記 号 処 理　40－5
（１９８７．１．１３）

ＨｉＬＩＳＰインタプリタの高速制御方式

湯浦　克彦　　安村　通晃

（株）日立製作所　中央研究所

Common Lisp に準拠した高速の Lispシステム HiLISP のインタプリタ制御方式について述べる。関数制御では、多値返答の拡張が課題となったが、単値と多値とで関数復帰先を分けて別経路で処理することを可能とした。変数管理では、静的スコープやクロージャ機能の拡張が課題となった。通常の処理での変数結合をスタック上の深い結合で高速に実現しながら、適宜ヒープに値を保存する処理と間接ポインタを用いて値を共有する方式によって、クロージャの特殊な問題を解決した。このほか、動的変数と静的変数の効率のよい判定法、局所関数のスコープと環境のクロージャによる表現法なども明らかにした。なお、本処理系は、HITAC M シリーズ上に作成した。

High Speed Methods for the HiLISP Interpreter

Katsuhiko YUURA and Michiaki YASUMURA

Central Reserch Laboratory, Hitachi,Ltd.
1-280,Higashi-koigakubo,Kokubunji-shi,Tokyo 185,Japan

The authors report on high speed methods for the HiLISP (High Perfermance List Processor) interpreter which is based on Common Lisp. Multiple values return is implemented by a method in that single value return and multiple values return have different paths. Lexical scoping and function closure features are implemented by the stack of deep binding method with properly saving values in heap and with using invisible pointers. A method for judgement of special/lexical variable and a method for transformation of local function and environment into closure are also developed. The HiLISP interpreter runs on HITACHI M-series computers.

１．はじめに

自然言語解析、設計自動化、知識処理システム等の基本言語としてLispが注目されている。我々は、Lispの共通言語となりつつある Common Lisp[1] に準拠した HiLISP (High Performance List Processor) のインタプリタ／コンパイラの基本方式を設計し、HITAC M シリーズ計算機上に処理系を作成した。[2][3]

HiLISPの設計の目標としては、 compiled-functionの実行速度を向上させることと、インタプリタでの性能も軽視せずに高速性を維持することを取り上げた。

Lisp言語のインタプリタに関しては、すでに変数管理法などについていくつかの研究や高速の処理系が発表されている。[4] Common Lispのインタプリタに対しても、Lisp言語のプログラミング過程がインタプリタ中心であることから、同様に高速の処理系が求められている。Common Lispでは、多くの機能が拡張されている。ことに多値返答機能の導入、変数管理規則（スコープとエクステント）の厳密化、局所関数等環境概念の拡大などを実現するには、インタプリタの基本制御をより一般性の高いものとすることが必要であり、これを素直に実現すれば負荷の重い制御となる。これら拡張機能の特徴を分析して、基本制御をより負荷の軽い制御として最適化することが課題となった。

高速化にあたっては、Lisp言語の中心的な機能と利用頻度の低い拡張機能とを区別し、実用効果を狙って、中心的な機能をより高速に実現することにした。

本稿では、多値返答を含む関数復帰処理、変数管理規則の実現法、局所関数の実現法について述べ、Mシリーズ上に作成した処理系の、性能測定結果の一部を報告する。

２．多値返答を含む関数制御

２．１　ＨｉＬＩＳＰの基本方針

以下の基本方針のもとに、HiLISPの設計をすすめた。

（１）メモリ

スタックとヒープを置く。ヒープは特定のガーベジ・コレクションを意識しない。

（２）データ型

特定のタグ判定機構を持たない汎用機上での高速実現のため、タグをポインタ側に持たせる。

（３）スタック

制御スタックと結合スタックを設ける。制御スタックには、動的リンク、静的リンク、静的結合に関する情報などを置く。結合スタックには、動的結合に関する情報を置く。

（４）レジスタ

汎用レジスタをスタック・ポインタ、プログラム・ベース、関数値およびワークなどに用いる。

（５）関数インターフェース

引数はスタック渡し、結果はレジスタで返答する。

２．２　多値返答の問題

Lisp 言語は関数型言語であり、 関数制御の速度が、インタプリタはもとより、compiled-function の極限性能を決める第１の要因となる。

従来Lisp言語では、関数は唯一の値（以下、単値と称す）を返答するのみであったが、Common Lisp では多値を返答する機能が拡張された。Common Lispにおける多値の生成と受け取り側の処理のパターンを図２．１に示す。末尾以外のユーザ定義関数呼出しの復帰は、Lisp 動作中かなり多く発生するが、このケースで、単値が返答される場合と多値が返答される場合とでは、受け取り側の処理を分けなければならない。そこで、ユーザ定義関数では、単値を返答するか多値を返答するかを受け取り側に知らせる処理が必要であり、末尾以外の呼出しでは、返答が単値か多値で場合分けする処理が必要となった。

