

Horn Clauses and SLD Resolution

Marco Piastra

Back to Propositional Logic

Horn Clauses (in L_P)

- Definition

A **Horn Clause** is a wff in CF
that contains at most one literal in positive form

- Three types of *Horn Clauses*:

Rule: two or more literals, one positive

Examples: $\{B, \neg D, \neg A, \neg C\}$, $\{A, \neg B\}$ (equivalent to: $(D \wedge A \wedge C) \rightarrow B$, $B \rightarrow A$)

Facts: just one positive literal

Examples: $\{B\}$, $\{A\}$

Goal: one or more literals, all negative

Examples: $\{\neg B\}$, $\{\neg A, \neg B\}$

More terminology:

Rules and facts are also called *definite clauses*

Goals are also called *negative clauses*

Lost in Translation...

Many wffs can be translated into Horn clauses:

$$(A \wedge B) \rightarrow C$$

$$\neg(A \wedge B) \vee C$$

$$\neg A \vee \neg B \vee C$$

(rewriting \rightarrow)

(De Morgan - CF – it is a rule)

$$A \rightarrow (B \wedge C)$$

$$\neg A \vee (B \wedge C)$$

$$(\neg A \vee B) \wedge (\neg A \vee C)$$

$$(\neg A \vee B), (\neg A \vee C)$$

(rewriting \rightarrow)

(distributing \vee)

(CF – two rules)

$$(A \vee B) \rightarrow C$$

$$\neg(A \vee B) \vee C$$

$$(\neg A \wedge \neg B) \vee C$$

$$(\neg A \vee C) \wedge (\neg B \vee C)$$

$$(\neg A \vee C), (\neg B \vee C)$$

(rewriting \rightarrow)

(De Morgan)

(distributing \vee)

(CF – two rules)

But not all of them:

$$(A \wedge \neg B) \rightarrow C$$

$$\neg(A \wedge \neg B) \vee C$$

$$\neg A \vee B \vee C$$

(rewriting \rightarrow)

(De Morgan)

$$A \rightarrow (B \vee C)$$

$$\neg A \vee B \vee C$$

(rewriting \rightarrow)

SLD Resolution

Linear resolution with Selection function for Definite clauses

■ Algorithm

Starts from a set of *definite clauses* (also the *program*) + a *goal*

- 1) At each step, the *selection function* identifies a *literal* in the *goal* (i.e. *subgoal*)
- 2) All *definite clause* applicable to the *subgoal* are selected, in the given order
- 3) The resolution rule is applied generating the *resolvent*

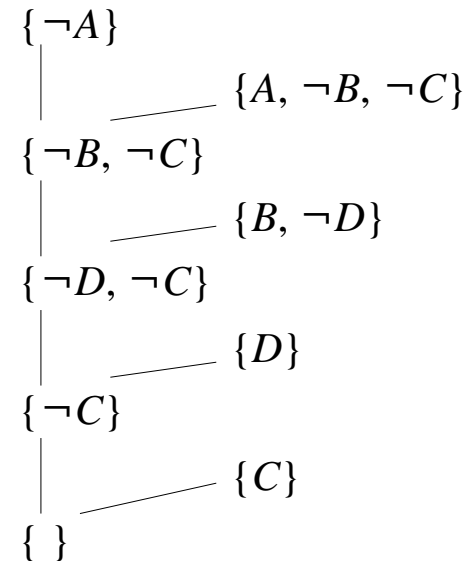
Termination: either the empty clause $\{ \}$ is obtained or step 2) fails.

Example:

Selection function: leftmost subgoal first

Definite clauses: $\{C\}$, $\{D\}$, $\{B, \neg D\}$, $\{A, \neg B, \neg C\}$

Goal: $\{\neg A\}$



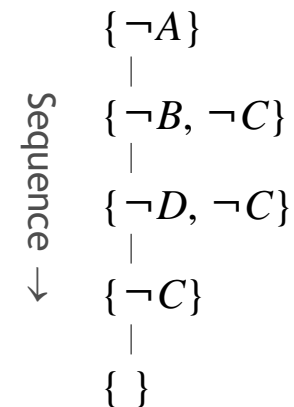
SLD trees

SLD derivations

Example: $\{C\}$, $\{D\}$, $\{B, \neg D\}$, $\{A, \neg B, \neg C\}$ goal $\{\neg A\}$

