TAO における代入計算機構

日本電信電話公社 武蔵野電気通信研究所
大里延康　竹内郁雄　奥乃 博

1. はじめに

我々は，Lispマシン ELIS[1] 上に，高性能の知能的プログラミング環境 NUE（New Unified Environment）を構築する研究を進めている．NUEの核となる言語 TAOは，S式をベースに，Lisp の機能（逐次型の実行を主体とする言語），Prolog の機能（ユニフィケーションとバックトラックを主体とする論理型言語），Smalltalk の機能（メッセージ授受を主体とするオブジェクト指向言語）を，プログラミングを遂行するための機能素材として含んだ統合化言語である．[2] これら3つの中心的な言語機能のほか，TAOは，アドレスを主体とする Fortran型の計算機構をも具備している．

本報告では，Lispとして見たときのTAOの機能のうち，代入についての考え方と，その機能の実現法を，ELISのマイクロプログラムを活用した実装技術の観点から論ずる．「代入」をLispの側面として述べる理由は以下のとおりである．論理型言語における「代入」は，ユニフィケーションに伴う論理変数の具体化（インスタンシエーション）によって起こるので，所謂従来的な意味での代入ではないと考えられること．オブジェクト指向におけるインスタンス変数への代入操作は，インスタンスへのメッセージ伝達の結果起こるものであるので，これも通常の意味での代入ではないこと．もちろん，メソッドの中でのインスタンス変数の代入は可能だが，その形式は Lisp の代入式と同じである（すなわち，インスタンス変数へのメッセージ伝達ではない —— インスタンス変数それ自身はオブジェクトではない．このことは Smalltalkでも同じである[3]）．

TAOは，他のどの Lispよりも一般的な代入式をもっている．一般に，Lispの代入の書法は，型宣言を基本とする Algol系言語に比べて劣る．Algol系言語では代入がすべて

左辺値 := 右辺値

と書けるのに，Lispでは左辺値の型に応じて代入関数名が異なるのがふつうである．Zetalisp[4] などでは，左辺値の種類によって，それに対応する代入形式をつくり出して実行する setf というマクロを使用し，代入計算の内部的不統一を表層でとりつくろうことが行なわれている．たとえば，

```
(setf a 3)              =>    (setq a 3)
(setf (aref q 2) 56)    =>    (aset 56 q 2)
(setf (cadr w) x)       =>    (rplaca (cdr w) x)
```

などであるが，使用上の制限が強い欠点がある．たとえば，setf の返す値は，生成される式の返す値によって変わる．このため，setq からの類推で，setf の右辺値が値になるというつもりでいると虫になる可能性がある．（Zetalisp では，上記の第3例，すなわち rplacaの形になるものは，setcar という内部的な関数を用いる式に展開し，一応この問題を避けているが，一般の setf の結果については保証していない．）すなわち，ユーザは setf の展開形について意識していることが必要である．このほかにも，マクロ展開であるがゆえの制限（あるいは意図しない結果）が起こりうるという問題がある．

②. 代入式の統一

TAOは，ELISの水平型マイクロプログラミングというハードウェアの特徴を生かして，変数や配列への代入のみならず，リストの書き換え(rplaca, rplacd)などをも，単一の計算機構で実現する汎用代入式を導入した．この汎用代入式は上記のsetfと同様，言語の表層レベルで代入の形式を統一し，さらに，その実現機構においても統一的な手法によっており，setfのようなマクロ展開ではないことから，上述のような欠点は取り除かれている．

2-1. TAOの代入式

TAOの代入は，次の形をしている．

(:α β)

ここで，"(:"は1組の記号であって，αが左辺値，βが右辺値である．この式は，"assign"というタグで標識づけられたリストである(図1).*

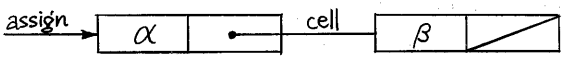


図1. TAOの代入式

TAOのインタープリタ(eval)は，"assign"のタグが立ったリストを見ると，それを「代入式」として処理する．なお，この式はデータとして見ると，タグが異なること以外，(α β)と同じで，carもcdrもとれる．

