ICOT Technical Report: TR-343

TR-343

A Simple Programming System Written in GHC and its Reflective Operations

by
J. Tanaka

February, 1988

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

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Simple Programming System Written in GHC and Its Reflective Operations

Jiro Tanaka

ICOT Research Center,
Institute for New Generation Computer Technology,
1-4-28 Mita, Minato-ku, Tokyo 108, Japan

Abstract: A programming system can be defined as an environment where one can input and execute programs. In this paper, we try to describe a simple programming system written in GHC. We try to capture the function of *metacall* first. Input/Output problems in GHC are also considered. After describing *shell*, we try to assemble these parts into a simple programming system. How to add new features such as *reflective* operations to the programming system is also discussed. This paper assumes the basic knowledge of Parallel Logic Languages such as PARLOG, Concurrent Prolog or GHC.

1. Introduction

Various kinds of *Parallel Logic Languages*, which are based on and-parallel execution of programs, have been proposed so far. PARLOG [Clark 85], Concurrent Prolog [Shapiro 83] and GHC [Ueda 85] are the examples of such languages. In these languages, we can create *processes* dynamically and express the *synchronization* between processes pretty easily. Therefore, it seems to be quite natural to try to describe an operating system in these languages.

Trying to write an operating system in a logic language is not new. SIMPOS [Takagi 84] may be one of that approach. However, the resulting system was full of side effects and far from the logic programming. In parallel logic languages, PPS (Parlog Programming System) [Foster 86, Clark 87] and Logix [Silverman 86] has already been developed. However, PPS or Logix are kinds of empirical system which consists of huge amounts of codes and their overall structures are not clear enough.

In this paper we try to describe a simple programming system written in GHC. Our objective is not building up the practical programming system such as PPS or Logix. Our interest rather exists in expressing an *simple* programming system in more systematic manner. We also would like to test new features of a programming system such as *reflective* operations. We also believe that our efforts lead to the development of new programming techniques in parallel logic languages and prove the expressive power of GHC in the long run.

The organization of this paper is as follows: In section 2, we describe *metacalls* which present us the basic functions of our programming system. In section 3, input and output handling in the programming system are discussed. In section 4, UNIX-like *shell* and the overall structure of a simple programming system are described. In section 5, *reflective* operations in the programming system are described.

2. Metacalls and their descriptions

User programs can be executed on a programming system. However, the programming system cannot be failed even if a user program fails. Therefore, we need the *metacall* mechanism [Clark 84, Foster 87] which protects the programming system from the failure. This *metacall* predicate executes the given goal and reports the execution result.

2.1 Various metacalls

Various kinds of *metacalls* have already discussed in [Clark 84]. Here, we briefly review their works. The simplest metacall is the following one argument metacall.

```
call(G)
```

This metacall simply executes goal *G*. However, this form of metacall does not help us much because it is too simple. Therefore, the following two argument metacalls has been proposed.

```
call(G,R)
```

This metacall executes goal *G* and return the result by instantiating *R* to *success* when succeeded and to *failure* when failed.

The next extension is the following three argument metacall, which has slightly been modified from [Clark 84].

```
call(G,In,Out)
```

Here, *In* is called *input stream* and used for communication from the system to the metacall. *Out* is called *output stream* and used for communication from the metacall. We can suspend, resume or abort the goal execution by instantiating *In* to `[suspend|In']`, `[resume|In'']` or `[abort|In']`. The current state of the metacall can also be asked by instantiating *In* to `[state(S)|I']`. When the execution of the metacall finishes successfully, *Out* is instantiated to `[success]`. When failed, *Out* is instantiated to `[failure(R)]`, where *R* is instantiated to the message which shows the cause of the failure.

2.2 Metacall and meta-interpreter

The next problem is how to implement these metacalls. Considering efficiency, they should be implemented as a built-in predicate. In fact, metacalls are prepared as an primitives in [Clark 84]. On the other hand, Shapiro takes the meta-interpreter and program transformation approach to keep flexibility [Silverman 86].

