記　号　処　理 39－6
（1986. 11. 10）

CommonLoops のためのあるメッセージ送信機構

井田昌之
青山学院大学

CommonLoops におけるメッセージ送信を高速化するための基本的な考察と、それに基づくハードウェア機構について述べる。CommonLoops のインタプリタである D-Eval を示し，メソッドの選択時における2つの着眼点を見出した。クラス間の階層関係の判定とメソッドキャッシュである。これらを4Kbit連想メモリチップを中心としたCAM ボード上で高速検索する。CAM ボードアクセスの高速化のためにアドレスバス上に検索情報等をのせる。この回路は現在，設計が終了した段階で，VMEバスを使用した 68020 ベースの CommonLisp システムに対する機能ボードとして動作するように試作を進めている。

On a Message Sending Mechanism for CommonLoops

Masayuki Ida
Aoyama Gakuin University
1 Morinosato Aoyama, Atsugi, Kanagawa, Japan 243-01

This paper describes a basic considerations for designing an accerelator for the message sending mechanism of CommonLoops, and a resulting design of an associative memory based hardware. Whole research is carried with a CommonLoops interpreter named D-Eval of the author. This paper focused on the two points; class precedence judgement and method cache. They are considered to be most important keys to speed up the message send. They are assisted by a CAM board with 4Kbit CMOS associative memory. To minimize a triggering, address bus is used to contain retrieval information. The design of the circuit was finished. And the experimental board is under construction for a 68020-VME based Common Lisp implementation.

1. はじめに

CommonLoops は，Common Lisp 用のオブジェクト指向機構であり，その概要は［bob86］に示されている。CommonLoops の特徴は，メソッド定義とクラス定義を分離している点，メッセージ送信を関数呼出しと同一の形式で扱う点にある。ラディカルな見方をすれば，すべての関数呼出しはメッセージ送信の中に包含される。

一般に，オブジェクト指向機構には，実用化のために乗り越えるべきハードルとして，メッセージ送信の高速化が課題として存在する。このためにはソフト／ハードを問わずキャッシュの利用が多く見られる。CommonLoopsについても同様のことが言える。そのポータブル処理系である PCL では，各セレクタに付随させて存在する discriminator object 中にソフト的なキャッシュ表（標準で8エントリ）を持ち，それによりメソッドの選択の高速化を計っている。

筆者は，PCL が公開された1986年2月以前より CommonLoops に興味を持ち，Xerox Parc との交流を続けてきた。特に CommonLoops の基準メカニズムを明らかにさせる点を中心として，［ida85］に d-eval 及び d-evlis を示し，［ida86］に具体的な基本構造を示した。

ここでは，D-Eval 処理系［ida86］に対して，今回，連想メモリを用いたハード的な支援機構の開発を進めているので，これに関連する基本設計について述べる。

2. D-Eval

D-Eval は，CommonLoops のインタプリタであり，結果として Common Lisp の Eval を包もうとするものである（しかし，現在の時点では CommonLoops に重点を置いており，完全な Common Lisp インタプリタを主眼としてはいない）。

D-Eval の主要部分の定義を図1に示す。また D-apply の中で最も鍵となる部分である most-specific-method 関数の定義を図2に示す。クラス定義は継承を記述する :include の処理と，後述するクラスID の処理を付加した defstruct で行なう。

図3に，D-Eval をトップレベルにおいた会話例を示す。☆印の行はデバッグ情報である。この例ではクラスとくては，Common Lisp 組込みの型を用いているがユーザ定義のクラスでも全く違いはない。

まだ，はっきりとした結論はでていないが，この D-Eval の試用を通して，次の2点に着眼してハードウェアサポートを作成する方針をたてた．これらはいずれも most-specific-method 関数（図2）の高速化のためのものである。

1. クラス間の階層関係を判定する SUBTYPEP
2. ハードウェアキャッシュを利用した手続的探索の short cut

most-specific-method は3引数の関数で，第一引数には選択対象となるメソッドリスト，第二引数には discrimination の鍵とすべき引数の個数，第三引数には，discrimination を起動する class-specifier のリストを与える。クラス間の優先順位（このメッセージ送信が行なわれる直前に決定される順位に基づいて）を考慮しながら，与えられた candidates の中で最も specific なもの決定しなければならない。

