記　号　処　理 70－3
（１９９３．　９．　１７）

# 循環構造に適用可能な参照カウント方式GC

前田 宗則　小中 裕喜　石川 裕　友清 孝志　堀敦史
技術研究組合 新情報処理開発機構 つくば研究センタ

本稿では，循環参照カウント方式(CRC) を基礎とする新しいGC アルゴリズム $CRC_{IW}$ を提案する．CRC は，ポインタによる循環構造も含めた任意の使用不能なメモリブロック（オブジェクト）を回収可能なGC 方式であるが，対象言語がコンビネータに制限されること，循環構造を管理するアルゴリズムが逐次的であることという二点により，並列マシン上の一般の高級言語にはそのまま適用できなかった．$CRC_{IW}$ は，各オブジェクトに順序数を与えることで任意の言語に適用可能とし，複数のプロセスによって並列に循環構造を管理するようアルゴリズムを拡張している．さらに，分散メモリを持つ並列マシンにおいてGC による通信オーバーヘッドを低減するために，参照を3 タイプに分けて管理することと各参照に重みを与えることが考察される．

# A GC algorithm based on Reference Counting able to collect cycles

Munenori Maeda, Hiroki Konaka, Yutaka Ishikawa, Takashi Tomokiyo, Atsushi Hori
Tsukuba Research Center, Real World Computing Partnership
16F Mitsui Bldg., 1-6-1 Takezono, Tsukuba-shi, Ibaraki, 305, JAPAN

In this paper, we propose a new garbage collection algorithm based on Cyclic Reference Counting, CRC, which is able to reclaim any memory block, called object, in disuse including a cyclic structure by pointers. However, CRC has difficulty in applying to high-level languages on multicomputers because: i)its target is not a general language which allows unrestricted pointer operations but only combinator, and ii)its algorithm managing cycles is sequential and not incremental. Our GC scheme, called $CRC_{IW}$, extends CRC in terms of above two points. A unique ordinal number for each object and a multi-processed cycle management enable $CRC_{IW}$ to manage cycles safely in parallel independently of languages. In addition, to reduce communication overhead on a distributed environment, we investigate classifying types of reference into three and adding weight for each reference.

# 1 はじめに

ガーベッジコレクション(GC)[1, 7] とは，ヒープ領域から変数に割り当てられたメモリブロック（オブジェクト）が使用不能になったときに，自動的にヒープ領域に戻して再利用可能とする自動メモリ管理機構である．Lisp や Smalltalk に代表される高級言語は，GC を用いることでプログラマから直接的なメモリ操作を隠蔽し，プログラムの信頼性と抽象度を高めることに成功している．

近年，いくつかの疎結合 MIMD 並列マシンが利用可能となっており，今後より身近な存在となろう[1]．これらの並列マシンの性能は，リモートメモリアクセスに非常にセンシティブであり，それゆえ GC は，通信コストまで考慮した実行効率の高いものでなければならない．

本稿では，分散メモリ管理のための GC 方式を提案する．本方式は $CRC_{IW}$ と呼ばれ，循環構造をなしたゴミの除去を可能とするよう，参照カウント方式 (Reference Counting; RC) を拡張した循環参照カウント方式 (Cyclic RC; CRC)[4] に基づくものである．CRC に対する本方式の拡張点は，(1) 適用言語の拡大，(2) アルゴリズムの並列化，(3) 通信オーバーヘッドの低減という 3 つである．

本稿の構成を明らかにする．まず，2節で研究の背景を述べる．次に3節で用語と参照の基本操作，循環参照管理方式を明らかにする (拡張点 (1))．4節は，3節で示した，オブジェクトの用不用管理操作をマルチプロセス化する話題について考察する (拡張点 (2))．また，5節では，参照に重み付けすることでリモート参照の複写と削除における通信オーバヘッドを軽減することを試みる (拡張点 (3))．最後に，6節で $CRC_{IW}$ の特徴をまとめる．

# 2 研究の背景

代表的な GC 方式としては，RC 方式の他にマーク&スウィープ方式 (Mark-Sweep; MS) が挙げられる．RC は使用不能になったオブジェクトを直接に回収する方式であり，MS はポインタをトレースすることで使用中のオブジェクトを識別し，そうでないオブジェクトを間接的に回収する方式であると特徴づけることができる．

