複合タグ方式の Lisp 処理系におけるガベージ・コレクタの実現とその問題点†

高橋 俊成††

最近の汎用マイクロプロセッサは32ビット・アドレスのものが主流であるため，ポインタ・タグによるLisp処理系での性能は期待できない．今後はアドレス・タグとオブジェクト・タグとを用いた複合タグ方式が主流になろう．本論文では，複合タグ方式を採用した処理系の例としてUtiLisp 32を取り上げ，そこで実現したmark & sweepによる一括型ガベージ・コレクションのアルゴリズムを報告する．これはインタラクティブな使用に十分耐える高速なものであり，主にMorrisのアルゴリズムを基本として，より一般的な複合タグ方式の処理系に適用可能な形に改良したものである．ここではUtiLisp 32に依存する形で述べているが，系統的な修正により一般の処理系にも活用できる．さらにはガベージ・コレクションとLisp処理系の信頼性との関係に触れ，処理系の一部としてのガベージ・コレクション実現の問題点をあげ，その解決法を論じる．

1. はじめに

近年のLisp処理系はリスト・プロセッサとしての性格が薄れ，汎用のプログラミング言語として実用的な機能を持つようになったため，Lispオブジェクトとして取り扱うデータの型が多様化している．一方，Lisp処理系の性能はメモリ・レイアウトおよびタイプ・チェックの方法に強く依存するので，データ格納の方式は複雑になりがちである．これに伴い，Lisp処理系の弱点とも言えるガベージ・コレクション（以下GCと略す）に大きな負担が掛かることになる．

GCについてはコピー方式や実時間型，並列型，オブジェクトの存在時間考慮のものなど各種論じられているが[14)~20)]，それらの多くは，典型的なリスト・モデル上での実現の域を出ることが困難であったり，特別のハードウェアを必要としたり，肝心のリスト処理の速度を犠牲にしているなど，汎用計算機上での利用に限定して考えた場合は意義の少ないものである．

本論文の趣旨はGCの手法に関する単なる一般論ではなく，実在する複雑な処理系におけるGCの具体的実現の一方法を提示することである．なお，この方法はUtiLisp[6),7),10)]の32ビット版であるUtiLisp 32[11)~13)]で採用している．UtiLispはもともとメインフレームおよびMC 68000[8),9)]をプロセッサとする計算機用に開発された．当時はこれらの計算機のアドレスバスが24ビットであることを利用し，ポインタ・タグ方式により実行速度を上げていた．今回，UtiLispをMC68020, VAXなど32ビット・アドレスの汎用マイクロプロセッサ上で実現するために，新たなGCの方法を考える必要が生じた．メインフレームのUtiLispは複式ヒープ領域のコピーGCでありMC 68000のUtiLispのGCは基本的には本方式と同じである．

第2章ではモデルとするLispシステムの特長を示し，第3章ではそこでのGCの実現方法を紹介する．また，第4章ではGCの存在に起因するLispシステム自体の構成の難しさについて論じる．

2. UtiLisp 32の複合タグ方式タイプ・チェック

UtiLisp 32において，Lispオブジェクトのタイプ・チェックは次の2つの方法を組み合わせて実現している．

1) 各々のLispオブジェクトに直接タグを書く．（オブジェクト・タグ）
2) ヒープ領域を分割し，存在位置によって知る．（アドレス・タグ）

一般に，前者はメモリ配分の効率を重視するものであり，後者はタイプ・チェックの速度を重視するものである．UtiLisp 32は，シンボルおよびリストには後者を用い，それ以外のもの（以下othersと呼ぶ）*には前者を用いることにより，性能の維持を適切に行っている（図1）．UtiLispの経験では，othersのオブジェクトはそれ自身へのメモリ参照が多く，タイプ・チ

† Garbage Collector for a Complex-Tagged Lisp Processor and the Related Issues by TOSHINARI TAKAHASHI (Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部計数工学科

* flonum, bignum, stream, string, vector, codeの6種類がある．

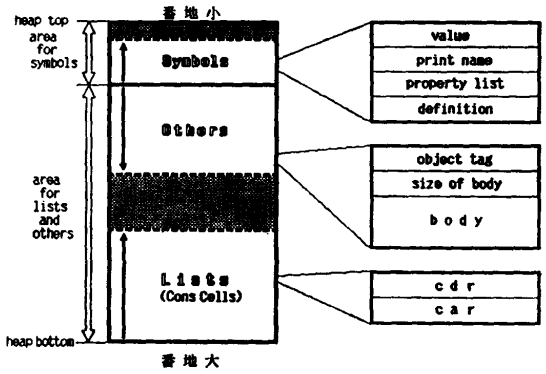

図 1 UtiLisp32 におけるヒープ領域の使用法
Fig. 1 Heap division of UtiLisp32.

