U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

U.PORTO
FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2025/2026 – 1st Semester

Logic Programming

Unification and Execution Model

Jácome Cunha
jacome@fe.up.pt

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Unification
- Execution Model

Substitution

- Recall everything in Prolog is a term

- Terms can be either
  - **Ground** - there are no variables in the term (completely instantiated)
  - **Unground** - there are variables in the term

- Unification is how Prolog matches two terms
  - Two terms are unifiable if
    - they are the same, or
    - they can be the same after variable substitution

Substitution

- A **substitution** θ is a set of pairs $X_i = t_i$ where
  - X_i is a variable
  - t_i is a term
  - $X_i \neq X_j$ for all $i \neq j$
  - X_i does not occur in any t_j, for all i and j

- To apply a substitution θ to a term T ($T\theta$) is to replace in T all occurrences of X_i for t_i, for all pairs X_i=t_i in θ

  T = father(X, bart)

  θ = {X=homer}

  $T\theta$ = father(homer, bart)

- A is said to be an **instance** of B if there is a substitution θ such that $A = B\theta$

  `father(homer, bart)` is an instance of `father(X, bart)`

Substitution

- A term ***T*** is a **common instance** of T_1 and T_2 if there are substitutions θ_1 and θ_2 such that $T = T_1\theta_1$ and $T = T_2\theta_2$

  `parent(homer, bart)` is a common instance of `parent(X, bart)` and `parent(homer, Y)`

- A term ***G*** is **more general** than term ***T*** if ***T*** is an instance of ***G*** but ***G*** is not an instance of ***T***

  `parent(X, bart)` is more general than `parent(homer, bart)`

- A term ***V*** is a **variant** of a term ***T*** if they can be converted into one another by a simple variable renaming

  `parent(Y, bart)` is a variant of `parent(X, bart)`

Unification

- Given two atomic sentences, *p* and *q*, a unification algorithm returns a substitution θ (the most general unifier) that makes them identical (or fails if such substitution does not exist):

  Unify(p, q) = θ where $p\theta = q\theta$

- θ is said to be the (most general) unifier of the two sentences

- The **most general unifier** (MGU) is the one that compromises the variables as little as possible
  - the respective instance is the most general

    Unify(*parent(X, bart)*, *parent(Y, Z)*) produces θ = { Y=X, Z=bart }

Unification Algorithm of terms T_1 and T_2

initialize θ to empty
push T_1=T_2 into the *stack*
while *stack* is not empty do
    pop X=Y from the *stack*
    case
        X is a variable that does not occur in Y:
            substitute Y for X in the *stack* and in θ
            add X=Y to θ
        Y is a variable that does not occur in X:
            ...
        X and Y are identical constants or variables:
            continue
        X is $f(X_1, ..., X_n)$ and Y is $f(Y_1, ..., Y_n)$, for some functor f
            push X_i=Y_i, i=1...n, on the *stack*
        otherwise:
            return *failure*
return θ

Occurs check

Example

Unification of *f(X, a)* *and* *f(b, Y)*

1. Push *f(X,a)* = *f(b,Y)*
2. Pop → same functor *f*, arity 2 → push *X* = *b* and *a* = *Y*
3. Pop *a* = *Y* → *Y* is variable and doesn't occur in *a* → add *Y* = *a* to θ
4. Pop *X* = *b* → *X* is variable and doesn't occur in *b* → add *X* = *b* to θ
5. Stack empty → return θ = {*X* = *b*, *Y* = *a*}

Unification in Practice

- Both terms are constants: the terms unify if they are the same
- One of the terms is a variable: it is instantiated to the other term
- If both terms are variables, they are bound to each other
- Two compound terms unify if
  - They have the same functor and arity
  - All the corresponding arguments unify
  - All substitutions are compatible

Unification Overview

Case	Action	Example	Result
X is var not in Y	Substitute Y for X	X, a	{X = a}
Y is var not in X	Substitute X for Y	b, Y	{Y = b}
Identical	Continue	a, a	{}
Same functor/arity	Push args	f(X,a) = f(b,Y)	Stack: X=b, a=Y
Otherwise	Fail	f(a), g(a)	failure

Occurs Check

- Standard unification algorithms start with an *occurs check*
  - Verification of whether the variable occurs in the other term
  - To avoid infinite substitutions, e.g.,

```
| ?- something(X) = X.
X = something(something(something(something(something(something(something(somethin
g(something(something(something(...))))))))))) ?
yes
| ?- unify_with_occurs_check(something(X), X).
no
```

- Prolog's typical unification algorithm skips this step, to increase efficiency

- However, we can force occurs check using the predicate

  `unify_with_occurs_check/2`

Agenda

- Unification
- Execution Model

Computation

- Program P composed of Clauses
  - Clauses are universally quantified logical sentences
    - `A:- B1, ..., Bk`, k >= 0
    - `A` and `Bi` are goals

