Prolog

Roger Oksanen

April 12, 2004

1. Introduction

The idea behind logic programming is to describe a domain and ask questions about that domain. The domain description and the question is formed using logical statements found in mathematical logic. To make logic programming usefull, the mathematical logic is extended by adding control found in programming languages. The idea is best described by Kowalski [1]:

”algorithm = logic + control”

1.1 History

Prologs logic notation is based on first-order predicate calculus (often just called first-order logic) that dates back to Gottlob Friege's Begriffschrift ("Consept Writing" or "Conseptual Notation") and is an extension on the propostional logic (also called Boolean logic) that was a too weak language to represent commonsense knowledge from the real world [2].

First-order logic was refined to a tool for building AI systems by McCarthy (1958), then for

linguistic analysis and extended to be programming language, Prolog (PROgramming LOGic, Colmerauer et.al 1973).

2. Prolog syntax

Prolog uses clausual form semantics, also called Horn clauses (the name comes from the logician Alfred Horn who first pointed out the significance of such clauses in 1951) that has prefix notation to represent logical statements[3]. The prolog program describes a domain, which is called a **theory**. A theory is a collection of statements called **facts** and **rules**. For a program to be usefull, the program must have a task to do. In prolog , this is called the **goal** statement. Theories and goals consists terms. A term is either a **constant**, a **variable** or a **compound term**. Constants are integer terms or **atoms** such as "nil" and "=". Variables will be discussed in 2.4. A compound term comprises a **functor** and a sequence of one or more **arguments** [3]. A statement ends with a dot ".", much like pascal procedures end with the statement "End".

2.1 Facts

A fact is a minimal description of the domain. If we want to describe Finland as a republic, we might want to define the following two facts:

```
president(”Mannerheim”, 1944, 1946).
president(”Kekkonen”, 1956, 1982).
```

Here we find "president" as a functor that takes tree arguments. The programmer has decided that the first argument is the name of the President, the second is the start of his/her regime and the third is the end date of the regime. When only refering to the functor name, we talk about a **predicate**. Functor collections are polymorphic, so we can add a new fact about the current finnish president to our domain:

```
president(”Halonen”, 2000).
```

As Mrs. Halonen is still the President of Finland, there is no known end date for her period. In prolog we can't define uncertain (or "fuzzy") logic facts such as "there is a 50% chance that she won't get re-elected in 2006", "there is a 1% chance that the parlament decides that Halonen is President for life", etc.

2.2 Goals

The goal in prolog is like the main function is C-programs. Having only a theory in prolog is like only having a collection of functions in C (a library). The goal is a question that the program is supposed to answer. In an interactive prolog interpreter the question starts with an "?-", like a command prompt. If we want to know who was the President of Finland (see 2.1) between 1944 and 1946 we might ask prolog the following question:

```
?- president(”Mannerheim”, 1944, 1946).
```

Prolog will now consult its theories that we defined, and answer "yes". Asking another question:

```
?- president(”Väyrynen”, 1994, 2000).
```

Will result in a "no" answer, luckily.

Prolog lets us also use variables when asking questions. A variable is distinguished by a initial capital letter eg.

Xi Prez Value

Asking questions with variables is easy:

```
?- president(”Mannerheim”, Start, End).
```

Prolog will now tell us that ”Start = 1944” and ”End = 1946”. We can also ask the question who was elected 2000:

```
?- president(Prez, 2000).
```

And prolog will tell us that Prez=”Halonen”. Failing questions with variables will result in a ”no” result:

```
?- president(”Väyrynen”, Start, End).
No
```

When a variable value in indifferent for us, one can unse an **anonymous variable**, ”_”.

```
?- president(Prez, _, _).
Prez = ”Mannerheim” ?;
Prez = ”Kekkonen”
```

The question still didn't match ”Halonen”, because the goal functor took three arguments. I will later unify the facts in 2.3 using rules. A question statement can consist of several goals, which makes it possible to do more complex questions:

```
?- president(Prez, Start, End), Start <= 1945, End >= 1945.
Prez = ”Mannerheim”
Start=1944
End=1946
```

The comma ”,”, is interpreted as logical **AND** (Λ). Logical **OR** (v) is written as a semi-colon **”;”**.

```
?- president(Prez, Start, End)
     ,( ( Start <= 1945, End >= 1945 )
     ; ( Start <= 1979, End >= 1979 ) ).
Prez = ”Mannerheim”, Start = 1944, End = 1946 ?;
Prez = ”Kekkonen”, Start = 1956, End = 1982
```

