記 号 処 理 50 － 3
（ 1 9 8 9. 3. 1 4 ）

TAO/ELISにおける論理型プログラムのコンパイラ

山崎 憲一
NTTソフトウェア研究所

マルチパラダイム言語 TAO の論理型パラダイムコンパイラの設計および実装方法について述べる。論理型以外のパラダイムとの融合機構を持つ言語のコンパイラを WAM 方式をベースとして Lisp マシンのアーキテクチャ上に実現する方法について述べる。現在の Prolog コンパイラでは過度の最適化のためコンパイルされたコードのデバッグは事実上不可能であるが、本コンパイラでは実行履歴を積極的に残すことでこれを可能にした。今回開発した最適化技法のうちの幾つかは従来の WAM コンパイラにも適用可能である。

The Compiler for the Logic Paradigm of TAO

Yamazaki Kenichi
NTT Software Laboratories
9-11 Midori-Cho 3-Chome, Musashino-Shi, Tokyo 180

A logic paradigm compiler for a multiple programming paradigm language TAO/ELIS is described. The main problem we attack is to design and develop a logic paradigm compiler which 1)provides the same programming environment as the interpreter and 2)generates efficient compiled codes. Compiled codes keep minimum information of its execution history so that a programmer can debug compiled codes as if they were interpreted codes. This paper also describes many optimization techniques adopted in TAO/ELIS system, some of which are applicable to ordinary WAM based Prolog compiler as well.

1. はじめに

TAO[86Takeuchi] は Lisp 専用計算機 ELIS 上のマルチパラダイム言語である。現在、TAO の 3 つのパラダイム、即ち関数型パラダイム、オブジェクト指向パラダイム、論理型パラダイムのうち前者 2 つのコンパイラは既に存在する。今回論理型プログラムのコンパイラを設計、製作したので報告する。

TAO はプロトタイピングを効率良く行うためにインタープリタを重視しインタープリタでも十分な実行速度を提供するように設計、実装されている。このため TAO ではコンパイラはプログラムの完成後に主に速度向上の目的で用いられる。論理型パラダイムにおいてもこのことは同様であり、ワークステーション上などの Prolog インタープリタに比べ数倍から 10 倍程度の実行速度を持つ。しかしインタープリタにおいてはインデキシングなどによる高速化が難しいため、データベースなどの大量データの処理プログラムにおいては十分な実行速度が得られず、結局プロトタイピングの段階からコンパイラが必要となる。このためプロトタイピング環境を重視するという TAO の思想を満たし、かつ実行速度を向上させるコンパイラが必要である。

2. TAO の論理型プログラミングパラダイム

TAO の論理型パラダイムは機能的には Prolog にほぼ等しいが、他パラダイムとの融合機構などのためにそのインプリメントは従来の方法と大きく異なる。本節では従来の Prolog と異なる点に関しまず言語仕様について、続いてインタープリタの実装方法について述べる。

2.1 TAO の論理型プログラミング

TAO の論理型プログラムは S 式で記述される (例 1)。

例 1: リストを連結する述語 &append

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

TAO では関数と述語の区別は無いが、プログラムの読解性を上げるため論理型で記述されているものには & で始まる名前を付ける慣習がある。ここで重要なのはこれにより構文上の違いだけでなくユニフィケーションにおいて重要な違いが生じるということである。例えば TAO では例 2 のようなユニフィケーションが記述できる。

例 2: 第 1 引数を _x に、残りの引数のリストを _y にユニファイする述語 &foo

```
(assert (&foo _x . _y) (write _x) (write _y))
```

次に他パラダイムとの融合機構について述べる。2 つの方法で論理型プログラムから関数型プログラムを呼び出すことができる。1 つは述語のボディ部から直接関数を呼び出す方法である。例 2 でボディ部に書かれた S 式 (write _x) は実は関数 write を呼び出す。実は TAO では (関数型パラダイムの)「関数」と (論理型パラダイムの)「述語」という区別はない。以下では便宜的に defun で定義されたものを関数、assert で定義されたものを述語と呼ぶ。従って述語も値を返す。実行が失敗すると述語は nil を返し、これによりバックトラックが起動される。同様にある関数が nil を返してもやはりバックトラックが起動される。従って例えば Prolog の組み込み述語 fail は TAO では nil と書くだけで良い (例 3)。

