数理解析研究所講究録
第 586 巻 1986 年 224-247

SINGLE QUEUE COMPILATION IN EXTENDED CONCURRENT PROLOG[1]

Jiro Tanaka[2], Makoto Kishishita[3], Takashi Yokomori[4]

田中二郎　岸下　誠　横森　貴

ABSTRACT

Extended Concurrent Prolog (ECP) [Fujitsu 84, Tanaka 85] is an variant of Concurrent Prolog (CP) [Shapiro 83] with OR-parallel, set-abstraction and meta-inference features. In this paper, we describe the implementation of Extended Concurrent Prolog (ECP) "compiler" by showing how these extended features of ECP can be compiled to a Prolog program. Our ECP compiler has only one scheduling queue to which all the AND-related goals and all the OR-related clauses are enqueued. This scheduling method is designated "Single Queue Compilation." This "Single Queue Compilation" method makes it possible to handle all kinds of AND-relations and OR-relations in a uniform manner.

1 INTRODUCTION

Concurrent Prolog (CP) [Shapiro 83] is a parallel logic language which includes a commit operator and read-only annotation as language constructs. Extended Concurrent Prolog (ECP) [Fujitsu 84, Tanaka 85] is an variant of Concurrent Prolog (CP) [Shapiro 83] with OR-parallel, set-abstraction and meta-inference features.

We have already implemented the "interpreter" and the "compiler" for our ECP. Since we have already described the implementation details of our ECP "interpreter" in [Tanaka 85], we focus on the implementation details of our ECP "compiler" in this paper. This paper assumes familiarity with CP

[1] This research has been carried out as a part of Fifth Generation Computer Project.

[2] ICOT Research Center, Mita-kokusai-building 21F, 1-4-28, Mita, Minato-ku, Tokyo 108, Japan

[3] International Institute for Advanced Study of Social Information Science (IIAS-SIS), Fujitsu Limited, 1-17-25, Shinkamata, Ohta-ku, Tokyo 144, Japan

[4] IIAS-SIS, Fujitsu Limited, 140 Miyamoto, Numazu-shi, Shizuoka 410-03, Japan

[Shapiro 83] and ECP [Tanaka 85], however we summarize the main features of ECP below.

2 BRIEF INTRODUCTION TO ECP

As mentioned above, ECP is an extension of CP with OR-parallel, set-abstraction and meta-inference features. These features are as follows:

2.1 AND-parallelism and OR-parallelism

AND-parallelism and OR-parallelism are the basic parallel inference mechanisms of ECP. The former is the mechanism which evaluates AND-related goals in parallel. This mechanism has already been implemented in Shapiro's Interpreter [Shapiro 83]. On the other hand, the latter is the mechanism which realizes the parallel evaluation of guards, when there exists more than one potentially unifiable clause with the given goal. This was not implemented in Shapiro's Interpreter. The following program is an example of exploiting OR-parallelism.

```
solve(P,Mes):- call(P) | ... .
solve(P,Mes):- find_stop(Mes) | ... .
```

When "solve" is called, the above two clauses are executed in parallel by OR-parallelism. The first clause executes "P." However, as soon as "stop" is found in "Mes" in the second clause, the second clause is committed and the first clause is aborted. This realizes the "solve" with abort.

2.2 Set-abstraction

Set-abstraction is a mechanism for realizing the all-solution-search feature in a parallel environment. The following two predicates have been proposed [Fujitsu 84].

```
eager_enumerate({X|Goals}, L)
lazy_enumerate({X|Goals}, L)
```

In the above description, "Goals" is the sequence of the goals defined in a

Pure Prolog world. We assume that the Pure Prolog world is defined as follows:

```
pp((<head> <- <body>)).
```

That is, the Pure Prolog world is asserted as the set of "facts" which have a functor name "pp."

These two "enumerate" predicates solve the Goals in the Pure Prolog world and put the set of all solutions in L in stream form. The following is an example of "eager_enumerate."

```
eager_enumerate({X|grand_child(jiro,X)}, L)
```

We assume that the Pure Prolog world is defined as follows:

```
pp((grand_child(X,Z) <- child(X,Y),child(Y,Z))).
pp((child(jiro,keiko) <- true)).
pp((child(yoko,takashi) <- true)).
pp((child(jiro,yoko) <- true)).
pp((child(keiko,makoto) <- true)).
```

In this case, L is instantiated as [takashi,makoto]. The difference between "eager" and "lazy" is the way it instantiates the second argument. "eager_enumerate" instantiates it actively. On the other hand, "lazy_enumerate" instantiates it passively in accordance with the request from the stream consumer. In the following example, a solution list "L" is created in accordance with the request from "display."

```
:- lazy_enumerate({X|prime(X)}, L?),
     display(L, Mes?), keyboard(Mes).
```

2.3 Meta-inference

Meta-inference means to solve a given goal using knowledge defined in a user-defined world [Furukawa 84]. We set up the predicate "simulate" with the following form.

```
simulate(World, Goals, Result, Control)
```

Here, "World" is the name of a world, "Goals" is the goal sequence to be solved, "Result" is the computation result, and "Control" is the stream through which we can stop and resume the computation. We assume that knowledge of the world is given as a set of facts whose principal functors are the name of the world. That is, knowledge of the world has the following format.

```
world_name((<Head> <- <Guard> | <Body>)).
```

As an example of meta-inference, we give the "shell" example [Clark 84] which can run the foreground and background jobs. In this example, "&" shows the sequential operator. The foreground job always checks its control information while running. The background job runs steadily without looking up its control information.

```
shell([], _).
shell([fg(G)|N],C) :-
      simulate(f_world,G,R,C)&
      remove(C, NewC)&
      shell(N?,NewC).
shell([bg(G)|N],C) :-
      simulate(b_world,G,R,_),
      shell(N?,C).

:- shell([fg(problemA),bg(problemB)],C), control(C).
```

In this example, "problemA" defined in "f_world" runs on foreground and "problemB" defined in "b_world" runs on background. Execution of the foreground job is controlled by "C."

