PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

WHAT IS A VARIABLE IN PROLOG?

Hideyuki Nakashima, Satoru Tomura and Kazunori Ueda[1]

Electrotechnical Laboratory
1-1-4, Umezono,
Sakura-mura, Niihari-gun, Ibaraki 305 Japan

C&C Systems Research Laboratories
NEC Corporation
4-1-1, Miyazaki, Miyamae-ku, Kawasaki 213 Japan

ABSTRACT

We review the treatment of variables in Prolog. Prolog does not have sufficient features to manipulate variables as data objects. This paper introduces two new language concepts: freezing and melting of variables in terms. Accordingly, system predicates having the freezing and melting capabilities, i.e., those for input-output and database handling, are revised and made more basic.

This revision of primitives increases Prolog's ability to handle variables, hence clauses, as data objects. As a result, a more sophisticated debugger, global logical variables, efficient meta-inference, and so on become realizable.

We present an efficient implementation for the freeze and melt operations. In particular, melting of a frozen term is achieved in constant time irrespective of its size.

1 INTRODUCTION

Prolog provides several primitive predicates to manipulate variables themselves as data objects. In the case of DEC-10 Prolog (Pereira et al. 1978), such primitives include `var`, `nonvar`, `numbervars`, ==(literal identicalness), and \==. From the viewpoint of variable manipulation as data objects, however, these primitives are neither sufficient nor well separated. We will examine this point and give a solution.

Use of Prolog in various projects has led us to identify some basic requirements for an underlying logic programming language. One project contains a natural language processing system with learning capability for new sentences. During the learning process, a human teacher must be able to tell the program the meaning of new grammatical categories. For example, if the program starts without knowing the concept of present participial, at a certain stage of the learning process, the teacher introduces it by giving an example such as:

The cat is eating.

At the time, the teacher must also indicate how to get the basic form of "*eating*", i.e., "eat". Borrowing DCG format, the new grammar may become

```
s(s(NP,T_ING)) -->
      np(NP),be,ing(ING), {trans(ING,T_ING)}.
```

[1] Order not significant.

The syntactic part

```
s(s(NP, T_ING))--> np(NP), be, ing(ING)    (1)
```

is constructed automatically by the program. What the teacher must give from a terminal is the semantic part

```
{trans(ING, T_ING)}.                       (2)
```

The problem here is that we have no means to tell the system that the variable `ING` in (2) is the same as the variable`ING` in (1). In other words, Prolog treats variables in an input term as completely different from those in the program in execution.

As a clearer example of this problem, let us consider the following assertion:

```
test:-read(V),
    p(X, Y), % X and Y become instantiated
    write(V).
```

The question is:

> Can we utilize `read` to specify which argument of `p` to be printed out?

Suppose that `p` is defined as

```
p(a, b).
```

and we type `X` for `read`. `Print` will (at best) just echo back `X`, or in DEC-10 Prolog, something like `_123`. Anyway, printing `a` is beyond hope in current Prolog systems. In Lisp, on the other hand, we can read a variable name at run time and examine its value. That is, an input atom, say `X`, is just a data object and 'eval' interprets it as the variable `X`.

In summary, Prolog lacks 'eval' and 'quote' capability.

2 THE CURRENT STATUS

DEC-10 Prolog distinguishes variables from constants when they are input from the terminal. Each variable is given special internal representation, i.e., separate positions in the stack frame. Its printed representation is something like

```
_123.
```

However, we cannot use this printed representation to designate from the terminal specific variables in the system. If we give the variable

```
_123
```

from the terminal, it is treated as an entirely new variable with its own internal representation, and might be printed as

```
_291.
```

Prolog/KR distinguishes variables from constants during unification. Internal representation of a whole clause is nothing but a list structure, and each variable holds its print name. The printer prints a variable as

```
*X_123
```

where "*" is a prefix for variables; "X" is the name of the variable; and "_123" is the unique postfix for this variable to distinguish it from other variables with the same name. However, here again, we cannot use this identifier to specify a specific variable. If the variable

```
*X_123
```

is given from the terminal, "X_123" is treated as the name of the variable and might be printed as

```
*X_123_456,
```

where "_456" is a new postfix for the variable.