図 2 fixnum の格納形式
Fig. 2 Format of a fixnum.

図 3 bound symbol の格納形式
Fig. 3 Format of a bound symbol.

ェックの速度は支配的にならないことが多いので，オブジェクト・タグでもさほど性能は低下しないであろうと判断した．

特別の扱いをするものとして fixnum（単精度整数）がある．これは他のオブジェクトのようにポインタで間接的に指されるのではなく，**図 2** のようにポインタを格納するセルに直接に値を書き込む（上位 2 ビットは fixnum 識別用，下位 2 ビットは GC 用の作業領域（後述）である）．したがって，fixnum の使用によってメモリ領域を消費することがなく，GC の後も fixnum はそのままにしておけば良い．

また，特殊なものとして bound symbol がある．これはシャロウ・バインディングを行う変数シンボルへのポインタであり，バインディングを解く際にシンボルへの普通のポインタと区別するため，**図 3** のように最上位のビットを立てる．

3. UtiLisp 32 における GC の実現方法

3.1 マーキング

マーキングは，具体的には参照可能なすべてのオブジェクトの先頭のセルにマーク・ビット（図中Ⓜと略す）を付けることで実現する．それには，すべての Lisp オブジェクトのアドレスを 4 の倍数にすることにより余った下位 2 ビットをマーク・ビットおよびストップ・ビット（後述）として使用する．fixnum は下位 2 ビットを作業用に残しておき，他のオブジェクトと混在しても同一の処理ができるようにする．ここで，“参照可能”とは現在必要なオブジェクトへのポインタの配列（ルートと呼ぶ）からポインタをたどって到達可能なことである（つまり“ゴミ”ではない）．（本論文ではスタック内のポインタなどもルートの一部として呼ぶ．）

マーキングは循環を避けるため preorder traversal（先行順走査）[5] により実現する．ただし再帰的方法では作業領域が必要となり好ましくない．これは以下に述べる逆転リンクを用いた方法（Schorr, Waite[4]）により解決する．

まず，1 つのポインタ（A）で訪問先（現在走査しているオブジェクトまたはそのセル）を記憶する．また，もう 1 つのポインタ（B）で訪問先のオブジェクトに到達する 1 つ手前のオブジェクトを記憶する（逆転リンクの出発点となる）．ルートへは B から逆転リンクをたどれば到達できる（**図 4**）．よって，余分な作業領域を全く使用せずに走査を続行・完了することができる．

一般に，1 つのオブジェクトは複数のセルを持つので，1 段バック・トラックするごとにあるオブジェクトのマーキングがどのセルまで終了しているのかを調べなければならない．これは訪問先の型と番地から知ることもできるが，それでは 2 段階の処理が必要である．そこで訪問先のマーキングが終了したことを示すためにストップ・ビット（図中Ⓢと略す）を用いる．例えば，シンボルの場合は，**図 5** のように最初に先頭のセルにストップ・ビットを付け，一番下（番地の大きい方）のセルから上（番地の小さい方）に向かってマーキングを行い，ストップ・ビットが現れるまで続ける．なお，付けたストップ・ビットはそのオブジェクトのマーキング終了時に取り去ってしまう．

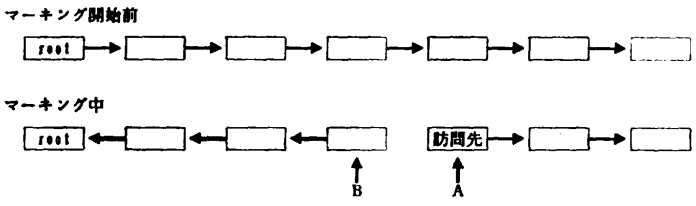

図 4 逆転リンクを用いたマーキング
Fig. 4 Marking with a reversed link method.

