ICOT Technical Report: TR-136

TR-136

Design and Evaluation of a Prolog Compiler

by
M. Kishimoto, T. Shinogi, Y. Kimura
and A.Hattori (Fujitsu Ltd.)

September, 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

DESIGN AND EVALUATION OF A PROLOG COMPILER

M. Kishimoto, T. Shinogi, Y. Kimura, and A. Hattori

FUJITSU LIMITED
1015 Kamikodanaka Nakahara-ku, Kawasaki 211, Japan

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. INTRODUCTION

A Prolog compiler has been created that runs on the FACOM α, a dedicated machine for symbolic data processing (Akimoto 1985). This paper discusses the design philosophy of the compiler, optimization methods, execution results, and evaluates them.

This Prolog language processor is based on DEC-10 Prolog. The language is identical to DEC-10 Prolog except for part of the syntax and the built-in predicates (Warren 1977).

The Prolog language processor is intertwined with the LISP language processor. The interpreter is started by executing the LISP function PROLOG.

The Prolog interpreter and garbage collector (GC) are implemented by microprograms. The garbage collector is shared by both the LISP and Prolog language processors, using data tags to differentiate the object code. Most of the Prolog I/O routines and built-in predicates are coded in LISP, but some are written in Prolog. The compiler discussed in this paper is also written in LISP.

When executed by the interpreter, programs run at about 12 thousand logical inferences per second (KLIPS), several times slower than required for practical use. A compiler speeds up execution of predicates to 30 to 40 KLIPS.

Section 2 of this paper outlines the compiler and the machine instructions. Section 3 explains the extended mode declarations, which greatly improve execution speed, and methods of fast goal invocation. Section 4 presents some measured results of the execution characteristics of compiled predicates and evaluates the performance.

2. COMPILER DESIGN

2.1 Data Structures

Compiled predicates have basically the same data structures as when they are interpreted, as shown below.

(1) Structure sharing is used. The data length is 4 bytes, consisting of a 3-byte pointer and a 1-byte data tag. Molecules consist of 8 bytes, including a skeleton and environment.

(2) A compound term is a tuple (LISP vector type) or a CONS cell. For compound terms that do not include variables, there are two special types: constant list, and constant tuple.

(3) Logical inferences are performed using three stacks. The local stack is implemented in hardware. The global stack and trail stack are in main storage, where they can be compacted by the garbage collector.

2.2 Separation of Control Frames

In the interface to the compiled predicates, the actual arguments are placed on the local stack before the predicate is called. Since the FACOM α is a stack machine, it does not have any registers manipulated by machine instructions. The method of calling the predicate after its actual arguments have been prepared is the same as the Warren's structure copying method, except for the following differences (Warren 1983, Tick 1984):

(1) Actual arguments are passed on the local stack instead of registers. The compiler is therefore spared the complex task of register allocation. Tail recursion can be performed by changing pointers on the local stack, because it is not necessary to globalize the unsafe variables. Unsafe variables, however, must be considered in in-predicate loop as described below (Warren 1980).

(2) Actual arguments are dereferenced before being passed to predicates. The compiler can determine whether dereferencing is necessary, so it generates code without needless dereference processing. Therefore logical inference can be performed with less dereferencing than when dereferencing is postponed until it becomes necessary.

The FACOM α accesses the local stack by the frame pointer. In Prolog, however, it is determined at run time whether recent choice pointer (RCP) frames are allocated. Since the frame length cannot be calculated at compilation time, stack access by the frame pointer cannot be used. This problem is solved by separating the predicate frame into two parts (Figure 1).

