ICOT Technical Report: TR-118

TR-118

Prolog コンパイラの設計と評価

岸本光弘, 篠木　剛, 木村康則, 服部　彰
(富士通)

June, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Ｐｒｏｌｏｇコンパイラの設計と評価

岸本光弘，猿木　剛，木村啓則，服部　彰
富士通株式会社

ABSTRACT

This paper discusses a Prolog compiler for the FACOM α, a symbolic data processing machine. The compiler includes several optimization algorithms, such as separated predicate frames, extended mode declaration, and fast goal invocation. Compiled programs run at 30 to 40 KLIPS.

1．はじめに

記号処理専用マシンＦＡＣＯＭα［富本83］上で動作するPrologコンパイラを作成した。本稿では、コンパイラの設計方針，採用した高速化技法，実行結果及びその評価について報告する。

作成したProlog処理系は、DEC-10 Prolog に準拠しており、シンタックスや組込述語は一部を除いてDEC-10 Prolog と同一である［War 77］［富本84］。Prolog処理系はLISP処理系と共存しており、PROLOGというLISP関数を実行することによりインタプリタが起動される。

PrologのインタプリタやＧＣ（ガーベジ・コレクタ）はマイクロプログラムで実現されている。ＧＣはLISP処理系，Prolog処理系で共通であり、タグを用いてオブジェクトの切り分けを行っている。Prologの入出力ルーチンや組込述語の大部分はLISPで記述されており、一部がPrologで書かれている。ほか本稿で述べるコンパイラもLISPによって記述されている。

処理系はインタプリタ実行で約12 KLIPSの速度を得ているが、実用的な処理系とするためにはさらに数倍の高速化が求められている。作成したコンパイラにより述語の実行速度は30～40 KLIPSまで向上する。

本稿の２節では作成したコンパイラとマシン命令の概略を述べる。高速化に大きく寄与しているモード宣言の拡張と高速なゴール呼出法については３節で説明する。４節ではコンパイルした述語の動作特性の測定結果を示し、評価結果を５節で述べる。

2．コンパイラの設計

2．1　データ構造

コンパイルされた述語は以下に示すように、基本的にインタプリタと同一のデータ構造を用いる。

(1)　構造共有法である。データはポインタ３Byte，タグ１Byteの４Byte長であり、モレキュールは構造体と環境の８Byteからなる。

(2)　構造体はリスト（コンスセル）とスケルトン（ＬＩＳＰのベクタ型）を用いる。さらに変数を含まない構造体用に定数リスト，定数スケルトンという別のタイプを用意した。

(3)　３本のスタックを用いて推論を行う。ローカル・スタックはハードウェア・スタックを用いて実現する。グローバル・スタック，トレール・スタックは

主記憶上に配置しＧＣによるコンパクションの対象となる。

(4) Prologで用いているデータタイプの一覧表を Table 1 に示す。

2．2 制御フレームの2分化

コンパイルされた述語は、実引数をスタック上に用意して述語呼出を行うインタフェースになっている。これはＦＡＣＯＭαがスタックマシンであり、マシン命令で扱かえるレジスタが存在しないからである。実引数を用意したのち述語呼出を行うのはWarrenの構造体複写法の実現と同じであるが以下の点が異なっている〔War 83〕〔Tic 84〕。

(1) 引数をレジスタ渡しではなくスタック渡しとしている。従ってコンパイラによる複雑なレジスタ割当が不要となる。終端再帰呼出の際のunsafe変数のグローバライズが不要になり、ローカルスタック上でのポインタの張替えで終端再帰が行なえる。ただし後述する述語内ループ化ではunsafe変数を考慮しなければならない〔War 80〕。

(2) 実引数をデレファレンス処理したのち述語に渡している。コンパイル時にデレファレンス処理の必要／不要の検出が行えるので、コンパイラはデレファレンス処理を省略したコードを生成する。従来の必要になるまでデレファレンス処理をしない場合と比べ、デレファレンス処理の少ない推論動作が実現できる。

ＦＡＣＯＭαはスタックをフレーム単位でアクセスするスタックマシンである。しかしPrologは実行時にＲＣＰフレーム（別候補を持ちバックトラックする最新のフレーム）の有／無が決る。フレームの長さをコンパイル時に計算できないのでフレームベースのアクセス機構は使えない。そこで述語のフレームを2つに分離することで解決している（Fig. 1）。

① 制御，引数，変数フレーム

コンパイラが大きさを計算できるフレームでフレーム・ポインタを基準にアクセスする。

② 述語呼出用（作業）フレーム

節本体のゴール呼出で用いる領域。①との間にＲＣＰフレームが存在することがあるので、フレーム・ポインタ相対ではなくスタックトップ相対でアクセスする。

制御フレームの内容は、ほぼインタプリタと共通であるが、幾つかのフレームエントリをコンパイルした述語の高速化のため異なる目的で使用している。インタプリタとコンパイラの制御情報の一覧を TABLE 2 に示す。

