小型 Common Lisp 処理系のための記憶領域管理法とその実現†

山本 強††* 青木由直††

本論文では小型，高速の Common Lisp 処理系を実現するための記憶領域管理の方式を提案し，それにもとづいた処理系，HCL (Hokkaido Common Lisp) について報告する．本論文で提案する単一ヒープ2領域法は一個の連続するヒープ領域を下向きに成長する可変長オブジェクト領域と上向きに成長する固定長オブジェクト領域に分割して管理するものである．本方式は領域の細分化を行わないためページ型の管理を行う処理系に見られるページ残量がオブジェクトサイズに満たない場合に生ずる無効領域が発生しない方式の一つである．またガーベジコレクションに関してアプリケーションプログラムの動的な特性解析を行った結果，一般にコンス領域が大量消費，大量回収の傾向があることが明らかになり，その特性を考慮した新しいガーベジコレクションの制御法を提案し実装した．HCL においてそれを用いない単純な制御方式と比較した結果 GC 時間について15%程度の改善が可能であることが示された．

1. まえがき

Lisp 処理系の標準化案として Common Lisp の仕様[8]が定義されて以来，事実上の Lisp の標準として認知され，実際に KCL[10]を始めとしていくつかの処理系のインプリメントが報告されている．言語処理系のインプリメントは計算機工学のもっとも基本的な分野であり，様々な形式のインプリメントが行われることはその言語仕様がより充実したものになるために必須であると考えられる．Common Lisp は言語仕様を定めるがインプリメントに対しては自由度が与えられており，最適なインプリメントの形式を求める研究は継続的に行われている．本論文は Common Lisp 処理系を対象とした記憶領域管理法の一方式の提案とその実現について述べる．

本論文で取り扱う処理系，HCL[1] (Hokkaido Common Lisp) は現在流通している Common Lisp 処理系のいくつかの問題点に着目し，それらを改善してより快適な Common Lisp 環境を提供するために継続的に開発されている処理系である．HCL は一部の限定はあるものの CLtL[7] に定義されている620関数をすべて含むフルセットの Common Lisp である．

HCL の設計において以下の項目を重視した．

1. 実行時の記憶要求量を小さく抑える
2. コンパイルされた関数の実行速度は他の手続き型言語と同等
3. インタプリタの速度は従来の非 Common Lisp 系の処理系と同等

これらの目標は Common Lisp がアプリケーションレベルで実用的な処理系として認知されるためには必要な条件と考えられる．これらの実現可能性を検討すると，現在一般に使われている Common Lisp のインプリメント技法のいくつかが基本的な問題となる．必要記憶空間についていえば二つの問題点がある．一つは Common Lisp の仕様が巨大であるためコード自体が大きくなる点と，現時点でもっとも広く用いられている標準の Copy 型 GC (Garbage Collection)[9] を用いる限りにおいてはオブジェクト空間が2セット必要であるため記憶空間の使用効率が良くない点である．現在のワークステーションでは大容量の主記憶が実装され仮想記憶もサポートされるため記憶要求量にこだわる必要はないように思われるが，マルチプロセス環境では巨大なプロセスの存在は他のプロセスのスワッピングの頻度を上げるため好ましくない．コンパイルされたコードの実行速度を左右する要因としては仮想マシンの形式，オブジェクトの内部表現などがあり，設計の初期に十分検討しておく必要がある．Common Lisp の場合には変数に関する宣言が可能なため，これをコンパイラが利用することによってCなどの手続き型言語と同等の速度が得られる可能性がある．しかし，KCL に見られるように仮想マシンをC処理系に設定した場合にはその速度の上限は明らかにCそのものでありそれを越えることはない．実際には最適にCでコーディングした場合よりもかなり低下すると考えられる．したがって仮想マシンモデルの選択は速度を重視する場合，言語構造と目的マシンのアーキテクチュアを十分考慮した上でなされるべきである．インタプリタの速度は現在の Common Lisp 処理

† A Memory Management Scheme for Compact Common Lisp System and Its Implementation by TSUYOSHI YAMAMOTO and YOSHINAO AOKI (Department of Information Engineering, Faculty of Engineering, Hokkaido University).
†† 北海道大学工学部情報工学科
* 現在 北海道大学大型計算機センター

