COMPILING EXTENDED CONCURRENT PROLOG -SINGLE QUEUE COMPILATION-[1]

Jiro Tanaka*, Makoto Kishishita**

* ICOT, Mita-kokusai-building 21F, 1-4-28, Mita, Minato-ku, Tokyo 108, Japan

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

ABSTRACT

Extended Concurrent Prolog (ECP) [Fujitsu 85, Tanaka 85a] 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 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" makes it possible to handle all kinds of AND-relations and OR-relations in a uniform manner.

1 INTRODUCTION

"Parallel Logic Languages," which permit concurrent programming and parallel execution, are currently attracting wide spread attention all over the world. Parlog [Clark 85], Concurrent Prolog (CP) [Shapiro 83] and GHC [Ueda 85b] are the examples of such "Parallel Logic Languages." Although there are differences, the basic computation mechanisms of these languages are quite similar: Horn clauses with guards are used to define predicates, goals are executed in parallel, and they have some synchronization mechanisms between goals.

In this paper, we consider the effective implementation method of these parallel logic languages. As an example of such parallel logic languages, we assume Extended Concurrent Prolog (ECP) [Fujitsu 85, Tanaka 85a], which is an variant of Concurrent Prolog (CP) with OR-parallel, set-abstraction and meta-inference features. Since we have already described the implementation of our ECP "interpreter" in [Tanaka 85a], we focus on the implementation of our ECP "compiler" in this paper. However, we should note that the techniques discussed in this paper can be applicable to all parallel logic languages such as GHC or Parlog. In theses cases, the implementation may become simpler because they do not generate multiple environments accompanied with OR-parallelism.

Although this paper assumes familiarity with CP and ECP, we briefly 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:

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

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 85].

```
        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.

2.3 Meta-inference

Meta-inference means to solve a given goal using knowledge defined in a user-defined world [Furukawa 84]. This primitive can be used to execute user program in system. By using this meta-predicate, system program can be protected from the failure. We have prepared 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, 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([bg(primes),fg(primes)],C), control(C).
```

In this example, the "primes" programs to compute the infinite sequence of prime numbers runs both foreground and background jobs. Execution of the foreground job can be controlled by "C."

2.4 Other features of ECP

Accompanied with the realization of OR-parallelism, we have introduced two important concepts in ECP. One is "otherwise" predicate proposed in [Shapiro 83]. This predicate could not have the sufficient meaning without implementing OR-parallelism. The other is the "write-early" annotation. In ECP, each OR-clause has its local environments to prevent clauses to interfering with each other. When one OR-clause is committed, its local environment is exported to the global environment. However, this invokes great inconveniences when we want to debug a program. There is no way to observe the computation until it is committed. The "write-early" annotation is a primitive introduced to solve such a problem. If a variable is annotated as "write-early," local environment is not created for that variable.

The other feature of ECP is its module structure. Module structure is very important when writing a large program. Module structure can be defined in ECP. It is based on message passing between modules like Logix [Shapiro 85]. This module structure has been designed taking into consideration ECP's distributed implementation in multi-processing environment. Multiple-inheritance can be defined in this module structure. The "part-of" relation can also be realized by throwing goals to other modules. Although we omit the detailed explanation in this paper, the detailed description is in [Tanaka 85c]

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. D-list is usually used to implement a scheduling queue. D-list is expressed by a pair of variables. The enqueuing and dequeuing are executed as follows:

(1) Enqueuing to the scheduling queue. If the current scheduling queue is expressed as "`Head\Tail`," unify this D-list with "`NewHead\[A|NewTail]`." This means to enqueue A to the scheduling queue. "`NewHead\NewTail`" expresses the renewed scheduling queue.

(2) Dequeuing from the scheduling queue. If the current scheduling queue is "`Head\Tail`," unify this D-list with "`[A|NewHead]\NewTail`." This means to dequeue one item from the scheduling queue and "`NewHead\NewTail`" expresses the renewed scheduling queue.

(3) Enqueuing and dequeuing can be carried out at the same time. If we want to enqueue "A1, A2, A3" and dequeue "B1, B2," we just need to unify the current scheduling queue with "`[B1,B2|NewHead]\[A1,A2,A3|NewTail]`."

