1Q-6

JSI AI ワークステーション (6)
—Prolog コンパイラの最適化技法

田村直之 浅川康夫 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

1. はじめに

FORTRAN や Pascal といった言語に比較して、Prolog をコンパイルする場合、以下のような点が問題になる。

(1) 変数のタイプ
Prolog は基本的にはタイプレス言語であり、変数には任意のタイプの値を代入できる。

(2) 代入の双方向性
すべての引数は入力にも出力にも使用できる。同一の引数を、構造体の作成にも読み出しにも利用することが可能である。

Warren の抽象マシン命令セットでは、(1) の点はタグでデータ・タイプを区別することにより、(2) の点はリード／ライト・モードでユニフィケーションの方向を示すことにより解決している。

Tick と Warren の Pipelined Prolog Engine では、タグやリード／ライト・モードをテストするためのハードウェアが用意されているため、これらのテストはオーバーヘッドにならない。しかし、通常のマシンでは、実行時のテストには数命令を必要とする。したがって、通常のマシンの場合、コンパイル時にこれらのテスト命令を最適化することが速度改善につながる。

2. 低レベル命令の使用

しかしながら、Warren の命令セットの場合、これらのテストは明示的な命令としては現れておらず、最適化をおこなうには高レベルすぎる。

たとえば、get_list(a(1)) という命令 (Warren の命令セットでは get_list A1) は以下のような処理を必要とする。

(1) レジスタ a(1) のタグをテストする
(2) もし a(1) が束縛された変数なら
値を a(1) に入れて、(1) に戻る
(3) もし a(1) が束縛されていない変数なら
リストのセルを作成し
トレイルに変数のアドレスをつみ
モードをライトにする
(4) もし a(1) がリストなら
リストのアドレスをもとめ
モードをリードにする
(5) それ以外なら
失敗する

get_list をプリミティブな命令として利用している限り、a(1) が常にリストだとしても最適化することは不可能である。

そこで我々のコンパイラでは、get や unify などの命令を上記のような低レベル命令に分解し、データ・タイプやリード／ライト・モードのテストが明示的に現れるようにして、コンパイラのよる最適化を可能にした。

このような低レベル命令を中間言語として採用することにより、冗長なタイプやリード／ライト・モードのテストの削除などの最適化が可能になった。

3. 最適化

最適化は、ターゲットマシンのアーキテクチャに依存しないレベル（中間言語レベル）と、マシンに依存したレベル（コード生成レベル）の二つのレベルで行っている。

3.1 マシン非依存の最適化

マシン非依存のものとしては

(1) 冗長なタイプ・テストの削除
(2) 冗長なモード・テストの削除
(3) 絶対に実行されないコードの削除

などの最適化を行っている。これらの最適化は、中間言語のグラフ表現を利用して行なわれる。すなわち、いったん中間言語をグラフ表現に変換し、そのグラフをより効率的なグラフに最適化したのち、ふたたび中間言語表現に戻す。

また、グラフの最適化は

(1) グラフのトレース
(2) グラフの書き換え

JSI AI Workstation (6) — Optimization techniques in Prolog Compiler
Naoyuki Tamura, Yasuo Asakawa, Hideaki Komatsu, Toshiaki Kurokawa
Science Institute, IBM Japan, Ltd.

の二つのステップを繰り返すことにより行なわれる。

まず、グラフのトレースのステップでは、可能なデータ・タイプやリード／ライト・モードを推論しながらグラフ全体をトレースする。また、ユーザが宣言した情報は assertion 文として埋めこまれているから、それらの情報も有効に利用して推論を行う。

次にグラフの書き換えステップでは、推論した情報とグラフの書き換えルールを利用して、グラフの最適化を行う。

書き換えルールには以下のようなものがある。

(1) case 文中で、絶対に選ばれない選択肢があれば、それを削除する。

(2) 分岐した先の case 文で、常にただ一つの選択肢だけを選ぶのなら、その case 文を飛び越して選択肢に直接分岐するようにする。

(3) 実行されえない命令を削除する。

最適化のようすを図1のプログラムを例にとって説明する。このプログラムは append のコードの一部で、ラベル l_1 にある命令は Warren の命令セットでの switch_on_term に、ラベル l_5 から l_8 までは get_list に、ラベル l_9 は unify_variable に相当している。このプログラムは次のような過程を経て、図2のプログラムに変換される。

まず、一行目の assertion 文は、レジスタ a(1) のタイプが nil または list またはそれらが代入された変数であることを意味しているから、最初の case 文中の ref(undef) と atom+int+struct のエントリーは決して選ばれない。したがって、(1) のルールにより削除できる。次に、a(1) が list の場合、ラベル l_5 へジャンプするが、(2) のルールによりその場所での case 文をスキップして、直接ラベル l_8 への goto 命令に変更することができる。さらに、ラベル l_9 では、常にモードがリードになることが推論できるから、ライトの選択肢は削除してよい。最後に、ルール (3) により、実行されえない命令を削除する。

実際には、これらの最適化が一度に行なわれるではなく、何度かトレースと書き換えを繰り返すことにより最終的な結果が得られる。

3.2 マシン依存の最適化

マシン依存の最適化では、アーキテクチャの違いによる部分の最適化を行う。例えば、効率的なタグの取扱いは、マシン・アーキテクチャの違いによって大きく異なってくる。そこで、我々のコンパイラでは、タグの値の付け方やテストの順序などが、ターゲット・マシンごとのコストの違いなどを配慮して決められている。具体的には、タグをテストするために必要なサイクル数とか、メモリー・アクセスのサイクル数などを考慮して、最適なコードを生成している。

```
l_1 : assertion(type(a(1))=list+nil+ref(^undef));
      case type(a(1)) {
        ref(^undef)          -> goto(l_2);
        ref(undef)           -> goto(l_3);
        nil                  -> goto(l_4);
        list                 -> goto(l_5);
        atom+int+struct      -> failure
      };
l_2 : deref(a(1));
      goto(l_1);
l_3 : ..................
l_4 : ..................
l_5 : assertion(type(a(1))=list+nil+ref(^undef));
      case type(a(1)) {
        ref(^undef)          -> goto(l_6);
        ref(undef)           -> goto(l_7);
        list                 -> goto(l_8);
        nil+atom+int+struct  -> failure
      };
l_6 : deref(a(1));
      goto(l_5);
l_7 : get_list_w(a(1));
      trail(a(1));
      setmode(write);
      goto(l_9);
l_8 : get_list_r(a(1));
      setmode(read);
l_9 : case mode {
        write  -> unify_variable_w(x(1),0);
        read   -> unify_variable_r(x(1),0)
      };
```

図1　最適化前のプログラム

```
l_1 : assertion(type(a(1))=list+nil+ref(^undef));
      case type(a(1)) {
        ref(^undef)          -> goto(l_2);
        nil                  -> goto(l_4);
        list                 -> goto(l_8);
      };
l_2 : deref(a(1));
      goto(l_1);
l_4 : ..................
l_8 : get_list_r(a(1));
      setmode(read);
      unify_variable_r(x(1),0);
```

図2　最適化後のプログラム