融合型言語TAOにおける構造データと未定義値の扱い

山崎憲一[†1] 吉田雅治[†2]
天海良治[†3] 竹内郁雄[†4]

Lisp に論理型言語の機能を融合する場合には，論理型言語特有の内部データ構造である未定義値（UNDEF）と，参照（REF）をどのように表現するかということが問題となる．本論文では，我々が TAO86 および TAO という 2 つの言語で採ったアプローチについて述べる．UNDEF には，必然的に構造データ中の「番地」という概念がともなうが，これを Lisp からどのように隠蔽するかがポイントとなる．TAO86 では，UNDEF を即値とし，REF を処理系が自動的にたどる方法によって実装を行った．これにより言語が単純なものとなるが，いくつかのケースにおいては特有の問題を生じる．たとえば，REF や UNDEF を含む構造データの同一性を判定するために，アドホックな関数を必要とする．TAO では同様の方法を採りつつも，論理変数を述語の値として返す関数型の機能，パターンマッチによる同一性の判定の機能などを用いて，これらの問題を解決する．また，本論文では TAO のタグ割当てなど，実装の詳細についても述べる．

Structured Data and Undefined Value in TAO

Kenichi Yamazaki,[†1] Masaharu Yoshida,[†2] Yoshiji Amagai[†3] and Ikuo Takeuchi[†4]

Lisp can have much expressive power by incorporating logic-programming facilities. In this paper, we discuss implementation issues of this incorporation, especially for internal data representation such as undefined value ('UNDEF') and reference ('REF'). We designed two Lisp-based multi-paradigm programming languages, TAO86 and TAO, to solve these issues. In TAO86, UNDEF is a first-class immediate value, and REF is dereferenced by the system automatically. This solution is quite simple but did not have enough power to handle the whole set of data so that TAO86 provides some adhoc builtin functions. TAO, a thoroughly redesigned successor of TAO86, has new features such as functional predicates and pattern-matching mechanism that give an elegant solution of TAO86 issues. This paper also describes the implementation of internal data representation and data handling mechanisms in detail.

1. はじめに

記号処理あるいは知識処理は，より多くの分野で必要となりつつあり，Lisp は記号処理言語の代表として，今後とも重要な言語の 1 つであり続けるであろう[1)]．Prolog に代表される論理型言語は，もう 1 つの記号処理言語の代表である．論理型言語では宣言的な記述が可能であり（関数型言語も宣言的な記述が可能であるが，本論文では Lisp を対象としている），また単一化によるパターン処理と後戻り探索という独自の計算機構を持つ．このため Lisp あるいは関数型言語から論理型計算の機能を用いたいという要求は多く，さまざまな研究がなされてきた[2),3)]．

それらの研究は，関数型と論理型を融合した新しい計算モデルを構築しようとするものと，言語の基本的な枠組みを保ったままで融合を図るものとに大別できよう．前者では，より単純で理論的に扱いやすい計算モデルを構築することを目指し，その結果として新しい言語を提案する．一方，後者の立場では，蓄積されたプログラムやプログラミング技法などを再利用しつつ，他のプログラミングパラダイムをも利用可能な言語を構築する．我々は，後者の立場をとり，融合に関する機能を用いない限り従来からの Lisp プログラムがそのまま動くこと，論理型計算については Prolog 相当の機能を持ち，多くの Prolog プログラムが機械

†1 NTT ドコモ
NTT DoCoMo
†2 エヌ・ティ・ティアイティ
NTT-IT
†3 NTT
NTT
†4 電気通信大学
The University of Electro-Communications

的変換程度で動作することを重視する．そのうえで，論理型計算によって生成された任意のデータを Lisp によって自由に操作できることを目指す．

本論文では，Lisp と論理型言語のこのような融合においてデータ操作に関して生じる問題を考察するが，まず比較のために前者の立場の典型的な研究である Curry [4),5)] について見てみよう．Curry では，residuation と narrowing という 2 つの方法で計算を行うことができる．たとえば，

```
f 0 = 2
f 1 = 3
f eval flex
```

と定義された関数 f は，flexible な関数と呼ばれ narrowing に基づき実行される．

```
f x
```

を実行すると，

```
{x=0} 2 | {x=1} 3 .
```

という答が返る（x=0 という置換の下では 2 が値，x=1 では 3 という意味）．一方，flex のかわりに

```
f eval rigid
```

と定義すると，f x の実行はサスペンドする．

このような言語では，変数の値が具体化されていない状態（以降，未定義状態と呼ぶ☆）をファーストクラスデータとして扱うことはできない．

我々の採る後者の立場，つまり Lisp の基本的な枠組みを保ったまま融合するという立場では，未定義状態を Lisp から何らかの方法で扱えるようにすることが自然であろう．未定義状態を Lisp から操作したときに実行がサスペンドするのでは，（少なくとも従来の）Lisp とは呼べない．このような立場での融合に関する研究も古くからなされている．Lisp Machine Lisp 上に Prolog を実装した LM-Prolog [6)] においては，未定義状態は locative と呼ばれるデータである．Pop-11 との融合を行った POPLOG [7)] でも，未定義状態（および後述する参照 REF も）は Pop-11 にとってファーストクラスデータである．それらの言語では未定義状態を陽に操作できる．しかし，このことはデータの抽象度を下げるということであり，プログラマの負担が増える，あるいは扱いやすい形に変換するためのデータコピーが必要となるといった問題がある．本研究の目標は，データに関するこれらの問題を生じない融合方法を提案することにある．

また，実際の Prolog 処理系は，記述力を高めるためにピュアな Prolog にはないさまざまな述語を提供する．たとえば，組込み述語 var は未定義状態を調べる機構そのものである．また，ゴールの全解を求める述語 bagof のように後戻りを超えて値を保持する機構は，未定義状態と副作用の問題に深く関連する．このような処理を言語の枠内で記述できるようにすることも，我々の目標である．

我々は，これまで言語 TAO86 と TAO を提案してきた．TAO86 [8)] は，融合型言語を目指して我々が最初に設計した言語である．TAO86 では，Lisp 部分を設計した後に論理型計算の機構を導入したという歴史的な経緯から十分な検討を行わず単純なアプローチを採った．本論文では，まず単純な融合方法を採るとどのような問題が生じるかを TAO86 を例として説明し問題点を明らかにする．その後，TAO において採った解決法について述べる．このような順序で述べるのは，問題点がより明確になるという理由のほかに，言語の改良のステップを示すこと自体にも学術的な価値があると我々は考えるからである．

TAO については，すでにいくつかの観点から報告してきた [9)]．本論文では，上記の問題を解決するための新機能として，値を返す述語といくつかのプリミティブを提案する．この機能追加を行った言語は新 TAO と呼ぶべき言語であるが，本論文では TAO と呼ぶ．文献 9) に対する新規の機能は「本論文での提案」と明記する．

