Knowledge Representation and Reasoning with First Order Logic

Module 3

Deepak Khemani

The Syllabus

Introduction: Overview and Historical Perspective

First Order Logic: A logic with quantified variables.

Module 1 (2 hours): Syntax, Semantics, Entailment and Models, Proof Systems, Knowledge Representation.

Module 2 (2 hours): Skolemization, Unification, Deductive Retrieval, Forward Chaining, Backward Chaining

Module 3 (2 hours): Resolution Refutation in FOL, Horn Clauses and Logic Programming

Text book

Deepak Khemani. A First Course in Artificial Intelligence (Chapters 12 & 13), McGraw Hill Education (India), 2013.

A not so easy problem

Given the following knowledge base (in list notation)

{(O A B), (O B C), (not (M A)), (M C)}

What is the KB talking about? What is the semantics?

Depends upon the interpretation ϑ = <D, I> !

Two interpretations….

Recap

Interpretation 1

{(O A B), (O B C), (not (M A)), (M C)}

Domain: Blocks World

Predicate symbols

I(O) = On

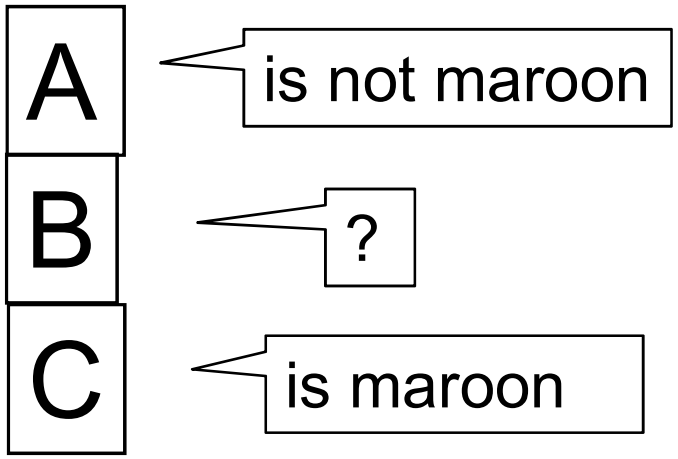
I(M) = Maroon

Constant Symbols

A, B, C → blocks

A is on B

B is on C



Recap

Interpretation 2

{(O A B), (O B C), (not (M A)), (M C)}

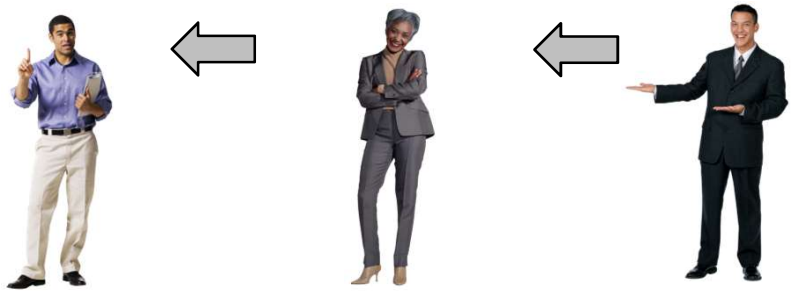
Domain: People

Predicate symbols
I(O) = LookingAt
I(M) = Married

Constant Symbols
I(A) = Jack
I(B) = Anne
I(C) = John

Anne is looking at John

Jack is looking at Anne



John

is married

Anne

?

Jack

is not married

Recap

The Goal

{(O A B), (O B C), (not (M A)), (M C)}

Given the KB and the goal

(exists (x y) (and (O x y) (not (M x)) (M y)))

or equivalently (and (O ?x ?y) (not (M ?x)) (M ?y))

…is clearly entailed

Interpretations are,

Blocks World: *Is there a not-maroon block on a maroon block?*

People: *Is a not-married person looking at a married one?*

Recap

Incompleteness of Backward and Forward Chaining

Given the KB,

{(O A B), (O B C), (not (M A)), (M C)}

And the Goal,

(and (O ?x ?y) (not (M ?x)) (M ?y))

Neither Forward Chaining nor Backward Chaining
is able to generate a proof.

Both are Incomplete!

Next, we look at a proof method,
the Resolution Refutation System,
that is Sound and Complete for FOL

Recap

Resolution Method in Propositional Logic

The resolution method requires that the formula is in conjunctive normal form (*CNF*). A formula *F* is in *CNF* if it has the following structure.

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

That is the formula is a conjunction of clauses where each clause C_i is a disjunction of literals,

$$C_i = D_{i1} \vee D_{i2} \vee \ldots \vee D_{ik(i)}$$

Each literal is either a proposition or the negation of a proposition. A formula in *CNF* is also conventionally represented as a set of sets as follows.

$$F = \{\{D_{11}, D_{12}, \ldots, D_{1k(1)}\}, \ldots, \{D_{n1}, D_{n2}, \ldots, D_{nk(n)}\}\}$$

Any formula → CNF

Any formula in propositional logic may be converted into *CNF* by the use of substitution rules based on the tautological equivalences. Conversion into *CNF* generally results in an increase in the size of the formula, often reaching exponential number of clauses in the number of propositions.

$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of $\vee$
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of $\wedge$
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	DeMorgan's Law
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	DeMorgan's Law
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of $\wedge$ over $\vee$
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of $\vee$ over $\wedge$
$(\alpha \supset \beta) \equiv (\neg\beta \supset \neg\alpha)$	contrapositive
$(\alpha \supset \beta) \equiv (\neg\alpha \vee \beta)$	implication

The Resolution Rule

The single rule used in the refutation method, called the resolution rule, takes two clauses that have a complimentary literal as follows.

From: $R_1 \vee R_2 \ldots \vee R_k \vee Q$

And: $P_1 \vee P_2 \ldots \vee P_m \vee \neg Q$

Infer: $R_1 \vee R_2 \ldots \vee R_k \vee P_1 \vee P_2 \ldots \vee P_m$

Both the literal Q and its negation $\neg Q$ are removed and the remaining literals are combined to form a new clause, called the *resolvent*. With the smaller clause the rule becomes,

From: $R \vee Q$

And: $P \vee \neg Q$

Infer: $R \vee P$

Validity of the Resolution Rule

The validity of the resolution rule can be established by showing that adding the resolvent does not change the set of clauses logically. That is, the sets before and after are equivalent. It suffices to show that,

$$((R \vee Q) \wedge (P \vee \neg Q)) \equiv ((R \vee Q) \wedge (P \vee \neg Q) \wedge (R \vee P))$$

Deduction Theorem

Given a set of premises $\{\alpha_1, \alpha_2, \ldots, \alpha_N\}$ and the desired goal β, we want to determine if the formula,

$$((\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \supset \beta)$$

is true. This follows from the well known Deduction theorem that asserts that,

$$\{\alpha_1, \alpha_2 \ldots, \alpha_n\} \models \beta \quad \text{iff} \quad \models ((\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \supset \beta)$$

$((\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \supset \beta)$ is a *tautology*

iff $\neg((\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \supset \beta)$ is *unsatisfiable*

Proof by Contradiction

Show that $\neg((\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \supset \beta)$ is unsatisfiable. We convert this into *CNF*, the form that is required by the resolution method.

$$\neg((\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \supset \beta) \equiv \neg(\neg(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \vee \beta)$$

$$\equiv ((\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \wedge \neg\beta)$$

$$\equiv (\alpha'_1 \wedge \alpha'_2 \wedge \ldots \wedge \alpha'_n \wedge \neg\beta')$$

To generate the input for the resolution method we simply need to negate the goal and add it to the set of clauses. We may need to convert each of the premises and the negated goal into the clause (*CNF*) form denoted by α'_i and β' in the last line of the transformation.

The basic algorithm

Given a set of clauses, pick any two clauses and add the resolvent to the set.

If at any time we generate a resolvent that is the empty clause (or null clause) then the procedure can terminate.

This is because the empty clause, or $\perp$, evaluates to false. We also use the symbol $\square$ to stand for the empty or null clause.

An empty clause can be generated from two clauses S and $\neg S$ by the application of the resolution rule.

Now if a conjunctive formula (the database) has both S and $\neg S$ then it must be false

Show that a formula is unsatisfiable

Let the original formula be $\{P_1, P_2, \ldots, P_N\}$. Remember this stands for a conjunction of the N clauses. To this we add a sequence of resolvents $R_1, R_2, R_3, \ldots$ culminating with $\perp$. The databases at all stages are logically equivalent, because the resolution rule is sound.

$$\{P_1, P_2, \ldots, P_N\} \equiv \{P_1, P_2, \ldots, P_N, R_1\}$$
$$\equiv \{P_1, P_2, \ldots, P_N, R_1, R_2\}$$
$$\equiv \{P_1, P_2, \ldots, P_N, R_1, R_2, R_3\}$$
$$\equiv \{P_1, P_2, \ldots, P_N, R_1, R_2, R_3, \ldots, \perp\}$$

Now since the last set of clauses evaluates to *false* (because it contains the empty clause) the set we started with, which is logically equivalent, also evaluates to *false*. Thus $\{P_1, P_2, \ldots, P_N\}$ is *false*.

