An Efficient Implementation of Reflection in KL1

Toshiyuki Takahashi Masayuki Takeda

Department of Information Sciences,
Faculty of Science and Technology,
Science University of Tokyo
2641 Yamazaki, Noda-Shi, Chiba, 278, Japan
E-mail: {tosiyuki,takeda}@is.noda.sut.ac.jp

Abstract

This paper proposes the reflective architecture to realize an efficient implementation of reflection in KL1. The previous approaches to implement the reflection facility in logic-based languages are almost all based on meta-interpreter methods which are far from practical. This paper aims at an efficient implementation of reflection in KL1. To this end, we developed a reflective architecture based on the notion of 'metabody' which is able to customize/control the execution of KL1 programs without meta-interpreter methods. Several examples using this architecture are shown and analyzed. This paper assumes a basic knowledge of parallel logic language.

1 Introduction

Computational Reflection was defined by Maes[5] as the behavior exhibited by a reflective system, where a *reflective system* is a computational system which is about itself in a causally connected way. A *computational system* is a computer-based system whose purpose is to answer questions or support actions in some domain. A computational system is said to be *causally connected* to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other.

Recently, computational reflection as a software paradigm has a great deal of attention. Implementation of a reflective facility into the programming language promotes the modularity of the program, and enhances the reusability and understandability of the code. Also, reflection makes module to change to another module dynamically. This will make the selection of executing strategy complying to the changing environment, and it contributes for the efficiency of the program execution.

Our aim is to achieve an efficient implementation of reflection in KL1. There are some experiments for implementation of the reflection facility into Parallel Logic Languages, and most of these approaches are based on meta-interpreter methods. Using meta-interpreter methods makes it easy to build the self-representation which is causally-connected to the aspects of the system. These methods are far from practical since interpretation methods cause overhead in execution, which results in poorer to the performance. To eliminate overhead

problems, application of partial evaluation has been suggested in some papers. But, these applications do not have enough performance and are roughly 10 times slower compared to the non-reflective programing language. And we have developed a reflective architecture based on the notion of '**metabody**' which is able to customize/control the execution of KL1 programs without meta-interpreter methods. This approach gains efficiency because it translates the enhanced KL1 code to the pure KL1 code.

In section 2, we explain metabody as an essential concept in reflective architecture. Section 3 presents how the body part of the base-level code customized by metabody and makes concept of meta-process group distinct. In section 4, we present two kinds of application programs, and in section 5, we discuss about suggestion of these expansions to KL1.

2 Fundamental Concepts

Reflective language which is designed using, base-level and meta-level codes that are written separately. We are going to use base-level codes to describe its domain. and then use meta-level codes to describe the customization of base-level codes and how to control their execution.[1] The codes for dynamic extention/modification to adapt to the new problems and environments are meta-level ones. For example, on a parallel computer, the codes solving the puzzle are in base-level and the codes for dynamic load balancing are meta-level codes.

Original KL1 has several primitives which assist description of the meta-level operation. For example, '`@node`' and '`@priority`' are used in meta-level operation. '`@node`' is the notation to specify the processor to execute the goal and '`@priority`' is the notation to specify the execution priority of the goal. In KL1 the programmer has to write compounded codes which contain two level codes, base-level and meta-level codes.

Our first motivation to implement reflective facility in KL1 is dividing these two level codes into separate codes. To achieve this, we made some extentions to KL1 to supplement the meta-level code into the existing base-level code.

Figure 1 illustrates the extended image of the KL1. The arrow stands for control flow, and the dotted line stands for division between base-level and meta-level codes. The clause starting with `p` is the base-level code, and the clause starting with `meta_p` is the supplemented meta-level code.

As we can conjecture from the control flow, if clause `p` commits, the body part of the clause `p` will not be executed, but instead `meta_p` will be executed. The `$execbody` in meta-level is the special goal in which the execution of this goal will pull the trigger for the base-level body's execution. In the illustrated example in figure 1, two goals `x` and `y` are invoked in the clause `meta_p`. In result, five goals `x,y,r,s,t` are invoked.

[1] The meta-level codes can customize/control the meta-level codes themselves.