［ida86］では，TYPE-OF を「最も specific な」型を返す関数として定義しており，それに基づき SUBTYPEP を用いてクラス間の階層関係を規定させている。SUBTYPEP

```
;;;;
;;;;    Discriminating Eval
;;;;                                  by Masayuki Ida 1986.10.06 1st version
(in-package 'pcl)

;;;; ---- D-eval ----
(defmeth d-eval ((x number))     (values x (type-of x)))
(defmeth d-eval ((x string))     (values x 'string))
(defmeth d-eval ((x character)) (values x 'character))
(defmeth d-eval ((x null))       (values x 'null))
(defmeth d-eval ((x symbol))
   ;; caution : this version does not take an account of lexical scoping or binding
   (let (( val (cond ((keywordp x) x)
                     ((eq x t)      x)   ; singular point in symbol
                     ((boundp x)   (symbol-value x))
                     (T     (error "Attempt to eval an unbound symbol ~s" x))
                     )))
                  (values val (type-of val))))
(defmeth d-eval ((x cons))
    (declare (special x))
    (let (( val     (d-eval-cons (car x)(cdr x))))
        (values val (type-of val))))

;;;; ---- D-eval-cons ---- for d-eval with cons type
(defmeth d-eval-cons ((fn symbol) args)
    (declare (special x))
    ;; if (car form) is symbol,
    ;;   it should be a macro name, special-form name, or a function name
    (cond      ((macro-function fn) (d-eval (macroexpand x)))
               ((special-form-p fn) (d-eval-special-form x))
               ((fboundp fn) (d-apply fn args))
               (t (eval x)))
               )
(defmeth d-eval-cons ((fn cons) args)
        (cond   ((eq (car fn) 'lambda) (apply fn args))
                (T (error "illegal form for D-eval-cons ~s" fn))))

;;;; ---- D-eval-special-form ----
(defun d-eval-special-form (x)
        (cond
        ;; caution : only QUOTE is processed
              ((eq (car x) 'quote) (values (cadr x) (type-of (cadr x))))
        ))

;;;; ---- D-apply ----
(defun d-apply (fn args)
    (format t "d-apply entered. fn = ~s args = ~s" fn args)
    (let ((fn-body (symbol-function fn)))
      (if (and (consp fn-body)(consp (car fn-body))
               (eq (caar fn-body) 'method) )
          ;; then,
          (multiple-value-bind (x y) (d-evlis args)
                 (apply (most-specific-method fn-body (length y) y) x) )
                        ;; will find the most-specific method in fn-body
                        ;; keys are (length y) : number of the elements types in y
                        ;;          y  :  type-specifiers of the arguments
          ;; else
          ;;   escape to the usual apply
          (apply fn (d-evlis args)))
       ))

(defmacro ndefmeth (selector args &rest forms)
   (if (not (fboundp selector)) (setf (symbol-function selector) nil))
   (push
   (let ((types (mapcar #'(lambda (x) (if (consp x) (cadr x) t)) args)))
  (list 'method (length types) types
 ((list 'lambda (mapcar #'(lambda (x) (if (consp x)(car x) x)) args )
  (list* 'block selector forms))))

    (symbol-function selector)
  )
    (list 'quote selector)
   )
```

