PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Constructing and Collapsing a Reflective Tower in Reflective Guarded Horn Clauses

Jiro Tanaka† and Fumio Matono‡

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

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

Abstract

The meta-level representation of *Guarded Horn Clauses* (GHC) is considered and a GHC meta-computation system is constructed by enhancing the simple GHC meta-program. Then the *Reflective Guarded Horn Clauses* (RGHC) system is described, where a *reflective tower* can be constructed and collapsed in a dynamic manner, using *reflective predicates*. The implementation of RGHC is shown. Finally a simple execution example is also shown. 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 development of the programming language is 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. Though the concept of *computational reflection* goes back to [Weyhrauch 80, Smith 84], this concept is becoming popular especially in the object-oriented language community [Maes 88].

In this paper, we assume the parallel logic programming language *GHC* [Ueda 85, Tanaka 86] as our underlying language. The reasons for picking this language are in its structural simplicity, semantical clearness and applicability to the system programming.

We have already proposed *reflection* mechanism and shown several application examples [Tanaka 88, Tanaka 90, Tanaka 91]. However, *reflection* has been introduced in an *ad hoc* manner. It lacks the generality seen in 3-Lisp [Smith 84]. We would like to propose *Reflective Guarded Horn Clauses* (RGHC), which has the expressive power comparable 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 a 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 latter computational system is called the *object-system*. The program of *meta-system* is called *meta-program*.

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 a given goal in an interpretive manner. We can see two levels, i.e., the *meta-level*, where the top level execution is performed, and the *object-level*, where 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 variables at the meta-level and those at the object-level. The object-level variables cannot be manipulated at the meta-level.
- The predicate "reduce(P,Q)" finds *potentially unifiable clauses* for the given argument "P." The object-level program must also be defined as a program. Therefore, the object-level program cannot be manipulated without using *assert* or *retract*.
- This program only simulate the top level execution of the program. The more detailed executing information such as *current continuation*, *environment* or *execution result* is not *explicit* in this interpreter.

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

We would like to have a real meta-computation system which does not have the disadvantages described in the previous section. As a first step, we consider the meta-level representation of the object-system.

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. Though *quote* is usually used to separate the *data* from the *program*, there exists a clear separation between predicates and functors in logic programming languages. 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 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, compared 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 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* the pointers when the value of a variable is needed.

2.2.3. Terms and object-level programs

Keeping the 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 are expressed as a *ground term* at the meta-level, where all variables are replaced by "var(number)" notation. Note that "var(number)" is just a *ground term* and not the *special ground term*.

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. Note that, this representation also works at the meta-meta-level, since "var(number)" is just a *ground term*.

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 the *goal queue* and the *environment* in its arguments for expressing *continuation* and *variable bindings*.

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

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

For given object-level goal "Goal" and given object-level program "Db," "m_ghc" outputs 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 contains the environment of this goal representation.

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

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

Note that *environment* is expressed as a pair of a number and a list. The first element of the pair shows how many variables are allocated in the environment. In this case, two numbers have already been allocated. The second element of the pair shows the variable bindings.

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

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

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

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]. A reflective system can be defined as a computational system which takes itself as its problem domain.

In 3-Lisp, an infinite tower of meta is conceptually assumed. A program is not executed directly. Instead, it is assumed to be executed on the bottom of the infinite tower of meta. A meta-system executes the object-system in an interpretive manner. Similarly, the meta-meta-system executes the meta-system in an interpretive manner. Conceptually, the infinite tower of meta all

moves in a synchronous manner. A reflective function can be defined as a mechanism which shift the control one level up.

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

How should such reflective capability be implemented? Apparently, implementing an infinite tower of meta directly is not possible.

There exist two approaches to realizing such reflective system. One is utilizing a pre-exist layer of meta-systems. We can modify the meta-program and add the means of communication between the levels, namely, we prepare a set of built-in predicates which can catch or replace the current state of the object 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 88, Tanaka 90, Tanaka 91], 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 merit in that the implementation is relatively straightforward, we should note that this approach is not an 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. The *reflective* function may also be 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. When the meta-level computation terminates, the control automatically returns to the one-level-lower computation system.

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 of RGHC are shown in the following sections.

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.