Sometimes we need to treat variables in terms just as data objects. For example, when we write a pretty printer or a interpreter in Prolog itself, it is annoying if a variable in a term matches a constant in the program and become instantiated. For example, a routine for printing out a list must first filter out the "variable case":

```
printlist(X):-write('['), printtail(X).
printtail(X):-var(X), !, write('|'),
        print(X), write(']').
printtail([]):-!, write(']').
printtail([X|Y]):-!, print(X),
        write(','), printtail(Y).
printtail(X):-!, write('|'),
        print(X), write(']').
```

Without the first clause of `printtail`, the "variable case" is treated by the second clause, and a list

```
[a, b, c |X]
```

will be printed out as

```
[a, b, c].
```

Moreover, if the second and the third clauses were interchanged, the same list might be printed out as

```
[a, b, c, _1, _2, _3, ...,
```

which is a disaster.

To avoid this inconvenience, `numbervars` is sometimes used to make variables ground terms. However, since `numbervars` binds variables to constants, its effect must usually be undone afterwards by backtracking. The convention for this is to use double "not"s:

```
\+\+((numbervars(X,0,_), printlist(X))).
```

3 NEW CONCEPTS: FREEZE AND MELT

Here we propose two new language concepts: freezing and melting of terms. Freezing makes a copy of a term, inactivating all its uninstantiated variables. A frozen term looks like and can be manipulable as a ordinary ground term; the difference is that it can be "melted back", as will be described later. Variables in frozen terms are in a sense quoted so that they act just as constants.

Melting is the reactivation of the variables in a frozen term. If a frozen term is melted, the resultant term should be the same as the original term.

In current Prolog systems, freezing and melting are usually done automatically, and are not isolated as primitive actions.

The current `assert` implicitly freezes a clause before it is put into database, although there is no direct access to the frozen clause. When `assert` is called, instantiated variables in the argument clause are replaced by their value as usual. On the other hand, uninstantiated variables are frozen, and lose relation with the original variables. For example,

```
:- ..., A=a, assert(p(A,X,X)), ...  .
```

and

```
:- ..., assert(p(a,Y,Y)), ...  .
```

have exactly the same effect as long as `X` and `Y` are uninstantiated upon the execution of `assert`.

Printing a term also involves freezing. The printed representation of a variable is no more variable; it cannot be instantiated.

`Clause`, `retract` and the call of an asserted clause melts the clause but the variables do not get back the connection to the original ones. That is, a virtual copy of the clause is created and used. This feature is sometimes used to `rename` a structure (i.e. to make a new structure by systematically replacing old one's variables):

```
rename(Old,New):-assert(t(Old)),retract(t(New)).
```

`Read` also melts a term as a new one. Therefore, when the system reads back what it printed, the created term is not the same as the original one, if it contains variables. That is,

```
write(X), read(Y), X==Y.
```

does not hold even if `read` just reads back what was written by `write`. Of course,

```
write(X), read(Y), X=Y
```

and

```
write(X), read(Y), variants(X,Y)
```

will hold, where `variants` is a predicate that tests whether two terms are the same allowing the renaming of variables.

3.1 New Primitives

An example of what we want is "read without melting". If we distinguish "reading a term" from "melting

and making a virtual copy", we can solve the problem stated in Section 1, i.e., we can utilize read to specify an individual variable.

Before describing the process in detail, let us define several primitives. First, we define a new primitive for "freeze"ing a term:

```
freeze(<term>, <frozen-term>).
```

The freeze predicate freezes <term> into <frozen-term>. Variables in <frozen-term> are inactivated and are not regarded as variables (until they are reactivated by melting). Note that

```
freeze(X, Y), X==Y
```

holds if and only if X is a ground term.

Then, we define a new primitive melt:

```
melt(<frozen-term>, <new-term>).
```

The melt predicate melts <frozen-term> to get <new-term>. The value of <new-term> becomes literally identical with the original term. That is,

```
freeze(X,Y), melt(Y,Z), X==Z
```

holds.

Further, we need another kind of melt operation which melts a frozen term and makes a virtual copy, just as the current clause does. Let us name this operation melt_new. Since melt_new creates an equally-structured but new term,

```
freeze(X,Y), melt_new(Y,Z), X==Z
```

does not hold unless X is a ground term.