2．3 ユニフィケーション法

コンパイルした述語による実行では、節の本体部分に対応したマシン命令が実引数

0	7 bit	
3	0	1
0	FIX-SHORT	RESERVED
1	LOCAL-VAR	LOCAL-REF
2	GLOBAL-VAR	GLOBAL-REF
3	FLOAT	←
4	STREAM	←
5	LISP-REFERENCE	←
6	VECTOR	CONST-VECTOR
7	STRING	K-STRING
8	CODE-PIECE	←
9	SYMBOL	←
A	CONS	CONST-CONS
B	MSUBR	GPP
C	MRSUBR	TRS
D	SPECIAL	GPH
E	VECTOR-HEAD	MOLECULE
F	UNDEFINED	←

Tag-bit is located at 0-3 & 7 bit in 4 byte data.

Table 1 Data Types of α

の設定とゴール呼出を行い、節の頭部に対応したマシン命令が実引数とのユニフィケーションを行う。モード宣言されている時は、①入力実引数，②普通の実引数，③出力実引数という順序でユニファイする。各引数が構造体の場合は、深さ優先にユニフィケーションを行うよう全レベルがコンパイルされる。ユニフィケーションはハードウェア・スタック上で高速に実行されるが、マシン命令で扱えるレジスタが存在しないためポインタを用いないユニフィケーションを実現している。

すなわちWarrenらの実現 [War 80] では、GET-LIST命令は引数がリストであることを検査したのち構造体へのポインタを張るだけであるのに対し、ＦＡＣＯＭαのGET-LIST命令は引数の検査をしたのちリストを分解しCDR 部，CAR 部の順序でスタックにプッシュする（Fig. 2 参照）。このとき分解した要素のデレファレンス処理を行うが、モレキュールの作成は行わず、環境へのポインタ（ENV ）を明に持ち歩く。変数とのユニフィケーション（GET-LOCAL-VARIABLE命令等）の時のみモレキュールの作成処理（MAKE-MOLECULE 命令）を行う。モレキュールの作成を抑制することにより、複雑な構造体データのユニフィケーションの効率を向上させている。

2．4 インデキシング法

候補節の高速な絞り込みを実現するためインデキシングおよびハッシングを採用し

	ENTRY	CONTENT
0	LEVEL#	Depth of this frame
1	CALL#	Call No. for tracer
2	FPS	Frame pointer save
3	TRS	Pointer to Trail stack
4	GFP	Pointer to Global stk
5	PFP	Parent frame pointer
6	RCP	Recent choice pointer
7	ARG	Argument goal
8	SU	Success return address
9	CC	Current clause
10	FL	Failure return address
11	LVAR#	Number of local vars

(a) Interpreter

	ENTRY	CONTENT
0	LEVEL#	Always 0
①	PN	Predicate name
2	FPS	Frame pointer save
3	TRS	Pointer to Trail stack
4	GFP	Pointer to Global stk
5	PFP	Parent frame pointer
6	RCP	Recent choice pointer
⑦	ARG#	Number of arguments
8	SU	Success return address
⑨	SUP	Success frame pointer
10	FL	Failure return address
11	LVAR#	Number of local vars

○ : Use for different purpose

(b) Compiled Predicate

Table 2 Entries of Control frame

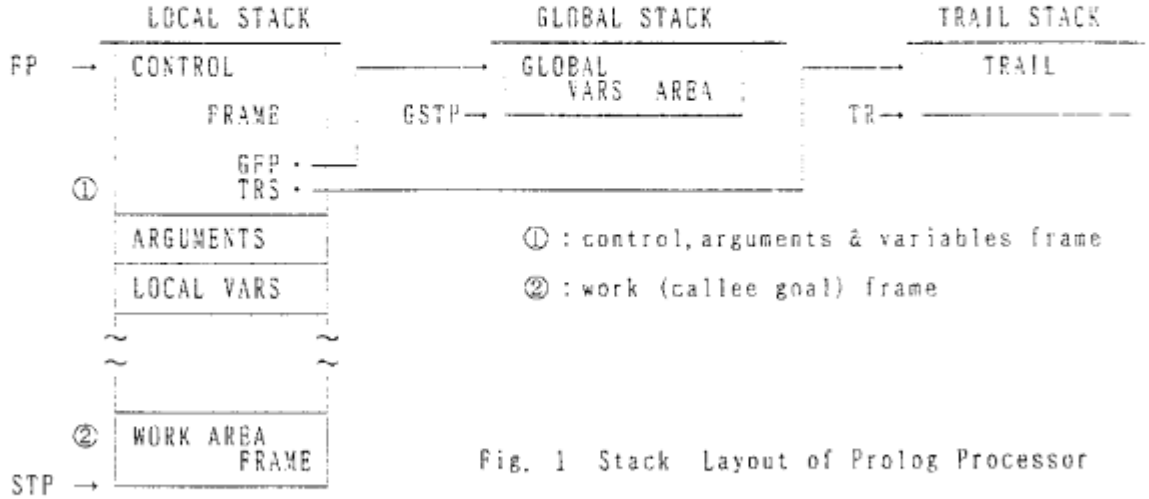


Fig. 1 Stack Layout of Prolog Processor

ている〔War 77〕。4方向分岐命令であるSWITCH-ON-TERM命令を4byte命令として実現するため、分岐テーブルを用いている（Fig. 3(a)）。また短分岐の高速な実現のため、相対アドレスが全て8Bitで表される場合は、分岐テーブルを3Byteのオペランドに押し込むSWITCH-ON-TERM-SHORT命令を用いる（Fig. 3(b)）。