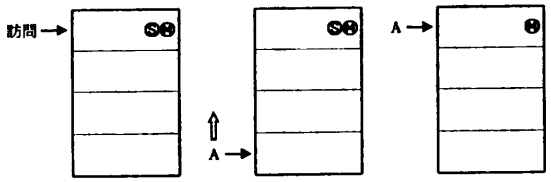

図 5 ストップ・ビットを用いたシンボルのマーキング
Fig. 5 Marking of a symbol using a stop-bit.

以下，各オブジェクトごとにマーキングの方法を述べる．

1) シンボル，リストのマーキング

シンボルやリストのように，ポインタだけを持つオブジェクトの場合は，前述（図5）のようにマークする．

2) flonum, bignum, string のマーキング

それ自体ポインタを持たないものは単にタグ部にマークを付けるだけで良い．

3) vector のマーキング

vector はタグとサイズのセルのほかに任意個のポインタを持つ．これは**図6**のように先頭のポインタ部にストップ・ビットを付け，サイズの値を基に一番下の位置を知り，1)の場合と同様に下からマークする．

4) stream のマーキング

stream は1つのポインタ (path-name) と複数のポインタでないセルを持つ．これは**図7**のようにポインタ部にストップ・ビットを付け，要素が1つの vector と同様に扱う．

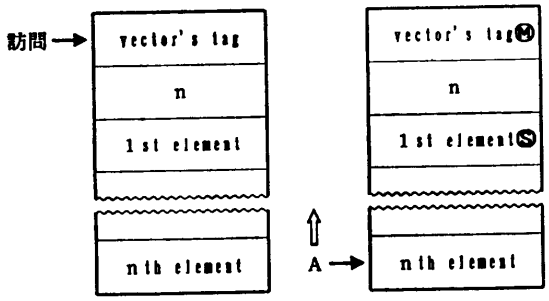

図 6 vector のマーキング
Fig. 6 Marking of a vector.

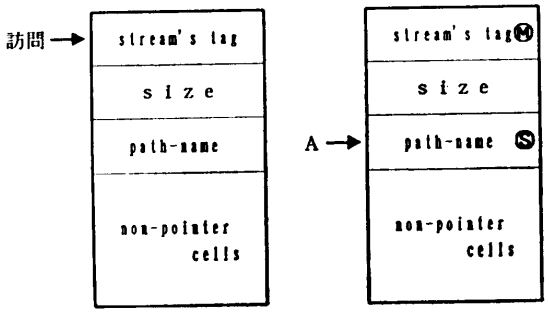

図 7 stream のマーキング
Fig. 7 Marking of a stream.

5) code のマーキング

code のマーキングは特殊である．code の持つポインタは function-name（関数名を表すシンボル）および quote-vector（コンパイルされた関数が使用する Lisp オブジェクトへのポインタ）の2種類であるが，それぞれは離れた位置にありしかもその相対位置は変化する（**図8**）．quote-vector の位置と大きさは size と maxparam（関数の引数の最大値）で知る．まず quote-vector をマークする時は先頭にストップ・ビットだけでなくマーク・ビットも付けると共に，maxparam を quote-vector の上に移動しておき，下からマークする．これで quote-vector のマーク終了後に（Aの値とマーク・ビットとから，quote-vector のマークをしていたことがわかり）1つ上の maxparam の値から function-name の場所を知る．以降は stream の場合と同様である．

6) reference のマーキング

reference とは vector 中の1つの要素を直接参照するポインタである（**図9**）．まず reference から指されている vector であることを示すために，そのタグ部にストップ・ビットを付ける．また vector のタグを消し，そこに reference への相対位置を書き込む（タグ部に付けたストップ・ビットは，そのオブジェクトが vector であることと，reference から指さ