- Computation of a Logic Program P:
  - Find an instance of a given query Q logically deducible from P
  - Query is an existentially quantified conjunction
    - `A1, ..., An`, n > 0
    - `Ai` are goals
  - Goal: Atom or compound term

Computation

- Given a program **P** and an initial query **Q**
  - Computation terminates
    - With success – (an instance of) Q was proven
      - Multiple successful computations (solutions) may exist
    - Without success – Q cannot be proven
  - Computation may not terminate (no result)

- Non-termination comes from recursive rules that may not end
  - Avoid left-recursive rules

```
ancestor(X, Y) :-
    ancestor(X, Z),
    parent(Z, Y).
```

```
married(homer, marge).
...
married(X, Y):- married(Y, X).
```

Computation

- **Resolvent** is a conjunctive question (query) with the set of goals still to be processed
- **Trace** is the evolution of the computation (sequence of resolvents) with information regarding:
  - Selected goal
  - Rule selected for reduction
  - Associated substitution
- **Reduction** is the replacement, in the resolvent, of a goal G with the body of a clause whose head unifies with G

Abstract Interpreter

- Abstract interpreter algorithm, given program **P** and query **Q**

Let *resolvent* be Q

While *resolvent* is not empty **do**

1. Choose a goal A from *resolvent*
2. Choose a renamed clause B :- B_1, ..., B_n from P such that A and B unify with an MGU θ (**exit** if no such goal and clause exist)
3. Remove A from *resolvent* and add B_1, ..., B_n to resolvent
4. Apply θ to *resolvent* and to Q

If *resolvent* is empty, **return** Q; else **return** failure

MGU = most general unifier

Execution Model

- An implementation of Logic Programming needs to instantiate the abstract interpreter, making choices that influence how the computation is performed
  - Choice of goal from resolvent
  - Choice of clause
  - Add goal(s) to resolvent

- Different languages / implementations may make different choices to implement the abstract interpreter

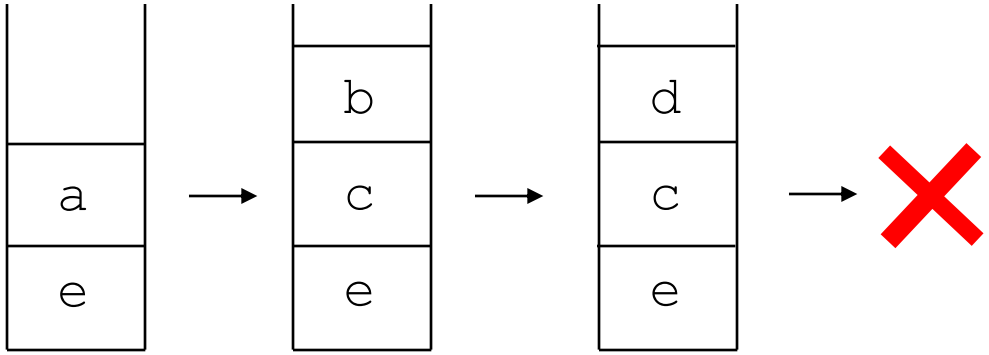
Prolog's Execution Model

- Prolog's implementation of the abstract interpreter
- Choice of goal from resolvent: **left to right**
  - Choice is arbitrary, does not affect computation (logical meaning, not operational)
- Choice of clause: **top to bottom with backtracking**
  - Choice affects computation
- Add goal(s) to resolvent: **at the beginning**
  - Results in a depth-first search
  - If it were to be added to the end, it would result in a breadth-first search (assuming leftmost goal is chosen next)

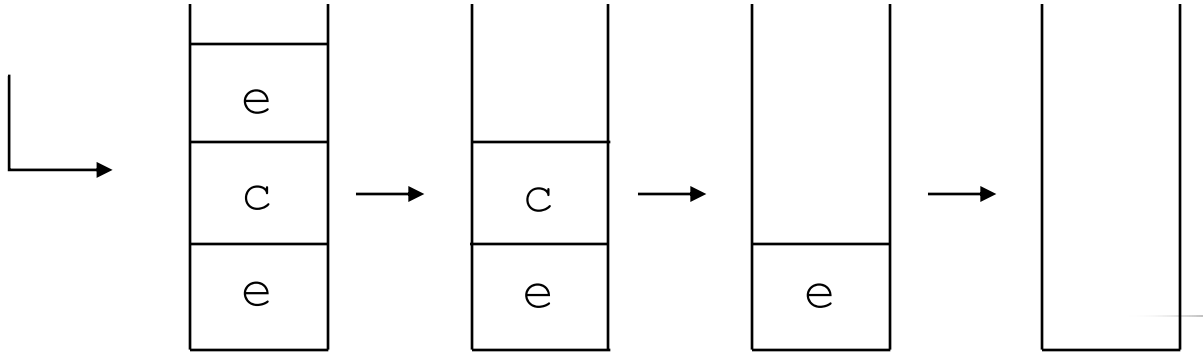
Prolog's Execution Model

- Resolvent can be seen as a stack
  - With auxiliary data (**backtracking** points)

```
a :- b, c.
b :- d.
b :- e.
c.
e.
```




```
| ?- a, e.
```