本論文は，次のように構成される．まず，2 章において問題とその背景を明確にし，3 章では TAO86 のアプローチについて述べるとともに，実際に TAO86 を使用する過程の中で発見された問題点を指摘する．それらの問題を TAO でどのように解決したかを 4 章で述べ，TAO の実装法を 5 章で述べる．次に関連研究との比較を行い，最後にまとめを行う．

2. 背景と問題

2.1 論理型言語特有のデータ構造

ここでは，簡単に論理型言語のデータ構造について，WAM（Warren Abstract Machine）[10),11)] による実装を中心として説明する．

WAM では，未定義状態の変数どうしを単一化すると，一方の変数に，もう一方の変数のアドレスが参照を示すタグ付きで代入される（リンク処理）．このよ

☆ 変数束縛が環境中に見つからない状態とは異なる．

うな関接ポインタを REF と呼ぶ☆．未定義状態の変数の値は，自分自身を指す REF である．単一化の際に，変数の値が REF のとき，自分へのポインタである場合は未定義状態として扱い，そうでない場合は REF の指す番地を読み出して単一化を続ける（deref 処理）．論理型言語の重要な機能である後戻りを実現するためには，単一化を取り消す必要がある．このため，単一化による代入がなされた番地を記憶しておき（トレール処理），これを用いて後戻り時に変数の値を未定義状態に戻す（取消し処理）．

単一化での代入は，未定義状態の変数に対してのみなされ，その代入のためには番地が必要となる．自分自身への REF を利用して未定義状態を表現するという WAM の技法は，未定義状態には「番地」の概念が密接に関連していることを示唆しているといえよう．

2.2 基本的な問題と設計方針

このように未定義状態と番地とは密接に関連はしているものの，ここでは概念的な検討をするために異なるものとして考える．以降では，未定義状態を UNDEF と呼ぶ．UNDEF と REF をどのように Lisp に見せるべきかについて，次の 3 つが考えられよう．

- UNDEF も REF も Lisp からは見えない．
  REF は処理系により自動的に deref される．UNDEF にアクセスしようとするとそれが具体化するまで処理はサスペンドする．これは前述の Curry のように，新しい計算モデルを構築するアプローチなどに見られる方法である．Lisp の保持を目指す我々の目的には，この方法は適さない．
- UNDEF も REF も見える．
  UNDEF も REF もファーストクラスデータであり，REF が現れたらプログラムで陽にたどる．前述の POPLOG がこれに該当する．また，LEDA [12] は，さまざまな計算パラダイムを実装可能な言語であり，ここでも UNDEF や REF はプログラムの対象である．ユーザが陽に deref を記述することは負担が多く，また従来の Lisp プログラムを修正して UNDEF を扱おうとしたときの修正量が多くなるという欠点もある．
- UNDEF は見えるが，REF は見えない．
  Lisp に対する新しいファーストクラスデータとして UNDEF のみが加わる．これは我々が TAO86 と TAO で採った方法である．

次に実装に関する選択肢として，UNDEF を（たとえばセルのような）ヒープ上に場所を占めるデータとするのか即値とするのかがある．前者の場合，セルと同様に何らかのプリミティブを使ってメモリを確保し回収は GC に任せることになる．この方法では，論理変数が出現するたびに新しくメモリが必要になり大量のメモリを消費すると予想される．一方，後者ではスタック上に UNDEF を直接置くことができ，メモリ効率は向上する．このため，我々は UNDEF を即値とすることにしたが，UNDEF は番地の概念をともなったデータであり，即値として扱うことにより番地情報が失われる．本論文で考察する問題の多くは，実はこのことに起因する．

☆ REF は WAM の用語で reference の意味である．一般に forwarding pointer とも呼ばれるが，ここでは WAM の用語を用いる．

3. TAO86 のアプローチ

TAO86 の論理型プログラムは，いわば S 式で書いた Prolog であり機械的な変換により Prolog プログラムがほぼそのまま動くことを目指した（実際，TAO86 の市販版である TAO/ELIS システムには DEC-10 Prolog 互換パッケージが備えられていた）．本章では，TAO86 において UNDEF と REF がどのような問題を引き起こすのかについて述べ，TAO の解決すべき問題点を具体的に明らかにする．

3.1 TAO86 の概要

TAO86 で記述した `append` を示す．

```
(assert (append () _x _x))
(assert (append (_a . _x) _y (_a . _z))
        (append _x _y _z))
```

述語は `assert` によって定義される．述語を呼び出すには，次のような式を実行する（→は，実行結果の値を示す）．

```
(let (_x)
  (append (a b) (c d) _x) )
→ t
(let (_x)
  (append (a b) (c d) _x) _x)
→ (a b c d)
```

述語は関数の一種であり，このように関数と同じ構文によって呼び出す．「_」で始まる名前の変数は論理変数と呼ばれる．論理変数と通常の変数（以降では Lisp 変数と呼ぶ）は，初期値（論理変数の初期値は UNDEF，Lisp 変数は `nil`）と単一化での扱いのみが異なり，それ以外はまったく同じように処理される．

述語呼び出し式の引数は，関数の引数のようには評価されずパターンと解釈され，論理変数があるとその値が単一化の対象となる．

述語は失敗すると `nil` を返し，成功すると非 `nil` を

返す．例：

```
(assert (foo a))
(assert (foo b) 3)
(foo a) → t
(foo b) → 3
(foo c) → nil
```

後戻りは，ボディ中の式（ゴール）を評価した結果が nil であるときに起動される．上のプログラムに以下の述語 bar を加えて実行する．

```
(assert (bar) (print 'first) (foo c))
(assert (bar) (print 'second) nil)
(assert (bar) (print 'third))

(bar)
→
 first      ; 印字
 second     ; 印字
 third      ; 印字
t           ; 評価結果
```

bar の最初の節では foo の実行が失敗し nil を返すため第 2 の節へ後戻りする．第 2 の節ではボディ中の nil によりただちに後戻りが起動される．第 3 の節は成功する（なお，TAO86 の関数 print は印字したデータを返す）．

述語は失敗したときに nil を返すという意味で，返す値に制御の意味が重ねられている．つまり述語は任意の値を返すことができない．これは述語と eq や atom などのテスト関数とを同一視することを重視し，同じ値を返すべきと考えたためである．しかし，その後の我々の経験では述語が失敗したことを呼び出しの値として知りたいことは多くはなく，むしろ nil に制御としての意味を重ねたことが問題となることが多かった．後述するように TAO では，述語の失敗は大域脱出と考え，成功して終わることが普通の状態と考える．

3.2 より詳細なデータの意味

論理型言語としての TAO86 の意味はこれまで述べてきたように単純なものであるが，Lisp の代入などとの組合せで生じる問題（3.4 節）について考えるためには UNDEF と REF に関してより厳密に意味を規定する必要がある．このために，ファーストクラスではないシステム内部のデータとして invs と invm を導入する．これらはデータを指すポインタを表現する内部データで，REF に相当する．

invs と invm は，UNDEF の論理変数どうしを単一化することにより生成される．構造データ中に存在する値を指す場合には invm が用いられ，その他の場合には invs が用いられる．簡単な例として次の単一化を考えてみよう．

```
(let (_x _y) (== _x _y))
```