The Resolution Refutation Method

The Resolution Refutation method was devised by Alan Robinson in 1965. He showed that the Resolution Refutation method is complete for deriving the null clause.

A proof by resolution method is a *proof by contradiction*. We start with the set of premises, *negate* the goal and add it as another clause, and show that it *leads to a contradiction* (something that is false or not possible).

If the input formula is unsatisfiable there always exists a derivation of the null clause.

The Alice problem revisited

Recap

Atomic sentences in Propositional Logic can stand for anything. Consider,

Alice likes mathematics and she likes stories. If she likes mathematics she likes algebra. If she likes algebra and likes physics she will go to college. She does not like stories or she likes physics. She does not like chemistry and history.

Encoding: P = Alice likes mathematics. Q = Alice likes stories. R = Alice likes algebra. S = Alice likes physics. T = Alice will go to college. U = Alice likes chemistry. V = Alice likes history.

Then the given facts are,

$(P \wedge Q)$

$(P \supset R)$

$((R \wedge S) \supset T)$

$(\sim Q \vee S)$

$(\sim U \wedge \sim V)$

If the above sentences are true is it necessarily true that "Alice will go to college"?

That is " Is T *true*?"

We answer this by producing a proof (of T)

Formulating the problem

To show that the following formula is a tautology.

$$(((P \wedge Q) \wedge (P \supset R) \wedge ((R \wedge S) \supset T) \wedge (\neg Q \vee S) \wedge (\neg U \wedge \neg V)) \supset T)$$

We take each of the premises and convert it to a clause. The first premise ($P \wedge Q$) gives us two clauses, as does the last one. We also add the negation of the goal $\neg T$ as a clause.

Deriving the null clause

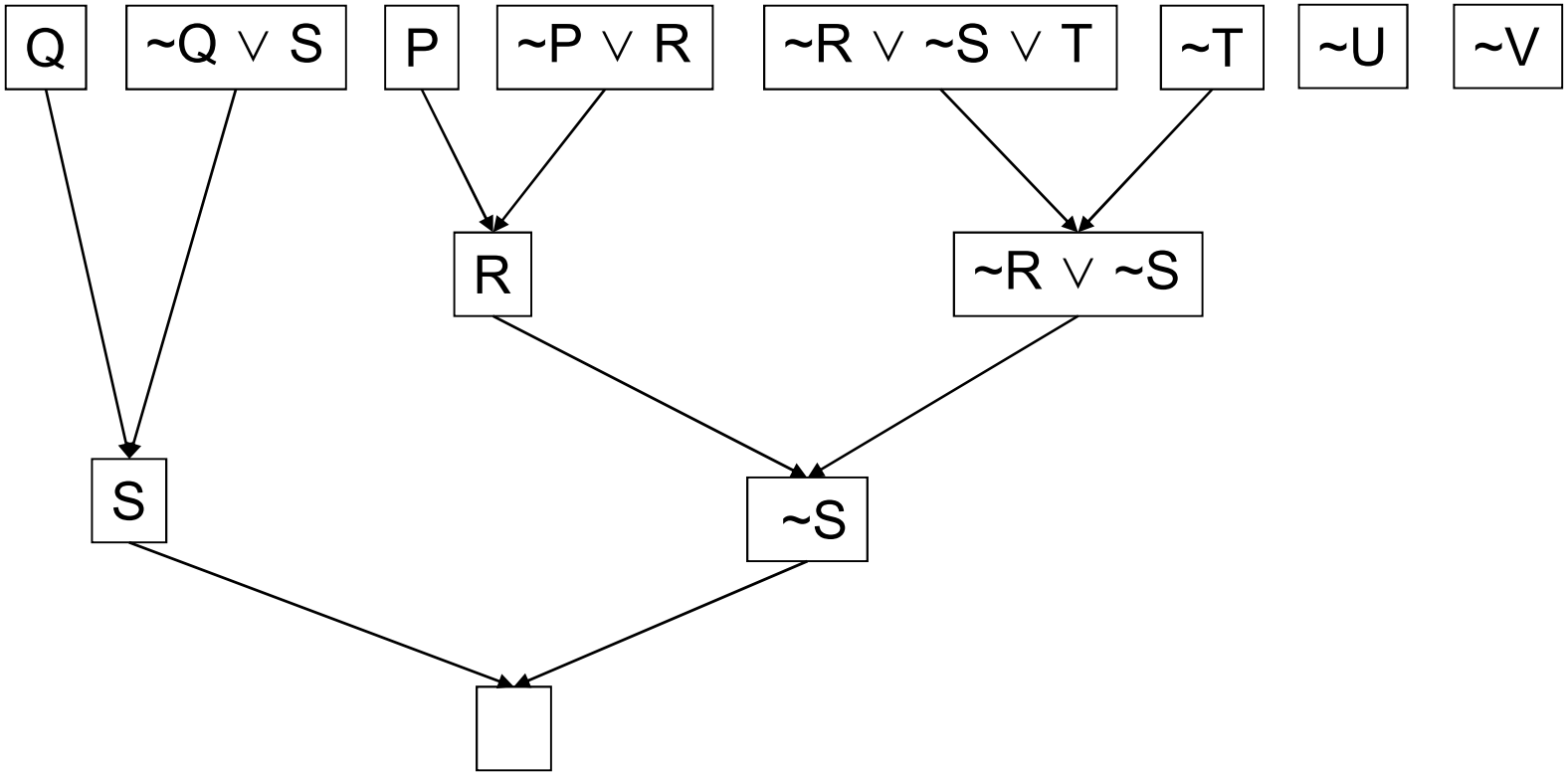
1. P
2. Q
3. $\neg P \vee R$
4. $\neg R \vee \neg S \vee T$
5. $\neg Q \vee S$
6. $\neg U$
7. $\neg V$
8. $\neg T$ negated goal

The resolvents are,

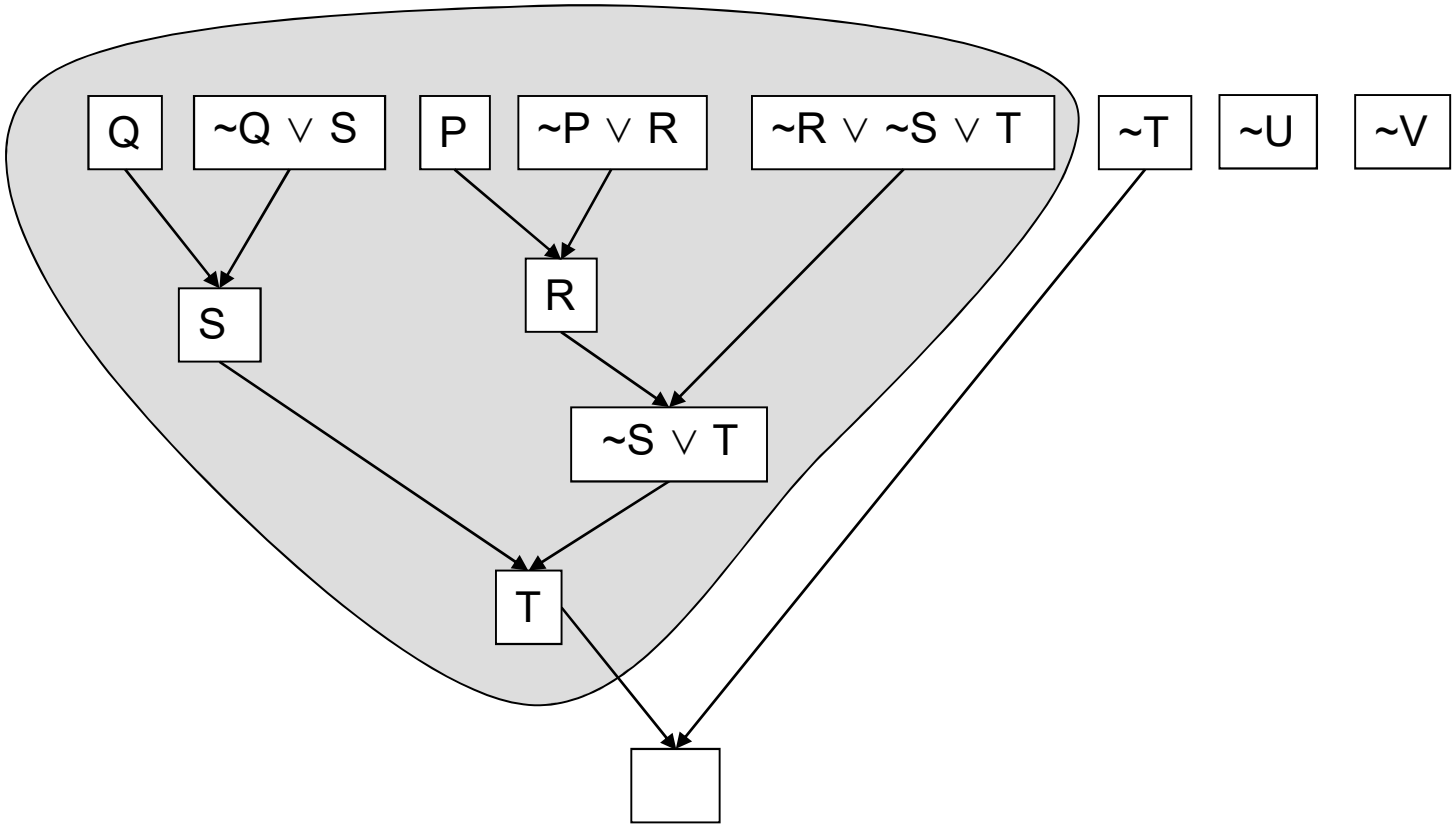
9. $\neg R \vee \neg S$ from 4, 8
10. R from 1, 3
11. $\neg S$ from 9, 10
12. $\neg Q$ from 11, 5
13. $\square$ from 2, 12

q.e.d

The Proof as a Directed Acyclic Graph (DAG)



An alternative derivation



Clause form in FOL

A first order formula is in clause form if it is of the following form,

$$\forall x_1 \forall x_2 \ldots \forall x_V (C_1 \wedge C_2 \ldots \wedge C_N)$$

where each C_i is a *clause* made of disjunction of *literals*, and each literal is an atomic formula or its negation

The clause form contains only universal quantifiers. The consequence of having only universal quantifiers and all of them bunched up in the left is that one can in fact ignore the quantifiers during processing. That makes writing programs a little bit simpler.

