Term Description:
A Simple Powerful Extension to Prolog Data Structures

Hideyuki Nakashima

Electrotechnical Laboratory
Umezono 1-1-4, Ibaraki, Japan

ABSTRACT

Term description is a simple, powerful extension of terras. For example, functional notation and lazy execution of a program is introduced in a very natural manner without changing the basic mechanism of the computation, such as unification and backtracking. Especially, the readability of functional languages is introduced without actually introducing functional concepts.

X Introduction

A. Term Description

A term description is an extension to Prolog data structure. A term description is a term with some description (constraints) on it:

<term> : <descr±ption>

It means that <term> must satisfy <description> which is a predicate. In other words, whenever the term is unified with another term, the substitution must satisfy the description.

B. Motivation

Although unification in Prolog is a useful tool for manipulating structures, it lacks the ability to express complex patterns and operations over them. In Prolog, dividing a list into its first element and the rest, and constructing a list from its first element and the rest are very easy. There is no need to call a procedure to perform the operation. A simple list notation, [Carl Odr] does both of them. However, it is not so easy to divide a list into two lists or construct a single list from two lists. This operation is usually called "append" and requires a special program.

Some operations are expressed as patterns and others are expressed as procedures. This destroys the readability and coherence of program notations. The distinction is not the essential part of the programming.

Using term description, a pattern for la described as:

Z:cons(X,Y,Z)

as well as a pattern for the concatenation of two lists, X and Y:

Z:append(X,Y,Z)

The definition of "cons" is:

cons(X,Y,cons(X,Y)).

And the definition of "append" is:

append(nil,X,X).
append(axis(A,X),Y,oons(A,Z):append(X,Y,Z)).

I did not use the list notation, [X, Y], In this example on purpose. The notation is simply a syntax sugar for a term cons(X,Y). We could similarly give a syntax sugar for Z:append(X,Y,Z), eg. X::Y.

XI Semantics of Term Description

a term description, <term>:<constraints> is unified with another term T, <term> is first unified with T and then <constraints> is checked. A constraint is described as a Prolog program, and executed as if it were written at the top-level. If the execution of the constraint fails, the unification also fails.

A constraint is executed only when it is necessary, ie., only when the term description is unified with non-variable terms.

When two term descriptions are unified, only one of them is executed first. For example, when two term descriptions: X:p(X,Y) and Z:g(Z) are unified, X is unified with Z:q(Z) first*, and p(Z:q(Z),Y) is executed. Then Z:q(Z) is in turn unified with the first argument of 'p'.

A term description may be used to produce a value. For example, X:plus(l,3,X) behaves just as 4.

The term description is similar to the macro In ESP [Chikayama 83] in its effect. ESP provides two different expansion orders to distinguish value-constraining macros and value-generating macros. The same effect is achieved Implicitly in term descriptions because of its demand driven execution.

* The selection is arbitrary and implementation dependent.

III Features Provided by the Term Description

A. Typed Variables

We could type variables by adding a constraint as:

```
X:integer(X)
```

The above term description is unifiable only with integers. Hence we oould regard X as having the type integer.

B. Functional Natations

The term description is useful to simulate "functional" notations, for example, a sequence of function applications:

```
(h (g (f x)))
```

is written as:

```
W:h(Z:g(Y:f(X,Y),Z),W)
```

If we follow the convention to place the result at the last argument position, we can further introduce a special syntax:

```
!f(X)
```

which stands for

```
Y:f(X,Y).
```

Now the previous example becomes:

```
!h(!g(!f(X)))
```

This form is translated into a normal term description at read-in time. A unique variable names are attached to each pattern.

Using the notation, a function factorial is defined as:

```
factorial(0,1).
factorial(N, !times(N, !factorial( !subl(N)))).
```

C. Equality for Terms

1. Equality and Reducibility

Term descriptions introduce equality for terms in a very efficient way compared with other approaches [Kahn 81, Kbrnfeld 83]. Checking the equality is nothing more than executing a program.

Let us consider defining more than two terms equal. To assert that morning star and eveningstar in fact refer to the same object "Venus", we may say:

```
morning_star(venus).
evening_star(venus).
```

Then the three terms: "! morning-star", "!evening-star" and "venus" become unifiable. A term description !p may be thought of as an intention of p (thus ! may be regarded as an intentional operator).

Let us consider another example. What is expressed by a program such as:

```
animal(X) :-bird(X).
animal(X):-mammal(X).

bird(X):-penguin(X).
bird(X):-canary(X).
panguin(pOOl).
```

is not *equality* but reducibi;ity [Tamaki 84, Shibayama 84]. A term, !animal is reducible to bird, which is further reducible to ! penguin, which is finally reducible to pOOl. A set of reducible terms (intentions) of !animal is a super set of the set of reducible terms of Ibird.

In tne case of "morningstar" and "eveningstar", two different terms are unified through a unique individual "venus". This can be done very efficiently. In the case of "birds" on the other hand, the numbers of individual is much larger than the original terms. Therefore unifying 'animal and Ibird usually requires lots of backtracking. Further research is required here.