Since we have introduced the concept of frozen terms, we must define unification involving frozen terms. Unification of two frozen variables succeeds if and only if they are identical, that is,

```
freeze(X, Xf), freeze(Y, Yf), Xf=Yf
```

holds if and only if X==Y holds. Frozen variables are regarded just as ground atoms in unification, so the above rule is implied by the following more general one:

> Unifying two non-variable terms (possibly including frozen variables) succeeds if and only if they have the same principal functor with the same arity and each pair of their arguments are recursively unifiable. Note that unifying a frozen variable and an (normal) non-variable term causes a failure.

If a frozen term and an uninstantiated variable are unified, they become an identical frozen term. For instance,

```
freeze(f(X,Y),F), F=f(Z,Z)
```

fails because frozen X cannot be unified with frozen Y.

In order to keep primitives disjoint, we must further introduce slightly modified versions of assert, retract, clause, read, and write. We call these assert*, retract*, clause*, read*, and write* respectively. These new primitives are the same as the original ones except that they do no automatic freezing or melting. The originals would be defined as follows using new ones:

```
assert(C):-freeze(C, C2), assert*(C2).
retract(C):-rename(C, C2), retract*(C2),
        melt_new(C2, C).
clause(H, B):-rename(H, H2), clause*(H2, B2),
        clausemelt_new((H2:-B2), (H:-B)).
read(X):-read*(X2), melt_new(X2, X).
write(X):-freeze(X, X2), write*(X2).
```

where

```
rename(Old,New):-freeze(Old,X),melt_new(X,New).
```

The definition of retract and clause must use rename, because the arguments of retract* and clause* will be instantiated to frozen terms. These definitions may not be efficient since they involve term copying in freeze, as will be described in Section 5. However, in an actual implementemention, copying can be avoided by using invisible destructive assignment.

Assert*ing an unfrozen term and clause*ing or retract*ing it can be used to realize global logical variables (see Section 4.2). Applying clause* on an unfrozen term should be almost the same as clause except that doing it more than once retrieves the identical clauses. For example, suppose there is an unfrozen clause

```
p(X).
```

then,

```
clause*(p(Y),YY), clause*(p(Z),ZZ), Y=a
```

also instantiates Z to a.

Write*ing an unfrozen variable should cause an error in a sequential Prolog system (which we assume in this paper), because it has no "print name". In parallel systems such as Parlog and Concurrent Prolog, on the other hand, the printer could wait the variable to be instantiated, instead of causing an error.

3.2 Printed Representation

Using read*, the example in Section 1 can be achieved via

```
test:-write(X or Y), write('?'),
    read*(Vf), melt(Vf, V),
    p(X, Y), % X and Y may become instantiated
    write(V).
```

If the user chooses and types the first alternative, the first argument of p will be printed.

Now read* combined with melt should exactly be the reverse of write. Thus after printing a certain term, read*ing and melting it should produce the original term.

To achieve this, the printed representation must be carefully designed. The easiest solution is to assign a unique print name to each variable. Suppose a variable X is printed out as _014, then read* together with melt should map the sequence of characters " ", "_", "0", "1", "4" and " " to the original occurrence of X.

In Prolog/KR, a variable is represented as the combination of its name and a unique identifier for the variable. For example,

```
*x_123
```

is the variable *x with the identifier 123. This convention will provide better user interface.

4 APPLICATIONS

4.1 Debugger

Once the complete reversibility of input and output is achieved by read* and melt, the Prolog debugger can be improved. We can tell the debugger to bind a certain variable to a constant, or even to another variable; we can examine the current value of a (previously unbound) variable; and so on. An example follows.