Converting to Clause Form

It was shown by Thoralf Skolem that every formula can be converted into the clause form. Take the existential closure of α. This ensures that there are no free variables in the formula.

1. Standardize variables apart across quantifiers. Rename variables so that the same symbol does not occur in different quantifiers.
2. Eliminate all occurrences of operators other than $\wedge$, $\vee$, and $\neg$.
3. Move $\neg$ all the way in.
4. Push the quantifiers to the right. This ensures that their scope is as tight a possible.
5. Eliminate $\exists$.
6. Move all $\forall$ to the left. They can be ignored henceforth.
7. Distribute $\wedge$ over $\vee$.
8. Simplify
9. Rename variables in each clause (disjunction).

Conversion: an example

$(girl(y) \wedge \forall x\ (boy(x) \supset likes(x, y)) \vee \exists x \exists z\ (boy(x) \wedge girl(z) \wedge loves(x,z))$

1. $\exists y((girl(y) \wedge \forall x\ (boy(x) \supset likes(x, y)) \vee \exists x \exists z(boy(x) \wedge girl(z) \wedge loves(x,z)))$
2. $\exists y((girl(y) \wedge \forall x(boy(x) \supset likes(x,y)) \vee \exists x_1 \exists z(boy(x_1) \wedge girl(z) \wedge loves(x_1,z)))$
3. $\exists y((girl(y) \wedge \forall x(\neg boy(x) \vee likes(x,y)) \vee \exists x_1 \exists z(boy(x_1) \wedge girl(z) \wedge loves(x_1,z)))$
4. no change
5. no change
6. $(girl(sk\text{-}y) \wedge \forall x(\neg boy(x) \vee likes(x,sk\text{-}y))) \vee (boy(sk\text{-}x_1) \wedge girl(sk\text{-}z) \wedge loves(sk\text{-}x_1,sk\text{-}z)))$

(girl(y)∧∀x (boy(x)⊃likes(x, y))∨∃x∃z (boy(x)∧girl(z)∧loves(x,z))

7.(∀x((girl(sk-y)∧(¬boy(x)∨likes(x,sk-y))∨(boy(sk-x_1) ∧girl(sk-z)∧loves(sk-x_1,sk-z)))

8. (girl(sk-y)∨boy(sk-x_1)) ∧ (girl(sk-y)∨girl(sk-z)) ∧ (girl(sk-y)∨loves(sk-x_1,sk-z)) ∧ (¬boy(x)∨likes(x,sk-y)∨(boy(sk-x_1)) ∧ (¬boy(x)∨likes(x,sk-y)∨girl(sk-z)) ∧ (¬boy(x)∨likes(x,sk-y)∨loves(sk-x_1,sk-z))

9. no change

10.(girl(sk-y)∨boy(sk-x_1)) ∧ (girl(sk-y)∨girl(sk-z)) ∧ (girl(sk-y)∨loves(sk-x_1,sk-z)) ∧ (¬boy(x)∨likes(x,sk-y)∨(boy(sk-x_1)) ∧ (¬boy(x_5)∨likes(x_5,sk-y)∨girl(sk-z)) ∧ (¬boy(x_6)∨likes(x_6,sk-y)∨loves(sk-x_1,sk-z))

Resolution Refutation for FOL

As with the propositional logic the procedure for finding a proof by the resolution refutation method is as follows,

1. Convert each premise into clause form
2. Negate the goal and convert it into clause form
3. Add the negated goal to the set of clauses
4. Choose two clauses such that two opposite signed literals in them can be unified
5. Resolve the two clauses using the MGU and add the resolvent to the set
6. Repeat steps 4-5 till a null resolvent is produced

The Resolution Rule for FOL

For the sake of completeness the resolution rule is defined as follows. A literal is an atomic formula. A clause is a disjunction of literals. Let C_i and C_k be two clauses with the structure,

$$C_i = (L_1 \vee L_2 \vee \ldots \vee L_k \vee P_1 \vee P_2 \vee \ldots \vee P_n)$$

$$C_k = (\neg R_1 \vee \neg R_2 \vee \ldots \vee \neg R_s \vee Q_1 \vee Q_2 \vee \ldots \vee Q_t)$$

If θ is the MGU for $\{ L_1, L_2, \ldots , L_k, R_1, R_2, \ldots, R_s\}$ then we can resolve C_i and C_k to give us the resolvent,

$$(P_1\theta \vee P_2\theta \vee \ldots \vee P_n\theta \vee Q_1\theta \vee Q_2\theta \vee \ldots \vee Q_t\theta)$$

That is we throw away all the positive literals L_j and negative literals $\neg R_i$, and combine the remainder after applying the substitution θ.

The Socratic argument

The three clauses for the Socratic argument are,

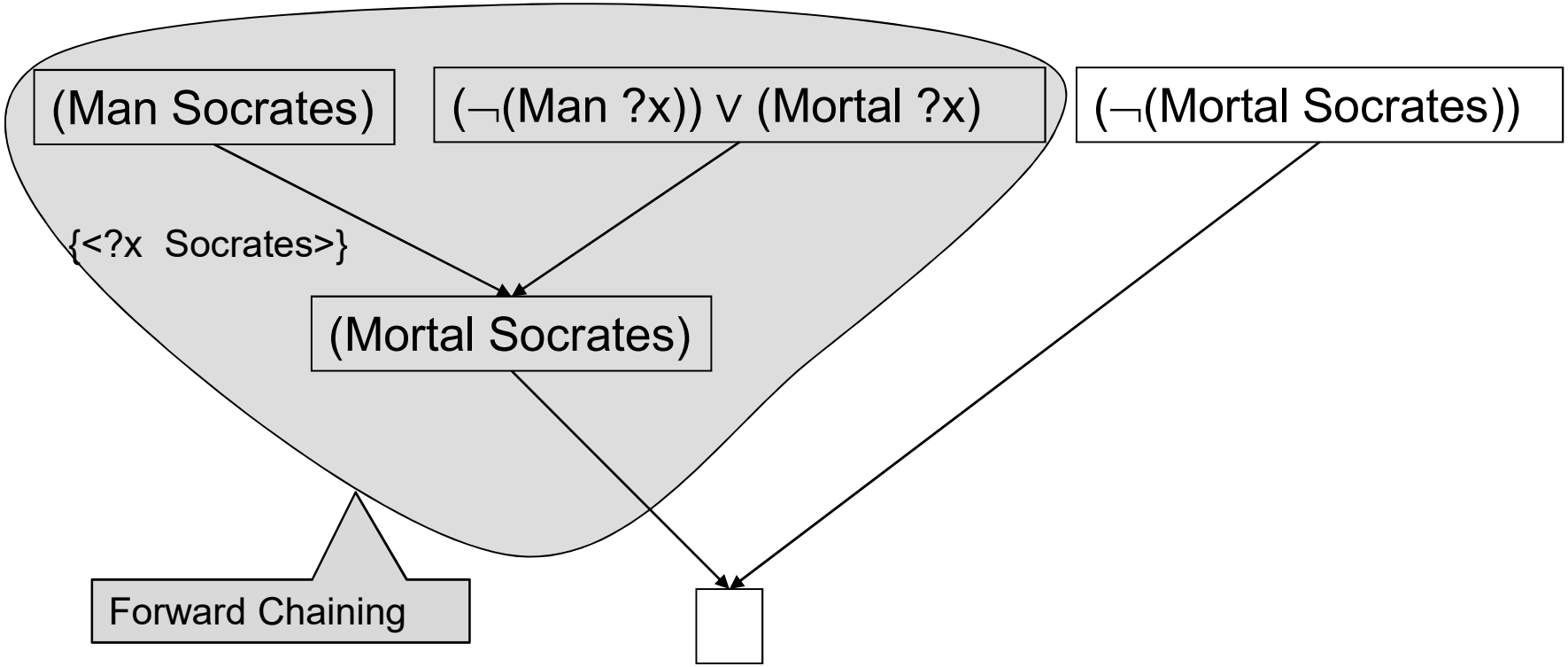
C_1	=	$(\neg(\text{Man ?x})) \vee (\text{Mortal ?x})$	premise
C_2	=	(Man Socrates)	premise
C_3	=	$(\neg(\text{Mortal Socrates}))$	negated goal

