解　説

論理型言語指向の推論マシン

2. 論理型言語の逐次実行処理方式†

横　田　　実††

1. はじめに

FORTRAN やC言語に代表されるこれまでのプログラミング言語は，「データに対する操作」をその実行順序に従って書き下すものである（手続き型言語）．これに対して論理型言語ではデータ間の「論理関係」を記述する．プログラミングパラダイムとしてみれば「計算機の実行すべき手順」の記述から「問題解決のための知識」の記述への革新と言える．このような発想の大きな転換を要求するため，論理型言語というのはこれまでのノイマン型プログラミングに慣れ親しんだ人には分かりにくいようである．確かに 80 年代初頭に Prolog が注目され始めたころは閉世界仮説と否定の問題など論理型言語のセオリ的側面の（難解な）議論が活発であった．しかし，プログラミングツールとして使用してみれば従来の言語とそれほど変わらない．実際，推論機構は手続き呼び出しと似た方法でコンパイルされている．むしろ，論理型言語のもつ特性を正しく踏まえた上で「道具」の一つとして活用することが重要である．本論文では論理型言語の実行に特徴的なメカニズムについて概説し，最適化コンパイル方式として確立している WAM 方式について述べる．

2. Prolog における推論処理

図-1 は家族の関係を Prolog でプログラムした例である．ピリオドで区切られた各行はホーン節，あるいは単に「節 (Clause)」と呼ばれる．「:-」の左辺の述語をヘッドゴール，右辺に並んだ述語をボディゴールと呼ぶこともある．

手続き型言語に慣れた人がこのようなプログラムを眺めて，まず困惑することは「プログラムはどこから始まってどういう順序で実行するのか？」ということであろう．Prolog プログラムとは事実や推論ルールが「宣言として」並んでいるだけで goto 文もなければプログラムの入口らしきものも見あたらない．

```
mother(mary, betty).
father(john, jim).
father(X, Z) :- hasband(X, Y), mother(Y, Z).
hasband(john, mary).
```

図-1 Prolog プログラム例

図-1 で示される事実関係から，われわれは john が betty の父親でもあることを当然と思うが，そのような判断が可能なのは，次のような推論を行っているからである．

- mary は betty の母親である．

(図-1 で陽に宣言されている事実)

- john は mary の夫である．

(図-1 で陽に宣言されている事実)

- 母親の夫である男を父親と呼ぶ．

(「常識」と呼ばれているルール)

図-1 のプログラムもこのような常識に相当する節，

father (X, Z) :- hasband (X, Y), mother (Y, Z).
(X が Y の夫で，Y が Z の母親ならば X は Z の父親である)

を含んでいることにより，同様の推論を実行できる．大文字で始まる名前は変数であり，この節を一般化されたルールにする働きをしている．

Prolog の利点は，プログラム上で明示的に宣言されている個別の知識が限られたものであっても，推論機能によってより多くの結論を導き出せる点にある．図-1 で，さらに兄弟を表す一般ルール「brother (X, Y) :- father (Z, X), father (Z, Y).」を追加すると jim と betty が兄弟（妹）であることが推論できる．これまでの「言われたことしかできないプログラム」に較べて少し利巧にな

† Implementation for Sequential Logic Programming Languages by Minoru YOKOTA (Computer System Research Laboratory C&C Systems Research Laboratories NEC Corporation).

†† 日本電気(株) C&C システム研究所コンピュータシステム研究部

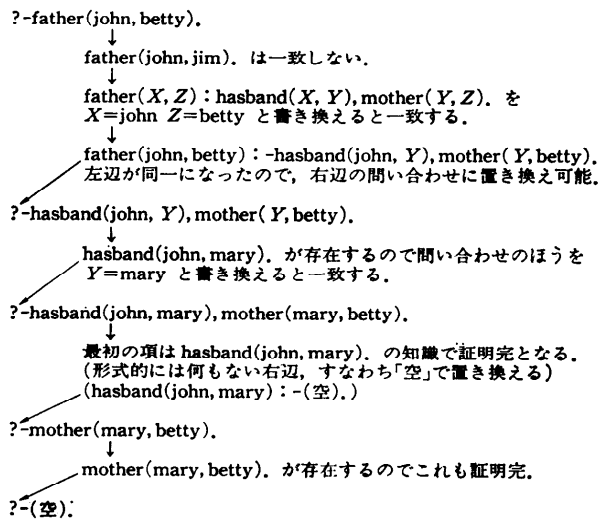
ったと言える．しかも知識や推論ルールは実行時に動的に追加／削除できるようプログラムすることが可能で，自己適応性を備えている．

3. 推論機構の実現

3.1 問い合わせの書換えによる推論実行

先の例における father (john, betty) が真か否かの証明は，プログラム中に「father (john, jim).」型の知識（事実：fact）で直接的に宣言されているか，もしくは「father (X, Z) :- hasband (X, Y), mother (Y, Z).」のようなルール（公理：axiom）を使って推論可能かを調べることになる．前者は一字一句が対応するものを捜す処理になるが，後者は :- の左辺に一致するものがあれば，その右辺が成立するかを証明する問題に置き換えられる．いずれの場合も，変数が含まれている場合には一致させるための変数の書換え操作が必要である．

このような同一化の操作をユニフィケーションと呼ぶが，プログラムで宣言されている知識を使って問い合わせを変形していく操作となる．



3.2 手続き呼び出しによる推論実行

上記のような「書換え」操作によって推論を実現する考え方はリダクションと呼ばれる．全体を一度に見渡せる人間にとっては直感的である．また，多数のプロセッサがおのおの異なるルールを用いて，問い合わせを並列に「書き換え」ていくような並列推論処理のモデルとして素直な考え方である[Onai 85]．しかし，このような節の書換えを忠実に実現しようとすると文字列操作になり，効率のよい実行は難しい．

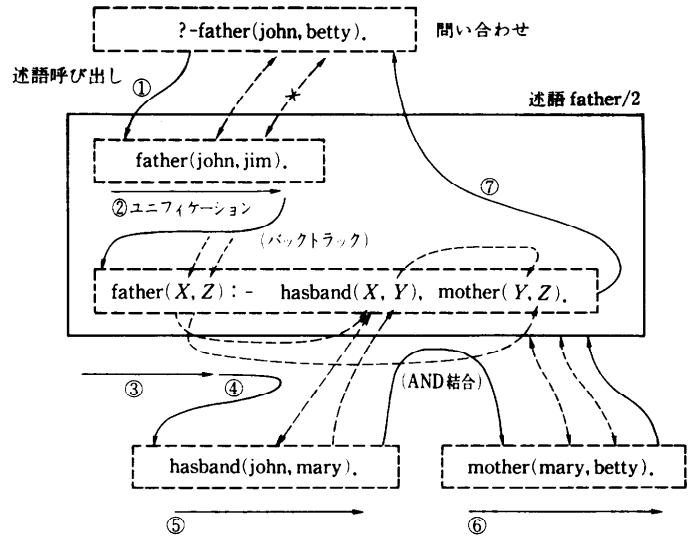


図-2　手続き呼び出しによる推論メカニズムの実現

実際に書き換えられるのは変数部分だけであるから，節のデータ構造をコード部と変数セルに分離できる．さらに，:- の左辺をユニフィケーションを実行するプログラム，右辺をほかの述語の呼び出し処理に変換すれば従来の手続きと同じようなものになる．その場合に，証明操作とは次のように考えることができる．

証明　　－〉 述語呼び出し処理
結果が真　－〉 呼び出し処理が正常に終了（サクセスリターン）
結果が偽　－〉 呼び出し処理が失敗（フェイル）

先の例を手続きとして実行する様子を図-2 を使って示す．

①まず，述語の呼び出しは同じ述語名 (father) を左辺にもつ節を探索することから開始される．このとき，引数の個数 (Arity) が異なる述語同士は一致しないため，実際には father/2 というように述語名／Arity で検索される．

②左辺が father/2 の節は二つ存在するが最初の節は引数データが呼び出し元ゴール引数と一致しないためユニフィケーションは失敗する.

