Prolog Shell ---- Prolog with Modality

Masami Hagiya
Research Institute for Mathematical Sciences
Kyoto University

ABSTRACT

A shell called Prolog Shell is presented, which, in addition to using Prolog for the query part of the command language, treats the usual UNIX commands as modal operators. By doing so, it can keep the shell environment in a consistent manner and can process the user's imperatives or subjunctives as well as his queries. As a result, a description file of the UNIX make command can be programmed as a set of modal clauses.

1. Introduction

Shell is the command interpreter of the UNIX Time Sharing System [6]. Beginning with the shell of UNIX Version Six (V6), the first public version of UNIX, various shells were developed by many people, each of which had, of course, its own advantageous and characteristic features. The so called Bourne Shell [1] of UNIX Version Seven (V7) was the direct descendant of V6 Shell and allowed the users to write command sequences in the style of structured programming. C Shell [5] of the Berkley Version of UNIX, in addition to providing the control structures similar to those of the C language, has the very powerful job control mechanism, the aliases, the history mechanism etc. Recently, a shell called vsh, i.e. Visual shell, is now beginning to be used, which is a display oriented, menu-driven command invoker. Besides those explicitly quoted above, there should be an uncountable number of 'local' shells implemented in specific sites and used for special purposes.

It is no doubt that the effectiveness of the shell greatly affects the productivity of program development on UNIX, since the shell is the direct human interface to the computer. It is said that the name of 'shell' is chosen, because it surrounds the kernel of the operating system and the usual user can only see the outermost 'shell'.

Of course, the recognition of the importance of the command language is not specific to the UNIX System, but seems to be the general paradigm of the current researches on the programming environment. It also seems that there is a trend to merge the programming language and the command language, the typical example of which is C Shell above, and this is, of course, from the intent of raising the user's friendliness to the command language, and also of equipping the command language with the full power of conventional programming languages.

In this short note, we report a shell called Prolog Shell, which is abbreviated as psh. Prolog is a programming language based on logic programming originated by R. A. Kowalski. Needless to say, the first implementation of Prolog was carried out in the University of Marseille and the name of 'Prolog' was also coined then. The currently most authorized version of Prolog is the DEC-10 implementation of Prolog, known as DEC-10 Prolog [7], implemented in the University of Edinburgh.

It is very natural to use Prolog for the query part of the command language, e.g. for listing file directories, for finding files with specific properties, for querying who is now doing what, etc. But the essence of the interaction between the user and the computer lies in the user's modification of the state of the computer by the invocation of commands and user programs. This is similar to saying that the human conversation consists of not only queries, but also imperatives, and sometimes even subjunctives. To extend the formalism of logic programming to cope with such kind of 'mood' seems to be a new challenge for the further development of logic programming.

In the system reported here, we tried to extend the logic programming by adding modality, regarding commands as
modal operators. This is inspired by dynamic logic (or process logic) of V. R. Pratt and D. Harel [5], which is a logical device for verifying the correctness of programs. In our formalism with modality, the invocations of commands are completely separated from the proof procedure of Prolog unlike the usual treatment of the extralogical features in Prolog, where side-effects are performed by the selection of an atom from a clause in the course of the usual refutation. The imperatives and the queries are clearly distinguished as distinct input modes in our system.

Another problem for using Prolog or 'logic' as a command language is concerned with how to treat the environment of the user. The typical element of the UNIX shell environment is the current directory, the value of which can be altered by the change directory (cd) command. Related to this is the problem of the unification between files; e.g. if the current directory is /usr/masami, then the terms test.c and /usr/masami/test.c denote the same file and so they should be unified with each other, but after changing the current directory to, say, the parent directory /usr, test.c and /usr/masami/test.c should no longer unify with each other. This example also suggests that the system should have some mechanism of treating 'tense', which, as is expected, is realized by the use of modality in our system.

Now let us give the flavor of the current version of Prolog. Since it includes (Pure) Prolog as its subsystem, it can use the full capacity of Prolog for queries; e.g. the find command of UNIX can be easily implemented by several clauses. As for the imperative part, the system is designed so that it can play the role of the make command [2] of UNIX.