代入式の使用例を次に示す．

```
(:x 123)                                  変数xに123を代入.
(:(nthv n v) (fn x y))                    ベクトルvの第n要素に代入.
(:(table i j k) 1)                        配列tableの第i,j,k要素に1を代入.**
(:(car x) value)                          =(rplaca x value)ただし値はvalue.
(:(member n y) (+ n 1))                   リストyの中の最初のnを1増やす.
(:(cond ((null flip) (car x)) (t (cdr x))) flop)   flipがnilならxのcarを,
                                          それ以外ならxのcdrをflopで置き換える.
```

これらの使用例からわかるように，TAOの代入式は，およそ「場所」をアクセスしたと考えられるときには，いつでも使用可能と考えてよい．

2-2. 代入レジスタ

TAOの代入式の実現機構を以下に述べる．(:α β)の評価機構は以下のとおりである．

(i) まず，αを「被代入評価」する．被代入評価は，αが変数のときのみ，特別の処理をする．特別の処理とは，「代入レジスタ」と呼ばれる2つのレジスタ

---

*) TAOのタグは，データそのものに付随しているのではなく，そのデータをさしているポインタの方に付随している．(所謂オブジェクト・タグではなく，ポインタ・タグ.) 図では assign ⟶ のように表わす．以下同じ．

**) TAOの配列はapplobj (applicable object ── 関数と考えてよい)の一種である．上式の場合，tableという名前の変数が3次元の配列へのポインタを押さえている．i, j, kは，その配列applobjへの引数である．こうすることによって，見かけ上，ふつうの意味で使われる配列表現に近い形式が得られる．

の設定である．αが変数のときは，その計算の環境においてαのさしている場所（prog 変数や関数の仮引数などの場合はスタック上の場所，大域変数であれば，識別子（アトムのこと．TAOでは id と呼ぶ）の中に設けられた大域値用のスロット，carやcdrであればそのセルのアドレス，など）を求め，その場所の情報を代入レジスタに格納する．αが変数以外（リスト）のときは，通常の evalと全く同じ動作をする．ただし，この evalと同じ動作の中のどこかで代入レジスタの設定が行なわれていることが期待されている．被代入評価の中で代入レジスタの設定が正当に行なわれなかったときは，代入不可能という旨のエラーを起こす．

(ii) βを，ふつうの意味で評価する．

(iii) βの値を，αの被代入評価の中で設定された代入レジスタのさしている場所に格納する．

なお，上記(i)において，被代入評価中に代入レジスタの設定が何度か起こる可能性もある．この場合，最後に行なわれた設定が有効である．

代入レジスタは，CPUのもつ汎用レジスタ（全部で32個ある）のうちの2個を固定的に割り当てて使用している．2個のレジスタは次のように使用される．

rpr (replace register) --------- r20 を使用

代入すべき場所をアドレスで示す．そのアドレスは，メモリ上のアドレスであることもスタック上のアドレスであることも，あるいは，ビット配列の，物理的な場所であることもある．

rpf (replace flag) ---------- r19 を使用

代入場所が何であるかという性質を表わすコードが格納される．たとえば，car部，cdr部，locative bit（後述）のオフセットなど．rpf使用法を表1に示す．rprの値をどう解釈するかは，このrpfによって決定される．

表1. rpf のコード
（rpfの下位4ビット）

コード	意味
0	エラー（代入不可能）
1	cell または その類似品 (car)
2	cell または その類似品 (cdr)
3	stack location
4	1 bit locative
5	2 bit locative
6	4 bit locative
7	8 bit locative
8	16 bit locative
9	32 bit locative
10	64 bit locative
11	property
12	id's global value

代入レジスタの設定は，ほとんどの場合，メモリアクセスの陰のサイクルで行なわれるので，実行時のオーバヘッドは全くない．たとえば，αが（cdr x）だったとすると，2つの代入レジスタは，xのcdrを読み出すためのメモリサイクルの間に並行的に設定される．ただし，この場合のxのcdrの値そのものは無駄である．なお，rpfをrprのタグ部に埋め込むなどにより，代入レジスタを1個にすることも可能ではあるが，carやcdrなどの基本関数の効率を保持する必要性から，2個のままにしている．（2つの情報をマージするためのステップを入れる余裕はない．）