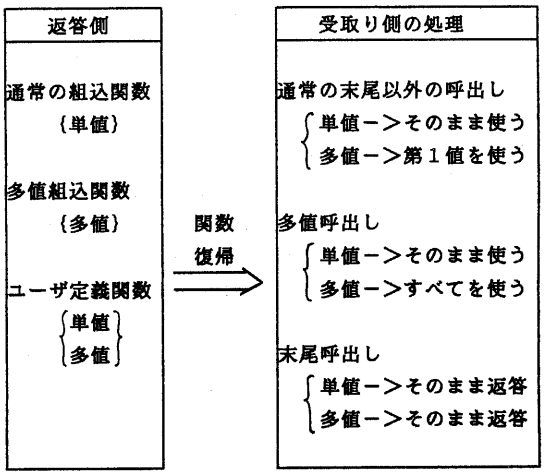


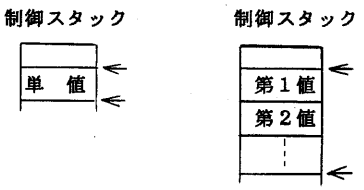
図２．１　　多値の返答と処理

２．３　多値処理方式の検討

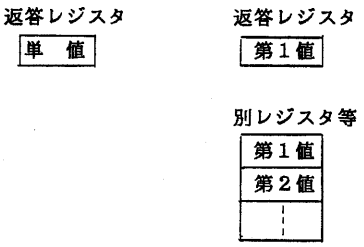
関数復帰の処理方式を、図２．２に３つの方式について比較検討した。

（１）スタックに値を返答する方式 [5]

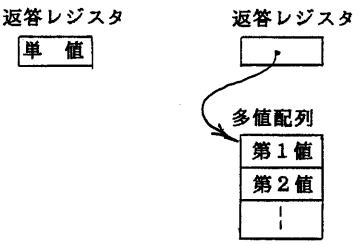
単値と多値は同型式で保持される。受け取り側の処理は単値でも多値でも区別する必要がなく、多値が返答された場合でも特殊な処理を要しない。しかし、大多数の返答は単値であり、単値の処理にスタック操作を必ず用いることの負荷は少なくない。



（１）スタック型式



（２）第１値抽出型式



（３）多値配列型式

図２．２　　多値保持の諸型式

（２）返答レジスタに第１値を設定する方式

多値のうち第１値（値数０のときは、既定値 nil ）を単値と同様に返答レジスタに設定しておけば、通常の末尾以外の呼出しでは、返答が単値か多値かの判定は不用となる。しかし、一旦生成された多値の残りの値を、別の機会に誤って参照しないためには、例えば、末尾以外の呼出しの返答値の処理時に別レジスタ等を必ず初期化するなどの、多値無効化の処理が必要となる。

（３）多値を配列等で単純に返答する方式

末尾以外の呼出しで返答が単値か多値かを判定する負荷が問題である。

```
(defun f (x)
   (let ((y (g x)))
     (h y)))
```

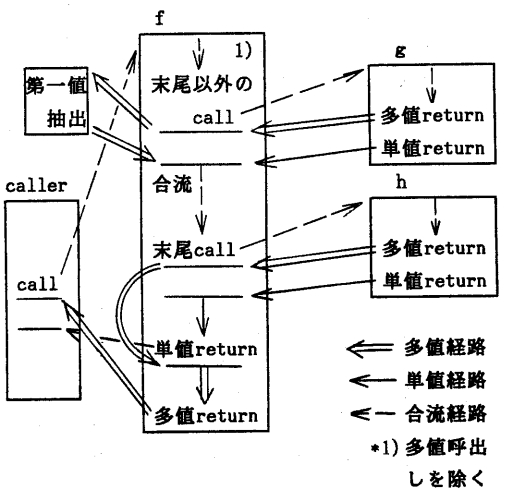


図２．３　多値経路振分け方式

２．４　ＨｉＬＩＳＰにおける多値経路振り分け方式

HiLISPでは、上記(3)の保持型式のまま、単値と多値で関数復帰時の戻りアドレスを分けて、単値／多値の判定を省略した。今回開発した多値経路振り分け方式の動作を図２．３に示す。

