Advanced Logic and Functional Programming

Lecture 1: Programming paradigms. Declarative programming. From first-order logic to Logic Programming.

Mircea Marin

Programming paradigms

Programming paradigm (software engineering) = style or way of programming hat prescribes a number of programming constructs and how to use them in order to model and solve efficiently a wide range of problems.
Major programming paradigms:

1. Imperative (or procedural) programming
2. Object-oriented programming (OOP)
3. Functional programming (FP)
4. Logic programming (LP)

Computation = sequence of **commands** that act on an implicit program state, until we reach a state containing the desired result.

- Way of thinking similar to everyday routines (e.g., food recipes)
- Typical examples of commands: blocks, conditionals, loops, assignment
- Main programming construct is procedure:

  abstracts the actions of one or more commands, and can be called as a single command.
- Representative programming languages: `Fortran`, `C`
- Oldest programming paradigm: hardware-oriented way of thinking–instruct the computer what to do
  - Other programming paradigms encourage a more human-oriented way of thinking

Mathematical function

- Computes a result from the values of its arguments
- Does not affect the implicit or global state of a program/system etc.

$\Rightarrow$ different calls of a mathematical function with same input args. always behave the same (are *indistinguishable*)

Procedure

- Group of commands that can be executed at once.
- Can have visible side effects, e.g.: delete a file, change the value of a global variable, etc.

$\Rightarrow$ different calls of a procedure with same input args. can behave different.

Computation=answering questions by search according to a fixed, predictable strategy.

- Way of thinking useful for solving problems related to the extraction of knowledge from basic facts and relations.
- **Programming** = encoding of knowledge as basic facts and rules, and collecting them in a program
- LP is a **declarative programming** paradigm
  - The user most know only how to encode what he knows
  - The method to find the answers to problems (*how?*) is predefined: there is a search strategy, built into the interpreter or compiler of the language
- Representative language: Prolog

Computation=compute by evaluating expressions

- Evaluation = stepwise reduction of the function calls in the expression, until all function calls have been reduced.
  For example:

  $\underline{(1+2)} * 3 - 4 \Rightarrow \underline{3 * 3} - 4 \Rightarrow \underline{9 - 4} \Rightarrow 5$.
- Programming = defining mathematical functions.
- **Functions are values:** They can be passed as arguments to function calls, stored in composite values, and returned as results of computations.
- Representative languages: Lisp, Haskell; Common Lisp, Racket.
- Functional programming (sub)styles:
  - Strict: Function calls are evaluated by first evaluating all their function arguments
  - Lazy: Function calls evaluate arguments only if they are really needed

- Strict evaluation:

$$0 * (3 + (\underline{8 - 2}) / 3) \Rightarrow 0 * (3 + \underline{6 / 3}) \Rightarrow 0 * (\underline{3 + 2}) \Rightarrow \underline{0 * 5} \Rightarrow 0$$

- Lazy evaluation: we know that 0 multiplied by anything is 0, thus

$$\underline{0 * (3 + (8 - 2) / 3)} \Rightarrow 0$$

- The earliest functional programming languages were strict. (E.g., Lisp and its dialects)
- Lazy evaluations are harder to analyse.
  - Haskell is a lazy programming language.

Imperative Programming:

```
minList(L, n)
    r ← L[0]
    i ← 1
    while i < n do
        r ← min(r, L[i])
    end while
    return r
```

```
min(a, b)
    if a < b then
        m ← a
    else
        m ← b
    endif
    i ← i + 1
    return m
```

Logic Programming:

```
min(X,Y,X) :- X =< Y.
min(X,Y,Y) :- Y < X.
minList([M],M).
minList([X,Y|T],M) :- minList([Y|T],N), min(X,N,M).
```

Functional programming:

```
minList(L, n)
   if (n == 1)
      return L[0]
   else
      return min(L[n − 1], minList(L, n − 1))
```

In FP, we use recursion to define repetitive computations.

- Note that `minList` is a recursive function.

