計算機アーキテクチャ 77－8
（1989. 7. 14）

HCL上のPortable Common Loopsのインプリメントと高速化技法

山本 強　青木由直
北海道大学工学部

Common Lispのオブジェクト指向プログラミング機能としてCLOSの仕様が固まりつつある。Common Lisp処理型は今後CLOSの機能を要求される事になるが、Common Lispの仕様はそれ自体がCLOSの存在を仮定して書かれたとは言えず、CLOSインプリメントの段階で不自然な記述を行う必要があるようである。我々は以前にワークステーション環境での小型・高速のCommon Lisp処理系としてHCL(Hokkaido Common Lisp)を開発したが、今回それに対するCLOSのサポートとしてPCL(Portable Common Loops)を移植した。HCLでのPCLインプリメントでは、HCLの特徴である小型・高速性を損なう事なくCLOSの高度な機能を実現するためにPCLの総称関数の実現母体であるfuncallable instanceをHCLの組み込みオブジェクトとして実現している。PCLを組み込むことによるHCLの記憶要求量の増加は約650KBであり、きわめて小さいCLOS処理系が実現できた。

Porting of Portable Common Loops to HCL and Its Speed Up Technique

Tsuyoshi Yamamoto and Yoshinao Aoki

Faculty of Engineering, Hokkaido University

N-13,W-8,Kita-ku,Sapporo 060, Japan

Specification of CLOS (Common Lisp Object System), that is an object-oriented extension to Common Lisp, has been fixed up recently. From now on, each Common Lisp system will be requested to support CLOS features. However, original specification of Common Lisp was designed without concerning CLOS specific features, so that some tricky programming techniques must be used to implement CLOS on them. HCL (Hokkaido Common Lisp) is an implementation of Common Lisp designed by us. The goal of HCL is to realize a small and fast Common Lisp environment on workstations. In this report, we report about porting of PCL(Portable Common Loops) to HCL as CLOS support for it. In this implementation, we extend internal data representation of HCL to realize funcallable instance object that is fundamental mechanism to realize generic function of CLOS. As the result, HCL with PCL increase its size only 650K bytes.

1 まえがき

Lisp 処理系の標準化案として Common Lisp の仕様 [6] 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[8] 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。

最近、ANSI によって Common Lisp のオブジェクト指向プログラミングのスタイルが CLOS(Common Lisp Object System) が規格化され公開された。そのため、今後は Common Lisp の処理系は CLOS の仕様を含めて考える必要が生じてきている。CLOS は必然的に従来の Common Lisp の仕様の枠内で記述されるべき物であるが Common Lisp の仕様決定の段階で CLOS の構造が意識されていたとは言いがたく、代表的なインプリメントである PCL[4](Portable Common Loops) を見ても Common Lisp の枠を越えたオブジェクト構造やアクセス関数を用いているのが実際である。また Common Lisp 自体が巨大であり、小型の処理系が少なく、かりに PCL を既存の処理系にインプリメントしても小型のワークステーションでは快適な使用環境が得られないのが現状である。

また、これまでにいくつかの報告で CLOS のインプリメントは実際に速度を犠牲にする事なく行うことが可能であるとの報告 [1, 5] があるが具体的に実際に速度比較を行った報告が見られないため、インプリメントの水準がどのレベルにあるのか多少曖昧でもある。

本報告では小型・高速の CLOS を実現するための問題点を PCL を例として検討し、それに基づき我々が開発した Common Lisp 処理系である HCL[2] (Hokkaido Common Lisp) に移植した結果について具体的な数値を含めて述べる。

2 HCL

本論文で取り扱う処理系、HCL[2](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 Common Lisp 処理系である。

HCL は一部の限定はあるものの CLtL[6] に定義されている 620 関数を総て含むフルセットの Common Lisp である。

HCL の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 Common Lisp 系の Lisp 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。これらを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 2 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十分大きい点と、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)[7] 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。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。Common Lisp 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 Common Lisp 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。Common Lisp は仕様上、過去によく用いられた shallow binding[3] 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ている。

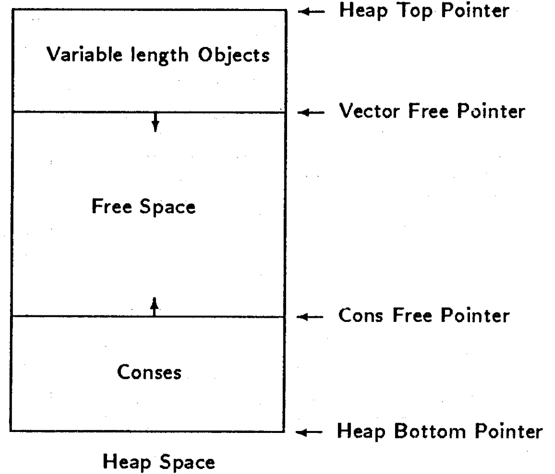

図 1: 単一ヒープ 2 領域記憶管理法

3 HCL の内部構造

まず PCL をインプリメントするために必要な HCL の内部構造について説明をする。

3.1 HCL の記憶管理モデル

HCL は全てのオブジェクトを単一のヒープ空間に格納する。HCL では記憶空間の使用効率を上げるためにインプレース型の GC を行っている。HCL のメモリモデルを図 1にしめす。

