TR-350

Deriving a Compilation Method for Parallel Logic Languages

by
Y. Kohda & J. Tanaka

March, 1988

Deriving a Compilation Method for Parallel Logic Languages

Youji KOHDA
International Institute for Advanced Study
of Social Information Science,
FUJITSU LIMITED
1-17-25, Shinkamata,
Ota-ku, Tokyo 144, JAPAN

Jiro TANAKA
Institute for New Generation
Computer Technology,
21F, Mita Kokusai Building
1-4-28, Mita,
Minato-ku, Tokyo 108, JAPAN

Abstract

It is already known that a Concurrent Prolog program can be compiled into an equivalent Prolog program. Using a Concurrent Prolog interpreter written in Prolog, we converted Concurrent Prolog programs step by step to compiled codes in Prolog. Each conversion step was successfully executed on a Prolog language processor. We examined how each conversion step contributed to performance improvement, using sample Concurrent Prolog programs such as a meta interpreter.

1. Introduction

Concurrent Prolog (CP) is an AND parallel logic language developed by Shapiro [3]. He also developed an interpreter for CP written in Prolog. The interpreter was considered significant as a working specification; however, it suffered from poor execution performance. Ueda and Chikayama have solved this problem by developing a compiling technique from CP to Prolog [4]. Their compilation technique is based on the scheme of Shapiro's CP interpreter. Therefore, it seems feasible to derive their compilation method from the CP interpreter scheme.

The concept of partial evaluation is familiar. It is widely known that the partial evaluation of an interpreter for language *L2* in language *L1* and a program in *L2* results in a compiled code in *L1* for that program. In our case, *L1* and *L2* correspond to Prolog and CP. However, we do not give a precise way to make a partial evaluator in this paper. Instead we will give a method of partial evaluation from CP to Prolog by *derivation*. It should be emphasized that a derivation path from CP to Prolog exists.

In program transformation, only an initial program is given, and it is transformed in small stages. The final program after program transformation will usually be too complicated to understand. This means that the correctness of each conversion step is very important in ensuring the correctness of the total transformation. This paper gives both initial and final programs, our interest lying in how to bridge the gap between them. This is why we use the term *derivation* instead of *transformation*.

We use an append program as a working example throughout this paper. We begin with Shapiro's CP interpreter and an append program in CP, and modify them step by step. Finally, the append is compiled into Prolog. This is identical to what Ueda and Chikayama have developed.

The derivation method is useful both theoretically and practically. It can help us to design a compiled code; each conversion step can be executed directly on a Prolog processor, and by observing the execution speed, the effectiveness of specific conversion methods can be studied. We can experiment with many new compiling techniques before deciding upon the final one. Kursawe gave a full account of this idea in his paper [2]. He showed that successive conversion of an append program in Prolog can finally yield a *Warren-type Abstract Prolog Machine Code*.

2. CP and CP Interpreters

A CP program is a set of *guarded Horn clauses*. A Horn clause has the form: *H*:-*G*|*B*. Here, *H* is a predicate called the *head*, *G* is a sequence of subgoals called the *guard* part, *B* is a sequence of subgoals called the *body* part. When G is *true*, the Horn clause is abbreviated as H:-B. When B is also *true*, it is simply written out as H. CP is characterized by the *commit* operator, |, introducing nondeterminism, and the *read-only* annotation, ?, introducing data driven control.

Computation in CP is a series of reductions. To reduce a given goal, first, choose a set of clauses each of whose heads is unifiable with the goal, including only the clauses whose guard part is recursively reducible. Second, randomly select one of these clauses. That is, the clause is *guarded* at the | from the rest in the set. Finally, replace the given goal by the body part of the selected clause. Each new goal is subjected to successive reductions. The commit operator, |, achieves coordination among parallel executing goals.

The symbol ? can be attached to any variable to indicate that the variable is used only for a *read-out* operation: a read-only variable cannot be unified with non-variables until some value is assigned to it. This assures a unidirectional flow of information. The read-only annotation, ?, achieves cooperation among parallel executing goals.

(1) CP interpreter in CP

The reduction process of CP can be written in CP as shown below [1]. This is a meta interpreter.

```
(M0)   mcall(true).                                               % halt
       mcall((A,B)) :- mcall(A?), mcall(B?).                      % fork
       mcall(G) :- system(G) | G.                                 % system
       mcall(G) :- nonsystem(G), G≠true, G≠(A,B) |                % reduce
                  reduce(G?,Body), mcall(Body?).
```

When clause goals conjoined by , are given, *mcall/M0*† reduces each goal recursively. When a single goal is given, it tries to reduce the goal. If the goal is *true*, it simply terminates. If the goal is a system-defined predicate, it executes the predicate directly. Otherwise, the reduction is carried out by *reduce*, and the returned body goals are solved recursively by *mcall/M0*.

