ICOT Technical Report: TR-598

TR-598

Implementing Reflection in GHC

by
J. Tanaka (Fujitsu)

October, 1990

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

Implementing Reflection in GHC

(Extended Summary)

Jiro Tanaka

IIAS-SIS, FUJITSU LIMITED,
1-17-25 Shinkamata, Ota-ku, Tokyo 144, JAPAN
email: jiro@iias.fujitsu.co.jp

Abstract

Implementation of reflection in GHC is presented in this paper. Though GHC is an *parallel* dialect of Prolog, we can regard it as a parallel object-oriented language since it has the notion of *object* and *communication* between objects.

After reviewing the language features of GHC as a parallel object-oriented language, we consider the realization of GHC meta-computation system first. Based on the meta-level representation of GHC, an enhanced GHC meta-program is proposed. Then we propose Reflective GHC, where *reflective tower* can be constructed and collapsed in a dynamic manner using *reflective predicates*.

Reflective GHC has actually been implemented. All codes shown in this paper are running on our Reflective GHC system.

1. GHC as a parallel object-oriented language

Recently, with the advent of parallel/distributed hardware, parallel programming languages are attracting wide spread attention. The language we are interested in is the parallel logic language which *parallelizes* logic programming language Prolog. Though several languages, such as PARLOG [Clark 85] and Concurrent Prolog [Shapiro 83a], have already been proposed and those are very similar to each other, we have chosen *Guarded Horn Clauses* (GHC) [Ueda 85] as our target and implementation language since it has the simplest syntactical structure.

1.1. Basic syntax and computation rules

Unlike conventional programming languages, problem solving in GHC consists of two parts, i.e., a *program* and a *goal clause*.

A GHC program can be defined as a set of *guarded* Horn clauses of the following form:

$$H :- G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_k. \qquad (m, k > 0)$$

The operator | is called a commitment operator. The part of a clause before | is called a guard, and the part after | is called a body. Note that guard goals and body goals can be empty. In such cases, we use the special goal "true" to denote empty goals.

A *goal clause* has the following form:

$$:- B_1, ..., B_n. \qquad (n > 0)$$

We can regard each *guarded* Horn clause as the *rewriting rule* and the given *goal clause* as the *initial query*. The GHC program execution is performed by consecutively applying *rewriting rules* to a goal in the given *initial query*. This can be done in a fully parallel manner under the following rules.

1. Pick up a goal from the *initial query*.
2. If it is a system-defined goal, execute it directly. This results to instantiate variables in the *initial query*.
3. If it is a user-defined goal, find the candidate *guarded* Horn clauses from the program.
4. Compute the guard of each candidate clause. If the head unification and guard goals of a clause all succeeds, that clause is "committed" exclusively and the goal of the given goal clause is replaced to the body goals of the committed clause.
5. Computation invoked in the guard of a clause cannot instantiate the caller of that clause. If this happens, the computation suspends until that caller is instantiated by some other goal. This provides the basic synchronization mechanism of GHC.

If the *goal clause* becomes empty, it means that the computation is finished successfully. At that time, the computation result is given as the bindings of variables in the *initial query*.

It must be stressed that under the rules stated above, anything can be done in parallel: Goals in a given goal clause can be executed in parallel; candidate clauses for a goal can be tested in parallel; head unification and the execution of guard goals can be done in parallel. However, it is even more important to stress the fact that we can also execute a set of tasks in an arbitrary order or in an arbitrary partial order as long as it does not change the intended meaning of the program.

1.2. An example program

The following is an example of the GHC program.

```
gen(N,Max,Ns) :- N=<Max|
       Ns=[N|Ns1], N1:=N+1, gen(N1,Max,Ns1).
gen(N,Max,Ns) :- N>Max|Ns=[].

square([P|Ns1],Ms) :- true|
       P1:=P*P, Ms=[P1|Ms1], square(Ns1,Ms1).
square([],Ms) :- true|Ms=[].

sum([P|Ms1],Int,Sum) :- true|
       Int1:=Int+P, sum(Ms1,Int1,Sum).
sum([],Int,Sum) :- true|Sum:=Int.
```

