Top-Down Execution

CS240B Notes

Notes based on Section 9.4 of *Advanced Database Systems*—Morgan Kaufmann, 1997

C. Zaniolo, March 2002

Top-Down Execution of Datalog

- A strict bottom-up execution strategy is frequently not natural nor efficient.
- Pure top-down, SLD-resolution, Prolog
- Mixing top-down and bottom-up in deductive databases

Passing Bindings from Goals to Heads

$$r_1 : \texttt{part\_weight}(\texttt{No}, \texttt{Kilos}) \leftarrow \quad \texttt{part}(\texttt{No}, \_, \texttt{actualkg}(\texttt{Kilos}))$$

$$r_2 : \texttt{part\_weight}(\texttt{No}, \texttt{Kilos}) \leftarrow \quad \texttt{part}(\texttt{No}, \texttt{Shape}, \texttt{unitkg}(\texttt{K})), \texttt{area}(\texttt{Shape}, \texttt{Area}), \texttt{Kilos} = \texttt{K} * \texttt{Area}.$$

$$r_3 : \texttt{area}(\texttt{circle}(\texttt{Dmtr}), \texttt{A}) \leftarrow \quad \texttt{A} = \texttt{Dmtr} * \texttt{Dmtr} * 3.14/4.$$

$$r_4 : \texttt{area}(\texttt{rectangle}(\texttt{Base}, \texttt{Height}), \texttt{A}) \leftarrow \quad \texttt{A} = \texttt{Base} * \texttt{Height}.$$

The goal $\texttt{area}(\texttt{Shape}, \texttt{Area})$ in rule r_1 can be viewed as a call to the procedure $\texttt{area}$ defined by rules r_3 and r_4.

Thus $\texttt{Shape}$ is instantiated to $\texttt{circle}(11)$ by the execution of r_3, and $\texttt{rectangle}(10, 20)$ and r_4.

Passing Bindings from Goals to Heads–cont.

- Instantiated to c means "assigned the value of the constant c". $\mathtt{Shape}/\mathtt{rectangle}(10, 20)$ denotes that $\mathtt{Shape}$ has been instantiated to $\mathtt{rectangle}(10, 20)$.
- Arguments can be complex; thus the passing of parameters is performed through a process known as *unification*.
- $\mathtt{Shape}/\mathtt{rectangle}(10, 20)$ is made equal to (unified to) the first argument of the second `area` rule, `rectangle(Base, Height)`, by setting $\mathtt{Base}/10$ and $\mathtt{Height}/20$.

Substitutions

Substitutions: A substitution θ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i is a distinct variable, and each t_i is a term distinct from v_i. Each t_i is called a *binding* for v_i.

The substitution θ is called a *ground substitution* if every t_i is a ground term. (Then X/θ is an instantiation of X to θ.)

Eθ denotes the result of applying the substitution θ to E; i.e., of replacing each variable with its respective binding.

For instance, if $E = p(x, y, f(a))$ and $\theta = \{x/b, y/x\}$. Then $E\theta = p(b, x, f(a))$. If $\gamma = \{x/c\}$, then $E\gamma = p(c, y, f(a))$.

Thus variables that are not part of the substitution are left unchanged.

Composition of Substitutions

Let θ = $\{u_1/s_1, \ldots, u_m/s_m\}$ and $\delta = \{v_1/t_1, \ldots, v_n/t_n\}$ be substitutions.
Then the *composition* $\theta\delta$ of θ and δ is the substitution obtained from the set

$$\{u_1/s_1\delta, \ldots, u_m/s_m\delta, v_1/t_1, \ldots, v_n/t_n\}$$

by deleting any binding $u_i/s_i\delta$ for which $u_i = s_i\delta$ and deleting any binding v_j/t_j for which $v_j \in \{u_1, \ldots, u_m\}$.

Composing Substitutions: Example

Let $\theta = \{(x/f(y), y/z)\}$ and $\delta = \{x/a, y/b, z/y\}$.
Then $\theta\delta = \{x/f(b), z/y\}$.

$$\begin{array}{ccc} \theta & \delta & \theta\delta \\ x/f(y) & x/a & x/f(b) \\ y/z & y/b & y/y \\ z/z & z/y & z/y \end{array}$$

Unification

A substitution θ *is called a* unifier *for two terms* A *and* B *if* $A\theta = B\theta$*.*

Example The two terms $p(f(x), a)$, and $p(y, f(w))$ are not unifiable, because the second arguments cannot be unified (i.e., they cannot be made identical)

The two terms $p(f(x), z)$, and $p(y, a)$ are unifiable, since

$\delta = \{y/(f(a), x/a, z/a)\}$ is a unifier.