(2) CP interpreter in Prolog

Shapiro's CP interpreter written in Prolog is shown below [3]. Notice that the overall program structure is similar to that of *mcall/M0*. The core of the interpreter consists of *solve/S0* and *reduce/R0*.

```
(S0)   solve(['$END'],_,_) :- !.                                  % halt
       solve(['$END'|H],[],d) :- !, fail.
       solve(['$END'|H],['$END'|T],nd) :- !,
                 solve(H,T,d).
       solve([A|H],T,_) :-                                        % system
                 system(A), !, A,
                 solve(H,T,nd).
       solve([A|H],T,F) :-                                        % reduce and fork
                 reduce(A,B,F,NF),
                 schedule(B,H,T,NH,NT), !,
                 solve(NH,NT,NF).
(R0)   reduce(A,B,_,nd) :-                                        % reduce
                 guarded_clause(A,G,B),
                 schedule(G,X,X,H,['$END'|T]),                    % create a new queue
                 solve(H,T,d), !.                                 % solve guard G in it
       reduce(A,suspended(A),F,F).
(G0)   guarded_clause(A,G,B) :-
```

† *The Name/Identifier* format is used to distinguish different versions of clauses.

```
                guarded_clause(A,B1), find_guard(B1,G,B).
        guarded_clause(A,B) :-
                functor(A,F,N), functor(A1,F,N),
                clause(A1,B), unify(A,A1).                        % CP unify
        find_guard((A|B),A,B) :- !.
        find_guard(A,true,A).
(C0)    schedule(true,H,T,H,T) :- !.                              % breadth-first
        schedule(suspended(A),H,[A|T],H,T) :- !.
        schedule((A,B),H,T,H2,T2) :- !,
                schedule(A,H,T,H1,T1),
                schedule(B,H1,T1,H2,T2).
        schedule(A,H,[A|T],H,T).
```

Solve(H,T,F) has a goal queue as a difference list, *H* and *T*, and a deadlock detection flag, *F*. The queue has a cycle marker *'$END'* to inform *solve/S0* that every goals in the queue has been tried. *Solve/S0* takes a goal out of the head of the goal queue and attempts to reduce it. *Reduce/R0* tries to reduce the goal. When the reduction succeeds, i.e., when a clause unifiable‡ with the goal is found and its guard is successfully solved, *reduce/R0* returns its body goals to the second argument. When the reduction fails, it returns *suspended(A)*, where *A* is the suspended goal. The third and fourth arguments of *reduce/R0* are the deadlock flags. When the given goal has been reduced, *nd (no deadlock)* is forced to be set to the fourth argument. When it has been suspended, the deadlock flag at the third argument is passed through to the fourth argument. Initially, *d (deadlock)* is set to the deadlock flag. Deadlock can be detected by confirming that the deadlock flag has never fallen to *nd* while the goal queue goes round.

Schedule/C0 schedules the reduction result in the goal queue breadth-first; the goals are appended at the tail of the queue. A depth-first scheduling strategy can also be used; the goals are inserted in front of the head of the queue. In that case, *schedule/C1* shown below is used instead of *schedule/C0*. When the given goal is marked as *suspended*, however, it is a suspended goal, and is always scheduled breadth-first.

```
(C1)    schedule(true,H,T,H,T) :- !.                              % depth-first
        schedule(suspended(A),H,[A|T],H,T) :- !.
        schedule((A,B),H,T,H2,T2) :- !,
                schedule(B,H,T,H1,T1),
                schedule(A,H1,T1,H2,T2).
        schedule(A,H,T,[A|H],T).
```

3. Compilation Techniques

As already stated in Section 1, Ueda and Chikayama developed a compiling technique from CP to Prolog. Their method is twofold. The first idea eliminates the overhead of goal invocations. The goal queue and deadlock detection flag in *solve* are included in each goal as extra arguments, thereby eliminating the processing time via *solve*.

The second idea implements depth-first scheduling which can manage the depth of a reduction tree; when depth-first reduction reaches the current depth limit, the search path is suspended and the current depth limit is increased in preparation for the coming deeper search, then another unreached path is tried. To implement this idea, a decrement counter is included in each goal as an extra argument. The counter remembers how many times reductions can take place before reaching the current depth limit. They called

‡ *Unify* in *guarded_clause/G0* is a special unification procedure; it has been tailored for CP to handle the read-only annotation, ?.

this scheduling strategy *N-bounded depth-first* scheduling, where N is the step size of the depth limit. The counter is initialized to N every time it reaches zero.

Because these two ideas are independent of each other, we can produce four variations as shown below.

Scheduling strategy	Interpreting	Compilation
Breadth/Depth-first	(i) Shapiro's interpreter	(ii) Breadth/Depth-first compilation
N-bounded depth-first	(iii) Enhanced interpreter	(iv) N-bounded depth-first compilation

Variation (i) is Shapiro's CP interpreter. Variation (ii) adopts the first idea; CP programs are compiled incorporating either breadth-first or depth-first scheduling. Variation (iii) adopts the second idea; it is an enhanced CP interpreter incorporating N-bounded depth-first scheduling. Variation (iv) adopts both ideas; CP programs are compiled incorporating N-bounded depth-first scheduling.

We will derive (ii) from (i) in Section 4. and (iv) from (iii) in Section 5. Enhanced interpreter (iii) incorporating N-bounded depth-first scheduling will be shown in Subsection 5.1. Sample compilations for (ii) and (iv) are shown in Subsections 4.4 and 5.2, respectively.

4. Compilation Incorporating Breadth-first or Depth-first Scheduling

This and the following sections will show that programs in CP can be converted to a Prolog program. The CP interpreter is incorporated into CP programs gradually in this derivation process. The CP interpreter in this section uses a simple scheduling strategy, which is either breadth-first or depth-first. The compiled code derived here is a simplified version of Ueda and Chikayama's compiled code. The following section will show a full derivation using N-bounded depth-first scheduling.

The derivation process consists of four major stages, some of which may consist of several minor steps. The first stage converts a CP program to its Prolog version using the definition of *reduce/R0*. The program is further modified in the second stage using the definition of *solve/S0*. In the third stage, indirect goal invocations are replaced by direct invocations. In the final stage, auxiliary predicates are unfolded to obtain the target code.