（ここで，==は単一化を行うプリミティブである）．この例では，単一化終了時の_y の値は_x への invs となる（_x の値が_y への invs になることもある）．

もう 1 つの例として，次の単一化を考えてみよう．

```
(let (_x _y _z)
  (== _z (_x))
  (== _z (_y)))
```

この結果_x と_y とが単一化されるが，これらはリストという構造データ中に出現するから，一方の値はもう一方への invm となる．より複雑なケースを理解するには実装について触れる必要があり，これは次節で述べる．

新たに導入したデータ invs と invm に対する評価の意味は次のようになる．まず，評価した結果が invs であった場合には，invs が続く限り deref 処理が行われる．一方，評価した結果が invm であっても deref は行われない．invm の deref は，それの指す値が必要になったときに初めて実行される．

このような invs と invm の非対称性は，invm が無限存続期間（indefinite extent）のデータへのポインタであるのに対し，invs は動的存続期間（dynamic extent）に対するものであることに起因する．

3.3 内部データ表現と実装

TAO86 の実装において本論文に関連する主なタグは以下のとおりである．

invs: スタックへのポインタであることを示すタグ

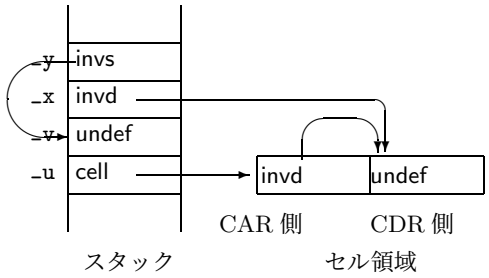
inva, invd: セル領域の CAR または CDR へのポインタであることを示すタグ

undef: UNDEF を表すタグ

論理変数の初期値は UNDEF である．論理変数を宣言するとスタック上にスロットが確保され，タグ部には undef，データ部には UNDEF ごとにユニークな UNDEF 番号が格納される．UNDEF 番号は UNDEF を印字したときの識別のためである．前節で導入した invs はタグ invs を用いて表現され，invm は inva と invd とにより表現される．これは TAO86 の実装された専用マシン ELIS が CAR 空間と CDR 空間とが分離したアーキテクチャであったためであり本質的ではない．

典型的な内部データ構造の例として，

```
(let (_u _v _x _y)
  (== _u (_x _x))
  (== _v _y) )
```



（セル中の UNDEF が，CAR 側にくることもある）

図 1 TAO86 の内部データ構造
Fig. 1 Internal data representation of TAO86.

というプログラムを実行した後のデータ構造を**図 1** に示す．

単一化の実装においては，特にリンクを張る方向が重要である．スタック上に置かれた UNDEF どうしを単一化すると，浅い方から深い方へ向かうように invs が張られる．また，スタック上とヒープ上（つまり構造データ内）の UNDEF どうしの場合はスタックからヒープへ向かう invm（タグ inva または invd）が張られる．これは dangling ポインタを避けるための WAM と同様の技法である．ただし，WAM では構造データはグローバルスタック上に置かれるが，TAO86 ではヒープに置かれ，浅い/深いという関係は生じないためヒープ内で張られる invm は方向を考慮しない．

リンクを張る方向がこのように決まっていることから，invm を deref した結果として invs が現れることはない．また，invs は論理変数どうしの単一化でのみ生成されるから invs は論理変数の値としてしか現れない．このため前節で述べた，評価の結果が invs であったら deref を行うという処理は，論理変数の値が invs であるかを調べるだけで実現できる．

一方，invm のチェックと deref 処理は，各プリミティブにおいて値が必要になったときに行われる型チェックにおいて行う．専用マシン ELIS にはタグチェックのためのハードウェアが備えられており，オーバヘッドなしに invm のチェックができる．

3.4 実際に起きた問題

我々は，TAO86 を用いて論理型プログラミングのライブラリやいくつかの実用的プログラムを作成した．それらの経験を通して，次のような TAO86 の問題点が判明した．

- 主に破壊的代入との関連で生じる問題
  問題 1： 代入する場所を制御できない．
  問題 2： 破壊的代入を取り消せない．
- 論理型計算の生成したデータの利用に関する問題
  問題 3： invm は自動的に deref されない．
  問題 4： 逆に invs は自動的に deref される．
  問題 5： 構造データが後戻りで破壊されてしまう．
  問題 6： UNDEF の同一性判定ができない．

以下では，それぞれについて簡単に説明する．

[**問題 1**]

```
(let (_x _y)
  (== _x _y)
  (!_x 'a)
  _y)
```

ここで，3 行目の式は TAO86 の代入式である．ここでは Lisp の SETQ と同等と考えればよい（SETQ との違いについては後述する）．以降では，単一化による代入と区別するために破壊的代入と呼ぶ．このプログラムの実行結果は，a が返される場合と UNDEF が返される場合とがある．2 行目の単一化の結果，_x と _y のスタック内の番地に依存して，_y から _x への REF（invs）となる場合とその逆の場合とがあるからである．前者のときは，_x の値を書き換えると _y の（deref した）値も変わる．後者のときは，_x への破壊的代入により REF が書き換えられ，_y は UNDEF のまま残る．代入先が REF であったときに，代入すべき場所を制御することができない．

[**問題 2**]

破壊的代入に関するもう 1 つの問題は，論理型計算の中で破壊的代入を実行したとき，その代入を越える後戻りが起きても代入を取り消すことができないという点である．後戻りは，Lisp の行った副作用には関与しないからである．

[**問題 3**]

```
(goal (== _u (_x _x))
      (== _x a)
      (!*x* (car _u))
      nil)
```

ここで，goal は論理型計算のトップレベル実行に用いるマクロで，そのボディを順に論理型計算により実行していく．1 行目の単一化により図 1 のようなセルが作られる．最後の nil により後戻りが起き単一化が取り消されるが，その結果 *x* の値は UNDEF となる．これは，3 行目の式の (car _u) が invm（タグ invd）を返すため，*x* の値が _x への invm となるからである．後戻りで _x の値が取り消されると，*x* の（deref した）値は UNDEF となってしまう．

これを避けるには，陽に deref を行うための組込み関数 peelinv を用いればよい．先の破壊的代入を，

```
(!*x* (peelinv (car _x)))
```

とすれば，*x*の値は後戻り後も a である．

[**問題 4**]

```
(let (_x _y z)
  (== _x _y)
  (!z _y)
  (== _x a)
  z)
→ UNDEF
```

2 行目の単一化の結果，_y が UNDEF のままであったり invs となったりするが，3 行目の破壊的代入の右辺で invs は自動的に deref されてしまうため，いずれにしろ z には UNDEF が直接入る．_x と関係のないこの UNDEF は 4 行目の単一化で書き換えられることはない．

