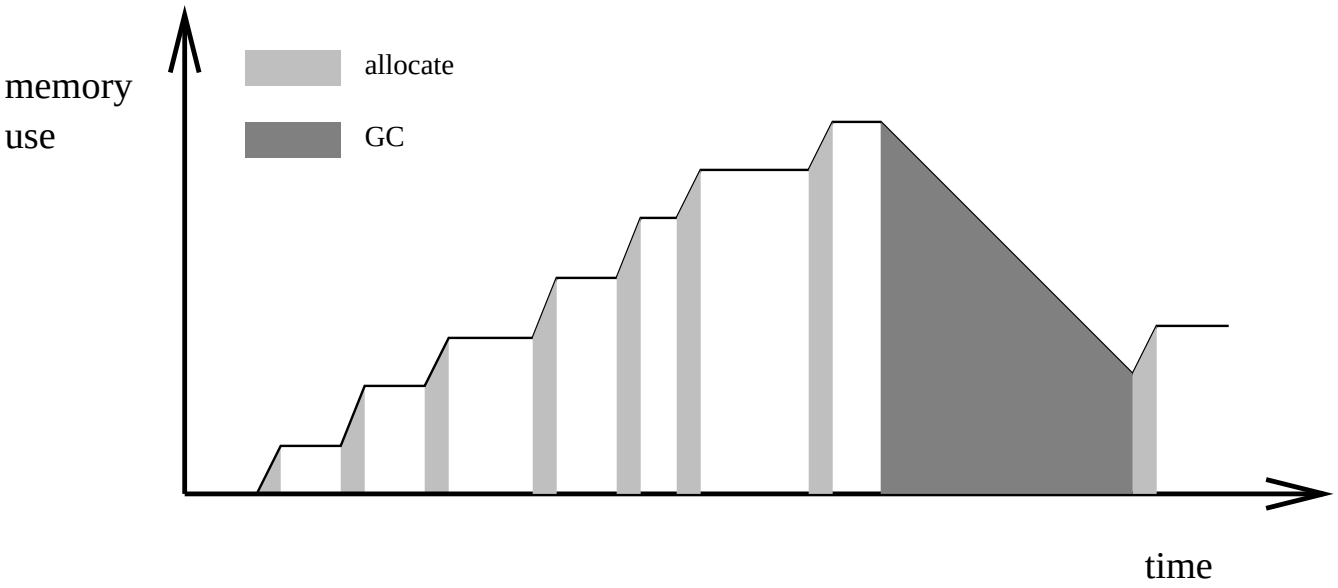

Figure 6: Tracing GC の場合の時間とメモリ使用量の関係 (イメージ図)。GC が起こって初めて領域の再利用が可能になる

ジェクトが確保されてもすぐに使い終り、生き残るものは少ないという状況である。特に、Scheme や ML などの関数型言語ではそのようなプログラムは比較的多い。もちろん逆に、オブジェクトがほとんど生き残ってしまう ($M/2 - L$ がほとんどゼロ) プログラムもたくさんあり、その場合に GC は悲惨なことになる。

5.2 Mark-sweep v.s. copying

今度は mark-sweep GC と copying GC の比較を行なう。持っていきたい結論は「古い教科書には "copying の方が優れている" と書かれているが、アルゴリズムの工夫を行なうと、そう一概には言えなくなった」である。

まず mark-sweep の 1 回の GC 時間を考える。マークフェイズでは live object を全てマークするので、時間は L に比例する。スイープフェイズではヒープの全オブジェクトのマークビットを調べるので M に比例する。以上より、mark-sweep の GC 時間は $mL + sM$ となる。

一方 copying GC の場合、live object を全てコピーすれば終りなので、時間は L のみに比例し、cL となる。

よって「M に実行時間が依存する mark-sweep よりも、L だけに依存する copying の方が有利である」というのが通説だったのだが、最近の mark-sweep GC の実装は lazy sweeping という改良が行なわれており、一概には成り立たない。

