Prologの最適化方式†

阿部重夫†† 川端薫†† 黒沢憲一††

非決定的な実行過程で特徴づけられるPrologの実行の高速化を図るためには，できるだけ決定的な処理となるように最適化することが必要である．このためプロセジャの実行方式により，決定的なタイプ1，シャローバックトラックしか生じないタイプ2，シャローバックトラックとデープバックトラックが生じるタイプ3に分類し，各タイプに適した最適化方式を提案した．これによりタイプ2では，バックトラック情報の退避／回復が不要となり，またタイプ3では，バックトラック情報の退避は必要であるが，シャローバックトラック時の回復が不要となり，これらのタイプのプロセジャの高速化が可能となった．内蔵型Prologプロセッサ IPP を用いた性能評価では，quick-sort, 8-queen で各々，従来方式の2.4倍 (921 kLIPS)，2.8倍 (1137 kLIPS) の性能向上を達成した．また Prolog で書かれた IPP の最適化コンパイラの一部を静的に解析したところ，タイプ1，2およびデープバックトラックしか生じないプロセジャを除いたタイプ3のプロセジャが全体の92%に達し，今回開発した最適化方式が一般のプログラムでも適用可能であることを明らかにした．

1. はじめに

論理型言語 Prolog は，言語自身に推論機能を持ち，知識処理に適した言語である．推論性能が低いという欠点も Warren の命令セット[1]により大幅に改善されつつあり，この命令をベースにした処理系の開発が進められている[2)~9)]．

Prolog の実行は，ユニフィケーションによるパターンマッチング処理と，それに失敗したときのバックトラック処理で特徴づけられるが，特にバックトラックによる性能低下が著しい．Warren の命令セットでは，プロセジャ呼び出し時に，第1引数の種類によって飛び先を決めるインデキシングにより別解を刈り，できるだけ不要なバックトラックが生じないようにしている．これにより Prolog の性能測定用プログラム append は，決定的となり大幅な性能向上が可能となる．しかしながら Warren の命令セットの欠点は，決定的なプログラムと非決定的なプログラムとの性能差が大きいことにある．これに対し，文献8)，10)，11)では，最適な引数でインデキシングすることにより，別解をできるだけ刈る方式を提案している．さらに文献12)では，非決定的なプログラムを決定的なプログラムに変換する方式を提案している．

これらの方式を用いても，非決定的なプログラムを完全になくすことはできないため，Prolog の高速化のためには，バックトラック自体の高速化が不可欠である．バックトラックには，失敗したクローズと同一名称のクローズ内でバックトラックするシャローバックトラックと，デープバックトラックとに分類される．しかしながらバックトラックの高速化について論じた論文は極めて少ない．文献13)，14)では，カットによりシャローバックトラックしか生じないことがある場合に着目して，シャローバックトラックを高速化する方式を提案している．

本論文では，文献8)等の最適引数方式を拡張するとともに，文献13)等のシャローバックトラックの高速化方式を一般化する．さらにこれらの方式をベンチマークプログラムで評価した結果を述べる．

2. 高速化の考え方

同一ヘッド名称のクローズからなるクローズの集合をプロセジャと呼ぶが，k 個のクローズからなるプロセジャが次のように与えられるとする．

$h(H_1^1, \cdots, H_n^1) :- b\_in^1, !, b^1(B_1^1, \cdots, B_{m_1}^1), \cdots$
$h(H_1^2, \cdots, H_n^2) :- b\_in^2, !, b^2(B_1^2, \cdots, B_{m_2}^2), \cdots$
$\cdots\cdots\cdots\cdots$
$h(H_1^{k-1}, \cdots, H_n^{k-1}) :- b\_in^{k-1}, !, b^{k-1}(B_1^{k-1}, \cdots, B_{m_{k-1}}^{k-1}), \cdots$
$h(H_1^k, \cdots, H_n^k) :- b\_in^k, b^k(B_1^k, \cdots, B_{m_k}^k), \cdots$

ここで b_in^i $(i=1, \cdots, k)$ は，決定的な組込述語であり，それらの引数は省略してある．

上記の k 個のクローズへのカットの入り方によりプロセジャの実行の仕方が異なってくる．そこでプロセジャを次のように3つのタイプに分類する．

(1) タイプ1：$k=1$ の場合すなわち決定的なプロセジャ
(2) タイプ2：$b\_in^1 \sim b\_in^{k-1}$ の後に必ずカットがありシャローバックトラックしか生じないプロ

† Optimization of Prolog Programs by SHIGEO ABE, KAORU KAWABATA and KEN-ICHI KUROSAWA (Hitachi Research Laboratory, Hitachi, Ltd.).
†† (株)日立製作所日立研究所

