# Overview of Automated Reasoning

Peter Baumgartner

Peter.Baumgartner@nicta.com.au

http://users.rsise.anu.edu.au/~baumgart/

## What is Automated Reasoning?

Theme

Building push-button technology (software) for mathematical-logical reasoning on computer

Relevant fields

- Mathematical logic and philosophy: formal logics and calculi
- Theoretical computer science: computability theory, complexity theory
- Applied and practical computer science: artifical intelligence, data structures and algorithms

Applications: Software verification, hardware verification, analysing dynamic properties of reactive systems, databases, mathematical theorem proving, planning, diagnosis, knowledge representation (description logics), logic programming, constraint solving

**Automated Reasoning systems parametrized in**
**logic and reasoning service**

## Logics and Reasoning Service: Constraint Solving

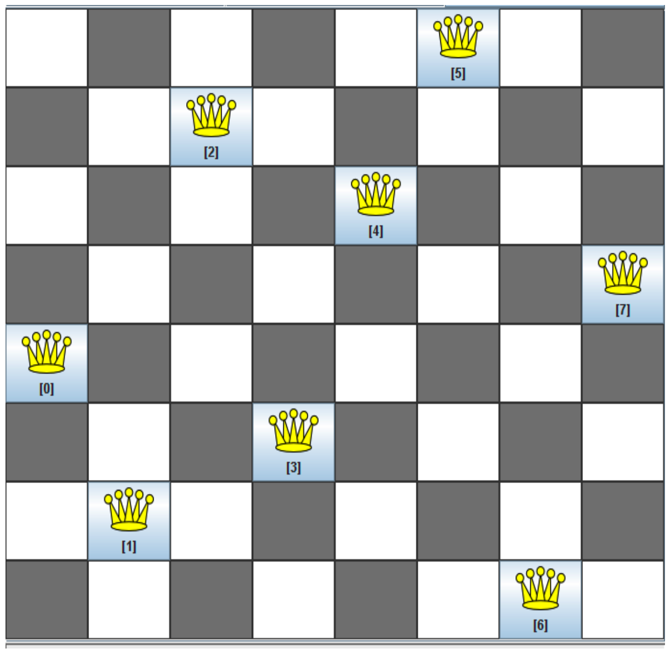
The n-queens problem:

Given: An $n \times n$ chessboard

Question: Is it possible to place $n$ queens so that no queen attacks any other?

A solution for $n = 8$

$p[1] = 6$
$p[2] = 3$
$p[3] = 5$
$p[4] = 8$
$p[5] = 1$
$p[6] = 4$
$p[7] = 2$
$p[8] = 7$



Use a **constraint solver** to find a solution.

## Logics and Reasoning Service: Constraint Solving

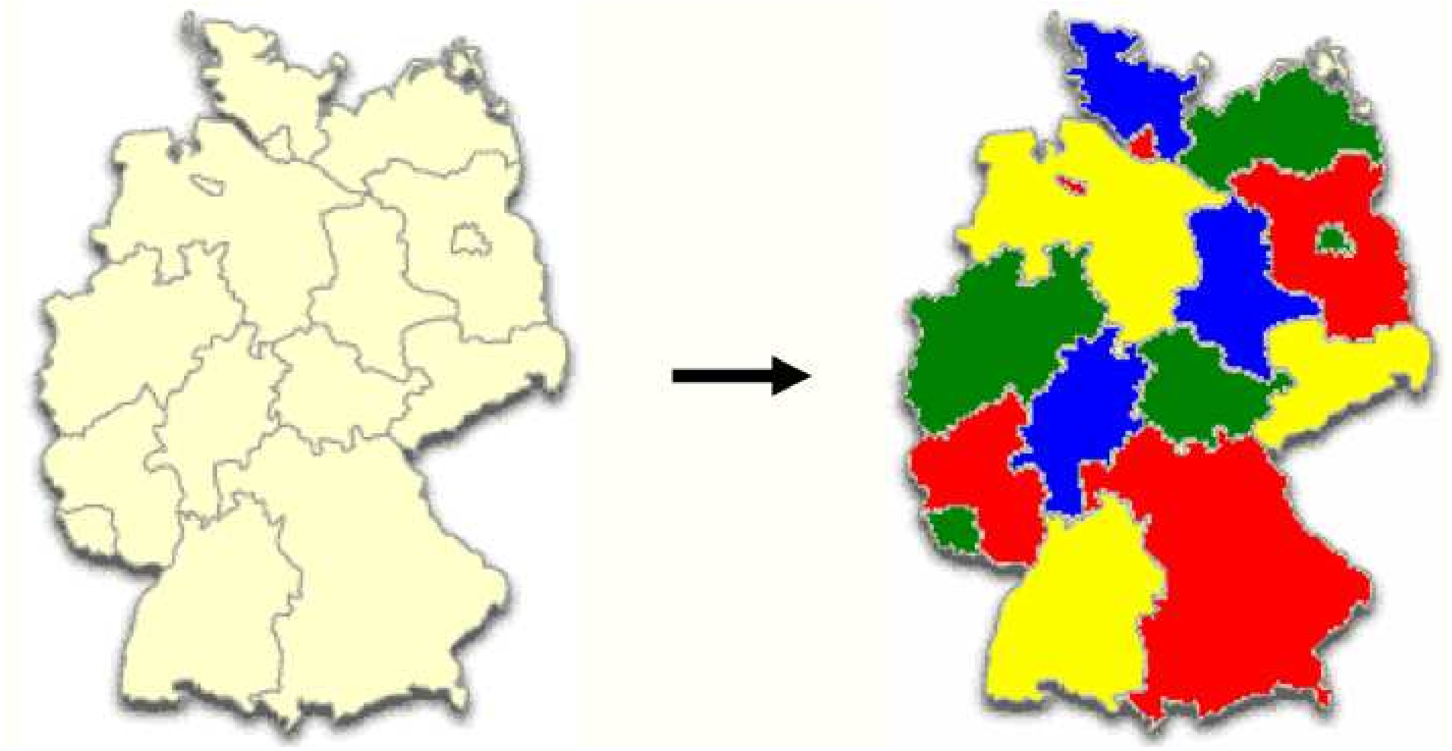
A **Zinc** model, ready to be “run” by a constraint solver:

```
int: n = 8;
array [1..n] of var 1..n: p;
constraint
        forall (i in 1..n, j in i + 1..n) (
                p[i]     != p[j]
        /\      p[i] + i != p[j] + j
        /\      p[i] - i != p[j] - j
        );
solve satisfy;
output ["Solution: ", show(p), "\n"];
```

Logic: finite integers, arithmetic, equality (arrays are syntactic sugar)

Reasoning Service: satisfiability: search assignments for **fixed** $n$ variables s.th. constraint is satisfied

## Logics and Reasoning Service: Three Coloring Problem

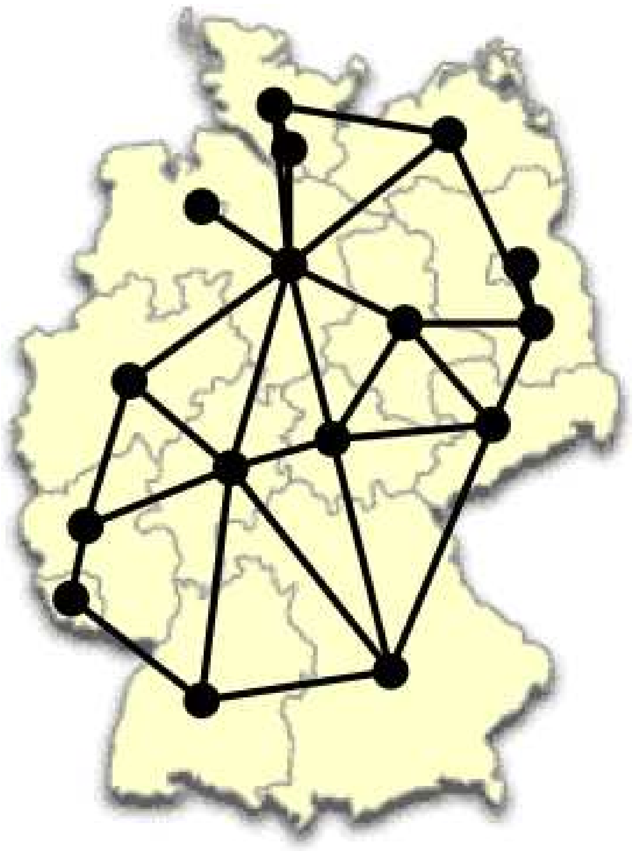


**Problem:** Given a map. Can it be colored using only three colors, where neighbouring countries are colored differently?
(Three-coloring is NP-complete)

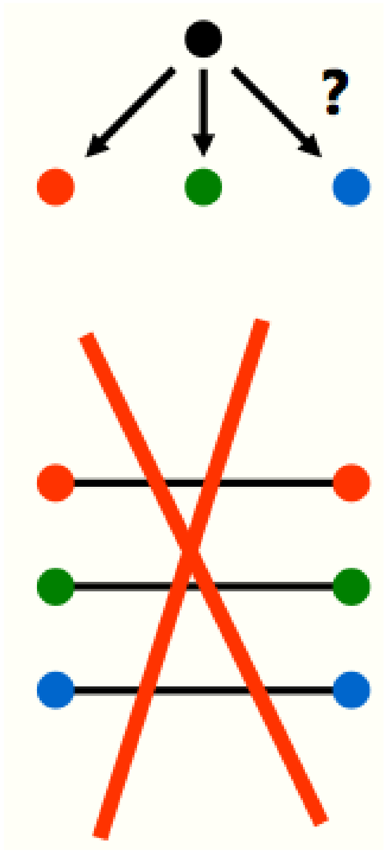
## Logics and Reasoning Service: Three Coloring Problem

### Graph Theory Abstraction

**Problem Instance**



**Problem Specification**



## Logics and Reasoning Service: Three Coloring Problem

**Formalization: let $3C$ be the conjunction of the following three formulas:**

Every node has at least one color

$$\forall N\ (\mathrm{red}(N) \lor \mathrm{green}(N) \lor \mathrm{blue}(N))$$

## Logics and Reasoning Service: Three Coloring Problem

**Formalization: let $3C$ be the conjunction of the following three formulas:**

Every node has at least one color

$$\forall N \; (\mathrm{red}(N) \vee \mathrm{green}(N) \vee \mathrm{blue}(N))$$

Every node has at most one color

$$\begin{aligned}\forall N \; ((&\mathrm{red}(N) \to \neg\mathrm{green}(N)) \wedge \\ (&\mathrm{red}(N) \to \neg\mathrm{blue}(N)) \wedge \\ (&\mathrm{blue}(N) \to \neg\mathrm{green}(N)))\end{aligned}$$

## Logics and Reasoning Service: Three Coloring Problem

**Formalization: let $3C$ be the conjunction of the following three formulas:**

Every node has at least one color

$$\forall N\ (\mathrm{red}(N) \lor \mathrm{green}(N) \lor \mathrm{blue}(N))$$

Every node has at most one color

$$\begin{aligned}\forall N\ ((&\mathrm{red}(N) \to \neg \mathrm{green}(N)) \land \\ (&\mathrm{red}(N) \to \neg \mathrm{blue}(N)) \land \\ (&\mathrm{blue}(N) \to \neg \mathrm{green}(N)))\end{aligned}$$

Adjacent nodes have different color

$$\begin{aligned}\forall M, N\ (\mathrm{edge}(M, N) \to (&\neg(\mathrm{red}(M) \land \mathrm{red}(N)) \land \\ &\neg(\mathrm{green}(M) \land \mathrm{green}(N)) \land \\ &\neg(\mathrm{blue}(M) \land \mathrm{blue}(N))))\end{aligned}$$

## Logics and Reasoning Service: Three Coloring Problem

### Problem

Given: A concrete graph $G = \{\text{edge}(n_0, n_1),\ \text{edge}(n_0, n_2), \ldots\}$

Question: Does $G$ admit a three-coloring?

### Solution 1

Logic: First-order logic

Reasoning service: Satisfiability

Is $G \cup 3C$ satsifiable?

Any model provides a solution – therefore can use any first-order theorem prover **that is capable of providing models**

## Logics and Reasoning Service: Three Coloring Problem

### Problem

Given: A concrete graph $G = \{\text{edge}(n_0, n_1),\ \text{edge}(n_0, n_2), \ldots\}$

Question: Does $G$ admit a three-coloring?

### Solution 2

Logic: Propositional logic

Reasoning service: Satisfiability

Is $G \cup \mathit{ground}(3C)$ satsifiable?

$\mathit{ground}(3C)$ is obtained from $3C$ by collecting all formulas that are obtained from $3C$ by replacing all variabes in all possible ways by the vertices in $G$

## Logics and Reasoning Service: Three Coloring Problem

### Problem

Given: A concrete graph $G = \{\text{edge}(n_0, n_1),\ \text{edge}(n_0, n_2), \ldots\}$

Question: Does $G$ admit a three-coloring?

### Solution 3

Logic: Existential Second-Order logic (ESO) over finite structures

Reasoning service: Evaluate ESO-formula on a given finite structure

Does "$G \models \exists \text{red}\ \exists \text{green}\ \exists \text{blue}\ 3C$" hold?

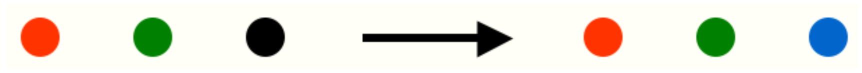
$G$ fixes the finite structure (finite graph). Essentially, this is what constraint solvers do

## Logics and Reasoning Service: Three Coloring Problem

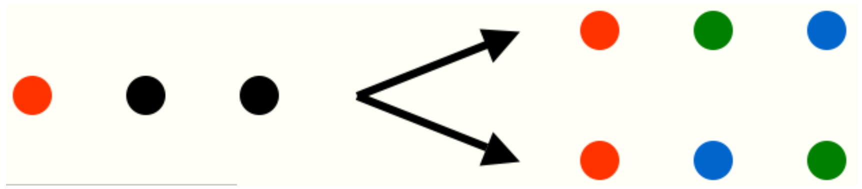
Sometimes it is not only useful to **solve** problems but also to **analyse** problems for their properties.

### Functional dependency

- Blue coloring depends functionally on the red and green coloring



- But blue coloring does not functionally depend on the red coloring



Why is it useful to know about functional dependencies?

Values for functional dependent variables do not need to be **searched** for.

Useful to speed up constraint solving.

## Logics and Reasoning Service: Three Coloring Problem

**Problem: show that the three coloring problem has functional dependencies**

Logic: First-order logic

Reasoning service: Validity

> **for every** graph $G$ and any two colorings $RGB_1$ and $RGB_2$ of $G$:
> if $RGB_1$ and $RGB_2$ are equal on the *red* and the *green* colorings
> then $RGB_1$ and $RGB_2$ are equal on the *blue* coloring.

Actual formalization: see file `threecolor-relational-fundep.p` and demo

Validity (that a property holds for all structures) cannot be established by constraint solving

**Need a wide spectrum of logics and reasoning service**

## Logics and Reasoning Service - Spectrum

**Logics**

Base logic: propositional/first-order/higher-order

Syntactic fragments
(Description Logics, Datalog, ...)

Classical/non-monotonic

Modalities (temporal, deontic, ...)

Over structures (finite trees, graphs,...)

Modulo Theories (equality, arithmetic, ...)

**Services**

Model checking
(evaluation)

Satisfiability
(minimal models)

Validity

Induction

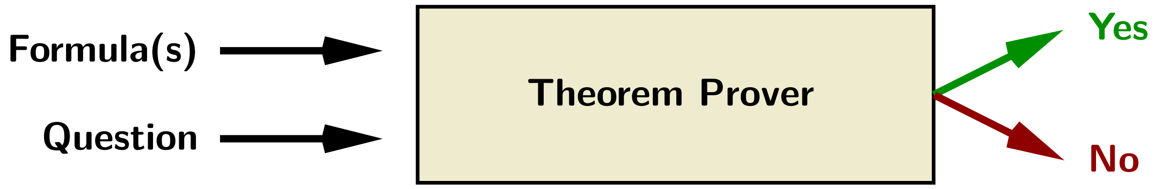
Abduction

Almost any subset of the left column (potentially) makes sense

**The challenge is to build “decent” calculi/theorem provers: theoretically analysed, avoiding redundancies, practically useful, meaningful answers (proofs, models), ...**

## Contents

## Propositional SAT Solving



Formula: Propositional logic formula $\phi$

Question: Is $\phi$ satisfiable?

(Minimal model? Maximal consistent subsets? )

Theorem Prover: Based on BDD, **DPLL**, or stochastic local search

**Issue:** the formula $\phi$ can be **BIG**

DPLL: Davis-Putnam-Logemann-Loveland method, 1960s

## DPLL as a Semantic Tree Method

(1) $A \lor B$ (2) $C \lor \neg A$ (3) $D \lor \neg C \lor \neg A$ (4) $\neg D \lor \neg B$

⟨empty tree⟩

$\{\} \not\models A \lor B$

$\{\} \models C \lor \neg A$

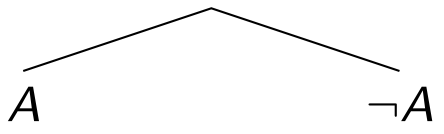
$\{\} \models D \lor \neg C \lor \neg A$

$\{\} \models \neg D \lor \neg B$

- A Branch stands for an interpretation
- **Purpose of splitting:** satisfy a clause that is currently falsified
- Close branch if some clause is plainly falsified by it ($\star$)

## DPLL as a Semantic Tree Method

(1) $A \lor B$ (2) $C \lor \neg A$ (3) $D \lor \neg C \lor \neg A$ (4) $\neg D \lor \neg B$

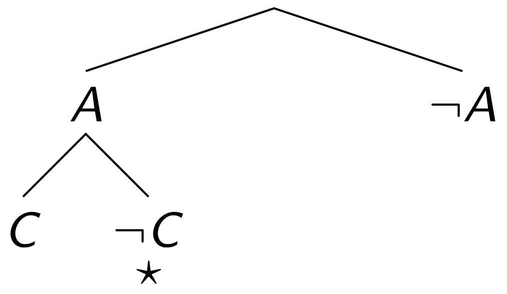


$$\boxed{\begin{array}{l} \{A\} \models A \lor B \\ \{A\} \not\models C \lor \neg A \\ \{A\} \models D \lor \neg C \lor \neg A \\ \{A\} \models \neg D \lor \neg B \end{array}}$$

- A Branch stands for an interpretation
- **Purpose of splitting:** satisfy a clause that is currently falsified
- Close branch if some clause is plainly falsified by it ($\star$)

## DPLL as a Semantic Tree Method

(1) $A \vee B$ (2) $C \vee \neg A$ (3) $D \vee \neg C \vee \neg A$ (4) $\neg D \vee \neg B$



$\{A, C\} \models A \vee B$

$\{A, C\} \models C \vee \neg A$

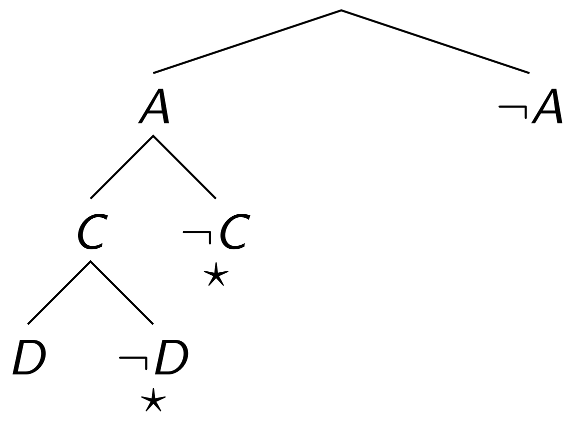
$\{A, C\} \not\models D \vee \neg C \vee \neg A$

$\{A, C\} \models \neg D \vee \neg B$

- A Branch stands for an interpretation
- **Purpose of splitting:** satisfy a clause that is currently falsified
- Close branch if some clause is plainly falsified by it ($\star$)

## DPLL as a Semantic Tree Method

(1) $A \lor B$ (2) $C \lor \neg A$ (3) $D \lor \neg C \lor \neg A$ (4) $\neg D \lor \neg B$



$\{A, C, D\} \models A \lor B$

$\{A, C, D\} \models C \lor \neg A$

$\{A, C, D\} \models D \lor \neg C \lor \neg A$

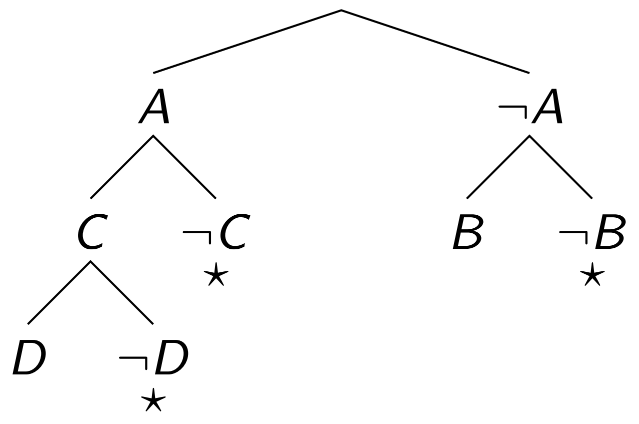
$\{A, C, D\} \models \neg D \lor \neg B$

Model $\{A, C, D\}$ found.