例 3: &foo の全解を表示する述語 &all-foo

```
(assert (&all-foo) (&foo _x) (progn (write _x) t) nil)
```

write が nil を返すこともあるので progn を使い必ず t を返すようにしている

より正確には、TAO ではあるフォームを評価する方式には「評価 (eval)」と「R 評価 (reval)」がある。前者はいわゆる Lisp の eval であり、以後では特に明示したい時に限り「L 評価」と呼ぶ。また後者の R は resolution の意味である。述語を L 評価すると結果に応じた値 (失敗した時は nil 成功した時は nil 以外) が返り、また同時に選択点の情報はすべて失われる。一方、R 評価では評価値が nil であれば最後の選択点 (Last Choice Point) までバックトラックが起こる。R 評価を行えるのは述語のボディ部の中の呼び出しだけであり、明示的に評価法を指定することはできない。

もう1つの方法はユニフィケーションによって呼び出すことである。TAOではユニフィケーションの実行直前に、あるフォームを評価することができる(例4)。

例4: 階乗を計算する述語 &fact

```
(assert (&fact 0 1))
(assert (&fact _n _result)
        (&fact ,(1- _n) _result1)
        (== _result ,(* _n _result1)) )
```

ここで == は第1引数と第2引数をユニファイする述語である

コンマ(,)が付けられた引数はユニフィケーションを開始する直前にL評価され、その値が実際のユニフィケーションの対象となる。

一方、関数型プログラムから論理型プログラムを呼び出す時は関数と同じように(述語名 引数$_1$ ... 引数$_n$)と書けば良い。述語がL評価され、もし失敗すればnilが評価値となる。

上記のような他パラダイムとの融合機構を用いたプログラムの例題としてn-queenのプログラムを例5に示す。コンマを用いたために中間変数が減り見やすくなっている。また組み込み関数を利用しているためプログラムサイズも減少している。さらに重要なことは速度が向上することである。8-queenの全解探索の実行時間はSUN3/260上のQuintus-Prologコンパイラで1.45秒であるのに対し、TAOはインタープリタで1.93秒である。(コンパイラでは1.01秒)。

例5: N人の女王問題を解くプログラム

```
(assert (&queen _n _l)
        (== _l1 ,(index _n 1 -1))
        (&try _n _l1 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 ,[_m - 1] _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) )
```

index, *not*, *memq* は TAO の組み込み関数

これまでの例で明らかなようにすべてのパラダイムのプログラム間でデータは完全に共有されており、この点に関しプログラマは何ら意識する必要はない。これもTAOの大きな特徴の一つであり後述するようにコンパイラの実装方式を決定する大きな要因である。

2.2 実装方式

Prolog処理系との最大の相違点はTAOではスタックが1本しかないことである。この理由は

- 通常の3本のスタックによるインプリメント方法では他パラダイムのプログラムとのデータ共有が難しい(次に述べる)。
- ELISは1本のハードウェアスタックを持つ。

である。このためglobalスタックに代わりセル領域を使う。セル領域はスタックのようにLIFO形式で使うことができないため、バックトラック時に領域を解放することはできなくなる。論理型プログラムが使うセル領域を他パラダイムのそれと分離すれば、セル領域をLIFOで使うことは可能であるが、データを共有することが不可能になるためこの方法は採っていない。またtrailスタックの代わりにuntrailする場所をリストで保持している。

以上の理由によりTAOでは非常に多くのセルを消費する。しかしTAOのGCが軽く、短時間で終了(16MByteで5秒以下)することに合わせ、5.2で述べる、組込み関数を積極的に用いるというプログラミングスタイルをとることにより実用上それほど問題は生じない。実際本コンパイラは論理型パラダイムを用いて記述されており主記憶が16MByteのELIS上で開発された。

3. コンパイラの設計

1節で述べたようにインタープリタの持つ良い性質を備えたコンパイラを目標として設計を行う。具体的には以下を本コンパイラが満たすべき条件とする。

1. 他パラダイムとの融合機構などがインタープリタと同じセマンティクスで提供される。
2. コンパイルされた述語とインタープリタ実行の述語が相互に呼びあえる。
3. コンパイルされたコードでもある程度のデバッグが可能。

