プログラミング言語 10－3
（1987. 2. 13）

内蔵型Prologプロセッサ IPPの最適コンパイル方式

阿部重夫　黒沢憲一　桐山　薫　坂東忠秋
（日立製作所　　日立研究所）

高い実効性能の実現と既存ソフトウェアとのリンケージとを主な開発課題として、汎用アーキテクチャにProlog高速処理機能を内蔵したPrologプロセッサIPPの開発を進めているが、本報告ではコンパイラの最適化方式を中心に述べる。

Warrenの命令セットは、特にappendプログラムが高速に実行できるアーキテクチャとなっている。このため実効性能を高めるためProlog命令の拡張について述べ、更に最適引数によるインデキシング、決定的な組み込み述語を渡って大域的なレジスタ割り当てを行なう最適化方式について述べる。最後にappend、q-sort、8-queenのベンチマークプログラムで性能評価を行ない、q-sort、8-queenで各々Warren方式の1.24倍、3.4倍の高速化が得られることを示す。

"Compiler Optimization for Integrated Prolog Processor IPP" (in Japanese)

by Shigeo ABE, Ken-ichi KUROSAWA, Kaoru KIRIYAMA, and Tadaaki BANDOH (Hitachi Research Laboratory, Hitachi, Ltd., 4026 Kuji, Hitachi, Ibaraki 319-12 Japan)

To realize high performance and utilization of a large amount of existing software, an integrated Prolog processor (IPP) is now being developed.

In this paper compiler optimization techniques and their performance evaluation are discussed. Based on the Prolog instruction set, which is an extension of Warren's, the Prolog compiler introduces new functions such as indexing by the optimal argument and global register assignment across determinate built-in predicates. In the quick-sort and 8-queen programs speed-up gains of 1.24 and 3.4 were obtained by the new compiler optimaization techniques.

１．はじめに

知識処理言語としてLispの他、近年Prolog[1]が注目を集めており、専用マシン及び汎用マシン上の処理系の開発が進められている。Prologは、宣言的に記述でき、従来の手続型言語に比較してプログラミング及びメインテナンスが容易であるが、次のような欠点を持っている。

（１）手続型言語と比較して実行速度が遅い。

（２）手続記述能力が比較的低い。

Warrenによって提案されたProlog命令セット[2]は、（１）の問題を解決するのに極めて適しており、この命令をベースにした専用マシン[3-6]、及び汎用マシン上の処理系[7]の開発が進められている。

（２）の問題に関しては、言語仕様を拡張して手続記述能力を高める試みと、手続型言語とリンケージする機能を設ける方向とがある。応用分野毎に大量の手続型言語で書かれたソフトウェアが存在することから、手続型言語とのリンケージの機能は、Prologにとって必須の機能である。

（１）、（２）の問題を解決するために、我々はWarrenの命令セットをベースにして内蔵型PrologプロセッサIPPの開発を進めている。[8-10]本報告では、IPPの設計方針、Prolog命令セット、最適化方式及び性能評価結果について述べる。

２．IPPの設計方針

AIの手法を実問題に適用する場合、しばしば推論と数値演算等の手続処理とを組み合わせることが必要となる。例えば計算機制御システムでは、オンラインデータに基づいてプラントの状態を計算し、その結果に基づいて推論を行なうことが必要となる。推論はPrologに適しているが、数値演算は、Prologより手続型言語の方が適している。従ってこの場合、Prologと手続型言語のリンケージが必要となり、またProlog、手続型言語のどちらとも高速に実行することが必要である。

Warrenの命令セットは、フォンノイマン型計算機で高速処理を実現するのに極めて適したアーキテクチャとなっているが、汎用マシン上でこの命令セットを実行したとき、データの型を示すタグの処理のオーバヘッドによる性能低下はまぬがれ得ない。このため次の方針に従いIPPの設計を行なった。

