ICOT Technical Report: TR-506

TR-506

An Experimental Reflective Programming System Written in GHC

by
J. Tanaka (Fujitsu)

September, 1989

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

An Experimental Reflective Programming System Written in GHC

Jiro Tanaka
IIAS-SIS, FUJITSU LIMITED,
1-17-25 Shinkamata, Ota-ku, Tokyo 144, JAPAN

Abstract

A programming system can be defined as an *environment* where one can input programs and execute goals. After describing two approaches, i.e., meta-extension and reflective-extension, for enhancing meta-interpreters, they are combined together and *reflective* operations on such meta-interpreters are discussed. Based on these meta-interpretation techniques, an experimental reflective programming system (ExReps) is described. The whole system consists of two layers, i.e., abstract machine layer and execution system layer, and both layers are totally written in *parallel logic language* GHC. Two examples of reflective programming, i.e., *load balancing* and *dynamic reduction count control*, are shown. An actual program execution example on ExReps is also shown.

1. Introduction

Various kinds of *parallel logic languages* have been proposed so far. PARLOG [6], Concurrent Prolog [15] and GHC [25] are examples of such languages. In these languages, we can create *processes* dynamically and express the *synchronization* between processes quite easily.

Therefore, it seems to be quite natural to try to describe an *operating system* in these languages. In fact, various proposals have been made for *systems programming* from the very beginning of *parallel logic languages* [5] [16]. PPS (PARLOG Programming System) [8] and Logix [18] are the examples of such systems.

In this paper, we try to describe an experimental reflective programming system written in GHC. A programming system can be defined as a small *operating system* where one can input programs and execute goals. Our objective is not building up a practical programming system like PPS or Logix. Rather, our interest exists in expressing a *simple* programming system more systematically and more concise manner. We would also like to test new features of a programming system such as *reflective* operations.

The organization of this paper is as follows. Section 2 describes two approaches for enhancing *self-description* of GHC. After describing meta-extension and reflective-extension, we combine them together and describes about *reflective* operations. Based on these techniques, an experimental reflective programming system (ExReps) is shown in Section 3. The whole system consists of two layers, i.e., the abstract machine layer and the execution system layer. Both layers are described using enhanced meta-interpreters. Two examples

of reflective programming are shown in Section 4. The actual program execution example on ExReps is shown in Section 5.

2. Enhanced self-descriptions

The original notion of self-description seems to derive from the description of EVAL in LISP. Both of programs and data structures are expressed as S-expressions. *Self-description* tries to describe the features of the language in itself, i.e., it tries to describe the evaluation step of the program in itself.

In Prolog world, the following 4-line program has been known as *Prolog in Prolog* or *vanilla* interpreter [1].

```
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. Though this 4-line program is very simple, it certainly works as *Prolog in Prolog*.

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. 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.

We would like to *enhance* this 4-line program and obtain various enhanced meta-interpreters. How should we extend our meta-interpreter? It seems that there exist two directions for that.

2.1. Meta-extension

One direction is developing various meta-interpreters to *control* program execution in the programming system. We call this extension as *meta-extension*. This approach is similar to those which have already been proposed by [10]. This extension aims at obtaining *object-level* information from *object-level* world to *meta-level* world. Here, *meta-level* means the *top-level* where the execution of programs are performed, and *object-level* means the interpreted-level where the execution of programs is done in an interpretive manner inside a meta-interpreter.

There exists various *object-level* information. The "`success`" and "`failure`" of goal execution can be considered as *object-level* information. Input and output can also be considered as such.

The simple *failsafe* meta-interpreter, such as seen in [5] can simply be obtained by making the notion of "success" and "failure" explicit in the 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).
```

Here "success" means that the given goals are all processed successfully. "failure" occurs when system goal is failed or there are no committable clauses in "reduce."

Similarly *interruptible* meta-interpreter can be defined as follows:

```
exec(true,In,Out):-true|Out=[success].
exec((A,B),In,Out):-true|
        exec(A,In,O1),exec(B,In,O2),merge_result(O1,O2,Out).
exec(A,In,Out):-sys(A),var(In)|sys_exe(A,In,Out).
exec(A@io,In,Out):-var(In)|Out=[A@io].
exec(A,In,Out):-not_sys(A),var(In)|
        reduce(A,In,Body,Out,NewOut),exec(Body,In,NewOut).

exec(A,[suspend|In],Out):-true|wait(A,In,Out).
exec(A,[abort|In],Out):-true|Out=[aborted].

wait(A,[resume|In],Out):-true|exec(A,In,Out).
wait(A,[abort|In],Out):-true|Out=[aborted].
```

This "exec" has also been proposed by [5]. It has two streams, "In" and "Out," which explicitly connect meta-level and object-level. The "success" and "failure" of the object goal execution is processed as a message to the meta-level by using "Out" stream. If the goal has the form "A@io," it means the i/o operation and the goal is sent to "Out" stream. Note that "var(In)" is the special predicate which checks the absence of messages in the argument variable. Also, "Out" stream can transmit *failure* or *exception* information, produced by "sys_exe" and "reduce," to the *meta-level.*

This "exec" is very useful if we would like to control the program execution from the meta-level. We can "suspend," "resume," or "abort" the execution of the given goal by sending appropriate messages from "In" stream.