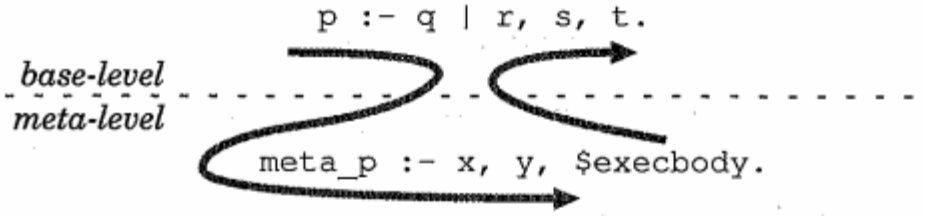

Figure 1: Extended control flow

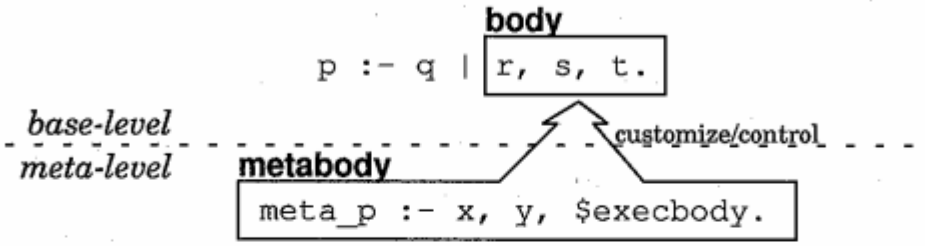

Figure 2: Metabody customization of the body

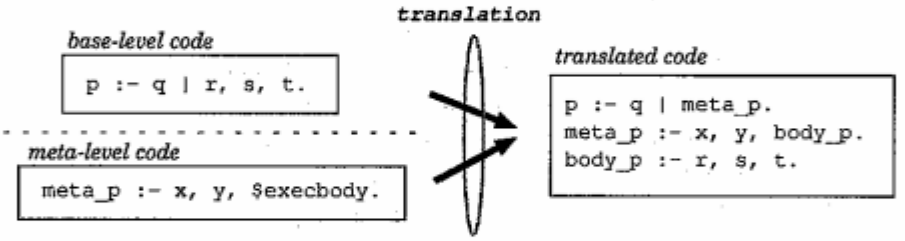

Figure 3: Translation to compounded codes.

Figure 2 is another representaion of figure 1 at the different point of view. Body part in the base-level can be customized by the meta-level code which is called **metabody**. [2]

The extended KL1 programs can be converted into normal ones. Figure 3 shows how the base-level and meta-level codes are translated into compounded codes.

3 Reflective Architecture

3.1 Metabody

In KL1 specification, the code which distributes goals to the computed `X` by using `loadb:assign_node/1` are expressed as below:

[2] While we describe the metabody in meta-level, it can also control the body part's execution

```
:-module sample1.
foo :- loadb:assign_node(X), foo1@node(X).
```

The same code described in extended KL1 specification are as follows:

```
:-module sample1.
%reflect metalb:distrib.
foo :- foo1.

:-metamodule metalb.
:-metabody distrib/0.
distrib :- loadb:assign_node(X), $execbody@node(X).
```

In the example above, base-level codes are between declaration `:-module` and declaration `:-metamodule`, and meta-level codes follow the declaration `:-metamodule`. *Metamodule* is the module which describes the meta-level code. The declarations of metamodule begin with `:-metamodule` *Metamodule*, and the meta-level code such as metabody follows. End of file or the declarations of another module (or metamodule) will terminate the metamodule description. Since there is a description in the base-level code that begins with `%reflect`, this specifies a relation between base-level clause and metabody. In this case, the base-level clause `foo` and the metabody `distrib` are related. That is, if the clause `foo` commits, the metabody `metalb:distrib` will be invoked.

Metabody is described into the metamodule. Definition of a metabody begins with a metabody declaration of the form `:-metabody` *Metabody/Arity* followed by its clauses. In the example `metalb:distrib/0`, when the `distrib/0` is invoked, the goal `loadb:assign_node/1` is called, which is an invocation of the `assign_node/1` inside the module($\neq$ metamodules). Then, by the goal `$execbody`, the body part of the base level will be executed. Under these circumstances, we implement a load balancing by specifying the processor with the pragma `@node`.