系がもっとも注意を払っていない点であるといえる．Common Lisp は仕様上，従来よく用いられた shallow binding[2] のようなインタプリタ高速化技法が使いにくいため一部の処理系を除いて原始的な連想リストによる変数束縛を行っており一般に低速である．この背景には，実際のコードの実行はコンパイルされたコードが行うため速いインタプリタの要求は少ない，あるいはデバック自体もコンパイルされたコードに対して行うほうが良いという認識がある．しかし，コンパイルされた関数の変数環境などをデバッガから見えるような構造を実現するためには関数のコード自体に工夫をする必要があり，それがコンパイルされたコードの実行速度を低下させる原因にもなりかねない．HCL はこれらの問題を念頭において設計・開発された．

HCL のインプリメント上の特徴は記憶領域管理法と GC アルゴリズムにある．開発段階において大規模プログラムの動作解析を行った結果，従来知られている GC アルゴリズムに改善の余地があることを示し，コンパクト性と GC の高速性を重視した，単一ヒープ2領域記憶管理法と2モードスライディング GC アルゴリズムを開発した結果，コンパクトかつ実行速度の速い処理系が実現できた．

2. Lisp の記憶領域管理の方式

Lisp はその特徴として記憶領域の管理機構を処理系が備えている点を上げることができる．そのため Lisp 処理系を対象とした記憶領域管理およびガーベジコレクタに関する報告は多数見られる．記憶領域管理について言えば基本的には単一のヒープを用いる方法，複数のヒープを用いる方法，ページ単位で管理する方法などが提案されている．Lisp 用の記憶管理法はその言語仕様によって影響を受ける．初期の Lisp 1.5[8] では単一長さのセルを取り扱えれば良かったが，近代 Lisp 処理系では言語仕様が可変長の線形ベクタを要求するためその管理は可変長セルの管理を要求される．可変長セルの取扱いを前提とした場合，GC はコンパクションを行う必要があり，そのアルゴリズムもかなり限定されたものとなる．現在，Common Lisp 処理系の代表的な記憶管理法はページ型である．これは汎用計算機上の処理系の場合にあらかじめ割り当ててある記憶領域を完全に消費しつくした場合でも追加割当が容易に行えることが最大の理由であると考えられる．反面，ページ型では可変長セルのデータに対して無効領域ができる確率が高く決して効率の良い方法とは言えないことも事実である．GC に関して見れば汎用機上の処理系ではコピー方式が多く採用されている．これはコピー方式がコンパクションを自然に行えることが主な理由であると考えられる．しかしコピー方式は実際に割り当てられている領域の半分を使うだけであり，残りは GC のために予約されるという問題があり，小型の処理系を実現しようとする場合大きな問題となる．大きな記憶空間を取り扱う場合には全体の GC 時間よりもそのための処理の中断時間が問題となるため GC のプロセスを細分化し長時間の処理中断を避ける分割回収型のアルゴリズム[3] を用いる処理系もある．しかし小型の処理系を考えるならば GC による中断は高々数秒であり，分割回収のためのオーバヘッドを考慮するとその利点は少ない．

我々は小型の Common Lisp 処理系に最適な記憶管理法と GC アルゴリズムとして本論文で単一ヒープ2領域法と2モードスライディング GC アルゴリズムを提案する．

2.1 単一ヒープ2領域記憶管理法

HCL では Common Lisp の全オブジェクトをヒープと呼ぶ線形空間に置く．ヒープはあらかじめ全領域が静的に割り付けられているとする．ヒープに格納されるオブジェクトは処理系によって決められた記憶領域を占有する固定長オブジェクト (Cons, Symbol など) と動的に領域が決定される可変長オブジェクトに分類される．Lisp の特性上，固定長オブジェクトの中でも Cons は特に大量に消費され使い捨てられる傾向にある．そのため Cons オブジェクトの高速な生成と回収は Lisp 処理系の性能を決定する重要なパラメータとなっている．HCL が採用した単一ヒープ2領域法では**図1**に示すようにヒープ領域を下向きに成長するベクタ領域と上向きに成長するコンス領域に分割して考える．これらの領域は Cons free pointer, Vector free pointer の2本のポインタで管理されそれらが衝突するまで成長できるものとする．衝突した時点で全領域が消費されたと判断し GC が起動され，各領域はそれぞれコンパクションされ，連続した自由空間が作られる．この方式の利点として，ページ式の記憶管理に見られる無効領域が発生しないこと，ベクタおよびコンス領域の割合をあらかじめ宣言する必要がないことが挙げられる．また Cons の生成コストはもっとも単純な自由リスト型の管理と同程度であり，ベクタ領域の割当も同様に簡単に行える．

