記 号 処 理　37－3
（１９８６．６．２０）

複数のアーキテクチャをターゲットにした高速Ｐｒｏｌｏｇコンパイラ

浅川康夫 田村直之 小松秀昭 黒川利明

（日本アイ・ビー・エム株式会社サイエンス・インスティチュート）

本論文ではIBM System/370及びIBM RT-PCをターゲットにしたPrologコンパイラの試作について報告する。

述語の使い方をより詳細にプログラマが記述できるようにするために、新たにタイプ宣言及びnotrail宣言の2つの宣言を導入した。コンパイラはPrologを中間言語に落とし、プログラマによって与えられたこれらの情報をもとに最適化を行い、オブジェクト・コードとして手続き型高級言語のプログラムを出力する。最終的には手続き型高級言語のコンパイラによってSystem/370とRT-PCの両方に対して効率のよいコードが得られる。

実測の結果、3090上で1MEGA LIPS、新しく導入した宣言をつければ1.4MEGA LIPSが得られている。 RT-PCではそれぞれ56KLIPSと87KLIPSが得られている。

A Fast Prolog Compiler Targeting for Multiple Architectures

Yasuo Asakawa, Naoyuki Tamura, Hideaki Komatsu, Toshiaki Kurokawa

(Science Institute, IBM Japan, Ltd.)

In this paper, we report on our experiment on Prolog compiler technology.

Type and notrail declarations are introduced for programmers to be able to specify the usage of predicates. Our compiler uses such information for optimization and generates programs in high-level procedural language as object code. Compiler for the high-level language enables us to get efficent code for both System/370 and RT-PC.

The generated code so far attained is so efficient to gain 1 MEGA LIPS on IBM 3090 and 1.4 MEGA LIPS with some newly introduced declarations. As for IBM RT-PC, 56KLIPS and 87KLIPS, respectively.

1. はじめに

近年Warrenによって考案された抽象マシン[1]に基づいたコンパイラの研究が盛んである。それらは節のインデキシングにおける工夫[2]、レジスタ割当ての最適化[2],[3]、冗長な代入の除去[1],[4]、組み込み述語の扱い方[2],[4]、変数のクラス分け[1],[2],[4]、モード宣言の拡張[2],[5],[6],[7]、モードや決定性の推論[5]といったものに分類できる。しかし、これらの多くは抽象マシン命令のレベルでの最適化であり、最終的に実行される計算機を考えた上での最適化ではない。また、これらの最適化のすべてがPrologに限った話というわけではなく、レジスタ割当てや冗長な代入の問題は、従来の手続き型言語のコンパイラにおいて既に研究されているテーマである。 我々は、予備的な実験によって、汎用計算機上ではタグの値による分岐やトレイルに関する処理が重いことを観察した。また、PL.8と呼ばれる手続き型言語のコンパイラ[8]の最適化処理は、レジスタ割当てや冗長な代入の除去といった抽象マシン命令レベルの最適化の多くの場合を処理できることも分かった。そこで我々は、基本的にはWarrenの抽象マシンを採用しながらも、

(1) データタイプ及びnotrail宣言の導入
(2) オブジェクト言語にPL.8を使用
(3) コンパイラを3つ(PL.8コンパイラを入れれば4つ)のフェイズに分け、各フェイズにおける最適化

といった方法をとることによって、ホスト(IBM System/370)及びワークステーション(IBM RT-PC,文献[9])をターゲットとした高速なPrologコンパイラを試作した[14],[15],[16]。 以下、本論文では、新たに導入された宣言、コンパイルの方法、及び評価について述べる。

2. 拡張Ｐｒｏｌｏｇ

DEC-10 Prolog[10]で導入されたモード宣言はPrologの述語の使われ方をコンパイラに知らせる唯一の方法として知られているが、我々は新たにタイプ宣言とnotrial宣言の2つの宣言を導入した。

