記 号 処 理 64－1
（1992. 3. 16）

TAO/SILENTの論理型プログラミング

山崎憲一　天海良治　竹内郁雄　吉田雅治
NTTソフトウェア研究所, NTT基礎研究所, NTTヒューマンインタフェース研究所

記号処理カーネルSILENT上に設計中の言語TAOの論理型プログラミングについて述べる. TAOは, その上にさまざまな言語を構築可能にすることを目指しており, そのためのデータ操作と実行制御のプリミティブを提供する. その意味でTAOは上位言語に対して機械語であるといえる. 本報告では, 論理型プログラミングを低レベルの機能に分類して, 論理型プログラミングのプリミティブとは何かを考察し, TAOでそれらがどのように提供されているかについて述べる. 特にバックトラックの制御機構に関してはLispの制御機構とより緊密な融合を行った. また, TAOは構文によりプログラムの意味を積極的に表現するという設計方針をとっており, 読みやすい言語となっている.

Logic Programming in TAO/SILENT

Kenichi Yamazaki　Yoshiji Amagai　Ikuo Takeuchi　Masaharu Yoshida
NTT Software Laboratories, NTT Basic Research Laboratories,
NTT Human Interface Laboratories

This paper discusses the logic programming on TAO, which is designed on a symbolic processing machine SILENT. The goal of TAO is to enable building a wide variety of high-level programming languages on top of TAO. TAO provides primitives of data operations and execution controls for the purpose. In this sense, TAO can be deemed a machine language. In this paper, we classify the features of logic programming, discuss what are the primitives of logic programming, and then describe how such primitives are provided by TAO. Especially, the control mechanism of backtracking is merged into that of Lisp. We also discuss that TAO is designed to express program semantics by its syntax, so that its program is easy to read.

1 はじめに

SILENT[1] は現在我々が設計・開発を行っている記号処理プロセッサである. TAO[2] は SILENT 上で動作する言語であり, 高速実行, マルチパラダイム, 並行記述, 実時間記述という特徴を持つ. TAO は, それ自身で完結した言語であることを目指すのではなく, むしろ言語を含む新しいプログラミング基盤を構築するためのシステムであることを目指している. これは, 並行記述, 実時間記述など現在盛んに研究されている分野, あるいはコネクティクス [3] など開拓が始まったばかりの分野をサポートするエンジンとして TAO/SILENT を考えた場合, 現段階で仕様を固定してしまうよりは, 可能な限りオープンなプログラミング基盤を提供するほうが好ましいであろうという判断による.

本報告では, このような目的を達成するための設計方針についてまず述べ, 次にその論理型プログラミングの機能について述べる.

2 TAO/SILENT の設計

2.1 設計方針

最も基本的な設計方針は「TAO は機械語である」ということである. TAO/SILENT はハードウェアアーキテクチャ SILENT の上に, ソフトウェアアーキテクチャ (SA) と呼ばれる言語システムを構成する [4]. SA は, いわば抽象計算機であり, TAO はその機械語である. TAO は SA 上で提供されるデータ型と制御に関するすべてのプリミティブを提供する. 一般のユーザは SA 上に新たに定義される言語 TAO_n を使う. TAO_n は TAO (TAO_n と明確に区別するために TAO_0 とも呼ばれる) の機能の組み合わせ, あるいは機能の削除によって, その仕様が定義される. 例えば, 教育用の言語として pure Lisp に近い Lisp, また型検査の厳しい Lisp などを SA 上に定義することができる.

今回報告する TAO_0 の論理型プログラミング機能も, この方針に従い, 機械語相当のレベルの機能を提供する. なお, 主にユーザが使用するであろうと思われる, TAO_1 以上の機能についても一部述べる.

2.2 TAO/ELIS の問題点

我々は専用計算機 ELIS 上のシステム TAO/ELIS[5] を開発し, 多くのプログラムを開発してきた (なお以下では ELIS 上の TAO を olD tAO の意味で DAO と呼ぶ). これにより DAO の持つ, さまざまな問題点が明らかになった. 論理型プログラミングに関しては, 以下のような問題点がある.

