プログラミング入門
第 14 回「配列、ガーベージコレクション」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

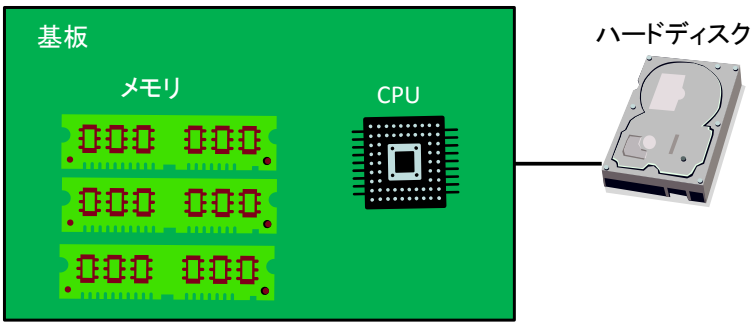
Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

5 レジスタ計算機での計算

今回は、プログラミング言語を使う上で、常識的に知っておいたほうが良いことについて勉強します。まず、プログラミング言語で必ずといっていいほど使われる(けど、この講義では今まで一度もでてこなかった) 配列(array)もしくはベクター(vector)と呼ばれるデータ構造について説明します。次に、C や C++以外のプログラミング言語を扱う上で必ず必要となるガーベージコレクション(英語では Garbage Collection, 通称 GC)について勉強します。ガーベージコレクションは効率的なプログラムを書くために必ず知っておかなければならないプログラミング言語処理系(コンパイラやインタプリタ)の基礎技術となっています。

5.3 記憶の割当てとガーベージコレクション

最初に配列について勉強しますが、その前にコンピュータアーキテクチャについてもちょっと勉強しておく必要があります。通常、コンピュータというのは CPU, メモリ、ハードディスクの三つからできていて、それらをつなぐ基板がある、という構成になっています(次の図参照)。



メモリにはプログラムや計算結果などがのっています。CPU はまずハードディスクからプログラムを読みだして、それらをメモリにのせます。続いて CPU はメモリにのったプログラムを順次読みだして、プログラムを実行します。CPU はプログラムに従って、数値やデータをメモリから読みだし何らかの計算をおこなって、その計算結果をまたメモリに書き込みということを行います。これを繰り返すことによって、実際の計算が行われるというわけです。計算をする CPU と、プログラムやデータ、計算結果を格納するメモリの二つが揃って初めてコンピュータとして様々な計算ができる、というわけです。

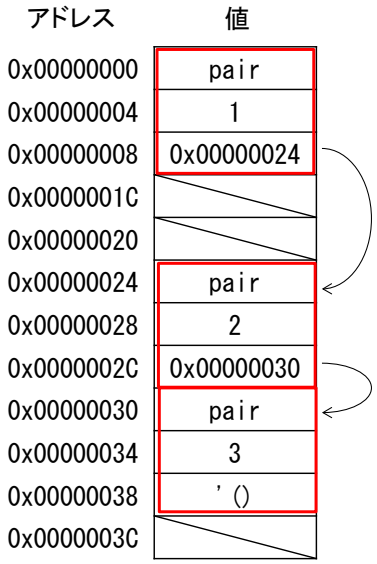
メモリの中はどうなっているのでしょうか。メモリは、デジタル情報が記憶されているところですが、アドレス(もしくは番地)と呼ばれる場所を指定して、その中に格納されている値を読んだり書いたりすることができます。各アドレスに対応する値はバイト単位で格納されています。1 バイトは 8 ビットから成っているので、一つのアドレスに格納できる値は 0～256 までの数字、16 進数でいうと、0～FF までの数字ということになります。例えば、メモリには次のようなかんじで値が格納されています。

アドレス	値
0x00000000	8E
0x00000001	12
0x00000002	03
0x00000003	9E
0x00000004	49
0x00000005	BE
…	…
0xFFFFFFFB	28
0xFFFFFFFC	0C
0xFFFFFFFD	0A
0xFFFFFFFE	29
0xFFFFFFFF	8A

