Deductive Databases and Logic Programming

(Summer 2017)

Chapter 3: Pure Prolog

- Prolog Syntax, Operators
- Minimal Herbrand Model
- SLD Resolution
- Four-Port/Box-Model of the Prolog Debugger

Objectives

After completing this chapter, you should be able to:

- write syntactically correct Prolog.
- use the operator syntax.
- determine the minimal Herbrand model of a given program.
- explain the immediate consequence operator T_P.
- develop an SQL-proof tree.
- understand the Prolog debugger output.

Overview

Introduction (1)

- A pure Prolog program is a set P of definite Horn clauses (clauses with exactly one positive literal).

  Prolog uses an un-sorted (or one-sorted) logic.

- A query or (proof) goal Q in Prolog is a conjunction of positive literals.

  I.e. its negation for refutation provers gives a Horn clause with only negative literals.

- The purpose of a Prolog system is to compute substitutions θ such that $P \vdash Q\,\theta$.

  I.e. one wants values for the variables such that the query is true for these values in each model of the program.

Introduction (2)

- In Prolog it is possible that the computed substitutions θ with $P \vdash Q\,\theta$ are not ground.
- E.g. consider the query $q(X)$ for the program

  $$q(X) \leftarrow p(X).$$
  $$p(X).$$

  Then it is not necessary to replace X in the query by any concrete value. The program implies $\forall X\; q(X)$.

  Then one is not interested in all substitutions with $P \vdash Q\,\theta$, but only in a set of substitutions that "subsumes" all other substitutions.
- In deductive DBs, rules and queries are restricted such that only ground answers are computed.

Introduction (3)

- While in mathematical logic, the concrete syntax is not very important (e.g. one assumes any alphabet), this chapter explains the exact Prolog syntax.
- In Chapter 5, some features will be explained that are necessary for many practical Prolog programs, but do not have a nice logical semantics.
- The classical "impure" feature is the cut, but also arithmetic predicates and I/O make Prolog semantics more complicated.

Introduction (4)

- In contrast to the examples in Chapter 1, now function symbols are permitted.
- Function symbols are supported in Prolog and some modern deductive database systems.

  Originally, function symbols are not permitted in deductive databases, because then termination of query evaluation cannot be guaranteed.
- Function symbols are interpreted as term constructors, e.g. for lists. In logic programming, one basically considers only Herbrand interpretations.

  I.e. function symbols are not interpreted ("free interpretation").

Lexical Syntax (1)

Prolog Atoms:

- Lowercase (Letter | Digit | _)*

  E.g.: thisIsAnAtom, x27, also_this_is_permitted.

- ' (arbitrary characters)* '

  If the sequence of characters contains ', one must escape it with "\", e.g. `'John\'s`. Modern Prologs support many more escape sequences starting with "\". If one needs "\" itself, one must write "\\" instead. Old Prologs used e.g. `John''s'`. Note: `'a'` and `a` are the same atom.

- `(#|$|&|*|+|-|.|/|:|<|=|>|?|@|\|^|~)`$^+$

  But: "." followed by whitespace marks the end of the clause. Another exception is "/*", which starts a comment.

- Special atoms: `!`, `;`, `[]`, `{}`.

Lexical Syntax (2)

Constants in Prolog:

- Atoms (see above): e.g. `red`, `green`, . . . , `monday`, . . .

  Atoms are internally represented as pointers to a symbol table.

- Integers: e.g. `23`, `-765`, `16'1F` (=31), `0'a` (=97)

  ⟨`Radix`⟩`'`⟨`Number`⟩ is the Edinburgh Prolog syntax. The ISO-Standard requires instead that hexadecimal numbers start with `0x`, octal numbers with `0o`, and binary numbers with `0b`. It supports `0'` for the ASCII-code. The Edinburgh Prolog syntax is probably more portable.

- Floating point numbers: e.g. `-1.23E5`.

- Strings: e.g. `"abc"`.

Lexical Syntax (3)

Strings in Prolog:

- In classical Prolog systems, strings are represented as lists of ASCII codes, e.g. `"abc"` is `[97,98,99]`.
- This makes string processing easy and flexible, but each character might need e.g. 16 bytes of storage.

  Also, `write("abc")` prints `[97,98,99]`.
- Modern Prologs often represent strings as arrays of characters (as usual in other languages).

  This creates, however, portability problems: Old programs might not run. In ECLiPSe, the conversion is done with `string_list(String, List)`. In SWI Prolog, it is `string_to_list(String, List)`.

Lexical Syntax (4)

Atoms vs. Strings:

- Atoms are internally represented as pointers into a symbol table (“dictionary”).
- Therefore comparing and copying them is very fast.
- However, creating a new atom takes some time.
- Also, once an atom is created, it is never deleted from memory (depending on the Prolog system).
- It is possible to create atoms dynamically (at run-time), but one should do this only if one expects to reference them again and again.

Lexical Syntax (5)

Predicates and Function Symbols in Prolog:

- Predicate and function symbol names are atoms.
- Prolog permits to use the same name with different arities. These are different predicates, e.g.:

```
father(Y) :- father(X,Y).
```

- In the Prolog literature, one normally writes p/n for a predicate with name p and arity n.

  Remember that the arity is the number of arguments. E.g. the above rule contains the predicate `father/1` in the head, and the predicate `father/2` in the body. There is no link between these two distinct predicates except what is explicitly specified with the rule.

Lexical Syntax (6)

Variables in Prolog:

- (Uppercase | _) (Letter | Digit | _)*
- Exception: "_" (anonymous variable): Each occurrence denotes a new system-generated variable.
- Many Prolog systems print a warning ("singleton variable") if a variable that appears only once in a rule does not start with an underscore "_".

  This helps to protect against typing errors in variable names: If the variable really appears only once, one could as well use the anonymous variable.

Lexical Syntax (7)

Comments in Prolog:

- From "`%`" to the line end (as in TeX).
- From "`/*`" to "`*/`" (as in C).

Logical Symbols in Prolog:

- "`:-`" for $\leftarrow$.
- "`,`" for $\wedge$.

Abstract Prolog Syntax (1)

- The abstract syntax describes the data structures that the parser creates (e.g. operator tree).
- The concrete syntax defines e.g. operator priorities, abbreviations, and special "syntactical sugar".

  E.g. the concrete input might contain parentheses and special delimiter characters that are not contained in the internal representation of the program.
- Prolog was originally an interpreted language.

  Today, it is typically compiled into byte code for the "WAM".
- Then the abstract syntax describes the data structures on which the interpreter works.

Abstract Prolog Syntax (2)

Program:

- A program is a sequence of clauses.

Clause:

- A clause is one of the following:
  - ◇ Fact: A literal.

    Literal means here always "positive literal".
  - ◇ Rule: Consists of a literal and a goal.

    The literal is called the head of the rule, and the goal is called the body of the rule.
  - ◇ Query/Command: A goal.

Abstract Prolog Syntax (3)

Goal (simple version, this chapter):

- A goal is a sequence of literals.

Goal (complex version, later):

- A goal is one of the following:
  - ◇ A literal.
  - ◇ A cut.
  - ◇ A conjunction of two goals.
  - ◇ A disjunction of two goals.
  - ◇ If goal, then goal, possibly else goal.

Abstract Prolog Syntax (4)

Literal (Positive Literal):

- A literal consists of
  - ◇ An atom p, and
  - ◇ n terms $t_1, \ldots, t_n$, $n \geq 0$.
- p/n is the predicate of the literal.
- t_i is the i-th argument of the literal.
- If A is a literal, let $pred(A)$ denote the predicate of A.

Abstract Prolog Syntax (5)

Term:

- A term is one of the following:
  - ◇ Variable (the anonymous variable is treated specially)
  - ◇ Atom
  - ◇ A composed term consisting of an atom f and $n \geq 1$ terms $t_1, \ldots, t_n$. (f/n is the functor of this term.)
  - ◇ Number: integer, real, possibly rationals etc.
  - ◇ String (if this is not a list of ASCII codes).
  - ◇ Stream (open file).
  - ◇ ... (possibly other types of objects).

Operator Syntax (1)

Example:

- `+(1, 1)` is a term in standard syntax.

  Standard syntax is $f(t_1, \ldots, t_n)$ for a composed term with functor f/n and arguments $t_1, \ldots, t_n$.

- `1+1` is the same term in operator syntax.

- This is only a more convenient input syntax.

  Internally, `+(1,1)` and `1+1` are the same term.

  There is absolutely no difference in their meaning.

- Operator syntax can be used also for literals, e.g.

  `X \= Y,    5 < 7.`

Operator Syntax (2)

Operators:

- Many operators are predeclared, but the Prolog user can declare new operators.

  Declaring an operator only modifies the input syntax of Prolog (in Prolog programs and user input read with the built-in predicate `read`). By itself, it does not associate any specific meaning with the operator.

- An operator has
  - ◇ Name: Any Prolog atom.
  - ◇ Priority: From 1 (high priority) to 1200 (low).

    E.g. $*$ (priority 400) binds more strongly than $+$ (priority 500).
  - ◇ Associativity: One of `fx`, `fy`, `xf`, `yf`, `xfx`, `yfx`, `xfy`.

Operator Syntax (3)

Operator Types:

- `fx`, `fy`: Prefix operator, e.g. "`-X`".
- `xf`, `yf`: Postfix operator, e.g. "`7!`".
- `xfx`, `yfx`, `xfy`: Infix operator, e.g. "`1 + 1`".

Associativity:

- `x`: term, the topmost operator of which has a numerically lower priority (which means really higher priority).
- `y`: term with numerically lower or equal priority.

Operator Syntax (4)

Example for Associativity:

- `+` has type `yfx`, i.e. another `+` can be in the left operand, but not in the right (except inside "`(...)`").
- Thus, the term `1+2+3` means `+(+(1,2),3)`.

  "`+`" is a left-associative operator.

Querying Declared Operators:

- "`current_op(Prio, Type, Operator)`" can be queried to get a list of all declared operators.

  "`current_op`" is one of many built-in predicates, i.e. predicates that are not defined by clauses, but by a procedure inside the Prolog system.

Operator Syntax (5)

Operator Declaration:

- A new operator is declared by calling/executing the built-in predicate "`op(Prio, Type, Operator)`".
- E.g. after executing the goal

  ```
  op(700, xfx, is_child_of)
  ```

  the following is a legal syntax for a fact:

  ```
  emil is_child_of birgit.
  ```

- It is completely equivalent to

  ```
  is_child_of(emil, birgit).
  ```

Operator Syntax (6)

Note:

- Besides facts and rules, a Prolog program can contain goals. One must write "`:- op(...).`"

  Unless one enters "`op(...)`" interactively, in which case the Prolog system is already in query mode. But if one should write "`op(...)`" as a fact in a file, one will probably get an error message that one tries to redefine a built-in predicate.

- The Prolog compiler executes this while compiling the program. It modifies the internal parser tables.
- One can then use the operator in the rest of the same input file and in later user input.

Operator Syntax (7)

Examples for Predefined Operators (Logic, Control):

Op.	Priority	Type	Meaning
`:-`	1200	xfx	"if" in rules
`:-`	1200	fx	marks a goal
`-->`	1200	xfx	syntax rule
`;`	1100	xfy	disjunction (or)
`->`	1050	xfy	then (for if-then-else)
`,`	1000	xfy	conjunction (and)
`\+`	900	fy	negation as failure
`=..`	700	xfx	convert term to list

Operator Syntax (8)

Operator Examples, Cont. (Arithmetic Comparisons):

Op.	Priority	Type	Meaning
`<`	700	xfx	is less than
`>`	700	xfx	is greater than
`>=`	700	xfx	greater than or equal
`=<`	700	xfx	less than or equal
`=:=`	700	xfx	is equal to
`=\=`	700	xfx	is not equal to
`is`	700	xfx	evaluate and assign

- These functions evaluate arithmetic expressions in their arguments (`is` only on the right side).