（１）汎用マシンに内蔵するPrologマシンアーキテクチャを開発する。

（２）ハードウェア及び最適化コンパイラの組合せで、Prologの最大性能のみならず実効性能の向上を図る。

IPPのハードウェアアーキテクチャに関しては別稿に譲り以下では、コンパイラの最適化方式を中心に述べる。

３．Prolog命令セット

Prologプログラムの実行の高速化は、次の２つによって実現される。

（１）ユニファイ可能なクローズの中から、最もユニファイしそうなクローズを選択するクローズインデキシング

（２）クローズに対応するコードの最適化

Warrenの命令セットでは、ゴール呼び出しの引数をアーギュメントレジスタと呼ぶレジスタを介して渡す方式を採っており、上記の最適化に適している。ルールクロ

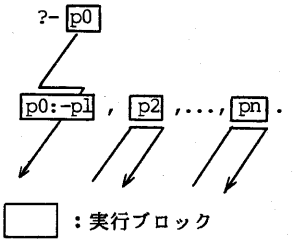


図１　Warrenの命令セットによるプログラムの実行

ーズの実行は、図１に示すようにゴール呼び出しによって不連続となる。ここでヘッドと第１ボディゴール、及び第２ボディゴール以降の各々のゴールを実行ブロックと呼ぶことにする。このときレジスタの使用は次の２つに限定される。

（１）呼び出された述語に引数の値を渡す。

（２）実行ブロックの中でアクセスされる変数の値を保持する。

もしクローズ中の変数がある１つの実行ブロックでしか現われないとき、それらはメモリに格納する必要がなくこれを一時変数と呼び、その他の変数を大域変数と呼ぶ。

引数をレジスタを介して渡すこと、及び一時変数の導入は、コンパイラの最適化を容易にし、大幅な性能向上へとつながる。しかしながら更に一層の性能向上を実現するためには、実行ブロックの拡張が必要である。またWarrenのクローズインデキシング命令では、ゴール呼び出しの第１引数を飛び先をチェックするための引数としているが、第１引数以外でインデキシングを行なった方が良い場合がある。Warrenの命令セットは、Prologのベースとなっている１階述語論理[11]の枠組を定義したものであるが、一般のプログラムでは組込述語のゴールに現われる比率が高く、[12]cutなどのように処理系を拡張しなければ実現できないものもある。このためWarrenの命令セットに対して次の拡張及び追加を行なった。

（１）算術演算等頻繁に現われる組込述語の命令化

(2) クローズインデキシング命令を引数番号でインデキシングできるように拡張
(3) cut及びif_then_elseの命令化
(4) 固定長リストの展開の命令化

4．コンパイラによる最適化

4．1 クローズインデキシング

Warrenの命令セットには次の2つのインデキシング命令がある。
(1) ゴール呼び出しの第1引数が、変数、定数、リストあるいは構造体かによって飛び先を決めるタグインデキシング命令
(2) ゴール呼び出しの第1引数が定数、構造体のものに対して、その引数のハッシュ値により飛び先を決めるハッシング命令
(1)に対応する命令は、switch_on_term命令であり、(2)に対応するものは、switch_on_constantとswicth_on_structure命令である。

Prologプログラムを分析した結果では、第1引数以外でクローズインデキシングを行なった方が良い場合があり、またゴール呼び出し時のインデキシングを行なう引数が変数の時、クローズとのユニフィケーションはリニアサーチとなってしまう。このため次の機能を導入した。
(1) 最適な引数によるインデキシング
(2) 2つの引数によるインデキシング
最適引数の選択は、同一ヘッドの集まりからなるプロシージャのヘッドの引数に関し次の基準により行なう。
(1) タグインデキシングにより別解を全て除く引数がある場合、その引数を最適引数とする。
(2) 定数あるいは構造体の個数を最も多く含む引数を最適引数とする。
(3) (1)、(2)の条件を満足するものがないときは、第1引数を選ぶ。
もし(2)を満足する引数が2つ以上ある場合は、左から順に2つ、2引数によるインデキシングの引数として選ぶ。

クローズインデキシングの機能を強化するために、callとexecuteの命令は、引数番号を指定するオペランドを追加して、switch_on_term命令と組合せ、switch_on_term命令は廃止した。またswitch_on_constant及びswitch_on_structure命令に引数番号を指定するオペランドを追加した。

```
.....,  p(X,b).

p(aa,a).
p(bb,b).
```

(a) ソースプログラム

```
1.      put_variable X,AG1
2.      put_const    b,AG2
3.      execute      AG1,(v1,c1c,fail,fail)
4. v1 : execute      AG2,(c1,c2c,fail,fail)
5. c1c: switch_on_constant AG1,2,(aa:c11,bb:c21)
6. c2c: switch_on_constant AG2,2,( a:c11, b:c21)
7. c1 : try_me_else c2
8. c11: get_const    aa,AG1
9.      get_const     a,AG2
10.     proceed
11.c2 : trust_me_else fail
12.c21: get_const    bb,AG1
13      get_const     b,AG2
14      proceed
```