インデキシングおよびハッシングは、述語内で候補節を高速に選び出すだけでなく、各節が別候補を持つ可能性を減らす。そのためこれまで別候補節を持っていた節を決定節として扱うことが可能となり効率のよいコード生成ができる。

2．5　マシン命令の設計

ＦＡＣＯＭαはLISP専用の命令として156個の命令を備えている〔篠木83〕。しかしPrologプログラムのコンパイルを考えると機能的，速度的に不充分なので、新たにProlog専用のマシン命令を65個追加した。マシン命令は全てマイクロプログラムで実現されており、6Byteの命令先取りバッファ，マシン命令のディスパッチ・メモリ等のハードウェアにより高速に実行される。マシン命令の設計においては、次のような点に留意した。

(1) Prologは従来の言語に比べ実行時判断が多い。そこでマシン命令のセマンティクスを高めに設定し実行時の判断はマイクロプログラム・レベルで処理した。マシン命令の分岐を抑える事で命令バッファのヒット率を高めている。

(2) デレファレンス処理，モレキュール作成処理等の共通機能は各命令から分離して専用命令として用意し、重複した処理の省略をコンパイル時に行った。

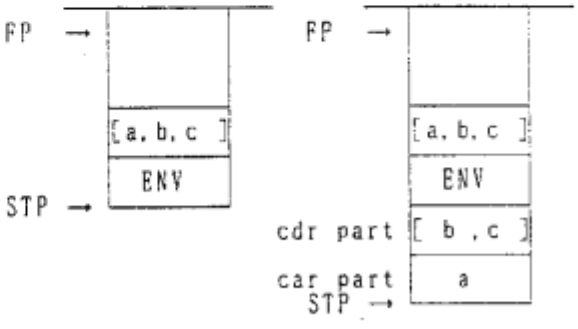
新たに作成したProlog専用命令は、①ユ

```
q( [X|Y ] ).          % predicate
?-  q( [a,b,c ] ).    % query
```

(a) sample predicate and query

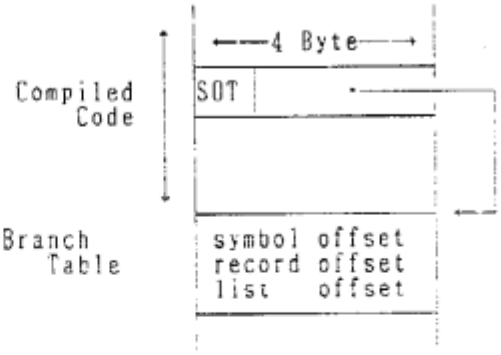
```
(LS A1)   % push first argument
(LS ENV)  % push environment
(GL)      % unify list
```

(b) codes for list unification

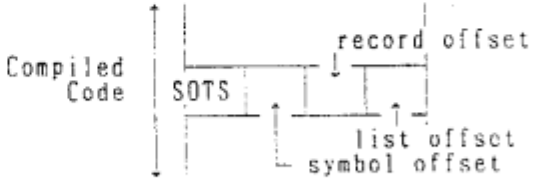


(c) Before (GL)　　(d) After (GL)

Fig. 2　List Unification



(a) SOT Instruction



(b) SOTS Instruction

Fig. 3　Switch-on-Term Instruction

Instruction	Code	Byte
GET-ATOM	GA	4
GET-INTEGER	GI	4
GET-CONSTANT	GC	4
GET-LIST	GL	2
GET-SKELETON	GS	6
GET-LOCAL-REFERENCE	GLR	2
GET-GLOBAL-VARIABLE	GGV	2
GET-GLOBAL-REFERENCE	GGR	2
CHECK-ATOM	CA	4
CHECK-INTEGER	CI	4
CHECK-CONSTANT	CC	4
STORE-ATOM	SA	4
STORE-INTEGER	SI	4
STORE-CONSTANT	SC	4
STORE-MOLECULE	SM	4

Table 3 GET Instructions

Instructions	Code	Byte
PUT-LOCAL-VARIABLE	PLV	2
PUT-GLOBAL-VARIABLE	PGV	2
PUT-LOCAL-DEREFERENCE	PLD	2
PUT-GLOBAL-DEREFERENCE	PGD	2
PUT-MOLECULE	PM	4
PUT-UNDEF	PU	2