Suppose we are tracing append, the current sequence would be something like:

```
|?- trace,append([a,b,c],[d],X).
(1) 0 Call : append([a,b,c],[d],_68) ?  c
(2) 1 Call : append([b,c],[d],_164) ?  c
(3) 2 Call : append([c],[d],_176) ?  c
(4) 3 Call : append([],[d],_188) ?  c
(4) 3 Exit : append([],[d],[d])
(3) 2 Exit : append([c],[d],[c,d])
(2) 1 Exit : append([b,c],[d],[b,c,d])
(1) 0 Exit : append([a,b,c],[d],[a,b,c,d])
X = [a,b,c,d]
```

We cannot, for example, examine the value of _68 at (2). However, if we could designate _68 through read* and melt, we could add another debugger command, say "e", for "e"xamining the value of a variable.

```
|?- trace,append([a,b,c],[d],X).
(1) 0 Call : append([a,b,c],[d],_68) ?  c
(2) 1 Call : append([b,c],[d],_164) ?  e _68
    Value of _68 : [a|_164] ?  c
(3) 2 Call : append([c],[d],_176) ?  e _68
    Value of _68 : [a,b|_176] ?  c
(4) 3 Call : append([],[d],_188) ?  c
(4) 3 Exit : append([],[d],[d])
(3) 2 Exit : append([c],[d],[c,d])
(2) 1 Exit : append([b,c],[d],[b,c,d])
(1) 0 Exit : append([a,b,c],[d],[a,b,c,d])
X = [a,b,c,d]
```

4.2 Global Logical Variables

In most current Prolog systems, a global database facility includes primitives such as assert and retract. However, since assert freezes its argument(s) and retract or the call of the predicate melts it as a new clause, we cannot use these predicates to manipulate data containing free variables.

Let us consider a natural-language parser trying to parse a sentence like

```
s --> np, vp.
```

Here we require the "agreement of number (i.e. singular or plural)" between NP and VP. We could pass an extra argument to carry that information as:

```
s --> np(P), vp(P).
```

But since we are usually passing a lot of information between NP and VP, it is easier to do it through a global database using assert. We assert the number agreement of NP while we are parsing NP and refer to the value while we are parsing VP. However, for some nouns, like "fish," we cannot determine whether they are singular or plural until we see the verb. In this case the information must be passed back from VP to NP. The current assert does not have this capability. Even if NP asserts

```
s_p(Unknown).
```

and VP retrieves it via,

```
s_p(Sp)
```

and later binds Sp to singular when it sees the verb "swims", this information cannot be passed back to the variable Unknown.

If we use assert* to assert s_p(Unknown), the variable Unknown is added to the database without freezing. Thus Unknown is unified with Sp when s_p(Sp) is executed in VP and can be further unified with singular when Sp is unified with singular.

In short, using assert*, we get global logical variables.

4.3 Meta-inference

Meta-inference deals with the demonstrability of a predicate from a clause set at the object-language level, which is introduced by (Bowen & Kowalski 1982) and whose utility has been shown in (Miyachi et al. 1984). To implement meta-inference, it must be possible to handle clause sets as Prolog terms and call them: the form would be

```
demo(Clause_set, Goal_to_be_proved).
```

Clause_set above may be given in the following format:

```
[(append([], X, X)),
 (append([A |X], Y, [A |Z]):-append(X, Y, Z))].
```

However, this format has the following problems.

1. The scope of variables in Clause_set must not extend outside it. In other words, a variable in a clause must

be distinguished from variables that appear outside `Clause_set` even if they have the same name.

2. Variables in the term `Clause_set` should not be instantiated by unification. So, clauses should be 'renamed' before they are selected and used for resolution.

If `Clause_set` is given in a frozen format, the first problem does not arise at all, and the `rename` operation in the second problem is replaced by the more efficient `melt` operation (See Chapter 5).

Use of a frozen form also allows a clause to have an 'undefined part' (Furukawa et al. 1984). Suppose we want to compile a generic sort module by the following predicate:

```
compile(Source, Calling_form, Object)
```

where

`Source`: source program(list of clauses)

`Calling_form`: list of the most general calling form of predicates to be called from outside

`Object`: machine-language program.

The predicate for element comparison must be left unspecified, i.e. it should be the undefined part of `Source` which will be instantiated later.

