ICOT Technical Report: TR-397

TR-397

Meta-interpreters and Reflective Operations in GHC

by
J. Tanaka (Fujitsu)

June, 1988

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

Institute for New Generation Computer Technology

Meta-interpreters and Reflective Operations in GHC

Jiro Tanaka

International Institute,
Fujitsu Limited,
1-17-25 Shinkamata, Ota-ku, Tokyo 144

Abstract

Starting from the simple self-description of GHC [Ueda 85a], we have derived various kinds of meta-interpreters by stepwise enhancement. A meta-interpreter which has variable management facility has been described. As far as I know, this is the first attempt to try to manage variable bindings in logic programming languages. Preliminary implementation has been performed with these meta-interpreters and the execution time has been measured for simple benchmark programs. It is shown that the overhead of variable management is not too much, comparing merits we can get there.

Various reflective operations in GHC are also described. Implementaition of these operation is shown using an enhanced meta-interpreter. Examples of the usage of these operations are also shown.

All the programs and examples shown in this paper are running on our GHC system. This paper assume a basic knowledge of parallel logic languages such as PARLOG [Clark 85], Concurrent Prolog [Shapiro 83] or GHC.

1. Introduction

The original notion of self-description seems to come from the description of EVAL in LISP. It tries to describe its language features by itself. Since both of programs and data structures are expressed as S-expressions in Lisp, the self-description of Lisp was not very difficult.

On the other hand, in Prolog world, the following 4-line program has been known as "Prolog in Prolog" [Bowen 83].

```
exec(true):-!.
exec((P,Q)):-!,exec(P),exec(Q).
exec(P):-clause((P:-Body)),exec(Body).
exec(P):-P.
```

The meaning of this meta-interpreter is fairly simple. The goal which should be solved is given as an argument of exec. If it is "true," the execution of the goal succeeds. If it is a sequence, it is decomposed and executed sequentially. In the case of a user-defined goal, predicate "clause" finds the definition of the given

goal and it is decomposed to the definition. Otherwise, it is assumed to be system-defined goals and solved directly. If all of these trials fail, these mean that the execution of the given goal fails. Though this 4-line program is very simple, it certainly works as as "Prolog in Prolog".

In this paper, we discuss the self-description of a parallel logic language GHC [Ueda 85a]. The GHC version of this meta-interpreter can similarly be written as follows:

```
exec(true):-true | true.
exec((P,Q)):-true | exec(P),exec(Q).
exec(P):-not_sys(P) | reduce(P,Body),exec(Body).
exec(P):-sys(P) | P.
```

This GHC program is almost the same as the Prolog program. However "!" is replaced by " | " since every clause definition includes the " | " operator in GHC. The predicate "clause" is also replaced by "reduce," which includes the operation of solving guard.

However, compared with Lisp, these 4-line program seems to be too simple since it only simulates the top-level control flow of the given program. Therefore, we would like to enhance this 4-line program and would like to obtain more useful information from the program.

2. Meta-interpreter enhancement

How do we extend this meta-interpreter is our next problem. We gained a hint from Kurusawe's paper [Kurusawe 86]. He assumed a abstract Prolog machine which can execute Prolog programs. He derived Warren-like code from the given Prolog program by changing the border between the machine and the program. Staring from the ordinary Prolog program, he makes explicit various hidden operations, such as unification or memory structure, step by step to get Warren-like code.

Our approach is similar to his approach in some sense. However, we are not much interested in transforming the source program. We are interested in the description of "abstract machine," which we try to express in the form of meta-interpreter.

In Prolog and parallel logic languages, various extension has already been proposed. They are:

(1) "demo" predicate by Bowen and Kowalski [Bowen 82]. This predicate is used in the form of "demo(Prog, Goals)" and shows that Goals are provable from "Prog." This is identical to the "exec," shown above, except that program definition is explicit in "demo" predicate.

(2) Fail-safe exec. "exec(G,R)" executes the given goal "G" and returns "success" if succeeded, "failure" if failed. This prevents the program from the failure even if the goal "G" fails. This predicate has been proposed by Clark in parallel programming language [Clark 84].