In this example each subgoal can be resolved in one mode only

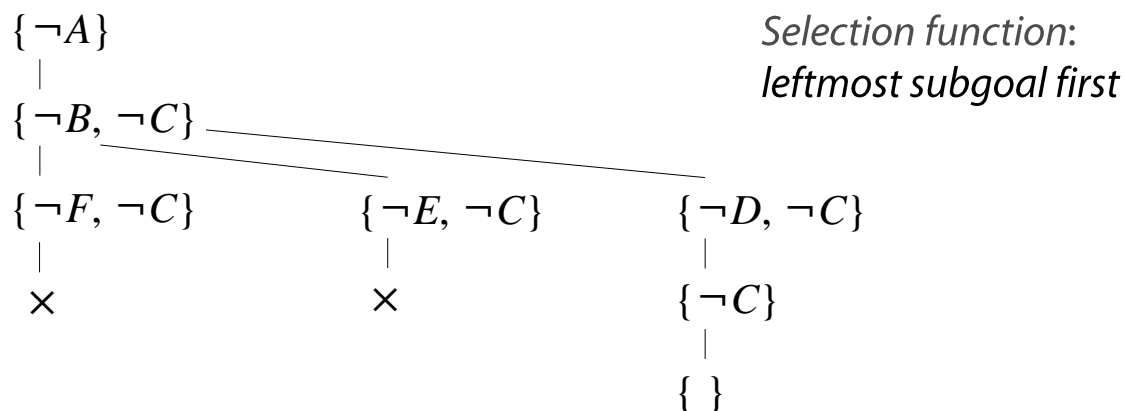
This is not true in general



- SLD trees (= trace of all SLD derivations from a goal)

Example: $\{C\}$, $\{D\}$, $\{B, \neg F\}$, $\{B, \neg E\}$, $\{B, \neg D\}$, $\{A, \neg B, \neg C\}$ goal $\{\neg A\}$

A few new rules have been added: there are now different possibilities



Each branch correspond to a possible resolution for a *subgoal*

SLD Resolution

- A resolution method for Horn clauses in L_P

It always terminates

It is *correct*: $\Gamma \vdash \varphi \Rightarrow \Gamma \models \varphi$

It is *complete*: $\Gamma \models \varphi \Rightarrow \Gamma \vdash \varphi$

- Computationally efficient

It has polynomial time complexity (w.r.t the # of propositional symbols occurring in Γ and φ)

SLD resolution in First-Order Logic

Horn Clauses in L_{FO}

The definition is very similar to the propositional case

- **Horn Clauses (of the skolemization of a set sentences)**

Each clause contains at most one literal in positive form

Facts, rules and goals

Fact: a clause with just an individual *atom*

$\{Greek(socrates)\}, \{Pyramid(x)\}, \{Sister(sally, motherOf(paul))\}$

Rule: a clause with at least two literals, exactly one in positive form

$\{Human(x), \neg Greek(x)\},$

$\forall x (Greek(x) \rightarrow Human(x))$

$\{\neg Female(x), \neg Parent(k(x),x), \neg Parent(k(y),y), Sister(x,y)\}$

$\forall x \forall y ((Female(x) \wedge \exists z (Parent(z,x) \wedge Parent(z,y))) \rightarrow Sister(x,y))$

$\{\neg Above(x,y), On(x,k(x))\}, \{\neg Above(x,y), On(j(y),y)\}$

$\forall x \forall y (Above(x,y) \rightarrow (\exists z On(x,z) \wedge \exists v On(v,y)))$

Goal: a clause containing negative literals only

$\{\neg Mortal(socrates)\}$

$\{\neg Sister(sally,x), \neg Sister(x,paul)\}$

Negation of $\exists x (Sister(sally,x) \wedge Sister(x,paul))$

SLD Resolution in L_{FO}

■ Input: a program Π and a goal ϕ

Program Π (i.e. a set of *definite clauses*: rules + facts) in some predefined linear order:

$\gamma_1, \gamma_2, \dots, \gamma_n$ (each γ_i is a *definite clause*)

Goal ϕ (i.e. a conjunction of facts in negated form), which becomes the *current goal* ψ

Note: the *selection function* for the *current goal* and *subgoal* will be discussed in the next slide

Procedure:

- 1) Select a negative literal $\neg\alpha$ (i.e. the *subgoal*) in the *current goal* ψ
- 2) Scan the program (in the predefined order) to identify a clause candidate literal γ_i
- 3) Try unifying $\neg\alpha$ and $std(\gamma_i)$ (i.e. apply the standardization of variables to α')
- 4) If there is a *unifier* σ of $\neg\alpha$ and $std(\gamma_i)$, replace the current goal with the *resolvent* of $std(\gamma_i)[\sigma]$ and $\psi[\sigma]$ (i.e. first apply σ to both $std(\gamma_i)$ and ψ and then generate the resolvent)
- 5) Then, if the *resolvent* is the empty clause, terminate with success, otherwise select a new *current goal* and resume from step 1)
- 6) Else, if the unification fails, scan the program and select a new candidate literal γ_i and resume from step 3)
- 7) Else, if there are no further clauses in the program, select a new *current goal* and resume from step 1)
- 8) If all the goals in the tree have been fully explored, terminate with failure

- *Two alternative selection functions:*

- **Depth-first** (which is the most common...)

- Always select the *most recent goal*, i.e. the *resolvent* which has been generated last, as the *current goal* ϕ
 - Then, in the current goal ϕ , select the leftmost *subgoal* $\neg\alpha$ not selected yet
 - When the current goal ϕ is fully explored and no new *resolvent* has been generated, select the next *most recent* goal in the tree (*backtracking*)

- **Breadth-first**

- Always select the *least recent* goal as the *current goal* ϕ
 - Then, in the current goal ϕ , select the leftmost *subgoal* $\neg\alpha$ not selected yet
 - When the current goal ϕ is fully explored select the next *least recent* goal in the tree

Comparison

Breadth-first is a *fair* selection function, in the sense that every possible resolution will be eventually attempted (i.e. 'it leaves nothing behind').