これは 32 ビットコンピュータの場合(アドレスの表現に 32 ビット使っている)ですが、アドレスは 0 番地から始まって、FFFFFFFF 番地まであり、それぞれに 1 バイト格納されています。つまり、$2^{32} = 4{,}294{,}967{,}296$ バイト $\cong 4$ ギガバイト(GB)となるので、32 ビットコンピュータでは 4GB のメモリまで扱えるということになります。64 ビットコンピュータだと $2^{64} = 18{,}446{,}744{,}073{,}709{,}551{,}616$ バイト $\cong 16$ エクサバイト(EB)となり、16EB のメモリまで扱えます。ギガの 1024 倍がテラで、テラの 1024 倍がペタで、ペタの 1024 倍がエクサなので、64 ビット CPU ではだいぶ大きなメモリが扱えることがわかります。

5.3.1 ベクターとしてのメモリ

メモリは上述のように 1 次元の列になっている、ということを理解しておくことが重要です。例えば、(1 2 3)というリストをメモリに格納しようと思ったら例えば次のように格納することになります。



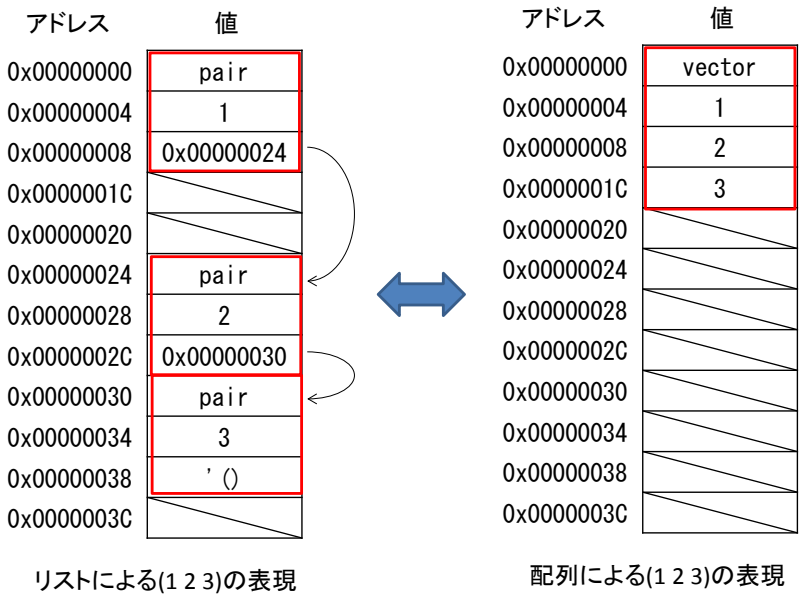
赤い枠がリストを構成する各ペアとなります。pair の下にペアの第一要素、第二要素が並ぶとします。0x00000000 がこのリストを指しているとすると、最初の要素が 1 で、次のペアは 0x00000024 で指されています。0x00000024 番地の値をみると、またペアがあって、その第一要素は 2 となっています。続いて、次のペアは 0x00000030 で指されているので、0x00000030 番地の値をみると、最後のペアである 3 と空リストがあることがわかります。実は今まで箱とポインタ記法で表現していたポインタというのはこのアドレスのことだったのです。

このようにしてリストはメモリ上で表現されているわけです。リストの利点として、上の図にあるように、1) 間に使われていないメモリ領域があっても良い、2)順番が前後しても大丈夫、3)リストを延長したいときは新しいペアをつくってリストをどんどん延ばすことができる、4) set-cdr!を駆使すれば、リストの途中や末尾に新しい要素を追加することも簡単にできる、ということがあります。しかし、その一方で、リストはn番目の要素の値を読むためにnステップの手続きが必要となり効率がよくありません。例えば、'(a b c d)というリストの 3 番目の要素を手に入れるためには、cdr→cdr→car の 3 ステップが必要となります。例えば、行列をリストで表現した場合、この行列の処理は非常に重たい計算となってしまいます。次のような行列があったとします。

$$\begin{bmatrix} \ddots & \vdots & \\ \cdots & a_{ij} & \cdots \\ & \vdots & \ddots \end{bmatrix}$$

a_{ij}の値を読むだけで$i-1$回の cdr と$j-1$回の cdr 操作が必要となります。

そこで、もっとコンパクトかつ高速にn番目の要素にアクセスできるデータ構造を考えます。下図の右のように連続したメモリ領域にデータを並べた構造ならn番目の要素に 1 ステップでアクセスできるようになります。