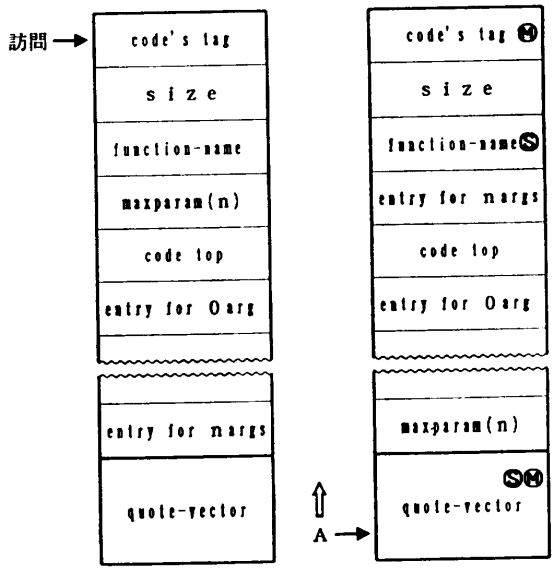

図 8 code のマーキング
Fig. 8 Marking of a code.

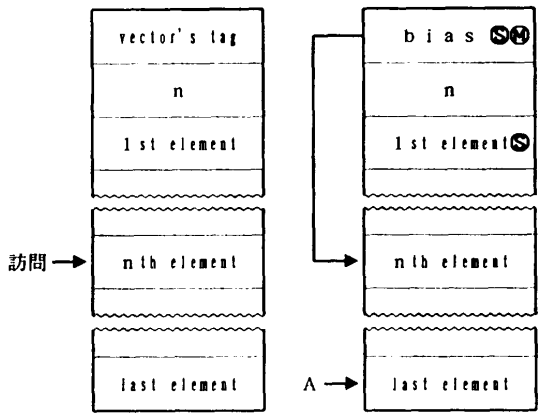

図 9 reference から指された vector のマーキング
Fig. 9 Marking of a vector pointed by a reference.

れていたことの両方を示している). あとは vector の場合と同様に行う.

3.2 コンパクション

マーキングの終了後にコンパクションを行う. 後述の手順により, シンボル, リスト, others の3つの領域についてスライディング・コンパクションを行うことができる. 基本的には, シンボルおよびリストについては Morris[1] のアルゴリズム, others については Knuth[3] のアルゴリズムを参考にしている. 当然, 前者と後者を単独に処理することはできないので, 2つのアルゴリズムを適切に組み合わせて実現している. 前者は2パスで完了するが後者は3パス必要である. others が2パスでは処理できない理由は次のとおりである. まず, others はオブジェクトの生成方向と反対向きには走査することができないので, 例えば Morris のアルゴリズムは適用できない. また, Jonkers[2] のアルゴリズムの適用は reference の存在が阻害する*. しかし3パス必要であることは結果的に問題にならなかった. なぜなら, 1パス目はオブジェクトの先頭に触れるだけであり, むしろポインタの更新作業が容易になったことによる効果の方が大きかったからである.

3.2.1 コンパクションに必要な基本操作

まず, アルゴリズムの説明に使用する用語の定義をしておく.

"thread する" とは, あるオブジェクトを指すポインタを逆転ポインタでつなぐことである. その際, 逆転ポインタは逆転ビット (図中Ⓡと略す) を付けることによって普通のポインタと区別する (図 10). マーキング後は, 既にストップ・ビットは消されているので, これを逆転ビットとして使う. シンボルへのポインタは普通のポインタと bound symbol としてのポインタを区別する. これには図 11 のように bound symbol の格納されていたセルへの逆転ポインタにマーク・ビットを付ける. アルゴリズムの手順により, このマーク・ビットはマーキングのために付けたマーク・ビットとは区別できる.

"下向きポインタ" とはそのポインタの格納番地よりも大きな番地を指すような普通のポインタである.

"上向きポインタ" とはそのポインタの格納番地よりも小さな番地を指すような普通のポインタである.

"ポインタを更新する" とはポインタの値を GC 終了後の値にすることである. 必ずしもそのポインタを新しい格納位置に移動することは意味しない.

"unthread する" とは thread されていた1組の逆転ポインタを解き, 自分自身を除くすべてのポインタを更新することである.

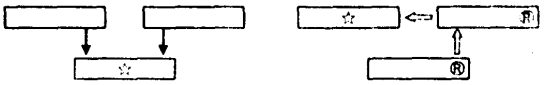

図 10 thread の方法
Fig. 10 Method of thread.

