

PrologマシンPEKにおける Prolog中間コードについて

宮本昌也* 和田耕一** 金田悠紀夫* 前川禎男*
* 神戸大学工学部システム工学部 ** 神戸大学大学院自然科学研究科

本論文では逐次実行型Prolog専用マシンPEKにおける高速コンパイラ用Prolog中間コードについて報告する。
中間コードの設計は、マイクロコードの実行時間の最小化を優先することを目指に行った。PEKのハードウェアの特性を十分に生かすため、ストラクチャシェアリング方式の処理を行う。D.Warrenに従い引き数のレジスタ割付けをインプリメントした上に、モード宣言、変数環境と成功情報環境の取り扱い等、PEK独自の最適化を行った。この定義を基に、コンパイラはPEKマイクロコードを生成する。
実測の結果、要素30の決定的appendプログラムで407.9KLIPSの速度を得ることができた。

PLM code on the Prolog Machine PEK

Masaya Miyamoto*, Kouichi Wada**, Yukio Kaneda*, Sadao Maekawa*

* Department of Systems Engineering, Kobe University.

** The Graduate School of Science and Technology, Kobe University
1-1 Rokkodai, Nada, Kobe, 657 Japan

In this paper, we report on our work about PLM code of the fast Prolog compiler on the sequential Prolog machine PEK.

It is speed efficiency that matters in our work. The PLM code is designed on the structure-sharing method to use the PEK hardware efficiently. It's optimizations are to put argument values on registers, to deal variable environment and success information in the special way and mode declaration.

Now we can gain 407.9KLIPS on a deterministic 30 elements append execution.

1. はじめに

近年、PrologがAI記述用言語として注目を集め、その処理系の研究が盛んである。Prologプログラムの効率のよい実行のためには、Prolog専用マシンが不可欠であり、また、それを効率よく実行するインタプリタ、コンパイラが必要である。

現在我々は、Prolog専用マシンPEKシステム(Prolog Engine of KUBE)を開発中である。このPEKシステムの高速実行には、PEKのハードウェアを効率よく利用しえるコンパイラシステムが必要である。そこで、我々は、PEKコンパイラシステムに必要なProlog中間コードを新たに開発した。

今回設計した中間コードは、主に時間的効率を追求し、メモリの使用効率等の優先度は低くした。あくまでオブジェクトコードの実行時間を最小にする、との観点から最適化手法をインプリメントした。基本的には、D.H.Warrenの中間コードに従い、

(1) 引数のレジスタ割付け。

を、インプリメントしながらも、

(2) モード宣言の導入。

(3) 変数環境のグローバルスタックへの割付け。

(4) CALL命令の多様化。

(5) ストラクチャシェアリング方式の採用。

の点において、PEK独自の最適化を導入した。本論文では、この中間コードの仕様、最適化、評価について述べる。

2. PrologマシンPEK

逐次実行型Prolog専用マシンPEKは、Prologプログラムの高速実行を目的として製作された。全体構成を図1に示す。システムはPrologプログラムの実行を行うPEKマシンと、PEKの実行管理、入出力管理を行うホストプロセッサからなる。ホストプロセッサはMC68000であり、Z-80はI/Oプロセッサである。ホストプロセッサからアクセス可能なメモリは、