(3) Controllable exec. It is used in the form of "exec(G,I,O)," where "I" is the input stream and "O" is the output stream. This "exec" is very useful if we would like to control the program execution from the

outside. We can "suspend," "resume," or "abort" the execution of the given goal. This predicate has also been proposed by Clark in parallel programming language [Clark 84].

(4) "exec(G,History)" which is the extension of the fail-safe exec. It returns the execution history instead of result. This exec is useful to build debuggers. Various works are done at ICOT in Prolog and parallel logic languages.

These proposals seems to answer the question how we extend our meta-interpreter. That is, we should extend our meta-interpreter to make explicit what we would like to know or what we would like to control.

3. Stepwise meta-interpreter enhancement

We sometimes need to understand the current state of the system. We also need to be able to modify and return the state to the system. These kinds of "reflective" capabilities, seen in 3-Lisp [Smith 84], seem to be very useful in writing an operating system. In 3-Lisp, we can easily obtain the current "continuation" and "environment" from the program. Smith used meta-circular interpreters as a mechanism to obtain information from the program.

We extend our meta-interpreter in a similar way to Smith's approach. The extension depends on what kind of resources we want to control. We try to enhance the 4-line GHC meta-interpreter and try to realize these "reflective" capabilities.

3.1 Two argument meta-interpreter

First extension is to get a "fail-safe" meta-interpreter by modifying the original 4-line GHC meta-interpreter. This modification is very simple and can be expressed as follows:

```
exec(true,R):-true | R=success.
exec(false,R):-true | R=failure.
exec((P,Q),R):-true | exec(P,R1),exec(Q,R2),and_result(R1,R2,R).
exec(P,R):-not_sys(P) | reduce(P,Body),exec(Body,R).
exec(P,R):-sys(P) | sys_exe(P,R).
```

In general, "failure" occurs when there are no unifiable clauses in "reduce." We assume "Body" is instanciated to "false" in such case. We should note that the "success" or "failure" of the given goal is handled as a message to the outside world.

3.2 Three argument meta-interpreter

We sometimes want to manage processes dynamically at execution time. Therefore, we introduce a "scheduling queue" explicitly in our meta-interpreter. Program "continuation" was explicit in the meta-circular interpreter in 3-Lisp. We have thought that the "scheduling queue" acts as a "continuation" in

GHC. The meta-interpreter which contains a scheduling queue inside the meta-interpreter becomes the following three argument "exec".

```
exec(T,T,R):-true | R=success.
exec([true | H],R):-true | exec(H,T,R).
exec([fail | H],T,R):-true | R=failure.
exec([P | H],T,Id,Mem,R):-not_sys(P) |
    reduce(P,T,NT),exec(H,NT,R).
exec([P | H],T,Id,Mem,R):-sys(P) |
    sys_exe(P,T,NT),exec(H,NT,R).
```

The first two arguments of "exec", "H" and "T," express the scheduling queue in Difference list form. The use of Difference list for expressing scheduling queue was originally invented by Shapiro [Shapiro 83]. We remove a goal from the top of the queue. Then "reduce" or "sys_exe" processes the goal. When the evaluation of goal suspends, it simply put the original goal to the tail of the scheduling queue again.

We should note that goals are processed "sequentially" because we have introduced a scheduling queue. However, it does not mean that the whole world is sequential. It simply means that the enqueuing and dequeuing processes are just sequential.

3.3 Five argument meta-interpreter

Then we introduce two more arguments, "MaxRC" and "RC," to control "reduction time." This kind of enhancement is motivated by [Foster 87]. We assume that this corresponds to the "computation time" in conventional systems. "MaxRC" shows the limit of the reduction count allowed in that "exec." "RC" shows the current reduction count.

```
exec(T,T,R,MaxRC,RC):-true | R=success(RC).
exec([true | H],T,R,MaxRC,RC):-true | exec(H,T,R,MaxRC,RC).
exec([fail | H],T,R,MaxRC,RC):-true | R=failure(RC).

exec([P | H],T,R,MaxRC,RC):-not_sys(P),MaxRC=<RC |
    reduce(P,T,NT,RC,RC1),exec(H,NT,R,MaxRC,RC1).
exec([P | H],T,I,R,MaxRC,RC):-sys(P),MaxRC=<RC |
    sys_exe(P,T,NT,RC,RC1),exec(H,NT,R,MaxRC,RC1).
exec([P | H],T,I,R,MaxRC,RC):-MaxRC>RC |
    R=count_over(R).
```

Notice that "reduce" or "sys_exe" increments "RC" by one when the actual computation takes place. However, "RC" is not incremented when suspended.

