Lisp のための新しいオブジェクト配置法†

湯浅 敬††

この論文は，我々が UtiLisp を 32 ビットプロセッサ MC68010/20 および VAX に移植するにあたって導入したオブジェクト配置ならびにタイプチェックの新しい手法について述べたものである．ダイナミックなタイプチェックを提供する Lisp では，そのスピードを上げることがシステム全体の性能を高めることにつながる．そのためには Lisp オブジェクトの構造や配置の方法という基本的なところからの設計が重要である．UtiLisp は元来スピードを重視し，これら基本的な部分の設計もきちんとなされていた．しかしながら今回我々がターゲットとした CPU のアーキテクチャは，旧システムが走っていた機械とは決定的な違いがあったため，基本部分の設計からやりなおす必要があった．

1. はじめに

UtiLisp 32[6] とは 32 ビットプロセッサ上で動く UtiLisp のことで，メインフレーム上で動いていたシステムを移植したものである．UtiLisp は元来スピードを重視したシステムであり，今回の移植にあたってもその点に留意した．

Lisp は動的なタイプチェック機能を持つ言語である．ヒープ内に配置された Lisp オブジェクトは，実行時にそのタイプが判別できなければならない．したがって，Lisp オブジェクトへのポインタとそのオブジェクトのタイプを表すタグを同時に管理するか，ポインタから何らかの方法でタイプを判別できるようにしておかなければならない．

タイプチェックは，Lisp インタプリタの根幹とも言えるエバリュエータの中で頻繁に行われるため，そのスピードが重視される．このタイプチェックの高速化がシステムの性能を決めると言っても過言ではない．

タイプチェックのコードをよくするためには，オブジェクトの配置法が重要になる．しかしメインフレームと今回移植の対象になった 32 ビット CPU の間にはアーキテクチャ上の違いがあり，旧システムの配置法をそのまま移植するのは不可能であった．そのため今回の移植では，言語仕様は同じにしたが，オブジェクト配置という根本の部分では新しい方法を導入した．

次の章では問題となっている旧システムでの配置法と，アーキテクチャの違い，3 章では 32 ビット CPU ですでに実現されている配置方法について述べる．4～7 章では我々のシステムで実現したオブジェクト配置法とタイプチェックについて説明し，8 章ではベンチマーク結果を基に我々のシステムの評価を行う．

2. 移植上の問題点

2.1 旧 UtiLisp のタイプチェック

元々の UtiLisp (以下旧 UtiLisp と呼ぶ)[1~4] は，ターゲットマシンとなったメインフレームや MC68000 の CPU アーキテクチャをうまく利用している．それは，レジスタは（アドレス専用のものも含めて）32 ビット幅であるが，アドレスバスは 24 ビット幅であるということである．メモリアクセスの時にアドレッシングされると，レジスタの上位 8 ビットは自動的に無視される．もちろんメモリ内では 4 バイト，すなわち 32 ビットのデータを保持できる．

旧 UtiLisp は，この上位 8 ビットをオブジェクトのタイプを表すタグとして利用した．これはポインタとそのタグを同時に持っているので，ポインタタグ方式と呼ばれる．ポインタで指されたオブジェクトのタイプを調べるには，レジスタの上位 8 ビットを見ればよい．ポインタとして内部を参照する際には下位 24 ビットが使われ，タグの部分は自動的にマスクされるのと同じことになる．

また旧 UtiLisp では，タグの割り当てでも工夫がなされている．リスト（コンスセル）には 0x80（2 進で 10000000）というタグを付け，シンボルやストリング，整数などのアトムオブジェクトには 0x80 未満，すなわち最上位ビットが 0 となるものを割り当てている．これによりタイプチェックのうち，リストかアトムかの判定 (**listp, atom** など) は，上位 8 ビットを抽出しなくてもポインタそのものの最上位ビットが 0

† A New Scheme of Object Allocation for Lisp Systems by KEI YUASA (Mathematical Engineering, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部計数工学科