Operator Syntax (9)

Operator Examples, Cont. (Arithmetics):

Op.	Priority	Type	Meaning
`+`	500	yfx	sum
`+`	200	fx	identity (monadic +)
`-`	500	yfx	difference
`-`	200	fx	sign inversion (monadic -)
`*`	400	yfx	product
`/`	400	yfx	division (quotient)
`div`	400	yfx	integer division (floor)
`//`	400	yfx	integer division (toward zero)
`mod`	400	xfx	modulo (division rest for `div`)
`rem`	400	xfx	modulo (division rest for `//`)

Operator Syntax (10)

Operator Examples, Cont. (Bit Operations):

Op.	Priority	Type	Meaning
/\	500	yfx	bitwise and
\/	500	yfx	bitwise or
>>	400	yfx	right shift
<<	400	yfx	left shift
\	200	fx	bitwise negation

Operator Syntax (11)

Operator Examples, Cont. (Term Comparisons):

Op.	Priority	Type	Meaning
`=`	700	xfx	does unify with
`==`	700	xfx	is strictly equal to
`\==`	700	xfx	is not strictly equal to
`@<`	700	xfx	comes before
`@>`	700	xfx	comes after
`@=<`	700	xfx	comes before or is equal
`@=>`	700	xfx	comes after or is equal

Operator Syntax (12)

Restrictions in Mixed Syntax:

- In standard syntax "$f(t_1, \ldots, t_n)$", one cannot put a space between "f" and "(".

  A space is necessary, when the argument of a prefix operator starts with a parenthesis, e.g. "`\+(1) > 2`" vs. "`\+ (1) > 2`".

- In standard syntax "$f(t_1, \ldots, t_n)$", the prioity of operators in the argument terms t_i must be < 1000.

  "," is an operator with priority 1000. Use parentheses if necessary.

- If a prefix-operator is used as atom without arguments, it must be put into parentheses: `(op)`.

List Syntax (1)

- The functor "`./2`" is used as list constructor.
- The left argument is the first element of the list.
- The right argument is the rest of list.
- The atom "`[]`" is used to represent the empty list.
- E.g. the list $1, 2, 3$ can be written as

  ```
  .(1, .(2, .(3, []))).
  ```

- However, Prolog accepts the abbreviation `[1, 2, 3]` for the above term.

  It is uncommon that one ever uses "." explicitly.

List Syntax (2)

- I.e. `[`t_1`, ...,` t_n`]` is an abbreviation for

  `.(`t_1`, ..., .(`t_n`, []) ...)`

- One can also write "`[X|Y]`" for "`.(X, Y)`".
- More generally, also the abbreviation

  `[`t_1`, ...,` t_n `|` t_{n+1}`]`

  for the following term is accepted:

  `.(`t_1`, ..., .(`t_n`,` t_{n+1}`) ...)`

  I.e. after the vertical bar "|", one writes the rest of the list. Before it, the first list elements. `[1 | 2, 3]` is a syntax error. `[1|2]` is not a syntax error, but it would be a type error if Prolog were typed.

List Syntax (3)

- E.g. the following are different notations for the list $1, 2, 3$:
  - ◇ `[1, 2, 3].`
  - ◇ `.(1, .(2, .(3, []))).`
  - ◇ `[1, 2, 3 | []].`
  - ◇ `[1 | [2, 3]].`
  - ◇ `.(1, [2, 3]).`
- If one tries `write(`t`)` for each of these terms, the system will always print `[1, 2, 3].`

List Syntax (4)

- Now list processing predicates are easy to define.
- E.g. `append(X, Y, Z)` is true iff the list `Z` is the concatenation of lists `X` and `Y`, e.g.

```
append([1, 2], [3, 4], [1, 2, 3, 4])
```

- It is defined as follows (some Prolog systems have it as a built-in predicate):

```
append([], L, L).
append([F|R], L, [F|RL]) :-
        append(R, L, RL).
```

- Exercise: Define `member(X, L)`: `X` is an element of `L`.

Formal Prolog Syntax (1)

- The input is a term of priority 1200, followed by a "full stop":

  Term(1200) "."

  White space (a space, line break, etc.) must follow so that "." is recognized as "full stop".

- The term is interpreted as clause: ":-" and "," are declared as operators.

- There are certain type restrictions, e.g. the head of the clause cannot be a variable or number.

  Prolog requires that the predicate is an atom, and e.g. not a variable.

Formal Prolog Syntax (2)

Term(N):

- Operator(N,fx) Term(N-1)

  Exception: "`-1`" is a numeric constant, not a composed term. Furthermore, if "Term(N-1)" starts with "(", a space is required.

- Operator(N,fy) Term(N)
- Term(N-1) Operator(N,xfx) Term(N-1)
- Term(N-1) Operator(N,xfy) Term(N)
- Term(N) Operator(N,yfx) Term(N-1)

Formal Prolog Syntax (3)

Term(N), continued:

- Term(N-1) Operator(N,xf)
- Term(N) Operator(N,yf)
- Operator(N,fx/fy)

  Prefix-operators that are used as atom count as term of their priority, and not as term of priority 0 as other operators.

- Term(N-1)

  I.e. it is not required that Term(N) really contains an operator of priority N. It may also contain an operator of numerically lower priority (which means higher binding strength), or contain no further operators outside parentheses (elementary terms are generated by Term(0) below).

Formal Prolog Syntax (4)

Term(0):

- Atom

  The atom cannot be declared as prefix operator, see above.

- Variable
- Number
- String
- Atom “(” Arguments “)”
- “[” List “]”
- “(” Term(1200) “)”

Formal Prolog Syntax (5)

Arguments:

- Term(999)
- Term(999) “,” Arguments

List:

- Term(999)
- Term(999) “,” List
- Term(999) “|” Term(999)

More Syntax Examples

- The rule "`p :- q, r.`" can also be entered in standard syntax: `:-(p, ',' (q, r)).`
- The following are all the same literal:
  - ◇ `X is Y+1`
  - ◇ `is(X, Y+1)`
  - ◇ `is(X, +(Y,1))`
  - ◇ `X is +(Y,1)`

  I.e. one can use arbitrary mixtures of operator syntax and standard syntax and even when an atom is defined as operator, one can use the standard syntax.

Example: A Riddle (1)

- Consider the following riddle:
  - ◇ A man needs to transport a cabbage, a goat, and a wolf from the left side of a river to the right side.
  - ◇ He has only a small boat, and can take only one of the three with him when he crosses the river.
  - ◇ When the wolf and the goat remain without the man on one side of the river, the wolf will eat the goat. Also the goat wants to eat the cabbage.

Example: A Riddle (2)

- The goal is to compute a sequence of moves that ends with all three on the right side of the river.
- A state in the riddle can be encoded with terms of the form

  ```
  st(Man,Wolf,Goat,Cabbage)
  ```

  where each of the arguments is `left` or `right`.
- One possibility is to define the allowed states by the 10 facts on the next slide (extensional definition).

Example: A Riddle (3)

```
allowed(st(left,  left,  left,  left )).
allowed(st(left,  left,  left,  right)).
allowed(st(left,  left,  right, left )).
allowed(st(left,  right, left,  left )).
allowed(st(left,  right, left,  right)).
allowed(st(right, left,  right, left )).
allowed(st(right, left,  right, right)).
allowed(st(right, right, left,  right)).
allowed(st(right, right, right, left )).
allowed(st(right, right, right, right)).
```

Example: A Riddle (4)

- However, the following two rules suffice:

```
allowed(st(M,W,M,C)) :-
        pos(M), pos(W), pos(C).
allowed(st(M,M,G,M)) :-
        pos(M), pos(G), M \== G.
```

  If the goat and the man are on the same side, nothing bad can happen. If the man and the goat are on different sides, the wolf and the cabbage must be with the man. “\==” means “not equals”.

- The auxillary predicate pos is defined by

```
pos(left).
pos(right).
```

Example: A Riddle (5)

- The predicate "`move(FromState, ToState, Action)`" defines the possible moves, where the third argument is a term of the form "`act(Object,ToSide)`".

```
move(st(F,W,G,C), st(T,W,G,C), act(alone,T)) :-
      pos(F), pos(T), pos(W), pos(G), pos(C),
      F \== T.
move(st(F,F,G,C), st(T,T,G,C), act(wolf,T)) :-
      pos(F), pos(T), pos(G), pos(C), F \== T.
move(st(F,W,F,C), st(T,W,T,C), act(goat,T)) :-
      pos(F), pos(T), pos(W), pos(C), F \== T.
move(st(F,W,G,F), st(T,W,G,T), act(cabbage,T)) :-
      pos(F), pos(T), pos(W), pos(G), F \== T.
```

Example: A Riddle (6)

- It would be tempting to define a predicate

  `reachable(State, ListOfActions)`

  as:

```
reachable(st(left,left,left,left), []).
reachable(S2, A2) :-
        reachable(S1, A1),
        move(S1, S2, A),
        allowed(S2),
        append(A1, [A], A2).
```

- However, this can produce infinitely long sequences of moves (e.g. moving the goat back and forth).

  Furthermore, the left recursion creates an infinite loop in Prolog.

Example: A Riddle (7)

- Thus, a third argument is used for a list of states that were already visited and may not be repeated:

```
reachable(Start, [], [Start]) :-
        Start = st(left,left,left,left).
reachable(S2, A2, [S2|V]) :-
        reachable(S1, A1, V),
        move(S1, S2, A),
        allowed(S2),
        \+ member(S2, V),
        append(A1, [A], A2).
solution(A) :-
        reachable(st(right,right,right,right), A, _).
```

"`\+ member(S2, V)`" means that `S2` is not contained in the list `V`.

Example: A Riddle (8)

- The above program has the right logical semantics and would work in a deductive DBMS that uses bottom-up evaluation (applies rules right-to-left).
- But in Prolog, it would generate an infinite loop.

  The recursion in the second rule does not finish (all arguments are unbound variables). In Prolog, the recursive call in the body must be nearer towards the termination condition than the call represented by the head. In the example, there must be more excluded states.
- In Prolog, rules are executed from left to right.

  Often, the distinction is not important, but sometimes it is necessary to view the predicates as procedures with input- and output-parameters. Here the excluded states must be an input parameter, not an output one.

Example: A Riddle (9)

- This version works in Prolog:

```
reachable(st(left,left,left,left), [], _).
reachable(S2, A2, V) :-
        move(S1, S2, A),
        allowed(S1),
        \+ member(S1, V),
        reachable(S1, A1, [S1|V]),
        append(A1, [A], A2).
solution(A) :-
        Goal = st(right,right,right,right),
        reachable(Goal, A, [Goal]).
```

The predicate `reachable(S, A, V)` is called with `S` and `V` given. States are constructed backwards.

Example: A Riddle (10)

- If one does not like the default negation "\+", one can define

```
not_member(X, []).
not_member(X, [F|R]) :-
        X \== F,
        not_member(X, R).
```

Defining a replacement for the built-in predicate \== (for the application in this program) is left as an exercise for the reader.

Overview

Logic Programs (1)

Definition:

- A Logic Program P is a set of definite Horn clauses, i.e. formulas of the form

$$A \leftarrow B_1 \wedge \cdots \wedge B_n.$$

  where $A, B_1, \ldots, B_n$ are atomic formulas and $n \geq 0$.