If we forget the efficiency, it is possible to express these *metacalls* as *meta-interpreter*. The original notion of *meta-interpreter* comes from the self-description of EVAL in Lisp. In Prolog, the following self-description is very famous [Bowen 83].

```
exec(true) :- !.
exec((P,Q)) :- !, exec(P), exec(Q).
exec(P) :- clause((P:-Body)), exec(Body).
exec(P).
```

This meta-interpreter simply executes the goal which is given as the argument of *exec*. You may notice that this meta-interpreter corresponds to the implementation of one argument metacall. The GHC version of this meta-interpreter can be written as follows:

```
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.
```

This program is quite same to the Prolog program except that every clause definition includes | operator.

The two argument metacalls can be written quite similarly by modifying this one argument meta-interpreter.

```
exec(true,R) :-  true | R=success.
exec(false,R) :-  true | R=failure.
exec((P,Q),R) :-  true | exec(P,R1),exec(Q,R2), and_result(R1,R2,R).
exec(P,R) :- not_sys(P) | reduce(P,Body), exec(Body,R).
exec(P,R) :- sys(P) | sys_exe(P,R).
```

2.3 Three argument metacall implementation

It is already mentioned that metacalls should be implemented as a built-in predicate considering efficiency. However, the three argument metacall may be too complex to implement it so. Its specification also needs to be flexible. Therefore, we adopted a kind of hybrid approach. We split the metacall into two parts, the *exec* part which realizes the basic function of metacall and the *exec_server* part which is in charge of other services. If we forget the efficiency, the *exec* can be expressed as follows:

```
exec(true,In,Out) :- true | Out=[success].
exec(false(R),In,Out) :- true | Out=[failure(R)].
exec((A,B),In,Out) :- true | exec(A,In,O1),
        exec(B,In,O2), omerge(O1,O2,Out).
exec(A,In,Out) :- sys(A),var(In) | sys_exe(A,In,Out).
exec(A,In,Out) :- is_io(A),var(In) | Out=[A].
exec(A,In,Out) :- not_sys(A),var(In) |
        reduce(A,In,Body,Out,NewOut),exec(Body,In,NewOut).
exec(A,[susp|In],Out) :- true | wait(A,In,Out).
exec(A,[abort|In],Out) :- true | Out=[aborted].

wait(A,[resume|In],Out) :- true | exec(A,In,Out).
```

```
wait(A,[abort|In],Out) :- true | Out=[aborted].
```

The unique feature of this *exec* is *In* and *Out* which connect the object level and the meta level. Note that I/O is handled as a message to the meta level. Also *var(In)* is the special predicate which checks the absence of messages in the argument variable.

The *exec_server* part which is in charge of other services can be expressed as follows:

```
exec_server(State,G,EI,[success|EO],In,Out) :- var(In) |
        Out=[output([success,goal=G])].
exec_server(State,G,EI,[failure(R)|EO],In,Out) :- var(In) |
        EI=[abort],
        Out=[output([failure,reason=R])].
exec_server(State,G,EI,[G|EO],In,Out) :- var(In), is_io(G)|
        Out=[G|Out1],
        exec_server(State,G,EI,EO,In,Out1).
exec_server(State,G,EI,[undefined(G)|EO],In,Out) :- var(In) |
        Out=[input([undefined_goal=G, expected_result?],NG)|out1],
        G=NG,
        exec_server(State,G,EI,EO,In,Out1).
exec_server(State,G,EI,EO,[C|In],Out) :- true |
        control_receiver(C,State,G,EI,EO,In,Out).
```

This *exec_server* has six arguments. The first argument shows the internal state of the metacall. The second argument keeps the initial goal *G* and used for user output. The third and fourth arguments are connected to *exec*. The fifth and the sixth arguments are used for communication to the system. When the *exec_server* receives a success or failure message from the *exec*, it transmits the message to the system by adding the appropriate message. I/O messages are also forwarded to the system. When undefined goal G appears in *exec*, *exec_servers* sends the message to the user and urges us to input the new goal NG which corresponds to the execution result. When *exec_server* receives control message from the outside, *exec_server* invokes *control_receiver*.

The *exec* and *exec_server* can be connected as follows:

```
call(G,In,Out):- true|
        exec_server(run,G,EI,EO,In,Out),
        exec(G,EI,EO).
```

We should notice that the pair of *exec_server* and *exec* works as a metacall as a whole.

