記 号 処 理 44－1
（１９８８．１．２２）

汎用ミニコン上の最適化C-Prologコンパイラ

松本憲幸　小林茂　落合正雄　本位田真一

（株）東芝

汎用スーパーミニコン上に、高速の Prolog 実行環境を構築すべく開発された最適化C-Prologコンパイラの高速化手法に関して報告する。本コンパイラは、パイプライン制御アーキテクチャーをもつ標準的な計算機を主なターゲットとして設計されており、条件分岐やアドレス修飾によるパイプライン制御の乱れを生じにくいEAPIS(Extended Abstract Prolog Instruction Set) を中間コードとして採用している。また、リスト・構造体のパターンマッチングやバックトラッキング処理など、汎用アーキテクチャ上でPrologを実行する際に比較的負荷の大きい処理を対象とした最適化を導入して、スーパーミニコン上でインタプリタ比20～30位程度の実行速度をもつ高速コードの生成を可能としている。

"Optimizing C-Prolog Compiler for general purpose super-mini-computer"
(in Japanese)

by Noriyoshi MATSUMOTO, Shigeru KOBAYASHI, Masao OCHIAI, and Shinichi HONIDEN
(Toshiba Corporation, 1 Toshiba-cho, Fuchu-shi, TOKYO, 183 JAPAN)

The serious disadvantage of Prolog execution on general purpose computers is that the Prolog program compiled into APIS(Abstract Prolog Instruction Set, D.H.D Warren, 1983) frequently takes conditional branching, which may cause execution delay of the hardware instructions especially on advanced control system. Some of the branchings are due to unclarity of the description of Prolog program, while the others are peculiar to APIS.

In this paper, we propose the improved prolog instruction set based on APIS and the optimization technique, which minimize the conditional branches.

1．はじめに

知識情報処理の分野で使用される論理型言語Prologは、手続型言語には見られない興味深い特性を持っている。

(1) 非決定的論理に基づくプログラミング

(2) 述語間での値受渡しの多値性および双方向性

(3) 不確定リストを利用した強力なパターンマッチング

(4) バックトラッキングによる実行環境の巻戻し

しかし、Prologの最大の弱点は、実行効率の悪さであり、インタプリタ処理の場合、典型的には5MIPS の性能を持つミニコン上で5KLIPS(1秒間に5000節を証明する)程度の実行効率しか得られないため、高性能のコンパイラが要求される。

D.H.D. Warren の提案した APIS (Abstract Prolog Instruction Set) は多くのPrologコンパイラ処理系に採用されているが、もともと仮想Prologマシンを想定したものであり、汎用命令セットしか持たないミニコンやワークステーションには不利な面を持つ。APISを使用するコンパイラによりスーパーミニコンや高性能ワークステーション上で 120～150 KLIPS 程度の性能を得ることができるが、さらに高い処理性能を得るためにはAPISを効率よく実行するような特殊なアーキテクチャが必要となる。

我々は、汎用アーキテクチャ上で生じるAPIS処理に関する負荷を軽減する機能を持つ最適化コンパイラをスーパーミニコンDS6060およびG8050 上に開発し、従来方式より50～100%程度高速なProlog実行環境を構築した。本報告では、DS6060に搭載した最適化コンパイラの中間コードとして開発された EAPIS (Extended Abstract Prolog Instruction Set) 処理およびその最適化の概要と効果について述べる。

2．設計思想

最適化コンパイラによる処理の高速化は、基本的に次の３つの手法により達成する。

(1) APIS処理の中から実行時の条件分岐要素を取除く。
(2) アドレス修飾直前のレジスタ値読込みを回避する。
(3) メモリアクセスを減少させる。

(1) および(2) は、多くのミニコンやワークステーションに採用されているパイプライン方式による命令の先廻り制御を活かし、ハードウェアの最高性能を引出すことを目的としている。コンパイラが中間コードとして使用するEAPIS は、APISに比べて条件分岐を含む命令が少なく、また、EAPIS 中に含まれる条件判定のある程度をコンパイラが最適化により取り除くことを可能とするように設計されている。アドレス修飾に関する問題は、EAPIS を機械語プログラムに変換するコードジェネレータ部で大部分を解決することができる。

ところでAPISの中にはコンパイルすると１～２個の汎用命令に変換されるものが存在しており、これらの命令は汎用計算機環境でも十分に高速実行が可能である。

例 get-variable Xj Ai, put-variable Xj Ai,etc

一方でProlog専用マシンに比べてかなりの負荷を生じる命令が存在し、汎用環境でPrologを高速実行するためにはこれらの命令を効率よく実行することが必要不可欠である。

例 switch-on-term,get-list,get-structure,
get-nil,etc.

