記 号 処 理　40－2
（１９８７．１．１３）

ＣｏｍｍｏｎＬｏｏｐｓのＵｔｉＬｉｓｐ上の実現

泉　寛幸　　吉田裕之　　加藤英樹
（㈱富士通研究所）

Common Lisp 上のオブジェクト指向プログラミングシステムとして，CommonLoops が提案されている．その基本部分を UtiLisp上に実現したので報告する．我々は，本システムをUtiLoops (CommonLoops on UtiLisp) と呼称している．UtiLoopsは，システムをコンパクトにすること，土台としてのUtiLisp を活用すること，メソッドの適用を高速に行うことを考慮して実現された．土台としてのUtiLisp を活用すべく，UtiLisp のデータ型を継承し，UtiLisp システムを変更しないように作られた．また，メソッドの適用を高速に行うために，メソッド優先順位の決定方法やスロットアクセス方法を工夫した．本報告では，CommonLoops とUtiLoopsとの差異，UtiLoopsの実現上の工夫，特に，メソッド特定を高速に行うための手法として，メソッド適用表の利用および新しい高速なメソッド特定アルゴリズムについて述べる．

Implementation of CommonLoops on UtiLisp

Hiroyuki IZUMI,　Hiroyuki YOSHIDA,　Hideki KATO
Fujitsu Laboratories Ltd.

CommonLoops has been proposed to an objected oriented system on Common Lisp. We have implemented its kernel on UtiLisp, which we call UtiLoops (CommonLoops on UtiLisp). UtiLoops was implemented so that it is compact, uses the underlying UtiLisp, and is fast for method specialization. In order to utilize UtiLisp, UtiLoops makes use of the data-types of UtiLisp and was implemented without any modification of UtiLisp system. We introduced new methods to determine the orderings of method precedences for fast method specialization and to access slots fast. This paper describes differences between CommonLoops and UtiLoops, and some design considerations for implementation, especially the use of method precedence tables and a new fast algorithm for method specialization.

1．はじめに

Common Lisp 上のオブジェクト指向プログラミングシステムとして，CommonLoops ［1］が提案されている．CommonLoops の基本部分を，UtiLisp システム［4］上に実現したので報告する．我々は，作成したシステムを，UtiLoopsと呼称している．

我々がCommonLoops に注目したのは，(1)その基本部分がコンパクトであり，容易にインプリメントできる，(2)メッセージパッシングとLispの関数呼出しが同じ構文であり，オブジェクト指向とLispが融合してシステム全体が洗練されている，(3)多重継承と多重メソッド機構を有している，(4)簡潔なプログラムが期待できる，等に魅力を感じたからである．

本報告では，UtiLoopsとCommonLoops との差異，インプリメント上の工夫について述べる．特に，適用メソッドの特定に用いるメソッド適用表の構造と利用法を述べる．さらに新しい高速なメソッド特定アルゴリズムを考案したので，それについて詳しく述べる．

2．CommonLoops との差異

土台としてのUtiLisp とCommon Lisp との機能および開発目的の違いにより，UtiLoopsは，いくつかの点で，CommonLoops とは異なる．本章は，その主な違いについて述べる．

(1) 基本部分を中心に実現

CommonLoops は，［1］の論文では基本部分と拡張部分に分けられており，我々は，その基本部分を主に実現した．

(2) メタクラス機能をサポートしない．

メタクラスの詳細な機能がはっきりしていなかったことと，我々にとっては当面の必要性がなかったことから，インプリメントを行なわなかった．

(3) UtiLoops 独自の型構造

土台としてのUtiLisp の型の概念に合うように，UtiLoopsでは，型とその階層関係をUtiLisp のデータ型に合せて独自に定めた．

(4) アクセッサのメソッド化

同名のメソッドやアクセッサを他のクラスでも定義できるように，スロットアクセッサをメソッドとみなした．またCommonLoops では，スロットアクセスをconc名で行う方向になりつつある［2］が，我々は［1］で論じられているようにスロット名のみでスロット値をアクセスできるほうがより自然であると考えた．

