Prolog

Wykład p.t.

Prolog

Language and terms

Antoni Ligęza

ligeza@agh.edu.pl
http://home.agh.edu.pl/~ligeza
http://homeagh.edu.pl/~ligeza/wiki

Some quoted materials:
http://www.swi-prolog.org/

Alphabet and Notation

The alphabet of PROLOG consists of atomic symbols denoting individual objects, variables, functional symbols (complex object constructors) and relational symbols (predicative symbols, predicates).

- C — a set of constant symbols (or constants, for short),
- V — a set of variable symbols (or variables, for short),
- F — a set of function (term) symbols,
- P — a set of relation (predicate) symbols.

There are also logical connectives:

- `:-` is equivalent of implication (**if**),
- `,` is equivalent of conjunction (**and**)n,
- `;` is equivalent of disjunction (**or**).

Some auxiliary symbols are: `.` is the end of a clause, and parentheses are in use.

All the sets are assumed to be countable (or finite, at least in specific applications).

Constants denote specific objects, items, elements, values, phenomena, etc. Constants names start with a lower-case letter.

Variables are used to denote the same elements in case the precise name of an element is currently not known, unimportant, or a class of elements is to be represented. Variable names start with an upper-case letter.

Functional symbols serve as complex objects constructors. Such objects have a root symbol (an element of F) and a number of arguments. They follow the tree-like structure.

The Role of Variables

The role of variables in first-order calculus is three-fold. It is worth examining the role in some details here, since it will influence design and properties of various classes of rule-based systems.

In short, variables place the role of:

- unknown but specific objects,
- place-holders, assure consistency with the arity of a functional symbol,
- coreference constraints and data carriers.

First of all, variables may be used to denote *unknown but specific objects*; some variable $X \in V$ may denote an object the properties of which are specified without specifying the object by its name. This means that a class of objects can be defined in an implicit way, or one may refer to a group of objects using universally quantified variables.

Second, any functional and predicate symbol have assigned a constant number of arguments they operate on; this is called the *arity* of a symbol. This is denoted as:

$$f/n,$$

where n is the arity of the constant number of arguments of f. Since the number of arguments cannot change — no argument can be missing. This means that if some of the arguments are unknown, variables must be used in place of specific names.

Last but not least, variables play the role of *coreference constraints* and data carriers. Two or more occurrences of the same variable in an expression denote the same object; if any replacement of an occurrence of some variable takes place, all the occurrences of this variable must be replaced with the same symbol or value. In this way data may be passed from rule input to output of the rule — a variable occurring in preconditions and conclusion of a rule will carry its value over the rule after being unified with some values during matching of preconditions against current state formula.

In the presented notation *variables* are denoted with single characters or strings, always beginning with an upper case letter or underscore, *constants* are denoted with any other strings of characters and special symbols or single letters (most typically a sequence of lower-case letters and possibly underscore characters; this convention is used in order to improve readability).

IMPORTANT: In PROLOG variables can be substituted with certain values. This means that a variable can be assigned some value or be *bound* to it. The assignment can be annulled as a result of backtracking and then a new value can be assigned to the variable. But once a value is assigned it cannot be overwritten!!! The variable must be free first.

In PROLOG there is nothing like:

```
X = X + 1
```

```
X := X + 1
```

```
X = 1,
X = 2
```

Function and Predicate Symbols

Function symbols denote, in general, mappings; however, in logic they are mostly applied to form record-like structures for representing more complex objects. In this case, one can assume that a function maps its arguments into the resulting structured object.

Predicate symbols are used to specify relations holding for certain objects. These objects are specified as the arguments of a predicate symbol.

It is assumed that for any functional symbol $f \in F$ and any predicate symbol $p \in P$ there is a unique function a defining its number of arguments (arity) of the form

$$a\colon F \longrightarrow \{1, 2, 3, \ldots\}$$

and

$$a\colon P \longrightarrow \{0, 1, 2, 3, \ldots\}$$

By convention, functional symbols of no arguments are considered to be constants. If for a certain function symbol f (predicate symbol p) the number of arguments is n, f (p) is called an n-place or n-ary function (predicate) symbol.

Functions and predicates are to be denoted with any arbitrary characters or strings; they are easily recognizable by their position in any expression. Further, proper names are frequently used so as to provide some intuitions and refer to some specific examples at hand. If necessary, indices can be occasionally used, so as to provide relatively precise definitions and theorems.