3. Input and output

Handling of input and output in logic programming is the important problem. We assume virtual processes which correspond to the actual devices. We consider that there exists a single stream which connects these virtual processes to the system. These virtual processes are always *consuming* a stream. For output we send the message of the form `output(Message)`. The input message has the format `input(Message_list, X)`. In this case, *Message_list* is printed first, then the user's input is instantiated to *X*. (Other

possibility may assume two streams which correspond to input and output. However, the synchronization of input and output becomes difficult in such case.)

In our approach, virtual processes can be created by *create* predicate in a program. This predicate is the special one in which we can execute only once in our program for each virtual process.

3.1 Window

Window is usually created on a bitmap display and we can input and output messages from there. For example, a window is created when `create(window,X)` is executed. The input and output to the window will be expressed as messages to stream X. (The actual input is completed when we put the cursor on the window and type in messages from the keyboard.) The virtual processes which correspond to devices will be deleted by instantiating X to $[]$.

3.2 Keyboard controller

As mentioned above, input and output will be performed by the request of the program. That is, the system does not accept the keyboard input without the demand of the program. Therefore we need to make the program which always generates the demand. Keyboard controller acts as such a program in a programming system. Keyboard controller can be written as follows:

```
keyboard(Out,In) :- true|
        Out=[input([@],T)|Out1],
        keyboard(T,Out1,In).

keyboard(halt,Out,In) :- true|
        Out=[],
        In=[].

keyboard(T,Out,In) :- goal_or_command(T) |
        In=[T|In1],
        Out=[input([@],T1)|Out1],
        keyboard(T1,Out1,In1).
```

3.3 Database server

In the programming system, we need database capability which can add, delete and check program definitions. The database capability is a kind of I/O in a broader sense. Therefore, we imagine the virtual process which corresponds to the database. The operations to the database can be realized by messages to the virtual process.

In fact, PPS tries to realize the database in such a way [Clark 87]. However, every *exec* needs to carry the stream to database in that case. This is very complicated and this may cause the database access bottleneck. Therefore, we have implemented *db_server* using side effect as follows:

```
db_server([add(Code)|In],ready,Out):-true|
        add_definition(Code,Done,Out,Out1),
```

```
        db_server(In,Done,Out1).
db_server([delete(Name)|In],ready,Out):-true|
        delete_definition(Name,Done,Out,Out1),
        db_server(In,Done,Out1).
db_server([definition(Name)|In],ready,Out):-true|
        definition(Name,Done,Out,Out1),
        db_server(In,Done,Out1).
```

The *db_server* predicate has three arguments. The first argument is the input from the system. The second argument is used to sequentialize the database access. The third argument is the output to the system.

4 Building a programming system

We have already discussed about the description of *metacall* and the handling of I/O in the programming system. The next step is the construction of a programming system. In this section, we describe *shell* which plays the central role in the programming system first. Then we try to assemble these parts into a simple programming system.

4.1 Shell

Shell creates the user task or enter the program to the database, depending on messages from the user. The following is the programming example for *shell*.

```
shell([],Val,Db,Out):-true|
        Val=[],
        Db=[],
        Out=[].
shell([goal(Goal)|In],Val,Db,Out):-true|
        Val=[record_dict(Goal,NGoal)|Val1],
        create(Window,WOut),
        keyboard(KO,PI),
        exec_server(run,NGoal,EI,EO,PI,PO),
        exec(NGoal,EI,EO),
        shell(In,Val1,Db,Out),
        merge(KO,PO,WOut).
shell([db(Message)|In],Val,Db,Out):-true|
        Db=[Message|Db1],
        shell(In,Val,Db1,Out).
shell([binding(Message)|In],Val,Db,Out):-true|
        Val=[Message|Val1],
        shell(In,Val1,Db,Out).
```

The *shell* has four arguments; the first argument is the input stream, the second one is the stream to the variable dictionary, the third one is the stream to database server, and the fourth one is the output stream. This *shell* are connected to the variable dictionary where user can freely define variables and its bindings. This variable dictionary presents us a kind of user interface and macro facilities. This *shell* program works as follows:

(1) If the input stream is □, it means the end of input. All streams will be closed in this case.

(2) If `goal(Goal)` is in the input stream. *Goal* is sent to the variable dictionary. The variable dictionary checks the bindings of every variable in *Goal* and creates *NGoal* where every variables are bound to the current bindings. Also a window, a keyboard controller, an *exec_server* and an exec will be created.