(5) ダイナミックスロットの導入

CommonLoops では，ダイナミックスロットは基本部分に入っていない．しかし，ダイナミックスロットを導入することによる速度上のデメリットはない．また，メモリ効率上のデメリットは，ダイナミックスロット機能の便利さに見合うものと判断した．

(6) 引数の数が固定されている

Common Lisp と異なり，UtiLisp では可変個の引数を取る関数をユーザが定義できないので，UtiLoopsのメソッドも，固定個の引数を取るものとした．

3．実現の方針

システムの実現は，次の各事項を考慮してなされた．

(1) コンパクトなデータ構造

UtiLoopsでは，処理の高速化とメモリ節約のために，クラス，UtiLisp のデータ型，インスタンス，メソッドの各データ構造を，ベクタで表現した．

(2) 土台としての UtiLispシステムの活用

UtiLisp 上に作成された他のシステムを，UtiLoops上でも利用できるように，Lispオブジェクトとその階層関係を UtiLispの型に合うように定めた． UtiLoops では，UtiLisp のデータ型として図1の階層関係で示されるものを扱う．

CommonLoops やUtiLoopsでは，関数呼出しとメソッド呼出しとが同じ構文を持つため，関数EVALは，メソッドを特定し起動する機能を持たなければならない．UtiLoopsでは，メソッドのセレクタ名の関数定義部にメソッドを起動するための手続きを格納することにより，関数EVALを書き換えることなく実現した．その手法は4章で述べる．

(3) メソッド呼出しの高速化

メソッド呼出し時に必要なメソッド優先順位リストの作成時間の短縮を図って，予め，メソッド適用表を作成しておくことにした．メソッドが呼び出されたとき，この適用表を参照して実行される可能性のあるメソッドの優先順位を決定する．詳細は5章に述べる．

また，UtiLoopsでは，スロットアクセッサをメソッドとみなしている．スロットアクセスをメソッドとして高速に行うために，メソッド適用表内にインスタンスであるベクタの要素を直接参照するコードを置くようにした．これについては，6章で説明する．

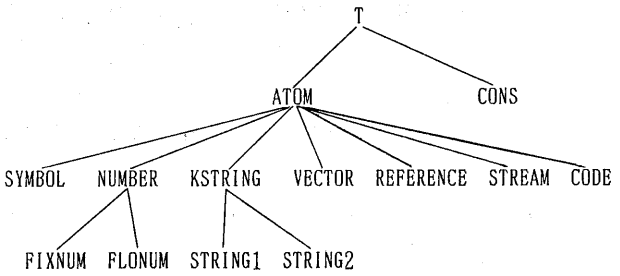

図1 UtiLisp データ型の階層関係

4．メソッド適用関数

UtiLoopsでは，関数EVALを書き換えることなくメソッド呼出しを処理するために，メソッド名の関数定義部には，メソッドを特定し起動する手続きを格納した．

例えば，メソッド(defmethod MOVE ((OBJ block) (X fixnum) (Y fixnum)) ･･･)を定義したとき，シンボルMOVEの関数定義部は，次のようになる．

```
(LAMBDA (OBJ X Y) (APPLY-METHOD 'MOVE (LIST OBJ X Y)))
```

メソッドが呼ばれたとき，関数APPLY-METHODが呼び出され，APPLY-METHODは次のステップを順に実行する．

Step 1 メソッドの実引数のクラスをもとに，後述のメソッド適用表からメソッド優先順位リストを求める．

Step 2 メソッド結合用関数 run-superのために関数名，引数，メソッド優先順位リストを大域変数に束縛する．

Step 3 最も優先順位の高いメソッドを起動する．メソッド実行中に run-superが呼ばれたならば，run-superがメソッド優先順位リストの次の要素であるメソッドを起動する．