③第一候補節とのユニファイに失敗したので,二番目の候補節とのユニフィケーションを試みる. これがバックトラックと呼ばれる後戻り操作である. 二番目の節のヘッド引数は変数となっており,初めての出現なのでまだ未定義状態にある. X=john, Z=betty を代入することにより呼び出し元と一致させることができる.

④ユニフィケーションは成功したので, :- を通過してボディゴールの呼び出し処理に移る.

⑤述語 hasband の呼び出しでは引数 Y がまだ未定義のままであるが, Y=mary とすることにより,ユニフィケーションは成功する (サクセスリターン).

⑥述語 mother の呼び出し時点ではすでに Y=mary, Z=betty が代入されているが,これとまったく一致する節が宣言されている (サクセスリターン).

⑦ボディゴールの呼び出しは全て成功したので (AND 結合),最終的に問い合わせが開始されたレベル (トップレベル) へサクセスリターンし,証明は完了する.

要約すると,Prologにおける推論メカニズムとは

1) ボディゴールの呼び出し

2) ユニフィケーション

の繰り返しによって行われる. 繰り返しがうまくいかなくなるのはユニフィケーションに失敗した場合で,そのときにはバックトラックによりほかの候補節を試みることになる.

4. Prolog の実行順序制御

4.1 候補節の検索順序

複数の候補節が存在する場合,その適用順序は本来証明には無関係であり,問い合わせを満たす解が複数あっても良い. しかしながら逐次的に処理をするためにはなんらかの順序を決めざるをえない. 逐次版 Prolog では宣言順 (ソースリストで上から) ユニフィケーションを試みる. 同様に, :- の右辺に並んだボディゴールの証明も論理的には順序に依存せず同時に真か偽かを判定すべきであるが,実際には実行順序が結果を左右するため左から順に行うと定める.

結果として,逐次処理方式ではプログラムが表現している「宣言的な意味」(Denotational Semantics) と,プログラムの「実行によって生ずる意味」(Procedual Semantics) とが異なる. また,この実行順序は複数の候補節を OR 結合,ボディゴールを AND 結合とみた AND-OR 型の証明木を左優先・深さ優先 (Left-Most-Depth-First) で探索する戦略に相当する (図-3).

Prolog の逐次処理システムではこのような「順序の存在」のもとに一つの解を求めることしかできないため,途中で証明が失敗した場合に後戻りして別の候補節を選び直す作業が必要となる. これがバックトラックと呼ばれる処理である. 最適解を求めるなど,ほかの可能性も探索する必要がある場合は意図的にバックトラックが利用される.

4.2 ユニフィケーション

1) 変数の書換え処理

ユニフィケーションは引数同士が同じデータか

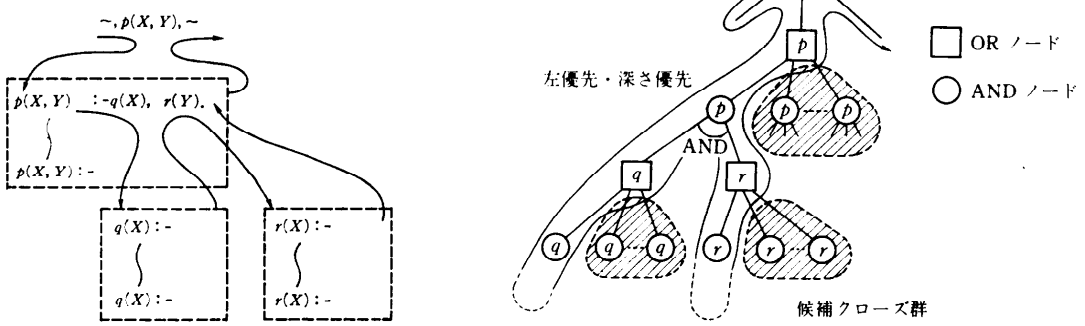


図-3 探索空間の AND-OR 木表現

否かという比較処理だけでなく，一致させるために変数への代入が行われる点に特徴がある．未定義変数は呼び出し元／呼び出された側のどちらに現れても良いため，変数への代入は「呼び出し元 -〉呼ばれた側」あるいは「呼ばれた側 -〉呼び出し元」へという両方向が発生する．特に呼び出し元引数の値が未定義のままでも述語呼び出しが可能である点は従来言語にはなかった新しい機構である．

この特徴は，実行とともにデータ構造が決まっていくような処理（たとえば言語処理における構文木の生成など）に非常に有効である．これは関数型言語で近年活発に研究されている遅延評価[Henderson 76],[Friedman 76]と同様の考え方である．Prologではこのような未定義変数を含むデータ構造を不完全構造（Incomplete Structure）と呼んでいる．

2)　デレファレンス

対応する引数が双方とも未定義変数の場合には両変数が論理的には同一であるとされる．具体的には双方の変数セルが参照ポインタでつながれて，一方へのユニフィケーション（代入）結果がもう一方の変数にも反映する仕掛になっている(図-4)．このような変数同士のユニファイにより，変数セルには値が直接格納されている場合と，ほかの変数セルへのアドレスが格納されている場合が生じる．また，このような変数がユニファイを繰り返すことにより参照ポインタは2段以上になり得る．したがって，Prologでの変数アクセスには値を取り出すために参照ポインタをたぐる操作が必要で，「デレファレンス（dereference）」と呼ばれる．

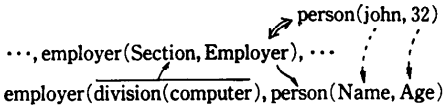
3)　変数への単一代入則

変数の書き換えは，問い合わせが真か偽かという最終ゴールに向かっての証明を進める上での「必要条件」として行われるため，後で勝手に書き換えてはいけない．ユニフィケーションにおいて「一致した」とみなされた引数の関係は未来永劫続いてくれなければ困る．このように変数に一度しか値が代入されない特徴は関数的プログラミング[Backus 78]の利点の一つと唱われた「単一代入則」と同じものである．

しかし，この特徴が逆にPrologを堅苦しい言語にしている点もいなめない．たとえばPrologの変数は，従来言語では普通に行われる途中結果を保存するための作業領域として書き換えて使うことが許されない．また，構造体データは制御テーブルのような書換えを前提とした集合データの保存のために使用したいことが多く，単一代入則を満たさない使い方が望まれる．第五世代プロジェクトで開発された推論マシンPSI上の論理型言語KL 0では，これを解決するために単一代入則を満たす構造体（スタックベクタ）と満たさない構造体（ヒープベクタ）の2種類を設けた[Chikayama 83b]．ヒープベクタの場合は物理的に同一のベクタを指している場合に限りユニファイ成功とし，たとえ将来，要素データが書き換えられたとしても同一である事実が破綻しないよう考慮されている．