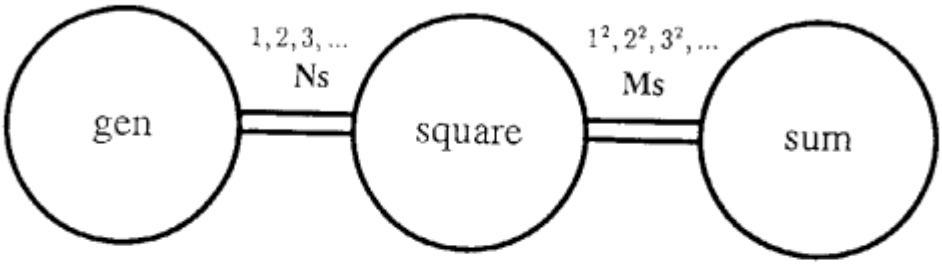

Figure 1: Computing *sum_of_square* number

This program consists of *six guarded* Horn clauses. It computes the *sum_of_square* number for the given query. Assume that the initial goal clause is given as follows:

```
:- gen(1,100,Ns),square(Ns,Ms),sum(Ms,0,Sum).
```

In this case, three goals, i.e., "`gen`," "`square`" and "`sum`," are created as shown in Figure 1 and $1^2 + 2^2 + 3^2 + ... + 100^2$ is computed as the binding of variable "`sum`" via stream communications between goals.

Looking at this example more carefully, we find that GHC programming is done in somewhat object-oriented manner. We can regard tail recursive goals, i.e., "`gen`," "`square`" and "`sum`," as *objects*. An *internal state* of a object is expressed as the values of the goal arguments. Goal arguments are also used to express *communication* between objects. If an argument is shared between two goals, we can send messages from one goal to the other by partially instantiating the shared variable. In this example, "`Ns`" and "`Ms`" are used as such shared variables.

Thus, we can consider GHC to be a *parallel object-oriented language* though it only have very primitive framework in expressing *object* and *communications* between them. At the same time, we notice that GHC is much simpler comparing to other existing object-oriented languages,

Note that Shapiro is the first to notice the similarity between parallel logic languages and object-oriented languages [Shapiro 83b]. He tried to realize other specific features of object-oriented languages, such as "class-subclass" hierarchy and "inheritance" mechanisms, in parallel logic languages. Also note that *A'UM* is another approach to develop Smalltalk like language in GHC [Yoshida 90].

2. Implementing a 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)|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 solved 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 such 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 [Smith 84] and Gödel [Lloyd 88] adopt this approach. However, we do not adopt this approach. Our claim is that there is little *practical* 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 we distinguish it from the ordinary *ground term*.

Our choice is expressing variables by "@number" at the meta-level, in which its own number is assigned for each variable at the object level. Though this representation looks too much low level comparing to the approach using *quote*, we have chosen this approach for implementation simplicity.

We also assume that the 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 are 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 *dereference* 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" program

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

is expressed as

```
[(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 goal "Goal" and given 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 in this predicate. The fourth argument contains the identification number which should be used next and the fifth contains the *environment* of this goal representation.

If we input "exam([H|T],T)" to "Goal,"
"transfer(exam([H|T],T),GRep,1,Id,Env)" is executed and the computation result is

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

The enhanced "exec" executes this *goal representation* and the computation result 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.*

