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

山崎憲一[†] 吉田雅治[††]
天海良治[†] 竹内郁雄[†††]

TAOは，関数型，論理型，オブジェクト指向のプログラミング機能を持つマルチパラダイム言語である．TAOの論理型計算は次の2つの特徴を持つ．パターンマッチとガードによって節を選択し，深いバックトラックは陽に呼び出す．関数と述語は互い呼び出すことができ，任意のデータを渡せる．本論文では，このような論理型計算機構を実装するための抽象マシンを提案する．この抽象マシンは，WAMをベースとしており，以下のような特徴を持つ．1) 構造データをスタックでなくヒープ上に表現する．2) 単一化やパターンマッチでは，レジスタを極力使用しない．3) プロセススイッチする可能性がある時点では，データを必ず無矛盾に保つ．また，他のProlog処理系と比較評価し，Lispとの融合によって性能が劣化しないことを示す．

Implementation of Logic Computation in a Multi-paradigm Language TAO

Kenichi Yamazaki,[†] Masaharu Yoshida,[††] Yoshiji Amagai[†]
and Ikuo Takeuchi[†††]

TAO is a Lisp-based multi-paradigm programming language which incorporates functional, logic and object-oriented programming paradigms. This paper describes the implementation of logic computation in TAO, which is different from Prolog in the following two points. A clause is selected according to pattern matching and guard testing, and deep backtracking is invoked explicitly. Functions and predicates can invoke each other and pass any type of data between them. We propose an abstract machine, based on WAM (Warren's abstract machine), which has the following features. 1) Structured data are represented in heap memory instead of stack. 2) Almost no extra registers are used at unification and pattern maching. 3) Memory configuration is consistent at any potential process-switching point. We also evaluated our implementation comparing with other Prolog processors, and showed that the paradigm fusion does not degrade the performance.

1. はじめに

TAO[7]は，様々な側面を持った実世界の問題を扱うために設計された言語であり，以下のような特徴を持つ．

- 記号処理に重点をおきLispをベースとする．
- 複数のプログラミングパラダイムを融合する．
- 実時間処理に関する記述が可能．

具体的には，20種以上のデータタイプと，それらを処理する豊富な関数を持ったLispをベースとし，これに，逐次型論理プログラミングの機能と，メッセージ送信メタファーのオブジェクト指向の機能を融合している．また，実時間制約を記述するプリミティブ（タイムアウト処理，割込み処理，スケジューリング制御，同期通信，排他制御など[2]）と実時間応答にすぐれた並行GC[9]を備え，きわめて高速な実時間OS[8]の機能を持つ．

本論文では，このうち主に論理型プログラミングの機能[14]の実装について述べる．TAOの論理型プログラミング機能には，次のような特徴がある．自動的な浅いバックトラックとガードとで節を選び，深いバックトラックは陽に呼び出す．パターンマッチにより引数を渡し，単一化はボディで陽に記述する．ボディ，ガード，および単一化の中から関数を呼び出すことができる．本論文では，これらを実現するうえでの問題を指摘し，WAM（Warren's Abstract Machine）[11]

† NTT未来ねっと研究所
NTT Network Innovation Laboratories
†† NTTサイバースペース研究所
NTT Cyber Space Laboratories
††† 電気通信大学
The University of Electro-Communications

をベースとした抽象マシンを提案する．

我々は，この抽象マシンをバイトコードインタプリタ方式で実装したが，実装にあたっては，実時間 OS 上で並行 GC が動作するというシステム環境に対応するために，次のような問題を解決しなければならない．すなわち，WAM では，他プロセスの存在を前提としていないため，メモリが矛盾状態になる期間が存在する．その間に並行 GC が走行したり，他プロセスが共有データを参照したりすると，致命的なエラーとなる．

これまでにも融合型言語の研究はなされてきたが，いわゆるピュアなものが多く，副作用や動的脱出などを含む実用的な Lisp と融合させた研究，さらに上記のような問題を考慮した実装の研究はあまり見られない．TAO は，専用計算機 SILENT 上に実装されているが，抽象マシンよりも上のレベルの実装法については従来アーキテクチャの計算機にもほぼ適用可能である．また，SILENT ハードウェアに特に依存する実装については，論文中で随時指摘する．

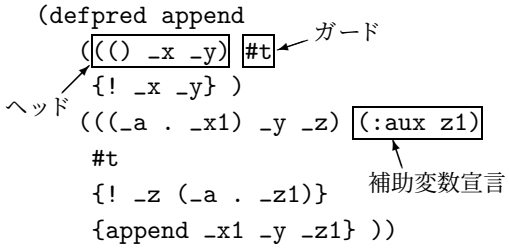
本論文は次のように構成される．まず，2 章において，TAO の論理型プログラミング機構について簡単に説明する．3 章では，実装についての基本的な目標とその概要について述べる．4 章で，我々が行った実装について詳細を述べ，5 章において他の処理系と比較し評価を与える．6 章で，関連研究との比較を行う．

なお，本論文では，紙面の都合上，WAM の知識を前提とする．これについては文献 1) に詳しい．

2. TAO の概要

データ表現は，本論文の範囲では Scheme とほぼ同じである（ブール値は`#t`/`#f`，ベクタは`#(a b c)`など）．独自のデータとして，未定義値を表す *undef* がある．*undef* は，単一化やパターンマッチの対象となるとき以外は，単なる即値と見なせる．記号`_`を評価すると *undef* になる．

TAO の主な構文を付録に示す．ここでは，本論文のために必要な部分について例を用いて説明する．リストを連結するプログラムは，次のように書く．



このプログラムは，Prolog の次のプログラムと対比すると理解しやすい．

```
append([],X,Y) :- X=Y.
append([A|X1],Y,Z) :-
    Z=[A|Z1],
    append(X1,Y,Z1).
```

上の TAO 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とり，それを第 2 引数と連結して，第 3 引数に返すもので，

```
{append (a b c) (c d) _x}
```

などのように呼ぶ．ただし，Prolog のように，入出力の方向を逆にして呼び出すことはできない（このプログラムの場合はエラーとなる）．

TAO の論理型プログラムの主な特徴は次のとおり．

- 述語呼び出しと関数呼び出しは異なる構文を持つ．
- 引数はヘッドとの（単一化ではなく）パターンマッチで渡される．
- ヘッドに現れない変数は補助変数宣言をする．
- パターンマッチが成功するとガードを評価し，`#f` 以外を返した場合のみボディが実行される．
- パターンマッチが失敗するか，ガードの値が`#f` のときは，次の節に実行が移る．
- 単一化はボディ中に次のような式で陽に記述する．