  For Prolog execution, the sequence of the rules is important. Then a Pure Prolog Program is a list of definite Horn clauses.

- Such formulas are called rules. A is called the head of the rule, and $B_1 \wedge \cdots \wedge B_n$ the body of the rule.

Logic Programs (2)

Note:

- In this Chapter, we assume that the signature is one-sorted.
- This corresponds to Prolog being untyped, and makes the formalism simpler.
- Often, a signature is not explicitly given (Prolog needs no declarations).
- However, given a logic program P, one can always construct a signature Σ of the symbols that appear in P.

Logic Programs (3)

Definition:

- A fact is a rule with empty body and without variables. The empty body is seen as "true".

  A conjunction is true iff all its conjuncts are true. If there is none, this is trivially satisfied.

- A fact is written as "$A \leftarrow .$" or as "$A.$".
- Sometimes facts are identified with the positive ground literal in the head.
- One also sometimes says "fact" when one really means "positive ground literal" (fact is shorter).

Logic Programs (4)

Note:

- The definitions become simpler when facts are seen as special cases of a rule.
- Of course, in deductive databases one separates
  - ◇ predicates that are defined only by facts (EDB predicates: classical relations).
  - ◇ predicates that are defined by proper rules (IDB predicates: views).

    EDB: Extensional Data Base. IDB: Intensional Data Base.
- In deductive databases, often no function symbols are permitted.

Herbrand Interpretations (1)

- This is only a repetition. See Chapter 2.
- It is difficult to consider arbitrary interpretations.
- Herbrand interpretations have a
  - ◇ fixed domain: Set of all ground terms.
  - ◇ fixed interpretation of constants as themselves.
  - ◇ fixed interpretation of function symbols as term constructors (“free interpretation”).
- Thus, only the interpretation of the predicates can be chosen in an Herbrand interpretation.

Herbrand Interpretations (2)

Definition:

- The Herbrand universe $\mathcal{U}_\Sigma$ for a signature Σ is the set of all ground terms that can be constructed with the constants and function symbols in Σ.

  If the signature should contain no constant, one adds one constant a (so that the Herbrand universe is not empty).

- For a logic program P, the Herbrand universe $\mathcal{U}_P$ is the set of ground terms that can be built with the constants and function symbols that appear in P.

  I.e. if a signature is not explicitly given, one assumes that the signature contains only the constants and function symbols that appear in P. Must must again add a constant if $\mathcal{U}_P$ would otherwise be empty.

Herbrand Interpretations (3)

Definition:

- The Herbrand base $\mathcal{B}_\Sigma$ is the set of all positive ground literals that can be built over Σ.

  Again, one must ensure that the set is not empty be adding a constant if Σ does not contain any constant.

- I.e. the Herbrand base is the set of all formulas of the form $p(t_1, \ldots, t_n)$, where p is a predicate of arity n in Σ, and $t_1, \ldots, t_n \in \mathcal{U}_\Sigma$.
- Again, if instead of a signature Σ, a logic program P is given, one constructs the signature of the symbols that appear in P.

Herbrand Interpretations (4)

- A Herbrand interpretation $\mathcal{I}$ can be identified with the set of all positive ground literals $p(t_1, \ldots, t_n)$ that are true in $\mathcal{I}$, i.e. with $H := \{A \in \mathcal{B}_\Sigma \mid \mathcal{I} \models A\}$.
- Conversely, $H \subseteq \mathcal{B}_\Sigma$ denotes the Herbrand interpretation with

$$\mathcal{I}[p] := \{(t_1, \ldots, t_n) \in \mathcal{U}_\Sigma^n \mid p(t_1, \ldots, t_n) \in H\}.$$

  Otherwise, $\mathcal{I}$ is fixed, because it is a Herbrand interpretation: For the single sort s, $\mathcal{I}[s] := \mathcal{U}_\Sigma$, for constants c, $\mathcal{I}[c] := c$, and for function symbols f of arity n: $\mathcal{I}[f](t_1, \ldots, t_n) := f(t_1, \ldots, t_n)$.

- Thus, in the following, Herbrand interpretations are subsets of $\mathcal{B}_\Sigma$.

Herbrand Interpretations (5)

Definition:

- A Herbrand model of a logic program P is a Herbrand interpretation $\mathcal{I}$ that is a model of P.

Exercise:

- Name two different Herbrand models of P:

$$\begin{array}{l} p(a). \\ p(b). \\ q(a, b). \\ r(X) \leftarrow p(X) \wedge q(X, Y). \end{array}$$

- Please name also a Herbrand interpretation that is not a Herbrand model of P.

Minimal Herbrand Model (1)

- A model of a logic program can be "too large" (contain unnecessary ground literals).
- The rules enforce only that if the body is true, also the head must be true.
- If the body is false, the rule is automatically satisfied. Nothing is required for the truth of the head.

  That is important because there can be several rules with the same head. If the body of this rule is false, the body of another rule with the same head can be true. Thus, one cannot require that if the body is false, the head must also be false.

Minimal Herbrand Model (2)

- E.g. the entire Herbrand base (the interpretation that makes everything true) is a model of every logic program.
- Of course, one wants a model that contains only those ground literals that must be true because of the rules.
- That is the minimal Herbrand model. It is the declarative semantics of a logic program.

  At least in the area of deductive databases. As we will see, Prolog's SLD-Resolution corresponds more the to set of supported models.

Minimal Herbrand Model (3)

Definition:

- A Herbrand interpretation $\mathcal{I}$ is called Minimal (Herbrand) Model of a logic program P iff
  - ◇ $\mathcal{I}$ is model of P ($\mathcal{I} \models P$), and
  - ◇ there is no smaller model of P, i.e. no Herbrand interpretation $\mathcal{I}'$ with $\mathcal{I}' \models P$ and $\mathcal{I}' \subset \mathcal{I}$ ($\mathcal{I}' \neq \mathcal{I}$).

Theorem:

- Every logic program has a unique minimal model.

  It is the intersection of all Herbrand models.

Minimal Herbrand Model (4)

Relation to Databases:

- As explained above, a relational database state is an interpretation with finite extensions of the relation symbols and no function symbols.

  For simplicity, we ignore datatype operations.

- If the logic program contains no function symbols, the minimal model is a relational DB state.
- If a predicate is defined only by facts, it is interpreted in the minimal model as exactly these facts.
- Rules then define views (derived predicates).

Minimal Herbrand Model (5)

Theorem:

- Let $\mathcal{I}$ be the minimal Herbrand model of a logic program P.
- For every positive ground literal $A \in \mathcal{B}_\Sigma$: If $\mathcal{I} \models A$, then $P \vdash A$.

  Of course, the same holds for a conjunction of positive ground literals.
- For every positive ground literal $A \in \mathcal{B}_\Sigma$: If $\mathcal{I} \not\models A$, then $P \nvdash A$.

  This is trivial.

Minimal Herbrand Model (6)

Note:

- The above theorem explains the importance of the minimal model for query evaluation: It is a prototypical model and instead of logical consequence we can talk about truth in this model.
- It does not hold for formulas that contain variables.

  E.g. $P = \{p(a), p(b)\}$. If a and b are the only constants in Σ (and there are no function symbols), $\forall X\ p(X)$ is true in the minimal model, but it is not implied.
- However, in deductive databases, one normally ensures that all variables in the query must be bound.

Minimal Herbrand Model (7)

Exercise:

- What is the minimal model of this logic program?

$$
\begin{array}{l}
\texttt{mother(arno, birgit).}\\
\texttt{father(birgit, chris).}\\
\texttt{parent(X, Y)} \leftarrow \texttt{mother(X, Y).}\\
\texttt{parent(X, Y)} \leftarrow \texttt{father(X, Y).}\\
\texttt{ancestor(X, Y)} \leftarrow \texttt{parent(X, Y).}\\
\texttt{ancestor(X, Z)} \leftarrow \texttt{parent(X, Y)} \wedge \texttt{ancestor(Y, Z).}
\end{array}
$$

- Guess a model $\mathcal{I}$ and explain for each $A \in \mathcal{I}$ that there cannot be a model without A.

Minimal Herbrand Model (8)

Example:

- Consider the following logic program:

```
a_list([]).
a_list([a|X]) :- a_list(X).
```

- This program has an infinite minimal model:

$$\mathcal{I} = \{\texttt{a\_list([])}, \texttt{a\_list([a])}, \texttt{a\_list([a,a])}, \ldots\}.$$

- This explains e.g. why Prolog answers
  - ◇ `a_list([a,a])` with “yes”,
  - ◇ `a_list([a,b])` with “no”.

Overview

Computing the Min. Model

- One can compute the minimal model by iteratively inserting known facts for the body literals to compute a new literal:

$$\begin{tikzcd}
p(a,b) & & & & \\
p(a,X) \arrow[u] & \leftarrow & q(X,Y) & \wedge & r(Y,a). \\
& & q(b,c) \arrow[u] & & r(c,a) \arrow[u]
\end{tikzcd}$$

- This rule application is formalized by the immediate consequence operator T_P.

Substitutions (1)

- This is a repetition, see Chapter 2.

  Here the definition is slightly simpler, because no sorts are considered.

- A substitution is a mapping $\theta: \mathit{VARS} \to \mathit{TE}_\Sigma$, such that the set $\{V \in \mathit{VARS} \mid \theta(V) \neq V\}$ is finite.

  The restriction ensures that a substitution can be finitely represented. It is not a real restriction because formulas anyway contain only finitely many variables.

- A substitution is usually written down as a set of variable/term-pairs in the form $\{X/a, Y/Z\}$.

  This means the substitution θ with $\theta(X) = a$, $\theta(Y) = Z$, and $\theta(V) = V$ for all other variables V.

Substitutions (2)

- The domain of a substitution can be extended from the set of variables successively to terms, literals, and rules (or arbitrary formulas).
- This is done by replacing the variables inside the term, literal, rule as specified in the substitution and leaving the rest unchanged.
- E.g. the substitution $\theta = \{X/a, Y/Z\}$ applied to the literal $p(X, Y, V, b)$ gives the literal $p(a, Z, V, b)$.
- The postfix notation is often used for applying a substitution, e.g. $A\theta$ means $\theta(A)$.

Substitutions (3)

- Note that a substitution is applied only once, not iteratively. E.g. $\theta = \{X/Y, Y/Z\}$ maps $p(X)$ to $p(Y)$, and not to $p(Z)$.
- A substitution θ is a ground substitution for a rule F iff it replaces all variables that occur in F by ground terms.
- Thus, the result of applying a ground substitution to a rule F is a ground rule.

  I.e. a ground substitution replaces all variables by concrete values. For Herbrand interpretations, ground substitutions and variable assignments are basically the same.

Ground Instances (1)

Definition:

- A rule F_1 is an instance of a rule F_2 iff there is a substitution θ with $F_1 = \theta(F_2)$.
- A ground instance is an instance that is variable-free (the result of applying a ground substitution).
- We write $ground(P)$ for the set of all ground instances of rules in P.

Ground Instances (2)

Example:

- E.g. $\texttt{parent(arno, birgit)} \leftarrow \texttt{mother(arno, birgit)}$ is a ground instance of $\texttt{parent(X, Y)} \leftarrow \texttt{mother(X, Y)}$.
- The ground substitution is $\theta = \{\texttt{X/arno},\ \texttt{Y/birgit}\}$.
- E.g. $\texttt{parent(arno, chris)} \leftarrow \texttt{mother(arno, chris)}$ is another ground instance of the same rule.
- E.g. $\texttt{parent(chris, birgit)} \leftarrow \texttt{mother(birgit, doris)}$ is not a ground instance of the above rule.