2.2.1 構文に関する問題

DAO では, (Lisp) 関数と (論理) 述語は作用オブジェクト (applicative object) であり, 区別はなかった. また述語は失敗すると `nil` を返すが, 述語のボディ中の作用オブジェクトが `nil` を返すと, それが関数であっても, 述語であっても, バックトラックが起きる. これは「述語は成功か失敗を返す関数である」と考えることに相当する. このようにすると関数と述語を統一して扱うことができる. DAO はこのような言語モデルに基づいて設計された.

しかし, 実用面を考えると両者の構文が同じであるため, 次のような問題が生ずる. 例えば

```
(foo _x _y)
```

という式があった時, `foo` が関数であるのか述語であるのかは, `foo` の定義を調べなければわからない. 我々は DAO を用いて幾つかのプログラムを作成した. これにより次のようなことがわかった. 述語の中から関数を呼ぶときには, 次のいずれかを行いたい場合が多い.

- 条件判定
  関数の引数は必ず値が確定しており, いわゆる逆方向の演算に使われることはない.
- 副作用を伴う操作
  論理型プログラミングの世界外に影響を与える.

関数であることがわかれば, 関数名より上記のどちらであるかは予測できるから, プログラムが読みやすくなると思われる.

このように, プログラムの意味を構文に反映させることで, プログラムの読みやすさが向上する. TAO ではこの考えを進め, 構文により積極的にプログラムの意味を表すというアプローチを採用する. このためには, 強い

型付き言語にするのも1方法ではあるが，プロトタイピング向きでなくなるので，型なし言語の範囲で行う．

2.2.2 ユニフィケーションに関する問題

DAOのユニフィケーションには次のような問題があった．

1. 配列，ベクタの要素へ代入ができない
   構造を持つデータのうちユニフィケーションで扱えるのはリストだけである．
2. 論理変数と(Lispの)変数が区別されていた
   論理変数名はアンダースコアで始まらなければならない．2種類の変数が存在するために統一性に欠け，言語を複雑にしていた．
3. 評価を呼び出す記法がわかりにくい
   ユニフィケーションのパターン中の項の前に“,”を書くことで，その項を式として(Lispの意味で)評価してからユニフィケーションすることができる．“,”はその他にも，場所により評価に似た意味がオーバーロードされており，わかりにくく，言語を複雑なものにしていた．

TAOでは，以上の3点の問題を，ユニフィケーション中から評価を呼び出す記法を拡張することでまとめて解決している．

3 論理型プログラミング

3.1 論理型プログラミングの機械語

前述のようにTAO(より正確にはTAO_0)は，ソフトウェアアーキテクチャにおける機械語である．それらの組み合せで上位言語を構築することが目的であるから，1つ1つのプリミティブが細かい程，上位層が柔軟になりうる．もちろん，あまり細かすぎても(例えばレジスタ転送のレベルにしても)意味がないのは明らかであり，十分に低レベルでありながら，論理型プログラミングの利点を失ってはならない．

論理型プログラミングを構成するプリミティブとして，次のような機能を挙げることができる．

- ユニフィケーション
- 述語呼び出し
- 浅いバックトラック
- 深いバックトラック
- カットなどの実行制御機構

ユニフィケーションをさらに分解して，比較とトレール付きの代入などのレベルのプリミティブとすることもできる．しかし，そのようなレベルでは，論理型プログラミングというよりは手続き的プログラミングと呼ぶべきであろう．このためユニフィケーションは，それ以上分割不可能なプリミティブであるとする．

次にバックトラックについてであるが，バックトラックをプリミティブとして分離することは，論理型プログラミングを，宣言的なプログラミングとしてとらえず，より手続き的にとらえることを意味する．純粋な宣言的プログラミングとしてとらえる立場では，バックトラックによるSLD木の左方優先探索などは，あくまでも実装の問題である．しかし，実際に論理型プログラミング，特にPrologでのプログラム例を考えると，カットを多用した，実行順序を意識したプログラムの方が多いと思われる．TAOでは，こういった点からバックトラックを分離することとした．さらに，バックトラックを2種類に分けたのは，両者の意味が異なるからである．すなわち浅いバックトラックは条件分岐，つまりLispにおけるcondと同様の意味であるのに対し，深いバックトラックは，述語全体の失敗を意味し，別の枝を探索するべきであることを意味する．

