プログラミング言語　25－5
（１９９０．　６．８）

オブジェクト指向言語ＣＯＢにおける自動メモリ管理

久世 和資　　上村 務

日本アイ・ビー・エム(株)　東京基礎研究所
東京都千代田区三番町5-19　〒102

ＣＯＢは，Ｃベースの安全性を重視したオブジェクト指向言語で，プログラムの記述性や再利用性が高い．ＣＯＢに自動メモリ管理機構を導入するにあたって，メモリ管理システムの高い移植性と，多くのアプリケーションプログラムにおいて10%以内のオーバーヘッドとなることを目標にした．すでに提案されているメモリ管理の手法を，効率向上のための改良を施してＣＯＢで実現した．本論文では，Ｃベースの言語における各メモリ管理手法の問題点，評価および効率に影響する要因を述べる．さらに，各メモリ管理手法の組合せや選択による使用の可能性についても考察する．

Automatic Memory Management for an Object-Oriented Programming Language COB

Kazushi Kuse　　Tsutomu Kamimura

IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo 102

COB is a new object-oriented language, upward compatible with C. The overall design goal of COB is to increase the productivity of programming activities by providing safe language constructs. We attempt to introduce the automatic memory management mechanism into COB. We aim at high portability of the management system with less than 10% overhead for typical COB application programs. We implemented memeory management methods and improved their perfomance. We present problems, evaluations and limitation of these methods. The possibility of combination and selective usage of these methods are also examined.

1．はじめに

オブジェクト指向言語ＣＯＢにおけるオブジェクトのためのメモリ管理手法とそれらの性能評価について述べる．一般にオブジェクト指向プログラムでは，オブジェクトの生成と消去に伴うメモリの管理はプログラマにとって容易ではなく，プログラムの記述性を低下する一因となっている．自動メモリ管理は，この複雑な仕事をプログラマから解放すると同時に，プログラムの信頼性および再利用性の向上に役立つ．

ＣＯＢは，Ｃを基本にしたオブジェクト指向言語であり，強い型づけ，充実した実行時の型チェック，インタフェースとインプリメンテーションの完全な分離などの特徴を持つ．また，プログラムの記述性と信頼性の向上を目的とする自動メモリ管理機能も特徴の一つである．さらに，Ｃとの完全互換性を保ち，実行効率も各種の最適化を行うことによりＣ＋＋と同等の性能を持つ[Kamimura][Onodera]．

本論文では，ＣＯＢの特徴の一つである自動メモリ管理機能について実現方式と性能評価を述べる．自動メモリ管理機能を設計するにあたって以下のような方針にしたがった．

ａ）自動メモリ管理を行うことによる言語設計上の制限を最小限にする．
ｂ）実行時システムは移植性を重視し，アセンブラ等は使用せずＣまたはＣＯＢで作成する．
ｃ）メモリ管理によるオーバーヘッドは１０％程度以内とする．
ｄ）アプリケーション・プログラムの性質にかかわらずｃ）の性能を達成する．
ｅ）メモリ管理の対象はＣＯＢのオブジェクトのみとする．

これまでに実現されている自動メモリ管理システムのオーバーヘッドを比較検討し，アプリケーションプログラムに対する許容値を考慮した結果，１０％以下のオーバーヘッドが，妥当である判断した．また，ＣＯＢプログラムでは，一般にデータをオブジェクトとして扱うので，対象をオブジェクトに限定してもメモリ管理の効果は十分に得られる．

メモリ管理の手法として，まず，ユーザプログラムに対する適用性と高い性能を考慮して遅延参照カウント方式を使用した．遅延参照カウント方式の適用のためにスタック上のオブジェクトへのポインタを識別する必要があった．さらに，２つの表を使用するかわりに，リストと世代数を利用する手法を考案し組み入れた．これによりオーバーヘッドを５０％以上軽減することができた．

次に，環状につながれたオブジェクトの回収も考慮してコピー方式のメモリ管理機構を作成して実験を行った．コピー方式をＣＯＢに適用する際の一番の問題はコピーによるオブジェクトの移動に伴うスタック上のオブジェクト変数の変更である．今回は，オブジェクトの間接参照機構を導入してこの問題を解決した．また，世代の導入を行って性能を評価した．さらに有効オブジェクト数の予想によるごみ回収時間の最適化方式を考案し実現した．

最後に，マーク・スイープ方式に基づくメモリ管理機構を実現した．非参照のオブジェクトを消去（スイープ）するために，スタック上のオブジェクトポインタ識別用のオブジェクト表を利用した．

以降，自動メモリ管理機構の導入のための言語上の制約を最初に述べる．つづいて，実現した遅延参照カウント方式，コピー方式，マーク・スイープ方式について改良点を中心に紹介する．最後に，三方式の比較をもとにして，各方式の問題点，限界，考慮すべき要因をのべ，各方式の混合やユーザの選択による使用の可能性を議論する．