リストによる(1 2 3)の表現　　配列による(1 2 3)の表現

このようなデータ構造のことを配列(array)またはベクター(vector)と呼びます。配列がp番地にあるなら、最初の要素は$p+4$番地の値をみればよいし、次の要素は$p+8$番地の値をみればよいことになります。つまり、n番目の値をみたければ$p+4n$番地の値をみ

ればよい、というわけです。しかもリストによる表現と比べて非常にコンパクトになることがわかります。

Scheme では配列を実現するために次の関数が用意されています。

(vector? $\langle obj \rangle$) … $\langle obj \rangle$が配列なら#t を返す。そうでなければ#f を返す。

(vector $\langle a_1 \rangle \langle a_2 \rangle \cdots \langle a_n \rangle$) … $\langle a_1 \rangle \langle a_2 \rangle \cdots \langle a_n \rangle$を要素とする配列を返す。

(make-vector $\langle k \rangle$) … k個の要素を持つ配列を返す。

(make-vector $\langle k \rangle \langle fill \rangle$) … k個の要素を持つ配列を返す。ただし、各要素の値は$fill$となる。

(vector-ref $\langle vector \rangle \langle n \rangle$) … 配列$vector$の$n$番目の要素を返す。

(vector-set! $\langle vector \rangle \langle n \rangle \langle value \rangle$) … 配列$vector$の$n$番目の要素を$value$に設定する。

(vector-length $\langle vector \rangle$) … 配列 vector の大きさ(要素数)を返す。

いいことづくめの配列ですが、逆に欠点としては、1) 使用するメモリ領域が連続していないといけない、2) 順番を変えて並べられない、3) 配列を延長することはできない、4) 配列の途中や末尾に新しい要素を追加することができない、ということがあります。つまり、リストの長所が配列の短所となっているわけです。特に、3)や 4)に書かれてあるように、配列には新しい要素を追加することができません。つまり、配列は最初に大きさを決めてやる必要があり、最初に決めた大きさから変更することができません。しかし、配列はコンパクトに表現されるし、1 ステップでどの要素にもアクセスできるという長所があります。どういったときにリストで表現すべきで、どういったときに配列で表現すべきかよく考えて用いるようにする必要があります。

※ただし、最近は、延長できる配列、というのが利用できるプログラミング言語が増えてきていて、リストと配列の長所を兼ね備えたデータ構造としてよく用いられるようになってきています。

5.3.2 無限メモリーの幻想の維持

最初に説明したようにメモリというのは 1 次元の列となっていてアドレスを指定することでデータにアクセスすることができます。リストや配列などなんらかのデータ構造をつくるときは、そのたびにメモリの空き領域から必要なメモリを確保してメモリを利用するようになります。そして、使われなくなったメモリは開放して、メモリの空き領