Declarative programming paradigms

FP and LP are declarative programming paradigms:

- The programmer must only focus on programming what he knows:
  - In LP: encode knowledge by facts and rules
  - In FP: encode reusable patterns of computation by function definitions
- The computational strategy to find answers/results is predefined, e.g.
  - SLDNF resolution (Prolog)
  - Strict evaluation (Lisp, Racket)
  - Lazy evaluation (Haskell)

Declarative programming paradigms

FP and LP are declarative programming paradigms:

- The programmer must only focus on programming what he knows:
  - In LP: encode knowledge by facts and rules
  - In FP: encode reusable patterns of computation by function definitions
- The computational strategy to find answers/results is predefined, e.g.
  - SLDNF resolution (Prolog)
  - Strict evaluation (Lisp, Racket)
  - Lazy evaluation (Haskell)

This lecture is concerned with declarative programming styles: FP and LP, and deepening our knowledge about them.

- Languages of concern: Prolog (for Logic Programming), Racket (for strict Functional Programing), and Haskell (for lazy functional programming)

- The most successful formal system, used to describe meaningful sentences in a precise and unambiguous way.
- Used in mathematics, philosophy, linguistics, and computer science.
- It has a simple and easy-to-understand syntax and semantics.
  - $\Rightarrow$ can be used to specify knowledge in a precise and unambigous way.
  - $\Rightarrow$ language for automated reasoning:
    - Knowledge is encoded by first-order formulas
    - Computation = deriving new knowledge by applying rules of deduction.

In First-Order Logic (FOL), knowledge consists of properties and relations which hold between objects of interest:

- objects are represented by terms
- properties and relations btw. them are represented by formulas

A **term** t is either

1. a **constant** which represents an atomic object.

   Examples: numbers: 1 (an integer), 13.5 (a floating-point); strings: "Hello"; symbols: `flower`, `red`, `john`; ...
2. a **variable**, for an unknown object.

   In logic programming, variables start with uppercase letter
3. $f(t_1, \ldots, t_n)$ where f is a function symbol (or constructor) and $t_1, \ldots, t_n$ are terms. It represents the construction of a composite object with f from the objects represented by terms $t_1, \ldots, t_n$.

   Examples: `car(skoda,red,engine(gas),2014)`, `list(1,2,3)`, `+(sin(X),/(Y,2))`

Note: Often, we write terms using other notations, e.g.: `X+Y/2` instead of `+(X,/(Y,2))`; `[1,2]` instead of `list(1,2)`, etc.

Sentences are represented by atomic or composite formulas.

- An atomic formula (or atom) is of the form $p(t_1, \ldots, p_n)$ where p is a predicate symbol and $t_1, \ldots, t_n$ are terms.
  - it indicates that relation p holds between objects represented by terms $t_1, \ldots, t_n$.
  - when $n = 1$, $p(t_1)$ indicates that t_1 has property p.
- A **composite formula** is built with logical connectives and quantifiers: If A and B are formulas and X is a variable, then the following expressions are also formulas:

formula	intended reading
$\neg A$	not A
$A \wedge B$	A and B
$A \vee B$	A or B
$A \rightarrow B$	A implies B
$A \leftrightarrow B$	A if and only if B
$\forall X.A$	for all X, A
$\exists X.A$	there is an X such that A
true	always holds
false	never holds

Abbreviations: If $A, A_1, \ldots, A_n$ are formulas, then

- $$\bigwedge_{i=1}^{n} A_i = \begin{cases} \mathit{true} & \text{if } n = 0 \\ A_1 & \text{if } n = 1 \\ \left(\bigwedge_{i=1}^{n-1} A_i\right) \wedge A_n & \text{if n>1.} \end{cases}$$
- $$\bigvee_{i=1}^{n} A_i = \begin{cases} \mathit{false} & \text{if } n = 0 \\ A_1 & \text{if } n = 1 \\ \left(\bigvee_{i=1}^{n-1} A_i\right) \vee A_n & \text{if n>1.} \end{cases}$$
- If $X_1, \ldots, X_n$ are the free variables in A then
  - $\tilde{\exists}.A$ abbreviates $\exists X_1. \cdots . \exists X_n.A$
  - $\tilde{\forall}.A$ abbreviates $\forall X_1. \cdots . \forall X_n.A$

REMARKS: An empty conjunction is *true*, and an empty disjunction is *false*.

From natural language to first-order logic

Most meaningful sentences from a natural language (English, Romanian, etc.) can be translated onto formulas from FOL, with the same meaning.