As stated before, we use an append program as an example for the derivation. *Append/A0* below is a CP program.

```
(A0)    append([X|Xs],Ys,[X|Zs]) :-                          % CP program
                append(Xs?,Ys?,Zs).
        append([],Ys,Ys).
```

4.1 First Stage

In the first stage, *append/A0* in CP is converted to its Prolog version, *append/A1*.

(1) Partial evaluation of reduce/R0

Reduce/R0 is partially evaluated, with respect to *append/A0*. Since *reduce/R0* is written in Prolog, the result is also a Prolog program.

First, consider the second clause of *reduce/R0*, as it is simpler than the first clause. It deals with reduction failure. It reports the failure by returning the suspended goal wrapped by *suspended*, and transmits the deadlock flag. The following clause has the same effect.

```
        reduce(append(L,M,N),suspended(append(L,M,N)),F0,F0).
```

Now turn to the first clause of *reduce/R0*. It searches a clause whose head is unifiable with the given goal by *guarded_clause/G0* and tries to solve the clause's guard by *solve/S0*. Because *append/A0* consists of two clauses, both of whose guards are *true*, *guarded_clause/G0* eventually returns the two clauses. The following clauses have the same effect.

```
reduce(append(L,M,N),append(Xs?,Ys?,Zs),F0,nd) :-
          unify(append(L,M,N),append([X|Xs],Ys,[X|Zs])),
          schedule(true,X,X,H,['$END'|T]), solve(H,T,d), !.
reduce(append(L,M,N),true,F0,nd) :-
          unify(append(L,M,N), append([],Ys,Ys)),
          schedule(true,X,X,H,['$END'|T]), solve(H,T,d), !.
```

The evaluation process can go further. First, schedule(true,X,X,H,['$END'|T]),solve(H,T,d) always turns out *true*, and can be deleted. Second, *unify* can proceed whenever arguments are given even partially. Two auxiliary predicates, *ulist* and *unil* are introduced to help *unify* proceed. They take charge of the subcomputation of *unify*, and are the same as those used in [4]. *Ulist(A,H,T)* and *unil(L)* have the same meaning as *unify(A,[H|T])* and *unify(L,[ ])*, but they are optimized enough.

Thus, the final result of this stage is *append/A1* below.

```
(A1)    reduce(append(A1,Ys,A3),append(Xs?,Ys?,Zs),_,nd) :-
                  ulist(A1,X,Xs), ulist(A3,X,Zs), !.
        reduce(append(A1,A2,A3),true,_,nd) :-
                  unil(A1), unify(A2,A3), !.
        reduce(append(X,Y,Z),suspended(append(X,Y,Z)),F,F).
```

4.2 Second Stage

In this stage, *append/A1* and *solve/S0* are converted step by step to *append/A4* and *solve/S4*. It is the key point of this stage that, in *solve/S0*, *schedule* and *solve* are always invoked after *reduce* returns. We will successively move them from *solve/S0* to *append/A1*.

(1) Migration of schedule to append

Solve/S0 includes two successive invocations as follows:

```
        reduce(A,B,F,NF), schedule(B,H,T,NH,NT)
```

They are integrated into the single invocation below by eliminating shared local variable *B*. This results in *solve/s2* and *append/A2*.

```
        reduce_schedule(A,H,T,NH,NT,F,NF)
```

At the same time, *append/A1* is converted to *append/A2*; *schedule* is moved from *solve/S0* to *append/A1*.

```
(A2)    reduce_schedule(append(A1,Ys,A3),H,T,NH,NT,_,nd) :-
                  ulist(A1,X,Xs), ulist(A3,X,Zs), !,
                  schedule(append(Xs?,Ys?,Zs),H,T,NH,NT).
        reduce_schedule(append(A1,A2,A3),H,T,NH,NT,_,nd) :-
                  unil(A1), unify(A2,A3), !,
                  schedule(true,H,T,NH,NT).
        reduce_schedule(append(X,Y,Z),H,T,NH,NT,F,F) :-
                  schedule(suspended(append(X,Y,Z)),H,T,NH,NT).
(S2)    solve(['$END'],_,_) :- !.
        solve(['$END'|H],[],d) :- !, fail.
        solve(['$END'|H],T,nd) :- !,
                  schedule(suspended('$END'),H,T,NH,NT),
                  solve(NH,NT,d).
        solve([A|H],T,_) :-
                  system(A), !, A,
                solve(H,T,nd).
        solve([A|H],T,F) :-
                reduce_schedule(A,H,T,NH,NT,F,NF),
                !, solve(NH,NT,NF).
```

(2) Migration of solve to append

Solve/S2 involves two successive invocations as follows:

```
        reduce_schedule(A,H,T,NH,NT,F,NF), !, solve(NH,NT,NF)
```

They are integrated into the following invocation by eliminating shared local variables *NH*, *NT*, and *NF*. This results in *solve/s3* and *append/A3*.

```
        $(A,H,T,F)
```