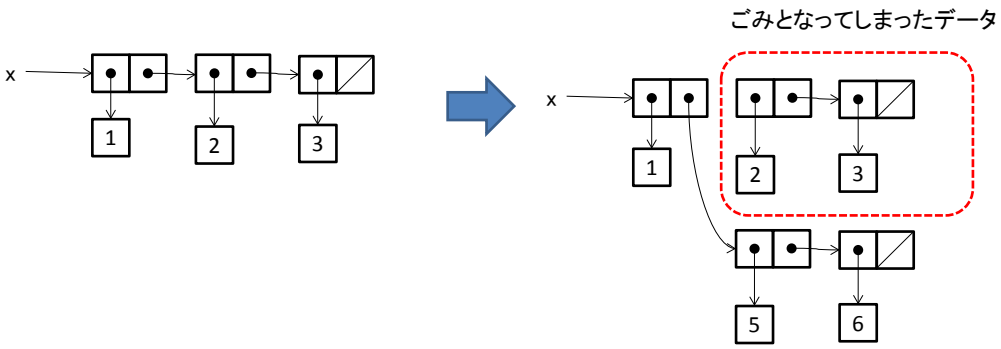
域にもどしてあげればよいわけですが、メモリは 1 次元の列となっているため、そんなに簡単に空き領域からメモリをとってきたり、返したりすることはできません。基本的に大きな問題が二つあります。一つは、開放すべきメモリ(つまり、もう必要ないメモリ領域)をどうやって検出するか、という問題で、もう一つは、メモリの断片化(fragmentation)と呼ばれるメモリの空き領域が断片化する問題があります。

まず、使われないメモリはどのように発生するのかみてみましょう。例えば、次のように実行したとしましょう。

```
(define x (list 1 2 3))

(set-cdr! x (list 5 6))
```

箱とポインタ記法で x の変化を表現すると次のようになります。



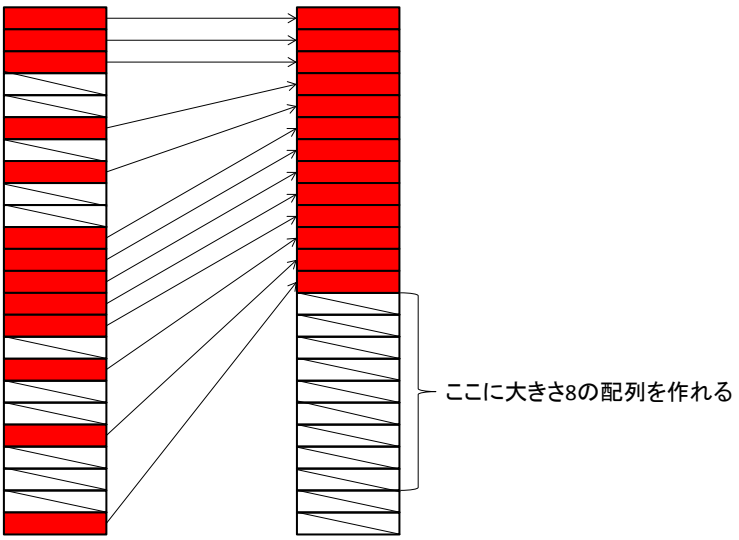
赤の点線で囲まれた部分は set-cdr!の実行の結果どこからも参照されなくなっています。このようにどこからも参照されていないデータは、どこからも使われることが決してない不要なデータなので、ゴミ(garbage)と呼ばれます。その他にも、例えば、

```
(reduce + 0 (filter odd? (range 0 n)))
```

と実行したとき、range で作成した整数のリストや、filter で生成した整数のリストは reduce を実行したあと、まったく参照されなくなります。こういった中間生成物や、変数の再定義のためどこからも参照されなくなったデータ、スコープを外れたためどこからも参照されなくなったデータはゴミとなってメモリ空間上に残ってしまいます。これらのどこからも参照されないデータが残り続けると、無駄にメモリが占有されてしまうため、ゴミをきちんと認識し、再び新たなメモリ資源として利用することが必要になります。ゴミを集めて再利用する仕組みは「ガーベージコレクション(英語では Garbage Collection, 通称 GC)」と呼ばれます。ほとんどの高級言語にはガーベージコレクションが実装されています。

もう一つの問題である断片化についても説明します。仮にメモリのどの部分がゴミであるかどうかわかったとしても、ゴミがメモリ上で断片化しているとメモリの利用効率が悪くなります。例えば、メモリが次のように使われていたとします。