代入レジスタの設定は動的に行なわれる．したがって，代入レジスタの値を変えない関数を外側に重ねても代入が行なえることに注意しよう．なお，ふつうのsetqはマクロでTAOの代入式に展開される．

2-3. 自己代入

代入レジスタを導入したことにより，一旦計算した場所をそのまま保持しておくことが可能となった．これを用いると，左辺値を2回評価する無駄をしない代

入が実現できる．これを TAO では自己代入と呼ぶ．自己代入式は，

```
(::fn ... :α ...)
```

と書く．ここでαは，右辺値の計算式の中にも現われる左辺値の計算式で，fnの式の入れ子の中に入っていてもよい．自己代入式を評価すると，αが副作用をもたなければ，

```
(:α (fn ... α ...))
```

と同じ意味になる．たとえば，

```
(::+ :(table i j k) n)
```

は，配列 tableの第 i, j, k要素にnを足す．これは，

```
(:(table i j k) (+ (table i j k) n))
```

に比べて明らかに簡便であるのみならず，配列のインデックス計算や境界チェックなどの計算が1度だけしか行なわれないので高速である．

自己代入式も，通常の代入式と同様，“self-assign”というタグの立ったリストである（図2）．代入の対象となる左辺値αは，“colon”というタグで標識づけられている．自己代入式の中には複数の colonが存在しうる．そして，その評価のたびごとに代入レジスタが設定される．したがって，最後に評価された colon が代入の起こる場所を決定する．

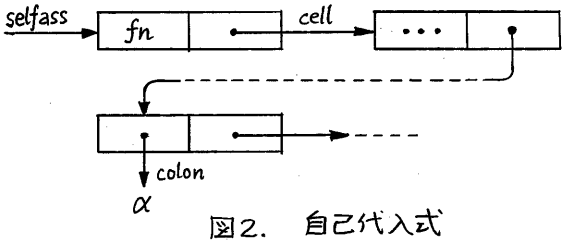


図2. 自己代入式

自己代入の実現においては，代入レジスタ rpr と rpf の設定には注意が必要である．自己代入式自身は通常の Lispの式として評価されるのであるから，この式が rpr，rpf の設定を期待するということは，別の情報として記憶しておかなければならない．しかも，この rpr，rpf は，ひとつの自己代入式に対して唯1組対応する．そこで，自己代入式に対応する rpr，rpf 用の保持スロットをスタック上に動的に設定し，その場所を rpsp（replace stack pointer）というレジスタ（汎用レジスタのうち r18 を固定的に使用している）で押さえておき，colon 式を評価するたびにその都度 rpr と rpf をそのスロットに格納するようにしている．すなわち，自己代入式の評価手続きは以下のとおりである．

(i) 現在の rpsp をスタックに保存する（自己代入式が入れ子になったときの対策）．しかるのち，rpf，rpr の2個分の空プッシュをし，そのときのスタックポインタの値を rpsp に格納する．

(ii) 関数 fn の実行のためのフレームを作る．（システム定義関数以外）

(iii) fn の実行環境において，colon が現われたらその式を被代入評価し，rpr と rpf を，rpsp によって示される スタックのスロットに格納する．

(iv) fn の出口で，fn の式の値自体を，rpsp の示す rpr，rpf で指定された場所（この自己代入式における左辺値）に格納する．rpr，rpf のスロットはポップされ，もとの rpsp が回復される．

(v) 自己代入式自身の値として，fn の値を返す．

2-2 および 2-3 で述べた代入式および自己代入式の評価は，すべて ELIS のマイクロプログラムで実現されている．

3. Locative pointer への代入

3-1. Locative pointer

TAOでは，メモリのアドレスに対応する locative という一群のデータ型を提供している．このデータ型によって，Fortran 風の計算機構が実現できる．アドレスを直接押さえているので，そのアドレスに対するデータの格納，計算が行なえ，ゴミ回収の絶対起きない計算が実現できることになる．表2に，TAOで提供している locative 型の一覧を示す．