4)　構造体データの生成／分解

ユニフィケーションでは要素データから構造体を生成したり，その逆に，構造体データを要素に分解したりできることも特徴である．



この例で，呼び出し元の変数Sectionが未定義なら呼ばれた側の構造体division (computer) が代入される．ただし，このような「即値で構成されている構造体データ」は定数としてあらかじめコード中に埋め込まれており，変数にはそこへのポインタが代入されるに過ぎない．一方，変数Employerが未定義の場合は，代入される構造体person (Name, Age) が変数を含んでいるため，変数セルを内蔵する構造体データを動的に生成しなければならない．後述するがこのような構造体データの割付けには専用のスタックが用意される．

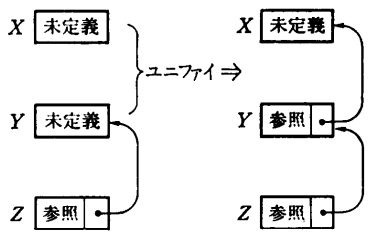


図-4　未定義変数同士のユニフィケーション

一方，変数 Employer にたとえば person (john, 32) という構造体データがすでに代入されている場合は，ユニフィケーションにより呼ばれた側の変数 Name, Age におのおの要素データ John, 32 が取り出される．

このように構造体データに対する操作が視覚的に表現できるのも Prolog の特徴であり，たとえば Lisp の基本関数である CAR・CDR は Prolog において次のようなリストデータとのユニフィケーションとして視覚化される．

```
…, p([1 2 3 …]), …        /* リスト(1 2 3 …)
  p([X | Y]) :- …          /* X=1, Y=[2 3 …]
```

4.3 バックトラック

バックトラックを実現するには複数の候補節が存在する選択点を通過するたびに，後戻りできるよう実行環境や次の候補節のアドレスを記憶しておかなければならない．バックトラックの制御がスタックを用いて行われることもあって，Last-In-First-Out，すなわち最後に通過した選択点が最初に再実行される．

図-5 の例では，述語 $p/2$ の呼び出し時点①でいくつか候補節が存在するが，まず最初の節（左優先・深さ優先の探索）を選びユニフィケーションを実行する．ユニファイ成功で右辺の処理に移るが，$q/1$ の呼び出し②，$r/1$ の呼び出し③におのおの 10 通りの候補節がある．ここでも最初のクローズ ($q(0).$ と $r(0).$) から選択される．その後，バックトラックが発生すると，最後に通過した選択点③に戻って，次の候補節「$r(1).$」が再試行される．バックトラックが何度も起こり，述語 r に試みる節がなくなると，次に古い選択点である②へ戻り述語 q の候補節が選ばれる．その後に続く述語 r の呼び出しはふり出しに戻って $r(0)$ から選択される．

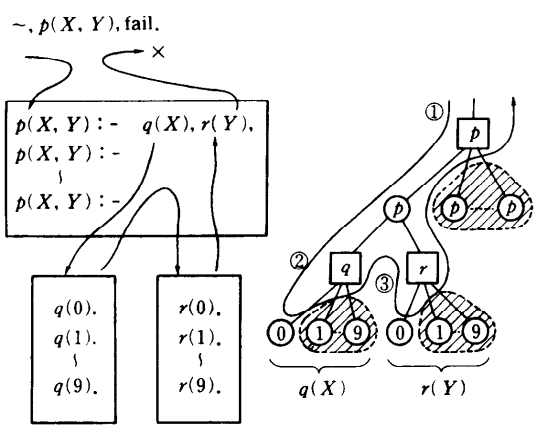


図-5 バックトラックの順序

このようなバックトラックにより変数 X/Y には順次，次のような値がユニファイされることになる．

$0/0, 0/1, \cdots, 0/9, 1/0, \cdots, 1/9, 9/0, \cdots, 9/9 \cdots,$

スタックに格納される情報は次に実行すべき候補節のアドレス，ユニフィケーションをやり直すための「呼び出された時点での引数情報」，関連するスタックの先頭アドレスなどである．バックトラック発生時にはスタックの一番上に記憶された（すなわち最後に通過した）選択点情報までポップアップ（クリア）し，呼び出し時の実行環境を復元してから，次の候補節とのユニフィケーションを再実行すれば良い．

なお，ユニフィケーションではスタックの底のほうに割りつけられた変数セルへの代入も発生することがある．スタックのポップアップだけではこのような深い位置に置かれた変数セルを未定義状態に戻すことができない．したがって，代入が行われた変数セルアドレスを記憶しておき，バックトラック発生時に順に未定義状態に戻す操作が行われる．変数セルアドレスを記憶するスタックをトレイルスタックと呼ぶ．

4.4 カットによるバックトラックの制御

Prolog の実行順序は AND-OR 木の上での探索順序に対応しており，OR 結合された候補節はバックトラック機構を用いて全て実行できる．しかしながらいくつかの選択が排他的に行われる場合には，残りの候補節を実行する必要がない．バックトラックが無駄であるばかりかプログラマの予期しない探索経路へ入り込むことになる．カット（！と記す）は残っているよけいな候補節を文字どおり「カット」し，不要なバックトラックを抑制する働きをする．

プログラムの実行途中のある時点を考えると，各 OR ノードを構成している候補節は，すでに実行され失敗した節，現在選択されている節，まだ候補として残っている節の 3 種類に分類できる (**図-6**)．カットはこのような残っている候補節を切り捨てる．たとえば図-6 における $p(X, Y) :-$

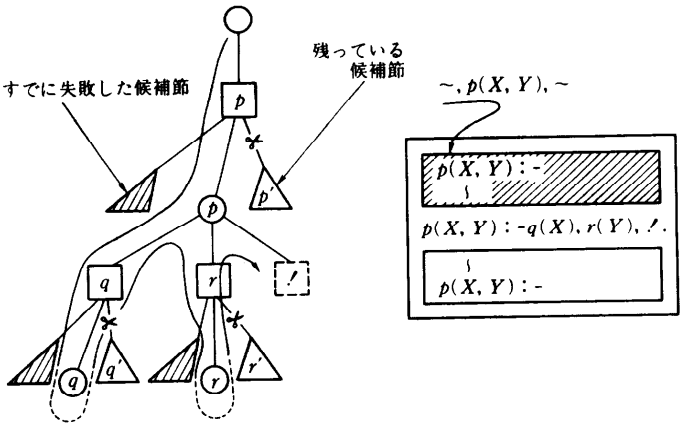


図-6　カットによるバックトラックの抑止

$q(X)$, $r(Y)$, !. のカットの作用は述語 q と r, および自分自身 (p) の残っている候補節を切り捨てることである．言語処理系によっては，q, r の選択肢のみ切り捨てる「弱いカット」や，もっと上位の呼び出しレベルまで切り捨てる「相対カット」[Chikayama 83a, 83b] なども提案されており，バックトラックをより柔軟に制御する方法として活用されている．

