TAOにおける論理型プログラミングとその処理方式†

山崎憲一†† 奥乃博††† 竹内郁雄†††

TAOは記号処理を伴う大規模なアプリケーションの記述のために開発されたマルチパラダイム言語であり，関数型，論理型，オブジェクト指向の3つのパラダイムから構成される．本論文ではこのうち論理型パラダイムの機能と実現方法および他パラダイムとの融合方法について述べる．本方式では，述語を関数の一種としたこと，すべてのパラダイムのデータ構造を共通化したことなどによりこれまでの融合型言語に比べ，より簡単にパラダイム間の相互呼び出しが可能となった．また実用性を重視して設計，実装しインタプリタで10 KLIPS，コンパイラで40 KLIPSを達成した．

1. はじめに

エキスパートシステムに代表される人工知能技術を用いたアプリケーションは年々巨大化し複雑になりつつある．このようなシステムは手続き的な処理，パターンマッチ，複数解の探索などさまざまな構成要素からなるが，各要素の記述に適したパラダイムは異なっていることが多い．これらをLisp，Prologなど単一のパラダイムからなる言語を用いて記述すると，プログラムが理解し難くなるだけでなく，実行効率も悪化する．TAO[6]はこのような問題を解決するために開発された言語であり，次のような特徴を持つ．

- マルチパラダイム言語である．
- インタプリタが実用に耐える速度で動作する．
- マルチプロセス，マルチユーザに対応する．

マルチパラダイムとしてはLispを基本に，論理型パラダイム，オブジェクト指向パラダイム，番地計算型パラダイムなどを融合している．ユーザはこれらの中から問題の記述に最も適したパラダイムを選択できる．

本論文ではTAOの論理型パラダイムについて，およびそれと他のパラダイムとの融合部分について，機能と実現方法を述べる．TAOでは論理型言語の述語を関数の一種と考えることでLispとの融合を行った．これにより，これまでの論理型言語に対する融合型言語[1),2),5)]に比べ，より均質で密に結合した言語となっている．またそれらにおいては実用的なレベルに達した処理系は少なかったが，TAOは実用に耐えうることを目標に設計された．現在，TAOは専用計算機ELIS[13)]上に実現されており，インタプリタで10 KLIPS，コンパイル・コードで40 KLIPSを達成している．融合型言語では絶対速度の他にパラダイム間の速度比が小さいことが重要である．実行速度が重視されるシステムの記述においては最高速のパラダイムを選択せざるを得ないが，各パラダイムがバランスしていれば問題に適したパラダイムを選択できる．TAOは速度比を小さくすることを目標に設計され，論理型パラダイムの実行速度は，Lisp，オブジェクト指向と比較して，2倍程度の速度低下に抑えられた．

なお，本論文の構成は以下のとおりである．第2章で論理型言語とLispを融合する際の問題点を指摘し，TAOの言語機能を述べることで，1つの解決法を提示する．第3章では，論理型パラダイムと関数型パラダイムとの相互呼び出しの詳細，およびオブジェクト指向との融合の機構について述べる．第4章で実現方法について述べ，第5章で処理系の評価を行う．

以下ではDEC-10 Prologに準拠したPrologを単にPrologと呼ぶ．また→は実行の結果返される値を表す．

2. TAOの論理型プログラミング

TAOでは関数型パラダイムおよびオブジェクト指向パラダイムがそれぞれ論理型パラダイムと融合されているが，各融合部分では，データ，変数の意味，バックトラックの動作などについて共通しているところが多い．まず本章において，より理解しやすいと思われる関数型との融合について述べ，オブジェクト指向との融合については次章で説明する．

2.1 Lispと論理型プログラミングの融合

異なる言語を融合するには，融合のレベルが問題となる．PrologからCの手続きを呼び出せる処理系な

† Logic Programming in TAO and Its Implementation by KENICHI YAMAZAKI (NTT Software Laboratories), HIROSHI OKUNO and IKUO TAKEUCHI (NTT Basic Research Laboratories).

†† NTTソフトウェア研究所
††† NTT基礎研究所

どがあるが，容易なパラダイム選択を可能とするためにはより細かい粒度での融合が望ましい．TAOでは論理型言語と関数型言語が計算の最小単位のレベルで相互呼び出しができ，また両者がまったく同じデータ構造を扱えることを目標とした．これを実現するには次のことを検討する必要がある．

- 関数と述語の関係．
- 変数の扱い．
- 論理型言語特有のデータ構造の処理．
- 非決定性（バックトラック）の機構．

以下ではこれらの問題の解決法について述べるが，まず簡単な例を用いて，TAOの構文をPrologと対応付けて説明する．2つのリストを連結する述語appendをPrologで記述した場合には次のようになる．

```
append([], X, X).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

これをTAOで記述すると次のようになる．

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

両者の対応はほぼ明らかであろう．このようにTAOの構文はS式であり，変数はアンダースコア（_）をつけて表す．appendを呼び出すには，

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

とする．変数の宣言（この場合はlet）が必要な点がPrologとの最大の相違点である．

2.2 レゾルバ—関数としての論理型計算

論理型プログラミングは，演算的側面から眺めると，ユニフィケーションと非決定的実行（バックトラック），という2つの基本機能に分解できる．TAOではこれらを2つの新たなタイプの関数，**レゾルバ**（*resolver*）によって表現する．すなわち，

- **Uレゾルバ**：ユニフィケーションを行う．
- **Cレゾルバ**：非決定的実行を行う．

レゾルバは関数の一種である．Lispの関数に値を適用（apply）することにより（関数型の）計算が行われるのと同様に，レゾルバに値を適用することにより論理型の計算が行われる．Uレゾルバ式の構文は

```
(&+パターン [(&aux 補助変数…)] ボディ…)
```

である（…は繰り返し，[] は省略可能部分を表す）．&auxは補助変数の宣言である．これは次のようなLispのlambda式の構文と対応づけられる．