4. assert,retract がコンパイルされたコードに対しても可能。

1 は TAO の基本機能である。3 は実行履歴などのデバッグに必要な情報を出来る限り保持するという意味である。Prolog 処理系で開発を行う場合、デバッグの終了した述語はコンパイルし末了のものはインタープリタで実行することが多い。現在の Prolog 処理系ではインタープリタによる実行とコンパイルされたコードの実行速度比が 10 倍以上であるためインタープリタだけでは遅すぎるからである。コンパイルされた述語にバグが潜んでいた場合はインタープリタに戻してもう一度バグを再現する必要がある。3 の機能があれば、すぐにデバッグ作業を開始できる。本コンパイラでは過度の最適化をせずスタックフレームにデバッグ情報を記録しておき、コンパイラされたコードでも実行履歴を調べられるようにする。4 の機能は従来の Prolog コンパイラでは実現されていないため、プログラマはコンパイルされている述語とそうでない述語を常に意識する必要がある。4 の実現により、よりインタープリタに近い環境を提供できる。

次にインプリメント方法を決定する際に検討すべきことについて述べる。本コンパイラは WAM [83Warren] 方式を基本としているが以下の点については異なるインプリメントをしなければならない。

1. (ハードウェア) スタック (32K 語) が 1 本。
2. スタックと主記憶のアクセス速度比が大きい。
3. 他パラダイムとスタックフレーム、データ構造などがあらゆる点で混在する。
4. リストが基本データ構造である。

1 は最も重要である。前述 (2.2 節) のような理由からコンパイラにおいてもスタックは 1 本とする。2 もアーキテクチャ上の制約である。このためできるだけスタックを活用するように設計する。3 および 4 は TAO の言語仕様によるものである。パラダイムに関係なく同じデータ構造を持ちかつ共有し、またパラダイム間の呼びだしが入れ子になっても変数のバインディングや catch-throw などが正しく機能しなければならない。

4. コンパイラの概要

4.1 コンパイラの構成

ELIS は Lisp 専用マシンでありマイクロプログラムが可能である。従ってコンパイラがマイクロプログラムを出力すれば高速な実行が可能であるが、ELIS のアーキテクチャおよび WCS のメモリサイズの点で無理である。そこで中間言語を生成する方式とする。中間言語の多くは WAM コードに 1 対 1 で対応する。図 1 に本コンパイラの構成を示す。

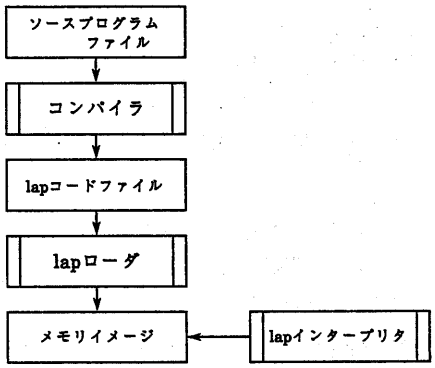

図1 コンパイラの構成

ソースコードはまず lap コードと呼ばれる中間言語にコンパイルされ、これが lap ローダによってメモリ上にロードされる。これを lap インタープリタが解釈実行する。コンパイラ、lap ローダは TAO で記述されており、それぞれ約 650、500 行であり、ほとんどが論理型パラダイムを用いて記述されている。lap インタープリタはマイクロプログラムであり約 1350 ステップである。

4.2 述語および節の内部表現

述語は適用可能オブジェクト (applobj: applicable object) の 1 つとしてベクタによって内部表現されている。ここでは述語の applobj を述語オブジェクトあるいは述語ヘッダと呼ぶ。同様に節のそれを節オブジェクトあるいは節ヘッダとよぶ。コンパイルされた一つの述語はこれらによって図 2 のように表現される。

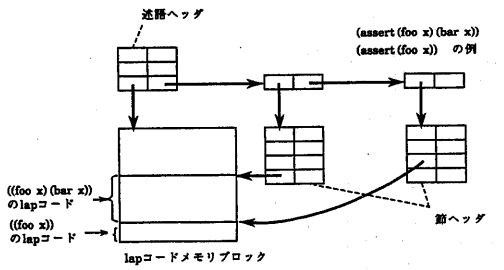

図2 コンパイルされた述語定義の内部表現

この述語に assert が為されると述語ヘッダにフラグが立ち、節ヘッダリストの適当な位置にインタープリタ実行の節ヘッダが挿入される。述語の実行過程は図3のようになる。