データタイプは、実際にはusage宣言の中でモードと合わせて宣言される。述語ごとに補助的に付けられるusage宣言では、その述語の各引数がどういうタイプのデータを入力としてもらうか、あるいは出力として出すかが宣言される。宣言できるタイプは、atom、integer、list、nil、structure、variableといった基本的なデータタイプ及びそれらの任意の組合せに制限している。空リスト(nil)の扱いはリスト処理の最適化を考えた場合に重要なので、独立したデータタイプとした。

notrail宣言は決定的にしか使われない述語に対して、トレイルの処理すら必要ないことを宣言するものである。通常、usage宣言と合わせて、関数的な述語に付けられる。 例えば、2つのリスト(空リストかもしれない)を入力としてもらい、それらを結合したリストを返す述語appendは図1のように定義できる。

```
<- usage append(in:(list+nil),in:(list+nil),out:(list+nil)).
<- notrail append(*,*,*).
append({},L,L).
append({X|L1},L2,{X|L3}) <- append(L1,L2,L3).
```

図1　宣言付append

これは、appendが呼ばれた時、第1及び第2引数はlist又はnilで、第3引数は変数であり、appendは必ず、そして1度だけ成功して、第3引数にはlist又はnilが結合されることをあらわしている。

データタイプの導入は、LISPなども含めて殆どの高級言語において行われているという意味においても自然である。一般に、データタイプの導入は

(1) 信頼性の向上

(2) 効率のよいオブジェクトの生成

といったメリットをもたらすが、Prologにおいては特に効率のよいオブジェクトを生成する上で極めて重要な役割を果たす。(1)の意味ではユーザー定義可能なデータタイプの導入も重要であるが、(2)の意味でのメリットがあまり期待できそうもないので、今回はあらかじめ用意された基本データタイプ及びその組合せに限定した。

データタイプとは異なり、notrail宣言はPrologに特有のものである。一般に、変数に束縛が起きた時には、将来バックトラックが起きた時に再び未束縛な状態に戻すために、その変数を覚えておかなければならない。通常、トレイル・スタックという特別なスタックにその変数のアドレスを積むという操作で実現される。実際には、無条件にスタックに積んでしまう方法と、その変数がいつ作られたものかを調べ、バックトラックが起きたとしても生きている変数だけをスタックに積む方法とがある。前者は、後者に比べてトレイル処理は軽いものの、トレイル・スタックの伸びが速い、バックトラックで既に必要でなくなった変数に対して再初期化をしてしまう場合があるといった欠点を持つ。いずれにしても、専用計算機[11]ではそのコストが問題にならないトレイル処理も、特別なハードウェアを持たない汎用計算機においてはかなり重い処理になる。Warrenのコンパイルの方法[1]では、変数の最初の出現とのユニフィケーションは単なる代入ですましトレイル処理は必要でなくすることができるが、appendの第3引数のように、出力用の変数に結合が起きる場合には、最終的に積むか積まないかは別にしても何らかのトレイル処理はさけられない。そのトレイル処理を取り除こうというのがnotrail宣言を導入した意図である。

notrail宣言された述語を使うには多少の注意が必要である。1つの節内で非決定的に実行が進んでいるところからはnotrail宣言された述語を呼ぶことはできない。しかし、そういう状況はプログラマはある程度分かっているし、その可能性のある所をコンパイラが見付けて対処することもできる。

我々が、導入したタイプ宣言(usage宣言)やnotrail宣言は、DEC-10 Prologにおけるモード宣言と同様にあくまでも補助的な宣言であり、書いても書かなくてもよい。効率を上げたい時に、使い方が決まっている述語に付けられるものである。今までは効率のよいプログラムを書こうとした時に、プログラムの性質を記述する方法がなかった。これらはそういった状況を改善すると思われる。

3. コンパイラの概要

System/370及びRT-PCに対して最適化されたコードを生成するために、WILと呼ばれる中間言語を導入し、コンパイラを、

フェイズ1: PrologからWILへのコンパイル
フェイズ2: WILレベルでの最適化
フェイズ3: WILからPL.8プログラムの生成

の3つのフェイズに分けた。

WILはWarrenの抽象マシン命令をベースにした中間言語であり、フェイズ2における最適化のためにタイプなどの補足的な情報を付加できるように変更してある。また、Warrenの抽象マシン命令は、最適化を行うにはレベルが高過ぎるという理由から、より低レベルの命令も導入された。例えば、"get_list A1"という命令は、