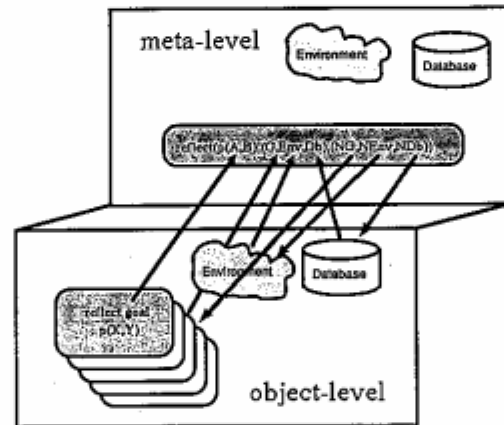

Figure 1: Execution of the reflective predicate

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.
```

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 represented as "`(G,Env,Db)`," where "`G`" represents the *remaining goals* which should be executed at the object-level, "`Env`" represents the *environments* and "`Db`" represents the object-level program. Note that they are the representations of the state and we can freely access and manipulate these arguments. "`(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 `(NG,NEnv,NDb)` are instantiated while executing meta-level goals. When the execution of this reflective goal is finished, 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 whether the variable is bound or not and the result is added to the environment list as a value of "R."

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." This kind of modification can be performed in a quite straightforward manner.

3.3. Shift-down and shift-up

It has been 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 representation "Exp" to the one-level lower representation "Down_Exp." "shift_up(Exp,Up_Exp)" transforms the given representation "Exp" to the one-level higher representation "Up_Exp." We should note that "shift_down" and "shift_up" just converts the representations. Therefore, they do not need to use environment for the conversion. 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.

"meta" and "global" predicates are prepared 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.
```

It is also assumed 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,Env),
        exec([GRep],Env,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))],
        (3,[(@1,undf),(@2,undf),(@3,undf)])
        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. "(GRep,Rest,Env,Db)" is shifted down and the representation "(D_GRep,D_Rest,D_Env,D_Db)" is generated.

Then "exec" at line 6 starts the meta-level computation using these arguments. This "exec" essentially responsible for the creation of the meta-level computation system. The *trick* of the program is in using the same "exec" at the meta-level computation. Note that variables at the meta-level computation are assigned differently from the object-level computation.

Therefore, in our implementation, the meta-level computation is executed at the same speed as its object-level computation.

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

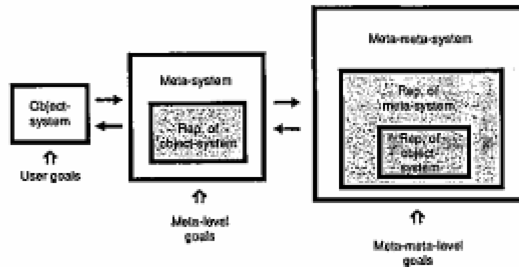

Figure 2: Constructing and collapsing a reflective tower

@1	undf
@2	a
@3	@2
@4	f(@1,@2)
@5	undf
@6	undf
...	...

Figure 3: Vector representation of variable bindings

The prototype implementation of RGHC has already been finished up 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, the *vector* data type is used 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 [Koshimura 90] and it has turned out to be very 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`." It can be seen that this "`set_vector_element`" predicate is defined in a quite declarative manner. However, at the language implementation level, the KL1 system is managing the reference count for the vector and destructive assignment is performed when no other goals are pointing the vector.

Our policy for RGHC implementation is as follows: A *vector* is used instead of *list* for *internal* implementation. However, we still continue to use *list* structures for the *external* representation. Therefore, the *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

The snapshots of a program execution example are shown in Figures 4 and 5. When the RGHC system is started, "`I/O WINDOW Level 1`" is automatically opened, where "`level 1`" means the *object level*. The initial query can be typed in from this window. In this example, we have typed in "`<- test(Q,A,B).`" The definition of "`test`" predicate is shown in the "`pmacs_window`," located to the right side of Figure 4, for the reference.

As seen 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. Variables at the meta-level are 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.

```
I/O WINDOW Level 1
?- test(Q.A.B).
```

```
I/O WINDOW Level 2
Shift up.
?- reflect(get_q(@!5).([append([a.b.c].[d].
@!7)].[(@!0.undf).(@!1.undf).(@!2.[@!9!@!13
]))....].[((test.3).[(test(var(1).var(2).va
r(3)):-true!...]).(@(1).@(2).@(3)))

success
reduction = 42
@(1) = [append([a.b.c].[d].@!7)]

@(2) = [(@!1.[append([a.b.c].[d].@!!7)]).(@
!0.undf).(@!2.[@!9!@!13]).(@!3.undf).
(@!4.undf).(@!5.@1)....]

@(3) = [((test.3).[(test(var(1).var(2).var(
3)):-true!append([1.2].[3].var(2)).get_q(va
r(1)).append([a.b.c].[d].var(3)))]).((appen
d.3).[(append([var(1)!var(2)].var(3).var(4)
):-true!unify(var(4).[var(1)!var(5)]).appen
d(var(2).var(3).var(5))).(append([].var(1).
var(2)):-true!unify(var(2).var(1)))]).((ref
lect.3).[(reflect(get_q(var(1)).(var(2).var
(3).var(4)).(var(5).var(6).var(7))):-true!.
..]
```

```
pmacs_window/9
%%  Sample program  %%

test(Q.A.B):- true!
     append([1.2].[3].A),
     get_q(Q),
     append([a.b.c].[d].B).

append([H!T].Y.Z):- true!
     Z=[H!Z2].
     append(T.Y.Z2).
append([].Y.Z):- true!
     Z=Y.

reflect(get_q(V).(G.Env.Db).(NG.NEnv.NDb)):-
     true!
     shift_down(G.G2),
     add_env((V.G2).Env.NEnv),
     (NG.NDb)=(G.Db).
```

