PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

Meta-interpreters and Reflective Operations in GHC

Jiro Tanaka

International Institute (IIAS-SIS),
Fujitsu Limited, 1-17-25 Shinkamata, Ota-ku, Tokyo 144
e-mail: jiro%flab.fujitsu.junet@uunet.uu.net

Abstract

Starting from the simple self-description of GHC [Ueda 85a, Tanaka 86a], we have derived various kinds of meta-interpreters by stepwise enhancement. Special attention is paid to the meta-interpreter which has variable management facility. 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 with its merits.

Various reflective operations, such as seen in [Smith 84, Maes 86], are also described. Implementations of these operations are shown using those enhanced meta-interpreters. Applications of reflective operations, i.e., "reduction count control," "garbage collection" and "load balancing," are also shown. This paper assumes 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 derive from the description of EVAL in LISP. It tries to describe its language features by itself. Since both of programs and data structures were 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):-sys(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, the predicate "clause" finds the definition of the given goal and the goal is decomposed to its definition. If it is a system-defined goal, it is solved directly. If all of these trials fail, this means that the execution of the given goal fails. Though this 4-line program is very simple, it certainly works as "Prolog in Prolog".

In this paper we discuss the self-description of parallel logic language GHC [Ueda 85a, Tanaka 86a]. 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 " | ." In GHC every clause definition includes the " | " operator. The predicate "clause" has also been replaced by "reduce." "reduce(P,Body)" finds clauses whose heads are unifiable to "P," selects one clause which satisfies guard, and instantiates "Body" to the body part of that clause. It is possible to define "reduce" as a user defined predicate. However, we assume it as a "primitive" function for the time being for simplicity. A more complete definition of "reduce" appears in Section 4.

If we compare this 4-line program with the self-description of Lisp, it seems to be too simple. It only simulates the top-level control flow of the given program. Therefore, we would like to "enhance" this 4-line program and obtain more useful information from the program.

2. Meta-interpreter enhancement

How we extend this meta-interpreter is our next problem. We gained a hint from Kurusawe's paper [Kurusawe 86]. He assumed an abstract Prolog machine which could 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 made explicit various hidden operations, such as unification or operations which handle memory structures, step by step to get Warren-like code.

However, we are not too interested in transforming the source program. We are interested in the description of "abstract machine," which we try to express in the form of a "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) Failsafe 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 & Gregory 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 & Gregory in parallel programming language [Clark 84].

(4) "exec(G,History)" which is the extension of the failsafe exec. It returns the execution history instead of result. This exec is useful to build debuggers. Various works have been done at ICOT in parallel logic languages.

These proposals seem 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 control.

3. Stepwise enhancement of meta-interpreters

What we would like to do is to derive various enhanced meta-interpreters, starting from the simple self-description of GHC. Our approach is similar to those which have already been proposed by Hirsch & Safra [Hirsch 86, Safra 86]. They have proposed various enhanced meta-interpreters, such as "trusted," "failsafe," "interruptible" and "controlled" meta-interpreters. Their motivations were mainly to develop those meta-interpreters suited for program execution in their "programming system." A "programming system" can be considered as a mini-operating system where one can input and execute user programs. However, our motivation is slightly different. We are more interested in "reflective" capabilities, such as seen in [Weyhrauch 80, Smith 84, Maes 86, Tanaka 88a, Watanabe 88]. We sometimes want to catch the current state of the system and modify it dynamically. These kinds of "reflective" capabilities seem to be very useful in writing a programming system. In 3-Lisp [Smith 84], we could 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 realize these "reflective" capabilities.