- "Every bird has wings."
  This is the same as saying: "For all X, if X is a bird then X has wings."

$$\forall x.(bird(x) \rightarrow hasWings(x))$$

- "The sky is blue and the sun is shining."

$$blue(sky) \wedge shining(sun)$$

- "Nobody is immortal"
  This is the same as saying "For all X, X is not immortal."

$$\forall x.\neg immortal(x)$$

- "Somebody uses the computer."

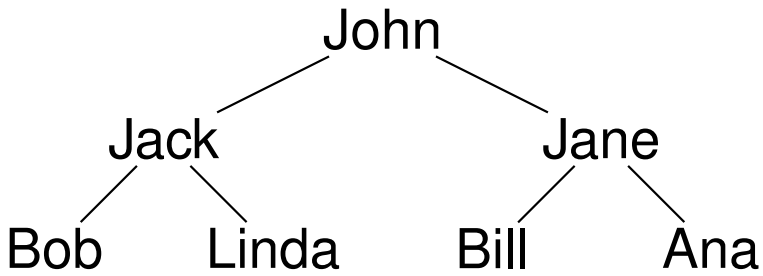
$$\exists X.uses(X, computer).$$

Use logic programming (LP) to answer the following question:

Who are the ancestors of John?

if we know that

Jack is the father of John. Jane is the mother of John. Bob is the father of Jack. Linda is the mother of Jack. Bill is the father of Jane. Ana is the mother of John.



We can use SWI Prolog to program what we know, and to find answers to questions about what we know. (See next slide)

Logic as programming language
Writing knowledge as facts

Program=collection of facts and rules that describe what we know.

- A fact is an elementary sentence (an atom).

```
father(john,jack).   % Jack is the father of John
father(jane,bill).   % Bill is the father of Jane
father(jack,bob).    % Bob is the father of Jack
mother(john,jane).   % Jane is the mother of John
mother(jack,linda).  % Linda is the mother of Jack
mother(jane,ana).    % Ana is the mother of Jane
```

Logic as programming language
Writing knowledge as facts

Program=collection of facts and rules that describe what we know.

- A fact is an elementary sentence (an atom).

```
father(john,jack).   % Jack is the father of John
father(jane,bill).   % Bill is the father of Jane
father(jack,bob).    % Bob is the father of Jack
mother(john,jane).   % Jane is the mother of John
mother(jack,linda).  % Linda is the mother of Jack
mother(jane,ana).    % Ana is the mother of Jane
```

REMARKS:

1. To write correct programs, we must respect the syntax of Prolog: (1) predicate names and symbolic constants must start with lowercase letters; (2) facts and rules must end with a dot character `'.'`; etc.
2. There are 6 facts which define the predicates `father` and `mother`
3. We can use rules to define other useful predicates (see next slide)

- A rule is a statement of the form "A if B_1 and ... and B_n." In Prolog, we write it as A `:-` $B_1, \ldots, B_n$`.`
- A is the head, and $B_1, \ldots, B_n$ is the body of the rule.
- We can use rules to define the well-known predicates `parent` and `ancestor`:

```
% Case 1: (Y is a parent of Y) if (Y is the father of X)
parent(X,Y) :- father(X,Y).
% Case 2: (Y is a parent of X) if (Y is the mother of X).
parent(X,Y) :- mother(X,Y).
% Case 1: (Y is an ancestor of X) if (Y is a parent of X)
ancestor(X,Y) :- parent(X,Y).
% Case 2: (Y is an ancestor of X) if it is an ancestor of
%         one of its parents (let's call it Z)
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

REMARK: Lines starting with '`%`' are comments with human-readable explanations

- We can use logical variables to write facts and rules about generic objects:
  - they must start with an uppercase letter. E.g., `X, Y, Z, ...`
  - every fact *A* is equivalent with the rule

    A `:- true .`
- After we write and load a program (using the `consult` command), we can ask questions (or queries) about what we know:

```
?- ancestor(john,Y). % which Y is an ancestor of John?
Y = jack ;
Y = jane ;
Y = bob ;
Y = linda ;
Y = bill ;
Y = ana ;
false.
```

Logic as programming language

- Logic programming allows to encode knowledge with rules of the form $\forall X_1 \ldots \forall X_m.(B_1 \wedge \ldots B_n \rightarrow A)$ where $A, B_1, \ldots, B_n$ are atoms. This formula is equivalent with the formula $\neg B_1 \vee \ldots \vee \neg B_n \vee A$ where all variables are implicitly universally quantified.
  In Prolog, we write A `:-` $B_1, \ldots, B_n$`.`
  When $n = 0$ (the body is empty), the rule becomes a fact; we write A`.`
- The facts and rules that describe our knowledge are stored in a program.
- The questions we can ask are called queries or goals. A goal is a formula of the form $\exists X_1 \ldots \exists X_m.A_1 \wedge \ldots \wedge A_n$ where $A_1, \ldots, A_n$ are atoms.
  In Prolog, we write `?-` $A_1, \ldots, A_n$`.`
  where all variables are implicitly existentially quantified.

Prolog is a language for logic programming (LP) where both programs and queries are represented by **Horn clauses**.

- A literal is either an atom or the negation of an atom.
- A Horn clause is a formula of one of the following forms, where $A_1, \ldots, A_n, B$ are atoms:

  Definite clause: $\neg B_1 \vee \ldots \vee \neg B_n \vee A$, which is logically equivalent with $B_1 \wedge \ldots \wedge B_n \rightarrow A$, and with the intended reading "Assume that, if B_1 and $\ldots$ and B_n hold, then A holds too."

  Fact: A, with intended reading "Assume A holds."

  Goal clause: $A_1 \wedge \ldots \wedge A_n$, with intended reading "Show that $\tilde{\exists}.(A_1 \wedge \ldots \wedge A_n)$ holds."

  REMARK: Goals are proven by contradiction: We prove that $\neg\tilde{\exists}.(A_1 \wedge \ldots \wedge A_n)$ does not hold $\Leftrightarrow \tilde{\forall}(\neg A_1 \vee \ldots \vee \neg A_n)$ does not hold.

- "Every man is mortal" can be encoded as the definite clause

$$man(X) \rightarrow mortal(X)$$