また，カットなどの実行制御機構も，手続き的に解釈したほうが自然であるから，関数としてTAOに取り入れることとする．

3.2 関数と述語

前述のように，構文によりプログラムの意味を積極的に表す，という設計方針から，関数呼び出し式と述語呼び出し式は異なる構文を持つ．それぞれ，

関数呼び出し式… (関数名 引数$_1$ 引数$_2$ ... 引数$_n$)
述語呼び出し式… { 述語名 . パターン }

という構文により表現される．これにより，定義本体を参照せずに，その呼び出しが何に対するものであるのかを識別できる(オブジェクト指向におけるメッセージ式も固有の構文を持つが，それについては文献[2]を参照

のこと). このような呼び出し式は, 評価が可能なところであれば, どこでも書くことができる. 原則として述語呼び出しが失敗すると nil が返され, 成功するとtが返される.

このように構文を分けたことにより, 1つのシンボルに対し関数と述語を同時に定義することが可能となった. 例えば DAO では car という述語を定義することは不可能であったが, TAO では可能である. 実際, システム関数で, このような意味でオーバーロードされているシンボルとしては, lambda, expr, equal, and, append などがある.

TAO では, 通常の Lisp の apply, funcall はプリミティブではない. また, 式の car は評価されないため, ((lambda ...) ...) のような式も許されない[1]. それぞれの詳細な説明は [2] で報告したので, ここでは本報告で使われる式の説明のみにとどめる. 通常の Lisp の

(apply fn arg_1 arg_2 ... arglist)

は TAO では

(,fn arg_1 arg_2 ... . arglist)

と表される. ","は評価の意味を表し, 上の式全体では, fn を評価して, その値を式の car として, 式全体を評価することを意味する. 式全体としては通常の関数呼び出しと同じ形をしているので, 例えば引数がない apply は, 単に

(,fn)

と書く. また lambda を式の car で直接使うには

(,(lambda ...) ...)

と書く. 述語呼び出しにおいては次の節で述べるようにアンダースコアが評価の意味を持つので, 後述する論理型プログラミングにおける lambda は,

{_{lambda ...} ...}

と表される.

3.3 ユニフィケーションと変数

2.2.2 節の 3 つの問題を解決するために, ユニフィケーション中から評価を呼び出す新たな記法を導入する. ユニフィケーションのパターン中の項の前に, アンダースコア ("_") がつけられると, ユニフィケーション前にその項が式として評価される. そして, その値に対しユニフィケーションがなされる. この評価は正確には, Lisp の eval と異なる. 評価結果が未定義値のときには, その場所が保持され, ユニフィケーションにより, そこに代入をすることができる. 式として変数も許されるので, 結局ユニフィケーションのパターン中の "_x" は変数 x を意味する. 変数は Lisp の世界で定義されたものでも良いから, リストを連結する述語 append を次のように呼び出せば, リスト (1 2 3 4) が印刷される.

```
(let ((x '(1 2))
      (y '(3 4))
      _z)
  {append _x _y _z}
  (print z))
```

let を含む TAO のあらゆる変数宣言においてアンダースコアがつけられた変数の初期値は未定義値である. 上の例では z の初期値が未定義値になり, append により リスト (1 2 3 4) が代入される. なお, アンダースコアだけがユニフィケーションのパターン中に現われた場合, これは Prolog の無名変数を表す. すなわち, いつでも任意のデータとパターンマッチする変数を表す.

式として関数呼び出し式も許されるので, この機能を用いて未定義値の入った配列へユニフィケーションで代入できる. 例えば

```
(let ((a (array 10
           :initial-element (undef))))
  {foo _(aref a 3)})
```

により, 要素が未定義値の大きさ 10 の配列が生成され, 添え字 3 の要素が述語 foo に渡される. そして foo のユニフィケーションによって値が代入され得る[2].