```
(lambda (引数…) ボディ…)
```

lambda式が無名の関数を表すようにUレゾルバ式は無名のUレゾルバを表す．Cレゾルバ式の構文は

```
(Hclauses Uレゾルバ…)
```

である．これらに値を適用するには例えば

```
(レゾルバ式 . 引数パターン)
```

というフォームを評価する．なお関数，レゾルバなどはTAOではオブジェクトの一種であり，値が適用可能であることから *applobj* (*applicable object*) と呼ばれる．

次に各レゾルバの動作について述べる．CレゾルバはPrologにおける節の非決定的選択と同様の動作をする．Cレゾルバに値が適用されると，まずボディのUレゾルバのリストのcdrで選択点を作り，carのUレゾルバに引数リストを適用する．その結果がnil以外であれば，それをCレゾルバの値として返す．nilであれば，バックトラックが起こる．すなわち，直前に作った選択点のUレゾルバリストについて上記を繰り返す．

UレゾルバはPrologにおける節に対応する．これに値が適用されるとパターンと引数リストのユニフィケーションが行われ，失敗するとユニフィケーションでの代入が取り消された後，nilが返される．これはPrologでの浅いバックトラックに相当する．一方成功すると，ボディがなければ t が返される．もしボディがあればボディリストの要素をその評価値がnilでない限り順に評価し続け，最後の要素の評価値を返す．ボディの要素がnilを返すと直前の選択点までのユニフィケーションが取り消され，その選択点を作ったCレゾルバにnilが返される．これは深いバックトラックである．

このようにレゾルバは関数と同様に必ず値を返す．その値がnilならレゾルバの実行の失敗，非nilなら成功を意味し，これによりバックトラックが制御される．レゾルバの例を挙げる．次のCレゾルバ式を評価すると b が印刷され，b が返される．

```
((Hclauses (&+ (a) (print 'a))
           (&+ (_x) (print _x))) b) → b
```

レゾルバには関数defineを用いて名前を付けることができる．例えば先に挙げたappendは次のように書ける．

```
(define append
  (Hclauses (&+ (() _x _x))
            (&+ ((_a . _x) _y (_a . _z))
```

```
                (append _x _y _z))))
```

さらにこれは，特殊関数 assert を使って 2.1 節の例のように書くことができる．このように名前づけされたレゾルバを述語と呼ぶ．なお，実際には append は TAO のシステム関数であり，再定義はできない．このため，また述語と関数の区別を容易にするため，慣例として述語名は & で始める．したがって 2.1 節の例ではユーザは通常 &append という名前で定義する．

2.3 変数とユニフィケーション

論理変数はアンダースコアで始まるシンボルである．論理変数は Lisp 変数と同様に宣言が必要である．ヘッド部のパターンに現れる変数はそこに現れることで宣言がなされるが，ボディで初めて現れる変数は &aux を用いて宣言する．

```
(define &foo (&+ () (&aux _x) (print _x)))
```

論理変数の初期値は undef という型を持つ値である．undef はデータが未定義な状態を表しユニフィケーションで値を代入することができる．なお，assert を使うと変数は自動的に宣言される．次の述語は先の例の述語と等価である．

```
(assert (&foo) (print _x))
```

変数のスコープは Lisp の変数と同様であり，例えば declare を使って大域変数であることを宣言できる．

```
(defun foo (&aux _*x*)
    (declare (special _*x*))
    (!_*x* 3)
    (&test))

(assert (&test)
        (declare (special _*x*))
        (&write _*x*))

(assert (&write _x) (print _x))
```

ここで “(!**左辺 右辺**)” は**右辺**を評価し**左辺**に代入 (Common Lisp の SETF) をする．この例で (foo) を評価すると 3 が表示される．

&+ によるUレゾルバでは変数のスコープは静的でそのレゾルバに閉じるが，&+ の代わりに &+dyn を使うとスコープを動的にすることもできる．&+dyn はスコープの境界を作らず，それを呼び出した親の環境を自分の環境とする．

論理変数にはすべてのデータ型を代入できるが，ベクタ，配列などの要素にユニフィケーションでアクセスすることはできない．ユニフィケーションによってデータ構造を分解生成できるのはリストだけである．なお，プログラム自身がリストで表されているため，通常の Prolog ではできない次のような任意個の引数の受け渡しができる．

```
(assert (&foo _x . _rest))
```

例えば (&foo 1 2 3 4) に対しては _x に 1 が，_rest には (2 3 4) が代入される．

2.4 バックトラック

Cレゾルバによるバックトラック機構は既に述べたように Prolog のそれにほぼ対応するが，レゾルバが関数から呼び出された場合は若干異なる．あるレゾルバがレゾルバから呼び出された場合は，制御が呼び出した側に戻ってもバックトラックが起きる可能性があるため選択点は残される．しかし関数から呼び出された場合，呼び出されてから値を返すまでに生成された選択点の情報はすべて消去されるのである．以下の例で説明しよう．

```
(assert (&foo) (print 'a))
(assert (&foo) (print 'b))

(assert (&fail) nil)

(defun func () (&foo) (&fail))

(assert (&pred) (&foo) (&fail))
```

この例では (func) の実行により *a* が印刷されるが，&foo の選択点は消去され，&fail の失敗によって &foo にバックトラックが及ぶことはなく，(&fail) の値である nil が返される．一方 (&pred) の実行では *a* と *b* が印刷された後 (&pred) の失敗を意味する nil が返される．

関数とレゾルバが入れ子になっても同様である．上の例にさらに，

```
(defun call-foo () (&foo))

(assert (&nest-pred) (call-foo) (&fail))
```

が加わったとして，(&nest-pred) を実行すると *a* のみが印刷され nil が返される．これは call-foo が関数であるために，(&foo) の実行後 call-foo に戻った時に選択点の情報が消去されるからである．

Prolog のカット “!” に対応する機構も用意されており，ボディに “!” と書く．TAO のカットは Prolog

のそれとは異なり自分が含まれる（述語でなく）Cレゾルバの作った選択点までしか消去しないため，例えば