GC 方式を決定するにあたっては，以下の観点から対象とするアプリケーションや並列マシンの特性を鑑みる必要がある．

1. スループット vs. レスポンスタイム（ソフト）
2. メモリサイズとメモリ占有率（ハード，ソフト）
3. プロセッサを跨ぐ通信のコスト（ハード）

一般に，疎結合並列マシンでは，ディスクを用いたデマンドローディングが相対的にコスト高で，オンメモリの処理が重視される．また，リモートメモリアクセスは，ローカルメモリアクセスと比べてかなりコスト高である．

こういった並列マシン上でユーザーとのインタラクティブな処理やリアルタイム処理をターゲットとする場合には，スループットよりもレスポンスタイムを保証しなければならない．そこで，処理はオンメモリで行なうこと，大域的な GC を行なう場合にはプロセッサ間通信をできるだけ少なくすることが要請される．

リアルタイム処理を指向するとき，インクリメンタルな GC が必要となる．インクリメンタルな GC とは，“GC プロセスは時間の短い複数のステップに分割可能であって，アプリケーションの実行を長時間中断しない” ものである．標準的な RC は生来的にインクリメンタルであり，インクリメンタルに拡張された MS[8] と比較して，実装とその正当性の証明が単純である．

RC はオブジェクトのメモリの占有率に関わらず定常的なパフォーマンスを持つが，オブジェクト単位の操作を行なうため総体としては処理量が多い．他方，MS はメモリ占有率とゴミ生成の割合に依存してパフォーマンスが変化する．メモリ占有率とゴミ生成率が高い場合はパフォーマンスが悪化する．逆に，メモリ占有率とゴミ生成率が低い場合は非常に有効である．

大域的な GC を行なうときの通信コストの観点で，分散 MS[2, 10] と分散 RC[3, 5] を比較してみる．分散 MS では，使用中であるオブジェクトをトレースすること及び大域 GC の完了を検出することのためにバリア同期を含めたプロセッサ間通信が必要となる．他方，分散 RC では，リモートポインタを削除するために被参照オブジェクトのあるプロセッサへ非同期通信することが必要であるが，分散 MS よりもスケーラビリティの点で優れている．

以上のように，我々のターゲットに関しては，RC は MS より有効であると言える．しかしながら，RC と分散 RC は，循環構造をなすゴミオブジェクトを回収できないという問題を持つ．これに対して，いくつかの CRC[4, 6] 技法が提案されている．

Brownbridge の CRC[4] は，ポインタを 2 種類に分けてサイクルを管理するアルゴリズムである．コンビネータを対象言語としており，生成されたポインタに正しくポインタタイプが割り当てられることを前提にしている．任意のポインタ操作を許す言語では，タイプの正しい割

[1] 例えば，RWC-1[9] は，1,000 台規模の超並列計算機であり，1996 年度から稼働予定である．

り当てが困難であることが指摘されている．また，ゴミオブジェクトの判定手続きが逐次的であり，並列マシンへの適用が考慮されていない．

Lins の CRC[6] は，ローカル MS という概念に基づいている．ゴミと疑われるオブジェクトから到達可能なオブジェクトのクロージャが使用中のオブジェクトからの参照を含んでいるかどうかを MS によって判定する．この点で MS と RC のハイブリッドな方式である．

# 3 CRC 適用可能言語の拡大

## 3.1 用語の定義

本節では，文献[4] に従っていくつかの用語を定義する．あるオブジェクト $S$に格納される，他のオブジェクト $T$へのポインタを参照という．これを<$S,T$>と表記する．

あるオブジェクトがゴミであることの直観的な定義は，そのオブジェクトが二度と実行時にアクセスされないことである．形式的には，まず使用中のオブジェクトを定義し，全オブジェクトのうち使用中でないオブジェクトをゴミであると定義する．使用中のオブジェクトとは，ルートと呼ばれる特別な集合から参照を有限回辿って到達可能なオブジェクトである．ルートとは，レジスタやスタック，大域変数など現在の計算状態を完全に決定する必要十分な要素のうち，オブジェクトへのポインタを含んでいるものである．