ユニフィケーションでの代入はすべて履歴が保存される (トレールされる). 代入されるのが配列であっても, トレールされる. Prolog などの単一代入言語において, 配列の書き替えを論理型言語の枠組の中で実現する方法として, 新たに配列を作り, 代入する要素以外のすべての要素を複製する (正確には複製したように見せる) という方法があるが, TAO ではこの方法はとらず, ユニフィケーションで代入ができるのは, その要素が未定義値の

[1] Common Lisp と異なり (lambda ...) は評価されて初めて関数オブジェクトになる

[2] キーワード引数を, 説明の便宜上用いたが, TAO_0 での取り扱いは検討中である.

ときのみであるとする．これは配列の書き換えはまさに副作用であり，そういった操作はLispの世界で行うべきであるからである．もちろん，Lispにおける副作用はバックトラックでは取り消せないが，副作用は本来取り消せるべきものではない．

3.4 実行の制御

述語を定義する特殊形式defpredは次のような構文で書かれる．

```
(defpred 述語名
    (パターン [(:aux 変数...)] ガード式
              [(:catch [名前])] [ボディ式...])
    (パターン [(:aux 変数...)] ガード式
              [(:catch [名前])] [ボディ式...])
    ...
    (パターン [(:aux 変数...)] ガード式
              [(:catch [名前])] [ボディ式...]) )
```

ここで，:auxで始まるリストは局所変数の宣言である．例えばリストを連結する述語appendは

```
(defpred append
    ((() _x _x) t)
    (((_a . _x) _y (_a . _z)) t
     {append _x _y _z}) )
```

のように記述される．

述語定義中で，パターンで始まるリストを節とよぶ．述語中に現われる局所変数は節間で独立である．各節で現われる局所変数は宣言されなければならない．パターン中に現われる局所変数は，現われることをもって宣言とされるが，ガード，ボディ中に初めて現われる局所変数は:auxで始まるリストにより宣言されなければならない．なお，TAO_1以上ではよりPrologに近い構文を持つマクロassertが提供される．これを使えば変数宣言などを意識する必要はない．assertの構文は次のとおりである．なお，以下では読みやすさのため，適宜assertを用いて説明を行う．

```
(assert { 述語名 . パターン }
        [[ガードリスト] | [ボディ式...]])
```

あるいは

```
(assert { 述語名 . パターン }
        [ガードリスト] || [ボディ式...])
```

ガードリスト，ボディ式に現われる変数は局所変数として，自動的に変数宣言に加えられる．また，defpredではガード式に複数の式を書きたい場合には{and ガードリスト}のようにして全体で1つの式にしなければならないが，assertではガードリストに複数の式があれば，自動的にand式にするのでこの必要はない．また，後述するオプショナルの(:catch)がある節ではガードとボディの区切りが，"||"になる．

ガードはGHCなどのガードと似ている．ガードにはシステムで許された述語と，Lisp関数が書ける．逆にユーザ定義の述語は記述できないため，ガード中に非決定的な述語は書けない．GHCなどと異なりガード中のユニフィケーションで変数に代入することは自由である．ガードの実行中においては，その結果がnilであれば失敗を意味する．従ってガード中に直接Lispの式，例えば(> x y)を書けば，xとyの大小関係が判定でき，それによってガードを失敗させることができる．

述語呼び出しが行われてからの動作は次のようになる．まず，呼び出し側の引数と最初の節のパターンのユニフィケーションがなされる．これが成功すると，必要な局所変数が宣言され，ガードの実行に入る．ガードの実行が失敗するとこの節の実行は失敗し，変数の代入が取り消されたのち，次の節に対して上記の動作を行う．もし次の節がない場合には実行時エラーとなる．一方ガードが成功するとボディの実行にはいる．ボディの実行は基本的にLispの暗黙のprognに相当する．すなわち，ボディの要素の返す値に関係なく，実行は左から右へと進む．そして，最後の要素の返す値が，その述語の値として返される．ボディのない節，いわゆる事実節はtを返す．