  `{!` パターン$_1$ パターン$_2$`}`
- 単一化や述語呼び出しの引数パターン中の`_`で始まる式は，強制評価式と呼ばれ，式の評価値が操作の対象となる．次の例では変数 x の値と 3 とが単一化される．

  ```
  {! _x _(+ 1 2)}
  ```

  なお，`_x` のように変数だけの強制評価式は，実装上は特別に扱われる．以降では強制評価式とは，変数以外の場合を指すものとする．
- 述語は，ボディの最後の式の値を返す．

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わなければならない．チョイスを生成するには，下の例のようにガードの次に`(:choice)` と書き，制御移動には，組込み関数 `fail` を用いる．

```
(defpred foo
   (() #t (:choice)
    (print 'clause1) (bar))
   (() #t (print 'clause2)) )
(defun bar () (fail _))
```

`{foo}`を実行して，最初の節の選択が成功すると（直感的には`:choice` に実行が到達すると），2 番目の節を継続とするチョイスが生成される．`clause1` が印

字され，関数 bar の実行により，fail が起動される．fail は，まず最新のチョイスを探し，それが生成された以降の単一化による代入を（もしあれば）取り消し，チョイスの継続に制御を移す．この場合のチョイスの継続は，foo の第 2 節であり，これを実行した結果 clause2 が印字される．

なお，チョイスには名前をつけることができ，fail の引数を用いて名前でチョイスを指定できる．この例のように引数が *undef* の場合には，最新のチョイスが指定される．TAO では，fail による制御移動を大域的脱出の一種として扱う．たとえば，fail によって unwind-hook （Common Lisp の unwind-protect に相当）の内から脱出するときには，後始末処理関数（同じく cleanup form に相当）が自動的に実行される．

述語呼び出し式の評価には，$\mathcal{L}$ 評価と $\mathcal{P}$ 評価の 2 種類がある．$\mathcal{P}$ 評価は述語のボディでの評価であり，$\mathcal{L}$ 評価はそれ以外のすべての評価である．述語を $\mathcal{L}$ 評価すると，実行の成功/失敗がブール値で返される．より正確には，述語呼び出し式 E の $\mathcal{L}$ 評価とは，次のような意味である．1）#f を E の値として返すという継続を持つチョイスが生成される（これを暗黙のチョイスと呼ぶ）．2）E を $\mathcal{P}$ 評価する．3）暗黙のチョイスを含め，それより新しいチョイスをすべて削除し，#t を主値，ステップ 2 の評価値を副値とする多値を返す．

なお以降では，:choice を含む節を NDET 節，そうでない節を DET 節と呼ぶ．また，述語呼び出し，パターンマッチと単一化に基づく計算機構を LP と呼び，その他の計算機構を Lisp と呼ぶ（機構としての “Lisp”と，言語としての Lisp は異なるものであるが，誤解は生じないであろう）．これらの計算機構は，言語仕様上・実装上は融合されており，厳密に分けることはできない．あくまでも説明の便宜のためである．

3. 基本設計

ハードウェアと言語仕様から生じる設計上の制約，および実装の大枠について述べる．

3.1 専用計算機 SILENT

SILENT[15] は次のような特徴を持つ．

- タグアーキテクチャ（8 bit タグ+32 bit データ）
- クロック 33 MHz，キャッシュ 320 KB，主記憶 640 MB
- 2 語幅（80 bit）バス（セルを 1 回で書き込み可）
- マイクロプログラム制御
- 型判定による条件分岐が演算と同時に実行可
- ハードウェアスタックによる高速スタックアクセス
- バイトコード実行支援（自動フェッチとデコード）

3.2 言語仕様上の制約

我々は，実世界の様々な問題記述においては，発見的に（つまり対話的かつ段階的に）プログラムを記述していくことが重要であると考える．このため，TAO は，発見的プログラミングを重視して実装される．これが実装設計上最も重要な制約である．実時間処理を意識した発見的プログラムにおいては，デバッグ時でも最終的な実行と同等の速度で動作させなければならない．つまり，インタプリタを用いて開発し，実システムはコンパイルして動かすといったことはできない．さらに，Lisp のリフレクティブなプログラム操作も効率的に行える必要がある．以上の点から，たとえば最適化コンパイルに時間をかけなければ性能が発揮できないようなシステムは，我々の目的に適合しない．

このため，我々は，高レベルのバイトコードへコンパイルし，これをマイクロプログラムによって解釈実行する方式を採用した．このコンパイラには，まずコンパイルが高速であることが求められる．また，フレームを解釈するために必要な情報などは,（たとえ速度低下を招いても）デバッグやリフレクティブな操作のために保持しなければならない．

このような前提のもとで実行モデルとメモリ構成を決定した．

［実行モデル］

実行モデルには大別してスタックマシンとレジスタマシンがある．TAO では，Lisp 部分の基本実行モデルに前者を採用した．前者は，デバッグ情報を残しやすいという点においても，プロセススイッチなどの実時間性能が高いという点においても優れている．また，実行速度については，SILENT ハードウェアにより，十分な性能を達成できると考えられる．一方，LP の実行モデルとしては後者を採用した．一般に，LP はレジスタマシンの方が高速であり，また WAM の技法を適用しやすい．レジスタセーブの手間に関しては，SILENT の高速スタックを利用して，使用レジスタ数を減らすことで対処する．

［メモリ構成］

WAM では 3 本のスタックを用いており，構造体はグローバルスタック上に表現する．後戻り時やトップレベルゴールの終了時には，スタック機構によってメモリが回収され，効率が良い．しかし TAO で同様の方法をとると，重大な問題が生ずる．単一化で作ったデータを自由に Lisp に渡すことができるからである．Lisp の破壊的代入によって，そのデータが大域変数な

どから指されたときに，スタックポップでデータを捨てると，いわゆる dangling pointer が発生する．

また，実時間処理の観点からは，WAM で通常行われるスタックシフト，スタックコンパクションは，一般に割込み禁止時間が長くなるという意味でも好ましくない☆．以上のことから，構造データはヒープ上に表現するものとした．

その他の制約としては次のような点がある．

- TAO は厳密なハードリアルタイムの保証はしないが，割込みから 100 μ 秒以内に処理ルーチンが起動することを目指している．このため割込みが入ったかを頻繁にチェックする☆☆．原則として，各バイトコードの終了時点で，また 1 つのバイトコード内に長時間のループがあれば，その内部でチェックを行う．チェックの結果，プロセススイッチする可能性がある．TAO では，単一化の不可分性を保証していないため，単一化途中のデータが見えることは許されるが，致命的なエラーを生じうる状態が他プロセスから見えてはならない．
- 上のケースの 1 つとして，プロセススイッチした結果，GC プロセスが起動する可能性がある．GC は，ユーザプロセスではたどれないようなデータをも操作するため，実装上の注意がより必要である．