2．言語設計上の制限

ＣＯＢに自動メモリ管理を導入し，かつ，言語の安全性を保持するために以下の制限がある．

１）オブジェクトはヒープにのみ作成する．
２）クラス型に適用できる型構成子は，配列，関数，クラス構成子である．
３）オブジェクトのメンバのアドレスをとれない．

クラスを要素に持つstructをクラスを持たないstructにキャストできるので，誤ったごみ回収をするおそれがある．また，クラスへのポインタを許すと，インクリメントやデクリメントの際にクラス以外の領域を指す可能性がある．クラスの配列は許されているが，通常のＣと違ってポインタへの型変換は起こらない．配列は上限と下限のチェックも行う．

インスタンス変数が，オブジェクトへのポインタの場合，そのアドレスを使って内容を操作するとメモリ管理システムは，オブジェクトの参照関係を正しく把握するのが困難になる．

３．遅延参照カウント方式

遅延参照カウント方式(delayed reference counting)は，スタック上のポインタのカウントは，アプリケーションの実行中には行なわず，ごみの回収時にスタックをスキャンする事によって行なう[Deutsch76]．この方式によるオーバーヘッドは，通常の参照カウント方式[Collins，Goldberg]より小さく約10%を達成している[Deutsch84]．

ＣＯＢに遅延参照カウント方式を導入するために，解決すべき問題点がいくつかあった．特に，ＣＯＢでは，スタックにオブジェクトのポインタ以外のデータが積まれるので，オブジェクトポインタを識別する必要がある．

また，遅延参照カウンタ方式自体も改良した．オリジナルでは，参照数がゼロのオブジェクトをゼロ参照表に登録して管理するが，登録と取り消しのコストを削減するために，ゼロ参照リストで実現した．さらに，特別な表を使用せずに，世代数を利用することによって，高速にごみの回収を行なう方式を実現した．

３．１　ゼロ参照リストの導入

通常の遅延参照カウント方式では，オブジェクトの参照数に応じて２つの表を利用する．この表は，参照数０のオブジェクトの登録用のゼロ参照表と参照数２以上のオブジェクト登録用の多重参照表である．２つの参照表を利用するとオブジェクトの登録と消去が頻繁に起こり，そのオーバーヘッドが大きい．ごみ集め時にはゼロ参照表の検索と削除が頻繁に起こる．

そこで，２つの参照表のかわりに，ゼロ参照リストを導入した．まず，多重参照表をなくすために，参照数は各オブジェクトで保持することにする．つぎにゼロ参照表であるが，ハッシュ表を利用するかわりに，参照数が０になったものは，そのオブジェクト自体をゼロ参照リストに双方向リンクでつなぐ．双方向リンクでつなぐのは，ごみ集め時や参照数が１になった時に探索をせずに削除できるようにするためである．

通常の参照数の変化に伴うオブジェクトの登録と削除は，それぞれ双方向リンクの付け替えのための三つの代入文で実現できる．ごみ集めの際に，参照数０のオブジェクトへのポインタがスタックに積まれている時には，参照表からの削除と次回に用いる新しい参照表への登録が起こる．しかし，ゼロ参照リストを次に述べる世代数とともに用いることによって，スタックからも指されていないごみのオブジェクトの削除操作のみに簡約化できる．

３．２　世代数を利用したごみ集め

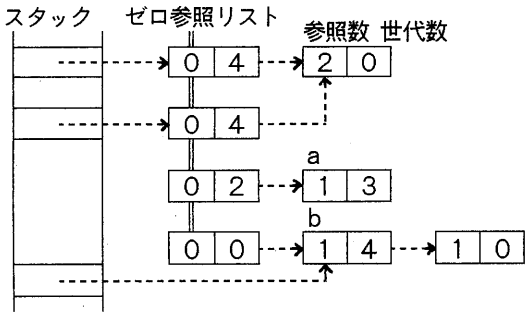
ごみの回収時に，回収されたオブジェクトが指しているオブジェクトの参照数が０となり，再びごみの対象となることはよく起こる．このごみの取扱い方によっていくつかの手法が考えられる．ひとつは，回収時にゼロ参照となったごみも，一定数以上であれば，その回に回収してしまう方式である．この方式では，全体のごみの回収率は上がる．しかしながら，１回のごみ集めでごみが爆発的に増大しスタックのスキャンが頻範に起こり，ごみ回収によるユーザプログラムの中断時間が大きくなる可能性がある．もう一つの方式は，回収時にゼロ参照となったごみは，その回には回収せずに次回の回収時にまかす方法である．これは，最初の方式に比べ，ユーザプログラムの中断を大きくする心配はない．しかし，アプリケーションによっては，全体としてほとんどごみが回収されない可能性もある．（これらの方式の評価は，ゼロ参照表を用いる初期版で行なった）