Prologではバックトラックが虫により起きたときに，それがどの述語で起きたのかを調べるのが非常に難しい．TAOでは，浅いバックトラックが最後の節で起きると，実行時エラーとなり，そこで停止するため簡単にわかる．

以上述べたように，TAOの論理型プログラミングはデフォルトでは浅いバックトラックしかサポートしない．しかし，SLD導出のsoundnessが失われたわけではない．実行が成功したときには，それは正しい答えだからである．この意味で，述語を用いてプログラムする限り，それは「論理型」プログラミングであるといえる．一方，TAOでは深いバックトラックは導出を制御するものとして，論理型プログラミングとは別なものと考える．こ

のため，深いバックトラックの制御は次に述べるように関数によって制御される．

3.5 深いバックトラック

TAOで深いバックトラックを行うには，ユーザがプログラム中に（:catch）と 関数（throw）を書いて明示的に制御しなければならない．（:catch）が書かれた節はガードが成功すると，チョイスポイントと呼ばれる情報を記録する．ただし，最後の節に（:catch）が書かれていても，チョイスポイントは記録されず，単に無視される．一方関数（throw）が呼び出されると最も新しいチョイスポイントを探し，それに対応する節の次の節に実行が移される．同時にそのチョイスポイントまでの代入を取り消す．なお，Lispのcatchについては後述する．

例えば

```
(assert {foo} || {bar})
(assert {foo})

(assert {bar} | (throw))
```

という定義があるとする（"||"は（:catch）に展開されることに注意）．{foo}が呼ばれると，barの（throw）によって，fooの第2節へ実行が移る．

（:catch）と（throw）によってPrologのプログラムをTAOに機械的に変換することができる．基本的には

H_1 :- B_{11}, B_{12}, ...

H_2 :- B_{21}, B_{22}, ...

H_3 :- B_{31}, B_{32}, ...

という述語Hを定義するPrologプログラムは

```
(assert H'1 || B'11 B'12 ...)
(assert H'2 || B'21 B'22 ...)
(assert H'3 | B'31 B'32 ...)
(assert {H . _} (throw) |)
```

というTAOのプログラムに変換される（B′はBという述語呼び出し式をTAOの構文に変換したもの）．

重要なのは最後の節である．もし最後の節がないと，最初の3つの節がすべてユニフィケーションに失敗したとき，もう節がないにも関わらず，浅いバックトラックが起きるため実行時エラーとなる．これを避け，H'_3の失敗によって深いバックトラックを起動するためには，最後の節が必要となる．また，最後から2番めの節（ここではH'_3の節）だけが"||"でなく，"|"であることにも注意されたい．最後の節は単にバックトラックするだけであるから，H'_3の節は，そのユニフィケーションが成功したら，最後の節にバックトラックで戻る必要はないので，"|"で良いわけである．これは実装上はスタック消費を減らす効果があり，WAMと同程度のスタックしか消費せずに済む．

（throw）は，当然Lispの（catch ...）式[3]に対する大域脱出としても使える．実はチョイスポイントは，Lispのcatchによって作られるキャッチポイントと，実行制御に関しては機能的に等価である．ここで説明の便宜上，キャッチポイントとチョイスポイントを総称してCポイントと呼ぶ．（:catch 名前）と書いてチョイスポイントに名前を付けることができる．（throw 名前）が評価されると，その名前のCポイントが探される．関数throwの構文とその意味を以下に示す．

（throw）... 最新のCポイントを探す．

（throw 名前）...

名前で指定された最新のCポイントを探す．

（throw nil）...