It is also possible to hand over the argument to the metabody to make the possibility of customizing the body execution for the furthur advancement. The argument to hand over to metabody other than ground term is the name of the variable in the body part of the clause, and the group stream which we describe in the next section.

In the example below, the parameter `^P` which denotes the the variable `P` in the body part is passed to the metabody `msample2:mfoo/1`.

```
:-module sample2.
%reflect msample2:mfoo(^P).
foo(P,Q):-calc1(P),calc2(Q).

:-metamodule msample2.
:-metabody mfoo/1.
mfoo(X):-$rep_v(X,Y),base:calc3($X,Y),$execbody.
```

In the example of metabody msample2:mfoo/1, the received variable name is used for two purposes. First, it is transfered to invoked goal: base:calc3/2 as an argument. In the metabody, a variable[3] which has modifier symbol '$' on its name is unified with the variable in base-level which is indicated by value of received variable itself. In above example, 'received variable': X=^P, therefore $X=P. Secondary, it is used for replacing the base-level variable with another one. It is possible to replace variable in base-level body (received variable's name) with another one before execution of $execbody/0, using special goal $rep_v/2 in metabody.

$rep_v/2 is a clause that replaces the variable given in first argument (base-level variable) with another one given in second argument. The Program described above is equivarent to the following one:

```
:-module sample2.
foo(P,Q):-base:calc3(P,Y),calc1(Y),calc2(Q).
```

Below is the attractive example of $rep_v/2:

```
:-module sample3.
main :- calc(X),output(X).
%reflect compe:compe([1,2,3],^X).
calc(X):-task(X).
task(X):- ... X=[Ans].

:-metamodule compe.
:-metabody compe/2.
compe(NL,Out) :- merge(R,S),
  waitans(S,$Out), compe2(NL,Out,R).
compe2([N|NL],Out,R) :- R={R1,R2}, compe2(NL,Out,R2),
  $rep_v(Out,R1),$execbody@node(N).
compe2([],_,R) :- R=[].
waitans([Ans|_],Out) :- Out=[Ans].
```

Metabody introduced in above code, compe:compe/2, is the meta level code which can get the result of calculation as fast as possible, by calculating in each node separately. The clause compe:compe/2 generates copy of the goal in the body of base level clause, and distribute them on each node. Then, it returns the result of the node that finishes the fastest as global answer. In this example, The same goal task/1 is distributed to the three nodes that has No. 1-3, then the result that is returned in first from one of these nodes is unified to X which is to accept the answer originally. To receive the answer, we use merge/2 which is built-in predicate of KL1. It should be noticed that meta-level code for distributed computation and receiving result is separated from base-level code clearly by application of extended KL1.

Customization such as replacement of goal in base-level body with another one can be made by use of metabody. The following is an example of the meta-level code for replacement of goal (exists in base-level body) which is given in

[3]this variable should have the name of variable that appears in base-level body.

actual argument of metabody in the form of 'functor/arity' with another goal. Special goal `$rep_g/2` described in metabody is similar to `$rep_v/2`. It replaces the goal given in first argument (of metabody) in the form of 'functor/arity' with one given in second argument:

```
:-module sample4.
%reflect msample4:mfoo(calc2/1).
foo(P,Q):-calc1(P),calc2(Q).

:-metamodule msample4.
:-metabody mfoo/1.
mfoo(X):-$rep_g(X,base:calc1/1),$execbody.
```

Above code is equivarent to next one:

```
:-module sample4.
foo(P,Q):-calc1(P),base:calc1(Q).
```

3.2 Meta-process Group

In this section, we introduce the concept of **meta-process-group**. We call an execution on the base-level as **base-process**, and an execution of a metabody as **meta-process**. For resource management on meta-level, coordination between meta-process belonging to the same group sharing the same resource is necessary. We named this mechanism for the coordination **meta-process-group**. It consists of **group-stream**, which is the common communication path for both meta-process and the group, and **group-controller** which controls resources shared in the group.