[**問題 5**]

```
(goal (append (a b) (c d) _x)
      (!*x* _x)
      nil)
```

このプログラムを実行した後の*x*の値は，

```
(a . UNDEF)
```

である．これは後戻りにより，単一化の行った代入が取り消され構造データを作り出した処理の一部も取り消されてしまうからである．これを避けるためには，*x*に破壊的代入する際に copy-tree をするなどして構造データ全体をコピーする必要がある．

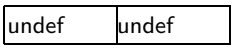
[**問題 6**]

あるゴールのすべての解を求める述語として，Prolog には bagof などの述語がある．TAO86 では，このために goal-all-list が用意されている．例：

```
(assert (foo a))
(assert (foo b))
(goal-all-list _x (foo _x)) → (a b)
```

goal-all-list は，1 つのゴールを実行し終えたら，解をコピーして後戻りで戻されないようにした後，後戻りを起動する．これを繰り返してすべての解を求める．ここでのコピーは，UNDEF の同一性まで含めて正しくコピーする必要がある．copy-tree で図 1 の _u の値をコピーすると**図 2** のようになってしまい問題がある．

goal-all-list が用いる内部関数 unified-copy は，copy-tree と似ているが，コピー中に出現した UNDEF を a-list に登録しておき，後で同じ UNDEF が出現したときにはそれへの REF を使い UNDEF の同一性を保つようにコピーする．ただし，TAO86 では UNDEF の同一性判定ができないため，unified-copy では UNDEF 番号を用いて判定しており問題がある（たとえば図 2 では UNDEF 番号は同じだが UNDEF としては異なる）．UNDEF の同一性を保ったコピーを行うには，UNDEF の同一性判定機能が必要である．



（CAR と CDR の UNDEF 番号は同じ）

図 2 copy-tree 後のセル

Fig. 2 A cell created by copy-tree.

以上の問題は，（peelinv や copy-tree などの使用により）TAO86 で解決されたものもあるが，TAO の設計において十分検討する必要がある．

なお，実装上の大きな問題としてトレール量の増加がある．WAM では構造データがグローバルスタック上に置かれることを利用してトレール処理を簡略化するが，TAO86 では構造データはヒープ上に置かれるためこの技法は使えない．TAO では，UNDEF に世代番号をつける方法によりこの問題を解決したが，これについては報告済み[13] であり本論文では割愛する．

4. TAO のアプローチ

前章で述べた問題の多くは REF と UNDEF の見せ方に起因する．たとえば，REF をファーストクラスデータとし UNDEF をヒープに置くデータとすれば，ほとんどの問題は解決する．しかし，2.2 節で述べたように，TAO86 での UNDEF と REF の見せ方は我々の目指す融合型言語には必須のものである．そこで我々は，Lisp の評価方法は（TAO86 のまま）変更せず，論理型計算で REF や UNDEF を扱うべきではないかと考えた．我々の基本的なアイデアは，REF や UNDEF は論理型計算で操作し，その結果を述語の返す値を利用して Lisp に戻すというものである．

4.1 TAO の概要と関数型述語

append を TAO で定義すると次のようになる（なお，TAO では真偽値を表すデータは#t と#f である）．

```
(defpred append                          ガード
    ((() _x _y) (:guard #t)
    {! _x _y} )
ヘッド
    (((_a . _x1) _y _z) (:aux z1)
    (:guard #t)                  補助変数宣言
    {! _z (_a . _z1)}
    {append _x1 _y _z1} ))
```

述語は defpred で定義され，その定義は述語名と複数の節のリストからなる．各節は，ヘッド，ガード，およびゴールのリスト（ボディ）から構成される．なお，キーワードで始まるリストは特殊形式（この場合は defpred）の構文の一部であり，オプショナルな意味を付与する（たとえば，補助変数宣言がないこともある）．

上のプログラムによって定義される 3 引数の述語 append は，第 1 引数にリストを受け取り第 2 引数と連結して第 3 引数に返すもので，

```
(let (x)
  {append (a b) (c d) _x} )
→ #t
(let (x)
  {append (a b) (c d) _x} x)
→ (a b c d)
```

となる．このように TAO では述語呼び出しと関数呼び出しは異なる構文を持つ．また，TAO では論理変数と Lisp 変数とは区別されない．すべての変数の初期値は UNDEF である．パターン中の「_」は，強制評価式と呼ばれる特殊な式（4.2 節）を示す記法であって TAO86 のように論理変数名の一部ではない．

TAO の論理型計算の主な特徴を述語の実行過程に沿って説明すると次のようになる．あるゴール（述語呼び出し式）の実行とは以下の手順である．

- 述語呼び出し式の引数を構造コピーする（4.2 節）．
- 述語の定義から節を 1 つ選び，その節のヘッドとのパターンマッチにより引数を渡す（4.2 節）．
- ヘッドに現れずボディにのみ現れる変数を補助変数宣言する．
- パターンマッチが成功するとガード（の引数）を評価し，その値が#f 以外の場合のみボディの実行が許される．
- パターンマッチが失敗するかガードが#f のときは，次の節に実行が移る．
- 単一化は，ボディ中で次のような式を実行することによってのみ行うことができる．

  {! パターン$_1$ パターン$_2$ }
- ボディのゴールを順に実行し，最後のゴールの値を返す．

パターンマッチが失敗するかガードの値が#f となることを節選択が失敗したという．すべての節選択が失敗するとエラーとなる．Prolog のように（深い）後戻りは自動的には起動されない．後戻りを行うにはチョイスと呼ばれる継続の生成とチョイスへの制御移動を陽に行う（本論文では詳細は割愛する）．

データの問題に関して重要な役割を持つのが本論文で提案する値を返す述語である☆．TAO86 では述語の返す値に制御の意味を重ねたため，任意の値を返すことができなかった．TAO の述語は，大域脱出を実行しない限り呼び出し元に任意の値を返すことができる☆☆．ボディに返す値が陽に書かれた述語を，以降では関数型述語と呼ぶ（ボディのない述語は#t を返す）．関数型述語を使うことにより，前章で述べた多くの問題を解決することができる．

TAO86 と TAO との相違をまとめると，次のようになる．

	TAO86	TAO
引数渡し	単一化	構造コピーとパターンマッチ
述語の返す値	成功か失敗	任意の値
論理変数	Lisp 変数と区別	区別なし
後戻り	Prolog と同じ	後戻り時にフックを実行可

4.2 構造コピーとパターンマッチ

TAO86 同様に invs，invm は内部データである．評価に関してのこれらの意味は TAO86 と同じ（3.2 節）である．本節では，これらを用いて構造コピーとパターンマッチの意味について述べる．

述語呼び出し式を実行したとき，その式の引数は関数の引数のようには評価されず，かわりに構造コピーがなされる（TAO86 では構造コピーは実装法の一部であったが，TAO では強制評価の環境やパターンマッチの意味を明確にするために構造コピーを言語の意味に含める）．構造コピーは，引数の構文上の表記からデータ構造を生成する操作である．定数の表記を構造コピーすればその表記に対応する定数が返され，構造データの表記を構造コピーすれば新たに構造データが生成されて返される．「_」で始まる式（強制評価式）は，_を除いた部分の表記が式として強制評価され，その結果が構造コピーの結果となる．たとえば，

```
(let ((x 1)) {pred1 a (b) _x})
```