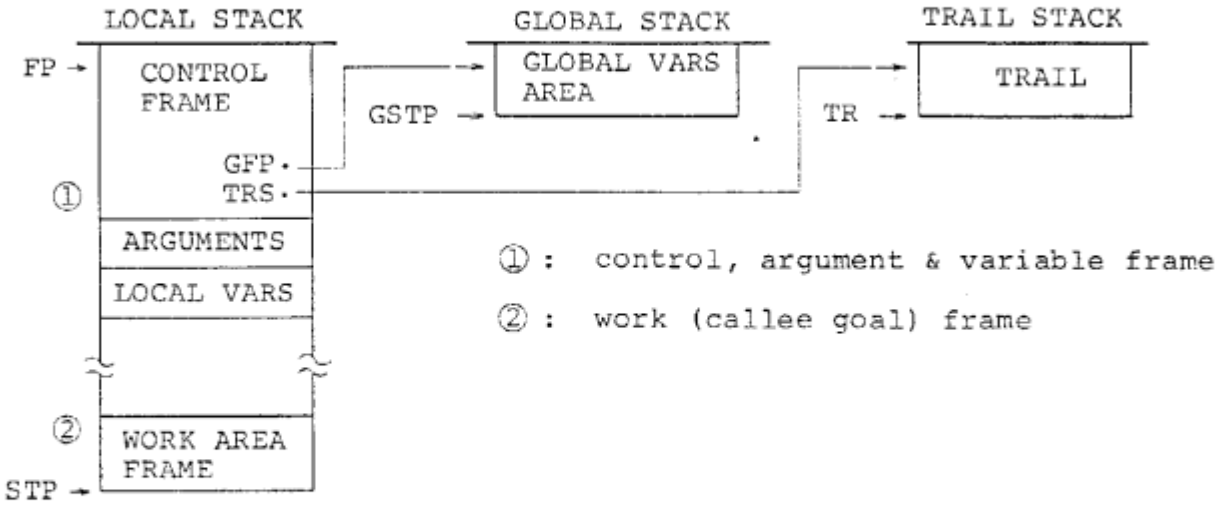

Fig. 1 Layout of the stacks for the Prolog processor

① Control, argument, and variable frame

The compiler can calculate the size of this frame, which is accessed by means of the frame pointer.

② Predicate call (work) frames

This frame is work area used for goal invocation in the clause body. Since there may be an RCP frame between this type of frame and a frame of type ①, access is relative to the stack top pointer rather than the frame pointer.

The contents of the control frame are nearly the same as ones for the interpreter. Three entries are used for different purpose to have compiled predicates run faster.

2.3 Unification

When a compiled predicate is executed, machine instructions corresponding to a body of a clause set actual arguments and invoke a goal. While other machine instructions corresponding to the head of the clause perform unification with actual arguments. If there is a mode declaration, unification is performed in the order of (1) input actual arguments, (2) normal actual arguments, (3) output actual arguments. If each argument is a compound term, all levels of objects are compiled to unify in the depth first manner. Unification is performed at high speed on the hardware stack, without using registers, since there are no registers available to the machine instructions.

In Warren's implementation (Warren 1980), the GET-LIST instruction simply checks whether an argument is a list, then sets a pointer to the compound term. In contrast, the FACOM α GET-LIST instruction checks whether an argument is a list, then decomposes it, and pushes first the cdr part then the car part onto the local stack. (See Figure 2.) The decomposed elements are dereferenced at this time, but no molecules are generated, and the environment pointer (ENV) is explicitly carried about. Molecules are generated (by the

MAKE-MOLECULE instruction) only ahead of a unification for variables (by GET-LOCAL-VARIABLE and other instructions). Suppressing generation of molecules speeds up unification of complex compound terms.

(a) sample predicate and query

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

(b) codes for list unification

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

(c) Before (GL) (d) After (GL)

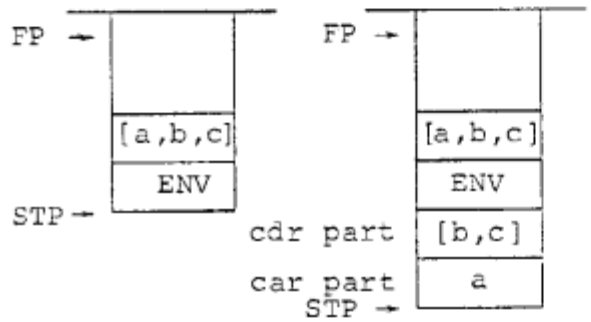

Fig. 2 List unification

2.4 Design of Machine Instructions

The FACOM α has 156 dedicated instructions for LISP. From the standpoints of both function and speed, these instructions are inadequate for Prolog processing, so 65 new machine instructions were added. All machine instructions are implemented by microprogram. A high execution speed achieved by using a 6-byte instruction prefetch buffer, a machine instruction dispatch memory, and other hardware features. The following points were considered in the design of the machine instructions.