! above is negligible, because *reduce_schedule* fails immediately when it backtracks and the clause including the ! is the last clause of *solve/S2*. At the same time, *append/A2* is converted to *append/A3*; *solve* is moved from *solve/S2* to *append/A2*.

```
 (A3)   $(append(A1,Ys,A3),H,T,_) :-
                ulist(A1,X,Xs), ulist(A3,X,Zs), !,
                schedule(append(Xs?,Ys?,Zs),H,T,NH,NT),
                solve(NH,NT,nd).
        $(append(A1,A2,A3),H,T,_) :-
                unil(A1), unify(A2,A3), !,
                schedule(true,H,T,NH,NT),
                solve(NH,NT,nd).
        $(append(X,Y,Z),H,T,F) :-
                schedule(suspended(append(X,Y,Z)),H,T,NH,NT),
                solve(NH,NT,F).
 (S3)   solve(['$END'],_,_) :- !.
        solve(['$END'|H ],[],d) :- !, fail.
        solve(['$END'|H ],T,nd) :- !,
                schedule(suspended('$END'),H,T,NH,NT),
                solve(NH,NT,d).
        solve([A|H],T,_) :-
                system(A), !, A,
                solve(H,T,nd).
        solve([A|H],T,F) :-          % !,
                $(A,H,T,F).
```

(3) Deletion of solve

Next, *solve* is deleted. *Solve/S3* invokes *$*, which, in turn, invokes *solve/S3*. Thus, *solve/S3* is merely used as an intermediary and can be deleted completely. *Solve/S3* really does two things: taking a goal out of the goal queue, and classifying the goal and reducing it according to its type. We will move the first action into *append/A3*. In preparation, *solve/S3* is divided into two.

```
 (S3')  solve([A|H],T,F) :-
                '$2'(A,H,T,F).
        '$2'('$END',[],_,_) :- !.
        '$2'('$END',H,[],d) :- !, fail.
        '$2'('$END',H,T,nd) :- !,
                schedule(suspended('$END'),H,T,NH,NT),
                solve(NH,NT,d).
        '$2'(A,H,T,_) :-
                system(A), !, A,
                solve(H,T,nd).
        '$2'(A,H,T,F) :-
                $(A,H,T,F).
```

All the *solve* invocations are unfolded, consulting *solve/S3*. For example, solve(NH,NT,d) is unfolded into NH=[G|NH2],'$2'(G,NH2,NT,d). All the '$2' invocations can be also replaced by $ invocations. This replacement is safe on the assumption that no user-defined predicates are the same as either '$END' or system-defined ones. As a side effect, system-defined predicates must be explicitly handled as shown below. Only *unify* is shown as a representative. The other system-predicates must be programmed in the same way.

```
  (A4)  $(append(A1,Ys,A3),H,T,_) :-
                ulist(A1,X,Xs), ulist(A3,X,Zs), !,
                schedule(append(Xs?,Ys?,Zs),H,T,NH,NT),
                NH=[G|NH2], $(G,NH2,NT,nd).
        $(append(A1,A2,A3),H,T,_) :-
                unil(A1), unify(A2,A3), !,
                schedule(true,H,T,NH,NT),
                NH=[G|NH2], $(G,NH2,NT,nd).
        $(append(X,Y,Z),H,T,F) :-
                schedule(suspended(append(X,Y,Z)),H,T,NH,NT),
                NH=[G|NH2], $(G,NH2,NT,F).
  (S4)  $('$END',[],_,_) :- !.
        $('$END',H,[],d) :- !, fail.
        $('$END',H,T,nd) :- !,
                schedule(suspended('$END'),H,T,NH,NT),
                NH=[G|NH2], $(G,NH2,NT,d).
        $(unify(X,Y),H,T,_) :- !,
                unify(X,Y),
                H=[G|H2], $(G,H2,T,nd).
```

4.3 Third Stage

Now both *append/A4* and *solve/S4* consist solely of $ clauses. The $ clause actually invoked is determined only by its first argument. The symbol $ is, therefore, redundant and can be eliminated by promoting the first argument to a predicate. This is performed in two steps. The first step is transient and deletes $ from the head of a clause. The second step deletes $ from the body.

(1) Deletion of $ from the head

$ is deleted from the head. First, the first argument of each head is promoted to a new predicate; the argument's principal functor becomes a new predicate name. Second, the new predicate is extended and takes the remaining arguments *H*, *T*, and *F* as shown below, where *X*, *Y*, and *Z* are the original arguments of *append*.

```
        append(X,Y,Z,H,T,F)
```

The result of this step is shown below. All the $s have been successfully deleted from all the heads. Note how the arguments of *solve/S0* are added to the original arguments of *append/A0*. $ in the body, however, still remains unchanged. To compensate for this discrepancy, *$* clause marked as (#) in *solve/S5* below is temporarily reintroduced. This *$* will be deleted in the next step.

```
(A5)    append(A1,Ys,A3,H,T,_) :-
                ulist(A1,X,Xs), ulist(A3,X,Zs), !,
                schedule(append(Xs?,Ys?,Zs,H0,T0,F0),H,T,NH,NT),
                NH=[G|NH2], $(G,NH2,NT,nd).
        append(A1,A2,A3,H,T,_) :-
                unil(A1), unify(A2,A3), !,
                schedule(true,H,T,NH,NT),
                NH=[G|NH2], $(G,NH2,NT,nd).
        append(X,Y,Z,H,T,F) :-
                schedule(suspended(append(X,Y,Z,H0,T0,F0)),H,T,NH,NT),
                NH=[G|NH2], $(G,NH2,NT,F).
(S5)    '$END'([],_,_) :- !.
        '$END'(H,[],d) :- !, fail.
        '$END'(H,T,nd) :- !,
                schedule(suspended('$END'(H0,T0,F0)),H,T,NH,NT),
                NH=[G|NH2], $(G,NH2,NT,d).
        unify(X,Y,H,T,_) :- !,
                unify(X,Y),
                H=[G|H2], $(G,H2,T,nd).
(#)     $(G,H,T,F) :-
                functor(G,_,A),
                arg(A,G,F),
                A1 is A-1, arg(A1,G,T),
                A2 is A-2, arg(A2,G,H),
                call(G).
```