```
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).
```

The meaning of this program is self-explanatory. Though we omit the detailed explanation, we easily note that this is the extension of the simple GHC meta-program in Section 2.1.

3. Reflection and Reflective GHC

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. 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 also note that *unit* of *reflection* we describe here is not the *object-level*, which is most popular in object-oriented world as seen in [Maes 86] and [Watanabe 88]. Instead, we are interested in more global, i.e. system level, *reflection* as seen in [Smith 84].

3.1. Two approaches 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. In this case, the object-system works as a *reflective* 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. We actually adopted this approach in implementing *reflection* in [Tanaka 88] and [Tanaka 90]. This approach has a merit that the implementation is relatively straightforward. However, at the same time, we should note that this approach is not the accurate implementation of *reflection* since 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 [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* 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 GHC*. Reflective GHC 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 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. We can define reflective predicates and use wherever we want, in the user program or in the initial query.

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 extra arguments, i.e., "`(G,Env,Db)`" and "`(NG,NEnv,NDb)`" 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* which contains the program. "`(NG,NEnv,NDb)`" 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* and "`NDb`" expresses the new *database*.

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)`" 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) |
           NEnv = [(R,unbound)|Env],
```

```
        (NG,NDb)=(G,Db).

reflect(var(X,R),(G,Env,Db),(NG,NEnv,NDb))
        :- bound(X,Env) |
           NEnv = [(R,bound)|Env],
           (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 *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),
           NEnv=[(N,X)|Env],
           (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."

3.3. Implementing Reflective GHC

In implementing Reflective GHC, there exists several possibilities. The most efficient implementation is re-designing the abstract machine code, which corresponds to Warren code, for Reflective GHC. 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 Reflective GHC system as an interpreter on top of 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 Reflective GHC using this method. In this case, the top level description of Reflective GHC 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 simple 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. "`(GRep,Rest,Env,Db)`" is shifted down and the meta-level 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 dereference 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 6 shows how the reflective tower is constructed by calling reflective predicates and how it is collapsed by finishing up their execution.

4. Related works and conclusion

It seems to be that [Smith 84], [Weyhrauch 80] and [Maes 88] present us the general background for our research. Regarding to related works, Lloyd is proposing Gödel which is a meta-extension of Prolog [Lloyd 88]. However, his interest mainly exists in the reconstruction of Prolog which has cleaner semantics.

The features of our *Reflective GHC* 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. 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.

We have already finished up the prototype implementation of Reflective GHC using PSI-II workstation. We used GHC to describe the core part of meta-program. User interface and i/o part are written in Prolog. All codes shown in this paper are running

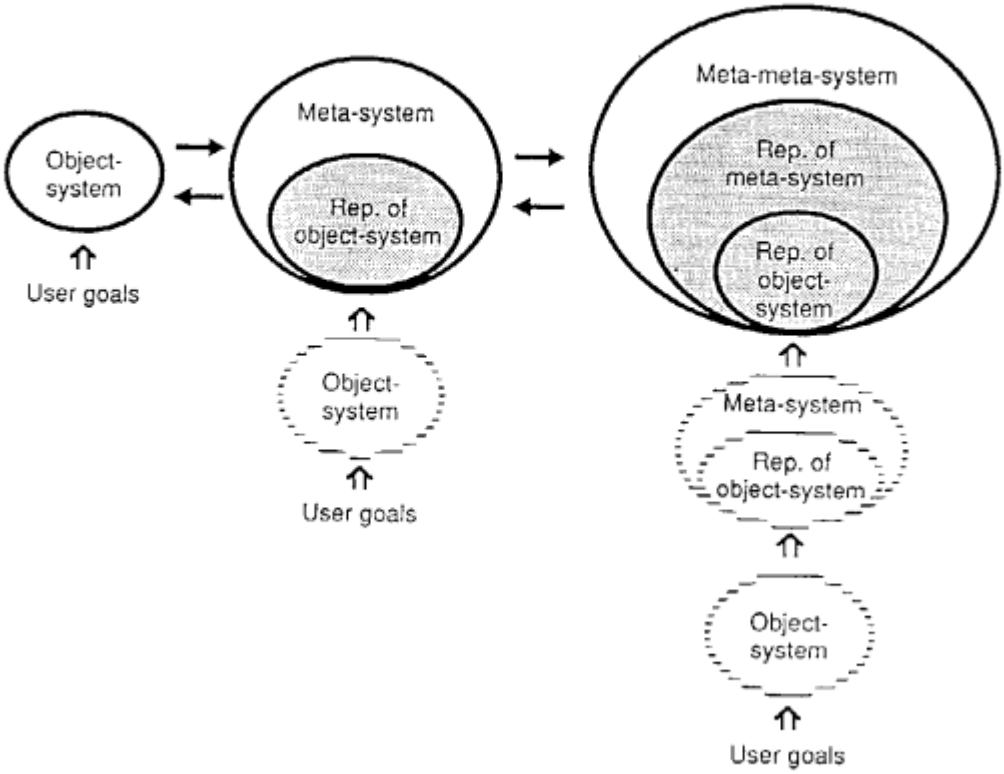

Figure 2: Constructing and collapsing a reflective tower

on our Reflective GHC system, though these are a little bit simplified than the actual implementation.

Also note that the execution speed of our meta-program was not slow than imagined. Though the cost of variable management is expensive in our implementation, this becomes negligible by using *vector*, where the *index search* is possible, in implementing *environment* [Fujita 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 88] [Tanaka 90].

5. 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 Fumio Matono, Yukiko Ohta, Hiroyasu Sugano and Youji Kohda their useful discussions.

References

[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.

[Clark 85] K. Clark and S. Gregory: PARLOG, Parallel Programming in Logic. Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, Revised 1985.

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

[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.

[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

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

[Shapiro 83b] E. Shapiro and A. Takeuchi: Object Oriented Programming in Concurrent Prolog, ICOT Technical Report, TR-004, 1983.

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

[Tanaka 88] 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, Y. Ohta and F. Matono: 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, in Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pp.16-36, ICOT, November 1988.

[Watanabe 88] T. Watanabe and A. Yonezawa: Reflection in an Object-Oriented Concurrent Language, in Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications, San Diego, September 1988, pp.306-315.

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

[Yoshida 90] K. Yoshida: A stream-based concurrent object-oriented programming language, Ph.D. Thesis, Keio University, 1990.