では，pred1 の引数は，シンボル a，リスト (b)，数字の 1 である．強制評価は，評価結果の値に加え，その値が存在していた場所の情報を付随して返す特殊な

☆ 文献 9) では，述語は，成功か失敗かを表す真偽値と述語の返した値の多値を返すものであった．

☆☆ 厳密には，失敗により大域脱出が起きると，デフォルトの脱出ハンドラが#f を返す．デフォルトのハンドラを変更することにより，失敗したときの動作を変えることができる．

評価である．返された値が UNDEF であった場合は，付随情報を使って UNDEF が存在していた場所への REF が生成され，それが構造コピーの値となる．たとえば，

```
(let (x y) {pred2 _x _x _y})
```

では，どの引数も deref した値は UNDEF であるが，第 1 引数と第 2 引数は変数 x の値が存在する場所への invs であり，y への invs である第 3 引数とは異なる．

パターンマッチは，ヘッドの表記と構造コピーされた引数とを比較しながら，ヘッドに初出した_の付いたシンボルの変数束縛を行う操作である（単一化とは異なり，引数が UNDEF であっても引数への代入はなされない）．上のプログラムに対し pred1 が次のように定義されていたとする．

```
(defpred pred1 ((a _u _v) (:guard #t)))
```

第 1 引数のパターンマッチは成功し，第 2 引数のパターンマッチではシンボル u は初出であるため変数 u と (b) とが変数束縛される．第 3 引数のパターンマッチでは変数 v と invs が変数束縛される．

_の付いた同じシンボルが複数回現れてもよい．上の pred2 の例に対し，次の定義を考える．

```
(defpred pred2 ((_u _u _v) (:guard #t)))
```

第 2 引数のパターンマッチにおいては，第 1 引数のパターンマッチで束縛した u の値（つまり x への invs）と第 2 引数とが比較される．この比較は，Lisp の eq と同じであるが，値が UNDEF どうしの場合はそれらが存在する場所が同じ場合に等しくなる．これにより，UNDEF の同一性判定が TAO では次のように自然に定義でき，TAO86 の問題 6 が解決できる．

```
(defpred eq ((_x _x) (:guard #t)))
(let (x) {eq _x _x}) → #t
(let (x) (eq x x)) → #t
(let (x y) {eq _x _y}) → #f
(let (x y) (eq x y)) → #t
```

4.3 いくつかのプリミティブ

本節では本論文で提案するいくつかの TAO 独自のプリミティブについて述べる．TAO86 の peelinv に相当する TAO の組込み述語は，述語 deref である（本論文での提案）．述語 deref は引数を deref し値を返す．

```
{deref 3} → 3
{deref (a)} → (a)
(let (x) {deref _x}) → UNDEF
```

deref により peelinv と同様に問題 3 を解決できるが，これに加え deref の特徴は破壊的代入のための場所を付随することにある☆．たとえば，

```
(let (x y)
  {! _x _y}
  (!{deref _x} 'a)
  y)
```

のような代入ができる．問題 1 の例とは異なり，必ず a が返される．関数 peelinv を機能拡張することでは同様のことを実現できない．peelinv は関数であるため引数の invs はただちに deref されてしまい，その場所の情報は peelinv には渡らないからである．

値を返す述語の特徴を端的に示す例が次のように定義される述語 globalize である．

```
(defpred globalize
  ((_x) (:guard #t) x) )
```

述語の最後のゴールが返した値が invs だった場合には，（通常の評価のような単なる deref でなく）次の処理がなされる．まず，ヒープ上にメモリ（現在の実装ではセル）を確保し，そこに UNDEF を格納する．次に，元のスタック上の UNDEF からヒープ上の新たな UNDEF へと invm を作り，その invm を述語の値として返す（この処理は，WAM では globalize と呼ばれる）．これにより UNDEF に対する一種のラップが生成でき，場所の情報を失うことなく UNDEF を Lisp に渡すことができる．たとえば，問題 4 の例を次のようにする．

```
(let (x y z)
  {! _x _y}
  (!z {globalize _y})
  {! _x a}
  z)
→ a
```

3 行目の globalize が返した invm は deref されずに z に代入される．4 行目の単一化により，この invm の指す先の UNDEF が書き換えられ z の値が a となる．

4.4 破壊的代入の取消し

TAO では述語ボディの実行中に関数を登録しておき，後戻り時にその関数を実行することができる．これについてはすでに報告済み[9] であるため，代入時と

☆ TAO，TAO86 では，破壊的代入の左辺に任意の式を書ける．式が評価され値を返すときに，その値が存在した場所が値に付随して返される．破壊的代入式は，左辺の値があった場所へ右辺の値を代入する．たとえば，

```
(!(cdr list) (cddr list))
```

のような記述が可能である．この左辺の評価機構を用いて TAO の強制評価は実現されている．

後戻り時に実行すべき部分プログラムについてのみ述べる．

変数 x を newval で書き換えるという問題を考える．x が REF を経由しない直接の値であるときは，次のようにすれば十分である．

```
代入時：      (!oldval x)
              (!x newval)
取り消し時：  (!x oldval)
```

一方，x が REF である可能性があり，それを deref した場所に代入したいときには，

```
代入時：      (!oldval {deref _x})
              (!{deref _x} newval)
取り消し時：  (!{deref _x} oldval)
```

とすればよい．

このほか newval も REF の可能性がある場合は，これに対しても deref を行うなど，データ構造を想定できるならば問題 2 の破壊的代入の取消しが可能となった．

4.5 例

ここでは関数型述語を用いた例として，リストをコピーするプログラムを示す．ただし，元のリストに UNDEF が含まれたら，コピーにより新たに生成されたリストから REF を張り，元の UNDEF を共有するようにする．

まず，UNDEF を考慮しない単なるコピーは次のような単純なプログラムである（各テスト関数は TAO 独自のものもあるが，その意味は明らかであろう）．

```
(defun copy (x)
  (cond ((atom? x) x)
        ((cons? x)
         (cons (copy (car x))
               (copy (cdr x))))))
```

次に，セルに直接 UNDEF が埋め込まれている場合に対処するため car や cdr で得た値を globalize する．

```
(defun copy (x)
  (cond
    ((atom? x) x)
    ((undef? x) x)
    ((cons? x)
     (cons
       (copy {globalize _(car x)})
       (copy {globalize _(cdr x)})))))
```

car が次のように定義されていたとすると，次のように書くこともできる（cdr も同様に定義されているとする）．

```
(defpred car (((_x . _)) (:guard #t) x)
(defun copy (x)
  (cond ((atom? x) x)
        ((undef? x) x)
        ((cons? x)
         (cons (copy {car _x})
               (copy {cdr _x}))))))
```