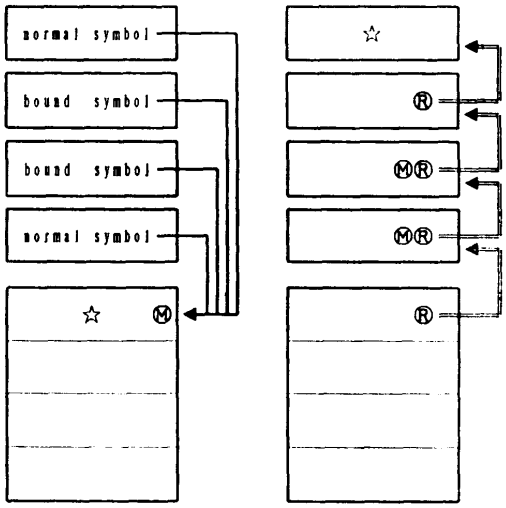

図 11 シンボルの thread
Fig. 11 Thread of symbols.

3.2.2 コンパクションのアルゴリズム

コンパクションの実現は次の流れによる.

1° others を上から走査する. タグとサイズを1箇所に圧縮して出来た領域にそのオブジェクトの移動先アドレスを書く (図 12).

2° ルートを走査する. シンボル, リストへのポイ

* 参照先もまたポインタであることに注意.

表 1 コンパクション実行中におけるポインタの更新手順
Table 1 Sequence of updating pointers in compaction phase.

ポインタの種類		処理手順							
存在場所	指し先	1°	2°	3°	4°	5°	6°	7°	8°
ルート	シンボル	O	O→R	R	R→N	N	N	N	N
	リスト	O	O→R	R	R	R→N	N	N	N
	others	O	O→N	N	N	N	N	N	N
シンボル	シンボル (下向き)	O	O	O	O→R→U	U	U	U→N	N
	シンボル (自分自身)	O	O	O	O→U	U	U	U→N	N
	シンボル (上向き)	O	O	O	O	O	O	O→R→N	N
	リスト	O	O	O	O→R	R→U	U→N	N	N
	others	O	O	O	O→U	U	U	U→N	N
リスト	シンボル	O	O	O	O	O	O→R	R→N	N
	リスト (下向き)	O	O	O	O	O→R→U	U→N	N	N
	リスト (自分自身)	O	O	O	O	O→U	U→N	N	N
	リスト (上向き)	O	O	O	O	O	O→R→N	N	N
	others	O	O	O	O	O	O→N	N	N
others	シンボル	O	O	O→R	R→U	U	U	U	U→N
	リスト	O	O	O→R	R	R→U	U	U	U→N
	others	O	O	O→U	U	U	U	U	U→N

O：旧ポインタ
R：逆転ポインタ
U：更新されたポインタ (値が新しくなっただけで，まだ格納場所は移動していない)
N：更新されたポインタ (しかも新しい番地に配置されている)

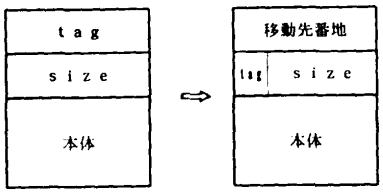

図 12 others の1パス目
Fig. 12 Others' pass 1.

ンタは thread し，others へのポインタは更新する．ポインタでないものはそのままにする．

3° others を上から走査する．シンボル，リストへのポインタは thread し，others へのポインタは更新する．

4° シンボルを上から走査する．逆転ポインタであれば unthread し，その後リストへのポインタとシンボルへの下向きポインタは thread し，others へのポインタは更新，シンボルへの上向きポインタはそのままにする．自分自身へのポインタは更新する．

5° リストを上から走査する．逆転ポインタは unthread し，リストへの下向きポインタは thread，シンボルと others へのポインタおよびリストへの上向きポインタはそのままにする．自分自身へのポインタは更新する．

6° リストを下から走査する．逆転ポインタは unthread し，シンボルへのポインタとリストへの上向きポインタは thread，others へのポインタは更新する*．リストへの下向きポインタはそのままにする．また，自分自身へのポインタもそのままにする**．いずれの場合も同時に新しい番地に配置する．

7° シンボルを下から走査する．逆転ポインタは

* シンボルから others へのポインタは1パス目に更新したが，リストから others へのポインタは2パス目に更新する．これは，1パス目に更新すると余分なタイプ・チェックを要するからである．