セジャ

(3) タイプ3: $b\_in^1 \sim b\_in^{k-1}$ の後にカットがないクローズがあり，シャローバックトラックと，デープバックトラックとが生ずるプロセジャ

なおタイプ3には後で示すようにシャローバックトラックが全く生じない場合も起こりうる．

ここで Warren の命令セットでは，2つ以上のクローズからなるプロセジャを決定的とするために，プロセジャ呼び出し時の第1引数が，変数，定数，リスト，構造体のいずれであるかによりインデキシングする方法を採っている．さらに第1引数が，定数，構造体のときは，ハッシングを行っている[1]．したがってこのインデキシングで分かれたクローズに対して上記分類をすればより効果的となる．プロセジャを決定的とすることにより大幅な性能向上が得られるため，インデキシングの強化は必須である．

バックトラックが生じるのはタイプ2，3のプロセジャであるが，タイプ2のプロセジャの例としては，次に示す quick-sort[15]の split がある．

```
split([X|L], Y, [X|L1], L2) :-
        X=<Y, !,
        split(L, Y, L1, L2).
split([X|L], Y, L1, [X|L2]) :-!,
        split(L, Y, L1, L2).
split([ ], _, [ ], [ ]).
```

ここで比較演算=<は，決定的であるのでこの比較に成功すれば，直後にカットがあるため，デープバックトラックは生じない．また第2番目のクローズのヘッドの直後にもカットがあるため，第2番目のクローズのヘッドのユニフィケーションに成功した後にはデープバックトラックは生じない．このように k 個のクローズから成るプロセジャの最初 $k-1$ 個のクローズのヘッドあるいはそれに続く決定的な組込述語の後にカットがあればシャローバックトラックしか生じないプロセジャとなる．

タイプ3のプロセジャの例としては次に示す 8-queen の select がある．

```
:-select(+, -, -).
select([A|L], A, L).
select([A|L], X, [A|L1]) :-
        select(L, X, L1).
```

select の第1引数のみが入力で，しかもどちらのクローズの第1引数も同じリストであり，入力がリストであれば成功する．したがって入力がリストのときは，デープバックトラックしか生ぜず，それ以外のときは失敗する．すなわちこの例では，シャローバックトラックは生じない．

バックトラック情報をセーブ／リストアする Warren の indexing 命令は，デープバックトラック用の命令であり，したがって特にタイプ2のプロセジャに対しては効率が悪く，これらのシャローバックトラックを効率良く実行する方式の開発が必要である．

3. クローズインデキシング

クローズインデキシングをより効果的にするためには，プロセジャ呼び出しの引数が入力，出力，入出力であることを示すモード情報に基づいて最適な引数を選択することが必要である．モード情報は，モード宣言，あるいはモード推論[17]によって得ることができる．モード情報が得られない場合は，引数のモードが入出力であるとして扱う．

入力および入出力モードの引数の中から次の基準に従って最適な引数を選択する．

(1) プロセジャのヘッドの引数が，定数，リスト，構造体のどれかになり，しかも各々に対応するクローズの数が1個以下となるときその引数を最適引数とする．

(2) ヘッドにおいてリスト，構造体中には現れない変数が，ヘッドの直後の型チェックの組込述語中に現れるときその変数を最適引数とする．

(3) 定数あるいは構造体を最も多く含む引数を最適引数とする．

入力モードで(1)の条件を満たす引数がある時は，タイプ1プロセジャとなる．このときは，変数が入力として入ってきたらフェイルする必要があるから，入力が変数に対して，クローズをリニアサーチするコードの生成は不要である．

(2)の条件を満たす引数のあるプロセジャがタイプ1となるのは，次の形のときである．

```
p(…, X, …) :-var(X), !, …
p(…, X, …) :-…
```

ここで，第1番目のクローズの入力，あるいは入出力モードの引数は，Xだけか，ほかにもある時は，必ず成功するか，失敗する可能性のある時は第2番目のクローズと同じことが必要である．型チェックの組込述語は，nonvar(X)でも良い．上記条件を満たすときは，Xが変数であれば，第1番目のヘッドとユニフィケーションに成功し，しかも var(X) の後にカットが

```
1.   :-p(?, -).
2.   p(X, a):- var(X), !.
3.   p(_, b).
4.   q(X, Y):- p(X,Y).
```

(a) タイプチェックの例

```
1. p1: get_constant a, A2
2.     proceed
3. p2: get_constant b, A2
4.     proceed
5.     execute A1, (p1,p2,p2,p2)
```

(b) 本方式によるコード

```
1. c1: try_me_else c2
2. p1: get_constant a, A2
3.     var A1
4.     fcut
5.     proceed
6. c2: trust_me_else fail
7. p2: get_constant b, A2
8.     proceed
9.     execute A1, (c1,p2,p2,p2)
```

(c) Warrenの方式によるコード

図 1 タイプチェックプログラムに対するコード
Fig. 1 The code for a type checking program.