Lazy sweeping とはアイデア的にはごく簡単で、マークフェイズが終ったらもうユーザプログラムを再開してしまう。その後、ユーザプログラムによる allocate 要求がなされて初めて、マークビットの検査を少しづつ行なっていく (そして空き領域を見つけた時点でユーザに渡したり、フリーリストにつないだりする)。Lazy sweeping によって、allocate のオーバヘッドが増えてしまうのだが、(1)GC の停止時間が短くなり、mL と考えることができる、(2) 一度に行なうスイープよりも局所性の面で優れている、という利点がある。

さてこの状況でもう一度、定数 m と c に注目しつつ、mark-sweep GC(停止時間 mL) と copying GC(停止時間 cL) を比べてみる。Mark-sweep GC が一つの live object に対して行なう仕事は、オブジェクト全体 read が 1 回、ビット変更 (マークビット)、スタックへの push/pop が 1 回づつである。一方、copying GC の場合は、オブジェクト全体 read が 2 回、オブジェクト全体 write が 1 回、さらにポインタ付け替えが加わる。よって $m < c$ であり、停止時間については mark-sweep の方が有利と考えられる[13]。

大ざっぱにまとめると、

- 1 度の GC の実行時間は mark-sweep GC(+lazy sweeping) の方が有利
- Allocate のコストは copying GC の方が有利

6 Tracing GC の改良技法

何度も述べているように、tracing GC の欠点は、一気に生きた全オブジェクトを探索するために停止時間が長くなることである。それを改善するための方法である、incremental GC と generational GC を紹介する。これらは単独のアルゴリズムではなく、これまでのアルゴリズムの改良方法である。「Generational mark-compact GC」「Incremental generational copy GC」などがありうる。

6.1 Incremental GC: ちょっとづつ GC

Incremental GC は GC の探索処理を細切れにして、ユーザプログラムと交互に動かす。これにより一回あたりの停止時間を短くするアプローチであり、リアルタイム性の必要なプログラムに適している。交互で動かす最もメジャーな方法は、allocate 処理を行なうときに、こっそり少しづつ GC を進めるというものである。以下では、mark sweep GC を incremental 化する場合について述べる。

Incremental GC を実装するには、いくつか問題点をクリアしなければならない。

- GC 処理のスケジューリング。これまでのように、ヒープが満杯になったら GC を始める... のでは間に合わない。GC の再帰探索が一通り終るより先に、メモリが尽きるという状況は極力避けなければ

[13] この傾向は、ヒープ中に大きいオブジェクトが多いと顕著になる。Copy GC を改良し、大きいオブジェクトを (mark-sweep 的に管理される) 第 3 のヒープに確保することによってこの問題を緩和することはできる

ならない。ちなみに、GC の仕事を少し行なったからといって、メモリをすぐに解放してくれるわけではない (=省メモリ性は相変わらず悪い) ことに注意 (図 5 と図 7 の違いに注意)。

- 後述の write barrier が必要になる。

Incremental GC は普通の GC と異なり、GC 最中にユーザプログラムが勝手に動作するという特徴がある。ユーザプログラムが行なう「オブジェクトの書き換え」が GC を破綻させてしまう可能性がある。

そのような例を図 8 に示す。マークフェイズの途中で a が黒になった時点 (b, c はまだ白) で、ユーザプログラムが動き出すとする。ユーザプログラムが a から c へのポインタを作り、b から c へのポインタを消す (NULL などで上書き) とする。この状況で GC が再開すると b まではマークを行なうが、c のマークは一切されないため c は誤って解放されてしまう。

このような状況を防ぐためには、ユーザプログラムがオブジェクトを書き換えたことを GC が知る必要がある。このためにコンパイラが、オブジェクト書き換えの際に必ず GC に知らせる (というマシンコードを吐く) ようにする。このように、オブジェクトへの書き込みの際に特別な処理を行なう機構を write barrier という[14]。

さて write barrier がどういう処理をするかというと、流儀は大きくわけて 2 つある。

- 黒→白のポインタの生成を問題視する方法 (incremental-update)。黒オブジェクト a に白オブジェクト c へのポインタを書き込む時点で、a か c のどちらかを灰色にする。

- ポインタの消滅を問題視する方法 (Snapshot-at-beginning)。b から c へのポインタを消す時点で、消される対象の c を灰色にする。

以上のどちらかを実装すれば良い[15]。

これまで incremental mark sweep GC について述べたが、incremental copying GC も存在する。一方、mark-compact GC を incremental 化することはなかなか困難である。