Our ECP compiler has only one scheduling queue. All the AND-related goals and all the OR-related clauses, which appears in executing a program, are enqueued to this global queue. On the other hand, in Shapiro's CP interpreter [Shapiro 83], a scheduling queue is created for each AND-relation. And OR-parallelism has not been implemented.

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

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.

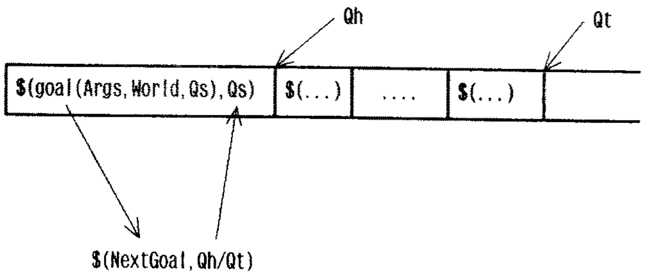
(2) Add scheduling queue to "markers."

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)."

Note that the same variable "Qs" appears twice in enqueued form. This form makes the goal easier to get the current scheduling queue when it is taken out from the queue. The following figure shows the instance when a "process" is taken out from the scheduling queue.



Here, "Qs" is unified with the current scheduling queue "Qh\Qt." Since "Qs" is the shared variable, this results the goal "goal(Argument, World, Qs)" to have the current scheduling queue.

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.

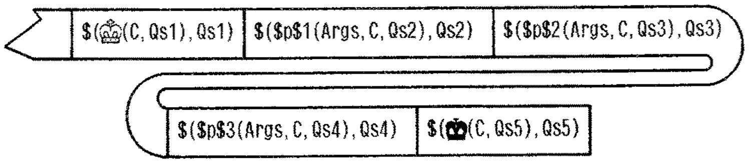
The following ECP clauses

```
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).
```

This means to put goals at the tail of the scheduling queue in the following form.



Here, "Qsi" stand for the scheduling queue. Note that every process in the 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 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

[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.

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 "`$p$n`" is picked up, it schedules the pre-scheduled goals to the tail of the scheduling queue, following the definition of "`$p$n`" .

(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 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."

[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."

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. However, we omit the detailed description because of space limitation.

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 ECP PROGRAM EXAMPLE

We examine the "shell" program in this section. This is a more realistic version of the shell program discussed in 2.3. The shell program in 2.3 can run only one foreground job and multiple background jobs. We can control the execution only for the foreground job. However, in our "realistic" version, there is no distinction between foreground and background jobs, so we can run and control multiple jobs at the same time. In this "shell" program, every job has a process-ID and the execution of jobs can be controlled by commands which include process-IDs. A job may be aborted, suspended and resumed. The realistic "shell" program is shown below:

```
(1) shell :- shell(I?, []), in(I).

(2) shell([proc(ID,Goals)|Input], IDlist) :-
       true | print_result(IDlist,IDlist1),
              print_process(ID,Goals),
              simulate(*, Goals, R, C),
              shell(Input?, [(ID,R,C)|IDlist1]).

    shell([wproc(ID,W,Goals)|Input], IDlist) :-
       true | print_result(IDlist,IDlist1),
              print_wproc(ID,W,Goals),
              simulate(W, Goals, R, C),
              shell(Input?, [(ID,R,C)|IDlist1]).

    shell([Com | Input], IDlist) :-
       otherwise | print_result(IDlist,IDlist1),
              print_com(Com),
              send(IDlist1, Com, NewIDlist),
              shell(Input?,NewIDlist).
```

```
(3) send([],_,[]).
    send([(ID,R,C)|IDList],Com,
            [(ID,R,NewC)|IDList]) :-
                Com =..  [M,ID] | C = [M|NewC].
    send([(ID,R,C)|IDList],Com,
            [(ID,R,C)|NewIDlist]) :- otherwise |
                send(IDlist,Com,NewIDlist).
```

The meaning of this "shell" program is as follows:

(1) "shell" is the top level predicate. It calls the two-argument-"shell" and "in."

(2) Two-argument-"shell" is the main part of this program. The first argument of "shell" is a stream which receives commands from the goal "in." The second argument is the list of processes controlled by "shell." This list of processes is called "IDlist." A process is expressed as (ID,R,C), where ID is an identifier of a process, R is a variable which sends a message to the outside, and C is a variable which controls its execution.