カットの具体的な動作はスタック中に保存されている選択点情報を消去することであるが，カットを含むボディゴールの実行途中ではスタック領域がまだ使用中であり，ただちにスタックを畳むというわけにはいかない．したがってカット処理では後戻りに使う選択点情報を変更するだけにとどめ，スタックの畳みこみはボディゴールを全て実行した後，呼び出し元へのリターンにともなうスタックのポップアップと一緒に行われる．

4.5 組込述語

以上述べたように Prolog の実行はユニフィケーションというパターンマッチング処理を繰り返すことにつきるが，それだけでは算術演算や入出力のような副次効果をともなうシステム機能を実現できない（原理的には可能であっても実用的な実行速度では実現できない）．さらに節の生成／登録やカットのようなメタ機能を実現する手段も必要である．これらの機能を実現するために言語処理系は多数の組込の述語を用意している．

［例］ ～, read(X), Y is $X+1$, ～（ファイルからデータを読んで，1 を加える）

［例］ ～, assert($p(X) : - q(X).$), ～（節の知識ベースへの登録）

組込述語は Prolog の枠組みではできないことを述語呼び出しの顔をして実行するものとも言える．したがって，その中にはバックトラックによっても元に戻らない機能を実行するものも多い．たとえば入出力動作は元に戻すことができない．

組込述語はいろいろな強力な機能を述語呼び出しの形で素直に含めることが可能であり，Prolog を実用的なプログラミング言語とするために大いに役だっている．

5. Prolog の処理系

5.1 インタプリタとコンパイラ

インタプリタ方式は Prolog プログラムをデータとして読み込み，言語処理系が解釈・実行していくもので，例外処理やメモリ管理など，プログラムの実行にともなって必要となる言語特有の処理を折り込みながら実行していくことができる．したがって処理系の開発も比較的容易である．Prolog のように対話的な実行環境を要求する言語ではインタプリタは必須であり，言語処理システムのトップレベルでインタプリタが動いているのが普通である．

一方，コンパイル方式ではコンパイラによる最適化によりソースプログラムが無駄のない機械語命令列に変換でき，直接ハードウェアで実行されるため実行速度はインタプリタの 10～20 倍高速化できる．しかし，プログラムの動的変更に対応しにくく，実行時に追加された節に対してはインタプリタを呼び出すのが普通である．また，例外処理やメモリ管理などをランタイムルーチンとして用意しておき，ユーザプログラムと有機的にリンクさせる工夫が必要となる．具体的にはコード中に直接ランタイムルーチンの呼び出しを埋め込むか，ハード的なトラップを起こして間接的にランタイムシステムを呼び出すことになる．

実際の処理系としては両者の中間となるエミュレータ方式も含め，図-7 に示すような処理方式が提案されている．推論専用マシンではインタプリタやエミュレータがファームウェアで実現されていることが多い．実行性能，処理系のつくりや

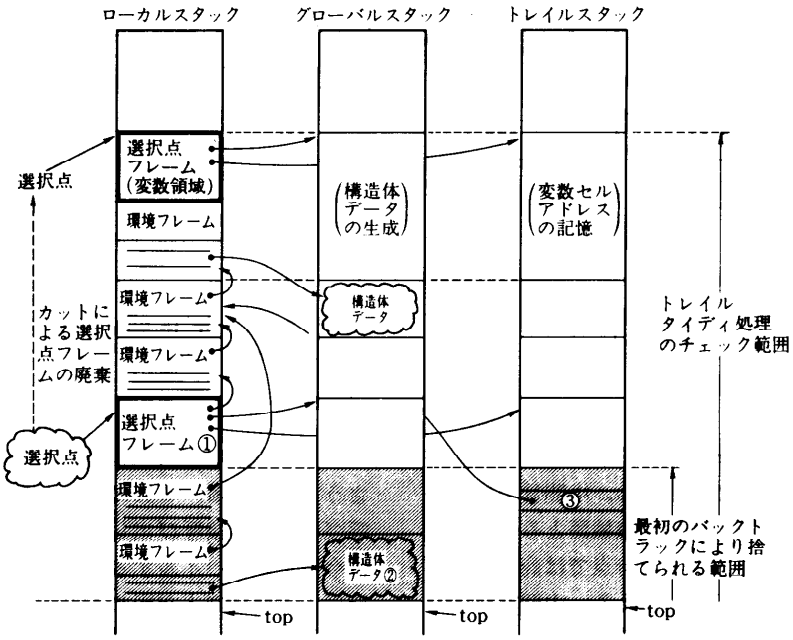


図-9　実行順序制御用スタック

位置，次に実行すべき候補節アドレスなど，バックトラックが発生した際に環境を復元するために必要な情報が格納されている．

これらスタックに格納された情報は呼び出しからのリターン時にポップアップされるのが普通であるが，呼び出された述語に候補節が残っている場合はバックトラックに備えて選択点フレームが残される（図-9 のフレーム①）．したがって，このようなフレームの存在によりローカルスタックはそれ以上縮まらない．

バックトラック発生時にはスタックの最も上にある（すなわち最後に通過した）選択点フレームを使って環境の復元／再試行が行われる．それよりも上に積まれている環境フレームは最も最近に選択された候補節の実行によって生じたものであり，全て無効化される．

2)　グローバルスタック

3.2.4)で述べたように，変数を含む構造体データと未定義変数とのユニファイは新しい構造体データを生成する．このような動的に生成される構造体データの割付け領域としてグローバルスタックが使用される．

呼び出された節の変数セルは環境フレームに割り付けられ，リターン時には環境フレームのポップアップとともに消滅する．しかし，ひとたび生成された構造体データはユニフィケーションによってどこかの変数セルから参照されている可能性があり，述語呼び出しからのリターン後も消去することはできない．特定の述語呼び出しとは関連しないという意味で「グローバル」である．この特性からスタックではなく，コードと同じようにヒープ領域に割り付けることも考えられるが，スタックとすることの利点は生成された時刻がスタック上の物理的な位置に対応していることである．これによりバックトラックの際にはある時間領域（たとえば最後の選択点を通過してから現在まで）に生成された構造体データをスタックを畳むことで捨ててしまえる(図-9 構造体データ②)．Prolog プログラマもこの辺を心得ており，意図的にバックトラックを起こしグローバルスタックを縮めることによりメモリ消費の爆発を防ぐことが行われている．次の例では mainjob の実行で伸びたスタックを強制的に畳んでから，次のサイクルが開始される．

［例］ toplevel-loop : - repeat, mainjob, !, fail.