あるため，第2番目のクローズを別解とする必要がない．またXが変数でないときは，第2番目のクローズを選択すれば良いから，タイプ1のプロセジャとなる．

例えば**図1**(a)に対するコードは，(b)のようになり，Warren 方式の(c)のコードに比較して，極めて効率が良くなる．図の execute Ai, (c_v, c_c, c_s, c_l) は，引数レジスタ Ai の内容が変数，定数，構造体，リストかに従って，c_v, c_c, c_s, c_l に飛ぶ命令で Warren の命令を拡張したものである[13]．図1(b)では，var(x) のコード生成も不要となる．

atom(X), integer(X)[18] 等の型チェックの組込述語もあるが，これらの組込述語が，図1(a)の var(X) の替りにあってもタイプ1のプロセジャとはならない．それは，atom(X) 等は，定数の1つの属性であり，定数であっても atom でない場合別解を探す必要があるからである．

型チェックの組込述語に，定数，構造体，リストであることを判定する述語 constant(X), structure(X), list(X) を追加することは，プログラムの分かりやすさ，処理速度を向上する上で極めて有効である．例えば**図2**のプログラムの第1引数でインデキシングするとすると，第1引数は，第2クローズから順に定数，リスト，構造体，変数である．したがって第5クローズが，第2から第4クローズの各々の別解となる．もし第5クローズの変数Xの種類が，定数でこれを型チェック組込述語 constant(X) で明示することとすれば，第5クローズは，第2クローズのみの別解で，第3，4クローズは，決定的となる．

```
1.  :- p(+,...).
2.  p([],...).
3.  p([X|Y],...):-
4.  p(f(a),...):-
5.  p(X,...):- constant(X),..
```

図 2 タイプチェックが効果的な例
Fig. 2 An example of effective type checking

(3)の条件を満たす引数が入力モードでないときは，次に多くの定数あるいは構造体を含む引数があればそれを選び2つの引数でインデキシングするようにし，最初の引数が，プロセジャ呼び出し時変数となっても2つ目の引数でインデキシングすることによりリニアサーチとなることを避ける．(2引数によるインデキシングのコード生成方式および，定数，構造体等のハッシング方式に関しては，文献11)参照)

4. バックトラックの最適化

4.1 考え方

バックトラックを実現するための Warren の indexing 命令には次のものがある．

try C	try_me_else C
retry C	retry_me_else C
trust C	trust_me_else fail

try 命令は，バックトラックに必要な情報を持ったチョイスポイントをローカルスタックに格納する．チョイスポイントの格納後，オペランドに指定されたクローズとのユニフィケーションを行い，ユニフィケーションに失敗すると，try 命令の直後の命令を実行する．retry 命令は，ラストチョイスポイントからバックトラック情報の回復処理を行い，以降 try 命令と同様の処理を行う．trust 命令は，別解の最後のクローズについて用いられ，ラストチョイスポイントをスタックからポップアップする以外は retry 命令と同じである．try_me_elst, retry_me_else では try 命令とは逆にこれらの命令の後のクローズとのユニフィケーションを行い，それに失敗したときオペランドで指定された別解のクローズを実行する．

上記のインデキシング命令は，デープバックトラックのために考えられたもので，タイプ2，タイプ3のプロセジャでのシャローバックトラックに対してチョイスポイントの退避回復を行っていたのでは処理効率が極めて悪くなる．このためシャローバックトラック

ができるための必要十分なバックトラック情報を決めることが必要となる.

チョイスポイントの内容は以下のとおりである[1].

(1) 別解へのポインタ (ACP)
(2) 引数レジスタ (A_1〜A_m, m: 引数個数)
(3) top-of-heap レジスタ(H)
(4) top-of-trail レジスタ (TR)
(5) ラストチョイスポイントレジスタ(B)
(6) 継続プログラムポインタレジスタ (CP)
(7) ラストエンバイロンメントレジスタ(E)

バックトラックは, チョイスポイントに格納されたACP から開始されるので, シャローバックトラックは ACP を書き替えることによって制御できる.

プロセジャ呼び出し時, プロセジャの引数は, 引数レジスタにロードされている. 引数レジスタとユニフィケーションを行う get 命令を実行した後で, 引数レジスタが unify, put 命令で書き替えられていなければ, undo 処理をすることにより引数レジスタの内容を回復することができる. したがって get 命令でユニファイされた引数レジスタの退避は不要である.