3.2 オブジェクト内部表現

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。

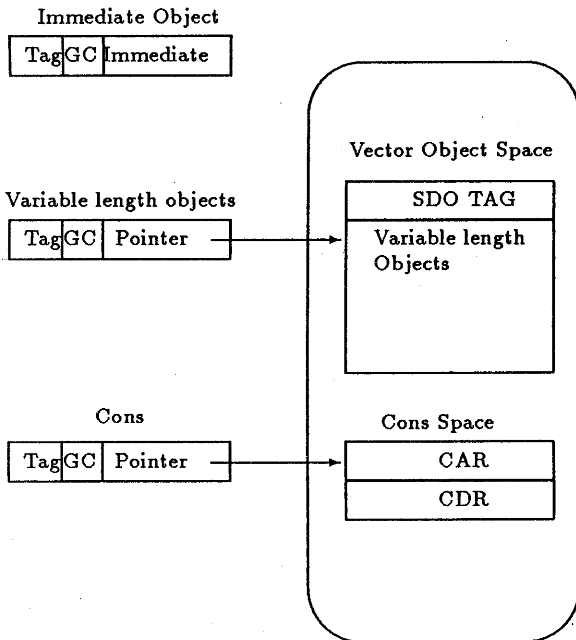

図 2: HCL オブジェクト内部表現

HCL ではオブジェクトを 32bit で表現するものとし最大ヒープ領域を 16MB に限定してポインタ部を 24bit、タグ部を 8bit 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。24bit 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事となる。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 でのタグパターンの分布は一様ではない。これは使用頻度の高いオブジェクトの確認が高速に行えるようにする配慮である。Cons のタグは $00_{16} \sim 3E_{16}$の全てが割り当てられている。これによりオブジェクトを 32bit 整数と見たときにそれが 0 より大であればそのまま Cons へのポインタと見なして良いことを意味する。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ようになる。Nil は空 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事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 か否かで行われるためその判別が高速に行えることは速度の改善に効果がある。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 以外の全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。この結果、もっとも頻繁に行われる型判定述語は表 1のように単純な数値比較に置き換えることができる。Symbolp も多用される述語であるが Symbol に対するタグは $80_{16} \sim BE_{16}$を割当ている。このパターンは 8bit 長で 2 倍した時にオーバーフローを生じるパターンであるため汎用計算機の整数加算命令とオーバーフローフラグによって高速に判定される。Nil は 2 倍によってオーバー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ーヘッドは小さい。整数オブジェクトは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 24bit で表現され

表 1: 基本型判定述語の数値的取扱い

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*

る整数を Fixnum として即値形式で表現する。24bit 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 32bit の大小判定により数値比較が可能となっている。

4 PCL の移植

PCL[4](Portable Common Loops) は Xerox PARC において開発され公開されている CLOS のインプリメンテーションである。PCL は Common Lisp で記述されているため完全な Common Lisp 処理系であればそのままインプリメント可能であるが、CLOS の基本的な概念である generic function(総称関数) を Common Lisp の枠内で実現すると速度が低下すると言われている。そのため各 Common Lisp のインプリメントにおいて generic function の内部表現形式である funcallable instance を関数オブジェクトの内部表現の拡張として実現する。Funcallable instance とは関数であると同時に standard class の instance であるというオブジェクトである。Standard class に属するオブジェクトは wrapper-slot,static-slot,dynamic-slot の 3 スロットを持たねばならない。PCL における高速化手法として重要なものにスロットアクセス及びメソッド探索におけるキャッシュ技法がある。特にメソッド探索のキャッシュ化はオブジェクト指向言語処理系で普通に用いられる方法であり、高速のキャッシュアクセスは実行速度をかなり改善できる。HCL における PCL のインプリメントでは funcallable instance に注目し、その実現に HCL のオブジェクト構造を一部拡張した。

