TR-534

A Proposal for Reflective GHC

by

J. Tanaka & F. Matono

February, 1990

A Proposal for Reflective GHC

Jiro Tanaka† and Fumio Matono‡

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

‡FUJITSU SOCIAL SCIENCE LABORATORY LIMITED,
1-6-4 Osaki, Shinagawa-ku, Tokyo 141, JAPAN

Abstract

After defining *meta* and *reflection* as basic terminology, we define *meta-computation system* in logic programming languages. Especially, we consider the correspondence between the representation at the meta-level and the object at the object-level. Based on these considerations, we propose a *reflective system* in parallel logic language GHC. 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 and collapsed in a dynamic manner.

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 concept which is called as *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] and shown several examples, mainly from application aspects [Tanaka 88a] [Tanaka 88b] [Tanaka 90]. However, the reflection mechanism has been introduced as special built-in predicates and they lacked the generality as seen in 3-Lisp [Smith 84]. Therefore, we would like to propose *Reflective GHC*, 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 define *meta* and *reflection* first. In section 3, we examine the disadvantages of the simple 4-line meta-programs. In Section 4, we propose a real meta-system which does not have the disadvantages of the 4-line meta-programs. In Section 5, we propose *Reflective GHC* where the reflective tower can be constructed dynamically. Related works and conclusions are described in Section 6.

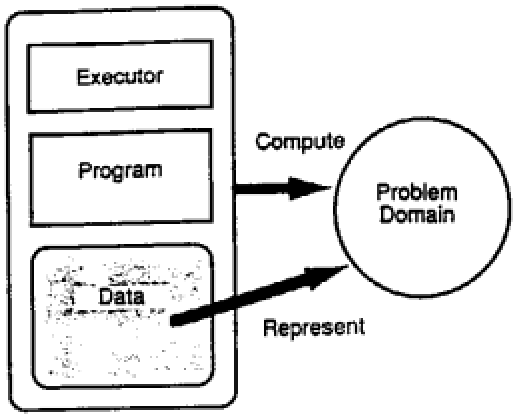

Figure 1: A computational system

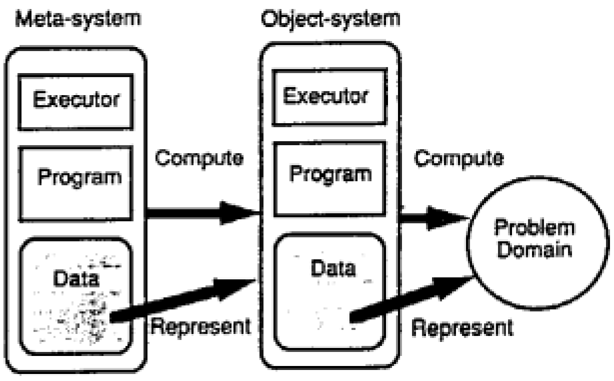

Figure 2: A meta-system

2. Meta and reflection

We would like to model a *computational system* first. We follow the description of Maes [Maes 86] in this section.

Though there exists various way to express a computation system, we assume that it consists of an *executor*, a *program* and *data*, as shown in Figure 1.

Here, the *executor* corresponds to the CPU of the ordinary computer. We model certain features of the problem domain into the *program* and *data* of the computational system. Generally speaking, the algorithm of the problem solving is modeled into the *program* and the structures of the problem domain is expressed by the *data*. The *executor* tries to compute a certain feature of the problem domain using the program and the data.

2.1. Meta-system

A meta-system can be defined as a computational system whose problem domain is another computational system, as shown in Figure 2.

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

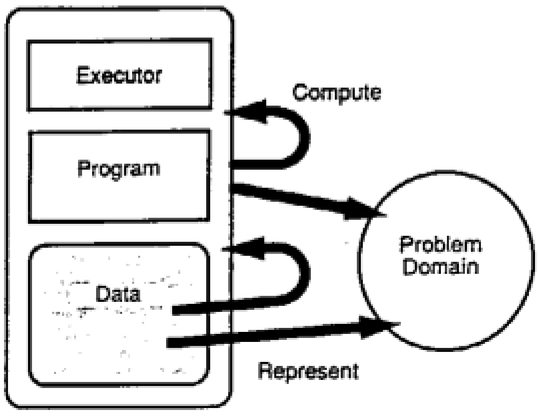

Figure 3: A reflective-system

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*.