名前のない最新のCポイントを探す．

述語の（:catch）とLispのcatchは動的にネストすることも可能なので，関数throwの正確な意味は次のようになる．まず，throwの引数で決められるCポイントを探す．探されたCポイントより新しいCポイントの中にチョイスポイントがあれば，代入を戻す．探されたCポイントがチョイスポイントであれば，その節の次の節へ制御を移す．それがcatchが作ったキャッチポイントであれば，Lispと同様の動作になる．どちらの場合でも，途中のunwind-protectはすべて処理される．

なお，assertにおいては，"||"のかわりに，"| 名前 |"[4]と書くと，（:catch 名前）にマクロ展開される．

[3] TAO_0ではcatcherというCommon Lispとは異なる大域脱出プリミティブが提供されるが，ここでは説明の便宜上catchを用いる．

[4] TAOはCommon Lispと異なり"| 名前 |"はシンボルを意味しない．

3.6 無名述語

Lisp において lambda を用いて無名関数を作れるように, 次のようにして無名の述語を作ることができる.

```
{lambda
    (パターン [(:aux 変数...)] ガード ボディ式...)
    (パターン [(:aux 変数...)] ガード ボディ式...)
    ...
    (パターン [(:aux 変数...)] ガード ボディ式...)}
```

DAO でも特殊形式 hclauses と &+ を使って, 無名述語を定義できたが, TAO では両者を lambda に統合した.

lambda を用いると, Prolog ではプリミティブであった制御述語が記述できる. 例えば, or (Prolog の ";") は次のように書ける.

```
(assert {foo} |
        {_{lambda (() t (:catch) {bar})
                  (() t {baz}) }})
```

{foo} が呼ばれると, {bar} が成功するが, その後, もし (throw) が呼ばれると, {baz} に制御が移る.

なお, {lambda ...} 式はレキシカル環境を取り込む. 一方レキシカル環境を取り込まない {expr ...} も提供されている. 例えば, defpred は内部でこれを用いて, 述語を定義している.

3.7 その他

or の他に and, if なども lambda を用いて表せるため, 実行制御プリミティブはカットだけである. カットは, assert では Prolog と同じシンボル "!" が準備されるが, TAO_0 では関数 (cut) によって呼び出す. カットも深いバックトラック同様に実行制御を行うものであるから, 述語でなく関数として提供されるわけである.

(cut) が呼び出されると, それ自身をレキシカルに含む述語が作ったチョイスポイントより新しいすべてのチョイスポイントを削除する. (cut) は Prolog のように, レキシカルにカットする述語を決定する. 一方, (cut 名前) と名前を指定して呼び出すと, (cut) と同様の動作をした後, 自分をレキシカルに含む述語を呼び出した述語を上にたどり, その名前を持つ述語までのチョイスポイントを削除する. すなわち, 名前付きのカットは, 大域的なカットである.

4 例題

TAO による 8-queen の記述を以下に示す.

```
(assert {queen _n _l} |
        {try _n _(index n 1 -1)
             () _l () ()} )

(assert {try _ () _l _l _ _})
(assert {try _m _s _l1 _l _c _d} |
        {select _s _a _s1}
        {equal _c1 _(+ m a)}
        (if (memq c1 c) (throw))
        {equal _d1 _(- m a)}
        (if (memq d1 d) (throw))
        {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} )
```

ここで, {equal ...} は第1引数と第2引数のユニフィケーションを行う述語である. 8-queen はバックトラックにより解を探索するプログラムであるが, (:catch) ("!!") と (throw) によって, 深いバックトラックがどこで生ずるかが極めて明らかになる. 逆にいうとプログラマはどこで深いバックトラックが起きるのかを意識してプログラムしなければならない. しかし, 多くの場合は, 意識してプログラムすることが多いので, 負担にはならないと思われる. 一方このプログラムを初めて読む者にとっては, バックトラックの構造が明確で, バックトラックしない部分は左から右に読めば良いため, 読みやすいであろう.

浅いバックトラックをデフォルトとしたときのもう1つの利点は, カットを減らせる可能性があることである. カットは他の選択肢を破棄したい場合に用いるが, その他に, 本来カットは不要であるのに, 単にスタックを削るためだけにカットを使うことがある. TAO では真にバックトラックが必要な節のみが, そのことを (:catch) を用いて陽に示すので, 無駄にスタックが使われることはない. このため, カットは本来の使用法のみに用いられ, プログラムが理解しやすくなると思われる.

