ICOT Technical Report: TR-641

TR-641

Reflective Guarded Horn Clauses: Language Implementation and Programming

by
J. Tanaka & F. Matono (Fujitsu)

May, 1991

© 1991, ICOT

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Reflective Guarded Horn Clauses: Language Implementation and Programming

Jiro Tanaka[†] and Fumio Matono[‡]

[†]IIAS-SIS, FUJITSU LABORATORIES LTD.,
1-17-25 Shinkamata, Ota-ku, Tokyo 144, JAPAN
Email: jiro@iias.flab.fujitsu.co.jp

[‡]FUJITSU SOCIAL SCIENCE LABORATORY LTD.,
Parale-Mitsui-Build., 8 Higashida-cho, Kawasaki 210, JAPAN

Abstract

We consider the meta-level representation of *Guarded Horn Clauses* (GHC) first. We implement GHC meta-computation system by enhancing the simple 4-line GHC meta-program. Then we describe *Reflective Guarded Horn Clauses* (RGHC) system, where *reflective tower* can be constructed and collapsed in a dynamic manner. This system has the following two features comparing to the previous approaches: First, this system is formulated without using *quote*. Secondly, it has the *reflective* mechanism in which *reflective tower* can be constructed using *reflective predicates*. RGHC has actually been implemented. A simple execution example is also shown in this paper. This paper assumes a basic knowledge of parallel logic languages.

1. Introduction

If we look for an ideal programming language, it must be simple and, at the same time, powerful language. Looking back the history of programming language, we note that the developments of the programming language are generated by the repeated trials which look for such languages within a limitation of the available hardware.

Recently, it seems that the mechanism, called *meta* or *reflection*, is attracting wide spread attention in programming language community [Maes 88]. We have already proposed to introduce *reflection* mechanism into parallel logic program language GHC [Ueda 85] [Tanaka 86] and shown several examples, mainly from application aspects [Tanaka 88a] [Tanaka 88b] [Tanaka 90]. However, the reflection mechanisms have been introduced in an *ad hoc* manner and they lacked the generality as seen in 3-Lisp [Smith 84]. Therefore, we would like to describe *Reflective Guarded Horn Clauses* (RGHC), which has the expressive power compatible to 3-Lisp, in this paper.

The organization of this paper is as follows. In Section 2, we try to describe the meta-computation system of GHC. After considering meta-presentation of the object-level system, we describe GHC meta-computation system by enhancing the simple 4-line

GHC meta-program. The language features of RGHC and several reflective programming examples are described in Section 3. RGHC implementation is described in Section 4. An actual program execution example is shown in Section 4. Related works and conclusion are described in Section 6.

2. Meta-computation system in GHC

A meta-system can be defined as a computational system whose problem domain is another computational system. The program and data of the meta-system model another computation system. This another computational system is called the *object-system*. Especially, the program of *meta-system* is called *meta-program* and it models the algorithm of the problem solving at the object-level. On the other hand, the data of *meta-system* models the structure of the *object-system*, i.e., the data of *meta-system* contains the representation of the *object-system*.

2.1. A simple GHC meta-program

In Prolog world, a simple 4-line program is well-known as *Prolog in Prolog* or *vanilla* interpreter [Bowen 83]. The GHC version of this program can be described as follows:

```
exec(true):-true|true.
exec((P,Q)):-true|exec(P),exec(Q).
exec(P):-user_defined(P)|reduce(P,Body),exec(Body).
exec(P):-system(P)|sys_exe(P).
```

Using this meta-program, we can execute a goal as an argument of "`exec`." This program tries to execute the given goal in an interpretive manner. We can see two levels here, *meta-level*, where the top level execution is performed, and *object-level*, where the goal execution is simulated inside the meta-program.

The meaning of this meta-interpreter is as follows: If the given goal is "`true`," the execution of the goal succeeds. If it is a sequence, it is decomposed and executed separately. In the case of a user-defined goal, the predicate "`reduce`" finds the clause which satisfies the guard and the goal is decomposed to the body goals of that clause. If it is a system-defined goal, it is executed directly.

Though this 4-line program is very simple, it certainly works as *GHC in GHC*. However, this *GHC in GHC* is insufficient as a real meta-program because of the following reasons.

- There is no distinction between the variable at the meta-level and the one at the object-level. Therefore, we cannot manipulate or modify object-level variables at the meta-level. For example, we cannot check whether the given variable is bound, nor can we check whether the given variable is identical to the other one.
- The predicate "`reduce(P,Q)`" finds *potentially unifiable clauses* for the given argument "`P`." In this case, object-level program must also be defined as a program. Therefore, we cannot manipulate the object-level program without using *assert* or *retract*.

- This program only simulate the top level execution of the program and we cannot obtain the more detailed executing information such as *current continuation*, *environment* or *execution result*.

Therefore, we would like to propose the real meta-computation system which does not have the disadvantages described above.

2.2. Meta-level representation of the object-level system

First, we consider how the object-level construct of GHC system should be represented at the meta-level. Those can be summarized as follows:

2.2.1. Constants, function symbols and predicate symbols

We assume that constants, function symbols and predicate symbols are expressed by the same symbols. The other possibility is using *quote* to distinguish the level. In this approach, '3 (quote three) corresponds to the 3 at the object-level. 3-Lisp and Gödel [Lloyd 88b] adopt this approach. However, we do not adopt this approach.