3 SCHEDULING QUEUE

A "scheduling queue" is often used in implementing a parallel logic language on a sequential machine. As mentioned before, AND-parallelism is the mechanism which evaluates "goals" in parallel. This is easily implemented by using a scheduling queue. The basic algorithm for the usage of the scheduling

queue is as follows:

(1) "goals" which should be solved are enqueued to the tail of the scheduling queue.

(2) A "goal" is dequeued from the head of the scheduling queue.

(3) A dequeued "goal" is solved. If the principal functor of the goal is a system predicate, the goal is solved immediately. If the principal functor of the goal is a user-defined predicate, the goal is reduced to new "goals."

(4) Newly created goals are appended to the tail of the scheduling queue.

In general, the computation model of CP can be expressed by AND-OR tree. AND-OR tree consists of two kinds of nodes, i.e., AND-goal-nodes and OR-clause-node. Shapiro's interpreter [Shapiro 83] processes AND-goal-nodes by creating a scheduling queue for each AND-relation. Therefore, AND-parallelism has been implemented in his interpreter. However, OR-relation is processed by backtracking. OR-parallelism is not realized in his interpreter.

Our ECP compiler has only one scheduling queue and all the AND-related goals and all the OR-related clauses are enqueued to this global queue. It is possible to handle all kinds of AND-relations and OR-relations in a uniform manner. In this paper, focusing on the role of the "scheduling queue," we outline the implementation method for realizing extended features, and show how one can nicely handle those features in a uniform manner.

4 ECP COMPILER

Our compiler translates the "ECP source program" to "Prolog" program. Since we already have the "Prolog compiler" which translates a "Prolog" program to the machine language, the "ECP source program" can be translated to the machine language.

"ECP program" and "Prolog program" have lots of similarity. Therefore, the translation of the former to the latter is much simpler than the direct translation to the machine language.

4.1 Program Compilation

At first, we show the overall compilation strategy.

Comparing ECP "compiler" with ECP "interpreter," we notice that there is no change with the scheduling of goals. The ECP "compiler" can easily be made from the ECP "interpreter" by changing the following points.

(1) Add scheduling queue to "goals".

In the compiled program, every goal is modified to include scheduling queue in itself. For example, if a goal is "goal(Args)," this goal is compiled as "goal(Args, World, Qs)." Two arguments are always added to the original ECP goal. One is the world name where the goal should be solved. If nothing is specified with world name, world name "*" which shows the global database world is automatically assigned. The other is the variable which will be unified with the current scheduling queue when the goal is dequeued from the scheduling queue.

(2) Add scheduling queue to "markers."

We have prepared various "marker" to realize the extended features of ECP. We also add an argument "Qs" to "marker" in order to process it as exactly the same manner as the ordinary "goal." If the original marker is "marker(Args)," the compiled marker becomes "marker(Args, Qs)."

(3) Every process is put on the scheduling queue in the form of "$(Element, Qs)."

Processes in the scheduling queue are expressed as a binary term whose principal functor is "$." Note that the "Element" is either "goal" or "marker," i.e., if "Element" is the goal "goal(Args)," the enqueued process becomes "$(goal(Args, World, Qs), Qs)." The same variable appears twice as the first argument of "goal" and as the second argument of the enqueued "process." This form makes the goal easier to get the current scheduling queue when it is taken out from the queue. At that time,

the second argument of the process is unified with the current scheduling queue by head unification. Since Qs is the shared variable, this results the goal "goal(Argument, World, Qs)" to have the current scheduling queue.

In summary, the main difference of the ECP compiler from the ECP interpreter is that all "Elements" keeps the current scheduling queue as its argument.

4.2 Compiled Code and Its Execution

The ECP compiler compiles the ECP programs to Prolog programs. The followings are the rough outline how various extended features of ECP can be compiled and executed.

4.2.1 OR-parallelism

The following two arguments are added to the compiled ECP clauses.

(1) The world name to which the given clause belongs. If the clause is defined in the global database world, world name "*" is assigned as a default value.

(2) Scheduling queue the tail of which OR-clauses are expanded with markers so that they can be processed in parallel.

The following ECP clauses

```
p(Args) :- G1 | B1.
p(Args) :- G2 | B2.
p(Args) :- G3 | B3.
```

is compiled as follows:

```
p(Args,
      '*', %World name
      [$(NextGoal, Qh\Qt)|Qh]\
      [$(♔(C,Qs1),Qs1),
            $($p$1(Args,C,Qs2),Qs2),
            $($p$2(Args,C,Qs3),Qs3),
            $($p$3(Args,C,Qs4),Qs4),
      $(♚(C,Qs5),Qs5)|Qt]) :- !, exec(NextGoal).
```

"Qsi" stand for the scheduling queue. D-list "Qh\Qt" is also used to express a scheduling queue. Note that every process in the scheduling queue has the form "$(Element, Qs)." OR-clauses from "p1" to "p3" are sandwiched in between the marker ♔ and ♚. The ECP compiler enumerates all the OR-clauses which have the same principal functor and generates the names from "p1" to "p3." The variable C contains the information whether one of the OR-clause is committed or not.

Each "pn" corresponds to the definition of original ECP program and it has the following format.[1]

```
$p$n(Args, C,
   [$(NextGoal, Qh\Qt)|Qh]\
   [$(♙(C,Fn,V,CVn,Qs1),Qs),
   <head unification processes>, <guard processes>,
         $(♟(Fn, [<body processes>|Bt]\Bt,
         Qs2),Qs2)|Qt]) :- !  exec(NextGoal).
```

"NextGoal" is used to get the goal which should be executed next. "Qh\Qt" express the renewed scheduling queue and it is passed to "NextGoal" by head-unification. The argument Fn of the markers ♙ and ♟ shows whether the n-th OR-clause has failed or not. The argument V is a list of variables

[1] You may notice that 'pn" need not be separated, i.e., we only need one big structure in which all OR-clauses are packed. The reason that we did not adopt this strategy comes from the regulations of DEC-10 Prolog compiler, i.e., DEC-10 Prolog Compiler does not accept the structure which includes more than 50 variables.

which contains all variables in the original goals. The argument CVn is the copied list of V. The body part of each clause is kept in the second argument of the marker ♔in D-list form. You may notice that processes between markers ♔and ♚are OR-related and processes between markers ♙and ♟are AND-related.