Most General Unifier

- A unifier θ for two terms is called a *most general unifier* (mgu), if for each other unifier γ, there exists a substitution δ such that $\gamma = \theta\delta$.
- $\delta = \{y/(f(a), x/a, z/a)\}$ is not the mgu of $p(f(x), z)$, and $p(y, a)$.
  A most general unifier for these two is
  $\theta = \{y/(f(x), z/a\}$. Note that δ = $\theta\{x/a\}$.
- There exist efficient algorithms to perform unification: such algorithms either return a most general unifier or report that none exists.

Resolvent of rule r and goal g

A rule $r : A \leftarrow B_1, \ldots, B_n$, and A query goal $\leftarrow g$, r and g have no variables in common.

If $\exists$ a *most general unifier* (mgu) δ for A and g, the goal list:

$$\leftarrow B_1\delta, \ldots, B_n\delta.$$

is called *resolvent of* r *and* g.

SLD-Resolution Algorithm

Input: A first-order program P and a goal list $\mathbf{G}$.

Output: A $\mathbf{G}\delta$ that was proved from P, or *failure*.

begin Set *Res* = **G**;

While *Res* is not empty, repeat the following:

Choose a goal g from *Res*;

Choose a rule $A \leftarrow B_1, \ldots, B_n (n \geq 0)$ from P

such that A and g unify under the mgu δ,

(renaming the variables in the rule as needed);

If no such rule exists, then output *failure* and exit.

else Delete g from *Res*;

Add $B_1, \ldots, B_n$ to *Res*;

Apply δ to *Res* and **G**;

If *Res* is empty then output $\mathbf{G}\delta$

end

SLD-resolution. Example

$$\mathtt{s}(\mathtt{X},\mathtt{Y}) \leftarrow \ \mathtt{p}(\mathtt{X},\mathtt{Y}),\mathtt{q}(\mathtt{Y}).$$

$$\mathtt{p}(\mathtt{X},3).$$

$$\mathtt{q}(3).$$

$$\mathtt{q}(4).$$

1. The initial goal list is: $\leftarrow \mathtt{s}(5,\mathtt{W})$
2. This unifies the head of the first rule with mgu: $\{\mathtt{X}/5,\mathtt{Y}/\mathtt{W}\}$.
   New goal list: $\leftarrow \mathtt{p}(5,\mathtt{W}),\mathtt{q}(\mathtt{W})$
3. Say that we choose $\mathtt{q}(\mathtt{W})$ as a goal: it unifies with the fact $\mathtt{q}(3)$, under the substitution $\{\mathtt{W}/3\}$: $\leftarrow \mathtt{p}(5,3)$
   This unifies with the fact $\mathtt{p}(\mathtt{X},3)$ under the substitution $\{\mathtt{X}/5\}$.
   The goal list becomes empty and we report success.
   Thus, a top-down evaluation returns the answer $\{\mathtt{W}/3\}$ for the query $\leftarrow \mathtt{s}(5,\mathtt{W})$. from the example program. But if we choose

Examples of SLD-Resolution

Any realization of the top-down evaluation procedure will have to make two choices at each step by selecting

1. the next goal from the goal list and
2. the rule whose head unifies with the selected goal.

In general, there will be more than one goal and many rules to choose from. The choice affects the efficiency of the deduction process and also the actual result when the search falls into an infinite loop.

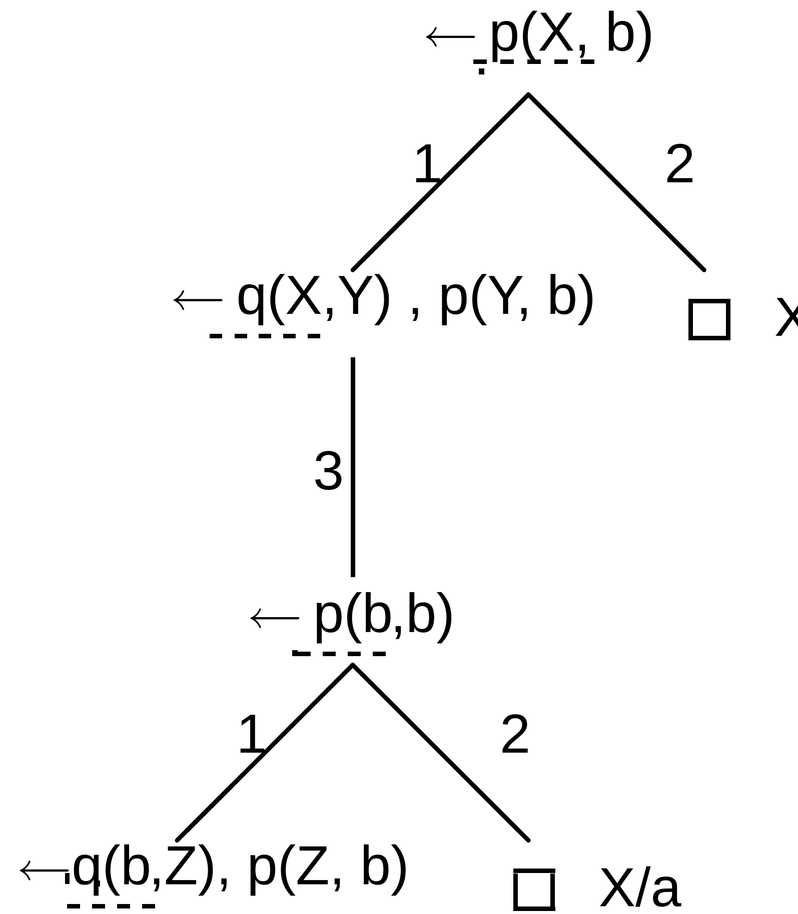
PROLOG interpreters usually choose goals in a left-to-right order and rules in a sequential order that corresponds to a depth-first search of the SLD-tree with backtracking when failure occurs. Thus, PROLOG treats the goal list as a stack onto which goals are pushed or popped, depending on success or failure.