- A Branch stands for an interpretation
- **Purpose of splitting:** satisfy a clause that is currently falsified
- Close branch if some clause is plainly falsified by it ($\star$)

## DPLL as a Semantic Tree Method

(1) $A \vee B$ (2) $C \vee \neg A$ (3) $D \vee \neg C \vee \neg A$ (4) $\neg D \vee \neg B$



$\{B\} \models A \vee B$

$\{B\} \models C \vee \neg A$

$\{B\} \models D \vee \neg C \vee \neg A$

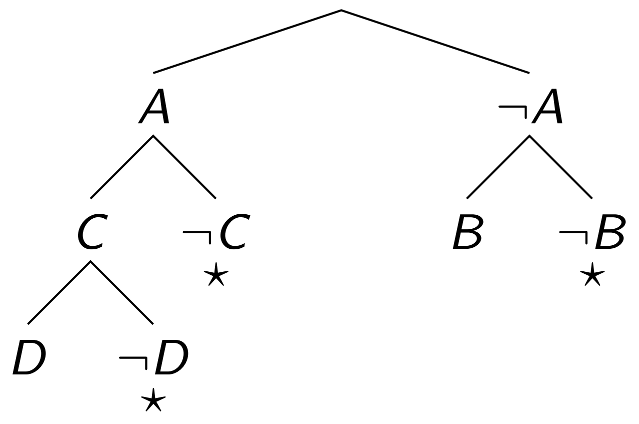
$\{B\} \models \neg D \vee \neg B$

Model $\{B\}$ found.

- A Branch stands for an interpretation
- **Purpose of splitting:** satisfy a clause that is currently falsified
- Close branch if some clause is plainly falsified by it ($\star$)

## DPLL as a Semantic Tree Method

(1) $A \vee B$ (2) $C \vee \neg A$ (3) $D \vee \neg C \vee \neg A$ (4) $\neg D \vee \neg B$



$\{B\} \models A \vee B$

$\{B\} \models C \vee \neg A$

$\{B\} \models D \vee \neg C \vee \neg A$

$\{B\} \models \neg D \vee \neg B$

Model $\{B\}$ found.

- A Branch stands for an interpretation
- **Purpose of splitting:** satisfy a clause that is currently falsified
- Close branch if some clause is plainly falsified by it ($\star$)

DPLL is the basis of most efficient SAT solvers today

## DPLL Pseudocode

**literal** $L$: a variable $A$ or its negation $\neg A$

**clause**: a set of literals, e.g., $\{A, \neg B, C\}$, connected by “or”

```
function DPLL($\phi$)   %% $\phi$ is a set of clauses, connected by "and"
  while $\phi$ contains a unit clause $\{L\}$
    $\phi$ := simplify($\phi$, $L$);
  if $\phi = \{\}$ then return true;
  if $\{\} \in \phi$ then return false;
  L := choose-literal($\phi$);
  if DPLL(simplify($\phi$, $L$)) then return true;
  else return DPLL(simplify($\phi$, $\neg L$));

function simplify($\phi$, $L$)  %% also called unit propagation
  remove all clauses from $\phi$ that contain $L$;
  delete $\neg L$ from all remaining clauses;
  return the resulting clause set;
```

## Making DPLL Fast

### Key ingredients

- Lemma learning: add new clauses to the clause set as branches get closed. Goal: reuse information that is obtained in one branch for subsequent derivation steps.
  (See subsequent slides.)
- Replace chronlogical backtracking by “dependency-directed backtracking”, aka “backjumping”: on backtracking, skip splits that are not necessary to close a branch.
- Randomized restarts: every now and then start over, with learned clauses.
- Variable selection heuristics: what literal to split on. E.g., use literals that occur often.
- Make unit-propagation fast: 2-watched literal technique.

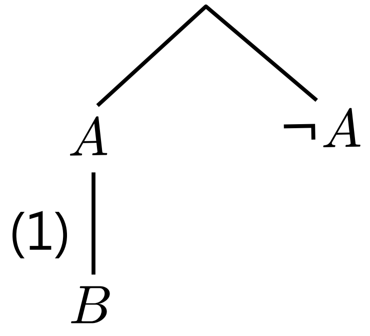
## Lemma Learning in DPLL

**"Avoid making the same mistake twice"**

$$\ldots$$

| | |
|---|---|
| $B \lor \neg A$ | (1) |
| $D \lor \neg C$ | (2) |
| $\neg D \lor \neg B \lor \neg C$ | (3) |

**w/o Lemma**

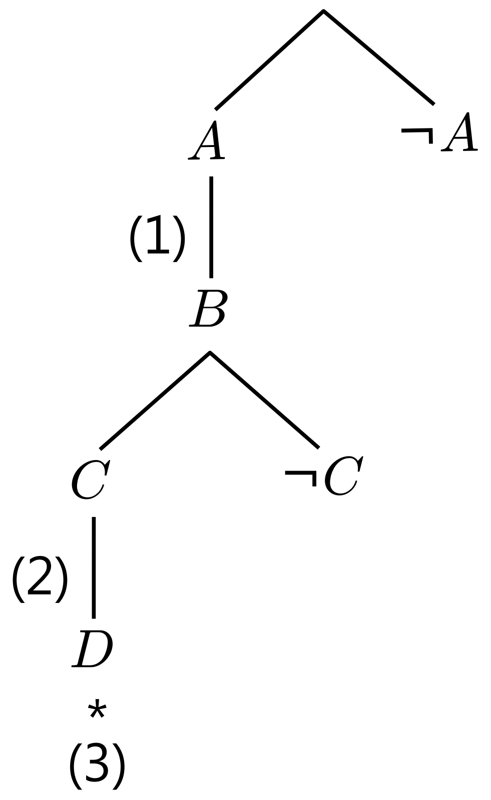


## Lemma Learning in DPLL

**"Avoid making the same mistake twice"**

$$\begin{array}{lr} & \dots \\ B \vee \neg A & (1) \\ D \vee \neg C & (2) \\ \neg D \vee \neg B \vee \neg C & (3) \end{array}$$

**w/o Lemma**

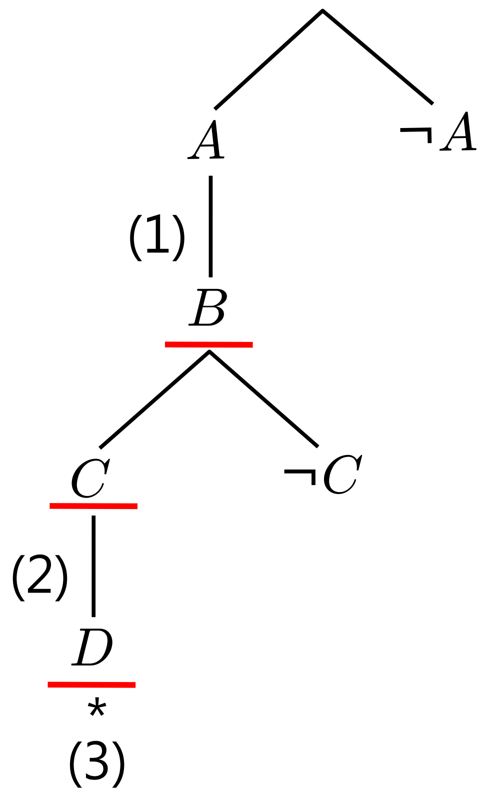


## Lemma Learning in DPLL

**"Avoid making the same mistake twice"**

$$\begin{array}{ll} & \dots \\ B \lor \neg A & (1) \\ D \lor \neg C & (2) \\ \underline{\neg D} \lor \underline{\neg B} \lor \underline{\neg C} & (3) \end{array}$$

**w/o Lemma**



## Lemma Learning in DPLL

**"Avoid making the same mistake twice"**

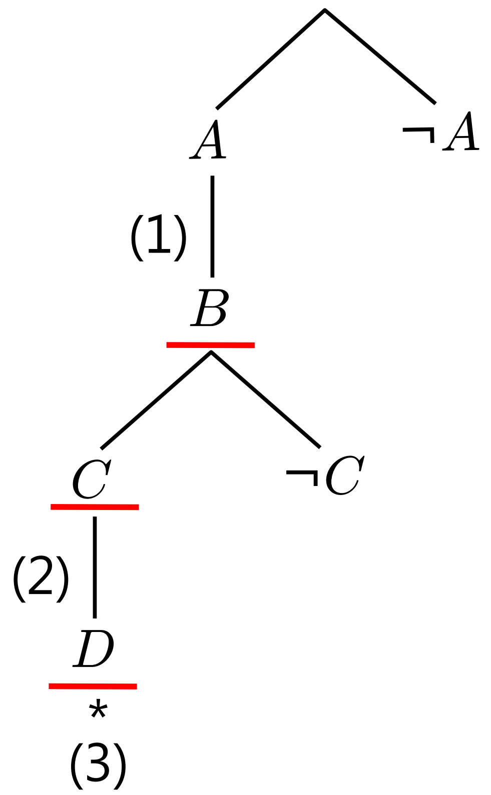
$\ldots$

| | |
|---|---|
| $B \lor \neg A$ | (1) |
| $D \lor \neg C$ | (2) |
| $\underline{\neg D} \lor \underline{\neg B} \lor \underline{\neg C}$ | (3) |

**Lemma Candidates by Resolution:**

$\underline{\neg D} \lor \neg B \lor \neg C$

**w/o Lemma**



## Lemma Learning in DPLL

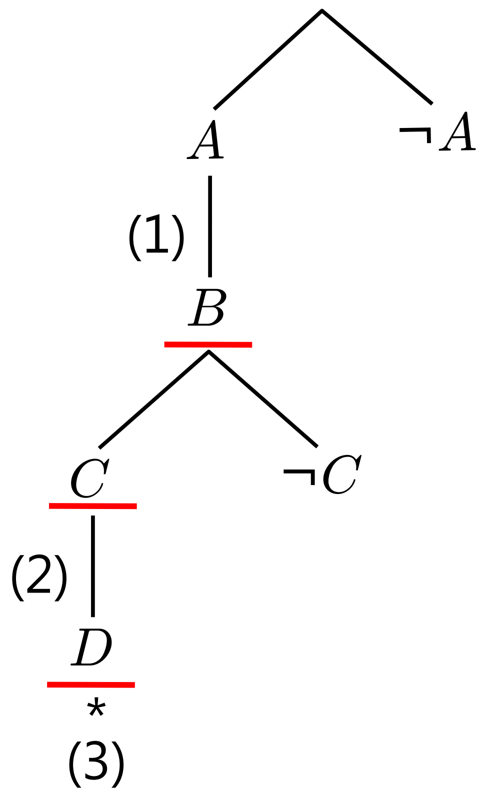
**"Avoid making the same mistake twice"**

$$\begin{array}{ll} \dots & \\ B \lor \neg A & (1) \\ D \lor \neg C & (2) \\ \underline{\neg D} \lor \underline{\neg B} \lor \underline{\neg C} & (3) \end{array}$$

**Lemma Candidates by Resolution:**

$$\frac{\underline{\neg D} \lor \neg B \lor \neg C \qquad \underline{D} \lor \neg C}{\boxed{\underline{\neg B} \lor \neg C}}$$

**w/o Lemma**



## Lemma Learning in DPLL

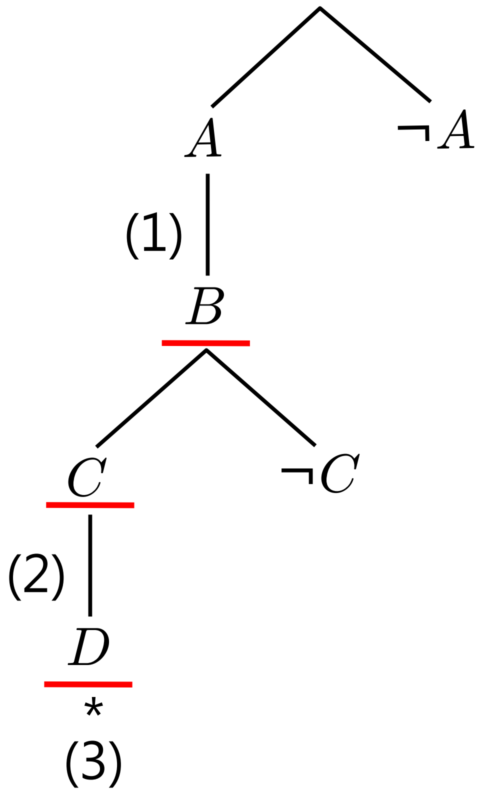
**"Avoid making the same mistake twice"**

$$\begin{array}{lr} & \ldots \\ B \vee \neg A & (1) \\ D \vee \neg C & (2) \\ \underline{\neg D} \vee \underline{\neg B} \vee \underline{\neg C} & (3) \end{array}$$

**Lemma Candidates by Resolution:**

$$\dfrac{\dfrac{\underline{\neg D} \vee \neg B \vee \neg C \qquad \underline{D} \vee \neg C}{\boxed{\underline{\neg B} \vee \neg C}} \qquad \underline{B} \vee \neg A}{\boxed{\neg C \vee \neg A}}$$

**w/o Lemma**



## Lemma Learning in DPLL

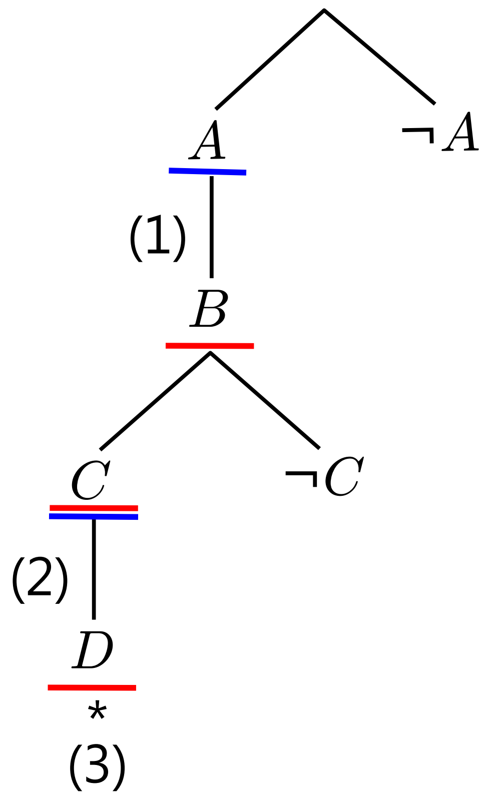
**"Avoid making the same mistake twice"**

| | |
|---|---|
| $\ldots$ | |
| $B \lor \neg A$ | (1) |
| $D \lor \neg C$ | (2) |
| $\underline{\neg D} \lor \underline{\neg B} \lor \underline{\neg C}$ | (3) |

**Lemma Candidates by Resolution:**

$$\dfrac{\dfrac{\underline{\neg D} \lor \neg B \lor \neg C \qquad \underline{D} \lor \neg C}{\boxed{\underline{\neg B} \lor \neg C}} \qquad \underline{B} \lor \neg A}{\boxed{\underline{\neg C \lor \neg A}}}$$

**w/o Lemma** **With Lemma**



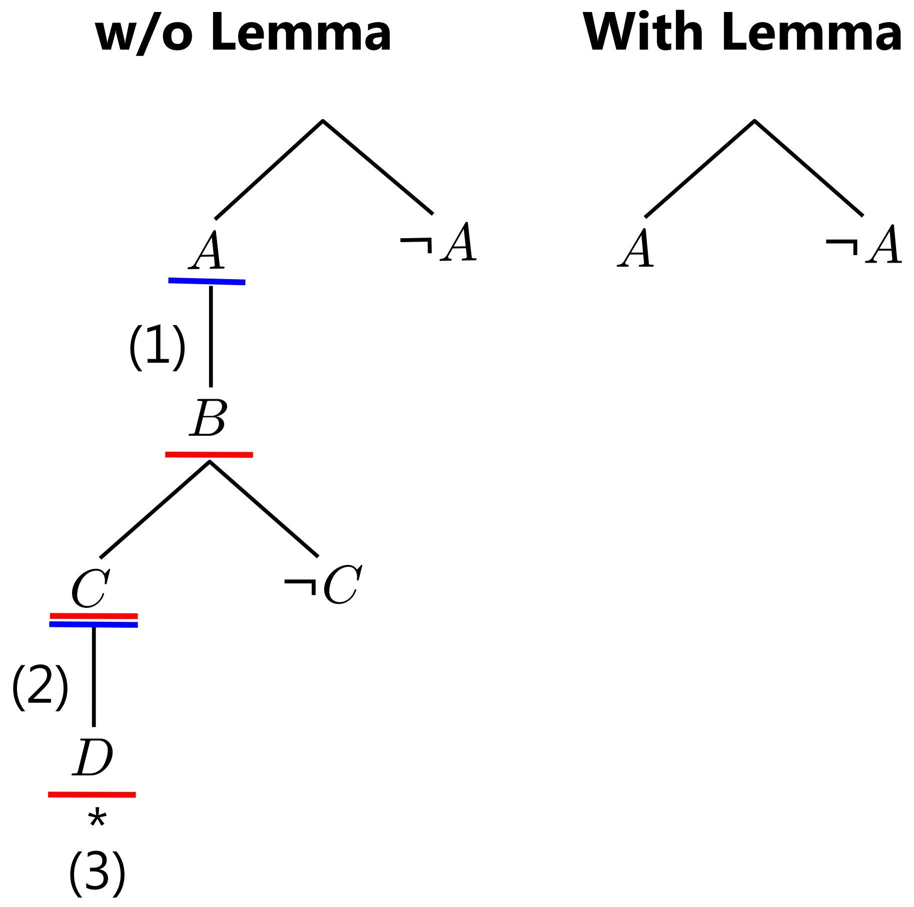
## Lemma Learning in DPLL

**"Avoid making the same mistake twice"**

$$\begin{array}{ll} & \ldots \\ B \vee \neg A & (1) \\ D \vee \neg C & (2) \\ \underline{\neg D} \vee \underline{\neg B} \vee \underline{\neg C} & (3) \end{array}$$

**Lemma Candidates by Resolution:**

$$\dfrac{\dfrac{\underline{\neg D} \vee \neg B \vee \neg C \qquad \underline{D} \vee \neg C}{\boxed{\underline{\neg B} \vee \neg C}} \qquad \underline{B} \vee \neg A}{\boxed{\underline{\neg C \vee \neg A}}}$$



# Lemma Learning in DPLL

**"Avoid making the same mistake twice"**

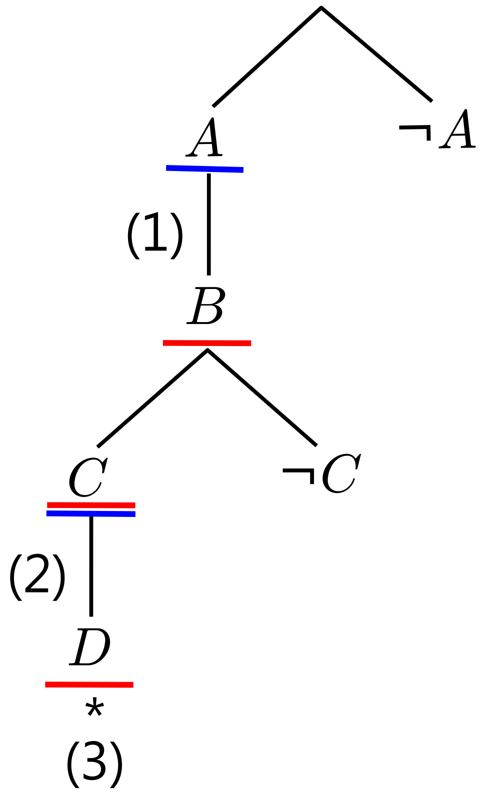
$$\ldots$$

| | |
|---|---|
| $B \vee \neg A$ | (1) |
| $D \vee \neg C$ | (2) |
| $\underline{\neg D} \vee \underline{\neg B} \vee \underline{\neg C}$ | (3) |

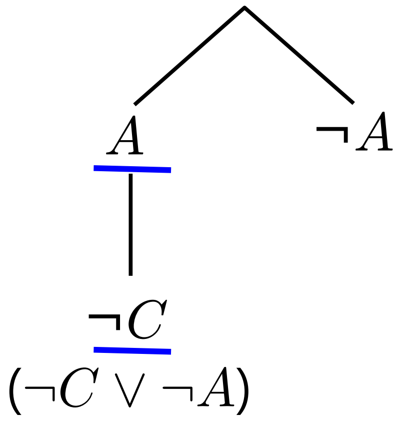
**Lemma Candidates by Resolution:**

$$\dfrac{\dfrac{\underline{\neg D} \vee \neg B \vee \neg C \qquad \underline{D} \vee \neg C}{\boxed{\underline{\neg B} \vee \neg C}} \qquad \underline{B} \vee \neg A}{\boxed{\underline{\neg C \vee \neg A}}}$$

**w/o Lemma**



**With Lemma**



## Making DPLL Fast

### Backjumping and Lemma Learning

- By construction, the lemma clause closes the branch: for each literal in the lemma clause $(\neg C \vee \neg A)$ there is a complementary literal on the branch ($C$ and $A$).
- On backtracking, the other branch literals ($B$ and $D$) can be ignored – they are not necessary to close the branch.
- In particular, if one of these literals is added to the branch by a splitting, that split can be skipped on backtracking ("non-chronological backtracking", "backjumping").