3.1 Two argument meta-interpreter

First extension is to get a "failsafe" 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).
```

The notion of "success" and "failure" appears in this two argument meta-interpreter. Here "success" means that the given goals are all processed successfully. "failure" occurs when system goal is failed or there are no commitable clauses in "reduce." We assume "Body" is instantiated to "false" in the latter case.

Note that the notion of "suspension" or "deadlock" cannot be detected by this meta-interpreter. When "suspension" occurs in "sys_exe" or "reduce," the execution of that predicate simply suspends.

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. The "continuation" of the program 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],T,R):-true | exec(H,T,R).
exec([false | H],T,R):-true | R=failure.
exec([P | H],T,R):-not_sys(P) |
    reduce(P,T,NT),exec(H,NT,R).
exec([P | H],T,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 that goal. In the former case, the goal is decomposed to sub-goals and they are simply appended to the tail of the scheduling queue.

We should note that goals are processed "sequentially" because we have introduced a scheduling queue. Thus far, we assumed that the goal execution was suspended when "suspension" occurred in "reduce" or "sys_exe." However, it means the suspension of the whole

system in this interpreter. Therefore, we have modified the meaning of "reduce" slightly. We assume that suspended goals are simply appended to the tail of the scheduling queue.

However, we should note that the introduction of the scheduling queue does not mean that the whole world becomes sequential. It simply means that the enqueuing and dequeuing processes just become sequential and the rest of the system can still work in parallel.

3.3 Five argument meta-interpreter

Then we introduce two more arguments, "MaxRC" and "RC," to control "reduction count." This kind of enhancement is motivated by [Foster 87]. We assume that "reduction count" 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([false | 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,R,MaxRC,RC):-
    sys(P),MaxRC>=RC |
    sys_exe(P,T,NT,RC,RC1),
    exec(H,NT,R,MaxRC,RC1).
exec([P | H],T,R,MaxRC,RC):-
    MaxRC<RC |
    R=count_over(RC).
```

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

3.4 Comparison with Hirsch's work

As mentioned before, Hirsch has proposed various enhanced meta-interpreters, such as "trusted," "failsafe," "interruptible" and "controlled" ones. Though motivations are mutually different, there is some correspondence between our approach and Hirsch's approach.

Our initial 4-line program corresponds to Hirsch's "trusted" meta-interpreter. Our two-argument meta-interpreter corresponds to his "failsafe" one. However, there is no correspondence to the three argument or the five argument meta-interpreters. Since "interruptible" and "controllable" meta-interpreters have been invented for their programming system, we are not too interested in them and have developed our meta-interpreters to the other direction.

We should note that our approach is consistent with Hirsch's one. It is possible to add Hirsch's extension to our three argument or the five argument meta-interpreters. Though we do not mention those approach in this paper and concentrate on "reflective" operations, the examples of such approaches can be seen in [Tanaka 88b].

Hirsch & Safra has also emphasized the use of partial evaluation or source-to-source program transformation techniques to increase the execution efficiency of enhanced meta-interpreters [Hirsch 86, Safra 86]. These techniques are also applicable to our extensions [Kohda 88].

4. Variable managing meta-interpreter

Varieties of meta-interpreters are described in the previous section. However, compared with Lisp, these meta-interpreters are "incomplete" since variables can be "shared" in our meta-interpreter and accessed from outside. It means that the execution of a goal can be influenced by the outside environment.

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

```
:- exec(X=0,R1),exec(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. In both cases, the whole system fails when X=2 is executed. This example shows that even the "failsafe" meta-interpreter may fail, in the case it has shared variables. To realize "complete" meta-interpreters, meta-interpreters must have the facility to manage its own local variable environment. Therefore, we consider the meta-interpreter which explicitly handles variables.

The toplevel description of a variable managing meta-interpreter can be expressed as follows:

```
m_ghc([FGoal | In],Out):-FGoal≠halt |
    transfer(FGoal,NGoal,1,Id,Env),
    H=[NGoal | 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 four processes, i.e., "transfer," "exec," "memory" and "print_result" processes. Among these, "transfer" process is a relatively short-live process. The remaining three processes live

long until the execution of the given goal "FGoal" is completed. Figure 1 shows the snapshot how processes are generated in accordance with the user's input.