か1か，すなわち符号を見ることによって実現できるようになっている．

2.2 32ビットマシンの問題点

旧システムは24ビットアドレスという「古い」アーキテクチャに依存するところが非常に大きい．現在主流となっているワークステーションのCPUはどれもみな32ビット幅のアドレスバスを持っているため，上位8ビットを無視してくれない．したがって旧 UtiLisp をそのまま移植することは不可能で，根本的にタイプチェックの方法，ひいてはオブジェクトの配置法から考えなおす必要がある．

この新しいオブジェクト配置の設計にあたっては，以下の点に留意した．

（1） リスト，シンボルのチェックは高速に行う

ポインタタグ方式が使えないことにより，ある程度効率が落ちることは避けられない．しかしながら，旧システムがリストには速いタイプチェックを提供していたように，重要なオブジェクトは速くチェックができるようにする．

リストとシンボルは，エバリュエータの中で特に重要な役割を果たす．なぜならインタプリタ下ではプログラムはリストで表されている．またシンボルは変数および関数名として使われる．この2つのオブジェクトに対してはよりよいタイプチェックを提供すればシステムとしての性能が上がるはずである．

（2） Fixnum はポインタの中に持つ

小さな整数 (Fixnum) は，ヒープの中に配置してそこへのポインタで保持するのではなく，直接ポインタの中にコーディングする．これは，小さくて同じ数を表すオブジェクトが大量にヒープを占有するのを避けることと，メモリアクセスを避けて数値演算を高速に実行できるようにするという意図がある．

3. ポインタタグに代わる方法

今回我々がターゲットとしている，32ビットすべてをアドレッシングするようなCPUの上でも，過去にいくつかの Lisp が作られている．この章では，それらのシステムで，どのようなオブジェクト配置・タイプチェックの方法を採っているかを紹介する．

3.1 タグのマスキング

この方法の基本的考え方はポインタタグ方式と同じである．アドレス幅が32ビットになったといっても，必ずしもそのような大きな領域を使う必要はない．またたいていの OS では 32 ビットものアドレス空間の使用を認めてくれない．したがってアドレス 32 ビットの内，上位の何ビットかは無意味である．そこでこの上位何ビットかをポインタタグとして使うことができる．

よく見られるのが5ビットと 27 ビットに分け，上位5ビットをタグ，下位 27 ビットをアドレスとする方式である．5ビットあれば 32 種類のタグを作ることができ，通常の Lisp システムであれば十分な幅である．またヒープも 27 ビット（最大で 128 M バイト）あれば十分な大きさである．

この方法の欠点は，ポインタをアドレッシングする際に，上位のタグ部をソフト的にマスクしなければならないことである．論理 AND 命令1つであるが，アドレッシングの際には毎回必要になる．また MC68000 のような CPU ではアドレスレジスタでは論理 AND 命令が実行できないので大きな手間となる．またタイプチェックの場合にも，上位にある5ビットをなんらかの方法で抽出しなければならない．

下位2ビットをタグとするシステムもある．バイトアドレスを用いている機械に 32 ビットのオブジェクトを並べると，それらはすべて下位2ビットが0になる4バイトバウンダリに並ぶ．そのためポインタの下位2ビットは無意味なので，タグに使おうというものである．ただしこの手法では，タグで4種類のものしか区別できない．またアドレッシングについても，マスクは必要ないが，インデスクを用いて4バイトバウンダリに戻してやらなければならない．

3.2 オブジェクトタグ

32ビットレジスタの中に，ポインタとタグの2つの情報を持つのが困難なため，レジスタにはポインタのみを持たせる．タイプタグはヒープ内に配置されたオブジェクト自身が持つようにする．

この方法では，アドレッシングの際にはマスキングが必要ないが，タイプチェックの際のコストが大きい．すなわちポインタのアドレスだけからはタイプを知ることができないため，必ず1回オブジェクトの先頭に書かれたタグにアクセスしなければならない．これは非常に大きなコストとなる．またメモリ効率のことを考えても，すべてのオブジェクトがタグを持つため，必要以上にヒープを占めてしまうという欠点がある．