2. Query

Under is a built-in predicate of the system; under(\d, *x) means that the file *x is under the directory \d. A variable beginning with an asterisk (*) is called a file variable and matches with a simple file name (which does not contain slashes (/)), or a fragment of a simple file name. A variable beginning with a tilde (~) is called a path variable and matches with an arbitrary path name of UNIX. Now suppose that the current directory is /usr/masami and there are files named include.h, error.c and main.c under the current directory. The query

\under(\d, *x)\?

asks the system if there is a file of the form *x.c under the current direc-
tory (.). The system replies with

'*x = 'error

(Ignore the quotation marks (')) now.) When all the solutions are needed, the query

\under(\d, *x)\?

suffices, to which the system replies with

'*x = 'error
'*x = 'main
That's all.

If you want to know the C source files under the directory /usr/hagiya, then

\under(\d, *x)\?

will bind to *x the names of the C source files under /usr/hagiya with .c stripped off. Since the current directory is /usr/masami,

\under(\d, *x)\?

works as well. The variable ~ is an anonymous path variable and cannot be omitted here, because the searched directory is not the current one. Note that

\under(\d, *x)\?

does not work, since it is equivalent to

\under(\d, *x)\?

The answer to both will be Fail.

The explanation at the beginning of this section using under(\d, *x) was somewhat misleading; we should have said under(\d, *d/*x) or under(\d, *f).

'under' is distinguished from the usual evaluable predicates of Prolog in that it can backtrack i.e. it may produce multiple answers just as the ordinary predicates.

\d = \ is a built-in predicate for checking equality. To

\d = ?

the system replies with

'd = '/usr/masami

The reply to

/usr/masami/main.c = main.c?

is just

Success.
An identifier beginning with an upper case letter is called a metavariable and matches with an arbitrary term, which may or may not contain files. (Psh has three kinds of variable.)

\[ X = *.c, \text{under}(., X)?? \]

also asks the C source files under the current directory, but the answer is

\[ X = \text{error}.c \]
\[ X = \text{main}.c \]
That's all.

which is more beautiful than above. * is, or course, an anonymous file variable.

Let us define predicate `below' as the transitive closure of 'under'.

\[ \text{below}(X, Y) :- \text{under}(X, Y); \]
\[ \text{below}(X, Z) :- \text{under}(X, Y), \]
\[ \text{below}(Y, Z); \]

The defining clauses should end with two semicolons (;;). Now

\[ X = ^/\text{if}.c, \text{below}(/\text{usr}/\text{sources}, X)?? \]

will list all the C source files below /usr/sources.

As was said before, the system includes (Pure) Prolog as its subsystem. E.g. the `append' program for lists can be programmed as follows:

\[
\text{append}([], Y, Y); \\
\text{append}(\text{[A|X]}, Y, \text{[A|Z]}) :- \\
\text{append}(X, Y, Z); \\
\]

[ , , ... ] is the list constructor of psh, whose syntax is just borrowed from DEC-10 Prolog.

Many built-in predicates are prepared, with which familiar concepts can be programmed fairly easily. E.g.

\[ \text{newer}(X, Y) :- \\
\text{mtime}(X, M), \text{mtime}(Y, N), \\
M > N; \]

defines the predicate `newer', which checks if the first argument is a newer file than the second. Mtime and > are built-in; mtime(X, M) sets the modification time of the file X to M, measured from Jan 1, 1970 and counted in seconds.

### 3. Modality

Suppose that error.c and main.c both include the C include file include.h. The following clause

\[ \text{object}(\text{*x.o}, \text{*x.c}) :- \\
\text{newer}(\text{*x.o}, \text{include}.h), \\
\text{newer}(\text{*x.o}, \text{*x.c}); \]

says that *x.o is (supposed to be) the object file of the source file *x.c, if *x.o is newer than include.h and *x.c. On the other hand, to make object(*x.o, *x.c) hold, we should only issue the command cc -c *x.c. This fact is expressed in psh by the following clause

\[ \text{cc -c *x.o : [*x.o]}, \]
\[ \text{object(*x.o, *x.c)}; \]

The formula of the form

\[ \{ \ldots \} \]

is read: after ... has been successfully executed, necessarily holds. [ ... ] is the modal operator and its general form is

\[ \{ \text{command line : affected file list} \}

where, after the colon, we write a list of the files that may be modified, created or removed, on the execution of the command. The use of this list is explained later. The general form of a modal clause is

\[ \{ \ldots \} \text{conclusion :- precondition}; \]

where the conclusion and the precondition may be conjunctions. Dynamic logic uses square brackets ([ ... ]) for modal operators in connection with the boxes of modal logic; we use curly braces ({ ... }) instead of the square brackets because they are reserved for lists.