  (The example does not show this effect, it would have to contain splits inbetween $C$ and $D$)
- One may also restart the derivation right away with the new lemma clause added. Cf. the right tree in the example.

## Making DPLL Fast

### 2-watched literal technique

A technique to implement unit propagation efficiently.

- In each clause, select two (currently undefined) “watched” literals.
- For each variable $A$, keep a list of all clauses in which $A$ is watched and a list of all clauses in which $\neg A$ is watched.
- If an undefined variable is set to 0 (or to 1), check all clauses in which $A$ (or $\neg A$) is watched and watch another literal (that is true or undefined) in this clause if possible.
- As long as there are two watched literals in a $n$-literal clause, this clause cannot be used for unit propagation, because $n - 1$ of its literals have to be false to
- Important: Watched literal information need not be restored upon backtracking.

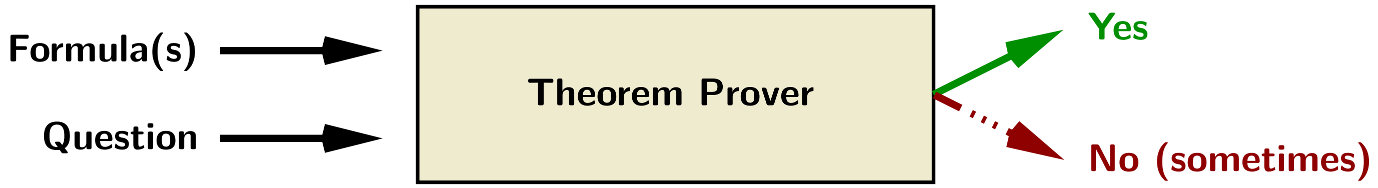
## Further Information

The ideas described so far heve been implemented in the SAT checker **zChaff.**

Further information:
Lintao Zhang and Sharad Malik: The Quest for Efficient Boolean Satisfiability Solvers, Proc. CADE-18, LNAI 2392, pp. 295–312, Springer, 2002.
Robert Nieuwenhuis, Albert Oliveras, Cesare Tinelli: Solvin SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland precedure to DPLL(T), pp 937–977, Journal of the ACM, 53(6), 2006.

## Contents

## First-Order Logic and Clause Normal Forms



Formula: First-order logic formula $\phi$ (e.g. the three-coloring spec above)

Usually with equality $\doteq$

Sometimes from syntactically resricted fragment (e.g., Description logics)

Question: Is $\phi$ formula valid? (satisfiable?, entailed by another formula?)

Calculi: Superposition (Resolution), Instance-based methods, Tableaux, ...

**Issues**

- Efficient treatment of equality
- Decision procedure for sub-languages or useful reductions?
- Built-in inference rules for arrays, lists, arithmetics (still open research)

## First-Order Logic

“The function $f$ is continuous”, expressed in (first-order) predicate logic:

$$\forall\varepsilon(0 < \varepsilon \to \forall a \exists\delta(0 < \delta \land \forall x(|x - a| < \delta \to |f(x) - f(a)| < \varepsilon)))$$

### Underlying Language

Variables $\varepsilon$, $a$, $\delta$, $x$

Function symbols $0$, $|\_|$, $\_ - \_$, $f(\_)$

Terms are well-formed expressions over variables and function symbols

Predicate symbols $\_ < \_$, $\_ = \_$

Atoms are applications of predicate symbols to terms

Boolean connectives $\land$, $\lor$, $\to$, $\neg$

Quantifiers $\forall$, $\exists$

The function symbols and predicate symbols comprise a signature $\Sigma$

## First-Order Logic

"The function $f$ is continuous", expressed in (first-order) predicate logic:

$$\forall\varepsilon(0 < \varepsilon \rightarrow \forall a \exists\delta(0 < \delta \land \forall x(|x - a| < \delta \rightarrow |f(x) - f(a)| < \varepsilon)))$$

### "Meaning" of Language Elements – Σ-Algebras

Universe (aka Domain): Set $U$

Variables $\mapsto$ values in $U$ (mapping is called "assignment")

Function symbols $\mapsto$ (total) functions over $U$

Predicate symbols $\mapsto$ relations over $U$

Boolean connectives $\mapsto$ the usual boolean functions

Quantifiers $\mapsto$ "for all ... holds", "there is a ..., such that"

Terms $\mapsto$ values in $U$

Formulas $\mapsto$ Boolean (Truth-) values

## Semantics - $\Sigma$-Algebra Example

Let $\Sigma_{PA}$ be the standard signature of Peano Arithmetic

The standard interpretation $\mathbb{N}$ for Peano Arithmetic then is:

$$
\begin{array}{rcl}
U_{\mathbb{N}} & = & \{0, 1, 2, \ldots\} \\
0_{\mathbb{N}} & = & 0 \\
s_{\mathbb{N}} & : & n \mapsto n + 1 \\
+_{\mathbb{N}} & : & (n, m) \mapsto n + m \\
*_{\mathbb{N}} & : & (n, m) \mapsto n * m \\
\leq_{\mathbb{N}} & = & \{(n, m) \mid n \text{ less than or equal to } m\} \\
<_{\mathbb{N}} & = & \{(n, m) \mid n \text{ less than } m\}
\end{array}
$$

Note that $\mathbb{N}$ is just one out of **many possible** $\Sigma_{PA}$-interpretations

## Semantics - Σ-Algebra Example

### Evaluation of terms and formulas

Under the interpretation $\mathbb{N}$ and the assignment $\beta : x \mapsto 1, y \mapsto 3$ (to evaluate the free variables) we obtain

$$\begin{array}{lcll}
(\mathbb{N}, \beta)(s(x) + s(0)) & = & 3 & \\
(\mathbb{N}, \beta)(x + y \doteq s(y)) & = & \mathit{True} & \\
(\mathbb{N}, \beta)(\forall z\ z \leq y) & = & \mathit{False} & \\
(\mathbb{N}, \beta)(\forall x \exists y\ x < y) & = & \mathit{True} & \\
\mathbb{N}(\forall x \exists y\ x < y) & = & \mathit{True} & \text{(Short notation when } \beta \text{ irrelevant)}
\end{array}$$

### Important Basic Notion: Model

If $\phi$ is a closed formula, then, instead of $I(\phi) = \mathit{True}$ one writes

$$I \models \phi \qquad \text{(“}I\text{ is a model of }\phi\text{”)}$$

E.g. $\mathbb{N} \models \forall x \exists y\ x < y$

## Reasoning Services Semantically

E.g. “entailment”:

$$\text{Axioms over } \mathbb{R} \land \text{continuous}(f) \land \text{continuous}(g) \models \text{continuous}(f+g) \ ?$$

### Services

Model($I$,$\phi$): $I \models \phi$ ? (Is $I$ a model for $\phi$?)

Validity($\phi$): $\models \phi$ ? ($I \models \phi$ for every interpretation?)

Satisfiability($\phi$): $\phi$ satisfiable? ($I \models \phi$ for some interpretation?)

Entailment($\phi$,$\psi$): $\phi \models \psi$ ? (does $\phi$ entail $\psi$?, i.e.
for every interpretation $I$: if $I \models \phi$ then $I \models \psi$?)

Solve($I$,$\phi$): find an assignment $\beta$ such that $(I, \beta)(\phi) = \mathit{True}$

Solve($\phi$): find an interpretation and assignment $\beta$ such that $(I, \beta)(\phi) = \mathit{True}$

Additional complication: fix interpretation of some symbols (as in $\mathbb{N}$ above)

## Reasoning Services Semantically

E.g. “entailment”:

$$\text{Axioms over } \mathbb{R} \wedge \text{continuous}(f) \wedge \text{continuous}(g) \models \text{continuous}(f+g) \ ?$$

### Services

Model($I$,$\phi$): $I \models \phi$ ? (Is $I$ a model for $\phi$?)

Validity($\phi$): $\models \phi$ ? ($I \models \phi$ for every interpretation?)

Satisfiability($\phi$): $\phi$ satisfiable? ($I \models \phi$ for some interpretation?)

Entailment($\phi$,$\psi$): $\phi \models \psi$ ? (does $\phi$ entail $\psi$?, i.e.
for every interpretation $I$: if $I \models \phi$ then $I \models \psi$?)

Solve($I$,$\phi$): find an assignment $\beta$ such that $(I, \beta)(\phi) = \mathit{True}$

Solve($\phi$): find an interpretation and assignment $\beta$ such that $(I, \beta)(\phi) = \mathit{True}$

Additional complication: fix interpretation of some symbols (as in $\mathbb{N}$ above)

**What if theorem prover’s native service is only “Is $\phi$ unsatisfiable?” ?**

## Reduction to Unsatisfiability

- Suppose we want to prove an entailment $\phi \models \psi$

## Reduction to Unsatisfiability

- Suppose we want to prove an entailment $\phi \models \psi$
- Equivalently, prove $\models \phi \rightarrow \psi$, i.e. that $\phi \rightarrow \psi$ is valid

## Reduction to Unsatisfiability

- Suppose we want to prove an entailment $\phi \models \psi$
- Equivalently, prove $\models \phi \rightarrow \psi$, i.e. that $\phi \rightarrow \psi$ is valid
- Equivalently, prove that $\neg(\phi \rightarrow \psi)$ is not satisfiable (unsatisfiable)

## Reduction to Unsatisfiability

- Suppose we want to prove an entailment $\phi \models \psi$
- Equivalently, prove $\models \phi \rightarrow \psi$, i.e. that $\phi \rightarrow \psi$ is valid
- Equivalently, prove that $\neg(\phi \rightarrow \psi)$ is not satisfiable (unsatisfiable)
- Equivalently, prove that $\phi \wedge \neg\psi$ is unsatisfiable

## Reduction to Unsatisfiability

- Suppose we want to prove an entailment $\phi \models \psi$
- Equivalently, prove $\models \phi \rightarrow \psi$, i.e. that $\phi \rightarrow \psi$ is valid
- Equivalently, prove that $\neg(\phi \rightarrow \psi)$ is not satisfiable (unsatisfiable)
- Equivalently, prove that $\phi \land \neg\psi$ is unsatisfiable

**Basis for (predominant) refutational theorem proving**

## Reduction to Unsatisfiability

- Suppose we want to prove an entailment $\phi \models \psi$
- Equivalently, prove $\models \phi \rightarrow \psi$, i.e. that $\phi \rightarrow \psi$ is valid
- Equivalently, prove that $\neg(\phi \rightarrow \psi)$ is not satisfiable (unsatisfiable)
- Equivalently, prove that $\phi \wedge \neg\psi$ is unsatisfiable

**Basis for (predominant) refutational theorem proving**

Dual problem, much harder: to disprove an entailment $\phi \models \psi$ find a model of $\phi \wedge \neg\psi$

**One motivation for (finite) model generation procedures**

## Normal Forms

Most first-order theorem provers take formulas in **clause normal form**

### Why Normal Forms?

- Reduction of logical concepts (operators, quantifiers)
- Reduction of syntactical structure (nesting of subformulas)
- Can be exploited for efficient data structures and control

## Normal Forms

Most first-order theorem provers take formulas in **clause normal form**

### Why Normal Forms?

- Reduction of logical concepts (operators, quantifiers)
- Reduction of syntactical structure (nesting of subformulas)
- Can be exploited for efficient data structures and control

### Translation into Clause Normal Form

$$\begin{tikzcd}
\textbf{Formula} \arrow[r] & \textbf{Prenex normal form} \arrow[r] & \textbf{Skolem normal form} \arrow[r] & \textbf{Clause normal form} \arrow[r] & \boxed{\textbf{Clausal Theorem Prover}}
\end{tikzcd}$$

**Prop:** the given formula and its clause normal form are equi-satisfiable

## Prenex Normal Form

**Prenex formulas** have the form

$$Q_1 x_1 \ldots Q_n x_n \; F,$$

where $F$ is quantifier-free and $Q_i \in \{\forall, \exists\}$

## Prenex Normal Form

**Prenex formulas** have the form

$$Q_1 x_1 \ldots Q_n x_n \; F,$$

where $F$ is quantifier-free and $Q_i \in \{\forall, \exists\}$

Computing prenex normal form by the rewrite relation $\Rightarrow_P$:

$$\begin{array}{rcll}
(F \leftrightarrow G) & \Rightarrow_P & (F \rightarrow G) \wedge (G \rightarrow F) & \\
\neg Q x F & \Rightarrow_P & \overline{Q} x \neg F & (\neg Q) \\
(Q x F \; \rho \; G) & \Rightarrow_P & Q y (F[y/x] \; \rho \; G), \; y \text{ fresh}, \; \rho \in \{\wedge, \vee\} & \\
(Q x F \rightarrow G) & \Rightarrow_P & \overline{Q} y (F[y/x] \rightarrow G), \; y \text{ fresh} & \\
(F \; \rho \; Q x G) & \Rightarrow_P & Q y (F \; \rho \; G[y/x]), \; y \text{ fresh}, \; \rho \in \{\wedge, \vee, \rightarrow\} &
\end{array}$$

Here $\overline{Q}$ denotes the quantifier **dual** to $Q$, i.e., $\overline{\forall} = \exists$ and $\overline{\exists} = \forall$.

## In the Example

$$\forall\varepsilon(0 < \varepsilon \rightarrow \forall a\exists\delta(0 < \delta \land \forall x(|x - a| < \delta \rightarrow |f(x) - f(a)| < \varepsilon)))$$

$$\Rightarrow_P$$

$$\forall\varepsilon\forall a(0 < \varepsilon \rightarrow \exists\delta(0 < \delta \land \forall x(|x - a| < \delta \rightarrow |f(x) - f(a)| < \varepsilon)))$$

$$\Rightarrow_P$$

$$\forall\varepsilon\forall a\exists\delta(0 < \varepsilon \rightarrow 0 < \delta \land \forall x(|x - a| < \delta \rightarrow |f(x) - f(a)| < \varepsilon))$$

$$\Rightarrow_P$$

$$\forall\varepsilon\forall a\exists\delta(0 < \varepsilon \rightarrow \forall x(0 < \delta \land |x - a| < \delta \rightarrow |f(x) - f(a)| < \varepsilon))$$

$$\Rightarrow_P$$

$$\forall\varepsilon\forall a\exists\delta\forall x(0 < \varepsilon \rightarrow (0 < \delta \land (|x - a| < \delta \rightarrow |f(x) - f(a)| < \varepsilon)))$$

## Skolem Normal Form

$$\begin{tikzcd}
\textbf{Formula} \arrow[r] & \textbf{Prenex normal form} \arrow[r] & \textbf{Skolem normal form} \arrow[r] & \textbf{Clause normal form} \arrow[r] & \boxed{\textbf{Clausal Theorem Prover}}
\end{tikzcd}$$

**Intuition:** replacement of $\exists y$ by a concrete choice function computing $y$ from all the arguments $y$ depends on.

Transformation $\Rightarrow_S$

$$\forall x_1, \ldots, x_n \exists y\ F \quad \Rightarrow_S \quad \forall x_1, \ldots, x_n\ F[f(x_1, \ldots, x_n)/y]$$

where $f/n$ is a new function symbol (**Skolem function**).

## Skolem Normal Form

$$\begin{tikzcd}\text{Formula} \arrow[r] & \text{Prenex normal form} \arrow[r] & \text{Skolem normal form} \arrow[r] & \text{Clause normal form} \arrow[r] & \boxed{\text{Clausal Theorem Prover}}\end{tikzcd}$$

**Intuition:** replacement of $\exists y$ by a concrete choice function computing $y$ from all the arguments $y$ depends on.

Transformation $\Rightarrow_S$

$$\forall x_1, \ldots, x_n \exists y \; F \quad \Rightarrow_S \quad \forall x_1, \ldots, x_n \; F[f(x_1, \ldots, x_n)/y]$$

where $f/n$ is a new function symbol (**Skolem function**).