HおよびTR は, シャローバックトラック開始直前のクローズのユニフィケーションでヒープあるいはトレイルスタックに書き込みがあればそれらをリセットしなければならないので退避することが必要である.

シャローバックトラックが始まった時点では, 新たにチョイスポイントを作る必要がないためBの退避は不要である. またシャローバックトラック時, CP は変更されないから CP の退避は不要である. エンバイロンメントを, バックトラックが発生しなくなった時点で生成するようにすれば, Eの退避は不要である.

以上よりシャローバックトラック時に退避の必要性が生じる可能性のあるレジスタは, ACP, A_1〜A_m, H, TR のみとなる.

4.2 バックトラックの枠組

タイプ3のうちデープバックトラックしか生じないプロセジャに対するコードは try 命令を indexing 命令として使うと**図3**のようになる. これは従来の Warren 方式に基づいたコードである. これに対して, シャローバックトラックしか生じないタイプ2のプロセジャに対するコードは, **図4**のようにすればよい.

ステップ1で ACP をワークレジスタ wk1 に退避し, ステップ2でシャローバックトラックのアドレス

```
1.      try d1
2.      retry d2
        .....
3.      trust dk
4.  d1: code for h(H1_1,...):-
5.  d2: code for h(H2_1,...):-
        .....
6.  dk: code for h(Hk_1,...):-
```

図3 デープバックトラックしか生じないタイプ3プロセジャに対するコード
Fig. 3 The code for a type 3 procedure with no shallow backtracking.

```
1.  s1: move ACP, wk1          /* save ACP */
2.      movea s2, ACP          /* set ACP */
3.      move H,wk2             /* save H */
4.      move TR,wk3            /* save TR */
5.      code for h(H1_1,...):- b_in1
6.      move wk1, ACP          /* restore ACP */
7.      code for b1(B1_1,...),..
8.  s2: undo wk3, TR           /* undo variables */
9.      move wk2, H            /* restore H */
10.     movea s3, ACP          /* set ACP */
11.     code for h(H2_1,...):-b_in2
12.     move wk1, ACP          /* restore ACP */
13. s3:
        .....
14. sk: undo wk3, TR           /* undo variables */
15.     move wk2, H            /* restore H */
16.     move wk1, ACP          /* restore ACP */
17.     code for h(Hk_1,...):-...
```

図4 タイプ2プロセジャに対するコード
Fig. 4 The code for a type 2 procedure.

s2 を ACP にロードする. ただし movea s2, ACP は, s2 のアドレスを ACP に転送することを意味するとする. そしてステップ3, 4でH, TR をワークレジスタに退避して, ステップ5を実行する. もしステップ5でユニフィケーションに失敗すれば, バックトラックにより s2 へ飛び, そうでなければ, ステップ6で wk1 に退避した ACP の値を回復して, ステップ7を実行する.

ステップ8では, undo 命令[16]により, wk3 から TR までに積まれた拘束された変数のアドレスを基に, 変数の解放処理を行う. ステップ9は, ヒープのリセットのためであり, 以下ステップ12までは, 前と同様である. 最後のクローズでは, ステップ16で退避していた ACP を回復することが以前のクローズと異なる.

シャローバックトラックしか生じないため, 退避情報はすべてレジスタ上に格納することができ, 図3のコードに比較して格段にバックトラック処理のオーバヘッドを削減できる. しかも後で示すように, 場合によってはHの退避/回復, TR の退避と, undo 処理は削除することも可能となる.

```
1.  e1: try d1
2.  e2: retry d2
        .....
3.  ek: trust dk
4.  d1: movea s2, ACP              /* set ACP */
5.      code for h(H1_1,..):-b_in1
6.      movea e2, ACP
7.      code for b1(B1_1,...),..
8.  s2: undo TR_B, TR              /* restore TR */
9       move H_B, H                /* restore H */
10.     movea s3, ACP
11. d2: code for h(H2_1,...):-b_in2
12.     movea e3, ACP
13.     code for b2(B2_1,...),..
        .....
14. sk: undo TR_B, TR              /* undo variables */
15      move H_B, H                /* restore H */
16.     move B_B, B                /* pop up choice point */
17. ek: code for h(Hk_1,...):-...
```

図5 タイプ3プロセジャに対するコード
Fig. 5 The code for a type 3 procedure.