例えば、この状態では大きさ 8 の配列をつくることができません。ガーベージコレクションをした結果、再利用可能な領域が上のようにあったとしても、連続した大きさ 9 の領域がないと大きさ 8 の配列をつくることができないからです。このように利用可能な領域があちこちに散在することを断片化(fragmentation)といいます。そこで、次のようにゴミを回収するだけでなく、利用可能な領域を一か所にまとめれば大きさ 8 の配列もつくれるようになります。



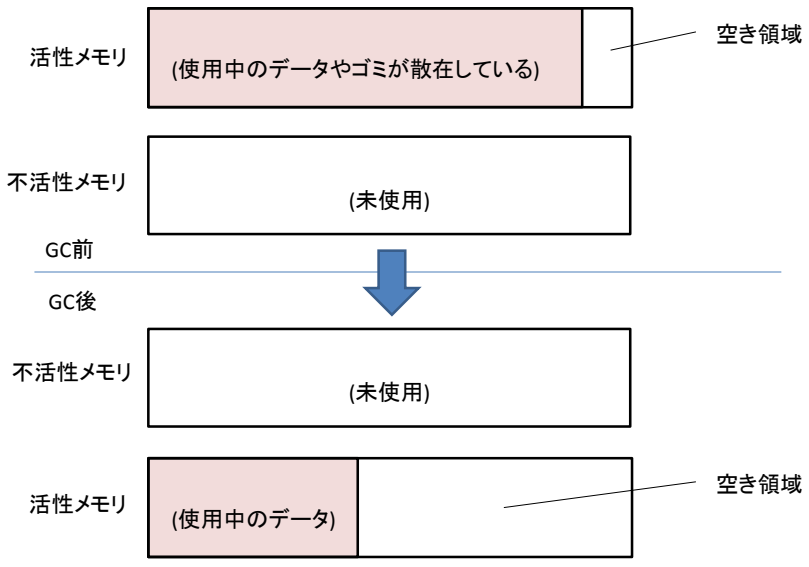
このように使用中の領域を一か所に集めて、連続した空き領域を作ることはコンパクションと呼ばれます。

ガーベージコレクションの方法は様々な方法が研究されていて、参照カウンター、ストップアンドコピー、マークスウィープなどの方法や、それらのハイブリッドなどがあります。ここではストップアンドコピーについて説明します。ストップアンドコピーGCはゴミを回収すると同時にコンパクションも行うガーベージコレクション方式となっています。

ストップアンドコピーGC の実装

今有効なデータに全てアクセスできるポインタ root が与えられているとします。これは今利用可能な環境を全て指すリストがあればそれが root となります。また、空き領域が空になる、もしくは空に近くなってきたら、ガーベージコレクションが行われます。

まず、ストップアンドコピー方式ではメモリ領域をまず半分ずつに分割します。そして、片方を利用可能なメモリ領域として使います。もう片方はガーベージコレクションが行われるまで使われません。つまり、ストップアンドコピー方式では、全メモリの半分しか利用出来ない、ということです。次の図をみてください。



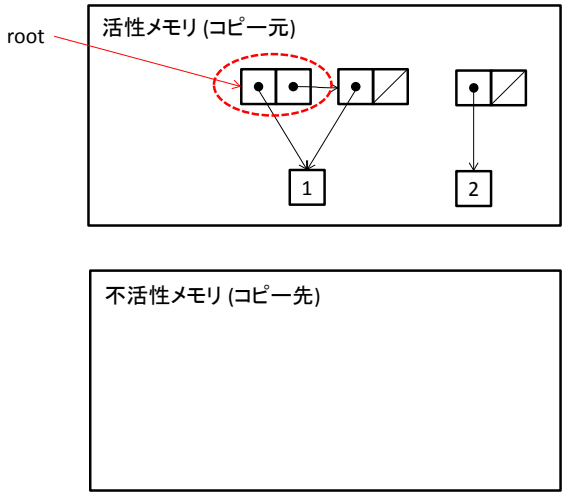
図の上半分にある二つの箱がガーベージコレクション前の二つのメモリ領域に対応します。上側の箱には有用データとごみが混在していて、下側の箱は全て空き領域となっています。説明の都合上、上側の有用データとごみが混在しているメモリを「活性メ