オブジェクトの参照は strong と weak の 2 種類に分類される．参照<$R,S$> のタイプ（強度）は，<$R,S$>.*type* で示される．参照に強弱が導入されたことに従い，各オブジェクト $R$は strong 参照カウント $R.SRefc$ と weak 参照カウント $R.WRefc$ を保持する．

参照強度は，参照先オブジェクトと参照ポインタ自身にそれぞれ 1 ビット与えることで実装され，それらの排他的論理和⊕によって決定される．

$$\begin{cases} strong & \text{if } <R,S>.rbit \oplus S.obit = 0 \\ weak & \text{otherwise} \end{cases}$$

ここで，$S.obit$ は参照先オブジェクト $S$に保持されるビットであり，<$R,S$>.*rbit* は，参照側オブジェクト $R$に保持される参照<$R,S$>に与えられるビットである．

さて，参照の強弱は，以下の不変則に従って保持されなければならない．

**不変則 1** *strong* 参照はルートから到達可能なパス上にのみ与えられ，かつ *strong* 参照のみからなる任意の部分パスは循環構造を含まない．ルートから到達可能な任意のオブジェクトは，*strong* 参照のみから成るルートからのあるパス上に存在する．

**系 1** $R.SRefc > 0$ *for all* $R$ *in use*

すなわち，使用中のオブジェクトは，1 個以上の strong 参照と 0 個以上の weak 参照によって指される．

参照に関する 3 つの基本操作を以下のように定義する．

- NEW($R$) フリーリストからオブジェクト $S$を割り当て，$R$から $S$への strong 参照を生成し $R$に格納する．$S.SRefc$ は 1 に設定する．
- COPY($R$,<$S,T$>) 参照<$S,T$> をコピーすることで $R$から $T$への参照を生成する．このとき，参照<$R,T$> の強度は不変則 1 に違反しないように決定されなければならない．
- DELETE(<$R,S$>) $R$から $S$への参照を削除し，$S$の対応する参照カウントを 1 減ずる．もし参照カウントの総和が 0 となったならば，$S$の持つ全ての参照に対して DELETE 操作を再帰的に起動した後，$S$を解放しフリーリストに戻す（**再帰的解放**）．

最後の DELETE 操作は，RC にはない CRC 特有の以下のような操作を含む: もし $S$の strong 参照カウントが 0 になったならば，$S$が使用中か否かを決定する．もし使用中であるならば，不変則 1 を保つために，$S$への参照と strong, weak それぞれの参照カウントを補正する．

まず，$S$への全ての weak 参照を strong 参照に変換する．これは，$S.obit$ のビット反転によって容易に達成できる．参照タイプの変換によって生じた，strong 参照による循環構造を取り消すために以下のような補助操作を実行する．

- SUICIDE(*Start*,<$R,S$>) 起点オブジェクト *Start* から strong 参照<$R,S$>を辿ってオブジェクトをトレースし，サイクルができないようにある条件を満たす参照<$R,S$>を weak にする．

## 3.2 順序数によるサイクル管理

オブジェクトベース言語のような任意のポインタ操作を許す言語のもとで，いかにサイクルを管理するかが問題である．参照の強弱の割り当ては，COPY($R$,<$S,T$>) 操作におけるオブジェクト $R$, $S$, $T$の局所的な情報のみによって決定されることが望ましい．

$CRC_{IW}$では，以下のようなアイデアに基づいて，各オブジェクトに順序数を与えることでこの問題を解決する．

任意の全順序集合$G$に対して，$X < Y \wedge Y < Z$なる関係を満たす$G$の相異なる3つの要素$X, Y, Z$は，推移律$X < Z$を常に満たす．つまり，循環関係$X < Y \wedge Y < Z \wedge Z < X$が決して起こらない．

さて，オブジェクト上の全順序関係<を導入し，それに伴い各オブジェクトに順序数を格納する．オブジェクト$R$の順序数は$R.ord$で表記する．

**不変則 2** オブジェクト$R$, $S$に対して，もし$R$から$S$への参照が*strong*であるならば$R$の順序数は$S$の順序数より小さい．すなわち$R.ord < S.ord$.

この不変則は，参照の基本操作のもとで保存されなければならない．

- NEW($R$) 元のNEWと同様に，フリーリストからオブジェクト$S$を割り当て，$R$から$S$へのstrong参照を生成し$R$に格納する．このとき，$S$に現在最大の順序数を格納する．

  $$S.ord := \textbf{ordinal\_max};$$

  最大の順序数**ordinal_max**を与える方法については，3.3節で検討する．直観的には，**ordinal_max**は以下のような関数で実装することが可能である．

```
int ordinal_max()
{ static int ord=0; return ord++; }
```