Depth-first tends to save memory and be more efficient, but it is NOT *fair* (more to follow)

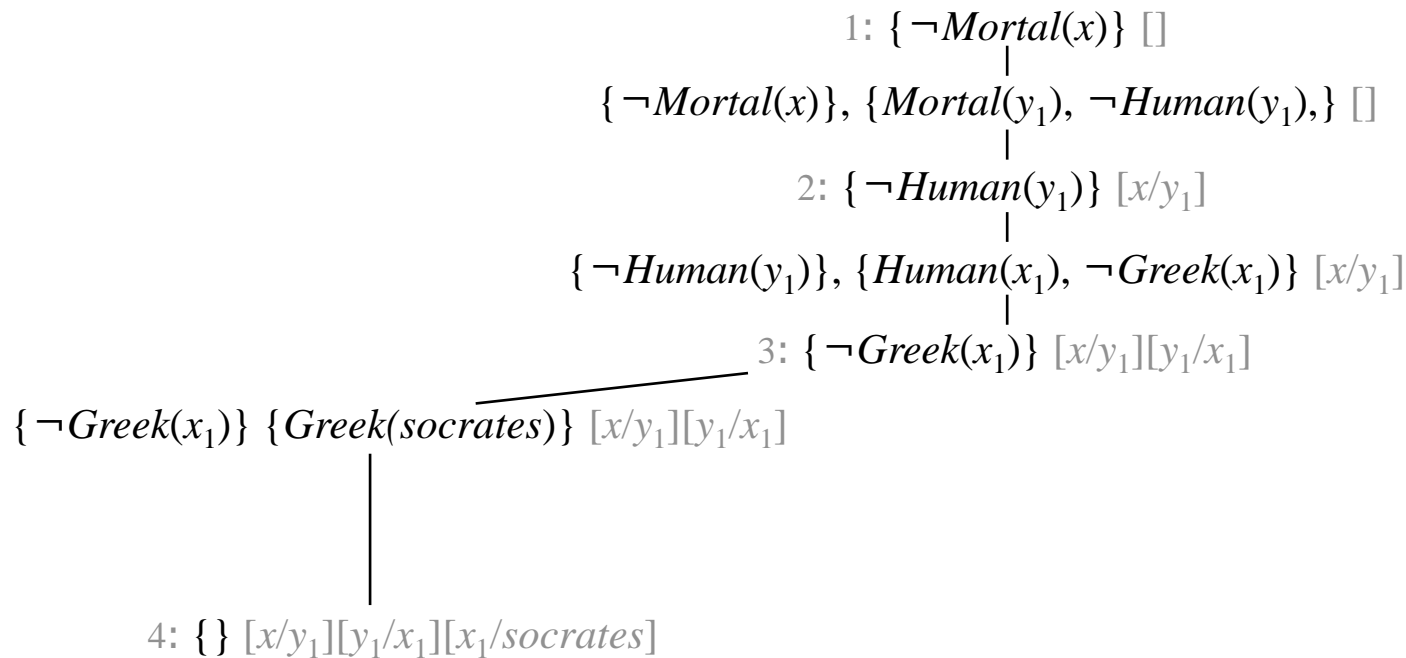
SLD Trees

- Example (depth-first selection function):

$\Pi \equiv \{ \{Human(x), \neg Greek(x)\}, \{Mortal(y), \neg Human(y)\},$
 $\{Greek(socrates)\}, \{Greek(plato)\}, \{Greek(aristotle)\} \}$

$goal \equiv \{ \neg Mortal(x) \}$

"Is there anyone who is mortal?"