The following resolvents are generated,

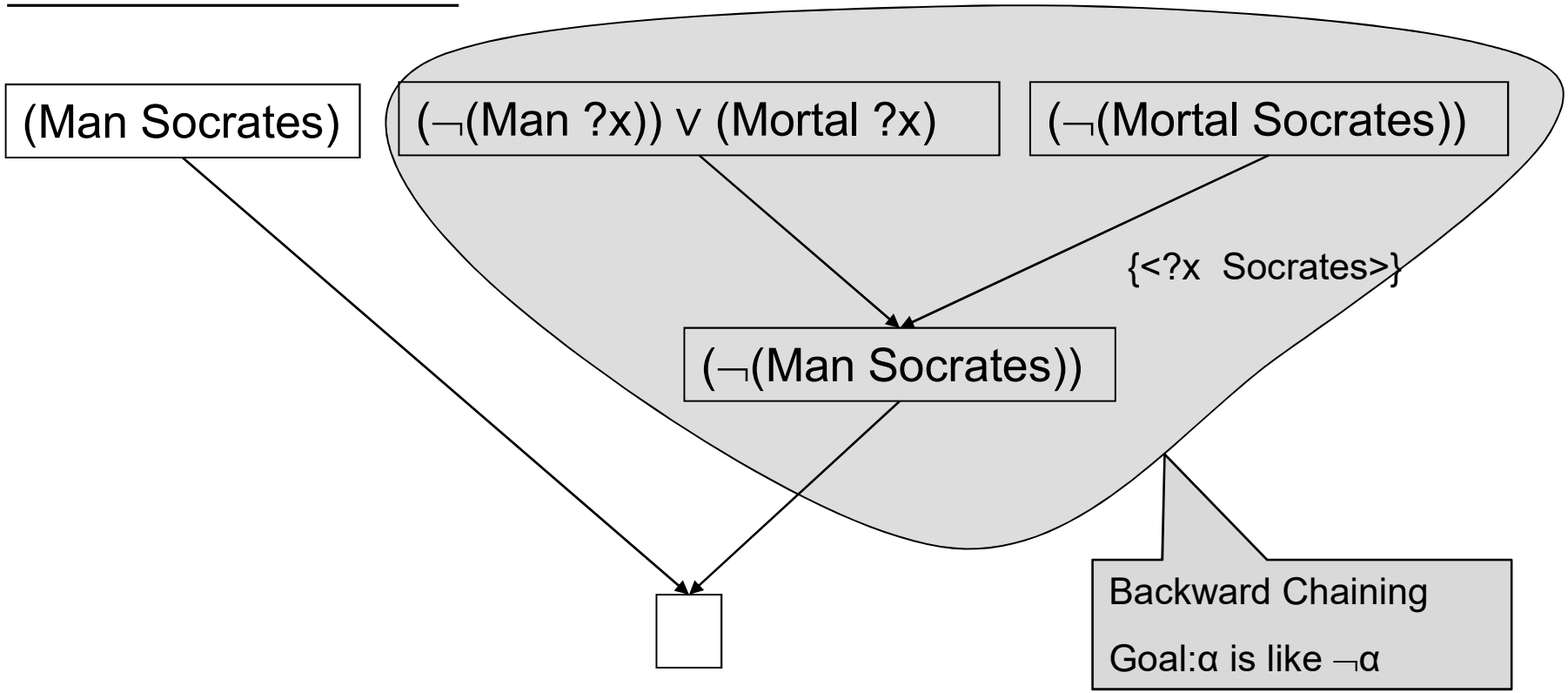
R_1	=	(Mortal Socrates)	C_1, C_2, {<?x Socrates>}
R_2	=	$\perp$	C_3, R_1

Remember that applying a substitution {?x = socrates} is a kind of instantiation.

The Proof as a Directed Acyclic Graph (DAG)



Another derivation



The Alice problem

The clauses are

1.(likes Alice Math)

2.(likes Alice stories)

3.(¬(likes ?x Math)) ∨ (likes ?x Algebra)

4.(¬(likes ?y Algebra)) ∨ (¬(likes ?y Physics)) ∨ (goesTo ?y College)

5.(¬(likes Alice stories)) ∨ (likes Alice Physics)

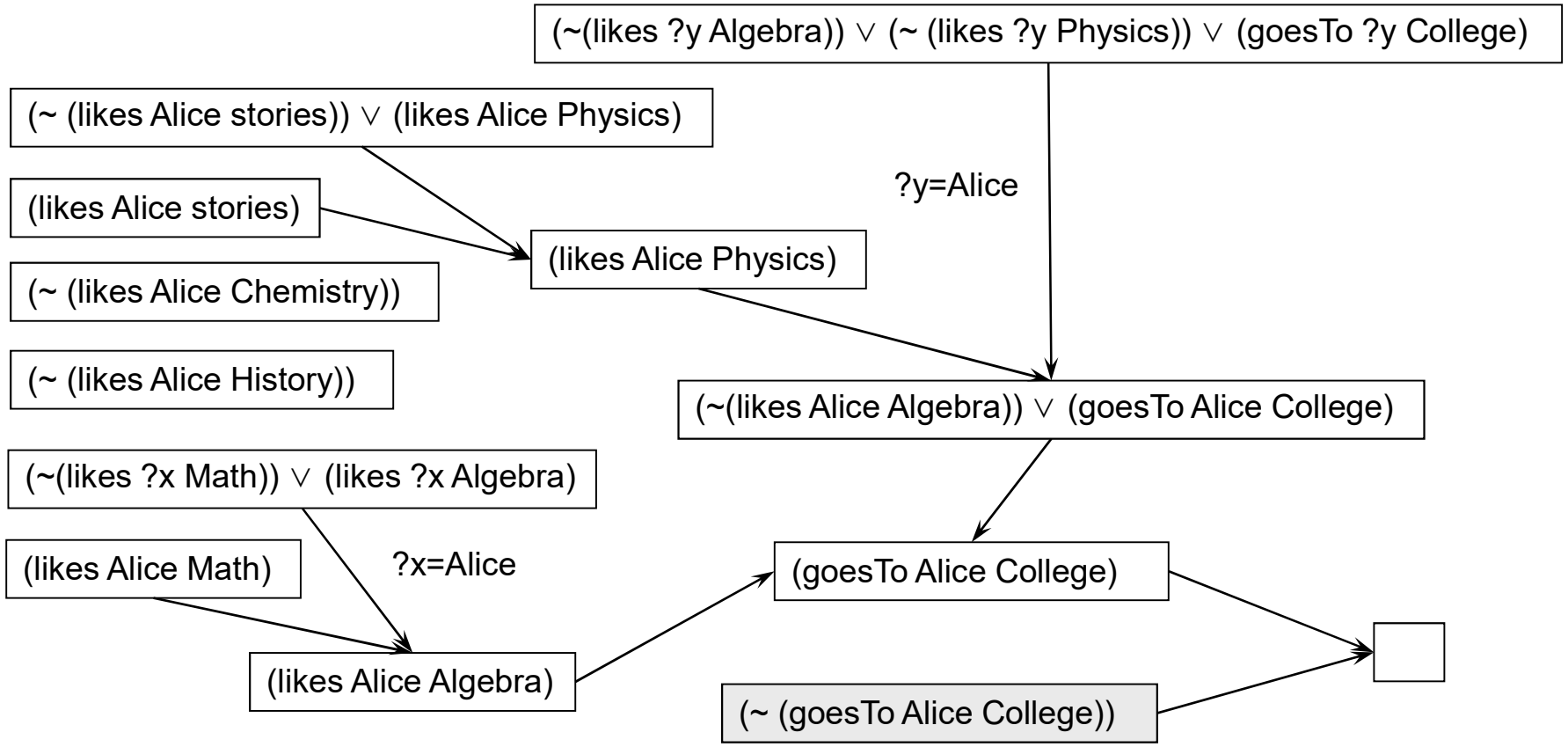
6.(¬(likes Alice Chemistry))

7.(¬(likes Alice History))

8.(¬(goesTo Alice College))

All but the last clause come from the premises. The last clause is the negated goal clause.

A resolution refutation



Incompleteness of Backward and Forward Chaining

Given the KB,

{(O A B), (O B C), (not (M A)), (M C)}

And the Goal,

(and (O ?x ?y) (not (M ?x)) (M ?y))

Neither Forward Chaining nor Backward Chaining
is able to generate a proof.

Both are Incomplete!