  One must of course replace all occurrences of the same variable in a rule by the same value (when computing a single ground instance of a single rule).

Ground Instances (3)

Exercise:

- Let the following rule be given:

$$p(a, X) \leftarrow q(X, Y) \wedge r(Y, a).$$

- Which of the following rules are ground instances of the given rule?

☐ $p(a, a) \leftarrow q(a, a) \wedge r(a, a).$

☐ $p(a, b) \leftarrow q(a, b) \wedge r(b, a).$

☐ $p(a, b) \leftarrow q(b, c) \wedge r(c, a).$

☐ $p(b, a) \leftarrow q(a, a) \wedge r(a, a).$

☐ $p(a, b) \leftarrow q(b, Y) \wedge r(Y, a).$

T_P-Operator (1)

- Let a logic program P be given.
- The immediate consequence operator T_P maps Herbrand interpretations to Herbrand interpretations:

  $T_P(\mathcal{I}) := \{F \in \mathcal{B}_\Sigma \mid$ There is a rule
  $A \leftarrow B_1 \land \cdots \land B_n$ in P
  and a ground substitution θ,
  such that
  - $B_i\,\theta \in \mathcal{I}$ for $i = 1, \ldots, n$, and
  - $F = A\,\theta\}$.

- Note that the case $n = 0$ is possible, then the condition about the body literals is trivially satisfied.

T_P-Operator (2)

- The input interpretation $\mathcal{I}$ consists of facts that are already known (or assumed) to be true.
- The result $T_P(\mathcal{I})$ of the T_P-operator consists of those facts that are derivable in a single step from the given facts and the rules in the program.
- I.e. for each ground instance $A \leftarrow B_1 \wedge \cdots \wedge B_n$ of a rule in P, if the precondition $B_1 \wedge \cdots \wedge B_n$ is true in $\mathcal{I}$ (i.e. $\{B_1, \ldots, B_n\} \subseteq \mathcal{I}$), then $A \in T_P(\mathcal{I})$.

T_P-Operator (3)

Exercise:

- Let the following logic program P be given:

$$\begin{array}{l} p(a, b). \\ p(c, c). \\ q(X, Y) \leftarrow p(X, Y). \\ q(Y, X) \leftarrow p(X, Y). \end{array}$$

- Let $\mathcal{I}_0 := \emptyset$.
- Please compute $\mathcal{I}_1 := T_P(\mathcal{I}_0)$, $\mathcal{I}_2 := T_P(\mathcal{I}_1)$, and $\mathcal{I}_3 := T_P(\mathcal{I}_2)$.

T_P-Operator (4)

Theorem:

- Let P be any logic program and let
  - $\mathcal{I}_0 := \emptyset$,
  - $\mathcal{I}_{i+1} := T_P(\mathcal{I}_i)$ for $i = 0, 1, \ldots$.
- If there is $n \in \mathbb{N}_0$ with $\mathcal{I}_{n+1} = \mathcal{I}_n$ then $\mathcal{I}_n$ is the minimal Herbrand model of P.
- If $ground(P)$ is finite, there is always such an n.

T_P-Operator (5)

Exercise:

- Please compute the minimal model of the following logic program P by iteratively applying the T_P-operator:

$$\begin{array}{l}
\mathtt{mother(arno, birgit).} \\
\mathtt{father(birgit, chris).} \\
\mathtt{parent(X, Y)} \leftarrow \mathtt{mother(X, Y).} \\
\mathtt{parent(X, Y)} \leftarrow \mathtt{father(X, Y).} \\
\mathtt{ancestor(X, Y)} \leftarrow \mathtt{parent(X, Y).} \\
\mathtt{ancestor(X, Z)} \leftarrow \mathtt{parent(X, Y)} \wedge \mathtt{ancestor(Y, Z).}
\end{array}$$

- Does $\mathcal{I} \subseteq T_P(\mathcal{I})$ hold for arbitrary $\mathcal{I}$?

A Bit of Lattice Theory (1)

- Let a set $\mathcal{M}$ and a relation $\preceq \subseteq \mathcal{M} \times \mathcal{M}$ be given.
- $\preceq$ is a partial order iff for all $\mathcal{I}, \mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3 \in \mathcal{M}$ the following holds:
  - ◇ $\mathcal{I} \preceq \mathcal{I}$.

    I.e. $\preceq$ is reflexive.
  - ◇ $\mathcal{I}_1 \preceq \mathcal{I}_2$ and $\mathcal{I}_2 \preceq \mathcal{I}_3$ implies $\mathcal{I}_1 \preceq \mathcal{I}_3$.

    I.e. $\preceq$ is transitive.
  - ◇ $\mathcal{I}_1 \preceq \mathcal{I}_2$ and $\mathcal{I}_2 \preceq \mathcal{I}_1$ implies $\mathcal{I}_1 = \mathcal{I}_2$.

    I.e. $\preceq$ is antisymmetric.
- $(\mathcal{M}, \preceq)$ is then called a partially ordered set.

A Bit of Lattice Theory (2)

- Let $(\mathcal{M}, \preceq)$ be a partially ordered set.
- An element $\mathcal{I} \in \mathcal{M}$ is an upper bound of a set $\mathcal{N} \subseteq \mathcal{M}$ iff $\mathcal{J} \preceq \mathcal{I}$ for all $\mathcal{J} \in \mathcal{N}$.
- An element $\mathcal{I} \in \mathcal{M}$ is called least upper bound of a set $\mathcal{N} \subseteq \mathcal{M}$ iff
  - ◇ it is an upper bound, i.e. $\mathcal{J} \preceq \mathcal{I}$ for all $\mathcal{J} \in \mathcal{N}$,
  - ◇ $\mathcal{I} \preceq \mathcal{I}'$ for all $\mathcal{I}' \in \mathcal{M}$ that are an upper bound of $\mathcal{N}$.
- In the same way, one defines lower bound and greatest lower bound.

A Bit of Lattice Theory (3)

Lemma:

- If the least upper bound exists, it is unique.

Definition (Complete Lattice):

- A partially ordered set $(\mathcal{M}, \preceq)$ is a complete lattice if and only if
  - ◇ for every $\mathcal{N} \subseteq \mathcal{M}$ there is a least upper bound and a greatest lower bound.
- Then one writes $lub(\mathcal{N})$ for the least upper bound and $glb(\mathcal{N})$ for the greatest lower bound.

A Bit of Lattice Theory (4)

Definition (Top and Bottom Elements):

- A complete lattice $(\mathcal{M}, \preceq)$ always contains
  - ◇ a top element $\top := lub(\mathcal{M})$, and
  - ◇ a bottom element $\bot := glb(\mathcal{M})$.

Example:

- The set of all Herbrand interpretations (over a fixed signature) together with $\subseteq$ is a complete lattice:

$$lub(\mathcal{N}) = \bigcup_{\mathcal{I} \in \mathcal{N}} \mathcal{I}, \quad glb(\mathcal{N}) = \bigcap_{\mathcal{I} \in \mathcal{N}} \mathcal{I}, \quad \bot = \emptyset, \quad \top = \mathcal{B}_\Sigma.$$

A Bit of Lattice Theory (5)

Definition (Properties of Mappings):

- Let $T: \mathcal{M} \to \mathcal{M}$.
- T is monotonic iff $T(\mathcal{I}_1) \preceq T(\mathcal{I}_2)$ for all $\mathcal{I}_1 \preceq \mathcal{I}_2$.
- T is continuous iff $T(lub(\mathcal{N})) = lub(T(\mathcal{N}))$ for all $\mathcal{N} \subseteq \mathcal{M}$ such that every finite subset of $\mathcal{N}$ has an upper bound in $\mathcal{N}$. Here $T(\mathcal{N}) := \{T(\mathcal{I}) \mid \mathcal{I} \in \mathcal{N}\}$.

Lemma:

- If T is continuous, it is also monotonic.

A Bit of Lattice Theory (6)

Definition (Fixpoints):

- $\mathcal{I} \in \mathcal{M}$ is a fixpoint of T iff $T(\mathcal{I}) = \mathcal{I}$.
- $\mathcal{I} \in \mathcal{M}$ is the least fixpoint of T iff $T(\mathcal{I}) = \mathcal{I}$ and $\mathcal{I} \preceq \mathcal{J}$ for al $\mathcal{J} \in \mathcal{M}$ with $T(\mathcal{J}) = \mathcal{J}$.

Theorem:

- A monotonic mapping T in a complete lattice has always a least fixpoint, namely

$$glb(\{\mathcal{I} \in \mathcal{M} \mid T(\mathcal{I}) \preceq \mathcal{I}\}).$$

Let $lfp(T)$ be the least fixpoint of T.

A Bit of Lattice Theory (7)

Lemma:

- The immediate consequence operator T_P is monotonic and even continuous.

Lemma:

- $\mathcal{I}$ is a model of P iff $T_P(\mathcal{I}) \subseteq \mathcal{I}$.

Theorem:

- The least fixpoint of T_P is the minimal model of P.

A Bit of Lattice Theory (8)

Definition (Iteration of a Mapping):

- $T{\uparrow}0 := \bot$.
- $T{\uparrow}(n+1) := T(T{\uparrow}n)$.
- $T{\uparrow}\omega := lub(\{T{\uparrow}n \mid n \subseteq \mathbb{N}_0\})$.

Note:

- If there is $n \in \mathbb{N}_0$ with $T{\uparrow}(n+1) = T{\uparrow}n$, then $T{\uparrow}m = T{\uparrow}n$ for all $m \geq n$ and thus $T{\uparrow}\omega = T{\uparrow}n$.
- $T{\uparrow}\gamma$ can be defined for arbitrary ordinal numbers γ.

  $T{\uparrow}\gamma := T(T{\uparrow}(\gamma - 1))$ if γ is successor of $\gamma - 1$, and
  $T{\uparrow}\gamma := lub(\{T{\uparrow}\beta \mid \beta < \gamma\})$ otherwise (γ is a limit ordinal).

A Bit of Lattice Theory (9)

Lemma:

- If T is monotonic, $T\!\uparrow\! i \preceq T\!\uparrow\!(i+1)$ for all $i \in \mathbb{N}_0$.

Theorem:

- If T is continuous, it holds that $lfp(T) = T\!\uparrow\!\omega$.

  This implies the theorem on page 3-81 (minimal model computation). If T is only monotonic, there is an ordinal number γ with $lfp(T) = T\!\uparrow\!\gamma$.

Corollary:

- Even when the iteration of the T_P-operator does not terminate, every fact that is true in the minimal model is generated after finitely many iterations.

Supported Models (1)

Supported Model:

- A Herbrand model $\mathcal{I}$ of P is called supported model of P iff $T_P(\mathcal{I}) = \mathcal{I}$ (i.e. $\mathcal{I}$ is a fixpoint of T_P).

Note:

- Thus, for every fact A that is true in $\mathcal{I}$ there is a reason in form of a ground instance

$$A \leftarrow B_1 \land \cdots \land B_n$$

of a rule in P that permits to derive A (because $\mathcal{I} \models B_i$ for $i = 1, \ldots, n$).

Supported Models (2)

Corollary:

- The minimal model is a supported model.

Note:

- The converse is not true: Consider e.g. the program $P := \{p \leftarrow p\}$. The interpretation $\mathcal{I} := \{p\}$ is a supported model of P, but it is not minimal.

  Practical example: $\texttt{married\_with}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{married\_with}(\texttt{Y}, \texttt{X})$.