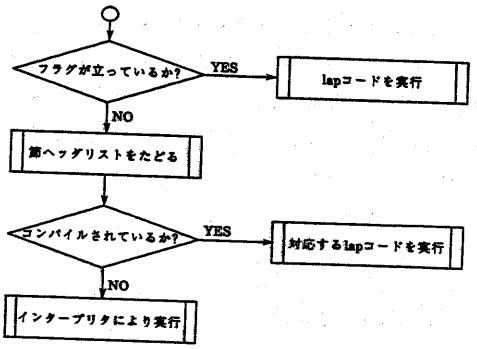

図3 実行までの制御の流れ

コンパイルされた節ヘッダには自分の lap コードの場所が記録してあるためこれが可能となる。なお assert、retract をしないことが明らかである場合は、述語ヘッダのフラグを調べることなく直接 lap コードメモリブロックを実行するようにコンパイルすることもできる。

4.3 スタックフレーム

図4にスタックフレームの一例を示す。

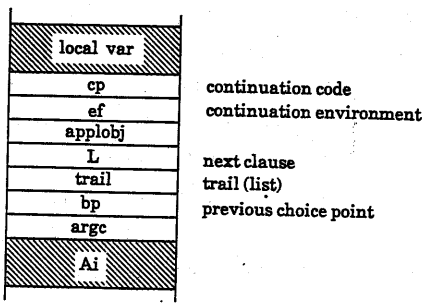

図4 スタックフレーム

これは WAM で Choice Point と Environment の両方が存在する場合に相当するフレームである。WAM ではある述語が Choice Point と Enviroment を持つときは両者が **continuation code** と **continuation environment** のフィールドを持つ。ELIS のスタックはあまり大きくないためスタックフレームの大きさをできる限り小さくする必要がある。そこで Choice Point と Environment があるときにはその2つのフィールドを重ね合わせ、共有する。本コンパイラではさらに applobj (後述) も両フレームに持つ必要があるため、結局3語の縮小が可能となる。これは WAM でも可能であり、マイクロプログラムによってインプリメントしてある場合、実行時間のオーバーヘッドはわずかである。

4.4 lap コード

まず lap コードの具体例を例6に示す。

例 6: 例1の &append の lap コード

```
(switch-on-type label4 label6 label1)
(label label1)
    (try-me-else label2)
(label label4)
    (allocate applobj3 1)
    get-nil
    (get-variable 0)
    (get-value 0)
    proceed
(label label2)
    (trust-me-else end)
(label label6)
    (allocate applobj5 4)
    get-list
    (unify-variable 0)
    (unify-cdr-variable 1)
    (get-variable 2)
    get-list
    (unify-value 0)
    (unify-cdr-variable 3)
    (argc 3)
    (put-value 1)
    (put-value 2)
    (put-value 3)
    (exec* &append)
```

多くが WAM コードに対応していることは明らかである。以下では本コンパイラ独特のコードについてのみ説明する。

・allocate 命令

WAM の allocate 命令と異なるのは 4.3 で示したスタックフレーム中の applobj と呼ばれるフィールドを生成することである。このフィールドはそのフレームを作った述語オブジェクトを指し、これを解析することでこのフレームを作った述語の名前、変数のバインディングの状態などを調べることができる。また applobj と ef のペアにより他パラダイムと同じ形のフレームを形成することができるため、これによりフレームの互換性が保たれ変数のバインディング、catch-throw などの動作がすべて保証される。

・リストのユニフィケーション

TAO の論理型パラダイムではリストが基本データであり、このユニフィケーションが全体の速度を決定するとも言えるためリスト専用に WAM を最適化する。

WAMではリストを構造体の1種と捕えているため1つの要素につき構造体名のユニファイ(unify_list)命令と構造体の要素のユニファイ(unify_constant)命令が必要となる。一方本コンパイラではリストをデータの並びと捕え「次の要素をユニファイする」という一連の動作をunify-constq命令が行う(例7)。

