記　号　処　理　66－1
（１９９２．９．２２）

C言語におけるリスト処理のための記憶管理方式

小田　利彦

リコー　中央研究所

C言語による知識情報システムの開発を支援するリスト処理機能（**clp**）を実現した。本論文ではその記憶管理法について述べる。数値表現は記憶を消費しない表現にした上で、特にくずとなりやすいコンスセルに対するくず集め方式として、頻度の高い圧縮によるGCを行なう。この圧縮GCは、関数実行毎に駆動可能でありGCの処理が分散化できる。さらに不要セル率等からGCの実行判定を行ない、その回収効率を高めている。**clp**を利用して開発した応用システムを示し実用性を評価した。

A MEMORY MANAGEMENT METHOD FOR LIST PROCESSING ON C LANGUAGE

Toshihiko Oda

RICOH Research and Development Center

16-1, Shinei-chou, Kouhoku-ku, Yokohama-shi, Kanagawa-ken, 223, Japan

This paper describes a memory management method for list processing under C program. Its numerical representation is designed to waste no memory space during excution. On its garbage collection, ephemeral cons cells are compressed more frequently than other data therefore the GC process can be distributed not to disturbe system's running. By using some parameters ex. garbage-ratio, we can avoid to execute the useless GC. We have developed a practical AI application system and the GC works on the system successfully.

1 始めに

再帰的な構造を持つリスト型データ構造は、データを柔軟に構成かつ変更でき、ポインタ表現であるためデータの共有性が高い。また、異なる型のデータを混在させることも容易である。こうした性質は、複雑な対象を表現したりデータの構成を頻繁に変化させる知識情報処理に適している。我々は、診断型エキスパートシステムのC言語による開発において、知識表現やその推論処理等のためにリストを基本的データ構造とした。そこで、C言語プログラム上でリスト処理を行なうために、それが容易かつ自然に記述できまた速度や記憶効率で実用性の高い機構（clpと呼ぶ）を実現した。

clpは、リスト処理に必要なシンボル管理、くず集めなどの記憶管理,リスト操作関数群等から構成されている。いわゆるProlog、LISP等の言語処理系と異なり、関数定義や評価実行等のインタプリタ機能はなく、C言語プログラム上で、データとしてリストを扱うために必要十分な機能の提供を目指している。そこでは、リストは一般のデータと同じようにC言語上の変数や構造体のメンバに代入や参照ができるため、C言語として違和感のないプログラミングスタイルを維持できる。clpのリスト処理機能として以下があり、関数ライブラリとして提供されている。

- リストの参照、
- リスト構造の変更、
- リストの置換、
- 整列と併合、
- 集合的操作、
- 連想リスト、
- 算術演算、
- データ型の述語、
- 等値のための述語、
- ハッシュテーブル、
- C言語データIF、
- ファイルIF、

リストの連結を行なう関数のプログラム例を示す。

```
LIST   clpAppend(list1,list2)
       LIST list1,list2;
{
       if (clpConsp(list1) == T) {
             return(clpCons(clpCar(list1),
               clpAppend(
                    clpCdr(list1),list2)));
       } else if (clpNull(list1) == T) {
            return(list2);
       } else {
            return(NIL);
       }
}
```

ところで、リストは部分リストへの他からの参照が容易であるため、どのリスト要素が不用で再利用ができるのかをプログラムから判断することは困難である。従って、リスト処理において効率よく記憶領域を使用するためには、実行中におけるリストの自動記憶管理機構が不可欠である。本論文では、clpの記憶管理方式、特にくず集めの方法について述べ、clpを用いて開発した応用システムからその実用性を評価する。

2 データ構成について

clpのデータ型は、コンス、非数値アトム、数値アトムに大別される。非数値アトムには、シンボル、文字列、ハッシュテーブル、ファイルや対象表現を行なうためのアトム類がある。数値アトムは、整数と浮動小数点のみである。これは、プログラマは倍精度や長整数を用いる負荷の高い数値計算はclpによるリスト処理でなく、通常のC言語スタイルで記述すればよいからである。

セルの構造については、コンスセルは、データ型を示す情報を保持するタイプ部（1byte）,CAR部（4byte）、CDR部（4byte）からなる。非数値アトムを構成するアトムセルは、タイプ部（1byte）、データ部（4byte）、プロパティ部（4byte）からなる。