After putting

\[ \text{compiled(*x.c) :-} \]
\[ \text{object(*x.o, *x.c)}; \]

the following subjunctive query

\[ \text{cc -c main.c compiled(main.c)?} \]

is answered with

Success.

In a subjunctive query, we may omit the affected file list and the preceding colon in the modal operator. Note that in the course of processing the query, the command cc -c main.c is not actually executed.

\[ \text{compiled(main.c)} \]

is an imperative statement and says: let compiled(main.c) hold, or let main.c be compiled. The reply of the system is

\[ \{\text{cc -c main.c} \}
\[ \text{compiled(main.c)} \]

which says: oh yes, compiled(main.c) will hold, provided that cc -c main.c has been successfully executed. The command is not executed in this case, either. The system just searches for the modality [ ... ] such that the for-
mula

{ ... } compiled(main.c)

is valid. To really execute the command,

compiled(main.c)!!

is used and

!cc -c main.c

will be printed, where the command is issued after the first line and ! is put out on the completion of the command. If compiled(main.c) already holds, then the system replies with

compiled(main.c)

to the imperative with one exclamation mark (!), indicating that since compiled(main.c) already holds, there is nothing to do, and it says just

!

to the imperative with two marks (!!). In processing an imperative, the usual non-modal clauses are always applied before the application of the modal clauses.

We now give a more sophisticated example, which makes the load module 'run' by compiling error.c and main.c.

sources([error.c, main.c])!!

ready(run) :- sources(X),
    all_newer(run, X),
    all_compiled(X);;

[cc error.o main.o -o run : [run]]
ready(run) :- sources(X),
    all_compiled(X);;

all_compiled([[]]);
all_compiled([A[X]]) :-
    compiled(A), all_compiled(X);;

all_newer(A, [X]);
all_newer(A, [B|Y]) :-
    newer(A, B), all_newer(A, Y);;

Now

ready(run)!!

will make 'run' exactly in the same fashion as the following make description file (makefile):

run: error.o main.o
    cc error.o main.o -o run
main.o error.o: include.h

If nothing exists except for the sources,

ready(run)!

will be answered with

[cc -c error.c]
[cc -c main.c]
[cc error.o main.o -o run]
ready(run)

If the conclusion of a modal clause is a conjunction as in

[cc -c *x.o *y.o : [*x.o, *y.o]]
    (object(*x.o, *x.o),
     object(*y.o, *y.o));

the current system only sees the first conjunct while processing an imperative, so, in some cases, we should put a clause with the conjuncts permuted as

[cc -c *x.o *y.o : [*x.o, *y.o]]
    (object(*y.o, *y.c),
     object(*x.o, *x.c));

(This is not a good example.)

4. Database

A file name may be used independently of the actual files. Suppose there are two men named 'ono' and 'ishihata', who are both married. (We cannot capitalize their names because they would be metavariables.) This fact can be expressed as

married(ono);
married(ishihata);

One semicolon (;) denotes that the clause should be put into the database. Consider the following modal clause:

[echo *x killed : [*x]] dead(*x);

Then

dead(ono)!

will be answered with

[echo ono killed]
dead(ono)

and the output after

dead(ono)!!
is

!echo ono killed
ono killed
!

The execution of the command changes the database. Since 'ono' is in the affected file list of the modal operator, the system considers that 'ono' has changed to be a different existence than before. Actually, the query

-4-
married(ono)?
will be answered with
Fail.
and
married(*x)??
will bind to *x only 'ishihata'. Of course,
dead(X)?
is answered with
X = ono
saying that 'ono' is dead.

This database keeping mechanism can, of course, be used for actual files and will be useful for memorizing some information about files that will change along with the executions of commands. The difference between
married(ono);
and
married(ono);;
is that the clause with two semicolons is considered to be a time-independent rule that holds at every moment of the interaction, while the clause with one semicolon expresses a fact that holds at this very present.

In the current version, in order to utilize the above mechanism, we should have declared the relevant predicate as in

keep('dead');;
to tell the system that the information concerning with 'dead' should be kept in the database. This is because the execution of a command ordinarily changes the environment of the computer and the result can be 'deduced' (or inferred) by the use of appropriate build-in predicates as in the case of the predicate 'object'.