Table 4 PUT Instructions

Instructions	Code	Byte
PRED-CALL	PCALL	2
CONT-PRED-CALL	CPCALL	2
PRE-LINK	PLINK	2
PRE-CONT-LINK	PCLINK	4
PRE-TR-LINK	PTLINK	4
LINK	LINK	4
CONT-LINK	CLINK	4
TR-LINK	TLINK	4
PROCEED	PCD	2
EXIT	EXIT	2
REFUTE-IF-EXPR	RIE	4
CHECK-TRUE-AND-POP	CTP	2
MAKE-PROLOG-FRAME	MPF	4
ALLOC-LOCAL	AL	2
ALLOC-GLOBAL	AG	2

Table 5 Procedural Instructions

Instructions	Code	Byte
SWITCH-ON-TERM	SOT	4
SWITCH-ON-ATOM	SOA	2
SWITCH-ON-SKELETON	SOS	2
TRY	TRY	4
RETRY	RETRY	4
TRUST	TRUST	4
TRY-ME-ELSE	TME	4
RETRY-ME-ELSE	RME	4
TRUST-ME-ELSE-FAIL	TMF	2

Table 6 Indexing Instructions

ニフィケーションを行うGET 系命令，②実引数の設定を行うPUT 系命令，③述語間の制御を行うPROCEDURAL系命令，④候補節の選択を行うINDEXING系命令，⑤その他に分類できる。命令の一覧をTABLE 3 ～ 7 に示す。

2．6 コンパイラの使用例

コンパイラはPrologとLISPを用いて記述されている。Prologで記述しているのはユーザとのインタフェースを行う部分だけであり、コンパイラの本体はLISPで約 3,300 行である。実際にコンパイラが生成したコンパイルコードを Fig. 4 に示す。

Fig. 4 はモード宣言あり（モード違反の検査コードは抑制してある）で述語内ループ化を行う決定的なAPPENDのコンパイルコードである。

3．高速化技法

本節ではコンパイルによって推論動作が高速化される理由を述べたのち、本コンパイラで採用した特徴のある高速化技法である「モード宣言の拡張」と「高速なゴール呼出法」について述べる。

3．1 コンパイルによる高速化の理由

コンパイルすることにより実行が高速化されるのは、インタプリタによるプログラムの読出し，解釈といったオーバヘッドが

Instructions	Code	Byte
BRANCH-ON-REFERENCE	BOR	4
BRANCH-NON-REFERENCE	BNR	4
FAIL	FAIL	2
CUT	CUT	2
LOAD-PRED-DEFINITION	LPD	6
LOAD-RCP	LRCP	2
LOAD-EFECTIVE-ADDRESS	LEA	4
LOAD-STACK-POINTER	LSP	2
SET-FRAME-POINTER	SFP	2
SET-TRAIL-POINTER	STP	2

Table 7 Miscellaneous Instructions

なくなることに加え次のような理由による。

(1) ユニフィケーションの特殊化

ユニファイを行う2つの「項」のうち、節頭部に書かれている「項」はコンパイル時に完全に解析できる。ゴール側の「項」も（もし宣言されていれば）モード宣言によって未束縛変数とそれ以外に分けることができる。これらの情報から汎用のユニファイ命令ではなく、特殊化されたユニファイ命令を生成しユニフィケーションを高速化する。

(2) 変数の再分類

コンパイラでは節全体の変数出現と使用法の様子を解析できるので、インタプリタよりも精密に変数を分類する事ができる。その結果、グローバル変数をローカル変数にし，ユニファイ命令をロード，ストア命令に置き直す。またインタプリタにないボイド変数を導入している。

例えばFig. 4に示したAPPENDの第1節の変数は次のように再分類される。

X　具体化したグローバル変数

L1　具体化したローカル変数

L2　具体化したローカル頭部変数

L3　未束縛なグローバル変数

(3) 候補節の絞り込み

インデキシングやハッシングの手法を用いて候補節を高速に絞り込む。また絞り込んだ節が決定節となり、トレール操作が不要になる機会が増える。

3．2　モード宣言による高速化

本コンパイラは節頭部に現れる構造体を完全にコンパイルする方式である。そこでDEC-10 Prolog 風のモード宣言（＋，－）を拡張し、より深いモード宣言（++）を導入した。モード宣言（++）は節頭部に現れる範囲で値が具体化しているという宣言である（Fig. 5）。図中の1，3番目の問は宣言を満たしている。2番目の問は変数L1に対応する実引数が未束縛でありモード宣言に違反している。モード宣言（++）を用いれば構造体の内部のオブジェクトに対しても、効率のよいコードを生成できる。

デレファレンス処理したのち実引数を受渡すというインタフェースでは、モード宣言（＋）の宣言された引数位置にある変数

```
append( [ X| L1] ,L2, [ X| L3] ):-
        append(  L1 ,  L2 ,  L3 ).

append( [] ,  X ,  X ).

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

(a) Source Program of APPEND

```
APPEND
    (SOT LA0001 NEXT0002 LL0003)