リスト処理では、データへのポインタからそのデータ型の高速な判別が必要である。clpにおけるポインタ値のデータ形式は、間接参照を避けるためポインタ値だけでデータ型判定が可能なポインタタグ方式とデータの記憶実体にデータ型情報を保持するオブジェクトタグ方式を併用する。ここでは、実行計算機の機械語長が32bitであることを前提とする。

ポイントタグ方式では、ポインタ値をタグ部（上

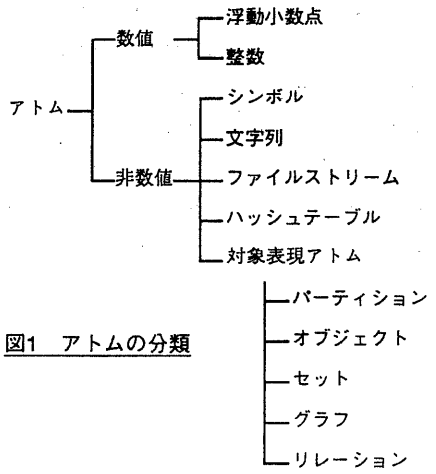

図1 アトムの分類

位2bit）と情報部（下位30bit）に分ける。タグ部のビットパタンは、

コンス　：　00
非数値アトム　：　01
整数アトム　：　10
浮動小数点アトム：　11

である。コンスや非数値アトムの情報部はセルへのアドレスであり、数値アトムの情報部は値そのものを表現している。

整数は、30bitの情報部で2の補数によって表現する。浮動小数点は、1bitの正負符号、8bitの指数部、21bitの仮数部から構成され、指数部は-127のバイアスにより-127から128を表現し、仮数は正規化によってその最上位数（必ず1）が省略されている。

このように数値は、2bitをタグに使うため精度や表現範囲が犠牲になっている。しかし、数値は繰り返し文の制御変数や算術演算の一時的な値等の寿命の短いデータになりがちで、このような即値形式にすることで記憶領域を消費せずくずが発生しない。さらに参照時にポイント先のデータをアクセスしないので高速であるという利点が得られる。

ポイントタグ方式は、論理的アドレス空間を均等分割した空間とデータ型との対応付けといえるが、使用可能な仮想記憶空間の大きさよりアドレス空間を小さく分割することはできない。従って、データ型の種類が多い場合、この方式だけで判定させることはできない。そこで、clpにある非数値アトムの9種のデータ型や4種あるコンスセルのタイプ（後述）の判定はオブジェクトタグ方式により行なう。参照回数が多いNILアトムやTアトムは定数で表現することで、その判定を高速化している。

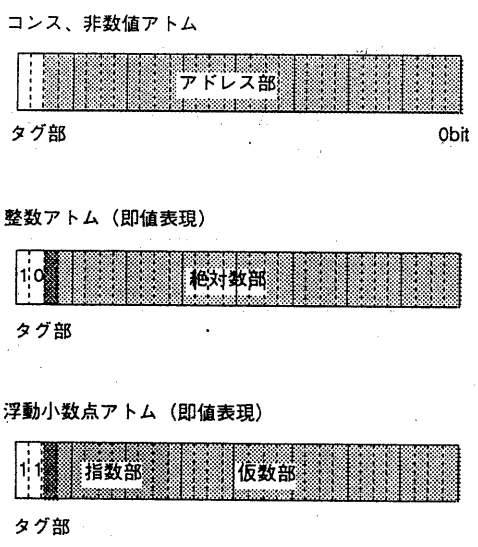

図2　ポインタ値の表現形式

3　くず集め方式

くずとは、プログラムの実行中に生じる不用になった記憶実体（オブジェクト）のことである。コンスセルを大量に消費するリスト処理では、プログラムの実行中くずを調べて回収し再利用するくず集め（以後GCとする）を自動的に行なう機能が不可欠である。