4. Variable manageable meta-interpreter

Varieties of meta-interpreters are described in the previous section. However, compared with Lisp, these meta-interpreters are incomplete since variables are not managed by themselves. Variables are shared. It means that the execution of a goal is influenced by the outside environment.

Consider the following example, which is adapted from [Ueda 86].

```
:- exe(X=0,R1),exe(X=1,R2),X=2.
```

Assume that "exec" is the two argument meta-interpreter, explained before. The shared variable X becomes 0, if the first "exec" is executed first. If the second "exec" is executed first, "X" becomes 1. Both cases, the whole system fails when X=2 is executed. This example shows that the "fail-safe" meta-interpreter may cause the unexpected result, in case it has shared variables.

In this section, we consider a meta-interpreter which explicitly handles variables. The toplevel description of variable manageable meta-interpreter is as follows:

```
m_ghc([FGoal | In],Out):-true |
    transfer(FGoal,NGoal,1,Id,Env),
    schedule(NGoal,H,T),
    exec(H,T,Id,Mem,Res),
    memory([enter(Env) | Mem],[ ]),
    print_result(Res,NGoal,Out1),
    merge(Out1,Out2,Out),
    m_ghc(In,Out2).
m_ghc([halt | In],Out):-true |
    Out=[halted].
```

The toplevel goal "m-ghc(In,Out)" has two arguments. "In" is the input from the user and "Out" denotes the output to the user. Every time it accepts the goal "FGoal" from the input, it generates five processes, i.e., "transfer," "schedule," "exec," "memory" and "print_result" processes. Among these, "transfer" and "schedule" processes are relatively short-live processes. The remaining three processes live long until the goal execution is completed. Figure 1 shows the snapshot how processes are generated accordance with the user input.

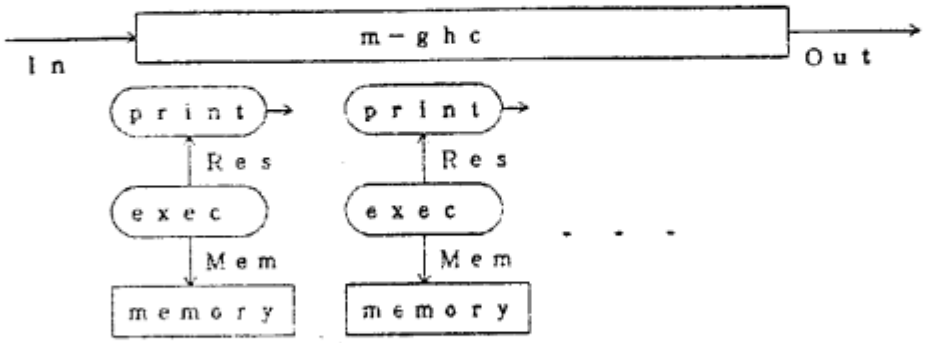

Figure 1 Creation of processes in m_ghc

The "transfer" generates "NGoal" from "FGoal." In "NGoal," every variable has been replaced by special identification numbers which have the format "@number." The third argument of "transfer" specifies the starting identification number of variables. The fourth argument "Id" denotes the identification number which should be assigned next. For exampie, when "transfer" allocated numbers from 1 to n, n+1 is assigned to "Id." The fifth argument contains the correspondence between the identification numbers and its value.

In "Env" or "memory," variables are contained in the forms of "(@number value)." For example, the meaning of this format is as follows:

(@1 undf) ... the value of variable @1 is undefined
(@2 100) ... the value of variable @2 is 100
(@3 ref(@2)) ... the value of variable @3 is the reference pointer to variable @2

For example, when the predicate "transfer" is given FGoal "exam([H|T],H)," it generates NGoal "exam([@1|@2],@1])." At that time, "Env" becomes [(@1 undf)(@2 undf)] and is entered to "memory."