(b) コード

図2 2引数によるインデキシングの例

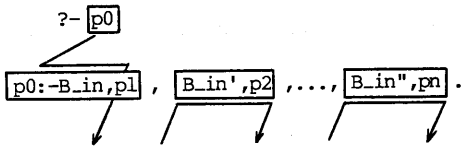
図2は、2引数によるインデキシングの例を示している。もしゴール呼び出しの第1引数が変数の時、図2(b)のステップ3で、第1引数がチェックされ、ステップ4に移り、ここで第2引数がインデキシングのためチェックされる。

3引数以上のインデキシングも容易に実現できるが、文献12)によれば、引数の平均の個数は2から3の間である。従って2引数によるインデキシングで通常の場合十分であると考えられる。

4．2 クローズコードの最適化

4．2．1 実行ブロックの拡張

文献12)によれば、組込述語のルール中のゴールに占める割り合いは、平均して50%である。しかもそれらの多くは決定的な組込述語である。従って頻繁に使われる組込述語を命令化すれば、図3に示すように実行ブロックを拡大することができる。このとき一時変数の定義は



B_in, B_in', B_in"：決定的な組込述語

図3 実行ブロックの拡張

自然に拡張されて、組込述語を渡たるレジスタ割りあてが可能となる。コンパイラの最適化をより効果的とするために、組込述語命令のオペランドは任意の一時及び大域変数を指定できるようにした。

４．２．２　最適化の考え方

クローズの最適化は、各々の実行ブロックに限定される。Warrenの命令セットでは、ゴール呼び出しの引数はレジスタを介して渡される。従ってルールにおけるヘッドと第１ゴールの実行は、ゴール呼び出しの引数を保持したレジスタから、第１ボディゴールの呼び出しのための引数を保持するレジスタへのデータフローと見ることができる。文献８)では、レジスタ競合による命令ステップ数の増加を最小にする最適化手法が述べられているが、本報告では、４．２．１項で定義された実行ブロックに対して文献８)の手法を拡張することにする。

組込述語命令によるレジスタ競合をさけるため、これらの命令では、任意のアーギュメントレジスタ及び大域変数をオペランドとして定義できる。これにより文献８)の最適化手法は、組込述語命令がある時に自然に拡張することができる。図３のヘッドp_0、組込述語B_in、ゴールp_1を含む実行ブロックのコード生成を考える。(ファクト、あるいはp_1より後のゴールのコード生成は、この特別な場合である。) この実行ブロックの実行は、ヘッドp_0の引数を保持するレジスタから、ゴールp_1の引数を保持するレジスタへのデータフローと見ることができる。データフローは、連結部分グラフの集合として定義され、各々の連結部分グラフの順序は、各々の連結部分グラフの実行においてレジスタ競合が発生しないように決める。その後連結部分グラフ間で、組込述語命令の展開順序を決め、コード生成を行なう。

４．２．３　有向グラフ化

Warrenの命令セットでは、リスト、構造体の要素は、連続した命令に展開する必要がある。この制約を考慮して、ヘッドp_0、ゴールp_1に現われる変数を次のように定義する。

(１) もし変数Ｘ、Ｙが同一のリスト、構造体に含まれるとき、Ｘ及びＹを同一の変数の集合Giに含める。

(２) もし変数Ｘが、Ｘしか含まれないリスト、構造体に現われるとき、あるいはＸはリスト、構造体中には現われないとき、Ｘは、Ｘしか含まない集合に含まれる。

次に連結部分グラフDSGi＝Ii→OiをGiを用いて次のように定義する。

(１) もし変数Ｘ∈Giがヘッドp_0のj番目の引数に現われるとき、j∈Iiとする。

(２) もし変数Y∈Giが、ゴールp_1のk番目の引数に現われるとき、k∈Oiとする。

４．２．４　連結部分グラフの順序づけ

２つの連結部分グラフDSGiとDSGjにおいて、

$$Oj \cap Ii \neq \phi、かつ Oi \cap Ij = \phi \qquad (1)$$

が成立するとき、DSGiの実行をDSGjの実行より前に行なえばDSGiとDSGjの間でレジスタ競合は生じない。この関係を次のように表現する。

$$DSGi > DSGj \qquad (2)$$

もし任意の２つの連結部分グラフで与えられる局所的な順序関係が、大域的な順序関係の矛盾をひき起こさなければ、(２)式により定まる順序関係で実行すれば、連結部分グラフ間でのレジスタ競合がなく実行することができる。