This compiled code is executed as follows:

(1) When a user-defined goal is called, it finds the definition clause for the given user-defined goals from the specified world. If it is found, enqueue the scheduled goals to the tail of the queue, dequeues a goal from the top of the queue, and executes this goal.

(2) When a system-defined goal is called, it computes the system defined goal. If it succeeds, next goal is dequeued from the top of the queue and executed.

(3) When a "marker" is called, it performs various computation depending on the markers and renew the scheduling queue. New goal is picked up from the queue and executed.

We should note that every "goal" or "marker" has scheduling queue in it. Every time new "goal" or "marker" is called, the renewed scheduling queue is put on it.

When "markers" are picked up, they are processes as follows:

(1) When marker ♔(C, Qs) or ♚(C, Qs) is picked up, the marker is aborted if "committed" is set in argument "C." Otherwise, the marker is put on the tail of the scheduling queue.

(2) When marker ♔is picked up and the top of the queue is marker ♚, i.e., the markers ♔and ♚are neighbors, this shows that all guards failed for a given goal. Since the "failure" of all guards means the "failure" of the given goal, "failure" is transmitted to the AND-relations to which they

belong.[1]

(3) When "pn" is picked up, it schedules the pre-scheduled goals to the tail of the scheduling queue, following the definition of "pn" .

(4) When marker ♙(C,Fn,V,CVn,Qs) is picked up, it checks whether "committed" is set in argument "C" or "failed" is set in argument "Fn." In these cases, all goals from ♙ to ♟ are removed from the scheduling queue.

(5) When marker ♙(C,Fn,V,CVn,Qs) is picked up and the top of the queue is marker ♟(Fn,Bn,Qs), i.e., the markers ♙(C,Fn,V,CVn,Qs) and ♟(Fn,Bn,Qs) are neighbors, it means that all goals of a guard succeed. In this case, we set "committed" to the argument C, unify V and CVn, and schedule Bn.

(6) When marker ♟(Fn,Bn,Qs) is picked up, the marker is simply put on the tail of the scheduling queue.

4.2.2 Set-abstraction

In the case of set-abstraction, there is no change in compiled code. "eager_enumerate" and "lazy_enumerate" are compiled in exactly the same manner as the ordinary goals.

When "eager_enumerate({X|p(X),q(X)},L)" is executed, this goal is reduced to the following processes and they are put on the tail of the scheduling queue.[2]

$(♔(Qs1),Qs1), $(♙♟(M,{X|p(X),q(X)},Qs2),Qs2), $(♚(M,L,Qs3),Qs3)

Two pairs of markers appear again. The meanings of these markers are slightly different from the previous ones. However it is still true that the markers ♔and ♚express OR-relation, and the marker ♙♟express AND-relation. The markers ♔and ♚surround the OR-relation and work as a

[1] If the goal is at the top level, it means the total failure of the computation.

[2] Although we chose to expand "set primitives" dynamically, it is possible to expand it at compilation time. This is also true for "meta-inference primitives."

solution collector. The solutions are collected in "L" in stream form. The marker ♙ compute one solution. The computed value is substituted into the argument "L."

These processes can be executed as follows:

(1) When marker ♔ is picked up and the top of the queue is marker ♚, i.e., the markers ♔ and ♚ are neighbors, this means that all solutions for the given goal have already been computed. We put [] onto the tail of the argument "L" in this case.

(2) When marker ♙(M,{X|p(X),q(X)},Qs2) is picked up, we find definition clauses for the leftmost goal of this set. If more than two clauses are found, it is broken up into several goals. The argument "M" is also reproduced by fission.

(3) When marker ♚(M,L,Qs3) is picked up, the argument "M" is checked. If it is instantiated, its value is sent to the stream "L" and the marker is appended to the tail of the scheduling queue.

The following is an example of fission. Assume that the marker is picked up, and P is defined in the Pure Prolog world as follows:

```
pp((p(X) <- B1,B2)).
pp((p(X) <- B3)).
pp((p(X) <- true)).
```

There are three clauses. The marker ♙ breaks up into three goals and they are appended to the scheduling queue in the following form:

```
$(♔(Qs1'),Qs1'),
$(♙(M1,{X|B1,B2,q(X)},Qs4),Qs4),
$(♙(M2,{X|p(B3,q(X)},Qs5),Qs5),
$(♙(M3,{X|q(X)},Qs6),Qs6),
$(♚([M1,M2,M3],L,Qs3'),Qs3')
```

We can get all solutions for the given goal by invoking fission. Notice that the

solutions are computed by the depth-first search based on OR-parallelism.

The basic mechanism of lazy-enumeration is almost the same as that of eager-enumeration. When "`lazy_enumerate({X|G},L)`" is executed, this goal is reduced to the following processes.

`$(♔(L, Qs1),Qs1), $(♙♟(M,{X|G},Qs2),Qs2), $(♚(Qs3),Qs3)`

As you notice, the form of the reduced processes are almost same, although the "markers" works slightly different. The variable "M" compute one solution. The variable L is the variables used for the bounded buffer communication to the outside world.

These "markers" are processed as follows:

(1) When marker ♔ is picked up, it checks whether "L" is a variable or not. If "L" is a variable, it means that there is no demand of solution yet. In this case, all goals from ♔ to ♚ are simply put onto the tail of the scheduling queue.

(2) When marker ♔ is picked up and "L" is instantiated to `[]`, it means that the demand from outside world is ended. In this case, all goals from ♔ to ♚ are removed from the scheduling queue.

(3) When marker ♔ is picked up and "L" is instantiated as `[X|L1]`, it checks the top of the scheduling queue. If the top of the queue is marker ♚, i.e., the markers ♔ and ♚ are neighbors, this means that all solutions for the given goal have already been computed. We instantiate X to "`$END_OF_SOLUTION$`," and all goals from ♔ to ♚ are removed from the scheduling queue.