- However, non-recursive programs (see below) have only one supported model, namely the minimal model.

Overview

Unification (1)

- Unification is used in Prolog for parameter passing: It matches the actual parameters with the formal parameters of a predicate. It can fail.
- It can also be seen as an assignment that is that is
  - ◇ symmetric: $X = a$ and $a = X$ are both legal and have the same effect (X is bound to a),
  - ◇ one-time: Once a variable is bound to a value, it is always automatically replaced by that value. It is impossible to assign a new value.
- Unification does pattern matching of trees.

Unification (2)

Definition (Unifier):

- A unifier of two literals A and B is a substitution θ with $A\theta = B\theta$.
- A and B are called unifiable if there is a unifier of A and B.
- θ is a most general unifier of A and B if for every other unifier θ' of A and B there is a substitution σ with $\theta' = \theta \circ \sigma$.

  $\theta \circ \sigma$ denotes the composition of θ and σ, i.e. $(\theta \circ \sigma)(A) = \sigma(\theta(A))$.

Unification (3)

Examples:

- $p(X, b)$ and $p(a, Y)$ are unifiable with most general unifier $\{X/a, Y/b\}$.
- $q(a)$ and $q(b)$ are not unifiable.
- Consider $q(X)$ and $q(Y)$:
  - ◇ $\{X/Y\}$ is a most general unifier of these literals.
  - ◇ $\{Y/X\}$ is another most general unifier of these literals. (It maps both literals to $q(X)$).
  - ◇ $\{X/a, Y/a\}$ is an example for a unifier that is not a most general unifier.

Unification (4)

Lemma:

- If there is a unifier of A and B, there is also a most general unifier (MGU).
- The most general unifier is unique up to variable renamings, i.e. if θ and θ' are both most general unifiers of A and B there is a substitution σ which is a bijective mapping from variables to variables such that $\theta' = \theta \circ \sigma$.

Notation:

- Let $mgu(A, B)$ be a most general unifier of A and B.

Unification (5)

```
unify(Literal/Term $t$, $u$): Substitution $\theta$
    **if** $t = u$ **then**
        $\theta := \{\}$;
    **else if** $t$ is a variable that does not occur in $u$ **then**
        $\theta := \{t/u\}$;
    **else if** $u$ is a variable that does not occur in $t$ **then**
        $\theta := \{u/t\}$;
    **else if** $t$ is $f(t_1, \ldots, t_n)$ and $u$ is $f(u_1, \ldots, u_n)$ **then**
        $\theta := \{\}$;
        **for** $i := 1$ **to** $n$ **do** $\theta := \theta \circ \text{unify}(t_i\theta, u_i\theta)$;
    **else** /* Different Functors/Constants */
        $\theta :=$ "not unifiable";
```

Unification (6)

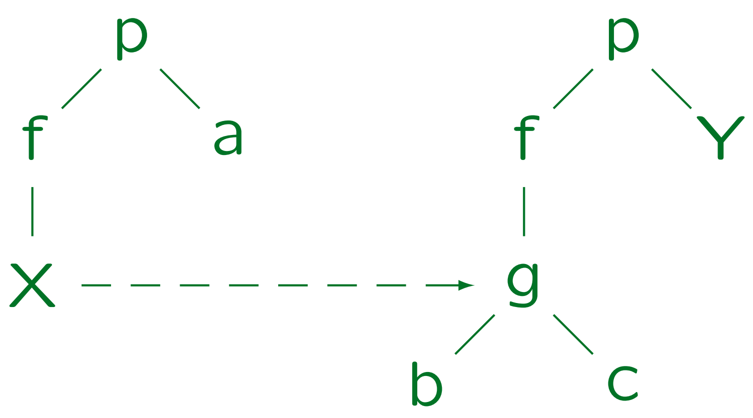
Example:

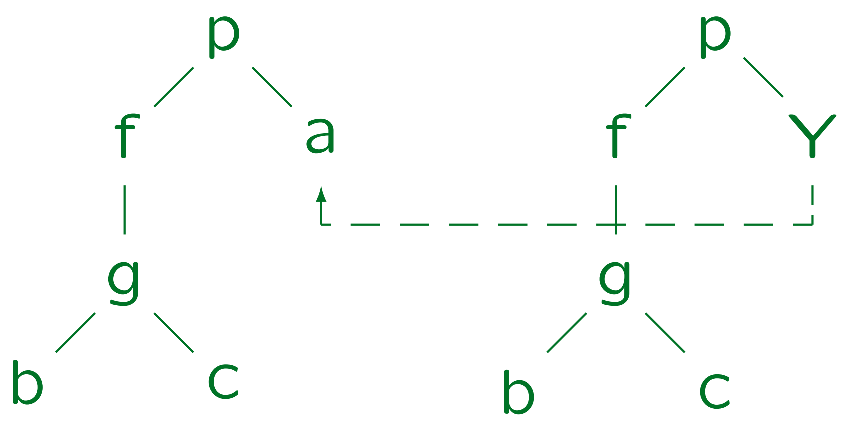
- $p(X, X)$ and $p(a, b)$ are not unifiable:
  - ◇ The first argument is unified with X/a.
  - ◇ However, then one has to unify $p(a, a)$ and $p(a, b)$. That is not possible.
- $p(X, X)$ and $p(Y, f(Y))$ are not unifiable:
  - ◇ First, one unifies X and Y, e.g. with $\{X/Y\}$.
  - ◇ Then one has to unify $p(Y, Y)$ and $p(Y, f(Y))$. It is not possible to bind Y to $f(Y)$, because Y occurs in $f(Y)$.

    $\{Y/f(Y)\}$ would not make the terms equal.

Unification (7)

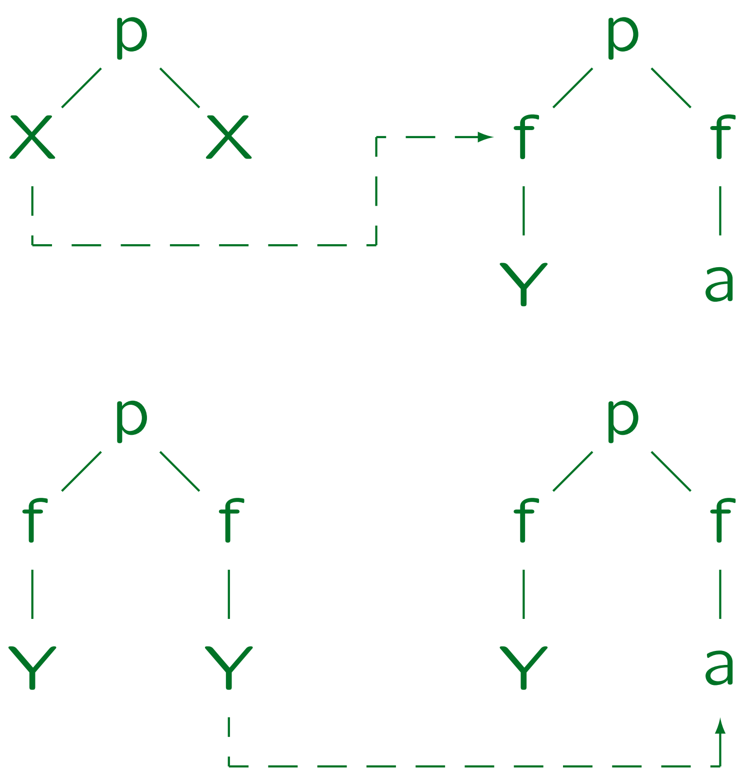
Example:





Unification (8)

Example:



Unification (9)

Exercises:

Compute the most general unifier if possible:

- $length([1,2,3], X)$ and $length([], 0)$.
- $length([1,2,3], X)$ and $length([E|R], N1)$.
- $append(X, [2,3], [1,2,3])$ and $append([F|R], L, [F|A])$.
- $p(f(X), Z)$ and $p(Y, a)$.
- $p(f(a), g(X))$ and $p(Y, Y)$.
- $q(X, Y, h(g(X)))$ and $q(Z, h(Z), h(Z))$.
- Use Prolog to check the solution.

Occur Check (1)

- Suppose that the following to literals are unified:
  - ◇ $p(X_1, \ldots, X_n)$,
  - ◇ $p(f(X_0, X_0), \ldots, f(X_{n-1}, X_{n-1}))$.
- The unifier is

$$\begin{aligned}\theta = \{&X_1/f(X_0, X_0),\\ &X_2/f(f(X_0, X_0),\ f(X_0, X_0)),\\ &\ldots\}.\end{aligned}$$

- The test, whether X_k appears in t_k ("occur check") costs exponential time.

  An explicit representation of θ would cost exponential time, too. But one normally uses pointers from variables to their values to represent a substitution internally: Then common subterms are stored only once.

Occur Check (2)

- Unification is the basic step in Prolog evaluation. It is bad if it can take exponential time.
- Solutions:
  - ◇ Unification without occur check: dangerous.

    This can give wrong solutions: E.g. consider the program consisting of $p \leftarrow q(X, X)$ and $q(Y, f(Y))$. Prolog systems without occur check answer "p" with "yes". It is also possible that unification or the printing of terms get into infinite loops.
  - ◇ With better data structures, the occur check has linear runtime.
  - ◇ Static analysis of a Program can show where no occur check is needed.

SLD-Resolution (1)

- SLD-resolution is the theoretical basis of Prolog execution.
- It is a theorem proving procedure that is complete for Horn clauses.
- SLD stands for "Linear resolution for Definite clauses with Selection function".

  In resolution, the basic derivation step is to conclude $A \vee C$ from $A \vee B$ and $\neg B \vee C$: I.e. one matches complementary literals (with a unifier) and composes the rests of the two clauses. It is a refutation proof procedure that starts with the negation of the proof goal and ends with the empty clause (the obvious contradiction). In linear resolution, one of the two clauses is always the result of the previous step.

SLD-Resolution (2)

- The idea of SLD-resolution is to simplify the query (proof goal) step by step to "true".

  If seen as refutation proof procedure, the current clause is the negation of the query, and one ends with "false".

- Each step makes a literal from the query and a rule head from the program equal with a unifier.
- Then literal in the query is replaced by the body of the rule. This gives a new query (hopefully simpler).
- Facts are treated as rules with empty body. Using facts makes the query shorter.

SLD-Resolution (3)

Example:

- Consider the following program:

  (1) $\texttt{ancestor(X, Y)} \leftarrow \texttt{parent(X, Y)}$.
  (2) $\texttt{ancestor(X, Z)} \leftarrow \texttt{parent(X, Y)} \wedge \texttt{ancestor(Y, Z)}$.
  (3) $\texttt{parent(X, Y)} \leftarrow \texttt{mother(X, Y)}$.
  (4) $\texttt{parent(X, Y)} \leftarrow \texttt{father(X, Y)}$.
  (5) $\texttt{father(julia, emil)}$.
  (6) $\texttt{mother(emil, birgit)}$.

- Let the query be

$$\texttt{ancestor(julia, birgit)}.$$

SLD-Resolution (4)

- The given query is the first proof goal:

$$\texttt{ancestor(julia, birgit)}.$$

- The only literal in the proof goal can be resolved with rule (2).
- The most general unifier of query literal and rule head is $\{\texttt{X}/\texttt{julia},\ \texttt{Z}/\texttt{birgit}\}$.
- Now the new proof goal is

$$\texttt{parent(julia, Y)} \land \texttt{ancestor(Y, birgit)}.$$

SLD-Resolution (5)

- Prolog always works on the first literal of the proof goal (this is a special selection function):

$$\underline{\texttt{parent(julia, Y)}} \land \texttt{ancestor(Y, birgit)}.$$

- It can be resolved with rule (4), this gives

$$\underline{\texttt{father(julia, Y)}} \land \texttt{ancestor(Y, birgit)}.$$

- Then the fact (5) is applied (with unifier $\{\texttt{Y/emil}\}$). This gives the proof goal:

$$\texttt{ancestor(emil, birgit)}.$$

SLD-Resolution (6)

- For the proof goal `ancestor(emil, birgit)`, one can e.g. apply rule (1) and replace it by

  `parent(emil, birgit)`.