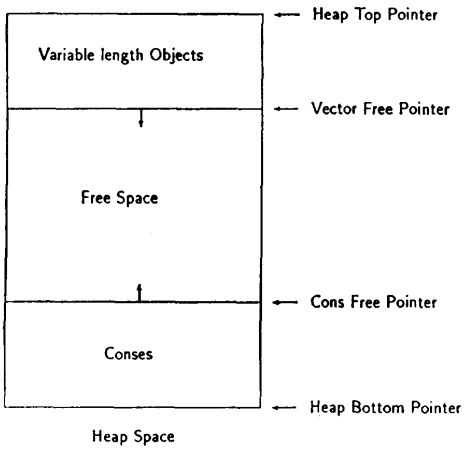


図 1 単一ヒープ2領域記憶管理法
Fig. 1 Single heap two regions memory management scheme.

Lisp プログラム実行において Cons はもっとも基本的なデータ型でありその全記憶領域中に占める割合は大きい．そのため Cons のメモリ上での占有領域は処理系の小型化にとって大きな意味を持つ．HCL では Cons を固定長オブジェクトとして専用の領域に置くため GC のための特別なフィールドは予約されておらず8バイトで表現されている．

2.2 2モードスライディング GC アルゴリズム

単一ヒープ2領域記憶管理法を採用した場合，GC はコンス，ベクタ領域とともコンパクションを行う必要がある．コピー法はこの問題を簡単に解決できるがヒープの領域を2倍要求するため HCL の目的にそぐわない．コピー法以外のコンパクションを行う GC はコンパクションフェーズのコストがかなり大きく，なるべく低コストのコンパクション GC アルゴリズムを開発することの意義は大きい．そのため HCL の開発過程においていくつかの大規模問題を用いて HCL の実行中のベクタ領域とコンス領域の消費と回収の動特性を計測した．**表1** は数式処理システム REDUCE 3.3[6] および標準ベンチマーク問題[5]を実行させてそれが使用した Cons とそれ以外のオブジェクトの消費量を計測した結果である．この例に限らず通常の Lisp アプリケーションプログラムでは Cons の大量消費，大量回収の傾向がみられ，すべての GC 要求に対して全ヒープ領域を回収，コンパクションを強要することは効率が良いとは言えない．この特性を考慮しより高速な GC 制御アルゴリズムとして2モードスライディング GC を考案した．この制御アルゴリズムはコンス領域のみを対象とするCモード GC (Cons only mode GC)，全領域を対象とするFモード GC (Full GC) の2方式の GC アルゴリズムを用意し，これを条件によって切り換えることにより全体の GC 時間を短縮しようとするものである．Cモード GC は実際にはFモード GC のサブセットであり，Fモード GC のいくつかのフェーズを選択的に実行するものでありベクタ領域のスライディングが不用であること，またベクタ領域のマーキングフェーズにおいてポインタの逆転による連鎖化が不要となることから処理が簡単になる．そのため可能な限りCモード GC を行い必要に応じてFモード GC を行うような制御が行えれば GC のために費やされる CPU 時間は減少させることができる．

表 1 アプリケーションレベルでのオブジェクト消費傾向
Table 1 Object space consumption in application programs.