シャローバックトラックとデープバックトラックの両方が生じうるタイプ3のプロセジャのコードは，**図5**のように作る．この場合は，デープバックトラックも生じうるので，try 命令でチョイスポイントを作成し，シャローバックトラック時は，チョイスポイントから必要データのみを回復して使用する．以下で添字Bは，チョイスポイント内のデータであることを示すこととする．

ステップ1～3は，図3のステップ1～3と同じであるが，図3の場合と異なり，チョイスポイント内の ACP は後で書き替えられるため，ステップ1，2で ACP への値の書き込みは省略できる．ステップ4で ACP をシャローバックトラックのアドレス s2 に書き替える．ステップ5で成功すれば，ステップ6へ，そうでなければ，ステップ8へ進む．ステップ6ではデープバックトラックに備えて，ACP をアドレス e2 に置き替え，ステップ7を実行する．

シャローバックトラックが生じた時は，ステップ8で，チョイスポイント内の top-of-trail レジスタの値 TR_B から TR までに拘束された変数の解放処理を行う．ステップ9，10でチョイスポイント内の top-of-heap レジスタの値 H_B にHの値をリセットし，シャローバックトラックに備えて，ACP を s3 に書き替える．以下ステップ12，13は，ステップ6，7と同様である．

シャローバックトラックにより最後のクローズを実行するときは，ステップ14，15で変数の解放，ヒープのリセットを行う．ステップ16は，ステップ1で退避したチョイスポイントをポップアップするためのものである．

ステップ8，9，14，15は条件によっては削除が可能である．図3の場合に比較して，最初のクローズ実行の直前，およびデープバックトラック開始時点では，チョイスポイントの生成，および回復処理が入るが，シャローバックトラック時は，必要な情報のみの回復に限定できるために，バックトラックのオーバヘッドを削減できる．

あるクローズから次のクローズへの実行が，シャローバックトラックか，デープバックトラックかによってさらに最適化することも可能である．例えば，第1番目のクローズの b_in^1 の後にカットがあるとすれば，ステップ2，6は不要となる．

4.3 スタック管理の最適化

（1） ローカルスタックの管理

ローカルスタックに積まれるフレームには，チョイスポイントと，エンバイロンメントとがある．チョイスポイントは，前述したようにタイプ3のプロセジャでは生成するが，タイプ1，2のプロセジャでは生成しない．エンバイロンメントは，ボディゴールを渡って現れるパーマネント変数およびゴールの実行のための制御情報を記憶する．エンバイロンメントを生成するか否かは，クローズに依存するが，生成した後でシャローバックトラックが生じると，その生成がオーバヘッドとなる．したがってエンバイロンメントの生成は，シャローバックトラックに失敗する可能性がなくなってからにする．

（2） ヒープの管理

出力となる引数を除いて，引数レジスタの内容と，ヘッドの引数とのユニフィケーションでヒープにコピーが作られるのは，次のいずれかの条件が成り立つときである．

（i） 入力となるヘッドの引数が多重のリストあるいは構造体である．

（ii） 入出力モードのヘッドの引数がリストか構造体である．

（iii） b_in^i が＝を含みそのどちらの引数も出力モードでなく，入力および入出力モードの変数が含まれる．

上記の条件が満たされないときは，シャローバックトラック時の top-of-heap レジスタの退避/回復処理は不要となる．(図4ステップ3，9，15，図5ステップ9，15)

（3） トレイルスタックの管理

変数が拘束されたとき 拘束された変数のアドレス

は，トレイルスタックに積まれる．入力および入出力モードに対応する引数レジスタ中の変数は，次のどれかの条件を満たすときに拘束される．

(i) (2)のコピー生成の条件をヘッドの引数が満たす．
(ii) ヘッドの入力あるいは入力モードの変数中に同一の変数がある．
(iii) b_in' が is を含み，左辺の引数が入力引数中のリストあるいは構造体中に現れるか，入出力引数中に含まれる．

もし上記の条件が満たされないときは，シャローバックトラック中の変数の undo 処理および TR の退避は不要である．(図4ステップ4，8，14，図5ステップ8，14)

ここで文献1)に示された Prolog 命令では，現在のチョイスポイントより上に積まれた変数の拘束については，トレイルに積んでない．もしこの方式を採るとするならば，タイプ2のプロセジャで undo 処理が必要な場合，図4において，ステップ1の後で，Bの値をワークレジスタに退避し，Bの値にローカルスタックの先頭アドレスを設定し，ユニフィケーションに成功したステップ12，およびステップ14の後でワークレジスタに退避した値をBに回復すればよい．

5. クローズの最適化

文献11)では，ユニフィケーションに成功する時の実行ステップ数が最少となるようにレジスタ競合を解消する方式を述べているが，この方式ではユニフィケーションに失敗したときにオーバヘッドが増える可能性がある．この問題を解決するため我々は，