3.3 ヒープ分割

ヒープをオブジェクトのタイプによって分割する方式で，レジスタにはアドレスそのものが入るために，

アドレッシング時のマスキングは不用である．またタイプチェックは，分割された境界のアドレスとの比較（高々2回）によってできるので，メモリアクセスは必要ない．

この方式の問題点はそれぞれのオブジェクトに割り振るヒープの大きさが，予測不可能ということである．どのタイプのオブジェクトがよく使われるかということは，プログラムによってまちまちである．しかしながら，リストを大量に消費するプログラムに対して，リスト領域を十分割り当てないと，ガーベージコレクションが頻繁に起こり効率が落ちてしまう．かと言って状況に応じてヒープの境界を動的に動かすということは非常に困難なことである．

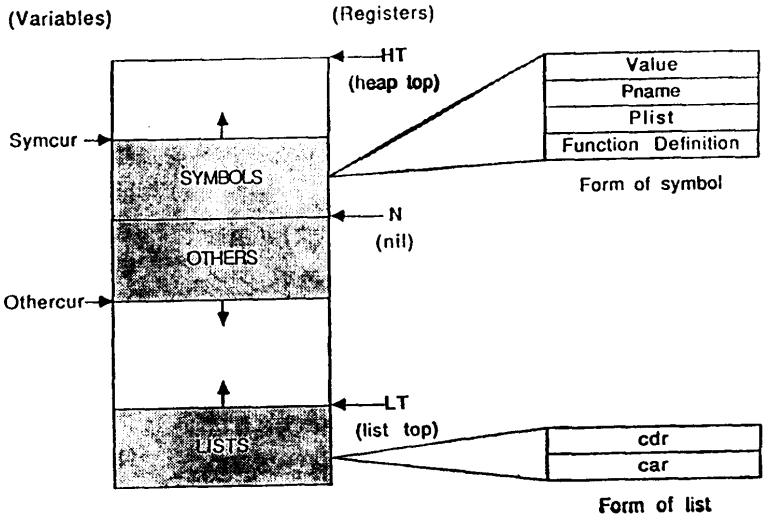

図 1 ヒープエリアの構造
Fig. 1 Heap area.

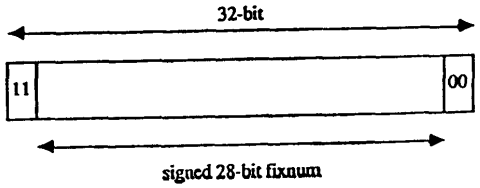

図 2 整数の構造
Fig. 2 Pointer coded fixnum.

4. UtiLisp 32 におけるヒープ分割

我々が UtiLisp 32 で用いた方法は，オブジェクトタグと，ヒープ分割の混合方式である．基本的方針は先にも述べたように，よく用いられるリストとシンボル，それに Fixnum にはよいタイプチェックを提供し，エバリュエータの効率を上げるということである．

UtiLisp 32 のヒープは**図 1** のように3つに分割される．1番上（アドレスの若い方）がシンボル領域，下がリスト領域，中央がその他のオブジェクトが入る領域である．Fixnum はポインタの中にコーディングするが，その印として上位2ビットを1にして持っている．また後で述べるように上位2ビットを別の用途でも用いるため，ヒープへのポインタは上位2ビットが0になるようにする．すなわちヒープは 0x40000000 番地を越えてはならないという制限がある．

5. タイプチェックとオブジェクト操作

さて前章のように各 Lisp オブジェクトが配置されるわけであるが，この章では問題のタイプチェックがどうなるか，および整数などに対する操作がどうなるかについて述べる．