(2) Deletion of $ from the body

The auxiliary *$* introduced in the previous step is deleted. When *$* is invoked, it gives its extra arguments to a goal which is also given as an argument, and invokes the goal. Fortunately, a *$* invocation is always immediately preceded by a *take-out* queue operation. By changing the goal queue structure, taking a goal out of the queue and setting extra arguments in the goal are accomplished by a single operation using the following technique. This technique was originally developed by Ueda and Chikayama.

Let each element in the goal queue have the following form:

$\$(\mathrm{head}(A_1,..,A_m,B_1,..,B_n),B_1,..,B_n)$

Head($A_1,...,A_m,B_1,...,B_n$) is an extended goal, where $A_1,...,A_m$ are the original arguments of the goal, and $B_1,...,B_n$ are the extra arguments. This form has a function similar to the λ-expression. $B_1,...,B_n$ serve as formal arguments; each argument B_i in *head* can be initialized by unifying a value and the B_i outside the *head*.

In the case of *append*, this λ-like expression has the form like this:

$(append(X,Y,Z,H0,T0,F0),H0,T0,F0)

where *H0*,*T0*, and *F0* are the extra arguments coming from *solve/S0*. The following program fragment does two things: taking a goal G from a goal queue H, and setting NT and NF to H0 and T0. At the same time, the rest of the goal queue, *H2*, is set to *H0*.

H=[$(G,H2,NT,NF)|H2]

Five unifications occur simultaneously: *G=append(X,Y,Z,H0,T0,F0)*, *H2=H0*, *NT=T0*, *NF=F0*, and *H=[_|H2]*. These unifications accomplish the intended task as a whole.

The result of this step is shown below. All the *$* invocations have been successfully deleted from all the bodies, thereby successfully deleting all the *$* invocations from all the clauses. Instead, as can be seen in *append/A6* and *solve/S6*, a substitute for *$* appears in the goal queue.

```
(A6)    append(A1,Ys,A3,H,T,_) :-
                ulist(A1,X,Xs), ulist(A3,X,Zs), !,
                schedule($(append(Xs?,Ys?,Zs,H0,T0,F0),H0,T0,F0),H,T,NH,NT),
                NH=[$(G,NH2,NT,nd)|NH2],
                call(G).
        append(A1,A2,A3,H,T,_) :-
                unil(A1), unify(A2,A3), !,
                schedule(true,H,T,NH,NT),
                NH=[$(G,NH2,NT,nd)|NH2],
                call(G).
        append(X,Y,Z,H,T,F) :-
                schedule(suspended($(append(X,Y,Z,H0,T0,F0),H0,T0,F0)),H,T,NH,NT),
                NH=[$(G,NH2,NT,F)|NH2],
                call(G).
(S6)    '$END'([],_,_) :- !.
        '$END'(H,[],d) :- !, fail.
        '$END'(H,T,nd) :- !,
                schedule(suspended($('$END'(H0,T0,F0),H0,T0,F0)),H,T,NH,NT),
                NH=[$(G,NH2,NT,d)|NH2],
                call(G).
        unify(X,Y,H,T,_) :- !,
                unify(X,Y),
                H=[$(G,H2,T,F)|H2],
                call(G).
```

4.4 Final Stage

Once the scheduling strategy is fixed, *schedule* can be unfolded. *Append/A7* and *solve/S7* shown below are obtained by unfolding *schedule*, consulting *schedule/C0*. The result is compiled code incorporating breadth-first scheduling, which corresponds to variation (ii) in Section 3.

```
(A7)    append(A1,Ys,A3,H,T,_) :-
                ulist(A1,X,Xs), ulist(A3,X,Zs), !,
                T=[$(append(Xs?,Ys?,Zs,H0,T0,F0),H0,T0,F0)|NT],
                H=[$(G,H2,NT,nd)|H2],
                call(G).
        append(A1,A2,A3,H,T,_) :-
                unil(A1), unify(A2,A3), !,
                H=[$(G,H2,T,nd) |H2],
                call(G).
        append(X,Y,Z,H,T,F) :-
                T=[$(append(X,Y,Z,H0,T0,F0),H0,T0,F0)|NT],
                H=[$(G,H2,NT,F)|H2],
                call(G).
(S7)    '$END'([],_,_) :- !.
        '$END'(H,[],d) :- !, fail.
        '$END'(H,T,nd) :- !,
                T=[$('$END'(H0,T0,F0),H0,T0,F0)|NT],
                H=[$(G,H2,NT,d)|H2],
                call(G).
        unify(X,Y,H,T,_) :- !,
                unify(X,Y),
                H=[$(G,H2,T,F)|H2],
                call(G).
```

