記 号 処 理 48－1
（1988. 11. 29）

HCLの開発と視覚的モニタ方式の提案

山本 強・青木 由直
北海道大学工学部

本報告では小型・高速のCommon Lisp 処理系の実現例として我々が開発したHCL(Hokkaido Common Lisp) について述べる。HCL は単一のHeapを可変長オブジェクトと固定長オブジェクトの２空間に分割しオーバーヘッドの少ない記憶管理を実現している。またGarbage Collectionについても置き換え法、スライディング法によるIn-Place型のアルゴリズムを採用した結果、最小で1MB 程度の記憶領域から動作するCL処理系が実現できた。HCL は記憶管理の構造が単純であるため外部からその振る舞いが容易に把握できる特長があり、これを利用して視覚的な動作モニタ機構を組み込んだ結果、プログラムのダイナミックな動作が認識できる事が明らかになった。HCL を汎用のワークステーション上で動作させた結果、既存の処理系よりも小型かつ高速であることが確認できた。

IMPLEMENTATION OF HCL AND ITS VISUAL MONITORING MECHANISM

Tsuyoshi YAMAMOTO and Yoshinao AOKI

Deparment of Information Engineering, Hokkaido University,
N-13,W-8,Kita-ku,Sapporo,060 Japan

The implementation of HCL(Hokkaido Common Lisp) that is an attempt to realize small and fast Common Lisp interpreter and compiler is reported. HCL features single-heap two-regions memory management mechanism in which one heap space is divided into two sub-spaces or fixed and variable length object spaces. The garbage collection scheme of HCL is not a copying GC scheme that requires dual heaps but an in-place scheme based on replacing and sliding. By these schemes, minimum requirement of memory to run HCL is less than 1MB. Simple memory management scheme of HCL gives us simple mechanism to monitor the dynamic behavior of program execution. HCL has visual monitoring mechanism to give the user dynamic information of program execution. HCL has been implemented on general purpose UNIX workstations and tested. The results shows that its size and timing are smaller and faster than currently available Common Lisp system.

1. はじめに

Lisp処理系の標準化案としてCommon Lispの仕様[1]が定義されて以来、事実上のLisp処理系の標準規格として認知され、実際にKCL[2]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は字句的な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。本報告ではその一つとして我々が実現したUNIX-WS を対象としたCL処理系、HCL(Hokkaido Common Lisp) について述べる。HCL は現在流通しているCL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CL環境を提供するために継続的に開発されているCL処理系である。HCL は一部の限定はあるもののCLtLに定義されている820 関数を総て含むフルセットのCLである。

HCL の開発目的はより実用的かつ教育的なCL処理系の実現である。当初の設計目標は以下の通りである。

1. 最小の必要記憶空間は1MB 程度
2. コンパイルされたコードの実行速度は最良の場合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CL系のLispと同等

これらの目標はCL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。これらを実現可能性を検討すると、現在一般に使われているCL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2つの問題点がある。一つはCLの仕様が巨大であるためコード自体が十分大きい点と、現時点でもっとも広く用いられているCopy型GCを用いる限りにおいてはHeap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。1MB という大きさはかなり小さいものであり、現在のWSの規模から言えばそれにこだわる必要は無いが、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。CLの場合には変数に関する宣言が可能なため、それをコンパイラが利用する事によって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有り得ない。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。従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CL処理系がもっとも注意を払っていない点であると言える。CLは仕様上、過去に開発されたshallow binding 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、あるいはデバッグ自体もコンパイルされたコードに対して行う方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 はこれらの問題をクリアする事を目的としている。

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が明らかになり、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モードスライデイングGCアルゴリズムを開発した。また、この解析の過程で処理系の動作状態のモニタ機構の重要性が認識され、内部状態変数の視覚的なモニタ機構を組み込んだ結果、プログラムのふるまいが容易に認識できるようになった。