- "John is a man" can be encoded as the fact

$$man(john)$$

- "John is not mortal" can be encoded as the goal $\neg mortal(john)$.

From First-Order Logic to Logic Programming

Prolog = programming language developed in the 1970s by A. Colmerauer and his coworkers in Marseille, France.

- Program = set P of facts and definite clauses.
  - Not all knowledge can be encoded as set of definite clauses and facts, but most of it can.
- Computation: Given a goal clause $G = A_1 \wedge \ldots \wedge A_n$ and a program P, find
  - all answers $\theta = [X_1 \rightarrow \ldots, \ldots, X_m \rightarrow \ldots]$ for the variables $X_1, \ldots, X_m$ in G, such that $G\theta = A_1\theta \wedge \ldots \wedge A_n\theta$ can be deduced from the knowledge encoded in program P.
  - If no answer exists, return *false*.

The computation rule of Prolog is SLD: **S**elective **L**inear **D**efinite clause resolution:

- It proves that $G' = \neg A_1 \vee \ldots \neg A_n$ does not hold.
  - All variables of G' are, by default, universally quantified.

A rule of deduction (or rule of inference) is of the form

$$\frac{H_1 \quad \ldots \quad H_n}{C}$$

with the intended reading "if the formulas $H_1, \ldots, H_n$ hold, then the formula C holds too."

- $H_1, \ldots, H_n$ are the **hypotheses** of the rule
- C is the **conclusion**

Examples (rules of inference)

- Resolution rule (propositional logic): $\dfrac{A \vee B \quad \neg B \vee C}{A \vee C}$

  Special case: modus ponens $\dfrac{P \quad P \rightarrow Q}{Q}$

SLD resolution: generalization of the resolution rule from propositional logic, which takes into account the variables in formulas:

$$\frac{G' = (\neg B_1 \vee \ldots \vee \neg B_i \vee \ldots \vee \neg B_n) \quad C = (K_1 \wedge \ldots \wedge K_m \rightarrow B) \in P}{G'' = (B_1 \vee \ldots \vee B_{i-1} \vee K_1 \vee \ldots \vee K_m \vee B_{i+1} \vee \ldots \vee B_n)\theta}$$

where θ is a most general unifier between B_i and B.

- B_i is the atom selected in G', and C is the clause selected from P.