基本的には、incremental GC は停止時間を短くするのが目的であって、GC の仕事の量を減らそうとするものではない。このため、非 incremental GC と比べてトータル実行時間は短くならない。逆に仕事の切替えオーバヘッドや write barrier の分、遅くなるだろう。Incremental を使うべきかどうかはプログラムの性質に応じて決める必要がある。

6.2 Generational GC: 若いやつらを優先する GC

オブジェクトの寿命について、多くのプログラムで観測されるある傾向が知られている。それは「古くから生き残っているオブジェクトはそのまま生き残りやすく、新しく確保されたオブジェクトほどすぐにゴミになりやすい」(*) という傾向である。GC の実行効率は生き残るオブジェクトが少ないほどよいのだから、古いオブジェクトはほうっておいて (無条件に生き残るとみなして) 新しいオブジェクトを重点的に探索することによって利益が得られそうである。

Generational GC (世代型 GC) は、新しめのオブジェクトのためのヒープと、古めのオブジェクトのためのヒープを用いる (図 9)。Copying GC を基にするのであれば、図には示されていないが、新世代のヒープと旧世代のヒープのそれぞれが from/to に 2 等分される。

オブジェクトはまず新世代ヒープに確保される。新世代ヒープが一杯になったら、ヒープ全体の GC を行なう代わりに、新世代ヒープだけを対象とした GC(minor collection) を行なう。Minor collection を繰り返した結果、ある程度以上長寿命なオブジェクトが見つかったなら (例えば GC が n 回起こる間生き残ったら)、そのオブジェクトは旧世代ヒープに移される (殿堂入り、または promote と呼ぶ)。

さて、minor collection においては旧世代ヒープ中のオブジェクトをスキャンする必要がない (旧世代オブジェクトは無条件に「生き」とみなされる)。図ではオブジェクト b 以降の探索は行なわない。このため、minor collection はヒープ全体を探索するよりも速く終了する。

[14] コンパイラの協力が得られない場合でも write barrier を実現することは可能である。大抵の OS ではメモリ領域のアクセス権限を変えることができる (UNIX の mprotect() など) ので、ヒープを write 不可にしておくことによって、write をつかまえることができる