2. HCL の記憶領域管理方式

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事ができる。そのためLisp処理系を対象とした記憶領域管理及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ある。Lisp用の記憶管理法はその言語仕様によっても影響を受ける。初期のLisp1.5 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、CL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事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出来る確率が高く、決して効率の良い方法とは言えない。また大きなヒープ領域を予め宣言し、その中に前オブジェクトを格納する方式を採用している処理系もあるがこの方式では無効領域は出来ない反面、全Heap領域を消費しつくすと実行が停止してしまう欠点がある。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事が主な理由である。しかしコピー方式はプログラムの動作中に有効に使われるのは全領域の半分だけであり、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。HCL では小型のCL処理系に最適な記憶管理法とGCアルゴリズムとして単一ヒープ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し実装した。

2.1　単一ヒープ２領域記憶管理法

HCL ではCLの全オブジェクトをHeapと呼ぶ線形空間に置く。Heapは予め全領域が静的に割り付けられているとする。そこに格納されるオブジェクトは２タイプに分類できる。一つは固定長のConsであり、残りはそれ以外の可変長アトミックオブジェクトである。Lispの特性上、Consは大量に消費されまた使い捨てられる傾向にある。そのため、Consの高速な生成と回収速度はLisp処理系の性能を決定する重要なパラメータである。HCL が採用した単一ヒープ２領域法は図１に示す様にHeap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Cons領域に分割して考える。これらの領域は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、Vector及びConsの領域の割合を予め宣言する必要が無いことを上げられる。また、Consの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、Vector領域の割当も同様に簡単に行える。

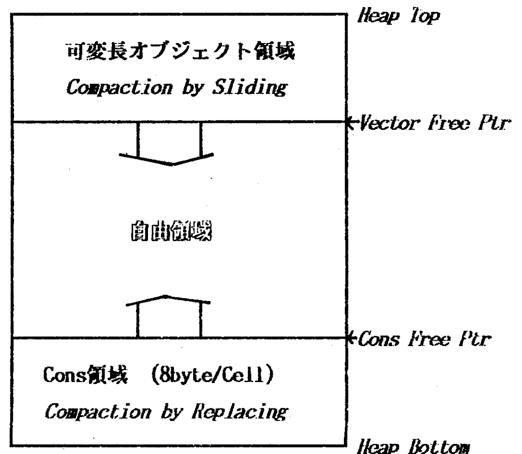

図１単一ヒープ２領域記憶管理

2.2　２モードスライディングGCアルズリズム

単一ヒープ２領域記憶管理法を採用した場合、GCはCons、Vector領域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Heap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。オブジェクト空間の動作特性を調べるためにHCL の開発過程においていくつかの大規模問題を用いて実行中のVector領域とCons領域の消費と回収の動特性を計測した。図２は数式処理システムREDUCE3.3 のベンチマーク問題（数式積分）の実行過程を1 秒毎に計測しそれをグラフ化したものである。この例に限らず通常のLispアプリケーションプログラムにはConsの大量消費、大量回収の傾向がみられ、全てのGC要求に対して全Heap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。この点を考慮しHCL では２モードスライディングGC

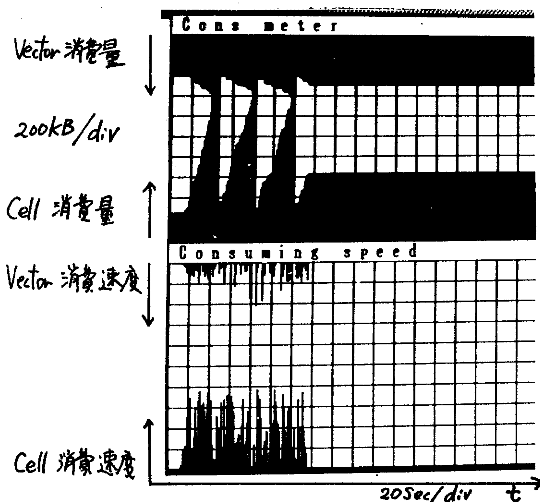

図２　Vector/Cons 領域の消費速度比較

としてCons領域のみを対象とするＣモードGC(Cons mode GC)、全領域を対象とするＦモードGC(Full GC) の２モードをもつGCアルゴリズムを開発した。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るが、Vector領域のスライディングが不用であること、Vector領域のマーキングフェーズがスライディングが不用なために処理が簡単になり、かなり高速に実行でき、全体のGCに要する時間を削減できる。