今回，提案する方式は，回収の世代数を利用することにより，１回のごみ回収で，関連するすべてのごみを低コストで回収することができる．まず，各オブジェクトに回収の世代数をいれるフィールドを用意する．世代数に関する規則は以下に述べる．

１）オブジェクトの生成時には，この世代数は０にセットする．
２）スタックスキャン時には，参照数にかかわらずスタックから指されているオブジェクトの世代数は，現在の回収の世代（最初からの回収回数）をセットする．
３）ゼロ参照リストを先頭からたどり，その世代数が現在の世代より小さいときには，ごみとして消去する．
４）消去する際にそのオブジェクトが指しているオブジェクトの参照数は１減らす．その時，参照数が０で世代数が現在の世代より小さいときは，そのオブジェクトもごみとして消去する．もし参照数が０になっても世代数が現在の世代数と等しいときには，ゼロ参照リストの最後につなぐ．

世代数は，そのままではオーバーフローするので，最大整数で割った余りを用いる．このために，最大数以上のごみ回収が起きた時には，余分なごみが消去さ

れない可能性があるが，ごみでないオブジェクトを消去する危険はない．



現在の世代数は４．オブジェクトａは世代数が４より小さいので消去されるが，ｂはゼロ参照リストにつながれる

図１　世代数を利用したごみ集めの概要

３．３　評価

ゼロ参照表を用いる初期版と本方式を比較すると，ごみ集めによるオーバーヘッドが５０％以上削減された．性能の比較は，ＣＯＢで記述された３つのプログラムを用いて行なった．

ｍａｚｅプログラムは，迷路盤をランダムに作成し，その上をねずみ（探索者）が，出発点からゴールをdepth-firstに探索する．バックトラックする際には，その経路上のねずみオブジェクトは，ごみとなる．ｅｘｐは，四則演算子，カッコ，変数を含む数式を計算するプログラム．数式の項に対して１つのオブジェクトが生成される．ｌｉｓｐは，ｌｉｓｐプログラムを入力し，項ごとにオブジェクトを生成し，実行するインタプリタである．評価では，複数の素数を生成するｌｉｓｐプログラムの実行をおこなった．それぞれのＣＯＢプログラムは，約1000，2000，および3000ステップである．

表１では，各ユーザプログラムごとに，ゼロ参照表とゼロ参照リスト＋世代数方式のオーバーヘッドを並べて示した．項目は，全実行時間，全メモリ管理時間（カウント時間＋回収時間），参照数のカウント時間，ごみ回収時間と回収率である．

参照カウント方式の欠点として，環状に結合されたオブジェクトは回収できないことがある．この問題解決のためにいくつかの方式が提案されているが，付加的な再帰スキャンが必要である[Standish, Brownbridge]．ＣＯＢでも参照カウント方式において環状のオブジェクトを回収する方法について考察したが，実行時のオーバーヘッドが大きく，実現していない．

ﾌﾟﾛｸﾞﾗﾑ	方式	全実行時間	ｶｳﾝﾄ＋回収時間	ｶｳﾝﾄ時間	回収 時間	回収 率
maze	表1	20.3	4.7(23)	1.3	3.4	34
	表2	17.4	1.8(10)	1.3	0.5	1
	ﾘｽﾄ	16.2	0.6 (3)	0.2	0.4	66
exp	表1	46.6	27.5(59)	13.9	13.6	89
	表2	40.8	21.7(53)	13.9	7.8	72
	ﾘｽﾄ	26.0	6.9(26)	5.3	1.6	91
lisp	表1	21.0	10.3(49)	2.1	8.2	57
	表2	17.3	6.6(38)	2.1	4.5	51
	ﾘｽﾄ	12.6	1.9(15)	0.9	1.0	74

（単位：sec，（）内と回収率は%）

表１は，ごみ回収時に再帰的に関連するごみを回収する方式，表２は，関連するごみは，次の回収にまかせる方式

表１　参照表方式と参照リスト＋世代数方式の効率比較

この他に，参照数の増減の際に判定と表への出し入れを簡略化する方法も，作成し実験した．この方法はオブジェクトの生成時には参照数を０にセットするだけでゼロ参照表に登録しない．参照数が増える時も，数を１増やすだけで，チェックや表への出し入れは行なわない．参照数が減るときにだけチェックとゼロ参照表への登録を行なう．この方法を用いるとプログラムによっては，ごみ回収のオーバーヘッドが約１０％削減できた．しかし，回収できないごみがある可能性がある．