4. 抽象マシンとその実装

4.1 基本的な実行方式

メモリ構成は，フレーム表現のための（ハードウェア）スタックが 1 本（Lisp と共用），構造データ表現のためのヒープ領域，トレールスタックを実現するメモリブロックチェインである．内部データは，SILENT アーキテクチャを利用して，ポインタ 32 ビットとそのポインタが指すデータの型を表すタグ 8 ビットにより表現する．他の Lisp 処理系にはない TAO 独自のデータである *undef* は，専用のタグ値（`undef`）を持った即値データとして表現されるが，詳細は後述する．

レジスタには，次のようなものがある．

CP	リターンアドレス	LP 専用
NC	次の節のアドレス	
AN	引数の個数	
A_i	引数	
TR	トレールスタックトップ	
GN	現在の世代番号	
MaxGN	最大の世代番号	

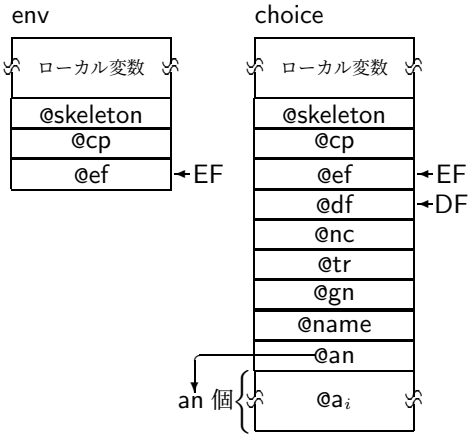


図 1 フレームの構成（図の下がスタックの底方向）
Fig. 1 Structure of a frame.

EF	環境フレーム	Lisp と共用
DF	動的フレーム	
SP	スタックポインタ	
PC	次の命令を指すポインタ	

このうち，CP，TR，A_i は WAM の同名のレジスタと，EF は WAM の E と，DF は B と，それぞれほぼ同じ意味である．引数 A_i には，SILENT の 64 本の汎用レジスタのうち，32 本を割り当てる☆☆☆．CP，EF には，印をつけられる（タグ 8 ビットの中の 1 ビットを使用）．印およびその他のレジスタの意味については以下で適宜説明する．

フレームには env と choice の 2 種類がある（**図 1**）．フィールド名は@をつけて表す．env フレームは WAM の “環境”に相当し，choice は WAM の “選択点”と環境とをマージしたものである．デバッグなどのためにフレームにはつねにローカル変数を残すため，WAM とは異なり選択点には必ず環境がともなう．このため，選択点と環境に共通するフィールドを統合した．@skeleton は，フレームを解釈するために必要な情報で，具体的には，このフレームを作った節とその変数割当ての情報である．@name はこのチョイスの名前であり，`fail` がチョイスを探すときに使用する．

TAO のバイトコードは，実際は 10 ビット長である．現在，842 種類のコードがあり，このうち LP に関連するものは約 240 である．簡単なピープホール最適化を行っている[13]．

4.2 実行制御

1 つの述語は，各節に対応したコードと，それらの節の間の実行を制御するコードからなり，前者は節の追加・削除を行っても再コンパイルが不要である．以下，実行順に説明する．

☆ 実時間 GC に関する研究[5),6)] もあるが，ある決まった時間内に GC の不可分操作が終了することを保証するのは困難である．

☆☆ このチェックは，ハードウェアの支援によりオーバヘッドなしに行われる．

☆☆☆ 32 本に入りきらない場合は，専用のメモリ領域が用意される．

［節間制御］

述語が呼ばれると，まず節間制御部が実行される．節間制御部は，インデキシングを含む節選択の制御とフレームの生成を行う．1つの述語中の節は3つのグループに分割して制御される．最初の NDET 節から最後の NDET 節までを NDET 部，それより前を第1DET 部，後を第2DET 部と呼ぶ．DET 部，NDET 部のことを節部位と呼ぶ．NDET 節の属す節部位は必ず NDET 部であるが，DET 節はどちらの節部位にも属しうる．さらに，節の動的な追加・削除が許されているため，ある DET 節の属す節部位は静的には決められない．以下のようなコードが生成される．

```
do        節 DET1_1ラベル  }
redo      節 DET1_2ラベル  } 第 1DET 部に対応
...                        }
delenv
try       節 NDET_1ラベル  }
redo      節 NDET_2ラベル  } NDET 部に対応
...                        }
delchoice
do        節 DET2_1ラベル  }
redo      節 DET2_2ラベル  } 第 2DET 部に対応
...                        }
redolast  節 DET2_nラベル  }
```

各バイトコードの動作を疑似的な C コードで記述する．ここでの単位データの大きさは，40 ビット（つまりポインタ部とタグ部）であることに注意されたい．

```
do ラベル
    *--SP = EF;
    EF = SP;
    *--SP = CP;
    *--SP = 0;
    NC = PC;
    PC = ラベル;

try ラベル
    Ai～CP をセーブ
    EF と DF を do と同様に更新
    NC = PC;
    PC = ラベル;

redo ラベル
    NC = PC;
    PC = ラベル;

redolast ラベル
    NC = &error;
    PC = ラベル;

delenv
    SP = EF;
    EF = *SP++;

delchoice
    SP = EF;
    EF = *SP;
    SP += SP->@an + 7;
```

ここで&error は，エラーを起動するバイトコードのアドレスである．DET 部の実行中には env が，NDET 部の実行中には choice が，必ず積まれている．try での EF 更新では，それの指しているフレームが choice であることを示すための印が付けられる．なお NDET 節がない場合には，第 2DET 部だけのコードとなる．

［1 つの節のコンパイル］

節はおおよそ次のようにコンパイルされる．

```
setskel   スケルトン   ; @skeleton のセット
allocvars N            ; 変数領域の確保
<パターンマッチ>
<補助変数の束縛>
<ガード>               ; ガードの値がプッシュされる
commit                 ; その値が#f なら NC へ
setAN     N            ; 引数の個数をセット
<引数処理>             ; 最初の節の引数処理
pcall     述語名       ; 最初の節の呼び出し
...
setAN     N
<引数処理>
exec      述語名       ; 最後の節の呼び出し
```