プログラム	Cons 消費量 (M_1†)	Cons 以外 (M_2†)	M_1/M_2
REDUCE 3.3 (積分例題集)	38,027	4,249	8.9
REDUCE 3.3 (因数分解例題集)	25,365	3,585	7.1
Boyer††	1,881	0	—
Browse††	3,911	41	95.4

† 消費量の単位は KByte
†† Gabriel Benchmark[6]

2.3 Cモード GC アルゴリズム

Cモード GC は全領域をマーキングするが回収およびコンパクションはコンス領域のみを対象に行う高速 GC モードである．Cモードの基本アルゴリズムはいわゆるマークスイープ法であり，回収段階で置き換え法によるコンパクションを行う．Cモード GC は3パスで実行を完了する．第一パスはマークフェーズで GC 要求の時点で変数，定数，データスタックから参照される可能性のあるオブジェクトを再帰的に探索しマークを付加する．第二パスではコンス領域の下向きのコンパクト化を行う．コンス領域のオブジェクトは固定長であるから置き換え法[4]により高速なコンパクト化が可能でありその結果回収されたコンス領域が線形空間として再使用できるようになる．第三フェーズでコンパクト化によって移動したセルを指すポインタの書換えとマークのリセットを行い，GC が完了する．Cモード GC はコンス領域のみを回収の対象とするため高速である反面，ベクタ領域がガーベジを多量に含んでいる場合には不十分である．

2.4 Fモード GC アルゴリズム

Fモード GC はCモード GC が回収しないベクタ領域も回収の対象とする完全な GC である．Fモード GC は4パスからなる．第一パスはCモードと同様にマークフェーズであるがFモードではベクタ領域のセルに対してはスライディングコンパクションに備えてポインタの逆転処理を行う．その結果ベクタ領域のオブジェクトに対する複数のポインタによる参照は連鎖化 (Chaining) される．第二パスはCモード GC と同様にコンス領域のコンパクションを行う．第三パスはベクタ領域のスライディングによるコンパクションを行う．第四パスはCモードの第三パスに相当するものでコンス領域のコンパクションによって場所が変わったセルを指すポインタを書き換える．Fモード GC をCモード GC と比較するとマークフェーズのオーバヘッドとベクタ領域のスライディングコンパクションのコストがかかる点が処理時間の増加分となる．HCL にインプリメントしてFモードとCモードの実行時間を比較した結果を**表2**に示す．平均してCモードはFモードの 60% の時間で終了することが確認された．

2.5 GC モード切り替えアルゴリズム

表 1, 2 から容易に推察されるように可能な限り Cモード GC を行い，必要に応じてFモード GC を行うことによって全体の GC に要する時間を短縮できる可能性がある．この切り替えはプログラムの実行を中断することなく自動的に行われるべきである．その判断基準を与えるために実行中のいくつかのパラメータを用いて全 CPU タイムに占める GC 時間を記述する．ここでは以下のパラメータを用いる．

T_{GCF} 一回のFモード GC に要する時間
T_{GCC} 一回のCモード GC に要する時間
D_V 単位時間当りのベクタ領域消費量
D_C 単位時間当りのコンス領域消費量

表 2 GC モードによる回収時間の比較
Table 2 Timing comparison between F mode and C mode GC schemes.

	Fモード GC	Cモード GC
HCL 起動直後 C領域 150 KB V領域 184 KB	1.233 sec	0.727 sec
REDUCE 3.3 起動後 C領域 336 KB V領域 420 KB	2.667 sec	1.417 sec

注：ヒープ領域を 1,840 KB 使用，使用計算機は SUN 3/60

S ヒープ領域の大きさ

ここで問題を簡単にするため消費されたすべてのセルは GC によって回収されるものと仮定する．またSはシステムが使用する全ヒープ領域ではなくそれから固定的に使用される領域を除いた大きさを意味する．この仮定はプログラムがある程度進行して変数や配列の初期化が済んだ状態を想定している．あるプログラムが実行を完了するためには GC 時間を含まない CPU 時間 T_1 が必要であるとする．この時，Fモード GC のみを用いて実行した場合の平均 GC 起動回数 N_{GC}, GC 時間 T_{GC}, 全 CPU タイム T_{CPU} はそれぞれ式 1, 2, 3 で与えられる．

$$N_{GC}=\frac{(D_V+D_C)T_1}{S} \tag{1}$$

$$T_{GC}=N_{GC}T_{GCF} \tag{2}$$

$$T_{CPU}=T_{GC}+T_1 \tag{3}$$

ここで GC の効率を表現する量として GC 時間率, T_{GC}/T_{CPU} を考え，それを最小とすることを考える．Fモード GC とCモード GC を切り替える判断基準としてその時点で消費されているヒープ領域に占めるベクタ領域の割合，K を導入する．K は0～1の間の実数であり，GC 要求があった時点で K があらかじめ決められた値より大きければFモード GC が起動される．そうでなければCモード GC を起動するものとする．この制御を行う場合にFモード GC が最初に実行される実効 CPU 時刻を T_2 とする式4で示される．