- Now one can apply rule (3) and get the proof goal

  `mother(emil, birgit)`.

- This is given as a fact (line (6) in the program), and one gets the empty proof goal "$\square$".

- Thus, the query indeed follows from the given program, and the answer "yes" is printed.

SLD-Resolution (7)

- A sequence of proof goals that starts with a query Q and ends in the empty goal is called a derivation of Q from the given program.
- In the above derivation, the right program rule was “guessed” in each step. Prolog will try all possibilities with backtracking.
- If a query contains variables, the answer computed by a derivation is the composition of all substitutions applied.

SLD-Resolution (8)

Definition (Selection Function):

- A selection function is a mapping that, given a proof goal $A_1 \wedge \cdots \wedge A_n$, returns an index i in the range from 1 to n. (I.e. it selects a literal A_i.)

Note:

- Prolog uses the first literal selection rule, i.e. it selects always A_1 in $A_1 \wedge \cdots \wedge A_n$.
- As we will see, in deductive databases, a good selection function is an important part of the optimizer.

  The Prolog selection function also does not guarantee completeness for the answer "no". However, it is easy to implement with a stack.

SLD-Resolution (9)

Definition (SLD-Resolution Derivation Step):

- Let $A_1 \wedge \cdots \wedge A_n$ be a proof goal (query).
- Suppose the selection function chooses A_i.
- Let $B \leftarrow B_1 \wedge \cdots \wedge B_m$ be a rule from the program.
- Replace the variables in the rule by new variables, let the result be $B' \leftarrow B'_1 \wedge \cdots \wedge B'_m$.
- Let A_i and B' be unifiable, $\theta := mgu(A_i, B')$.
- Then the result of the SLD-resolution step is
  $(A_1 \wedge \cdots \wedge A_{i-1} \wedge B'_1 \wedge \cdots \wedge B'_m \wedge A_{i+1} \wedge \cdots \wedge A_n)\theta$.

SLD-Resolution (10)

Definition (Applicable Rule):

- In the above situation, the rule $B \leftarrow B_1 \wedge \cdots \wedge B_m$ is called applicable to the proof goal $A_1 \wedge \cdots \wedge A_n$.
- I.e. after renaming the variables in the rule, giving $B' \leftarrow B'_1 \wedge \cdots \wedge B'_m$, the head literal B' unifies with the selected literal A_i in the proof goal.

Note:

- Several rules in the program can be applicable to the same proof goal.

  This leads to branches in the SLD-tree explained below.

SLD-Resolution (11)

- It is important that the variables of the rule are renamed such that there is no name clash with a variable in the proof goal.

  Or a previous substitution, see computed answer substitution below.

- E.g. suppose the proof goal is $p(X, a)$ and the rule to be applied is $p(b, X) \leftarrow$.
- There is no unifier of $p(X, a)$ and $p(b, X)$.
- However, variable names in rules are not important. If the variable in the rule is renamed, e.g. to X_1, the MGU is $\{X/b,\ X_1/a\}$.

SLD-Derivations (1)

Definition (SLD-Derivation, Successful SLD-Deriv.):

- Let a logic program P, a query Q, and a selection function be given.
- An SLD-derivation for Q is a (finite or infinite) sequence of proof goals Q_0, $Q_1, \ldots, Q_n$, ... such that
  - ◇ $Q_0 = Q$ and
  - ◇ Q_i for $i \geq 1$ is the result of an SLD-derivation step from Q_{i-1} and a rule from P.
- An SQL-derivation is successful iff it is finite and ends in the empty clause $\square$.

SLD-Derivations (2)

Definition (Failed SLD-Derivation):

- An SLD-derivation $Q_0, \ldots, Q_n$ is failed iff it is finite, the last goal Q_n is not the empty clause $\square$, and the given program does not contain a rule that is applicable to Q_n.

Summary: Classification of SLD-Derivations:

- Successful: Finite, ends in $\square$.
- Failed: Finite, ends not in $\square$, no applicable rule.
- Incomplete: Finite, there is an applicable rule.
- Infinite.

SLD-Derivations (3)

Example (shown also on next page with applied rules):

- $\mathtt{ancestor(julia, birgit)}$.
- $\mathtt{parent(julia, Y)} \land \mathtt{ancestor(Y, birgit)}$.
- $\mathtt{father(julia, Y)} \land \mathtt{ancestor(Y, birgit)}$.
- $\mathtt{ancestor(emil, birgit)}$.
- $\mathtt{parent(emil, birgit)}$.
- $\mathtt{mother(emil, birgit)}$.
- $\square$.

SLD-Derivations (4)

$\texttt{ancestor(julia, birgit).}$

$\boxed{\texttt{ancestor(X, Z)} \leftarrow \texttt{parent(X, Y)} \land \texttt{ancestor(Y, Z).}}$

$\texttt{parent(julia, Y)} \land \texttt{ancestor(Y, birgit).}$

$\boxed{\texttt{parent(X, Y)} \leftarrow \texttt{father(X, Y).}}$

$\texttt{father(julia, Y)} \land \texttt{ancestor(Y, birgit).}$

$\boxed{\texttt{father(julia, emil).}}$

$\texttt{ancestor(emil, birgit).}$

$\boxed{\texttt{ancestor(X, Y)} \leftarrow \texttt{parent(X, Y).}}$

$\texttt{parent(emil, birgit).}$

$\boxed{\texttt{parent(X, Y)} \leftarrow \texttt{mother(X, Y).}}$

$\texttt{mother(emil, birgit).}$

$\boxed{\texttt{mother(emil, birgit).}}$

$\Box$

SLD-Derivations (5)

Exercise:

- Let the following logic program be given:

```
append([], L, L).
append([F|R], L, [F|A]) ← append(R, L, A).
```

- Give a successful SLD-derivation for

```
append([1], [2], [1,2]).
```

- What are the applied rules and most general unifiers in each step?

Computed Answers (1)

Definition (Computed Answer Substitution):

- Given a logic program P and a query Q, let
  $$Q_0 = Q,\ Q_1, \ldots, Q_n$$
  be a successful SLD-derivation for Q, and $\theta_1, \ldots, \theta_n$ be the most general unifiers applied in the SLD resolution steps.
- Let θ be the composition $\theta_1 \circ \cdots \circ \theta_n$ of these unifers, restricted to the variables that occur in the query Q.
- Then θ is a computed answer substitution for Q.

  Or: The answer substitution computed by this SLD-derivation.

Computed Answers (2)

Example (For Program on Slide 3-108):

- A successful derivation for $\mathtt{parent}(\mathtt{X}, \mathtt{Y})$ is as follows:
  - ◇ Goal: $\mathtt{parent}(\mathtt{X}, \mathtt{Y})$.
    Rule: $\mathtt{parent}(\mathtt{X}_1, \mathtt{Y}_1) \leftarrow \mathtt{mother}(\mathtt{X}_1, \mathtt{Y}_1)$.
    MGU: $\theta_1 := \{\mathtt{X}/\mathtt{X}_1,\ \mathtt{Y}/\mathtt{Y}_1\}$.
  - ◇ Goal: $\mathtt{mother}(\mathtt{X}_1, \mathtt{Y}_1)$.
    Rule: $\mathtt{mother}(\mathtt{emil}, \mathtt{birgit})$.
    MGU: $\theta_2 := \{\mathtt{X}_1/\mathtt{emil}, \mathtt{Y}_1/\mathtt{birgit}\}$.
  - ◇ Goal: $\square$.
- $\theta_1 \circ \theta_2 = \{\mathtt{X}/\mathtt{emil},\ \mathtt{Y}/\mathtt{birgit},\ \mathtt{X}_1/\mathtt{emil},\ \mathtt{Y}_1/\mathtt{birgit}\}$.
- Computed answer substitution: $\{\mathtt{X}/\mathtt{emil},\ \mathtt{Y}/\mathtt{birgit}\}$.

Computed Answers (3)

Theorem (Correctness of SLD-Resolution):

- For every program P, query Q, and computed answer substitution θ: $P \vdash Q\,\theta$.

  I.e. the program (set of Horn clauses) logically implies the query (conjunction of positive literals) after the answer substitution is applied to the query. As always, variables are treated as universally quantified.

Theorem (Completeness of SLD-Resolution):

- For every program P, query Q, and substitution θ with $P \vdash Q\,\theta$, there is a computed answer substitution θ_0 and a substitution θ_1 such that $\theta = \theta_0 \circ \theta_1$.

  I.e. for every correct answer substitution, SLD-resolution either computes it, or it computes a more general substitution.

Computed Answers (4)

Note (On the Completeness):

- E.g. consider the program consisting of the rule

$$p(f(X)) \leftarrow .$$

- Let the query be $p(Y)$.
- The substitution $\theta := \{Y/f(a)\}$ is correct, i.e. it satisfies $P \vdash Q\,\theta$, but SLD-resolution computes the more general substitution $\theta_0 := \{Y/f(X)\}$.
- θ_0 is more general than θ, because it can be composed with $\theta_1 := \{X/a\}$ to give θ.

Computed Answers (5)

Note (On Prolog):

- The correctness result holds only if the Prolog system does the occur check, e.g. try the program P:

  ```
  p ← q(X,X).
  q(X, f(X)).
  ```

  Prolog systems without occur check answer "`p`" with "`yes`", but `p` is not a logical consequence of P.

- The completeness result holds only if the Prolog system terminates. Prolog might run into an infinite loop before it finds all answers.

SLD-Trees (1)

- There are usually more than one SLD-derivation for a given query, because for every proof goal, more than one rule might be applicable.
- Every successful SLD-derivation computes only one answer substitution, but a query might have several distinct correct answer substitutions.

  Thus, it is important for the completeness of SLD-resolution, that there can be several SLD-derivations for the same query.
- The different SLD-derivations for a given query are usually displayed in form of a tree, the SLD-tree.

SLD-Trees (2)

Definition (SLD-Tree):

- The SLD-tree for a program P and a query Q (and a given selection function) is constructed as follows:
  - ◇ Every node of the tree is labelled with a proof goal (query). The root node is labelled with Q.
  - ◇ Let a node $\mathcal{N}$ be labelled with the proof goal $A_1 \wedge \cdots \wedge A_n$, $n \geq 1$. Then $\mathcal{N}$ has a child node for every rule $B \leftarrow B_1 \wedge \cdots \wedge B_m$ in P that is applicable to $A_1 \wedge \cdots \wedge A_n$. The child node is labelled with the result of the corresponding SLD-resolution step.