また，すべてを述語にすると次のようになる．

```
(defpred copy
  ((_x) (:guard (atom? x)) x)
  ((_x) (:guard (undef? x)) x)
  (((_car . _cdr)) (:guard #t)
   {cons _{copy _car} _{copy _cdr}}))
```

このように関数型述語を用いて，さまざまな記述方法で UNDEF を扱うことができる．

5. TAO の実装

5.1 内部データ表現

我々は，TAO を専用マシン SILENT[14] 上に実装した．内部データ構造は TAO86 とほぼ同じである（ただし，SILENT では CAR/CDR 空間の区別がなくなりフラットなバイトアドレッシングとなったため，inva，invd は統合され invm となった）．

タグの割付けは，SILENT と同時に設計されさまざまな工夫がなされている（付録 A.1）．本節では，特徴的な点について典型的なマイクロコードを用いて説明する．**図 3** は，CAR をとるプログラムである．図でアンダーラインを付けた部分が特にタグ操作に関連する部分である．1 つの S 式が 1 ステップを要し，通常は，行番号 1 → 18 → 7 の 3 ステップで CAR が終了する．この 3 ステップの中で，メモリアドレスとして読んでよいデータかのチェック（boc 命令），タグが CAR/CDR 可能なデータかのチェック（sptagcase 分岐）が行われている．なお，引数が REF だったときには，1 → 10 → 29 → delinvc サブルーチン→ 30 → . . . と処理が進む．

5.2 関数型述語

値を返す機構の実装についてはすでに報告済み[13]であるため，ここでは新たに提案した部分の実装について述べる．単一化式および述語呼び出し式の引数中に強制評価式として述語呼び出しが現れた場合には，

```
{! _フレッシュな変数 _述語呼び出し式}
```

の形に変形する．たとえば，

```
{foo _{bar}}
```

```
 1:  (,(dyte-code 'car)                  car というバイトコードのエントリ
 2:       (mov <sp> car0)                <sp>を car0 レジスタへ
 3:       (boc <sp> mdr1)                YBR を番地としてメモリを読み mdr1 へ
 4:                                      この時 YBR が即値のタグだったら読まない
 5:       ldrpr                          前のステップの計算結果を rpr（場所レジスタ）に保持する
 6:       (bsr sptagcase car-case))      car-case をサブルーチンコールする
 7:  (    (mov car1 <sp>)                car1 レジスタを<sp>へ
 8:       (dytej hap laphap))            次のバイトコードにディスパッチ

; サブルーチン car-case の定義
 9:  (.case !car-case 3                  3 方向分岐エントリ
10:   (0                                 REF の可能性がある時
11:      (mov car0)
12:      clrrpf                          rpf レジスタのクリア
13:      (j taginv *4))                  タグが invs,invm の時はラベル*4 へ
14:   (1                                 アトムの時
15:      (mov usrmode)                   現在のモードレジスタの値を出力
16:      clrrpf
17:      (br tagnempty *1))              空リストの判定
18:   (2                                 CAR/CDR 可能データの時
19:      (setrpf cell-car)               rpf レジスタのセット
20:      rts)                            サブルーチンからのリターン
21:  )                                   end of case

22:  (.case *1 2                         2 方向分岐エントリ
23:   (0 (empty car1)                    car1 レジスタに空リストをセット
24:      (br mode-car-empty-error *3))   usrmode レジスタのフラグを見る
25:   (1 エラー処理) )                   空リストでない時はエラー
26:  (.case *3 2                         2 方向分岐エントリ
27:   (0 rts)                            空リストの CAR は OK
28:   (1 エラー処理) )                   空リストの CAR はエラー
29:  (!*4 (bsr taginvc delinvc))         deref サブルーチンコール
30:  (    (mov car0)
31:       (boc car0 mdr1)
32:       ldrpr
33:       (br tagcase car-case))         deref した結果でもう一度分類をする
```

図 3 典型的なマイクロプログラム
Fig. 3 A sample microprogram.

という式は，フレッシュな中間変数 temp を用意し，

```
{! _temp _{bar}} {foo _temp}
```

のように変形する．構造コピーの途中で直接ネストして述語呼び出しをすると，その中で選択点がスタックに積まれスタックフレームが壊される可能性がある．上記のような変形により単一化や述語引数中の述語呼び出しをボディのトップレベルで実行できる．

変形後，この単一化式は，

```
pcall-in-unif bar
true
put-pop R
```

とコンパイルされる．pcall-in-unif は，その引数の述語定義をコールする．関数型述語は返す値をスタックにプッシュし pcall-in-unif の次の次の命令へリターンする．返す値を指定されていない述語は pcall-in-unif の次の命令（true）へリターンする．ボディのない述語の値である#t が true によりプッシュされる．put-pop は，スタックトップの値をポップしレジスタ R へ格納する．

5.3 そ の 他

TAO86 では論理変数が構文上区別されていたため，論理変数の評価においてのみ invs を deref するという最適化が可能であった．TAO では論理変数が区別されないため，現在の実装では強制評価式として 1 回でも現れた変数は，それ以降の出現ですべて deref を行う．

6. 関 連 研 究

本研究のように Lisp の枠組みを保持したまま融合を狙うという立場に近い研究は，あまり多くはない．ここでは幅広い観点から関連研究との比較を行う．

前述の LM-Prolog[6] は，Lisp Machine Lisp 上で Prolog との融合を図った言語である．LM-Prolog から Lisp を呼ぶには，たとえば，

```
(lisp-value LOGVAR FORM
            &optional MODE)
```

という式を実行する．Lisp の式 FORM を実行しその結果を LOGVAR と単一化する．ここで FORM の中に UNDEF が含まれるときにどのようなデータ変換を行うかを MODE で指定する（たとえば，UNDEF を Flavors のインスタンスに変換できる）．このような変換は必然的にデータのコピーをともなう処理となる．TAO では，Lisp と論理型計算の間でデータは完全に共通であり何らの変換も必要としない．

値を返す述語という点に着目すると，RELFUN [15] は Prolog の計算モデルをできるだけ保ったまま関数型計算を融合するという立場の言語であり，TAO の関数型述語とほぼ同等の記述が可能である．RELFUN は Prolog がベースであるからデータに関し本論文で述べたような問題は生じない．しかし当然のことながら，破壊的な代入など Lisp 特有の計算を行うことはできない．

最後に，論理型言語の内部実装などに現れるデータの扱いについて考える．後戻りと構造データに関して最も問題となるのは bagof の実装である．多くの処理系では，特殊な assert を使ってコード領域に解を保持する．また，通常の bagof はリストの形で解を集めるが，Mercury [16] では汎用の述語（Collector と呼ぶ）を使って解を集めることができる．このために通常のグローバルスタックと中間解保持用のグローバルスタックを用意し，Collector を動作させる前にグローバルスタックをスワップするという技法を用いている．これらは言語の枠外（つまり処理系内部）で実現せざるをえない．一方，TAO ではグローバルスタックではなくヒープによってメモリを管理しており，破壊的代入を使えば後戻りを越えてデータを保持することは容易である．TAO ではユーザ定義だけで Mercury のような bagof を実現することも可能である．