```
(assert (&foo) ((Hclauses (&+ () !) (&+ ()))))
(assert (&foo))
```

という述語ではカットにより消去される選択点はHclausesで作られた無名Cレゾルバのものだけであり，&fooの選択点は消去されない．

以上のようにUレゾルバのボディの要素はLispの評価とは異なる点がある．Uレゾルバのボディでの評価は特に**R評価**と呼ばれる．R評価とLispの意味での評価の違いはこれまで述べてきたことも含めて次の3点がある．第1は**選択点に関する意味**である．述語を評価するとそれが終了した時点で，評価中に生成された選択点は消去される．一方R評価では選択点は残され，バックトラックが起こる可能性もある．第2は**カット(!)の意味**である．カットを評価した場合にはシンボル“!”の評価とみなされ，それに束縛されている値があれば返される．第3は**未定義述語に対する動作**である．未定義述語をR評価すると通常はエラーとなる．これは述語名のタイプミスなどの発見を容易にする．しかし未定義述語の実行を失敗とすることで否定の意味を表したい場合には，

```
(negation-as-failure t)
```

とする．これによりモードが変わり未定義述語のR評価は失敗してバックトラックが起こる．なお未定義述語を評価すると常にエラーとなる．

計算が評価によって進む世界とR評価によって進む世界を分けて考えると理解しやすい．以降では評価によって計算が進むプログラム部分をLisp世界という．

3. 他パラダイムとの融合

関数型との融合については前章で述べたとおりであるが，主にPrologとの対応で説明した．本章では融合部分に特有な点について説明する．また，オブジェクト指向との融合について説明する．

3.1 Lisp世界からのレゾルバの呼び出し

Lisp世界からレゾルバを呼び出すには単に評価をすれば良い．レゾルバが返す値は成功したか否かを示す．別に値を得たい場合には論理変数をLisp世界側で宣言しそれをレゾルバに渡す．ただし，レゾルバの処理した結果をLisp世界側で使いたい場合は注意が必要である．

```
(defvar *var* nil)
(assert (&foo)
        (&append _x _y (a b))
        (progn (print _x) (push _x *var*) t)
        nil)
```

(&foo)を実行すると，(), (a), (a b)が印刷されるが*var*には次のような値が代入される．

```
((#{undef}125732 . #{undef}125729)
 (#{undef}125732 . #{undef}125729)
 ())
```

(#{undef}125732などはundefデータを表す．)これはバックトラックによりユニフィケーションが戻され，結果的にリストが破壊されるからである．バックトラックにより戻される可能性のあるリストはバックトラックの起こる前にコピーする必要がある．ただし，典型的にはレゾルバは成功してLisp世界に戻るという使い方をするために，このような配慮は必要ない．

3.2 論理型プログラムからのLisp世界の呼び出し

レゾルバからLisp世界を呼び出すには2つの方法がある．第1はUレゾルバのボディで関数などのLisp世界を呼び出すフォームをR評価する方法である．R評価した結果がnilであればバックトラックが起きる．例えば大きい方の引数を印刷する述語は次のようになる．

```
(assert (&print-greater _x _y) (> _x _y)
                               (print _x))
(assert (&print-greater _x _y) (< _x _y)
                               (print _y))
```

レゾルバのボディにおけるR評価とPrologの節のボディの実行の違いを明確にするために次の例を考える．

```
p(X) :- X.                                  (1)

(assert (p _x) _x)                          (2)
(assert (p _x) , _x)                        (3)
(assert (p _x) (eval _x))                   (4)
```

Prologで記述された節(1)では，Xの値を述語として解釈しそれを呼び出す．一方TAOでは，節(2)の場合，ボディ中のシンボル_xが評価されその束縛値が返される．この値がnilならバックトラックが起き，それ以外の時は成功する．このような違いは変数に対する意味が異なるために生ずる．Prologには変数名とその値という考え方はないが，TAOでは変数

名はシンボルであり，それを評価することによって初めてその束縛値が得られるのである．節(1)と同様のことを TAO で行うために記法 "," が用意されており，節(3)のように書く．これにより，_x の束縛値がこのUレゾルバの環境中でR評価される．節(4)は節(3)と似ているが，eval は関数であるから，2.4 節で述べたように eval 内へバックトラックが及ぶことはない．

レゾルバから Lisp 世界を呼び出す第2の方法は，ユニフィケーションからの呼び出しである．これにはパターンの前に "," を書く．ユニフィケーションが行われる直前に "," のついたパターンが（R評価でなく）評価され，その結果に対してユニフィケーションが行われる．例えば階乗を計算する述語は次のように書ける．

```
(assert (&fact 0 1))
(assert (&fact _n _f)
        (&fact ,(1- _n) _f1)
        (== _f ,(* _n _f1)))
```

ただし，==は次のように定義されている．

```
(assert (== _x _x))
```

"," を付けたパターンをパターンの中に書いてもよい．

```
(let (_x) (== _x (,(+ 1 1))) _x)  →  (2)
```

アンダースコアで始まらない通常の変数はユニフィケーションにおいてはパターンとして扱われるが，"," を付けることでこれを評価してその値を参照することができる．さらに "," によって例えば関数 aref を呼び出して配列の値を参照できる．ただしこれは評価した値を参照するだけであり，その値が undef であってもユニフィケーションによる代入はできない．undef は Lisp 世界では即値として扱われるからである．なお，"," により Prolog で多用される中間的な変数を削減できることは先の階乗の例でも明らかであろう．

3.3 オブジェクト指向との融合

Prolog では述語名のスコープは大域的であるため，モジュール化や階層的なルールの記述などが困難となる．TAO ではこれを2つの点から解決している．1つはシンボルのパッケージ機構である．述語名はシンボルであるから，この機構によりあるパッケージの述語を他のパッケージから隠すことができる．この際ユニフィケーションで扱われるシンボルも別パッケージになる．

もう1つの解決法が本節で述べるオブジェクト指向との融合である（以下論理型オブジェクト指向プログラミングと呼ぶ）．オブジェクト指向パラダイムは手続きとデータの抽象化のための枠組を与える．我々は述語は手続きであり，グラウンドユニット節はデータであると考えてこの枠組を論理型プログラミングに適用した．論理型オブジェクト指向プログラミングには論理メソッドとインスタンスファクトが導入された．直観的には論理メソッドはオブジェクト指向のメソッドに対応し，インスタンスファクトはインスタンス変数に対応する．すなわち，論理メソッドはあるクラスに属する全インスタンスから参照できる述語であり，インスタンスファクトはあるインスタンスから参照できるグラウンドユニット節である．論理メソッドの構文は