GCに関しては様々な方式が知られている。マークスィープ法[1]は、メモリ領域が溢れてコンスセルの割当ができなくなると、まず到達可能な全てのオブジェクトに印付け、その後印のないものを不用オブジェクトとして回収し再利用する。巡回リストの回収が可能であり記憶領域の利用効率もよいが、GCが行なわれている一定期間本来の処理の実行が中断する。参照計数法は、オブジェクトへの参照数を計数し、0になったオブジェクトを検知して回収する。この方式は、巡回リストの存在や参照数の溢れから完全にくずを回収できないが、マークスィープ法の実行頻度を減らすことができる。さらに、参照数の計数に若干の実行負荷を要する問題については、変数参照テーブルやゼロ参照テーブル等を用いて、特に参照が多いスタック上の変数からの参照を計数しなくてもよい方式[2]もある。コピー方式では、記憶領域を二つの領域に分け、実行中は一方の領域だけを使用する。GCでは、使用中の領域（A）に含まれる有用なオブジェクトだけを全て片方の領域（B）に移し変え、その後はBの領域を使用する。このGC毎に使用領域がAとBで入れ替わる。この方式は、領域の走査を一回行なえば圧縮を伴うGC処理が完了するが、実行中は常に半分のメモリ領域しか使えない。このコピーを一括して行なわず、CONSやCAR等の中で少しずつ行なうように分散すれば実時間GC[3]となる。次に、オブジェクトの生存時間とくずになる傾向との相関を考慮した効率の良い方式として、複数の小さな領域に分け、それぞれの新旧に応じた頻度でコピーを行なう方式[4]がある。また、過去、現在、未来の3つの領域に分け、変化の少ないオブジェクトを過去領域に移し、現在と未来の領域間でコピーを行なう方式[5]がある。これらのくず集め方式では、

・記憶領域の高い利用効率
・データの局所性の維持
・システム本来の実行を阻害しない

等の条件を満足すること求められる。

clpのくず集めの特徴は、くずとなりやすいコンスセルに対して、関数の実行終了毎に起動可能な圧縮によるGCを行なっていることである。関数単位という比較的粒度の小さい単位で実行頻度が高いGCを行なうことでその処理が分散され、システムの実行が中断することを防ぐ。本章では、clpにおけるGCの詳細な説明をする。

3.1 コンスセルのタイプ

clpには、記憶管理法が異なる4タイプのコンスセルがある（下表参照）。しかし、構造は同じで、リスト操作上では同一に扱える。

一時的コンスセルは、一定サイズを越えると使用済みのセルを強制的に再利用するため、ファイル入力時等の単なるバッファとして用いる。ユーザー関数で一般に使用されるのが動的コンスセルであり、これが最もくずになりやすいため、頻度の高いGCが行なわれる。準静的コンスセルには低い頻度でGCが行なわれ、シンボルの属性リストやハンシュテーブル内のリストはこのコンスセルで生成する。

このように、データの寿命の性質に応じてコンスセルのタイプを明示的に使い分けることで、くずの発生率に応じたGCを行ないその処理の負荷を軽減できる。

3.2 GCに必要な情報の管理

次に、GCを可能にするために必要な情報を管理する処置について述べる。

・リストの束縛実体の管理

印付けを伴うGCでは、プログラムの実行中に到達可能な、即ち有用なリストを知るため、リストを束縛するあらゆる実体を認識しなければならない。clpでは、リストの束縛実体のルートとして以下がある。

1. シンボルテーブル
2. 静的コンスセル
3. LIST型の変数
4. リストを束縛する構造体等のデータ実体

1および2はclpの中で管理している。しかし3および4はユーザプログラム中に現われるため、これらを登録する束縛実体テーブル（ハッシュテーブル）を用意する。

C言語プログラム上でリストを束縛するLIST型の変数は、記憶クラスとしてstaticとautoがある。外部変数や関数内部の静的変数は、コンパイル時に記憶割付が行われるstatic記憶クラスであり、プログラム実行中にその変数アドレスを束縛実体テーブルに登録する。関数の局所変数や仮引数はauto記憶クラスであり、clpのGCではこれらの束縛は考慮しなくてもよいため、テーブルに登録しない。構造体等からリストを束縛するデータ実体については、static変数と同様に登録が必要だが、そのデータ実体を記憶解放するときには登録を抹消する。

