Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer

[The art of Prolog: 3.3, 3.4; Learn Prolog Now!: 5.1 - 5.3]

Composition of Prolog programs

- Merging the procedural and declarative perspective
- Top-down design of logic programs

Procedural vs. declarative

- Program understanding is typically procedural

- Outside of computer programs, our thinking tends to be declarative
    - Definitions of meaning
    - Descriptions

Design recipe for logic programs

- Mixing the procedural and declarative view
  - Set up procedurally
    - Construction is easier this way
    - Focus on one use case
  - Interpret declaratively (control)
    - What other uses are there?
    - Does the program generally make sense?

Design recipe: Case study

- Removing an element from a list
- Procedure: `delete(L1, X, L2)`
- Two input parameters
  - The original list (L1)
  - The element to be removed (X)
- An output parameter
  - The results list (L2)
- Meaning: All basic instances in which L2 is the list L1 without all X
- Example application:

```
?- delete([a, b, c, b], b, L2).
```

  - Where the answer `L2 = [a, c]` is

Procedural thinking

Design recipe: Case study

- Composition of the recursive procedure
- Structural recursion via L1 (input parameter)
  - We write for L1: `[X | Xs]`
- Two recursive cases
  - The head X is the element to be removed
  - The head X is not the element to be removed
- A base case

Procedural thinking

Design recipe: Case study

- Case 1: Header X is the element to be removed
  - Result: recursive removal of the element from from the rest of the input list
  - `delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).`
- Case 2: Header X is not the element to be element to be removed
  - Result: List with X as header and rest as above
  - ```
    delete([X|Xs], Z, [X|Ys]) :-
           X \= Z, delete(Xs,Z,Ys).
    ```
- Base case:
  - The empty list no longer contains any elements
  - `delete([], X, []).`

Procedural thinking
    ```

Design recipe: Case study

- `delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).`
  - "The deletion of X from [X|Xs] is Ys,
    if the deletion of X from Xs is Ys"
  - Split variable X expresses that the header is equal to the element to be
    element to be deleted

- ```
  delete([X|Xs], Z, [X|Ys]) :-
      X \= Z, delete(Xs,Z,Ys).
  ```
  - "The deletion of Z from [X|Xs] is [X|Ys],
    if Z is different from X and
    the deletion of Z from Xs is Ys."
  - Differences between X and Z must be explicitly must be specified

Declarative thinking
  ```

Design recipe for logic programs

- Top-down approach with gradual refinement
  - Formulation of the general problem
  - Break down into sub-problems

Design recipe: Case study

- Permutation-sort
  - Finding a sorted permutation

- `sort(Xs, Ys)`: Ys contains all elements of Xs in ascending order

- ```
  sort(Xs,Ys) :-
        permutation(Xs,Ys), ordered(Ys).
  ```

- Partial problems:
  - What does it mean that Ys is a permutation of Xs?
  - What does it mean that Ys is ordered?
  ```

Design recipe: Case study

- `ordered([X]).`
- `ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).`

More on comparisons and arithmetic later

Design recipe: Case study

- Permutation
  - Non-deterministic selection of an element
  - Prepend this element to the results list
  - Remainder of the result list: Permutation of the remainder of the input list
    (without the selected element)

```
permutation(Xs, [Z|Zs]) :-
     select(Z, Xs, Ys), permutation(Ys, Zs).
permutation([], []).
```

- Selection
  - Remove an occurrence of an element from the list

```
select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :- select (X, Ys, Zs).
```

Unification

- So far:
  - Definition of the meaning of Prolog programs only for
    - Basic goals
    - Basic instances of rules
- Unification
  - Core concept of logic programming
  - "Standardization" of expressions

Terminology

- Common instance
  - A term t is a
  - Common instance of two terms t_1 and t_2
  - If there are two substitutions θ_1 and θ_2 such that
  - $t = t_1\ \theta_1$ and $t = t_2\ \theta_2$
- General instance
  - A term s is more general than a term t,
  - If t is an instance of s,
  - But s no instance of t

Terminology

- Alphabetical variant
  - A term *s* is an alphabetical variant of a term *t*,
  - If *s* is an instance of *t* and
  - *t* is an instance of *s*

- Alphabetical variant: "identical except for naming"
  - Example
  - `member(X,tree(Left,X,Right))`
  - `member(Y,tree(Left,Y,Z))`

Terminology

- Unifier
  - A substitution that
  - Makes two terms identical

- Each common instance is created by a unifier
- Each unifier creates a common instance

- Example
  - t_1 = `append([1,2,3], [3,4], List)`
  - t_2 = `append([X|Xs], Ys, [X|Zs])`

  - Unifier: `{X=1,Xs=[2,3], Ys=[3,4], List=[1|Zs]}`
  - Common instance: `append([1, 2, 3], [3, 4], [1|Zs]).`

Least common unifier

- "Least common unifier" or
  "Most general unifier" of two terms
  - A unifier of both terms
  - The unifier with the most general common instance

- If two terms can be unified, then the least common unifier is
  - clear
  - determinable

- Used in complete (abstract) interpreter to find common instance of target and rule header.

Comparisons in Prologue

(SWI)Prologue Notation

- x < y `X < Y.`
- x ≤ y `X =< Y.`
- x = y `X == Y.`
- x ≠ y `X =\= Y.`
- x ≥ y `X >= Y.`
- x > y `X >Y.`