5．メソッド適用表の利用

5.1　メソッド優先順位リストを動的に計算する必要性

CommonLoops やUtiLoopsにおいては，同名のセレクタに対する適用メソッドの優先順位は，実引数のクラスにより定まる．例えば，図2のようなクラス定義の上で，一引数のメソッドにおけるメソッド優先順位リストを考えてみよう．すでに図3のようにクラスD，E，tの上にはメソッドS0が定義され，クラスA，B，Cには，定義されてないものとする．そのとき，Bのクラス優先順位は｛B，D，E，t｝，Cのクラス優先順位は｛C，E，D，t｝となり，DとEの順序が逆転している．そのため，メソッド優先順位リストは，図5のようになり，実引数のクラスがAやCの場合とBの場合とで適用メソッドの優先順位が異なる．

しかしながら，CommonLoops やUtiLoopsでは第一引数だけでなくすべての引数のクラスを考えねばならず，あらかじめあらゆる実引数のクラスの組合せに対して，メソッド優先順位リストを静的に作成しておくのは，必要なメモリ量の点で不可能である．それゆえ，メソッドを実際に起動するときに，その実引数をもとに適用可能なメソッドの優先順位リストを作成せざるを得ない．

```
(defstruct (A (:include B C)) ‥)
(defstruct (B (:include D E)) (スロットｂ ‥))
(defstruct (C (:include E D)) (スロットｃ ‥))
(defstruct D ‥)
(defstruct E ‥)
```

図2クラス定義の例

```
(defmethod S0 ((X D)) 式1)
(defmethod S0 ((X E)) 式2)
(defmethod S0 ((X t)) 式3)
```

図3　一引数のメソッドの例

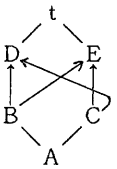

図4　クラス階層

```
実引数がAの場合，（（式2）（式1）（式3））
〃　　B　〃　　（（式1）（式2）（式3））
〃　　C　〃　　（（式2）（式1）（式3））
〃　　D　〃　　（（式1）（式3））
〃　　E　〃　　（（式2）（式3））
```

図5　メソッド優先順位リストの例

5.2　適用表の構造

UtiLoopsにおいては，メソッド優先順位リストの作成時間の短縮を図るため，メソッド優先順位リストをあらかじめ部分的に計算したものをメソッド適用表に格納しておき，メソッド呼出し時にはメソッド適用表をもとにメソッド優先順位リストを計算する方式をとっている．ここで用いるメソッド適用表は，第一引数に適用可能なすべてのクラスをキーとし，適用可能なメソッドを並べた連想リストである．図3の一引数のメソッドの場合と，図6の二引数の場合のメソッド適用表の構造を表1に示す．

```
(defmethod S1 ((X D) (Y D)) 式1)
(defmethod S1 ((X D) (Y E)) 式2)
(defmethod S1 ((X E) (Y D)) 式3)
(defmethod S1 ((X t) (Y E)) 式4)
```

図6　二引数のメソッドの例

5.3　表の利用法

Step 1　メソッドを初めて起動するとき，このメソッド適用表が作られる．

Step 2　二度目以後このメソッドが起動されるとき，

Step 2-1　引数が一個のメソッドならば，

Step 2-1-1　実引数のクラスをキーとして表を検索し，メソッド優先順位リストをとって来る．

Step 2-1-2　このメソッド優先順位リストの先頭のメソッドを起動する．

Step 2-2　引数の個数が二個以上の場合，

Step 2-2-1　第一実引数のクラスをキーとして一次メソッド優先順位リストをとってくる．一次メソッド

優先順位リストとは，第一実引数のクラスだけで，適用メソッドを分類し，その分類をソートしたリストである．

Step 2-2-2　この一次メソッド優先順位リストを，第二の実引数以下のすべての実引数のクラスでさらにソートしてできたリストをメソッド優先順位リストとする．

Step 2-2-3　このメソッド優先順位リストの先頭のメソッドを起動する．

Step 3　メソッド実行中に run-superが呼び出されると，この優先順位リスト内の次のメソッドを起動する．

表1　メソッド適用表の例

引数の個数が一個の例

実引数のクラス	メソッド優先順位リスト	
	仮引数のクラス	ラムダ式
A	E D t	式2 式1 式3
B	D E t	式1 式2 式3
C	E D t	式2 式1 式3
D	・・・	・・・