```
(deflogic-method （クラス名［コンストラクタ］
                    論理メソッド名）
  パターン［(&aux 補助変数…)］ボディ…)
```

である．また次のようなマクロが用意されている．

```
(within-class クラス名
  (assert (論理メソッド名．パターン)ボディ…))
```

TAO のオブジェクト指向プログラミング[9]および論理型オブジェクト指向プログラミングにおいてメッセージ送信はブラケット ("[ ]") で囲まれた式で表現される．次のような式でインスタンスにメッセージを送り論理メソッドを呼び出す．

［インスタンス論理メソッド名．引数パターン］

これによりインスタンスのクラスに定義されている論理メソッドが探され，そのヘッドのパターンと引数パターンとがユニファイされる．成功すると変数 self がインスタンス自身に束縛され，ボディが実行される．ボディの実行はレゾルバのそれとまったく同じである．失敗した場合にはレゾルバと同様に直前の選択点にバックトラックする．論理メソッドは，呼び出される節がインスタンスによって決定されるようなレゾルバであり，変数の意味，扱えるデータ，カットの動作，バックトラックの順序などはすべてレゾルバのそれと同じである．例として次のプログラムを考える．

```
(defclass カラス () () () :logical-class)
(within-class カラス (assert (色 黒)))

(defclass ツバメ () () ():logical-class)
(within-class ツバメ (assert (色 白と黒)))
```

defclassによってクラスを宣言する*. “カラス”と“ツバメ”のインスタンスを生成し，論理メソッド“色”を送るとそれぞれの色がわかる.

```
(!crow (make-instance 'カラス))
(!swallow (make-instance 'ツバメ))

(let (_col) [crow 色 _col] _col)      →黒
(let (_col) [swallow 色 _col] _col) →白と黒
```

論理メソッドはメソッド同様に継承されうる．同名の論理メソッドが継承関係にある複数のクラスに定義されていた場合は，下位クラスから上位方向にバックトラックによって探索される．つまり上位の論理メソッドほど後で宣言されたように動作する．この宣言順序はコンストラクタを用いて変更できる．最後に宣言される(:after)，最初に宣言される(:before)，より上位の論理メソッドを無視する(:cut)，という動作をする3つのコンストラクタがある.

インスタンスファクトはインスタンスに&assertという論理メソッドを次のように送ることにより動的に定義する.

[インスタンス &assert . パターン]

ただし，パターンは undef のデータを含んではならない．すなわちグラウンドでなければならない．インスタンスファクトの参照は論理メソッド“&”を送る.

[インスタンス & . 引数パターン]

これによりインスタンスファクトのパターンとユニフィケーションが行われ，成功すれば t が，失敗すれば nil が返される．もちろん，ユニファイ可能なインスタンスファクトが複数あれば選択点が作られる．インスタンスファクトの例を示す.