LL0003
    (AG 2)      % alloc global vars
    (PU)        % alloc local  var
    (LS A1)     % unify 1st argument
    (LS ENV)
    (GL)        % deconstruct list
    (MKM 2)
    (GGV G1)    % unify X
    (MKM 1)
    (SSP X4)    % unify L1
    (POP 2)
    (LS A3)     % unify 3rd argument
    (SM '(G1 . G2))
                % store molecule
    (LS X4)     % 1st argument
    (PGV G2)    % 3ed argument
    (SSP A3)    % set argument
    (SSP A2)
    (POP1)
    (SOT LA0001 NEXT0002 LL0003)
                % indexing
LA0001
    (LS A1)     % unify 1st argument
    (CA [] )
    (LS A2)     % unify 2nd argument
    (SL A3)
    (LRCP)
    (PCD)

NEXT002         % case of record
    (TMF)
    (LRCP)
    (FAIL)
```

(b) Compiled Code of APPEND

Fig. 4 Determinate APPEND

とモード宣言（++）の宣言された引数位置にある構造体中の変数はリファレンスでないことが保証されている（具体化している）。このような位置に少なくとも一度出現した変数を「具体化した変数」と呼ぶ。

「具体化した変数」はデレファレンス処理が不要で、ユニフィケーションはストア命令で、引数としての設定はロード命令で代用できる。APEEND述語のコンパイルコード（Fig. 4）では、 X, L1, L2が「具体化した変数」であり、 L1 のユニフィケーションはストア命令(SSP X4)で行われる。

同様にデレファレンス処理が省略できる場合として「真のリファレンス」と呼ぶ場合がある。実引数をデレファレンス処理したのち設定しているので、モード宣言（－）された実引数は、その時点でリファレンスでありリファレンスの先が必ず未束縛の変数値セルを指している。一般にはリファレンスの先は、定数や別の変数値セルへのリファレンス等も許されており、リファレンスの先にデレファレンス処理なしで書込みを行うのは危険である。それに対し「真のリファレンス」ではデレファレンス処理を省略できる。実引数が「真のリファレンス」であることが保証されているのは、述語が呼び出されてからユニフィケーションがおこるまでの間である（その述語に割り当てられた変数域へは書込みを行なってもよい）。APPEND述語の例（Fig. 4）では、第３引数が「真のリファレンス」でありデレファレンス処理を行わず直接リファレンスの先へモレキュールを書込むストア命令(SM '(G1 . G2)) を生成している。

本コンパイラで導入したもう１つの新しいモード宣言に準具体化モード宣言（?±, -±）がある。値が準具体化されているとは、デレファレンス処理を行うとリファレンス以外の値がもとまるという意味である。モード宣言（?±）は述語の呼出時にはモード宣言（？）の意味で、終了後にはその引数は具体化されている事を宣言する。同様にモード宣言（-±）は述語呼出時は入力宣言（－）であり、終了後具体化する事を宣言している。準具体化モード宣言（?±, -±）は、コンパイル時に変数がunsafe変数でないことを検出するために用いることができる。すなわち本体のゴール呼出中で出力宣言または準具体化宣言されている変数はunsafe変数ではないと判断する。

従来は自分自身にたいするモード宣言のみを利用していた。本方式では呼出すゴールに関するモード宣言も利用できる。例題を用いて説明する。8-QUEEN の一部であるNOT-TAKE1 述語の定義をfig. 6に示す。変数 N1, M1はNOT-TAKE1 述語に対するモード宣言からはunsafe変数でないとは判断できない。しかし組込述語 is に対して

```
:- mode is(?±, ++).
```

が宣言されているのでいずれもunsafe変数でないことがわかりNOT-TAKE1 は述語内ル

```
append( [X|L1] ,L2, [X|L3] ):-
:-mode append(--, ++, -).
```

(a) Declaration