・未来参照コンスセルスタック

動的コンスセルの圧縮GC（後述）では、最も新しい部分領域を対象としてコンスセルの再配置を行な

表1 コンスセルセルのタイプ

	用途	回収方法	メモリ構成
一時的コンスセル	バッファ等の短命なデータ	強制的に再利用	連結した小領域
動的コンスセル	一般的な利用	関数の実行単位で行なう圧縮により再配置	単一領域
準静的コンスセル	内容変更が少ないKBやDB	自由コンスセルリストに連結	連結した小領域
静的コンスセル	全く変化しないデータ	行わない	連結した小領域

表2 GCのための情報管理機構

	登録内容	登録時期	削除時期
束縛実体テーブル	外部変数アドレス	変数が使用される以前	削除しない
	内部static変数アドレス	変数宣言した関数の実行開始時	削除しない
	データ実体のアドレス	記憶確保（malloc）直後	記憶解放（free）直前
未来参照セルスタック	新しいセルを参照するセル	RPLACA,RPLACDの使用時	関数の圧縮GC実行後
自由コンスセルスタック	自由動的コンスセルの位置	関数実行開始時	関数実行終了時

う。この時、この領域を参照するコンスセル（古い領域にあるコンスセル）があれば、参照先のコンスセルを保存して次にその参照番地を再配置後の番地に更新しなければならない。そこで、このようなコンスセル（未来参照コンスセル）が生じた場合に登録しておくスタックを持つ。

・自由コンスセルスタック

自由コンスセルスタックは、関数実行開始時の自由な動的コンスセルの位置をスタックに保持していて、スタックの動作はリスト関数（リスト処理を行なう）のネストと同期している。

3.3 圧縮によるGC

圧縮によるGCとは、セルの連続領域を有用セルだけが詰め込まれた状態にし、空いた領域のセルを再利用できるようにする処理である。

ところで、一般に古いコンスセルに比べて、新しく生成されたコンスセルはくずになりやすい。そこで、ある関数の実行中に新たに生成したばかりのコンスセルだけをGCの対象にすれば、回収率が高くGCの効率がよいと言える。clpでは、動的コンスセルに対してこのような関数の実行単位に基づくGCを行なう。即ち、ある関数の開始から終了までの実行期間に成長した動的コンスセル領域を対象に、その関数の終了時に圧縮によるGCを行なう。

動的コンスセル領域は、単一の領域で領域の下部から上部へとセルを使用していく。関数の実行開始時の自由セルの位置から終了時の位置までの範囲がその関数が生成したセルであり、その関数から呼び出した関数が生成したセルもその範囲に含まれる。

この圧縮GCで考慮すべき束縛実体のルートは

1）関数の戻り値
2）束縛実体テーブル
3）未来参照コンスセルスタック

の3つである。auto記憶クラスの変数からの束縛については、このGCの対象範囲と実行時期において無視できる。なぜなら、ある関数Fの実行が終了した時点では、Fの局所変数や仮引数は消滅するため、それらからの束縛は無効になるからである。また、Fを呼び出している関数の局所変数についても、C言語の局所変数は静的な束縛によるスコープであり、それからFで生成したコンスセルを直接に束縛することはない。

静的や準静的コンスセルは上の3つのルートに比べて束縛実体数が大きいため、この圧縮GCでは静的や準静的コンスセルから動的コンスセルへの束縛がないことを前提とする。そこで、RPLACA、RPLACDでは動的コンスセルから静的や準静的コンスセルへコピーされている。

動的コンスセルの圧縮によるGCは以下の3つの手続きからなる。

印付け処理

3つのルートの束縛実体からリストを辿り、圧縮GCの対象領域にある動的コンスセルに出会うと印付けを行なう。静的や準静的コンスセルに出会えばそれから先は辿らない。

再配置処理

圧縮の対象領域の上部から印付けられたコンスセルを走査するActiveConsPtと下部から印がないものを走査するFreeConsPtの2つのポインタを用意する。それぞれのポインタが目的のコンスセルを見つけると、ActiveConsPtが指すコンスセルの内容をFreeConsPtが指すコンスセルにコピー（再配置）する。さらに、ActiveConsPtが指すセルに移動したことを示すフラグと移動先番地を残して印を消す。そして2つのポインタは次のコンスセルを探す。ActiveConsPtとFreeConsPtが出会えばこの処理は終了し、出会った位置から上部の領域は自由コンスセ