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及びコンパクションはCons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おこなう。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総てにマークを付加する。第二パスではCons領域をknuth による置き換えアルゴリズムにより下向きに圧縮する。第三フェーズでは圧縮で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Cons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Vector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Vector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。第二パスはＣモードGCと同様にCons領域のコンパクションを行う。第三パスにはVector領域のスライディングによるコンパクションである。第四パスはＣモードの第三パスに相当するものでCons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ーヘッドとVector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モードの実行時間の比較した結果を表１に示す。平均してＣモードが約２倍高速である事が確認された。

表１ Ｆモード/ＣモードGC時間 (OMRON 9100SX)

	Fモード GC(Sec)	Cモード GC(Sec)
HCL 起動直後 C 93KB,V 166KB	1.217	0.700
REDCE3.3起動後 C 229KB,V 360KB	2.283	1.083

2.5 GCモード切り替えアルゴリズム

図２、表１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を行う事によって全体のGCに要する時間を短縮できる可能性がある。この切り替えは自動的に行われるべきである。その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に要した時間を記述する。ここでは以下のパラメータを用いる。

Tcpu	CPU 時間
Tgc	GC時間
Trun	実効CPU 時間(Tcpu-Tgc)

Dvect	単位時間当りのベクタ領域消費量
Dcons	単位時間当りのコンス領域消費量
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。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Trun=T1 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、Ngc、GC時間Tgc 、全CPU タイムTcpu、はそれぞれ(1)-(3)のように記述される。

$$Ngc = (Dvect+Dcons)*T1/S \quad (1)$$

$$Tgc = Ngc*Tgcf \quad (2)$$

$$Tcpu = Tgc+T1 \quad (3)$$

となる。ここでGCの効率を表現する量としてGC時間率、Tgc/Tcpuを考え、それを最小とする事を考える。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 が予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。そうで無ければ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T2とすると式(4) で示される。

$$T2 = S*K / Dvect \quad (4)$$

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が起動される事となるが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Tci(n)とするとそれは式(5) で書ける。

$$Tci(n) = \frac{S - Dvect * \sum_{i=1}^{n-1} Tci(i)}{Dvect + Dcons} \quad (5)$$

従って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。

$$T(n) = \sum_{i=1}^{n} Tci(i) \quad (6)$$

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Ncを求めると(7) となる。

$$Nc = \frac{\log\left(1 - \frac{DVect + Dcons}{2 \cdot DVect + Dcons} \cdot K\right)}{\log\left(\frac{DVect + Dcons}{2 \cdot DVect + DCons}\right)} \quad (7)$$

従ってこの場合のGC時間率をG とすると式(8) で書かれる。

$$Trun = \frac{S \cdot (DVect+Dcons)}{DVect \cdot (2DVect+Dcons)} \left(1-\left(\frac{DVect+Dcons}{2DVect+Dcons}\right)^{Nc}\right)$$

$$Tgc = Tf + Nc \cdot Tgcc$$

$$G = Tgc/(Trun+Tgc) \quad (8)$$

G を最小とするK は、式(8) をK について微分してそれを0 とする事により求められる。

つまり、Vector領域とCons領域の消費速度があらかじめ分かっていればそれから定数K を設定し、GC要求があった時点で直前のGCからのCons,Vector 領域の消費量の比率をもとめ、これがK より大きいか小さいかによってGCモードを決定する事ができる。K=0 では常にFモードGCが起動される事になる。