The "exec" process can be described as follows:

```
exec(T,T,Id,Mem,Res):-true | Res=success,Mem=[ ].
exec([fail | H],T,Id,Mem,Res):-true | Res=fail,Mem=[ ].
exec([true | H],T,Id,Mem,Res):-true | exec(H,T,Id,Mem,Res).
exec([P | H],T,Id,Mem,Res):-not_sys(P) |
    reduce(P,T,NT,Id,Id1,Mem,Mem1),exec(H,NT,Id1,Mem1,Res).
```

```
exec([P | H],T,Id,Mem,Res):-sys(P) |
    sys_exe(P,T,NT,Mem,Mem1),exec(H,NT,Id,Mem1,Res).
```

This "exec" is almost the same as before, except it explicitly send messages to "memory" when the value of a variable is needed in "reduce" or "sys_exe." The third argument of "exec" contains the identification number which should be assigned next in "reduce." The fourth is the stream to "memory" and the fifth is the variable which contains the execution result.

The "reduce" predicate can be described as follows:

```
reduce(P,T,NT,Id,Id1,Mem,NewMem):- true |
    clauses(P,FClauses),
    resolve(P,FClauses,Body,Id,Id1,Mem,NewMem),
    schedule(Body,T,NT).

resolve(P,[FClause | Cs],Body,Id,Id1,Mem,Mem2):- true |
    transfer(FClause,Clause,Id,IdTemp,LoEnv),
    try_commit(P,Clause,Body,LoEnv,Res,Mem,Mem1),
    resolve1(Res,P,Cs,Body,Id,IdTemp,Id1,Mem1,Mem2).
resolve(P,[ ],Body,Id,Id1,Mem,NewMem):- true |
    Body=P,
    NewMem=Mem,
    Id1=Id.
resolve1(success,_,_,_,_,IdTemp,Id1,Mem1,Mem2):- true |
    Id1=IdTemp,
    Mem2=Mem1.
resolve1(susp,P,Cs,Body,Id,_,Id1,Mem1,Mem2):- true |
    resolve(P,Cs,Body,Id,Id1,Mem1,Mem2).
```

In "reduce," "clauses" constructs the list of potentially unifiable clauses "FClauses" from the given goal "P." "resolve" finds one "FClause" which can be committed. Then the goal "P" is expanded to the body of that "FClause." "schedule" puts this "Body" to the tail of the scheduling queue. We should note that "transfer" is called for each "FClause" to create a local environment in "resolve."

Next, we describe "try_commit" predicate. This predicate performs "head_unification" between the goal and the head of the candidate clause, solves the guard of the candidate clause, and tries to "commit" this clause if "head_unification" and "guards" are solved successfully.

```
try_commit(Goal,(Head:-G | B),Body,LoEnv,Res,Mem,NewMem):- true |
    head_unification(Goal,Head,LoEnv,LoEnv1,Res1,Mem,Mem1),
    solve_guard(G,LoEnv1,LoEnv2,Res2),
    and_result(Res1,Res2,Res3),
    commit(Res3,B,Body,LoEnv2,Res,Mem1,NewMem).

commit(success,B,Body,LoEnv,Res,Mem,NewMem):- true |
    Mem=[enter(LoEnv) | NewMem],
    Body=B,
    Res=success.
commit(susp,_,_,_,Res,Mem,NewMem):- true |
    NewMem=Mem,
    Res=susp.
```

We should note that "head_unification" does not generate any global bindings in "memory." It only references the global bindings. The local environment of a candidate clause becomes global only after it is committed.

The "head_unification" predicate has seven arguments. "Goal" and "Head" are put to the first and the second arguments. The local environment of the clause is put to the third argument. The fourth argument stores the new local environment after the "head_unification" is completed. The result of "head unification," whether it is succeeded or suspended, is put to the fifth argument. The sixth and the seventh is used for keeping the communication to "memory."

It can be described as follows:

```
head_unification(Goal,Head,LoEnv,NewLoEnv,Res,Mem,NewMem)
    :-true |
    Mem=[deref(Goal,GV) | Mem1],
    deref(Head,LV,LoEnv),
    h_unify(Gv,LV,LoEnv,NewLoEnv,Res,Mem1,NewMem).
```

It dereferences "Goal" and "Head," respectively. Dereference means to get the contents of variables by tracing the reference chain. Notice that the dereference of the Goal is realized by sending message to "memory." On the other hand, the dereference of "Head" is realized by calling "deref" predicate directly. The "h_unify" predicate is called after "Goal" and "Head" is dereferenced. The following Table 1 shows how "h_unify" works.