$$T_2=\frac{SK}{D_V} \tag{4}$$

最初のFモード GC が起動されるまでに何回かのCモード GC が起動されることとなるが $n-1$ 回目のCモード GC から n 回目のCモード GC までの CPU 時間を $T_{CI}(n)$ とするとそれは式5で書ける．

$$T_{CI}(n)=\frac{S-D_V\sum_{i=1}^{n-1}T_{CI}(i)}{D_V+D_C}$$

$$=\left(1-\frac{D_V}{D_V+D_C}\right)T_{CI}(n-1) \tag{5}$$

ここで $T_{CI}(1)=\dfrac{S}{D_V+D_C}$ である．

n 回目のCモード GC が起動された時点の実効 CPU 時間，$T(n)$ は式6となる．

$$T(n)=\sum_{i=1}^{n}T_{CI}(i)$$

$$=\frac{S}{(D_V+D_C)}\left(1-\frac{D_C}{(D_V+D_C)}\right)^{n-1} \qquad (6)$$

式 4,6 から一回のFモード GC が起動されるまでに起動されるCモード GC の平均回数 N_C を求めると式7となる.

$$N_C=\frac{\log(1-K)}{\log\frac{D_C}{D_V+D_C}} \qquad (7)$$

したがって最初のFモードGCが起動され，終了するまでに費やされた全 GC 時間を $T3$ とするとそれは式8で記述される.

$$T3=T_{GCF}+N_CT_{GCC} \qquad (8)$$

GC 時間と実効 CPU 時間との比を最大にする K が最善であることは明らかであるから $T3/T2$ を最小とするように K を求めれば良い.

$$\frac{T3}{T2}=\left(T_{GCF}+\frac{\log(1-K)}{\log\frac{D_C}{D_V+D_C}}T_{GCC}\right)\frac{D_V}{SK} \qquad (9)$$

式 9 から明らかなように $T3/T2$ は $0<K<1$ で唯一の最小値を持つ．2の REDUCE 3.3 起動後の状態におけるGC速度を用いて最適なKを9から数値計算により求めると D_C/D_V が 5〜50 の範囲で変化する場合に K は 0.52〜0.23 となる．D_V, D_C は実行するプログラムにより変化するがこの K の傾向を参考にして固定した値を用いるのが現実的である．HCL では標準値として 0.33 を設定している．また，プログラム実行中に Cons のみを消費するループにおちいることがありその場合CモードGCのみが多数回起動されることになる．その場合，CモードGCのみが短い時間間隔で反復されることになり，GC 時間率が低下する．そういった状況を避けるため，一定回数以上CモードGCが連続した場合にFモードGCを強制し可変長オブジェクト領域をコンパクト化する．これによりCモードGCの起動間隔を大きくしGC時間率を小さくできる．プログラムがベクタ空間のみを消費するサイクルに陥ることも考えられるがその場合でも毎回Fモード GC が起動されることになるだけでありGC 時間が本制御方式を用いない場合よりも増加することはない.

3. オブジェクト内部表現

GC のアルゴリズムとオブジェクトの内部表現は密接に関連している．HCL では小型化を達成するためにアドレス空間を限定しデータ型情報はなるべくポインタ側に持たせる方針で内部表現を設計している．HCL ではオブジェクトを 32 bit で表現するものとし最大ヒープ領域を 16 MB に限定してポインタ部を 24 bit，タグ部を 8 bit 使用する．タグは MSB 側に置かれるものとしタグ部の LSB は GC がマークビットとして使用するため常に 0 であるものとする.

HCL におけるオブジェクト内部表現は図 2 に示す3形式に分類される．24 bit のポインタ部を即値形式で使用する型としては Nil, Fixnum, Character, Invisible Pointer がある．他の2形式ではポインタ部は基本的にヒープ領域を指すこととなる．可変長オブジェクト領域に格納されるオブジェクトの実体にはその大きさと必要に応じて詳細な構造情報を格納するためのタグ領域 (SDO-TAG) が割り当てられている．このフィールドはFモードGCにおいて複数参照のチェーン化のためのポインタ格納にも用いられる．多用されるコンス領域のオブジェクトには付加的なタグフィールドは用いていない.