また，ごみの回収を開始するタイミングはゼロ参照オブジェクトの数を参考にする．ゼロ参照数が一定値を越えたら回収を始めるが，ごみ回収が起きても回収率が低い時には，動的にこのしきい値の更新を行い最適なタイミングでごみ回収が起きるようになっている．

４．コピー方式

コピー方式のメモリ管理機構は，Bakerによって提案されたBarker Semispaceアルゴリズムが基本となっている[Barker]．この方式は，メモリを２つに分割して使用し，一方のメモリが一杯になると生きているオブジェクトをすべてたどって，他方のメモリにコピーすることによってごみの回収を行なう．ごみの回収と同時にメモリ領域の詰合せ(compaction)もできるのが特徴である．しかし，ごみ回収の時点で生きているすべてのオブジェクトをコピーするオーバーヘッドは大きい．この問題点を解決する手法として考案され，いくつかのシステムで実用化されているのが，世代に基づいた方式である[Ungar86, Ungar84]．この方式では，長く生き残ったオブジェクトは古い世代領域に固定して，ごみ回収時にコピーの対象としない．

ＣＯＢにコピー方式のメモリ管理方式を適用するためには，オブジェクトの移動に対する影響を考慮しなければならない．一方のメモリ領域から他方の領域にオブジェクトをコピーして移動する際に，そのアドレスを変更する必要がある．しかし，スタック上のオブジェクトポインタを正確に識別することは困難である．そこで，オブジェクトは間接に参照する機構を導入し，コピー方式の実験を行なった．連続した間接参照領域を設け，オブジェクトのデータ参照とメソッド呼び出しはこの領域を通して行なうこととした．オブジェクトの移動が生じても，間接参照領域内のデータの変更ですむので，ユーザプログラムの動作に影響しない．

次に，生きているすべてのオブジェクトをコピーするオーバーヘッドを削減するために世代を導入して実験を行なった．アルゴリズムは，任意の世代数に適用できるが，今回は２世代のメモリ領域を扱うごみ集めシステムを実現した．実験の結果，テストプログラムでは２世代目に移動されるオブジェクトが少いために，ごみ回収時のコストは小さくなるが，全体のコストとしては大きな変化はなかった．

最後に，コピー方式でオーバーヘッドを削減する手法として，スタック長を実行時にモニタしてごみ集めのタイミングを調節する方法を考案し，実現したので解説する．コピー方式では，生きているすべてのオブジェクトをコピーするので，ごみ集め時には生きているオブジェクトの数が少ないのが望ましい．そこで，スタック長と生きているオブジェクトの関係に着目して，スタック長が短い時には，生きているオブジェクトが少ないと仮定する．オブジェクトの生成時にスタック長を観察して，現在のメモリ消費量との関連を調べて，スタック長が最小値に近い時にごみ集めを起こす．

４．１　オブジェクトの間接参照機構の導入

ＣＯＢにコピー方式を適用するためには，ごみ集め時のオブジェクト移動に対して方策が必要である．今回は，オブジェクトの参照は固定した間接参照領域を通して行なうことによって，この問題を解決した．

まず，間接参照領域として一定の領域を確保する．この部分は，オブジェクトが，ごみ集めによって消去された場合は，フリーリストにつなぐことによって再利用される．この領域は，連続した領域にとられサイズは固定である．これは，スタック上のオブジェクトポインタの識別を容易にするためである．

オブジェクトの生成時は，オブジェクト本体はメモリ領域に生成され，間接参照領域の１セルが，その先頭を指す．オブジェクト変数には，そのセルのアドレスが代入される．オブジェクト変数の代入が起こると，同じセルのアドレスを指すことになる．オブジェクトの型変換が生じると，新たに１セルを設け，そのセルから変換結果にしたがってオブジェクトの途中または先頭を指す．

この方式の欠点は，メソッド呼び出しとインスタンス変数の参照に１間接のオーバーヘッドがかかることである．

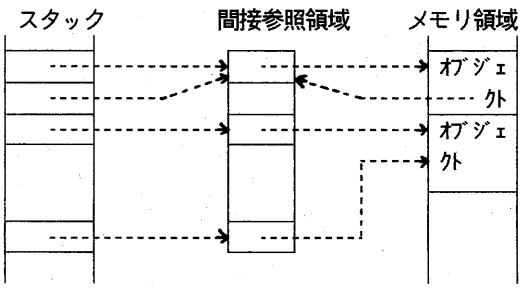


図２　オブジェクトの間接参照の概要

実際にポインタをたどってオブジェクトをコピーするのは，各クラスに標準で用意されたコピー用のメソッドである．そのメソッドの再帰的な呼び出しが，ポインタをたどることに対応する．

4．2　スタック長を考慮したGC

コピー方式で，ごみ回収時にコピーされるオブジェクト数を削減する方式を考案し実現した．これは，スタック長を実行時にモニタすることによって，なるべくスタック長が短い時にごみ回収を開始する方法である．