(4) If the top of the queue is `(♙♟(M,{X|G},Qs2),Qs2)` in case (3), M is unified with X and `♙♟(M,{X|G},Qs2)` is executed after putting ♔(L1, Qs1') to the tail of the queue.

When `♙♟(M,{X|G},Qs2)` is executed, "G" is reduced to find a solution.

If a solution is found, it is substituted for X and all goals to ♚(Qs3) is moved to the tail of the queue. If the reduction fails, this goal is simply aborted and the next goal is executed from the queue.

4.2.3 Meta-inference

There is also nothing special with "meta-inference" predicates, "simulate" is compiled as same as the ordinary goals.

However, "simulate(W, (G1(Args1), G2(Args2)), R, C)" is called at execution time, this goal is reduced as follows:

$(♙(R,C,Qs1),Qs1),
$(G1(Args1,W,Qs3),Qs3),
$(G2(Args2,W,Qs4),Qs4),
$(♟(R,Qs2),Qs2)

Note that all processes between "markers" are defined in world "W."

The following summarizes the actions when markers are taken from the scheduling queue.

(1) When marker ♙(R,C,Qs1) is picked up and "failure" is already set in argument "R," all goals from ♙ to ♟ are removed from the scheduling queue.

(2) When marker ♙(R,C,Qs1) is picked up and the top of the queue is marker ♟(R,Qs2), i.e., it is empty between marker ♙ and marker ♟, we set "success" to the argument "R."

(3) When marker ♙(R,C,Qs1) is picked up and "C" is instantiated as [..., abort | variable], all goals from ♙ to ♟ are removed from the scheduling queue and "abortion" is set to the variable "R."

(4) When marker ♙(R,C,Qs1) is picked up and "C" is instantiated as [..., stop | variable], all goals from ♙ to ♟ are enqueued onto the tail of the scheduling queue without reducing these goals.

(5) When marker 🧍(R,C,Qs1) is picked up, and "C" is a variable or instantiated as `[ ..., cont | variable]`, the marker is just appended to the tail of the scheduling queue.

(6) When marker 🧍(R,Qs2) is picked up, the marker is appended to the tail of the scheduling queue.

Just as before, the markers 🧍 and 🧍 express AND-relation. If a goal between 🧍 and 🧍 fails, "failure" is set to "R." Goals between 🧍 and 🧍 are processed as exactly the same manner as the ordinary goals, except that goals are reduced in a specified world. No special problems are created even if OR-parallelism, set abstraction and meta-inference are nested within each other.

5 RELATED WORKS

Here, we would like to survey related works on ECP.

Generally speaking, the language specification of ECP's extended features is based on the conceptual specification of Kernel Language Version 1 (KL1) at ICOT [Furukawa 84]. The related works of each extended feature can be summarized as follows:

(1) For OR-parallelism, Levy [Levy 84] proposed the CP interpreter using a global queue. His interpreter is based on the lazy copying scheme. ICOT also implemented various CP interpreters which realized OR-parallelism using several implementation schemes [Miyazaki 85, Sato 84, Tanaka 84].

(2) The research in set abstraction is preceded by POPS [Hirakawa 84]. POPS is a Pure Prolog interpreter written in Concurrent Prolog. It enumerates all solutions for the given goals in stream form. In our approach, the enumeration of all solutions is directly realized by the scheduling queue.

(3) The key issue in meta-inference is how to implement the interpreter of the target language. In this field, research has been done by writing meta-interpreters [Shapiro 84, Clark 84]. We have implemented meta-inference predicates directly onto the scheduling queue. Compared with

the traditional approach, our approach is more direct.

In relate to the "compiler," our ECP compiler is greatly effected by the CP Compiler written by Chikayama and Ueda [Ueda 85a]. Our compiler is essentially the "revised" version of their CP compiler to allow OR-parallelism and various extended features of ECP.

After finishing up our compiler, we knew that Clark and Gregory [Clark 85] also made the Parlog compiler which compiles Parlog program to Prolog. We also happened to know that Murakami and Miyazaki designed the similar GHC compiled code which allows OR-parallel execution [Murakami 85].

6 CONCLUSION

In this paper, we described the various extended features of ECP and its compilation. Although we have omitted here, there are various problems which occur in the actual implementation, such as the copying variables, suspension of head unification, etc..

As mentioned before, our ECP "compiler" converts ECP source program to Prolog program. Although it is impossible to remove the scheduling queue, we see all guard and body goals are completely pre-scheduled to the queue in our "compiled" program. Therefore we can expect the speed up of the "compiled" program compared to the interpretive execution of the program.

The current version of our ECP compiler only compiles the scheduling. However, we can expect further optimization of this compiler. Examples of such optimization are as follows:

(1) The compilation of unification.

When enqueuing the head unification processes, we can call specialized unifiers such as "ulist," "uvect," "uatom," instead of calling general unifier "unify."

(2) The compilation of the immediate guard.

If the guard part of a clause only consists of system functions, we can solve it immediately instead of enqueuing all OR-clauses to the queue.

These compilation techniques are already implemented in [Ueda 85a] or [Miyazaki 85] and the effect of these optimizations are proved to be very effective. We can adopt these techniques without any difficulty.

By the way, the scope of this "single queue compilation" method is not limited to Concurrent Prolog. This method is also applicable to GHC [Ueda 85b]. In this case, the implementation becomes simpler because it does not generate multiple environments in implementing OR-parallelism.

ACKNOWLEDGMENTS

This research was carried out as a part of the Fifth Generation Computer Project. We would like to thank Kazunori Ueda, Toshihiko Miyazaki and other members of the KL1 implementation group at ICOT for their useful comments and suggestions. We would also like to thank Dr. Furukawa, the chief of the First Research Laboratory, ICOT, Dr. Kitagawa, the president of IIAS-SIS, Fujitsu, Dr. Enomoto, the director of IIAS-SIS, Fujitsu, and Mr. Yoshii, Fujitsu Social Science Laboratory, for giving us the opportunity to pursue this research and helping us with it.