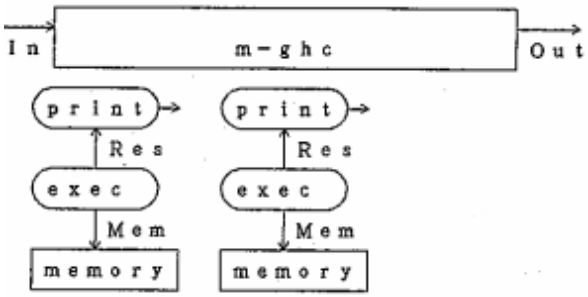

Figure 1 Creation of processes in m_ghc

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 initial identification number of variables. The fourth argument "Id" denotes the identification number which should be assigned next. For example, when "transfer" allocated numbers from 1 to n, n+1 is assigned to "Id." The fifth argument "Env" contains the correspondence between the identification number and its value, i.e., the correspondence is kept as a list of "(@number, value)" tuples in "Env." Example of these forms are shown below:

```
(@1, undf)      ... the value of @1 is undefined
(@2, 100)       ... the value of @2 is the integer 100
(@3, ref(@2)) ... the value of @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" predicate can be described as follows:

```
exec(T,T,Id,Mem,Res):-true |
    Res=success,Mem=[ ].
exec([false|H],T,Id,Mem,Res):- true |
    Res=failure,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 that it has a "channel" to "memory" as its fourth argument. When the value of a variable is needed in "reduce" or "sys_exe," it explicitly sends messages to "memory.' The third argument of "exec" contains the identification number which should be assigned next in "reduce." The fifth argument contains the execution result.

The following is the definition of predicate "reduce."

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

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

In the predicate "reduce," "clauses" constructs the list of potentially unifiable clauses "FClauses" from the given goal "P." "resolve" selects one "FClause" from "FClauses" and instantiates "Body" to the body of that "FClause." "Body" is appended to the tail "T" of the queue and "NT" denotes the new tail of the queue.

The predicate "resolve" picks up one "FClause" from "FClauses." "transfer" generates "NClause" and local environment "LoEnv" from "FClause." "transfer" predicate is exactly the same as before except that "NClause" is created instead of "NGoal." "try_commit" checks whether "NClause" can be committed for the given goal "P" or not. "Res" is instantiated to "success," "failure" or "suspension," depending on the result. We should note that "failure" is handled as exactly same as "suspension" in this meta-interpreter. Though it is certainly possible to distinguish "failure" from "suspension," we omit that because of implementation simplicity.

The predicate "resolve1" sets "Id1" and "Mem2" of "resolve," if try_commit is succeeded. Otherwise it calls "resolve" again.

The predicate "try_commit" can be described as follows:

```
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(Res3,_,_,_,Res,Mem,NewMem):-
    Res3≠success |
    NewMem=Mem,
    Res=suspension.
```

This "try_commit" 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.

In "head_unification" predicate, "Goal" and "Head" expresses two terms which should be unified. "LoEnv" is the local environment of the clause. The new local environment after head_unification is stored in "LoEnv1." The result of head_unification is put to "Res." "Mem" and "Mem1" are used for keeping the communication to "memory."

"solve_guard" tries to solve the given guard "G" using the local environment "LoEnv." "Res2" and "LoEnv2" stores the evaluation result and new local environment, respectively.