SLD-Trees (3)

Example:

- Consider the following program:

$$
\begin{array}{l}
(1)\ \texttt{parent(X, Y)} \leftarrow \texttt{mother(X, Y).}\\
(2)\ \texttt{parent(X, Y)} \leftarrow \texttt{father(X, Y).}\\
(3)\ \texttt{father(julia, emil).}\\
(4)\ \texttt{mother(julia, frida).}\\
(5)\ \texttt{father(ian, emil).}\\
(6)\ \texttt{mother(ian, frida).}
\end{array}
$$

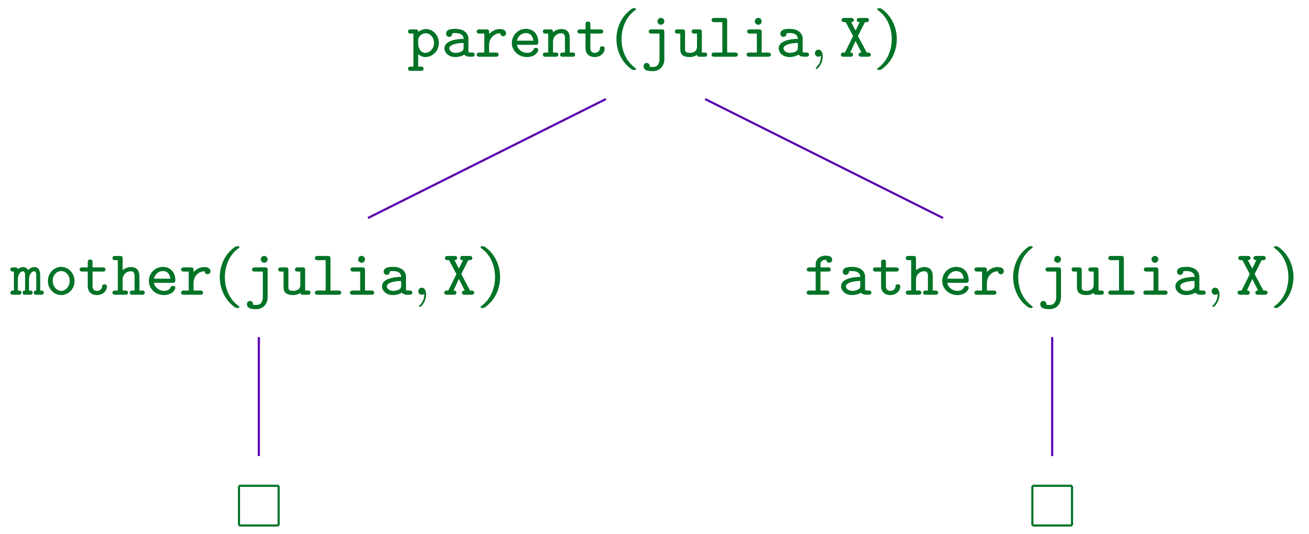
- Let the query be

  `parent(julia, X).`

- The SLD-Tree is shown on the next page.

SLD-Trees (4)

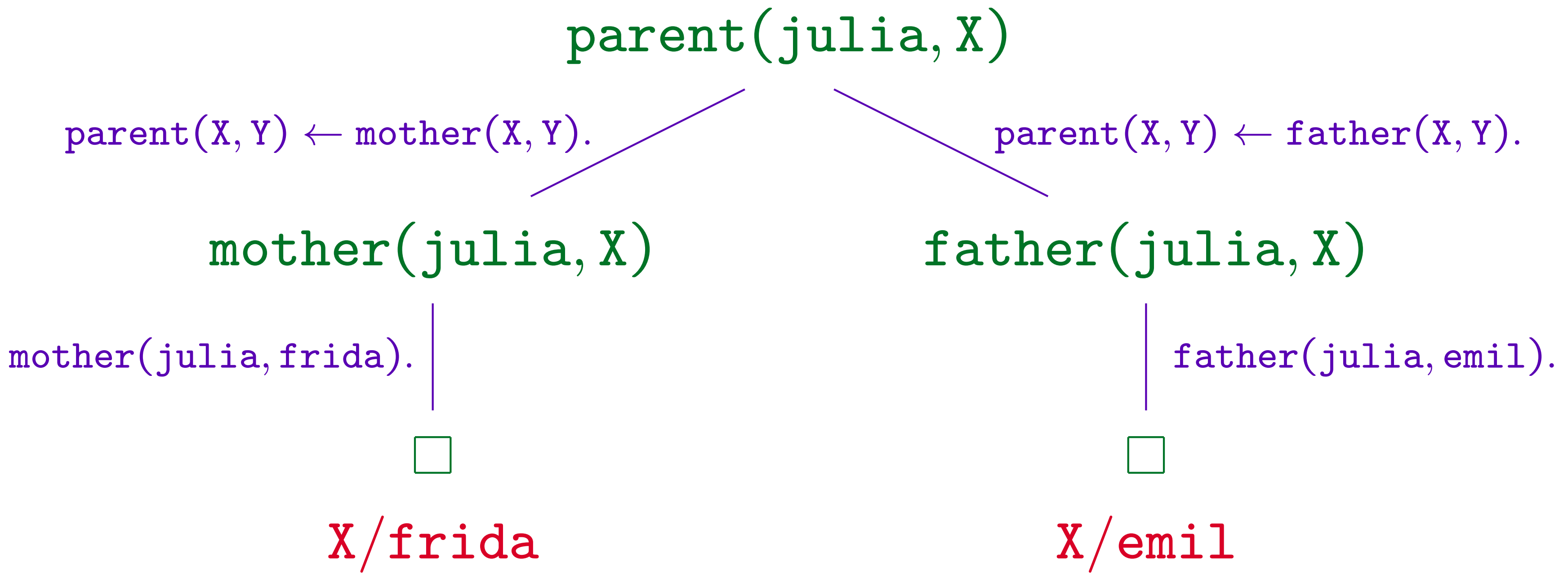
SLD-Tree:



- Often, it is also useful to know the applied rules and/or the computed answers. This information is shown in the variant on the next page.

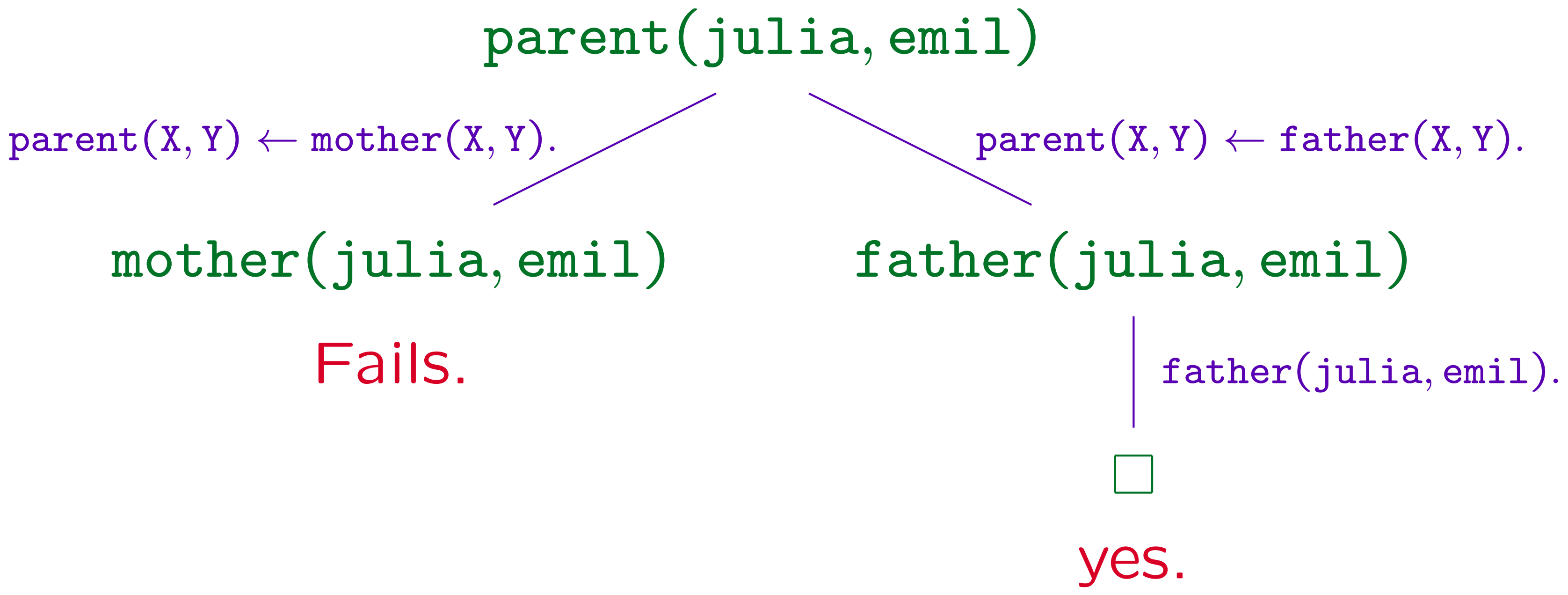
SLD-Trees (5)

SLD-Tree (with applied rules and computed answers):



SLD-Trees (6)

Another Example (Is `emil` parent of `julia`?):



SLD-Trees (7)

- Please note that branching in an SLD-tree happens only when there are several applicable rules.

  There is exactly one child node for each applicable rule, i.e. a rule of which the head literal is unifiable with the selected literal in the current node. I.e. the branching is done only for disjunctions ($\vee$).

- If a rule has several body literals, these are added together to the current goal.

  I.e. for conjunctions ($\wedge$) no branching is done (otherwise the binding of common variables would become difficult). If there is always only one applicable rule, the SLD-tree is a single path from root to leaf, even if the rules have many body literals. In the examples on the slides, the rules have only a single body literal, because there is little space. On Slide 3-120 an SLD-derivation (a single branch in the SLD-tree) is shown in which a rule has two body literals.

SLD-Trees (8)

Exercise:

- Consider again the program for list concatenation:

```
(1)  append([], L, L).
(2)  append([F|R], L, [F|A]) ← append(R, L, A).
```

- What is the SLD-tree for

```
append(X, Y, [1,2]).
```

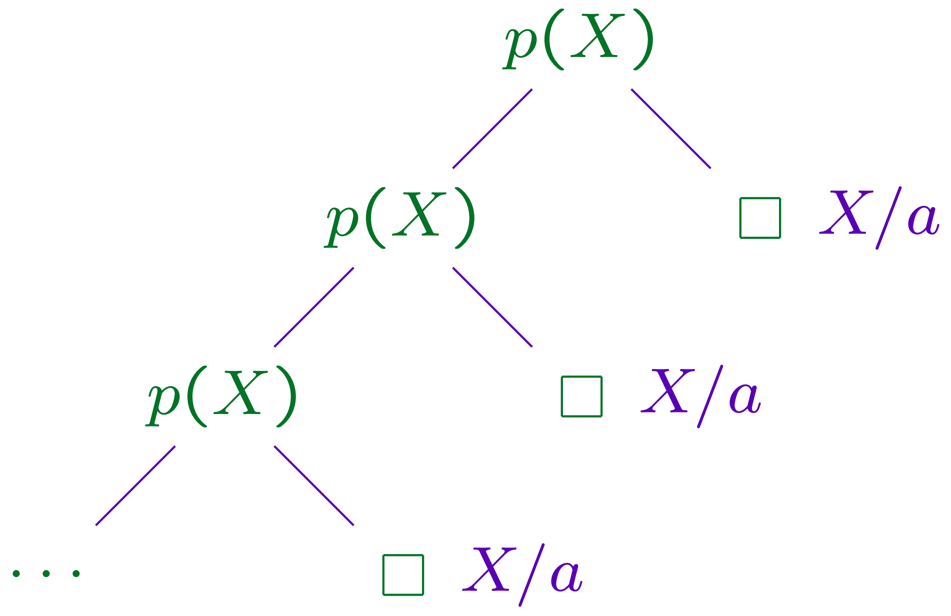
- Which answers do the different paths in the SLD-tree (i.e. the SLD-derivations) compute?