(i) 失敗をできるだけ早く検出する最適化方式
(ii) プロセジャ内の共通のユニフィケーションをシャローバックトラック時に削除する最適化方式

を開発した[13),14)]．

すなわち，ユニフィケーションの中で失敗する可能性のあるものを検出し，これらのコードを代入となるユニフィケーションおよびゴール生成のコードの前に作成する．前者のコードをチェックコード，後者のコードをロードコードと呼ぶ．チェックコード，ロードコードの各々で生じるレジスタ競合は，文献11)のように実行順序を変えて解消する．これに対して両者にまたがるレジスタ競合は，ワークレジスタを割りあてて解消する．これは不要な代入処理をシャローバックトラックの前にしないためである．

チェックコードの引数レジスタは，リスト，構造体要素のユニフィケーション中に破壊されても，次のクローズで同一のユニフィケーションがあるときは，回復する必要はない．このときはユニフィケーションを最初に行い後は省略できる．そうでないときはワークレジスタを割りあてて競合解消を図る．コード生成の詳細な方法については文献13)，14)を参照して頂きたい．

6. コード生成例

(例1) 文献15)の微分のプログラム中のプロセジャ d を第1引数でハッシングしたときの構造体 "+" の下記の別解を考える．

```
:-d(+, +, -).
d(U+V, X, DY) :-!, d(U, X, DU), d(V, X, DV),
                    simp-plus(DU, DV, DY).
d(X, X, 1) :-!.
d(C, X, 0) :-atomic(C), C ¥==0.
```

上記はタイプ2のプロセジャとなり，**図6**のコードが生成される．第2番目のクローズで失敗する可能性のあるのは，構造体 "+" とのユニフィケーションの

```
1.  s1: move ACP, A4          /* save ACP */
2.      movea s2, ACP         /* set ACP */
3.      move H, A5            /* save H */
4.      move TR, A6           /* save TR */
5.      get_structure "+", A1
6.      allocate
7.      unify_variable A1
8.      unify_variable V
9.      get_variable X, A2
10.     get_variable DY, A3
11.     put_variable DU, A3
12.     move A4, ACP          /* restore ACP */
13.     call A1, d
14.     put_value V, A1
15.     put_value X, A2
16.     put_variable DV, A3
17.     call A1, d
18.     put_unsafe_value DU, A1
19.     put_unsafe_value DV, A2
20.     put_value DY, A3
21.     deallocate
22.     execute A1, simp_plus
23. s2: movea s3, ACP         /* set ACP */
24.     get_value A1, A2
25.     get_constant 1, A3
26.     move A4, ACP          /* restore ACP */
27.     proceed
28. s3: undo A6, TR           /* undo varaibles */
29.     move A5, H            /* restore H */
30.     move A4, ACP          /* restore ACP */
31.     atomic A1
32.     noteq A1, 0
33.     proceed
```

図6 プロセジャ "d" に対するコード
Fig. 6 The code for procedure "d".

みであるから，allocate でエンバイロンメントを生成する前に，ステップ5で構造体とのユニフィケーションを行っている．第2番目のクローズでは，失敗するまで変数の拘束，ヒープのコピーが生じないため，第3番目のクローズの先頭での undo 処理，ヒープの回復処理が不要となる．これに対し第3のクローズでは，入力モードの第1引数と第2引数とのユニフィケーションにより，変数の拘束，コピーの生成の可能性があるため，undo 処理，ヒープの回復処理は，省略できない．(ステップ28, 29)

(例2) quick-sort の次のプロセジャを考える．

```
:-mode split(+, +, −, −).
split([X|L], Y, [X|L1], L2) :-X=<Y, !,
                split(Y, L, L1, L2).
split([X|L], Y, L1, [X|L2]) :-split(Y,L,L1,L2).
split([ ], _, [ ], [ ]).
```

これに対するコードは図7のようになる[13),14)]．split はタイプ2のプロセジャとなり，シャローバックトラック時の変数の undo 処理，およびヒープの回復処理は不要である．また第2番目と第3番目のクローズの第2引数は同一であるので，第3番目のクローズでのユニフィケーションは省略できる．なお ACP の退避と設定をステップ5，6で行っているのは，ステップ1，2で失敗すれば，クローズ2でも失敗するためである．