2.3 Rules

With inference rules in prolog, there is a way to create predicates that depend on the outcome of other predicates by the rule of modus ponens. In 2.2, asking a simple question such who was president in 1945 resulted in a question with several goals. If the question would had been who was president in 2001, one would have to know that it would be the current president, with a two argument functor. The easy solution is to define a predicate depending on some other predicate:

```
president(X, Year) :- president(X, Start, End), Start <= Year,
End >= Year.
```

The rule should be read right to left. The ":-" can be read as implies "←", note the direction of the arrow. This still doesn't catch Mrs. Halonen, so a second rule is needed for the current president:

```
president(X, Year) :- president(X, Start), Start <= Year.
```

2.4 Variables

The lexical scope of a variable is restricted to the statement where it is defined. Explicit variable declaration inside a statement is done with the "is" operator. The "is" operator can be used both as an infix or prefix operator { X is 3 <=> is(X,3) }. When functors become large the need to define a temporary variable for storage raises, like in this gcd program:

```
gcd(N,N,N).
gcd(A,B,Res):-
        A>B,
        C is A-B,
        gcd(B,C,Res).
gcd(A,B,Res):- A < B, gcd(B,A,Res).
```

In the second rule, the C variable is needed to store A-B, since the gcd functor needs variables as arguments.

2.5 Inheritance

With rules, a chain of inheritance can be made. A rule can inherit the capabilities from other rules and facts. Concider the following rules in a domain:

```
mammal(X) :- animal(X), warm_blooded(X).
human(X) :- mammal(X), intelligence(X, 75).
```

A mammal is an animal that is warm_blooded. A human is a mammal with an intelligence that is 75 or higher. Now we can observe our friend George and define some facts about him:

```
warn_blooded(”George”).
animal(”George”).
intelligence(”George”, 85).
```

From this theory, we find that George is a human. He is alsoa mammal, without explictly stating it, eg. we can ask the following questions, and prolog knows the answers:

```
?-mammal(”George”).
yes
?-human(”George”).
yes
```

3. Goal matching

When prolog is matching several goals, it first tries to satisfy the first goal, then the second, and so on. The matching is short circuit, so if the first goal does not match, the the next goals won't be compared. Prolog will walk through all predicates in order and try to satisfy the goal, until there are no more predicates. When it the end of the list is

reached, the goal will fail. When having several goals, the right-hand goal will always reset at the **choice point**[4] when failing, and prolog will again try to satisfy the left-hand goal. This is called **backtracking**[3].

We tell prolog that ”Mannerheim” was the Marshal of Finland. Then we decide that that a person that is a marshal is important. To be politically correct, we also define that the current president and state minister are important persons:

```
marshal(”Mannerheim”).
state_minister(”Vanhanen”).
important(Person) :- state_minister(Person).
important(Person) :- marshal(Person).
important(Person) :- president(Person, 2004).
```

Then we ask prolog what presidents are also important persons.

```
?- president(Prez,_), important(Prez).
```

Now prolog has two goals it has to satisfy. It starts with the first functor named president that takes two arguments. It first matches the fact that Halonen is a president (we defined the president fact about Halonen before we defined the rule president).

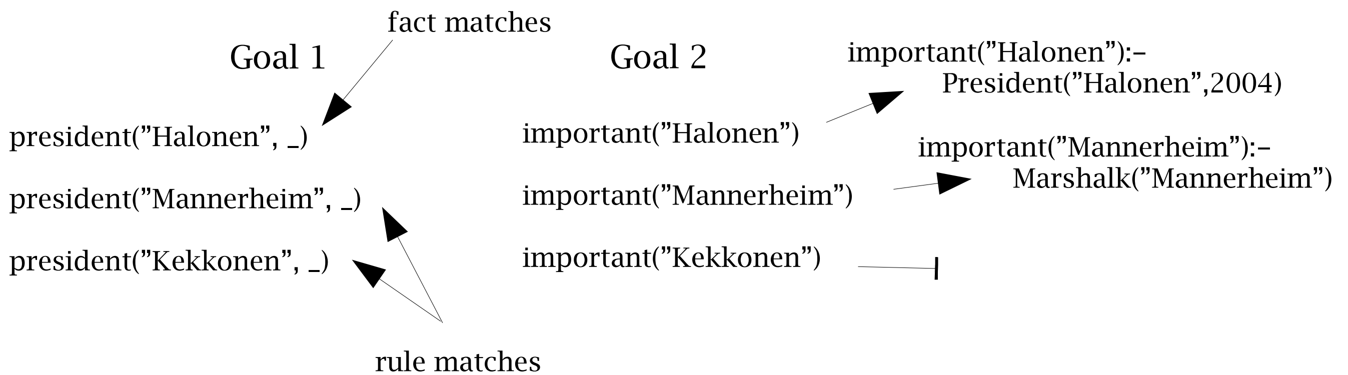
- state_minister(”Halonen”) => no
- marshal(”Halonen”) => no
- president(”Halonen”,2004) => yes, both goals satisfied, print Prez = ”Halonen”.