(3) If the message to the *database_server* or the variable dictionary is received, the message is sent to the appropriate stream.

The function of the variable dictionary is to memorize the value of variables as its internal state. It works as a kind of a user interface.

The following Figure 1 shows the snapshot where processes are created in accordance with the user input.

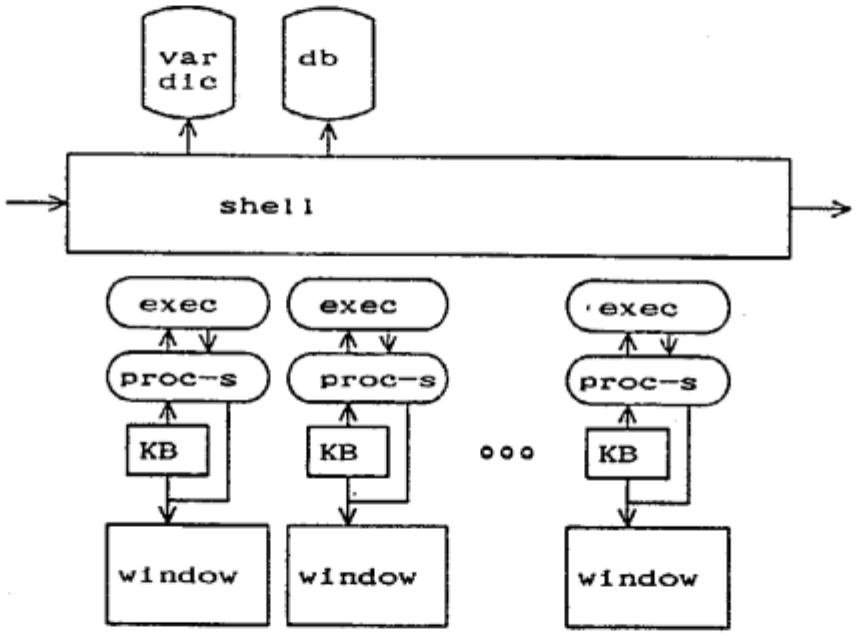

Figure 1. The creations of processes in *shell*

Corresponding to the input goal, four processes, i.e., exec, process server, keyboard controller and window, are dynamically created.

Since four processes have their own window and keyboard, they can run quite independently from *shell*. Keyboard controller always sends the read request to the window, and user can input the control commands to the process server from the window.

4.2 An example of a programming system

We show an example of a programming system by connecting the components described before.

```
create_world :- true|
        create(window,Out),
        keyboard(Out,In),
        shell(In,Va,Db,Out2),
        vr_dictionary(Va,□,Out3),
        db_server(Db,ready,Out4),
        merge4(Out1,Out2,Out3,Out4,Out).
```

This program can be illustrated as follows:

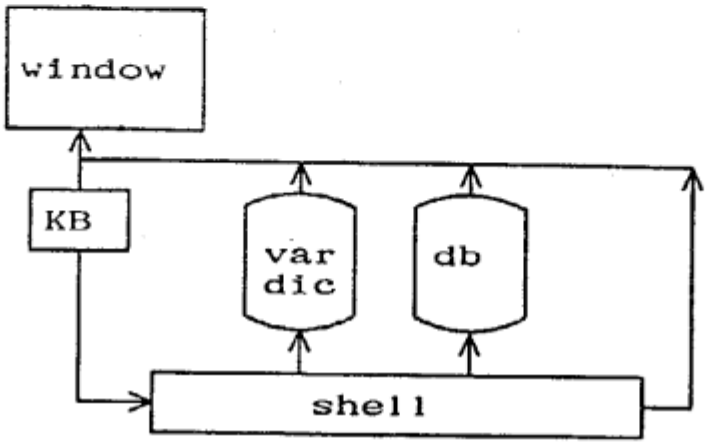

Figure 2. A simple programming system

Here, we create system window, keyboard controller, shell, variable dictionary and database server. We connect the outputs of shell, variable dictionary and database server to system window, together with the output of keyboard controller.

Since keyboard controller always generates the read request to the system window, we can input goals from there. Commands to the database server or variable dictionary can also be entered.