- レジスタA1のタグをテストする
- 必要ならポインタをたどる
- 未束縛な変数ならリスト・セルを作り、実行モードをwriteにセットする
- 必要ならトレイル・スタックに変数のアドレスを積む
- リストならリスト・セルのアドレスをSレジスタ、実行モードをreadにセットする
- さもなくば失敗とする

という操作を含んでいるが、WILではこれらの個々の操作に対応する命令が存在する。

3.1 フェイズ1

フェイズ1においては、Prologソースプログラムから WILへのコンパイルが行われる。このレベルでは、Prologに関する知識を使わなければできない最適化が行われる。 終端呼び出しの最適化もその1つであるが、それ以外にも

- ○ 入力モードの引数から優先的に統一化を行う
- ○ タイプ、モードの情報を使い"unsafe"な変数の数を減らす
- ○ タイプ情報を使って最適な組み込み述語を選択する
- ○ 処理の軽い組み込み述語を一般の述語呼び出しとは区別することにより、
  変数のクラス分けの最適化を行う

などを行っている。ここではレジスタは無限にあることを仮定している。Warrenの抽象マシンでのレジスタ割当てや冗長なget/put命令の除去といった問題は、普通このレベルで解決されるが、我々のコンパイラではPL.8コンパイラによって解決されるので、フェイズ1では行っていない。また、フェイズ2で必要なモードやタイプの情報はここで付加される。図2は図1のappendに対して出力されるWILの一部であり、インデキシングの部分と2番目の節の第1番目の引数に相当する部分である。

```
asa3:
   assertion(type(a(1)) = list + {} + ref( ^ undef));
   select type(a(1)) of {
     ref( ^ undef) -> {
        deref(a(1));
        goto(asa3) };
     {} -> goto(asa1);
     list -> goto(asa2);
     otherwise -> failure};
   ..............................
asa2:
   get_list(a(1))
        where trail = no &
              type(a(1)) = list + {} + ref( ^ undef) &
              deref = no;
   unify_variable(x(1),0);
   unify_variable(x(2),1);
   ..............................
```

図2　フェイズ1の出力例 - 最適化前のWIL

assertion文は非実行文でselect文に対する前条件を示すものであり、asa3においてa1はlist、nil、又はそれらへの参照であることを示している。select文はWarrenの抽象マシン命令ではswitch_on_termやswitch_on_constantに相当するものであり、最適化がしやすいように1つにまとめてある。また、デレファレンスのループもこの中に陽に書かれている。 get_listにおいてもwhere以下に前条件が示されている。"deref=no"は既にデレファレンスがすんでいることを示す。 unify命令は、Sレジスタをインクリメントするのをやめ、オフセットを指定するように変更されている。

3.2 フェイズ2

フェイズ2では、最適化に先立って、まずフェイズ1の出力をより低いレベルのWILコードに落とす。このコードは、まずコントロール・フロー・グラフに展開される。最適化は、このグラフに対してトレース・書き換えを繰り返すことによって行われる。トレースの目的は、既にわかっているタイプやモードのもとで到達可能な最大の部分グラフを

求めることである。そのために、各ノードに対応する命令が、与えられたタイプやモードのもとでどのように振舞うかを命令の意味定義表を参照しながら推論すると同時に、その命令の実行後のタイプやモードに関する情報を生成する。

書き換えは、次の点について行われる。

(1) マークされなかったノード、すなわち実行され得ない命令を削除する
(2) 分岐命令の前条件から考えて、選択され得ない分岐先のチェックを削除する
(3) 分岐命令に入ってくるパスに付随している条件のもとでは常に決まった所にしか分岐しない場合、その分岐先にパスを付け換える

このようなグラフの書き換えによって、図2は図3のように最適化される。

```
asa3 :
    case type(a(1)) of {
        ref( ^ undef) ->
            goto(tn82);
        {} ->
            goto(tn77);
        list ->
             goto(tn62)
    };
tn82 :
    deref(a(1));
    goto(asa3);
    ...................
tn62 :
    get_list_r(a(1));
    setmode(read);
    unify_variable_r(x(1),0);
    unify_variable_r(x(2),1);
    ...................
```