2. Equations

In KRC [Turner 81], equations in which the same term appear on both sides such as

```
integers = l:(addl integers)
```

are allowed*. The term description also covers this kind of equations. Since "integers" is a function with no argument, it is translated into Prolog predicate with one argument to return its value:

```
integers([1! !map(addl,!integers)]).
```

"Map" is used to apply "addl" to all the elements of a list, and defined as:

```
map(Pred, [X, Y], [IPred(X): Imap(Pred,Y)]).
```

The computation is infinite and hence we need "lazy execution."

D. Lazy Execution and Infinite Data Structure

A demand driven lazy execution is realized naturally as "lazy unification" of term descriptions. Since a variable is unifiable to any term, it is also unifiable to any term descriptions. Therefore, there is no need to execute the constraint when a term description is unified with an uninstantiated variable. The description is executed only when the result is actually necessary.

As the direct consequence of the lazy execution, indefinite data structure is manipulatable. The following example depicts the use of the infinite list in "Sieve of Eratosthenes".

The predicate "integers" produces an infinite list of integers beginning N.

```
integers(N, [M !integers( !addl(N))]).
```

Note the recursive call of "integers" itself as the term description in the second argument. If this call is moved to the body, a call for "integers" runs infinitely and never returns. When the term description is used, only the minimum part required is computed (demand driven computation).

* ":" is the concatenation operator.

The predicate "sift" filters a list of integers using "sieve". Only those which are not products of the previous elements remain in the second argument.

```
sift( [P! Rest], [P: !sift(!sieve(P,Rest))]).

"Sieve" removes those which are products of P.
sieve(P, [X:remainder(X,P,0)I Y], !sieve(P,Y)).
sieve(P, [X Y],[X2 !sieve(P,Y)]).
```

Now a call

```
integers(2,1 ),sift(I,P).
```

returns P an infinite list of prime numbers.

There are other, special purpose, primitives to deal with infinite data structures: Prolog-II [Oolmerauer 1982] has 'geler' (freeze) to manipulate infinite data structures; Par log [Clark and Gregory 1984] and Concurrent Prolog [Shapiro 1983] have read only annotations for variables which provides synchronization among processes.

IV Implementation

A subset of the term description is implemented on Uranus, a successor of Prolog/KR [Nakashima 82]. Only those which is written in functional notations are supported. A term description !p(X) is written in Uranus as [p *x].

This notation is extended to the top-level of Uranus. A user can type in a predicate call just as if it is a function. For example,

```
[+ 1 3]
```

echoes back 4. If we define primitive lisp functions as predicates, then the user can use the system just as if it were Lisp, just by using "[" and "]" instead of "(" and ")". Here are some examples:

```
[cons 1 2]                --> (1 . 2)
[car [cons 1 2]]          --> 1
[car (cons 1 2)]          --> cons
[append (1 2) (3 4)]      --> (1 2 3 4)
```

Note that we do not need "'". We can simply use "(" and ")" to denote a quoted list.

In usual, the description is replaced by the result once it is executed. Thus the multiple execution of the same description is avoided. However, in some cases, it is impossible to optimize the execution automatically. User should be careful and responsible for the efficiency.

As the final comment on implementation, it is worth noting that the implementation of lazy unification on Prolog with structure sharing is efficient. Since the form is shared, delaying the unification does not require extra storage. The storage required for saving the environment is just as large as is required for backtracking.

V. Conclusion

Prolog with term description may not be pure Prolog any more. However, the basic mechanism of the computation such as unification and backtracking are the same.

If Prolog ever needs any extension such as introducing functions, it should be kept as small as possible and that the term description is one of the smallest solutions.

ACKNOWLEDGMENTS

The author gives many thanks to Satoru Tomura and Kokichi Putatsugi at ETL, Koichi Furukawa at ICOT, Taku Takeshima at Fujitsu, Kazunori ueda at NEC, and members of ICOT WG2, especially Etsuya Shibayama, for their detailed discussions.

REFERENCES

[1] Takashi Chikayama: ESP — *Extended Self-contained PROLOG* — as a Preliminary Kernel Languages *of Fifth Generation Computers* New Generation Computing, Vol. 1, No. 1, pp.11-24 (1983)

[2] Keith L. Clark, Steve Gregory: PARLOG; Parallel *Programming in logic,* Research Report, Dept. of Computing, Imperial College (1984)

[3] Kenneth M. Kahn: *Uniform* — A Language Based upon *Unification which Unifies (Much of) LISP, Prolog and Ret 1,* IJCAI-VII, pp. 933-939, (1981)

[4] William A. Kornfeld: *Equality for Prolog,* Proc. of IJCAI-VIII, pp. 514-519 (1983)

[5] Hideyuki Nakashima: *Prolog/KR - Language Features,* Proc. of the First International Logic Programming Conference, pp. 65-70 (1982)

[6] Ehud Shapiro: A *Subset of Concurrent PROLOG and Its Interpreter,* ICOT TR-003 (1983)

[7] Etsuya Shibayama: personal communication (1984)

[8] Hisao Tamaki: *Semantics of a Logic Programming Language with a Reducibility Predicate,* Proc. of the 1984 International Symposium on Logic Programming, pp. 259-264 (1984)

[9] D. A. Turner: *The Semantic Elegance of Applicative Languages,* Proc. Conf. on Functional Programming Languages and Computer Architecture, pp. 85-96 (1981)

[10] David H. D. Warren: Higher-Order Extensions *to Prolog* - Are they *Needed?,* D.A.I. Research Paper No. 154, University of Edinburgh (1981)