5.1 Fixnum

Fixnum とはポインタ内にコーディングできる小さな整数を意味する．32ビットのうち，上位2ビットは常に1であり，これによってそれがポインタでなく実際のデータであることを表す．またガーベージコレクションの都合上すべてのオブジェクトの下位2ビットも0に固定してあるため，Fixnum は32ビットのうち中央の28ビットを占める．**図 2** に整数の構造を示す．

Fixnum の最上位ビットは常に1であり，ヒープ内に配置されたオブジェクトへのポインタの最上位ビットは0である．したがって，Fixnum のタイプチェックはこの最上位ビットが1であることを確かめればよい．レジスタの最上位ビットは符号を表すので，レジスタが負であれば Fixnum，正ならばそれ以外のオブジェクトということになる．レジスタの符号は，レジスタ転送命令あるいはテスト命令1回でできる．

Fixnum は実体がレジスタの中央に位置しているため，数値演算には機械語の演算命令をそのまま使うことはできない．図2にあるように符号は第 29 ビットにある．そのため加減演算を行うにはまず左に2ビットシフトしてから行う．演算の結果オーバフローなどが起これば，28ビットに収まらないということである．その場合は Bignum というヒープに配置するオブ

ジェクトへの変換が行われる．また乗除算についてもそれぞれ適当なシフト操作が必要になる．

5.2 リストとシンボル

ヒープ分割によって，リストとシンボルについてはポインタのアドレス比較によってそのタイプがチェックできる．

シンボルは3つに分割されたヒープの1番上の領域に配置される．そのためポインタのアドレスが，シンボル領域とその他領域の境界よりも小さければシンボルであることがチェックできる．この時ポインタと境界との比較を符号なしで行う．すると最上位ビットが1になっている整数は，負の数ではなく正の大きな数とみなされるので，問題の境界よりも大きい．すなわちシンボルではない側にはいる．

リストはヒープの下部に配置されるので，リスト領域とその他領域の境界との比較をすればよい．今度は符号付きの比較をする．すると整数は負の数とみなされるので，境界よりも小さい方になる．

このように比較を符号付きか符号なしかを使い分けることによって，シンボルもリストも比較1回でタイプチェックができる．シンボルもリストも，それぞれの領域の底から配置され上に向かって延びていく．シンボル領域の底，すなわち1番最初には nil が配置されており，レジスタの1つがこれを常に指している．シンボル領域とその他領域の境界には，このレジスタが使われる．またリスト領域は現在配置されているリストの1番上にあるものを，別のレジスタが指していて，これがその他領域との境界として用いられる．結局シンボルとリストに関しては，レジスタ同士の比較1回でタイプチェックが可能である．

5.3 その他のオブジェクト

その他の領域に配置されるオブジェクトは，Bignum (無限長の整数)，Flonum (64ビット実数)，文字列，ストリーム，コードピースそれにベクタである．これらは1つの領域に入れられるので，オブジェクトタグを付けて区別する．また Flonum 以外はオブジェクトの長さが不定であるため，そのサイズも記入しておくことが必要である．そこでこの領域のオブジェクトは (Flonum も含めて)，最初に4バイトのオブジェクトタグ，次にオブジェクトのサイズを持っている．図3に例として文字列の構造を示す．

オブジェクトタグの最初の2ビットはすべて10というパターンになっている．これは UtiLisp では，どんな Lisp オブジェクトにも出てこないパターンである．したがってオブジェクトの最初のセルを参照して，その上位2ビットが 10 であれば，それは必ずオブジェクトタグである．シンボルやリストは最初のセルは通常の Lisp オブジェクトが入っている．したがってその他領域にあるものをチェックする場合は，ポインタがこの領域に入っていることは確認しなくて，いきなり先頭のセルとタグとを比較すればよい．ただし最上位ビットが立っている Fixnum をポインタとしてアドレッシングすると，メモリマネジメントエラーを越こすので，事前に Fixnum でないことはチェックする必要がある．