例7: リスト(a b c)をユニファイするlapコード

```
get_list A1              get-list
unify_constant a         (unify-constq a)
unify_list               (unify-constq b)
unify_constant b         (unify-constq c)
unify_list               unify-cdr-nil
unify_constant c
unify_nil
```

WAMコード　　lapコード

これにより次のような効果が期待できる。

- リストが長い場合lapコード量はWAMのほぼ半分まで減少する。
- コードのフェッチ回数も半分に減少するため実行速度が向上する。

Prologでは幾つかの要素が並んだリストのパターンよりもcdr部が変数であるようなパターンとのユニフィケーションの方が多いが、TAOで構造体に相当するデータを表現するためにはリストを用いるので前者のパターンが多くなる。従ってこの最適化はTAOでは特に有効であるがWAMでも適用可能である。

・可変個引数の述語の処理

Prologでは述語の名前とその引数の個数の組み合わせで述語を特定(例えばappend/3など)しているが例2のようなユニフィケーションが可能なためTAOでは述語名でしか述語を特定できない。コンパイル時に引数の個数が決まっている述語かどうか判断し可変個の場合は、引数の個数に対応する節に分岐する命令switch-on-argcを生成する。また例2のような場合_yに対してはget-cdr-variable命令が生成される。これは呼び出し側の残り(第2引数以降)の引数をリストにし_yに代入する。

4.5 その他の高速化・効率化の手法

・バイトコードとレジスタによる命令キャッシュ

lapコードは固定長ではなく1バイト単位の可変長バイトコードである。これはlapコードのオペランドが可変であることによる。特に多くのlapコードはオペランドが無い。ところでELISでは64bit長の汎用メモリレジスタ内の各バイトを間接レジスタを用いてバイト単位およびワード(16bit)単位でアクセスすることが可能である。そこで汎用メモリレジスタを8バイトのlapコードキャッシュとして用いる。これによりメモリアクセスの回数を減らすことができ、最大8命令までが命令フェッチ無しに実行できる。例えば例7の(a b c)のユニフィケーションの例では、一連のlapコードは以下のようになる。

```
get-list        => #30
unify-constq a => #3 #3
unify-constq b => #3 #4
unify-constq c => #3 #5
unify-cdr-nil  => #15
```

合計8バイトであるから、この命令すべてがキャッシュに入り切ることもありうる(この一連の命令が必ずセル境界に配置されるわけではない)。

・スタック上の仮想引数レジスタ

WAMでは引数はレジスタによって渡される。ELISは32本の汎用レジスタを持つがシステムが使用するレジスタなどがあるため、すべてを使うことはできない。もし引数レジスタを実レジスタで実現するとしても高々4本程度しか使えない。この場合実レジスタに入りきらない引数の例外処理が必要になり、また各実レジスタごとに専用処理ルーチンが必要になりマイクロプログラムが非常に多くなる。一方ELISはスタックをオートインクリメントしながら参照する場合にはレジスタと同じ速度でスタックメモリのアクセスが可能である。ユニフィケーションは一般には第1引数から順に行われるため、スタック上に仮想的な引数レジスタを置けば引数は順にアクセスされ、オートインクリメントの機構を利用できる。また引数レジスタの個数の制限は実用上無くなる。さらにユニフィケーションを順に第1引数から行うことを前提にしたため、何番目の引数のユニフィケーションかを指定するオペランドが不要となりバイトコードを小さくできる。

Prologコンパイラの中にはユニフィケーションの順番を変更し、ヘッド部の値の決まっている引数から順にユニフィケーションをすることで失敗をより早く検出するという最適化を行っているものがある。本コンパイラではこの最適化は適用できないが、仮想引数レジスタ方式を採らない場合のデメリットを考えると

やむをえないと思われる。

5. 評価

5.1 実行速度およびその解析

本来は本コンパイラで用いられた手法の効果を一つずつ測定するべきであるが実際上は困難であるのでまず他の処理系と比較を行う。長さ 2000 のリストの結合 append-2000、長さ 30 のリストの反転 reverse-30、8 人の女王問題の全解探索をする 8-queen のプログラムについて、TAO インタープリタ (TAO(I)) とコンパイラ (TAO(C))、SUN3/260 上の Quintus-Prolog インタープリタ (Q(I)) とコンパイラ (Q(C)) での実行速度を次に示す。但し 8-queen は例 5 の組み込み述語を論理型プログラムで書いてある。