### In the Example

$$\forall \varepsilon \forall a \exists \delta \forall x (0 < \varepsilon \rightarrow 0 < \delta \wedge (|x - a| < \delta \rightarrow |f(x) - f(a)| < \varepsilon))$$

$$\Rightarrow_S$$

$$\forall \varepsilon \forall a \forall x (0 < \varepsilon \rightarrow 0 < d(\varepsilon, a) \wedge (|x - a| < d(\varepsilon, a) \rightarrow |f(x) - f(a)| < \varepsilon))$$

## Clausal Normal Form (Conjunctive Normal Form)

Rules to convert the matrix of the formula in Skolem normal form into a conjunction of disjunctions:

$$
\begin{array}{rcl}
(F \leftrightarrow G) & \Rightarrow_K & (F \rightarrow G) \wedge (G \rightarrow F) \\
(F \rightarrow G) & \Rightarrow_K & (\neg F \vee G) \\
\neg(F \vee G) & \Rightarrow_K & (\neg F \wedge \neg G) \\
\neg(F \wedge G) & \Rightarrow_K & (\neg F \vee \neg G) \\
\neg\neg F & \Rightarrow_K & F \\
(F \wedge G) \vee H & \Rightarrow_K & (F \vee H) \wedge (G \vee H) \\
(F \wedge \top) & \Rightarrow_K & F \\
(F \wedge \bot) & \Rightarrow_K & \bot \\
(F \vee \top) & \Rightarrow_K & \top \\
(F \vee \bot) & \Rightarrow_K & F
\end{array}
$$

They are to be applied modulo commutativity of $\wedge$ and $\vee$

## In the Example

$$\forall\varepsilon\forall a\forall x(0 < \varepsilon \to 0 < d(\varepsilon, a) \land (|x - a| < d(\varepsilon, a) \to |f(x) - f(a)| < \varepsilon))$$

$$\Rightarrow_K$$

$$0 < d(\varepsilon, a) \;\lor\; \neg(0 < \varepsilon)$$

$$\neg(|x - a| < d(\varepsilon, a)) \;\lor\; |f(x) - f(a)| < \varepsilon \;\lor\; \neg(0 < \varepsilon)$$

**Note:** The universal quantifiers for the variables $\varepsilon$, $a$ and $x$, as well as the conjunction symbol $\land$ between the clauses are not written, for convenience

## The Complete Picture

$$\begin{aligned} F \quad & \overset{*}{\Rightarrow}_P \quad Q_1 y_1 \ldots Q_n y_n \ G && (G \text{ quantifier-free}) \\ & \overset{*}{\Rightarrow}_S \quad \forall x_1, \ldots, x_m \ H && (m \leq n,\ H \text{ quantifier-free}) \\ & \overset{*}{\Rightarrow}_K \quad \underbrace{\underbrace{\forall x_1, \ldots, x_m}_{\text{leave out}} \ \bigwedge_{i=1}^{k} \underbrace{\bigvee_{j=1}^{n_i} L_{ij}}_{\text{clauses } C_i}}_{F'} \end{aligned}$$

$N = \{C_1, \ldots, C_k\}$ is called the **clausal (normal) form** (CNF) of $F$

**Note:** the variables in the clauses are implicitly universally quantified

## The Complete Picture

$$
\begin{aligned}
F \quad & \overset{*}{\Rightarrow}_P \quad Q_1 y_1 \ldots Q_n y_n \; G && (G \text{ quantifier-free}) \\
& \overset{*}{\Rightarrow}_S \quad \forall x_1, \ldots, x_m \; H && (m \leq n,\; H \text{ quantifier-free}) \\
& \overset{*}{\Rightarrow}_K \quad \underbrace{\underbrace{\forall x_1, \ldots, x_m}_{\text{leave out}} \; \bigwedge_{i=1}^{k} \; \underbrace{\bigvee_{j=1}^{n_i} L_{ij}}_{\text{clauses } C_i}}_{F'}
\end{aligned}
$$

$N = \{C_1, \ldots, C_k\}$ is called the **clausal (normal) form** (CNF) of $F$

**Note:** the variables in the clauses are implicitly universally quantified

**Instead of showing that $F$ is unsatisfiable, the proof problem from now is to show that $N$ is unsatisfiable**

## The Complete Picture

$$F \overset{*}{\Rightarrow}_P Q_1 y_1 \ldots Q_n y_n \; G \qquad (G \text{ quantifier-free})$$

$$\overset{*}{\Rightarrow}_S \forall x_1, \ldots, x_m \; H \qquad (m \leq n,\; H \text{ quantifier-free})$$

$$\overset{*}{\Rightarrow}_K \underbrace{\underbrace{\forall x_1, \ldots, x_m}_{\text{leave out}} \bigwedge_{i=1}^{k} \underbrace{\bigvee_{j=1}^{n_i} L_{ij}}_{\text{clauses } C_i}}_{F'}$$

$N = \{C_1, \ldots, C_k\}$ is called the **clausal (normal) form** (CNF) of $F$

**Note:** the variables in the clauses are implicitly universally quantified

**Instead of showing that $F$ is unsatisfiable, the proof problem from now is to show that $N$ is unsatisfiable**

**Can do better than "searching through all interpretations"**

**Theorem: $N$ is satisfiable iff it has a Herbrand model**

## Contents

## Proof Procedures Based on Herbrand's Theorem

A **Herbrand interpretation** (over a given signature $\Sigma$) is a $\Sigma$-algebra $\mathcal{A}$ such that

## Proof Procedures Based on Herbrand's Theorem

A **Herbrand interpretation** (over a given signature $\Sigma$) is a $\Sigma$-algebra $\mathcal{A}$ such that

- The universe is the set $\mathsf{T}_\Sigma$ of ground terms over $\Sigma$
  (a **ground term** is a term without any variables ):

$$U_{\mathcal{A}} = \mathsf{T}_\Sigma$$

## Proof Procedures Based on Herbrand's Theorem

A **Herbrand interpretation** (over a given signature $\Sigma$) is a $\Sigma$-algebra $\mathcal{A}$ such that

- The universe is the set $\mathsf{T}_\Sigma$ of ground terms over $\Sigma$ (a **ground term** is a term without any variables ):

$$U_{\mathcal{A}} = \mathsf{T}_\Sigma$$

- Every function symbol from $\Sigma$ is "mapped to itself":

$$f_{\mathcal{A}} : (s_1, \ldots, s_n) \mapsto f(s_1, \ldots, s_n), \text{ where } f \text{ is } n\text{-ary function symbol in } \Sigma$$

## Proof Procedures Based on Herbrand's Theorem

A **Herbrand interpretation** (over a given signature $\Sigma$) is a $\Sigma$-algebra $\mathcal{A}$ such that

- The universe is the set $\mathsf{T}_\Sigma$ of ground terms over $\Sigma$ (a **ground term** is a term without any variables ):

$$U_{\mathcal{A}} = \mathsf{T}_\Sigma$$

- Every function symbol from $\Sigma$ is "mapped to itself":

$$f_{\mathcal{A}} : (s_1, \ldots, s_n) \mapsto f(s_1, \ldots, s_n), \text{ where } f \text{ is } n\text{-ary function symbol in } \Sigma$$

Example

- $\Sigma_{Pres} = (\{0/0, s/1, +/2\}, \ \{</2, \leq/2\})$
- $$U_{\mathcal{A}} = \{0, s(0), s(s(0)), \ldots, 0+0, s(0)+0, \ldots, s(0+0), s(s(0)+0), \ldots\}$$
- $0 \mapsto 0, s(0) \mapsto s(0), s(s(0)) \mapsto s(s(0)), \ldots, 0+0 \mapsto 0+0, \ldots$

## Herbrand Interpretations

Only interpretations $p_{\mathcal{A}}$ of predicate symbols $p \in \Sigma$ is undetermined in a Herbrand interpretation

- $p_{\mathcal{A}}$ represented as the set of ground atoms

  $\{p(s_1, \ldots, s_n) \mid (s_1, \ldots, s_n) \in p_{\mathcal{A}}$ where $p \in \Sigma$ is $n$-ary predicate symbol$\}$

- Whole interpretation represented as $\bigcup_{p \in \Sigma} p_{\mathcal{A}}$

### Example

- $\Sigma_{Pres} = (\{0/0, s/1, +/2\},\ \{</2, \leq/2\})$ (from above)
- $\mathbb{N}$ as Herbrand interpretation over $\Sigma_{Pres}$

$$\begin{aligned} I = \{ \quad & 0 \leq 0,\ 0 \leq s(0),\ 0 \leq s(s(0)),\ \ldots, \\ & 0+0 \leq 0,\ 0+0 \leq s(0),\ \ldots, \\ & \ldots,\ (s(0)+0)+s(0) \leq s(0)+(s(0)+s(0)), \ldots \quad \} \end{aligned}$$

## Herbrand’s Theorem

**Proposition**

A Skolem normal form $\forall\phi$ is unsatisfiable iff it has no Herbrand model

## Herbrand's Theorem

**Proposition**

A Skolem normal form $\forall\phi$ is unsatisfiable iff it has no Herbrand model

**Theorem (Skolem-Herbrand-Theorem)**

$\forall\phi$ has no Herbrand model iff some finite set of ground instances $\{\phi\gamma_1, \ldots, \phi\gamma_n\}$ is unsatisfiable

## Herbrand's Theorem

**Proposition**

A Skolem normal form $\forall\phi$ is unsatisfiable iff it has no Herbrand model

**Theorem (Skolem-Herbrand-Theorem)**

$\forall\phi$ has no Herbrand model iff some finite set of ground instances $\{\phi\gamma_1, \ldots, \phi\gamma_n\}$ is unsatisfiable

Applied to clause logic:

**Theorem (Skolem-Herbrand-Theorem)**

A set $N$ of $\Sigma$-clauses is unsatisfiable iff some finite set of ground instances of clauses from $N$ is unsatisfiable

## Herbrand's Theorem

**Proposition**

A Skolem normal form $\forall\phi$ is unsatisfiable iff it has no Herbrand model

**Theorem (Skolem-Herbrand-Theorem)**

$\forall\phi$ has no Herbrand model iff some finite set of ground instances $\{\phi\gamma_1, \ldots, \phi\gamma_n\}$ is unsatisfiable

Applied to clause logic:

**Theorem (Skolem-Herbrand-Theorem)**

A set $N$ of $\Sigma$-clauses is unsatisfiable iff some finite set of ground instances of clauses from $N$ is unsatisfiable

**Leads immediately to theorem prover "Gilmore's Method"**

## Gilmore's Method - Based on Herbrand's Theorem

**Preprocessing:**

Given Formula

Clause Form

$$\begin{tikzcd} \boxed{\begin{array}{l}\forall x\ \exists y\ P(y,x)\\ \ \land \forall z\ \neg P(z,a)\end{array}} \arrow[r] & \boxed{\begin{array}{l}P(f(x),x)\\ \neg P(z,a)\end{array}} \end{tikzcd}$$

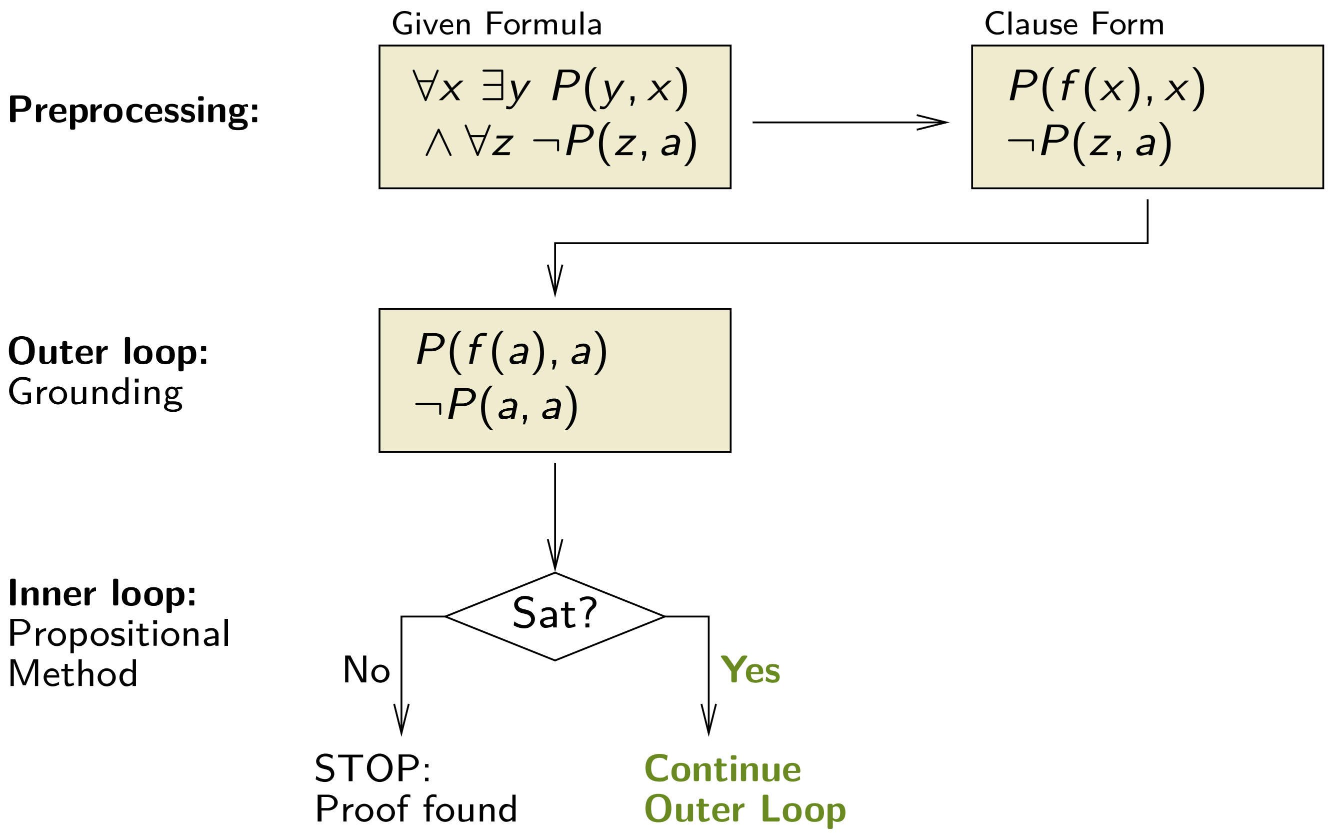
**Outer loop:**
Grounding

**Inner loop:**
Propositional
Method

## Gilmore's Method - Based on Herbrand's Theorem

$$\begin{tikzcd}
 & \text{Given Formula} & \text{Clause Form} \\
\textbf{Preprocessing:} & \boxed{\begin{array}{l} \forall x\ \exists y\ P(y,x) \\ \ \land \forall z\ \neg P(z,a) \end{array}} \arrow[r] & \boxed{\begin{array}{l} P(f(x),x) \\ \neg P(z,a) \end{array}} \arrow[dl] \\
\begin{array}{l} \textbf{Outer loop:} \\ \text{Grounding} \end{array} & \boxed{\begin{array}{l} P(f(a),a) \\ \neg P(a,a) \end{array}} & \\
\begin{array}{l} \textbf{Inner loop:} \\ \text{Propositional} \\ \text{Method} \end{array} & &
\end{tikzcd}$$

## Gilmore’s Method - Based on Herbrand’s Theorem



## Gilmore's Method - Based on Herbrand's Theorem

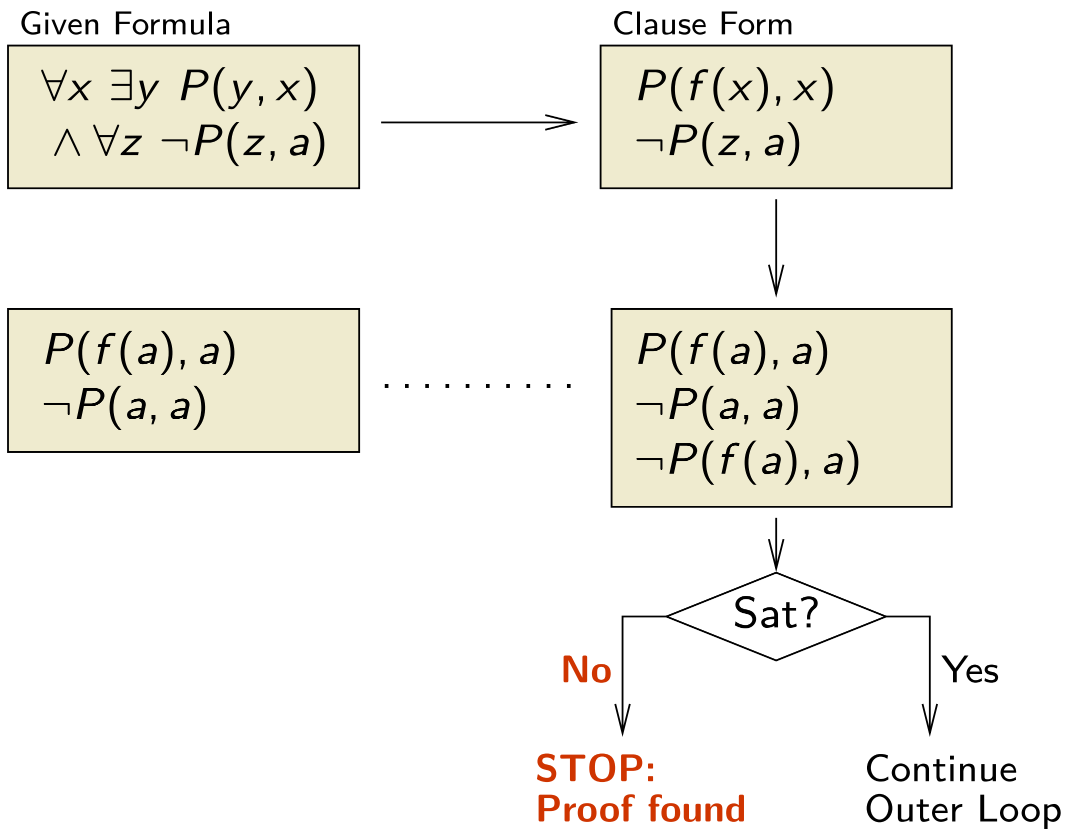
$$\begin{tikzcd}
 & \text{Given Formula} & \text{Clause Form} \\
\textbf{Preprocessing:} & \boxed{\begin{array}{l}\forall x\ \exists y\ P(y,x)\\ \ \land \forall z\ \neg P(z,a)\end{array}} \arrow[r] & \boxed{\begin{array}{l}P(f(x),x)\\ \neg P(z,a)\end{array}} \arrow[d] \\
\begin{array}{l}\textbf{Outer loop:}\\ \text{Grounding}\end{array} & \boxed{\begin{array}{l}P(f(a),a)\\ \neg P(a,a)\end{array}} \arrow[r, dash, dotted] & \boxed{\begin{array}{l}P(f(a),a)\\ \neg P(a,a)\\ \neg P(f(a),a)\end{array}} \\
\begin{array}{l}\textbf{Inner loop:}\\ \text{Propositional}\\ \text{Method}\end{array} & &
\end{tikzcd}$$

# Gilmore's Method - Based on Herbrand's Theorem

**Preprocessing:**

**Outer loop:**
Grounding

**Inner loop:**
Propositional
Method



## Contents

## The Resolution Calculus

- Gilmore's method reduces proof search in first-order logic to propositional logic unsatisfiability problems

## The Resolution Calculus

- Gilmore's method reduces proof search in first-order logic to propositional logic unsatisfiability problems
- Main problem is the unguided generation of (very many) ground clauses

## The Resolution Calculus

- Gilmore's method reduces proof search in first-order logic to propositional logic unsatisfiability problems
- Main problem is the unguided generation of (very many) ground clauses
- All modern calculi address this problem in one way or another, e.g.
  - **Avoidance:** Resolution calculi do not need to generate the ground instances at all

    Resolution inferences operate directly on clauses, not on their ground instances
  - **Guidance:** Instance-Based Methods are similar to Gilmore's method but generate ground instances in a guided way (see below)

## The Resolution Calculus

- Gilmore's method reduces proof search in first-order logic to propositional logic unsatisfiability problems
- Main problem is the unguided generation of (very many) ground clauses
- All modern calculi address this problem in one way or another, e.g.
  - **Avoidance:** Resolution calculi do not need to generate the ground instances at all

    Resolution inferences operate directly on clauses, not on their ground instances
  - **Guidance:** Instance-Based Methods are similar to Gilmore's method but generate ground instances in a guided way (see below)

**Next: propositional Resolution, lifting, first-order Resolution**

## The Propositional Resolution Calculus *Res*

Modern versions of the first-order version of the resolution calculus [Robinson 1965] are (still) the most important calculi for FOTP today.

**Propositional resolution inference rule**:

$$\frac{C \vee A \qquad \neg A \vee D}{C \vee D}$$

Terminology: $C \vee D$: **resolvent**; $A$: **resolved atom**

**Propositional (positive) factorisation inference rule**:

$$\frac{C \vee A \vee A}{C \vee A}$$

These are **schematic inference rules**:

$C$ and $D$ – propositional clauses

$A$ – propositional atom

"$\vee$" is considered associative and commutative

## Sample Proof

| | | |
|---|---|---|
| 1. | $\neg A \lor \neg A \lor B$ | (given) |
| 2. | $A \lor B$ | (given) |
| 3. | $\neg C \lor \neg B$ | (given) |
| 4. | $C$ | (given) |

## Sample Proof

| | | |
|---|---|---|
| 1. | $\neg A \vee \neg A \vee B$ | (given) |
| 2. | $A \vee B$ | (given) |
| 3. | $\neg C \vee \neg B$ | (given) |
| 4. | $C$ | (given) |
| 5. | $\neg A \vee B \vee B$ | (Res. 2. into 1.) |

## Sample Proof

| | | |
|---|---|---|
| 1. | $\neg A \vee \neg A \vee B$ | (given) |
| 2. | $A \vee B$ | (given) |
| 3. | $\neg C \vee \neg B$ | (given) |
| 4. | $C$ | (given) |
| 5. | $\neg A \vee B \vee B$ | (Res. 2. into 1.) |
| 6. | $\neg A \vee B$ | (Fact. 5.) |

## Sample Proof

| | | |
|---|---|---|
| 1. | $\neg A \vee \neg A \vee B$ | (given) |
| 2. | $A \vee B$ | (given) |
| 3. | $\neg C \vee \neg B$ | (given) |
| 4. | $C$ | (given) |
| 5. | $\neg A \vee B \vee B$ | (Res. 2. into 1.) |
| 6. | $\neg A \vee B$ | (Fact. 5.) |
| 7. | $B \vee B$ | (Res. 2. into 6.) |

## Sample Proof

|  |  |  |
|---|---|---|
| 1. | $\neg A \vee \neg A \vee B$ | (given) |
| 2. | $A \vee B$ | (given) |
| 3. | $\neg C \vee \neg B$ | (given) |
| 4. | $C$ | (given) |
| 5. | $\neg A \vee B \vee B$ | (Res. 2. into 1.) |
| 6. | $\neg A \vee B$ | (Fact. 5.) |
| 7. | $B \vee B$ | (Res. 2. into 6.) |
| 8. | $B$ | (Fact. 7.) |

## Sample Proof

| | | |
|---|---|---|
| 1. | $\neg A \vee \neg A \vee B$ | (given) |
| 2. | $A \vee B$ | (given) |
| 3. | $\neg C \vee \neg B$ | (given) |
| 4. | $C$ | (given) |
| 5. | $\neg A \vee B \vee B$ | (Res. 2. into 1.) |
| 6. | $\neg A \vee B$ | (Fact. 5.) |
| 7. | $B \vee B$ | (Res. 2. into 6.) |
| 8. | $B$ | (Fact. 7.) |
| 9. | $\neg C$ | (Res. 8. into 3.) |

## Sample Proof

| | | |
|---|---|---|
| 1. | $\neg A \vee \neg A \vee B$ | (given) |
| 2. | $A \vee B$ | (given) |
| 3. | $\neg C \vee \neg B$ | (given) |
| 4. | $C$ | (given) |
| 5. | $\neg A \vee B \vee B$ | (Res. 2. into 1.) |
| 6. | $\neg A \vee B$ | (Fact. 5.) |
| 7. | $B \vee B$ | (Res. 2. into 6.) |
| 8. | $B$ | (Fact. 7.) |
| 9. | $\neg C$ | (Res. 8. into 3.) |
| 10. | $\perp$ | (Res. 4. into 9.) |

## Soundness of Propositional Resolution

**Proposition**

Propositional resolution is sound

**Proof:**

Let $I \in \Sigma$-Alg. To be shown:

1. for resolution: $I \models C \vee A,\ I \models D \vee \neg A \ \Rightarrow\ I \models C \vee D$
2. for factorization: $I \models C \vee A \vee A \ \Rightarrow\ I \models C \vee A$

## Soundness of Propositional Resolution

**Proposition**

Propositional resolution is sound

**Proof:**

Let $I \in \Sigma$-Alg. To be shown:

1. for resolution: $I \models C \vee A,\ I \models D \vee \neg A \ \Rightarrow\ I \models C \vee D$
2. for factorization: $I \models C \vee A \vee A \ \Rightarrow\ I \models C \vee A$

Ad (i): Assume premises are valid in $I$. Two cases need to be considered:
(a) $A$ is valid in $I$, or (b) $\neg A$ is valid in $I$.