```
?-append( [a,b,c ]  [d ] ,ANS).
    X=a      L1= [b,c ]    -- OK
?-append( [a|Y ]  [d ] ,ANS).
    X=a      L1= UNDEF     -- NO
?- Y= [b|Z ],
   append ( [a|Y ]  [d ] ,ANS).
    X=a      L1= [b| UNDEF]   -- OK
```

(b) Usage

Fig. 5 Deeper Mode Declaration

ープ化を行うことができる。このようにコンパイラでは組込述語に対するモード宣言を予め保持している。組込述語に対するモード宣言の一部をFig. 7に示す。

3．3　高速な呼出法

ゴール呼出を高速化するため、次のようなゴール呼出法を用意している。これらの高速なゴール呼出法は、コンパイルされていない述語が呼べなくなったり，述語の変更が反映しなかったりするトレードオフがあるため、(1)以外はコンパイル・オプションとして利用する。

(1)　継続呼出（Continuation Call ）

終了処理を高速化するために継続呼出を採用している。継続呼出は非決定節でも用いることができるが、通常呼出と同量のスタックを消費する。

(2)　リンク呼出（直接呼出）

コンパイルされた述語同志の呼出を高速化するためにリンク呼出を採用している。通常の呼出は述語名と引数個数からシンボルヘッダを経由して述語定義を探し出し呼出を行なうが、リンク呼出では述語の先頭番地に直接分岐する。リンク呼出は呼出される側がコンパイルされた述語であることが必要でり、実行中に呼出される側の述語を再定義しても反映されないが、述語定義を探すオーバーヘッドが無くなるため、高速な呼出が可能となる。

(3)　終端再帰呼出（TRO [Mar 80] ）

終了処理を高速化するために終端再帰呼出を採用している。終端再帰呼出は決定節でしか採用できないがローカルスタックの消費量を大幅に抑制することができる。

(4)　述語内ループ化

決定節でかつ最終ゴールが自分自身である場合，再帰呼出をループ（繰返し）に変換する。述語内ループ化は無条件相対分岐命令で実現されるので，(3)リンク呼出と(2)終端再帰呼出を組合せたものよりさらに高速に動作する。

4．性能測定

作成したコンパイラの速度性能及び動作特性に関する測定を行なった。本節では測定結果とそれに対する考察を述べる。

4．1　実行時間

Prologコンテスト［奥野84］の問題に対する実行時間の一部ををTABLE 8 に示す。3節で説明した高速化技法がどの程度寄与しているかを示すため、12つのケースについて実行速度を測定した。12つのケース分けと測定結果をTABLE 9 に示す。測定結果

```
not-take1( [] ,N,M).

not-take1( [X|L ] ,N,M)
   :- X = = N,
      X = = M,
      N1 is N+1,
      M1 is M+1,
      not-take1(L,N1,M1).

:- mode not-take1(++,+,+).
```

Fig. 6　Predicate NOT-TAKE1

```
% ARITHMETIC
  :- mode is(?±,++),
          '>'(++,++),
          '=:='(+-,++),
           ...  .

% CONVENIENCE
  :- mode length(+,?±).