Search Trees

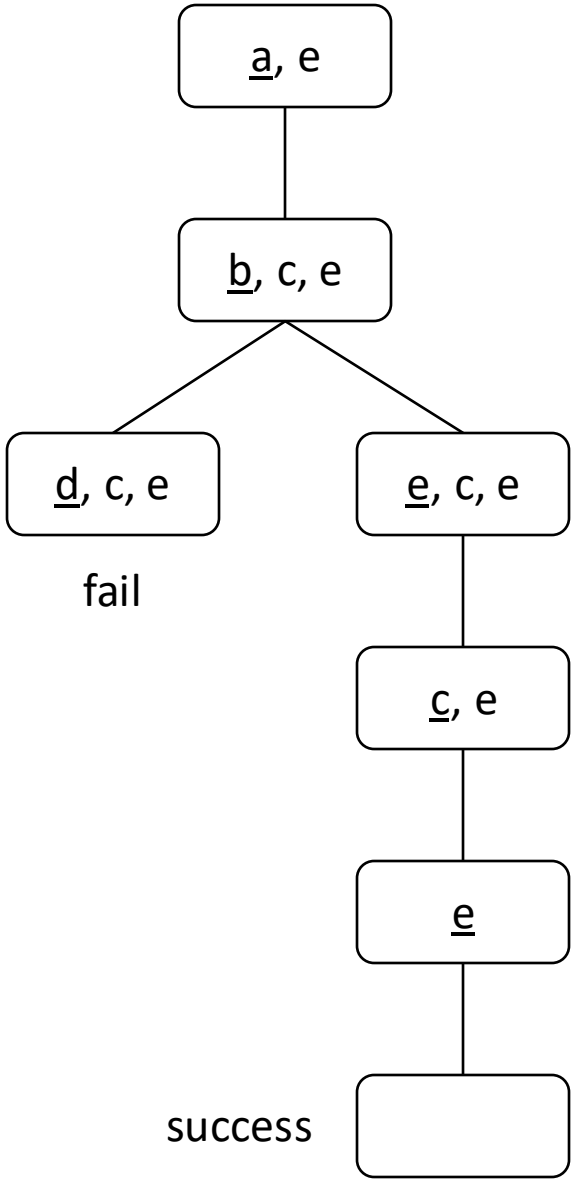
- A search tree contains all possible search paths
  - **Root**: Query Q
  - **Nodes**: resolvents, with selected goal
  - **Edges**: one edge for each clause in P whose head unifies with the selected goal in the source node
    - Includes substitution from the unification
  - **Leaves**: success nodes, if empty resolvent; or fail nodes
  - **Paths from root to leaves**: computation of Q using P

Search Trees

- Example:

```
a :- b, c.
b :- d.
b :- e.
c.
e.
```

```
| ?- a, e.
```



Search Trees

- It is independent of the **clause selection** criteria (it contains all alternatives)
- There can be different search trees for the same query and program, depending on the **goal selection** criteria
- The number of **success nodes** is the same in all trees
- Contains all answers
  - it is named **search** tree because a concrete interpreter needs a strategy to traverse the tree searching for solutions
  - Depth-first search, breadth-first search, parallel search, ...

Alternatives

- Depth-first search is not complete
  - It may not find a solution (infinite search branch)
- Breadth-first search is complete
  - If a solution exists, it is found
- OR parallelism
  - Search all branches of the search tree in parallel
- AND parallelism
  - Execute all goals of the resolvent in parallel

Clause and Goal Order

- We cannot ignore Prolog's execution model
  - Changing the order of clauses changes the order in which the search tree is traversed, and so the order in which answers are found
  - Changing the order of goals changes the search tree (may generate trees with different sizes – search effort can be different)
    - May lead to an infinite search branch!
    - If efficiency is important, different versions of a predicate may be required depending on variable instantiation

```
grandparent(X, Y):- parent(X, Z), parent(Z, Y).
grandparent(X, Y):- parent(Z, Y), parent(X, Z).
```

Goal Order

- Some heuristics can be devised, based in the principle of failing as fast as possible (failing means cutting the search tree, and thus reaching the solution faster)
  - Place tests (guards) first
  - Place goals with fewer solutions first
    - Depends on the database
  - Place goals with more ground terms first
    - Depends on the use