## Soundness of Propositional Resolution

**Proposition**

Propositional resolution is sound

**Proof:**

Let $I \in \Sigma$-Alg. To be shown:

1. for resolution: $I \models C \vee A,\ I \models D \vee \neg A \ \Rightarrow\ I \models C \vee D$
2. for factorization: $I \models C \vee A \vee A \ \Rightarrow\ I \models C \vee A$

Ad (i): Assume premises are valid in $I$. Two cases need to be considered:
(a) $A$ is valid in $I$, or (b) $\neg A$ is valid in $I$.

a) $I \models A \Rightarrow I \models D \Rightarrow I \models C \vee D$

## Soundness of Propositional Resolution

**Proposition**

Propositional resolution is sound

**Proof:**

Let $I \in \Sigma$-Alg. To be shown:

1. for resolution: $I \models C \vee A,\ I \models D \vee \neg A \ \Rightarrow\ I \models C \vee D$
2. for factorization: $I \models C \vee A \vee A \ \Rightarrow\ I \models C \vee A$

Ad (i): Assume premises are valid in $I$. Two cases need to be considered: (a) $A$ is valid in $I$, or (b) $\neg A$ is valid in $I$.

a) $I \models A \Rightarrow I \models D \Rightarrow I \models C \vee D$

b) $I \models \neg A \Rightarrow I \models C \Rightarrow I \models C \vee D$

## Soundness of Propositional Resolution

**Proposition**

Propositional resolution is sound

**Proof:**

Let $I \in \Sigma$-Alg. To be shown:

1. for resolution: $I \models C \vee A,\ I \models D \vee \neg A \ \Rightarrow\ I \models C \vee D$
2. for factorization: $I \models C \vee A \vee A \ \Rightarrow\ I \models C \vee A$

Ad (i): Assume premises are valid in $I$. Two cases need to be considered:
(a) $A$ is valid in $I$, or (b) $\neg A$ is valid in $I$.

a) $I \models A \Rightarrow I \models D \Rightarrow I \models C \vee D$

b) $I \models \neg A \Rightarrow I \models C \Rightarrow I \models C \vee D$

Ad (ii): even simpler

## Completeness of Propositional Resolution

**Theorem:**

Propositional Resolution is refutationally complete

## Completeness of Propositional Resolution

**Theorem:**

Propositional Resolution is refutationally complete

- That is, if a propositional clause set is unsatisfiable, then Resolution will derive the empty clause $\bot$ eventually

## Completeness of Propositional Resolution

**Theorem:**

Propositional Resolution is refutationally complete

- That is, if a propositional clause set is unsatisfiable, then Resolution will derive the empty clause $\perp$ eventually
- More precisely: If a clause set is unsatisfiable and closed under the application of the Resolution and Factorization inference rules, then it contains the empty clause $\perp$

## Completeness of Propositional Resolution

**Theorem:**

Propositional Resolution is refutationally complete

- That is, if a propositional clause set is unsatisfiable, then Resolution will derive the empty clause $\perp$ eventually
- More precisely: If a clause set is unsatisfiable and closed under the application of the Resolution and Factorization inference rules, then it contains the empty clause $\perp$
- Perhaps easiest proof: semantic tree proof technique (see blackboard)

## Completeness of Propositional Resolution

**Theorem:**

Propositional Resolution is refutationally complete

- That is, if a propositional clause set is unsatisfiable, then Resolution will derive the empty clause $\perp$ eventually
- More precisely: If a clause set is unsatisfiable and closed under the application of the Resolution and Factorization inference rules, then it contains the empty clause $\perp$
- Perhaps easiest proof: semantic tree proof technique (see blackboard)
- This result can be considerably strengthened, some strengthenings come for free from the proof

## Completeness of Propositional Resolution

**Theorem:**

Propositional Resolution is refutationally complete

- That is, if a propositional clause set is unsatisfiable, then Resolution will derive the empty clause $\perp$ eventually
- More precisely: If a clause set is unsatisfiable and closed under the application of the Resolution and Factorization inference rules, then it contains the empty clause $\perp$
- Perhaps easiest proof: semantic tree proof technique (see blackboard)
- This result can be considerably strengthened, some strengthenings come for free from the proof

**Propositional resolution is not suitable for first-order clause sets**

## General Resolution

Propositional resolution:

- refutationally complete,
- in its most naive version: not guaranteed to terminate for satisfiable sets of clauses, (improved versions do terminate, however)
- in practice clearly inferior to the DPLL procedure (even with various improvements).

But: in contrast to the DPLL procedure, resolution can be easily extended to non-ground clauses (but see below First-order DPLL)

## General Resolution through Instantiation

Idea: instantiate clauses appropriately:

$$\begin{tikzcd}
P(z',z') \lor \neg Q(z) \arrow[d, "{[a/z', f(a,b)/z]}"'] & \neg P(a,y) \arrow[d, "{[a/y]}"'] \arrow[dr, "{[b/y]}"] & & P(x',b) \lor Q(f(x',x)) \arrow[d, "{[a/x', b/x]}"] \\
P(a,a) \lor \neg Q(f(a,b)) \arrow[d] & \neg P(a,a) \arrow[dl] & \neg P(a,b) \arrow[d] & P(a,b) \lor Q(f(a,b)) \arrow[dl] \\
\neg Q(f(a,b)) \arrow[d] & & Q(f(a,b)) \arrow[d] & \\
{} \arrow[dr] & & {} \arrow[dl] & \\
 & \bot & &
\end{tikzcd}$$

## General Resolution through Instantiation

Problems:

- More than one instance of a clause can participate in a proof.
- Even worse: There are infinitely many possible instances.

Observation:

- Instantiation must produce complementary literals (so that inferences become possible).

Idea:

- Do not instantiate more than necessary to get complementary literals.

## General Resolution through Instantiation

Idea: do not instantiate more than necessary:

$$\begin{tikzcd}
P(z',z') \lor \neg Q(z) \arrow[d, "{[a/z']}"'] & & & \neg P(a,y) \arrow[dl, "{[a/y]}"'] \arrow[dr, "{[b/y]}"] & & & P(x',b) \lor Q(f(x',x)) \arrow[d, "{[a/x']}"] \\
P(a,a) \lor \neg Q(z) \arrow[dr] & & \neg P(a,a) \arrow[dl] & & \neg P(a,b) \arrow[dr] & & P(a,b) \lor Q(f(a,x)) \arrow[dl] \\
& \neg Q(z) \arrow[d, "{[f(a,x)/z]}"'] & & & & Q(f(a,x)) \arrow[d] & \\
& \neg Q(f(a,x)) \arrow[drr] & & & & Q(f(a,x)) \arrow[dll] & \\
& & & \bot & & &
\end{tikzcd}$$

## Lifting Principle

Problem: Make saturation of infinite sets of clauses as they arise from taking the (ground) instances of finitely many **general** clauses (with variables) effective and efficient.

Idea (Robinson 1965):

- Resolution for general clauses:
- **Equality** of ground atoms is generalized to **unifiability** of general atoms;
- Only compute **most general** (minimal) unifiers.

## General Resolution through Instantiation

Significance: The advantage of the method in (Robinson 1965) compared with (Gilmore 1960) is that unification enumerates only those instances of clauses that participate in an inference. Moreover, clauses are not right away instantiated into ground clauses. Rather they are instantiated only as far as required for an inference. Inferences with non-ground clauses in general represent infinite sets of ground inferences which are computed simultaneously in a single step.

## Substitutions and Unifiers

- A **substitution** $\sigma$ is a mapping from variables to terms which is the identity almost everywhere

  Example: $\sigma = [y \mapsto f(x),\ z \mapsto f(x)]$

## Substitutions and Unifiers

- A **substitution** $\sigma$ is a mapping from variables to terms which is the identity almost everywhere

  Example: $\sigma = [y \mapsto f(x),\ z \mapsto f(x)]$

- A substitution can be **applied** to a term or atom $t$, written as $t\sigma$

  Example, where $\sigma$ is from above: $P(f(x), y)\sigma = P(f(x), f(x))$

## Substitutions and Unifiers

- A **substitution** $\sigma$ is a mapping from variables to terms which is the identity almost everywhere

  Example: $\sigma = [y \mapsto f(x),\ z \mapsto f(x)]$

- A substitution can be **applied** to a term or atom $t$, written as $t\sigma$

  Example, where $\sigma$ is from above: $P(f(x), y)\sigma = P(f(x), f(x))$

- A substitution $\gamma$ is a **unifier** of $s$ and $t$ iff $s\gamma = t\gamma$

  Example: $\gamma = [x \mapsto a, y \mapsto f(a),\ z \mapsto f(a)]$ is a unifier of $P(f(x), y)$ and $P(z, z)$

## Substitutions and Unifiers

- A **substitution** $\sigma$ is a mapping from variables to terms which is the identity almost everywhere

  Example: $\sigma = [y \mapsto f(x),\ z \mapsto f(x)]$

- A substitution can be **applied** to a term or atom $t$, written as $t\sigma$

  Example, where $\sigma$ is from above: $P(f(x), y)\sigma = P(f(x), f(x))$

- A substitution $\gamma$ is a **unifier** of $s$ and $t$ iff $s\gamma = t\gamma$

  Example: $\gamma = [x \mapsto a, y \mapsto f(a),\ z \mapsto f(a)]$ is a unifier of $P(f(x), y)$ and $P(z, z)$

- A unifier $\sigma$ of $s$ is **most general** iff for every unifier $\gamma$ of $s$ and $t$ there is a substitution $\delta$ such that $\gamma = \sigma \circ \delta$; notation: $\sigma = \mathrm{mgu}(s, t)$

  Example: $\sigma = [y \mapsto f(x),\ z \mapsto f(x)] = \mathrm{mgu}(P(f(x), y), P(z, z))$

  There are (linear) algorithms to compute mgu's or return "fail"

## Substitutions and Unifiers

Let $E = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ ($s_i, t_i$ terms or atoms) a multi-set of **equality problems.** A substitution $\sigma$ is called a **unifier** of $E$ if $s_i\sigma = t_i\sigma$ for all $1 \leq i \leq n$.

If a unifier of $E$ exists, then $E$ is called **unifiable.**
The rule system on the next slide computes a most general unifer of a multiset of equality problems or "fail" ($\perp$) if none exists.

## Rule Based Naive Standard Unification

$$
\begin{array}{rll}
t \doteq t, E & \Rightarrow_{SU} & E \\
f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n), E & \Rightarrow_{SU} & s_1 \doteq t_1, \ldots, s_n \doteq t_n, E \\
f(\ldots) \doteq g(\ldots), E & \Rightarrow_{SU} & \bot \\
x \doteq t, E & \Rightarrow_{SU} & x \doteq t, E[t/x] \\
 & \multicolumn{2}{l}{\text{if } x \in var(E), x \notin var(t)} \\
x \doteq t, E & \Rightarrow_{SU} & \bot \\
 & \multicolumn{2}{l}{\text{if } x \neq t, x \in var(t)} \\
t \doteq x, E & \Rightarrow_{SU} & x \doteq t, E \\
 & \multicolumn{2}{l}{\text{if } t \notin X}
\end{array}
$$

## Main Properties

The above unification algorithm is sound and complete:
Given $E = s_1 \doteq t_1, \dots, s_n \doteq t_n$, exhaustive application of the above rules always terminates, and one of the following holds:

- The result is a set equations in **solved form**, that is, is of the form

  $$x_1 \doteq u_1, \dots, x_k \doteq u_k$$

  with $x_i$ pairwise distinct variables, and $x_i \notin var(u_j)$.
  In this case, the solved form represents the substitution $\sigma_E = [u_1/x_1, \dots, u_k/x_k]$ and it is an mgu for $E$.
- The result is $\perp$. In this case no unifier for $E$ exists.

## Resolution for First-Order Clauses

$$\frac{C \vee A \qquad D \vee \neg B}{(C \vee D)\sigma} \quad \text{if } \sigma = \text{mgu}(A, B) \qquad \text{[resolution]}$$

$$\frac{C \vee A \vee B}{(C \vee A)\sigma} \quad \text{if } \sigma = \text{mgu}(A, B) \quad \text{[factorization]}$$

In both cases, $A$ and $B$ have to be renamed apart (made variable disjoint).

## Resolution for First-Order Clauses

$$\frac{C \vee A \qquad D \vee \neg B}{(C \vee D)\sigma} \quad \text{if } \sigma = \mathrm{mgu}(A, B) \qquad \text{[resolution]}$$

$$\frac{C \vee A \vee B}{(C \vee A)\sigma} \quad \text{if } \sigma = \mathrm{mgu}(A, B) \quad \text{[factorization]}$$

In both cases, $A$ and $B$ have to be renamed apart (made variable disjoint).

### Example

$$\frac{Q(z) \vee P(z, z) \quad \neg P(x, y)}{Q(x)} \quad \text{where } \sigma = [z \mapsto x, y \mapsto x] \qquad \text{[resolution]}$$

$$\frac{Q(z) \vee P(z, a) \vee P(a, y)}{Q(a) \vee P(a, a)} \quad \text{where } \sigma = [z \mapsto a, y \mapsto a] \quad \text{[factorization]}$$

## Completeness of First-Order Resolution

**Theorem:** Resolution is **refutationally complete**

## Completeness of First-Order Resolution

**Theorem:** Resolution is **refutationally complete**

- That is, if a clause set is unsatisfiable, then Resolution will derive the empty clause $\perp$ eventually

## Completeness of First-Order Resolution

**Theorem:** Resolution is **refutationally complete**

- That is, if a clause set is unsatisfiable, then Resolution will derive the empty clause $\bot$ eventually
- More precisely: If a clause set is unsatisfiable and closed under the application of the Resolution and Factorization inference rules, then it contains the empty clause $\bot$

## Completeness of First-Order Resolution

**Theorem:** Resolution is **refutationally complete**

- That is, if a clause set is unsatisfiable, then Resolution will derive the empty clause $\perp$ eventually
- More precisely: If a clause set is unsatisfiable and closed under the application of the Resolution and Factorization inference rules, then it contains the empty clause $\perp$
- Perhaps easiest proof: Herbrand Theorem + Completeness of propositional resolution + **Lifting Lemma**

## Lifting Lemma

**Lemma 0.1** *Let $C$ and $D$ be variable-disjoint clauses. If*

$$\begin{tikzcd} D \arrow[d, "\sigma"] & C \arrow[d, "\rho"] \\ D\sigma & C\rho \end{tikzcd}$$

$$\frac{D\sigma \qquad C\rho}{C'} \qquad \textit{[propositional resolution]}$$

*then there exists a substitution $\tau$ such that*

$$\frac{D \qquad C}{C''} \qquad \textit{[general resolution]}$$

$$\begin{tikzcd} C'' \arrow[d, "\tau"] \\ C' = C''\tau \end{tikzcd}$$

## Lifting Lemma

An analogous lifting lemma holds for factorization.

**Corollary:** if $N$ is a set of clauses closed under resolution and factorization, then also the set of all ground instances of all clauses from $N$ is closed under resolution and factorization.

With this result, it only remains to be shown how a given set of clauses can be closed under resolution and factorization. For this use, e.g., the “Given Clause Loop”.

## The “Given Clause Loop”

As used in the Otter theorem prover:

Lists of clauses maintained by the algorithm: `usable` and `sos`.

Initialize `sos` with the input clauses, `usable` empty.

**Algorithm** (straight from the Otter manual):

```
While (sos is not empty and no refutation has been found)
  1. Let given_clause be the ‘lightest’ clause in sos;
  2. Move given_clause from sos to usable;
  3. Infer and process new clauses using the inference rules in
     effect; each new clause must have the given_clause as
     one of its parents and members of usable as its other
     parents;  new clauses that pass the retention tests
     are appended to sos;
End of while loop.
```

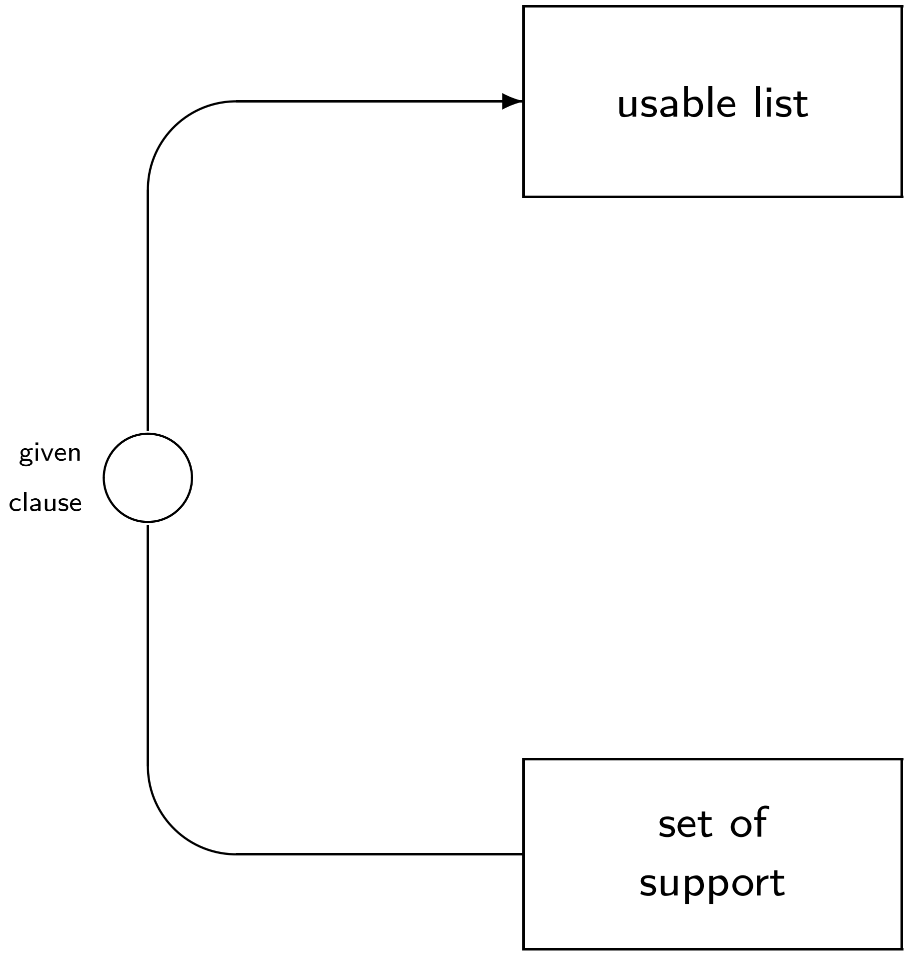
**Fairness:** define clause weight e.g. as “depth + length” of clause.