2.2. Reflective-extension

The other direction for extending meta-interpreter is to realize *reflective* capabilities, such as seen in [12] [19]. We sometimes want to catch the current state of the system and modify it dynamically. We call this extension as *reflective-extension.* By using this extension *object-level* program can obtain *meta-level* information. This capabilities seem to be very useful in writing advanced programming system.

The extension depends on what kind of resources we want to control. We sometimes want to manage processes dynamically at execution time. Therefore, we introduce a *scheduling queue* explicitly in our meta-interpreter. The *enhanced* meta-interpreter can be shown as follows.

```
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 [15]. 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 appended to the *tail* of the scheduling queue.

Next we introduce two more arguments, "MaxRC" and "RC," to control *reduction count*. This enhancement is motivated by [9]. 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):-MaxRC<RC|
        R=count_over(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).
```

Notice that "reduce" or "sys_exe" increments "RC" by one when the actual computation takes place.

2.3. Combining together

Though we showed meta-extension and reflective-extension in the previous sections, they are not conflicting to each other. In fact, they can be combined together as shown below.

```
exec(T,T,In,Out,MaxRC,RC):-true|
        Out=[success(reduction_count=RC)].
exec([true|H],T,In,Out,MaxRC,RC):-true|
        exec(H,T,In,Out,MaxRC,RC).
```

```
exec(H,T,In,Out,MaxRC,RC):-MaxRC>=RC|
        Out=[count_over].

exec([A|H],T,In,Out,MaxRC,RC):-
        sys(A),var(In),MaxRC>RC|
        sys_exe(A,T,NT,RC,RC1,Out,NOut),
        exec(H,NT,In,NOut,MaxRC,RC1).
exec([A@io|H],T,In,Out,MaxRC,RC):-
        var(In),MaxRC>RC|
        Out=[A@io|NOut],
        RC1=RC+1,
        exec(H,T,In,NOut,MaxRC,RC1).
exec([A|H],T,In,Out,MaxRC,RC):-
        not_sys(A),var(In),MaxRC>RC|
        reduce(A,T,NT,RC,RC1,Out,NOut),
        exec(H,NT,In,NOut,MaxRC,RC1).

exec(H,T,[Mes|In],Out,MaxRC,RC):-true|
        control_exec(Mes,H,T,In,Out,MaxRC,RC).
exec(H,T,[goal(G)|In],Out,MaxRC,RC):-true|
        NH=[G|H],
        exec(NH,T,In,Out,MaxRC,RC).
```

As mentioned before, *meta-extension* essentially cares for *object-level* information to the *meta-level*. On the other hand, reflective extension cares for *meta-level* information to the *object-level*. In this interpreter, we see these two extensions are combined in a harmonious manner.

2.4. Reflective operations in an enhanced interpreter

Implementing various reflective operations, such as seen in [12] [19] is not too difficult, once we get the enhanced meta-interpreters.

Reflective operations must be executed *urgently*. However, there is no notion of job priority in our interpreter. Therefore, we introduce the notion of *express queue* to execute *express goals*, which have the form "G@exp," as a preparation.

We add two more arguments, which correspond to the *express queue*, to our interpreter. The following two definitions describe the transition between the normal state and express state.

```
exec([G@exp|H],T,In,Out,MaxRC,RC):-var(In)|
        exec([G|ET],ET,H,T,In,Out,MaxRC,RC).
exec(ET,ET,H,T,In,Out,MaxRC,RC):-var(In)|
        exec(H,T,In,Out,MaxRC,RC).
```

The first two arguments of eight-argument "exec" correspond to *express queue*. If the normal six-argument "exec" comes across the express goal, the eight-argument "exec" is called and it enters the *express state*.

In *express* state, the normal execution of goals are *frozen*. It "exec" only executes express goals, and the reduced goals are also entered to the express queue. If the express queue becomes empty, it simply returns to the normal state.

We consider four kinds of reflective operations, "get_rc," "put_rc," "get_q" and "put_q," of which *object-level* program can make use. The meaning of these operations is shown in Figure 1.