Fig. 1 D-Eval, D-apply definitions ([Ida86])

```
;;;; find the most specific method

(defun most-specific-method (fn number-of-specifiers type-specifiers)
(do* ((methods fn (cdr methods))
                (method (car methods)(car methods))
                (candidate nil)
                (types (caddr method) (caddr method)))

        ((null methods)
        (if (null candidate)
                (error "Method not found")
  (progn (print 'the-most-specific-method-is)(print (cadddr candidate)))))
  ;;; main body -------- make candidate list --------
  (when (eql number-of-specifiers (cadr method))   ; check the length
        (if (equal types type-specifiers) (return (cadddr method)))
  ;;; condition of the comparison method-specifiers and type-specifiers
  ;;;  1. should have the same length
  ;;;  2. left-to-right search
  (do ((ftype type-specifiers (cdr ftype))
        (mtype types (cdr mtype)))
  ((null ftype)
                ;; find the most specific
                (if (null candidate) (setq candidate method)
                (let ((ctypes (caddr candidate)))
                (do ((ctype ctypes (cdr ctype))
                (mtype types (cdr mtype)))
                ((null ctype) (setq candidate method))
                ; type comparison(the current candidate and the current method)
                ; was not interrupted.
                ; so, it should be a candidate
        (unless
                (or (eq (car mtype)(car ctype))
                (subtypep (car mtype) (car ctype)))
                (return nil))))
                ;; break the loop
                ))
  (unless
  (or (eq (car ftype)(car mtype))
  (subtypep (car ftype)(car mtype)))
  (return nil)))
))
)
```

図2 most-specific-method 関数

は，most-specific-method 関数の主ループの中で毎回呼出され，かつ主要な役割を果たしている。SUBTYPEPは，次節で述べるクラス優先順位表（CPT，Class-Precedence-Table）を毎回参照する。クラスの生成と継承の変更は動的におこりうるので，原理的にも，また，実際的にも表検索は避けられないものとなる。

3. CPT

Common Lisp の型階層自身，多重継承が存在する。サーチの順序はCPT上で定義する。2，3の例を示すと次のような規定が必要である。

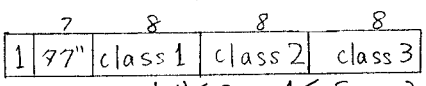
```
 (Get-class-precedence-list 'NULL)
==> (NULL LIST SEQUENCE SYMBOL T)
SIMPLE-VECTOR has
 (Get-class-precedence-list 'SIMPLE-VECTOR)
==> (SIMPLE-VECTOR VECTOR SEQUENCE SIMPLE-ARRAY ARRAY T)
and, SIMPLE-STRING has
 (Get-class-precedence-list 'SIMPLE-STRING)
==> (SIMPLE-STRING STRING VECTOR SEQUENCE SIMPLE-ARRAY ARRAY T)
```

CommonLoops での規則は "left-to-right, up-to-join" であり，例えばNULL型指定に対しては，SEQUENCE のほうがSYMBOLより優先度が高い。SIMPLE-STRING等の例を見てもわかるようにSEQUENCE 優先が原則となっている。

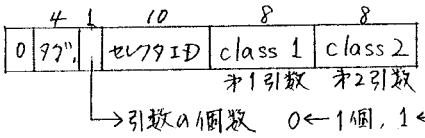
```
D-Eval>(ndefmeth bar (x y) (print 'the-most-generic-case) (cons x y))

bar
; symbol
D-Eval>(ndefmeth bar ((x list) y) (print 'classical-method) (cons x y))

bar
; symbol
D-Eval>(ndefmeth bar ((x number) (y cons)) (print 'multi-method-1) (cons x y))

bar
; symbol
D-Eval>(ndefmeth bar ((x fixnum) (y list)) (print 'multi-method-2) (cons x y))

bar
; symbol
D-Eval>(bar 'a 'b)
☆{d-apply entered. fn = bar args = ('a 'b)
 the-most-specific-method-is
 (lambda (x y) (block bar (print 'the-most-generic-case) (cons x y)))
the-most-generic-case
(a . b)
; cons
D-Eval>(bar 1 '(a b))
☆{d-apply entered. fn = bar args = (1 '(a b))
 the-most-specific-method-is
 (lambda (x y) (block bar (print 'multi-method-2) (cons x y)))
multi-method-2
(1 a b)
; cons
D-Eval>(bar 1.2 'a)
☆{d-apply entered. fn = bar args = (1.2 'a)
 the-most-specific-method-is
 (lambda (x y) (block bar (print 'the-most-generic-case) (cons x y)))
the-most-generic-case
(1.2 . a)
; cons
D-Eval>(bar 2/3 '(1 / 3))
☆{d-apply entered. fn = bar args = (2/3 '(1 / 3))
 the-most-specific-method-is
 (lambda (x y) (block bar (print 'multi-method-1) (cons x y)))
multi-method-1
(2/3 1 / 3)
; cons
D-Eval>(ndefmeth foo ((x number) y)(print 'foo-number-t) (cons x y))
☆{selector = foo args = ((x number) y) forms = ((print 'foo-number-t)
                                                 (cons x y))

foo
; symbol
D-Eval>(ndefmeth foo ((x fixnum)(sy symbol))(print 'foo-fixnum-symbol)(cons x sy))
☆{selector = foo args = ((x fixnum) (sy symbol)) forms = ((print
                                                           'foo-fixnum-symbol)
                                                          (cons x sy))

foo
; symbol
D-Eval>(symbol-function 'foo)
☆(d-apply entered. fn = symbol-function args = ('foo)
((method 2 (fixnum symbol)
           (lambda (x sy)
             (block foo (print 'foo-fixnum-symbol) (cons x sy))))
 (method 2 (number t)
           (lambda (x y) (block foo (print 'foo-number-t) (cons x y)))))
; cons
D-Eval>(exit-d-eval)
nil
```