種々のユーザプログラムについて，スタックの伸縮の様子を観察したところ2桁から3桁の差の伸縮が頻繁に起こることがわかった．オブジェクト生成時のスタック長とその時点で生きているオブジェクト数の関係について以下の仮定をする．

仮定a

オブジェクト生成時のスタック長が下降状態から上昇に転じた時点では，それ以外の点より生きているオブジェクトが少ない．

仮定b

複数の上昇に転じた時点では，スタック長が短い方が，生きているオブジェクト数が少ない．

仮定a，仮定bが正しいとすると，スタック長が下降から上昇に転じた時点で，スタック長が最も短い時に，ごみ回収を開始すれば，生きているオブジェクトが少ないために効率よくコピーによって回収ができると考えられる．

この方式を実現するために以下の操作を行なう．

1）オブジェクト生成時に現在のスタック長を記録する．

2）前回，記録したスタック長と比較して，下降していれば下降状態のフラグをセットする．

3）下降状態の時に，スタック長が前回より長くなっていたら，上昇に転じた時点としてマークする．

4）上昇に転じた点では，前のスタック長をごみ回収のトリガーの候補として，記録しておく．同じスタック長が再び現われたらその回数もカウントしておく．

5）メモリの残り容量が一定数を下回っている時に，記録したスタック長に一致すれば，ごみ回収開始を検討する．それ以下のスタック長が出現した回数と残り容量から，スタック長が現在より下回る可能性を計算する．その可能性がなければごみ回収を開始する．

6）5）によってごみ回収が開始されることがなければ，メモリの残り容量がなくなったところで通常のごみ回収が起こる．

4．3　評価

コピー方式全体の性能と間接参照のオーバーヘッドについては，6．で述べる．ここでは，主に世代の導入とスタック長のモニタによるオブジェクト数最小予想方式の効果について実験結果を示す．

世代の導入では，ごみ集めのコストは，約10%減少する．しかし，今回のテストプログラムに関しては，同じメモリ容量で比較した場合，各オブジェクトに世代のフィールドが増えるのと，世代の管理のオーバーヘッドがあり，全体のコストとしては，大きな向上は得られなかった．テストに用いたプログラムでは，2世代領域に移されるオブジェクトの絶対量が，それぞれ生成オブジェクトの1%以下と少ないためと考えられる．より長時間使用するエディタなどの会話システムでは，世代の効果がより期待できる．

スタック長を利用した最小オブジェクト時点の予想方式では，パーサーで3%の性能向上が得られた．ごみ回収時の性能は，より大きく向上するが，オブジェクト生成時のスタック長のチェックのコストがかかるため，全体では約3%にとどまる．スタック長のチェックを毎回行なわずに，サンプリングすることによって，モニタリングのコストは落とすことができるが，予想の正確さも小さくなる．

プログラム	世代	全実行時間	ごみ回収時間	2世代オブジェクト
lisp	なし	23.0	1.2	0
	あり	22.3	1.0	916
parser	なし	18.3	4.2	0
	あり	18.0	2.6	38

表2　世代導入の効果

プログラム	予想	全実行時間	ごみ回収時間	コピーしたオブジェクト	回収回数
lisp	なし	42.9	1.0	12594	7
	あり	42.6	0.8	8985	9
parser	なし	19.3	4.5	35725	15
	あり	18.7	4.0	12138	16

表3　予想方式の効果

5．マーク・スイープ方式

マーク・スイープ方式を実現するためには，スイープのために生成されたオブジェクトを管理する必要がある．ここでは，スタック上にあるオブジェクトポインタを識別するためのオブジェクト表を利用した．

ＣＯＢプログラムの実行では，スタックにオブジェクトへのポインタ以外のデータも積まれるので，スタックのスキャン時にオブジェクトへのポインタを識別する必要がある．そこで，オブジェクトのアドレスを登録するオブジェクト表を用意した．

オブジェクトの生成時に，そのオブジェクトのアドレスをこの表に登録し，オブジェクトでは登録したエントリを記憶しておく．スタック上にオブジェクトのアドレスらしき値が見つかると，オブジェクト表のアドレスとの照合によって，正しいアドレスかどうか判定できる．この方式は，参照カウント方式でも使用している．

ＣＯＢの処理系は，オブジェクトのメンバをたどってマーキングをするメソッドを生成する．マークは，オブジェクト表につける．スイープは，オブジェクト表をスキャンして，マークされていないオブジェクトを解放する．スイープ時に，オブジェクト表の詰合せも行なう．