[15] 前者の方が write barrier のコストを最適化しやすいのだが、探索終了判定に時間がかかる問題がある。筆者の論文 [5] でも議論

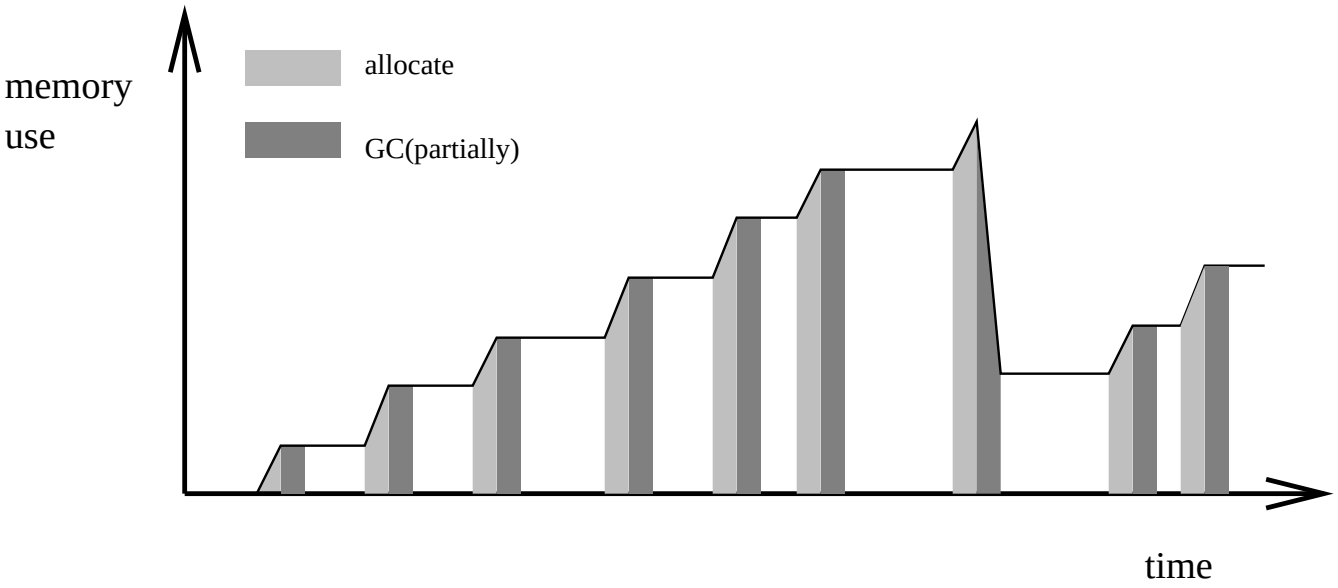

Figure 7: Incremental GC の場合の時間とメモリ使用量の関係 (イメージ図)。GC 処理を少しづつ進め、一通り終った時に始めて領域が解放される

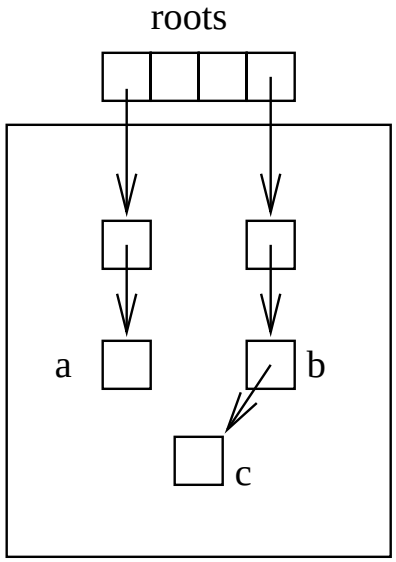

Figure 8: Incremental GC には write barrier が必要なことを示す図。マークの途中で、ユーザプログラムが a から c のポインタを作り、b から c へのポインタを消すと、そのままでは破綻する

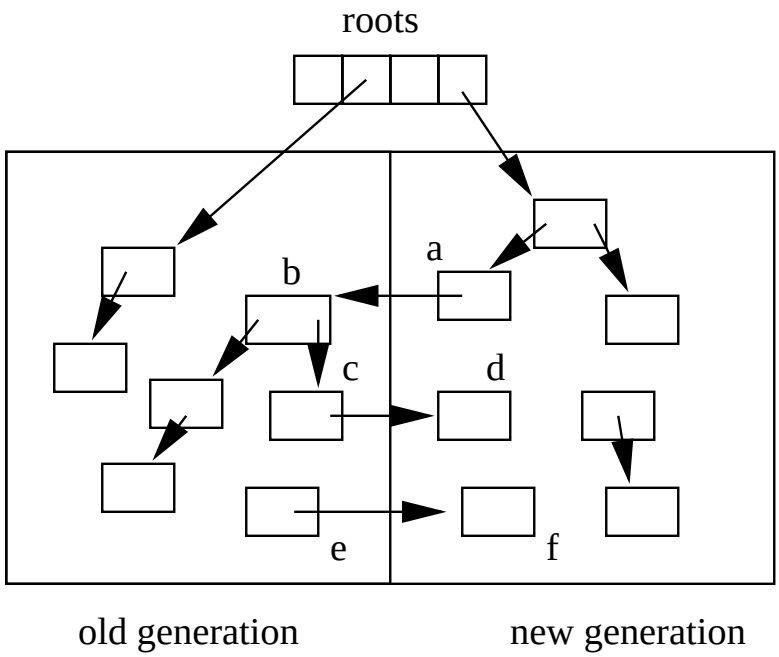

Figure 9: Generational GC のヒープ。新世代のみを探索する場合、$c \to d$, $e \to f$ のポインタに注意する必要がある

入れ替わりの激しい新世代ヒープと対照的に、旧世代ヒープは殿堂入りオブジェクトによってゆっくりと満たされていく。やがて旧世代ヒープさえ一杯になったら、そこではじめて、ヒープ全体を探索する GC(major collection) を試みる。

