5.1 Arithmetic

Freitag, 5. Juni 2015 08:30

5. The programming language Prolog

Prolog: most popular logic prog. language, developed in the 1970s by Kowalski + Colmerauer.

Essentially, Prolog uses the same syntax as the logic programs in Sect 4.

- function + pred. symbols: strings starting with lower-case letter, strings consisting of special symbols (e.g., <-->, +, <, ...), ...

  Variables: strings starting with upper-case letter or with _ (e.g., _G192). Special anonymous variable _ : its instantiation is not included in answer substitutions and several occurrences of _ can be instantiated differently.

  Ex: Prog `p(a,b,c).`

  `?- p(_,_,X).`

  Answer: X = c

- Prolog allows overloading of fct. and pred. symbols:

One may have different symbols with the same name, but different arity:

p(a,b,c).
p(a,c).

← 2 different p-symbols that have no connection

To distinguish such symbols, we often write p/3 and p/2.

- To gain efficiency, Prolog does not implement proper unification, but it does not perform the occur check.

To unify X with a term t, one has to check whether X occurs in t. In that case, X and t are not unifiable (unless X=t).

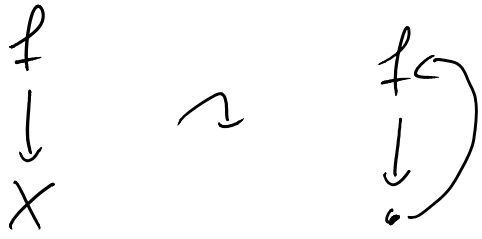
Prolog omits this check.

So far Prolog, X and f(X) are unifiable

The mgu instantiates X by f(f(....)) (infinitely many).

More precisely: X is replaced by a pointer to f(X).

⇒ we obtain



Such terms are called rational terms
(can be represented by finite graphs).

Good prog. style: avoid this problem, do not write programs where the occur check would fail, do not write programs that construct such infinite terms.

Prolog has many pre-defined predicates, including a predicate for proper unification:

unify_with_occurs_check

```
?- unify_with_occurs_check(X, f(X)).
false
?- unify_with_occurs_check(X, f(Y)).
X = f(Y)
```

5.1. Arithmetic
5.2. Lists
5.3. Operators
5.4. Cut + Negation
5.5. Input + Output
5.6. Meta-Programming

FRI: Lectures now in AH 1 (Same room as exercise course)

5.7. Parsing with Rdog

5.1 Arithmetic

All data objects have to be represented as terms.
Suitable for data structures like lists, trees, graphs, ...
Unsuitable for numbers:

One can represent $\mathbb{N}$ by the fct. symbols
$0 \in \Sigma_0$ and $s \in \Sigma_1$.

Then
$$\begin{aligned} 0 &\hat{=} 0 \\ s(0) &\hat{=} 1 \\ s(s(0)) &\hat{=} 2 \\ &\vdots \\ s^{1000}(0) &\hat{=} 1000 \end{aligned}$$

Drawbacks:
- one can't use efficient arithmetic operations of the operating system/processor
- unsuitable for large numbers

$\Rightarrow$ Prolog has built-in numbers.

Addition algorithm on user-defined numbers:

To implement a function of arity n, one needs a predicate of arity $n+1$.

add/3 : add(t_1, t_2, t_3) iff $t_1 + t_2 = t_3$

```
add(X, 0, X).
add(X, s(Y), s(Z)) :- add(X, Y, Z).
```

```
?- add(s(0), s(s(0)), X).
```
← computes 1+2

$X = s(s(s(0)))$

```
?- add(X, s(s(0)), s(s(s(0)))).
```
← computes 3-2

$X = s(0)$

The same alg. can be used for addition and subtraction
⇒ Bidirectionality

```
?- add(s(0), Y, Z).
```
← infinitely many answer substitutions

⇒ Even simple (and reasonable) programs may have an infinite SLD-tree if one uses an unfortunate query
⇒ termination depends on query

Built-in numbers in Prolog:

arithmetic expression: term built from
- numbers (0, 1, 2, ...)
- variables (X, Y, ...)

- binary infix function symbols for arithmetic:

  +, -, *, //, **, ...

  // : integer division

  ** : exponentiation

- unary negation symbol -

In principle, arithmetic expressions are ordinary terms. Most Prolog predicate symbols treat them as ordinary terms

```
equal(X,X).

?- equal(X,Y).
X=Y.

?- equal(3,1+2).
false.
```

← Reason: 3 and 1+2 cannot be unified.