REFERENCES

Clark K, Gregory S (1984) Notes on Systems Programming in Parlog. Proceedings of the International Conference on Fifth Generation Computer Systems 299-306

Clark K, Gregory S (1985) PARLOG: Parallel Programming in Logic. Research Report DOC 84/4. Department of Computing, Imperial College of Science and Technology. Revised June 1985

Fujitsu (1984) The Verifying Software of Kernel Language Version 1 –Detailed Specification– PART II. In: The 1983 Report on Committed Development on Computer Basic Technology, in Japanese

Fujitsu (1985) The Verifying Software of Kernel Language Version 1 – the Revised Detailed Specification and the Evaluation Result–, PART I. In: The 1984 Report on Committed Development on Computer Basic Technology, in Japanese

Furukawa K et al. (1984) The Conceptual Specification of the Kernel Language Version 1. Technical Report TR-054. ICOT

Hirakawa H et al. (1984) Eager and Lazy Enumeration in Concurrent Prolog. Proceedings of the Second International Logic Programming Conference 89-100

Levy J (1984) A Unification Algorithm for Concurrent Prolog. Proceedings of the Second International Logic Programming Conference 333-341

Miyazaki T et al. (1985) A Sequential Implementation of Concurrent Prolog Based on Shallow Binding Scheme. Proceedings of 1985 Symposium on Logic Programming 110-118

Murakami K (1985) The study of "unifier" implementation in multi-processor environment. Multi-SIM study group internal document, ICOT

Sato H et al. (1984) A Sequential Implementation of Concurrent Prolog - based on the Deep Binding Scheme. Proceedings of the First National Conference of Japan Society for Software Science and Technology 299-302, in Japanese

Shapiro E (1983) A Subset of Concurrent Prolog and its Interpreter. Technical Report TR-003. ICOT

Shapiro E (1984) Systems Programming in Concurrent Prolog. Conference Record of the 11th Annual ACM Symposium on Principles of Programming Language 93-105

Tanaka J et al. (1984) A Sequential Implementation of Concurrent Prolog –

based on the Lazy Copying Scheme. Proceedings of the First National Conference of Japan Society for Software Science and Technology 303-306, in Japanese

Tanaka J et al. (1985) AND-OR Queuing in Extended Concurrent Prolog. Proceedings of the Logic Programming Conference '85 215-224, in Japanese

Ueda K, Chikayama T (1985a) Concurrent Prolog Compiler on Top of Prolog. Proceedings of 1985 Symposium on Logic Programming 119-126

Ueda K (1985b) Guarded Horn Clauses. Technical Report TR-103. ICOT

APPENDIX A COMPILATION EXAMPLE

We show the ECP source of "merge" program and its compiled code as an example. Note that ECP compiler automatically generates names such as '$merge$m$n' where 'm' shows the arity of that predicate and 'n' shows its OR-clause number.

```
/* Source code of Merge program in ECP */

  merge([],Y,Y).
  merge(X,[],X).
  merge([X|Xs],Y,[X|Z]) :- true | merge(Xs?,Y?,Z).
  merge(X,[Y|Ys],[Y|Z]) :- true | merge(X?,Ys?,Z).


/* Compiled code */

  :-fastcode.

  :-public merge/5.
  merge(A,B,C,*,
      [$(D,E\F)|E]\
      [$( '$GS'(G,H),H),
         $( '$merge$3$1'(A,B,C,G,I),I),
         $( '$merge$3$2'(A,B,C,G,J),J),
         $( '$merge$3$3'(A,B,C,G,K),K),
         $( '$merge$3$4'(A,B,C,G,L),L),
       $( '$GE'(G,M),M)|F]):- ! ','exec(D).


  :-public '$merge$3$1' /5.
  '$merge$3$1'(A,B,C,G,
       [$(D,E\F)|E]\
       [$( '$G'(G,H,[A,B,C],[I,J,K],L),L),
        $(u(I,[],M),M),$(u(J,N,O),O),$(u(K,N,P),P),
        $( '$G'(H,Q\Q,R),R)|F]):- ! ','exec(D).

  :-public '$merge$3$2' /5.
  '$merge$3$2'(A,B,C,G,
       [$(D,E\F)|E]\
       [$( '$G'(G,H,[A,B,C],[I,J,K],L),L),
        $(u(I,M,N),N),$(u(J,[],O),O),$(u(K,M,P),P),
        $( '$G'(H,Q\Q,R),R)|F]):- ! ','exec(D).
  :-public '$merge$3$3' /5.
  '$merge$3$3'(A,B,C,G,
       [$(D,E\F)|E]\
       [$( '$G'(G,H,[A,B,C],[I,J,K],L),L),
        $(u(I,[M|N],O),O),$(u(J,P,Q),Q),$(u(K,[M|R],S),S),
        $( '$G'(H,[$(merge(N?,P?,R,*,T),T)|U]\U,V),
           V)|F]):- ! ','exec(D).

  :-public '$merge$3$4' /5.
  '$merge$3$4'(A,B,C,G,
       [$(D,E\F)|E]\
       [$( '$G'(G,H,[A,B,C],[I,J,K],L),L),
        $(u(I,M,N),N),$(u(J,[O|P],Q),Q),$(u(K,[O|R],S),S),
        $( '$G'(H,[$(merge(M?,P?,R,*,T),T)|U]\U,V),
           V)|F]):- ! ','exec(D).
```

APPENDIX B ECP COMPILER PROGRAM

We show the ECP compiler program. In the actual implementation, we use different "markers" instead of them shown in the main part of this paper. The correspondence between these "markers" are as follows:

- OR-parallelism

♔(C,Qs)	...	$GS(C,Qs)
♘(C,Fn,V,CVn,Qs)	...	$G(C,Fn,V,CVn,Qs)
♟(Fn,Bn,Qs)	...	$G(Fn,Bn,Qs)
♚(C,Qs)	...	$GE(C,Qs)