## The “Given Clause Loop” - Graphically

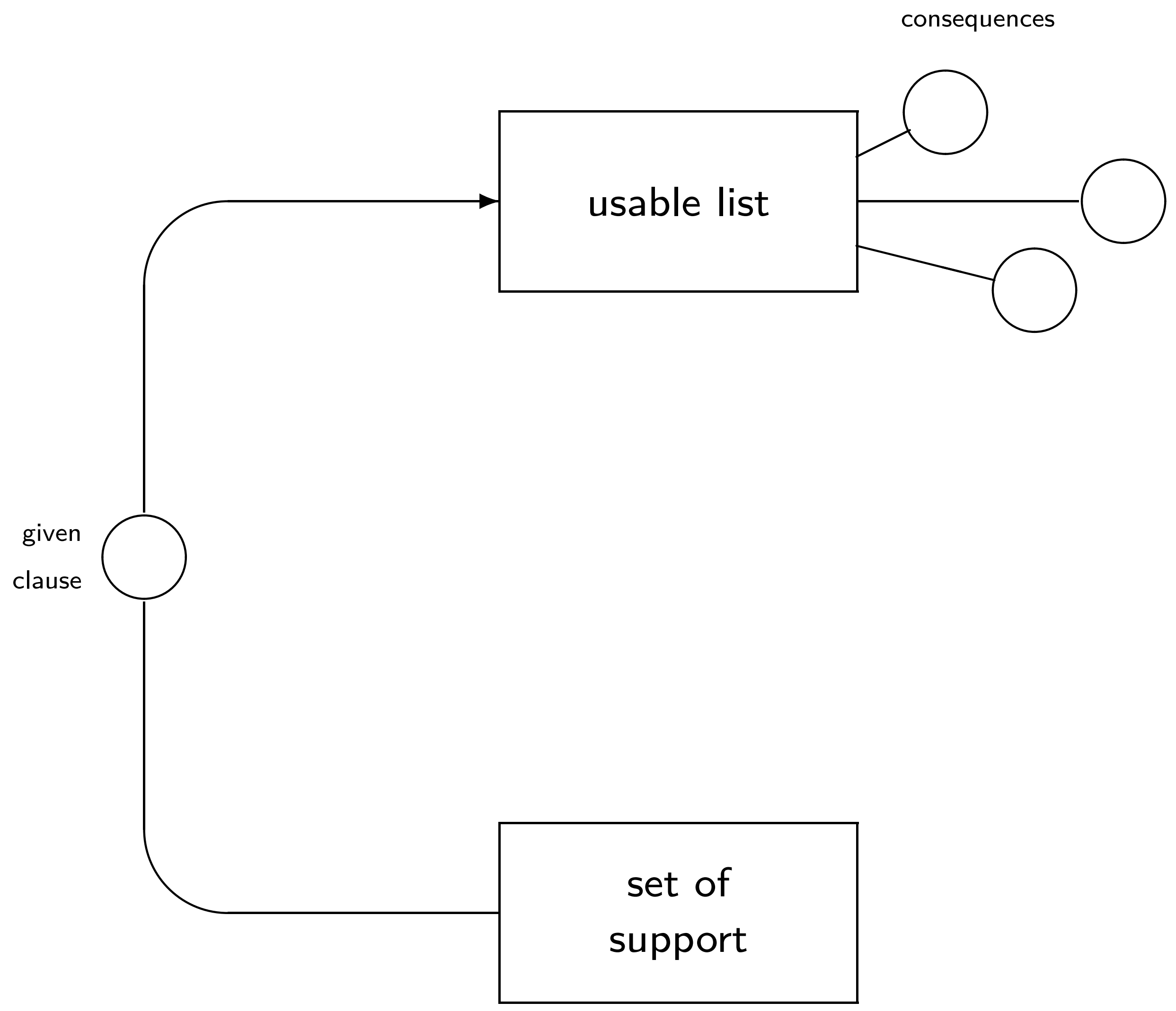
## The “Given Clause Loop” - Graphically

usable list
set of support

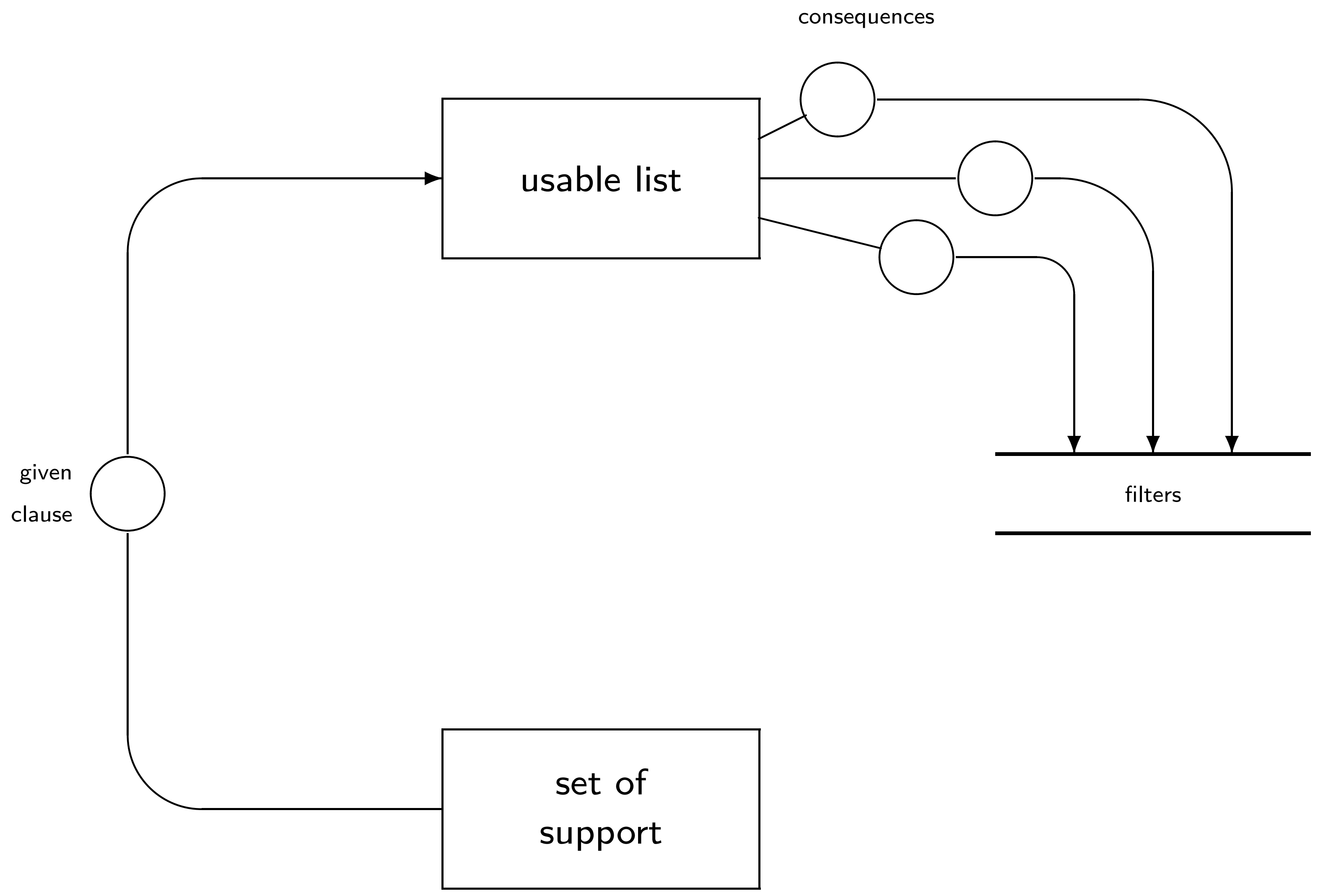
## The “Given Clause Loop” - Graphically



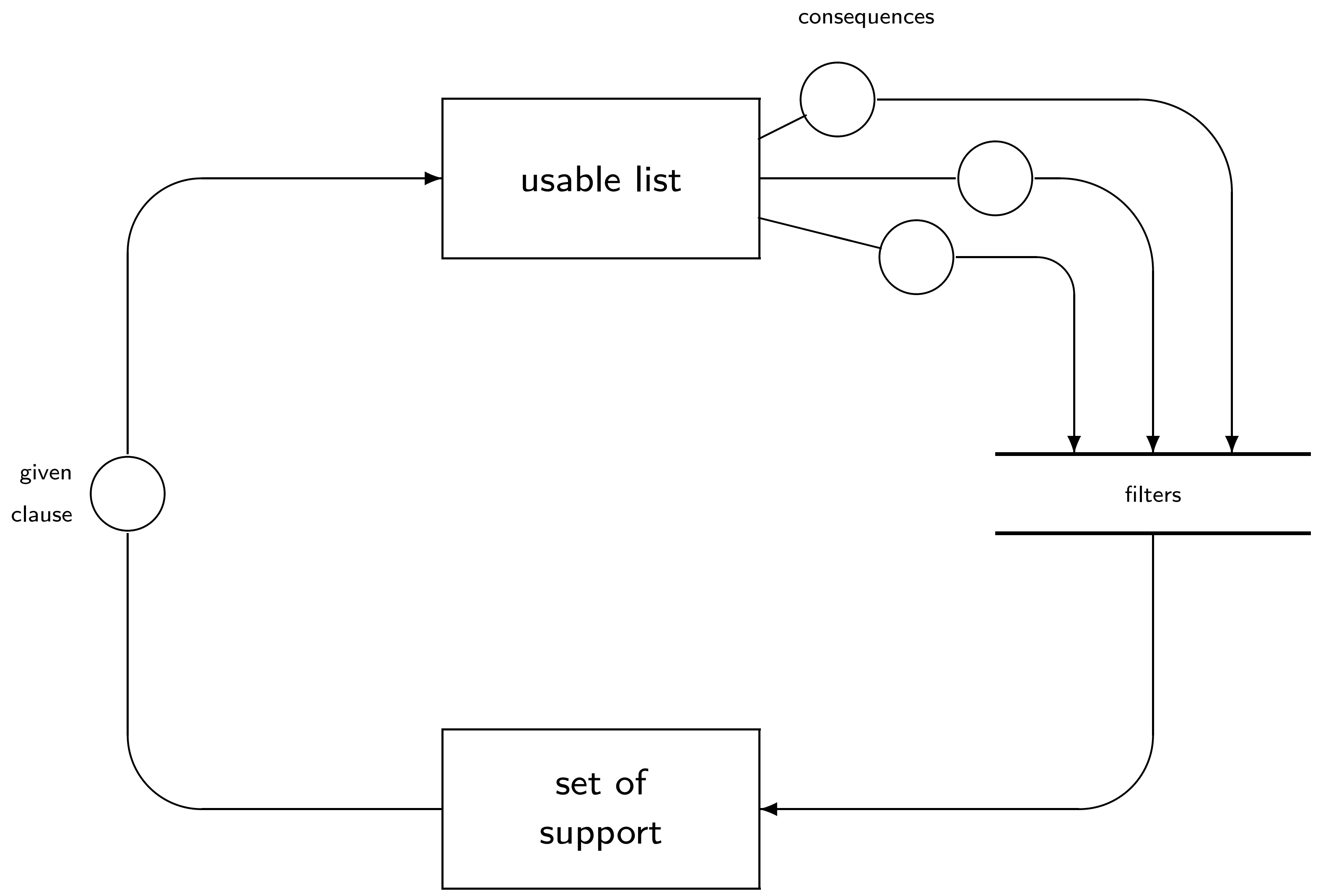
## The “Given Clause Loop” - Graphically



## The “Given Clause Loop” - Graphically



## The “Given Clause Loop” - Graphically



## Resolution – Further Topics

### Overcoming the search space

- Restricting inference rules, in particular by ordering refinements.

  A-ordered resolution permits resolution inferences only if the literals resolved upon are maximal in their parent clauses.

- Resolution strategies, to compute (hopefully small) subsets of the full closure under inference rule applications.

  Set-of-support, Linear Resolution, Hyperresolution (see below), and more.

- Deleting clauses that are not needed to find a refutation.
  In particular subsumption deletion: delete clause $C$ in presence of a (different) clause $D$ such that $D\sigma \subseteq C$, for some substitution $\sigma$.

- Simplification of clauses.

Implementation techniques: in particular term indexing techniques

## Hyperresolution

There are **many** variants of resolution. (We refer to [Bachmair, Ganzinger: Resolution Theorem Proving] for further reading.)

One well-known example is hyperresolution (Robinson 1965):

$$\frac{D_1 \vee B_1 \quad \ldots \quad D_n \vee B_n \qquad C \vee \neg A_1 \vee \ldots \vee \neg A_n}{(D_1 \vee \ldots \vee D_n \vee C)\sigma}$$

with $\sigma = \mathrm{mgu}(A_1 \doteq B_1, \ldots, A_n \doteq B_n)$.

Similarly to resolution, hyperresolution has to be complemented by a factoring inference.

## Contents

## Instance-Based Methods

Recall:

- Gilmore's method reduces proof search in first-order logic to propositional logic unsatisfiability problems

## Instance-Based Methods

Recall:

- Gilmore's method reduces proof search in first-order logic to propositional logic unsatisfiability problems
- Main problem is the unguided generation of (very many) ground clauses

## Instance-Based Methods

Recall:

- Gilmore's method reduces proof search in first-order logic to propositional logic unsatisfiability problems
- Main problem is the unguided generation of (very many) ground clauses
- All modern calculi address this problem in one way or another, e.g.
  - **Avoidance:** Resolution calculi do not need to generate the ground instances at all

    Resolution inferences operate directly on clauses, not on their ground instances
  - **Guidance:** Instance-Based Methods are similar to Gilmore's method but generate ground instances in a guided way

## Two-Level Instance-Based Methods

Idea:

- Overlaps of complementary literals produce instantiations (as in resolution);
- However, contrary to resolution, clauses are not recombined.
- Clauses are temporarily grounded – replace every variable by a constant – and checked for unsatisfiability; use an efficient propositional proof method, a “SAT-solver” for that.
- Main variants: (ordered) semantic hyperlinking [Plaisted], resolution-based instance generation (Inst-Gen) [Ganzinger and Korovin].

## Resolution-Based Instance Generation

Resolution-based instance generation has only one inference rule:

$$\frac{D \vee B \qquad\qquad C \vee \neg A}{(D \vee B)\sigma \qquad (C \vee \neg A)\sigma} \qquad [\textbf{Inst-Gen}]$$

if $\sigma = \text{mgu}(A, B)$ and at least one conclusion is a proper instance of its premise.

The instance-generation calculus saturates a given clause set under Inst-Gen and periodically passes the ground-instantiated version of the current clause set to a SAT-solver.

A refutation has been found if the SAT-solver determines unsatisfiability.

## One-Level Instance-Based Methods

- Other methods **do not** use a SAT-solver as a subroutine;
- Instead, the **same** base calculus is used to generate new clause instances and test for unsatisfiability of grounded data structures.
- Main variants: tableau variants, such as the disconnection calculus [Billon; Letz and Stenz], and a variant of the DPLL procedure for first-order logic, FDPLL [Baumgartner and Tinelli].

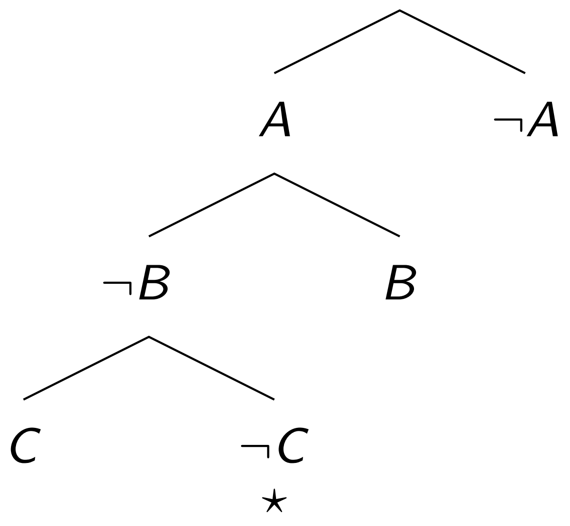
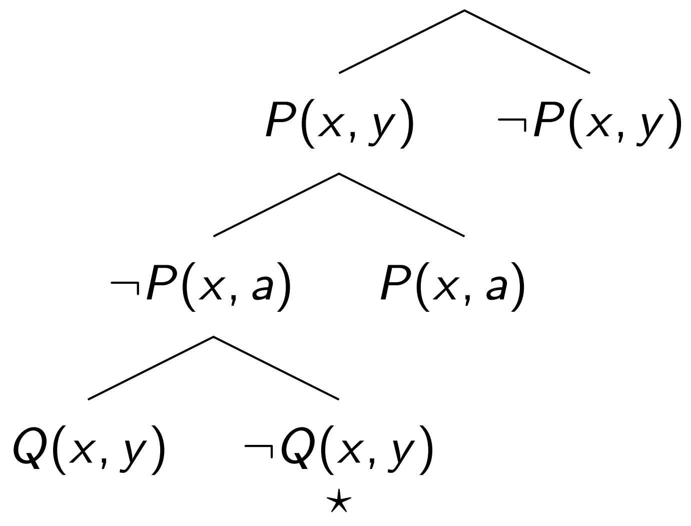
## Instance-Based Method – FDPLL

### Lifted data structures:

| | **Propositional Reasoning** | **First-Order Reasoning** |
|---|---|---|
| **Clauses** | $\neg A \lor B \lor C$ | $\neg P(x,x) \lor P(x,a) \lor Q(x,x)$ |

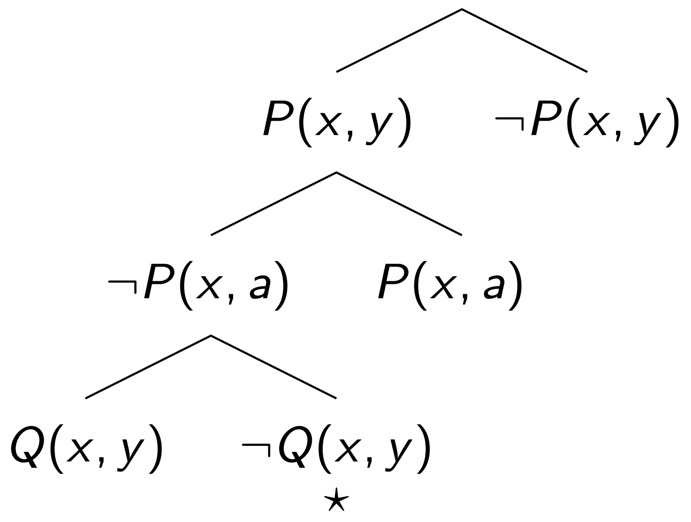
## Instance-Based Method – FDPLL

**Lifted data structures:**

| | **Propositional Reasoning** | **First-Order Reasoning** |
|---|---|---|
| **Clauses** | $\neg A \vee B \vee C$ | $\neg P(x,x) \vee P(x,a) \vee Q(x,x)$ |
| **Trees** |  |  |

**First-Order Semantic Trees**

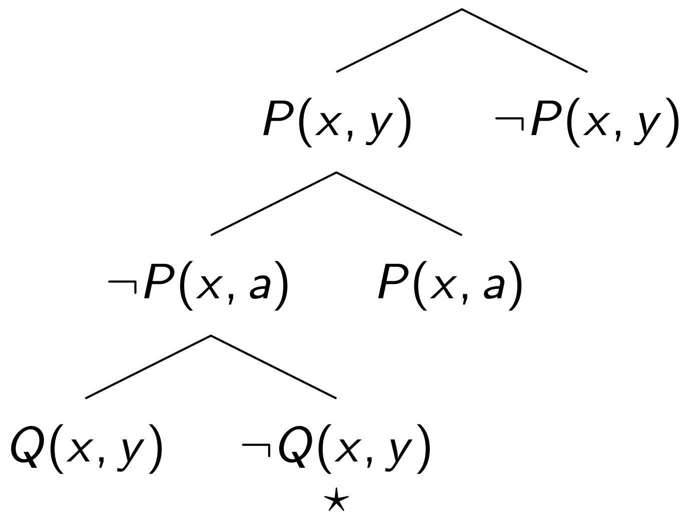
## First-Order Semantic Trees



Issues:

- One-branch-at-a-time approach desired

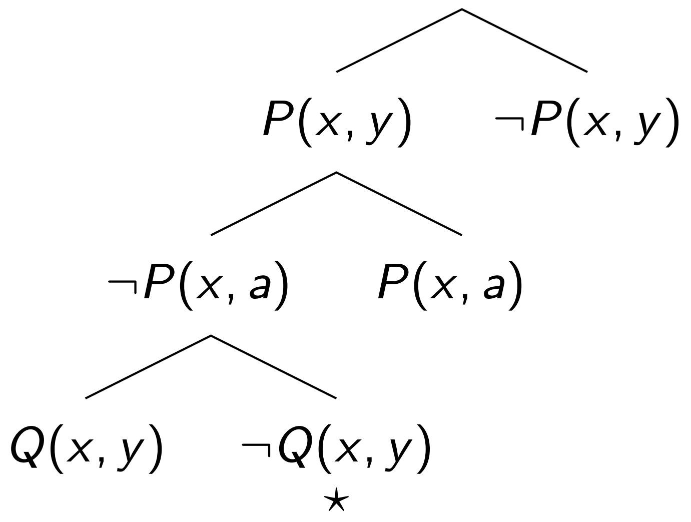
## First-Order Semantic Trees



Issues:

- One-branch-at-a-time approach desired
- How to extract an interpretation from a branch?