HCL でのタグパターンは頻繁に使用されるオブジェクトの型判定が高速に行えるように配慮している．例えば Cons のタグは 00_{16} であるが 02_{16}〜$7E_{16}$ は故意に使用しない．その結果，オブジェクトを 32 bit 整数と見たときにそれが0より大きければそのまま Cons へのポインタと見なして良いことが保証される．その結果，基本アクセス関数である CAR, CDR は型検査を含めても3機械命令で実行される．00_{16} を持つ例外的なオブジェクトとして Nil がある．Nil は

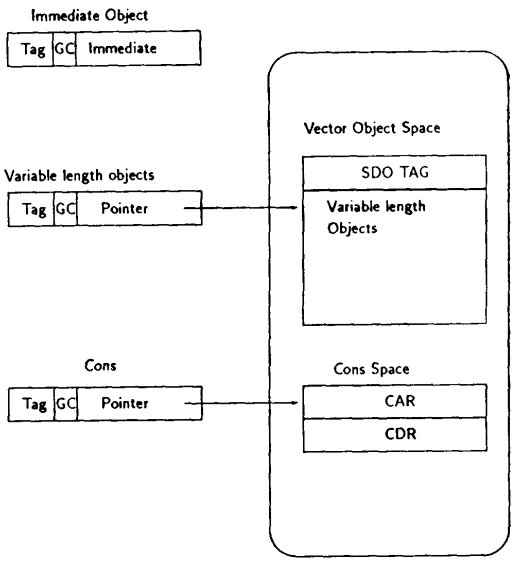


図 2 HCL オブジェクト内部表現
Fig. 2 Internal representation of objects in HCL.

空 List としての意味と Symbol としての意味がある特別なオブジェクトであるが HCL では特別なパターン 00000000_{16} として表現されている．Nil は Symbol としての機能が要求されるため内部では隠れたオブジェクトとして Nil の実体に相当する Symbol の構造体が存在する．ある関数が Symbol としての Nil の構造体メンバのアクセスを要求する場合は特例として取り扱われることになるがその頻度は極めて少なく，その多くは例外的な処理であるため速度に与える影響は少ない．Lisp における条件判定は Nil か否かで行われるのが普通であり，多くの処理系はそれが高速に行えるような配慮をしている．HCL では Nil は数値としての 0 であるから汎用計算機上で高速に判定できる．Nil と Cons 以外のすべてのオブジェクトは MSB が 1 であるようなタグを与えられている．これにより頻繁に行われる型判定述語は**表 3** のように単純な数値比較に置き換えることができた．Symbolp も多用される述語である．Symbol に対するタグは 80_{16}〜BE_{16} をすべて割り当てている．このパターンは 8 bit 長で 2 倍した時にオーバフローを生じるパターンであり，汎用計算機の整数加算命令とオーバフローフラグによって高速に判定することが可能である．Nil は 2 倍によってオーバフローを生じないため特別に扱う必要があるが 0 の判定であるからそのオーバヘッドは小さい．整数オブジェクトもプログラム中で多用され，特に 0 付近の小さな整数の高速な取扱いは処理系の速度を左右する．HCL では 24 bit で表現される整数を Fixnum として即値形式で表現している．24 bit の数値部は 2 の補数ではなくオフセットとして 2^{23} を加えた正数として表現されているため，タグを含めた 32 bit の大小判定により数値比較が可能となっている．

4. 処理系の開発および評価

HCL は本論文で提案した記憶管理手法を 680X0 CPU をターゲットとしてインプリメントした処理系である．HCL は Common Lisp の仕様を満たすことを前提としており，核は C および 680X0 CPU を意識した仮想アッセンブラで記述されている．**表 4** はモデルケースおよび大規模なアプリケーションによって計測した本手法による GC 時間短縮を示している．表 4 の測定では $K=0.33$ とし連続する 10 回の C モード GC の後には F モード GC を強制される制御を行っている．改善率は例題によって変動するが大規模なアプリケーションプログラムの実例として REDUCE 3.3 の因数分解テストプログラムを用いた計測の結果 15% 以上の改善が確認された．