実際のメモリのアロケーションとフリーは，参照カウント方式と同様，ＯＳが提供する標準ルーチンを利用した．

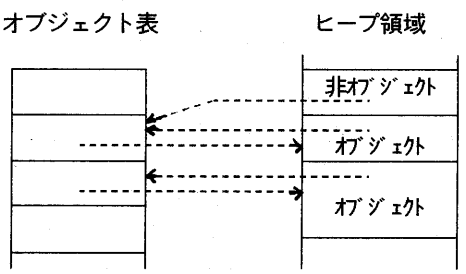


図３　オブジェクトの識別とマークのためのオブジェクト表

6．自動メモリ管理方式の比較と問題点

6．１　ＣＯＢにメモリ管理を導入する際の考慮点

まず，ＣＯＢの言語仕様および処理系からは，以下の考慮点が挙げられる．

１）スタック上のオブジェクトポインタの識別
２）グローバル領域上のオブジェクトポインタの識別
３）型変換されたオブジェクトの先頭アドレスの計算
４）型変換によるオブジェクトポインタの位置
５）オブジェクトの消去時ルーチンｆｉｎａｌの実行

ＣＯＢは，Ｃの上位互換言語であり，クラス以外に，Ｃで許される型を持つデータが使用できる．したがって，ＣＯＢのオブジェクトを管理するためには，スッタクとグロバール領域上のオブジェクトポインタを識別する必要がある．ＣＯＢのコモン変数は，グローバル領域にとられるので，グローバルなオブジェクトポインタも多いことが予想される．

ＣＯＢは，オブジェクトの上位クラスおよび下位クラスへの型変換を明示的に行なう機能を持っている．このため，オブジェクトへのポインタが，その先頭を指しているとは限らないし，あるオブジェクトへの複数のポインタが異なる位置を指す可能性もある．オブジェクトに付加されたメモリ管理用の情報を参照したり，オブジェクトを消去するには，オブジェクトの先頭を計算しなければならない．

ＣＯＢでは，オブジェクトの生成時と消去時に自動的に実行されるｉｎｉｔとｆｉｎａｌ関数が用意され，その内容はプログラマが定義することができる．オブジェクトの生成時と消去時には，これらの関数を実行しなければならない．

6．２　ＣＯＢプログラムの特性

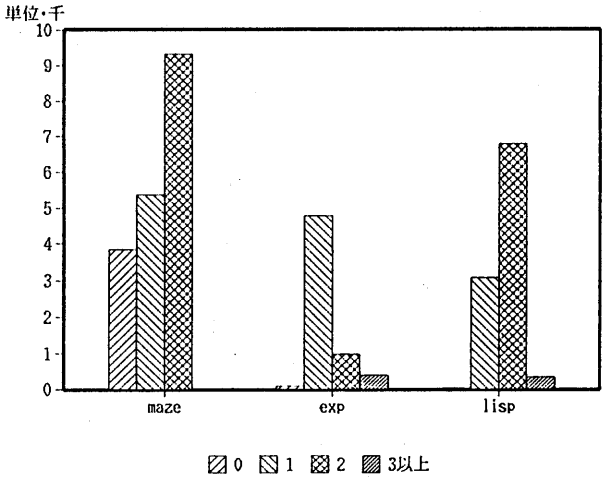
次に，ＣＯＢで記述したアプリケーションプログラムの特性を以下に述べる．

１）オブジェクトが，環状に結合されやすい．
２）オブジェクトへのポインタ数が２以上のことが多い．

遅延参照カウンタ方式が有効に動作するには，「ほとんどの場合，参照数が１である」という仮定がある．しかし，オブジェクト指向プログラムでは，オブジェクト間でメッセージをやりとりするために，各オブジェクトは，単純な木構造ではなく，ネットワーク状に

結合されている可能性が高い．テストプログラムで調べた結果，参照数2が多いことがわかった．

また，同じような理由で，オブジェクトの結合が環状になっていることも多い．テストに用いた5つのプログラムのうち，2つに環状の構造があった．



グラフ1　オブジェクトの参照数の分布

6．3　自動メモリ管理の効率

メモリ管理の効率を比較するには，以下の要因が考えられる．

1）メモリ管理の時間
2）ごみ回収回数
3）ごみ回収1回あたりの中断時間
4）使用メモリ総量
5）ページフォルトの影響
6）生きているオブジェクトの割合による影響

三方式の比較を，同じメモリ容量を与えて行なった．遅延参照カウント方式は，ゼロ参照オブジェクトの数が一定数を越えたときにごみ回収を開始する機構になっている．そこで，その数を変化させて実行した時の使用メモリ容量を，それぞれコピー方式とマークスイープ方式に与えて性能を測定した．コピー方式では，2分割でメモリを使用するので，オブジェクトが最大使用できるのは参照カウント方式の半分のメモリである．評価は，lispインタプリタとCOBパーサー[Kuse]を用いて行なった．