We ordinary realizes the *meta-system* by representing the algorithms of problem solving using meta-interpreters. However, in general, we only need the capability to represent a computational system in some way. Historically, FOL [Weyhrauch 80] is the well-known example of a meta-system which does not rely on the meta-interpreter. On the other hand, EVAL in Lisp 1.5 [McCarthy 65] is a well-known example of the meta-interpreter.

2.2. Reflective-system

Reflection is the capability to feel or modify the current state of the system dynamically [Smith 84] [Maes 86]. A *reflective system* is shown in Figure 3.

As shown in the figure, the program and data of the computational system point to themselves, besides pointing to the specific problem domain. We can look the reflective system as a specific meta-system where the meta-system and the object-system are overlapping each other. In the meta-system, we could observe and manipulate the behavior of the object-system. However, in the reflective system, we can observe and manipulate ourselves.

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.

We usually utilize a meta-system to realize these reflective capabilities. Since object-system is expressed as data in the meta-system, we simply add the means of communication between the meta-system and the object-system. We prepare the way which takes out the current computation state from the object-system. We also need the mean which replaces the current state of the object-system to the modified state.

There exist two approaches realizing such frameworks. One is just preparing built-in functions which can catch or replace the current state of the system. We actually adopted this approach in implementing *reflection* in [Tanaka 88b]. This approach has a merit that the implementation is relatively straightforward. However, at the same time, it has a disadvantage that this may easily cause the confusion of levels because meta-information is processed at the object-level. This approach is not the accurate implementation of *reflection* since meta-level state may change while processing meta-level information at the object-level.

The other way is to create meta-system dynamically when needed. If a *reflective* function 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 [Smith 84]. 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*, though the implementation becomes more complicated.

2.3. Reflective tower

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 is also possible to imagine the infinite tower of meta. We can also think that the infinite tower of *meta* exists from the very beginning and they move together in a synchronized way.

3. Description of the meta-system in logic programming languages

As shown in Figure 4, the logical computational system consists of an *executor*, a *database*, *execution goals* and *variable bindings*. We easily notice that this is the specialization of Figure 1.

Comparing these two, we note that the *data* in Figure 1 corresponds to the *execution goals* and *variable bindings* in Figure 4. The *execution goals* contain the current goal sequence to be computed. It initially contains the query to be processed. The binding information of the variables in the *execution goals* is kept in the *variable bindings*.

3.1. Simple meta-programs

In Prolog world, the following 4-line program has been known as *Prolog in Prolog* or *vanilla* interpreter [Bowen 83].

```
exec(true):-!.
exec((P,Q)):-!,exec(P),exec(Q).
exec(P):-my_clause((P:-Q)),exec(Q).
exec(P):-sys(P),!,P.
```

Here, the program of the object system is defined in "my_clause" predicate. Initial goal "p" is executed by the form "exec(P)." This "exec" program works as follows:

1. If the goal is "true," the execution of the goal succeeds.

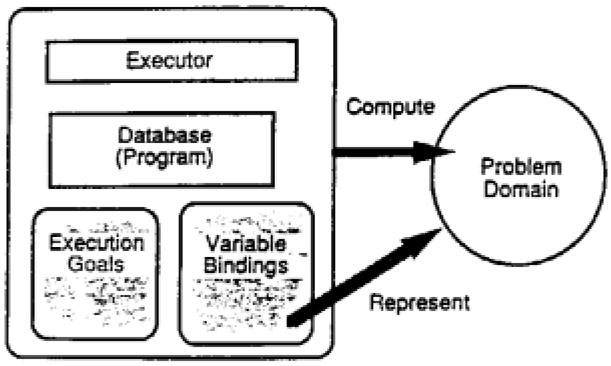

Figure 4: A logical computation system

2. If it is a sequence, it is decomposed and executed sequentially.
3. In the case of a user-defined goal, the predicate "`my_clause`" finds the definition of the given goal and the goal is decomposed to the body part of its definition.
4. 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 [1]:

```
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.
```