		Goal		
		Variable @1	Atom	Compound Term
Head	Variable @2	@2 <- ref(@1)	@2 <- atom	@2 <- Term
	Atom	suspend	success / fail	fail
	Compound Term	suspend	fail	Decompose & unify

Table 1 Head unification table

Here, "@2 <- ref(@1)" means to "replace the value of @2 to the reference pointer to @1."

The transformation from the head unification table to the actual code is quite straightforward. The code shown below is the program fragment of "h_unify." Only part of "h-unify" is shown here because of our space limitation.

```
h_unify(GV,LV,LoEnv,NewLoEnv,Res,Mem,NewMem):-
    variable(GV),
    nonvariable(LV) |
    Res=susp,
    NewLoEnv=LoEnv,
    NewMem=Mem.
h_unify(GV,LV,LoEnv,NewLoEnv,Res,Mem,NewMem):-
    variable(GV),
    variable(LV),
    assign(LV,ref(GV),LoEnv,NewLoEnv),
    Res=success,
    NewMem=Mem.
```

Next, we describe "sys_exe" predicate. This predicate executes the system predicates existing in the body part of the clause. Here we show the description of unification and addition.

```
sys_exe((X=Y),T,NT,Mem,NewMem):- true |
    Mem=[unify(X,Y,Res) | NewMem],
    sys_exe1(Res,(X=Y),T,NT).
sys_exe(+(Z,X,Y),T,NT,Mem,NewMem):- true |
```

```
        Mem=[deref(X,XV),deref(Y,YV) | Mem1],
        add(Z,XV,YV,Mem1,NewMem,Res),
        sys_exe1(Res,+(Z,X,Y),T,NT).

add(Z,XV,YV,Mem,NewMem,Res):-
        ready_arg(XV,YV) |
        Ad:=XV+YV,
        Mem=[unify(Z,Ad,Res) | NewMem].
add(Z,XV,YV,Mem,NewMem,Res):-
        not_ready_arg(XV,YV) |
        Res=susp,
        NewMem=Mem.

sys_exe1(success,_,T,NT):- true |
        NT=T.
sys_exe1(susp,G,T,NT):- true |
        schedule(G,T,NT).
```

Memory part which manages the bindings of global variables can be described as follows:

```
memory([deref(Term,Value) | NMem],Db):- true |
        deref(Term,Value,Db),
        memory(NMem,Db).
memory([enter(Env) | NMem],Db):- true |
        enter(Db,Env,NDb),
        memory(NMem,NDb).
memory([unify(X,Y,Res) | NMem],Db):- true |
        unification(X,Y,Res,Db,NDb),
        memory(NMem,NDb).
memory([ ],Db):- true  | true.
```

You may notice that this "memory" is very intelligent. Instead of accepting low level primitives, such as read and write, it receives high level operations, such as "deref," "enter" and "unify." The access to this "memory" happens when (1) goal variables are dereferenced by head unification, (2) a clause is committed and local environment becomes global by "enter" operation and (3) system predicates at the body part are executed.

The "deref" and "enter" predicates are described as follows:

```
deref(Term,Value,Db):-
    variable(Term) |
    search_cell(Term,Cont,Db),
    deref1(Term,Cont,Value,Db).
deref(Term,Value,Db):-
    nonvariable(Term) |
    Value=Term.

deref1(_,ref(C),Value,Db):-true |
    deref(C,Value,Db). itemderef1(Cell,undf,Value,Db):- true |
    Value=Cell.
deref1(_,Cont,Value,Db):- true |
    Contref(_),
    Contundf |
    Value=Cont.

enter(Db,Env,NDb):- true |
    append(Env,Db,NDb).
```

The simplest list structure is assumed here for global database. The more complicated and the more efficient representation of the database are of course possible. Various optimization techniques, such as the use of Difference list and the optimizations of database update and retrieval are also possible.