本コンパイラの最も特徴的な最適化機能は、これらの命令と対象として、一定条件下で除去あるいは汎用命令で数命令の単純な逐次的処理に変換することを可能とする。とは言っても、Prolog処理の中にはコンパイラがコンパイル時には判定できず、実行時判定とならざるを得ない要素が含まれており、これが汎用環境上で大きな負荷となる。この不定因子を極力取除くためにlocal, modeという２つの最適化宣言を導入する。処理効率が重要な問題ではない場合は、これらの宣言は不要である。宣言が与えられた時、コンパイラは宣言に基いた最適化を行うが、最適化されたコードに対して宣言に反する処理を行うことはできない。逆に、正当な宣言は実行に対して何も影響を与えないものとする。

```
:-mode / append(+,+,-).
 append([X|Y],Z,[X|W]):-append(Y,Z,W).
 append( [], Z,Z).
```

例１ 最適化宣言の使用例

例１は、append述語に対する最適化宣言の使用例であり、append述語は第１引数と第２引数が変数以外、第３

引数は変数で呼び出され、かつappend述語をローカル述語（外部からアクセス不可能、再定義もできない）とすることを表している。

最適化宣言は厳密に言えば、最適化された述語実行に一定の制限を与えることになるが多くの場合は事実上容易に宣言を付加することが可能であり、インタプリタ実行と最適化コード実行の間での不整合性は問題とはならない。

３．処理系の概要

最適化コンパイラは、DS6060 VMPおよび G8050 OS/V上にC-Prologインタプリタの組込述語として搭載されており、C-Prologで記述されたプログラムはコンパイラにより、図１に示す手順に従って、各マシンの機械語プログラムに変換され、リロケータブルなPrologオブジェクトプログラムとなる。このプログラムはC-Prologインタプリタにローディングして使用する。

図１に示されるようにコンパイル処理は基本的に３つのフェーズに分けられ、各フェーズごとに次の処理を行う。

Phase1 : C-Prolog コードをEAPIS に変換する。一時変数・パーマネント変数割付けや固定リスト・構造体のリテラル化等のクローズ単位での最適化を実施する。

Phase2 :最適化宣言をもとにしてEAPIS の再配置・変換・消去等を行う。述語を単位として、複数クローズ間に渡る最適化を実施する。

Phase3 : EAPISを機械語プログラムに変換する。ただし、EAPIS はPhase1やPhase2における最適化により与えられる情報に依有して多様な機械語列生成パターンを持つ。アドレス修飾の問題を解決するため、Phase3は複数個のEAPIS を単位として機械語生成を行う。

Phase1処理の第１段階(C-Prolog プログラムの構文解析）はC-Prologインタプリタに内蔵されており、ソースプログラムファイル中に含まれる命令（否定節）はこの構文解析の段階で実行される。これは、Prologプログラムの内容の解釈が一般には否定節の実行結果に依存することに起因する。例えば、例２のプログラムが述語fooを定義するものであるか、述語bar を定義するものであるかは、否定節に含まれる演算子宣言と解釈することなしに決定することはできない。

```
:- op(300,fy,foo).
:- op(400,yf,bar).
 foo X bar :- boo(X),baz(X).
```

例２　否定節に依存する構文

４．EAPIS

最適化コンパイラは、Prologプログラムを機械語プログラムに変換する中間過程でEAPIS という仮想的な中間コードイメージを使用している。EAPIS の基本処理は、D.H.D.WarrenのAPISに基いているが、APIS処理の中から実行時に動的に判定される要素を極力取り除いている。APISとの主な相異点としては以下のものが挙げられる。