3)　トレイルスタック

ユニフィケーションによって生じた変数への代入を無効化するのに必要な変数セルアドレスを格

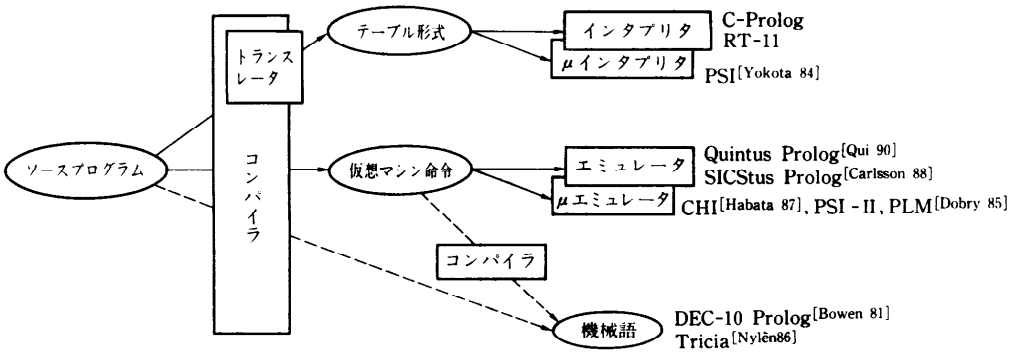


図-7 処理系の構築法

すさ，ポータビリティを考えると，コンパイラによる最適化を効かしていったん，中間言語に展開し，この中間言語を仮想的な機械語とみなしてエミュレーションする方式が現時点では商業的に最もバランスの良い処理系と言える．中間言語としては後述する WAM 命令が標準である．

5.2 クローズの内部データ形式

節の追加／削除はプログラム実行中にも可能であり，述語の呼び出しは静的には決められない．したがって，呼び出そうとする述語名を文字列ストリングの形でもっておき，実行時に名前表を索くのが基本的な処理手順となる．現実には検索の高速化のために名前のアトム番号化（文字列から登録番号への変換）やハッシング手法が活用される．さらに，実行時に呼び出し元の引数データタイプで候補節を絞り込む高速化手法（クローズインデキシング）もある．いずれにしても，ひとつの節に対応するオブジェクトコードは追加・削除が容易なようにポインタで連結される(図-8)．

ただし，実際には動的に変更される述語の数は全体のごく一部であることが多く，最近の処理系では，そのような追加・削除可能な述語 (dynamic predicate) をユーザに陽に宣言させ，分離してしまう方向にある．これにより，残りの多くの述語呼び出しを直接呼び出し（相手のコードへの分岐命令）にコンパイルすることが可能となり，実行速度の大幅な向上が図れる．

5.3 実行に必要なスタック構造

これまで述べてきた Prolog の実行メカニズムを実現するには，変数や制御情報の動的な生成・管理が必要となる．時系列での管理，メモリ領域の管理という観点からスタック構造が適しており，最終的には以下に述べる3本のスタックの働きに集約される（図-9）．

1) ローカルスタック

ローカルスタックは実行順序の制御に使用され，述語呼び出し／リターンのための情報（環境フレーム）と，バックトラックのための情報（選択点フレーム）との2種類が格納される．

環境フレームには述語呼び出しからの戻りアドレスや変数セル領域など，従来型言語での手続きの call/return に必要な情報と同じものが格納される．一方，選択点フレームは Prolog 特有のもので，述語呼び出し時点の引数情報やスタック先頭

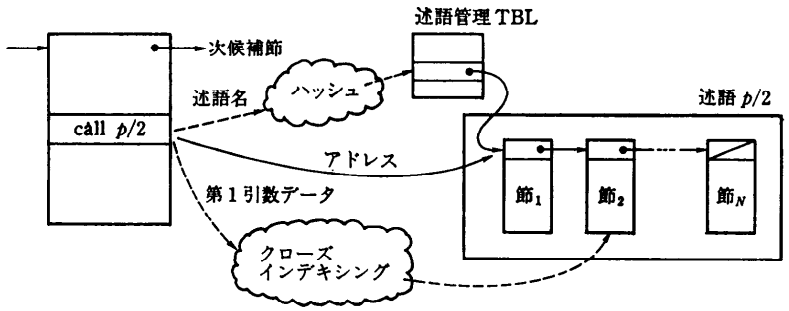


図-8 クローズの内部データ形式

納するのに使用される．バックトラック発生時にはスタックをポップアップしながら該当変数セルを未定義に戻す（図-9の③）．

バックトラックの際にローカル／グローバルスタックは戻るべき選択点に対応する位置まで畳まれるため，その範囲に含まれる変数セルは自動的に消滅する．したがってこのような変数セルアドレスは記憶する必要がない．トレイルスタックには後述のタイディ処理など，コストの高い処理がともなうため，格納するアドレス数を極力減らすために，格納すべきか否かの事前チェックが行われる（トレイルチェック）．

カットはバックトラックしないように候補節を捨ててしまう操作であるが，実際には選択点フレームとそれに関連する環境フレームを畳む作業になる．問題は，この消滅する領域に含まれている変数セルのアドレスがトレイルスタックに取り残されてしまうことである（図-9の③）．そのままでは，どこかでバックトラックが発生した際に未定義に戻すための書き込みが行われてしまう．これを防ぐにはカット時にトレイルスタックをチェックして不要になった変数セルアドレスをただちに除去しておくことが必要であり，これをトレイルタイディ処理と呼ぶ．

6. 仮想機械語命令による実行

6.1 DEC-10 Prolog

DEC-10 Prolog[Warren 77a, 77b] は D. H. D. Warren らにより 70 年代の始めに開発された最初のコンパイラ版 Prolog 処理系であり，80年代のWAM方式の基本的な考え方はここでほぼ確立されている．DEC-10 Prolog ではボディゴールを call 命令を用いたコーリングシーケンスに，呼び出された側のヘッドゴールはユニフィケーションのための機械語命令列に展開することにより，推論機構を初めてコンパイルされた「手続き呼び出し処理」に置き換えることに成功した．ソースプログラムはコンパイラにより仮想的な Prolog 命令列に展開され，最終的には DEC-10 のネイティブコードに変換される．

処理系の作りは AI 研究用マシンとして設計された DEC-10 のアーキテクチャの特徴をフルに活用したものとなっている．たとえば，変数の値を取り出すデレファレンス処理には「インデックス修飾＋メモリ多重間接修飾」によるポインタの連続たどり機構が，構造体データの表現（structure sharing 法）には1語に二つのアドレス情報を格納できるというリスト処理向きの機構がそれぞれ活かされている．また，コンパイラによる高速化の手法としてクローズインデキシングやクローズの最後の述語呼び出しを最適化するテイルリカージョンオプティマイズ，変数の性質を分類してユニフィケーション命令を細分化する最適化手法などがすでに組み込まれている．

述語呼び出しに必要な処理の流れはいくつかに区分され，Prolog 命令として以下のようにまとめられた．

- ①enter 命令　処理の開始点として環境・選択点フレームの生成．
- ②try 命令　節へのジャンプ．
- ③init 命令　変数セル領域の生成と初期化．
- ④unify 命令　引数ごとのユニフィケーション処理．（引数のデータタイプに対応した uatom, uvar, uref などの命令群）
- ⑤neck 命令　スタック環境の「呼び出し側」への切り替え
- ⑥call 命令　ボディ側での述語の呼び出し
- ⑦foot 命令　呼び出し元へのサクセスリターン（候補節が残っていなければスタックを畳む）

DEC-10 Prolog ではまだ環境フレーム／選択点フレームの分離が行われておらず，呼び出された時点で関連する制御情報が全て格納されるため処理が重い．一方，変数に関しては初めての出現では（未定義状態のため）値を取り出す必要がないので，ユニフィケーション用命令を分ける最適化が図られている（uvar 命令：初めての出現，uref 命令：2回目以降の出現）．また，call 命令や try 命令ではジャンプ後の次命令アドレスが特定レジスタに格納されるハードウェア機構をうまく利用して，引数アクセスのベースアドレスや次の候補節アドレスを設定する工夫がなされている．