モリ」と呼ぶことにし、下側の空き領域だけのメモリを「不活性メモリ」と呼ぶことにしましょう。プログラムの実行中に必要となるメモリは、空き領域から取得されます。図の上半分では、活性メモリの空き領域が残り少なくなってきたので、ここでガーベージコレクションを行います。すると、図の下半分に書かれてあるような状態になります。活性メモリにあるデータを不活性メモリにコピーして、今まで利用していた活性メモリをまるごと廃棄します。次に、コピー先の不活性メモリを新しく活性メモリとして利用し、今まで使っていた活性メモリを不活性メモリとします。このコピーのときに、ゴミはコピーをせず、有用なデータのみをコピーするようにします。すると新しい活性メモリ領域ではゴミがなく有用なデータのみから構成されるようになります。

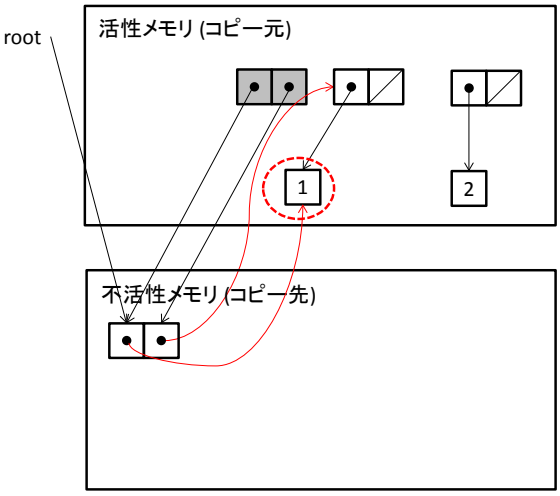
ストップアンドコピー方式は、ガーベージコレクションを開始したら、今実行しているプログラムを一時停止して、有用なデータのみを空いているもう一つのメモリ空間に移すことにより実現するガーベージコレクション方式と言えます。活性メモリと不活性メモリの役割を交互に交代して利用し続けることにより、ガーベージコレクションを繰り返し実行できるようになっています。

次に、有用なデータのみどうやって不活性メモリにコピーするか、ということについてですが、root から辿ることのできるデータを順に辿ることによって実現されます。ただし、構造共有をきちんと再現し、また、サイクルに対する無限巡回を避けるために、すでに巡回された箇所にはマークをし、マークされた箇所のコピー先がどこであるかを知る必要があります。