% DATA BASE
  :- mode recorded(+,?,?±),
          recoreda(+,?,?±),
          recoredz(+,?,?±),
          erase(+),
           ... .
```

Fig. 7　Mode Declaration for Builtin Predicate

から次のようなことが分る。

(1) 呼出法が同じでモード宣言が違う組合せ（①と⑤と⑨，②と⑥と⑩，③と⑦と⑪，④と⑧と⑫）の比較によればモード宣言（+，-）によって処理が0～20%高速化される。またモード宣言（++，-*）を用いると35～70%高速化する。これは（+，-）と比べても25～70%高速である。

(2) リンク呼出，終端再帰呼出によってゴール呼出が高速化される。実行時間からみると、リンク呼出のほうが終端再帰呼出よりも高速である。述語内ループ化を用いれば通常の呼出法と比べて最高45%高速化される。

4．2 スタック使用量

コンパイル実行時の3本のスタックの使用量の測定を行い、インタプリタ実行時の使用量と比較を行った。ローカル・スタックの使用量はＳＶＰ（サービス・プロセッサ）を用いてマイクロ命令を1ステップずつ実行させ、そのたびにスタックトップ・ポインタを読み出して測定した。グローバル・スタック，トレール・スタックの使用量はLISPの関数により測定前に領域を全部初期化し、試験プログラム実行後に使用済の領域を調べることで行なった。幾つかのプログラム実行によるスタック使用量を4．1と同じ8つのケースについて調べTABLE 10に示す。

(1) ローカルスタックの使用量は、終端再帰呼出および述語内ループ化を行うことにより大幅に減少する。NREVERSEでは述語NREVERSEが決定節であることが検出できないため使用量の減少はみられない。

(2) モード宣言なしの場合グローバルスタックの使用量はインタプリタ実行時とほぼ同じである。モード宣言を行うと変数が再分類されるのでグローバル

[msec]

benchmark prog	interpre	compile
APPEND (30)	2.74	0.75
NREVERSE (30)	39.8	15.5
QSORT (50)	52.1	18.6
DATABASE-1	51	2
LISP (FIB10)	1156	443

Table 8 Result of Benchmark

CASE	mode	goal invocation
①	×	normal invocation
②	×	linked call
③	×	tail recursive optim
④	×	trans iteration
⑤	○	normal invocation
⑥	○	linked call
⑦	○	tail recursive optim
⑧	○	trans iteration
⑨	◎	normal invocation
⑩	◎	linked call
⑪	◎	tail recursive optim
⑫	◎	trans iteration

× without Mode Declaration
○ with +,- Mode Declaration
◎ with ++,-* Mode Declaration

(a) Optional Case Explanation

CASE	NREVERSE		QSORT	
	msec	KLIPS	msec	KLIPS
①	33.1	(15)	31.9	(23)
②	28.9	(17)	28.7	(25)
③	27.9	(18)	29.8	(24)
④	26.5	(19)	26.7	(27)
⑤	29.6	(18)	30.1	(24)
⑥	25.3	(20)	26.9	(27)
⑦	26.9	(18)	28.0	(26)
⑧	26.9	(18)	23.0	(26)
⑨	22.3	(22)	23.7	(31)
⑩	18.3	(27)	20.5	(35)
⑪	19.5	(25)	21.6	(33)
⑫	15.5	(32)	18.6	(39)

(b) Result

Table 9 Result in Each Case

スタックの使用量が減る。

(3) コンパイルすることにより候補節の絞り込みが行われる。これまで別候補があった節が決定節になり、トレールスタックの使用量が減少する。

4．3 ＣＰＵ占有率

Prolog実行中において処理がどのような比率で行われているかを調べるため、ＣＰＵの占有率の測定をおこなった。測定はマイクロ命令ごとに割込みを起し、割り込まれた番地に対応するカウンタをインクリメントすることで行った。APPENDと8-QUEEN に対するインタプリタ実行とコンパイル実行におけるＣＰＵ占有率をFig. 8に示す。

(1) デレファレンス処理は3．2で示したモード宣言の利用によって、インタプリタ実行に比べ８割以上省略される。特にAPPENDにおいては完全に抑制される。

(2) 節頭部をコンパイルして特殊なユニファイを行うことによりユニフィケーションの手間が約半分に減少した。

(3) 組込述語の実行時間は固定分であり、8-QUEEN のように組込述語の比率の高いプログラムでは高速化が難しい。

[Word = 4Byte]

	CASE	LOCAL	GLOBAL	TRAIL
NREVERSE	INT	472	2822	435
	①	477	2763	0
	②	↓	↓	↓
	③			
	④	↓	↓	↓
	⑨	496	1023	0
	⑩	↓	↓	↓
	⑪			
	⑫	↓	↓	↓
QSORT	INT	754	1652	268
	①	818	1654	104
	②	↓	↓	↓
	③	197		
	④	↓	↓	↓
	⑨	868	1023	104
	⑩	↓	↓	↓
	⑪	197		
	⑫	↓	↓	↓

Table 10 Stack Consumption

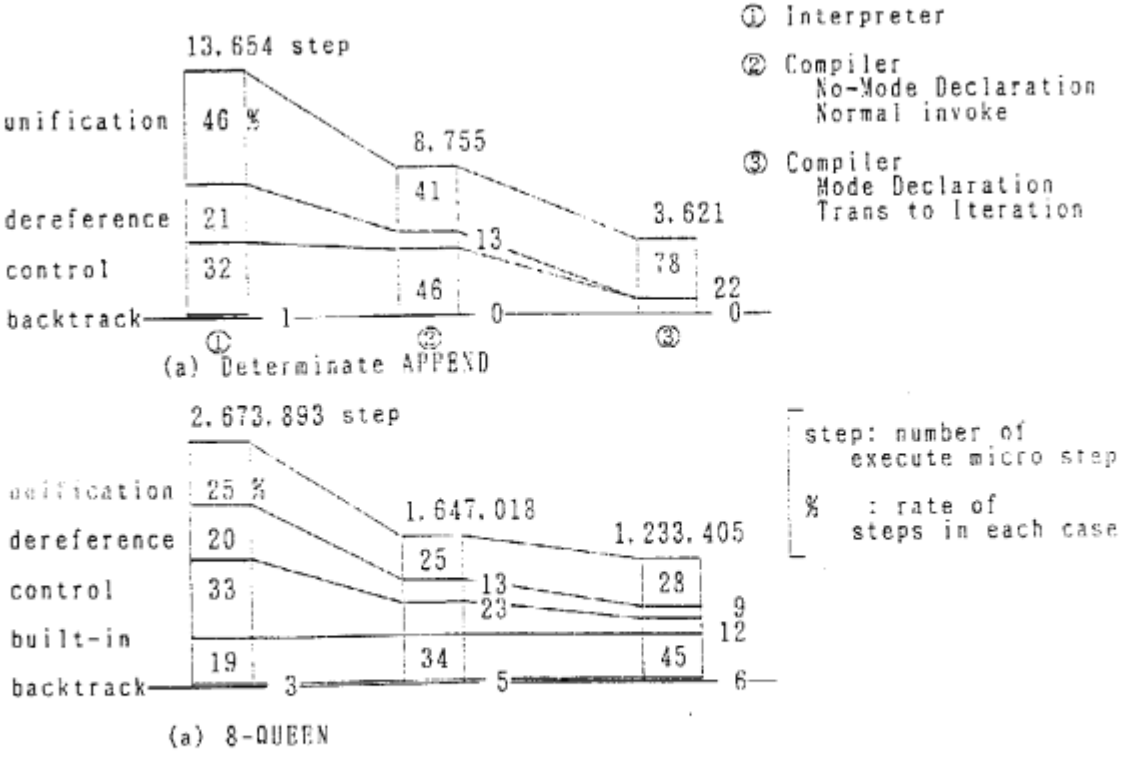