- COPY($R$,<$S,T$>) 参照<$R,T$>を生成する．参照強度は$R$と$T$の順序数により決定される．

  $$<R,T>.type := \begin{cases} strong & \text{if } R.ord < T.ord \\ weak & \text{otherwise} \end{cases}$$

- DELETE(<$R,S$>) $R$から$S$への参照を削除する．参照を削除した後，必要があれば，$S.obit$を反転させることで$S$を指す全てのweak参照をstrong参照に変換し，$S$の順序数をシステム中の最大数に置き換える．これは，不変則2を保つために要請される．その後SUICIDEプロセスを起動する．

  $S.obit := \textbf{NOT}(S.obit)$; $S.ord :=$ **ordinal_max**;
  **for** $T$ **in** $Sons(S)$ **do** SUICIDE($S$,<$S,T$>) **od**

  SUICIDEプロセスが終了した後，もし全てのstrong参照がweakに戻されているならば$S$はゴミであり，再帰的解放を行なう．

- SUICIDE(*Start*,<$R,S$>) 起点オブジェクト*Start*からstrong参照を辿ってオブジェクトをトレースすることで不変則1, 2を保全する．

SUICIDE(*Start*, <$R,S$>)
**if** <$R,S$>.*type* **is** *strong* **then**
**if** $S$ **is** *Start* **then** make <$R,S$>.*type* into *weak*
**elseif** $S.SRefc > 1$ **then** make <$R,S$>.*type* into *weak*
**elseif** $S.ord :=$ **ordinal_max**;
**for** $T$ **in** $Sons(S)$ **do** SUICIDE(*Start*, <$S,T$>) **od**
**end** of SUICIDE

以上のようにこれらの操作は，2つの不変則を常に保存するよう拡張されているので，strong参照による循環は決して生成されない．各オブジェクトは順序数を格納するために余分なスペースを必要とするが，拡張されたSUICIDEは，オリジナルと同じ時間複雑度で実現されている．

### 3.3 順序数集合の圧縮

DELETEおよびSUICIDE操作におけるオブジェクト順序数の再割り当ては，順序数の集合を疎にし，最終的には整数の桁溢れを引き起こす．これは**ordinal_max**が単調に増加するからである．

この問題を解決するためには，順序数集合の圧縮が必要となる．順序数集合の圧縮とは，理想的には，現在使用中のオブジェクト数を$n$とすると，各オブジェクトの順序数に0から$(n-1)$の整数のいずれかを矛盾なく割り当てることである．

順序数集合を圧縮するために，全ての使用中オブジェクトをルートからトレースするようなナイーブな方法は，参照の数を$e$とすると時間複雑度が$O(e+n\log n+n)$=$O(e+n\log n)$となる．ここで，$n\log n$の項は，順序数のソートによるものである．

全てのオブジェクトが“双方向リスト”で接続されていることを仮定すると，時間複雑度は容易に$O(n)$に減少させることが可能である．双方向リストを用いれば，リスト上でオブジェクトの順序を変更することがポインタの付け替えだけの定数時間で済むことが保証される．あるオブジェクトの順序数が現在の最大数に再割り当てされるとき，そのオブジェクトは，双方向リストの末尾に移動された上で**ordinal_max**が割り当てられる．3.2節でCプログラム風に記述された**ordinal_max**は，双方向リスト最適化のもとでは，一つ前方のオブジェクトの順序数に1

加えたものになる．また，ゴミオブジェクトの回収においては，双方向リストから削除し，フリーリストに繋ぎ直す操作が必要であるが，これも定数時間で終了する．

さて，双方向リストを用いて圧縮は以下のように行なわれる．(1) リストの先頭のオブジェクトに順序数として 0 を与える．(2) 続いてリストを前向きに手繰り，1 大きい整数を順序数として与える．(3) リストの末尾に到着するまで Step 2 を繰り返す．