処理系の小型化という点についてどの程度達成されたかを調べるために HCL 単体および代表的なアプリケーションプログラムを実行するために最低限要求される記憶空間の大きさを**表 5** に示す．表 5 の要求量は起動時に既に消費されている記憶量を表しており，実行のためにはアプリケーションに応じた記憶領域をあらかじめ用意する必要がある．

本論文で提案した記憶管理方式は単に効率が良いというだけではなく，動的な記憶管理のモニタリングの面でも新たな可能性を持っている．つまり，全領域が一本のヒープであり，数個のパラメータによって管理

表 3 基本型判定述語の数値的取扱い
Table 3 Numerical manipulation of basic type predicates.

述語	数値的取扱い
atom	*if* $x \leq 0$ *then t else nil*
consp	*if* $x > 0$ *then t else nil*
listp	*if* $x \geq 0$ *then t else nil*
null	*if* $x = 0$ *then t else nil*

表 4 GC 時間の改善
Table 4 Improvement of GC timings.

例題	F モードのみ		2 モード法 ($K=0.33$)			改善率 (%) $\frac{T_{GC}-T_{GC'}}{T_{GC'}}\times 100$
	N_{GCF}	T_{GC}	N_{GCC}	N_{GCF}	$T_{GC'}$	
例題 1†	22	39.883	21	6	36.900	7.5
例題 2††	21	35.867	21	2	28.050	21.8
REDUCE 3.3 (因数分解)	27	92.288	22	9	77.600	15.9

† Cons 消費量が Vector 消費量の 10 倍のモデルケース
†† Cons 消費量が Vector 消費量の 50 倍のモデルケース

表 5 HCL のアプリケーションにおける最小メモリ要求量
Table 5 Minimum memory requirement for HCL applications.

項目	値
最小記憶領域要求量 (1) (コンパイラをオートロード化)	0.963 MB
最小記憶領域要求量 (2) (全関数ロード状態)	1.339 MB
Portable Common Loops	1.984 MB
CLX	2.283 MB
REDUCE 3.3	3.160 MB

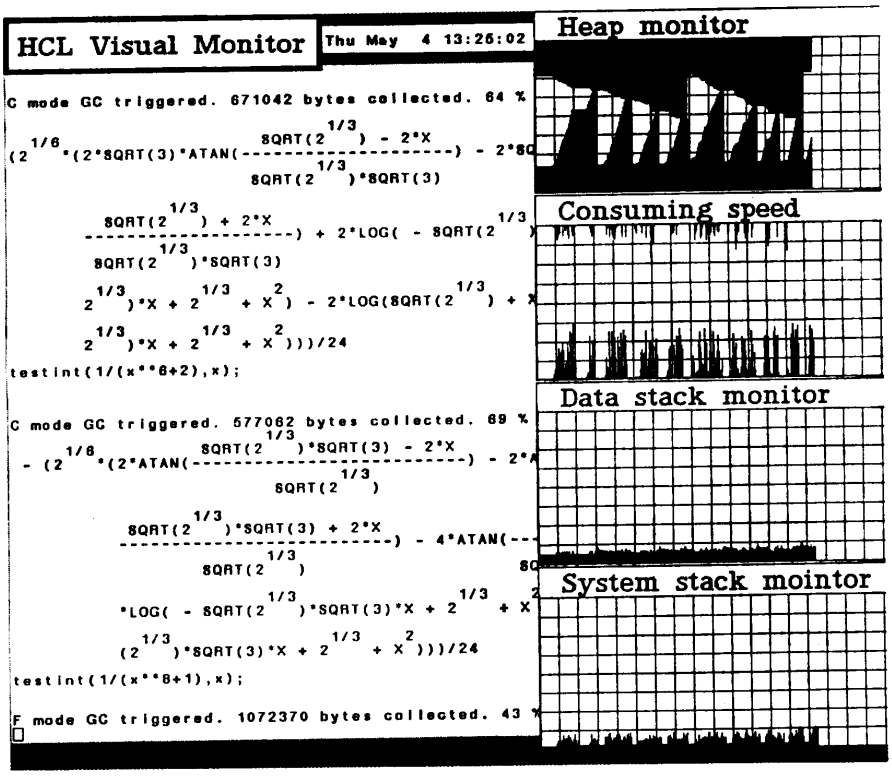


図 3 HCL に組み込まれた視覚的動作モニタ
Fig. 3 Built-in visual monitor of HCL.