The SLD-Refutation Procedure

Example: the goal $\leftarrow$ p(x,b) on program

1. p(x,z) $\leftarrow$ q(x,y), p(y,z)
2. p(x,x) $\leftarrow$
3. q(a,b) $\leftarrow$

A finite SLD-tree is shown at the left.
This SLD-tree comes from
the standard Prolog computation
rule that selects the leftmost atom.
Selected atoms are underlined.

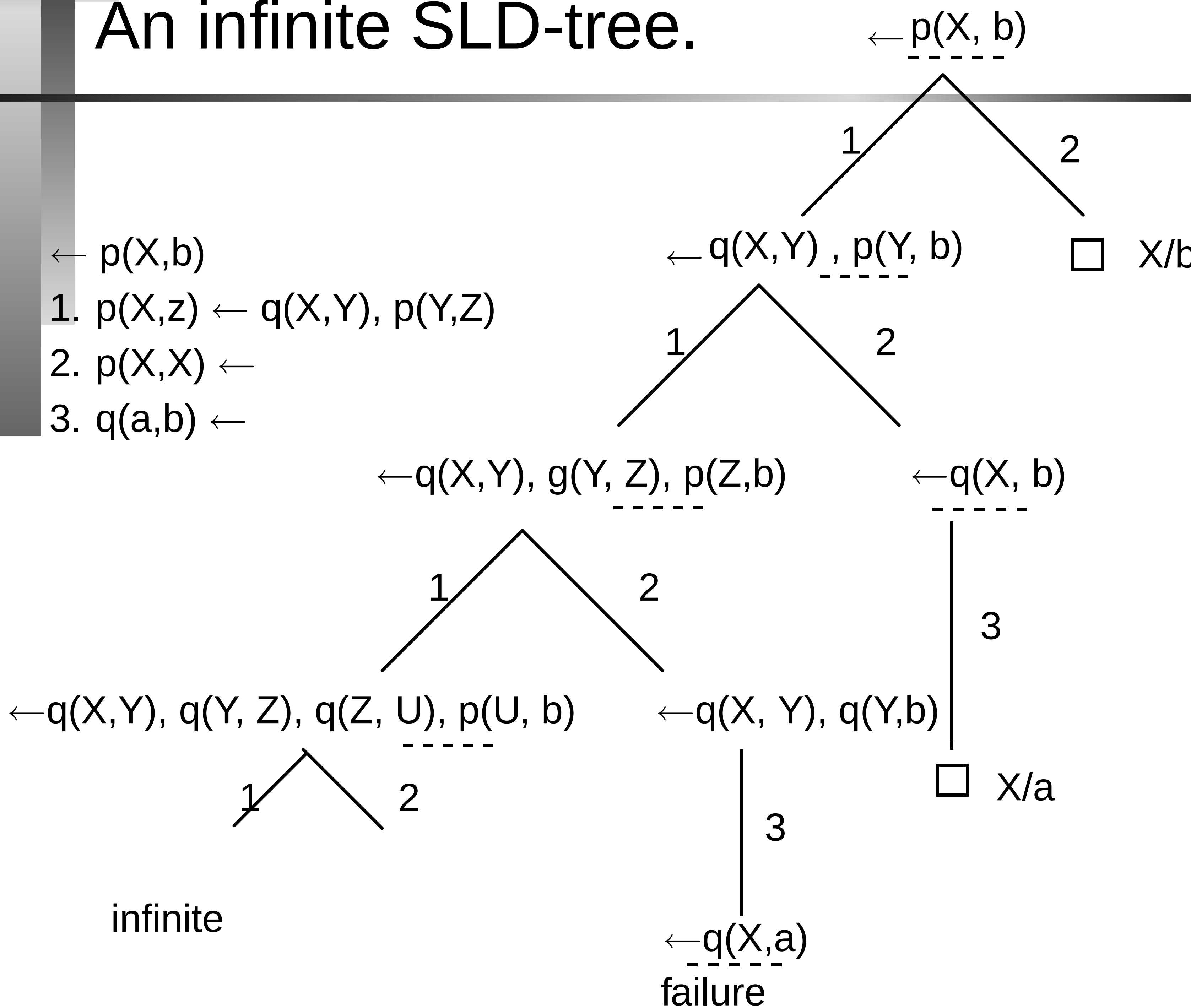


□ denotes success.