3. オブジェクト内部表現

HCL は全てのオブジェクト内部表現は図3の様にTag(8bit)+Pointer(24bit)の32bit で表現する。Tag の最下位bit はGC使用するので常に0 となっている。オブジェクトの基本形式はCons型、可変長オブジェクト型、即値型の三種がある。Tag の割り当ては使用頻度が高いものほど有利なように設定している。例えば、Consに対するTag は0 であり、それ以外のアトミックオブジェクトのTag はMSB が1 であるように割り当てている。その結果、頻繁に行なわれるConsとAtomの判定はオブジェクトを数値と見た時の正負判定で行なえる。特殊なオブジェクトとしてNil があるがNil は全bit が0 であるパターンを予約している。従ってNil をふくむAtomの判定はObj <= 0に、Nil を含むListの判定はObj >= 0となり、汎用マシン上で高速に行なえる。Symbolp も頻繁に行なわれる判定であるが、Symbolに対するTag は80X を与えており、2 倍してオーバーフローを検出する事によって判定される。ConsのTag が0 であるためもっとも頻繁に行なわれるCar,Cdr へのアクセスは基本的に1 命令で済む。Tag を8bitに設定したため短整数は24bit であり、それ以上は全てbignumとなる。Tag はポインタ側に配置されているのでConsセルは8 バイトで表現されている。しかし可変長オブジェクト型のオブジェクトはスライデ

5. 評価

HCL は680X0 MPU を使用したUnixワークステーションで動作する。カーネルの記述はC 及びアセンブラであり、関数の一部はHCL 自身で記述されている。
HCL の目的である小型化については動作に必要な最低限のメモリサイズはコンパイラを非常駐とした場合に1MBを実現できた。実用的には1.5MB 程度の占有領域から使用できる。実際に2MB 主記憶のSystem V上でコンパイラを含む動作が確認できている。
CLの仕様的な完備性を確認するのは容易ではないため、パブリックドメインとして入手できる大規模な応用プログラムを仕様確認のためにインプリメントしている。現在までに応用プログラムとしてCL版OPS5, CL版REDUCE3.3, XEROX版PCL の動作が確認できている。速度的な評価として代表的なGabriel ベンチマークの結果を表２に示す。表２にはパブリックドメインに近い他の処理系との速度比較をあわせて記してある。HCL,KCL に関しては全く同一のソースコードであるがFranz,PSL はCLでは無いために専用にコーディングされたプログラムが実行されている。HCL コンパイラはいくつかの最適化レベルを有しており変数の型宣言と組合せる事により、整数を取り扱う関数においては最高でC コンパイラと同程度の実行速度を得る事ができる事が確認された。表３はTARAI 関数を例に最適化レベルをかえてコンパイルした場合にどの程度の改善が得られるかを示したものである。

表２　Gabriel ベンチマーク(Sun3/260)

Problem	HCL	KCL	Franz	PSL
Boyer	8.7	24.6	59.0	7.0
Browse	11.2	****	105.0	n/a
Fft	38.0	53.5	n/a	33.0
Destruct	1.3	4.7	n/a	1.1
Puzzle	23.2	35.4	159.0	3.4
Tak	1.0	1.8	3.0	0.0
Traverse	32.0	53.2	174.0	59.3
Triangle	198.0	****	354.0	57.3

Courtesy of Mr.Cristopher Burdorf,The Rand Corp.
注: 計測は全て同一マシンによるシングルユーザ環境で
測定

表３　HCL コンパイラの最適化レベル

Level	内容	時間(約)
ｲﾝﾀﾌﾟﾘﾀ		96.4
(safety 3)	トレース可能コード	14.5
(safety 2)	Stack Overflowをチェック	6.5
(safety 1)	引き数個数チエックまで	3.8
(safety 0)	引き数型チェックのみ	3.3
fixnum宣言	型チェックは省略される	2.0
16bit 宣言	最高速の整数計算	1.8

注：(TARAI 10 5 0)にて計測。機種はOMRON 9100-SX

6. むすび

小型のCL処理系であるHCL の実現とその上に組み込まれた視覚的動作モニタ機構について報告した。HCL はフルセットのCLであるにも関わらず1MB 程度の実行イメージから動作し、スライディング型のGCを行なうため他の処理系と比べて記憶領域の要求量が少ない事が確認された。また、処理速度についても他のインプリメントと比べて遜色のないものであることが確認できた。