(1) More run-time decisions are made in Prolog than in conventional languages. The semantics of the machine instructions are therefore set at a high level, and decisions are performed on the microprogram level at run time. The number of branch instructions is held down to improve the hit ratio on the instruction buffer.

(2) Dereferencing, molecule-making, and other common functions are performed by dedicated instructions to eliminate redundant processing at compilation time.

The instructions newly created for Prolog can be classified as follows: (1) GET instructions to perform unification; (2) PUT

instructions to set actual arguments; (3) PROCEDURAL instructions to handle inter-predicate control; (4) INDEXING instructions to select candidate clauses; (5) miscellaneous instructions.

3. OPTIMIZATION METHODS

This section discusses extended mode declaration and fast goal invocation, two special methods of optimization used in this compiler.

3.1 Speed-up by Mode Declaration

This compiler completely compiles compound terms appearing at the head of a clause. The DEC-10 Prolog like mode declarations (+, -) are therefore extended to allow a deeper mode declaration (++). It declares that all the variables even in compound terms are instantiated. (See Figure 5.) This declaration is satisfied by the first and the third items in Figure 5. In the second item, it is against the mode declaration that the actual argument of variable L1 is uninstantiated. Use of the (++) mode declaration enables an efficient object code to be generated for objects in a compound term.

(a) Declaration

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

(b) Usage

```
?-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
```

Fig. 5 Deeper mode declaration

In an interface in which actual arguments are passed after dereferencing, variables in argument positions with the (+) mode declaration and variables appearing in compound terms in argument positions with the (++) mode declaration are always instantiated (never references). A variable that appears at least once in such a position is called an instantiated variable.

For instantiated variables, dereferencing is unnecessary, so unification can be replaced by the store instructions, and setting of arguments by the load instructions.

A similar situation in which dereferencing can be eliminated is called a real reference. Since an actual argument is set after dereferencing, an argument with the (-) mode declaration is a reference, and the referenced location is always an uninstantiated variable value cell. In general, a reference indicates a constant

or a value cell of a different variable, so it is unsafe to write without dereferencing. The dereferencing can be omitted, however, in a real reference. A (-) mode actual argument is always a real reference during the interval from invocation of the predicate to the first unification.

Another new mode declaration used in this compiler is a half instantiated mode declaration (?#, -#). Half instantiated means that dereferencing would produce a value other than a reference. The (?#) mode declaration signifies the normal (?) mode declaration when a goal is called. If also declares that the argument is half instantiated after the goal termination. Similarly, the (-#) mode declaration is the input mode declaration (-) when a goal is called. It also declares that the argument is instantiated after exit. The half instantiated mode declarations (?#, -#) can be used to enable the compiler to detect not unsafe variables. The compiler determines that neither variables declared as input nor half instantiated in the goal invocation are unsafe.