図3 D-Eval インタプリタとの会話例 [Iida86] より)

(1) CPT エントリ



(2) メソッドキャッシュエントリ



ただし class ID 8bit (Max 256)
セレクタID 10bit (Max 1024)

図－5 CAM WORD の形式

```
 0: (T 0 0)
 1: (PATHNAME T 0)
 2: (STREAM T 0)
 3: (HASH-TABLE T 0)
 4: (READTABLE T 0)
 5: (RANDOM-STATE T 0)
 6: (SEQUENCE T 0)
 7: (SYMBOL T 0)
 8: (ARRAY T 0)
 9: (CHARACTER T 0)
10: (NUMBER T 0)
11: (LIST SEQUENCE T)
12: (CONS LIST SEQUENCE)
13: (NULL LIST SEQUENCE)
14: (NULL SYMBOL T)
15: (VECTOR SEQUENCE T)
16: (VECTOR ARRAY T)
17: (SIMPLE-ARRAY ARRAY T)
18: (STRING VECTOR SEQUENCE)
19: (BIT-VECTOR VECTOR SEQUENCE)
20*: (SIMPLE-STRING STRING  VECTOR SEQUENCE)
21: (SIMPLE-STRING SIMPLE-ARRAY ARRAY)
22*: (SIMPLE-BIT-VECTOR BIT-VECTOR VECTOR SEQUENCE)
23: (SIMPLE-BIT-VECTOR  SIMPLE-ARRAY ARRAY)
24: (SIMPLE-VECTOR VECTOR SEQUENCE)
25: (SIMPLE-VECTOR SIMPLE-ARRAY ARRAY)
26: (STRING-CHAR CHARACTER T)
27: (STANDARD-CHAR STRING-CHAR CHARACTER)
28: (RATIONAL NUMBER T)
29: (INTEGER RATIONAL NUMBER)
30*: (RATIO INTEGER RATIONAL NUMBER)
31*: (FIXNUM INTEGER RATIONAL NUMBER)
32*: (BIGNUM INTEGER RATIONAL NUMBER)
33: (FLOAT NUMBER T)
34: (SHORT-FLOAT FLOAT NUMBER)
35: (SINGLE-FLOAT FLOAT NUMBER)
36: (DOUBLE-FLOAT FLOAT NUMBER)
37: (LONG-FLOAT FLOAT NUMBER)
38: (COMPLEX NUMBER T)
```

Fig. 4 The Complete list of the start-up image of CPT

```
(defmeth f ((x symbol)) 'symbol)
(defmeth f ((x number)) 'number)
(defmeth f ((x fixnum) (y string))
          'fixnum-and-string)
```

⇓

```
((METHOD 1 (SYMBOL)
   (LAMBDA (X)(BLOCK F 'symbol)))
 (METHOD 1 (NUMBER)
   (LAMBDA (X)(BLOCK F 'number)))
 (METHOD 2 (FIXNUM STRING)
   (LAMBDA (X Y)
     (BLOCK F 'fixnum-and-string))))
```