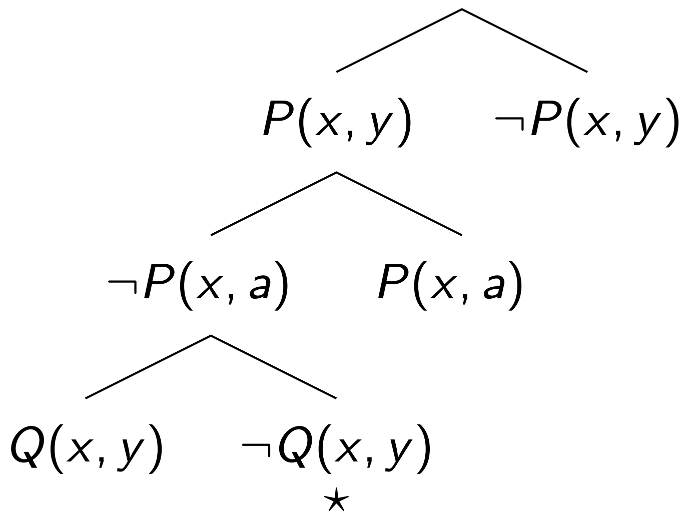
## First-Order Semantic Trees



Issues:

- One-branch-at-a-time approach desired
- How to extract an interpretation from a branch?
- When is a branch closed?

## First-Order Semantic Trees



Issues:

- One-branch-at-a-time approach desired
- How to extract an interpretation from a branch?
- When is a branch closed?
- How to construct such trees (calculus)?

## Extracting an Interpretation from a Branch

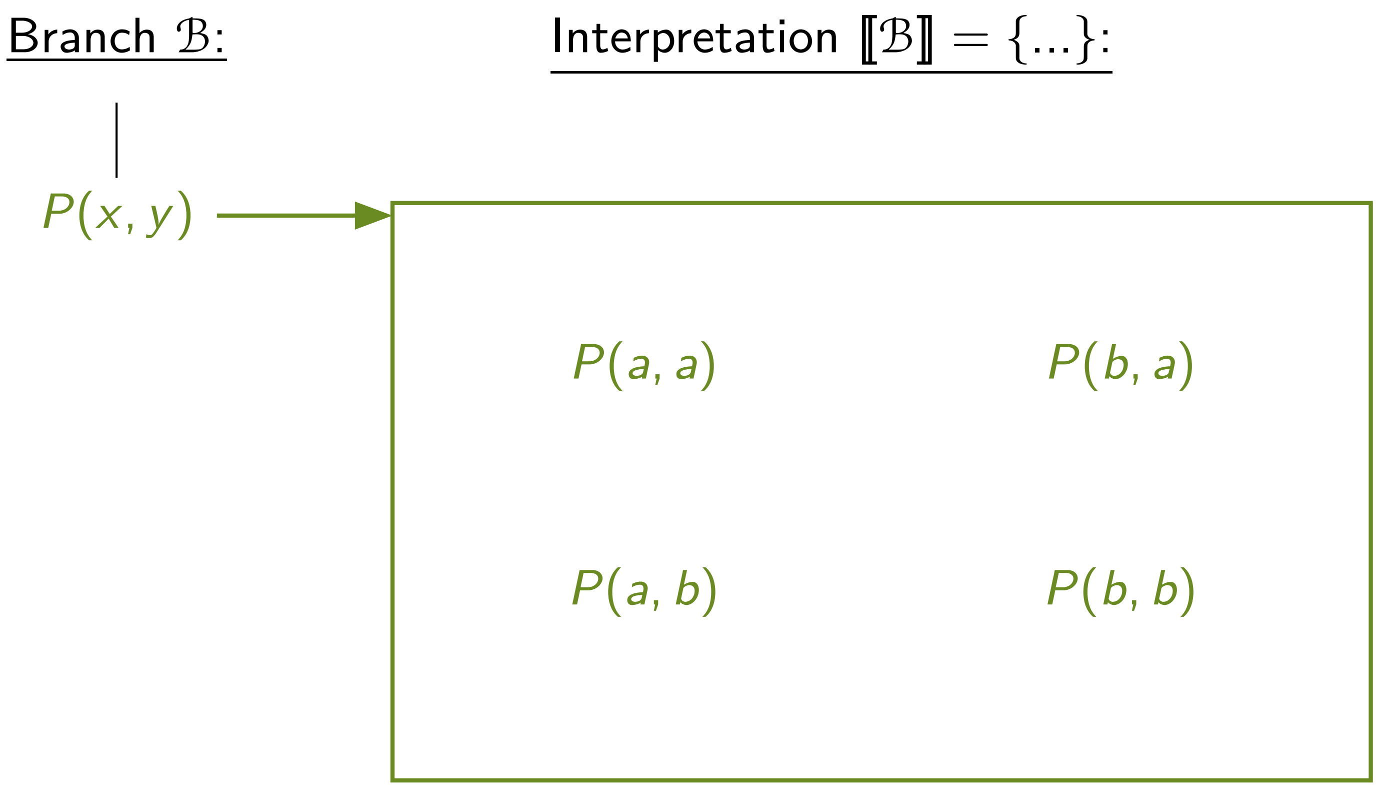
<u>Branch $\mathcal{B}$:</u>

$P(x, y)$

<u>Interpretation $[\![\mathcal{B}]\!] = \{...\}$:</u>

- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

## Extracting an Interpretation from a Branch



- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

## Extracting an Interpretation from a Branch

Branch $\mathcal{B}$:

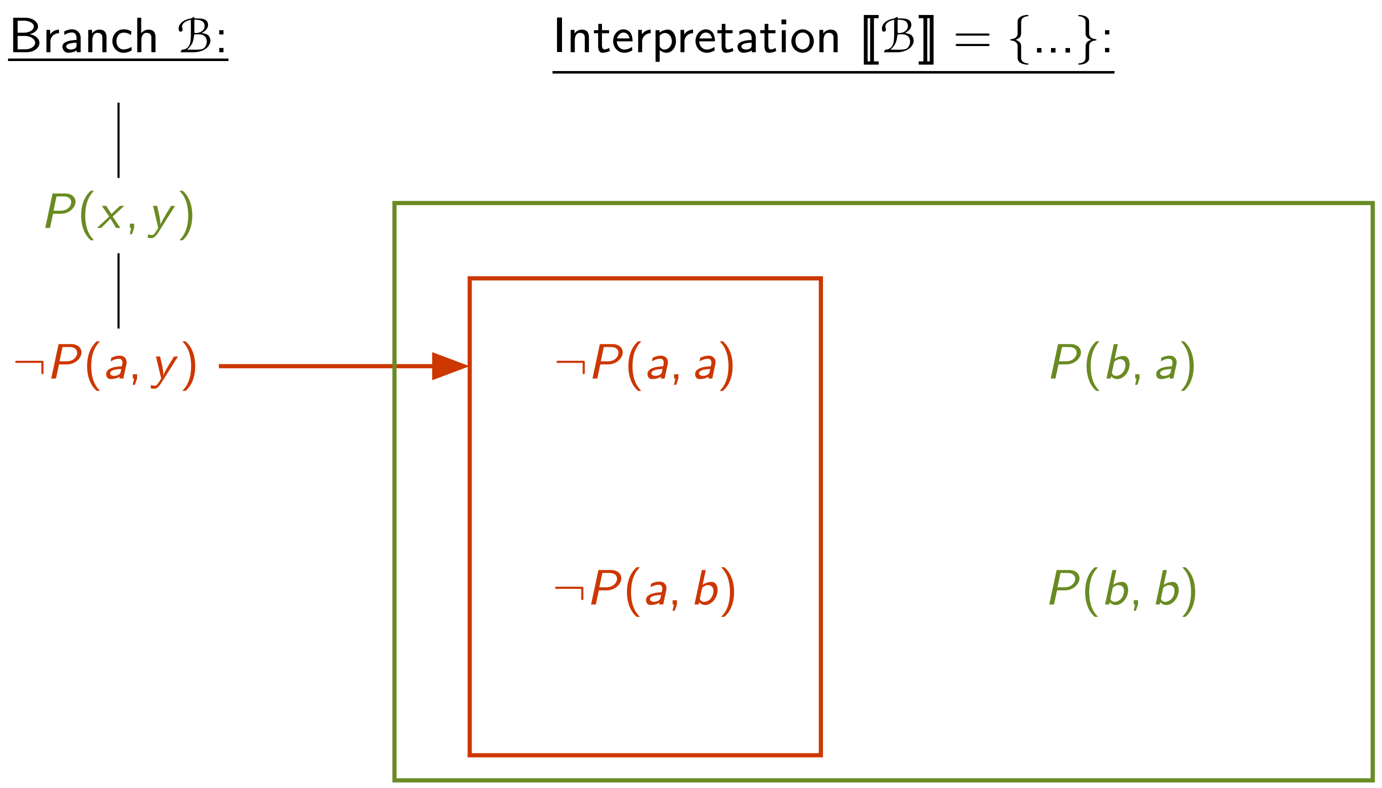
$$\begin{array}{c} | \\ P(x, y) \\ | \\ \neg P(a, y) \end{array}$$

Interpretation $[\![\mathcal{B}]\!] = \{\ldots\}$:

$$\boxed{\begin{array}{cc} P(a, a) & P(b, a) \\ P(a, b) & P(b, b) \end{array}}$$

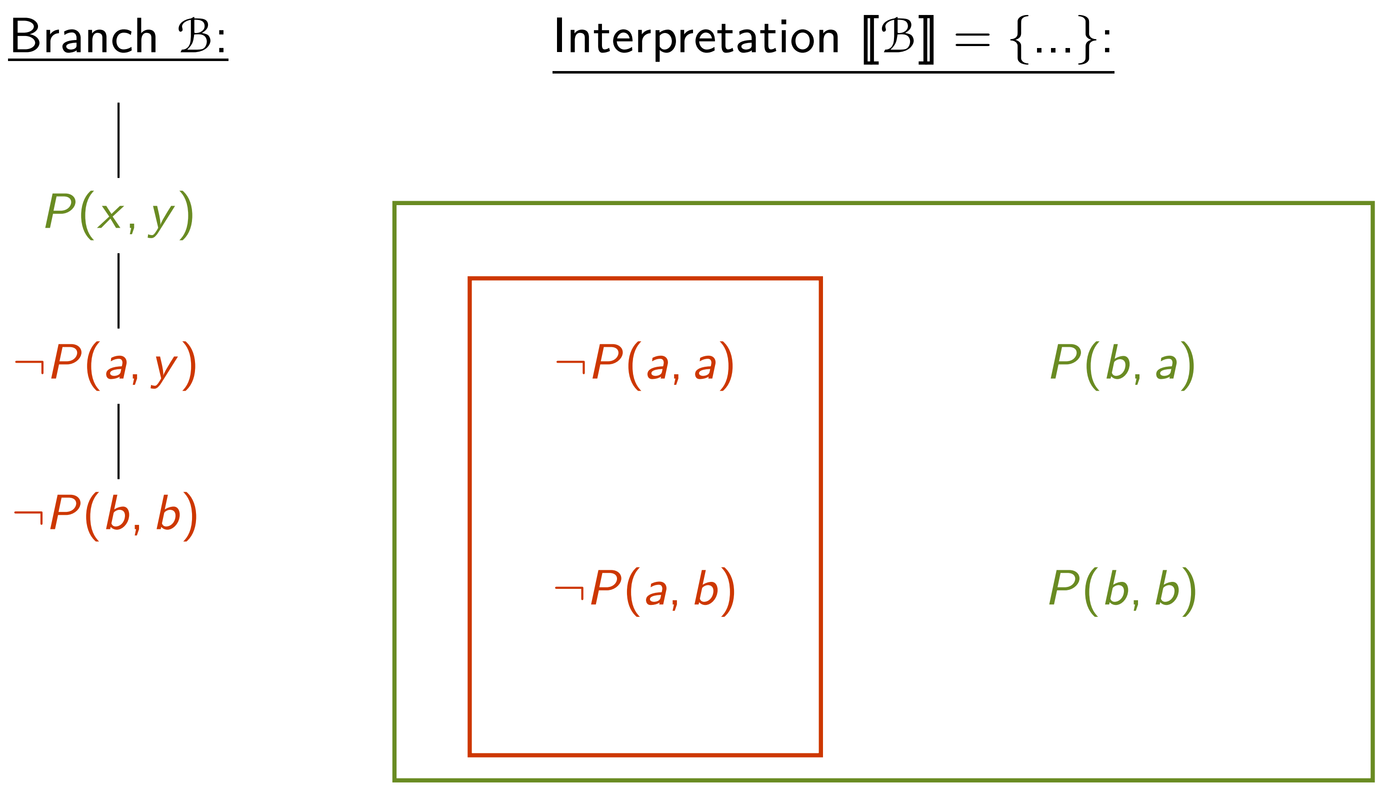
- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

## Extracting an Interpretation from a Branch



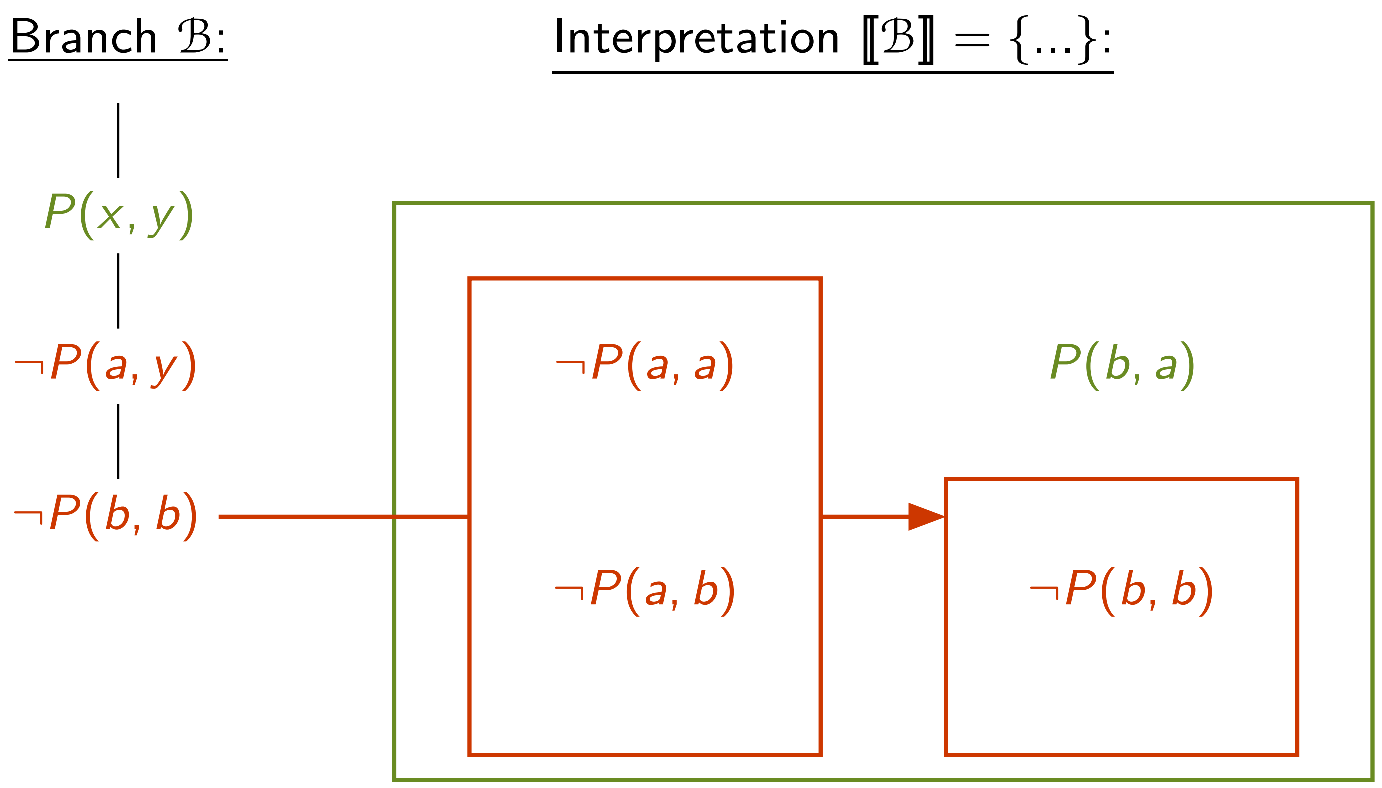
- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

## Extracting an Interpretation from a Branch



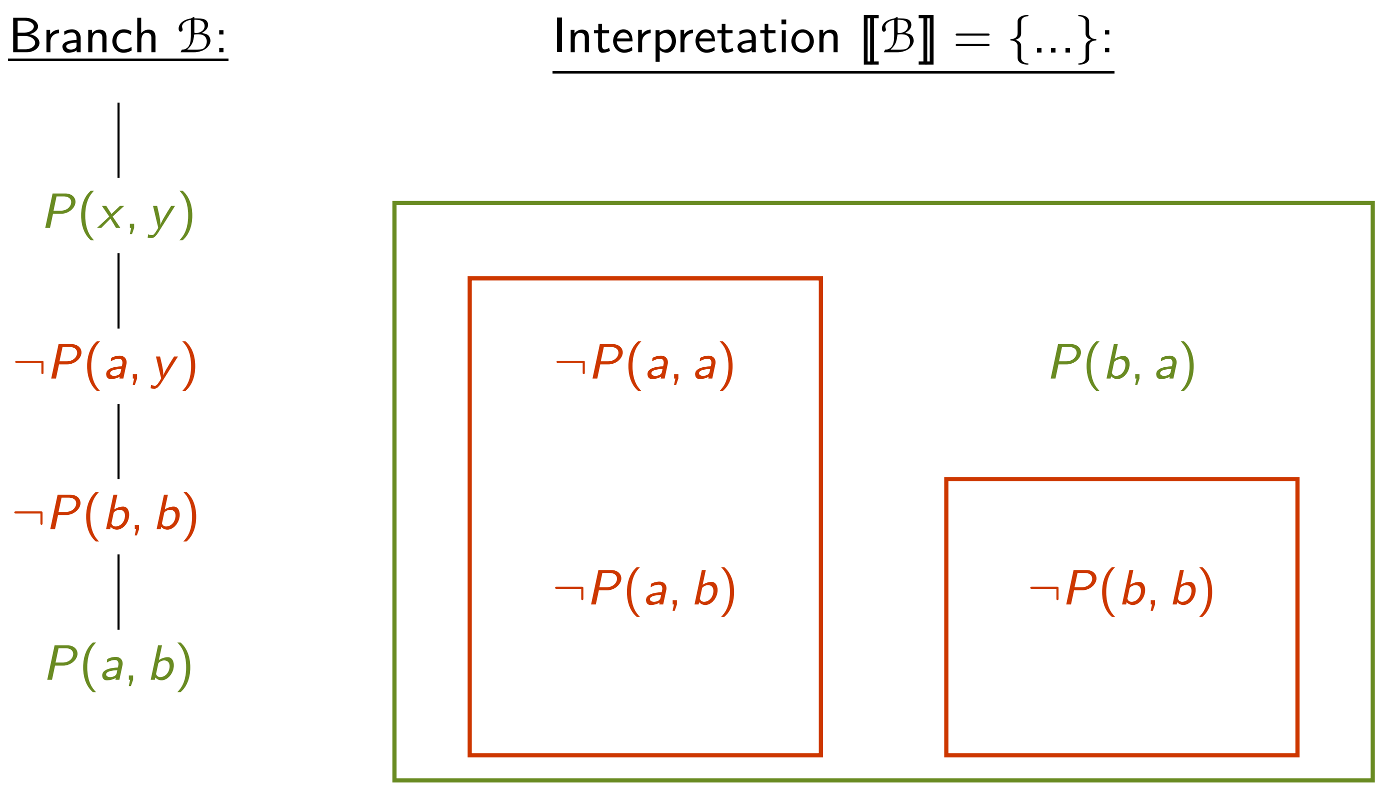
- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