図3　フェイズ2の出力例 - 最適化されたWIL

select文はlist、nil又はそれらへの参照しか渡されないという条件のもとにcase文に書き換えられている。case文での、それぞれのエントリーはガーディッド・コマンドのように排他的であり、実際の順番はフェイズ3で決められる。get_listやunify_variableといった命令はreadモードだけを処理する命令に置き換えられた。setmodeはすでに冗長な命令であり、最終的にはPL.8コンパイラによって取り除かれる。

3.3 フェイズ3

フェイズ2によって生成されるコードは、マシンに依存する情報はまったく使われずに作られた。フェイズ3では、このコードをPL.8のプログラムに変換するわけであるが、マシンに依存した情報を使った最適化も同時に行われる。すなわち、

(1) タグのデザイン
(2) プリミティブ・オペレーション
(3) タグにマッチする分岐のサーチ順

などがマシンによるコストの違いを配慮して決められている。図4はSystem/370に対するオブジェクト・コード(PL.8)である。

```
asa3:
 select;
 when(  shiftr(a1,28)=4 )
 goto tn62;
 when( a1= ('30000000'xb|('0FFFFFFF'xb &0)) )
 goto tn77;
 otherwise
 goto tn82;
 end;
 tn82:

  /*** DEREF(a1)***/
  a1 =
  ptr(
  a1
  ,memory)->w;
 goto asa3;
.........................
 tn62:

  /*** G_LIST_R(a1)***/
  s = ('00000000'xb|('0FFFFFFF'xb &a1));
 mode = rmode;

  /*** U_VAR_R(x1,0)***/x1=s->strfrm.w0;

  /*** U_VAR_R(x2,1)***/x2=s->strfrm.w1;
.........................
```

図4　フェイズ3の出力例 - PL.8プログラム

このPL.8のプログラムは、この後更にPL.8コンパイラによってレジスタ割当て、分岐といった点に関して大域的に最適化が行われる。PL.8コンパイラは、現在System/370及びRT-PCに対して極めて最適化されたコードを生成することができる。PL.8コンパイラは、RISCプロセッサもそのターゲットの1つとして開発されてきたこともあり、RT-PCに対しては、特によいコードを生成できる。

4. 評価

処理系はVM/PrologとPL.8で書かれた。PROEDIT[12]及びPL.8のデバッグ環境のおかげで開発は比較的短期間で行なわれた。

今回、我々が試作したコンパイラは、あくまで実験的なものであり、汎用計算機上でいかにして高速なコンパイラが作れるか、また、いったい何処まで高速にできるのかに焦点がおかれた。したがって、実用的な処理系には本来備わっているべき実行時の種々のチェック、ガーベッジ・コレクション、カット、多くの組み込み述語は実現されていない。しかし、基本的な算術述語をサポートするだけでベンチマークテスト[12]によるテストがある程度できた。ここで報告するデータはそういう状況での測定結果である。

測定はおもに3081K(VM/CMS)及びRT-PC上で行われ、いくつかのテストについては3090上でも行われた。3081K及び3090の実行時間は仮想CPU時間で、RT-PCではシングル・ユーザー時の実応答時間である。

図5は3081K及びRT-PCで、usage宣言やnotrail宣言を全く与えない場合とできる限り与えた場合について、試作したコンパイラ及びPL.8コンパイラの最適化の効果を測定してまとめた結果である。 PL.8での最適化の効果は、いわゆる節レベルの最適化[4]や大域的なレジスタ割当て・分岐の配慮による。 PL.8コンパイラは、Warrenの抽象マシン命令レベルで考えられていたこれらの問題をある程度解決していることがわかる。一方、これらのデータは、Warrenの抽象マシン命令をそのまま汎用計算機の命令に展開しても十分な性能がでないことをも示している。より低レベルでの最適化が有効であることが分かる。

	3081K		RT-PC	
	No hint	Hints	No hint	Hints
No opt.	1	1.06	1	1.09
Prolog only	1.68	1.92	1.62	1.86
PL.8 only	1.64	1.85	1.68	1.74
Prolog&PL.8	3.30	4.47	2.80	4.35

図5 最適化の効果(appendにおける最適化・ヒントなしに対するLIPS値の相対値)