表2. Locative 一覧

データ型	説明
64 bit unsigned-integer-locative	64ビット full word 符号なし整数
64 bit signed-integer-locative	64ビット full word 符号つき整数
64 bit floating-point-number-locative	64ビット full word 浮動小数点数
locative bit (locbit)	ビットデータ・メモリブロック内ポインタ型 *
1bit, 2bit, 4bit, 8bit, 16bit, 32bit, 64bit	

このような locative pointer 型を導入すると，それへの代入には若干混乱が起こりうる．すなわち，このような locative 自身はLispオブジェクトであるので，それ自身が変数の値であったり配列の内容であったり，あるいはリストの car 部に存在したりすることになる．したがって，Lisp変数などに locative を代入することと，Lisp変数などのさしている locative からさされているデータを書き換えることとをはっきり区別する必要がある．

図3-aは，8ビットの locbit をLisp変数xが保持しているようすを示している．「xへの代入」は，Lisp変数xへの代入であり，

(:x value)

と書かれる．これを実行すると当然，xはもとの locbit とは縁が切れ，単に value をもつものになる（図3-b）．

図3-aのxがさしている locbit のビットデータ（斜線の部分）を変更する代入も，特別な関数を用いないで，「自然である」と感じられる式で書けることが望ましい．TAOでは，このことを2とおりの方法で書くことができる．

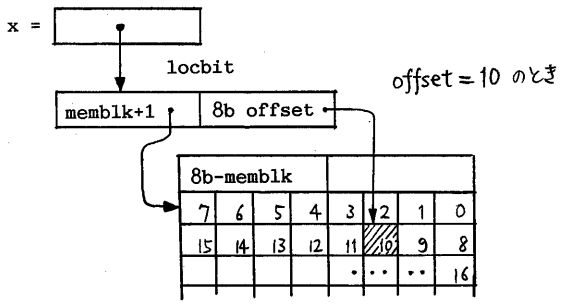


図3-a Locbit

(:x value)

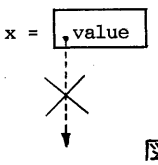


図3-b Lisp変数xへの代入

3-1-1. ":=" による代入

TAOでは，locbit はメッセージを受け取るオブジェクトである．TAOにおけるメッセー

---

*) ビットデータの locative を locbit と呼ぶ．Locbit は，ビットデータの連続するメモリ領域（locbit memblk）の内部を，先頭アドレスとオフセットで示す．

ジ授受の式は，

```
(receiver message args ...)
```

の形をしている[5]．そこで，locbit は := というメッセージを1つの引数とともに受け取ることとし，その結果，自分のさしているデータの変更を行なうということにした．引数が，新しくセットされる値である．したがって，

```
(x := value)
```

は，図3-cのような結果をもたらす．この式は，通常の Algol 系言語の代入式と似た形をしており，locbit のデータを書き換える代入式に関する限り自然な書き方と言えよう．

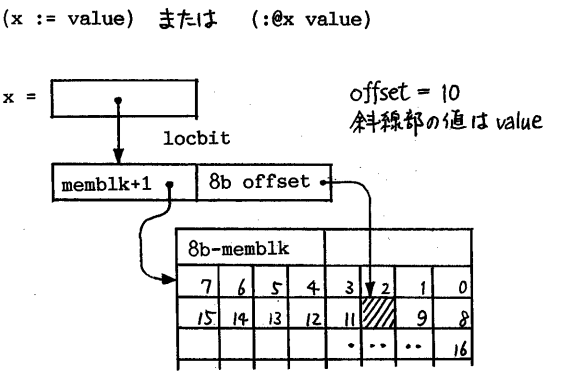


図3-c Locbit への代入

Locative integer など64ビット full word の locative への代入も同様である．たとえば，x を signed-integer-locative とする．x のさす locative への代入は，

```
(x := 100)
```