⇓

a	0		0	f	symbol	–
a+1	0		0	f	number	–
a+2	0		1	f	fixnum	string
⋮				⋮		

図－6
メソッドキャッシュの例

このクラス優先順位リストを図4のように表わす。若い番号の方が優先度が高い。例えばエントリ13と14の配置はSEQUENCE優先を表わしている。下型の根を除くと，エントリ20, 22, 30, 31, 32 を除いて，三項関係で表わせる。これに基づいて後述する連想メモリ上での構造を決定している。

4. メソッドキャッシュ

[bob86]では，現存コードの50%はオー引数によるdiscriminationのみであることを報告している。これは，まだCommon Loopsになって拡張されたmulti-methodが普及する以前のLoops, Flavorsの話であり，これから直接の結論を見出すことは

できない。筆者の近辺に存在するコードでは約90%が第2引数までで済んでいる。そこで，セレクタ名と2つの引数に対するメソッドキャッシュを設計した。

5. CAMボードの設計

現在，KCLを68020(16.67MHz)，主記憶4MB，VMEバスによるUNIBOX上に移植し(実作業はソード(株)による)，それに対してファンクションユニットとして動作するCAMボードを開発中である。CAM(Contents Addressable Memory)としては，NTT厚木研で開発された4kbit連想メモリ[Ogura83]を中心に構成している。4kCAMは32bit×128wの構成を持つ。これらから1kwの連想メモリを形成させる。

図5に，CAM WORDの形式を示す。CAMには3.及び4.で考察した2種の情報を混載させる。混載によるオーバヘッドは無い(検索マスクの存在)。クラスの総数はたかだか100～200なので8bitでクラスIDをなし，defstruct時にユニークな番号を与えてしまう。セレクタIDはdefmeth時に決定されるユニークIDで16bitをとった。CPTエントリに対する検索は，CAMエントリの有無及び下位16bitとの比較により真理値をうる。メソッドキャッシュエントリに対する検索は，検索して得られた語の番地を用いて，主記憶上の表のインデクスとし，そこに実行すべき形式をおくこととした。CAMの語長を拡げればすべてCAMボード上で処理できるが，最初の試作なのでこうした。

メソッドキャッシュのサーチは図1のD-Applyの下線部を次のように修正する。

```
(let ((ln (length y)) addr)
  (if (and (< ln 3) (setq addr (CAM-search fn y))) (load-body addr)
      (most-specific-method fn-body (n y))))
```

図6にボード構成を，図7に基本動作シーケエンスの主要部を示す。また，表1表2に使用する連想メモリチップの特徴を示す。

図7に示した基本シーケエンスのうち，複数選択検索はSUBTYPEP及びCAM-searchのなかで行なわれる。(詳細は紙面の都合で省略する)。

CAMボードのアクセスはVME Rev.Cの(A32, D16)で行なう。アドレスラインの32ビットのうち，上位2bitはボードデコードに利用する。最下位16ビット(lowerhalf)は，CAMアドレス，検索のキーを置く。CAMに対する制御は，表2に示すI0～I4及び制御信号を含んで1×bitがあてられる。CAMエントリロード時はデータバス16bitにより情報をおぎない，Half word Register to memoryのwriteによりすべてを行なう。CAMサーチ時には，サーチキーはなかなか16ビットなので，Fullword memory to registerのreadによりすべてを行なう。複数選択信号Pは，図5にある最上位ビットのかわりに，read時には最上位bitとして読出す。

CAMボードのサイクルは4MHzとしており，ホストからは250nsecのメモリのように見える。検索は1サイクルで完了する。総じてソフトのみによる場合に比して，少なくとも10倍の速度向上を予測している。

謝辞　本研究は文部省科研費奨励研究 No.61750344による。連想メモリの提供をうけたNTT厚木研究所 山田稜一郎氏，小倉武氏，長沼次郎氏に感謝する。68020-VMEについてはソード(株)よりの情報提供と協力があった。