5. File Unification

The files as terms are constructed by the following syntax (we use the YACC [4] notation, since the description might be more compact):

%nonassoc ROOT
%left ROOT SLASH '/'
%nonassoc CURDIR PARDIR
%left '.'
%
file: ROOT
ROOT SLASH simple_file
PATH_VARIABLE
file '/' simple_file
simple_file ;

simple_file :
CURDIR
CURDIR
PARDIR
simple_file '
file
FILE_VARIABLE
simple_file

ROOT and ROOT SLASH stand for a slash (/). ROOT, ROOT SLASH and '/' are distinguished by the surrounding characters. CURDIR and PARDIR stand for a period (.) and a double-period (..) respectively. CURDIR and '.' are also distinguished by the surrounding characters. A CONSTANT is an appropriate identifier; it should not begin with an upper case character, with an asterisk, with a tilde etc. and should not contain periods, slashes etc. Surrounding by double-quotes (" ) will allow an arbitrary string to be a CONSTANT. The period in, say main.c, is treated as a functor, and this allows *x.c to match with main.c. The slash is also a functor in psh. Note that the period and the slash associate to the right. Since numbers are of a distinct data type in psh, they should be double-quoted when used as a file name.

If a file is used as a term, an environment is attached to the file, which, however, is invisible to the user. Thus, while

main.c = /usr/masami/main.c?
succeeds, the subjunctive query

{cd ..} main.c
= /usr/masami/main.c?
fails. The environment attached to a file consists of

* what is the current directory
* the list of files that has been affected by the execution of commands up to then

and the unifier of psh, in the course of unifying two files, always checks their attached environment. The difference between a clause that is put into the database and a clause used as a usual rule is that the environment of the
defining time is used for a database clause, while, for a rule, the environment of the invocation of the rule is always used.

If a constant is to be used just as a string of characters, and not as a file, put a quotation mark ("") before it (just as in LISP). Thus, you may say

past('went', 'go');

The quotation marks in the answers such as

'*x = 'error

are because file variables and path variables are not considered to match a file with an environment, but rather to match a file without an environment (i.e. a file name or a fragment of a file name), and in the above case, *x is considered to match a string 'error'. If we use logicians' terminology, we may say that they are 'syntactical' variables.

In a command line of a modal operator, the command name or the options should be quoted as in

{"cc '-c *x.c : [*x.o]"
  object(*x.o, *x.c);"

because they are not files. (Psh does not know that cc is /bin/cc.) Since this is so tiresome, if we have declared

keyword('cc');
keyword('-c');

the preceding quotes may be omitted in command lines.

The linking of UNIX is not counted in the file unification; even if /usr/heaven/angel and /usr/hell/devil are linked to the same file, they do not unify in the file unification.

6. Affected File List

Suppose that we have a file named gram.y which is, of course, a source file to YACC. Let us put the following clauses:

  parser(y.tab.c, *x.y) :-
  newerr(y.tab.c, *x.y);
  [yacc *x.y : [y.tab.c]]
  parser(y.tab.c, *x.y);

and suppose that a.out can be created by compiling y.tab.c:

ready(a.out) :-
  object(a.out, y.tab.c),
  parser(y.tab.c, gram.y);

object(a.out, y.tab.c) :-
  newerr(a.out, y.tab.c);
[cc y.tab.c : [a.out]]
object(a.out, y.tab.c);

(Note the order of the conjuncts in the condition of the clause of ready(a.out).) Unfortunately, in the current system,

ready(a.out)!!

does not work, because after executing

cc y.tab.c to let object(a.out, y.tab.c), the system will try to make y.tab.c by invoking yacc gram.y, but doing this will cause y.tab.c to be changed and object(a.out, y.tab.c) to no longer hold. Here the affected file list of the modal operator [yacc *x.y : [y.tab.c]] tells that y.tab.c is changed (created) after yacc gram.y. (No commands are actually executed.) For doing what we intend, we should have defined

ready(a.out) :-
  parser(y.tab.c, gram.y),
  object(a.out, y.tab.c);

Then, after putting

ready(a.out)!!

the system goes as follows:

!yacc gram.y
!cc y.tab.c
!

(provided that there is no error in gram.y). This mechanism makes psh a more consistent tool for the development of programs.