各バイトコードの意味は次のとおり：

```
setskel スケルトン
    EF->@skeleton =
        スケルトン;

allocvars N
    SP -= N;

commit
    if(*SP++ == #f)
        PC = NC;

setAN N
    AN = N;

pcall 述語名
    check_argnum(述語名, AN);
    CP = PC;
    PC = get_code(述語名);

exec 述語名
    check_argnum(述語名, AN);
    if(older(DF,EF))
        SP = frame_bottom(EF);
    CP = EF->@cp;
    EF = EF->@ef;
    PC = get_code(述語名);

proceed-d
    SP = frame_bottom(EF);
    EF = EF->@ef;
    PC = CP;

proceed-nd
    EF = EF->@ef;
    PC = CP;
```

allocvars は，パターンマッチで束縛される変数領域を確保するが初期化は行わない．初期化未了の状態でプロセススイッチする必要が生じた場合は，未初期化変数を *undef* で埋めてから，プロセススイッチする．パターンマッチ，引数処理については後述する．補助変数の束縛は，初期値を順にプッシュしていき，これでフレームが完成する．commit で，節選択が失敗したときは NC へ制御を移すだけである．パターンマッチでは，引数レジスタ A_i は破壊されず，トレールもされないため，これで十分である．

pcall の動作は WAM とほぼ同じであるが，動的な述語定義を許しているため，引数のチェック（関数 check_argnum）が必要であり，コードのアドレスは実行時に得る（関数 get_code）必要がある．exec で，最新の choice が最新のフレーム（env のことも choice のこともある）よりも古いことが，関数 older により分かった場合，最新のフレームを捨てる．frame_bottom は，EF の印からフレームの種類を調べて，その底のアドレスを計算する関数である☆．

☆ 節が追加・削除されるたびに動的にコードを書き換えれば，実行時に計算する必要はないが，現在は追加・削除が軽いことを重視している．

proceed は，2 種類ある．ボディがない節にもフレームがあるため，WAM と異なる動作をする．proceed-d は DET 節用のもので，フレームを捨てて CP に戻る．NDET 節用の proceed-nd では，最新のフレームは有効な choice なので，フレームは捨てない．

4.3 データ操作

データ操作に関しては，パターンからの引数生成，引数とヘッドのパターンマッチ，単一化がある．紙面の都合上，特徴的なバイトコードのみを説明する．

全体を通じての目標は，以下のような点である．

- メモリ状態を並行 GC から見て無矛盾に保つ．
- レジスタ使用量を最小限にする．
- パターン中に複数の強制評価式があった場合，言語仕様どおり左から右へ行う．

ここで使用するレジスタは以下のとおり：

S	構造データの要素を指すポインタ
rpr	場所ポインタ

rpr は，SILENT の特殊なレジスタである．マイクロ命令で指示すれば計算結果をオーバヘッドなしに rpr に保持することができる．このレジスタは，TAO 独自の評価方法を実現するために用いられる．TAO の評価では，その評価結果の値が存在した場所を評価結果と共に返す．たとえば，次のようなことが可能である．

```
(defun foo (x) (car x))
(let ((x (cons 'a 'b)))
   (!(foo x) 'c)
   x )   →    (c . b)
```

ここで，(!式 1 式 2) は，代入式と呼ばれ，式 1 の返した値の存在する場所に，式 2 の値を代入する．この例では，x の car に代入が起きるが，これは foo の中で car のアドレスを計算したときに，それを rpr に保持し，そこに c を書き込むことで，実現される．

［変数のアクセス］

述語中で宣言された変数は，可能ならばレジスタに割り当てられる．レジスタに割り当てるかどうかの基準は WAM のテンポラリ変数のそれと同じである．

スタックに割り当てられた変数は，EF からのオフセットでアクセスされる．env でも choice でも同じオフセット位置から始まることは重要である．節の属す節部位は（つまりフレームの形も）動的に変わりうるが，変数のオフセットは同じであり，再コンパイルは不要である．

TAO の変数には，様々な変数（静的変数，動的変数，大域変数など）があり，それぞれ専用のバイトコードでアクセスする．さらに静的変数でも，環境の入れ子状態に最適化された多種のバイトコードがある．これらのバイトコードは，スタックマシンモデルに基づくもので，値をスタックトップにプッシュする．これらの各々に対応するレジスタマシン用のコードを用意すると，たとえば put_value だけで何十種類ものバイトコードが必要となる．これを避けるための命令が put-pop である．

put-pop R_i

```
R_i = *SP++;
if(tag(R_i) == undef) R_i = rpr;
```

put-pop は，スタックトップに積まれた値をレジスタに設定するが，その値が *undef* の場合には，値が存在する場所へのポインタをレジスタ rpr を用いて設定する．put-pop の前に，rpr をセットする命令を配置することはコンパイラの責任である．たとえば，述語呼び出しの引数に，動的変数 v が現れた場合のコードは次のようになる．

```
symbol v          ; 変数名 v をプッシュ
dyn-var           ; v の値をプッシュ
put-pop A_i       ; 値をポップして A_i にセット
```

ここで，dyn-var は，変数名からその動的束縛の値をプッシュし，同時にその場所を rpr にセットする．

変数に関する，WAM にはない概念として indirect 状態がある．WAM では，未定義値は自分自身へのリンクとして表現されている．ある変数の値をレジスタにセットすれば，その変数が未定義値の場合は，自動的にレジスタから変数へのリンクができる．

しかし，TAO では，未定義値を持つ変数には，*undef* が入っている（タグ部は undef，データ部は後述する世代番号）．この場合，その値をレジスタにセットすると，レジスタの値が *undef* となってしまう．これを避けるには，セットごとに *undef* かを調べ，そうならばリンクを張る，という操作が必要になる．

このチェックのコストは SILENT ではきわめて小さいが零ではない．しかし，変数が次の状態（indirect 状態）のとき，その値が具体値かリンクであることが保証できるため，チェックを省略した専用のバイトコードが使える．

- ヘッドで宣言された変数はつねに indirect．
  – 呼び出し側の環境へのリンクになるため．
- 構造データ中に現れた変数はそれ以降 indirect．
  – 構造データへのリンクが張られるため．

これまで述べてきた「リンク」は，未定義値の変数どうしを単一化した場合に，一方の変数からもう一方へ張る（専用のタグを持った）ポインタである．リンクは，値を得ようとしたときに自動的にたどられる．リンクが Lisp に渡されることもあり，Lisp 側でも自動的なたどりを行わなければならない．これは，従来の Lisp の実装にはない操作であるが，SILENT の分岐命令によって高速に判定される．

［引数処理］

引数は実レジスタに格納される．SILENT には，バイトコード中に埋め込まれた 5 ビットをレジスタ番号として，レジスタに間接アクセスする機構があり，引数レジスタも一般のレジスタと同じ速度でアクセスできる．