There appear two levels of variables in this example. One is the object-level variables which are compiled into target codes. The other is the meta-level variables which remain as parameters in the compiled code. They are distinguished as frozen vs. melted variables. Variables in a clause to be compiled should appear in frozen forms, while the undefined part should appear as a normal Prolog variable. The resulting object program Object will be a unary functor, which, applied to the name of a comparison predicate, becomes a complete program. For example, to sort a list using the 'standard' ordering, Object should be used in the following way:

```
Complete_object =..  [Object, '>'],
call(Complete_object), sort(L1, L2).
```

5 IMPLEMENTATION

Here is a brief description of the implementation. We assume the structure sharing method (Boyer & Moore 1972), since it enables very efficient melting. In structure sharing method, every nonvariable term is represented as the pair of a template (denoting the skeleton of a term) and an environment (a table of initially uninstantiated variables).

Freezing of a term requires scanning the whole structure and making a new template and an environment. This scanning probably cannot be avoided, since we have to identify all the occurrences of variables in the term anyway. When a term is frozen, we further mark the cell denoting the frozen term (i.e., pointing the template and its environment) with a "frozen mark".

For example, if a term

```
f(X,Y)
```

is frozen with substitution

```
{a/X, g(Z)/Y},
```

a new template

```
f(a, g(_1))
```

is created[1] with a new environment:

```
{Z/_}.
```

With the "frozen mark" on the cell, variables in the template are treated as constants by the unifier. Even if some of them get values after freezing, they are ignored.

When we melt back the frozen term with the original environment, we simply remove the mark from the cell (or more precisely, copy the cell ignoring its frozen mark). In the example above, since the variable _1 is bound to the original Z by the environment

```
{Z/_1}.
```

it restores the original value through this binding. If the variable Z has been bound to some term after freezing, the binding is now visible from the melted term.

The `melt` operation can be done in constant time as long as a frozen mark is put at the top level. A term to be melted may sometimes not have a frozen mark at its top-level, in which case the copy of the top-level template must be created and all the subterms must be recursively melted.

When we want to melt it with a new environment, on the other hand, we have to create a new environment

```
{}.
```

but it also can be done very efficiently. The actual implementation of an environment may well be a frame on the heap. In this case, a new environment can be created simply by allocating a new frame on the heap and setting each cell to 'undefined'.

6 CONCLUSION

We investigated features of the Prolog language for manipulating programs as data. We showed that certain features are not clearly distinguished in Prolog primitives, and are used implicitly in some of the system predicates. Recognizing and separating these features as new primitives in their own right should increase the expressive power of Prolog.

We also showed that an efficient implementation is possible for melting. On the other hand, the cost of freezing is proportional to the size of the term to be frozen.

[1] _1 indicates the first occurrence of a variable.

ACKNOWLEDGMENTS

The authors thank Monica Strauss, Katsushi Ikeuchi and Hiroshi Kashiwagi for reading and commenting on the earlier version of this paper. Thanks are also due to the members of ICOT WG2 for the useful discussions and suggestions.

REFERENCES

K. A. Bowen and Robert A. Kowalski, *Amalgamating Language and Metalanguage in Logic Programming,* in K. L. Clark and S. -Å. Tarnlund eds. Logic Programming, Academic Press, 1982, pp.152-172.

R. S. Boyer and J. S. Moore : *The Sharing of Structure in Theorem Proving Programs,* Machine Intelligence 7, Edinburgh Univ. Press, 1972.

K. Furukawa, S. Kunifuji, A. Takeuchi, K. Ueda, *The Conceptual Specification of The Kernel Language Version 1,* ICOT Tech. Report TR-054, Institute for New Generation Computer Technology, 1984.

ICOT, *Conceptual Specification of the Fifth Generation Kernel Language Version 1 (KL1),* Preliminary Draft, Institute for New Generation Computer Technology, 1983 (unpublished).

T. Miyachi, S. Kunifuji, H. Kitakami, K. Furukawa, A. Takeuchi, H. Yokota, *A Knowledge Assimilation Method For Logic Databases,* 1984 International symposium on logic programming, pp.118-125, 1984.

H. Nakashima, *Prolog/KR User's Manual,* METR 82-4, Dept. of Mathematical Engineering, University of Tokyo, 1982.

L. M. Pereira, F. C. N. Pereira and D. H. D. Warren, *User's guide to DECsystem-10 PROLOG,* Dept. of Artificial Intelligence, University of Edinburgh, 1978.