決定された連続部分グラフの順序に従って命令を展開すれば、ユニフィケーションが成功する時は、実行ステップの最小化が図られる。しかしながら失敗するとき、実行ステップ数は最小になるとは限らない。従って連結部分グラフに対して更に次の順序付けを導入した。

(１) Oi＝φとなるDSGi

(２) 順序づけられたDSGi及びIi、Oi≠φとなるDSGi

(３) Ij＝φとなるDSGj

もしB_inの入力変数のユニフィケーションがp_0で存在する場合、B_inを実行する前に行なう必要がある。従ってB_inの順序を決めるアルゴリズムは次のようになる。S_BinをB_inに含まれる入力及び出力の変数とする。このとき、

$$S\_Bin \cap Ii \neq \phi$$

を満足する全てのDSGiを求める。B_inの順序は、求められたDSGiの中の最後のDSGiの直後とすればよい。

４．２．５　コード生成

DSGiには定数、定数リスト及び定数構造体は含まれていない。またユニフィケーションに失敗する場合に対してコードの最適化を行なう必要がある。従ってコードの生成順は次のようにした。

(１) ヘッドp_0における定数データのユニフィケーション

(２) ヘッドp_0における同一変数のユニフィケーション

(３) 順序づけられたDSGi及びB_inの展開

(４) ボディp_1における定数データのロード

コード生成における最適化では次の項目が主要な課題となる。

(１) レジスタベースの命令に展開し、メモリアクセス回数を削減する。

(2) 'is' 'unify'の組込述語において、左辺の引数が変数でかつそれが最初に現われるとき、それらの組込述語を削除する。

4．2．6 例

例1 quick sortプログラムにおける次の述語splitの展開を考える。

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

'=<'及び'!'は組込述語命令であるから、変数の集合Giは次のようになる。

G1={X, L, L1},

G2={Y},

G3={L2}.

従って連結部分グラフDSGiは次のようになる。

DSG1={1, 3}→{1, 3}

DSG2={2}→{2}

DSG3={4}→{4}

DSG1からDSG3は独立に実行できる。またDSG2とDSG3の実行は冗長となる。S_Bin={X, Y}であるから、B_inとDSG1の展開順序は次のようになる。

DSG1>B_in

生成されたコードは、図4のようになる。Warrenの最適化方式では、コードは図5のようになると思われる。新方式により命令ステップ数は1/2に削減できる。

```
1. get_list AG1
2. unify_variable AG5
3. unify_variable AG1
4. get_list AG3
5. unify_value AG5
6. unify_variable AG3
7. less_equal AG5,AG2
8. first_cut
9. execute AG1,split
```

図4 splitに対するコード

```
 1. allocate
 2. markcut
 3. get_list AG1
 4. unify_variable AG1
 5. unify_variable L
 6. get_variable Y,AG2
 7. get_list AG3
 8. unify_value AG1
 9. unify_variable L1
10. get_variable L2,AG4
11. less_equal AG1,AG2
12. cut 4
13. put_value L,AG1
14. put_value Y,AG2
15. put_value L1,AG3
16. put_value L2,AG4
17. deallocate
18. execute AG1,split
```

図5 Warrenの最適化に基づくsplitのコード

例2 8-queenにおける次のルールを考える。

```
generate(N,[N|L]):-N>0, N1 is N-1,
                    generate(N1,L).
```

'>', 'is'及び'-'は、組込述語命令であるから、変数の集合Giは次のようになる。

G1={N, L}

G2={N1}

従ってDSGiは、次のようになる。

DSG1={1, 2}→{2},

DSG2= ϕ →{1}.

DSGiの順序は

DSG1>DSG2

となる。S_Bin={N,N1}であるから、順序は

DSG1>B_in>DSG2

となる。'is'の左辺の変数N1は、'is'で最初に出現する変数であるから、'is'を省略することができる。またB_inが実行されるときに、N1はレジスタにロードされるため、DSG2は冗長となる。従ってこのとき生成されるコードは図6のようになる。Warrenの最適化に従ったときのコードは、図7のようになると思われる。従ってこのと

きもステップ数は1/2以上削減される。

```
1. get_list AG2
2. unify_value AG1
3. unify_variable AG2
4. greater_than AG1,0
5. subtract AG1,1,AG1
6. execute AG1,generate
```