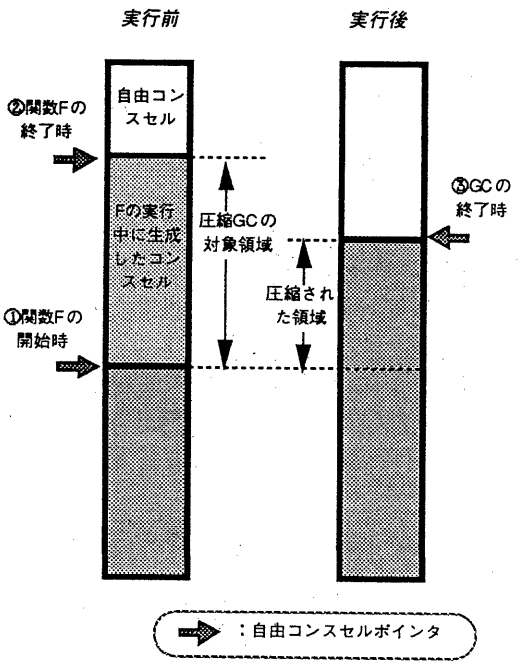

図3 動的コンスセル領域の圧縮の動作

ルとなっている。

更新処理

3つのルートの束縛実体の中で、再配置によって移動したコンスセルを参照している実体があれば、参照番地を移動先番地に更新する。

プログラムの実行中、関数のネストが浅くなるとGCの対象範囲が領域の底辺部に広がっていく。従ってこの圧縮GCをかけることで、外部変数や構造体等から束縛されているような寿命の長いコンスセルは領域の底に沈澱していく。

3.4 圧縮GCの実行制御

全てのリスト関数に圧縮GCを実行するとむやみにコンスセルが移動するだけである。そこで、次に述べる判定条件を満たした時のみ、実際に圧縮GCを行なうように制御する。

まず、関数のネストが一定数TH1以上深くなると自由コンスセルスタックの動作を停止しGCの機能も働かなくする。これはネストの深い関数は相対的にGCを必要とする程コンスセルの消費が多くないと考えられるからである。また、本来の処理に対してGCのための処理が及ぼす影響を軽減している。

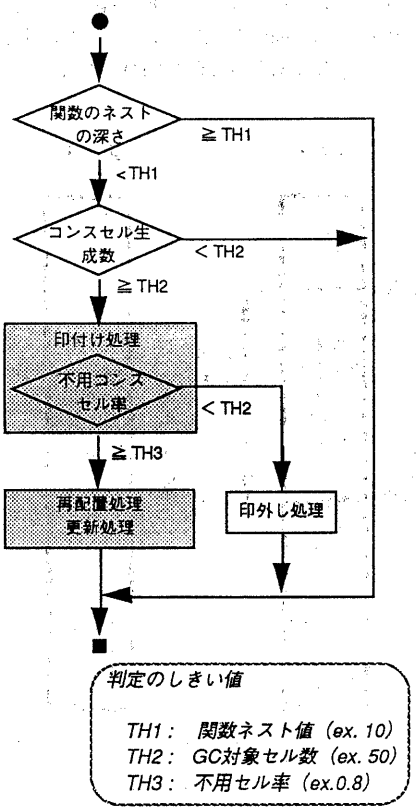

図4 圧縮GCの実行判定手順

次に、ある関数の実行中に一定個TH2以上のコンスセルが消費されかつそこに含まれる不用コンスセルの割合が一定値TH3以上であることである。この判定条件により、ある程度の粒度を持った関数に対して効率の良いGCだけを行なうことができる。さらに、ネストが浅くなれば（呼び出し側に戻ると）ネストの深い関数で生成したコンスセルが不用になることが多いというプログラムの挙動と対応している。不用コンスセル率は次の計算で得る。

$$\text{G-ratio} = \frac{\text{TotalCellNum} - \text{UsedCellNum}}{\text{TotalCellNum}}$$