(1) Environment をトリミングしない。

(2) 仮想パーマネント変数(Environmentの割付けなしに使用）が存在する。

(3) switch-on-termが分岐ターゲットとしてnil ラベルを持つ。

(4) get 系命令がインデキシングフラグを持つ。

以下にEAPIS の基本処理について述べる。

図１　コンパイル処理の流れ

4.1 Environment のトリミングの除去

APIS処理では、クローズ本体が複数ゴールから成時、各ゴールの実行ごとに不要となったパーマネント変数セルを解放しEnvironment のトリミングを行うが、EAPIS ではトリミングは行われず、deallocate命令実行まではEnivironmentの初期サイズがそのクローズのために確保される。

これは、スタック上に割付けられるEnvironmentおよびChoice Pointのサイズを初期割付け時に確定させることにより、後続の処理に対してEnvironment およびChoice Pointの割付け位置を決定的にすることを目的とする。APIS処理においては EnvironmentやChoice Pointの割付け位置はE(Environment Pointer)および B(Back tracking Pointer) の比較により割付け時に決定されるが、EAPIS 処理においては、これらの割付け位置は常にスタックトップポインタ(A) によって確定されており、図2、図3のように単純な処理となる。

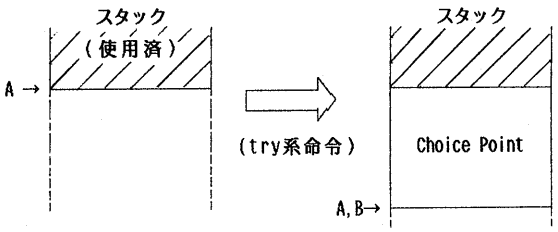

図2 EAPIS のChoice Point割付け処理

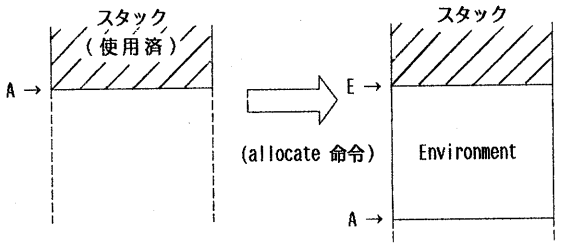

図3 EAPIS のEnvironment 割付け処理

Environment トリミング除去の利点および欠点として以下の点が挙げられる。

利点・Environment,Choice Pointの割付け処理が高速化される。
- call命令実行時のNパラメータ（必要なパーマネント変数の数）が不要となる。
- unsafe変数の減少。最終ゴール以外のパーマネント変数はすべてsafeとなる。

欠点・スタック使用量が増加する可能性がある。これは、クローズ本体のゴール数が3以上であり、ヘッド側（第1ゴール、第2ゴール、…）が決定的処理の場合に起こる。

4.2 仮想パーマネント変数

APIS処理では、allocate命令実行までEnvironment の割付け位置が確定しないため、パーマネント変数の使用以前に必ずallocate命令が実行されなければならない。一方、EAPIS 処理では割付け位置がスタックトップポインタAにより確定しているため、allocate命令実行以前であっても、ポインタEからのオフセットの代わりにポインタAからのオフセットを使用することによりパーマネント変数（を作成すべき領域）にアクセスすることが可能である。

故に、allocate命令により実際のEnvironment が作成される以前に仮想的なEnvironment を想定した処理を行うことができる。これにより、allocate命令実行をスタック使用述語呼出しの直前まで遅らせることができる。これは以下の利点を持つ。

(1) クローズヘッド部でバックトラッキングが生じる場合、allocateを実行する必要がない。
(2) allocate命令の移動（スタックを使用しない述語呼出しの領域で）によりdeallocate命令と対消滅の可能性を与える。この時、仮想Environment のみでのクローズ実行が可能となる。
(3) Environment 割付け直後のパーマネント変数使用の問題（アドレス修飾の問題）が避けられる。

EAPIS ではEnvironment とパーマネント変数は次の概念で取扱われる。

- allocate命令は、仮想Environment を実Environment に変換する。
- allocate命令実行以前はパーマネント変数は仮想変数でポインタAにより参照。

$$Vi \equiv [A+4(i+1)]$$

- allocate命令実行後はパーマネント変数は実変数でポインタEにより参照。

$$Yi \equiv [E+4(i+1)]$$

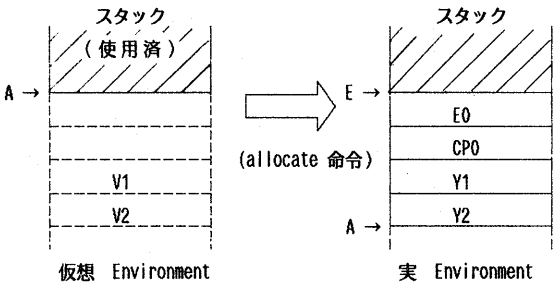

図4 EAPIS のパーマネント変数アクセス

4.3 switch-on-term命令の拡張

EAPIS のswitch-on-term命令は、変数、定数、リスト、構造体ラベルに加えてnil ラベルが追加されている。これは後に述べるget-nil 命令の最適化にも関係しているが、基本的には生成される機械語プログラムの処理効率を考慮したものである。