- Abbreviated notation: $G' \rightarrow_{\theta,C} G''$.

REMARK:In Prolog, B_i is always the leftmost atom of G'. This means that the subgoals of G are answered from left to right.

The computational model of Prolog

Looks for solutions of $?- A_1, \ldots, A_n$ by generating an SLD-tree:

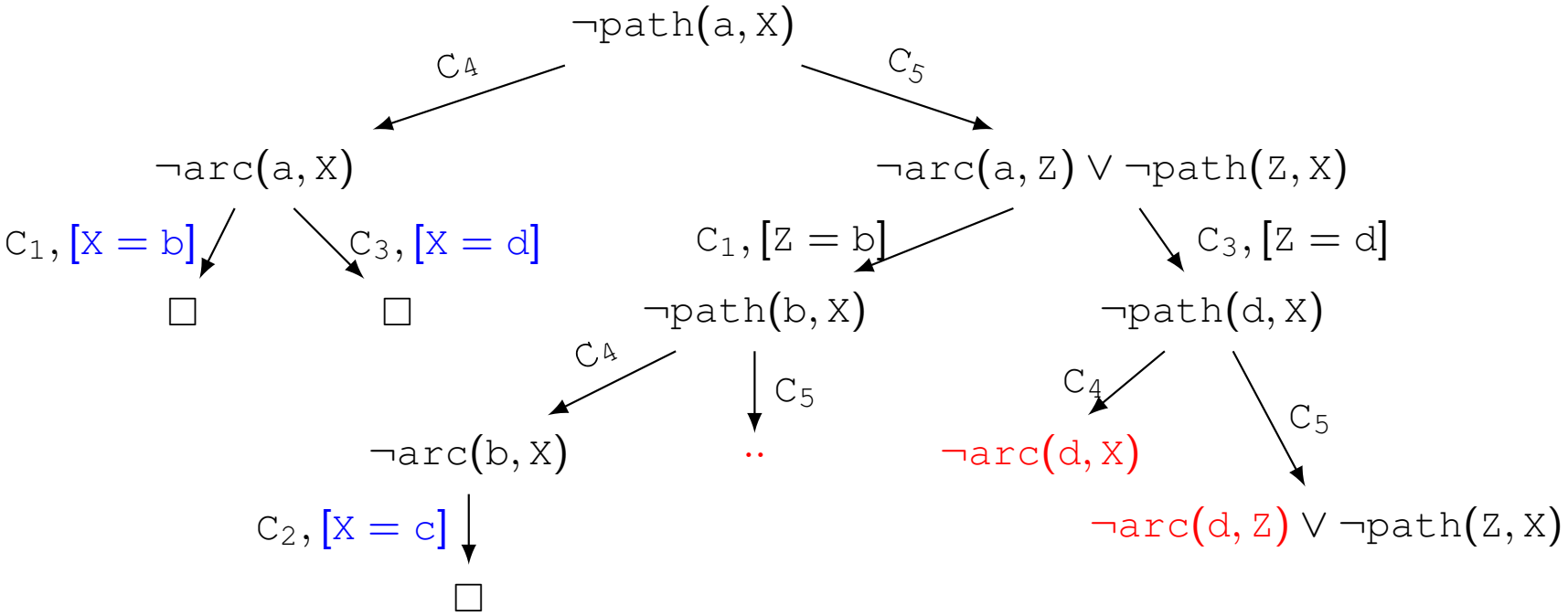
- The root is $\neg A_1 \vee \ldots \vee \neg A_n$
- Every node G has $m \geq 0$ children $G_1, \ldots, G_m$, enumerated from left to right, where
  - $G \rightarrow_{\theta_i, C_i} G_i$ and
  - In P, C_i appears before C_{i+1}, for all $1 \leq i < m$.
- There are two kinds of leaf nodes:
  1. Failure nodes: $G = \neg B_1 \vee \ldots \vee \neg B_n$ for which there is no clause $C = (K_1 \wedge \ldots \wedge K_m \rightarrow B)$ whose head B unifies with B_1.
  2. Success nodes: $\square$, which is the empty disjunction of literals. NOTE: $\square$ is the same as *false*
- The computed answers of G are obtained from the branches of the SLD-tree of G to success nodes:

  If $G \rightarrow_{\theta_1} G_1 \rightarrow_{\theta_2} \ldots \rightarrow_{\theta_n} G_n = \square$
  then $\theta = \theta_1 \theta_2 \ldots \theta_n$ is a computed answer.

Example of SLD tree

$P = \{$ arc(a, b). %C_1
arc(b, c). %C_2
arc(a, d). %C_3
path(X, Y) ← arc(X, Y). %C_4
path(X, Y) ← arc(X, Z) ∧ path(Z, Y).$\}$ %C_5

$G =$? − path(a, X).



Example of SLD tree

$P = \{$ $\mathtt{arc(a,b).}$ $\%C_1$

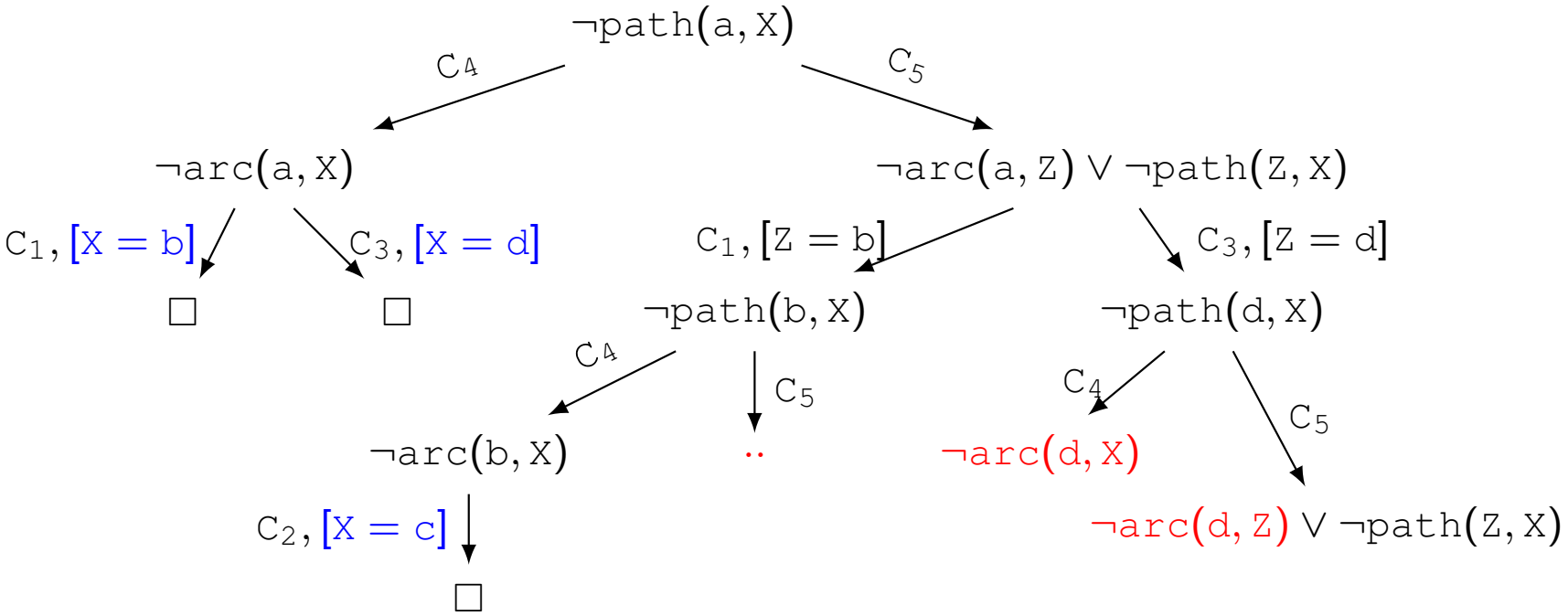
$\mathtt{arc(b,c).}$ $\%C_2$

$\mathtt{arc(a,d).}$ $\%C_3$

$\mathtt{path(X,Y)} \leftarrow \mathtt{arc(X,Y).}$ $\%C_4$

$\mathtt{path(X,Y)} \leftarrow \mathtt{arc(X,Z)} \wedge \mathtt{path(Z,Y).}\}$ $\%C_5$

$G = ? - \mathtt{path(a,X).}$



REMARK: The solutions $[X = b], [X = d], [X = c]$ are found by growing the SLD-tree from left to right.

References

1. W.F. Clocksin and C.S. Mellish: *Programming in Prolog*. 5th Edition. Springer Berlin. 2003.
2. K.R. Apt and R.N. Bol: Logic programming and negation: A survey. *Journal of Logic Programming* vol 19-20, pages 9-71. 1994.