In Lisp, both of programs and data are expressed as S-expression. In evaluating a program, *quote* is essentially used to separate data from the program and to stop the evaluation. However, in logic programming languages, there exists a clear separation between predicates and functors. Logic programming languages do not have a notion of *evaluation*. They simply find out the binding of variables contained in the initial query. Though the implementation of *quote* is not difficult, our claim is that there is little merit in using *quote* in logic programming languages.

2.2.2. Variables and variable bindings

As explained previously, we cannot manipulate object-level variables well if it is expressed as variables. To manipulate object-level variables, we need the information about the representation of variables, i.e., we need to know where and how the given variable is realized.

Therefore, we use a *special ground term* to express an object-level variable. This *special ground term* has a one-to-one correspondence to the object-level term and is distinguished from the ordinary *ground term*.

An object-level variables are expressed as "@number" at the meta-level. A unique number is assigned for each variable. Though we are afraid that this representation of variables is not abstract enough, comparing to the approach using *quote*, we have chosen it for implementation simplicity. Similarly, we also assume that the object-level variable is expressed as "@!number" at the meta-meta-level, "@!!number" at the meta-meta-meta-level, and so on.

The variable bindings at the object-level, i.e., *environment*, can *conceptually* be represented as a list of address-value pairs at the meta-level. The followings are the examples of such pairs.

```
(@1, undf) ... the value of @1 is undefined
(@2, a)    ... the value of @2 is the constant ''a''
(@3, @2)   ... the value of @3 is the reference pointer
```

```
                    to @2
(@4,f(@1,@2))
                ... the value of @4 is the structure whose
                    function symbol is ''f,'' the first argument
                    is the reference pointer to @1, and the
                    second argument is the reference pointer to @2
```

We can regard these pair as expressing the memory cells of the object-level. Similar to the ordinary Prolog implementation, reference pointers are generated when two variables are unified. Therefore, we need to *derefer* pointers when the value of a variable is needed.

2.2.3. Terms and object-level programs

Keeping consistency with the notations explained before, we denote object-level terms by corresponding meta-level *special ground terms*, where every variable is replaced by its meta-level notation.

For example, the object-level term "`p(a,[H|T],f(T,b))`" is expressed as "`p(a,[@1|@2],f(@2,b))`" at the meta-level. It is also expressed as "`p(a,[@!1|@!2],f(@!2,b))`" at the meta-meta-level.

On the other hand, the program of object-level, i.e., the collections of *guarded* Horn clause definitions, are expressed as a *ground term* at the meta-level, where all variables are replaced by "`var(number)`" notation. For example, the following "`append/3`" program

```
append([A|B],C,D):-true|
        D=[A|E], append(B,C,E).
append([],A,B):-true|A=B.
```

is expressed as

```
((append,3),
   [(append([var(1)|var(2)],var(3),var(4)):-true|
        var(4)=[var(1)|var(5)], append(var(2),var(3),var(5))),
    (append([],var(1),var(2)):-true|var(1)=var(2))])
```

at the meta-level.

2.3. An enhanced meta-program

The simple GHC meta-program in Section 2.1 can be enhanced to fit to the requirements of the real meta-program using the meta-level representation in Section 2.2. The enhancement can be done by making *explicit* what is *implicit* in the simple GHC meta-program.

- There was no distinction between the variable at the meta-level and the one at the object-level. We express object-level variables as *special ground terms* at the meta-level.
- We manipulate object-level program as a *ground term* at meta-level. "`exec`" keeps it program as its argument.

- "exec" also keeps its *goal queue* and *variable bindings* for expressing *continuation* and *environment* in its arguments.

The top level description of GHC meta-system can be written as follows:

```
m_ghc(Goal,Db,Out) :- true|
        transfer(Goal,GRep,1,Id,Env),
        exec([GRep],Env,Id,Db,NEnv,Res),
        make-result(Res,GRep,NEnv,Out).
```

For given object-level goal "Goal" and given object-level program "Db," "m_ghc" puts out the computation result to "Out." "transfer" changes given goal "Goal" to object-level representation "GRep." In "GRep," every variable in "Goal" has been replaced to "@number" form. The third argument of "transfer" stands for the starting identification number which is used to transfer the given goal. The fourth argument contains the identification number which should be used next. The fifth contains the variable bindings of this goal representation.

For example, if we input "exam([H|T],T)" to "Goal," "transfer(exam([H|T],T),GRep,1,Id,Env)" is executed. The computation result becomes

```
GRep = exam([@1|@2],@2)
Id = 3
Env = [(@1,undf),(@2,undf)].
```

The enhanced "exec" executes this *goal representation* and the computation result "Out" will be generated by "make_result" predicate.