引数をレジスタにセットする命令は，WAM の put 命令とほぼ同等であるが，構造データ（セルとベクタ）に関しては若干異なる．まず，セルに関しては，次の例のようになる．

例：(a b) というパターンのコード：

```
push-constant a    ; *--SP = &a;
push-constant b    ; *--SP = &b;
cons-sp-empty      ; *SP=cons(*SP,());
cons-sp-sp         ; *SP = rcons(*SP++,*SP);
                   ; rcons は cdr と car を cons する
put-pop A_i
```

いったん，要素をスタックに積み，ボトムアップで構造データを作っていく．この方式では，WAM のような中間データ保持のためのレジスタが不要となる．また，cons するときに car と cdr の値が確定しているというメリットもある．セルは生成された直後から GC にマークされる可能性があるため，WAM のように先に（値を書き込まずに）セルだけを確保し，後で値を書き込むことはできない．

一方，セルのアドレスが cons しないと決まらないため，変数が現れたときには，工夫が必要となる．変数 x が cdr 側に初出した場合は，次のようになる．

例：(a . _x) のコード：

```
push-constant a
cons-sp-variable X ; *SP = cons(*SP,GN);
                   ; *(EF-X) = &((*SP)->cdr);
                   ; X は変数 x のオフセット
```

初出でない場合は，cons-sp-value X が生成される．これは，x が *undef* だった場合に上と同じことを行う．一方，car 側に変数が現れたときは，次のようになる．

例：(_x . c) のコード：

```
push-variable X    ; *--SP = EF-X;
push-constant c
cons-sp-sp
put-pop A_i
```

cons-sp-sp は，car 側の引数*(SP+2) がスタックへのポインタだった場合には，そのポインタが指している場所から，新しく作られたセルの car へリンクを張る．x がテンポラリ変数のときや複数回現れたときは，さらに特別なコードが必要となるが省略する．

ベクタの場合は，その要素のマークを禁止するビットがあるため，ベクタを生成してから要素を順に初期化していけばよい．ただし，マークを禁止してしまうと，初期化に用いたデータがベクタ経由でマークされなくなる．そこで，ベクタの初期化が終わるまで，データ（構造データだけでよい）をスタックに残す．

例：#(a (b) c) のコード：

```
make-vector 3      ; S = *--SP = make_vector(3);
write-symbol a     ; *S++ = &a;
write-vector-save  ; *--SP = S;
push-symbol b
cons-sp-empty
write-vector-resume; R0 = *SP++;
                   ; S = *SP;
                   ; *S++ = *SP = R0;
make-vector-end 1  ; SP += 1;
                   ; permit_mark(*SP);
put-pop A_i
```

write-vector-resume は，S を元に戻し，リスト (b) をプッシュし直している．make-vector-end で，中間データをポップし，ベクタのマークを許可する．

［パターンマッチ］

左から右へ深さ優先でパターンマッチをしていき，パターンマッチが失敗したときは，NC へ制御を移す．基本的な動作は，単一化の read モードのときとほぼ同じなので省略する．

［単一化］

パターンどうしの単一化はパターンを分解すれば，変数とパターンの単一化に帰着できる．パターンに対し左から右へ深さ優先でコードが生成される．次の例は，

```
{! _x (a (b))}
```

のコードを示したものである．またコメントは，変数 x が R0 に割り当てられており，その値が (a *undef*) のときの動作だけを示したものであり，対応するバイトコードのすべての意味を記述したものではない．

例：

```
unify-list R0      ; R0 のタグがセルか調べる
                   ; mode = READ;
                   ; S = R0;
unify-constq/car a ; S->car が a か調べる
unify-list/cdr     ; S->cdr がセルか調べる
                   ; S = S->cdr;
unify-list/car     ; *--SP = S | mode;
                   ; S->car のタグが undef なので,
                   ; mode = WRITE;
                   ; S = S->car = cons(GN,GN);
unify-constq/car b ; S->car = &b;
unify-empty/cdr    ; S->cdr = ();
unify-resume       ; mode = *SP & mode_mask;
                   ; これにより read モードに戻る
                   ; S = *SP++ & ~mode_mask;
unify-empty/cdr    ; S->cdr が () か調べる
unify-list-end     ; mode = READ;
```

単一化では，モードがどのように移行するかがポイン

トである．構造データがネストするときには，read から write に移行する可能性がある．このときには，S レジスタと（S のタグビットを利用して）モードとをセーブする．構造データの終わりには，unify-resume が実行され，元のモードに戻る（なお，セルの cdr 方向のネストは，テールマージの形になるので，ネストの処理はしない）．

上記の例には，ほかに次のようなモード移行パターンがある．変数 x が *undef* のときは，unify-list の次の命令から unify-list-end までが write モード，x が（a. *undef*）のときは，unify-list/cdr の次から unify-list-end までが write モードである．

ベクタに関しても同様にベクタの開始でモードセーブ，終わりでモードリストアが行われる．

なお，write モードは，バイトコードの解釈用テーブルのアドレスを一時的に変更するという SILENT の機構を利用して実現している．したがって，上の mode は，実はプログラムの状態である．

4.4 代入のトレール

WAM では，グローバルスタックを用いて構造データを表現している．構造データの要素への代入において，最新チョイスの生成時のグローバルスタックポインタよりも浅い場所への代入であった場合は，トレールをしない．これは，バックトラック時にスタックを縮めることにより，最新のチョイス以降のデータが自動的に捨てられるからである．これによりトレールのメモリ使用量を削減できる．

一方，本方式のように構造データをヒープ上に表現する場合，WAM のようにアドレス比較によって，構造データの生成時刻を比較することはできない．そこで，世代番号を用いてデータ生成順の管理を行う．

チョイスが生成されると，MaxGN から世代番号が割り当てられる．その後 MaxGN はインクリメントされる．MaxGN はシステム共通のレジスタで，全システムを通して世代番号がユニークであることを保証する☆．

最新のチョイスの世代番号は GN が保持する．構造データを生成する場合には，GN の値を用いて初期化する（GN のタグは undef）．構造データの要素への代入時に，その要素の世代番号と GN とを比較する．両者が同じならば，最新のチョイスの生成以降に生成されたデータであることを意味するので，トレールしない．

ただし，この方法では，代入がなされるアドレスと代入前に入っていた世代番号の両方をトレールする必要があり，トレールのメモリ使用量が 2 倍になる．また，カットがあると効率が悪化する可能性がある[12]．