図6は新しく導入した宣言の効果を表したものである。モード宣言だけの場合に比べて、 データタイプも合わせて宣言した時の効果が大きい。またデータタイプが宣言されて最適化された時にはトレイルの処理はRT-PCにおいては全体の24%の処理をしめていた。メモリー・アクセスの遅いワークステーションにおいてはnotrail宣言によるトレイル処理の除去の効果が大きい。

	3081K		RT-PC	
	Trail	Notrail	Trail	Notrail
None	1	1.13	1	1.16
Mode	1.06	1.29	1.05	1.31
Mode&Type	1.29	1.38	1.48	1.95

図6 宣言の効果(appendにおける宣言なしに対するLIPS値の相対値)

ベンチマークテストの結果、appendに関してはRT-PCではusage宣言なしで55.9KLIPS、 usage宣言ありで87.0KLIPS、3081Kではそれぞれ611KLIPSと827KLIPというデータが得られている。また、3090ではそれぞれ1MLIPS、 1.4MLIPSが得られた。

8-QUEEN問題では、RT-PCではOSのオーバー・ヘッドこみで第1解は65ミリ秒、全解には1066ミリ秒、3081Kではそれぞれ5ミリ秒と92ミリ秒というデータが得られた。

5. 終わりに

この研究の目的は、汎用計算機上に高速なPrologコンパイラを実現するための基本的な技術を確立することにあった。 したがって、 実用を目指した処理系を作ろうとすれば、 更に多くの機能を追加しなくてはならない。 そのためには実行効率を犠牲にしなければならないかもしれない。しかし、今回のプロトタイプの作成・評価によって、我々のアプローチの正しさや可能性は確かめられた。

今後は、この成果をいかにしてシステムとしてまとめるか、タイプの扱いの整理・拡張、決定性やタイプの推論とその利用、より高速な実行モデルの提案といったことが重要なテーマになると思われる。

[参考文献]

[1] Warren, D.H.D., "An Abstract Prolog Instruction Set", SRI International Technical Note 309, October, 1983

[2] 黒沢他、"Prolog最適化コンパイラの開発(2)-(4)"、情報処理学会第32回全国大会、pp.377-382、1986

[3] 寺門・玉木、"Prologコンパイラにおける最適化技法"、情報処理学会第32回全国大会、pp.391-392、1986

[4] Park S. and DeGroot D., "Clause-Level Optimization of Abstract Prolog Instruction Set", Research Report RC11320, IBM Thomas J.Watson Research Center, 1985

[5] Mellish, C.S., "Some global optimizations for Prolog compiler", J. OF LOGIC PROGRAMMING, 1985 No.1 pp. 43-66

[6] 岸本他、"Prologコンパイラの設計と評価"、Proceedings of the Logic Programming Conference '85

[7] 竹内他、"論理型言語「LONLI」における最適化コンパイル方式の提案と評価"、情報処理学会第32回全国大会、pp. 375-386、1986

[8] Auslander, M. and Hopkins, M., "An Overview of the PL.8 Compiler", Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Volume 17, Number 6, June 1982

[9] International Business Machines Corporation, "RT Personal Computer Technology", No.SA23-1057

[10] Warren, D.H.D., "Implementing Prolog - compiling predicate logic program", Research Reports 39 & 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977

[11] Tick, E. and Warren, D.H.D., "Towards a Pipelined Prolog Processor", Proc. of 1984 International Symposium on Logic Programming, IEEE Computer Society, 1984

[12] Numao M. and Fujisaki T., "Visual Debugger for Prolog", The Second Conference on Artificial Intelligence Applications, pp.422-427, IEEE Computer Society, 1985

[13] 奥野 博、"第3回LISPコンテストと第1回PROLOGコンテストの課題案"、情報処理学会 記号処理研究会資料 28-4、1984

[14] Kurokawa, T., Tamura, N., Asakawa, Y., and Komatsu, H., "A Very Fast Prolog Compiler on Multiple Architecture", will be presented at ACM-IEEE C/S Fall Joint Computer Conference, November 1986

[15] Tamura, N., "Knowledge Based Optimization in Prolog Compiler", will be presented at ACM-IEEE C/S Fall Joint Computer Conference, November 1986

[16] Komatsu, H., Tamura, N., Asakawa, Y., and Kurokawa, T., "An Optimizing Prolog Compiler", will be presented at the Logic Programming Conference '86, 1986