Generational GC の長所は以下の通りである。

- Minor collection の実行時間はヒープ全体の GC に比べ短いので、プログラムの停止時間を改善できる。ただし、major collection が起こってしまえば長く停止してしまう。
- 上述の (*) という傾向が成り立つようなプログラムにおいて、5.1 節で述べた「メモリ解放のコストパフォーマンス」を改善することができる。つまり、トータル実行時間を改善できる。ただし (*) の逆の傾向にある (古いオブジェクトの方が先に死にやすい) プログラムでは逆効果となりうる。

Generational GC を実装する際には、以下のような落し穴に気をつける必要がある。Minor collection を行なう際に、旧→新のポインタに注意が必要である。たとえば $c \to d$ のポインタに気付かないと、到達可能なオブジェクト d を間違って解放してしまう。この図の場合、GC は新世代へのポインタを持っている c と e を全て覚えておき、minor collection はそれらもルートと見なす。この目的のために (incremental GC の節で説明したような) write barrier を用いることによって、普段から旧→新のポインタができたかどうかを全て覚えておかねばならない。

7 その他の話題

これまで述べたこと以外にも、GC の研究は色々な方向に向かっており、いくつか紹介する。

本稿には含まれていないのだが、finalizer, weak reference なども、Java プログラマに関係する話題として重要である。

7.1 Conservative GC: C 言語でも使える GC

これまで、GC は常にあらゆるメモリ内容の型 (ポインタか否か) を知ることができることを前提としていたが、C 言語のように、実行時に型が分からない環境で GC を行なう処理系も存在する。中でも Boehm GC[2] が有名であり、いくつかの著名プログラムも利用している (フリーの web ブラウザ w3m, gcc 3.x の一部な

ど) [16] 。これは C/C++プログラムから利用できる mark-sweep GC ライブラリで、このライブラリが提供するメモリ確保関数 (GC_malloc()) を呼ぶことによって GC 機能を利用できる。

再帰探索中に見つけた値のポインタ判定をどうするかというと、アイデアは簡単で、とりあえずポインタだと思ってみるという方針をとる (conservative GC(保守的 GC) と呼ばれる)。そしてヒープ領域以外を指していたり、空き領域のはずの箇所を指している場合は、明らかにポインタでないとして除外する。

もちろんこの方針を取る限り、ユーザプログラムがポインタのつもりで使っていない値を、GC がポインタだと思ってしまう (false pointer) 可能性は避けられない。その場合でも、少し省メモリ性が落ちるだけで、プログラムの続行にほぼ支障はないと報告されている[17]。

余談: 型が分からない環境で copying GC を実装するのは、少なくともそのままでは無理である。Copying GC ではメモリ内容の update(fromspace を指すポインタが tospace を指すように書き変わる) が起るためである。非ポインタなのに勝手に GC に値を書き変えられてしまっては、ユーザプログラムを正しく続行することができない。値の型が一部分かり、一部分からない状況で copying GC を行なう研究もなされている (mostly copying GC [1])。

簡単に GC 対応できないプログラムもあるものの (既存のコンパイル済みモジュールが malloc を直接使っていると困る)、プログラムの開発コストを大幅に減らすことができる可能性があるので、C/C++を使っている人はぜひ一度試してみてほしい。

7.2 マルチプロセッサマシンのための GC

複数のプロセッサが載ったマルチプロセッサマシンが、特にサーバ分野で増えてきている。そのようなマシン上で、遊んでいるプロセッサを用いて GC を効率的に実行する研究がされている。

マルチプロセッサを用いた GC は大きく 2 つの方針に分けられる。

- concurrent GC: GC 専用のプロセッサはいつも GC 処理を行い、それ以外のプロセッサ上のユーザプログラムと同時に動作する。メモリの飢餓状態 (GC による解放がユーザプログラムによる allocation 要求に間に合わない) さえなければ、ユーザプログラム停止はほとんど起らない。Incremental GC の応用と考えられ、要 write barrier。

- parallel GC: 全プロセッサは普段ユーザプログラムの実行を行い、メモリ不足が起った時点で、全プロセッサとも協調的に GC 処理を行う。探索の並列化によって、GC の速度向上を目的とする。Write barrier 不要だが、基本的 GC と同じで停止時間を食らう。

筆者は、parallel かつ concurrent な GC を、Boehm GC を基にして実装している [6, 5]。プロセッサ数が多くなったらそれに伴い GC 速度が向上することと、write barrier による影響をなるべく小さくすることを主眼においている。