トレールのデータは，プロセスごとに用意されるトレールスタックに保持される．トレールスタックは双方向チェインでつながれたメモリブロック（現在の実装では 8 K バイト）で表現される．

4.5 Lisp との融合

本節では，Lisp との融合に関連する部分について述べる．

述語呼び出し式の $\mathcal{L}$ 評価は，次のようにコンパイルされる．

```
l2p                    ; 暗黙のチョイス作成 (レジスタセーブ)
setAN N
<引数処理>             ; Ai をセット
pcall-from-lisp 述語名
true                   ; *--SP = #t;
p2l                    ; レジスタリストア
```

pcall-from-lisp は，暗黙のチョイスの@nc に p2l のアドレスを印付きでセットし，CP に true のアドレスを印付きでセットする．

ある述語呼び出しは通常は最後にボディのない節を呼び出し，proceed を実行して終了する．proceed により，CP の指すアドレス，つまり true に制御が移る．p2l は，いったん値をポップし，暗黙のチョイスからレジスタをリストアし，また値を積み直す（値が多値の場合もあるので，実際はもう少し複雑である）．これにより，述語の $\mathcal{L}$ 評価が#t を返して終わる．

次のように述語呼び出し式でない式で終わっている述語は，値を返さなければならない．

```
(defpred foo (() #t 1))
```

これは次のようなコードになる．

```
push-constant 1    ; *--SP = 1;
ret
```

ret は，CP に印が付いていたら，#t と*SP を多値としてスタックに積み，（true をスキップするため）CP+1 へと制御を移す．そうでない場合は，値をポップして CP へ制御を移す．

一方，fail が実行されて，暗黙のチョイスへ戻ることが@nc の印から分かった場合は，#f を積み，@nc の値（つまり p2l）へ制御を移す．

［述語からの関数の呼び出し］

述語から関数を呼び出すには，ボディでの関数呼び出し式の評価，ガードでの評価，引数処理や単一化における強制評価式の評価がある．いずれの場合も，呼び出しの前後でレジスタの状態，（戻り値が*SP に積

☆ MaxGN があるスレッショルドを超えたら，桁溢れを防止するため，メモリ中のすべての世代番号を 0 にする．この機能は未実装であるが，並行プロセスを起動する，並行 GC 中で行う，といった実装方式がある．

まれることを除いて）スタックの状態は変化しないため，単純に実行するだけである．

例：{! _x _(+ u v)}のコード

```
var U                ; *--SP = *(EF-U);
var V                ; *--SP = *(EF-V);
add                  ; *SP = *SP++ + *SP;
put-pop R1
unify-value R0 R1    ; 変数 x は R0 とする
```

［大域脱出］

fail の処理は基本的には WAM と同じである．ただし，たとえば unwind-hook の中で fail が呼ばれた場合には，unwind-hook の後始末関数を実行しなければならない．DF は Lisp と共通のレジスタであり，動的処理の必要なフレームがすべてリンクされている．fail は，これをたどりながら必要な処理を行う．

5. 実装の評価

Prolog のベンチマークプログラム[10] からいくつかを選び，メモリ使用量と実行速度の評価を行った．プログラム名のうち，q8 は queens_8，chat は chat_parser の略である．また，q8_all は，q8 のすべての解を得るプログラムである．

従来の Prolog プログラムを TAO に変換するには，主に次の 2 つが考えられる．

A すべての引数を入出力可能とするために，ヘッドパターンに互いに独立した変数を並べ，ボディで単一化をする．また，すべての節を NDET 節とする（つまり :choice を付ける）．

B プログラムを解析し，必ず具体化した値が渡される場合はヘッドに値を書く．ガードや DET/NDET 節も，適切に使い分ける．さらに，すべての引数が具体化されている述語は，可能なら関数で記述する．

タイプ A は，引数の入出力の方向に関係なく，Prolog と完全に同じ動作をする．しかし，実際の多くの Prolog プログラムでは入出力を意識し，インデキシング機能を利用したり，非決定性を予測したりする．TAO では，引数渡しをパターンマッチで行うから，プログラマは強く入出力を意識することになる（そうなるように言語が設計されている）．タイプ B は，そのような実際の状況に近いプログラムへの変換である．

簡単な Prolog → TAO 変換器（Prolog で 300 行程度）を記述し，上のような変換を行った．この変換器には，各述語の引数の入出力方向とバックトラックの種類を与えることができる．何の情報も与えずに変換するとタイプ A となる．タイプ B は，ベンチマークプログラムの動作を理解し，人手でモード情報を与えることでプログラムを生成した．なお，タイプ B で，実際に関数化したのは boyer の member，truep，falsep のみである．

表 1 TAO と SICS のメモリ消費量の比較
Table 1 Memory consumption of TAO and SICS.

プログラム	スタック		トレール		ヒープ
	A	B	A	B	
concat	44/0	8/0	8/0	0/0	1.0
nreverse	3.95	1.24	480/0	0/0	1.0
tak	–	1.61	–	0/0	0/0
qsort	39.1	1.29	5.12	0/416	1.0
q8	2.14	1.31	2.00	1.89	15.0
q8_all	1.96	1.19	2.00	1.89	199.0
chat	1.62	1.48	2.16	2.15	160.0
boyer	2.26	1.78	25.04	0.38	0.61

比較対象は SICStus Prolog Version 3.0（以降 SICS と呼ぶ）とした．SICS は，広く使われている処理系で，WAM をベースとしている．

［メモリ使用量］

まず，メモリ使用量を測定した．SICS はバイトコードエミュレーションのモード，また，TAO，SICS とも GC を自動的に起動しないモードである．SICS は，チョイスとローカルの 2 種類のスタックを使う実装をしているため，ここでは両者の和を「スタック」とした．測定結果を**表 1** に示す．SICS は 1 語が 4 バイトなので，TAO もタグを除いてバイトに換算した．つまり，基本ワード数の比となっている．また，A と B はプログラムの違いであるから行分類とすべきだが，ここでは A どうし，B どうしを比較しやすくするため，このような形式の表とした．ここで分数で書いてあるものは，分子（または分母）が 0 であったもので，分母（または分子）の値はバイト数である．ヒープは，A，B ともにまったく同じなので 1 つにまとめた．tak の A タイプは，TAO ではスタックオーバフローとなり測定不能であった．SICS では計算可能であったが，スタックを 2.4 M バイト使用した．これと B タイプとを比較しても差が大きすぎるため，SICS の tak にはカットを入れた．