＊の部分の優先順位は，動的に決定される．
(Step 2-2-2)

引数の個数が二個以上の例

第一実引数のクラス	一次メソッド優先順位リスト	
	仮引数のクラス	ラムダ式
A	E　D	式3
	D　E D　D	式2 式1　＊
	t　E	式4
B	D　E D　D	式2 式1　＊
	E　D	式3
	t　E	式4
C	E　D	式3
	D　D D　E	式1 式2　＊
	t　E	式4
・・・	・・・	・・・

6．　スロットアクセッサの書換え

UtiLoopsでは，スロットアクセッサをメソッドとみなしている．スロットアクセッサと同名のメソッドを定義することにより，一種のデモンを実現できる．

例えば図2の場合，クラスBとクラスCに対して，スロットbとスロットcは次のようにメソッドとして定義される．

```
(defmethod スロットb ((instance B)) (get-slot 'スロットb instance))……①
(defmethod スロットc ((instance C)) (get-slot 'スロットc instance))……②
```

クラスAは，BとCを親に持つので，これらのメソッドを継承する．さらに，クラスAのスロットcに関して，特別な処理をしたければ，次のように定義すればよい．

```
(defmethod スロットc ((instance A)) (・・・ (run-super) ・・・) ) ……③
```

このとき，表2(a)のようなメソッド適用表が作られ，③を実行中に run-superが呼び出されたとき，メソッド優先順位リストの次の要素である②が起動される．

ここで，高速なアクセスを実行するために，メソッド適用表内部には，インスタンスベクタを直接参照するコードを置くようにした．すなわち，スロットbとスロットcのアクセッサメソッドのメソッド適用表は，表2(b)のように修正される．表2(b)の中のスロットcのメソッド適用表において，ベクタの参照要素が第一と第二と異なっていることに注意して欲しい．

表2　スロットアクセッサのメソッド適用表の例

(a)　修正前の適用表

スロットbのメソッド適用表

実引数のクラス	メソッド優先順位リスト	
	仮引数のクラス	ラムダ式
A	B	①の本体
B	B	①の本体

スロットcのメソッド適用表

実引数のクラス	メソッド優先順位リスト	
	仮引数のクラス	ラムダ式
A	A C	③の本体 ②の本体
C	C	②の本体

(b)　修正後の適用表

スロットbのメソッド適用表

実引数のクラス	メソッド優先順位リスト	
	仮引数のクラス	ラムダ式
A	—	ベクタの第一要素を取るコード
B	—	ベクタの第一要素を取るコード

スロットcのメソッド適用表

実引数のクラス	メソッド優先順位リスト	
	仮引数のクラス	ラムダ式
A	A —	③の本体 ベクタの第二要素を取るコード
C	—	ベクタの第一要素を取るコード

7．新しいメソッド決定アルゴリズム

適用メソッド決定のための高速アルゴリズムを考案し，現在インプリメントを検討している．本節では，そのアルゴリズムの概要を述べる．

7.1　アルゴリズムの背景

メソッドの優先順位はクラス優先順位から求められる．同名のセレクタを持つメソッドの優先順位が，図5のように逆転する場合は，クラス優先順位リスト｛B，D，E，t｝と｛C，E，D，t｝とのDとEのように，クラスに半順序関係が成立しない場合である．しかしながら，このようにクラス優先順位を逆転して定義することはまれであり，一般の場合は，同名のセレクタに対して，クラスの優先順序は半順序関係になっていると考えられる．

このようにクラスの優先順序が半順序関係になっているときには，あらかじめ，そのクラス優先順位に従って同名のセレクタを持つすべてのメソッドを半順序にならべておくことができ，その中を検索していくことにより実引数に適用可能なメソッドを優先順位に従って選択することが可能になる．このようにすると，実引数のクラスの組合せごとに，メソッド優先順位リストを作ることがないため，計算時間と記憶領域がともに節約される．次節では，このような場合のメソッド特定の高速アルゴリズムを述べる．