An infinite SLD-tree.

← p(X,b)

1. p(X,z) ← q(X,Y), p(Y,Z)
2. p(X,X) ←
3. q(a,b) ←



Success Set

- SLD-derivations can be *finite* or *infinite*.
  - A finite SLD-derivation can be successful or failed.
  - A *successful* SLD-derivation is a finite one that ends in the empty clause. This is also called an *SLD-refutation*.
  - A *failed* SLD-derivation is a finite one that ends in a non-empty goal, where the selected atom in this goal does not unify with the head of any program clause.
- **Definition** Let P be a program. The *success set* of P is the set of all A $\in B_P$ such that P $\cup \{\leftarrow A\}$ has an SLD-refutation (i.e., there exist some successful derivation for it).

Equivalent Semantics

Theorem: *The success set of a program is equal to its least Herbrand model.*

- Equivalence of the three formal semantics. (Least Model, Least Fixpoint, and SLD-resolution).
- SLD-resolution is a form of theorem proving (an efficient one).
- In general, generation of the success requires that all choices are visited in a breadth-first fashion. This too inefficient for practical languages such as Prolog that use depth-first instead.

Satisfiability

Let S be a set of closed formulas. We say that S is *satisfiable* when there is an interpretation that is a model for S.

S is *valid* if every interpretation of L is a model for S.

S is *unsatisfiable* if it has no models.

Theorem: Let S be a set of clauses. Then S is unsatisfiable iff S has no Herbrand models.

We say F is a logical consequence of S if, every interpretation that is a model for S is also a model for F.

Note that if $S = \{F_1, \ldots, F_n\}$ is a finite set of closed formulas, then F is a logical consequence of S iff $F_1 \wedge \ldots \wedge F_n \rightarrow F$ is valid.

Refutation

$S = \{F_1, \ldots, F_n\}$ is a finite set of closed formulas, then F is a logical consequence of S iff $F_1 \wedge \ldots \wedge F_n \rightarrow F$ is valid.

Theorem: Let S be a set of closed formulas and F be a closed formula. Then F is a logical consequence of S iff $S \cup \{\neg F\}$ is unsatisfiable.

Thus to prove a goal G from a set of rules and facts P we simply have to prove that $P \cup \{\leftarrow G\}$ is unsatisfiable—i.e., we have to refute $P \cup \{\leftarrow G\}$.

Resolution theorem proving does exactly that: It refutes the goal list.

Prolog can be viewed in that light. But, actually, there is no real refutation—just procedural composition via unification. The term SLD stands for Selected literal Linear resolution (or refutation) strategy over Definite clauses.

Prolog

- Depth-first exploration of alternatives, where *goals are always chosen in a left-to-right order and the heads of the rules are also considered in the order they appear in the program.*
- The programmer is given responsibility for ordering the rules and their goals in such a fashion as to guide Prolog into successful and efficient searches.
- The programmer must also make sure that the procedure never falls into an infinite loop.

Example: The goal $?\mathtt{anc}(\mathtt{marc}, \mathtt{mary})$ on the program:

$$\begin{aligned} \mathtt{anc}(\mathtt{X}, \mathtt{Y}) &\leftarrow \mathtt{anc}(\mathtt{X}, \mathtt{Y}), \mathtt{parent}(\mathtt{Y}, \mathtt{Z}). \\ \mathtt{anc}(\mathtt{X}, \mathtt{Z}) &\leftarrow \mathtt{parent}(\mathtt{X}, \mathtt{Y}). \end{aligned}$$

This causes an infinite loop that never returns any result.

A solution to the previous problems is to put the exit rule before the recursive one.

$$\begin{aligned}\mathtt{anc}(\mathtt{X},\mathtt{Y}) &\leftarrow \mathtt{parent}(\mathtt{X},\mathtt{Y}).\\ \mathtt{anc}(\mathtt{X},\mathtt{Z}) &\leftarrow \mathtt{anc}(\mathtt{X},\mathtt{Y}),\mathtt{parent}(\mathtt{Y},\mathtt{Z}).\end{aligned}$$

Prolog loops after the generation of all the results. To make things work `parent` must be put before `anc` in the recursive rule. A skill not hard to learn.

Cycles in the parent database will also cause problems— SLD-resolution has no memory.