のように行なえる．ただし，locative integer および locative floating への := による代入式の右辺値は，3-2において述べるように，若干特殊な解釈をする．

3-1-2. ":"代入式による代入

Locbit のさす場所も，前に述べた代入式における一般的な左辺値の一種になりうるのが自然である．そこで，locbit に対する代入レジスタの設定（表1 参照）を行なうことによって，代入式による代入を行なえるようにしている．

関数 deref は，引数として locbit をとり，そのビットデータを数として返すとともに，rpr, rpf を設定する．より正確に言うと，

(i) rpf が locbit に設定されていたら，その rpf によって示された場所のデータを数として返す．rpr, rpf は破壊しない．

(ii) (i)以外のとき，引数が locbit ならばそのビットデータを数として返し，rpr, rpf をその locbit に設定する．それ以外のときはエラーとなる．

なお，(deref x) は，読み込みマクロによって @x と書ける．したがって，x が図3-a の状態にあるときは，

```
(:@x value)
```

によって，図3-cのような結果がもたらされる．上記(i)のように，deref の仕様が，あらかじめ設定された rpf を見る理由は後述する（3-3）．

3-2. Locative integer/floating への ":=" による代入式の右辺値

3種の 64 bit locative number への := による代入式は Fortran 風の計算を可能にする．これらの locative は := というメッセージを受け取ると，その引数のS式を（優先順位なしの）算術演算式として解釈し実行する．詳しく言うと，

(i) 最初にその式を評価したときに，右辺の式を Poland 記法に変換し，その式を，仮想的なスタック・マシンによって解釈実行するマイクロルーチンに渡す関数（64pol）への引数として与える式にマクロ展開する．Poland 記法の式は，演算を表わすトークン（opr ── operator の意）と，それへの引数からなるリストで

ある．トークンは，タグ opr をもつ内部的な特殊なコード（数の一種）である．

(ii) 64pol の式を実行し，その値を左辺の locative number に格納する．

Locative integer を用いて Fortran 風に繰り返しスタイルで書いた階乗の例を図4に示す．"signed-integer-locatives" は，locative を生成して引数である変数に代入する関数である．この例では，関数 fact に与えられる引数 (n) も locative integer であるとする．

```
(de fact (n)
  (prog (x)
        (signed-integer-locatives x)
        (x := 1)
   loop (if (n = 0) (return x))
        (x := (x * n))
        (n := (n - 1))
        (go loop) ))
```

図4. Fortran-style Factorial

3-3. 自動ポインタ更新

Locbit のさすビットデータのメモリ領域は，TAO レベルで書く入出力プログラムのバッファ，エディタのバッファ，データ対応に設ける何ビットかのフラグ，コンパイルドコードの格納領域など，多様な用途をもつ．これらの用途において，locbit のさすデータをアクセスするとともに，そのポインタを1だけ更新するという動作が頻繁に必要になる．TAO では locbit に対して4種の操作（関数またはメッセージ）を定義し，pdp11 アセンブラ[6]や C言語等に見られるようなこれらの機能を，構文としても非常に近い形で実現した．表3に示すように，locbit には ++ と -- というメッセージが送れる．また，同じく ++ および -- という関数がある．以下，これらの動作について述べる．

表3. 参照とポインタの更新

湯気	式	式の種類	意味
○	(p ++)	メッセージ	間接参照後増加
○	(p --)	メッセージ	間接参照後減少
×	(++ p)	関数	増加後間接参照
×	(-- p)	関数	減少後間接参照

(p は locbit を押さえている変数)

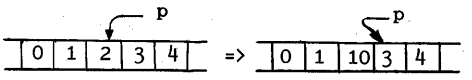
(i) (p ++) ----- p は locbit とする．まず，rpr，rpf を，現在の p によって設定する．しかるのち，p のオフセットを1だけ増加し，新しい p を値として返す．すなわち，(p ++) を実行すると，もとの p は壊されるが，rpr，rpf はもとの p をさしたままになっている．したがって，

```
(:@(p ++) 10)
```