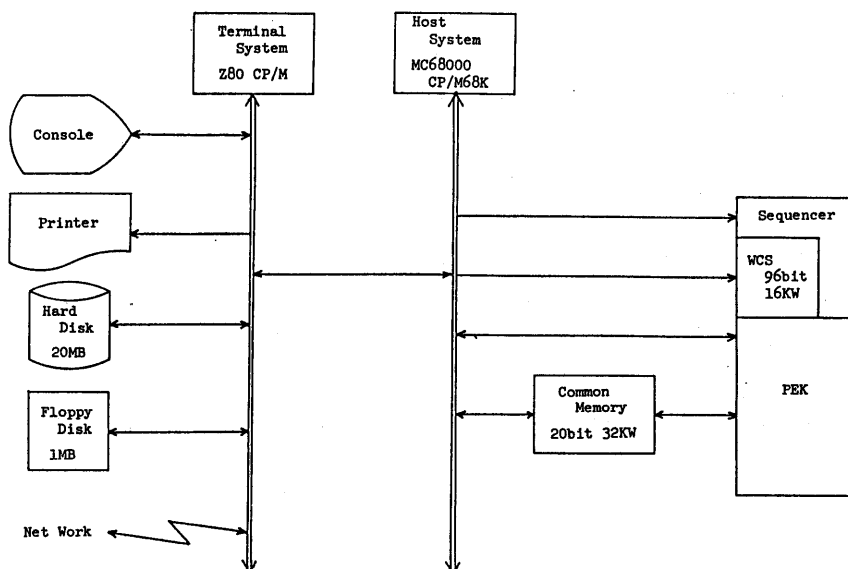


図1. PEKの全体構成

WCSとCOMMON MEMORY（以下、CM）の2種類である。WCSにはPEKマイクロコードが格納される。インタプリタ、コンパイラの実行コードはこのWCSに格納される。CMはPEK本体からもアクセス可能な共有メモリである。CMには、述語引き数の構造、スケルトンデータが格納される。図2にPEKのアーキテクチャの簡略図を示す。主な特徴は

- ・ ストラクチャシェアリング方式
- ・ メモリの専用化、分散化
- ・ 自動トレイル回路
- ・ 自動アンドゥ回路
- ・ 両方、片方のタグによる多方向分岐

である。

バスはフレーム、タグ、バリュー部の3構成で、34bits幅である。S、R、Yバスの3バス構成である。また、PEKは1ポート入力、2ポート出力の32ワード容量のレジスタファイルを持っており、プログラム実行中の各種データを保持する。

3. 中間コードの基本方針

今回設計した中間コード基本方針は

- ・ コンパイルコードの実行時間の最小化を最優先する

である。メモリの使用効率向上のためにマイクロステップが増加するのであればその導入は見合わせる。

3-1. 引き数のレジスタ割付け

D.H.Warrenに準じて、引き数内容をレジスタファイルに割り付けることにした。従来のコンパイラでは、述語引き数はCMに格納され、ユニフィケーション時にデリファレンスされ

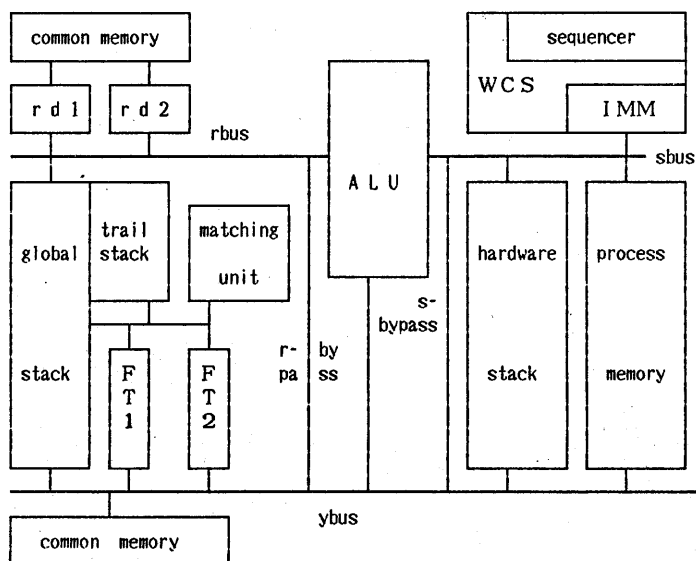


図2. PEKのアーキテクチャ

ていた。このため毎回ユニフィケーションの準備のため多くのマイクロステップが必要であった。レジスタファイルへの書き込みは、他のメモリへの書き込みと同時に進め、2ポート出力のためR、Sバスにそれぞれ別々のデータを出すことができる。このため、新たにマイクロステップを必要としない。

引き数のレジスタ割付け導入のため、中間コードセットはD.H.Warrenと似たものになっている。しかしながら、ストラクチャシェアリング方式のため、フレーム管理の命令が新たに必要である。

- `put_frame`
この命令はput命令列の先頭に置かれる。ボディ部の引き数をレジスタにセットする際、それらのグローバルスタックでのベースアドレスを与える。
- `neck`
これはヘッドの引き数を確保するために、グローバルスタックのベースアドレスの更新を行うものである。

これらの命令が今回新たに必要となったものである。

3-2. モード宣言

コンパイル時に各々の引き数の入出力情報を指定することは、効率のよいコンパイルコードを生成するのに非常に有効である。特に、インデキシングと並用した場合、第一引き数のタイプをかなり絞り込むことが出来る。