通常の末尾以外の呼出しでは、多値で戻ると第１値を抽出して単値の処理と合流する。末尾呼出しでは、単値で戻るとさらにそこから単値戻りアドレスへ、また、多値で戻るとさらにそこから多値戻りアドレスへと経路を振り分けて、単値を返答しているか多値を返答しているかの情報を伝播させる。

この方式による関数復帰処理を、従来のLisp 言語の単値のみ返答する仕様での関数復帰処理と比較する。多値が返答された場合には１ないし２回の分岐増などがあるが、単値が返答された場合には、従来 Lisp 言語での単値のみの関数復帰処理と同じシーケンスである。大多数の関数復帰が単値によるものであることを考え合せると、ほとんど負荷増はない。

３．　変数管理方式

３．１　スコープとエクステント

Common Lisp においては、変数他の環境の参照規則をスコープとエクステントという概念を用いて規定している。スコープ（有効範囲）はプログラムのテキスト上文脈的に確立される環境であり、エクステント（寿命）はプログラムの動作上時間的に確立される環境である。 Common Lisp で扱う環境を表３．１に示す。このうち、静的変数および動的変数の実現法については本章で、関数については次の章で述べる。静的変数および動的変数の参照規則は、それぞれ静的スコープ、無限エクステントおよび無限スコープ、動的エクステントによるが、本稿では、以降これを静的スコープおよび動的エクステントと省略して記述する。

表３．１　スコープとエクステント

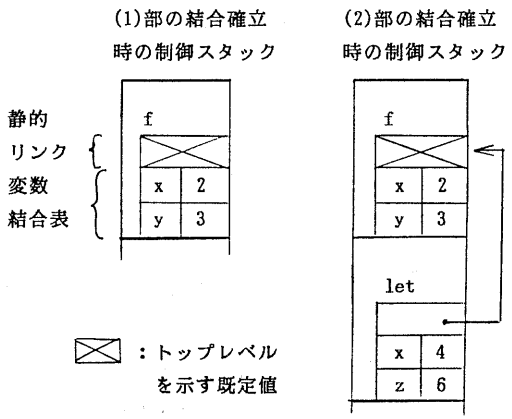
	動的 エクステント	無限 エクステント
静的 スコープ	block 名 go の飛び先	静的変数 関数
無限 スコープ	動的変数 catch のタグ	定数

```
(defun f (x y)              ———  (1)
  (let ((x 4)(z 6))         ———  (2)
    (list x y z)))
(f 2 3)   が入力されたとき
```

(1)部の結合確立時の a-list
((x . 2)(y . 3))

(2)部の結合確立時の a-list
((x . 4)(z . 6)(x . 2)(y . 3))

（ａ）　ａ－ｌｉｓｔ方式



（ｂ）　スタック結合方式

図３．１　　静的変数の 深い結合

3．2　静的変数とスタック結合方式

従来の Lisp 言語では、変数の参照規則を、compiled-function では静的スコープを採用しているのに対し、インタプリタでは、動的エクステントを採用している例がほとんどであった。動的エクステントは、処理効率の高い 浅い結合 (shallow binding) によって容易に実現されるが、下記のようにコンパイル前後で異なる値を返答する場合があり、使用上問題となっていた。

```
(setq x 5)
(defun f (x) (g))
(defun g () x )
(f 7)  -->  { 静的スコープでは  5
            { 動的エクステントでは  7
```

Common Lisp では、インタプリタの動作と compiled-function の動作の仕様統一のため、インタプリタにおいても変数参照に静的スコープを採用している（静的変数と称す）。動的エクステントでの参照をおこなう動的変数は、変数に special 宣言を付加し、区別して使用する。