Now let us turn to another scheduling strategy. *Append/A8* is obtained by unfolding *schedule/C1* which performs depth-first scheduling. *Append/A8* equals *append/A7* except for the first clause. Thus, only the first clause of *append/A8* is shown below. *Solve/S8* is the same as *solve/S7*, and it is omitted here. *Append/A8* can be converted further to *append/A8'*, because it is useless to put a goal at the head of the goal queue and immediately take it out. The result is a compiled code incorporating depth-first scheduling, which also corresponds to variation (ii) in Section 3.

```
(A8)    append(A1,Ys,A3,H,T,_) :-
                ulist(A1,X,Xs), ulist(A3,X,Zs), !,
                NH=[$(append(Xs?,Ys?,Zs,H0,T0,F0),H0,T0,F0)|H],
                NH=[$(G,NH2,T,nd)|NH2],
                call(G).
(A8')   append(A1,Ys,A3,H,T,_) :-
                ulist(A1,X,Xs), ulist(A3,X,Zs), !,
                append(Xs?,Ys?,Zs,H,T,nd).
```

5. Compilation Incorporating N-bounded Depth-first Scheduling

This section shows an enhanced CP interpreter incorporating N-bounded depth-first scheduling. To incorporate N-bounded depth-first scheduling into Shapiro's CP interpreter, the execution information of each goal must be explicitly specified, because such an enhanced interpreter must access the information at will to govern the whole reduction process. We will call such an information block a *PCB* (Process Control Block). The interpreter schedules each goal, referring to the information in the PCB attached to the goal.

A PCB and a goal are attached to each other by @ in the following way.

```
        goal@Pcb
```

An enhanced CP interpreter handling the PCB is outlined below.

```
(R1)    reduce(Head@PCB,Body,_,nd) :-
                firmware(policy,PCB,GuardPCBs,BodyPCBs),
                guarded_clause(Head,Guard0,Body0),
                attach(Guard0,GuardPCBs,Guard),
                schedule(Guard,X,X,H,['$END'|T]),
                solve(H,T,d), !,
                attach(Body0,BodyPCBs,Body).
        reduce(Goal@PCB,suspended(Goal@SuspendedPCB),F,F) :-
                firmware(policy,PCB,SuspendedPCB).
```

Reduce/R1 handles the form *goal@Pcb*. *Solve* in *reduce/R1* is the same as *solve/S0* in the original CP interpreter. *Firmware* calculates two PCB lists of child goals from the PCB of the parent goal, referring to the specified *policy*; *policy* decides how to calculate the PCB lists. *Attach* is an auxiliary predicate only for matching; it combines a goal list with a PCB list sequentially, and produces a *goal@Pcb* list.

5.1 Interpreter Incorporating N-bounded Depth-first Scheduling

Here, PCB is the form *(B,BC)*, where *B* is a decrement counter which limits the depth of a reduction tree and *BC* is the initial value for the counter. *Reduce/R2* incorporating N-bounded depth-first scheduling is shown below. *Reduce/R2* combined with *solve/S0* corresponds to variation (iii) in Section 3.

Here, the *policy* is defined as "If B has not yet reached 0, reduction can continue and *B* is decremented by 1. If not, reduction is suspended, and *BC* is set to B."

```
(R2)    reduce(Head@(B,BC),Body,_,nd) :-
                  B>0, NB is B-1,
                  guarded_clause(Head,Guard0,Body0),
                  attach(Guard0,(BC,BC),Guard),
                  schedule(Guard,X,X,H,['$END'|T]),
                  solve(H,T,d), !,
                  attach(Body0,(NB,BC),Body).
        reduce(Goal@(_,BC),suspended(Goal@(BC,BC)),F,F).
```

Since all the child goals that generated from a parent goal have the same PCB, *attach* here is simpler than *attach* in *reduce/R1*.

N.B.: *Schedule* in *solve* is *schedule/C1*, because breadth-first scheduling is meaningless here.

5.2 Compilation Incorporating N-bounded Depth-first Scheduling

A compiled code incorporating N-bounded depth-first scheduling is obtained by performing the program conversion described in Section 4, using *reduce/R2* instead of *reduce/R0*. The first stage is the only exception; it needs special consideration. *Reduce/R2* is partially evaluated, with respect to *append/A0*. This results in *append/A9* below.

```
(A9)    reduce(append(A1,Ys,A3)@(B,BC),append(Xs?,Ys?,Zs)@(NB,BC),_,nd) :-
                  B>0, ulist(A1,X,Xs), ulist(A3,X,Zs), !, NB is B-1.
        reduce(append(A1,A2,A3)@(B,BC),true,_,nd) :-
                  B>0, unil(A1), unify(A2,A3), !.
        reduce(append(X,Y,Z)@(_,BC),suspended(append(X,Y,Z)@(BC,BC)),F,F).
```

Here, the PCB can be included as extra arguments of each goal without any trouble, because it always goes with the goal. This results in *append/A9'* below.

```
(A9')   reduce(append(A1,Ys,A3,B,BC),append(Xs?,Ys?,Zs,NB,BC),_,nd) :-
                  B>0, ulist(A1,X,Xs), ulist(A3,X,Zs), !, NB is B-1.
        reduce(append(A1,A2,A3,B,BC),true,_,nd) :-
                  B>0, unil(A1), unify(A2,A3), !.
        reduce(append(X,Y,Z,_,BC),suspended(append(X,Y,Z,BC,BC)),F,F).
```