インデキシング命令は次の通りである。

- `switch_on_term( CODE, GVAR, ATOM, LIST, FANC).`

CODEは組込み述語のジャンプ先である。GVAR, ATOM, LIST, FANCは、変数、アトム、リスト、ファンクタ、各々のタイプに応じたジャンプ先である。また、モード宣言は次のように行われる。

- `:- mode(append(+,+, -)).`

＋、に宣言した引き数にはデリファレンスは行わない。たとえ未定義の変数がきたとしてもアトムと同様に扱われる。構造体に対してはそれらの引き数も入力用と考える。ユニフィケーションのパターンを表1に示す。

－、に宣言した場合、callee側変数は必ず未定義とする。ここでのユニフィケーション処理は単に変数セルへの値の代入である。

さて、次に第一引き数と他の引き数との違いについて表2に示す。

表1.ユニフィケーションのパターン

callee側引き数	ユニフィケーション操作	
	入 力 用	出 力 用
変数	caller側への代入	caller側への代入
アトム	タイプ、値のチェック	caller側への代入
リスト ファンクタ	レベル1のユニフィケーションの準備 タイプ、値のチェック	caller側への代入

表2.第一引き数と他の引き数との違い

	caller側	callee側	ユニフィケーション動作
第一 引 き 数	不定	変数	callee側への代入
	アトム	アトム	値のチェック
	リスト ファンクタ	リスト ファンクタ	レベル1のユニフィケーションの準備
他 の 引 き 数	不定	変数	callee側への代入
	アトム リスト ファンクタ	アトム	タイプチェック後、値の チェック
	アトム リスト ファンクタ	リスト ファンクタ	タイプチェック後、レベ ル1のユニフィケーシ ョンの準備

入力用第一引き数の場合、同一タイプとの組合せになるためタイプチェックの必要がなく値の一致のみ調べればよい。このため第一引き数と、他の引き数との中間コードは区別することにした。

3-3. TRO (Tail Recursion Optimization)について

TROはメモリの効率のよい利用に対して大変有用な手法である。しかしながら、プログラム実行中に頻繁にスタック管理を行わねばならず、時間的にはオーバーヘッドが生じる。そこで我々は、次のような手法によりTROを行わずにすむようにし、より簡単なスタック管理を実現した。

- ・変数を全てグローバルスタックに割り付ける

これによって、Warrenにおける変数環境がなくなることになる。変数環境がなくなると次に残るのは成功時情報となる。これに対しては我々は次のような手法を採用した。

- ・CALL命令を多様化する

```
a :- b,          b's put instructions.
    c,          first_call(b).
    d,          c's put instructions.
    e,          mid_call(c).
               d's put instructions.
               mid_call(d).
               e's put instructions.
               last_call(e).
```

図3. call命令の多様化

第一ボディ、最終ボディ、それ以外のボディに対して異なったCALL命令を生成する。図3におけるfirst_callは新たに成功時環境を生成する。Warrenにおけるallocate, callのシーケンスに対応する。mid_callは単に成功時情報のリセットを行う。last_callは、現在の成功時情報の環境を消去してジャンプする。これは、deallocate, callに相当する。

4. ユニフィケーション

ここでは入力用構造体引き数のユニフィケーションの例を用いてPEK内部のデータの流れを説明する。uin_gvar1(GVAR, SHD)命令は構造体の変数に対して作られ、caller側構造体の要素を変数セルに代入する。この命令のマイクロコードは次のようになる。

```
1: ft2 = alu( xrb(qq) = ra(gs_bottom);#GVAR )
2: gs(ft2) = alu( xrb(SHD) = rd2 ), push(ts)
3: alu( value(xrb(wk0)) = xrb(wk0) + 1 )
```

PEK内部でのデータの流れは図4に示す。それぞれのステップの説明を行う。

- 1: マイクロ命令のイミディエイトフィールドの変数番号と、レジスタファイル中の現在のフレーム値とを合わせて、レジスタファイルqqとft2に代入する。ft2はグローバルスタックのアドレッシングを行う。
- 2: CMにある要素をrd2レジスタから読みだして引き数レジスタとグローバルスタックの変数セルに書き込む。同時にトレイル処理も行う。
- 3: ワークレジスタをインクリメントして、次の要素のユニフィケーションのアドレスとする。