Ordinarily, mode declarations can be used only in relation to their own predicates. In this system, mode declarations can also be used in relation to invoked goals. As an example, the predicate NOT_TAKE1 in Figure 6 is a part of 8-QUEEN. From the mode declaration for the NOT_TAKE1 predicate, it cannot be determined that variables N1 and M1 are safe. But, both variables are proved not to be unsafe, and recursion can be changed to in-predicate loop. Since ":- mode is (?#, ++)" is declared for the built-in predicate. The compiler is equipped with ready-made mode declarations for the built-in predicates as shown in Figure 7.

```
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_TEKE1

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

% CONVENIENCE
  :- mode length(+,?#).

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

Fig. 7 Mode declaration of built-in predicates

3.3 High-Speed Invocation

Goal invocation is speeded up by the following means. Except for (1), these means are compiler options, because balanced against the high speed are trade-offs such as that non-compiled predicates cannot be invoked, and modifications in predicates are not reflected afterwards.

(1) Continuation call

Continuation calls are used to speed up termination. Continuation calls use the same amount of stack as ordinary calls, though they are applicable even to indeterminate clauses.

(2) Linked call (direct call)

Linked calls are used to speed up calls among compiled predicates. In an ordinary call, a predicate definition is searched from the predicate name and arity, but in a linked call a direct branch is made to the address of the predicate difinition.

(3) Tail recursive call (TRO) (Warren 1980)

A tail recursive call can be used only with a determinate clause, but it greatly reduces consumption of the local stack.

(4) Forming of loops within predicates (in-predicate loop)

When a clause is determinate and the last goal is itself, a recursive call is converted to a loop (iteration). Since loops within predicates are always formed using relative branch instructions, they execute even faster than the combination of a linked call (2) and a tail recursion (3).

4. PERFORMANCE MEASUREMENTS

We measured the speed performance and dynamic characteristics of the compiled predicate. This section presents and discusses the results.

4.1 Execution Time

Table 8 shows some of the execution times for the problems in the Prolog contest (Okuno, 1984). Execution time was measured for 12 cases to examine how the optimization methods explained in Section 3 contribute. The results indicated in Table 9. Following conclusions are lead from them.

(1) A comparison of sets of cases with the same invocation method but different mode declarations (for example ①, ⑤ and ⑨) shows that the (+, -) mode declaration provides a boost in speed of 0% to 20%. The (++, -#) mode declaration provides a boost in speed of 35% to 70%.

(2) Goal invocation is speeded up by linked call and tail recursion. In terms of execution time, the linked call is faster than the tail recursive call. When in-predicate loops are used, speed is improved by a maximum of 45% compared with the ordinary call.

Table 8 Result of benchmark

[msec]

Benchmark program	Interpreted code	Compiled code
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 9 Result in each case

(a) Optional case explanation

CASE	Mode	Goal invocation
①	x	Normal invocation
②	x	Linked call
③	x	Tail recursive optimization
④	x	Trans iteration
⑤	o	Normal invocation
⑥	o	Linked call
⑦	o	Tail recursive optimization
⑧	o	Trans iteration
⑨	◎	Normal invocation
⑩	◎	Linked call