Next, we look at a proof method,
the Resolution Refutation System,
that is Sound and Complete for FOL

Recap

A reolution refutation

1. (O A B)) premise
2. (O B C)) premise
3. (¬(M A))) premise
4. (M C)) premise
5. (¬(O ?x ?y)) ∨ (M ?x) ∨ (¬(M ?y)) negated goal

A derivation of the null clause is,

6. (¬(O ?x C)) ∨ (M ?x) 4,5, ?y=C
7. (¬(O A ?y)) ∨ (¬(M ?y)) 3,5, ?x=A
8. (¬(M B)) 1,7, ?y=B
9. (M B) 2,6, ?x=B
10. ⊥ 8,9

Domain: The Blocks World

1.	(On A B))	premise
2.	(On B C))	premise
3.	(¬(Maroon A)))	premise
4.	(Maroon C))	premise
5.	(¬(On ?x ?y)) ∨ (Maroon ?x) ∨ (¬(Maroon ?y))	negated goal

A derivation of the null clause is,

6.	(¬(On ?x C)) ∨ (Maroon ?x)	4,5, ?y=C
7.	(¬(On A ?y)) ∨ (¬(Maroon ?y))	3,5, ?x=A
8.	(¬(Maroon B))	1,7, ?y=B
9.	(Maroon B)	2,6, ?x=B
10.	⊥	8,9

Domain: People

1.	(LookingAt Jack Anne)	premise
2.	(LookingAt Anne John)	premise
3.	(¬(Married Jack))	premise
4.	(Married John)	premise
5.	(¬(LookingAt ?x ?y)) ∨ (Married ?x) ∨ (¬(Married ?y))	negated goal

A derivation of the null clause is,

6.	(¬(LookingAt ?x John)) ∨ (Married ?x)	4,5, ?y=John
7.	(¬(LookingAt Jack ?y)) ∨ (¬(Married ?y))	3,5, ?x=Jack
8.	(¬(Married Anne))	1,7, ?y=Anne
9.	(Married Anne)	2,6, ?x=Anne
10.	⊥	8,9

Logic Programming

- Robert Kowalski put forward the idea that logic can be used for every day computing.
  - "*The driving force behind logic programming is the idea that a single formalism suffices for both logic and computation, and that logic subsumes computation*."
- His idea is that the programmer should only focus on the logical relation between Input and Output, and that a machine (computer program) should figure out the control of flow.
  - Program = Logic + Control
- The basis of logic programming languages like Prolog.

Addition

Consider an infinite domain consisting of one constant 0 and one function “s” of arity 1. The elements of the domain are {0, s(0), s(s(0)), s(s(s(0))),…. }. We can call this the domain of Natural Numbers.

The following statements define addition of two elements.

$\forall x$ Plus(0, x, x)

$\forall x,y,z$ (Plus(x, y, z) $\supset$ Plus(s(x), y, s(z)))

Plus(x, y, z) is interpreted as x+y=z. This small KB entails the entire set of addition relation

Addition

$\forall x$ Plus(0, x, x)

$\forall x,y,z$ (Plus(x, y, z) $\supset$ Plus(s(x), y, s(z)))

Converting the KB in to clause form we get,

Plus(0, ?x, ?x)

~Plus(?x, ?y, ?z) $\vee$ Plus(s(?x), ?y, s(?z))

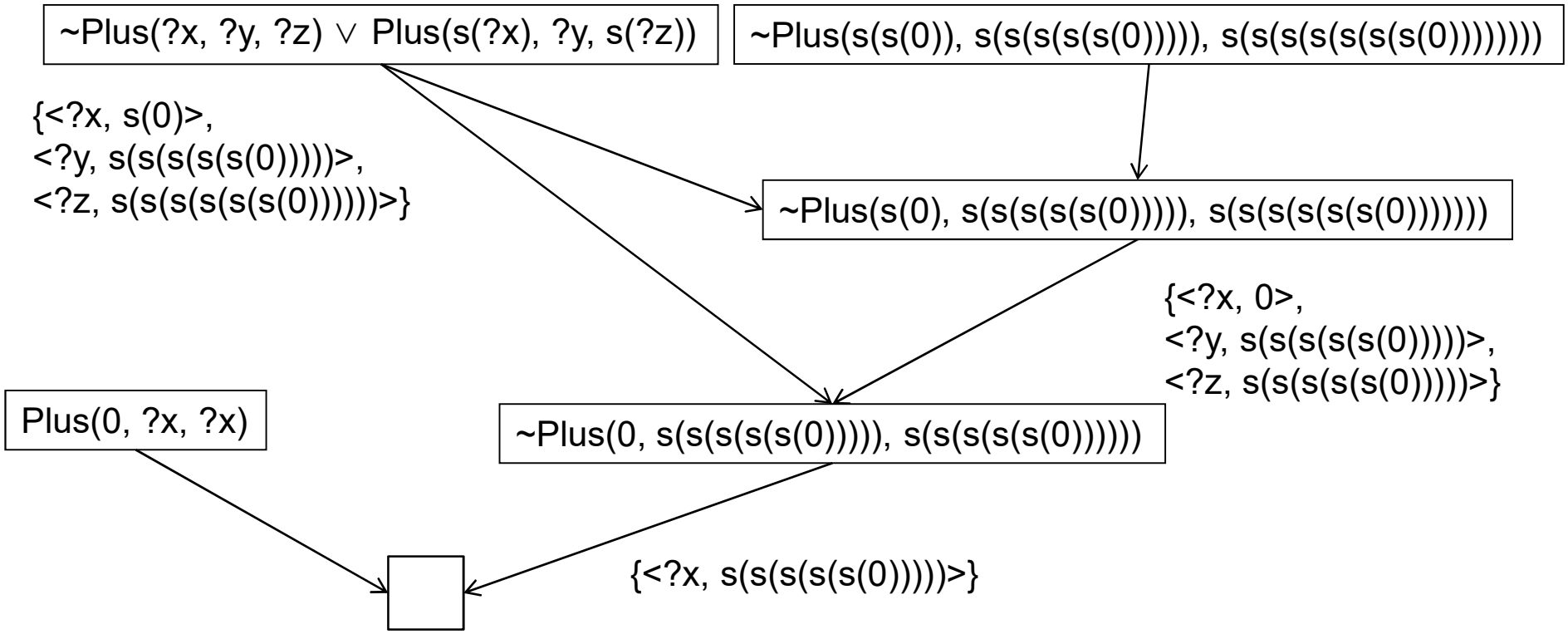
Let us consider two queries

Plus(2,5,7) written as Plus(s(s(0)), s(s(s(s(s(0))))), s(s(s(s(s(s(s(0))))))))

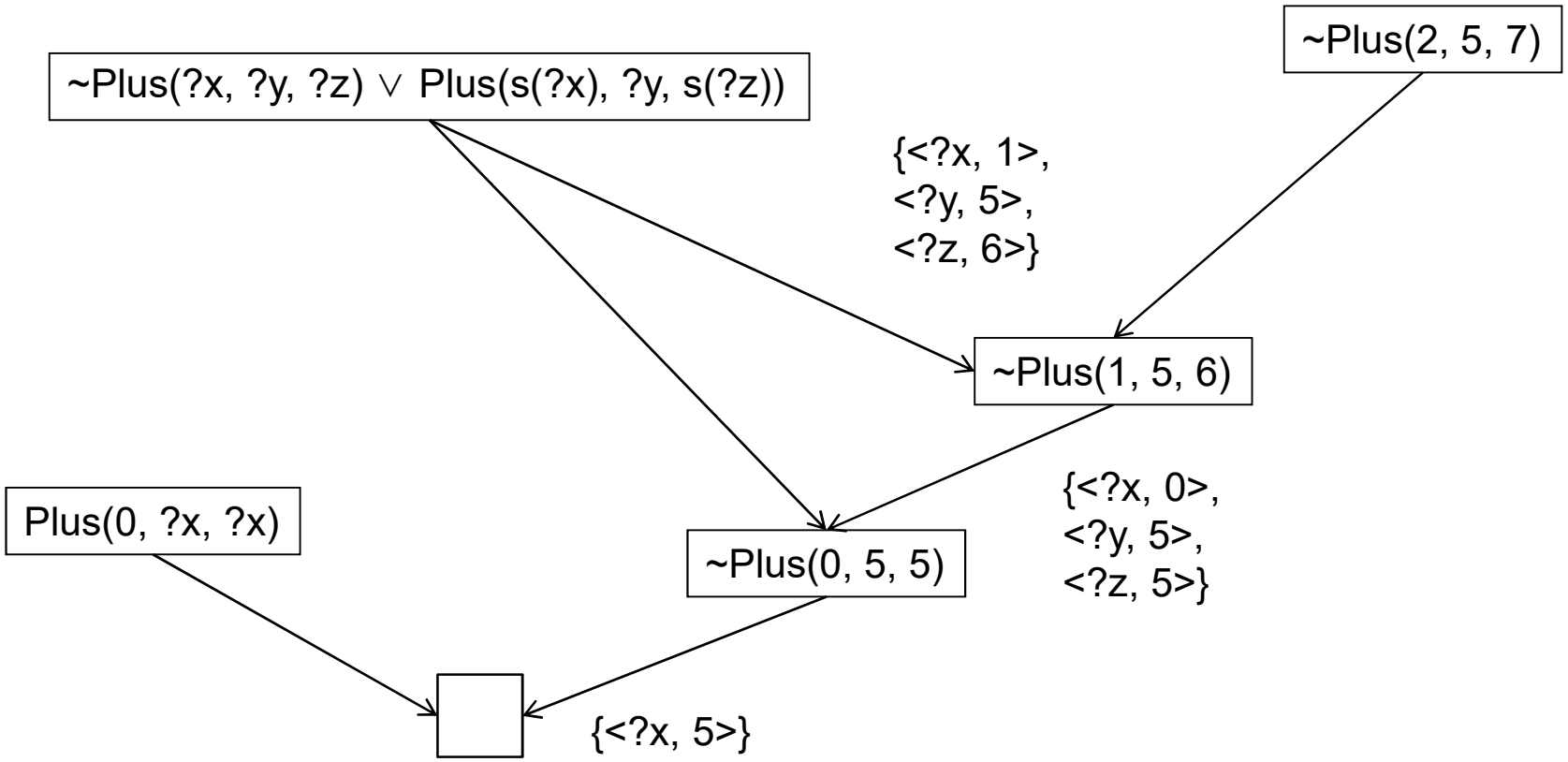
and $\exists x$Plus(2,5,x) written as $\exists x$Plus(s(s(0)), s(s(s(s(s(0))))), x)