図６　generateに対するコード

```
1. allocate
2. get_variable N,AG1
3. get_list AG2
4. unify_value AG1
5. unify_variable L
6. greater_than AG1,0
7. put_variable N1,AG1
8. put_value N,AG2
9. subtract AG2,1,AG3
10. is AG1,AG3
11. put_unsafe_value N1,AG1
12. put_value L,AG2
13. deallocate
14. execute AG1,generate
```

図７　Warrenの最適化に基づいたgenerateのコード

５．性能評価

IPPのマイクロプログラムをシミュレートするシミュレータを作成し、append、q-sort、8-queen[13]で最適化方式の評価を行なった。appendに対するコードは図８のようになる。

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

(a)　ソースプログラム

```
1. c1 :try_me_else l1
2. c2 :get_nil AG1
3.     get_value AG2,AG3
4.     proceed
5. l1 :trust_me_else fail
6. l2 :get_list AG1
7.     unify_variable AG4
8.     unify_variable AG1
9.     get_list AG3
10.    unify_value AG4
11.    unify_variable AG3
12.    execute AG1,(c1,c2,l2,fail)
```

(b)　コード

図８　appendプログラム

この場合は、Warren方式と新方式では、同一のコードとなる。３つのベンチマークプログラムに対する評価結果を表１に示す。appendでは性能向上はないが、q-sort

表１　最適化による性能向上

プログラム	方式１	方式２	方式３
append	1	1	1
q-sort	1	1	1.24
8-queen	1	2.1	3.4

方式１：Warrenの最適化
方式２：最適引数によるインデキシング
方式３：最適引数によるインデキシングとクローズコードの最適化

では、クローズコードの最適化により1.24倍高速化され、また8-queenでは、全体で3.4倍の高速化が達成できた。

６．おわりに

汎用アーキテクチャにProlog高速処理機構を内蔵したPrologプロセッサIPPの開発を進めているが、本報告では、コンパイラの最適化方式を中心に述べた。

Warrenの命令セットはPrologの１階述語論理の枠組を定義し、appendの高速処理に極めて適したアーキテクチャとなっている。このためappendのみでなく一般プログラムの高速化を図るために、Prolog命令セットの拡張を行なった。更に最適引数によるインデキシング、２引数によるインデキシング、及び決定的な組込述語を渡たる大域的なレジスタ割り当ての最適化方式を導入した。

ベンチマークプログラムによる性能評価では、appendではWarren方式と同じであるが、q-sort及び8-queenでは、各々新方式で1.24、及び3.4倍の高速化を実現できた。

参考文献

１）　W.F.Clocksin and C.S.Mellish, “Programming in Prolog,” Springer-Verlag, New York, 1981.

２）　D.H.Warren, “An Abstract Prolog Instruction Set,”Technical Note 309, Artificial Intelligence Center, SRI International, October 1983.

３）　R.Nakazaki et al.,“Design of a High-speed Prolog Machine (HPM),” Proceedings of the 12th International Symposium on Computer Architecture, June 1985, pp191-197.

4) T.P.Dobry et al., "Performance Studies of a Prolog Machine Architecture," ibid., pp180-190.
5) A.M.Despain, "A High Performance Prolog Co-processor," Proceedings of WESCON 85, 1985, no18/2.
6) 中島他, "マルチPSI要素プロセッサPSI-Ⅱのアーキテクチャ," 第33回情報処理全国大会, 昭和61年10月, pp149-150.
7) H.Komatsu et al.,"An Optimizing Prolog Compiler," Proceedings of the Logic Programming Conference 86, Tokyo, June 1986, pp143-149.
8) S.Abe et al., "A New Optimization technique for a Prolog Compiler," Proceedings of Compcon 86 Spring, San Francisco, March 1986, pp241-245.
9) 山口他, "Prolog最適化コンパイラの開発(Ⅰ)ー(Ⅳ)," 情報処理第32回全国大会, 昭和61年3月, pp375-382.
10) 桐山他, "Prolog最適化コンパイラの開発(Ⅴ)ー(Ⅷ)," 情報処理第33回全国大会, 昭和61年10月, pp491-496.
11) C-L Chang, R.C-T Lee, "Symbolic Logic and Mechanical Theorem Proving," Academic Press, New York, 1973.
12) 尾内他, "逐次型Prologプログラムの解析," Proceedings of the Logic Programming Conference 84, Tokyo, 1984, paper no.9.5.
13) 奥乃, "第3回LISPコンテストおよび第1回PROLOGコンテストの課題案," 情報処理学会記号処理研究会28-4, 昭和59年6月.