7.3 分散環境の GC

CORBA や Java RMI など、複数のコンピュータ (当然メモリ領域を別々に持っている) を用いた協調的なプログラムを作成する環境が広まりつつある。このような環境では、分散プログラミングを楽にするために、遠隔参照 (remote reference) という機能がある。これによって、ネットワーク越しにあるオブジェクトに対して、普通のオブジェクトとほぼ同じようにアクセスできる。

そうなると、各コンピュータ上でこれまでと全く同じ GC を動かしていてはいけない。「あるオブジェクトが内部的にはゴミに見えるが、他のコンピュータから使われているために、捨ててしまってはバグになる」という状況が起り得るからである。

分散環境の GC は大きく 2 種類に分けられる。

- 分散 reference counting: remote reference に限って、reference counting(3 章) を用いる。一方、各コンピュータ内の GC には tracing GC を用いるのが普通。Java RMI でも使われているそうだ。コンピュータ間にまたがったサイクル状のゴミを回収できない欠点がある。

[16] w3m 作者の伊藤 彰則さんによる解説ページもある (http://homepage2.nifty.com/aito/gc/gc.html)

[17] 筆者の経験からすると、consservative GC は double 浮動小数を多用するプログラムに弱いようだ。特に double の下位 word が false pointer となりやすい

- 分散 tracing GC: 内部でも外部でも tracing GC を用いる手法である。どんなサイクル状のゴミでも回収できるが、GC の度に全コンピュータの同期を必要とするのが欠点とされている。

Shapiro らのサーベイ論文 [10] は様々な分散 GC アルゴリズムを紹介している。

7.4 静的自動メモリ管理

型推論などを用いて、コンパイラ主導によって自動メモリ管理を行なう処理系もある。

- Escape analysis ([9] など)　　プログラム中のメモリ確保個所について、そこで確保されるオブジェクトが関数の外に飛び出すか否か判定する。寿命が関数内であれば、ヒープの代りにスタックに確保する。判定失敗すればヒープに確保するので、最終的には GC が必要。しかし、ヒープ使用量を減らせれば GC コストも減らせる。

- Region inference ([11] など)　　静的推論によってオブジェクトの寿命を判定し、メモリの解放コードをプログラム中に埋め込む。リージョン推論では解放できるが tracing GC では解放できないゴミも、その逆もあり得るそうだ。ML kit という処理系で実装されている。

- linear type ([8] など)　　各オブジェクトの作成後、それらが使用される (かもしれない) 回数を型推論によって求める。1 回のみ使用されるオブジェクトが見つかれば、その唯一の使用直後に free することができる。

8 GC を実装したい人へ

8.1 ヒープサイズはどれだけにすべきか？

本稿ではヒープのサイズが前もって決まっているとして、「ヒープが満杯になったら GC」と単純に述べてきたが、実際にはどれくらいのヒープサイズがふさわしいかを決めるのは難しい。

実際の処理系では、プログラム開始時には小さいヒープサイズにしておき、GC を行なっても足りないことがわかったら、そこでヒープを拡張するということが行なわれている。Sun Java VM の場合、-Xms(初期ヒープサイズ)、-Xmx(最大ヒープサイズ) オプションによりヒープサイズが調節可能である。

8.2 コンパイラによる援助

これまでに説明してきたように、GC はヒープやスタックの中身を検査する。そのために GC は、プログラム実行中に以下のような情報を取得できる必要がある。これらのうち多くを、コンパイラの援助により教えてあげる必要がある。

1. オブジェクトを指すポインタがあるとき、そのサイズが分かる (全ての GC)。
2. オブジェクトの途中を指しているかもしれないポインタがあるとき、オブジェクトの先頭位置が分かる (conservative GC では重要)。
3. GC 起動時にスタックの範囲が分かる (tracing GC, 場合によっては reference counting も)。
4. 全ての大域変数はどこに格納されているか分かる (tracing GC)。
5. GC 起動時に各レジスタが生きているか否か、生きているならポインタか否か分かる (tracing GC)。
6. オブジェクトやスタック, 大域変数に含まれる値を見たとき、それがポインタか否か分かる (全ての GC)。
7. ユーザプログラムによるオブジェクトへの書き込みが分かる (write barrier) (reference counting, incremental GC など)。
8. ユーザプログラムによるオブジェクトの読み込みが分かる (reference counting, 一部の incremental GC)。