The enhanced "exec" has six arguments. These six arguments, in turn, denote the *goal queue*, the *environment*, the *starting identification number*, the *program*, the new *environment* and the *execution result*. The enhanced "exec" can be programmed as follows:

```
exec([],Env,Id,Db,NEnv,R)
        :- true|
           (NEnv,R)=(Env,success).
exec([true|Rest],Env,Id,Db,NEnv,R)
        :- true|
           exec(Rest,Env,Id,Db,NEnv,R).
exec([false|Rest],Env,Id,Db,NEnv,R)
        :- true|
           (NEnv,R)=(Env,failure).
exec([GRep|Rest],Env,Id,Db,NEnv,R)
        :- user_defined(GRep,Db)|
           reduce(GRep,Rest,Env,Db,NGRep,Env1,Id1),
           exec(NGRep,Env1,Id1,Db,NEnv,R).
exec([GRep|Rest],Env,Id,Db,NEnv,R)
        :- system(GRep)|
           sys_exe(GRep,Rest,Env,NGRep,Env1),
           exec(NGRep,Env1,Id,Db,NEnv,R).
```

Though we omit the detailed explanation, the meaning of this program is self-explanatory. We easily notice that this is the extension of the simple GHC meta-program in Section 2.1. Note that the use of *list* for expressing *goal queue* imposes us inefficiency and some sequentiality. The *difference list* is used in the actual implementation. Also note that the use of *shared-variable* and *short-circuit* techniques [Hirsch 86] [Safra 86] might be effective in a truly parallel computing environment.

3. Reflective Guarded Horn Clauses

Reflection is the capability to feel or modify the current state of the system dynamically. The form of *reflection* we are interested in is the *computational reflection* proposed by [Smith 84] and [Maes 86]. A reflective system can be defined as a computational system which takes its computation system as its problem domain.

In 3-Lisp, the meta-system and the object-system are exactly the same computation system. A meta-system is dynamically created when *reflective* procedures are called at the object-level. However, there is a possibility that *reflective* procedures are called while executing the meta-system. In this case, the system creates the meta-meta-system and the control transfers to that system. Similarly, it is possible to consider the meta-meta-meta-system, the meta-meta-meta-meta-system, and so on. Conceptually, it creates the infinite tower of meta.

If a computational system has such reflective capability, it becomes possible to catch the current state while executing the program and takes the appropriate action according to the obtained information.

3.1. Two approaches in implementing reflection

There exist two approaches realizing such reflective system. One is utilizing a meta-system. We modify the meta-program and add the means of communication between the meta-level and the object-level, namely, we prepare a set of built-in predicates which can catch or replace the current state of the object-level system. If we adopt this approach, it becomes possible to catch or modify the *internal state* of the executing program by using those built-in predicates.

In [Tanaka 88b] and [Tanaka 90], we proposed several reflective built-in predicates, such as "`get_q`," "`put_q`," "`get_env`" and "`put_env`." "`get_q`" gets the current goal queue of "`exec`." "`put_q`" resets the current goal queue to the given argument. Similarly, "`get_env`" and "`put_env`" operate on the variable binding environment. Though this approach has a merit that the implementation is relatively straightforward, we should note that this approach is not the accurate implementation of *reflection*. It is because the *internal state* is always changing, even while processing the obtained information at the object-level.

The other way is to create meta-system dynamically when needed. If a *reflective* predicate is called from the object-system, the meta-system is dynamically created and the control transfers to the meta-level in order to perform the necessary computation. When the meta-level computation terminates, the control automatically returns to the object-level. This mechanism was originally proposed by B. C. Smith in 3-Lisp. Comparing to the first approach, this method has the merit that the distinction of levels are more clear. Also this is the more accurate implementation of *reflection* because the object-level

system is *frozen* while performing the meta-level computation. Note that we can realize the meta-system and the object-system using the same computation system. In such case, it becomes possible to execute *reflective* predicates also at the meta-level and dynamically create a meta-meta-level. Conceptually, it is possible to imagine the infinite tower of meta, i.e., *infinite reflective tower.*

We adopted the second approach in implementing *Reflective Guarded Horn Clauses* (RGHC). RGHC is the *reflective extension* of GHC and can be defined as a superset of GHC. Language features and the outline of the implementation are shown in the followings.

3.2. Reflective predicate

Reflective predicates are user-defined predicates which invoke *reflection* when called. *Reflective predicates* can be defined quite easily. It can used wherever we want, in the user program or in the initial query. Similar to 3-Lisp, it is possible to *access* or *modify* the internal state of the computation system by this predicates.

For example, reflective predicate for goal "`p(A,B)`" can be defined as follows:

```
reflect(p(X,Y),(G,Env,Db),(NG,NEnv,NDb)):- guard | body.
```

We should note that "`p(A,B)`" is changed to "`p(X,Y)`" and two extra arguments, i.e., "`(G,Env,Db)`" and "`(NG,NEnv,NDb)`" are added. When the goal "`p(A,B)`" is called at the object-level, we automatically shift one level up and this goal is executed at the meta-level. (See Figure 1.) At this level, "`p(A,B)`" is transformed to "`p(X,Y),`" where "`X`" and "`Y`" are the meta-level representation of the arguments. The computation state of the object-level is also expressed as"`(G,Env,Db),`" where "`G`" represents the *remaining goals* which should be executed at the object-level, "`Env`" represents the *variable bindings* and "`Db`" represents the object-level program. "`(NG,NEnv,NDb)`" denotes the new computation state of the object-level to which the system should return when the meta-level execution finishes. We assume that the value of these variables are instantiated while executing meta-level goals. When we finished executing this reflective goal, we automatically shift one level down and "`(NG,NEnv,NDb)`" becomes to the new object-level state.