Terms in Prolog

In order to denote any object — represented by a constant, a variable, or as a result of a mapping (a structured object), the notion of *term* is introduced. The set of terms TER is defined recursively in the following manner:

Definicja 1 *The set of* terms TER *is one satisfying the following conditions:*

- *if* c *is a constant,* $c \in C$*, then* $c \in TER$*;*
- *if* X *is a variable,* $X \in V$*, then* $X \in TER$*;*
- *if* f *is an* n*-ary function symbol,* $f \in F$*, and* $t_1, t_2, \ldots, t_n$ *are terms, then* $f(t_1, t_2, \ldots, t_n) \in TER$*;*
- *all the elements of* TER *are generated only by applying the above rules.*

The definition above imposes that only the expressions belonging to one of the above categories (i.e. constants, variables, and properly constructed structured objects) are terms. Furthermore, all of the expressions satisfying one of the above conditions are terms.

Note that the definition is recursive, i.e. in order to check if a certain expression is a term one is to check if one of the above conditions holds; in case of the third possibility, the verifying procedure must be applied recursively down to all the elements $t_1, t_2, \ldots, t_n$, provided that f is an n-ary function symbol.

Assume that $a, b, c \in C$, $X, Y, Z \in V$, $f, g \in F$, and arity of f and g is 1 and 2, respectively. Then, all the following expressions are examples of terms:

- a, b, c;
- X, Y, Z;
- $f(a), f(b), f(c), f(X), f(Y), f(Z)$;
  $g(a, b), g(a, X), g(X, a), g(X, Y)$;
  $f(g(a, b)), g(X, f(X)), g(f(a), g(X, f(Z)))$.

Note that even for finite sets of constants, variables, and functions, it is possible to build an infinite set of terms. Obviously, if the set of functional symbols F is empty, the set of terms $TER = C \cup V$.

Interpretation and Applications of Terms

Terms can be used to represent various complex data structures, such as record-like objects, lists, trees, and many other. For intuition, let us show how general and flexible terms are when applied for structure construction. Lest us look at the following examples.

Consider a book as an object having title, author, publisher, place and a year of publication. Further, let the author be a man having first name and surname. A book can be represented as a complex term of the form:

```
book (book_title,
      author(first_name,last_name),
      publisher_name,
      year_of_publication
     )
```

Note that many structures used in electronic documents, mathematics, formal languages and other systems are in fact terms. For example, in XML the example concerning the specification of a book can be represented as

```
<book>
      <book_title> Learning XML </book_title>
      <author>
            <first_name> Erik </first_name>
            <last_name>  Ray  </last_name>
      </author>
      <publisher_name> O'Reilly & Associates, Inc. </publisher
      <year_of_publication> 2003 </year_of_publication>
</book>
```

where each field is declared in an explicit way by its name, beginning and end with a pair `<name> contents </name>` and the contents is either atomic value or another XML structure. Note that the internal structure of a tree is preserved.

Consider another example concerning specification of mathematical formulae. The following formula

$$\frac{\frac{x}{y}}{\sqrt{1+\frac{x}{y}}},$$

is in fact defined in LaTeX as

```
\frac{
      \frac{x}{y}
     }
     {
      \sqrt{1+\frac{x}{y}}
     }
```

where `\frac` is a two-argument symbol of division and `\sqrt` is a single argument symbol of $\sqrt{(.)}$.

Next, consider a list structure, e.g. `[red,green,blue,yellow]`. A list is constructed as an ordered pair of two elements: its *head*, being the single first element and its *tail* being the rest of the list (the definition is obviously recursive). A list as the one above can be represented as the following term

```
list(red,list(green,list(blue,list(yellow,nil))))
```

where `nil` is an arbitrary symbol denoting an empty list. Note that a list can be used to represent a set, a multi-set (or a bag — a set with repeated elements) and a sequence.

Finally, consider a binary tree, for example of depth 2; it can be represented by a term according to the following scheme

```
tree (
      tree (left_left,  left_right),
      tree (right_left, right_right)
     )
```

More complex trees can be represented with the use of lists, e.g. with a structure of the form

```
tree (root,list_of_subtrees)
```

Terms can be also used to specify graphs (e.g. as a list of nodes and another list of vertices), forests, relations, matrices, etc. In fact, expressive power of terms highly overcomes the immediate expectations following their definition. Some further examples will be presented in the part concerning PROLOG programming language.

Some Prolog Predicates: Term Types

```
var(+Term)
    Succeeds if Term currently is a free variable.

nonvar(+Term)
    Succeeds if Term currently is not a free variable.

number(+Term)
    Succeeds if Term is bound to an integer or floating
    point number.

integer(+Term)
    Succeeds if Term is bound to an integer.

float(+Term)
    Succeeds if Term is bound to a floating point number.

rational(+Term)
    Succeeds  if Term is bound to  a rational number.
    Rational  numbers
    include integers.

atom(+Term)
    Succeeds if Term is bound to an atom.

atomic(+Term)
    Succeeds  if Term is bound to  an atom, string,
    integer or  floating point number.

compound(+Term)
    Succeeds  if Term is bound to a  compound term.
    See  also functor/3 and =../2.
```

```
callable(+Term)
    Succeeds  if Term is bound to an atom or a compound term,
    so it can be handed without type-error to call/1,
    functor/3 and =../2.

ground(+Term)
    Succeeds if Term holds no free variables.

cyclic_term(+Term)
    Succeeds  if Term contains cycles,  i.e. is an  infinite t
    See also acyclic_term/1 and section 2.16.

acyclic_term(+Term)
    Succeeds  if Term  does not  contain cycles, i.e.
    can be  processed recursively   in  finite   time.
    See  also   cyclic_term/1  and section 2.16.
```