7.2　アルゴリズム

（前処理）

Step 1　すべてのクラスが，クラス優先順位に従って半順序に並んでいるかどうか判定する．半順序に並んでいないとき，このアルゴリズムは適用できないので，その旨出力する．

Step 2　半順序関係があるとき，半順序に関係づけられていないクラス間にも適当な順序をつけて，もとの半順序を保存したまま，すべてのクラスを全順序集合に並べる．この全順序関係を仮にクラス全順序関係と呼ぶ．

Step 3　クラス全順序関係にもとづく辞書式順序関係に従い，メソッドの仮引数のクラスをキーとして，同じセレクタ名Sで定義された全メソッドを並べる．セレクタ名Sのメソッドを並べた結果をMS ＝｛MS1，MS2，MS3，…，MSn｝とする

（メソッド呼出し処理）

Step 4　セレクタSが実引数を伴って呼び出されたときに，MS1から始めて，適用可能となる最初の MSiを見付け

る．適用可能性は，次のようにして判定する：すなわち，メソッドの各実引数に対して，実引数のクラスの親クラスの集合をとってくる．その集合の中にMSi の仮引数のクラスが含まれているかどうかチェックする．すべての実引数が MSiの仮引数のクラスに対して，このチェックを満足したならば， MSiは適用可能と判定される．

Step 5　適用可能なメソッド MSiが見付かったならば，MSi を実行する．MSi を実行中にrun-super が呼ばれたならば， MSi+1から，Step 4を繰り返す．見つからなければ，エラー処理を行う．

7.3　インプリメント上の工夫

実際のインプリメントにおいては，さらに高速化するためにいくつかの手法が考えられる．すなわち，

7.3.1　自然数の割当て

クラス全順序集合関係の要素である各クラスに，優先順位の高さを表わす自然数を割りあてる．この結果，MSのソート演算が自然数の比較で高速にできる．また，その自然数をもとに，クラスの親集合をビットベクタで表現することにより，適用可能性判定における集合演算が，ビット演算により高速化される．

7.3.2　決定木型のメソッドの配列

さらに，仮引数のクラスに従って，メソッド優先順位リストMS を決定木の形に変形し，実引数があるクラスに属しているかどうかの判定の繰返しを省くようにする．具体的に，例を用いて説明する．

例えば，図7のような三引数のメソッド定義と，図8のようなクラス優先順位が与えられたものとする．辞書式順序により，図9のようにMS が与えられる．

```
(defmethod  S2 ((X  A) (Y  A) (Z  B))  式1)
(defmethod  S2 ((X  A) (Y  B) (Z  A))  式2)
(defmethod  S2 ((X  B) (Y  A) (Z  B))  式3)
(defmethod  S2 ((X  B) (Y  B) (Z  A))  式4)
```

図7　三引数のメソッドの例

図8　クラス階層

MS ＝ { ((A A B) 式1)　((A B A) 式2)　((B A B) 式3)　((B B A) 式4) }

図9　MS の例

Step 1　まずメソッドの仮引数のクラスをもとに，図10のような決定木を構成する．決定木のノードは，クラス検査用のノードとメソッド起動用のノードとから成る．クラス検査用ノードは検査すべき引数の位置k，検査すべきクラスC，検査に成功した時の遷移先Ｎ１，失敗したときの遷移先Ｎ２の４つのデータを持つ．クラス検査用ノードに遷移したとき，ｋ番目の実引数の親クラス集合がクラスCを含んでいればノードＮ１に遷移し，含まなければノードＮ２に遷移する．メソッド起動用ノードは起動すべきメソッドMとrun-super 時に再開する際のノードNの二つのデータを持つ．このノードに遷移した時は，再開ノードＮを記録しておいて，メソッドMを起動する．run-super が指定された時には，記録しておいたノードNから，引数のクラス検査またはメソッド起動を再開する．εは遷移すべきノードが存在しないことを意味し，エラー処理を行う場合である．