6.2 Warren Abstract Machine (WAM)

DEC-10 Prolog の出現によりようやく論理型言語が実用的プログラミング言語として使えるようになったが，DEC-10 自身がポピュラではなかっ

たため，処理系の普及には難点があった（第五世代プロジェクトで最初にしたことはDEC-20の購入であった）．さらにDEC-10/20は1語36ビットにアドレスを二つ格納するアイデアを採用していたためユーザのメモリ空間は18ビット分256K語しかなく，メモリを大量に消費しがちなPrologではメモリ空間の狭さが致命的であった．

第五世代プロジェクトではこれを克服するためにPSIという専用マシンを開発したが，D. H. D. Warrenはむしろ，VAXのような汎用コンピュータの使用を前提に高速化とメモリ消費量の低減を図った新しいコンパイル技法を提案した[Warren 83]．現在WAM方式として定着している方式で，提案されたprolog用の仮想マシン命令セットアーキテクチャをWAMと呼ぶ．

1)　WAM方式の特徴

Prologの実行は，結局，ユニフィケーションでの引数データへのアクセスと実行順序制御のためのスタックへのアクセスに尽きる．したがって，WAM方式では高速化のためにa）引数データをレジスタ上に置くこと，b）スタックへの格納情報を削減することを最大の特徴としている．

a）　引数レジスタ方式

DEC-10 Prologではメモリ上にとられた呼び出し元／呼ばれた側，双方の変数領域をアクセスしながらユニフィケーションが行われていたが，WAMでは呼び出し元引数情報をいったん計算機の汎用レジスタにロードしておき，ユニフィケーションはレジスタに対して行うよう改良された．見方によっては従来言語の手続き呼び出しに限りなく近づいたとも言える．この方式の最大の利点はメモリをハードウェアレジスタに置き換えたことによるアクセス時間の短縮という直接的効果よりも，コンパイラがレジスタの割付けを工夫することによる引数情報のメモリへの退避の削減，無駄なユニファイ命令の削減が可能となったことにある（後述）．ベンチマークで有名なappendプログラムの例では，再帰呼び出しのループを実行している間中，引数をレジスタで渡していくことができる．

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

商用計算機でユーザに開放されている汎用レジスタ数は32個程度であり，スタックなどの制御に使用する分を除くと引数レジスタとして使用できる数は限られてしまうが，統計的には4～8個ぐらいでも性能的には十分と言われている．

b）　バックトラック情報格納の最適化

もう一つの大きな改良点は実行順序制御のための情報を環境フレームと選択点フレームに分離し，選択点フレームはバックトラックが必要な場合に限り生成するよう最適化した点である．DEC-10 Prologではenter命令が常にバックトラックに関連する情報も一緒に格納していたが，WAMではコンパイラがソースプログラムを解析し，必要な場合だけ選択点フレームの作成を指示する（静的な最適化）．

さらに，実行時の最適化としてクローズインデキシングの活用がある．クローズインデキシングはもともとデータベースへの応用を高速化する目的で導入されたもので，候補節がある程度以上の規模になる場合にのみ使用されていた．WAMではこれを単純化し，通常の述語呼び出しに常用するようにした．その結果，呼び出し元引数のデータタイプにより選択される候補節が絞られ，静的には選択肢のある述語でも動的には選択肢がなくなる状況が検出可能となった．たとえば先のappendの例でも，呼び出し元の第1引数がリストかアトム（空リスト[]）かによっていずれかの節を決定的に選ぶことができ，選択点フレームは生成されない．

WAMではこのほかにも，コンパイラがプログラムの静的な解析から得られる情報を最大限に活用し，無駄のない最適な命令列を生成するきめ細かい工夫がなされている．

2)　処理の流れとWAM命令の対応

プログラムに対して実際にどのような命令が生成され，処理が進むかについて図-10の例を用いて説明する．呼び出し元の引数はすでに仮想的な引数レジスタ $A0$, $A1$ にロードされているとする．

①switch命令：候補節グループの選択

呼び出し元の第1引数のデータタイプによりユニファイ可能な候補節群を選択する．具体的には，$A0$ に格納されているデータのタイプに応じて②に述べるtry命令のブロックへの多方向分岐を行う．たとえばswitch-on-term命令は未定義／

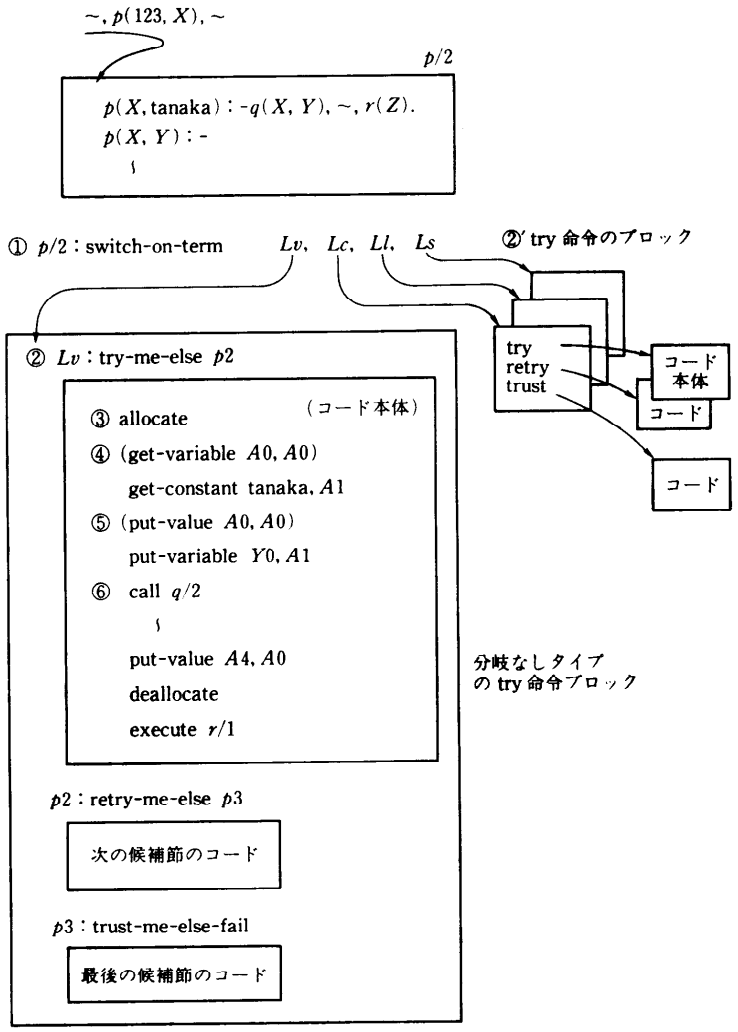


図-10 WAM 方式のオブジェクト命令列

アトム／リスト／構造体の4種類に対応して分岐する．第1引数がキーとなっているデータベース用にハッシュ機能付きの switch-on-constant 命令なども用意されている．

②try 命令：節の選択

switch 命令に対応して候補節はグループ化 (switch-on-term 命令なら4グループ) されるが，一つの節が複数のグループの候補となり得る．たとえば呼び出された側の引数が未定義変数の場合は呼び出し元のデータタイプが何であってもユニファイ可能なため，候補として選択されなければならない．このような節に対してはオブジェクトコードを共用するのが得策であり，節の呼び出し部分だけを try 命令列として独立させるのが良い．すなわち，try 命令を経由することで同一の節が異なるパスで選択可能となる．ただし，実際には高速化を考慮して節への分岐を行わない try-me-else 型の命令群も用意されている．

選択点フレームを作るのも try 命令の重要な仕事であるが，候補節の有無やバックトラック後の実行などの状況に合わせて処理内容を変えたバリエーションが用意されている．たとえば try, try-me-else は選択肢の存在する述語の最初の呼び出しに使用される（選択点フレームを生成）．retry, retry-me-else はバックトラック後の再試行で，かつ，選択肢がまだ存在する場合に使用される（選択点フレーム中の次候補節アドレスを更新する）．trust, trust-me-else-fail は最後の候補節呼び出し用

であるなど.