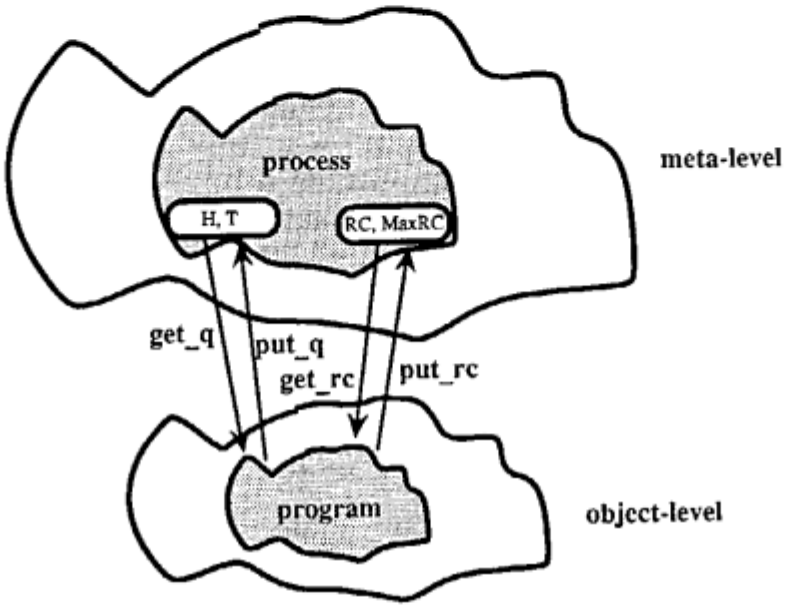

Figure 1 Reflective Operations in GHC

Two kinds of meta-information are *reified* here, i.e., *scheduling queue* and *reduction count*. "get" operations obtain meta-information for the *object-level*. On the other hand, "put" operations return the information to the *meta-level*.

The meaning of each operation is as follows: "get_rc" gets "MaxRC" and "RC" from the meta-level, "put_rc" resets "MaxRC" to the given argument, "get_q" gets the current scheduling queue, and "put_q" resets the scheduling queue to the given arguments. The followings are the definitions of these reflective operations:

```
exec([get_rc(Max,C)|EH],ET,H,T,In,Out,MaxRC,RC):-true|
        Max:=MaxRC,C:=RC,RC1:=RC+1,
        exec(EH,ET,H,T,In,Out,MaxRC,RC1).
exec([put_rc(C)|EH],ET,H,T,In,Out,MaxRC,RC):-true|
        RC1:=RC+1,
        exec(EH,ET,H,T,In,Out,C,RC1).
exec([get_q(NH,NT)|EH],ET,H,T,In,Out,MaxRC,RC):-true|
        RC1:=RC+1,NH=H,NT=T,
        exec(EH,ET,H,T,In,Out,MaxRC,RC1).
exec([put_q(NH,NT)|EH],ET,H,T,In,Out,MaxRC,RC):-true|
        RC1:=RC+1,
```

```
exec(EH,ET,NH,NT,In,Out,MaxRC,RC1).
```

The implementation is quite straightforward since *scheduling queue* and *reduction count* have already been *explicit* in the meta-interpreter.

3. ExReps programming system

We have already described the enhanced meta-interpreters. As stated before, these extensions provide us the basis for building up our programming system.

ExReps, which stands for "Experimental Reflective Programming System," has actually been built up based on these techniques [22]. Since our ExReps also consists of huge amount of codes, we try to show the simplified version of the system and describe how the programming system will be constructed using meta-interpreter techniques.

3.1. Overall structure

The overall structure of ExReps is shown in Figure 2. ExReps is implemented on PSI-II Machine [13]. Since the current version of PSI-II only understands ESP which is the object-oriented dialect of Prolog [2], we install GHC system first. This GHC system is a slightly modified version of [26] and executes the compiled GHC programs.

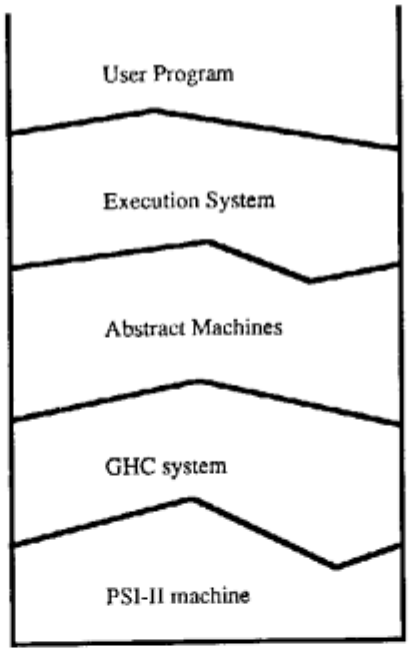

Figure 2 Overall structures of ExReps

We expect ExReps should also be adaptable to distributed hardware. Therefore, we construct distributed abstract machine layer on top of GHC system. The construction of this layer is originally motivated by [23]. The execution system, which loads user programs and executes user jobs, is constructed on top of these abstract machines.