The unification is described as follows:

```
unification(X,Y,Res,Db,NewDb):- true |
    deref(X,XV,Db),
    deref(Y,YV,Db),
    unify(XV,YV,Res,Db,NewDb).
```

Two arguments, X and Y, are dereferenced first and the "unify" predicate is called to perform unification. The following Table 2 shows how "unification" is carried out.

	Variable @1	Atom	Compound Term
Variable @2	@2 <- ref(@1)	@2 <- atom	@2 <- Term
Atom	@1 <- atom	success / fail	fail
Compound Term	@1 <- Term	fail	Decompose & unify

Table 2 Unification table

The examples of actual code fragments for "unify" are shown below:

```
unify(X,Y,Res,Db,NewDb):-
    variable(X),
    nonvariable(Y) |
    assign(X,Y,Db,NewDb),
    Res=success.
unify(X,Y,Res,Db,NewDb):-
    variable(X),
    variable(Y) |
    assign(X,ref(Y),Db,NewDb),
    Res=success.
unify(X,Y,Res,Db,NewDb):-
    nonvariable(X),
    nonvariable(Y),
    atomic(X),
    atomic(Y),
    X=Y |
    Res=success,
    NewMem=Mem.
```

5. Preliminary implementaition results

Preliminary implementaition has been carried out to evaluate the feasibility of our meta-interpreters. We have used PSI-II Machine [Nakashima 87] to measure the performance. Since the current version of PSI-II only understand ESP [Chikayama 84], we used GHC compiler which compiles GHC program to ESP.

Though ESP is a object-oriented dialect of Prolog, we did not use any object-oriented nature of ESP. Our GHC compiler is slightly modified from Ueda's Compiler [Ueda 85b], except that various system predicates are added for reflection support.

We have measured the execution time of the following two goals for for various types of meta-interpreters.

exec(append([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z],[end],S))

exec(qsort([4,2,7,1,6,3,5],S))

Here, "append" is the usual append program and "qsort" is the quicksort program. The measured execution time is shown in Table 3.

	Exec_1	Exec_2	Exec_3	Exec_4
append	1474	4708	4725	5034
qsort	1443	6741	15986	32635

(unit: msec)

Table 3 The measured execution time

Exec_1 denotes the simplest 4-line meta-interpreter. The "reduce" predicate is written in ESP in this case. Exec_2 also denotes 4-line meta-interpreter. However, "reduce" is written in GHC. Exec_3 is the three argument meta-interpreter, shown in Section 3.2. This interpreter include the scheduling queue. Exec_4 is the variable manageable meta_interpreter, shown in Section 4.

Our preliminary conclusion from these measurements are as follows:

(1) Exec_2 is currently slower than Exec_1. However, this problem can be solved shortly by implementing GHC directly by firmware.

(2) In the case of "qsort," Exec_3 is slower than Exec. We imagine this is invoked by the breadth-first scheduling algorithm which Exec_3 has been adopted. This problem is also conquered by adopting more sophisticated scheduling algorithm, such as bounded depth-first algorithm.

(3) Exec_4 is a little bit slower than Exec_3. However, it seems that variable management is not too much overhead for the meta-interpreter, comparing with the merits we can get from there. By applying various optimizations, mentioned before, we can expect much speedup to Exec_4.

Though our implementation is naive, the result we have got is not trivial. In sum, this shows the feasibility of variable manageable meta-interpreters.

6. Reflective operations in GHC

Implementing various reflective operations in GHC is not too difficult, once we get the enhanced meta-interpreter. We use the following seven argument "exec" in this section.

```
exec(H,T,Id,Mem,Res,MaxRC,RC)
```

Here, the first two arguments show the scheduling queue, the third is the starting identification number, the fourth is the communication channel to "memory," the fifth is the variable which receives the computation result, the sixth keeps the maximum reduction count allowed to his "exec" and the seventh is the current reduction count. Though we omit the program for this seven argument "exec," it can easily be constructed by combining various meta-interpreters, explained before.