** 5° に於ける自分自身へのポインタは，ここでは必ずしも自分自身へのポインタとして観測されない．下向きポインタに変化している場合もある．

unthread し，シンボルへの上向きポインタは thread し，その他のポインタはそのままにする．いずれの場合も同時に新しい番地に配置する．

8° others を上から走査する．タグとサイズを元に戻し，新しい番地に配置する．

なお，1°，4°，5° において連続するゴミに対し1つのリンクを張る（ゴミリンク）．ゴミリンクをたどることにより以降の走査は参照可能なオブジェクトだけで済む．経験的に，多くのプログラムにおいてリストと others は大半がゴミになるので，この効果は大きい．また，ゴミリンクの使用によってマークを消すことができるので，thread, unthread の作業が容易になる．

以上の処理中に各々のポインタが変化していく様子を示したのが**表1**である．

3.3 性　　能

GC の起動から終了まで一貫して作業領域（スタック等）を全く使わない．したがって，コンパクション方式であることと相まって，メモリ効率は最大限に良い（実際に，UtiLisp 32 の高速性をそこなうことなく，コンパクトなシステムが実現された）．また，処理時間は使用するヒープ領域の大きさに比例するという欠点はあるものの，例えば UtiLisp 32 のデフォルト値（ヒープサイズ 512 K バイト）の場合，SUN 3/260 で 0.1〜0.3 秒，VAX 8600 で 0.2〜0.7 秒程度であり，インタラクティブな使用環境で全く気付かないくらいの速度が実現できた．

4. GC の存在に起因する Lisp 処理系作成の問題点

GC の実現において，Lisp 処理系自体の構成と別個に考えることはできない．その理由のうちおそらく最も重大な点は次のことである．

GC は，それが起動された時点の環境（ヒープおよびスタックの内容，作業用メモリなどすべてを含む）に対して

1) Lisp オブジェクトへのポインタは更新する，

2) 単なるデータ（以下数値と呼ぶ）は保存する，

ことが必要である．ところが特にアドレス・タグを使う処理系の場合，各データがこのいずれの処理をすべきものであるかを，GC の側から識別することはできない．したがって処理系本体と GC との間で，すべてのデータの格納方式を取り決めておかなければならない．

以下，GC 側がその内容をポインタとして扱う領域をポインタ領域，数値として扱う領域をデータ領域と呼ぶ．（一般にはスタックおよびヒープをポインタ領域として扱うのが普通であろう．）

4.1 階層的プログラムにおけるデータ格納の実際

複合タグ方式の処理系の場合，サブルーチンを呼ぶ際のデータ受け渡しは次に示すように非常に複雑である．まず，呼ぶ際にレジスタにある値は，大きく分けて次の4通りに類別できる．

A° Lisp オブジェクトへのポインタ

B° GC から識別可能な数値，すなわちポインタとしては存在し得ない数値

C° GC から識別不可能な数値，すなわちポインタとみなすことのできる数値

D° 内容が不明なもの，例えば用途が一定していないものや未使用のレジスタの内容など*

以上それぞれについて，戻って来た時の値に関して次の2通りに分類できる．

1° 値は保存されていなければならない．（ポインタはポインタとしての値が）

2° 保存されている必要がない．例えば返し値を持つ場合や壊れても構わない場合．

一方，呼ばれた側の扱いとしては，レジスタの値を

P° 退避せずに変更する可能性がある．

Q° 全く変更しない．しかも GC は起こらない．

R° 変更する場合はポインタ領域に退避する．

S° 変更する場合はデータ領域に退避する．または

表2 サブルーチン呼出し時のレジスタ受け渡し法
Table 2 Register delivering method of a subroutine call.

	P	Q	R	S
A-1	×	○	○	××
A-2	○	○	△	△
B-1	×	○	△	△
B-2	○	○	△	△
C-1	×	○	××	○
C-2	○	○	××	△
D-1	×	○	××	××
D-2	○	○	××	△

* 未使用のレジスタはコーディング時にないがしろにされがちであるが，それも処理系の信頼性に強く関わっていることに注意が必要である．

GC の時も含めて全く変更しない.

の4通りに分類できる.

以上の 4×2×4=32 通りの場合について正しい方法であるかを示したのが**表2**である.