ステップ2で、引き数レジスタとグローバルスタックの両方に書き込みが行われている。また、トレイル処理も同時に行われている。

5. インタプリタとのインタフェース

ゴールが与えられた時、インタプリタはそれらのゴールの引き数情報をCMに割り付ける。そのため、コンパイルドコードを実行する場合、引き数をレジスタに割り付ける操作が必要になる。レジスタ (args) が、CM中の述語引き数の先頭アドレスを示している (図5)。この処理はコンパイルドコードへのジャンプ時に行う。

インタプリタへのリターン時には、必要な変数はグローバルスタックにあるため特別な処理は必要ない。

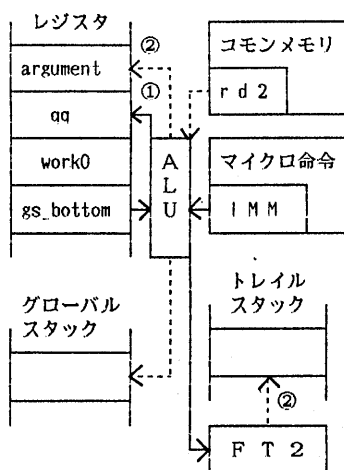


図4. PEK内部でのデータフロー

:-append([1,2,3],[],X).

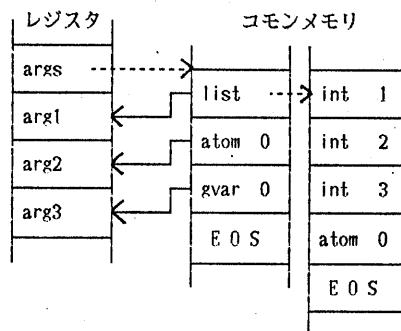


図5. インタプリタとのインタフェース

6. 中間コードセット

各中間コードの一覧表を表3に示す。ユニフィケーション命令はWarrenコードでのget命令にあたる。レベル0の命令は第一引き数か否かによって分類される。第一引き数ではswitch_on_term命令が前処理を行うため、他の命令よりコンパクトになる。switch_on_term命令がない時は、他のユニフィケーション命令を代用する。レベル1の命令は、構造体の最終引き数か、否かで分類する。最終引き数の場合、次のレベル0のユニフィケーションの準備をしなくてはならない。また、出力用のレベル1命令は必要ない。構造体が出力引き数に来る場合、レベル0命令がスケルトンを出力するのみである。

表3. 中間コード一覧

実行制御命令	put命令	ユニフィケーション命令			
switch_on_term switch_on_const switch_on_struct try_me_else retry_me_else trust_me_else _fail try retry trust first_call mid_call last_call neck proceed execute	put_frame put_var put_val put_atom put_skel put_shadow	入力用			
		uin_gvar0	uin_atom0	uin_skel0	uin_gref0
		uin_gvar0_1	uin_atom0_1	uin_skel0_1	
		uin_gvar1	uin_atom1	uin_skel1	uin_gref1
		uin_gvar1_1	uin_atom1_1	uin_skel1_1	uin_gref1_1
		出力用			
		uout_gvar0	uout_atom0	uout_skel0	
		uout_gvar0_1	uout_atom0_1	uout_skel0_1	
		uout_gref0			
		入出力用			
		ugvar0	uatom0	uskel0	ugref0
		ugvar0_1	uatom0_1	uskel0_1	ugref0_1
		ugvar1	uatom1	uskel1	ugref1
		ugvar1_1	uatom1_1	uskel1_1	ugref1_1

7. 評価

現在、コンパイラを製作中である。コンパイラはC言語で記述される。ここでの評価プログラムはハンドコンパイルによるものである。インタプリタとのレジスタの競合問題があり3引き数までプログラムが実行可能である。appendプログラムとその中間コードを図6に示す。また、その命令数と実行時間を表4に示す。1マイクロ命令のサイクルタイムは120nsと160nsである。

引き数のレジスタ割付けとモード宣言を導入したことにより、ユニフィケーションのステップが大幅に削減された。第2クローズのヘッドのユニフィケーションは9ステップで完了し、13ステップで1回の推論が完了する。実行速度は要素30の決定的appendでは407.9KLIPS、そのappendプログラムを使った要素30のreverseでは340.7KLIPSとなった。