- Set-abstraction

Eager-enumerate

♔(Qs)	...	$SSET(Qs)
♘♟(M,{X|...},Qs)	...	$SET(M,{X|...},Qs)
♚(M,L,Qs)	...	$ESET(M,L,Qs)

Lazy-enumerate

♔(L,Qs)	...	$LSSET(L,Qs)
♘♟(M,{X|...},Qs)	...	$LSET(M,{X|...},Qs)
♚(Qs)	...	$LESET(Qs)

- Meta-inference

♘(R,C,Qs)	...	$SIMU(R,C,Qs)
♟(R,Qs)	...	$SIMU(R,Qs)

```
:- fastcode.

/* Top level */

:- public (kl)/1.
:- mode kl(+).

kl(X) :- settime,
    solve(X,R),
    nl,write('Result'(R)),nl,
    prtime,!.

:- mode solve(+).
solve(X) :- solve(X,R),!.

:- mode solve(+,-).
solve(X,R) :-
   c_schedule(*,X,Y\Y,
              [$(Goal,Qh\Qt)|Qh]\
              [$('$SIMJ'(R,Q1),Q1),
               $('$SIMJ'(R,**,Q2),Q2)|Qt]),
   !,exec(Goal).


:- mode exec(+).
exec(X) :-
   incore(X),!.
exec(X) :-
   functor(X,F,A),
   arg(A,X,Q),
   dequeue_failed_goals(Q,Goal),!,
   exec(Goal).


/* Flags */

:- public '$SIMJ'/3.
:- mode '$SIMJ'(?,+,+).

'$SIMJ'(success,_,
   [$('$SIMJ'(success,_),_)]\[]) :- !.
'$SIMJ'(Res,_,Qh\Qt) :-
    Res==failure ,!,
    dequeue_simulate(Qh,Qh1),
    (Qh1==Qt ;
      Qh1=[$(Goal,Qh2\Qt)|Qh2],!,
      exec(Goal) ).
'$SIMJ'(success,_,[$('$SIMJ'(success,_),_),
                   $(Goal,Qh\Qt)|Qh]\Qt) :- !,
    exec(Goal).

'$SIMJ'(Res,Cntl,
         [$(Goal,Qh\Qt)|Qh]\
        [$('$SIMJ'(Res,Cntl,Q),Q)|Qt]) :-
    (var(Cntl) ; Cntl==** ),!,
    exec(Goal).
'$SIMJ'(abortion,Cntl,Qh\Qt) :-
    mem_abort(Cntl),
    dequeue_simulate(Qh,
                          [$(Goal,Qh1\Qt)|Qh1]),!,
    exec(Goal).
'$SIMJ'(Res,[stop,Cont|Cntl],
         [$(Goal,Qh\Qt)|Qh]\
        [$('$SIMJ'(Res,Cntl,Q),Q)|Qt]) :-
    Cont==cont ,!,
    exec(Goal).
'$SIMJ'(Res,[stop|Cntl],
         Qh\
         [$('$SIMJ'(Res,
                    [stop|Cntl],Q),Q)|Qt]) :-
    skip_simulate(Qh\Qt,Goal),!,
    exec(Goal).
'$SIMJ'(Res,[cont|Cntl],
         [$(Goal,Qh\Qt)|Qh]\
        [$('$SIMJ'(Res,Cntl,Q),Q)|Qt]) :- !,
    exec(Goal).

:- public '$SIMJ'/2.
:- mode '$SIMJ'(+,+).

'$SIMJ'(Res,[$(Goal,Qh\Qt)|Qh]\
            [$('$SIMJ'(Res,Q),Q)|Qt]) :- !,
    exec(Goal).

:- public '$GS'/2.
:- mode '$GS'(+,+).

'$GS'(Cmmt,[$(Goal,Qh\Qt)|Qh]\Qt) :-
    Cmmt==committed ,!,
    exec(Goal).
'$GS'(_,[$('$GE'(_,_),_)|Qh]\Qt) :-
    dequeue_failed_goals(Qh\Qt,Goal),!,
    exec(Goal).
'$GS'(Cmmt,[$(Goal,Qh\Qt)|Qh]\
            [$('$GS'(Cmmt,Q),Q)|Qt]) :- !,
    exec(Goal).

:- public '$GE'/2.
:- mode '$GE'(+,+).

'$GE'(Cmmt,[$(Goal,Qh\Qt)|Qh]\Qt) :-
    Cmmt==committed ,!,
    exec(Goal).
'$GE'(Cmmt,[$(Goal,Qh\Qt)|Qh]\
            [$('$GE'(Cmmt,Q),Q)|Qt]) :- !,
    exec(Goal).

:- public '$G'/5.
:- mode '$G'(?,+,+,+,+).

'$G'(Cmmt,Fail,_,_,Q) :-
    (Cmmt==committed
    ;Fail==failed),
    dequeue_guard(Q,Goal),!,
    exec(Goal).
'$G'(committed,_,CV,CV,
      [$('$G'(_,Qt\Qt1,_),_),
       $(Goal,Qh\Qt1)|Qh]\Qt) :-
```

```
    unify(OV,CV),!,
    exec(Goal).
'$G'(Cmmt,Fail,[],[],
     [$(Goal,Qh\Qt)|Qh]\
     [$('$G'(Cmmt,Fail,[],[],Q),Q)|Qt]) :- !,
    exec(Goal).
'$G'(Cmmt,Fail,CV,CV,
     [$(Goal,Qh\Qt)|Qh]\
     [$('$G'(Cmmt,Fail,CVV,CVV,Q),Q)|Qt]) :-
    copy(OV,CV,OVV,CVV),!,
    exec(Goal).
'$G'(_,_,_,_,Q) :-
    dequeue_guard(Q,Goal),!,
    exec(Goal).

:- public '$G'/3.
:- mode '$G'(+,+,+).

'$G'(Fail,BQ,
     [$(Goal,Qh\Qt)|Qh]\
     [$('$G'(Fail,BQ,Q),Q)|Qt]) :- !,
    exec(Goal).

:- public '$SSET'/1.
:- mode '$SSET'(+).

'$SSET'([$('$ESET'([],[],_),_),
         $(Goal,Qh\Qt)|Qh]\Qt) :- !,
   exec(Goal).
'$SSET'([$(Goal,Qh\Qt)|Qh]\
        [$('$SSET'(Q),Q)|Qt]) :- !,
   exec(Goal).

:- public '$SET'/3.
:- mode '$SET'(-,+,+).