Fig. 8 Dynamic Analyze of Prolog Execution

5．性能評価

前節で行った測定結果に基づいてPrologコンパイラとＦＡＣＯＭαの性能について論じる。

5．1　ＣＰＵ，ＭＳＣの稼働率

代表的なPROLOGプログラムとLISPプログラムに対するＣＰＵとＭＳＣ（主記憶制御装置）の稼働率をTable 11に示す。

インタプリタ同士比較するとLISPに比べてPrologでは、ＣＰＵ，ＭＳＣ共に稼働率が低いことがわかる。しかしコンパイルすることによって同程度となる。ＭＳＣの稼働率中　31%（Prolog）32%（LISP）は命令フェッチによるものである。

ＣＰＵのアイドル率は約２割であり、ＦＡＣＯＭαのＣＰＵと主記憶（読み出し 480ns，書込み 640ns）はバランスがとれていることが判る。

5．2　終端再帰呼出

引数の受渡しをスタックを用いて行うので、終端再帰呼出においてハードウェア・スタックのポインタ張換えが必要となる。ＦＡＣＯＭαはスタックマシンであるがスタックポインタ操作の制約から、ポインタの張換え処理に時間がかかる。全処理時間に対する終端再帰呼出命令の占める割合をTable 12に示す。この表から終端再帰呼出のコストが大きいことが判る。もし終端再帰呼出がリンク呼出と同程度のコストで実現できるなら、ケース⑪は31 KLIPS (NREVERSE)，37KLIPS(QSORT)で実行可能である。

5．3　トレール・スタック操作

Prologの実行のなかでトレール・スタックに関する操作には次のものがある。

①　トレイルプッシュ処理

②　バックトラック処理

	PROGRAM	ＣＰＵ	ＭＳＣ
inter	APPEND	97 %	32 %
	NREVERSE	78	29
	QSORT	89	29
comp	APPEND	78	59
	NREVERSE	82	53
	QSORT	83	48

(a) Prolog Programs

	PROGRAM	ＣＰＵ	ＭＳＣ
inter	SORT	95 %	51 %
	TARAI	97	54
	TPU	97	47
comp	SORT	86	52
	TARAI	91	50
	CPU	84	39

(b) LISP Programs

Table 11 RUN/IDLE Ratio of CPU,MSC

	NO-MODE	MODE
APPEND	22.0 %	33.6 %
NREVERSE	16.7	25.3
QSORT	10.4	12.9
8-QUEEN	7.2	8.4

Table 12　Rate of TRO Operation

	QSORT	8-QUEEN
trail-push	4.3 %	0.1 %
backtrack	3.2	4.7
undo	2.0	1.2
cut operation	2.3	0.6
tidy-up	1.3	0.0

Table 13　Rate of Trail Operation

③ undo処理

④ カット処理

⑤ tidy-up 処理

全処理時間に占めるこれらの操作の割合をTable 13に示す。ＦＡＣＯＭαはグローバル・スタック，トレール・スタック操作のためにスタックの先頭と境界をＣＰＵ内のレジスタに保持する程度で特別なハードウェアは持っていない。しかし表によれば、トレール・スタックに関する処理は処理全体の僅かな時間しか占めていない。

6．おわりに

ＦＡＣＯＭα上でPrologコンパイラを作成した。コンパイルした述語はインタプリタの約３倍，30～40KLIPS で推論を行う。高速化技法として、モード宣言を拡張して新しい工夫をこらした。拡張したモード宣言を用いれば宣言なしの場合に比べ35～70%処理が高速になる。従来のモード宣言と比較しても25～70%高速化される事が確認できた。また様々なゴール呼出法に関して定量的な評価を行った。コンパイルした述語の動作解析を行いＦＡＣＯＭαの評価を行った。

尚、本研究は第５世代コンピュータプロジェクトの一環としてＩＣＯＴの委託で行ったものである。

謝辞

日頃御指導頂く棚橋部長，林室長並びに有益な議論をして下さった研究室の諸兄に感謝します。またコンパイラの開発，データ収集に協力して下さった富士通ＳＳＬの杉野，山崎，猪野，山内の４氏に感謝します。

参考文献

[War 77] Warren, D.H.D., "Implementing Prolog - compiling predicate programs", D.A.I. Research Report 39-40, Univ. of Edinburgh (1977).

[War 80] Warren, D.H.D., "An improved Prolog implementation which optimizes tail recursion," 1980 Logic Programming Workshop, Debrecen, Hungary.

[War 83] Warren, D.H.D., "An abstract Prolog instruction set", Tech. Note 309 A.I.C. CRI International.

[Tic 84] Tick E. and Warren D.H.D., "Towards a pipelined Prolog processer," 84 International Symposium on Logic Programming.

[服部83] 服部ほか，"ALPHA- 高性能LISPマシン，" 情処学会　記号処理研究会資料 23-1

[服部84] 服部ほか，"PROLOG インタプリタの構成" 情処学会　29回全国大会

[篠木83] 篠木ほか，"LISP マシンALPHA のマシン命令" 情処学会　29回全国大会

[奥野84] 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，" 情処学会　記号処理研究会資料　28-4