TotalCellNum：関数内で生成したコンスセル総数
UsedCellNum：TotalCellNum中の有用コンスセル数
G-ratio：不用コンスセル率

有用コンスセル数は、印付け処理で印付けしたコンスセルの個数を計数することで得る。この時、有要コンスセル数をカウントアップと同時に不用コンスセル率に基づく判定を行ない、判定条件を満たさなくなった時点で印付け処理を中断しGCの実行を回避する。

TH2およTH3の値を小さく設定すると頻繁にGCがかかり、使用コンスセル領域の成長を抑制できるが、GCの効率が悪くそれに要する総時間が増す。逆に大きく設定するとGCの効率がよくなるが、一回当たりの処理時間が増す。

3.5 トップレベル関数でのGC

トップレベル関数とは、実行中に非リスト関数だけからネストされているリスト関数のことであり、この関数より浅い関数（リスト関数でない）ではリストを束縛するLIST型auto変数が存在していない。そこで、この関数の終了時では全てのauto記憶クラスのLIST型変数が無効になるため、準静的セルやアトムに対してもGCが可能となる。実行中にこの関数が現われると以下の3タイプのGCから選択し実行する。

GC-1：準静的コンスセル、アトムセルを対象にした不用セル回収とGC-2
GC-2：動的コンスセルの全領域の圧縮GC
GC-3：関数単位の動的コンスセルの圧縮GC

GC-2が必要な理由は、トップレベル関数において

関数単位の圧縮GC（GC-3）が回避されると、その時生成したコンスセルがそのまま残存してしまうからである。そこで、GC-3を回避した回数の一定頻度やコンスセル領域の使用率に基づき実行する。

GC-1では、不用セルを自由セルリストに連結して回収する。時間コストのかかるGC-1の実行頻度は他に比べ十分に低くしておくことが必要である。ここではGC-2の実行回数に基づく頻度でGC-1を実行する。

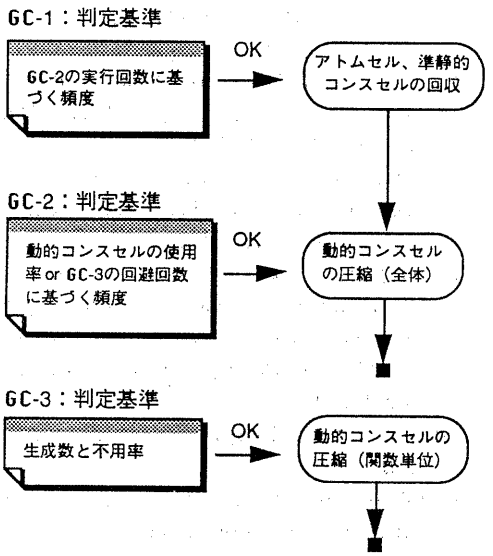

図5　トップレベル関数のGCの選択

4　プリプロセッサの設計

GCを機能させるためには関数に以下のような形式を与える。手続き的には、関数本体の実行前に自由コンスセルスタックを積み上げ、終了直前にGCの判定及びその実行の手続きを行ないスタックを戻す。

```
LIST   function (仮引数並び)
    仮引数宣言部
CLP_BEGIN   {
    内部変数宣言部

    実行部

    clpReturn（戻り値,type）;
     /* if LIST then type == 1,else 2 */
} CLP_END
```

ユーザプログラマがこの形式を意識しなくてもよいように、プログラムを自動的にこの形式に変換し、さらにLIST型静的変数の登録を行なうプリプロセッサを開発する。プリプロセッサの機能を示す。

1）全てのリスト関数（LIST型内部変数が出現する関数）に対し、関数本体ブロック文の前後にマクロを加える。そのブロック文中のreturnキーワードをcplReturnに置き換え、return値の型に基づきclpReturnに引数を与える。

2）LIST型内部静的変数や外部変数が出現する関数では、関数の内部変数宣言部の直後にそれらを束縛実体テーブルに登録する手続きを付加する。

但し、リストを束縛する構造体等のデータ実体の登録および削除についてはユーザプログラマの作業となっている。

5　実行評価

clpはSun-SPARC2上で利用可能であり、我々の研究室ではclpを用いて診断型エキスパートシステム（ES）の開発を完了し現在テスト運用を行なっている。このESで典型的な故障診断を実行した結果（下表）を示す。このESではUIにおける応答性が重要であるが、関数単位のGCは一回の処理時間が十分無視できる程度であり、ESの実時間応答を損なうこ