For example, a reflective predicate "`var(X,R),`" which checks whether the given argument "`X`" is unbound or not, can be defined as follows:

```
reflect(var(X,R),(G,Env,Db),(NG,NEnv,NDb))
        :- unbound(X,Env) |
           add_env((R,unbound),Env,NEnv),
           (NG,NDb)=(G,Db).

reflect(var(X,R),(G,Env,Db),(NG,NEnv,NDb))
        :- bound(X,Env) |
           add_env((R,bound),Env,NEnv),
           (NG,NDb)=(G,Db).
```

Since an object-level variable is handled as a *special ground term* and its value is contained in the *environment*, we examine the representation of *environment* to check

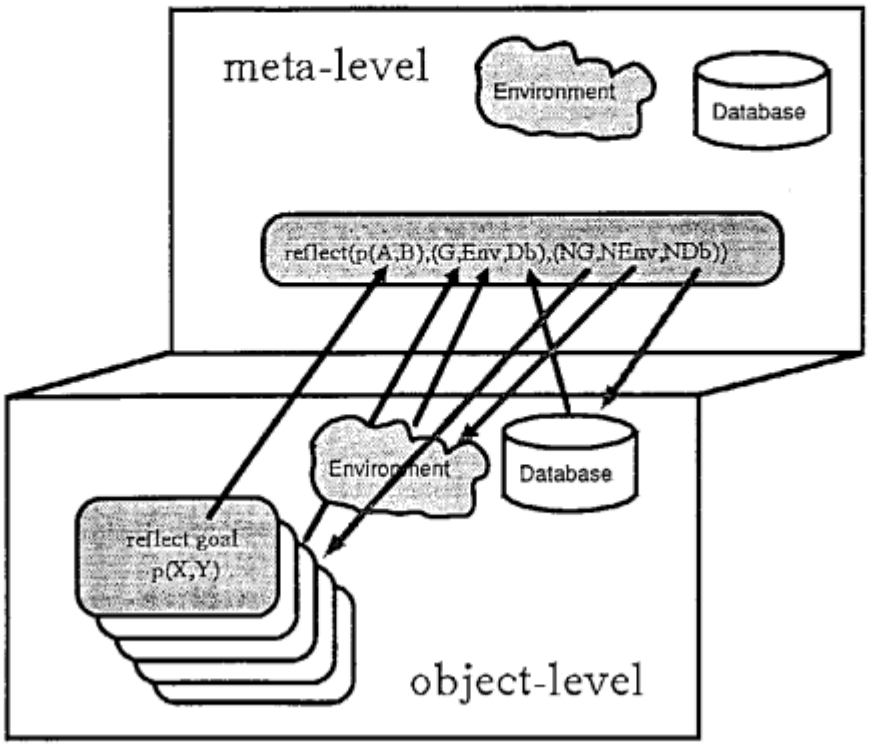

Figure 1: Execution of the reflective predicate

whether the variable is bound or not and the result is added to the environment list as a value of "R."

Note that the value of "R" may be the reference pointer to another variable. Therefore, the "add_env((R,Value),Env,NEnv)" predicate need to *derefer* "R" first. Then it replaces the value of the dereferred address. It can be written as follows:

```
add_env((N,Value),Env,NEnv)
        :- true |
           deref(N,Env,N2),
           remove(N2,Env,Env2),
           NEnv=[(N2,Value)|Env2].
```

The "current_load(N)" predicate, which obtains the number of goals in the *goal queue* of the object-system, can be defined as follows:

```
reflect(current_load(N),(G,Env,Db),(NG,NEnv,NDb))
        :- true |
           length(G,X),
           add_env((N,X),Env,NEnv),
           (NG,NDb)=(G,Db).
```

We shift up to the meta-level and computes the length "X" of "G." This value "X" is contained in the environment list as a value of "N."

The "add_clause(CL)" predicate, which adds a given clause definition to the program of the object-system can be defined as follows:

```
reflect(add_clause(CL),(G,Env,Db),(NG,NEnv,NDb))
        :- true |
           add_db(CL,Db,NDb),
           (NG,NEnv)=(G,Env).
```

The next example is the "interpretive" predicate which execute a given goal "p" in an interpretive manner.

```
reflect(interpretive(P),(G,Env,Db),(NG,NEnv,NDb))
         :- true |
            exec([P],Env,Db,NEnv,_),
            (NG,NDb)=(G,Db).
```

Note that this interpretive execution can be executed in parallel with other execution. Therefore, it is possible to execute the specific goals in an interpretive manner and execute others directly. One possibility is modifying this "exec" to keep the debugging information. In such case, this predicate can be used as a "debugger." These kinds of modification can be performed in a quite straightforward manner.

3.3. Shift-down and shift-up

It is explained that, when a reflective predicate is called, the system is automatically shifted one-level up. When the execution of the reflective procedure finishes, the system is automatically shifted one-level down. In that sense, shift-up and shift-down are automatically carried out by using *reflective predicates* and we do not need to specify them explicitly.

However, we sometimes need to obtain the information about the representation, not the information itself. This typically happens when we want to *implement* a reflective system. For such purposes, we prepare two built-in predicates, i.e., "shift_down" and "shift_up."

"shift_down(Exp,Down_Exp)" transforms the given expression "Exp" to the one-level lower expression "Down_Exp." "shift_up(Exp,Up_Exp)" transforms the given expression "Exp" to the one-level higher expression "Up_Exp." For example, if we *shift-down* "p(a,[@1|@2],f(@2,b))" we obtain "p(a,[@!1|@!2],f(@!2,b))."