The following is an example program using meta-process-group:

```
:-module sample5.
%defgrp metalb:initgrp(ns).
main :- gen(X),foo(X).
foo([X|L]) :- foo1(X), foo(L).
foo([]) :- true.
%reflect metalb:assign_node(ns).
foo1(X) :- calc(X).

:-metamodule metalb.
:-metagrp initgrp/1.
initgrp(*strm) :- demander(*strm).
:-metabody assign_node/1.
assign_node(*strm) :- *strm<<X, $execbody@node(X).
```

This program performs the coordinative load balancing. The diffrence between the sample of load balancing program(`sample1`) which we mentioned before, is

that it makes the meta process to perform the coordinative balancing strategy[4] opposed to the sample before which can only perform the incoordinative[5] one.

The notation `%defgrp` imported in module `sample5` is the declaration of the meta-process-group. The invocation of the group-controller and the name of the group stream are defined here. In the sample, the group controller `initgrp/1` descripted in the metamodule `metalb` is invoked, and `ns` will be defined as the "group-stream name". The declaration `:-metagrp` is added to the meta-level code. Declaration `:-metagrp` *Group-controller* will define the following clause as a group controller. In the example above, it defines `initgrp/1` as the group controller. When the `initgrp/1` is called, it invokes the goal `demamder/1`. In the meta-level code, we represent the name of the group-stream with prefix `*`. In the code above, `*strm` is a group stream. Also, in the sample above, `<<` is an operator which sends message `X` to `*strm`. By mediating the group-stream, message `X` which was sent to group-controller is calculated and unified.

4 Examples of Reflective Programming

4.1 Load Balancer

Program `sample5` shown in section 3.2 is translated into KL1 as follows:

```
:- module sample5.

main :- gen(X),foo(Ns,X),
  merge(Ns,Ns1),demander(Ns1).

foo(Ns,[X|L]) :- Ns={Ns1,Ns2}, foo1(Ns1,X), foo(Ns2,L).
foo(Ns,[]) :- Ns=[].

foo1(Ns,X) :- Ns=[X], foo1_1_1b(X)@node(X).
foo1_1_1b(X) :- calc(X).
```

4.2 Debugger

For a practical example, we show an application to the debugger. The meta-module for the debugger are shown below:

```
:-metamodule debugger.

:-metagrp init/1.
init(*io) :- debugger:init(*io).

:-metabody spy/1.
spy(*io) :- debugger:catch(*io,Cmd,$functor,$args),
```

[4] a strategy like choosing the node depending on statistical information, distribute the job to the unoccupied node at high priority, ... etc.

[5] a strategy like choosing the node at random, in turn, ...etc.

```
            spy2(Cmd).
spy2(cont) :- $execbody.
spy2(halt) :- halt.
```

Then, we show a program which calculates prime number, with debugging aids:

```
:-module primes.

%defgrp debugger:init(tio).
primes(Max,C) :- gen_primes(Max,Ps), count(Ps,C).

gen_primes(Max,Ps) :- gen(2,Max,Ns), sift(Ns,Ps).

sift([],Zs0):-Zs0=[].
%reflect debugger:spy(tio).
sift([P|Xs1],Zs0) :- Zs0=[P|Zs1],
  filter(P,Xs1,Ys), sift(Ys,Zs1).

filter(_,[],Ys0) :- Ys0=[].
%reflect debugger:spy(tio).
filter(P,[X|Xs1],Ys0) :- X mod P=\=0 | Ys0=[X|Ys1],
                                       filter(P,Xs1,Ys1).
filter(P,[X|Xs1],Ys0) :- X mod P=:=0 | filter(P,Xs1,Ys0).
```

The translated code for `sift` is as follows:

```
sift(Tio,[],Zs0) :- Tio=[], Zs0=[].
sift(Tio,Arg1,Zs0) :- Arg1=[P|Xs1] | Tio={Tioa,Tiob},
  debugger:catch(Tioa,Cmd,sift_2_2,[Arg1,Zs0]).
  sift_2_2_debugger_spy2(Cmd,Tiob,P,Xs1,Zs0).
sift_2_2_debugger_spy2(cont,Tio,P,Xs1,Zs0) :-
  sift_2_2b(Tio,P,Xs1,Zs0).
sift_2_2_debugger_spy2(skip,Tio,_,_,_) :- Tio=[].
sift_2_2b(Tio,P,Xs1,Zs0) :-
  Tio={Tioa,Tiob},
  Zs0=[P|Zs1], filter(Tioa,P,Xs1,Ys), sift(Tiob,Ys,Zs1).
```