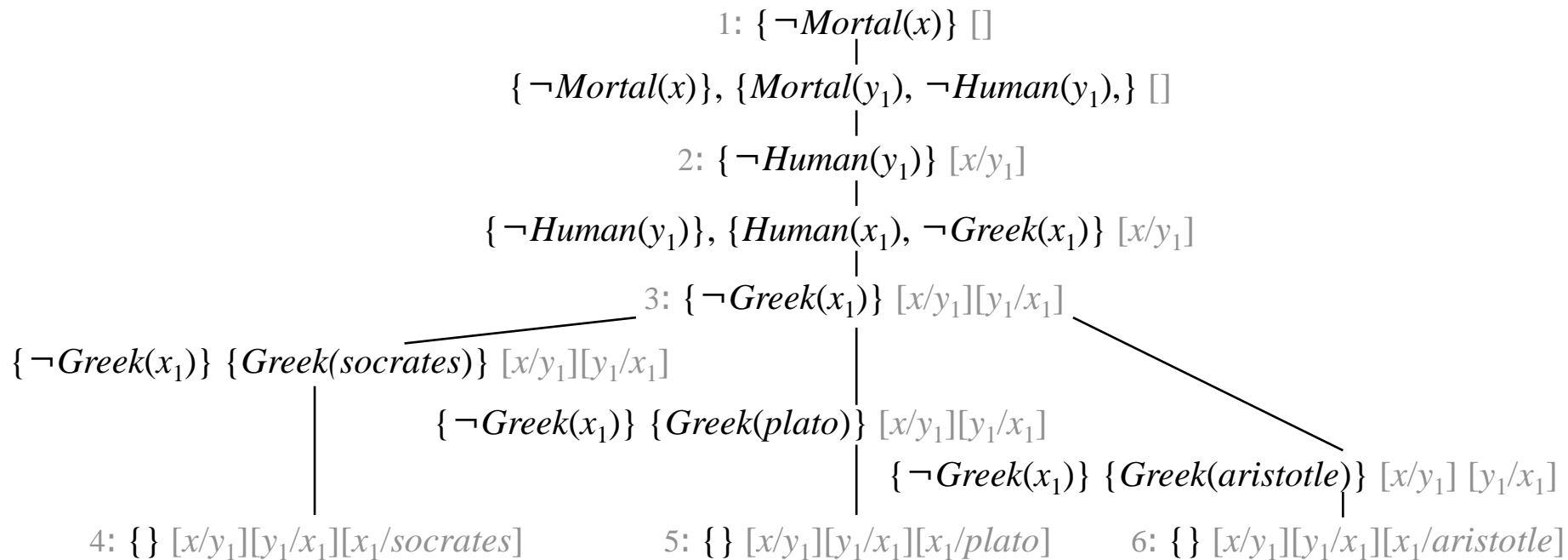
SLD Trees

- Example (depth-first selection function, forcing full exploration of SLD tree):

$\Pi \equiv \{ \{Human(x), \neg Greek(x)\}, \{Mortal(y), \neg Human(y)\},$
 $\{Greek(socrates)\}, \{Greek(plato)\}, \{Greek(aristotle)\} \}$

$goal \equiv \{ \neg Mortal(x) \}$

"Is there anyone who is mortal?"



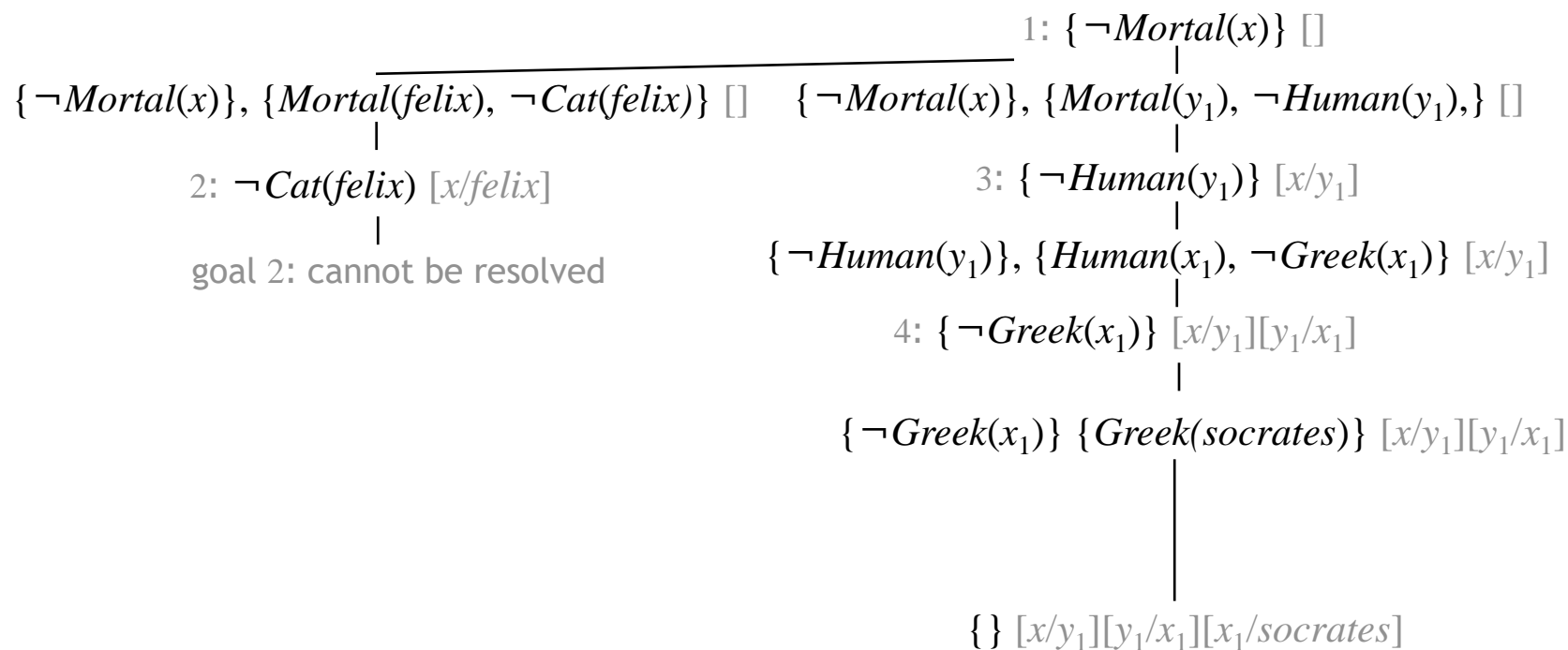
SLD Trees

- Another example (depth-first selection function):