という式は，pdp 11 アセンブラ風に書けば，

```
MOVB    #10,(P)+
```

の意味であり（8ビットの locbit の場合），図5のように動作することになる．



(:@(p ++) 10)

図5. 間接参照後増加

このように，(p ++) を実行した結果返される値の locbit p と rpr，rpf とは一時的に食い違った状態になる．この状態を，比喩的に「もとの locbit の場所から rpr，rpf による湯気が立っている」と言うことにする．

(ii) (p --) ----- この場合も (p ++) と同様である．オフセットが1だけ増加でなく減少する点が異なる．この式の場合も，「湯気」の立った状態が生ずる．(i) と (ii) はメッセージ伝達の式である．

(iii) (++ p) ------ この場合，++ は関数である．まず p のオフセットを1だけ増加する．しかるのち，rpr，rpf を新しい p に対して設定する．そして，こ

の式の値としては，新しい locbit p を返す．したがって，この場合にはpの値とrpr, rpfの一時的な食い違いは生じない．すなわち，「湯気」は立たない．

(iv) (-- p) ------この場合も(++ p)と同様である．湯気は立たない．したがって，

```
(:@(-- p) 10)
```

という式は，pdp 11 アセンブラ風に書けば，

```
MOVB    #10,-(P)
```

の意味である．動作については自明であろう．

```
(loop (!until ((offset-of p) > memblk-size))
      (:@(p ++) 0) )
```

または

```
(for i (index 0 (memblk-size - 1))
     (:@(p ++) 0) )
```

図6. ポインタ自動更新によるプログラムの例

上記4つの機能のうち，(ii)と(iii)に対応する機能はpdp 11 アセンブラにはない．ポインタの自動更新を用いて，locbitのメモリ領域をクリアするプログラムの例を図6に示す．

ポインタ自動更新命令を使うと，locbit が正規の範囲を越えることがある．図6の例でも，loopの出口では，pのオフセットはメモリ領域の範囲外に出ている．しかし，この時点では「範囲外エラー」は生じない．このあともし @p などを行なえばエラーになる．ただ，この時点から (-- p)を行なえば矛盾は生じないのでエラーは起きない．

Locbitによるメモリ領域間の転送は，p, q を各バッファ・ポインタとして，

```
(:@(p ++) @(q ++))
```

なる式でループすれば容易に行なえる．3-1-2の(i)で述べたように，derefはrpfが設定されていればそのrpfによって値をとるので，この式の右辺値はqの（オフセット増加前の）値となる．

4. おわりに

TAOの代入式について，その言語表層および内部的な実現機構を述べた．言語仕様上提供されるデータ型の多様性に合わせて，そのデータ型を更新するための代入式の実現も多様になる．TAOでは，この多様性を表層上も内部機構でも統一的に扱えるようにした．タグや汎用レジスタなどのELISのハードウェアを活用することによって，TAOの代入式は内部的な式の変換の必要もなく，直接的に解釈実行できる．代入式の統一的な扱いは，多数の宣言と静的な型チェックを行なうコンパイル型言語ではふつうに実現しているが，Lispではあまり実現されていない．TAOではこれを，代入場所情報を保持する内部レジスタを設けることで実現した．この計算機構は，代入という概念の存在しうるLispオブジェクトの変数からビットデータに至るまで同一である．

参考文献

1) 日比野 他：LispマシンELISの基本設計，情報処理学会記号処理研究会資料 12-15，1980年6月．
2) Okuno,H.G. et al.: TAO: A Fast Interpreter-Centered System on Lisp Machine ELIS, Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Aug. 6-8, pp. 140-149.
3) Goldberg,A. et al.: Smalltalk-80: The Language and its implementation, Reading, Massachusetts, Addidon-Weslsy, 1983.
4) Weinreb,D. et al.: Lisp Machine Manual, Fifth Edition, LMI, Jan. 1983.
5) Osato,N. et al.: Object-Oriented Programming in Lisp, Report of SIGSYM, 26-4, IPSJ, Dec. 1983.
6) DEC, pdp11 processor handbook.