A タイプのプログラムはすべての述語でチョイスを生成するため，比は大きくなる．また，チョイスが多く作られると，必然的にトレールの量も増加する．

B タイプは，より実際の TAO のプログラムに近いと考えられ，この結果についてより詳細に検討する．まず，スタックの差は主にフレームサイズの差に起因すると思われる．トレールについては，4.4 節で述べたとおり，比は普通 2.0 以上になる．しかし，B タイ

プの boyer や qsort では，SICS を大きく下回る値となっている．この理由は次のようなものである．boyer の典型的な節は次のようなものである：

```
difference(plus(X,Y),X,fix(Y)):-!.
```

一方，TAO (B タイプ) では次のようになる：

```
(defpred difference
   ...
   ((#(plus _x _y) _x _out2) #t
    {! _out2 #(fix _y)})
   ...)
```

これは，difference の第 1，2 引数は必ず値が具体化されているという情報を与えているからである．SICS ではチョイスポイントを作り，単一化を行った後に，カットを行う．したがって単一化時にトレールをする可能性がある．一方，TAO ではそもそもチョイスが生成されず，トレールも起こらない．

ヒープは，きわめて大きな差が生じている．しかし，この差が両方式の性能差でないことは注意する必要がある．あるプログラムの消費したメモリ量を M とすると，生成には最低 O(M) 時間を必要とする．回収はスタック機構が最も有効に働けば，O(1) である．一方，ヒープ方式も生成には O(M) を必要とし，回収は GC が行う．もし，生成したデータのほとんどがゴミであったとすると（実際 q8 や chat はその典型的なプログラム），TAO のマークスイープ方式の GC は O(M) でこれを回収する．つまり，両方式の実行時間のオーダーはトータルでは同じである．なお，boyer においては，TAO の方が使用量が少ないが，この原因は不明である．しかし，後述する B-Prolog で測定したところ，同比は 1.09 となり妥当な値であった．SICS の実装に何らかの問題があると考えられる．

［実行速度］

次に実行速度を測定した．異なるアーキテクチャ間での処理系の比較は難しいが，SILENT と同程度のハードウェアテクノロジである Sun SS20（SuperSparc 75 MHz）で測定した．また，SICS のコンパイルモードは，本論文の実装方式の特性を調べることを主目的として，アーキテクチャにチューンされたネイティブコードではなく，バイトコードを選択した．

結果を**表 2** に示す．concat，nreverse は，小さくて単純なプログラムであり，簡単な最適化が大きな効果をもたらす．TAO では，ほとんど最適化を行っていないため比は大きくなる．なお，nreverse に対し，setskel，setAN および第 2 引数セットの除去などの簡単な最適化を行ったところ，T/S = 1.73 となった．chat で比が大きいことの理由は，まだ十分解析されていないが，chat はほとんどの述語がチョイスを生成するため，A タイプと B タイプとの差が小さい．このことから，TAO と SICS とのチョイス生成やトレールの重さの違いが理由の 1 つとして考えられる．

表 2 TAO と SICS の実行時間の比較
Table 2 Execution speed of TAO and SICS.

プログラム	TAO		SICS	時間比（T/S）	
	A	B		A	B
concat	0.143	0.069	0.0325	4.41	2.13
nreverse	2.24	1.07	0.526	4.26	2.03
tak	–	165	158	–	1.04
qsort	2.10	1.02	0.957	2.19	1.07
q8	9.02	5.78	5.00	1.80	1.16
q8 all	155	99.2	84.4	1.84	1.16
chat	614	506	231	2.66	2.19
boyer	5430	1240	962	5.65	1.29

単位はミリ秒（有効数字は 3 桁）

表 3 関数の実行時間および関数と述語との時間比
Table 3 Comparison between function and predicate.

プログラム	関数の実行時間（mS）	実行時間比 関数/述語	サイズ比 関数/述語
concat	0.049	0.710	18/48
nreverse	0.766	0.716	43/102
tak	56.5	0.342	37/120
qsort	0.736	0.722	101/175

［Lisp と LP の速度比較］

ベンチマークのうち後戻りをまったくしないプログラムを TAO の関数によって書き直し，述語と比較した（**表 3**）．tak は，Lisp の再帰呼び出しに関する最適化が特に効果を持つプログラムであるため速度比は大きいが，それ以外ではほぼ一定である．この速度差は，Lisp がスタックマシンモデルのためバイトコードがコンパクトに表現されていることによると予想されるが，より詳細に解析する必要がある．コードサイズに起因する要素が支配的だとすると，複数のコードをマージしてサイズを縮小する，さらにはスタックマシンモデルで再設計するといった検討をする必要がある．

6. 議　　論

ここでは，論理型言語の基本制御方式，単一化の実装方式，割込みへの対応の観点から，他の研究と比較する．

［基本制御方式］ B-Prolog[16] の matching clause は，パターンマッチとガードによる節選択，節の決定性の陽な記述といった点で TAO と共通している．B-Prolog では，チョイスオペレータの種類やボディでの呼び出しパターンなどをもとに述語を 4 種類に分類し，この分類で述語のフレームの形を決める．一方，TAO では 2 種類のフレームがあり，1 つの述語でも節ごとに

フレームの形は異なる．TAO の方が実行速度で劣る場合もあるが，節の動的な追加・削除が容易である．B-Prolog の実装は，引数をスタックで渡す点が特徴の 1 つである．バイトコードエミュレーションの場合は，こちらの方が速いということが，この実装の根拠であるが，SILENT ではバイトコードから実レジスタを直接指定できるため，レジスタ渡しを採用した．しかし，スタックマシンによる実装は，コード表現がコンパクトで使用レジスタ数が少ないという利点がある．評価の項でも述べたように，より詳細な評価が必要である．

［単一化］ 本方式に最も類似した Meier の方式[4] と比較する．Meier では，構造データが入れ子になった場合は，レジスタを用いて中間状態を保持している．一方，我々の方式では，レジスタは使用せずスタックを使う．高速スタックを持つ SILENT では，実行速度はほぼ同じである．制御シーケンスについては，Meier では，read モードと write モードを 2 つのシーケンスに分け，動的に判定しながらシーケンス間をスイッチする．本方式では，シーケンスは 1 つで，現在のモード状態で翻訳動作を変える．Meier ではコードレベルの最適化は容易であるが，本方式には分岐がないという利点があり，総合的な実行速度はアーキテクチャに依存する．SILENT では，バイトコードレベルでの制御の移動はコストが高く，本方式の方が有利である．

［割込み処理］ KLIC[17] などの並行論理型言語では，一般に 1 つのリダクションが終わるまでは，原則として割込みは検出せず，したがってプロセススイッチも起きない．一方，TAO では 1 つのバイトコードごとに割込みを検知し，これにより，きわめて高い実時間性を達成することができる．ただし，割込みを検知しない方法に比べ，並行 GC に対処するためのオーバヘッドが生ずる．