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

The definition of "head_unification" is shown below:

```
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).
```

"head_unification" performs the dereferencing of "Goal" and "Head" first. Dereferencing means to get the contents of variables by tracing the reference chain. Notice that the dereferencing of the Goal is realized by sending message to "memory." On the other hand, the dereferencing of "Head" is realized by calling "deref" predicate directly. The "h_unify" predicate is executed after the dereferencing. 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	suspension	success/failure	failure
	Compound Term	suspension	failure	Decomp.& unify

Table 1 Head unification table

Here, for example, "@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 space limitation.

```
h_unify(GV,LV,LoEnv,NewLoEnv,Res,
    Mem,NewMem):-
    variable(GV),nonvariable(LV) |
    Res=suspend,
    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=suspend,
    NewMem=Mem.

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

```
T=[G | 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 |
    append(Env,Db,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) dereferencing of goal variables are requested 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).
deref1(_,Cont,Value,Db):-
    Cont≠ref(_) |
    Value=Cont.
```

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).
```

The dereferencing of two arguments, X and Y, takes place 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/failure	failure
Compound Term	#1 ← Term	failure	Decomp.& unify

Table 2 Unification table

For example, the following is a part of code fragments for "unify":

```
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):-
    atomic(X),
    atomic(Y),
    X=Y |
    Res=success,
    NewDb=Db.
```

5. Preliminary implementation results

Preliminary Implementations have been carried out to evaluate the feasibility of our meta-interpreters. We have measured the execution time of the following two goals 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([3,2,6,1,8,4,9,5,7],S))
```

Here, "append" is the usual append program and "qsort" is the quicksort program.

We have tried two kinds of implementations. The first implementation is done on PSI-II Machine [Nakashima 87]. Since the current version of PSI-II only understands ESP [Chikayama 84], we used GHC compiler which compiles GHC programs to ESP. Our GHC compiler is a slightly modified version of Ueda's Compiler [Ueda 85b] which compiles GHC programs to Prolog. Though ESP is an object-oriented dialect of Prolog, we did not use object-oriented nature of ESP in an essential manner.

The measured execution time is shown in Table 3.

	Exec_1	Exec_2	Exec_3	Exec_3'	Exec_4
append	1088	3455	3471	3466	7419
qsort	1535	8360	20875	8375	20168

(unit: msec)

Table 3 The measured execution time on PSI-II

Here, Exec_1 denotes the simplest 4-line meta-interpreter. We have written the "reduce" predicate in GHC. Exec_2 denotes the two argument meta-interpreter shown in Section 3.1. Exec_3 is the three argument meta-interpreter, shown in Section 3.2. This interpreter includes the scheduling queue. Exec_3' is also the three argument meta-interpreter. The difference is that we have used depth-first scheduling algorithm, instead of breadth-first scheduling algorithm used in Exec_3. Exec_4 is a variable managing meta_interpreter as shown in Section 4. Slightly different to Section 4, we have assumed depth-first scheduling algorithm to increase execution efficiency.

Our observations are as follows:

(1) Exec_2 is approximately three or five times slower than Exec_1.

(2) In the case of "append," Exec_3 is as fast as Exec_2. However, Exec_3 is much slower than Exec_2 in the case of "qsort." This seems to come from the breadth-first scheduling algorithm we adopt. If we use depth-first scheduling algorithm, the three argument meta-interpreter becomes as fast as Exec_2, as shown in Exec_3'.

(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 its merits.

In this implementation, GHC programs must be compiled to ESP programs before execution. Since the execution speed depends on how GHC programs are compiled, we have felt that it is very difficult to make the fair comparison of these interpreters. For example, the execution speed of Exec_1 highly depends on whether the "reduce" predicate is written in GHC or in ESP.

Therefore, the second implementation has been tried on Sequent S-81 (Symmetry) using PDSS [ICOT 88], which is the GHC system written in C. Because PDSS is directly written in C without using ESP or Prolog, we can make the fair comparisons of various meta-interpreters.

The measured execution time on Sequent S-81 is shown in Table 4.

	Exec_1	Exec_2	Exec_3	Exec_3'	Exec_4
append	80	830	890	850	7020
	116	2740	2795	2770	26810
qsort	130	2610	7100	2650	17710
	190	8370	21595	8419	54748

unit: { msec / reductions }

Table 4 The measured execution time on S-81

In this table, upper rows show the execution time measured in "msec" and lower rows show the measured "number of reductions," though the former is approximately proportional to the latter.

If we compare Table 4 with Table 3, we notice that Exec_1 is much faster in S-81. The relative speed of Exec_4, compared with Exec_3 or Exec_3', is a little slower in S-81. However, it does not seem to be the big difference between two. In summary, though our implementation method is very primitive and many optimizations are possible on our interpreters from now on, these measurements seem to show the feasibility of variable managing meta-interpreters.

6. Reflective operations in GHC

Implementing various reflective operations, such as seen in [Weyhrauch 80, Smith 84, Maes 86, Tanaka 88a, Watanabe 88], 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 initial 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 this "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 normal speed, i.e., 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: reduction count control

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

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

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

When "check_rc@exp" goal is executed, the system goes into the "express" state. In "express" state, we can assume that the normal execution of goals are "frozen." In "express" state, "check_rc" tries to get "MaxRC" and current "RC." Then it computes the remaining reduction count "RestRC." In the case "RestRC" is less than 100, it gets "AddRC" from the user, computes "NRC," stores this number as the new "MaxRC" of the system, and returns to the normal state after adding "check_rc@exp" goal to the tail of the current scheduling queue.

6.2 Reflective programming example 2: garbage collection

Next example of reflective programming is the program which use "get_env" and "put_env."

As described in Section 4, environment has been implemented as a list of (@number, value) tuples. When new clauses are committed or system predicates are executed, new variable bindings are entered to "memory." It means that environment increases monotonously, as the computation advances. Therefore, we sometimes want to "garbage collect" this environment.

The following program shows the "garbage collection" program. By inserting "gc@exp" goal to the appropriate place of the user program, garbage collection takes place when that goal is executed.