# 4 SUICIDE 操作の並列拡張

疎結合並列マシンでは，効率良くゴミを回収するために，各プロセッサ毎に局所的なメモリ管理を担当する SUICIDE プロセスを配置する必要がある．そこで SUICIDE のマルチプロセス実行のための基礎を与えるのが本節の目的である．さらに，オリジナルの CRC では考察されなかった SUICIDE プロセスの遅延起動について取り上げる．

## 4.1 マルチプロセス実行

ある SUICIDE プロセスが実行されているとき，不変則 1 が保持されない状況が一時的に生じうる．それゆえ，複数の SUICIDE プロセスが実行される状況では，ある SUICIDE プロセスは，他のプロセスが生成した strong 参照による循環構造を辿ることでループに陥る可能性がある．

あるオブジェクト $A$ に対してある SUICIDE が起動されると直ちに，$A$ には'SUICIDE 動作中' という印が付けられる．各 SUICIDE プロセスは，起点となったオブジェクトによって区別される．オブジェクト $A$ を起点とする SUICIDE プロセスを $\text{SUICIDE}_A$ と表記する．また，そのプロセスが実行中である場合には，"$A$ は，SUICIDE 動作中にある" といい，$\vec{A}$ と表記して $A$ と区別する．

先の SUICIDE 操作を複数のプロセスで並列に実行可能とするとき，次のような拡張方式が考えられる

- SUICIDE の第一引数をオブジェクトのポインタ *Start* からポインタのリスト *ListofStarts* に拡張することによって，他の SUICIDE 動作中のオブジェクトによって作られた循環もチェック可能となる．プロセス $\text{SUICIDE}_A$ が，あるオブジェクト $\vec{B}$, $B \neq A$ を訪問した時に，そのプロセスは $B$ へのポインタを第一引数に連接して再格納する．

- 各 SUICIDE プロセスに優先度を割り当てることで，プロセス同士の調停が可能になる．$\text{SUICIDE}_A$ が，それより低い優先度の SUICIDE が動作中のオブジェクト $\vec{B}$ を訪問した時，$\text{SUICIDE}_A$ は $\vec{B}$ のトレースを中断する．優先度の高低が逆の場合は，トレースを取り止める．

どちらの方式も単一 SUICIDE プロセスのみ動作中であるときはオーバーヘッドがない．しかしながら，複数のプロセスが動作中である状況では，前者は，ポインタの比較演算の回数が'SUICIDE 動作中' のオブジェクトに出合う度に増加することなる．それゆえ，$CRC_{IW}$ 方式では，時間効率の観点から後者の方式を採用する．

SUICIDE の並列拡張版は以下のように実行される．

1. $\text{SUICIDE}_A$ は，'SUICIDE 動作中' のオブジェクト $\vec{B}$ を訪問しない限り strong 参照を辿る．

2. $\text{SUICIDE}_A$ は，ある strong 参照 $F$ により指される $\vec{B}$ を訪問したとき，もし $\text{SUICIDE}_A$ の優先度が $\text{SUICIDE}_B$ よりも低いならば，$\text{SUICIDE}_A$ は，$F$ の参照タイプを weak に変更し，$\vec{B}$ のトレースを取り止め，オブジェクト $B$ に'visited' という印をつける．さもなくば，$\text{SUICIDE}_B$ の実行が終了するまで，$\text{SUICIDE}_A$ による $\vec{B}$ のトレースを中断する．SUICIDE プロセスの優先度としては，起点オブジェクトの順序数を利用する．

3. $\text{SUICIDE}_B$ の実行が終了したとき，中断していた SUICIDE プロセスの実行を一部巻き戻した上で再開する．巻き戻しは，あるオブジェクトから参照 $F$ を通って $B$ へ至るステップである．$\text{SUICIDE}_B$ の終了時，$F$ は weak 参照であるかもしれない．このとき，$B$ のトレースは不要である．もし $B$ に 'visited' とマークされているならば Step 4 へ，さもなくば Step 5 へ進む．

4. もし $B$ の strong 参照カウントが 0，weak 参照カウントが 0 でないならば $\text{SUICIDE}_B$ の再実行を行なう．これは，ある $\text{SUICIDE}_X$ プロセスが Step 2 にて strong 参照を weak に変更し，$B$ のゴミ判定に干渉したからである．