⑪	◎	Tail recursive optimization
⑫	◎	Trans iteration

x: Without mode, declaration

o: With +, - mode declaration

◎: With ++, -# mode declaration

(b) Result

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)	28.0	(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)

4.2 Stack Consumption

Consumption of the three stacks was measured during execution of compiled code and we compare it with consumption during execution of interpreted code. Several programs were run, and stack consumption was measured for eight of the same cases as in Section 4.1. Table 10 gives the results.

Table 10 Stack consumption

[Word = 4 byte]

Program	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	↓		

(1) Local stack consumption was greatly reduced by the tail recursive call and the in-predicate loops. For NREVERSE, stack consumption was not reduced, because the compiler cannot recognize the NREVERSE predicate is determinate.

(2) Without mode declaration, global stack consumption was about the same as in execution by the interpreter. With mode declaration, global stack consumption was reduced because variables were reclassified.

(3) Compilation have candidate clauses narrow down. Clauses that previously had alternatives became determinate, and trail stack consumption was reduced.

4.3 CPU Occupied Rate

The CPU occupied rate was measured during Prolog execution. Figure 8 shows the CPU occupied rate for APPEND and 8-QUEEN for compiled and interpreted codes.

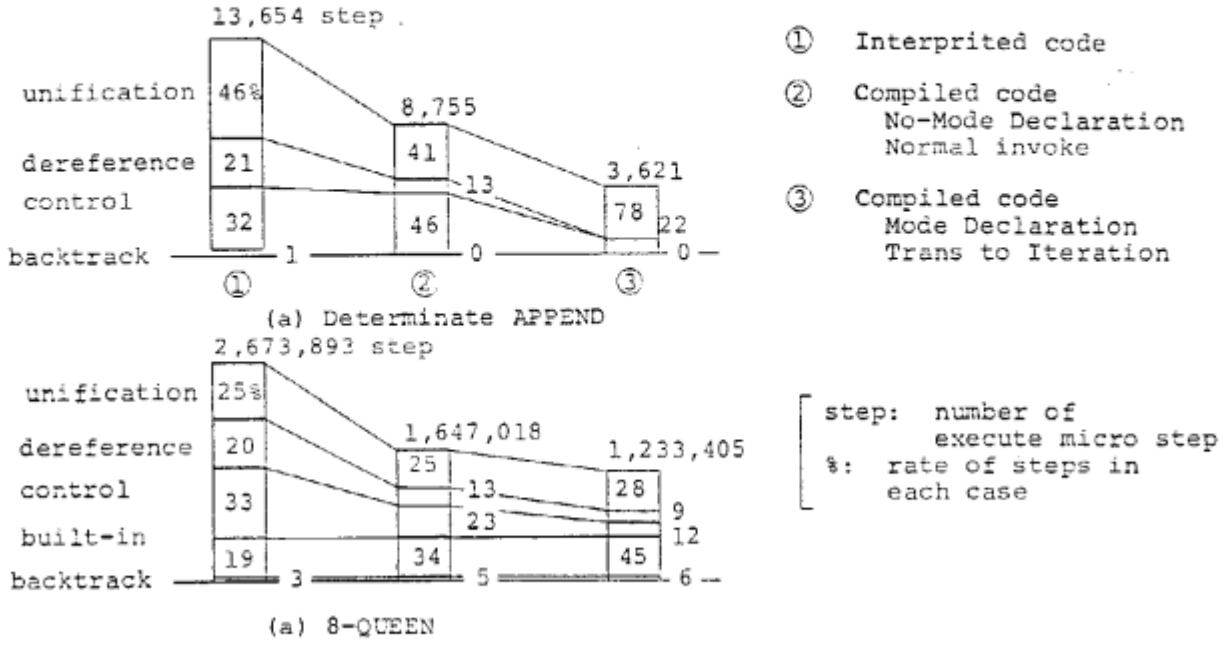

Fig. 8 Dynamic analysis of Prolog execution

(1) More than 80% of the dereferencing done by the interpreter was eliminated by using the mode declarations discussed in Section 3.2. In APPEND, all dereferencing was eliminated.

(2) Special unification, generated by compilation of clause head, cuts the unification operation roughly in half.

(3) It is difficult to speed up the execution of programs like 8-QUEEN which consist of a high proportion of built-in predicates, because the execution time of these predicates is fixed.

5. CONCLUSION

A Prolog compiler for the FACOM α has been created. Compiled predicates run at 30 to 40 KLIPS, about three times faster than when interpreted. Extended mode declarations were added to improve execution speed, giving improvements of 35% to 70% over no declarations, and 25% to 70% over conventional mode declarations. Various invocation methods were evaluated, and a dynamic analysis of compiled predicates was performed to evaluate the FACOM α.

This research was done at the request of the Institute for New Generation Computer Technology (ICOT) as part of the fifth-generation computer project.

ACKNOWLEDGMENTS

The authors wish to acknowledge the guidance received from Mr. Tanahashi and Mr. Hayashi, and the useful discussions with other members of the laboratory staff. They also wish to thank Mr. Sugino, Mr. Yamazaki, Mis. Ino, and Mr. Yamauchi of Fujitsu SSL for cooperation in developing the compiler and collecting data.

REFERENCES

AKIMOTO H (1985) Evaluation of the dedicated Hardware in FACOM ALPHA. IEEE 1985 COMPCON SPRING

Okuno H (1984) Proposed problems for 3rd LISP contest and 1st Prolog contest. (In Japanese) IPSJ, SYM, 28-4

Tick E, Warren DHD (1984) Towards a pipelined Prolog processor. 1984 International Symposium on Logic Programming

Warren DHD (1977) Implementing Prolog - compiling predicate programs. DAI Research Report 39-40, Univ. of Edinburgh

Warren DHD (1980) An improved Prolog implementation which optimizes tail recursion. 1980 Logic Programming Workshop, Debrecen, Hungary

Warren DHD (1983) An abstract Prolog instruction set. Tech Note 309, AIC SRI International