→ switch-on-term Ai;Lvar,Lconst,Llist,Lstruct ,Lnil

コンパイラのPhase3, すなわち機械語プログラムジェネレータはEAPIS 1 命令に対して最適化情報に依有した多様な機械語列生成パターンを持つ。nil を定数と分離することにより、リスト処理で出現頻度の高いlist & nil 処理をコンパイラが認識可能となり、効率の良いコード生成を行うことができる。逆にnil が定数群に含まれる場合、後続の処理は定数とnil の判別を行わなければならない。

また、インデキシング引数Aiがmode宣言により入力モード（＋）を宣言されている時、switch-on-term命令は、APIS処理に加えて以下の処理を行う。

- Aiの内容がリストならば、リストへのポインタをS(ストラクチャ ポインタ）にセットする。
- Aiの内容が構造体ならば、構造体へのポインタをSにセットする。
- Aiの内容が定数ならば、その定数を特定のワークレジスタ（実はS）にセットする。

これにより、後に述べるインデックス引数の最適化が可能となる。

5. 最適化

EAPIS 処理は、APIS処理に比較して実行時条件分岐やアドレス修飾の問題が生じにくいという特徴を持っているが、なお、汎用環境で実行するには大きな負荷となり得る要素が含まれている。最適化コンパイルはこれらの負荷を軽減することを目標として設計されている。
以下に本コンパイラの代表的な最適化方式について述べる。

5.1 インデックス引数の最適化

EAPIS やAPISを汎用命令列で実現することを考える時、処理効率・コードサイズともに問題となる可能性が高いものとして次の命令がある。

get-list,get-structure,get-constant,get-nil

引数モードの宣言により、処理効率・コードサイズともにかなり向上するが、これらの命令はデリファレンス処理を含むこともあり、EAPIS の中では処理効率の低い命令であると言える。

しかし、入力モードが宣言されたインデックス引数に対するこれらのEAPIS 命令に対しては 4.3で述べたswitch-term 命令の拡張機能により以下の最適化が可能となる。

インデックス最適化：Aiがインデックス引数、かつ入力モード（＋）宣言の時
- get-nil Aiを消去する。
- get-list Ai を消去する。
- get-constant Const,Ai →get-immediate-constant Constという変換を行う。(get-immediate-constant 命令は、特定のレジスタにセットされた定数と定数Const との比較を行う命令であり、デリファレンス処理は不要。)
- get-structure Functor,Ai→get-immediate-structure Functor という変換を行う。(get-immediate-structure 命令は、通常のget-stucture命令からストラクチャ ポインタSのセットまでの部分を省略した命令。)

本処理系では、ストラクチャ ポインタ(S) の値をChoice Point にセーブすることにより本最適化を代替節に適用することを可能としている。

5.2 カット除去最適化

Prologプログラム中に含まれるカット組込述語の中には、コンパイラが静的な評価によって取除くことが可能なものが存在する。これは、例えば、例３にような場合である。

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

例３　カットを含むプログラム

例３のpart述語が第１引数でインデキシングされる場合、実行ロジックに影響を与えるのは、第１クローズの第２ゴールとして現れるカットのみであり、他のすべてのカットはコンパイラの述語をスコープとした最適化により除去することができる。これは、次のようなカットを対象とする。

①try 系、retry 系命令の分岐先にならないクローズの本体第１ゴールのカット。

・例３の第２、第３クローズのカットがこれに当たる。これは、第２クローズのようにタグインデキシングによりカットが不要となる場合のほか、定数ハッシュや構造体ハッシュによってもこの種のカットが生じ得る。