[bob86] D.G Bobrow, et. al. CommonLoops: Merging CommonLisp and Object-oriented Programming, Proc. OOPSLA'86 ACM 1986
[ida85] M. Ida, An Interpretation of the CommonLoops Specification, WGSYM 85-3, IPSJ. Dec. 1985
[ida86] ―, Discriminating Eval, proc. Annual Conf. JSST, Nov. 1986
[Ogura83] 小倉 他, 4Kb CMOS 連想メモリLSI, 信学技報, SSD83-78 pp45-52, 1983

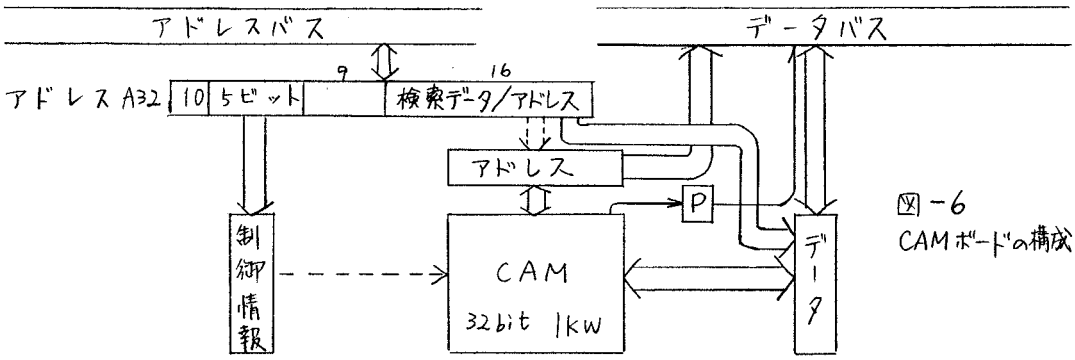


図-6
CAMボードの構成

表-1　4k連想メモリLSIの諸元と特徴

ワード構成	128w × 32bit
機能的特徴	・完全並列な一致検索機能 ・ビット直列ワード並列な関係検索機能 ・ビット方向への拡張性 ・並列部分書込み機能 ・ガベジタグによるGC
動作モード数	30
サイクルタイム	140ns
チップサイズ	10.3mm × 8.4mm
総素子数	71,300
プロセス技術	3μm CMOS 金属2層配線
消費電力	250mW (5MHz)
発表年	1983 by NTT
端子数	53 + 電源9

表-2　各端子の役割

データ	D1~D32	(32)	I/O	
アドレス	A1~A7	(7)	I/O	アドレス0~127
コマンド	I0~I4	(5)	I	30の動作モード
電源	VDD	(4)	I	+5V
	VSS	(5)	I	GND
制御	SCI	(1)	I	Scan-in
	CP	(1)	I	1相クロック
	PEI	(1)	I	
	PEO	(1)	Ō	複数選択信号
	FC	(1)	I	対チップ選択
	RC	(1)	I	Resolve clear
	CS	(1)	I	チップセレクト
	BS	(1)	I	分離選択
	SCŌ	(1)	Ō	Scan-out

#bits	operation (5)	data (32)	address (7-)	cs (1)	fc (1)	bs (1)	rc (1)
Initialization	1.ESA	-	-	-	-	1	-
	2.WMDA	0	0	1	0	1	0
	3.WID	0	-	1	0	1	-
	4.WASC	0	-	1	0	1	-
	5.ESA	-	-	-	-	1	-
NOP	WID	0/1	-	1	-	0	-
Retrieve multi	1.WMD	data	-	1	-	-	-
	2.WID	data	-	1	-	-	-
	3.SRD	-	-	-	-	0	-
	4.NOP	-	-	-	-	0	0
	5.RBBN	data	address	1	1	0	1
	(if RBBY, rc be 0)						
	6. loop to 4. while Pextout is ON.						

図-7　基本動作シーケンス

ESA データの無効化　　WASC 非検索選択WORD書込
WMDA マスクデータの書込　　SRD 検索
WID 検索データの書込　　RBBN 複数選択分離読出