4.1 Funcallable Instance の実現

PCL を移植する場合にまず解決しなければいけない問題は関数でありかつ外部からアクセスできる 3 個のスロットを持つオブジェクトをどの様に内部表現するかである。一般的な解決は compiled closure を用いてその閉じこめられた変数の領域 (普通は vector あるいは list) を拡張するものである。KCL 等ではこの方法を用いている。しかしこの方法は Common Lisp の仕様の範囲内では実現するのが困難である。つまり、compiled closure は Common Lisp の標準オブジェクトであるがその内部に閉じこめられた変数を外部からアクセスする方法を Common Lisp は与えていない。そのため、各処理系によって異なる closure の構造に対してアクセス関数を作製する事になる。閉じこめられた変数領域の拡張は closure が元々閉じこめていた変数に対して影響を与えてはいけないため、あらかじめ十分大きな領域をアロケートするという方法で行われる。この方針は処理系の小型化という観点から見ると問題がある。一つは、"十分大きな領域"がどのくらいかという点である。PCL の機種依存コードを見ると 15 程度の領域を仮定して、それで不足を生じた場合には間にダミーの関数 (trampoline function) を介在させる。この場合、funcallable instance の大きさは通常の closure 関数に長さ 15 の配列を加えた大きさになり、一般に関数オブジェクトが長さ 2-3 の配

列程度である事を考えると5倍くらい大きくなると考えられる。また、KCLのように閉じこめられた変数をリストの形式で保持している場合にはスロットをアクセスするコストがその長さに比例するという問題もある。もう一つの問題はclosureの呼び出しコストである。HCLではcompiled functionとしての内部形式としてsimple functionとcompiled closureの二通りあり、使用頻度の高いsimple functionの呼び出しはきわめて高速に行えるような配慮がなされているため、なるべく不要なclosureを発生しないようなfuncallable instanceの内部表現が望まれる。

HCLではPCLの移植にあたり、記憶効率、関数呼び出しコストの改善のために第三の関数形式としてfuncallable instanceを標準オブジェクトとして組み込む事にした。HCLでの関数オブジェクトの内部表現一覧を図3に示す。funcallable instanceはそれがfuncallable instanceである事を識別できるタグと追加された3スロットからなる必要最小限の大きさである。また、関数本体がclosureでなければ作られるfuncallable instanceもclosureとはならないので高速の関数呼び出し形式が使われる。

5 処理系の実現及び評価

用いたPCLのソースコードは1988年3月に公開されたPost ISO versionである。PCLソースの変更はlow.lisp,fin.lisp,defsys.lispの処理系依存コードに対して変更を加える事によって行い、それ以外の変更はあえて行っていない。

処理系の実行速度および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簡単なベンチマークを行った。目的はmethod lookupのコスト及びinstance variableのアクセスコストの測定である。使用したベンチマーク問題は、Lispで一般的なTak関数のCLOS版とユーザー定義クラスを用いた物である。図4,5にその定義を示す。各ベンチマーク例題は従来手法との速度比較を示すためにgeneric functionによらないコーディングも同時に記述してある。測定結果を表2に示す。