'$SET'(Mes,Cls,[$(Goal,Qh\Qt)|Qh]\Qt1) :-
   c_reduce_set(Cls,Mes,Qt1,Qt),!,
   exec(Goal).

:- public '$ESET'/3.
:- mode '$ESET'(+,-,+).

'$ESET'(Mes,S,
        [$(Goal,Qh\Qt)|Qh]\
        [$('$ESET'(Mes1,S1,Q),Q)|Qt]) :-
   collect_s(Mes,Mes1,S,S1),!,
   exec(Goal).

:- public '$LSSET'/2.
:- mode '$LSSET'(?,+).

'$LSSET'(S,Qh\[$('$LSSET'(S1,Q),Q)|Qt]) :-
   cvar(S,S1),!,
   skip_set(Qh\Qt,Goal),!,
   exec(Goal).
'$LSSET'([],Q) :- !,
   dequeue_set(Q,Goal),!,
   exec(Goal).

'$LSSET'(['$END_OF_SOLUTION$'|_],
         [$('$LESET'(_),_),
          $(Goal,Qh\Qt)|Qh]\Qt) :- !,
   exec(Goal).
'$LSSET'(S,[$('$LSET'(0,Cls,_),_)|Qh]\
           [$('$LSSET'(S1,Q),Q)|Qt]) :-
   wait(S,[0|S1]),!,
   '$LSET'(0,Cls,Qh\Qt).

:- public '$LSET'/3.
:- mode '$LSET'(-,+,+).

'$LSET'(0,Cls,Qh\Qt) :-
   c_1_reduce_set(Cls,0,Qt\Qt1),!,
   skip_set(Qh\Qt1,Goal),!,
   exec(Goal).
'$LSET'(0,_,
        [$('$LSET'(0,Cls,_),_)|Qh]\Qt) :-
   '$LSET'(0,Cls,Qh\Qt).
'$LSET'('$END_OF_SOLUTION$',_,
         [$(Goal,Qh\Qt)|Qh]\Qt) :- !,
   exec(Goal).

:- public '$LESET'/1.
:- mode '$LESET'(+).

'$LESET'([$(Goal,Qh\Qt)|Qh]\
         [$('$LESET'(Q),Q)|Qt]) :- !,
   exec(Goal).



/* Set */

c_reduce_set(Cls,Mes,T,T1) :-
   reducep(Cls,NextCls),!,
   c_fork_set(Cls,NextCls,Mes,T,T1),!.
c_reduce_set(Cls,'$SOL'(Mes),T,T) :-
   terminatep(Cls,Mes),!.
c_reduce_set(Cls,Mes,
             [$('$SET'(Mes,NCls,
                       Q1),Q1)|T],T) :-
   systemp(Cls,NCls),!.
c_reduce_set(_,'$FAIL$',T,T) :- !.


:- mode c_1_reduce_set(+,-,?).

c_1_reduce_set(Cls,0,T\T1) :-
   reducep(Cls,NextCls),!,
   c_1_fork_set(Cls,NextCls,0,T\T1).
c_1_reduce_set(Cls,Ans,T\T) :-
   terminatep(Cls,Ans),!.
c_1_reduce_set(Cls,0,T\T1) :-
   systemp(Cls,NCls),!,
   c_1_reduce_set(NCls,0,T\T1).
```

```
/* Simulate */

:- public simulate/6.
:- mode simulate(+,+,-,+,+,+).

simulate(World,Goal,Res,Contorl,_,
          [$(NextGoal,Qh\Qt)|Qh1]\
          [$('$SIMU'(Res,Cntl,
                     Q),Q)|Qt1]) :-
    c_schedule(World,Goal,Qh1\Qt1,
               Qh\[$('$SIMU'(Res,Q1),
                     Q1)|Qt]),
    exec(NextGoal),!.


/* Compiler */

:- public comp/2.

comp(IF,OF) :-
    c_assert(IF,[]\NL),
    reverse(NL,RNL),
    c_c(OF,RNL),
    abolish_all(NL),!.


c_assert([],L\L) :- !.

c_assert([F|R],L0\L) :-
    c_assert(F,L0\L1),
    c_assert(R,L1\L).

c_assert(F,L0\L) :-
    seeing(OF),see(F),
    read(X),c_assert1(X,L0\L),
    seen,see(OF).


c_assert1(end_of_file,L\L) :- !.

c_assert1((Head :- Body),L0\L) :-
    functor(Head,F,A),
    check_member(F/A,L0\L1),
    assertz((Head :- Body)),
    read(X),c_assert1(X,L1\L).

c_assert1(Cls,L0\L) :-
    Cls=..[World,(Head <-- Body)],!,
    functor(Head,F,A),
    check_member((World,F/A),L0\L1),
    assertz(Cls),
    read(X),c_assert1(X,L1\L).

c_assert1(Head,L0\L) :-
    functor(Head,F,A),
    check_member(F/A,L0\L1),
    assertz(Head),
    read(X),c_assert1(X,L1\L).


c_c(F,NL) :-
    telling(OF),tell(F),
   write((:- fastcode)),
   writenl('.'),nl,
    c_clause(NL),
    told,tell(OF).


c_clause([]) :- !.

c_clause([F/A|NL]) :-
    functor(P,F,A),
    bagof((P <-- B),clause(P,B),Clist),
    c_clauses(F/A,Clist,*),
    c_clause(NL).

c_clause([(W,F/A)|NL]) :-
    functor(P,F,A),
    Cls=..[W,(P <-- B)],
    bagof((P <-- B),Cls,Clist),
    c_clauses(F/A,Clist,W),
    c_clause(NL).


c_clauses(F/A,[Clause],W) :-
   A2 is A+2,
   functor(Head,F,A2),
   Head=..[F|Args],
   functor(Dummy,F,A),
   Dummy=..[F|DArgs],
   append(DArgs,