表4 .appendプログラムの評価

命 令	命 令 数		実行時間 [nsec]	
	[1,...30]-[30]	[]	[1,...30]-[30]	[]
switch_on_term	3*30=90	3	440*30=13200	440
uin_atom0_1		3		440
uout_gref0		2		240
proceed		3		480
uin_skel0_1	1*30=30		160*30=4800	
uin_gvar1	3*30=90		480*30=14400	
uin_gvar1_1	3*30=90		480*30=14400	
uout_skel0	2*30=60		280*30=8400	
neck	1*30=30		160*30=4800	
out_frame	1*30=30		160*30=4800	
out_var	1*30=30		160*30=4800	
execute	1*30=30		160*30=4800	
t o t a l	480 + '11 = 491		74400+1600=76000	

```
:-mode(app(+,+, -)).
```

```
app([],Y,Y).
```

```
app([H: X],Y,[H: Z]) :- app(X,Y,Z).
```

```
app:   switch_on_term(i_fail, x_0001, x_0002,
                      x_0004, i_fail).
```

```
x_0001: try_me_else(x_0003).
x_0002: uin_atom0_1('$30000').
        uout_gref0(arg3, arg2).
        proceed('$00001').
```

```
x_0003: trust_me_else fail.
x_0004: uin_skel0_1('$C7500').
        uin_gvar1('$80000',wk3).
        uin_gvar1_1('$80001',arg1).
        uout_skel0('$C7503',arg3).
        neck('$00004').
        out_frame.
        put_var('$80003',arg3).
        execute(app).
```

```
backtrack.
```

図6 .appendプログラムとその中間コード展開

8. 結論

今回の中間コードはPEKハードウェアを有効に利用し、かなりコンパクトにまとめることができた。とくにPEKのハードウェアが寄与したと思われる点は、次の通りである。

メモリへの書き込みと、レジスタへの書き込みが同一のマイクロ命令で可能である。引き数のレジスタ割付けの導入によるマイクロ命令の増加をおさえ、デリファレンス処理ステップの減少分が相殺されなかった。

タグによる多方向分岐が可能である。コンパイラにおいては、caller側の引き数のタイプのみによる分岐が多く現れる。PEKではタグのセットと多方向ジャンプ動作は2マイクロ命令で行われる。これにより、インデキシングのインプリメントに必要なマイクロ命令を抑えることが出来た。

トレイル処理が他の動作と並行して行える。これによって、全ての述語がバックトラックを行える。

ジャンプ動作を他の動作と並行して行える。ジャンプ動作はコンパイルコード中に数多く出現する。それらを他のマイクロ命令中に組み込めることは、マイクロコードのコンパクト化に大きく貢献した。

9. 終わりに

今回の研究の狙いは、PEK上でのPrologプログラムの実行速度の最高値を求めることにあった。今回のプロトタイプ之作製、評価によってPEKの基本的なPrologプログラムの実行能力は、十分実証されたと考える。今後、コンパイラにおけるより大規模なプログラムの実行時の問題として、ストラクチャシェアリング方式におけるユニフィケーションコードの最適化、引き数レジスタの数の制限をいかにして外すか、等の研究が必要と思われる。

10. 参考文献

- [Gabriel, 85] Gabriel, J., Lindholm, T., Lusk, E.L., and Overbeek, R.A.,
"A Tutorial on the Warren Abstract Machine". Mathematics and
Computer Science Division, Argonne National Laboratory
- [kaneda, 85] 金田悠紀夫, 田村直之, 和田耕一, 小畑正實, 前川禎男, "シーケンシャル実
行型PrologマシンPEK", 情報処理学会論文誌, Vol.26, No.5, Sep.1985
- [kaneda, 85] 金田悠紀夫, 田村直之, 和田耕一, 松田秀雄, 小林久和, 綾部雅之, 前川禎男
"PrologマシンPEK上のインタプリタの動特性", 情報処理学会第30回全国
大会, 7C-5, 1985
- [Warren, 77] Warren, D.H.D., "Implementing Prolog - compiling predicate logic
program", D.A.I. Research Report No.33
- [Warren, 83] Warren, D.H.D., "An Abstract Prolog Instruction Set", SRI
International Technical Note 309, October, 1983