This check by the list of the affected files is performed in such a way that the system searches through the partially constructed proof tree to determine whether the affected files occur in the tree or not. So, if you had defined

[cc y.tab.c : [a.out]]
object(a.out);
ready(a.out) :- object(a.out),
  parser(y.tab.c, gram.y);

then the check would have had no meaning, and

ready(a.out)!!

would have resulted in

!cc y.tab.c
!yacc gram.y
!

which is indeed an incorrect sequencing
of commands.

In dynamic logic, the modal operator

\[ x := \text{random} \]

is equivalent to the universal quantifier

\[ \forall x \]

This means that in a formula

\[ x := \text{random} \]

the variable \( x \) is bound by the operator and \( x \) in \( \forall \) is considered to be a distinct variable from \( x \) outside the modal formula. In our formalism, the affected file list explicitly declares those variables that are bound by the modal operator.

7. Direct Execution of Commands

The usual shell will be invoked by

\[ ! \] command line

just as in ed. But if the command matches with the modal operator of some modal clause, the corresponding new fact will be put into the database (if the predicate has been declared by 'keep') and the affected files are processed and remembered by the system. So after

\[ ! \text{echo one killed} \]
\[ \text{ono}
\]

the query

\[ \text{married(ono)?} \]

will be answered with

\[ \text{Fail} \]

8. Conclusion

We first summarize the intended usage of psh as follows:

* Queries are expressed by the input clauses and processed just as in Prolog interpreter. Complicated queries may be programmed by the usual Horn clauses.

* Small and ad-hoc commands such as invoking the editor or deleting the files are invoked by the direct execution of the command.

* Tasks such as generating the overall system or systematically updating the version of programs are expressed by the modal clauses and invoked by an imperative.

Next, we point out some problems concerning with the current version of our system. The treatment of the environment is indeed a point of discussion. It may seem better to treat the two elements of the file environment independently; e.g. the current directory of a file may always be taken from the definition time even for rules.

Another problem is that of universal quantification. The imperatives such as

Let all the C source files in the current directory be compiled.

cannot be treated in the current formalism. Since such kind of imperative is indispensable in the everyday conversation with the computer, the formalism should anyway be extended. (There are at least two interpretations of the above imperative. The first one is the search for the modality in

\[ \forall x \]
\[ \text{(under(., \*x.c) ->} \]
\[ \{ ... \} \text{ compiled(*x.c))} \]

and the other is the search for the modality in

\[ \{ ... \} \forall x \]
\[ \text{(under(., \*x.c) ->} \]
\[ \text{compiled(*x.c))} \]

The author believes that the former is more tractable than the latter.)

In natural languages, the action and its resulting state are often expressed by the same word or by the conjugated forms of the same word. E.g., if we 'open' the door, then it will be 'opened'. But, in psh, if we 'co -c main.o', then main.o will be such that 'object(main.o, main.o)'. The conjugation of verbs systematically relates the action and the state, and greatly reduces the number of words of the language. We think that we should incorporate this mechanism into our language.

The sequencing operator, i.e. a semicolon (;), of the UNIX shell can be interpreted by the following schema:

\[ <== \]
\[ \{ \text{command1; command2} \} \]
\[ \{ \text{command1} \mid \text{command2} \} \]

If command1 and command2 are 'independent' from each other, the semicolon may be replaced by an ampersand (&). If an answer to an imperative is of the following form

\[ \{ \text{command1} \}
\[ \text{command1} \text{> X} \]
\[ \{ \text{command2} \text{< X} \]

and the metavariable X is unbound, then
we can merge the operators using a pipe of UNIX as in

\[ \text{command1 | command2} \]

since X can denote any file, i.e. it is universally quantified. The automatic construction of the network of processes from the answer to an imperative seems to be an interesting problem.

We think that this research is a step toward building a more intelligent programming environment, in which the environment supports the user as an intelligent adviser. There are a large number of works being done or having been done along this line, but Prolog will be one of the keys for establishing that aim, since it has an efficient and natural built-in inference mechanism. The possibility of Prolog as a program development tool, not as a programming language, is indeed of a great interest.

The extensions to logic programming performed here have not yet been studied theoretically. For the further development of the system, such as adding universal quantification discussed above, it is vital to analyze the semantics of the language more rigorously.

Acknowledgement

The author thanks to Mr. T. Sakurai of the University of Tokyo for a cherryful discussion with him and also Dr. T. Kurokawa of ICOT for reading the draft of the paper and pointing out some problems of this work.

References