7. おわりに

Lisp に論理型言語を融合する際に，特にデータに関して生じる問題について議論した．

Lisp と論理型計算の枠組みをできるかぎり保持するという立場から，UNDEF を即値とし REF を処理系が自動的にたどるという方針のもと，まず TAO86 を提案した．TAO86 を実際に使用し，破壊的代入に関する問題および論理型計算の生成したデータの利用に関する問題があることを明らかにした．これらの問題を解決するために次のような特徴を持つ言語 TAO を提案した．

- 値を返す述語を導入することにより，REF を陽に扱うことは避けつつも REF を含んだデータを扱うことが可能となった．
- `deref` との組合せにより破壊的代入の機能が向上した．
- 副作用を取り消すための機構を持つ．
- パターンマッチによる UNDEF の同一性判定が可能である．

また，専用マシン上の実装に関してタグチェックの方法などについて述べ，本方式が大きなオーバヘッドとならないことを示した．

謝辞 本論文執筆にあたりご支援をいただいた NTT ドコモネットワーク研究所今井和雄所長に感謝します．PRO 研究会において議論に参加していただいた方々と有益なコメントをいただいた査読者に感謝します．

参考文献

1) 日本 Lisp ユーザ会：*Lisp User Group Meeting Japan* (2000).
2) Bellia, M. and Levi, G.: The Relation between Logic and Functional Languages: A Survey, *J. Logic Programming*, Vol.3, No.3, pp.217–236 (1986).
3) Hanus, M.: The Integration of Functions into Logic Programming: From Theory to Practice, *J. Logic Programming*, Vol.19&20, pp.583–628 (1994).
4) Hanus, M.: A Unified Computation Model for Functional and Logic Programming, *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.80–93 (1997).
5) Hanus, M.: Curry An Integrated Functional Logic Languages. http://www.informatik.uni-kiel.de/~curry/.
6) Kahn, K. and Carlsson, M.: How to implement Prolog on a LISP Machine, *Implementations of PROLOG*, Campbell, J.(Ed.), pp.117–134, Ellis Horwood (1984).
7) Mellish, C. and Hardy, S.: Integrating Prolog in the POPLOG environment, *Implementations of PROLOG*, Campbell, J.(Ed.), Ellis Horwood (1984).（最新版は http://www.cogs.susx.ac.uk/users/adrianh/poplog.html）.
8) 山崎憲一，奥乃 博，竹内郁雄：TAO における論理型プログラミングとその処理方式，情報処理学会論文誌，Vol.32, No.9, pp.1090–1101 (1991).
9) 山崎憲一，吉田雅治，天海良治，竹内郁雄：実行機構の類似性に着目した関数型言語と論理型言語の融合，情報処理学会論文誌，Vol.40, No.6, pp.2743–2754 (1999).

10) Warren, D.: An abstract Prolog instruction set, Technical Report Technical Note 309, SRI International (1983).
11) Aït-Kaci, H.: *Warren's Abstract Machine*, MIT Press (1991).
12) Budd, T.A.: *Multiparadigm Programming in Leda*, Addison-Wesley (1995).
13) 山崎憲一，吉田雅治，天海良治，竹内郁雄：マルチパラダイム言語 TAO における論理型プログラム処理系の実装，情報処理学会論文誌，Vol.41, No.1, pp.136–147 (2000).
14) 吉田雅治，竹内郁雄，天海良治，山崎憲一：新しい記号処理カーネル SILENT の設計，情報処理学会計算機アーキテクチャ研究会 84-1 (1990).
15) Boley, H.: Functional-logic integration via minimal reciprocal extensions, *Theoretical Computer Science*, Vol.212, pp.77–99 (1999).
16) Somogyi, Z., Henderson, F. and Conway, T.: The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language, *J. Logic Programming*, Vol.29, No.1–3, pp.17–64 (1996).

付　　録

A.1 TAO のタグ割当て一覧

タグは 7 ビットで構成される（bit7 は GC のマークビット）．bit6 は tage（tag extension）と呼ばれ，タグの意味を修整するために用いられる．残り 6 ビットのうち上位 3 ビット（tag5-3）により，タグ空間は 8 個のセクションに分割される（**図 4** 参照）．各セクションは次のような意味を持つ．

0：即値アトム
1：即値でない数，文字列，シンボル
2：複雑なデータ構造
3：プログラムと作用素
4：特殊（undef，invisible，自己記述）
5：乙種データ（ファーストクラスでないデータ）
6：コンカレンシとバッファ
7：コンス類（CAR/CDR が可能なデータ，cadable と呼ぶ）

セクション 0 とセクション 4，つまり tag4-3 が 0 のタグは，タグチェック付き読み出し命令 boc で読み出し番地として使われると，読み出しがハードウェアにより禁止される（付録 A.2）．

○セクション 0：即値アトム（boc できない）
00 `dnil` `#f` など
01 `fixnum`（fixed point number）
02 `flonum`（floating point number）
03 `char`（character）

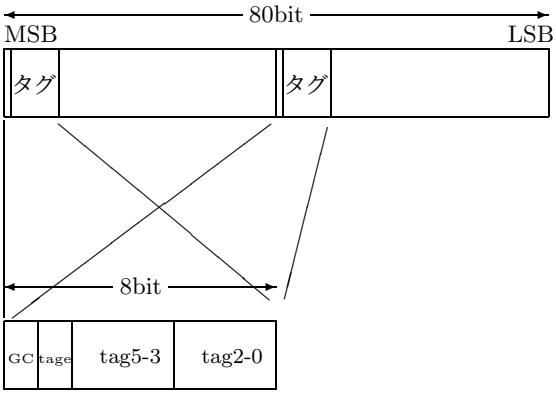


図 4 TAO の内部データ表現
Fig. 4 Internal data representation of TAO.

04 `special` `#t` など
05 `stkpt`（stack pointer）
06 `empty` 空リスト，空 cqueue
07 `keyword`

○セクション 1：即値でない数，文字列，シンボル
08 `doublefixnum`
09 `doubleflonum`
0A `complex`
0B `bignum`
0C 空き（rational? or other type number or string）
0D `str`
0E `substr`
0F `symbol`

○セクション 2：複雑なデータ構造
10 `vector`
11 `matrix`
12 `cube`（3 次元の配列）
13 空き
14 `symtab`
15 `package`
16 `context`
17 `object`

○セクション 3：プログラムと作用素
18 `wrapper`
19 `form`
1A `chore`
1B 空き
1C `fnskel`（function skeleton）
1D `predskel`（predicate skeleton）
1E `clamp` closure（tage 付き）/amphibious（tage なし）
1F `dolamb`（downward lambda = `op*`作用素）

○セクション 4：特殊 I（undef，invisible，自己記述

など．boc できない）
20 undef（undefined）
21 lazy
22 invm（invisible pointer to memory）
23 invs（invisible pointer to stack）
24 [reserved for logic paradigm]
25 elem（array element，自己記述タグ，ポインタとしては無意味）
26 free
27 mark