5. もし $B$ の strong 参照カウントが 0 であるならば $B$ はゴミである．$B$ より再帰的解放を実行する．

## 4.2 SUICIDE 起動の遅延

strong 参照カウントが 0，かつ weak 参照カウントが 0 でないオブジェクト $A$ がゴミであるか否かは，$\text{SUICIDE}_A$

の実行結果によって決定される．$A$ に対するゴミ判定は，直ちに行なわれる必要がなく，遅延実行が可能である．

こういったゴミと疑わしきオブジェクトへのポインタは，遅延スキャンを行なうためにSUICIDE タスクプールに格納される．もしあるオブジェクトが実際にはゴミでないならば，このオブジェクトを指すweak 参照の一部は，アプリケーション実行によって削除されるかもしれない．それゆえ，SUICIDE プロセスが遅延されている間に全てのweak 参照が消去されることもあるうる．あるオブジェクトに参照がまったくなければ，直ちにそのオブジェクトを再利用することができるので，SUICIDE の遅延実行は有利な戦略である．

SUICIDE を遅延実行するためには，各オブジェクトに'遅延' 状態を表す1 ビットを用意する．あるオブジェクトの最後のstrong 参照が削除されたときに，そのオブジェクトの遅延状態ビットが設定され，オブジェクトへのポインタがSUICIDE タスクプールに格納される．SUICIDE プロセス起動前に全ての参照が削除された場合には，タスクプールからそれへのポインタが削除され通常通り再帰的解放が行なわれる．他方，SUICIDE プロセスが実行中であるならば，SUICIDE プロセスにオブジェクトの解放が委任される．

## 5 遠隔通信オーバーヘッドの低減

3.2節で与えられたCOPY(R,<$S,T$>) 操作は，新たに生成される参照<$R,T$> に $R$と $T$の順序数のみから決定される強度を割り当てる．この強度は言語のセマンティックなコンテキストと独立に定まるという点が特色である．

あるオブジェクト $T$がリモートメモリ上に存在している場合には，$T$の順序数を取り出すためにリモートアクセスが必要となり，通信オーバーヘッドを引き起こす．それゆえ，疎結合並列マシンをターゲットとするためには，ストリクトに参照の強度を決定するこれまでの技法に代わるものが必要である．

### 5.1 参照強度 unknown の導入

表1に，COPY(R,<$S,T$>) 操作における $R$, $S$, $T$の順序数の大小関係と参照<$R,T$>の強度の割り当てを示す．最初の3カラムは，3つのオブジェクトを順序数の昇順に並べ表示している．参照<$S,T$>の強度はSUICIDE プロセスによってstrong からweak に変更される場合があるので，strong/weak と表記されている．

表 1: 参照強度の割り当て

| Order | | | <$S,T$> | <$R,T$> |
|---|---|---|---|---|
| R | S | T | strong/weak | strong |
| R | T | S | weak | strong |
| S | R | T | strong/weak | strong |
| T | R | S | weak | weak |
| S | T | R | strong/weak | weak |
| T | S | R | weak | weak |

参照<$R,T$>の強度を局所的な方法を用いて割り当て，表1を簡約化することを試みる．順序数によってstrong に割り当てられるべき参照<$R,T$>をweak とすることは，不変則1, 2 に反しないので常に許される．また，参照<$R,T$>の強度の集合が参照<$S,T$>の強度の集合を被覆する場合には，コピー先参照<$R,T$>の強度をコピー元参照<$S,T$>と同じに設定することができる．

$S.ord < T.ord < R.ord$ の成り立つ場合には，参照<$R,T$>は，<$S,T$>の強度とは独立にweak と割り当てられなければならない．ここで，$R$と $S$の順序数の大小が知られていることを仮定したとき，$S.ord < R.ord$ を満たす3 つの場合における，コピー先参照の強度<$R,T$>.$type$ は，常にweak と割り当てることが可能である．3.1節で示されたBrownbridge の実装技法によれば，参照強度は，参照元と参照先に用意されたそれぞれのビットの排他的論理和によって計算される．それゆえ，参照<$R,T$>にweak を割り当てるためには，$T.obit$ のリモートアクセスが必要になる場合がある．

こういったリモートアクセスを避けるために，新しい参照タイプとしてunknown を導入する．unknown は一時的なものであり，いずれはweak に確定されるものであるとする．このunknown タイプの導入によって，表1は表2のように簡約化される．表中のX は，参照タイプのマッチングを表す．