There is no notion of job priority in this "exec." We sometimes need to execute goals urgently. Therefore, we also introduce the "express queue" to execute "express goals" which have the form, "G@exp." This can be realized by adding two more arguments, "EH" and "ET", which correspond to the express queue, to the seven argument "exec". The following two definitions describe the transition between the seven argument "exec" and nine argument "exec."

```
exec([G@exp | H],T,Id,Mem,Res,MaxRC,RC)
    :- true |
    exec([G | ET],ET,H,T,Id,Mem,Res,MaxRC,RC).
exec(ET,ET,H,T,Id,Mem,Res,MaxRC,RC) :-
    :- true |
    exec(H,T,Id,Mem,Res,MaxRC,RC).
```

If we come across the express goal, we simply call the nine argument "exec." The nine argument "exec" executes express goals first, and the reduced goals are also entered to the express queue. If the express queue becomes empty, we simply return to the seven argument "exec."

The next thing is to realize the reflective operations. Here, we consider six kinds of reflective operations, i.e., "get_q," "put_q," "get_rc," "put_rc," "get_env" and "put_env." "get_q" gets the current scheduling queue of "exec." "put_q" resets the current scheduling queue to the given argument. Similarly, "get_rc" and "put_rc" operate on "MaxRC" and "RC," "get_env" and "put_env" to the variable binding environment.

Since we already have got these as an internal state of the enhanced meta-interpreter, the implementations of these operations are fairly easy.

```
exec([get_q(NH,NT) | EH],ET,H,T,Id,Mem,Res,MaxRC,RC)
    :- true |
```

```
        RC1 := RC+1,
        NH = H,
        NT = T,
    exec(EH,ET,H,T,Id,Mem,Res,MaxRC,RC1).
    exec([put_q(NH,NT) | EH],ET,H,T,Id,Mem,Res,MaxRC,RC)
        :- true |
        RC1 := RC+1,
        exec(EH,ET,NH,NT,Id,Mem,Res,MaxRC,RC1).
```

"get_rc" and "put_rc" can be implemented similarly. In the case of "get_env" and "put_env," they send the express message to "memory" and they are processed in "memory."

6.1 Reflective programming example-1

We show an example which uses these reflective operations. This example shows the program which checks the current reduction count of "exec" and changes it, if the remaining reduction count is fewer than 100 reductions.

```
    check_rc :- true |
        get_rc(MaxRC,RC),
        RestRC := MaxRC-RC,
        check(MaxRC,RestRC).

    check(MaxRC,RestRC) :- 100>=RestRC |
        get_q(H,T),
        input(AddRC),
        NRC := MaxRC+AddRC,
        put_rc(NRC),
        schedule(check_rc@exp,T,NT),
        put_q(H,NT).
    check(MaxRC,RestRC) :- 100<RestRC |
        get_q(H,T),
        schedule(check_rc@exp,T,NT),
        put_q(H,NT).
```

The expected effect is gained running "check_rc@exp" goal together with user goals in "exec."

6.2 Reflective programming example-2

Thus far "exec" has been used to express "user process." However, it can be considered as a kind of "virtual processor" since it has a scheduling queue and a channel to "memory." This view of "exec" opens the new world. By connecting "exec" and "memory" to the architecture we imagine, we can construct a "virtual distributed computer."

Most processor usually has I/O. Therefore, it is convenient if "exec" has "input" and "output" in order to see "exec" as a "virtual processor." We use following "exec" in this section.

```
exec(H,T,Id,Mem,In,Out,MaxRC,RC)
```

Here, "In" denotes the input to this "exec" and "Out" denotes the output. We assume that user goals can be entered from "Input" and the goals which have postfix "@out" are sent out from "Output."

We define the following ring-connected distributed computer as an example.

```
m_ghc(In):-true |
    exec(H1,T1,(1,1),Mem1,C1,C2,_,0),
    exec(H2,T2,(2,1),Mem2,C2,C3,_,0),
    exec(H3,T3,(3,1),Mem3,C3,C4,_,0),
    exec(H4,T4,(4,1),Mem4,C4,C5,_,0),
    merge(In,C5,C1),
    merge4([Mem1,Mem2,Mem3,Mem4],Mem),
    memory(Mem,[ ]).
```

Four processors are connected to the uni-directed ring. Memory is shared by all processors.