図4は各オブジェクトに対するタイプチェックの MC 68020 用のコードである．どれもチェックすべきオブジェクトはレジスタ A (a5) に入っており，チェックの結果それぞれのオブジェクトでない場合はエラーにジャンプするようになっている．N (d7) はシンボル領域の底にある nil，LT (d5) はリスト領域の先頭を指しているレジスタである．

6. スタックフレーム

UtiLisp 32では1つのスタックを，関数呼び出しおよび OS とのインタフェースに用いている．関数呼び出しの際には，スタックフレームが1つ作られる．ここには局所変数，帰り番地，コードベース，前のスタックフレームへのリンクそれにラムダ変数の束縛が入る．

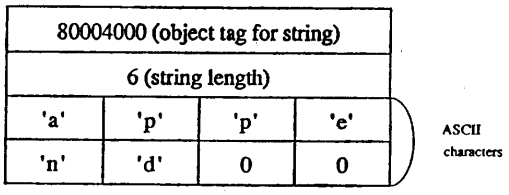

図3 文字列の構造
Fig. 3 String.

```
1. Fixnum
   movl A, DWW
   jpl      typerr
(DWW is a work register)
2. Symbol
   cmpl A, N
   jcs      typerr
(N points nil)
3. List
   cmpl A, LT
   Jgt      typerr
    (LT is list top)
4. Others (String)
   cmpl A, LT
   jne      typerr
   cmpl   $0x80004000, A@
   jne      typerr
($0x80004000 is a pointer tag for string)
```

図4 各オブジェクトに対するタイプチェック
Fig. 4 Type checkings.

UtiLisp ではスピードを重視するため，浅い束縛を行う．そのためラムダ束縛は以下の手順で行われる．まず束縛されるシンボルと，そこに束縛されていた古い値をスタックに積み，シンボルの方に印を付けておく．そしてそのシンボルにはあらたに束縛する値を入れる．束縛を解く時にはスタックをスキャンしてこの印を探し，見つかったらそのシンボルに古い値を入れる．

この束縛シンボルの印付けは，最上位ビットを1にすることで行う．スタック中には Fixnum も積まれることがあるが，これとの区別は第2ビットで見る．Fixnum は第2ビットも1であるので，束縛シンボルのほうは第2ビットは0でなければならない．そのため，シンボルは元々この第2ビットが1であると元に戻すことができなくなるため，シンボルは 0x40000000 番地よりも若い番地になければならない．

なおその他領域にあるオブジェクトの先頭にあるオブジェクトタグも，先頭が 10 で始まっているが，これが直接スタックに現れることはないので問題ない（**図5**）．

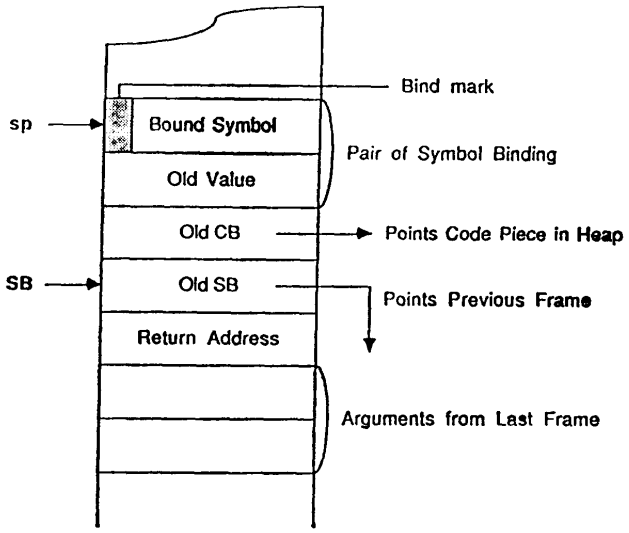

図 5 スタックフレームの構造
Fig. 5 Stack frame.