②カットの伝播を考える時、実カットの伝播経路上に存在するカット。

・カットの伝播として実カット・仮想カットの２つを想定し、以下の概念で取扱う。

(1) カットは、クローズの先頭方向から終端方向へ伝播する。

(2) 実カットは次の特性を持つ。

生成点：try 系、retry 系命令のターゲットとならないクローズの先頭、カット述語存在位置。

遷移点：再帰呼出し位置で実カットに遷移する。

消滅点：Choice Point作成の可能性を持つ述語の呼出し位置で消滅する。

(3) 仮想カットは次の特性を持つ。

生成点：実カットの遷移点

消滅点：Choice Point作成の可能性を持つ述語の呼出し位置で消滅する。

(4) 述語構成クローズのすべての終端に実カットまたは、仮想カットのいずれかが到着するとき、述語内の仮想カットは、すべて実カットに遷移する。

・例３の第１クローズ終端のカットはこれに当たる。

5.3 絶対deallocate

クローズに割付けられたEnvironment は、最終ゴール呼出し前に実行されるdeallocate命令によって、可能ならば解放される。このとき、deallocate命令は現在のEnvironment 作成以降にスタック上にChoite Pointが作成されているかどうかを判定する。

しかし、 5.2②で述べた実カット伝播経路上に存在するdeallocate命令からは、この条件判定を取り除くことができ、無条件にEnvironment を解放するような絶対型のdeallocate命令で置換えすることが可能である。これは、実カット伝播経路上では、現在のEnvironment 作成以降のChoice POintはすべて消去されているためである。

6．性能評価

いくつかの簡単なPrologプログラムに関してスーパーミニコンDS6060上で実効性能測定を行った結果を表１に示す。ただし、表１は、キャッシュメモリミスヒットの負荷を含んでおり、通常コンパイラの評価に用いられる最適化コードによるキャッシュメモリ100%ヒット時のappend実行効率は 320.5KLIPS となる。

表１中のEAPIS 標準コードは、最適化宣言なしでコンパイルした時に生成されるインタプリタ完全互換コードに関する実行効率を表し、最適化コードは、最適化宣言を付加した場合の効率を表している。

qsort が他のプログラムにまらべ比べて最適化コードの実行効率が悪いのは、以下の理由による。

(1) バックトラッキングにより、実際に実行された推論過程がカウントされない。

(2) 組込述語処理を含むが、この処理が推論としてカウントされない。

(3) 最適化の対象外となっている汎用の組込述語ルーチン呼出しを含む。

項目 プログラム	最適化コード (KLIPS)	EAPIS 標準コード (KLIPS)	インタプリタ (KLIPS)	実行速度比
append (リスト長100)	284	106	6	47：18：1
nreverse	228	102	5	46：20：1
qsort	110	77	4	28：19：1

表1　オブジェクトコード実行効率

一方、appendやnreverseのように組込述語呼出しを含まず最適化コードのみにより実行される場合は、最適化を行わない場合に比べて100%程度実行速度が向上している。

7．おわりに

Prologには、他のプログラミング言語に比べて、コンパイラが静的な構文解析によって決定することが、困難であるような要素が多く含まれており、その結果としてコンパイルコード実行時の条件判定が増加することになり、これが処理効率の低下を生じる。DS6060上に搭載した最適化コンパイラは、この不確定性による負荷の大きさを示しており、逆にコンパイラに対する簡単な宣言により汎用アーキテクチャ上でも十分な高速性が得られることを証明している。

本コンパイラの高速化手法は、特定のハードウェアをターゲットとしたものではなく、基本的にはコンパイラのPhase3の書換えにより他のハードウェア上に同等の実行環境を構築することが可能である。　現在、我々はAS3000, AS4000などのワークステーション上に本コンパイラをベースとした高速Prolog実行環境を構築中であり、汎用アーキテクチャー上のProlog実行環境の評価を行っていく予定である。

8．参考文献

[1] 松本他、“C-Prologコンパイラの開発(1) ～(2) ”情報処理第33回全国大会講演論文集（Ⅰ）、487～ 490、昭和61年10月。

[2] 松本他、“C-Prologコンパイラの開発(3) ”情報処理第34回全国大会講演論文集（Ⅱ）、1025～1026、昭和62年 3月。

[3] 小林他、“C-Prologコンパイラの最適化”情報処理学会プログラミング言語研究会、10-4,昭和62年 2月。

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

[5] 奥及、“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。