3.2. Communication to the external world

How to realize the communication to the external world is the critical problem to implement a programming system. One possibility is utilizing i/o operations which have side-effects. However, this means that i/o is handled as somewhat extra-logical things. Another possibility is assuming a conceptual process which corresponds to the actual device.

We adopted the latter approach, i.e., we assumed a conceptual process for each physical device. Every conceptual process has a single stream to which we can send messages. We have prepared "`catch`" predicate to catch the stream from the physical device. For example, a stream to the *printer* can be caught by executing "`catch(printer,X).`" This predicate is a special one, which we can execute only once in our program for each device.

For simplicity, we assumed these conceptual processes always *consume* the stream. We can manipulate the device by sending an appropriate message sequence from the stream. (Note that i/o operations executed inside meta-interpreters are actually transformed to the message of the *meta-level* stream in our system.)

3.3. Window and keyboard controller

The typical i/o device we assume here is a *window* which is created on a bitmap display. It is possible to input and output messages from there. Of course, *window* is not a physical device, since we can create as many number of windows as we want. It should be called as a *virtual* device. In our approach, *virtual* processes can be created by the *create* predicate in a program.

For example, a window is created when "`create(window,X)`" is executed. The input and output to the window are expressed as messages to stream "`X`." (The actual input is completed when we move the cursor to the window and type in messages from the keyboard.) We assume that the virtual processes which correspond to devices are deleted by instantiating the stream "`X`" to "`[]`."

However, we should note that input and output are executed as a message to the window stream, i.e., the window does not accept keyboard input without a request from the program. Therefore, we need *keyboard controller* program which always generates this request. This can be written as follows:

```
keyboard(Out,In):-true|
        Out=[input([@],T)|Out1],
        keyboard(T,Out1,In).

keyboard(halt,Out,In):-true|
        Out=[],
        In=[].
keyboard(T,Out,In):-goal_or_command(T)|
        In=[T|In1],
        Out=[input([@],T1)|Out1],
        keyboard(T1,Out1,In1).
```

Here, we assume that the first argument "`Out`" is connected to the window and the second argument "`In`" is connected to GHC programs. Note that "`input([@],T)`" out-

puts "@" first, and user's input will be entered into "T," i.e., "@" is used as a *prompt* for user input.

3.4. Abstract machine layer

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 reduction counter. This view of "*exec*" opens the new world. By connecting "exec" we can construct a *virtual distributed computers*.

To manage input and output from "exec," we also prepare the network manager "nm." For example, we can define the following ring-connected distributed computers by using "exec" and "nm."

```
d_machine:-true|
        nm(Nm4,Nm1,In1,Out1),exec(T1,T1,In1,Out1,_,0),
        nm(Nm1,Nm2,In2,Out2),exec(T2,T2,In2,Out2,_,0),
        nm(Nm2,Nm3,In3,Out3),exec(T3,T3,In3,Out3,_,0),
        nm(Nm3,Nm4,In4,Out4),exec(T4,T4,In4,Out4,_,0).
```

Four "nm" processes are connected to the uni-directed ring. The output of one network manager is connected to the input of the other network manager. Each "nm" is also connected to a "exec." The scheduling queue of "exec" is initially empty. User goals can be entered in "exec" from "In."

Inside of each "exec," the ordinary GHC program runs. However each "exec" can throw goals which has *pragma* [17] to other "exec" through "Out" stream. We assume that goal "A" which has pragma "@P" is expressed as "A@P." The kind of *pragma* depends on the topology of abstract machines. We assume that pragma "@forward" is used for uni-directed ring.

The handling of *pragma* is carried out by simply adding the following definition to the six-argument "exec."

```
exec([A@P|H],T,In,Out,MaxRC,RC):-
        is_pragma(P),var(In),MaxRC>RC|
        Out=[A@io|NOut],
        RC1=RC+1,
        exec(H,T,In,NOut,MaxRC,RC1).
```

Each "nm" delivers the goal with *pragma*. If the goal has the pragma, "nm" simple peels off the outermost pragma and sends the remaining part to the next "nm." The goal which has no pragma is dropped to the "exec." Therefore, goal "A@forward@forward" will be dropped to "exec" located ahead by two.