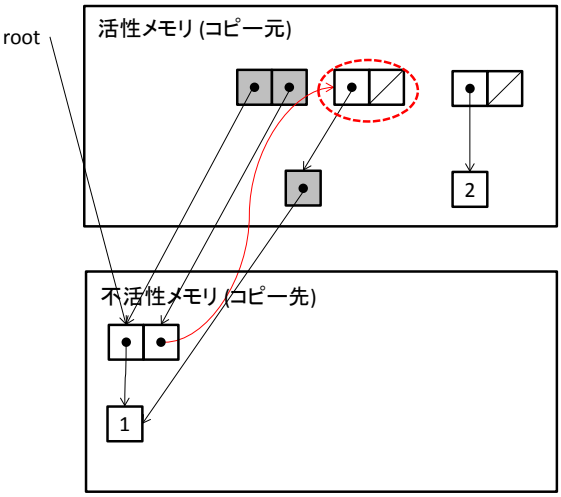
次のデータを見てください。



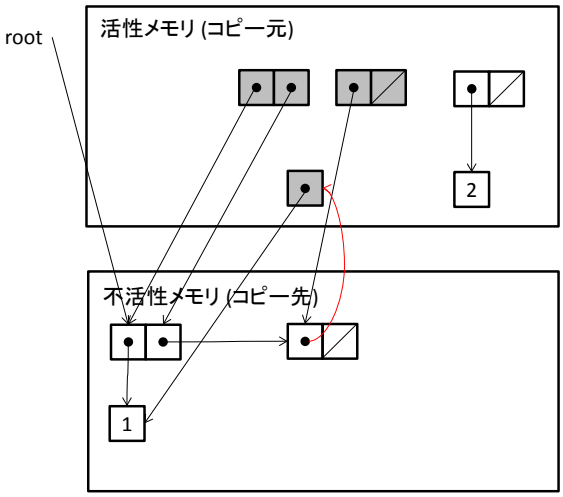
活性メモリには 1 を共有する短いリストが root から指されていて、このデータを不活性メモリにコピーしたいとします。2 を要素として持つリストは root から辿れないためゴミとして削除する、つまり、コピーしないようにします。図では赤線が処理中のポインタで、赤の点線で囲まれた楕円はコピー対象のデータとします。まず、root が指しているペアからコピーをします。



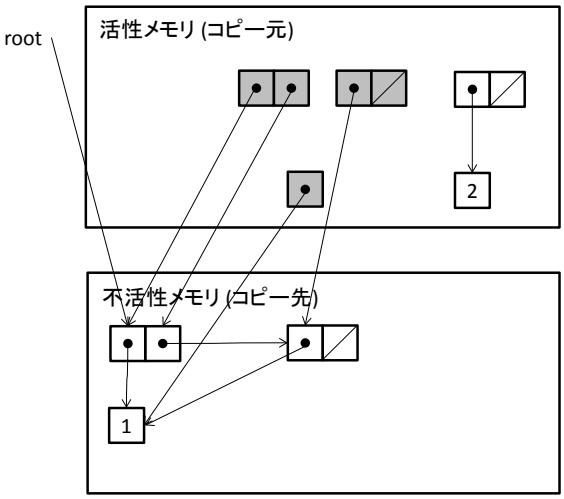
コピー元データはすでにコピー済みであることを示すためマークをつけておきます（図中では灰色で示しています）。また、コピー元はコピー先がどこであるかわかるようにコピー先へのポインタで上書きしておきます。続いて、1 が入っているデータをコピーします。



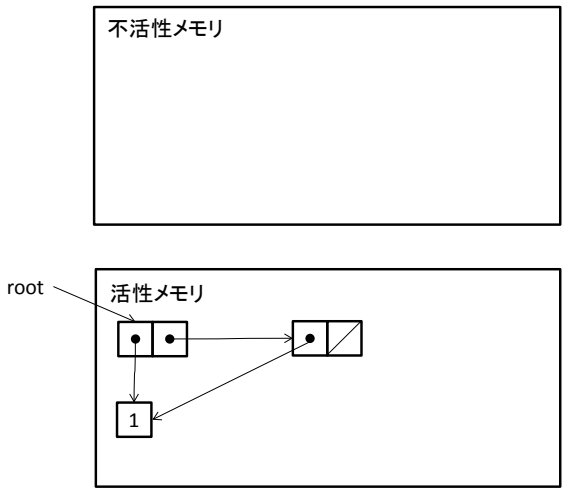
ここでも同様に 1 が入っていたボックスをコピー先を示すポインタで上書きし、またマークしておきます。次に、上の図において赤点線で囲まれたペアをコピーします。



続いて、上の図の赤線のポインタを処理するのですが、この赤線ポインタはマーク済みのデータを指しているので、コピーは行いません。次に、赤線ポインタを貼替えないといけないのですが、コピー元のデータがあったところにコピー先ポインタが残されているので、そのコピー先を指すようにポインタを貼替えます。すると次のように元々コピーしたかったリストが構造共有の情報を保ったままコピーされていることがわかります。



この時点で不活性メモリに全ての有効なデータを移し終わったので、活性メモリを消去し、活性メモリと不活性メモリの役割を入れ替えれば、ガーベージコレクションは終了ということになります。



ストップアンドコピー方式は、有効なデータをコピーする際に詰めてデータを並べることができるため、単に無効なデータを消去するだけではなく、コンパクションの機能を実現しています。