```
?- 2 < 4.
true
```

```
?- 2 =< 4.
true
?- 4 =< 4.
true
```

```
?- 4==4.
true
```

```
?- 4\=5.
true
?- 4\=4.
false
```

Arithmetic in Prolog

Prologue notation

- 6 + 2 = 8 `8 is 6+2.`
- 6 ∗ 2 = 12 `12 is 6*2.`
- 6 - 2 = 4 `4 is 6-2.`
- 6 - 8 = - 2 `-2 is 6-8.`
- 6 ÷ 2 = 3 `3 is 6/2.`
- 7 ÷ 2 = 3 `3 is 7//2.`
- 7 mod 2 = 1 `1 is mod(7,2).`

X is 6+2.

Integer division

Arithmetic in Prolog

```
add_3_and_double(X,Y) :- Y is (X+3)*2.
?- add_3_and_double(1,A).
```

true

```
A = 8
```

Arithmetic in Prolog

- Variables on the right-hand side of **is** must be instantiated

```
add_3_and_double(X,Y) :- Y is (X+3)*2.
?- add_3_and_double(1, A).
?- add_3_and_double(A, 8).
```

OK. The variable X from the definition is instantiated to 1.

Error. The variable X from the definition is not instantiated.

- Variables must be instantiated to numbers

```
?- X = 3, X < 4.
?- X = b, X < 4.
```

OK.

Error

Arithmetic and lists

- Recursive calculation of the length of a list
  - Base case:
    - `len([],0).`
  - Structural recursion
    - `len([S|T],N) :- len(T,X), N is X+1.`

Variable S is only used in one place. We can use "_" as an "anonymous variable":

```
len([_|T],N) :- len(T,X), N is X+1.
```

```
?- len([a,b,c,d,e,[a,b],g],X).
true
X = 7
```

Accumulators in Prolog

- Calculation of the length of a list with accumulator
- Accumulator variant
  - Number of elements visited so far
- Main procedure
  - Call the auxiliary function with the appropriate initial accumulator
  - No elements visited at the beginning: initial accumulator is 0
- `leng(List,Length) :- accLen(List,0,Length).`

Accumulators in Prolog

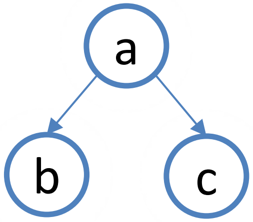
- Structural recursion
  - `accLen([_|T],A,L) :- Anew is A+1, accLen(T,Anew,L).`
  - Base case
    - `accLen([],A,A).`

Accumulator. Contains length of the list.

Same variable name for the "result".

Binary trees in Prolog

- Data structures are represented as a functor
  - `tree(Element, Left, Right)`
  - Empty tree: "void" atom



- Example

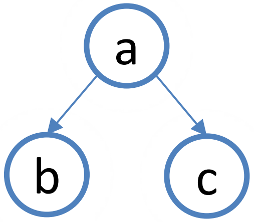
```
tree(a,tree(b,void,void),tree(c,void,void)).
```

- Type definition
  - Procedure that checks whether a value is a binary tree

```
/*binary_tree(Tree) :-
  Tree is a binary tree.*/
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :-
    binary_tree(Left), binary_tree(Right).
```

Binary trees in Prolog

- Data structures are represented as a functor
  - `tree(Element, Left, Right)`
  - Empty tree: "void" atom



- Example

```
tree(a,tree(b,void,void),tree(c,void,void)).
```

- Type definition
  - Procedure that checks whether a value is a binary tree

```
/*binary_tree(Tree) :-
  Tree is a binary tree.*/
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :-
    binary_tree(Left), binary_tree(Right).
```

double-recursive

Binary trees in Prolog

```
/* tree_ member (element, tree) ~
   Element is an element of the binary tree Tree. */
tree_member(X,tree(X,Left,Right)).
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Left).
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).

/* isotree( Treel, Tree2) :-
   Treel and Tree2 are isomorphic binary trees.*/
isotree(void,void).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Leftl,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Right2), isotree(Right1,Left2).
```

Binary trees in Prolog

```
/* tree_ member (element, tree) ~
   Element is an element of the binary tree Tree. */
tree_member(X,t                    t)).
tree_member(X,t                    t)) :- tree_member(X,Left).
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).

/* isotree( Tree1, Tree2) :-
   Tree1 and Tree2 are isomorphic binary trees.*/
isotree(void,void).
isotree(tree(X,Left1,Right1),tree(X,Left2
    isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Right2), isotree(Right1,Left2).
```

double-recursive

double-recursive

Double-recursive data structures

- Procedures for double-recursive data structures are themselves double-recursive

- Two alternative manifestations
  - Two different recursive cases (as with tree_member)
  - One (or more) recursive rules with two recursive applications of the procedure each (as with isotree)