○セクション 5：特殊 II（システム内部専用．ユーザには見えない）
28 buddy（buddy 領域へのポインタ）
29 pkghash（package につく hash table，特殊マーキングが必要）
2A macroskel（macro skeleton）
2B objcore
2C constructskel
2D rpftag
2E gcbt
2F dytes

○セクション 6：コンカレンシとバッファ
30 locker
31 semaphore
32 mailbox
33 process
34 eventbox
35 buffer
36 cqueue
37 stack

○セクション 7：コンス類（cadable）
38 cons 例：(a b c)
39 pcons（predicate cons）例：{p _x}
3A mcons（message cons）例：[x (msg: y)]
3B ucons（unification cons）例：{! _x _y}
3C hatcons（hat cons）
例：(^if ^test (car ^))
3D assign（assign cons）例：(!x y)
3E selfass（self-assign cons）例：(!!+ !x 123)
3F mvass（multiple-value-assign cons）
例：(.!(x y z) 1 3 4)

A.2 タグに関連する特殊な命令

○タグチェック付き読み出し操作命令（boc，rpic）
読み出しアドレスに指定したレジスタのタグの tag4-3 が 0 であったら読み出しをしない．チェックをしても実行速度は低下しない．

○ TAGALU，陰の比較器
TAGALU は，タグの演算を行うための ALU である．TAGALU の主な機能は，タグの入換え，GC ビットのセット/クリア，タグのマスク/クリア/OR である．TAGALU には，データ比較器が並列して置かれている．この比較器は，マイクロコードの指定に関係なく，つねに TAGALU への 2 つの入力（Asrc，Bsrc）を比較し続けている．この比較器の結果を,「陰の演算結果」と呼び，次に述べるように特定の分岐命令を用いて参照できる．

A.3 タグによる分岐条件一覧

SILENT では，きわめて多くの分岐条件を指定することができるが，そのうちタグに関係する分岐条件を以下に示す．分岐には，多方向分岐と，2 方向分岐がある．

○多方向分岐条件
多方向分岐では，指定したアドレス+N にジャンプする．以下の説明では，+N とその条件を記述する．

tag5-0　TAGALU の tag5-0 で 64 方向分岐．
tagcase　TAGALU の tag5-3 で 3 方向分岐．
+0: 以下の+1〜+2 の条件以外の場合 (special)
+1: if tag5 = 0 (atom)
+2: if tag5-3 = #b111 (cadable)
sptagcase　<sp>の tag5-3 で 3 方向分岐．
+0: 以下の+1〜+2 の条件以外の場合 (special)
+1: if tag5 = 0 (atom)
+2: if tag5-3 = #b111 (cadable)
taginvc　TAGALU の tag5-0 で 6 方向分岐．
+0: 以下の+1〜+5 の条件以外の場合
+1: if tag5-0 = #b100000 (undef)
+2: if tag5-0 = #b100001 (lazy)
+3: if tag5-0 = #b100010 (invm)
+4: if tag5-0 = #b100011 (invs)
+5: if tag5-0 = #b000000 (dnil)
sptaginvc　<sp>の tag5-0 で 6 方向分岐．
+0: 以下の+1〜+5 の条件以外の場合
+1: if tag5-0 = #b100000 (undef)
+2: if tag5-0 = #b100001 (lazy)
+3: if tag5-0 = #b100010 (invm)
+4: if tag5-0 = #b100011 (invs)
+5: if tag5-0 = #b000000 (dnil)
allgel　TAGALU と ALU の組合せで 3 方向分岐，39 ビット比較になる．若い番地から順に greater，equal，less．Lisp データのソートに必要．

○ 2 方向分岐条件
TAGALU の結果，スタックトップレジスタ（<sp>）のタグの値，暗黙の比較の結果による条件．
tag7　TAGALU の bit7（GC など）が立っている．
sptag6　<sp>のタグの bit6（tage）が立っている．
tag6　TAGALU の bit6（tage）が立っている．
tag5　TAGALU の bit5，atom でない．
tagnboc　TAGALU の tag4-3 が#b00，boc できな

い.

sptagncadbl <sp>の tag5-3 が#b111 でない．cadable でない.

tagncadbl TAGALU の tag5-3 が#b111 でない．cadable でない.

tagnil TAGALU の bit5-0 が全部ゼロ，nil である.

sptagnfix <sp>のタグが fixnum でない．つまり，tag5-0 が#b000001 でない.

taginv TAGALUが invisible pointer，つまり tag5-2 が#b1000.

sptaginv <sp>のタグが invisible pointer，つまり tag5-2 が#b1000.

tagnop TAGALU の tag5-2=#b0111 でない．operator でない.

tag/= Asrc と Bsrc の tag5-0 が等しくない.（陰の演算結果）

tag= Asrc と Bsrc の tag5-0 が等しい.（陰の演算結果）

tage/= Asrc と Bsrc の tag6-0 が等しくない.（陰の演算結果）

tage= Asrc と Bsrc の tag6-0 が等しい.（陰の演算結果）

(平成 12 年 10 月 25 日受付)
(平成 13 年 2 月 20 日採録)

山崎 憲一（正会員）

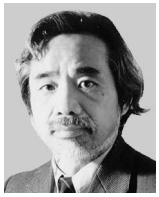
1961 年生．1984 年東北大学工学部通信工学科卒業．1986 年同大学院情報工学科修士課程修了．同年，日本電信電話（株）入社．2000 年 9 月から NTT ドコモネットワーク研究所主幹研究員．主として記号処理言語，日本語文書処理，ネットワークの研究に従事．ACM 会員.

吉田 雅治（正会員）

1953 年生．1976 年千葉大学工学部電気工学科卒業．1978 年同大学院工学研究科修士課程修了．同年，日本電信電話公社入社．現在，東日本電信電話（株）企画部所属，エヌ・ティ・ティアイティ（株）ファイル・映像ソリューション事業部に出向中．第 2 技術部担当部長．主として画像系のシステム開発に従事．ユーログラフィックス，電子情報通信学会会員.

天海 良治（正会員）

1959 年生．1983 年電気通信大学電気通信学部計算機科学科卒業．1985 年同大学院修士課程修了．同年日本電信電話（株）入社．以来，プログラミングパラダイム，計算機アーキテクチャ，計算機ネットワークの研究に従事．現在 NTT 未来ねっと研究所ネットワークインテリジェンス研究部主任研究員．1994 年度山下記念研究賞．日本ソフトウェア科学会会員.



竹内 郁雄（正会員）

1946 年生．1969 年東京大学理学部数学科卒業．1971 同大学院理学系研究科修士課程修了．同年，日本電信電話公社入社．日本電信電話（株）基礎研究所，ソフトウェア研究所を経て，1997 年から電気通信大学情報工学科教授．主として記号処理言語やシステムの研究に従事．工学博士．1990 年情報処理学会論文賞．ACM，日本ソフトウェア科学会各会員.