5. Reflective operations in the programming system

We sometimes need the ability to catch the current state of the system. Also we need is the capability of modifying and returning it to the system. These kinds of *reflective* capabilities, such as seen in 3-Lisp [Smith 84], seem to be very useful in the operating system. In 3-Lisp we can freely obtain the current ***continuation*** and ***environment*** from the program. Smith used meta-circular interpreters as a mechanism to get the information from the program.

Similar to Smith's approach, we extend our meta-interpreter. How we extend the meta-interpreter? It depends on what kind of resources we want to control. We would

like to control computation time, memories and processes in the programming system. Therefore, we introduce *scheduling queue* explicitly in our meta-interpreter. Program *continuation* was explicit in the meta-circular interpreter in 3-Lisp. We have thought that *scheduling queue* acts as *continuation* in GHC. We also introduce *reduction count* to control computation resources. We assume that this *reduction count* corresponds to the *computation time* in conventional systems.

The three argument *exec* in 2.3 becomes the following six argument *exec* when we perform the above mentioned modification.

```
exec(T,T,In,Out,MaxRC,RC) :- true |
        Out=[success(reduction_count=RC)].
exec([true|H],T,In,Out,MaxRC,RC) :- true |
        exec(H,T,In,Out,MaxRC,RC).
exec([false(R)|H],T,In,Out,MaxRC,RC) :- true |
        Out=[failure(R)].
exec([A|H],T,In,Out,MaxRC,RC) :- sys(A),var(In),MaxRC>RC |
        sys_exe(A,T,NT,RC,RC1,Out,NOut),
        exec(H,NT,In,NOut,MaxRC,RC1).
exec([A|H],T,In,Out,MaxRC,RC) :- is_io(A),var(In),MaxRC>RC |
        Out=[A|NOut],
        RC1=RC+1,
        exec(H,T,In,NOut,MaxRC,RC1).
exec([A|H],T,In,Out,MaxRC,RC) :- not_sys(A),var(In),MaxRC>RC |
        reduce(A,T,NT,RC,RC1,Out,NOut),
        exec(H,NT,In,NOut,MaxRC,RC1).
exec(H,T,In,Out,MaxRC,RC) :- MaxRC>=RC |
        Out=[count_over].
exec(H,T,[Mes|In],Out,MaxRC,RC) :- true |
        control_exec(Mes,H,T,In,Out,MaxRC,RC).
```

The first two arguments of *exec*, i.e., H and T, express the scheduling queue in Difference list form, which was originally used in [Shapiro 83]. Notice that goals are processed sequentially because we have introduced a scheduling queue. The third and the fourth arguments are the same as before. The fifth argument $MaxRC$ shows the limit of the reduction count allowed in that *exec*. The sixth argument RC shows the current reduction count.

There is no notion of job priority in this *exec*. We sometimes would like to execute goals by express. Therefore we also introduce *express queue* to execute *express goals* which have the form *G@exp*. This can be realized by adding two more arguments EH and ET, which correspond to the express queue, to six argument *exec*. The following two definitions describe the transition between six argument *exec* and eight argument *exec*.

```
exec(In,Out,[G@exp|H],T,RC,MaxRC):-var(In)|
        exec([G|ET],ET,In,Out,H,T,RC,MaxRC).
exec(ET,ET,In,Out,H,T,RC,MaxRC):-var(In)|
        exec(In,Out,H,T,RC,MaxRC).
```

The meaning of this program is pretty simple. If we come across the express goal,

we simply call the eight argument *exec*. If the express queue is empty, we simply return to the six argument *exec*.

The next thing is to realize the reflective operations. Here we consider four kinds of reflective operations, i.e., *get_rc*, *put_rc*, *get_q*, *put_q*. These can be defined as follows:

```
exec([get_rc(Max,C)|EH],ET,In,Out,H,T,RC,MaxRC):- true|
        Max := MaxRC,
        C := RC,
        RC1 := RC+1,
        exec(EH,ET,In,Out,H,T,RC1,MaxRC).
exec([put_rc(C)|EH],ET,In,Out,H,T,RC,MaxRC):- true|
        RC1 := RC+1,
        exec(EH,ET,In,Out,H,T,RC1,C).
exec([get_q(NH,NT)|EH],ET,In,Out,H,T,RC,MaxRC):- true|
        RC1 := RC+1,
        NH := H,
        NT := T,
        exec(EH,ET,In,Out,H,T,RC1,MaxRC).
exec([put_q(NH,NT)|EH],ET,In,Out,H,T,RC,MaxRC):- true|
        RC1 := RC+1,
        exec(EH,ET,In,Out,NH,NT,RC1,MaxRC).
```