Prolog tries to re-satisfy the goals, but there are no more predicates left in the second goal, so it backtracks to the first goal and tries to re-satisfy the goal. Mannerheim is the second president the prolog finds (using the rule president):

- state_minister(”Mannerheim”) => no
- marshal(”Mannerheim”) => yes, both goals satisfied.
- president(”Mannerheim”, 2004) => no.

Backtrack to goal 1, try to satisfy on Kekkonen:

- state_minister(”Kekkonen”) => no
- marshal(”Kekkonen”) => no
- president(”Kekkonen”) => no



Drawing 1:Matching two goals

When prolog backtracks again to goal 1, there are no more predicates to match anymore. Prolog tells that it cannot re-satisfty the goals anymore and prints "no".

When using more goals there will be more stages, and this can lead to performance problems. Prolog provides two ways to decrease the ammount of backtracking, the cut-operator "!" and the **fail** predicate. These mechanisms not only speed up execution on some problems, without them many problems cannot be solved at all.

With "fail", we can define rules that halt the matching process. In the president domain, we might have the following rule:

```
president(Person) :- forigner(Person), fail.
```

The rule tells that if a person is a forigner, he/she cannot be a president and there is no reason to try to match any other predicate. In another domain, we might have a rule about what an average taxpayer is:

```
average_taxpayer(Person) :- lotto_winner(Person), fail.
average_taxpayer(Person) :- .....
```

If the person is a lotto winner, the first rule matches and prolog will not continue the search, even though the second line might match the goal. Fail alone won't stop us from trying to re-satisfy the goal, so cut "!" will become handy.

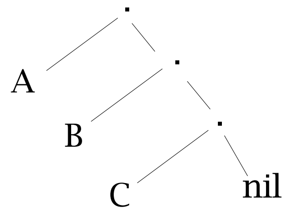
```
average_taxpayer(Person) :- lotto_winner(Person), !, fail.
average_taxpayer(Person) :- .....
```

Cut disables the backtracking and now prolog cannot re-satisfy the goal.

4. Lists

Lists in prolog are created with the same syntax as in Lisp. A list consisting of elements A, B and C is created with the "." -binary functor: .(A,.(B,.(C,nil))) or using syntactic sugar [A,B,C]. Strings are also converted into lists of **ASCII** numbers; "Mannerheim" becomes [77,97,110,110,101,114,104,101,105,109].

Using elements from the list is also done much Lisp like. Individual elements are split out recursively taking the head element away. Thus a theory to find out if a given element is in a list would look like this:



Drawing 2:The list [A,B,C]

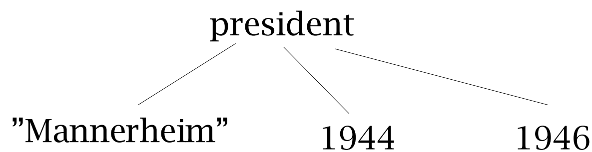
```
     % This rule matches if X is in the head of the list
member(X,[Y,_]) :- X = Y.
     % If the previous list did not match, test if X is in
     % the tail of the list (recursively)
member(X,[_,Y]) :- member(X,Y).
```

These rules enable us to ask questions such as

```
?- member(110, ”Mannerheim”).
```

And prolog will answer in a friendly manner ”yes”, as 110 is the ASCII number for 'n'.

Prolog programs are also represented as lists. The compound term ”president (Mannerheim,1944,1946)” is represented as a the list [president, ”Mannerheim”, 1944, 1946]. Prolog provides the ”=..” (univ) operator to instantiate a functor into a variable.



Drawing 3:The president functor

To use instatiated variables, the call functor is used.

```
?- F =.. [is, X, 3], call(F), Y is X+1.
F = 3 is 3
X = 3
Y = 4
```

F will be instatiated to ”X is 3” (remember that the is-operator can be used both as an infix and prefix operator). Univ can also be used to do the opposite:

```
?- is(X,3) =.. F.
F=[is, X, 3]
```

5. Prolog implementations

- Gnu Prolog
- Visual Prolog
- Mercury

References

[1] Robert Kowalski, Artificial Intelligence, Logic for problem solving, 1979.

[2] Stuart Russell, Perter Norvig, Artificial Intelligence: A modern approach 2:nd edition, Prentice Hall, 2003

[3] Applied logic - Its use and implementation as a programming tool, David H.D. Warren, 1983

[4] Hassan Ait-Kaci, Warren's Abstract Machine, The MIT press, 1991