We negate the queries and use the resolution method

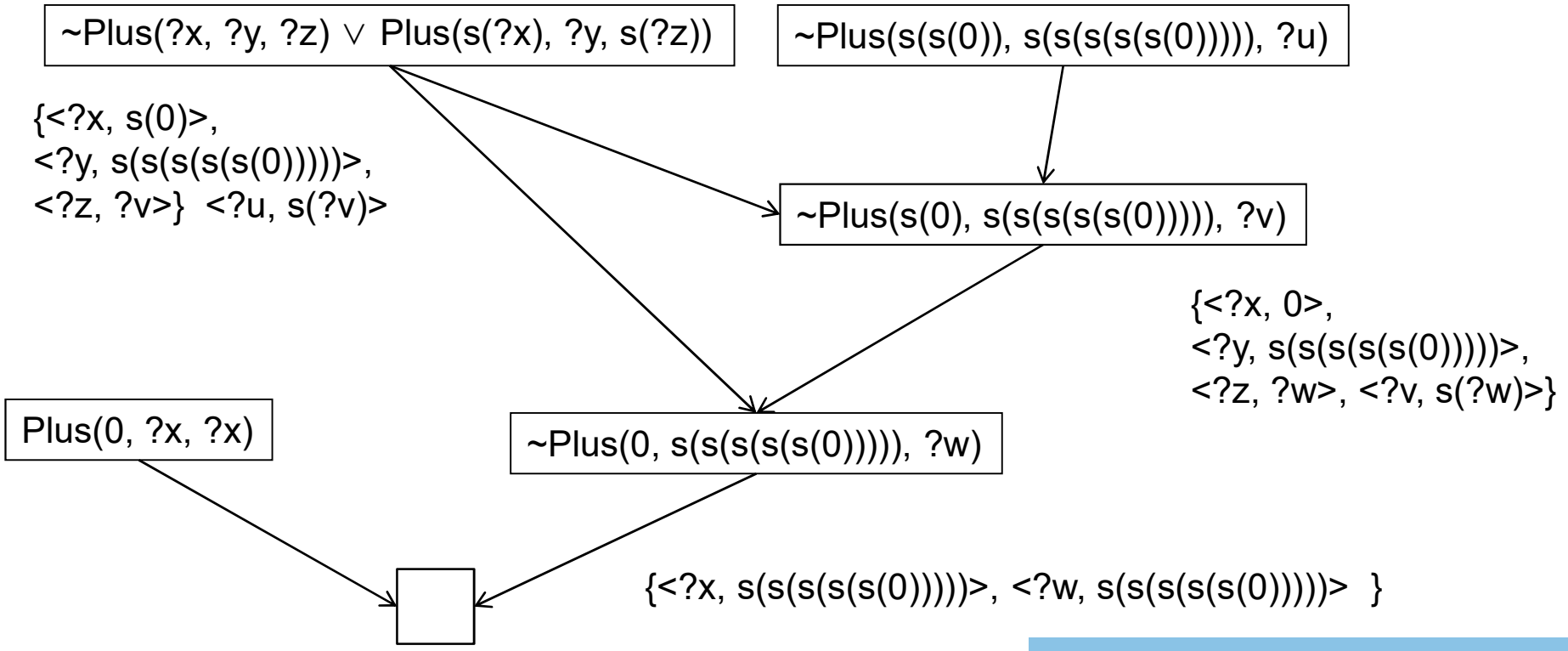
Query: Is 2+5=7?


~Plus(?x, ?y, ?z) ∨ Plus(s(?x), ?y, s(?z))
~Plus(s(s(0)), s(s(s(s(s(0))))), s(s(s(s(s(s(s(0)))))))
{<?x, s(0)>,
<?y, s(s(s(s(s(0)))))>,
<?z, s(s(s(s(s(s(0))))))>}
~Plus(s(0), s(s(s(s(s(0))))), s(s(s(s(s(s(0)))))))
{<?x, 0>,
<?y, s(s(s(s(s(0)))))>,
<?z, s(s(s(s(s(0)))))>}
Plus(0, ?x, ?x)
~Plus(0, s(s(s(s(s(0))))), s(s(s(s(s(0))))))
{<?x, s(s(s(s(s(0)))))>}

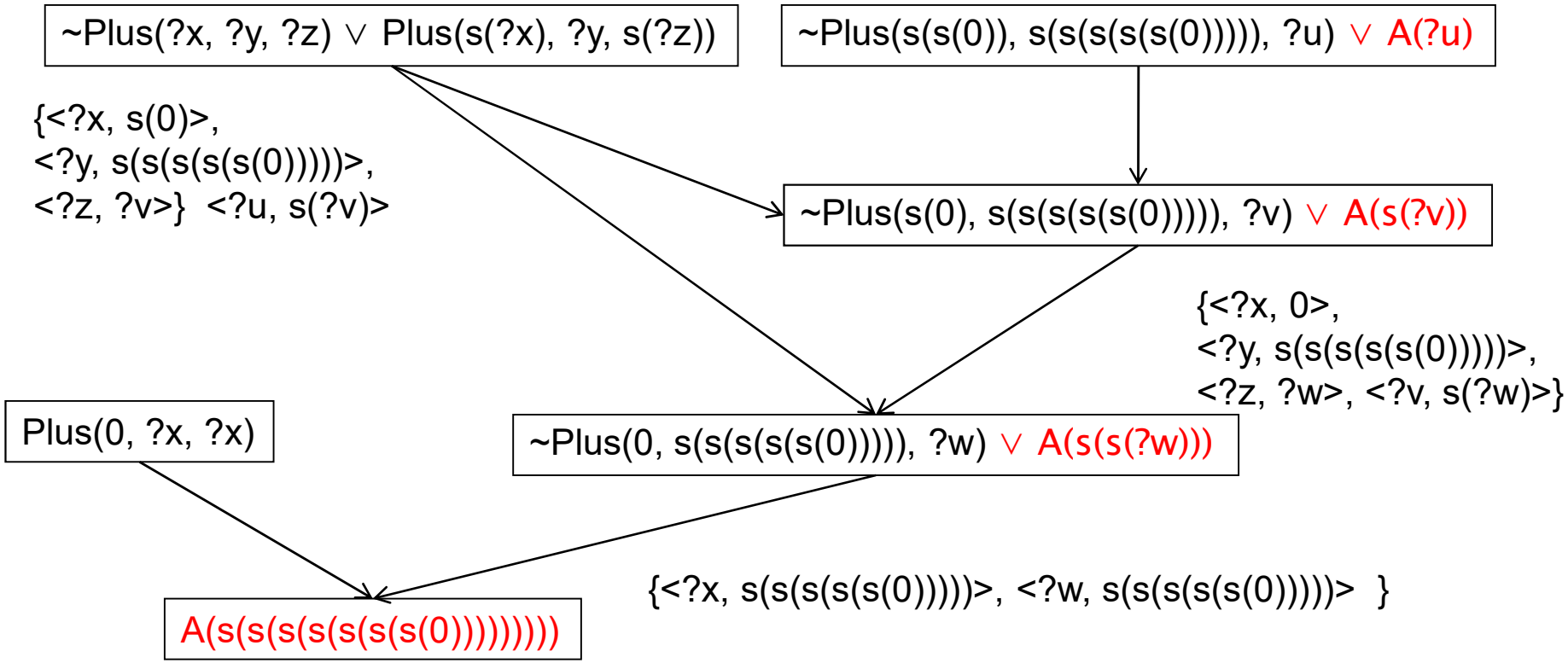
Using numerals



Existential query: Is there an X such that 2+5=X?



Using the Answer Predicate A(?u)



Horn Clauses

A clause with at most one positive literal is a Horn clause (named after the mathematician Alfred Horn).

$(\neg P \vee \neg Q \vee \neg R \vee \neg S \vee T)$ or

$(\neg(P\ ?x) \vee \neg(Q\ ?x) \vee \neg(R\ ?x) \vee \neg(S\ ?x)) \vee (T\ ?x)$

Equivalently they can be written as.