Step 2　次に，図10の決定木をみながら，クラス判定の検査が省略可能な所のポインタをはりかえる．例えば，ノード①でクラス検査に失敗した後は，②に行っても失敗することがわかるから，①からのポインタを②から③へはりかえる．また，⑤で失敗したときには，①で第一引数がクラスＡに属することはわかっているから，②でのクラス検査は必ず成功する．それゆえ，その検査を省くことができるから⑤からのポインタを②から⑥へはりかえる．以下，同様にして図11のような決定木を構成することができる．この例の中で，最も劇的に変化するのは⑨から②へのポインタで，②→⑥→⑩→③→⑦→⑪→④→⑧→⑫→εの順にはりかえられる．

（メソッド起動時）

Step 3　メソッドＳ２が実引数A，A，Bで与えられたとき，決定木に従い，式１が起動される．実行中に run-superが呼び出されたときには，式３が起動される．

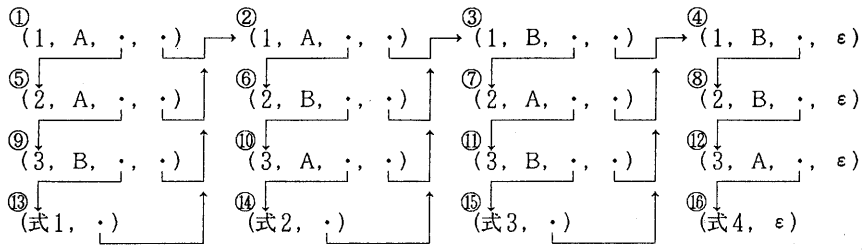

図10　適用メソッドの決定木の例

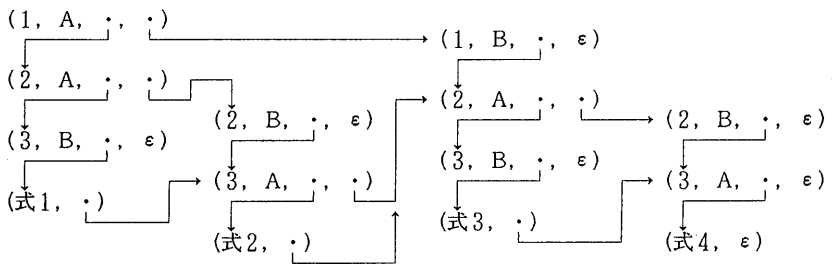

図11　最適化した適用メソッドの決定木の例

8．まとめ

CommonLoops の基本部分を UtiLisp上に実現したシステムとして， UtiLoops について述べた．UtiLoopsは，我々が現在構築中の知識表現システム，および図形表現システムの土台として用いられている．オブジェクト指向が図形表現に向くことは，確認できたが，多重メソッド等の機能の利点や欠点を実感できるほどまだ使いこなしていない．今後さらに処理の高速化としてコンパイラの作成，および必要な機能について研究を進めていく予定である．

謝辞

開発にあたり多大な貢献をしてくださった松木美保子嬢に深謝します．また，棚橋部長，林部長代理ならびに研究室諸兄の，日頃の御指導，御討論に感謝します．

文献

[1] D.G.Bobrow, K.Kahn, G.Kiczales, L.Masinter, M.Stefik and K.Zdybel: COMMONLOOPS, Pre-IJCAI'85 Draft, ISL-85-8, XEROX PARC, 12 AUGUST 1985.

[2] D.G.Bobrow, et al.: CommonLoops, Merging Lisp and Object-oriented Programming, OOPSLA'86, p.17-p.29, 1986.

[3] Common Lisp の動向に関する調査報告書，日本電子工業振興協会，1986.

[4] UTILISP 手引書，FACOM マニュアル，1985.

[5] 松木，泉，吉田，加藤：COMMONLOOPS の UtiLisp 上の実現，情報処理学会第33回全国大会講演論文集，p.479-p.480，1986.