表2において○を付けた方法が適切な受け渡し方である. ×を付けたのは明白な誤りであるが, ××を付けたのは GC が起きた場合のみ正しく動作しないので特に注意を要する. また△を付けたのは意味のない退避等が行われることを示しており, この方法が多用されると処理系の性能を落とすことになる.

4.2 Lisp 処理系の実現を容易にする手法

処理系の実現に際しては, 表2に示したきまりを常に守る必要があるが, それを処理系作成者の注意深さのみに依存していたのでは信頼性の保持は期待できない. そこで処理系の構成法に関して種々の制約を付けることによってインプリメントの負担を軽減する必要が生じる. 例えば次のような方法がある.

・個々のレジスタに対し, ポインタとして使用するかデータとして使用するかを決め, 他の使用法を禁止する. そして GC の有無に関わりなくポインタのデータ領域への退避・数値のポインタ領域への退避を禁止する.

このようなきまりのもとでの処理系構成の長所, 短所は,

1) レジスタ受け渡しの種類は**表3**のようになり表2の場合と比べ信頼性, 効率の両面で優れている.
2) レジスタ退避の効率が向上する. 例えば**図13**のプログラムにおいて alloc が各所から呼ばれる時, sample 1, sample 2 (呼び側) の平均処理速度はAよりもBの方が優れている. このことは, 基本的なリスト処理の速度に大きく影響する.
3) レジスタの使用法に融通がきかなくなるので, 複雑な機能の関数の記述時にレジスタが不足し, その高速化が困難になる恐れがある.

などである.

実際には処理系の仕様に依存して, 各種の制約を組み合わせることになろう.

表3 用途限定によるレジスタ受け渡し法
Table 3 Register delivering method by restricting register usages.

	P	Q	R	S
A-1	×	○	○	—
A-2	○	○	△	—
B-1	×	○	—	△
B-2	○	○	—	△
C-1	×	○	—	○
C-2	○	○	—	△
D-1	—	—	—	—
D-2	—	—	—	—

```
A :  sample1(·····;
            push pointer1, pointer2;
            call alloc;
            pop  pointer1, pointer2;
            ·····)
     sample2(·····;
            push pointer1, pointer3;
            call alloc;
            pop  pointer1, pointer3;
            ·····)
     alloc  (if exhausted then GC;
            allocate)

B :  sample1(·····; call alloc; ·····)
     sample2(·····; call alloc; ·····)
     alloc  (if exhausted then
                  (push pointer1, pointer2, pointer3;
                   GC;
                   pop  pointer1, pointer2, pointer3);
            allocate)
```

図13 レジスタの用途を限定することによる効率的なプログラム例
Fig. 13 An example of an effective program restricting register usages.

4.3 Lisp 処理系の信頼性に対する考察

処理系の保守をするため, 最終的にはユーザのバグ報告などに依存する面が残されているのが現状である. そのような意味もあり, GC 起動時に限って起こる異常は以下の理由により絶対に避けなければならない.

1) 再現性に乏しいのでデバッグが困難である.
2) 頻発性がないのでバグが水面下に隠れやすい.
3) 対症療法的なデバッグにより, その周辺すべてにバグを生み出す結果になる可能性が高い.
4) GC は暗に呼ばれる場合が多く, 作成時に行った注意をデバッグ時には忘れていがちである.

すなわち, 長期的な保守によって処理系の信頼性がどこに収束するかという観点から見れば, Lisp 処理系の信頼性評価の目安としては, 表面的なバグの量的な少なさよりも, いかに GC との整合性を考慮しているかということを重視すべきである.

5. むすび

本論文では UtiLisp 32 における GC の構成法を報告し，実現上の問題点を一般的に論じた．

GC による計算の停止を避けるため，実時間方式を採用するのも1つの逃げ道ではあるが，それでは本来の目的である計算そのものの性能を落とす．本論文での報告は，GC そのものの性能を向上させることにより問題を小さくしようとするものである．UtiLisp32 の場合は，その用途が汎用計算機での記号処理・数式処理等の一般的なものであると考え，後者の方式を採用することになった．