故障診断ESにおける実行結果

program size	：約40KLOC	判定しきい値
image size	：約5MB	TH1 : 10
実行内容	：典型的な6診断事例	TH2 : 50
所要時間	：約30分	TH3 : 0.7

表3　GCの実行状況

	実行回数	回避回数	平均実行時間	平均回収率
圧縮GC（関数単位）	904	603	0.032(sec.)	0.92
圧縮GC（全体）	13	–	0.217	0.70
全リストデータのGC	2	–	3.083	0.01

表4　セルの使用状況

	のべ使用数	最終時使用数	確保数
アトム	22770	22580	23000
動的コンス	424719	26877	80000
準静的コンス	187196	179382	180000
一時的コンス	190251	1310	2000
静的コンス	0	0	1000

とはなかった。

次に、ベンチマーク関数として知られるtakをSun-SPARC2上で実行した結果を示す。

clp : 300ms
real C : 83 ms
allegro-CL : 50ms
Austin-KCL : 717ms

clpはタグの付け外しやタイプ検査を行なうためreal Cに比べ4倍弱の処理時間を要している。また、LISPよりさほど良い結果ではないのはコンパイル時に再帰呼び出しを最適化していないためである。

clpによるtakのプログラム例

```
#define  LN1  0x80000001
LIST tak(x,  y,  z)
  LIST x,  y,  z;
{
   if  (clpGep(y,  x)  ==  T)  {
      return(z);
   }  else  {
      return  (  tak  (
            clpMinus(x,  LN1),  y,  z),
            clpMinus(y,  LN1),  z,  x),
            clpMinus(z,  LN1),  x,  y)));
```

6 最後に

今回実用システムの開発に十分利用可能なリスト処理ツールを目指して完成させることができた。特に、関数単位での圧縮GCでは、auto変数からの束縛を無視でき、さらに粒度を最適化することで実時間GCになる。最後に幾つか問題点について述べる。

まず、関数の実行終了時だけしかGCを行なう機会がないが、これでは関数の実行期間内に動的コンスセルが足らなくなる危険がある。clpでは、このような場合が生じると準静的コンスセルを代用することで対応している。

次に、束縛実体テーブルの走査に要する時間は圧縮GCの対象領域の大きさに依存せず一定値を占めるため、リスト束縛する構造体等が多量に存在するシステムでは印付け処理や更新処理がボトルネックになってくる。我々のシステム（ES）では約40000のリスト参照データ実体が存在し、圧縮GCの度にこれらの参照先を調べている。対応処置は、TH2の値を大きくし圧縮GCの実行回数を減らすことである。

clpではデータによってコンスセルのタイプを明示的に使い分けているが、動的コンスセルのGC生存回数に基づき準静的あるいは静的コンスセルに移動させることが考えられる。今後は、こうした点について改良を加えていきたい。

参考文献

[1] Allen J. ,Anatomy of Lisp, Mcgrow-Hill, New York, 1979

[2] L.P. Deutch and D.G. Bobrow, An Efficient Incremental Automatic Garbage Collector, Comm. of the ACM 19,9 (september 1976), 522-526

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

[4] H. Lieberman and C. Hewitt, A Real-Time Garbage Collector Based on the Lifetimes of Objects, Com m. of the ACM 26,6 419-429

[5] D.Uger, Generation Scavenging: A Non Disruptive High Performance Storage Reclaimation Algorithm, Proc. of Software Engineering Symposium on Practical Software Development Environments, ACM, April, 1989

[6] 稲田、Cでつくる Lisp処理系、archive, No1, CQ出版社

[7] Taiichi Yuasa, Design and Implementation of Kyoto Common Lisp , Journal of Information Procesing, Vol. 13, Number 3

[8] R. Gabriel, Lispシステムの評価・活用法、訳 和田、日刊工業新聞社

[9] 寺島、Lispのデータ配置に関する一技法について、情報処理学会記号処理研究会資料、SYM-58(1991)

[10] Guy L Steele Jr., Common LISP, 訳 井田、共立出版社