We omit the detailed explanation of this program since its meaning is self-explanatory. The correspondence between object-level and meta-level can be summarized as follows:

Object-level	Meta-level
constant	constant
variable	variable
function symbol	function symbol
predicate symbol	function symbol
defintion clause	special definition clause

3.2. Disadvantages of simple meta-programs

However, this "Prolog in Prolog" or "GHC in GHC" is insufficient as a meta-system because of the following reasons.

[1]Though *Prolog in Prolog* and *GHC in GHC* look very similar, one difference is that system-defined predicates have more stronger reason for their existence in GHC. In Prolog, system-defined predicates are assumed mainly for efficiency. On the other hand, they are first-class citizens in GHC.

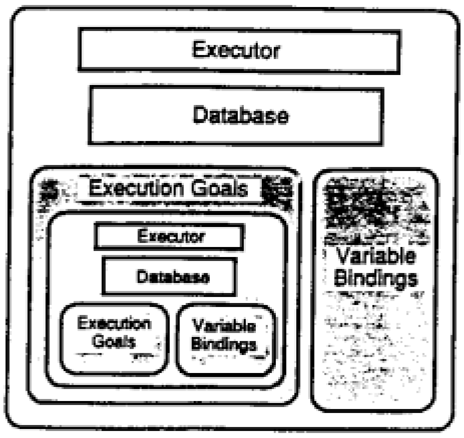

Figure 5: A logical meta-computation system

1. This program only simulate the top level execution of the program and we cannot obtain the more detailed executing information such as unification or guard execution from the program.

2. 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 bounded, nor can we check whether the given variable is identical to the other one.

3. At the meta-level, a function symbol corresponds to the predicate of the object-level. In the case of Prolog, a special definition clause "`my_clause`" corresponds to the object-level definition clause. Therefore, we can distinguish object-level definition clause from the meta-level one. However, we cannot modify these definitions at the meta-level without using *assert* or *retract*. Though this is not clear in the case of GHC, the situation is exactly the same if we look at the definition of "`reduce`."

4. Constructing a real meta-computation system

Therefore, we would like to propose the real meta-computation system which does not have the disadvantages described in the previous section. Our proposal for such meta-computation system is shown in Figure 5.

Here, the representations of the object-system, i.e., an *executor*, a *database*, *execution goals* and *variable bindings*, are all contained in the *execution goals* of the meta-computation system.

We show how the elements of a *object-system* should be expressed at the *meta-system* in the following subsections.

4.1. Constants, function symbols and predicate symbols

Constants and function symbols of the object-system are expressed by the same constants or function symbols. Predicate symbols of the object-system are expressed as function 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, R-Prolog [Sugano 89] or Gödel [Lloyd 88] adopts 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 functions. Logic programming languages do not have a notion of *evaluation*. They simply find out the binding of variables contained in the initial query. Our claim is that there is little *practical* merit in using *quote* in logic programming languages.

4.2. Variables

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 a object-level variable. This *special ground term* has a one-to-one correspondence to the object-level variable and we can distinguish it from the ordinary *ground term* at the meta-level. Since the tower of meta can be created to the arbitrary levels, the *special ground term* also needs to contain information about its level.

Variables are usually expressed by the identifiers which starts with a capital letter. We assume that a variable is expressed as "@number" at the meta-level, in which its own number is assigned for each variable at the object level. We assume that it is expressed as "!@number" at the meta-meta-level. Similarly, it is expressed as "!!@number," "!!!@number," "!!!!@number," and so on, as the level increases.

4.3. Terms

Keeping consistency with the notations of Sections 4.1 and 4.2, we denote object-level terms by corresponding meta-level *special ground terms*. In those corresponding *special ground terms*, every constant and function symbols are not changed. However, 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.

4.4. Variable bindings

Variable bindings at the object-level are represented as a list of address-value pairs at the meta-level. Note that this list is expressed as a *special ground term*. 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 *dereference* pointers when the value of a pair is required.