$(P \wedge Q \wedge R \wedge S) \supset T$ or

$((P\ ?x) \wedge (Q\ ?x) \wedge (R\ ?x) \wedge (S\ ?x)) \supset (T\ ?x)$

Positive Definite Horn Clauses

A positive Horn clause has exactly one positive literal

Facts have no negative literals

(T ?x) or (T a)

Rules have some number of negative literals

$((P\ ?x) \wedge (Q\ ?x) \wedge (R\ ?x) \wedge (S\ ?x)) \supset (T\ ?x)$

$(\neg(P\ ?x) \vee \neg(Q\ ?x) \vee \neg(R\ ?x) \vee \neg(S\ ?x)) \vee (T\ ?x)$

Prolog program is a set of Positive Definite Horn Clauses

In Prolog upper case arguments are variables and lower case arguments are constants. Further the syntax is different.

(T a)	Fact
((P ?x) ∧ (Q ?x) ∧ (R ?x) ∧ (S ?x)) ⊃ (T ?x)	Rule
T(a).	Fact
T(X) :- P(X), Q(X), R(X), S(X).	Rule

A Prolog KB (program)

```
outingPlan(X,Y,Z) :- eveningPlan(X), moviePlan(Y), dinnerPlan(Z).
eveningPlan(X) :- outing(X), likes(friend, X).
moviePlan(X) :- movie(X), likes(friend,X).
dinnerPlan(X) :- restaurant(X), likes(friend,X).
outing(mall).
outing(beach).
movie(theMatrix).
movie(artificialIntelligence).
movie(bhuvanShome).
movie(sevenSamurai).
restaurant(pizzaHut).
restaurant(saravanaBhavan).
likes(friend, beach).
likes(friend, theMatrix).
likes(friend, bhuvanShome).
likes(friend, sarvanaBhavan).
```

```
(if (and (restaurant ?x) (likes friend ?x)) (dinnerPlan ?x))
```

Recap

Negative Horn Clauses

- A Horn clause with no positive literal is a negative clause
- In Prolog such clause come only from a negated goal or query
- The null clause is a negative clause (no positive literal).
- The resolution rule has the following resolvents

  Positive clause and positive clause → positive clause

  $\neg(S\ ?x) \vee (T\ ?x)$ and $\neg(Q\ ?x) \vee (S\ ?x) \rightarrow (T\ ?x) \vee \neg(Q\ ?x)$

  Positive clause and negative clause → negative clause

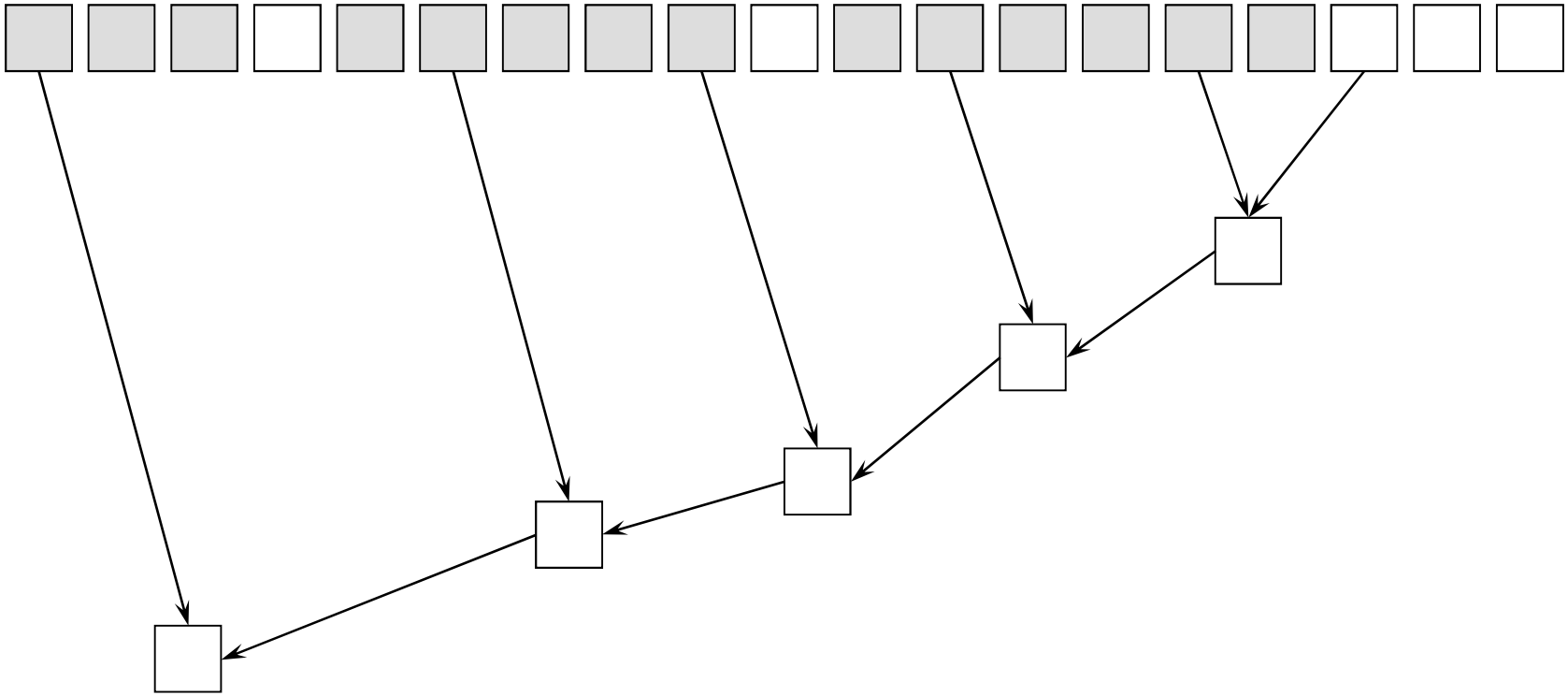
  $\neg(S\ ?x) \vee \neg\ (T\ ?x)$ and $\neg(Q\ ?x) \vee (S\ ?x) \rightarrow \neg\ (T\ ?x) \vee \neg(Q\ ?x)$

  Two negative cannot be resolved

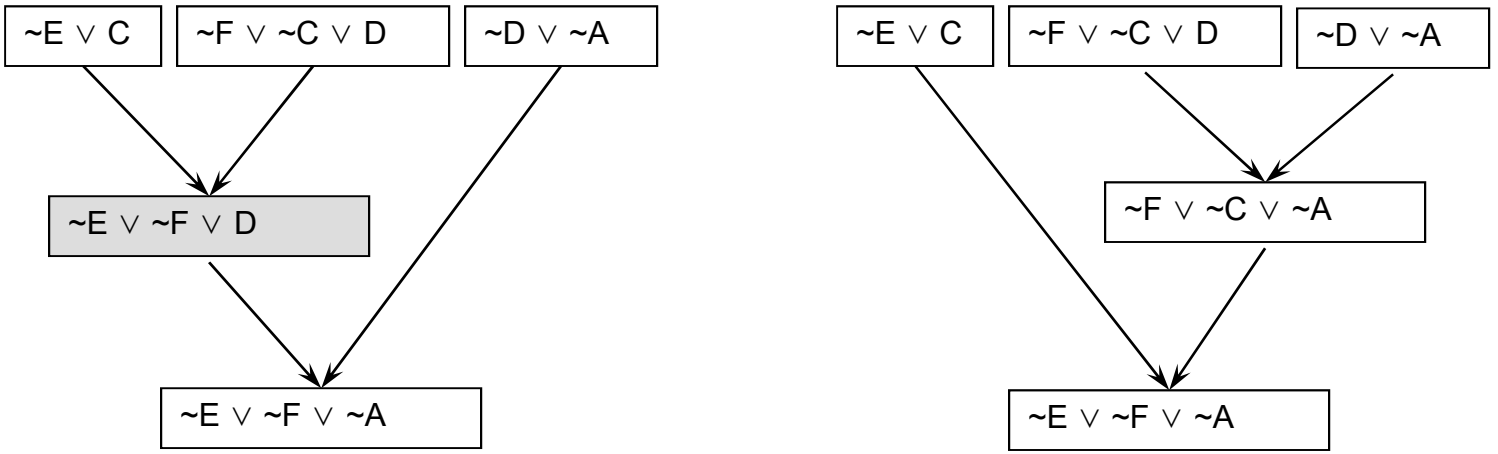
An SLD derivation

- A Prolog program is a set of positive definite clauses
- The negated goal is the only negative clause
- An SLD (selected literal, linear structure, definite clauses) derivation has the following structure
  - the first resolvent has one parent from the goal and one from the program
  - each resolvent in the derivation is negative (this is equivalent to doing backward chaining to generate a sub-goal)
  - the latest resolvent becomes one of the parents, and the other parent is a positive clause from the program