次に静的スコープの実現法について述べる。Lisp プログラムは、いくつかのスコープに頻繁に出たり入ったりしながら動作するため、変数名と値との結合表をスコープ毎に準備する 深い結合 (deep binding) が、唯一の変数名アドレスにその時点での値を設定する 浅い結合 より有利と考えられた。

深い結合 の実現方式としては、ヒープに変数名と値の対をリストで保持する a-list 方式が最も一般的である。a-list 方式 [5]（図３．１（ａ））は、後で述べるクロージャの機能の実現には適しているが、変数結合が生ずるたびに cons セルを消費することや変数参照のたびにリストの探索が必要なことで効率上好ましくない。

HiLISP では、図３．１（ｂ）に示すように、変数名と値との結合表をスタック上のアレイとして実現した。この方式をスタック結合方式と呼ぶ。スタック結合方式では、変数結合時にヒープから cons セルをアロケートする負荷が避けられ、変数参照時には、リストを手繰ることなくアレイの順次検索で済ますことができる。変数結合表のアレイは、スタック・フレーム毎に複数生成されるが、静的リンクを辿ることによって、静的スコープ内にあるものをすべて検索することができる。ここで検討した方式は、文献6) でも紹介されている。

3．3　Ｃｏｍｍｏｎ　Ｌｉｓｐのクロージャ機能

クロージャ (closure) とは、関数をその定義場所の静的環境とともに保持する機能である。Common Lisp では、特殊形式 function （#'と略記する）実行時のすべての静的環境、つまり、参照可能なすべての静的変数、局所関数などを保持し、さらに、以下の点についてサポートする。

（１）静的変数は無限エクステントであるから、動的エクステント外、つまり、もはやスタック上にない変数結合も参照できるようにする。

```
例  (let ((x 12))
       (setq f  #'(lambda () (1+ x))))

    (funcall f)  --->  13
```

（２）クロージャに取り込まれた静的変数は、クロージャを定義した場所からも、起動されたクロージャ内からも共通に参照、更新される必要がある。

```
例  (let ((x 12))
       (setq f  #'(lambda () (1+ x)))
       (setq x 7))

    (funcall f)  --->  8
```

（３）同一の静的変数を複数のクロージャが取り込み、おのおのクロージャが起動された場合には、それぞれのクロージャ内から共通に参照、更新される必要がある。

```
例  (let ((x 12))
       (setq f  #'(lambda () x))
       (setq g  #'(lambda () (setq x (1+ x)))))

    (funcall f)  --->  12
    (funcall g)  --->  13
    (funcall f)  --->  13
```

3．4　間接ポインタを用いたクロージャの実現

スタック結合方式のもとに上記の機能を実現する方法を図３．２に示す。静的変数は無限エクステントではあるが、特殊形式 function で引用された静的環境

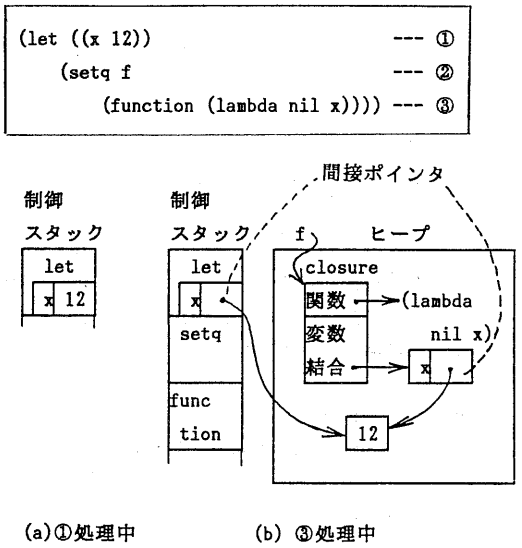


図３．２　間接ポインタの利用による
クロージャの実現

を呼び出したり、特殊形式 flet などを実行しない限りはスタック上に結合情報を持つ。特殊形式 flet などの処理方式については次の節で述べる。特殊形式 function の（１）の機能については、特殊形式 function が実行された時点でのみ、その文脈で参照可能なすべての静的変数をスタックの静的リンクを辿って検索し、それぞれの値をヒープ上に移動し、クロージャとして保存することにした。