③allocate 命令：環境フレームの作成

ローカルスタックに環境フレームを生成し，ローカル変数を初期化する.

環境フレームの生成を述語呼び出し処理に一体化せず，このように独立した命令として部品化することにより，コンパイラが最適処理を指示できる．たとえば $p(\ ):-q(\ )$ 型の（トランジット）節では述語 q の呼び出し時に自分の環境を重んでしまえるため初めから環境フレームを作る必要がなく allocate 命令は生成されない.

④get 命令：引数ごとのユニフィケーション処理

(unify 命令：構造体要素のユニフィケーション処理)

ユニフィケーションを実行するのは get 命令で，引数ごとにそのデータタイプに対応した get 命令（get-integer, get-constant など）が生成される．引数が変数の場合，節の中での最初の出現ならば未定義状態であるから呼び出し元の値を無条件で代入する get-variable 命令が生成され，2度目以降の出現なら変数の値をデレファレンスしてから比較をする get-value 命令が生成される．引数が構造体データの場合は最初の get 命令（get-list など）で構造体の型，大きさが比較され（場合によってはグローバルスタックにひな型の生成が行われる），その後に要素ごとの unify 命令が続く.

図-10 の例で変数 X は最初の出現であるため，呼び出し元引数の値をそのまま引数レジスタのどれかに代入すれば良い (get-variable $A?$, $A0$)．ところが X はボディ側での呼び出しでも第1引数であるため，$A0$ を割り当てるのが得策である．するとこのユニフィケーションは「get-variable $A0$, $A0$ ($A0$ の値を $A0$ に代入)」となり，結局，何もしなくても良いことが分かる．WAM ではこのような最適化により実行ステップ数を削減できるのが特徴である.

⑤put 命令：引数レジスタへのロード

述語呼び出しに必要な引数情報を引数レジスタにロードする．get 命令と同様，引数のデータタイプに対応したバリエーション (put-constant, put-variable, put-value など) が用意されている．図-10 の例では，引数レジスタ $A0$ を変数 X として割り当てたのでユニフィケーション完了時点で述語 q の呼び出しのための第1引数 ($A0$) はすでにセット済みとなっている．したがって対応する put 命令は生成されない．ここでもコンパイラが引数レジスタをうまく割り当てることにより実行ステップ数を削減できたことが分かる．この効果を上げるために WAM ではコンパイラが引数レジスタの内容を壊して別の変数を割り当て直すことも行う.

⑥call 命令：述語呼び出し

call 命令の役割は目的のコードへの分岐と戻りアドレスの退避である．WAM では高速化のために戻りアドレスは可能なかぎりレジスタ上に確保される.

節の最後で述語 r の呼び出しから戻ると，残った仕事は呼び出し元へのリターン処理だけである．そこで，呼び出した r から直接，p の呼び出し元へリターンする最適化が考えられる．これはテールリカージョン最適化として手続き型言語でも行われている手法である．その実現には使用していたスタック領域を重み，自分が呼び出される前の環境へ戻してから r の述語呼び出しを行うことが必要となる．このためのスタックを重む deallocate 命令，戻りアドレスの退避を行わない execute 命令（goto と同じ）が用意されている.

⑦proceed 命令：サクセスリターン

call 命令に対応したリターン命令．スタックを重むか否かは deallocate 命令としてコンパイラが指示するため，戻りアドレスに分岐するだけである．述語呼び出しがない場合（右辺がない）や右辺の最後が組込述語の呼び出しである場合に使用される.

7. 逐次実行処理方式の現状と今後の課題

7.1 より高速の処理系に向けて

WAM方式により処理ステップは必要最小のギリギリのところまで削減された．しかしながら処理ステップの短縮は皮肉にも数ステップを稼ぐ細かい最適化の効果をさらに高める結果となり，最適化競争はエスカレートすることになる．WAM をベースにした処理系ではさらに次のような最適化が導入されている.

- 仮想機械語命令の複合化

よく出てくる命令の組合せを1命令化すること

による最適化．たとえばリスト [X|Y] のユニフィケーションのための命令列 get-list/unify-variable/unify-variable を get-list-var-var 命令として1命令化するなど．

- if-then-else 機能の導入

従来言語での if-then-else 機能も Prolog ではバックトラックにより実現するしかないが，これをユーザが陽に指定するかコンパイラで検出してバックトラック処理を回避する．

```
p(X) :- X>0, !, sub 1(X, Y), ···
p(X) :- add 1(X, Y), ···
p(X) :- if(X>0) then {sub 1(X, Y), ···,}
                else {add 1(X, Y), ···,}.
```

- クローズインデキシングの拡張／強化

第1引数のみでなく，第2引数や複数の引数の組合せによりインデキシングする．あるいは特定データタイプのみ切り分ける2方向分岐（switch-on-list）を設けるなど，インデキシングの効果をより高めるための機能拡張．

- 入出力モードの活用

呼び出し元引数が未定義ではないという保証があればユニフィケーションでの代入方向を限定したり，トレイルチェックをなくすなどの最適化が可能となる．入出力モードはユーザに宣言させるのが普通であるがコンパイラで推定できる場合もある（ただし，プログラムミスで保証が崩れるとデバッグは悲惨）．

- RISC アーキテクチャへの対応

RISC マシンの高速性は魅力的であるが，コンパイラにより RISC マシン命令に直接変換する方法はオブジェクトコード量の爆発や，ポインタデータ操作中の割込みなどに配慮が必要である．前者について CISC の20倍という予想[Borriello 87]は大きすぎると思われるが[Maruyama 89]，後者の問題はガベージコレクタも含めて処理系を隅々まで作ってみないと分からないであろう．Quintus Prolog のように WAM エミュレータを RISC でインプリメントするのが現実的な解と思われる．

7.2 動的に追加される節の高速化（知識ベースの構築に向けて）

Prolog の処理系を複雑にしている理由の一つは，述語を動的に変更できることである．加えて，Prolog ではソースイメージの操作がいつでも可能であり Lisp よりも柔軟性が高いが，その点が単純に機械語命令列にコンパイルできない理由にもなっている．したがって，動的に追加される節はインタプリタにより処理されるのが普通であり，当然，その部分の実行速度は非常に遅い．

高速化の一つの手法はインクリメンタルコンパイルである．Prolog ではプログラムが節単位に独立しているため，節の追加時にそれだけをインクリメンタルにコンパイルすることが比較的容易である．ただし，ソースイメージを操作可能とするには，オブジェクトコード中にソーステキストをそのままの形で保持するか，展開された WAM 命令列から復元する方法[Buettner 86],[Konagaya 85],[Atarashi 90] が必要となる．