7. ガーベージコレクション

UtiLisp 32 のガーベージコレクタ（以下 GC と略す）はマーキング・コンパクション方式を採る．これはヒープに新しいオブジェクトを配置できなくなった時に起動されるが，陽に ***gc*** という関数で呼び出すこともできる．ヒープがなくなるのはシンボル領域がなくなる場合と，その他領域とリスト領域が衝突する場合の2通りがある．いずれの場合にも GC はヒープ内のすべてのゴミを回収する．

GC は最初にマーキングを行うが，これはインタプリタ内の「ルート」と呼ばれるポインタ群，およびスタックからたどれるオブジェクトすべてにマークを付ける．マークはポインタの最下位ビットに付ける．その際オブジェクト内でマークを始めたセルを記録するために，もう1ビット（ストップビット）必要になる．よって GC では下位2ビットを用いる．

すべてのゴミでないオブジェクトにマーキングした後，コンパクションを行う．ヒープは3つの領域があるが，シンボルとリストは下に，その他オブジェクトは上に有効なオブジェクトをコンパクションする．このコンパクションでは，シンボルとリスト領域には Morris[8] のアルゴリズム，その他領域には Knuth[7] のアルゴリズムを適用している．

8. 性能評価

この章ではいくつかのベンチマークの結果を基に，UtiLisp 32 の性能を評価する．測定は SUN ワークステーションと VAX の上で行われた．また他のシステムとの比較をするために，FRANZ LISP, KCL (Kyoto Common Lisp), FRANZ Common Lisp での測定も行った．ちなみに KCL ではオブジェクトタグ，FRANZ ではポインタタグのマスクを行っている．

ベンチマークプログラムは tarai, tak takl, ctak, boyer の5つである[5]．boyer は定理証明をするプログラムで，リスト処理やプロパティリストの処理が多く，この面での性能が測れる．他の4つは再帰呼び出しを多用したプログラムで，インタプリタ上では関数の呼び出しとリターンの速度，コンパイラでは再帰呼び出しの最適化などが問題になる．その他に tarai と tak では整数操作，takl ではリスト操作，ctak では catch-throw のスピードが測れる．

表1がベンチマークの結果である．まずインタプリタ上での結果では UtiLisp 32 は他のシステムにかなりの差をつけている．これは我々のシステムがインタプリタでの性能を特に重視したためである．Common Lisp 系の処理系ではコンパイラ重視となり，特に静的スコープを用いているためインタプリタはそれほど速度を上げることは難しい．

表 1 ベンチマーク結果
単位秒 (インタプリタ／コンパイラ)
GC 時間は除く

Table 1 Benchmarks.
Elapsed times are in seconds (Interpreter/Compiled code).
GC times are excluded.

		tarai	tak	takl	ctak	boyer
SUN 2/120	UtiLisp	117/13.5	23.8/2.38	182/14.9	40.1/9.15	313/63.2
SUN 3/52	UtiLisp	53.0/4.70	10.8/0.883	82.1/5.75	17.8/4.00	131/28.2
	FranzLisp	243/86.1	52.3/15.0	434/17.9	——	713/141
	Franz CL	1400/6.40	279/1.20	2260/5.90	348/3.00	—/38.0
SUN 3/260	UtiLisp	21.5/2.10	4.38/0.383	34.9/2.47	7.80/1.82	57.4/11.2
	FranzLisp	95.4/30.6	21.0/5.70	174/7.07	——	279/39.1
	Franz CL	581/2.57	113/0.467	956/2.73	159/1.20	—/14.6
VAX 8600	UtiLisp	23.8/2.79	4.73/0.417	39.2/2.70	8.87/1.98	61.8/12.5
	FranzLisp	81.3/31.6	15.7/4.36	150/6.02	——	270/45.6
	Franz CL	722/2.86	122/0.500	1130/2.68	122/1.02	—/9.98
	Kyoto CL	393/8.38	78.1/1.63	1570/3.33	134/15.9	——