値を移動することに伴って、（２）、（３）の機能の実現法が問題となるが、スタックの結合表の元の値の位置には移動先への間接ポインタを設定し、またクロージャ群の変数結合表にも同様の間接ポインタを設定して、互いの値の共有を可能にした。クロージャを呼び出す時には、ヒープに保存した間接ポインタ付きの変数結合表をスタック上に再現し、スタックを介して高速に参照する。

以上の方式により、特殊形式 function などの現れない通常の処理では、静的変数の参照時に、値が間接ポインタを介するものか否かの判定の追加のみで、クロージャ機能をサポートすることができた。

３．５　動的変数の実現

Common Lisp では、静的変数のほかに、無限スコープ、動的エクステントを持つ動的変数をサポートする。動的変数の結合は、結合スタックを退避領域として浅い結合 により実現した。動的変数の指定は、

```
(defvar  x 7)
```

のようにグローバルに宣言する方法と、

```
(let ((x 7))
    (declare (special x))  .  body  )
```

のように局所的に宣言する方法があり、後者の場合には、body に現れた変数 x を静的変数として参照するか、動的変数として参照するかの判定が問題となる。

HiLISP における動的変数の判定方式を図３．３に示す。宣言自体は静的スコープを持つので、スタック上の変数結合表に宣言に関する情報を置くことができる。浅い結合 によって値は変数シンボルに設定されている。スタック上の変数結合表には、通常は変数の値が設定されるが、 declare special 宣言された変数については、変数シンボルの値への間接ポインタを設定した。これによって、変数表のエントリ参照時に、間接ポインタの有無をチェックすれば、動的変数の参照か静的変数の参照かを制御することができることになる。

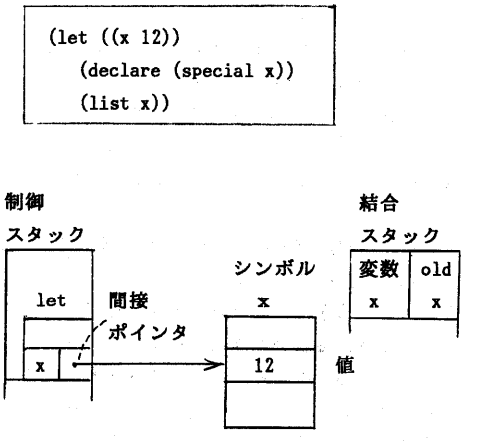


図３．３　動的変数と動的結合

4． 局所関数の実現

4．1 局所関数の機能

従来 Lisp 言語では、関数はすべてトップレベル（空環境）で、グローバルに定義されたものを意味したが、Common Lispでは、特殊形式 flet 等で定義される局所関数をも導入しており、以下の機能を実現しなければならない。

（1）関数についても静的スコープを持つ。

```
例  (defun f (x) (g x))
    (defun g (x) (1+ x))

    (flet ((g (x) (1- x)))
      (g 5))  --->  4   ;  局所関数 g を適用

    (flet ((g (x) (1- x)))
      (f 5))  --->  6   ;  グローバルな関数 g
                           を適用
```

（2）局所関数は、一般に静的環境（静的変数、局所関数など）を伴う。

```
例  (let ((x 2))
      (flet ((f (y) (+ x y)))
        (let ((x 4))
          (flet ((g () (f x)))
            (g )))))

    --> (g)
    --> (f 4)     ; 局所関数 g を適用、そして
                    g の環境の変数 x を参照
    --> (+ 2 4)   ; 局所関数 f を適用、そして
                    f の環境の変数 x を参照
    ---> 6
```

4．2 クロージャによる局所関数の実現

トップレベルでクローバルに定義された関数はラムダ式のみで表現するが、局所関数はクロージャとして実現した。

```
(sys:closure  lambda-list  variable.vector
      closure.vector)
```

クロージャは上記のようなリストで実現した。第1要素に指示子、第2要素に関数のラムダ式、第3要素に前章で述べた変数結合表、第4要素に第2要素の関数において参照可能な局所関数名と局所関数を表現したクロージャのベクタを置く。