There are some pre-defined Prolog predicates that evaluate arithmetic expressions. This is in contrast to ordinary logic programming where fct. symbols are never evaluated.

Pre-defined infix predicates for comparison of arithmetic expressions: <, >, =<, >=, =:=, =\=

for $\leq$ for $\geq$ for $=$ for $\neq$

For such a predicate op:

$?- t_1 \ op \ t_2 .$

Succeeds iff at the point of evaluation, t_1 and t_2 are fully instantiated arithmetic expressions and the result of evaluating t_1 is in relation "op" to the result of evaluating t_2.

- $?- 1 < 2.$
  true
- $?- 1 + 1 < 1 + 1.$
  true
- $?- 6 // 3 < 5 - 4.$
  false
- $p(1).$
  $q(2).$
  $?- p(X), q(Y), X < Y.$
  true

  this is fully instantiated when this literal is evaluated
- $?- X < 1.$

Program stops with error (X is not fully instad.)

- ?- a < 1.

  Program stops with error (a is no arithmetic expr.)

Problem: these predicates can't be used to instantiate variables

?- X =:= 1.

Will not result in answer subst X = 1,
but in a prog. error.

Thus: another pre-defined predicate "is".

?- t_1 is t_2.

succeeds iff t_2 is a fully instantiated arithmetic expression, t_2 evaluates to some number Z, and t_1 unifies with Z.

?- X is 2.

X = 2

?- X is 1+1.

X = 2

?- 2 is 1+1.

true

?- 1+1 is 2.

false

?- X is 3+4, Y is X+1.

X=7, Y=8

?- Y is X+1, X is 3+4.

prog. error

Equality predicates:

=:=	arithmetic equality, both left- and right-hand side are evaluated
is	arithmetic equality, only right-hand side is evaluated
=	syntactic unification (i.e., is defined by the only fact X=X.) (without occur check)
==	syntactic identity (no unification)

?- a=a.

true

?- 2=1+1.

false

?- 1+X = Y+1.

X=1, Y=1

?- f(1,X) = f(Y,1).

X=1, Y=1

?- X=3+4, Y is X+1.

X=3+4, Y=8

?- X==X.

true

?- X==Y.

false

Addition with built-in numbers:

```
add(X,0,X).
add(X,s(Y),s(Z)):-add(X,Y,Z).
```

} with user-def. numbers

```
add(X,0,X).
add(X,Y,Z):- Y>0, Y' is Y-1, add(X,Y',Z'),
                              Z is Z'+1.
```

?- add(1,2,X)

X = 3

?- add(X, 2, 3).

prog. error, because one reaches an is-literal where the right-hand side is not fully instantiated.

⇒ bidirectionality is lost, because the built-in arithmetic predicates are not bidirectional.

Why don't we use the following alg:

```
add(X,0,X).
add(X, Y+1, Z+1) :- add(X, Y, Z).
```

?- add(1,2,X).

false

?- add(1, 0+1, X).

X = 1+1

More arithmetic algorithms:

fact(t_1, t_2) holds iff $t_2 = t_1!$.

```
fact(0,1).
```

e.g. fact(4, 24), since $4 \cdot 3 \cdot 2 \cdot 1 = 24$

```
fact(X,Y) :- X>0, X' is X-1, fact(X',Y'),
                                Y is X*Y'.
```

```
?- fact(4,Y).
Y = 24
```

gcd (greatest common divisor):

gcd(t_1, t_2, t_3) iff t_3 is the gcd of t_1 and t_2 (for natural numbers)

```
?- gcd(28,36,Z).
Z = 4
```

```
gcd(X,0,X).
gcd(0,X,X).
gcd(X,Y,Z) :- X=<Y, X>0, Y' is Y-X,
                gcd(X,Y',Z).
gcd(X,Y,Z) :- X>Y, Y>0, X' is X-Y,
                gcd(X',Y,Z).
```

There is a pre-defined predicate number/1 to check whether the argument is a number.

```
?- number(2).
true
?- number(1+1).
false
?- X is 1+1, number(X).
X=2
```

For numbers, the built-in numbers lead to more readable and more efficient algorithms.

↑ arithmetic functions are evaluated using efficient arithmetic operations of the operating system.

For lists, there is also a pre-defined data structure to increase readability.