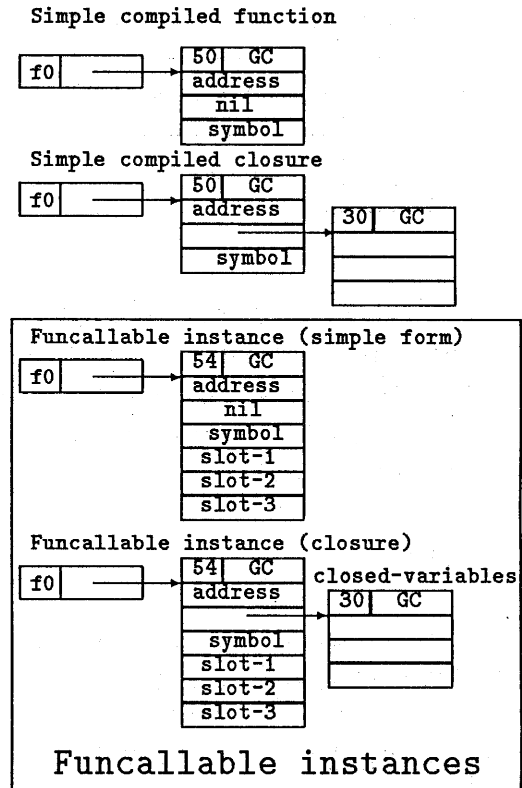

図3: HCLにおける拡張関数オブジェクト

現在のインプリメントでは組み込みオブジェクト型によるmethod look upでは20倍以上の速度差が認められるが、defclassによって定義されたクラスによるmethod look upではその差は2倍程度である事が明かとなった。組み込みオブジェクト型に対するtypecaseマクロはコンパイルの時点で型判定がインライン展開されるために特に差が広がったものと思われる。この程度のオーバーヘッドでgeneric functionが

```
;;
;;      "TAK" as generic function
;;
(defmethod tak-gf ((x integer) y z)
   (if (> x y)
       (tak-gf (tak-gf (1- x) y z)
               (tak-gf (1- y) z x)
               (tak-gf (1- z) x y))
       y))

(defmethod tak-gf ((x list) y z)
  (if (> (length x) (length y))
      (tak-gf (tak-gf (cdr x) y z)
              (tak-gf (cdr y) z x)
              (tak-gf (cdr z) x y))
      y))

(defmethod tak-gf ((x float) y z)
  (if (> x y)
      (tak-gf (tak-gf (- x 1.0) y z)
              (tak-gf (- y 1.0) x y)
              (tak-gf (- z 1.0) x y))
      y))
;;
;;      Typecase version
;;
(defun tak-typecase (x y z)
  (typecase x
        (integer
   (if (> x y)
       (tak-typecase (tak-typecase (1- x) y z)
                     (tak-typecase (1- y) z x)
                     (tak-typecase (1- z) x y))
       y))
        (float
  (if (> x y)
      (tak-typecase (tak-typecase (- x 1.0) y z)
                    (tak-typecase (- y 1.0) x y)
                    (tak-typecase (- z 1.0) x y))
      y))
        (list
  (if (> (length x) (length y))
      (tak-typecase (tak-typecase (cdr x) y z)
                    (tak-typecase (cdr y) z x)
                    (tak-typecase (cdr z) x y))
      y))))
```

図 4: Generic function 版 TAK ベンチマーク

```
;;
;;      Generic function benchmark II
;;
(defclass c1 () ((dummy :initform nil)))
(defclass c2 () ((dummy :initform nil)))

(defun test-generic-function (x)
  (let ((c1 (make-instance 'c1)))
      (time (dotimes (i x)
             (test-gf-internal c1)))))

(defmethod test-gf-internal ((c1 c1))(slot-value c1 'dummy))
(defmethod test-gf-internal ((c2 c2))(slot-value c2 'dummy))
;;
;;      Typecase version
;;
(defstruct s1 (dummy nil))
(defstruct s2 (dummy nil))

(defun test-typecase (x)
  (let ((s1 (make-s1)))
    (time (dotimes (i x)
           (test-typecase-internal s1)))))

(defun test-typecase-internal (x)
  (typecase x
    (s1 (s1-dummy x))
    (s2 (s2-dummy x))
    (otherwise (error "Don't know what to do with ~S" x))))
```

図 5: ベンチマーク問題 2

表 2: HCL 上の PCL ベンチマーク

ベンチマーク例題	実行時間 (CPU 時間)
(tak-gf 8 4 0)	4.03 Sec.
(tak-typecase 8 4 0)	0.18 Sec.
(test-generic-function 100000)	40.3 Sec.
(test-typecase 100000)	18.5 Sec

使用できるならばその自然な記述形式を考えると十分実用になるものと考えられる。ちなみに PCL を登載する事による HCL の必要記憶領域の増加は 650KB と小さなものであった。

6 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所である。CLOS の仕様に関しては少なくとも記述の厳密性はかなり改善されている。しかし機能的な豊富さが災いしてか PCL 以外のインプリメントは見あたらないのが現状である。CLOS の中心概念は generic function であると言っても過言ではなく、これは今後 Common Lisp のカーネルのレベルでサポートされるべきであろう。今回行った HCL での実験的なインプリメントはそういった方向性で行われたものである。今後様々な形式のインプリメントによって CLOS 及び 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。本論文で報告したインプリメント技法は CLOS 処理系の高速化・小型化に関してひとつの指針を与えるものであり、得られた処理系は小型のワークステーションでも実用的な環境を実現できるものであった。

参考文献

[1] 井田昌之他. *Common Lisp オブジェクトシステム*. 共立出版, Bit 別冊, 1989 .

[2] 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 .

[3] J. Allen. *Anatomy of LISP.* McGraw-Hill, New York,NY, 1978.

[4] D.G. Bobrow and et al. Commonloops: merging common lisp and object-oriented programming. *Proc. OOPSLA '86*, 1986.

[5] L.G. Demichiel and R.P. Gabriel. The common lisp object system. *Proc. ECOOP '87*, 1987.

[6] G. L. Steel Jr. *Common Lisp the Language.* Digital Press, Burlington,Massachusetts, 1984.

[7] R.Fenchel and J.Yochelson. A lisp garbage-collector for virtual-memory computer system. *CACM Vol.14, No.8, pp.611-612*, 1969.

[8] T. Yuasa and M. Hagiya. Kyoto common lisp report. *Kyoto University*, 1985.