Figure 4: Execution example of RGHC (1)

```
I/O WINDOW Level 1
?- test(Q.A.B).
success
reduction = 57
Q=[append([a.b.c].[d].@!7)]
A=[1.2.3]
B=[a.b.c.d]

?-
```

```
pmacs_window/9
%%  Sample program  %%

test(Q.A.B):- true!
     append([1.2].[3].A),
     get_q(Q),
     append([a.b.c].[d].B).

append([H!T].Y.Z):- true!
     Z=[H!Z2].
     append(T.Y.Z2).
append([].Y.Z):- true!
     Z=Y.

reflect(get_q(V).(G.Env.Db).(NG.NEnv.NDb)):-
     true!
     shift_down(G.G2),
     add_env((V.G2).Env.NEnv),
     (NG.NDb)=(G.Db).
```

Figure 5: Execution example of RGHC (2)

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 variable bindings at that time are also shown.

6. Related works and conclusion

In logic programming field, [Bowen 82] provides us the starting point for *meta-programming* research. There exists related research, such as [Porto 84, Eshghi 86]. Also some related research was carried out in parallel logic programming field [Shapiro 84, Clark 84, Hirsch 86, Safra 86].

Recently, two workshops on *meta-programming* in logic programming, i.e., Meta 88 and Meta 90, have been held [Lloyd 88a, Bruynooghe 90]. It seems that the attention has been paid to the theoretical foundation of meta-programming. Several considerations have been done for the meta-level *representation* of the object-system [Lloyd 88b, Lim 90]. Our representation, described in Section 2, is very close to Lim's approach.

Though we described the GHC meta-computation system first in this paper, it seems that our object-system representation is quite straightforward one and that most people agree with our representation scheme.

However, regarding to *reflection* in logic programming, there exist few research works so far. The exceptions are [Costantini 89, Sugano 90, Lamma 91]. They all worked for *reflection* in Prolog.

In [Costantini 89, Lamma 91] the interests mainly lie in controlling the program execution dynamically by redefining the *solve* predicate at the meta-level. In spite of his claim on *computational reflection*, their systems have only a very limited expressive power,

On the other hand, [Sugano 90] assumes the similar kind of *reflective predicate definition* as proposed in this paper. However, his interest is mainly in semantics. Not much consideration has done for the implementation, the execution efficiency and the application.

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

1. Reflection mechanism by *reflective predicate definition*. The user can freely define *reflective predicates* which invoke *reflection* when called. We can handle *current continuation*, *environment* and *program* at the meta-level. This mechanism is the GHC version of *reflective function* in 3-Lisp.

2. 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. By calling *reflective predicate* recursively, the arbitrary level of *meta* can be created dynamically.

3. Creation of the arbitrary layers of databases. We can define the meta-level database, which is different from the object-level by "meta" predicate. It is also possible to define the arbitrary layers of databases by using "reflect," "meta" and "global" predicates.

It seems that the unique feature of RGHC is the implementation simplicity of *reflection*. As shown in Section 4, the trick is in using the same "exec" at the meta-level computation. Therefore, the meta-level computation can be executed at the same speed as its object-level computation.

This elegance of the implementation mainly comes from the simplicity of the language. This seems to be its most critical difference from the implementation of *reflection* in Lisp or the one in object-oriented languages. Though we did not mentioned the semantics of RGHC, we should note that [Sugano 90] worked out for defining the semantics of his R-Prolog* by the extended Herbrant base with i/o pair.

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 88, Tanaka 90, Tanaka 91].

7. Acknowledgments

This research has been carried out as a part of the Fifth Generation Computer Project of Japan. The authors would like to express thanks to Yukiko Ohta, Simon Martin, 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.

[Clark 84] 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

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

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

[Koshimura 90] M. Koshimura, H. Fujita and R. Hasegawa: Meta-programming in KL1, SIGSYM Meeting, No.57-2, IPSJ, November 1990 (*in Japanese*).

[Lamma 91] E. Lamma, P. Mello and A. Natali: Reflection Mechanisms for Combining Prolog Databases, *Software Practice and Experience*, Vol.21, No.6, pp.602-624, June 1991.

[Lim 90] P. Lim and P.J. Stucky: Meta Programming as Constraint Programming, *Logic Programming, Proceedings of the 1990 North American Conference*, The MIT Press pp.416-430, 1990.

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

[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.

[Shapiro 84] 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.

[Smith 84] B.C. Smith: Reflection and Semantics in Lisp, 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 88] 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.

[Tanaka 91] J. Tanaka: An Experimental Reflective Programming System Written in GHC, *Journal of Information Processing*, Vol.14, No.1, pp.74-84, 1991.

[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.