DELETE 操作における，オブジェクトの参照ビット $obit$ の反転は，weak 参照をstrong 参照に変換するが，unknown 参照には影響を与えない．ここで，系1の対偶命題を拡張して以下の系2が得られる．

**系 2** あるオブジェクト $R$の *strong* 参照カウントと *unknown* 参照カウントが共に *0* であるならば，$R$はゴミである．

表 2: unknown タイプの導入による簡約化

| Order | | | <S,T> | <R,T> |
|---|---|---|---|---|
| R | S | T | X | X |
| R | T | S | X | X |
| S | R | T | * | unknown |
| T | R | S | X | X |
| S | T | R | * | unknown |
| T | S | R | * | unknown |

'*' は，don't care を表す．

参照の DELETE 操作に伴う SUICIDE プロセスは，unknown 参照を weak に確定する．あるオブジェクトの strong 参照カウント $SRefc$ が 0 になった場合には，他の 2 つの参照カウント weak 用の $WRefc$ および unknown 用の $URefc$ の値によって，異なるオブジェクト管理手続きが実行される．これを図 1 に示す．ここで，+は 0 以上の値を表現している．

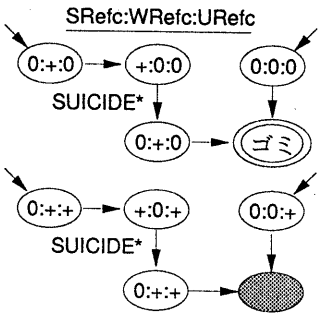


図 1: 参照カウントの遷移とゴミ判定

図上段は $URefc$ が 0 の場合である．weak 参照を strong 参照に変換した後，起点オブジェクトから strong 参照と unknown 参照を辿る．unknown 参照は weak に置き換えられる．strong 参照を辿った場合の処理はこれまでに示したものと同様である．図下段左の $WRefc, URefc$ 共に 0 でない場合も同様の処理を行なう．しかしながら，SUICIDE によって全ての strong 参照が weak に戻された場合は，系 2によってこのオブジェクトをゴミと判断することはできない．また，図下段右のように $URefc$ のみ 0 でない場合においても同様である．

こういったゴミと疑わしきオブジェクトに対しては，そのオブジェクトが保持する全ての参照を weak に変換する補助操作を実行する．各参照には逆ポインタが存在していないので，unknown 型入力参照を他の強度に直接置き換えることはできない．unknown 参照のみで構成されるゴミサイクルの除去のためには，こういった補助操作が必要となる．

## 5.2 重み付き参照

参照の操作に伴う大域的なプロセッサ間同期とメッセージ数を低減するために，分散 RC の一方式として重み付き参照カウント方式 (Weighted Reference Counting; WRC)[3] が提案されている．WRC 方式では，COPY(R,<S,T>) 操作において，コピー先参照<R,T>にコピー元参照<S,T>の半分の重みを分配することで，参照先 T へ参照カウントの増加を知らせる通信を削減することが可能になっている．

コピー先参照<R,T>の強度がストリクトに決定できないとき unknown が割り当てられるような $CRC_{IW}$ の枠組みにおいては，コピー元参照<S,T>は，その強度に関わらず，unknown 参照の重みを内包しなければならない．このために，参照は表 3 で示されるように重みの組で表される．以下で $ab_{(2)}$ は二進数 2 ビットで表現された識別子である．B は 2 ビット長に拡張された参照ビット $rbit$ を表す．strong, weak 参照は，$00_{(2)}$ または $01_{(2)}$ で表される．$11_{(2)}$ は，unknown 参照に直接対応する．

表 3: 参照の強度と重みの構成

| | |
|---|---|
| strong | $\{(B, W_s), (11_{(2)}, W_u)\}$ |
| weak | $\{(B, W_w), (11_{(2)}, W_u)\}$ |
| unknown | $\{(11_{(2)}, W_u)\}$ |

参照の COPY 操作における重みの分配は，3 つのオブジェクトが相対的にどこに存在するかによって異なる．以下では，S の置かれたプロセッサ上で COPY 操作が行なわれ，R と T は別々の分散プロセッサ上に配置されている場合を取り上げている．このとき，新しく生成される参照<R,T>の強度は，常に unknown となる点に注意する．