一方，第4章では Lisp 処理系作成時の実際的な難しさについて触れたが，この警告が今後処理系を作成する際に何らかの参考になれば幸いである．

謝辞 日頃から有益な助言をいただいている東京大学工学部計数工学科の和田英一教授および処理系作成に際して指導と協力をいただいた湯浅敬，金子敬一両氏，論文の作成の際に助言をいただいた寺田実，岩崎英哉両氏に感謝します．またガベージコレクションの実現法に関して富岡豊氏の発案を参考にさせていただきました．

参考文献

1) Morris, F. L.: A Time- and Space-Efficient Garbage Compaction Algorithm, *Comm. ACM*, Vol. 21, No. 8, pp. 662-665 (1978).
2) Jonkers, H. B. M.: A Fast Garbage Compaction Algorithm, *Inf. Process. Lett.*, Vol. 9, No. 1, pp. 26-30 (1979).
3) Knuth, D. E.: *The Art of Computer Programming Vol. 1*, p. 634, Addison-Wesley, Reading, Mass. (1968).
4) Schorr, H. and Waite, W. M.: An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, *Comm. ACM*, Vol. 10, No. 8, pp. 501-506 (1967).
5) Aho, A. V., Hopcroft, J. E. and Ullman, J. D.: *The Design and Analysis of Computer Algorithms*, p. 470, Addison-Wesley, Reading, Mass. (1974).
6) 近山 隆：Utilisp システムの開発，情報処理学会論文誌，Vol. 24, No. 5, pp. 599-604 (1983).
7) Chikayama, T.: Utilisp Manual, Technical Report, METR 81-6, Dept. of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, Univ. of Tokyo (1981).
8) 和田英一，富岡 豊：UtiLisp の MC 68000 への移植，情報処理学会記号処理研究会資料，84-29, pp. 15-21 (1984).
9) 湯浅 敬，金子敬一：UtiLisp の Macintosh への移植．その苦難の道のり，情報処理学会第27回プログラミング・シンポジウム報告集，pp. 131-141 (1986).
10) 寺田 実，岩崎英哉：UtiLisp の使い方，東京大学大型計算機センター・センターニュース，Vol. 18, No. 3, pp. 48-60 (1986).
11) Kaneko, K. and Yuasa, K.: A New Implementation Technique for the UtiLisp System, Preprints of WGSYM Meeting, Vol. 41, No. 7 (1987).
12) 金子敬一：VAX 上の UtiLisp，東京大学大型計算機センター・センターニュース，Vol. 19, No. 2, pp. 35-39 (1987).
13) 高橋俊成，石井 信：VAX 上の UtiLisp その後，東京大学大型計算機センター・センターニュース，Vol. 19, No. 7・8, pp. 69-73 (1987).
14) Fenichel, R. R. and Yochelson, J. C.: A LISP Garbage-Collector for Virtual-Memory Computer Systems, *Comm. ACM*, Vol. 12, No. 11, pp. 611-612 (1969).
15) Steele, G. L.: Multiprocessing Compactifying Garbage Collection, *Comm. ACM*, Vol. 18, No. 9, pp. 495-508 (1975).
16) Deutsch, L. P. and Bobrow, D. G.: An Efficient, Incremental, Automatic Garbage Collector, *Comm. ACM*, Vol. 19, No. 9, pp. 522-526 (1976).
17) Baker, H. G. Jr.: List Processing in Real Time on a Serial Computer, *Comm. ACM*, Vol. 21, No. 4, pp. 280-294 (1978).
18) 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, *Comm. ACM*, Vol. 21, No. 11, pp. 966-975 (1978).
19) Lieberman, H. and Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM*, Vol. 26, No. 6, pp. 419-429 (1983).
20) Rudalics, M.: Distributed Copying Garbage Collection, *Proc. ACM Conf. on Lisp and Functional Programming 1986*, pp. 364-372.

(昭和63年3月3日受付)
(昭和63年12月12日採録)

高橋 俊成（正会員）

1962年12月19日（水）生．1986年東京大学工学部計数工学科（数理工学専修）卒業．在学中はアナログ・レコードの光学式再生装置の研究・試作を経験．同年同大学院工学系研究科情報工学専門課程修士課程に入学し，計算機分野に転向．1988年同課程修了．同年(株)東芝入社．現在，同社総合研究所情報システム研究所にて基本ソフトウェアの研究に従事．