されているため実行中のプログラムの振舞いを少ないオーバヘッドによって視覚的表示することが可能である．HCL には X Window 上のインタフェースとしてプログラム実行中のヒープ領域の消費速度，累積消費量，スタック使用量をリアルタイムで表示するメカニズムが組み込まれており，必要に応じて起動することができるようになっている．**図 3** はアプリケーションプログラム実行中にモニタを起動した例である．これにより従来は把握しにくかった Lisp プログラムの振舞いが視覚的に把握できるようになった．

5. むすび

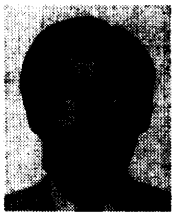
Common Lisp の仕様に関してはその規模，複雑さ，厳密性などに関して議論の分かれるところである．しかし事実上の標準として認知されていることも事実であり様々な形式のインプリメントによって Common Lisp の仕様が持つ問題点や追加すべき仕様などが明らかになるものと思われる．本論文で報告した記憶管理法に関する設計方針は Lisp 処理系の小型化に関してひとつの指針を与えるものであり，得られた処理系は従来の非 Common Lisp 系の処理系と比べて実行速度の点でも優るものであり小型のワークステーションでも実用的な規模を実現できた．

参考文献

1) 山本 強，青木由直：Hcl の開発と視覚的モニタ方式の提案，情報処理学会研究報告，88-SYM-48 (1988).
2) Allen, J.: *Anatomy of LISP*, McGraw-Hill, New York (1978).
3) Baker, H.: List Processing in Real Time on a Serial Computer, *CACM*, Vol. 21, No. 4, pp. 280-294 (1978).
4) Cohen, J.: Garbage Collection of Linked Data Structures, *ACM Comput. Surv.*, pp. 341-367 (1981).
5) Gabriel, R. P.: *Performance and Evaluation of LISP Systems*, MIT Press, Cambridge, MA (1985).
6) Hearn, A. C.: *REDUCE-3 User's Manual*, The Rand Corporation, Santa Monica, CA (1983).
7) Steel, G. L., Jr.: *Common Lisp the Language*, Digital Press, Burlington, MA (1984).
8) McCarthy, J.: *LISP 1.5 Programmer's Manual*, MIT Press, Cambridege, MA (1969).
9) Fenchel, R. and Yochelson, J.: A Lisp Garbage-Collector for Virtual-Memory Computer System, *CACM*, Vol. 14, No. 8, pp. 611-612

(1969).
10) Yuasa, T. and Hagiya, M.: Kyoto Common Lisp Report, Kyoto University (1985).

(平成元年5月9日受付)
(平成2年4月17日採録)



山本　強（正会員）

昭和28年生．昭和51年北海道大学工学部電子工学科卒業．昭和53年同大学院修士課程修了．同年富士通(株)入社．昭和55年同退職．同年北海道大学大学院博士後期課程入学(電気工学専攻)．昭和57年同中退．同年同大学講師，昭和61年同大学助教授．昭和62年同大学情報工学科助教授．平成元年より同大学大型計算機センター助教授．記号処理，画像工学，コンピュータグラフィックスの研究に従事．著書『THE 3-DIMENSIONAL-COMPUTER GRAPHICS』(CQ出版，昭58)．電子情報通信学会，IEEE，ACM各会員．工学博士．

青木　由直（正会員）

昭和16年生．昭和39年北海道大学電子工学科卒業．昭和41年同大学院修士課程修了．同年同大学講師，昭和42年同大学助教授，昭和54年同大学電気工学科教授，昭和62年同大学情報工学科教授．昭和58年より中国瀋陽工業大学客員教授．(財)札幌エレクトロニクスセンター運営委員会委員長．不可視波動情報処理，信号・画像処理，コンピュータグラフィックスの研究に従事．『マイクロコンピュータ講義』(共著，昭晃堂，昭58)，『BASIC 数値計算法』(コロナ社，昭59)，『波動信号処理』(森北出版，昭61)．電子情報通信学会，IEEE，応用物理学会，OSA，日本音響学会等各会員．工学博士．