$\Pi \equiv \{ \{ \text{Mortal}(\text{felix}), \neg \text{Cat}(\text{felix}) \}, \{ \text{Human}(x), \neg \text{Greek}(x) \}, \{ \text{Mortal}(y), \neg \text{Human}(y) \}, \{ \text{Greek}(\text{socrates}) \}, \{ \text{Greek}(\text{plato}) \}, \{ \text{Greek}(\text{aristotle}) \} \}$

$\text{goal} \equiv \{ \neg \text{Mortal}(x) \}$

"Is there anyone who is mortal?"



*The discreet charme of functions

- Representing data structures: *lists of items* $[a, b, c, \dots]$

Symbols in Σ

cons/2

it's a function that associates items (e.g. a) to a list (e.g. $[b, c]$)

$\text{cons}(a, \text{cons}(b, \text{cons}(c, \text{nil})))$ represents the list $[a, b, c]$

Append/3

it's a predicate: each pair of lists x and y is associated to their *concatenation* z

nil

it's a constant, represents the *empty list*.

Axioms (**AL**)

$\forall x \text{ Append}(\text{nil}, x, x)$

$\forall x \forall y \forall z (\text{Append}(x, y, z) \rightarrow \forall s \text{ Append}(\text{cons}(s, x), y, \text{cons}(s, z)))$

Examples of entailment

$\{\mathbf{AL} + \exists z \text{ Append}(\text{cons}(a, \text{nil}), \text{cons}(b, \text{cons}(c, \text{nil})), z) \}$

$\models \text{Append}(\text{cons}(a, \text{nil}), \text{cons}(b, \text{cons}(c, \text{nil})), \text{cons}(a, \text{cons}(b, \text{cons}(c, \text{nil}))))$

$\{\mathbf{AL} + \exists x \exists y \text{ Append}(x, y, \text{cons}(a, \text{cons}(b, \text{nil}))) \}$

$\models \text{Append}(\text{cons}(a, \text{nil}), \text{cons}(b, \text{nil}), \text{cons}(a, \text{cons}(b, \text{nil})))$

$\models \text{Append}(\text{nil}, \text{cons}(a, \text{cons}(b, \text{nil})), \text{cons}(a, \text{cons}(b, \text{nil})))$

$\models \text{Append}(\text{cons}(a, \text{cons}(b, \text{nil})), \text{nil}, \text{cons}(a, \text{cons}(b, \text{nil})))$

The world of lists

- Lists of items $[a, b, c, \dots]$

cons/2

it's a function that associates items (e.g. a) to a list (e.g. $[b, c]$)

$\text{cons}(a, \text{cons}(b, \text{cons}(c, \text{nil})))$ is the list $[a, b, c]$

Append/3

it's a predicate: each pair of lists x and y is associated to their concatenation z

nil

it's a constant, the empty list.

Shorthand notation (Prolog):

- $[] \Leftrightarrow \text{nil}$
- $[a] \Leftrightarrow \text{cons}(a, \text{nil})$
- $[a, b] \Leftrightarrow \text{cons}(a, \text{cons}(b, \text{nil}))$
- $[a/[b, c]] \Leftrightarrow \text{cons}(a, [b, c])$

Axioms (**AL**)

$\forall x \text{ Append}(\text{nil}, x, x)$

$\forall x \forall y \forall z (\text{Append}(x, y, z) \rightarrow \forall s \text{ Append}([s, x], y, [s, z]))$

The world of lists

Problem: $\forall x \text{ Append}(\text{nil}, x, x) \models \exists y \forall x \text{ Append}(\text{nil}, \text{cons}(y, x), \text{cons}(a, x))$

1: $\forall x \text{ Append}(\text{nil}, x, x), \neg \exists y \forall x \text{ Append}(\text{nil}, \text{cons}(y, x), \text{cons}(a, x))$ (refutation)

2: $\forall x \text{ Append}(\text{nil}, x, x), \forall y \exists x \neg \text{Append}(\text{nil}, \text{cons}(y, x), \text{cons}(a, x))$ (prenex normal form)

3: $\{\text{Append}(\text{nil}, x, x)\}, \{\neg \text{Append}(\text{nil}, \text{cons}(y, k(y)), \text{cons}(a, k(y)))\}$

($k/1$ is a Skolem function, clausal form)

(N.B. there is no *skolemization* in Prolog : the programmer does it)

The pair of **literals**

$\text{Append}(\text{nil}, x, x), \neg \text{Append}(\text{nil}, \text{cons}(y, k(y)), \text{cons}(a, k(y)))$

... contains the same predicate $\text{Append}/3$ but the arguments are **different**

There is however an MGU $\sigma = [x/\text{cons}(a, k(a)), y/a]$ that yields

$\{\text{Append}(\text{nil}, \text{cons}(a, k(a)), \text{cons}(a, k(a)))\}, \{\neg \text{Append}(\text{nil}, \text{cons}(a, k(a)), \text{cons}(a, k(a)))\}$

From this, the resolvent is the empty clause.