Though the use of "shift_up" and "shift_down" is not recommended for the casual user, we can use these predicates and obtain the information about the representation if we want. For example, "get_q" predicate which obtains the content of *execution goals* as its *representation* can be defined as follows:

```
reflect(get_q(Q),(G,Env,Db),(NG,NEnv,NDb))
        :- true |
           shift_down(G,Down_G),
           add_env((Q,Down_G),Env,NEnv),
           (NG,NDb)=(G,Db).
```

We need to shift down the *execution goals* because we want to get the content of *execution goals* as its *representation.*

On the other hand, "put_q" predicate, which replaces the contents of *goal queue* to the given expression "Q," can be defined as follows:

```
reflect(put_q(Q),(G,Env,Db),(NG,NEnv,NDb))
        :- true |
           shift_up(Q,NG),
           (NEnv,NDb)=(Env,Db).
```

Note that we cannot get the expected result, if we forget to *shift-up* "Q."

3.4. Meta-level databases

It is explained that reflective predicates are executed at the meta-level. The remaining question is how to build a meta-level computation system *dynamically* when the reflective predicate is executed.

Please see Figure 1 again. In general, a computation system of GHC consists of three elements, i.e., *goal queue*, *environment* and *database*. We have already explained how the meta-level *goal queue* is created, i.e., it only consists of the reflect goal. The meta-level *environment* only contains the binding information of this goal.

How about *database*? It must be different from the object-level database. If all of guard and body goals of the reflective predicate consist of system-defined goals, no problems occur. If it includes user-defined goals, they must be defined in the *database* at the meta-level.

How can we create meta-level database different from object-level database? We assume that *initially* only object-level database exists.

We have prepared "meta" and "global" predicate for such purpose. For example, if we would like to define a clause

```
G :- H | B.
```

at the the meta-level, we define it as

```
meta(G):- H | B.
```

at the object-level. When the meta-level is *dynamically* created by executing the reflective predicate, all *meta* definitions are searched from the object-level definition. Top level predicate "meta" is removed from those definitions and they are copied to the meta-level database. Similarly the meta-meta-level predicates can be defined as

```
meta(meta(G)):- H | B.
```

We also assume that *reflective* definitions are all copied to the meta-level, since they can be used recursively. The *global* predicate

```
global(G):- H | B.
```

is also prepared to define user-defined predicates which are common to all levels.

4. RGHC implementation

In implementing RGHC, there exists several possibilities. The most efficient implementation is re-designing the abstract machine code, which corresponds to Warren code, for RGHC. In this case, the abstract machine code must have the capability to handle system's *internal state* as *data*, or, conversely, to convert the given *data* into its *internal state*.

The other possibility is realizing RGHC system as an interpreter on top of an ordinary GHC system. Though we cannot expect too much for the execution efficiency in this case, this method has a merit that the implementation is relatively simple. We actually implemented RGHC using this method.

4.1. Description of RGHC

The top level description of RGHC can be expressed as follows:

```
r_ghc(Goal,Db,Out)
     :- true |
        transfer(Goal,GRep,1,Id,Env),
        exec([GRep],Env,Id,Db,NEnv,Res),
        make_result(Res,GRep,NEnv,Out).
```

Note that this code is exactly the same as that of "`m_ghc`" in Section 2.3. This means that we realize a *reflective system* as a object-level system in the meta-computation system.

However, "`exec`" must be enhanced to realize *reflection*. This can simply be performed by adding one program clause to the "`exec`" program in Section 2.3, as shown below.

```
exec([GRep|Rest],Env,Id,Db,NEnv,R)
        :- reflective(GRep,Db)|
           create_meta_db(Db,Meta_Db),
           shift_down((GRep,Rest,Env,Db),
                   (D_GRep,D_Rest,D_Env,D_Db)),
           exec([reflect(D_GRep,(D_Rest,D_Env,D_Db),(@1,@2,@3))],
                   [(@1,undf),(@2,undf),(@3,undf)],4,
                   Meta_Db,New_Meta_Env,_),
           deref_variable(@1,@2,@3),New_Meta_Env,
                   (D_Rest2,D_Env2,D_Db2)),
           shift_up((D_Rest2,D_Env2,D_Db2),
                    (N_Rest,N_Env,N_Db)),
           exec(N_Rest,N_Env,Id,N_Db,NEnv,R).
```

This program definition clause takes care of the creation of the reflective tower. "`create_meta_db`" creates the meta-database from the object-system database.

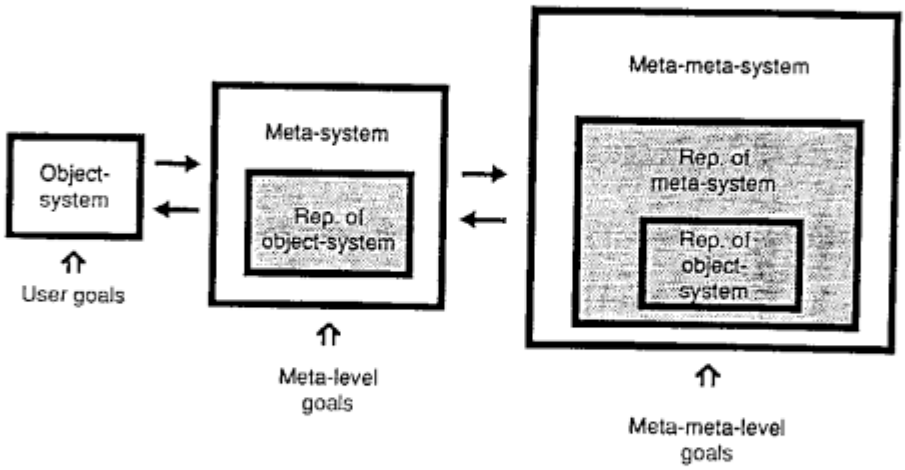

Figure 2: Constructing and collapsing a reflective tower

"(GRep,Rest,Env,Db)" is shifted down and the representation "(D_GRep,D_Rest,D_Env,D_Db)" is generated. Then "exec" starts the meta-level computation using these arguments.