5 今後の課題

5.1 実装

TAOの論理型プログラムは基本的にWAMと同様のコードにコンパイルされる．ただし，(:catch)のない節においては，浅いバックトラックのみに備えれば良いので，チョイスポイントに関する処理が簡単になる．このため，(:catch)を使わない述語においては速度向上が期待できる．(:catch)がある場合には，WAMと同じ方法でコンパイルできる．ただし，(:catch)がある節とない節とでバックトラックの方法が異なるため，これを表すモードが必要になる．

また，ユニフィケーションにおけるアンダースコアによる評価についての実装は，TAOの自己代入の評価機構をそのまま用いる．TAOでは例えば

```
(!!car !(car x))
```

という式を実行すると，xのcarのcarを取り，それをxのcarに代入する．これを自己代入と呼ぶ．自己代入のために，評価結果のある場所を保持するという機構が基本評価機構にすでに組み込まれており，アンダースコアによる評価の機構は，容易に実現可能である．

なお，SILENTのクロックはELIS-8100シリーズの6倍であり，ハードウェアサポートによりWAMのレジスタ転送命令は，4クロックから1クロックに短縮される．ELIS-8100上のDAOの論理型プログラムのコンパイルコードの実行速度は約40KLIPSであるから，SILENT上のTAOでは500KLIPS以上の性能が期待される．

5.2 マクロ

DAOにおいては関数呼び出しの式も述語呼び出しの式も同じ構文であるから，両者ともdefmacroで定義されたマクロによるマクロ展開が可能である．一方TAOでは述語呼び出しは構文が異なるため，defmacroで定義されたマクロは呼び出されない．述語呼び出しのためのマクロはdefpred-macroで定義される．このマクロでは，例えば展開結果が(foo _x _y)であっても，これは{foo _x _y}に変更され，これが本当の展開結果となる．orやandはこのマクロ機能を用いてlambdaに展開される．

defpred-macroはdefmacroと同様に定義する．すなわちマクロ展開はLispによって行われる予定である．しかし，論理型プログラミングでマクロ展開を行う方が，わかりやすい可能性もあり，現在検討中である．

6 さいごに

構文によりプログラムの意味を積極的に表すというアプローチに基づいて設計されたTAOの論理型プログラミング機能について述べた．TAOはTAO/SILENTソフトウェアアーキテクチャの機械語に相当する部分であり，細かい低レベルの記述ができなければならない．論理型プログラミングの機械語について考察し，より低レベルの記述を可能とする機能を取り入れた．今後は，言語の細部をリファインするとともに，実装について検討し，実際にSILENT上に実現する．また，オブジェクト指向論理型プログラミングはDAOのものを基に，新たに設計中であり，これについては別の機会に報告したい．

TAOの全般にわたって東大のJ. A. Robinson教授には熱心な議論を通して，貴重なご意見を頂いた．また，バックトラックをLispのcatch, throwととらえる考え方はNTTソフトウェア研究所の梅村恭司氏との議論からヒントを得た．両氏を始めとして，この研究について議論して頂いた多くの方々に感謝します．

参考文献

[1] 吉田, 竹内, 山崎, 天海: 新しい記号処理カーネルSILENTの設計, 記号処理研究会, 56-1, 1990.

[2] 竹内, 吉田, 天海, 山崎: 新しいTAOの設計, 記号処理研究会, 56-2, 1990.

[3] 竹内, 後藤, 尾内, 斉藤, 奥乃: コネクティクス構想, 日本ソフトウェア科学会第8回大会, pp.233-236, 1991.

[4] 天海, 竹内, 吉田, 山崎: TAO/SILENTのソフトウェア・アーキテクチャ, 日本ソフトウェア科学会第8回大会, pp.57-60, 1991.

[5] I.Takeuchi, H.Okuno, and N.Ohsato: A List Processing Language TAO with Multiple Programming Paradigms, *New Generation Computing, Vol.4 No.4*, 1986.