```
gc :- true |
    get_q(H,T),
    get_env(Env),
    do_gc(H,T,Env,NH,NT,NEnv),
    put_q(NH,NT),
    put_env(NEnv).

do_gc([G | H],T,Env,NH,NT,NEnv):-true |
    deref_all(G,NG,Env,Env1),
    NH=[NG | NH1],
    do_gc(H,T,Env,NH1,NT,Env2),
    do_merge(Env1,Env2,NEnv).

do_gc(T,T,Env,NH,NT,NEnv) :-true |
    NH=NT,
    NEnv=[ ].
```

In this program, "deref_all" performs the dereferencing of all variables in the given goal "G" using "Env." It creates the new goal "NG" and its new local environment "Env1" by cutting off the unnecessary tuples from "Env." These new local environments are merged by "do_merge" to create "NEnv."

6.3 Reflective programming example 3:

load balancing

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 would like to constract, we can make a "virtual distributed computer" [Tanaka 86b].

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

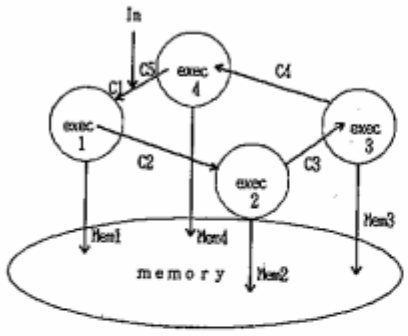

Figure 2 Ring-connected distributed computer

As shown in Figure 2, four processors are connected to the uni-directed ring. The output of one processor is connected to the input of the other processor. The memory channels are all merged and connected to one "memory" process. (Figure 2 is a little bit simplified in this point.)

It is possible to consider the "load balancing" problem on the top of these virtual computers using reflective operations. Load balancing can be programmed as shown below. If we enter "load_balance@exp" goal from the "Input" of "m_ghc," this goal automatically circulates among processors and performs load balancing.

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

balance(N,H,T):-
    N>100 |
    X:=N-100,
    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.
```

When "load_balance@exp" is executed, the current scheduling queue of the processor are taken out and the length of the queue is computed. If it is longer than 100, "throw_out(X,H,T,NH,NT)" picks up X excessive goals from the scheduling queue (H,T), throws these goals out and enters remaining goals to "NH" and "NT" in Difference list form. "load_balance@exp" goal is also thrown out to invoke load balancing to other processors.