However, you may notice that these distributed computers are isolated from the external world. The program of distributed machines with i/o becomes as follows:

```
d_machine:-true|
        create(window,0),
        keyboard(O1,I),
        merge(I,Nm4,Nm4'),
```

```
nm(Nm4',Nm1,In1,Out1),exec(T1,T1,In1,Out1,_,0),
nm(Nm1,Nm2,In2,Out2),exec(T2,T2,In2,Out2,_,0),
nm(Nm2,Nm3,In3,Out3),exec(T3,T3,In3,Out3,_,0),
nm(Nm3',Nm4,In4,Out4),exec(T4,T4,In4,Out4,_,0),
dist(Nm3,Nm3',O2),
merge(O1,O2,O).
```

Figure 3 shows the overall structure of the distributed machines. Comparing to the previous program, a *window* and a *keyboard controller* are added for the interface to the outer world. "`merge`" is added to join input stream "`I`" to "`Nm4`." We assume that goal "`A@io`" simply goes through "`nm`." "`dist`" captures the i/o goals and send them to "`O2`."

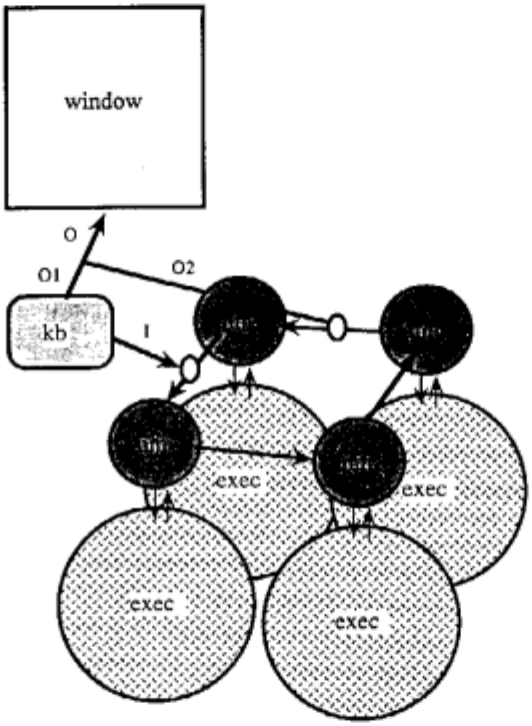

Figure 3 Distributed abstract machines

Note that the distributed computers shown here is the extremely simplified version of actual ExReps system. In the actual system, we can build various kinds of distributed abstract machines and there exist different *pragmas* for different topologies. Though "`@forward`" was used for uni-directed ring, we use "`@right`," "`@left`," "`@up`" and "`@down`" for square mesh. Potentially it is also possible to apply more complicated *pragma* strategy such as seen in [3] [11].

3.5. Execution system layer

In this section, we first describe the *shell* which plays the central role in the execution system. Then we describe *database server* which provides us the capability to load user programs. After that, we try to assemble these parts into the *execution system*.

3.5.1. Shell

The shell creates the user task or enters the program to the database, depending on messages from the user. The following is the program for the *shell*.

```
shell([],Db,MaxRC,Out):-true|
        Db=[],Out=[].
shell([goal(Goal)|In],Db,MaxRC,Out):-true|
        create(Window,WOut),
        keyboard(KOut,EIn),
        exec([Goal|T],T,EIn,EOut,MaxRC,0),
        shell(In,Db,MaxRC,Out),
        merge(KOut,EOut,WOut).
shell([db(Message)|In],Db,MaxRC,Out):-true|
        Db=[Message|Db1],
        shell(In,Db1,MaxRC,Out).
```

The "shell" has four arguments; the first is the input stream, the second is the stream to the *database server*, the third is the internal state which specifies the maximum reduction count allowed for the user process, and the fourth is the output stream.

This program works as follows: If the input stream of "shell" is "[]," it means the end of input. All streams will be closed in this case. If "goal(Goal)" is in the input stream, the enhanced meta-interpreter which contain "Goal" as its argument is created. A window and a keyboard controller are also created accompanied with *exec*. If "db(Message)" is in the input stream, "Message" is sent to the *database server*.

Figure 4 shows the snapshot where processes are created in accordance with the user input. Note that each *exec* has their own window and keyboard. Once created they run independently from the *shell*. Since the keyboard controller always try to request input from the window, the user can input the commands from the window to *exec*.