このうち 1. に関しては、allocate+手動 free においても必要だった。これを実現するには allocate 時に header にサイズを書いておくなどする。

5., 6., 7., 8. を実現するためには、コンパイラが余分なマシンコードを付け加えてあげる手法が一般的である[18] 。

以下では 6.(ポインタ型の区別) に注目し、これを GC に教える方法をいくつか紹介する。

タグ付け　あらゆる値に型を示すタグをつける、というのが最も実装の簡単な方法だろう。以下、1 ワード=16bit としてタグ付けの例を示す。

スタック、大域変数、オブジェクトに含まれる全ワードの下位 1bit をタグとする。1 ならオブジェクトへのポインタ、0 ならそれ以外 (整数など) とする。値を示すのに 15bit しか使えなくなるので、以下のようにする。

- オブジェクトへのポインタは必ず 2 の倍数とする。(実際のアドレス+1) を、「型付きの値」とする。
- 整数については、(実際の値 * 2) を、「型付きの値」とする。

このときコンパイラは、オブジェクト読み書き命令や、整数演算命令などが正しく動くようにしてやる必要がある。例えば、r3 に r1 と r2 の積を計算するとき、タグがなければ `mul r1, r2, r3` で済む。しかしタグがあると、1bit 分調整する必要があり、`mul r1, r2, r3; shr r3, 1` のようになる。

もっと細かく型を区別したい場合 (実数, char, boolean など) には 2bit または 3bit のタグが必要である。

スタックマップなど　しかしタグを付けたくない処理系もあるかもしれない (整数の精度が 16bit 必要なときなど [19])。その代りに、スタックフレーム中の各ワードの型を表す表 (スタックマップ) をコンパイラが作成する方法がある。スタックフレームは 1 関数の中でさえも、プログラムの動作によって刻々と型が変っていく。あらゆる瞬間の表を作るのは不可能なので、関数呼び出しのある瞬間/メモリ確保を行なう瞬間の表を作る。ヒープ中オブジェクトについては、オブジェクトのヘッダに、各ワードの型を表す表へのポインタを持っておく。

Tracing GC で必要となる 5.(レジスタの生死情報) についての基本アイデアは以下のようなものでなる。GC が起りうるタイミング (典型的にはプログラム中のメモリ確保が起る場所) に対してそれぞれ、レジスタ生死情報の表たちを、コンパイラが余分に出力しておく。GC は実行スタックをさかのぼって呼出元を得、正しい生死情報表を得る。

9　GC 関連 web ページ

GC に関する話題をなるべく網羅的に触れるようにしましたが、まだまだ説明していないことがあります。以下のページが参考になるでしょう。

- http://www.cs.ukc.ac.uk/people/staff/rej/gc.html　[7] の著者
- http://www.hpl.hp.com/personal/Hans_Boehm/gc/　Boehm の conservative GC のページ。フリーでライブラリをダウンロードできます [2]
- http://www.cs.utexas.edu/users/oops/　Paul Wilson グループ, [12][13] など
- http://www.yl.is.s.u-tokyo.ac.jp/gc/　筆者らによる並列/分散 GC ページ。マルチプロセッサ向け GC 処理系をフリーで公開中。

[18] 例外は、C 言語上で GC を行なうような場合 (7.1 節)
[19] この理由で Java 処理系ではタグは使えない

References

[1] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Researach Laboraory, February 1988.

[2] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[3] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[4] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(7):522–526, July 1976.

[5] Toshio Endo and Kenjiro Taura. Reducing pause time of conservative collectors. In *Proceedings of ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 12–24, June 2002.

[6] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of ACM/IEEE Conference on High Performance Networking and Computing (SC97)*, November 1997.

[7] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley & Sones, 1996. ISBN 0-471-94148-4.

[8] Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26st ACM Symposium on Principles of Programming Languages*, pages 29–42, 1999.

[9] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127, June 1992.

[10] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the 1995 SIGPLAN International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, 1995.

[11] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM, 1994.

[12] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 SIGPLAN International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, 1992.

[13] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the 1995 SIGPLAN International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, 1995.