プログラム	実行時間 (sec)	ヒープ使用量(k)	生成オブジェクト数
lisp	36.1	1092	68913
parser	16.8	5916	85835

表4　テストに使用したプログラムの動作特性

使用ヒープ (Kbyte)		214	300	505	880
参照カウント方式	合計	17	15	14	13
	カウント	9	9	9	9
	回収	8	6	5	4
コピー方式	合計	15	14	14	14
	間接	12	13	13	13
	回収	3	1	1	1
マークスイープ	合計	4	3	1	1

(単位：%)

表5　lispインタプリタにおける自動メモリ管理のオーバーヘッド

使用ヒープ (Kbyte)		711	1300	2292	3351
参照カウント方式	合計	33	28	25	24
	カウント	13	13	13	16
	回収	20	15	12	8
コピー方式	合計	10	8	4	7
	間接	3	3	3	3
	回収	7	5	1	4
マークスイープ	合計	5	1	2	5

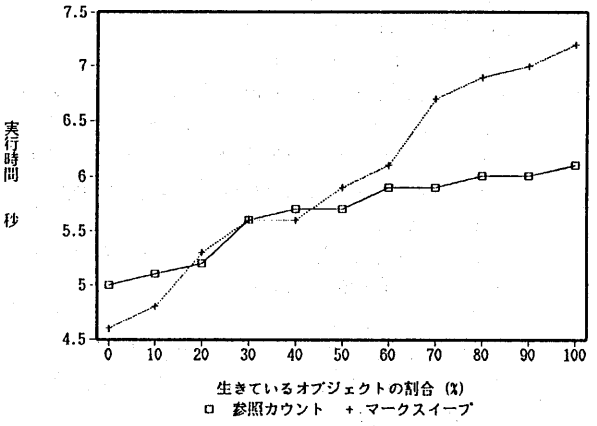
(単位：%)

表6　COBパーサにおける自動メモリ管理のオーバーヘッド

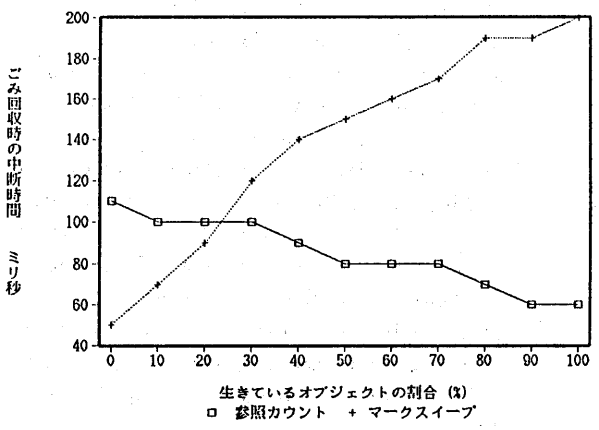
この評価結果からは，マークスイープが，もっともオーバーヘッドが少ないことがわかる．しかし，ごみ回収のオーバーヘッドに着目すれば，使用できるメモリ量が半分にもかかわらず，コピー方式のオーバーヘッドが小さいことがわかる．ごみ回収1回あたりの中断時間は，遅延参照カウント方式が有利である．なお，遅延参照カウント方式によるparserの実行では，環状のポインタも回収できるようにプログラムの一部を変更した．

表5と表6では，遅延参照カウント方式のオーバーヘッドが大きいが，これは，すべてのアプリケーションプログラムに単純にあてはまらない．グラフ2は，

簡単なプログラムで，各ごみ回収時点での生きているオブジェクトの割合を変化させた時のオーバーヘッドを示す．ごみ回収時で，生きているオブジェクトの割合が大きい時は，遅延参照カウンタ方式のオーバーヘッドは小さくなる．遅延参照カウンタ方式は，生きているオブジェクトの割合に対する影響が少ないこともわかる．また，ごみ回収による中断時間も，カウント方式が小さく，応答性が要求されるアプリケーションには適している．



グラフ２　生きているオブジェクトの割合とメモリ管理のオーバーヘッド



グラフ３　生きているオブジェクトの割合と中断時間

６．４　自動メモリ管理機構の移植性

遅延参照カウント方式とマークスイープ方式の移植性は高い．コピー方式は，ＯＳのメモリ割付けの方法に依存するため移植性が低い．

遅延参照カウンタ方式とマークスイープ方式は，PS/2とRT/PCのAIX上とPS/2のOS/2上に移植されている．２つのAIXでは，ヒープ領域の上限値と下限値の定数を変更するだけで対応できる．OS/2とAIXでは，ヒープとグローバルの領域のチェックが変わる．OS/2のメモリは，６４Ｋバイトずつのセグメント単位でとられるため，チェック用のコードが数行，余分に必要になる．