文献12)では，共通のユニフィケーションの省略のためにローカルスタックにユニファイされた変数を格納する方式を採っているが，本方式ではレジスタ上で処理することができ，退避回復処理が不要となるためさらに処理効率が良くなる．

```
1. s1:  get_list A1
2.      unify_variable A5
3.      unify_variable A1
4.      move ACP, A6              /* save ACP */
5.      movea s2, ACP             /* set ACP */
6.      leseq  A5, A2
7.      get_list  A3
8.      unify_value  A5
9.      unify_variable  A3
10.     move A6, ACP              /* restore ACP */
11.     execute A1, (fail, s3, s1, fail)
12. s2: move A6, ACP              /* restore ACP */
13.     get_list A4
14.     unify_value  A5
15.     unify_variable  A4
16.     execute A1, (fail, s3, s1, fail)
17. s3: get_nil A1
18.     get_nil A3
19.     get_nil A4
20.     proceed
```

図7 プロセジャ "split" に対するコード
Fig. 7 The code for procedure "split".

7. 性能評価

7.1 動的評価

IPPと最適化コンパイラとを用いて最適化方式の評価を行った．文献15)のベンチマークプログラムを用いて，次の4つの最適化方式について評価した．

(1) 文献1)により推定される Warren の最適化方式
(2) クローズインデキシングの最適化
(3) 項(2)＋クローズの最適化
(4) 項(2)，(3)＋バックトラックの最適化

表1にその結果を示す．append は決定的であるため最適化の効果はない．quick-sort では，例で示したとおり，split がタイプ2のプロセジャとなり，シャローバックトラックで2倍高速化されている．これに対して 8-queen では，クローズインデキシング，クローズの最適化の効果が大きく，シャローバックトラックの最適化の効果は少ない．

7.2 静的評価

一般的なプログラムでの最適化方式の効果を調べる

表1 最適化方式の効果 (kLIPS)
Table 1 Speedup by optimization techniques (in kLIPS).

No.	プログラム	方式1	方式2	方式3	方式4
1	append	1125 (1)	1125 (1)	1125 (1)	1125 (1)
2	q-sort	387 (1)	387 (1)	458 (1.2)	921 (2.4)
3	8-queen first	407 (1)	830 (2.0)	1133 (2.8)	1137 (2.8)
4	8-queen all	394 (1)	796 (2.0)	1079 (2.7)	1081 (2.7)

方式1：Warren の最適化
方式2：クローズインデキシングの最適化
方式3：方式2＋クローズの最適化
方式4：方式2，3＋バックトラックの最適化
( )：高速化率

表2 プロセジャの分類
Table 2 Classification of procedures.

方式	タイプ1	タイプ2	タイプ3	計
Warren 方式	67 (29)	—	168 (71)	235 (100)
インデキシング＋バックトラック最適化	79 (34)	113 (48)	43(23) (18)	235 (100)

( )：全体の割合%
タイプ3の(23)は，43個のうち，23個シャローバックトラックがあることを示す．

ために，IPP の最適化コンパイラの一部を用いて静的に評価した．シャローバックトラックの最適化方式を除いては，文献11)で評価しているため，プロセジャのタイプの分析を行った．**表2**に235個のプロセジャに対する分析結果を示す．Warren方式では，タイプ1とタイプ3のプロセジャしかないが，タイプ1の判定は第1引数の入出力モードに従って行った．Warren方式でタイプ1となったプロセジャは，67個であるがこのうち25個がもともと決定的なプロセジャ（すなわちクローズ数が1）であった．インデキシングの最適化により，タイプ1は79個（全体の34%）となった．この内訳は，最適引数の選択によりタイプ1となったのが7個で，型チェック組込述語によるインデキシングでタイプ1となったのが5個である．バックトラックの最適化によりタイプ2となったプロセジャは113個で，タイプ1とタイプ2で全体の82%，シャローバックトラックを含むタイプ3まで含めれば92%となった．IPPのコンパイラは1つの版しかないため動的評価まで行うことができなかったが，この静的評価によって一般のプログラムに対して本手法が有効であることが分かった．

8. おわりに

非決定的な実行過程で特徴づけられるPrologの実行の高速化を図るためには，できるだけ決定的な処理となるように最適化することが必要である．

このためプロセジャを実行方式により，決定的なタイプ1，シャローバックトラックしか生じないタイプ2，シャローバックトラックとデープバックトラックが生じるタイプ3に分類し，各タイプに適した最適化方式を提案した．これによりバックトラックのある時，従来不可能であったタイプ2，3のプロセジャの高速化が可能となった．

内蔵型 Prolog プロセッサ IPP を用いた性能評価では，quick-sort, 8-queen で各々，従来方式の2.4倍(921 kLIPS)，2.8倍（1137 kLIPS）の性能向上を達成した．また Prolog で書かれた IPP の最適化コンパイラの一部を静的に解析したところ，タイプ12，およびデープバックトラックしか生じないプロセジャを除いたタイプ3のプロセジャが全体の92%に達し，今回開発した最適化方式が一般のプログラムでも適用可能であることを明らかにした．