If it is shorter than 100, it simply forwards the "load_balance@exp" goal to output.

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 also described. Though the attempts to manage variable bindings in meta-interpreters have already been tried by [Hirata 87] and [Koshimura 87], we believe that this is the first attempt which tries to show the relations of meta-interpreters and reflective operations in parallel logic languages.

We used parallel logic language GHC in this paper. However, we imagine that the same kinds of arguments are also possible in Prolog. We feel that the use of the parallel logic programming language makes the programming easier. 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 an elegant manner.

Various reflective operations in GHC are also described. Implementations of these operations are shown using the enhanced meta-interpreters. Application examples of these reflective operations are also shown. Since our primary objective was to show the feasibility of reflective operations, we kept our meta-interpreters as simple as possible. Therefore, we cut off unessential implementation techniques. For example, we omitted the detection of deadlock and used busywaiting strategy for suspended goals. We do not think that techniques used in this paper directly scale to a distributed environment, although these are the research topic we currently working for [Tanaka 88b].

We also admit that, in a sense, 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.

These reflective operations are defined as an ad hoc 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 of Japan. I would like to express my thanks to Fumio Matono and Yukiko Ohta for their useful comments and discussions. Funio Matono also worked for the actual implementation and the performance evaluation. Thanks also go to Youji Kohda and Hiroyasu Sugano for their usuful discussions. Also 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

[Hirata 87] M. Hirata, Parallel List Processing Language Oc and its Self-description, Computer Software, Vol.4, No.3, pp.41-64, 1987 (in Japanese)

[Hirsch 86] M. Hirsch, W. Silverman and E. Shapiro, Layers of Protection and Control in the Logix System, Weizmann Institute of Science Technical Report CS86-19, 1986

[ICOT 88] ICOT, PDSS User's manual, Version 1.5, ICOT, 1988 (in Japanese)

[Kohda 88] Y. Kohda and J. Tanaka, Deriving a Compilation Method for Parallel Logic Languages, Logic Programming '87, LNCS 315, Springer-Verlag, pp. 80-94, 1988

[Koshimura 87] M. Koshimura, Description of Full GHC by Flat GHC, ICOT internal memorandum, PSH-I-A-KL1-598, 1987 (in Japanese)

[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

[Maes 86] P. Maes, Reflection in an Object-Oriented Language, in Preprints of the Workshop on Metalevel Architectures and Reflection, Alghero-Sardinia, October 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

[Safra 86] S. Safra and E. Shapiro, Meta Interpreters for Real, in Proc. of IFIP 86, pp.271-278, 1986

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

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

[Tanaka 86a] J. Tanaka et al., Guarded Horn Clauses and Experiences with Parallel Logic Programming, in Proc. of FJCC '86, ACM, Dallas, Texas, pp.948-954, November 1986

[Tanaka 86b] J. Tanaka et al., Distributed Implementation of FGHC -Toward the Realization of Multi-PSI System-, ICOT, Technical Report TR-159, March 1986

[Tanaka 88a] J. Tanaka, Metaprogramming and Reflection, In bit, Vol.20, No.5, pp.41-50, May 1988 (in Japanese)

[Tanaka 88b] J. Tanaka, A Simple Programming System Written in GHC and Its Reflective Operations, in Proc. of The Logic Programming Conference '88, ICOT, Tokyo, pp.143-149, April 1988

[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

[Watanabe 88] T. Watanabe and A. Yonezawa, Reflection in an Object-Oriented Concurrent Language, in Proc. of ACM Conf. on OOPSLA, San Diego, September 1988

[Weyhrauch 80] R. Weyhrauch R, Prolegomena to a Theory of Mechanized Formal Reasoning. In Artificial Intelligence 13, pp.133-170, 1980