The world of lists in Prolog

```
% Identical to built-in predicate append/3, although it uses "cons"  
% as a defined predicate, thus allowing trace-ability.
```

```
append(cons(S,X),Y,cons(S,Z)) :- append(X,Y,Z).  
append(nil,X,X).
```

```
% WARNING: express your queries with cons. Examples:
```

```
% ?- append(cons(a,nil), cons(b,cons(c, nil)),cons(a,cons(b,cons(c, nil)))).  
% ?- append(X,Y,cons(a,cons(b,cons(c, nil)))).
```

Infinite SLD Trees (*fairness of SLD*)

- An example:

$\Pi \equiv \{\{S(a,b)\}, \{S(b,c)\}, \{S(x,z), \neg S(x,y), \neg S(y,z)\}\}$

$\neg\phi \equiv \{\neg S(a,x)\}$

goal: $\neg S(a,x)$ []
|
 $\{\neg S(a,x)\}, \{S(a,b)\}$ []
|
 $\{\}$ [x/b]

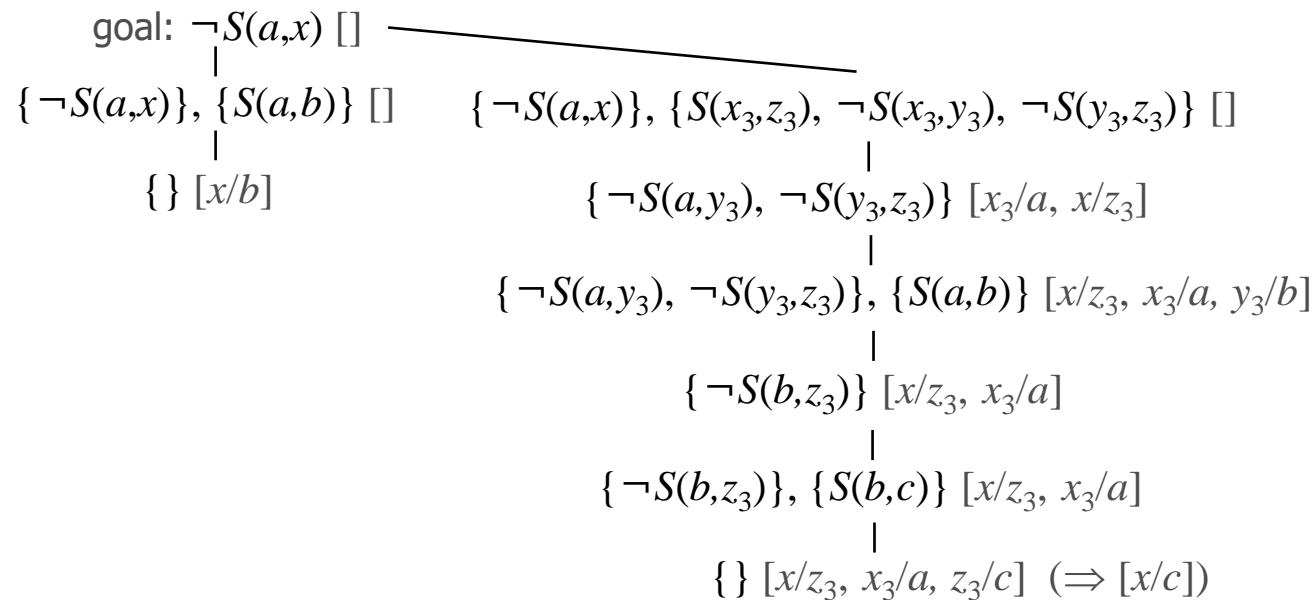
Easy...

Infinite SLD Trees (*fairness of SLD*)

- An example:

$$\Pi \equiv \{\{S(a,b)\}, \{S(b,c)\}, \{S(x,z), \neg S(x,y), \neg S(y,z)\}\}$$

$$\neg\phi \equiv \{\neg S(a,x)\}$$



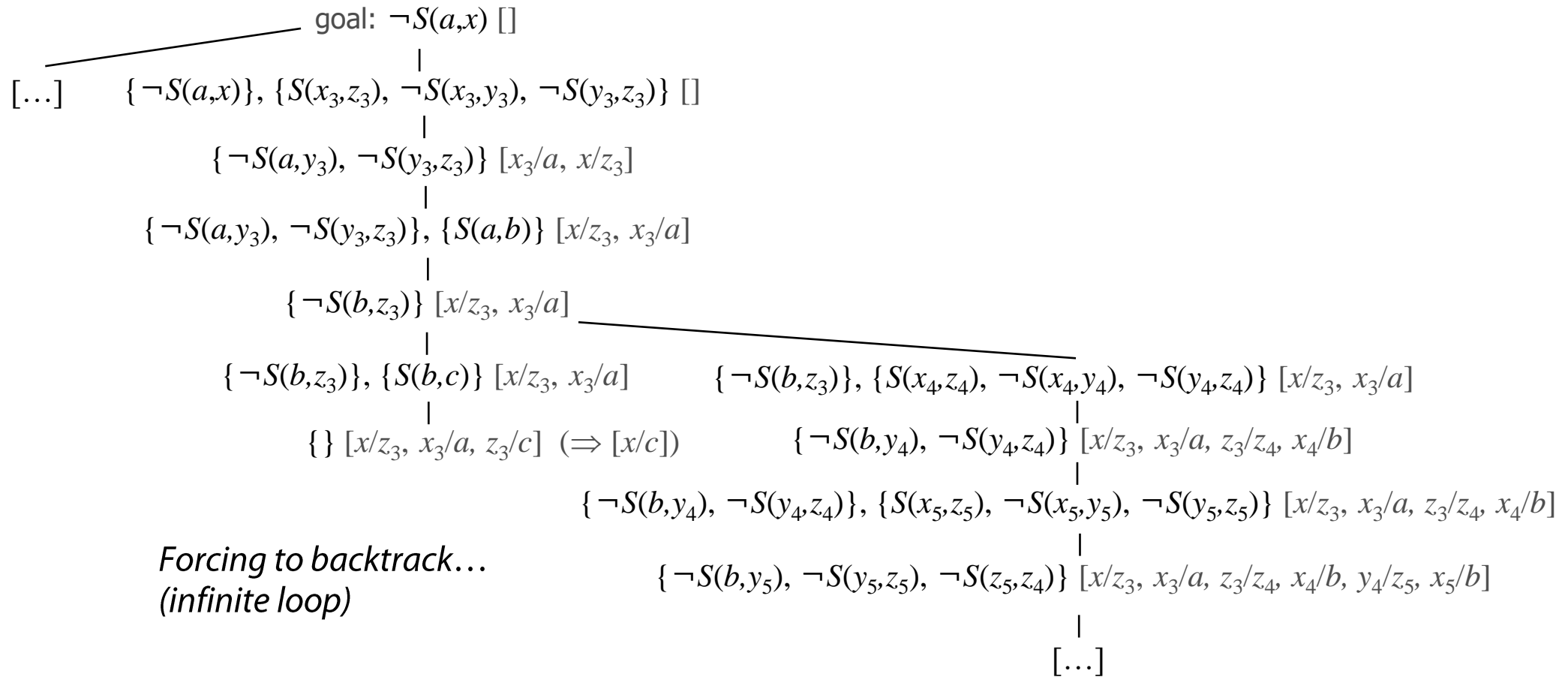
Forcing to backtrack...
(easy again)

Infinite SLD Trees (*fairness of SLD*)

- An example:

$$\Pi \equiv \{ \{S(a,b)\}, \{S(b,c)\}, \{S(x,z), \neg S(x,y), \neg S(y,z)\} \}$$

$$\neg\phi \equiv \{ \neg S(a,x) \}$$



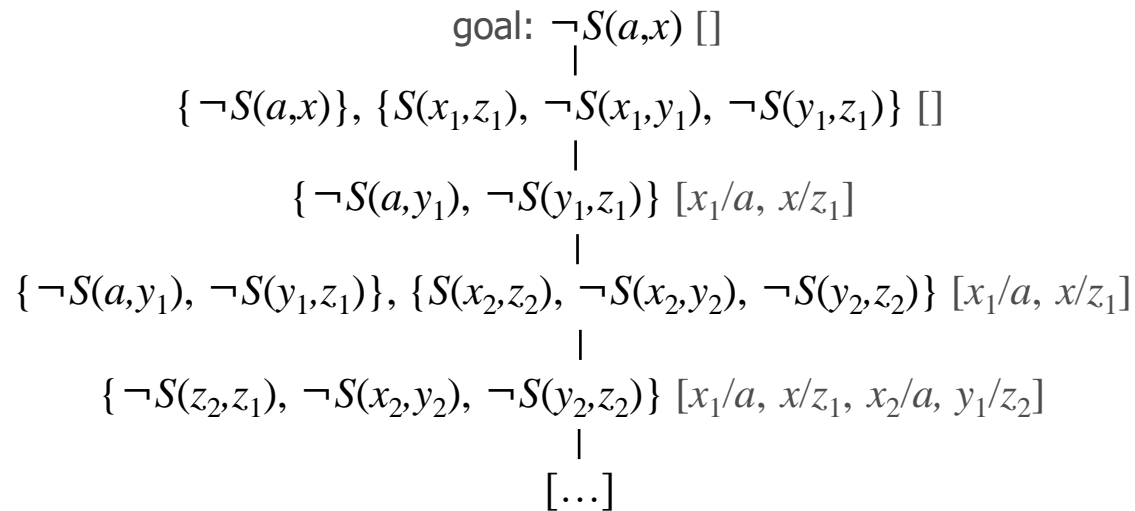
Infinite SLD Trees (*fairness of SLD*)

- A second example:

$$\Pi \equiv \{\{S(x,z), \neg S(x,y), \neg S(y,z)\}, \{S(a,b)\}, \{S(b,c)\}\}$$

$$\neg\phi \equiv \{\neg S(a,x)\}$$

Notice the change in clause ordering.....