現在組み込まれている視覚的モニタ機構は内部の主要状態変数をのぞく単純なものであるがより高度な制御構造の可視化やオブジェクト指向プログラミングでのメッセージの流れといったプログラムのセマンテイクスを直観的にユーザに与える事は教育的にも重要であり、その表示手法、インプリメント法について現在検討中である。

参考文献

1)G.L.Steel Jr.:"Common Lisp the Language", Digital Press, 1980
2)T.Yuasa,M.Hagiya:"Kyoto Common Lisp Report",1985 Kyoto University
3)湯浦他:"高速Common Lisp-HiLispの実現" 情報処理学会第33回全国大会2E-1
4)小林他:"CP/M68K 用STANDARD LISP の開発",情報処理学会記号処理研究会資料SYM29-9

イングのためにオブジェクト自信にも大きさとセルの内容をそ一示すTag が付加される。ポインタ部が24bit であるためHCL のオブジェクト空間は実質的に16MBに制限される。コンパイルされた関数の空間はこの中に入らない。

HCL が実行中に作成する環境オブジェクトはインタプリタ、コンパイラともリストではなく線形ベクタで内部表現されている。インタプリタの内部環境は環境スタックで表現されており、クロージャが生成された場合にのみコピーをベクタとしてクロージャ内に保存する。[3] コンパイルされたクロージャの場合も同様であるが、コンパイル時にクロージャ内から参照される可能性のある変数、関数、ブロックのみが選択的に環境ベクタにコピーされる。

31　　24	23　　　　　　　　　　　　　　0
Tag 8bit	Pointer or Immediate (24 bit)

図3　HCL オブジェクト表現

4．視覚的動作モニタ機構

最近のウインドウシステムではCPU の稼働状況をモニタするパフォーマンスメータ的な表示を行ない、プロセスの動作を視覚的に認識できるようになっている。このような表示はプロセスの実行中に突発的に発生する異常現象を発見したり、プログラムをチューニングする場合の補助的な情報を与えてくれる。しかしLispのプログラミング環境、実行環境はC 等と異なりLispの中から全ての操作が行なわれる事が多い。またOS側から見た場合にLispは一個のプロセスに見えるがそれは一般に巨大であり、CPU 資源のかなりの割合を消費するが、外部から見えるのは一個のプロセスとしてLisp全体が計量の単位となる。パフォーマンスメータのアナロジーからLisp内部の状態変数を視覚的に表示することによってLispプログラムの振る舞いや異常動作を認識しやすくなる可能性がある。HCL はその単純な記憶管理機構のために動作中に内部状態をオーバーヘッド少なくモニタできる。我々は実験的にウインドウシステムを用いて重要と思われる内部パラメータのリアルタイム表示機構をインプリメントした。

Lispの構造上、記憶領域の消費速度、残量、データ種別毎の消費量などはプログラムの動的な特性を解析するうえで有用なパラメータであるが、近年の複雑な記憶管理を行なう処理系では実行中のオーバーヘッドを伴わずにモニタする事は困難である。HCL の記憶管理は先に説明したように２本のポインタで全オブジェクト空間が管理されているため、このポインタを一定時間間隔でモニタすれば実行中のプログラムに一切影響を与えずプログラムの実行過程が表示できる。HCL に組み込まれた視覚的モニタはVector,Cons 領域の消費状況モニタの他に単位時間当たりの消費量(消費速度) 、プログラムスタックの深さ、データスタックの深さをモニタする。しかし、スタックについてはサンプリング間隔が粗いために表示された結果がスタックの最大成長点とは限らず、あくまでも参考程度の情報である。視覚モニタはスタート時のオプションとして起動される。表示はプログラムされた一定間隔 (System Vでは1 秒) に起動される割り込み処理により更新される。モニタを起動しなかった場合にはLispプログラムの実行速度の低下はない。図４はモニタを起動した場合の表示画面の例である。表示機構はOMRON 9100SXのGMW 上に作成されている。

図４　視覚的動作モニタを起動させた画面
(REDUCE3.3の因数分解を実行中)