In the remaining stages, the clauses are converted to *append/A10* in the procedure described in Section 4. *Append/A10* corresponds to variation (iv) in Section 3.

```
(A10)   append(A1,Ys,A3,B,BC,H,T,_) :-
                  B>0, ulist(A1,X,Xs), ulist(A3,X,Zs), !, NB is B-1,
                  append(Xs?,Ys?,Zs,NB,BC,H,T,nd).
        append(A1,A2,A3,B,BC,H,T,_) :-
                  B>0, unil(A1), unify(A2,A3), !,
              H=[$(G,H2,T,nd)|H2],
              call(G).
    append(X,Y,Z,_,BC,H,T,F) :-
              T=[$(append(X,Y,Z,BC,BC,H0,T0,F0),H0,T0,F0)|NT],
              H=[$(G,H2,NT,F)|H2],
              call(G).
```

6. Compiling a CP Meta Interpreter

The *append* we have used as an example so far is really a simple program. All the guards that appeared in *append* are *true*, i.e., *append* is merely a FCP program. FCP is a subset of CP where all guards are either *true* or system-defined predicates. The CP programs used in [4] are really FCP programs. Now we will develop a compilation technique for full CP. We use *mcall/M0* as the next example.

Mcall/M1 is a working version of *mcall/M0* in Section 2. *Mcall/M1* invokes several auxiliary predicates, system-defined *system*, *exec*, *clauses* and user-defined *resolve*. *System* detects a system-defined goal and *exec* executes it. *Clauses* collects candidate clauses that are unifiable with the given goal. *Resolve* selects one reducible clause from the candidate clauses.

```
(M1)    mcall(true).
        mcall((A,B)) :- mcall(A?), mcall(B?).
        mcall(A) :- system(A) | exec(A).
        mcall(A) :- clauses(A,Clauses) |
                  resolve(A,Clauses,Body), mcall(Body?).
(#)     resolve(A,[(Head:-Guard|Body)|Cs],Body) :-
                  unify(A,Head), mcall(Guard) | true.
        resolve(A,[(Head:- Body)|Cs],Body) :-
                  unify(A,Head), Body≠(_|_) | true.
        resolve(A,[C|Clauses],Body) :-
                  resolve(A,Clauses,Body) | true.
```

Mcall/M1 is really a full CP program, because the first clause includes **mcall(Guard)** and the third clause includes **resolve(A,Clauses,Body)** as a guard, and both are user-defined.

Compilation proceeds as before. We need no more tricks. In the first stage, *reduce/R0* is partially evaluated, with respect to *mcall/M1*. To make the following explanation brief, we will concentrate on the (#) clause in *mcall/M1*. The following clause is the intermediate result of unfolding *reduce(resolve(L,M,N),B,F0,F1)*, consulting the (#) clause above.

```
(#1)    reduce(resolve(L,M,N),true,F0,nd) :-
                  unify(resolve(L,M,N),resolve(A,[(Head:-Guard|Body)|Cs],Body)),
                  schedule((unify(A,Head),mcall(Guard)),X,X,H,['$END'|T]),
                  solve(H,T,d), !.
```

The first *unify* can proceed to a further computation. The second *unify* can be moved out of *schedule*, because *unify* is system-defined and is executed immediately by *solve*. The (#2) clause below is the second intermediate result.

```
(#2)    reduce(resolve(A,Arg2,Body),true,_,nd) :-
                  ulist(Arg2,(Head:-Guard|Body),Cs),
                  unify(A,Head),
                  schedule(mcall(Guard),X,X,H,['$END'|T]),
                  solve(H,T,d), !.
```

The rest of the task proceeds in the same way as *append*. The (#3) clause below is the final code after the derivation, incorporating depth-first scheduling. This result corresponds to variation (ii) in Section 3.

```
(#3)    resolve(A,Arg2,Body,H,T,_) :-
                ulist(Arg2,(Head:-Guard|Body),Cs),
                unify(A,Head),
                NH=[$('$END'(H0,T0,F0),H0,T0,F0)|NT],
                mcall(Guard,NH,NT,d), !,
                H=[$(G,H2,T,nd)|H2],
                call(G).
```

7. Performance Improvement in Practice

Sections 4 and 5 showed how to compile a CP program in Prolog. Since each program obtained this way is a complete Prolog program, it can be executed on any Prolog processor. The effect of each step on performance can be measured by checking the execution speed. Performance was measured by the total execution time of three sample problems.

The sample problems used in the measurement are as follows:

(a) *Reverse* a list three times through a pipe

nreverse(3,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],S).

i.e., **reverse([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],X), reverse(X?,Y), reverse(Y?,S).**

(b) Sort a random list using the *quick sort* algorithm

qsort([17,26,13,21,5,1,20,9,3,27,15,25,11,30,24,8,2,
28,29,4,23,19,16,22,31,6,10,14,32,12,7,18],S).

(c) *Quick-sort* a short random list on *meta interpreter*

mcall(qsort([4,2,3,5,1],S)).

We used **Quintus Prolog**^{TM} Release 1.5 that has a Prolog compiler. The experimental results are summarized in Table 1 on the next page.

The table organizes the results along two axes:

- Eight derivation steps among four stages
  Three problems in every step:
  nreverse, *qsort*, and *mcall*
- Two selections either using or not using the Quintus Compiler
  Three scheduling strategies in each selection:
  depth-first, breadth-first, and 10-bounded depth-first scheduling¶

The rows at the *first stage* contain real processing time in seconds. The other rows contain the ratio of the processing time in the stage to the corresponding time in the *first stage*. For example, when using the Quintus Prolog Interpreter and depth-first scheduling, *nreverse* took 14.83 sec in the *first stage*. *Nreverse* at the *initial stage* took 7.15 times as many seconds as in the *first stage*; it took 106.0 sec = 14.83 sec × 7.15.