An SLD derivation with Horn Clauses

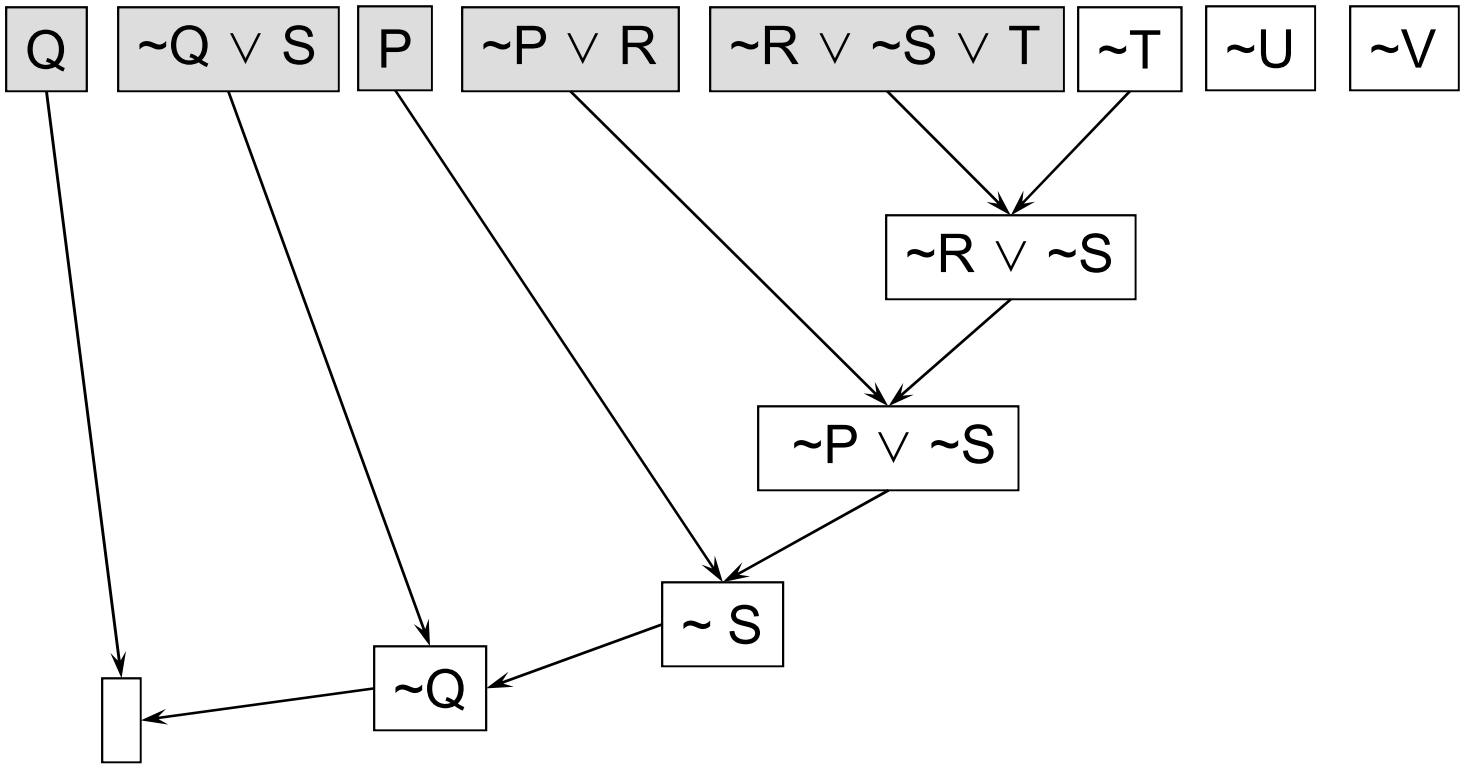


Eliminating positive resolvents



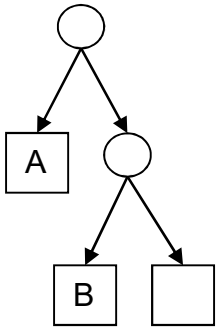
The figure on the left has a positive *resolvent*, shown in the shaded box. The equivalent derivation on the right has only negative resolvents. One can transform derivation with positive resolvents to one with only negative resolvents by picking the lowest positive resolvent and applying a transformation like the one above.

AN SLD derivation for the Alice problem

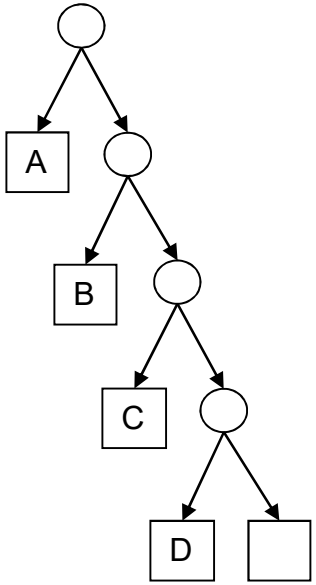


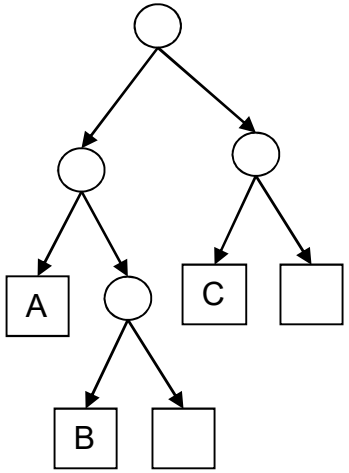
Lists are binary trees

[[A B] C] = (cons (cons A (cons B nil)) (cons (C nil)))



[A B] = (cons (A cons (B nil)))

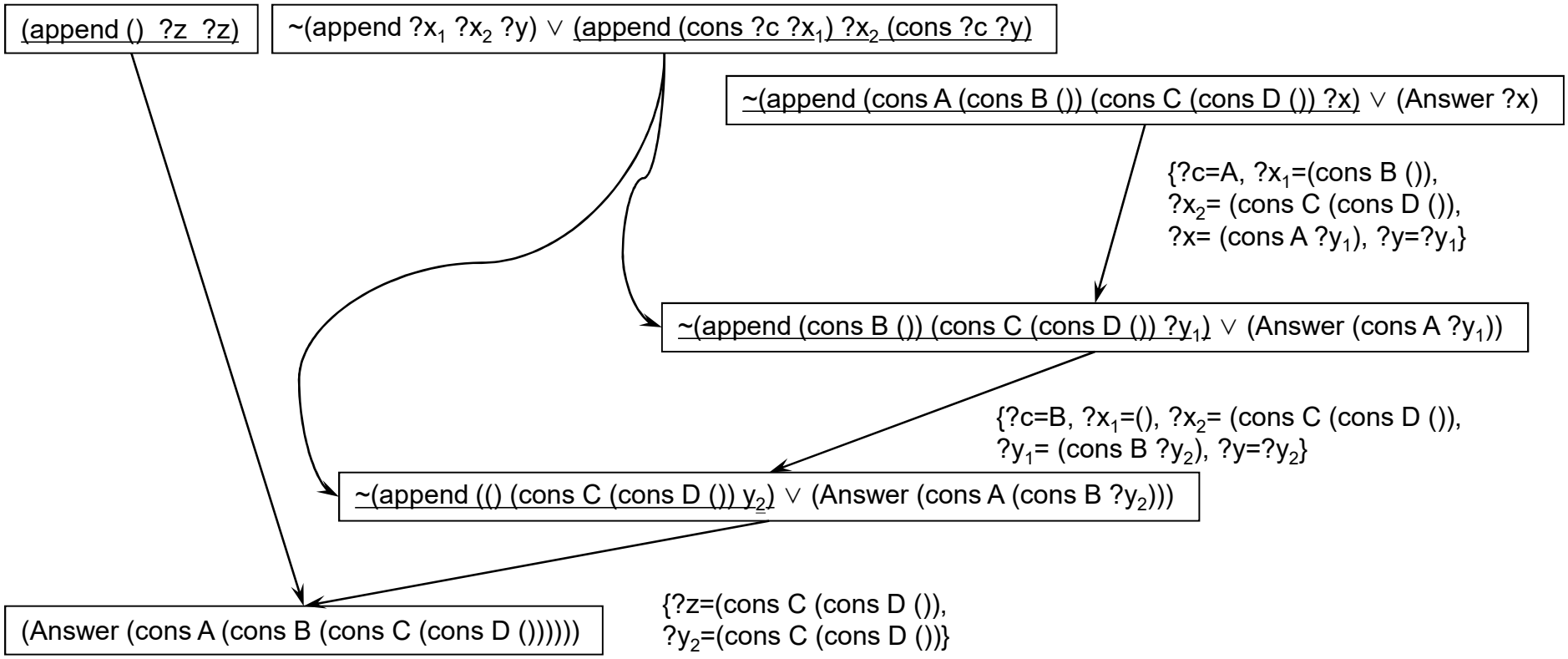




[A B C D] = (cons A(cons B (cons C (cons D nil))))

The tree structures for the lists [A B], [A B C D], and [[A B] C] respectively. The circular node is also known as the dotted pair or the cons pair. The left child points to the head of the list and the right child to a list that is the tail of the given list.

Computing Append ([A B] [C D] ?x)



End of Module 3