When the meta-level execution finishes, "@1,@2,@3" must be instantiated. We derefer these variables, shift up this information and get "(N_Rest,N_Env,N_Db)" which denotes the new object-level information. Then we return to the object-level execution using this information.

Figure 2 shows how the reflective tower is constructed by calling reflective predicates and how it is collapsed by finishing up their execution.

4.2. RGHC implementation on PSI-II

We have already finished up the prototype implementation of RGHC using PSI-II [Nakashima 87] sequential Prolog machine. We used KL1 [ICOT 89] and ESP [Chikayama 84] as our implementation programming languages. KL1 is the extension of GHC, running on PSI-II hardware. Various extensions has been made to GHC, considering the actual program development on PSI-II. KL1 is used to describe the core part of the program. On the other hand, user interface and i/o part are written in ESP, the object-oriented dialect of Prolog. The total size of the program is 1530 lines, where KL1 part consists of 985 lines.

Though the RGHC system can *conceptually* be described as shown in Section 4.1, this implementation is very expensive since *list* is used for expressing *environment*. It sequentially searches the element and it leads to the inefficiency when the length of the *list* becomes long.

Therefore, we used *vector* data type instead of *list* for *internal* implementation. KL1 provides us with *vector* data type, where the *index search* is possible. Figure 3 shows the *vector representation* which corresponds to the *variable bindings* shown in Section 2.2.2. This *vector* implementation is initiated by [Fujita 90] and it has turned out to be very

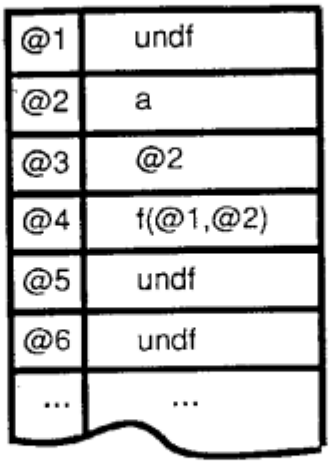

Figure 3: Vector representation of variable bindings

efficient.

In KL1, a vector can be created by executing "`new_vector(Vector,N)`" goal, where "`N`" is the input for specifying the vector size and "`Vector`" is the output keeping the reference pointer to the vector. The content of i-th element can be examined by "`vector_element(Vector,I,Element).`"

"`set_vector_element(Vector,I,OldElem,NewElem,NewVector)`" is used for setting value "`NewElem`" to the i-th element of vector "`Vector.`" We should note that the old value of i-th element is given as "`OldElem`" and the new reference pointer to the modified vector is given as "`NewVector.`" As you see, this "`set_vector_element`" predicate is defined in a quite declarative manner. However, at the language implementation level, KL1 system is managing the reference count for the vector and destructive assignment is performed when no other goal is pointing the vector.

Our policy for RGHC implementation is as follows: We use *vector* instead of *list* for *internal* implementation. However, we still continue to use *list* structures for the *external* representation. Therefore, *reflective* programming examples shown in the previous sections are still effective. On the other hand, *exec* must be modified slightly to handle *vectors*, though we omit the implementation details because of the space limitation.

Note that the use of the internal database, such as seen in DEC-10 Prolog [Bowen 83], may also be effective for the more efficient implementation. If we use the internal database, fast program look-up becomes possible using the key. Though we have not used the internal database in representing programs, these kinds of considerations become more important, especially when the size of the object-level program becomes larger.

5. Program execution example

We show the program execution example in Figures 4 and 5. When we start up the RGHC system, it automatically opens "`I/O WINDOW Level 1,`" where "`level 1`" means the *object level*. We can type in the initial query from this window. In this case,