Some Prolog Predicates: Term Comparison and Unification

```
+Term1 = +Term2
    Unify Term1 with Term2.  Succeeds if the unification
    succeeds.

+Term1 \= +Term2
    Equivalent to \+Term1 = Term2.

+Term1 == +Term2
    Succeeds  if Term1  is  equivalent to  Term2.
    A  variable is  only identical to a sharing variable.

+Term1 \== +Term2
    Equivalent to \+Term1 == Term2.

unify_with_occurs_check(+Term1, +Term2)
    As  =/2, but  using sound-unification. That is,
    a variable only unifies to a term if this term
    does notcontain the variable itself.
    To illustrate this, consider the two goals below:

    1 ?- A = f(A).

    A = f(f(f(f(f(f(f(f(f(f(...))))))))))
    2 ?- unify_with_occurs_check(A, f(A)).

    No

    I.e. the first creates a cyclic-term, which is
    printed as an  infinitely nested f/1 term
    (see the max_depth option of write_term/2).
    The second executes logically sound unification
```

and thus fails.

+Term1 =@= +Term2
Succeeds if Term1 is ‘structurally equal’ to Term2. Structural equivalence is weaker than equivalence (==/2), but stronger than unification (=/2). Two terms are structurally equal if their tree representation is identical and they have the same ‘pattern’ of variables. Examples:

```
        a  =@=  A       false
        A  =@=  B       true
   x(A,A)  =@=  x(B,C)  false
   x(A,A)  =@=  x(B,B)  true
   x(A,B)  =@=  x(C,D)  true
```

The predicates =@=/2 and \=@=/2 are cycle-safe. Attributed variables are considered structurally equal iff their attributes are structurally equal.

+Term1 \=@= +Term2
Equivalent to ‘\+Term1 =@= Term2’.

+Expr1 =:= +Expr2
Succeeds when expression Expr1 evaluates to a number equal to Expr2.

term_variables(+Term, -List)
Unify List with a list of variables, each sharing with a unique variable of Term. See also term_variables/3. For example:

```
?- term_variables(a(X, b(Y, X), Z), L).

L = [G367, G366, G371]
```

```
    X = G367
    Y = G366
    Z = G371

term_variables(+Term, -List, ?Tail)
    Difference list  version of term_variables/2.
    I.e.  Tail is the tail
    of the variable-list List.
```

Some Prolog Predicates: Term Composition

```
functor(?Term, ?Functor, ?Arity)
    Succeeds  if Term is  a term with  functor
    Functor and arity  Arity. If  Term is a variable
    it is unified with  a new term holding  only
    variables.    functor/3 silently  fails on
    instantiation faults  If
    Term  is an atom or  number, Functor will
    be unified with Term  and
    arity will be unified with the integer 0 (zero).

arg(?Arg, +Term, ?Value)
    Term  should be instantiated  to a term,
    Arg to an integer  between 1  and  the  arity of  Term.
    Value  is  unified with  the  Arg-th
    argument  of Term.   Arg may also  be unbound.
    In this case  Value
    will  be unified  with the  successive arguments
    of  the term.    On successful  unification, Arg
    is unified  with the argument  number.
    Backtracking  yields alternative  solutions.
    The predicate  arg/3 fails  silently if
    Arg= 0 or Arg > arity and raises  the exception
    domain_error(not_less_then_zero, Arg)if Arg <0.

?Term =.. ?List
    List  is a list which head is the functor of
    Term and  the remaining arguments are the arguments
    of the term.  Each of the arguments may
    be  a variable,  but not  both.   This predicate
    is called  ‘Univ’.
    Examples:
```

```
    ?- foo(hello, X) =.. List.

    List = [foo, hello, X]

    ?- Term =.. [baz, foo(1)]

    Term = baz(foo(1))

term_variables(+Term, -List)
    Unify  List with  a list of  variables, each
    sharing with a  unique
    variable of Term.  See also term_variables/3.
    For example:

    ?- term_variables(a(X, b(Y, X), Z), L).

    L = [G367, G366, G371]
    X = G367
    Y = G366
    Z = G371

term_variables(+Term, -List, ?Tail)
    Difference list  version of term_variables/2.
    I.e.  Tail is the tail
    of the variable-list List.
```