コンパイルされたコードで比較して見ると，ほとんどのテストで他のシステムに勝っている．これは我々のオブジェクト配置やタイプチェックの設計が成功したことの現れと言えるであろう．VAX の結果はあまりよくない．これは VAX 上のコンパイラのチューン不足と言える．これにはまだ向上の余地がある．

ctak の結果は決してよいとは言えない．これは UtiLisp 32 ではスタックフレームに catch 用のタグを入れる場所を作っていないためである．catch を実行すると，専用の 32 ビットの catch 用パターンと，タグをスタックに積んでいる．throw 時にはスタックをスキャンしてこのパターンを探すということを行っている．このため ctak のようなベンチマークは遅くなる．スタックに catch 用タグを設ければ，ctak の結果はよくなるが，通常の関数呼び出しが犠牲になる．我々は catch-throw は緊急の場面で用いられることが多いと考え，この方式は採らなかった．

またこれとは別に，旧システムが動く MC 68000 マシン上に，新しいオブジェクト配置法でインタプリタのみを試作した．結果は tarai が旧システムで 104 に対して 108 (単位秒) であった．Fixnum 演算でロスがあるが，リスト，シンボル，Fixnum の 3 つのオブジェクトを扱ったこのプログラムでは，旧システムの性能がほとんど落ちていないことがわかった．

9. 結　論

我々は 32 ビットのアドレスバスを持つ CPU 上に高速な Lisp システムを構成するために，新しいオブジェクトの配置方法とタイプチェックの機能を導入した．このシステムでは，オブジェクトのタイプによって異なるチェック方法が用いられる．すなわちエバリュエータ内で頻繁にタイプチェックが行われるシンボルとリストに対しては，よいタイプチェックを与える．また 28 ビット以内の整数については，ポインタの中にコーディングし，速く操作ができるようにした．

ベンチマークの結果から，同一機械上の他の Lisp システムよりも性能が高いことがわかり，我々の目的は達成できたと考える．

なお，UtiLisp 32 は現在 SUN 2, SUN 3, VAX といった UNIX マシンの上で動いており，PROLOG/KR のような旧 UtiLisp 上で走っていたアプリケーションも動かすことができる．

謝辞 著者の指導教官である和田英一教授に感謝致します．また UtiLisp 32 の開発にあたっては，寺田実，岩崎英哉，金子敬一，高橋俊成，石井信の各氏の協力があった．ここに併せて感謝します．

参 考 文 献

1) 近山　隆：UtiLisp システムの開発，情報処理学会論文誌，Vol. 24, No. 5, pp. 599-604 (1983).
2) 和田英一，富岡　豊：UtiLisp の 68000 への移植，情報処理学会記号処理研究会報告，84-29 (1984. 1).
3) 湯浅　敬，金子敬一：UtiLisp の Macintosh への移植，その苦難の道のり，第 27 回プログラミングシンポジウム，情報処理学会 (1986. 1).

4) Chikayama, T.: UtiLisp Manual, Technical Reports METR 81-6, Department of Engineering and Instrumental Physics, Faculty of Engineering, University of Tokyo (Sept. 1981).
5) Gabriel, R. P.: Performance and Evaluation of Lisp Systems, Research Reports and Notes, The MIT Press (1985).
6) Kaneko, K. and Yuasa, K.: A New Implementation Technique for the UtiLisp System, Reprints of WGSYM Meeting, *IPS Japan*, 41-7 (Jul. 1987).
7) Knuth, D. E.: *The Art of Computer Programming*, Vol. 1, Addison-Wesley (1969).
8) Morris, F. L.: A Time- and Space-Efficient Garbage Compaction Algorithm, *CACM*, Vol. 21, No. 8, pp. 662-665 (Aug. 1978).

(昭和63年9月13日受付)
(平成元年4月11日採録)

湯浅　敬（正会員）

昭和36年生．昭和59年東京大学工学部計数工学科卒業．昭和61年同大学院修士課程修了．平成元年同大学院博士課程修了．現在松下電器産業東京研究所．プログラム言語処理系，記号処理，マルチプロセッシングなどに従事．ACM 会員．