Compiling considerably affected the performance improvement in the derivation process. The improvement in the Prolog interpreter is apparent from Table 1. In the case of Prolog compiler, on the other hand, the improvement is dependent upon the selections of the problems and the scheduling strategies.

In both cases, the performance improved considerably in the first stage of the derivation, because the execution cost of *reduce/R0* or *reduce/R2* was substantially decreased. No improvement was found in the second stage, because some goal invocations in *solve* were simply moved to each *append*. The performance improved somewhat more in the third stage than in the second stage, because indirect invocations to the

¶ N is 10; the step size of depth limit is 10.

Table 1 Performance Improvement in the Derivation Process

Derivation	Problem	Quintus Prolog Interpreter			Quintus Prolog Compiler		
		Depth	Breadth	10-Depth	Depth	Breadth	10-Depth
Initial Stage	*nreverse*	7.15	5.01	6.05	20.4	14.4	8.89
original CP	*qsort*	7.67	4.59	5.44	22.2	15.3	8.74
program	*mcall*	5.32	5.38	5.05	4.79	4.84	4.35
First Stage	*nreverse*	14.83 sec	25.97 sec	18.10 sec	0.77 sec	1.35 sec	1.70 sec
partial evaluation	*qsort*	10.42 sec	41.67 sec	16.22 sec	0.50 sec	1.78 sec	1.52 sec
of reduce	*mcall*	3.90 sec	14.05 sec	15.12 sec	2.82 sec	2.83 sec	3.20 sec
Second Stage	*nreverse*	1.03	1.04	1.02	1.00	0.98	0.99
migration	*qsort*	1.01	1.01	1.04	1.03	0.87	0.99
of schedule	*mcall*	1.01	1.01	1.02	0.99	0.99	1.02
Second Stage	*nreverse*	0.84	0.84	1.06	0.89	0.93	0.86
migration	*qsort*	0.91	0.87	1.08	0.93	0.76	0.87
of solve	*mcall*	0.96	0.97	1.04	0.98	0.99	0.99
Second Stage	*nreverse*	0.88	0.88	1.09	0.78	0.77	0.80
deletion	*qsort*	0.96	0.94	1.12	0.80	0.68	0.83
of solve	*mcall*	0.99	1.00	1.06	0.98	0.99	0.99
Third Stage	*nreverse*	0.94	1.00	1.07	2.37	2.64	1.24
deletion	*qsort*	0.83	0.81	0.87	1.97	2.13	0.98
of $ from head	*mcall*	0.94	0.97	0.95	1.11	1.12	1.01
Third Stage	*nreverse*	0.60	0.61	0.78	2.24	2.43	1.12
deletion	*qsort*	0.63	0.52	0.69	1.87	1.95	0.91
of $ from body	*mcall*	0.83	0.84	0.85	1.08	1.10	0.99
Final Stage	*nreverse*	0.36	0.53	0.56	0.74	2.23	0.45
unfolding	*qsort*	0.46	0.48	0.56	0.93	1.78	0.60
of schedule	*mcall*	0.77	0.80	0.80	0.99	1.06	0.91

next goal were replaced by direct invocations, but this is only true in the case of the Prolog interpreter. In the case of the Prolog compiler the performance became worse. The performance improved further in the final stage, where the time-consuming scheduling calculation was eliminated.

8. Future Research

Each derivation stage is characterized as follows. The first stage utilizes partial evaluation. The second stage moves common goal invocations into each clause. The third stage promotes the first argument of each clause to a predicate. The fourth and final stage unfolds auxiliary goals, consulting their definitions. Of these four stages, the third stage is somewhat magical; an argument is promoted to a predicate and the goal queue structure is changed. We believe that such magical techniques are necessary to enrich the world of program transformation. Further investigation is necessary.

Section 7 showed that the meta interpreter can be compiled in Prolog. An interesting subject is the automation of a process in which a compiled code is generated from a given meta interpreter and a program, in other words, developing a specialized partial evaluator which can be used as a compiler. All we have to do is to make a meta interpreter which realizes the desired functionality. The specialized partial evaluator will do the rest of the work. It can compile programs written in the new functional style, consulting the meta interpreter. Hirsch has moved closer to this goal in the domain of CP.

9. Conclusion

Ueda and Chikayama developed a compiling technique from CP to Prolog. The technique might look complicated at a first glance. In this paper, we successfully separated several ideas; we derived their compiled code from the CP interpreter and a CP program. Specifically, we used an append program and a meta interpreter in CP as our examples, transforming them step by step and incorporating the CP interpreter gradually.

Acknowledgements

This research was conducted as part of the Fifth Generation Computer Project. We wish to thank the members of the First Research Laboratory at ICOT and Dr. Kitagawa and Dr. Enomoto at IIAS-SIS, FUJITSU LIMITED for giving us the opportunity to complete this study.

References

[1] Hirsch, M., Silverman, W. and Shapiro, E., Layers of Protection and Control in the Logix System, *CS86-19*, Weizmann Instit., 1986

[2] Kursawe, P., How to Invent a Prolog Machine, in *Proc. of 3rd Inter. Conf. on Logic Prog.*, 1986, pp. 134-148

[3] Shapiro, E., A Subset of Concurrent Prolog and Its Interpreter, *Tech. rep. TR-003*, ICOT, 1983

[4] Ueda, K., and Chikayama, T., Concurrent Prolog Compiler on Top of Prolog, in *Proc. of Symp. on Logic Prog.*, 1985, pp. 119-126