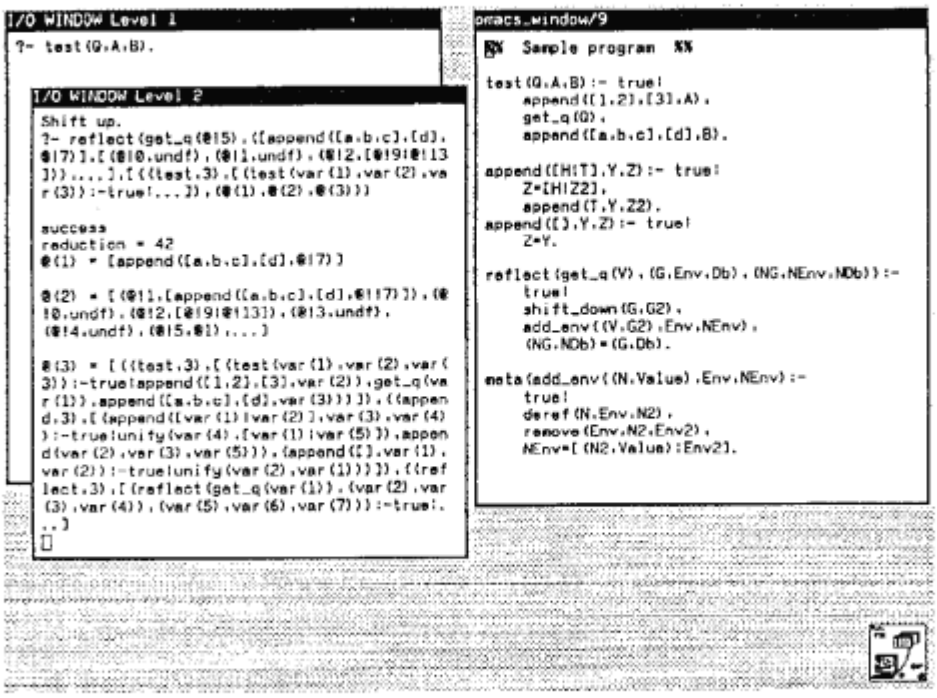

Figure 4: Execution example of RGHC (1)

we typed in "`<- test(Q,A,B).`" We showed the definition of "`test`" predicate in the "`pmacs_window`," located to the right side of Figure 4, for the reference.

As you see in this program, "`get_q(Q)`" is defined as a reflective goal. When this "`get_q(Q)`" goal is executed, the meta-level computation system is *dynamically* created and "`I/O WINDOW Level 2`" is opened.

Figure 4 shows the instant when the meta-level computation has just finished up. "`reflect(get_q(...)...)`" in "`I/O WINDOW Level 2`" shows the reflective goal to be executed at the meta-level. This window shows that the execution of this goal has been finished successfully by 42 steps. The bindings of variables allocated at the meta-level are also shown. As you see, variables at the meta-level is shown by `@(1)`,`@(2)`,`@(3)` and the object-level variables are shown by `@!5`,`@!7`. These representations are slightly different from those explained in Section 2.2.2, since it includes () at the meta-level. The differences mainly come from the regulations of our GHC system and are not essential.

When the meta-level computation terminates, "`I/O WINDOW Level 2`" also disappears. Figure 5 shows the instance when the whole computation terminates. The final computation result is shown in "`I/O WINDOW Level 1.`" It shows that the execution has been terminated successfully by 57 steps and the bindings of variables are also shown.

6. Related works and conclusion

As mentioned before, *meta* and *reflection* are attracting wide spread attention. Though the concept of *computational reflection* goes back to [Weyhrauch 80] and [Smith 84], this concept are becoming popular especially in object-oriented language community [Maes 88] [Ibrahim 90].

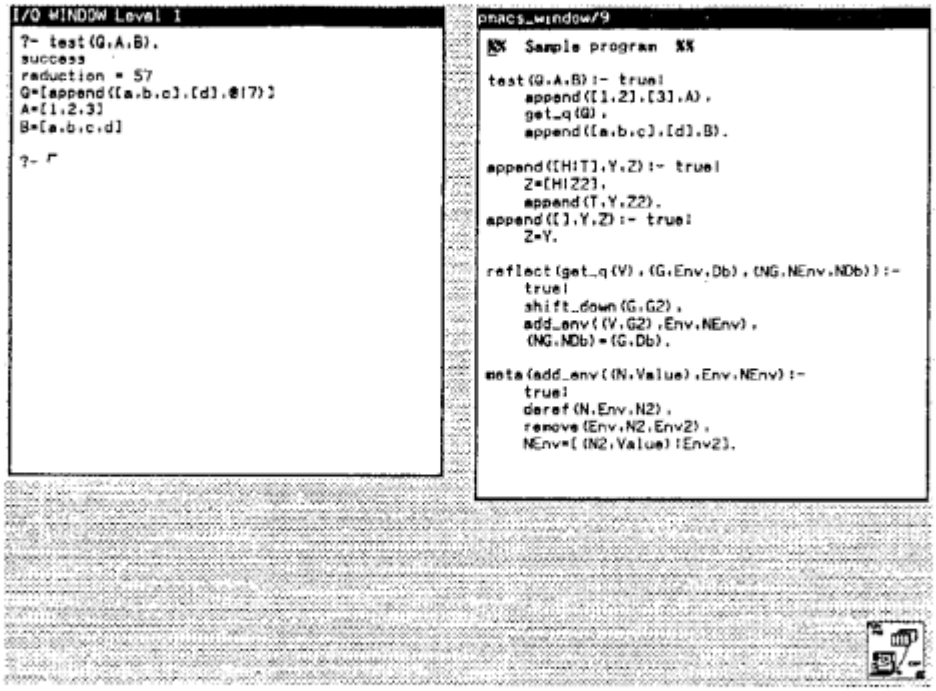

Figure 5: Execution example of RGHC (2)

In logic programming field, [Bowen 82] provides us the starting point for *meta-programming* research. There exists several related research, such as [Porto 84], [Eshghi 86] and [Lloyd 88b]. Two workshops, i.e. Meta 88 and Meta 90, have also been held in relate to *meta-programming* in logic programming [Lloyd 88a] [Bruynooghe 90]. However, it seems that their interests mainly exist in the reconstruction of Prolog which has cleaner semantics.