その他，融合型言語の研究として，実装の詳細にまで触れた言語として，Leda[3] がある．Leda では，たとえば，単一化もユーザレベルでプログラムする必要があり（むしろ，そのようなレベルの記述が可能なことが Leda の特徴），従来の実装技術を適用することは難しい．一方，TAO では基本的には WAM が適用可能である．

7. ま と め

本論文では，次のような特徴を持つ言語を実装するための抽象マシンを提案した．1）パターンマッチとガードによって節を選択し，後戻りを陽に行う．2）関数（Lisp）と述語（LP）が互いに呼びあうことができ，任意のデータを渡せる．

Lisp との融合と，並行 GC 下での実装により生じる問題を，次のようにして解決した．1）単一化の生成したデータが Lisp の副作用によって永続的に保持される可能性があるため，構造データをヒープ上に表現する．2）レジスタの使用量をおさえるために，単一化，引数処理，パターンマッチをスタックを用いて行う．3）つねにメモリ状態を無矛盾にするため，単一化途中でも正しいデータ構造を維持する．

また，他の処理系と比較し，スタック使用量は最大数十%程度の増加，トレール使用量は減る場合があること，実行速度は多くのプログラムで WAM とほぼ同じ特性を示すことなどを示した．絶対的な実行性能としては，ハードウェアテクノロジを考慮すると，他の処理系と同程度であることも確認した．

今後の課題としては，本論文で採用した各部分の実装方式を，詳細に個別評価する必要がある．

参 考 文 献

1) Aït-Kaci, H.: *Warren's Abstract Machine*, MIT Press (1991).
2) 天海良治，山崎憲一，中村昌志，吉田雅治，竹内郁雄：TAO のコンカレンシ・コントロール，情報処理学会記号処理研究会 69-3 (1993).
3) Budd, T.A.: *Multiparadigm Programming in Leda*, Addison-Wesley (1995).
4) Meier, M.: Compilation of Compound Terms in Prolog, *1990 North American Conference on Logic Programming*, pp.63–79 (1990).
5) Older, W.J. and Rummell, J.A.: An Incremental Garbage Collector for WAM-Based Prolog, *1992 Joint International Conference and Symposium on Logic Programming*, pp.369–383 (1992).
6) Pittomvils, E., Bruynooghe, M. and Willems, Y.: Towards a Real Time Garbage Collector for PROLOG, *1985 Symposium on Logic Programming*, pp.185–198, IEEE Computer Society (1985).
7) 竹内郁雄，天海良治，山崎憲一：新しい TAO の設計，情報処理学会記号処理研究会 56-2 (1990).
8) 竹内郁雄，吉田雅治，山崎憲一，天海良治：実時間記号処理システム TAO/SILENT における軽量プロセスの実現，情報処理学会論文誌，Vol.38, No.3, pp.595–605 (1997).
9) 竹内郁雄，吉田雅治，山崎憲一，天海良治：実時間記号処理システム TAO/SILENT の並行 GC，日本ソフトウェア科学会 SPA99 (1999).
10) Van Roy, P.: Aquarius Benchmarks, anonymous ftp from "ftp://gatekeeper.dec.com/pub/plan/prolog/AquariusBenchmarks.tar.Z".

11) Warren, D.: An abstract Prolog instruction set, Technical Report Technical Note 309, SRI International (1983).
12) 山崎憲一，天海良治，竹内郁雄，吉田雅治：ヒープを使用する論理型言語でのトレール方式，情報処理学会記号処理研究会 67-4 (1993).
13) 山崎憲一，天海良治，竹内郁雄，吉田雅治：TAO/SILENT のバイトコード実行方式，日本ソフトウェア科学会 SPA98 (1998).
14) 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).
15) 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).
16) Zhou, N.: Parameter Passing and Control Stack Management in Prolog Implementation Revisited, *ACM Trans.Prog.Lang.Syst.*, Vol.18, No.6, pp.752–779 (1996).
17) 関田大吾：Inside KLIC (1998). “http://www.klic.org/software/klic/inside/master.html” より入手可.

付　　録

A.1 TAO の主な文法

プログラム ::= 式$^+$
式 ::= 定数 | 変数 | 作用素呼び出し式 | メッセージ式
　　| 構文式 | 代入式 | 単一化式
定数 ::= シンボル | 文字列 | 数字 | `#t` | `#f`
変数 ::= シンボル
構文式 ::= 作用素生成構文 | 作用素定義構文
作用素定義構文 ::= (`define` シンボル 作用素生成構文)
作用素生成構文 ::= 関数生成構文 | 述語生成構文
関数生成構文 ::= (`op` 仮引数定義 式$^+$)
述語生成構文 ::= {`op` 節$^+$}
節 ::= ([`:clause`] ヘッド [(`:aux` 局所変数宣言$^+$)]
ガード
　　[(`:choice` [名前])] [(`:before-fail` 式)]
　　ボディ*)
ヘッド ::= ([`:head`] Hパターン*)
ガード ::= ([`:guard`] 式)
ボディ ::= 式 | (`:on-fail` 式)
H パターン ::= 定数 | _変数 | (H パターン . H パターン)
B パターン ::= 定数 | _式 | (B パターン . B パターン)
作用素呼び出し式 ::= 関数呼び出し式 | 述語呼び出し式
関数呼び出し式 ::= (関数名 式*)
述語呼び出し式 ::= {述語名 Bパターン*}
メッセージ式 ::= 関数メッセージ式 | 述語メッセージ式
関数メッセージ式 ::= [式 (メッセージ名 式*)]
述語メッセージ式 ::= [式 {メッセージ名 式*}]
代入式 ::= (`!`式 式)
単一化式 ::= {`!` B パターン B パターン}

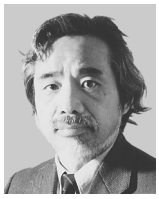
ここで，キーワード（`:choice` など）を car に持つ式は，構文の一部であり，関数呼び出しではない．本文中に現れる `defpred` は，(`define` 名前 {`op...`}) というマクロである．

(平成 11 年 5 月 18 日受付)
(平成 11 年 11 月 4 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．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．記号処理専用計算機，記号処理プログラミング言語，日本語文書処理の研究に従事．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，日本電信電話（株）NTT サイバースペース研究所サイバー入出力プロジェクト主任研究員．並列処理・画像生成・記号処理等の専用計算機，知能ロボット用センサ等のハードウェアの研究に従事．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 年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会員．