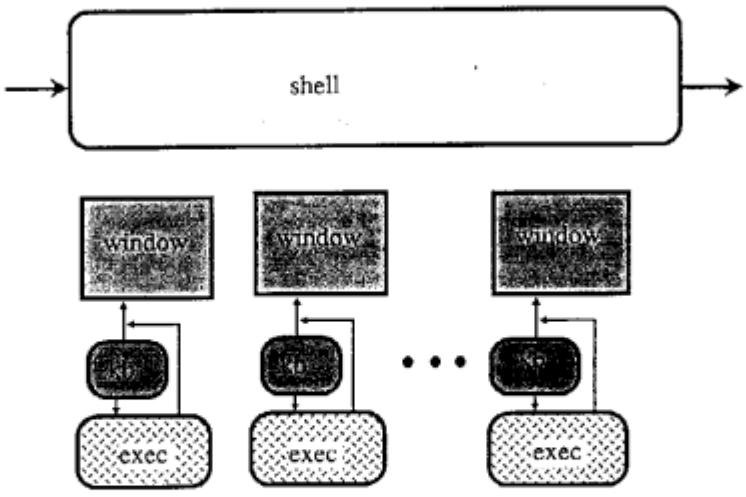

Figure 4 The creation of processes in *shell*

3.5.2. Database server

In the execution system, we need a database capability which can add, delete and check program definitions. In naive implementation, we need to consult *db_server* every time we execute user-defined predicates. In fact, PPS tries to realize the database in such way [7]. However, this is very complicated and this may cause a database access bottleneck. Therefore, we have implemented *db_server* as follows, using side-effects:

```
db_server([add(Code)|In],ready,Out):-true|
        add_definition(Code,Done,Out,Out1),
        db_server(In,Done,Out1).
db_server([delete(Name,Arity)|In],ready,Out):-true|
        delete_definition(Name,Arity,Done,Out,Out1),
        db_server(In,Done,Out1).
db_server([definition(Name,Arity)|In],ready,Out):-true|
        definition(Name,Arity,Done,Out,Out1),
        db_server(In,Done,Out1).
```

The "`db_server`" predicate has three arguments. The first argument is the input from the system. The second argument "`Done`" is used to sequentialize database access. The third is the output to the system.

3.5.3. Building up the execution system

Now we connect components together and building up the execution system [22]. The example is shown below:

```
exe_system:-true|
        create(window,Out),
        keyboard(Out1,In),
        shell(In,Db,MaxRC,Out2),
        db_server(Db,ready,Out3),
        merge3(Out1,Out2,Out3,Out).
```

This program can also be illustrated in Figure 5. A *window*, a *keyboard controller*, a *shell* and *database server* are connected together. The outputs of the *shell*, *database server* and *keyboard controller* are connected together to the *window*. Since the *keyboard controller* always generates the read request to the system window, we can always input goals from it.

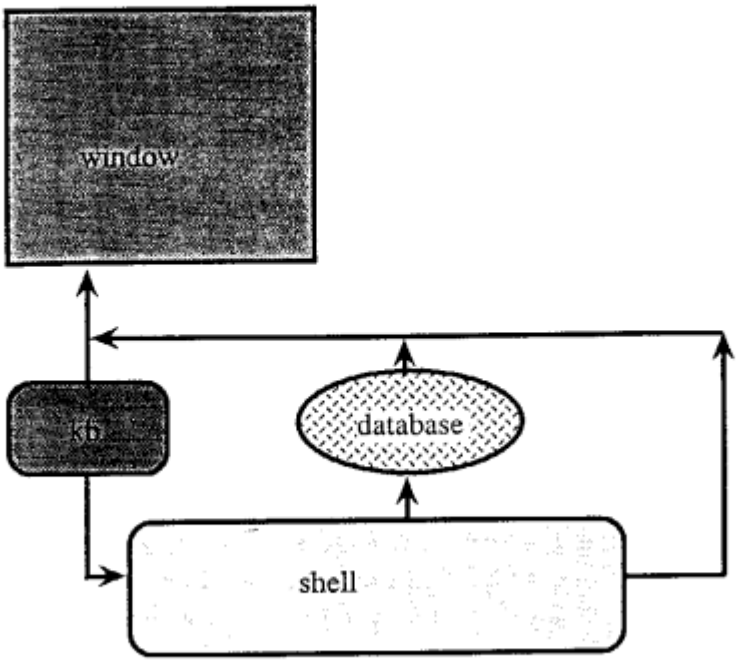

Figure 5 Execution system layer

This program should be installed on top of abstract machines. Since this program does not include *pragma*, all components are installed in one abstract machine. However, we should note that *user programs* can be executed on several abstract machines by putting appropriate *pragma* to user programs.