コピー方式をOS/2に移植するには，この６４Ｋバイトに適した調整をする必要がある．

６．５　議論

表７は，６．１から６．４までに述べた要因に対して各方式の適応度を示したものである．たとえば，効率２では，ごみ回収回数に関するもので，マークスイープ，コピー，カウントの順に多くなる．移植性は，６．４でも述べたが，AIX用のコピー方式のメモリ管理システムをOS/2に移植するのは難しい．AIXの場合，連続するヒープ領域を単純に２分割して使用できるが，OS/2の場合，６４Ｋバイトのセグメントを連結して使用する必要がある．

項目		参照カウント	コピー	マークスイープ
言語処理系	1	△	△	△
	2	○	×	×
	3	△	△	△
	4	△	×	△
	5	○	×	○
COBプログラム	1	×	○	○
	2	△	○	○
効率	1	×	△	○
	2	×	△	○
	3	○	×	×
	4	△	△	○
	5	△	△	×
	6	○	×	×
移植性		○	×	△

表７　各メモリ管理方式の比較

7. おわりに

オブジェクト指向言語ＣＯＢの自動メモリ管理についてその実現方式と性能評価を述べた．今回のテストプログラムでは，どの方式が最良であるか判断が難しい．しかしながら，各方式でどの部分にオーバーヘッドがかかっているのかを明らかにすることができた．特にオブジェクトの間接参照については，メソッド表へのポインタを間接参照領域に設けたり，ごみ回収が起こらない部分でのメンバの参照は直接行うなどの改良の余地が考えられる．

さらに，エディタなどの会話型アプリケーションを含む多くのテストプログラムで性能評価を行うとともに，自動メモリ管理システムのオーバーヘッドをさらに削減するのが，今後の課題である．

謝辞

遅延参照方式の初期版の設計と作成は，1988年にIBM東京基礎研究所の客員研究員であったJohn M. Lucassen氏によるものである．ＧＣに関連するCOBの言語仕様の設計と遅延参照方式のカウント用コードの生成は東京基礎研究所の小野寺民也氏が行なった．効率向上の手法に関しては同研究所の鈴木則久所長に多くの貴重なコメントをいただいた．MITのCarl Hewitt教授にも，貴重なコメントをいただいた．東京基礎研究所の渦原茂氏にも多くの有益なコメントをいただいた．

参考文献

[Barker] H. G. Baker, "List Processing in Real Time on a Serial Computer", A. I. Working Paper 139, MIT-AI Lab, Boston, MA, April 1977.

[Boehm] Boehm, H. J. and M. Weiser, "Garbage Collection in an Uncooperative Environment", Software Practice and Experience, September, 1988.

[Collins] G. E. Collins, "A Method for Overlapping and Erasure of Lists", Communications of the ACM 3, 1, December 1960, 655-657.

[Caudill] Patrick Caudill and Allen Wirfs-Brock, "A Third Generation Smalltalk-80 Implementation", OOPSLA '86 Conferance Proceedings, 119-130.

[Deutsch76] L. Peter Deutsch and Daniel G. Bobrow, "An Efficient Incremantal Automatic Garbage Collection", Communications of the ACM 19, 9, Sept. 1976

[Deutsch84] L. Peter Deutsch and Allan Schiffman, "Efficient Implementation of the Smalltalk-80 System", Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages, January 1984.

[Goldberg] Adele Goldberg and David Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[Kaehler] Ted Kaehler, Glenn Krasner, "LOOM Large Object-Oriented Memory for Smalltalk-80 Systems", in Smalltalk-80: Bits of History, Words of Advice, G. Krasner(ed.), Addison-Wesley, 1983, 251-271.

[Kamimura] 上村務，横内寛文ほか，"COBにおけるオブジェクト指向機能"，コンピュータソフトウェア，6, 1, 1989, pp. 4-16.

[Kuse] 久世和資，"オブジェクト指向言語COBによる構文解析系の設計と実現"，情報処理学会39回大会，1989.

[Lieberman] Henry Lieberman and Carl Hewitt, "A Real-Time Garbage Collection Based on the Lifetimes of Objects", Communications of the ACM, 26, 6, June 1983, pp. 419-429.

[Onodera] Tamiya Onodera, Kazushi Kuse and Tsutomu Kamimura, "Increasing Safety and Modularity of C Based Objects", IBM Tokyo Reserch Laboratory Technical Report.

[Ungar84] David Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm", Proceedings of the ACM Symposium on Practical Software Development Environments, Pittsburgh, April 1984.

[Ungar86] David Ungar, The Design and Evaluation of a High Performance Smalltalk System, ACM 1986 Distinguished Dissertation, MIT Press, 1987.

[Ungar88] David Ungar and Frank Jackson, "Tenuring Policies for Generation-Based Storage Reclamation", OOPSLA'88 Proceedings, pp1-17.