いずれにしても，このような述語の動的変更は「知識，推論ルールの追加・削除」であり，知識処理では必須機能である．Prolog のベンチマークではほとんど取り上げられないが，実際の応用システムの構築にはその高速化が重要である．

8. おわりに

論理型言語の逐次実行処理について基本的な考え方と WAM 方式について述べた．本論文では細部まで言及しきれなかったが，論理型言語の処理系は推論機構を実現しているという点で手続き型言語にはない工夫がなされており奥が深い．コンパイラによる最適化はほぼ極限近くまで進化しており，逐次型計算機上ではメモリアクセスが性能を抑える主要因となっている．今後は並列化／分散化に期待したい．商用の処理系ではC言語などの他言語呼び出し機能も実現されているが，Lisp のように従来型言語の機能を取り込む機能拡張の方向も考えられる．しかし，論理型であることの枠組みを崩さずにサイドイフェクトを導入するのは困難であろう．この辺に従来型言語との境界が残りそうであるが，論理型言語の備える推論機能，宣言的プログラミング，強力なメタ機能などはいずれもこれからの高度な知識応用システムの構築に有効なメカニズムであり，逐次版の実現方式の研究がその土台の一つを築いたといえるであろう．

最後に本論文をまとめるにあたり，多忙の中有用なコメントをいただいた NTT（株）後藤厚弘氏，（財）ICOT 近藤誠一氏，日本電気（株）新淳君に深謝する．

参考文献

[Atarashi 90] Atarashi, A., Konagaya, A., Habata, S. and Yokota, M.: Implementation and Evaluation of Dynamic Predicates on the Sequential Inference Machine CHI, Proc. of the 1990 International Conf. on Computer Languages, pp. 236-244 (Mar. 1990).

[Atarashi 87] 新 淳, 小長谷明彦: 小型化版 CHI の SUPLOG コンパイラにおける最適化技法, 情報処理学会第 *34* 回全国大会論文集, pp. 749-750 (1987).

[Backus 78] Backus, J.: Can Programming be Liberated from the von Neumann Style? A Functional Style and Ist Algebra of Programs, Comm. of the ACM, Vol. 21, No. 8, pp. 613-641 (1978).

[Borriello 87] Borriello, B., Cherenson, A. R., Danzig, P. B. and Nelson, M. N.: RISCs vs. CISCs for Prolog: A Case Study, In *Proc. of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (1987).

[Bowen 81] Bowen, D. L.: DECsystem-10 PROLOG USER'S MANUAL, Technical Report, Dept. of Artificial Intelligence, University of Edinburgh (1981).

[Buettner 86] Buettner, K. A.: Fast Decompilation of Compiled Prolog Clauses, Proc. of the Third International Conf. on Logic Programming (1986).

[Carlsson 88] Carlsson, M. and Widén, J.: SICStus Prolog User's Manual, Technical Report, SICS R 88007 B, Swedish Institute of Computer Science (Oct, 1988).

[近山 83 a] 近山 隆, 横田 実, 服部 隆, 高木茂行: 推論機械 SIM-P の構成と言語, 情報処理学会第 24 回プログラミングシンポジウム報告集.

[近山 83 b] 近山 隆, 横田 実, 服部 隆: Fifth Generation Kernel Language Version-0, ロジックプログラミングコンファレンス '83 (1983).

[Dobry 85] Dobry, T. P., Despain, A. M. and Patt, Y. N.: Performance Studies of a Prolog Machine Architecture, In *Conference Proceedings of the 12th Auunal International Symposium on Computer Architecture*, pp. 180-190 (June 1985).

[Friedman 76] Friedman, D. P. and Wise, D. S.: CONS Should Not Evaluate its Arguments, Automata, Languages and Programming, Edinburgh University Press, pp. 257-284 (1976).

[Habata 87] Habata, S., Nakazaki, R., Konagaya, A., Atarashi, A. and Umemura, M.: Cooperative High Performance Sequential Inference Machine: CHI, In *Proceedings of 1987 IEEE International Conference on Computer Design* (Oct. 1987).

[Henderson 76] Henderson, P. and Morris, J. M.: A Lazy Evaluator, Proc. 3rd Symp. Principles of Programming Languages, pp. 95-103 (1976).

[Konagaya 85] 小長谷明彦, 阪野正子, 梅村 護: 高性能 PROLOG マシン CHI における Prolog プログラムのコンパイル方式, 情報処理学会第 *31* 回全国大会論文集, pp. 61-62 (1985).

[Maruyama 89 b] 丸山 勉, 神津伸一, 横田 実, 森島 潔: AI 言語向き RISC アーキテクチャ, 情報処理学会第 *39* 回全国大会論文集 (1989).

[Nakashima 87] Nakashima, H. and Nakajima, K.: Hardware Architecture of the Sequential Inference Macnine: PSI-II Proc. of the 1987 Symposium on Logic Programming pp. 104-113 (1987).

[Nakata 88] 中田登志之, 新 淳, 中島 震: 高水準言語計算機の命令セットアーキテクチャ, 情報処理, Vol. 29, No. 12 (Dec. 1988).

[Nylén 86] Nylén, M., Hugosson, A. and Barklund, J.: *Tricia User's Guide*, Uppsala Programming Methodology and Artificial Intelligence Department of Computing Science, Uppsala University (July 1986).

[Onai 85] Onai, R., Aso, M., Shimizu, H., Masuda, K. and Matsumoto, A.: Architecture of a Reduction-Based Parallel Inference Machine: PIM-R, New Generation Computing, Vol. 3, pp. 197-228 (1985).

[Qui 90] Quintus Computer Systems, Inc.: *Quintus Prolog Reference Manual*, release 2.5 edition (Jan. 1990).

[Warren 77 a] Warren, D. H. D.: Implementing Prolog—Compiling Predicate Logic Program, Vol. 1-2, D. A. I. Research Report, No. 39-40, Department of Artificial Intelligence, Univ. of Edinburgh (1977).

[Warren 83] Warren, D. H. D.: An Abstract Prolog Instruction Sets, Technical Report 309, AI Center, SRI International (1983).

[Yokota 84] Yokota, M., Yamamoto, A., Taki, K., Nishikawa, H., Uchida, S., Nakajima, K. and Mitsui, M.: A Microprogrammed Interpreter for the Personal Sequential Inference Machine, Proc. of the Int. Conference on FGCS '84 (1984).

(平成 3 年 2 月 13 日受付)

横田 実 (正会員)

1950 年生. 1973 年慶應義塾大学工学部電気工学科卒業. 同年, 日本電気(株)に入社. 現在, 同社 C&C システム研究所コンピュータシステム研究部研究課長. 主に高級言語マシンのアーキテクチャ研究に従事し, これまでに COBOL マシン, Prolog マシン, Lisp マシンを研究開発. 1982 年から 1986 年の間通産省第五世代プロジェクトに出向し, 推論マシン PSI の開発に従事. 現在は使いやすいコンピュータの実現に関心を持ち, 並列オブジェクト指向言語の研究を進めている. 1986 年に慶應義塾大学より博士号を取得.