4.5. Definition clauses

The program of object-level, i.e., the collections of *asserted* clause definitions are also expressed as a *special ground term*, i.e., a list of *ground* representations of clause definitions, at the meta-level.

For example, the following "**append**" program

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

is expressed as follows.

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

4.6. Correspondences between the meta-level and the object-level

The above mentioned correspondences between the meta-level and the object-level can be summarized as follows:

Object-level	Meta-level
constant	constant
function symbol	function symbol
predicate symbol	function symbol
variable	special ground term
term	special ground term
variable bindings	special ground term
defintion clause	special ground term

5. Proposal for Reflective GHC

Based on the considerations described in the previous section, we propose Reflective GHC. Reflective GHC is the *reflective extension* of GHC and can be defined as a superset of GHC. We show the language features and the outline of the implementation in the following subsections.

5.1. Reflective predicates

Reflective predicates are user-defined predicates which invoke *reflection* when called. Similar to 3-Lisp, we can easily access to the internal state of the computation system and obtain them to the object-level by using *reflective predicates.* Or we can modify the internal state of the computation system from the object-level. We can define reflective predicates and use wherever we want, in the user program or in the initial query.

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

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

We should note that extra arguments, i.e., (G,Env,Db) and (NG,NEnv,NDb,R), are added to this definition. Here, (G,Env,Db) expresses the computation state of the object-level, where G expresses the *execution goals*, Env expresses the *variable bindings* and Db expresses the *database.* (NG,NEnv,NDb,R) denotes the new state to which the system should return when the execution of the reflective procedure finishes. NG expresses the new *execution goals*, NEnv expresses the new *variable bindings* NDb expresses the new *database* and R expresses the computation result of the reflective procedure.

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. At this level, we can handle "`p(X,Y)`" where X and Y are the meta-level representation of the arguments, and (G,Env,Db), which is the representation of the object-system. When we finished executing this reflective goal, we automatically shift one level down and (NG,NEnv,NDb,R) changes to the new object-level state.

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

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

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

We simply shift up one level and checks the value of the given argument from the environment list. If it is unbounded the execution of this predicate succeeds. If it is not, the execution of this predicate fails.

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

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

We shift up one level and computes the length X of G. This value X is contained in the environment list as a value of N, and the execution finishes successfully.

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

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

We dereference CL in the current environment and get NCL [2]. Then we simply add this NCL to Db.

5.2. Shift-up and shift-down

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 and we do not need to specify them explicitly.

By using *reflective predicates*, we could easily access to the internal state of the computation system and obtain them to the object-level, as shown in the previous subsection. However, the obtained information was always the ground information.

We sometimes need to obtain the meta-level information which includes the representation of variables. Or we sometimes would like to replace the meta-level information to a

[2]Since variables in NCL are used as formal arguments, it is also possible to perform renumbering here.

certain object-level information. For such purposes, we prepared two kinds of predicates, i.e., "shift_up" and "shift_down."

"shift_up(Exp,Up_Exp)" transforms the given expression "Exp" to the one-level higher expression "Up_Exp." "shift_down(Exp,Down_Exp)" transforms the given expression "Exp" to the one-level lower expression "Down_Exp."

For example, "get_q" predicate which obtains the contents of *execution goals* as its *meta-level notation* can be defined as follows:

```
reflect(get_q(Q),(G,Env,Db),(NG,NEnv,NDb,R))
        :- true |
           shift_up(G,Up_G),
           NEnv=[(Q,Up_G)|Env],
           (NG,NDb,R)=(G,Db,success).
```

We need to shift up the *execution goals* in this definition because we want to get the contents of *execution goals* as its *meta-level notation*.

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

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

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

5.3. Meta definitions

Reflective predicates are executed at the meta-level. If all of guard and body goals consist of system-defined functions, we have no problem. If it includes user-defined goals, they must be defined in the *database* of the meta-level computation system. Therefore, we prepare "meta" predicate to define the given definition clause to the meta-level database.