```
[crow &assert 名前 三郎]
(let (_x) [crow & 名前 _x] _x) →三郎
(let (_x _y) [crow & _x _y] (list _x _y))
                          →(名前 三郎)
```

4. 論理型プログラミング機能の実現

インタプリタのカーネル部分はすべてマイクロプログラムで記述されている．TAOの高速性はマイクロプログラムの実装技法によるところが小さくないが，ここでは一般的なProlog処理系の実現法と特に異なる点について説明する.

4.1 ユニフィケーションと内部データ表現

内部データ表現はELISのタグアーキテクチャに適合するポインタタグを用いた，Lisp世界との相互呼び出しとデータ引き渡しを実現するために，ユニフィケーションが処理するデータとLisp世界のデータとは同じ構造を持つ必要がある．このため構造コピー方式のユニフィケーションを行い，リストの互換性を保っている．通常のPrologでは構造データはグローバルスタック上に取られる．このデータはバックトラックで捨てられるだけでなく，トップレベルでのゴール実行が成功して終了した後においても捨てられる．Prologでは実行が終了した時点で構造データが外部から指されていることはないためこの方法で十分である．しかしTAOで同様の実装をした場合，Lisp世界に渡されたポインタが捨てられたスタック領域を指してしまう可能性がある．なぜならLisp世界へ渡すという操作はバックトラックでは取り消せないからである．この問題を避けるためスタックでなくセル領域に構造データ（リスト）を取る．これにより，例えばユニフィケーションによって作られる構造データを大域変数へ代入した場合，実行が成功した後でも正しいデータにアクセスできる．ただし，2.4節で述べたようにバックトラックが起こると構造データの一部が未定義状態に戻されることがある.

インタプリタでは“,”と論理変数の評価はユニフィケーション中に値が必要になった時に行われる．変数の値を得た後や“,”を評価して値を得た後にはモードが変更され，その値に論理変数のシンボルや“,”のついたフォームが含まれても単なるデータとしてユニフィケーションされる．なお，コンパイラでは“,”や論理変数はコンパイルされるために，このようなモード切替えはない.

論理型プログラム特有の変数の未定義状態を表すため，undefというタグが割り当てられている．この場合そのポインタ部(24ビット)には世代番号が入る．世代番号は未定義変数が生成される度に1ずつ増加する．未定義変数は世代番号付きで印刷されるため異なる未定義変数を表示上で区別できる.

未定義の変数同士をユニフィケーションした結果，「見えないポインタ」によりリンクが張られる．この際必ず，ローカルスタック内ではスタックの底の方向へ，またローカルスタックからセル領域の方向へリンクを張ることで，スタックのポップアップによるリンク先データの消失を避けている．見えないポインタは

* defclassの引数の意味等の詳細は文献9)を参照いただきたい.

特殊タグにより表現されている．このタグにはcar側へのポインタである inva (**in**visible **ca**r) と cdr 側へのそれである invd，およびスタックへのポインタである stkpt がある．例えば次のようなユニフィケーションをした結果，**図1**のようにポインタが張られる．

(let (_x _b _a) (== _a _b) (== _b (_x _x . _x)))

見えないポインタを含んだデータはそのまま Lisp 世界に引き渡されることもある．しかし Lisp の動的な型チェックの際，マイクロプログラムにより見えないポインタが自動的にたどられるため，ユーザはこれに関し意識する必要はない．ただしポインタをたどった結果到達した undef は即値として扱われる．undef が存在する場所は保持されないため，そこに代入することはできない．

バックトラックで変数をもとに戻すため代入の履歴を記録することをトレールという．多くの場合Prologではトレール情報は専用スタックに記録される．ELISでは唯一のハードウェアスタックは実行制御に用いられるため，トレールはリストの形式で保持している．また Prolog では直前の選択点より後で生成された変数への代入はスタック機構によりその変数が捨てられるためトレールの必要がない．しかし前述のようにTAO ではグローバルスタックの代わりにセル領域を使うため，このようなスタックの特性を利用した実装はできない．従って，セル領域への代入はすべてトレールしておき，バックトラック時にすべてをもとに戻す必要がある．このためトレールは通常の Prologに比べ頻度が高いことが予想される (5.1節参照)．

4.2 実行制御とスタックフレーム

前述のようにELISはハードウェアスタックを1本持っている．変数のバインディング，実行の制御などはすべてこのスタックだけで実現されている．

インタプリタでレゾルバが作るスタックフレームを**図2**に示す．スタックフレームは選択点と環境から構成され，前者はCレゾルバが後者はUレゾルバが作成する．環境のうち tf, ef と applobj の3つのフィールドとローカル変数の部分の形式は TAO のすべてのパラダイムで共通である．これにより論理変数と Lisp 変数が同じ扱いをうけることが可能となる．Lisp 世界から述語が呼ばれて Lisp 世界に戻った場合には，スタックトップレジスタが強制的にもとの状態に戻され，述語が残した選択点（および環境）がすべて削除される．

論理型プログラムに特に関係するレジスタには以下がある．

trail トレールリストを保持する．

tf ユニフィケーション時に親の環境を指す．

ef 現在の環境を指す．

undef 次の undef の世代番号．

カットは ef レジスタが指すところまでスタックをポップアップすることで実現される．ポップアップされる各フレームの trail の情報は連結して (Lisp の nconc) 1つのリストにされる．この時不要なトレール情報が捨てられトレールリストの圧縮がなされる．なお，末尾呼び出しの最適化はインタプリタでは行っていない．

4.3 論理メソッド

論理メソッドはオブジェクト指向のメソッド探索機構と論理型プログラムの実行機構を組み合わせることで実現されており，レゾルバをクラスによって選択する部分以外は論理型プログラムの実行と全く同じである．各クラスはメソッドテーブルと呼ばれるメソッド名とその applobj の対応表を持つ．その applobj としてメソッドでは関数が，論理メソッドではレゾルバが格納される．メッセージが送信されるとメソッドテーブルが探索されるが，最初の送信ではテーブル自身がないのでまずメソッドテーブルが作成される．こ

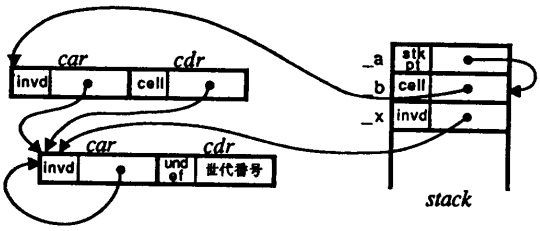

図1 データの内部表現
Fig. 1 Data representation.

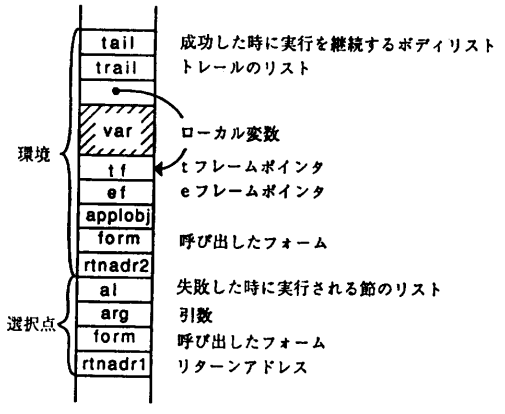

図2 インタプリタのスタックフレームの構造