Regarding to *reflection* in logic programming, there exists few research works so far. The exceptions are Reflective Prolog [Costantini 89] and R-Prolog* [Sugano 90]. It seems that the interest of Reflective Prolog mainly exists in controlling the program execution by re-defining *solve* predicate at the meta-level. In spite of his claim on *computational reflection*, his system has only a very limited expressive power,

On the other hand, R-Prolog* [Sugano 90] assumes the similar kind of reflective predicated as proposed in this paper. However, his interest exists mainly in semantics. Not much consideration has done on the actual implementation and for the execution efficiency.

The features of our RGHC system can be summarized as follows:

1. Simple formulation of reflection in GHC. Especially, we have formulated reflection without using *quote*. This is the critical difference from Lloyd's approach.

2. Ground representation of variables. In our system, variables are expressed as *special ground terms*. This representation essentially corresponds to quoted form in other systems.

3. Complete handling of database. In our system, we can define the meta-level database, which is completely different from the object-level by "`meta`" predicate. It is also

possible to define the arbitrary layers of databases.

4. Dynamic constructing and collapsing of a reflective tower. In our system, a new level is generated when a reflective predicate is called. When finished, that level is collapsed and the system automatically returns to its original level.

Though we have not mentioned about the declarative semantics of RGHC, we imagine that the extended Herbrant base with i/o pair [Sugano 90] fits to the GHC semantics. This direction matches to the effort to define GHC semantics in a denotational manner [Murakami 90].

Our final goal exists in building a sophisticated distributed operating system on top of the distributed inference machine such as PIM [Uchida 88]. Some trials for describing such systems can be seen in [Tanaka 88b] [Tanaka 90].

7. Acknowledgments

This research has been carried out as a part of the Fifth Generation Computer Project of Japan. The author would like to express thanks to Yukiko Ohta, Hiroyasu Sugano and Youji Kohda their useful discussions.

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, L. Byrd, F.C.N. Pereira, L.M. Pereira and D.H.D. Warren: DECsystem-10 Prolog User's Manual, University of Edinburgh, August 1983.

[Bruynooghe 90] M. Bruynooghe eds.: Proceedings of the Second Workshop on Meta-Programming in Logic, Leuven, Belgium, April, 1990.

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

[Costantini 89] S. Costantini and G.A. Lanzarone: A Metalogic Programming Language, Logic Programming, Proceedings of the Sixth International Conference, pp.218-233, The MIT Press.

[Eshghi 86] K. Eshghi: Meta-Language in Logic Programming, Ph.D. Thesis, Department of Computing, Imperial College, July 1986.

[Fujita 90] H. Fujita, M. Koshimura and R. Hasegawa: Meta-programming Library on KL1, *internal report*, ICOT, June 1990 (*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.

[Ibrahim 90] M.H. Ibrahim eds: ECOOP/OOPSLA Workshop, Reflection and Metalevel Architectures in Object-Oriented Programming, Ottawa, Canada, October 1990.

[ICOT 89] ICOT: PIMOS Manual, version 1, ICOT, July 1989 (*in Japanese*).

[Lloyd 88a] J. W. Lloyd eds.: Proceedings of the Workshop on Meta Programming in Logic Programming, University of Bristol, June 1988.

[Lloyd 88b] J. W. Lloyd: Directions for Meta-Programming, Proceedings of of the International Conference on Fifth Generation Computer Systems 1988, pp.609-617, ICOT, November 1988.

[Maes 86] P. Maes: Reflection in an Object-Oriented Language, Preprints of the Workshop on Metalevel Architectures and Reflection, Alghero-Sardinia, October 1986.

[Maes 88] P. Maes and D. Nardi eds: Meta-Level Architectures and Reflection, North-Holland, 1988.

[Murakami 90] M. Murakami: A Declarative Semantics of Flat Guarded Horn Clause for Programs with Perpetual Processes, Theoretical Computer Science, Vol. 75, No. 1/2, pp.67-83, 1990.

[Nakashima 87] 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.

[Porto 84] A. Porto: Two-level Prolog, Proceedings of of the International Conference on Fifth Generation Computer Systems 1984, pp.356-360, ICOT, November 1984.

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

[Smith 84] B.C. Smith: Reflection and Semantics in Lisp, Proceedings. of in Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, pp.23-35, ACM, January 1984.

[Sugano 90] H. Sugano: Meta and Reflective Computation in Logic Programs and its Semantics, Proceedings of the Second Workshop on Meta-Programming in Logic, Leuven, Belgium, pp.19-34, April, 1990.

[Tanaka 86] J. Tanaka, K. Ueda, T. Miyazaki A. Takeuchi, Y. Matsumoto and K. Furukawa: Guarded Horn Clauses and Experiences with Parallel Logic Programming, Proceedings of FJCC '86, ACM, Dallas, Texas, pp.948-954, November 1986.

[Tanaka 88a] 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.

[Tanaka 88b] J. Tanaka: Meta-interpreters and Reflective Operations in GHC, Proceedings of of the International Conference on Fifth Generation Computer Systems 1988, pp.774-783, ICOT, November 1988.

[Tanaka 90] J. Tanaka, Y. Ohta and F. Matono: Overview of Experimental Reflective Programming System ExReps, Fujitsu Scientific & Technical Journal, Vol.26, No.1, pp.86-97, April 1990.

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

[Uchida 88] 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.

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