5 Discussions

5.1 Relationship to Other Works

From the work ABCL/R2, we noticed that reflection is an useful mechanism to construct parallel/distributed system, and their achievement are the motivation of our work.[6][7][8]

The original idea for making the relationship between base-level and meta-level are from Meta Object Protocol in OpenC++.[1][2]

An issue for introducing reflection to GHC is currently done by Jiro Tanaka. Tanaka used meta-interpreter to implemenent Reflective GHC (RGHC) but

that implementation was far from practical since the overhead caused by the interpreter are huge. The reflective architecture(metabody, meta-process-group) which we proposed translates the program into KL1 code, and the compounded code after the translation will be quite similar to the code which does not use the Reflective Architecture, so it will be very efficient and practical.[10][11]

5.2 Future Works

- Our architecture restricts the reflection of the body part, but the argument about the effect on reflection is not enough.
- Metabody is described in KL1, so it is possible to make meta- meta-body, and by using that, we have to work on the reflective tower that we may construct.
- To appeal the usefullness of this reflective Architecture, we can apply our extended KL1, to the system for syntax error diagnosis of programming languages and this will be the subject of a future paper.

6 Acknowledgements

We would like to thank Kazuaki Rokusawa and Kasumi Susaki at ICOT for discussioning from the early period, and Akinori Yonezawa labo at The University of Tokyo, Jiro Tanaka labo at University of Tsukuba for their helpful comments, and Takashi Chikayama, Kazunori Ueda, the members of ICOT KLIC-TG also gave us many advices. We would also like to thank Tomosuke Takanashi and Jun Konosu for the help in writing the paper, and Ph.D. Kenzo Inoue who encouraged us.

References

[1] Chiba,S.and Masuda,T. *Open C++ and Its Optimization*, In Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures, Washington, D.C., 1993.

[2] Chiba,S. *Open C++ Release 1.2 Programmer's Guide*,Technical Report No.93-3, Dept. of Information Science, Univ. of Tokyo, 1993.

[3] Chikayama,T. *Introduction to KL1*, ICOT, 1994.

[4] Chikayama,T. *KLIC User's Manual*, ICOT, 1994.

[5] Maes,P. *Issues In Computational Reflection*, In Meta-Level Architectures and Reflection, pp.21-35, North-holland, 1988.

[6] Masuhara,H. and Matsuoka,S. and Yonezawa,A. *Designing an OO Reflective Language for Massively-Parallel Processors*, In OOPSLA'93 Proceedings,1993

[7] Masuhara,H Matsuoka, Watanabe,T Yonezawa,A. *Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently*, In OOPSLA'92 Proceedings, 1992, pp.127-144.

[8] Masuhara,H. *Study on a reflective architecture to provide efficient dynamic resource management for highly-parallelobject-oriented applications*, Masters Thesis, Univ. of Tokyo, 1994.

[9] Matsuoka,S. and Watanabe,T. and Yonezawa,A *Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming*, In ECOOP'91 Proceedings, 1991, pp.231-250.

[10] Tanaka,S. and Matono,F. *Constructing and Collaspsing a Reflective Tower in Reflective Guarded Horn Clauses*, In Proceedings of theInternational Conference on FGCS, 1992.

[11] Tanaka,J. *Meta-interpreters and Reflective Operation in GHC*, In Proceedings of the International Conference on FGCS, 1988.

[12] Watanabe,T. *An Actor-Based Metalevel Architecture for Group-Wide Reflection*, In Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands, Lecture Notes in Computer Science. Springer-Verlag, 1990.

[13] Watanabe,T. and Yonezawa,A. *Reflection in an Object-Oriented Concurrent Language*, In OOPSLA'88 Proceedings, 1988, pp.306-315.