Fig. 2 Stack frame of interpreter.

のとき継承関係にあるクラスは順にたどられ，同名の論理メソッドがあれば集めて1つのレゾルバが生成されメソッドテーブルに登録される．メソッドテーブル探索の結果レゾルバが見つかった場合には，変数 self の束縛をした後，論理型プログラムのインタプリタに制御を移す．以降は論理型プログラムとまったく同じように実行される．

インスタンスファクトは "&" という論理メソッドを特別に扱うことで実現されている．このメソッドが送られると，インスタンスの持つインスタンスファクト用ハッシュ表を使ってパターンが取り出され引数パターンとユニファイされる．他に候補がある場合にはレゾルバと同様の選択点をスタック上に作り，バックトラックに対応する．

4.4 コンパイラ

コンパイラはプログラムを WAM[8] 形式のバイトコードへ変換し，これをマイクロプログラムで解釈実行する方式をとる[14]．インデキシングおよび末尾呼び出し最適化を行っている．WAM と異なる点は主に

- スタック上に仮想引数レジスタを配置する．
- ユニフィケーションコードがリスト向きに最適化されている．

などである．また，スタックフレームをインタプリタおよび Lisp のそれと共通化することで一貫性を保っている．

5. 評　　価

具体的なプログラムを用いて他の処理系と比較しながら評価を行う．なお，以下，プログラムは文献[3]のものを使用した．nrev30 は長さ 30 のリストの反転，qsort は長さ 50 のあるリストのクイックソート，8 queen は 8 クイーン問題の最初の解を得るプログラムである．処理系名のあとの "(I)" はインタプリタ，"(C)" はコンパイラを表す．なお，論理型オブジェクト指向プログラミングについては現在インタプリタしかないため，簡単な評価にとどめる．

5.1 メモリ消費量

メモリ消費量について，WAM 方式の処理系である SB-Prolog[7] (以下 SB) と比較する．ローカルスタックについては TAO コンパイラも WAM を採用しているため，主に大きな差が生ずると思われるトレールスタックとヒープ領域 (SB ではグローバルスタック，TAO ではセル領域) について評価する．測定結果を表1に示す．

表 1 メモリー消費量の比較 (単位はワード，1ワードは4バイト)
Table 1 Memory consumption (in words, 1 word is 4 byte).

	nrev30		qsort		8queen	
	ヒープ	トレール	ヒープ	トレール	ヒープ	トレール
TAO(I)	936	932	756	958	2278	3188
TAO(C)	930	870	756	706	1352	1798
SB(C)	930	0	550	103	93	18

表 2 実行速度の比較 (単位はミリ秒)
Table 2 Execution speed (in milliseconds).

	nrev30	qsort	8queen
TAO (I)	48.8	51.3	376
TAO (C)	12.3	17.9	142
Quintus (C)	4.00	11.6	78.7
T/Q 比	3.08	1.54	1.81

nrev30 は決定的であるため，SB ではトレールは0になっている．一方 TAO ではセル領域への代入はすべてトレール処理を行うためこのような差が生じる．qsort は決定的なプログラムではあるが，親の環境を書き換えるユニフィケーションがあるためトレール処理が伴う．このような場合にはトレールのメモリ消費には nrev30 のような極端な差はなくなる．ヒープについては nrev 30 も qsort も極端な差はないが，バックトラックが頻繁に生じる 8queen では大きな差がある．これは次のような理由による．ヒープ領域がスタック構造になっているとバックトラック時にスタック機構により自動的にメモリが回収される．一方 TAO ではセルは使い捨てであり，GC (Garbage Collection，ゴミ集め) によって回収されるためこのような差が生じる．結局，トレールによるメモリ消費が激しいこと，ヒープ領域の消費はバックトラックが頻繁に生じない場合には大きな差はないこと，がわかる．

なお論理型オブジェクト指向プログラミングは，実装上は節選択の方法が論理型プログラミングと異なるだけであるためメモリ消費量については同じである．

5.2 実行速度

5.2.1 Prolog との比較

実行速度について，Quintus Prolog[4] コンパイラ (以下 Quintus) との比較により評価を行う．測定結果を表2に示す．Quintus は SUN-3/260 上で測定した．ここで T/Q 比とは TAO (コンパイラ) と Quintus

の実行速度比である．なお，この結果にはGCの時間は含まれていない．

まず，絶対速度について検討する．nrev30によりLIPSを求めると，TAOインタプリタは10.2K，同コンパイラは40.3K，Quintusは124Kとなっている．SUN-3はクロック25MHz，キャッシュ64KByteであり，ELISは16.6MHz，キャッシュなしである．両者のハードウェアはまったく異なるので比較は難しいが，それでもLIPS値での3倍の差は小さくはない．これはトレール処理の重さに起因するものと思われる．一方，相対値(T/Q比)はqsort，8queenでは小さくなる．これはqsort，8queenで組込みの述語が呼び出されたためと考えられる．TAOでは多くの組込み述語がマイクロコードで実装されており非常に高速に実行されるのである．組込み述語を多用するような実用的なプログラムでは両者の差は2倍以内と考えるのが妥当であろう．

本実装方式ではグローバルスタックとトレールをすべてセルを用いて表現している．このためGCの頻度が高くなり，GCを含めた総合的な実行速度は低下することが予想される．しかしProlog(あるいはWAM)のGCはヒープの圧縮(コンパクション)が必須であるのに対し，本実装方式では圧縮が不要であるためGC自身は高速である．さらに記憶空間を多くのスタックで分割するとスタックの自動移動などが必要となり，スタックを1本しか用いない本方式の方が優れている．これらの点と5.1節の結果を考え合わせると，GCを含めた総合的な実行速度について，決定的なプログラムではメモリ消費に激しい差はないためGCの速い本方式の方が有利となり，バックトラックを頻繁に起こすプログラムではメモリ消費が少なくGCの頻度が低いProlog方式の方が有利となると思われる．

5.2.2 パラダイム間の比較

パラダイム間の実行速度の差を評価するために，フィボナッチ関数をTAOの3つのパラダイムで記述し(**図3**)，測定した．それぞれのプログラムは同じアルゴリズムを用い，比較の回数なども極力同じになるように記述した．**表3**にその結果を示す．インタプリタではパラダイム間の速度比が2倍以下に押えられていることがわかる．コンパイラでは関数型が自己再帰における最適化をきめ細かく行っているために突出している．他のパラダイムでも同様の最適化は可能であるが，現時点ではまだ実装されていない．