          [W,[$(Goal,Qh\Qt)|Qh]\Qh1],Args),
   c_each_clause(Clause,DArgs,W,C,RQh\RQt),
   (C=='committed',
    Qh1=RQh,
    Qt=RQt
   ;Qh1=[$('$GS'(C,Q1),Q1)|RQh],
    RQt=[$('$GE'(C,Q2),Q2)|Qt]),
   writeq((:- public F/A2)),
   put("."),
   nl,
   pretty_print((Head :- !,exec(Goal))).
c_clauses(F/A,Cls,W) :-
   A2 is A+2,
   functor(Head,F,A2),
   Head=..[F|Args],
   functor(Dummy,F,A),
   Dummy=..[F|DArgs],
   append(DArgs,
          [W,[$(Goal,Qh\Qt)|Qh]\Qh1],Args),
   c_or_clauses(1,F/A,Cls,DArgs,W,
        Cmpd_Cls,CQh\CQt,Cmmt_Flag),
   (member_committed(Cmmt_Flag,N),
    picup_queue(N,Cmpd_Cls,Qh1\Qt),
    writeq((:- public F/A2)),
    put("."),
    nl,
    pretty_print((Head :- !,exec(Goal)))
```

```
    ;link_commit(Cmmt_Flag,C),
     Qh1=[$('$GS'(C,Q1),Q1)|CQh],
     CQt=[$('$GE'(C,Q2),Q2)|Qt],
     writeq((:- public F/A2)),
     put("."),
     nl,
     pretty_print((Head :- !,exec(Goal))),
     print_each_clause(Cmpd_Cls)).


c_or_clauses(K,F/A,[Clause|Clauses],
             Args,W,
             [(Head :- !,exec(Goal))|
              Cmpd_Cls],
             [$(Caller,Q)|Calls]\CQt,
             [C|Cs]) :-
   c_each_clause(Clause,Args,W,C,Qh1\Qt),
   append(Args,
          [C,[$(Goal,Qh\Qt)|Qh]\Qh1],NArgs),
   make_name(F,A,W,K,NF),
   Head=..[NF|NArgs],
   append(Args,[C,Q],CArgs),
   Caller=..[NF|CArgs],
   N is K+1,
   c_or_clauses(N,F/A,Clauses,Args,W,
                Cmpd_Cls,Calls\CQt,Cs).
c_or_clauses(_,_,[],_,_,[],CQt\CQt,[]) :- !.


c_each_clause((Head <-- VP),Args,
              W,C,RQh\RQt) :-
   var(VP),
   !,
   copyp(Args,CArgs),
   Head=..[_|HArgs],
   c_unif_queue(CArgs,HArgs,
          TQh\[$('$G'(F,
                     [$('$VP'(VP,Q3),
                        Q3)|BQt]\BQt,
                     Q2),Q2)|RQt]),
   queue_optimizer([$('$G'(C,F,Args,CArgs,Q1),
                     Q1)|TQh],RQh).
c_each_clause((Head <-- Guard : Body),
              Args,W,C,RQh\RQt) :-
   copyp(Args,CArgs),
   c_schedule(W,Body,X\X,BQ),
   c_schedule(W,Guard,Y\Y,
              GQh\[$('$G'(F,BQ,Q3),Q3)|RQt]),
   Head=..[_|HArgs],
   c_unif_queue(CArgs,HArgs,TQh\GQh),
   queue_optimizer([$('$G'(C,F,Args,CArgs,Q1),
                     Q1)|TQh],RQh).
c_each_clause((Head <-- Body),Args,
              W,C,RQh\RQt) :-
   copyp(Args,CArgs),
   c_schedule(W,Body,X\X,BQ),
   Head=..[_|HArgs],
   c_unif_queue(CArgs,HArgs,
                TQh\[$('$G'(F,BQ,Q2),Q2)|RQt]),
   queue_optimizer([$('$G'(C,F,Args,CArgs,Q1),
                     Q1)|TQh],RQh).


c_unif_queue([CA|CAs],[A|As],
             [$(u(CA,A,Q),Q)|TQh]\TQt) :-
   c_unif_queue(CAs,As,TQh\TQt).
c_unif_queue([],[],TQt\TQt).


:- public c_schedule/4.

c_schedule(W,G,
           Qh\[$('$VP'(G,W,Q),
                Q)|Qt],Qh\Qt) :-
    var(G),!.

c_schedule(W,(A,B),Q,Q2) :-
    c_schedule(W,A,Q,Q1),
    c_schedule(W,B,Q1,Q2).

c_schedule(_,true,Q,Q) :- !.

c_schedule(_,simulate(W,G,R,C),
           Qh\[$('$SIMJ'(R,C,Q),Q)|Qt1],
           Qh\Qt) :-
   c_schedule(W,G,Qh\Qt1,
              Qh\[$('$SIMJ'(R,Q1),Q1)|Qt]),!.

c_schedule(_,set({X:Goal},Str),
           Qh\[$('$SSET'(Q1),Q1),
               $('$SET'(Mes,{X:Goal},Q2),Q2),
               $('$ESET'(Mes,Str,Q3),Q3)|Qt],
           Qh\Qt) :- !.

c_schedule(_,lazy_set({X:Goal},Str),
           Qh\[$('$LSSET'(Str,Q1),Q1),
               $('$LSET'(_,{X:Goal},Q2),Q2),
               $('$LESET'(Q3),Q3)|Qt],
           Qh\Qt) :- !.

c_schedule(W,A,Qh\[$(CA,Q)|Qt],Qh\Qt) :-
   A=..[F|Args],
   append(Args,[W,Q],CArgs),
   CA=..[F|CArgs].
```