For example, if we would like to define a clause

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

to the meta-database, we can define it as

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

using "meta" predicate.

Though *reflective* or *meta* predicates are executed at the meta-level, these *reflective* or *meta* definitions are kept at the object-level database initially. It is explained that meta-level is *dynamically* created when a reflective predicate is called. At that time, *reflective* or *meta* definitions are transformed to the appropriate form and copied to the meta-level database.

5.4. Enhanced meta-interpreter and interpretive execution

The simple 4-line meta-program of GHC in Section 3.1 can be enhanced to fit to the requirements of Section 4. The enhanced "exec" has five arguments. These five arguments, in turn, denote the *execution goals*, the *variable bindings*, the *database*, the new *variable bindings* after the execution and the execution result.

This five-argument "exec" can be defined 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([G|Rest],Env,Db,NEnv,R)
        :- user_defined(G,Db)|
           reduce(G,Rest,Env,Db,NG,Env1,R1),
           exec(NG,Env1,Db,NEnv,R2),
           and_result(R1,R2,R).
exec([G|Rest],Env,Db,NEnv,R)
        :- system(G)|
           sys_exe(G,Env,Env1,R1),
           exec(Rest,Env1,Db,NEnv,R2),
           and_result(R1,R2,R).
```

By transforming these "exec" definitions to "meta" definitions, we can install them to the meta-level database. After that, it becomes possible to execute a given goal "p" in an interpretive manner. This is performed by using reflective predicate "interpretive," which is defined as follows:

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

Note that this interpretive execution can be mixed with other direct execution. Therefore, it is possible to execute the specific goals in an interpretive manner and execute other goals directly. Other possibility is modifying this "exec" and use this "exec" as a "debugger." We can perform these kinds of things in a quite straightforward manner.

5.5. Constructing and collapsing a reflective tower

In implementing *reflective system*, we initially create *supporting system* and execute *user goals* inside the supporting system. This supporting system can also be defined as an enhanced "exec." It can be defined as follows:

```
support([],Env,Db,R)
        :- true|R=success.

support([true|Rest],Env,Db,R)
        :- true|
           support(Rest,Env,Db,R).
support([G|Rest],Env,Db,R)
        :- user_defined(G,Db)|
           reduce(G,Rest,NG,Env,Db,Env1,R1),
           support(NG,Env1,Db,R2),
           and_result(R1,R2,R).

support([G|Rest],Env,Db,R)
        :- system(G)|
           sys_exe(G,Rest,NG,Env,Env1,R1),
           support(NG,Env1,Db,R2),
           and_result(R1,R2,R).