This "shell" behaves as follows:

- When it receives the message "proc(ID, Goals)" at its first argument, it calls "simulate," executes "Goals" in the global database world, and adds this process "(ID,R,C)" to the "IDlist."
- When it receives the message "wproc(ID, W, Goals)" as its first argument, it calls "simulate," executes "Goals" in world "W," and adds this process "(ID,R,C) to the "IDlist."
- When it receives other commands, such as "stop(ID)," "cont(ID)" or "abort(ID)," as its first argument, it sends that command to the control variable of the specified process.

The predicates "print_process," "print_wproc" and "print_com" are used just for printing out the message which "shell" received. The predicate "print_result" is used to print out the result when a process is aborted or ends successfully. In such cases, it prints out the process termination information and removes that process from the given "IDlist."

(3) "send" transmits a message such as "stop(ID)," "cont(ID)" or "abort(ID)" to the process with a process identifier "ID." It looks for the "IDlist." If it finds the process, it sends the message to the control variable of that process.

The following is an execution example of this "shell" program. We invoked two processes. One is the process "p01" which generates the infinite sequence of prime numbers. The second is the process "p02" which computes prime numbers up to 10 following the definition of prime in the world "s." In this example, we stopped "p01" after it printed out 2, 3 and 5, and resumed after "p02" printed out 2, 3, 5 and 7. The process "p02" is terminated after it prints out all primes up to 10. We also terminated process "p01" by sending the abort message to "p01."

```
?- solve(shell, R).
> proc(p01,primes)
> wproc(p02,s,prime(10))
2
3
        2 (s)
        3 (s)
5
> stop(p01)
        5 (s)
        7 (s)
> cont(p01)
> result([p02,success])
7
11
> abort(p01)
> result([p01,abortion])
```

6 RELATED WORKS

The language specification of ECP's extended features is based on the conceptual specification of Kernel Language Version 1 (KL1) at ICOT [Furukawa 84].

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 have designed the similar GHC compiled code which allows OR-parallel execution [Murakami 85].

7 CONCLUDING REMARKS

We have proposed the "Single Queue Compilation" method which compiles an ECP program to a Prolog program. It is surprising that the various features of ECP, such as OR-parallel, set-abstraction and meta-inference, can be implemented in a consistent manner. Our implementation has the following features:

(1) It realizes OR-parallelism.

(2) In set abstraction, we can reduce other goals while generating solutions.

(3) In meta-inference, we can compute several "simulate" predicates at the same time.

From the architectural point of view, our implementation is more "realistic" than Shapiro's implementation [Shapiro 83] because we use only-one-existing scheduling queue and scheduling queues are not created dynamically.

Actual implementation of our ECP compiler has done on DEC 2065. The programs are written in DEC 10 prolog and 430 lines long in total. The programs are consist of "code compilation part," "markers setting part," and "execution support part." The program sizes are approximately, 160 lines, 160 lines, and 110 lines, respectively.

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 in our "compiled" program.

The current version of our ECP compiler only compiles the scheduling. However, we can expect further optimization of this compiler. These 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.

ACKNOWLEDGMENTS

This research was carried out as a part of the Fifth Generation Computer Project. We would like to thank our colleague Takashi Yokomori, IIAS-SIS. Part of our research is indebted to him. We would also 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 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 (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

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

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

Shapiro E et al. (1985) Logix User Manual for Release 1.1. Weizmann Institute, Israel

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. (1985a) AND-OR Queuing in Extended Concurrent Prolog. Proceedings of the Logic Programming Conference '85 215-224, in Japanese. English version is to appear in Lecture Notes in Computer Science, Springer

Tanaka J et al. (1985b) Single Queue Compilation in Extended Concurrent Prolog. Mathematical Methods in Software Science and Engineering, RIMS Kokyuroku, Research Institute for Mathematical Science, Kyoto University

Tanaka J, Kishishita M. (1985c) Message Based Module Structure for Parallel Logic Languages. To appear in Proceedings of the Second National Conference of Japan Society for Software Science and Technology, 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).
```