	TAO(I)	TAO(C)	Q(I)	Q(C)	TAO/Q(C)
append-2000	203	60	846	21	2.9
reverse-30	48.8	14.5	250	6.1	2.4
8-queen	5444	2053	31117	1450	1.4

[単位は msec]

LIPS に換算すると append では 33.3KLIPS であり、reverse では 34.2KLIPS である。(但し現在 lap コードレベルの最適化を行っていない。ハンドオプティマイズしたところ 40.5KLIPS であった。) TAO(C) と Q(C) の比が 8-queen ではリスト操作のそれに比べ著しく小さい。これは 8-queen で用いられている組み込みの述語が TAO ではマイクロプログラムによって記述してあるためと思われ、純粋な処理系としての速度比はリスト操作の比に近く 2.5 倍程度であろう。一方、ELIS と SUN-3 のハードウェア性能を単純に CPU クロックで比較すると 16.6MHz と 25MHz であり約 1.5 倍である。この原因を調べるため append プログラムの一部について各 lap コードの実行マイクロステップ数を調べる。

ユニフィケーションのステップ数が大きいが中でも 3 つの write モードのユニフィケーションだけで 36% を占めている。write モードは主にセルの cons と trail の登録 (これも実は cons) を行っており、global スタックと trail スタックの代わりにセル領域を使っていることが速度低下の原因であると思われる。

lapコード	ステップ数
(switch-on-type ...)	13
(allocate applobj5 4)	16
get-list	10
(unify-variable 0)	6
(unify-cdr-variable 1)	7
(get-variable 2)	5
get-list	13 + 15 [write-mode]
(unify-value 0)	9 [write-mode]
(unify-cdr-variable 3)	17 [write-mode]
(argc 3)	6
(put-value 1)	6
(put-value 2)	6
(put-value 3)	6
(exec* &append)	14
合 計	149

次に applobj(4.4 節) のためのオーバーヘッドであるが、このフィールドを作るのは allocate 命令でありその 16 ステップのうち 6 ステップを占める。これは総ステップ数の 4% である。(多くの述語はこの例よりも長いのでこの値はさらに小さくなる)。これは applobj による多くの利点を考えると十分小さいといえる。

5.2 TAO の論理型パラダイムのプログラミングスタイル

TAO の論理型パラダイムでプログラムを行う場合、TAO の言語仕様および TAO 独特のインプリメントのため Prolog とは異なるプログラミングスタイルをとるべきであることがこれまでの議論より解る。つまり、TAO では

- スタックが小さく深い再帰ができない。
- 豊富でかつ非常に高速な組み込み述語を持つ。

であるから、すべてを論理型パラダイムで記述するより他パラダイムの方が適している時には積極的にそれらを用いることが望ましい。特に決定的に動作するリスト操作などは関数型パラダイムを用いることでインタープリタ実行でも Prolog コンパイラに近い実行速度を出すことも可能である。特に TAO のマイクロプログラムで記述された組み込み述語を多用するほど、例 5 で示したようにインタープリタとコンパイラの差が縮まってくる。また、スタックがあまり大きくないためトップレベルなどの繰り返しの部分はループによって記述しその中で論理型パラダイムを用いるのが望ましい。論理型のフォームを L 評価すると評価後にスタッ

クを捨て去るのでスタックを消費しない。

6. まとめ

TAO の論理型プログラムコンパイラについて述べた。本コンパイラはインタープリタに近い環境を提供することを目標とし、過度の最適化を避け述語の呼び出しの履歴や変数の状態などデバッグに有用な情報を保存する。またこれを実現するために必要なオーバーヘッドは4%程度に過ぎない。また従来の WAM 方式のコンパイラにも適用可能な最適化を検討しそれらを実装した。

今後は lap コードレベルの最適化を行い高速化を計るとともに、スタックフレームの情報を有効に利用したデバッガを作成する。

参考文献

[86Takeuchi] I.Takeuchi, H.Okuno, N.Ohsato: A List Processing Language TAO with Multiple Programming Paradigms, New Generation Computing No.4, 1986.

[83Warren] D.Warren: An Abstract Prolog Instruction Set, SRI Technical Note, October 1983.

付録

インタープリタのスタックフレームの例