謝辞 IPP のプロジェクトに参加した各位，特に IPP の Prolog 処理系の開発に協力頂いた日立プロセスコンピュータエンジニアリング鈴木弘氏，大津善行氏に感謝の意を表する．

参考文献

1) Warren, D. H. D.: An Abstract Prolog Instruction Set, Technical Note 309, Artificial Intelligence Center, SRI International (October 1983).
2) Nakazaki, R. et al.: Design of a High-speed Prolog Machine (HPM), *Proc. 12th ISCA*, pp. 191-197 (1985).
3) Dobry, T. P. et al.: Performance Studies of a Prolog Machine Architecture, *ibid*, pp. 180-190.
4) Despain, A. M.: A High Performance Prolog Co-processor, *Proc. WESCON 85*, No. 18/2 (1985).
5) Yokota, M. et al.: The Design and Implementation of a Personal Sequential Inference Machine: PSI, *New Generation Computing*, Vol. 1, pp. 125-144 (1983).
6) Kurokawa, T. et al.: A Very Fast Prolog Compiler on Multi Architecture, *Proc. FJCC 86*, pp. 963-968 (1986).
7) 新井ほか：Prolog コンパイラの設計，昭和62年度人工知能学会全国大会（第1回），pp. 185-188 (1987).
8) Abe, S. et al.: High Performance Integrated Prolog Processor IPP, *Proc. 14th ISCA*, pp. 100-107 (1987).
9) Yamaguchi, S. et al.: Architecture of High Performance Integrated Prolog Processor IPP, *Proc. FJCC 87*, pp. 175-182 (1987).
10) Abe, S. et al.: Performance Evaluation of Integrated Prolog Processor IPP, *Proc. Int. Workshop on Artificial Intelligence for Industrial Applications*, pp. 505-510 (1988).
11) 桐山ほか：内蔵型 Prolog プロセッサ IPP の最適化コンパイル方式の提案と性能評価，情報処理学会論文誌，Vol. 29, No. 6, pp. 589-595 (1988).
12) 松本ほか：PROLOG コンパイラにおける非決定性処理の最適化方式，情報処理記号処理研究会，45-2 (1988).
13) 阿部ほか：PROLOG におけるシャロウバックトラックの高速化方式，情報処理学会計算機アーキテクチャ研究会，pp. 25-31 (1988. 7).
14) 阿部ほか：モード情報を用いたシャローバックトラックの高速化方式，第37回情報処理学会全国大会論文集，pp. 613-614 (1988. 9).
15) 奥野：第3回 LISP コンテスト及び第1回 PROLOG コンテストの議題案，情報処理学会記号処理研究会資料，28-4 (1984. 6).
16) Kurosawa, K. et al.: Instruction Architecture for a High Performance Integrated Prolog

Processor IPP, *Proceedings of the Joint Fifth Symposium and Fifth International Conference on Logic Programming* (1988).
17) Debray, S. K. and Warren, D. S.: Automatic Inference for Prolog Programs, *Proceedings of 1986 Symposium on Logic Programming*, pp. 78-88 (1986).
18) Bowen, D. L.: DECSYSTEM-10 PROLOG User's Manual, D. A. I. Occational paper no. 27, University of Edinburgh (1981).

(昭和63年7月27日受付)
(平成元年3月7日採録)

阿部 重夫（正会員）

昭和22年生．昭和45年京都大学工学部電子工学科卒業．昭和47年同大学院修士課程（電気工学科）修了．工学博士．同年(株)日立製作所日立研究所に入社．現在同所主任研究員．昭和53年～54年テキサス大学客員研究員．電力系統の解析，アレイプロセッサ，Prologプロセッサの研究開発に従事．昭和59年電気学会論文賞受賞．電気学会，ソフトウェア科学会各会員．IEEE senior member.

川端 薫（正会員）

昭和35年生．昭和58年宇都宮大学工学部情報工学科卒業．同年(株)日立製作所日立研究所入社．Prolog最適化コンパイラに関する研究に従事．

黒沢 憙一（正会員）

昭和28年生．昭和55年東北大学大学院工学研究科修士課程情報工学修了．同年(株)日立製作所入社．知識処理計算機の命令アーキテクチャ，PROLOG言語の最適コンパイラの研究に従事．人工知能マシンアーキテクチャ，並列処理に興味をもつ．現在，同社日立研究所第8部研究員．電子情報通信学会会員．