5. The Programming Language Prolog

developed by Kowalski + Colmerauer in the 1970s

Fixes certain syntax rules:
- `:-`, `?-`, `...`

- function and predicate symbols: strings starting with lower-case symbol, strings of special symbols (`<-`, `)`
  strings in quotes `'X'`

- variables: strings starting with upper-case symbols or with `(_G 192)`

- special anonymous variable `_` ("don’t care")
  - each occurrence of `_` can be instantiated differently
  - instantiations of `_` are not contained in the answer subset.

Ex. Prolog contains the fact p(a, b, c).

?- p (_, _, X).

X = c

- Prolog allows overloading of pred/fct. symbols.

p(a, b, c). p/3 2 unrelated
p(d, e). p/2 pred. symbols
p one often writes pred/fct. symbols
in this way to distinguish pred./fct. symbols of different entries.

- Prolog uses a variant of unification without occur check to improve efficiency.

\[
equal(X, X).
\]

\[
?- equal(f(Y), g(Y)).
\]

\[
\text{false}
\]

\[
?- equal(f(0), f(Y)).
\]

\[
Y = 0
\]

\[
?- equal(Y, f(Y)).
\]

\[
Y = f(Y)
\]

\[
\text{class failure}
\]

\[
\text{succeeds in Prolog although one should have occur failure}
\]

\[
\text{Solutions: } Y/\text{f(f(...))}
\]

\[
\text{Infinite term}
\]

In general, one should avoid Prolog programs where possible occur failures could happen and where infinite terms are constructed.

Prolog has a pre-defined predicate for proper unification:

\[
?- \text{unify_with_occur_check}(Y, f(Y)).
\]

\[
\text{false}
\]

Now, we will introduce several features of Prolog
At least second pure logic programming.

Sect 5.1 + 5.2: typical pre-defined data types

5.1. Arithmetic

Prolog has no data types, but it only operates on terms.

⇒ Data objects have to be represented by terms.

Ex: IN can be represented by terms over 0 ∈ \(\Sigma\)

and 1 ∈ \(\Sigma_1\):

\[
\text{add}(X, 0, X) .
\]

\[
\text{add}(X, s(Y), s(Z)) :- \text{add}(X, Y, Z).
\]

Disadvantage: inefficient and hard to read

\[
1000 \equiv s(s(\ldots s(0))\ldots)
\]

1000 times

Prolog has built-in arithmetic that allows us to write

numbers as usual and to use efficient arithmetic operations

provided by the operating system.

Arithmetic expression: term built from numbers, variables,

binary infix symbols \(+,-,*,//,\times,\ldots\)

unary negation \(-\)

integer exponentiation \(\uparrow\)

division

In principle, these are terms as usual.

\[
\text{equal}(X, X).
\]

\[
? - \text{equal}(3, 1+2).
\]

false

\(\vdash -, +, \cdots\) are syntactic
false
\( ? = \text{equal} (Y, 1+2). \)
\( Y = 1+2. \)

are syntactic
fact symbols
that are not
evaluated in
syntactic unification.

There exist special pre-defined predicates which evalu-
ate arithmetic expressions: \(<, >, =, =, =, \)
\( =, \neq \)
equality
non-equality

For an operation \( \text{op} \) like this:
\( ? = t_1 \text{ op } t_2. \)

When evaluating this query, \( t_1 \) and \( t_2 \) must be
fully instantiated arithmetic expressions. Then
the pre-defined facts \( +, -, \ldots \) are evaluated
and the boolean result of the comparison
determines whether the query succeeds.

\( ? = 1 < 2. \)   \( ? = 1 \times 1 < 1 + 1. \)
true     true

\( ? = -2 > 1. \)
false

\( ? = X > 1. \)    \( ? = \text{monika} > 1. \)
error    error
These predicates cannot be used to instantiate variables:

?- X := 2.
true

Therefore, there is another predicate is/2:

?- t₁ is t₂.

When evaluating the query, \( t₂ \) must be a fully instantiated arithmetic expression. Afterwards, \( t₁ \) is unified with the result of evaluating \( t₂ \):

?- 2 is 1+1.
true

?- 1+1 is 2.
false

?- X is 1+1.

X = 2
false

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

X = 7, Y = 8

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

true

Prolog has several predicates for equality:

- \( =:= \) arithmetic equality where both arguments are evaluated
• is equality, where the right argument
  is evaluated and afterwards one
  performs unification

  • unification (corresponds to
    equal \((X, X)\). )

  without occur check

? - monika = monika
true

? - \(f(X) = f(\varphi)\).
\(X = \varphi\).

?= \(X = 1+1\).
\(X = 1+1\)  \(\text{false}\)

?= \(\varphi + 1 = 2\).
\(X = \varphi + 1\)

• unify_with_occurs_check

• syntactic equality

? - monika \(\equiv\) monika
true

? - \(f(X) \equiv f(Y)\).
false

Computing with arithmetic:
add \((X, 0, X)\).

without
built-in
add(X, 0, X).
add(X, s(Y), s(Z)) :- add(X, Y, Z).

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

Disadvantage: ?- add'(1, 2, X).
false
?- add'(1, 0+1, X). 
X = 1+1

Better:
add"(X, 0, X).
add"(X, Y, Z) :- Y>0, Y1 is Y-1,
                  add"(X, Y1, Z1), Z is Z1+1.

?- add"(1, 2, X).  
?- add"(X, 2, 3).
X=3  
error

because at some point one
has to evaluate an is-literal
where the right argument is
not fully instantiated
\[ \Rightarrow \text{We lose bidirectionality.} \]

Easier: \( \text{add}(X, Y, Z) : -Z \text{ is } X+Y. \)

and much more efficient.

Ex: \( \text{gcd on natural numbers} \)

\[
\text{gcd}(X, 0, X).
\]

\[
\text{gcd}(0, X, X).
\]

\[
\text{gcd}(X, Y, Z) : -X = Y, X > 0, Y > 0, X > Y, \]

\[
\text{gcd}(X, Y, Z).
\]

\[
\text{gcd}(X, Y, Z) : -X \geq Y, Y > 0, X > Y, \]

\[
\text{gcd}(X, Y, Z).
\]

\[ ?= \text{gcd}(28, 36, X). \]

\[ X = 4 \]

Again, this is not bidirectional. To implement bidirectional arithmetic programs: Constraint Logic Programming (CLP)

Pre-defined pred. \( \text{number/1} \) is true if
the argument is a number when the pred. is evaluated:

\[ ? \text{- number (2)}. \quad ? \text{- number (N+1)}. \]
true \quad false

\[ ? \text{- X is N+1, number (X)}. \]
X=2

\[ ? \text{- number (X)}. \]
false