The *infinite loop* occurs immediately ...

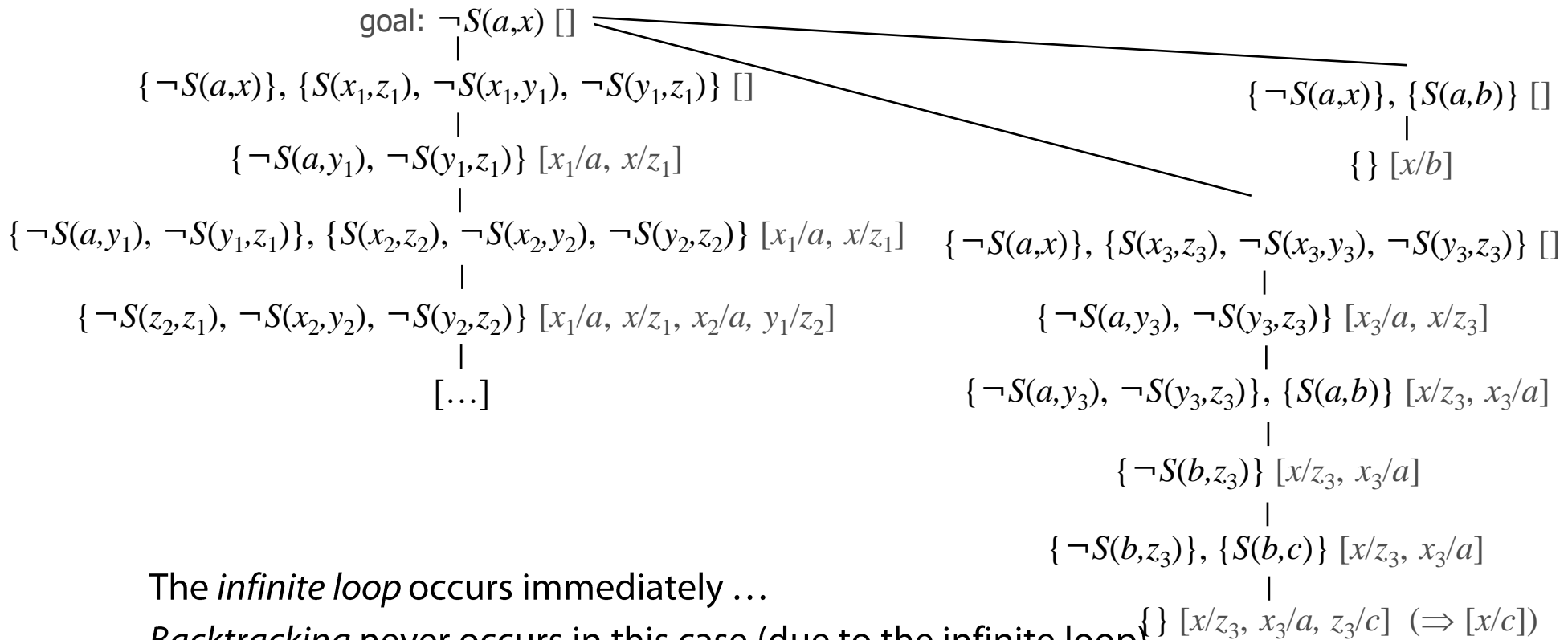
Infinite SLD Trees (*fairness of SLD*)

- A second example:

$$\Pi \equiv \{\{S(x,z), \neg S(x,y), \neg S(y,z)\}, \{S(a,b)\}, \{S(b,c)\}\}$$

$$\neg\phi \equiv \{\neg S(a,x)\}$$

Notice the change in clause ordering.....



The *infinite loop* occurs immediately ...

Backtracking never occurs in this case (due to the infinite loop), yet, if it occurred it would have produced the two correct results

Infinite SLD Trees (*fairness of SLD*)

■ Moral

- In both previous examples the infinite loop (i.e. *divergence*) is unavoidable
- Yet in the first one, the method first produces the right results and then diverges
- So in the first case the result is *complete* (i.e. all entailed formulae are derived) while in the second case the method is not

A ***fair*** selection function is such that no possible resolution will be postponed indefinitely: that is, any possible resolution will be performed, eventually.

In the two previous examples, we used a *depth-first* exploration method of the SLD tree: which is not complete (in the above sense)

A *breadth-first* exploration method is ***fair*** hence it is complete (in the above sense)

In actual programming systems (e.g. Prolog) the depth-first is preferred for memory efficiency since the breadth-first method forces to keep (most of) the whole SLD tree in memory