Infinite Paths (1)

- Consider the following program:

  (1) $p(X) \leftarrow p(X)$.
  (2) $p(a)$.

- The query $p(X)$ has the following SLD-tree:



Infinite Paths (2)

- Prolog searches the SLD-tree depth first.

  It also uses alternative rules always in the order that they are written down in the program.

- In this example, Prolog will get into an infinite loop and will not compute the correct answer substitution $\{X/a\}$. Thus, Prolog is not complete.
- However, if one would search the SLD-tree breadth-first, one would find all correct answer substitutions (because of the completeness of SLD-resolution).

Infinite Paths (3)

- But depth-first search is much more efficient to implement (with a stack).
- One solution is iterative deepening.

  First, one searches the SLD-tree depth-first, but e.g. only to depth 5. Then, one searches the SLD-tree again up to depth 10 (printing only answers below depth 5). And so on.
- In the XSB-system, it one can switch on "tabling" for selected predicates. Then the system detects when the same selected literal appears again.

  Then infinite loops can happen only when more and more complicated terms are constructed. For programs without function symbols (and built-in predicates), termination is guaranteed.

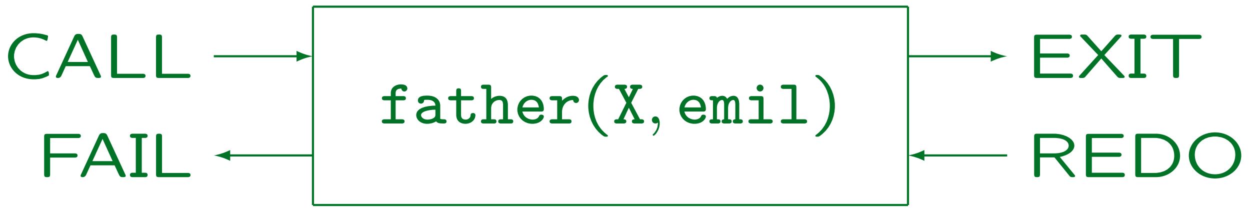
Overview

Box Model (1)

- Prolog uses SLD-resolution with
  - ◇ the first-literal selection function, and
  - ◇ depth-first search of the SLD-tree.
- However, the Prolog debugger does not show the entire proof goal (node label in the SLD-tree).
- Instead, it views predicates as nondeterministic procedures (procedures that can have more than one solution).
- The four-port debugger model is standard among Prolog systems.

Box Model (2)

- Each predicate invocation (selected literal in the SLD-tree) is represented as a box with four ports:
  - ◇ CALL A: Call of A, find first solution.
  - ◇ REDO A: Is there another solution for A?
  - ◇ EXIT A: A solution was found, A is proven.
  - ◇ FAIL A: There is no (more) solution for A.



Box Model (3)

- E.g. consider the following small program:

```
father(ian, emil).
father(julia, emil).
father(emil, arno).
```

- Debugger output for the query `father(X, emil)`:
  - `CALL father(X, emil)`
  - `EXIT father(ian, emil)`

    Note that the proven instance is shown.
  - Then the solution `X/ian` is displayed.

    Suppose one presses "`;`" to get more solutions.

Box Model (4)

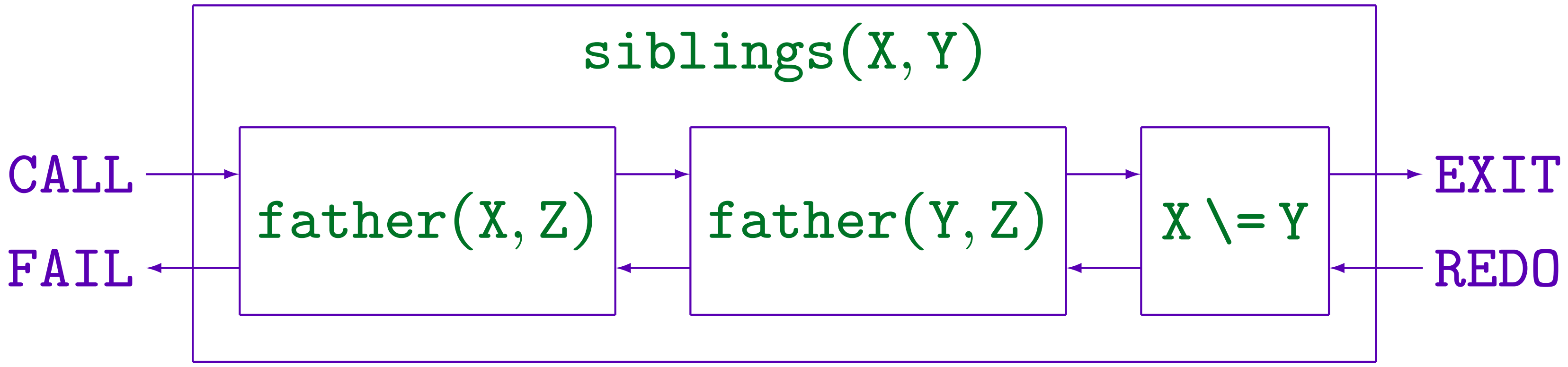
- Example debugger output, continued:
  - ◇ `REDO father(X, emil)`
  - ◇ `EXIT father(julia, emil)`
  - ◇ The solution `X/julia` is displayed. Some systems already know that there is no further solution. Otherwise, one can press again "`;`".
  - ◇ `REDO father(X, emil)`
  - ◇ `FAIL father(X, emil)`
  - ◇ The system prints "`no`".

Box Model (5)

- Suppose the program is extended with the rule

  $\texttt{siblings(X, Y)} \leftarrow \texttt{father(X, Z)} \wedge \texttt{father(Y, Z)} \wedge \texttt{X \textbackslash= Y}.$

- The box model is:



E.g. when the first or second body literal exists, the next body literal is called. When the last body literal is proven, `siblings` exits.

Box Model (6)

Debugger Output for the query `siblings(ian, Y)`:

```
(1)  0   CALL  siblings(ian, Y).
(2)  1   CALL  father(ian, Z).
(2)  1   EXIT  father(ian, emil).
(3)  1   CALL  father(Y, emil).
(3)  1  *EXIT  father(ian, emil).
(4)  1   CALL  ian \= ian.
(4)  1   FAIL  ian \= ian.
(3)  1   REDO  father(Y, emil).
(3)  1   EXIT  father(julia, emil).
(5)  1   CALL  julia \= ian.
(5)  1   EXIT  julia \= ian.
(1)  0   EXIT  siblings(ian, julia).
```

Box Model (7)

Remark:

- The exact form of the output depends on the Prolog system.
- The above output contains a box number in the first column and a nesting depth (call stack depth) in the second column.
- The asterisc "*" before `EXIT` marks that there are possibly further solutions (nondeterministic exit).

  Otherwise, the box is already removed, and not visited during backtracking (i.e. no `REDO`-`FAIL` will be shown). Because of such optimizations, the debugger output might violate the pure four-port model.

Box Model (8)

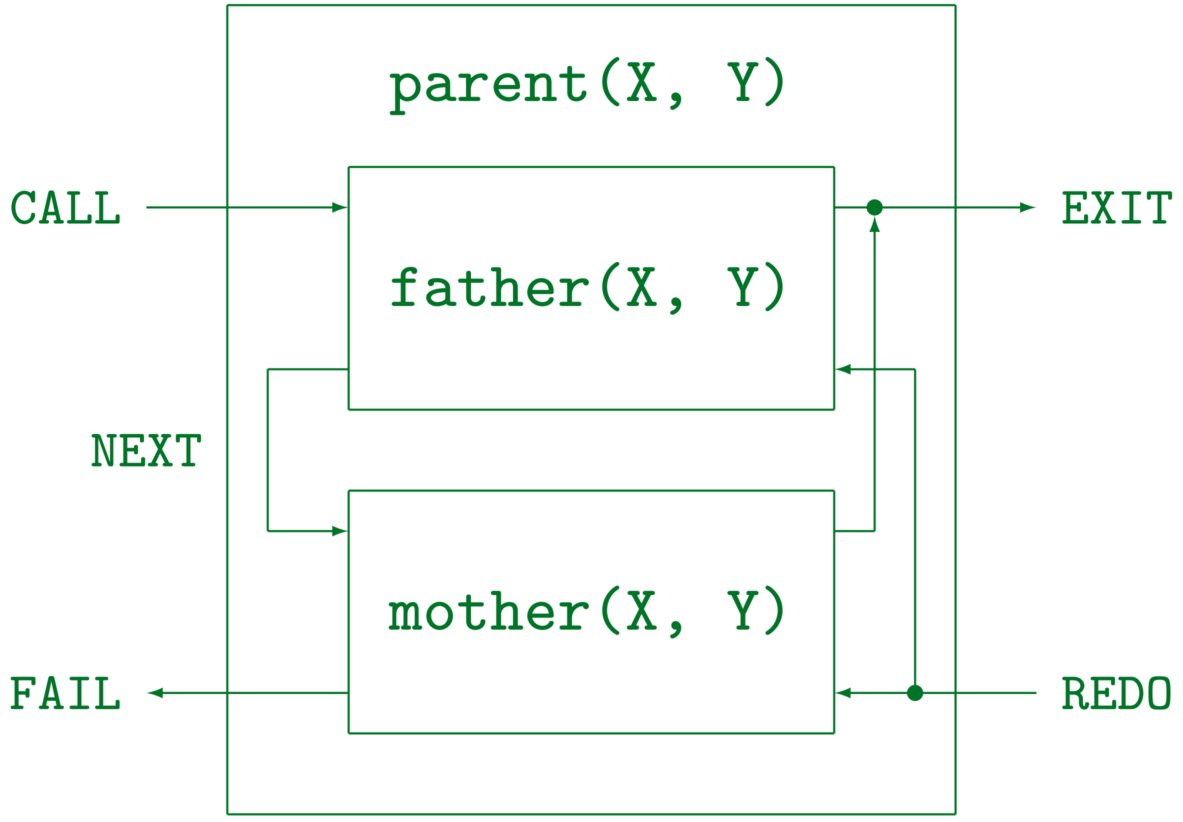
- Consider now a predicate defined with two rules:

$$\texttt{parent(X, Y)} \leftarrow \texttt{father(X, Y).}$$
$$\texttt{parent(X, Y)} \leftarrow \texttt{mother(X, Y).}$$

- The box model for `parent` is shown on the next page.

  There, also a port `NEXT` appears. This is a speciality of ECLiPSe Prolog. It shows when execution moves to another rule for the same predicate. In general, different Prolog systems have extended the basic Four-Port Model in various ways. E.g. SWI-Prolog can display a port "`UNIFY`" that shows the called literal after unification with the rule head.

Box Model (9)



REDO enters the inner box that was last left with EXIT.

Using the Debugger (1)

- The debugger output is switched on by executing the built-in predicate "`trace`" (as a query).

  It is switched off with "`notrace`". In SWI-Prolog, `trace` means only that the next query is traced.

- The debugger then displays a line for every port and waits for commands after each line.
- With "`Return`" one steps to the next port.
- Other commands are listed in the manual.

  Often, they are displayed when one enters "`?`". The command "`a`" should stop execution of the query ("abort").

Using the Debugger (2)

- It is possible to produce debugger output only selectively.
- One can set breakpoints ("spypoints") on a predicate with e.g.

```
spy father/2.
```

- If instead of "`trace`", one uses "`debug`", Prolog executes the program without interruption until it reaches a predicate with a spypoint set.

  Then one can continue debugging as above or "leap" to the next spypoint (usually with the command "`l`"). Of course, there are "`nodebug`" and "`nospy`".