**ソース側の処理:** コピー元参照<S,T>の重みのうち unknown 成分に対する重みを半分に分割する．もし unknown 成分に対する重みが 1 であるならば，標準的な WRC と同様に代理オブジェクト (proxy object) を置き，unknown 成分の重みを最大値に再設定した上で重みの分割を行なう．

**ターゲット側の処理:** 分割した重み W と T へのポインタを R にメッセージ送信する．R はそのメッセージを受信し，参照強度 unknown，重み W で初期化された参照<R,T>を生成する．

## 6 まとめと今後の課題

$CRC_{IW}$の特徴は以下のような3点にまとめられる.

- オブジェクトに順序数を導入し，参照に関する操作を拡張することによって，任意のポインタ操作を許す言語に適用可能である.
- 低い優先度プロセスによって干渉を受けたプロセスを十分遅延させた上で再実行することによって，複数のSUICIDEプロセスを健全に並列実行可能である.
- 3種類の参照タイプstrong, weak, unknownと参照の重みを導入したことによって，参照のCOPY操作における通信オーバーヘッドを低減している. unknown参照は，SUICIDEプロセスによってweak参照に置き換えられる.

また本稿は，オブジェクトの順序数の再割り当てによる順序数の桁溢れ問題を考察し，順序数集合の圧縮が最適な時間複雑度で行なえるようなオブジェクトの管理方式を示した.

今後の課題として，正当性の証明と実機での有効性の検証が残されている. 我々が現在設計を進めている超並列オブジェクトベース言語 *OCore*[11] のランタイムとして適切であるか，いくつかの商用並列マシン上に実験システムを構築し，評価する予定である.

## 謝辞

日頃から御討論頂いているRWC超並列プログラミング言語ワーキンググループおよび超並列ソフトウエアワークショップグループのメンバの方々，及び本稿の構成に関して助言を頂いた筑波大学の田中二郎氏に感謝致します。

## 参考文献

[1] Saleh E. Abdullahi, Eliot E. Miranda, and Graem A. Ringwood. Collection schemes for distributed garbage. In Y. Bekkers and J. Cohen, editors, *LNCS 637*, pp. 43–81. Springer-Verlag, 1992. IWMM'92 Proceedings.

[2] Lex Augusteijn. Garbage collection in a distributed environment. In J. W. de Bakker, A. J. Nijimans, and P. C. Treleaven, editors, *LNCS 259*, pp. 75–93. Springer-Verlag, 1987.

[3] D. I. Bevan. Distributed garbage collection using reference counting. In J. W. de Bakker, A. J. Nijimans, and P. C. Treleaven, editors, *LNCS 259*, pp. 176–187. Springer-Verlag, 1987.

[4] D. R. Brownbridge. Cyclic reference counting for combinator machines. In *LNCS 201*, pp. 273–288. Springer-Verlag, 1985. ACM Conference on Functional Programming Languages and Computer Architecture '85 Proceedings.

[5] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. *ACM SIGPLAN Notices*, Vol. 24, pp. 313–321, 1989.

[6] Richard E. Jones and Rafael D. Lins. Cyclic weighted reference counting without delay. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *LNCS 694*, pp. 712–715. Springer-Verlag, June 1993. 5th International PARLE Conference Proceedings.

[7] Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *LNCS 637*, pp. 1–42. Springer-Verlag, 1992. IWMM'92 Proceedings.

[8] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, No. 11, pp. 181–198, 1990.

[9] 坂井修一, 岡本一晃, 松岡浩司, 廣野英雄, 児玉祐悦, 佐藤三久, 横田隆史. 超並列計算機rwc-1の基本構想. 並列処理シンポジウムJSPP'93, pp. 87–94, 1993.

[10] 鎌田十三郎, 松岡聡, 米澤明憲. 超並列計算機上の高効率な大域的ガーベジコレクション. 情処研報, Vol. 93, No. 73, pp. 121–128, August 1993.

[11] 小中裕喜, 石川裕, 前田宗則, 友清孝志, 堀敦史. 超並列オブジェクトベース言語 *OCore* の概要. 情処研報, *Vol. 93, No. 73, pp. 89–96, August 1993.*