It is possible to consider the "load balancing" problem on top of these virtual computers using reflective operations. Load balancing can be programmed as follows:

```
load_balance:-true |
    get_q(H,T),
    length(H,T,N),
    balance(N,H,T).

balance(N,H,T):-
    N>100,sub(N,100,X) |
    throw_out(X,H,T,NH,NT),
    load_balance@exp@out,
```

```
        put_q(NH,NT).
    balance(N,H,T):-
        N=<100 |
        load_balance@exp@out,
        put_q(H,T).
```

This goal gets the scheduling queue of the processor which is executing this goal and computes the current length of the queue. If it is longer than 100, the excessive goals and "load_balance@exp" goal are thrown out from the processor. If it is shorter than 100, it simply forward the "load_balance@exp" goal to output. By entering "load_balance@exp" goal from the "Input" of "m_ghc," this goal automatically circulates among processors and performs load balancing.

7. Conclusion

Starting from the simple 4-line self-description of GHC, we have made the stepwise enhancement to this meta-interpreter. A meta-interpreter which has variable management facility is described. As far as I know, this is the first attempt to try to manage variable bindings in logic programming language.

Though we used parallel logic language GHC in this paper, we imagine that the same kinds of things are also possible in Prolog. However, we feel that the use of the parallel logic programming language makes the programming easilier. In our language, we can create processes dynamically and the communication between processes can be expressed as streams. These language features helped to express the communication between the execution block and the memory block in a elegant manner.

Various reflective operations in GHC are also described. Implementaition of these operation is shown using an enhanced meta-interpreter. Examples of the usage of these operations are also shown.

In a sense, these reflective operations are very dangerous because we can easily access and change the internal state of the system. However, we can say that privileged users must have these capabilities for advanced system control. Our interest currently exists in examining the possibilities of the reflective operations in GHC.

We also admit that these reflective operations are defined in a very primitive manner. However, the more sophisticated handling of reflective operations and security considerations should be the topic which should be considered later, along with the problem of reflective tower.

8. Acknowledgments

This research has been carried out as a part of the Fifth Generation Computer Project. I would like to express my thanks to Yukiko Ohta and Fumio Matono, Fujitsu Social Science Laboratory, for their useful comments. I am indebted to them for part of this research. I would also like to express my thanks to Toshio Kitagawa, Hajime Enomoto and Koichi Furukawa for their encouragements and giving me the opportunity

to pursue this research.

9. References

[Bowen 82] K. Bowen and R. Kowalski, Amalgamating Language and Metalanguage in Logic Programming, Logic Programming, pp.153-172, Academic Press, London, 1982

[Bowen 83] D.L. Bowen et al., DECsystem-10 Prolog User's Manual, University of Edinburgh, 1983

[Chikayama 84] T. Chikayama, Unique Features of ESP, in Proc. of the International Conference on Fifth Generation Computer Systems 1984, pp.292-298, ICOT, 1984

[Clark 84] K. Clark and S. Gregory, Notes on Systems Programming in Parlog, in Proc. of the International Conference on Fifth Generation Computer Systems 1984, pp.299-306, ICOT, 1984

[Clark 85] K. Clark and S. Gregory, PARLOG, Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, Revised 1985

[Foster 87] I. Foster, Logic Operating Systems, Design Issues, in Proc. of the Fourth International Conference on Logic Programming, Vol.2, pp.910-926, MIT Press, May 1987

[Kurusawe 86] P. Kurusawe, How to Invent a Prolog Machine, in Proc. of Third International Conference on Logic Programming, LNCS-225, pp.138-148, Springer-Verlag, 1986

[Nakashima 87] H. Nakashima and K. Nakajima, Hardware Architecture of the Sequential Inference Machine: PSI-II, in Proc. of 1987 Symposium on Logic Programming, San Francisco, pp.104-113, 1987

[Shapiro 83] E. Shapiro, A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003, 1983

[Smith 84] B.C. Smith, Reflection and Semantics in Lisp, in Proc. of 11th POPL, Salt Lake City, Utah, pp.23-35, 1984

[Ueda 85a] K. Ueda, Guarded Horn Clauses, ICOT Technical Report, TR-103, 1985

[Ueda 85b] K. Ueda and T. Chikayama, Concurrent Prolog Compiler on Top of Prolog, in Proc. of 1985 Symposium on Logic Programming, Boston, pp.119-126, 1985

[Ueda 86] K. Ueda, Guarded Horn Clauses, Doctor of Engineering Thesis, Information Engineering Course, University of Tokyo, 1986