```
(assert (&fib 0 1))
(assert (&fib 1 1))
(assert (&fib _n _f)
        (== _n1 ,(- _n 1)) (&fib _n1 _f1)
        (== _n2 ,(- _n 2)) (&fib _n2 _f2) !
        (== _f ,(+ _f1 _f2)) )

(defun fib (n)
   (cond ((= n 0) 1)
         ((= n 1) 1)
         (t (+ (fib (- n 1)) (fib (- n 2))))))

(defmethod (integer fib) ()
   (declare (type fixnum self))
   (cond ([self = 0] 1)
         ([self = 1] 1)
         (t [[[self - 1] fib] + [[self - 2] fib]]] ))
```

図3 3つのパラダイムによるフィボナッチプログラム
Fig. 3 Programs of Fibonacci written in each three paradigms.

表3 fib(20)によるパラダイム間での実行速度の比較(単位はミリ秒)
Table 3 Comparison of execution speed between paradigms by fib(20)(in milliseconds).

	論理型	関数型	オブジェクト指向
インタプリタ	2662	1353	1367
時間比	1.97	1.0	1.01
コンパイル・コード	1020	380	821
時間比	2.69	1.0	2.16

5.2.3 論理型オブジェクト指向

すでに述べたように論理メソッドは節選択部分だけが異なるレゾルバであるから，呼び出しについてのオーバヘッドが問題である．そこで上のプログラムをあるクラス(cls)の論理メソッドとして定義し直して測定した．たとえばappendは次のように定義した．

```
(within-class cls
   (assert (&append () _x _x))
   (assert (&append (_a . _x) _y (_a . _z))
           [self &append _x _y _z]))
```

表4より実行速度が15%程度低下していることがわかる．

5.3 プログラミング言語としての評価

プログラミング言語としての優劣の評価を行うことは難しいが，共通の問題を記述してその簡潔さで比較

することが多い．そこでTAOでの8queenの記述を試みた．Prologでの8queenのプログラムをTAOに単に変換するだけでなく，問題に適したパラダイムを積極的に用いると図4のようになる．例えばシンボルがリストに含まれるかを調べるために組込み関数memqを用いているが，述語を用いるのに比べ，引数の入出力の方向が逆になったり，バックトラックが発生したりしないことをより明示的に示せる．つまり，適したパラダイムを用いることによって，プログラムが理解しやすくなったと言える．また実行速度はインタプリタで117ミリ秒，コンパイラで52.4ミリ秒となり3倍程度向上する．

表4 論理メソッドのオーバヘッド（単位はミリ秒）
Table 4 Overhead of logic method.

	nrev30	qsort	8queen
述語(I)	48.8	51.3	376
論理メソッド(I)	57.1	58.0	419
速度低下率	17.0%	13.1%	11.4%

```
(assert (queen _n _l)
        (try _n ,(index _n 1 -1) nil _l nil nil) )

(assert (try _ nil _l _l _ _) !)
(assert (try _m _s _l1 _l _c _d)
        (&select _s _a _s1)
        (== _c1 ,(+ _m _a))
        (not (memq _c1 _c))
        (== _d1 ,(- _m _a))
        (not (memq _d1 _d))
        (try ,(1- _m) _s1 (_a . _l1) _l (_c1 . _c) (_d1 . _d)) )