4. Reflective programming examples

In this section, we show two examples of reflective programming. The first is the *load balancing* program, the second is *dynamic reduction count control.*

4.1. Load balancing

The first example is the *load balancing* program which is executed directly on top of abstract machine. It is possible to consider the *load balancing* problem using reflective operations. The *load balancing* program is shown below. If we enter "`load_balance@exp`" goal as a goal which is executed on the abstract machine, this goal automatically circulates among abstract machines 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@forward,
        put_q(NH,NT).
balance(N,H,T):-N=<100|
        load_balance@exp@forward.
```

When "load_balance@exp" is executed inside an abstract machine, it goes into the *express* state. In *express* state, the current scheduling queue of the abstract machine is 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 excessive goals from the scheduling queue (H,T), puts these goals into "X," throws "X" out and put remaining goals into "NH" and "NT" in Difference list form. "load_balance@exp" goal is also thrown out to invoke load balancing to other abstract machines. If it is shorter than 100, it simply forwards the "load_balance@exp" goal to other abstract machines.

4.2. Dynamic reduction count control

The second example is *dynamic reduction count control* program which is executed at the *user program level*. The following program shows how to define the "check_rc" predicate which checks the current reduction count of the system and changes it if allowed 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([reduction_increment,@],AddRC),
        NRC:=MaxRC+AddRC,
        put_rc(NRC),
        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, it 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.

We assumed these *reflective* operations in a very primitive manner. In a sense, these are very dangerous because we can easily access and change the internal state of the system. However, we need these capabilities for advanced system control. Our current interest exists in examining the reflective operations in a programming system.

5. User program execution on ExReps

An example of the actual program execution on ExReps is shown in Figure 6. As mentioned before, we install GHC system first on PSI-II. Then we execute the *abstract machine construction* program. The execution of this program opens "VM_window" in PSI-II display (the upper-left corner of Figure 6). We specify the topology and the size of the network. In ExReps system, we can build various distributed abstract machines, such as *linear array, square mesh, hexagonal mesh, ring, tree* and *hyper cube.*

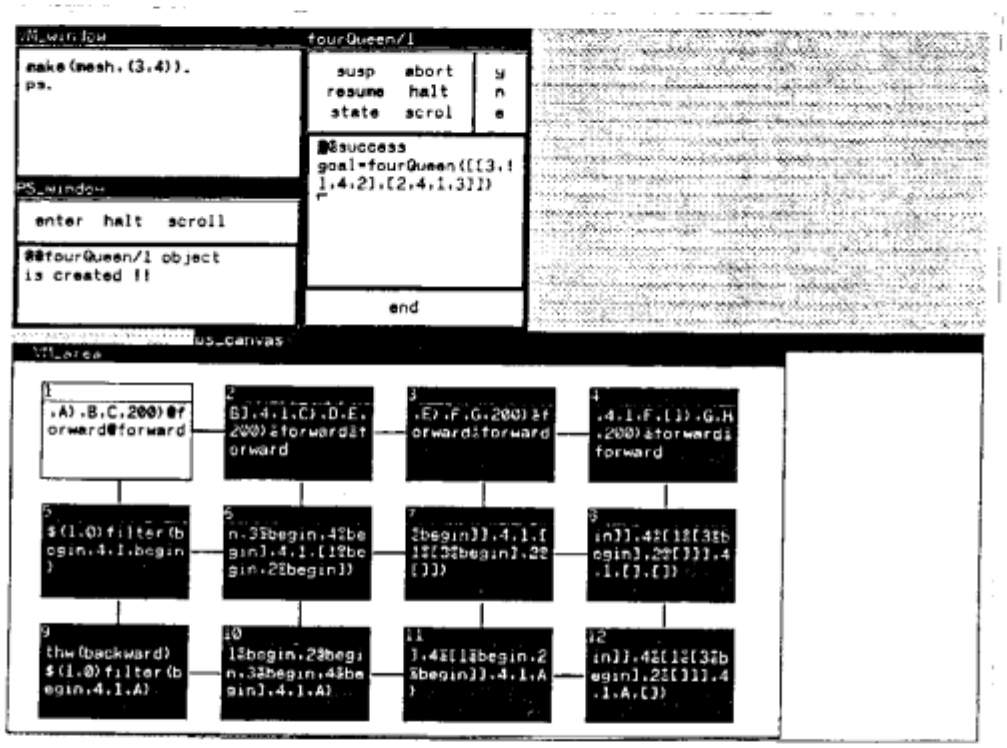

Figure 6 User program execution on ExReps

In this case, we typed in "make(mesh,(3,4))," which created the 3 by 4 mesh network. The abstract machines are shown in "VM_area" window (lower-left corner). Here, each abstract machine is also shown as a window and the executing goals are displayed in real time manner.

Then we construct *execution system layer* on top of *abstract machines.* This is done simply by typing in "ps" from "VM_window," which creates "PS_window." We can input user programs and goals from "PS_window." We loaded the distributed *four queen* program and executed it. When we input the goal, it automatically creates the user-process window ("fourQueen/1" window in this case). We can "suspend," "resume," or "abort" the execution of the goal dynamically by sending appropriate messages from this window.

This *four queen* program contains *pragma* and has been executed in a distributed manner.

6. Conclusion

Two approaches, i.e., *meta-extension* and *reflective-extension*, are shown for enhancing GHC meta-interpreters. Though several works, such as [5] [10] [14], have been investigated for *meta-extension*, no previous work is proposed for *reflective-extension* and *reflective operations* in parallel logic programming world.

In our approach, the extension depends on what kind of resources we want to control. Since we wanted to control the *scheduling* of processes and *computation time*, we introduced *scheduling queue* and *reduction counter* explicitly. Other resources, such as *variable environment*, can also be controlled in a similar manner [21]. Though *reflective* operations are defined as an ad hoc way, defining it reflective operations in more sophisticated way, such as seen in [19] [27], is also possible.