support([G|Rest],Env,Db,R)
        :- reflective(G,Db)|
           create_meta_db(Db,Meta_Db),
           shift_up((G,Rest,Env,Db),
                   (Up_Goal,Up_Rest,Up_Env,Up_Db)),
           support([reflect(Up_Goal,(Up_Rest,Up_Env,Up_Db),
                   (NRest,NEnv,NDb,R1))],[],Meta_Db,R1),
           shift_down((NRest,NEnv,NDb),
                     (Down_Rest,Down_Env,Down_Db)),
           support(Down_Rest,Down_Env,Down_Db,R2),
           and_result(R1,R2,R).
```

We easily notice that this "`support`" is almost same to the "`exec`" in Section 5.4 except the last definition clause.

This definition clause takes care of the creation of the reflective-tower. "`create_meta_db`" creates the meta-database by taking out *meta* and *reflective* definitions from the object-level database. `(G,Rest,Env,Db)` is shifted up and the meta-level representation `(Up_G,Up_Rest,Up_Env,Up_Db)` is generated. Then "`support`" starts the meta-level computation using these arguments. Note that meta-level environment is initially set to `[]` since the shifted-up expression does not include variables.

When the meta-level execution finishes, `(NRest,NEnv,NDb,R1)` are instantiated. We shift down this information and we get `(Down_Rest,Down_Env,Down_Db)` which denotes the new object-level information. Then we return to the object-level execution using this information.

Figure 6 shows how the reflective tower is constructed by calling reflective predicates and how it is collapsed by finishing up their execution.

6. Related works and conclusion

It seems to be that [Smith 84] and [Maes 88] present us the general background for our research. Regarding to related works, Sugano is proposing R-Prolog which is the reflective extension of Prolog [Sugano 89]. Lloyd is also proposing Gödel which is a meta-extension of Prolog [Lloyd 88]. Their approaches are quite similar to each other and their interests mainly exist in the reconstruction of Prolog which has cleaner semantics without using

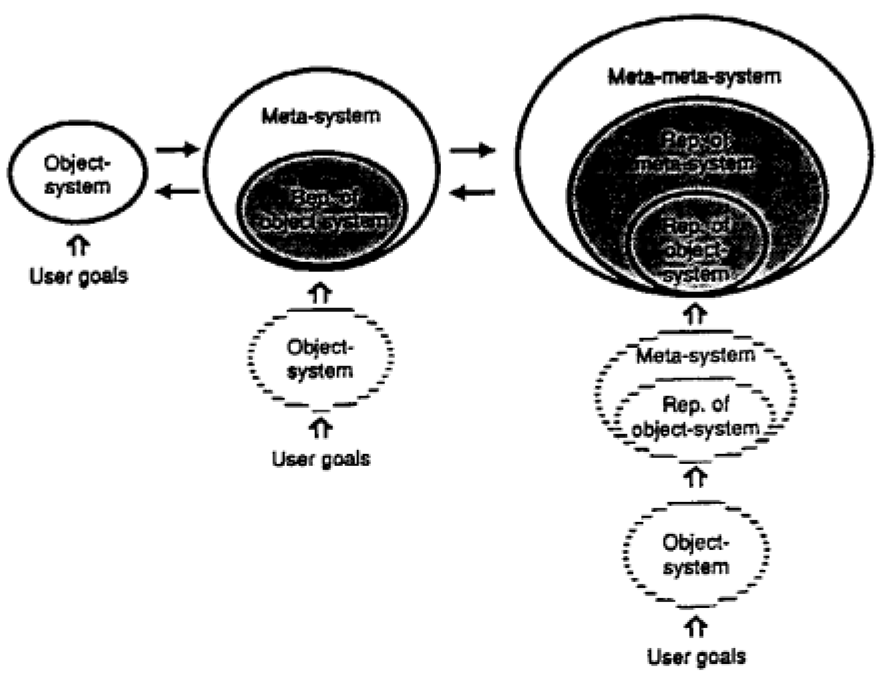

Figure 6: Constructing and collapsing a reflective tower

side-effects.

In this paper, we have proposed *Reflective GHC* which is the reflective extension of parallel logic language GHC. The features of our system can be summarized as follows:

1. Simple formulation of reflection in parallel logic language GHC. We have mentioned that logic languages have no notion of *evaluation* comparing to Lisp. Therefore, we have formulated reflection without using *quote*. This is the critical difference from Sugano's or Lloyd's approach.

2. Ground representation of variables. In our system, variables are expressed as *special ground terms* as described in Section 4.2. 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.

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 authors would like to express thanks to Yukiko Ohta for her useful comments, discussions and programming supports. Thanks also go to Hiroyasu Sugano, Youji Kohda and Masaki Murakami for their useful discussions.

References

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

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

[McCarthy 65] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart and M. I. Levin; LISP 1.5 Programmer's Manual, The M.I.T. Press, 1965

[Maes 86] P. Maes; Reflection in an Object-Oriented Language, in 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

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

[Sugano 89] H. Sugano: A Formalization of Reflection in Logic Programming, Research Report No.98, IIAS-SIS, Fujitsu Limited, 1989

[Tanaka 88a] J. Tanaka; A Simple Programming System Written in GHC and Its Reflective Operations, in 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, in Proceedings of of the International Conference on Fifth Generation Computer Systems 1988, pp.774-783, ICOT, November 1988

[Tanaka 90] J. Tanaka; An Overview of ExReps System, Fujitsu Scientific & Technical Journal, Vol.26, No.1, 1990, to appear

[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. In Artificial Intelligence 13, pp.133-170, 1980