(assert (&select (_a . _l) _a _l))
(assert (&select (_a . _l) _x (_a . _l1))
        (&select _l _x _l1) )
```

図4 TAOによる8queenプログラム
Fig. 4 8queen program written in TAO style.

6. 関連研究と議論

関数型言語と論理型言語を融合する研究としてLOGLISP[5]，POPLOG[1]などが有名である．また，Uranus[2]を融合型言語としてとらえることもできる．ESP[11]はオブジェクト指向との融合，副作用を持つなどの点でTAOと共通するところが多い．以下それぞれと比較する．LOGLISPは演繹過程にLISP簡約化を取り込んでいる．LISP簡約化は演繹を行う前に毎回行われる．簡約化は評価とは異なり，簡約化可能な部分だけを簡約化する．従って例えばxの値が決まっていない時には

(+(+1 x) (+2 3))

は

(+(+1 x) 5)

に簡約化される（LOGLISPでは小文字で始まるシンボルが変数である）．そしてxが値を持つとさらに簡約化がなされる．これにより遅延評価に似た動作が実現できる．しかし簡約化は毎回呼ばれるため，簡約化を必要としないプログラムも影響を受け，速度が低下する．TAOでは必要な場合にのみ","の評価が行われ，通常の処理には影響を与えない．LOGLISPは基本的に全解探索を行い，全解のリストを値として返す．バックトラックの機構は持たず，WAM方式の効率の良い実装方法を適用できない．

POPLOGはPOP-2の方言であるPOP-11に論理型プログラミングの機能を取り込んだ言語である．POPLOGでは1つの述語はPOP-11の1つの手続きにコンパイルされる．コンパイラベースであるうえ，論理型プログラムとPOP-11の構文がまったく異なるためassertなどを用いてプログラム自身を扱うことは難しい．TAOはインタプリタベースでありプログラムはS式ですべて表現されているためプログラムの操作は簡単である．POP-11から述語を呼ぶには，コンパイルされたPOP-11の手続きを呼ぶ．ユニフィケーションで作られたデータには間接ポインタ(TAOの見えないポインタ）が含まれる．POP-11側でそのようなデータにアクセスするときにはユーザが自分で間接ポインタをたどる必要がある．TAOでは見えないポインタは自動的に剝ぎ取られるためユーザは意識する必要はない．論理型プログラムからPOP-11の手続きを呼ぶには，基本的には述語isの右辺から呼び出し，その結果をisの左辺へ代入する．TAOのように引数に関数を書くことはできない．

Uranusは主に知識表現などの問題を扱うための言語であるが，論理型プログラミングの機能をLISP上に実現した処理系ととらえることもできる．この観点からTAOとの比較を行う．Uranusのユニフィケーションの特徴は項記述である．項記述により，ユニ

フィケーション時に述語を呼び出すことができる．TAO と異なるのはこの述語が失敗した場合にはユニフィケーションが失敗する点である．TAO では ","によって述語を呼ぶことはできるがこれはR評価でなく評価である．すなわち，述語が失敗して返されたnil がユニフィケーションの対象となる．また項記述における遅延実行の機能は TAO にはない．

ESP はオブジェクト指向パラダイムを論理型パラダイムに融合した言語である．メソッド（クラスに属する述語）とその継承，スロット（副作用を持つ変数），マクロ機能などが大きな特徴である．ESP のメソッドは TAO の論理メソッドに対応する．ESP は第1引数をメッセージ送信先のインスタンスとすることで構文を Prolog に近いものとしており，TAO は専用の構文を導入している．しかし機能的にはほぼ同等のものと考えられる．スロットは TAO のインスタンス変数に対応する．相違点はスロットに代入できるデータ型が限られているのに対し，インスタンス変数には TAO のすべてのデータ型が代入できる点である（ただし undef データの代入は即値の代入に過ぎずポインタによるリンクなどはされない）．これは ESP では幾つかの構造データ型をスタック上に置いているのに対し，TAO ではすべてヒープ領域に取っていることに起因する．ESP ではスタック上のデータを明示的にデータ変換をしてスロットに代入する必要がある．TAO ではこのようなことは不要であるが，これを実現するためにプログラムによっては多くのメモリを消費する．TAO のインスタンスファクトに相当する機能，インスタンス固有のデータをバックトラックで探す機能は ESP にはない．ESP のマクロは TAO の Lisp マクロを用いて等価な動作が実現可能ではあるが，ESP の方が Prolog 向きのマクロとなっている．

以上のことから TAO は，関数と述語が区別されない，パターンの中に関数が書ける，データの互換性に優れる，本来の実行速度を保ったまま融合が実現されているなどの点で優れていると言えよう．一方，ユニフィケーションによって変数に値が代入されるまで関数の実行を遅らせるといった遅延評価の機構は現在の TAO では実現されていない．またインスタンスファクトは TAO 特有の機能であるが，固定のメソッド名 "&" でなくパターンをメソッド名として呼び出せた方が有用であることが指摘されている[10]．

7. むすび

TAO の論理型プログラミング機能の次のような特徴について述べた．

- 関数と述語の区別がなく相互呼び出しができる．
- ユニフィケーション時に関数呼び出しができる．
- 関数型と論理型プログラムが共通のデータ構造を持つ．
- オブジェクト指向との融合機構を持つ．

また，処理系の評価を行いシステム関数などを多用する実用的なプログラムにおいては十分な実行速度を持つことを確認した．問題点としてはインスタンスファクトの仕様，トレール量の多さ，遅延実行機構の欠如などがある．筆者らは現在 TAO での経験をもとに SILENT マシン上の新 TAO[12] を設計中である．新 TAO ではこれらの問題の解決と，実行速度の 10 倍の向上を目指している．

マルチパラダイム言語では問題を適当に分割して適したパラダイムを選ぶことが重要である．8queen 程度のプログラムではこれは簡単であるが一般には難しい．この問題に関する検討も今後の課題である．

謝辞 ご討論いただいた NTT ソフトウェア研究所尾内理紀夫リーダと同僚に感謝いたします．NTT ヒューマンインタフェース研究所日比野靖リーダに感謝いたします．

参考文献

1) Mellish, C. and Hardy, S.: Integrating Prolog in the POPLOG Environment, *Implementations of PROLOG* (Campbell, J. A. ed.), pp. 147-162, Ellis Horwood (1984).
2) Nakashima, H.: Uranus Reference Manual for V-11, *Bulletin of the Electrotechnical Laboratory*, Vol. 50, No. 8 (1986).
3) Okuno, H. G.: The Report of the Third Lisp Contest and the First Prolog Contest, 情報処理学会記号処理研究会, 33-4 (1985).
4) Quintus Prolog Reference Manual, Quintus Computer Systems, Inc. (1985).
5) Robinson, J. A. and Sibert, E. E.: LOGLISP: Motivation, Design and Implementation, *Logic Programming* (Clark, K. L. and Tärnlund, S. A. eds.), pp. 299-314, Academic Press (1983).
6) Takeuchi, I., Okuno, H. and Ohsato, N.: A List Processing Language TAO with Multiple Programming Paradigms, *New Generation Computing*, Vol. 4, No. 4, pp. 401-444 (1986).

7) Warren, D. S., Dietrich, S. and Pereira, F.: The SB-Prolog System, A User Manual (Debray, S. K. ed.), Univ. of Arizona (1988).
8) Warren, D. H. D.: An Abstract Prolog Instruction Set, SRI Technical Note, 309 (1983).
9) 大里，竹内：複合パラダイム言語 TAO におけるオブジェクト指向プログラミングとその実現，情報処理学会論文誌，Vol. 30, No. 5, pp. 596-604 (1989).
10) 尾内：イベント計算とオブジェクト指向計算の融合のための一方式，信学技報，COMP 90-26, pp. 65-72 (1990).
11) 小型化 PSI ESP 説明書，新世代コンピュータ技術開発機構 (1988).
12) 竹内，天海，山崎：新しい TAO の設計，情報処理学会記号処理研究会，56-2 (1990).
13) 日比野，渡邊，大里：Lisp Machine ELIS のアーキテクチャーメモリレジスタの汎用化とその効果一，情報処理学会記号処理研究会，24-3 (1983).
14) 山崎：TAO/ELIS における論理型プログラムのコンパイラ，情報処理学会記号処理研究会，50-3 (1989).

(平成2年11月6日受付)
(平成3年6月13日採録)

山崎 憲一（正会員）

昭和59年東北大学工学部通信工学科卒業．昭和61年同大学院修士課程修了．同年，日本電信電話(株)入社．現在，記号処理専用計算機および実時間処理の研究に従事．

奥乃 博（正会員）

1972年東京大学教養学部基礎科学科卒業．同年電電公社（現NTT）武蔵野電気通信研究所入所．1986～1988年スタンフォード大学コンピュータ科学科知識システム研究所客員研究員．現在，NTT 基礎研究所情報科学部竹内研究グループ所属．主幹研究員．Lisp，演繹データベース，論理型プログラミング，プログラミング環境の研究を経て，現在は AI の並列処理，AI プログラミング・パラダイムの研究に従事．1990年度人工知能学会論文賞受賞．人工知能学会，日本認知科学会，日本ソフトウェア科学会，ACM，AAAI 各会員．本学会欧文誌編集委員，人工知能学会編集委員，日本ソフトウェア科学会企画委員．

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年同大学院理学系研究科修士課程修了．同年，電電公社電気通信研究所入社．現在，NTT 基礎研究所情報科学研究部主幹研究員．プログラミングパラダイム，AI マシン・言語などの研究を行っている．ACM，日本ソフトウェア科学会各会員．