In relate to ExReps, the current version is implemented on PSI-II. However, we imagine that the extension to distributed hardware, such as Multi-PSI [24], will not be difficult. Our approach can be classified as *software-oriented* approach. In contrast, PIMOS [4] tries to implement their "**exec**" (*Sho'en*) directly as a built-in predicate.

PIMOS tries to realize various features of distributed operating system in *machine-dependent* or *hard-wired* way. All key features, such as exception handing, task management and load balancing, are concentrated to this "exec" implementation.

On the other side, *reflective operations* work as *wires* which connect *meta-level* and *object-level* in our approach. The *object-level* world can obtain *meta-level* information through these *wires*. User can write *object-level* programs which handle *meta-level* information and the *modified met-level* information can be *reflected back* to the *meta-level*. These result the flexible and powerful system which consists of small core. Therefore, we believe that our approach is more suitable, especially in the case of distributed environment.

Also note that the programs shown here are the extremely simplified version. The more complete version of ExReps, running on PSI-II, has already been demonstrated at FGCS'88 and is available from the author [22].

7. 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 for their programming support. I am indebted to them for part of this research. I would also like to express my thanks to Youji Kohda, Hiroyasu Sugano, Kazunori Ueda and Koichi Furukawa for their discussions and encouragements.

References

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

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

[3] T. Chikayama, Load balancing in a very large scale multi-processor system, Proceedings of the Fourth Japanese-Swedish Workshop on Fifth Generation Computer Systems, SICS, 1986.

[4] T. Chikayama, H. Sato and T. Miyazaki, Overview of the parallel inference machine operating system (PIMOS), Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pp.230-251, ICOT, November 1988.

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

[6] 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.

[7] K. Clark and I. Foster, A Declarative Environment for Concurrent Logic Programming, TAPSOFT'87, Lecture Notes in Computer Science 250, pp.212-242, 1987.

[8] I. Foster, The Parlog Programming System (PPS), Version 0.2, Imperial College of Science and Technology, 1986.

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

[10] 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.

[11] Y. Kohda, New pragma scheme for parallel logic programming and its application to OS, Proceedings of the 35th Annual Convention IPS Japan, pp.743-744, 1987, in Japanese.

[12] P. Maes, Reflection in an Object-Oriented Language, Preprints of the Workshop on Metalevel Architectures and Reflection, Alghero-Sardinia, October 1986.

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

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

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

[16] E. Shapiro, Systems programming in Concurrent Prolog, Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, pp.93-105, ACM, January 1984.

[17] E. Shapiro, Systolic programming: A paradigm of parallel processing, Proceedings of the International Conference on Fifth Generation Computer Systems 1984, pp.458-471, ICOT, 1984.

[18] W. Silverman, M. Hirsch, A. Houri and E. Shapiro, The Logix System User Manual, Version 1.21, Weizmann Institute, Israel, July 1986.

[19] B.C. Smith, Reflection and Semantics in Lisp, in Proceedings. of Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, pp.23-35, ACM, January 1984.

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

[21] J. Tanaka, Meta-interpreters and Reflective Operations in GHC, Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pp.774-783, ICOT, November 1988.

[22] J. Tanaka, Experimental Reflective Programming System "ExReps," Demonstration material of the International Conference on Fifth Generation Computer Systems 1988, ICOT, November 1988.

[23] S. Taylor, E. Av-Ron and E. Shapiro, A Layered Method for Process and Code Mapping, chapter 22, Concurrent Prolog: collected papers, ed. by E. Shapiro, Vol.2, pp.78-100, The MIT Press, 1987.

[24] S. Uchida, K. Taki, K. Nakajima, A. Goto and T. Chikayama, Research and development of the parallel inference system in the intermediate stage of the FGCS project, Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pp.16-36, ICOT, November 1988.

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

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

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