局所関数を定義、実行する特殊形式 flet は、下記のようにクロージャに変換し評価する。特殊形式 flet では、局所関数 $l\text{-}fn_i(i=1,2,\cdots)$ の定義のもとに式 form を評価するので、式 form を本体とする関数（lambda () . form）についてのクロージャを生成することにする。このとき、局所関数 $l\text{-}fn_i$ もクロージャ化して、特殊形式 flet の評価場所で参照可能な局所関数のクロージャのベクタ $\#\langle c_0\rangle$ と合わせて保持する。$\#\langle c_0\rangle$ はまた、局所関数 $l\text{-}fn_i$ の関数の環境であり、特殊形式 flet がネストした場合に必要である。式form のなかで関数名 $l\text{-}fn_i$ が用いられている場合は、関数（lambda () . form）のクロージャ内に保持された局所関数 $l\text{-}fn_i$ のクロージャを呼び出す。

```
(flet ((l-fn1 l-ld1 l-body1)
       (l-fn2 l-ld2 l-body2) ··· )
      . form )
```

⇓

```
(sys:closure (lambda () . form)
      #<v0>
      #( l-fn1
         (sys:closure (lambda l-ld1 l-body1)
                      #<v0> #<c0> )
         l-fn2
         (sys:closure (lambda l-ld2 l-body2)
                      #<v0> #<c0> )
         ...
         . #<c0> ))
```

ただし、

- $l\text{-}fn_i$ ：局所関数名
- $l\text{-}ld_i$ ：局所関数のラムダリスト
- $l\text{-}body_i$ ：局所関数本体
- form ：局所関数定義のもとで評価される式
- $\#\langle v_0\rangle$ ：flet の現れた場所で参照可能な静的変数の変数結合表
- $\#\langle c_0\rangle$ ：flet の現れた場所で参照可能な局所関数を表わしたクロージャのベクタ

5. HiLISPの試作と性能測定

HITAC M シリーズ計算機上に処理系を作成した。Common Lisp の制御構造に関る機能は、すべてサポートしている。記述言語は、高速化のため L-code と称する専用の中間言語を設定し、おもにこれを用いた。L-code には、タグ判定、スタック操作、関数呼出し、cons アロケートなど Lisp 固有の機能のほか、if-then-else、do ループなどの構造化記述機能も含んでいる。L-code はまたコンパイラの中間出力としても用いている。

Lisp コンテスト[7]の例題のうち、いくつかのベンチマークについての性能測定結果を表５．１に示す。多値の拡張による性能の劣化は、ほとんどなかった。クロージャ機能の拡張に関しても、クロージャ機能が適用されるベンチマーク #7,#10 を除いて、ほとんど性能の劣化はなかった。

表５．１ LISPコンテスト代表１２題
HiLISP インタプリタ性能

#	ベンチマーク	実行時間 (msec)
1	Tarai-5	14000
2	List-tarai-4	630
3	String-tarai-4	950
4	Flonum-tarai-4	590
5	Bubble-50	330
6	Seq-100	35
7	BITA-6	27
8	Sort-100	2.2
9	TPU-6	3100
10	Prolog-sort-20	650
11	Diff-5	470
12	Boyer	30000

HITAC M280H にて測定

6. おわりに

Common Lisp 仕様の処理系、とくにインタプリタを高速に実現する方法を明らかにした。多値制御方式、変数管理方式などの最適化により、高速性は維持されていると考えられる。

参考文献

1) Guy L. Steele Jr. : "Common Lisp the Language",Digital Press,1984

2) 湯浦他：" 高速Commmon lisp - HiLISP の実現",情報処理学会第33回全国大会2E-1,1986

3) 高田他：" HiLISP コンパイラにおける高速化方式",ソフトウェア科学会第3回大会c-2-3,1986

4) 近山："Utilisp システムの開発",情報処理学会論文誌24-5,1983

5) 湯浅他："Kyoto Common Lisp の実現",情報処理学会 記号処理研究会資料34-1,1985

6) H. G. Okuno et.al. : "TAO : A Fast Interpreter-Centered System on Lisp Machine ELIS",ACM Conference on Lisp and Functional Programming,1984

7) 奥野：" 第３回LISPコンテストと第１回prologコンテストの課題案",情報処理学会 記号処理研究会資料28-4,1984