The *get_rc* gets *MaxRC* and *RC* from *exec*. The *put_rc* resets the *MaxRC* of *exec* to the given argument. The *get_q* gets the current scheduling queue of *exec*. The *put_q* resets the current scheduling queue to the given argument.

Here, we show an example which uses these reflective operations. This example shows how to define *check_rc* predicate which checks the current reduction count of the system and changes it if allowed less than 100 reductions.

```
check_rc :- true |
        get_rc(MaxRC,RC),
        RestRC := MaxRC-RC,
        check(MaxRC,RestRC).

check(MaxRC,RestRC) :- 100>=RestRC |
        get_q(H,T),
        input([reduction_increment,@],AddRC),
        NRC := MaxRC+AddRC,
        put_rc(NRC),
        T=[check_rc@exp|NT],
        put_q(H,NT).
check(MaxRC,RestRC) :- 100<RestRC |
        get_q(H,T),
        T=[check_rc@exp|NT],
        put_q(H,NT).
```

In a sense these *reflective* operations are very dangerous because we can freely access and change the internal state of the system. However, we can say that at least privileged user must have these capabilities for advanced system control.

6. Concluding remarks

In this paper, we discussed various factors in the programming system and showed an example of such systems. *Reflective* operations in the programming system are also considered.

You may notice these program fragments are quite simple and declarative. It seems that simpleness mainly comes from the extensive use of streams, processes and meta-interpretation techniques. The clear separation of the meta from the object also makes the system very clean. We must note that program fragments shown here are the extremely simplified version and the more complete version workable on DEC 2065 is available from the author.

We would like to answer the question why we need to make research in the programming system now? ... Looking back on the history of parallel logic programming, we decided the language specification first. Then we made interpreters and compilers. Now we have started for programming environments and applications for those languages.

It seems to be clear that we can program easily in GHC. And techniques for parallel problem solving such as in [Ohwada 87] are very similar to that of parallel programming systems. It seems to us that the research in the parallel programming system leads to that in parallel problem solving.

7. Acknowledgments

This research has been carried out as a part of the Fifth Generation Computer Project. Thanks to Yukiko Ohta, Fujitsu Social Science Laboratory for her useful comments. Actually part of this research is indebted to her. Also thanks to Koichi Furukawa, Deputy Director of ICOT, for his encouragement and giving us the opportunity to pursue this research.

References

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

[Clark 84] K.Clark and S.Gregory, Notes on Systems Programming in Parlog, in Proceedings of the International Conference on Fifth Generation Computer Systems 1984, pp.299-306, ICOT, 1984

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

[Clark 87] K.Clark and I.Foster, A Declarative Environment for Concurrent Logic Programming, Lecture Notes in Computer Science 250, TAPSOFT'87, pp.212-242, 1987

[Foster 86] I.Foster, The Parlog Programming System (PPS), Version 0.2, Imperial College of Science and Technology, 1986

[Foster 87] I.Foster, Logic Operating Systems; Design Issues, in Proceedings of the Fourth International Conference on Logic Programming, Vol.2, pp.910-926, MIT Press, May 1987

[Ohwada 87] H.Ohwada and F.Mizoguchi, Managing Search in Parallel Logic Programming, in Proceedings of the Logic Programming Conference '87, pp.213-222, ICOT, June 1987

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

[Silverman 86] W.Silverman et al., The Logix System User Manual, Version 1.21, Weizmann Institute, Israel, July 1986

[Smith 84] B.C.Smith, Reflection and Semantics in Lisp, in Proc. of 11th POPL, Salt Lake City, Utah, pp.23-35, 1984

[Takagi 84] S.Takagi et al., Overall design of SIMPOS, in Proc. of the Second International Logic Programming Conference, Uppsala, Sweden, July 1984, pp.1-12

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