## Extracting an Interpretation from a Branch



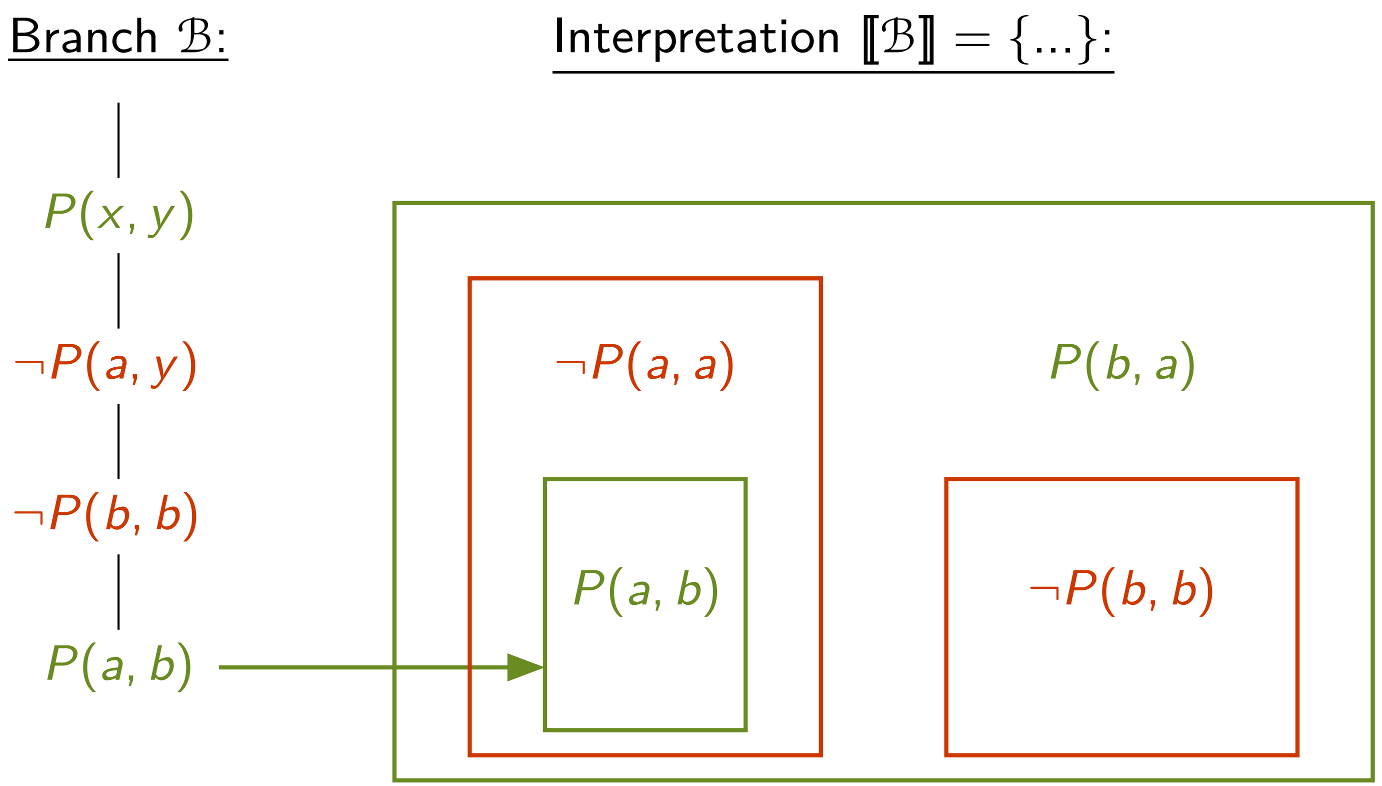
- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

## Extracting an Interpretation from a Branch



- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

## Extracting an Interpretation from a Branch



- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

## Extracting an Interpretation from a Branch

Branch $\mathcal{B}$:

$$\begin{tikzcd}
{} \arrow[d, no head] \\
P(x, y) \arrow[d, no head] \\
\neg P(a, y) \arrow[d, no head] \\
\neg P(b, b) \arrow[d, no head] \\
P(a, b)
\end{tikzcd}$$

Interpretation $[\![\mathcal{B}]\!] = \{\ldots\}$:

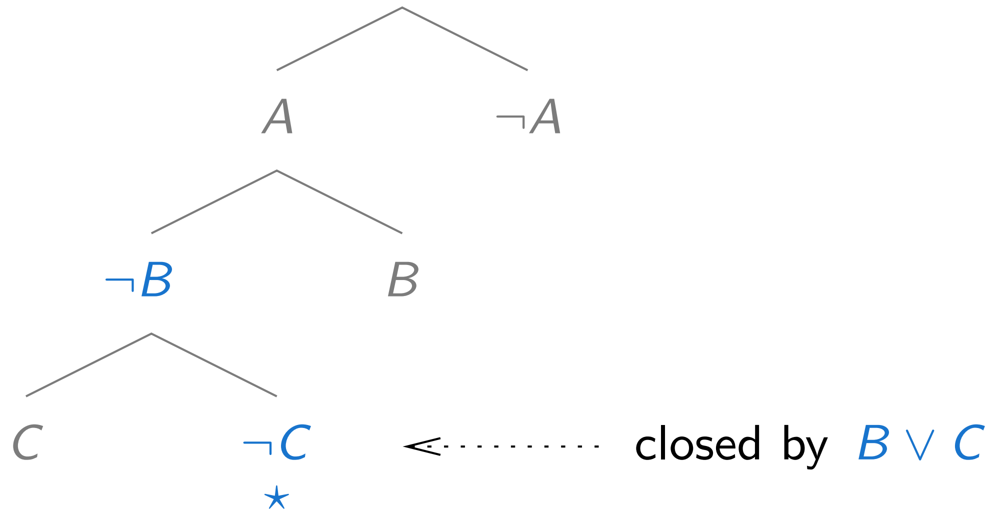
$$\{\quad \neg P(a, a) \quad , \quad P(b, a) \quad , \quad P(a, b) \quad , \quad \neg P(b, b) \quad \}$$

- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.
- The order of literals does not matter.

## Calculus: Branch Closure

**Purpose:** Determine if branch elementary contradicts an input clause.

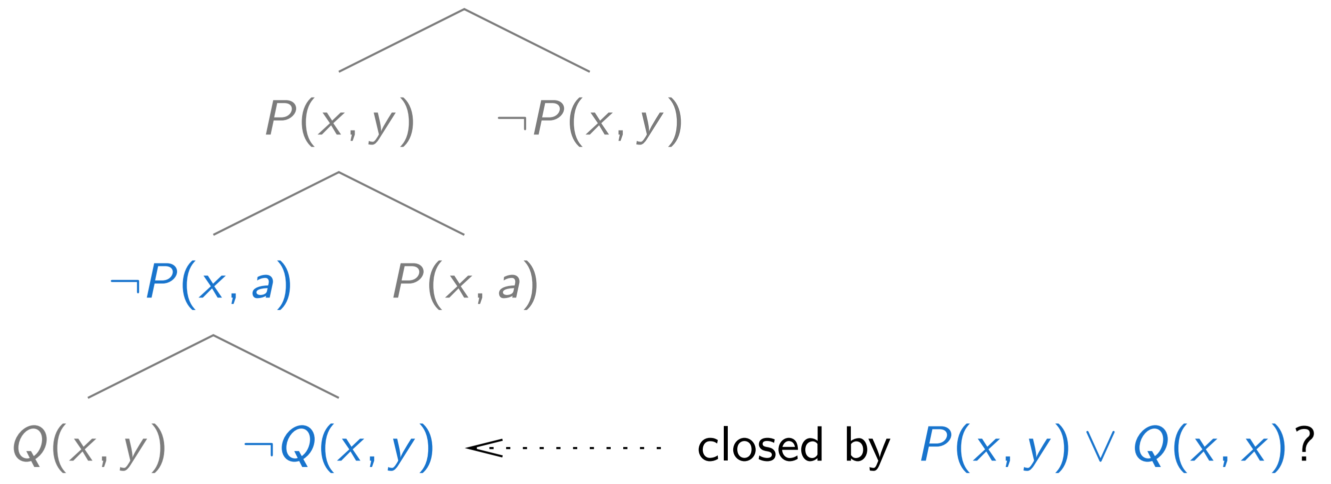
Propositional case:



## Calculus: Branch Closure

**Purpose:** Determine if branch elementary contradicts an input clause.

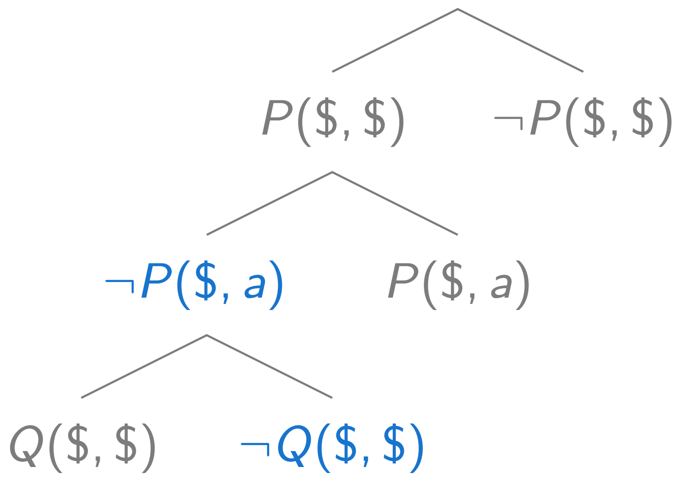
First-Order case:



## Calculus: Branch Closure

**Purpose:** Determine if branch elementary contradicts an input clause.

First-Order case:



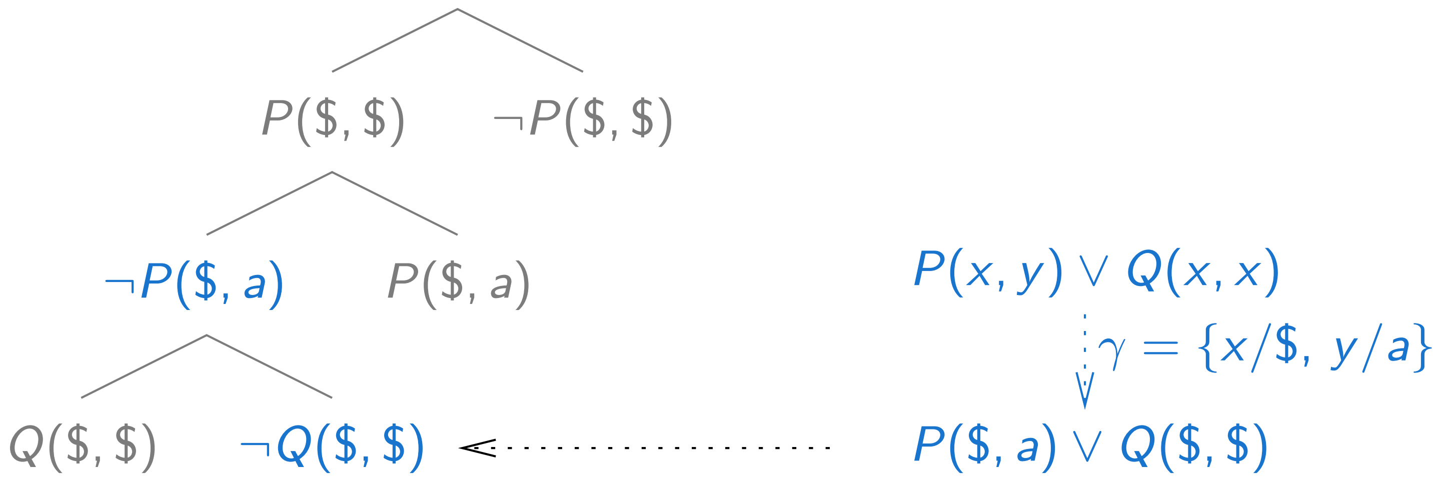
$P(x,y) \vee Q(x,x)$

1. Replace **all** variables in tree by a constant \$. Gives propositional tree
2.
3.

## Calculus: Branch Closure

**Purpose:** Determine if branch elementary contradicts an input clause.

First-Order case:

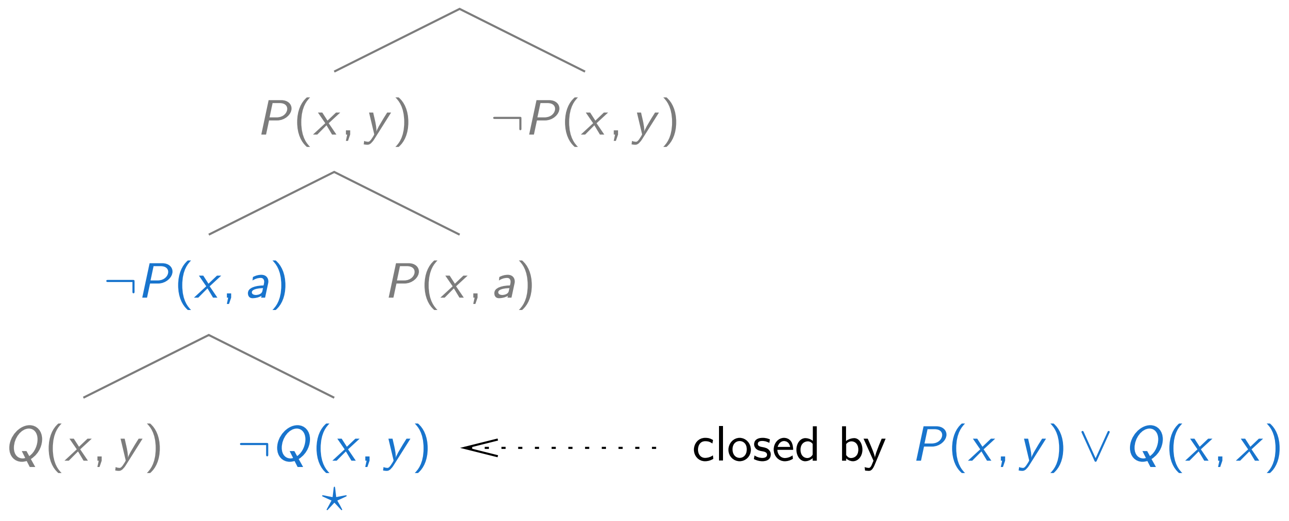


1. Replace all variables in tree by a constant \$. Gives propositional tree
2. Compute matcher $\gamma$ to propositionally close branch
3.

## Calculus: Branch Closure

**Purpose:** Determine if branch elementary contradicts an input clause.

First-Order case:

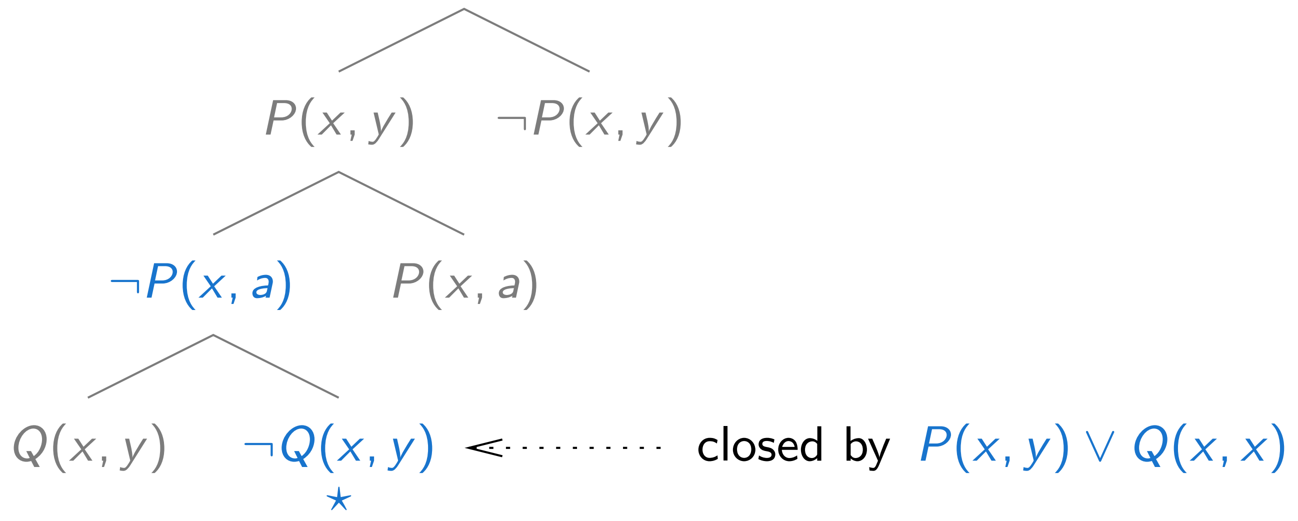


1. Replace all variables in tree by a constant \$. Gives propositional tree
2. Compute matcher $\gamma$ to propositionally close branch
3. Mark branch as closed ($\star$)

## Calculus: Branch Closure

**Purpose:** Determine if branch elementary contradicts an input clause.

First-Order case:



1. Replace all variables in tree by a constant \$. Gives propositional tree
2. Compute matcher $\gamma$ to propositionally close branch
3. Mark branch as closed ($\star$)

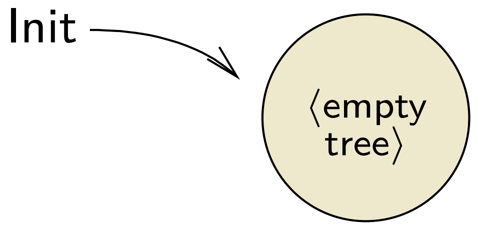
**Theorem:** FDPLL is sound (because propositional DPLL is sound), and splitting can be done with **arbitrary** literal.

## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** “unsatisfiable” or “satisfiable” (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:

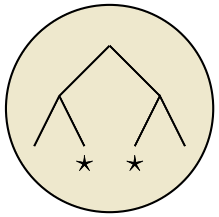


## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** "unsatisfiable" or "satisfiable" (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:

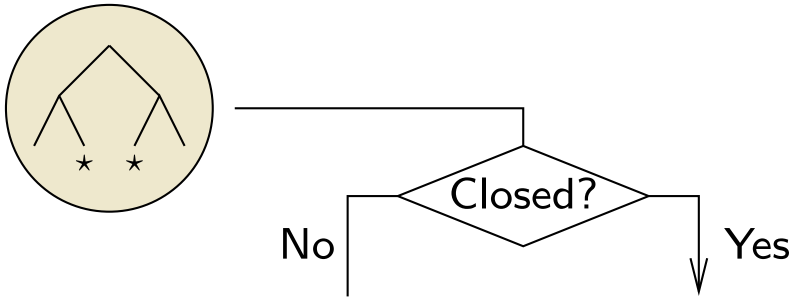


## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** "unsatisfiable" or "satisfiable" (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:

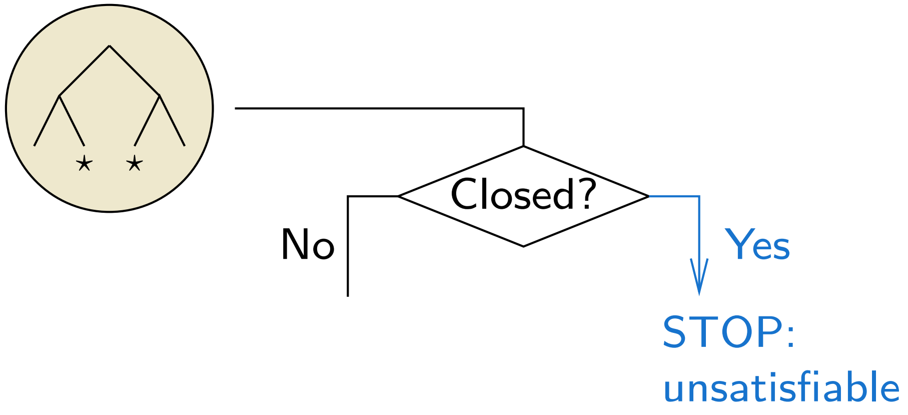


## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** “unsatisfiable” or “satisfiable” (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:

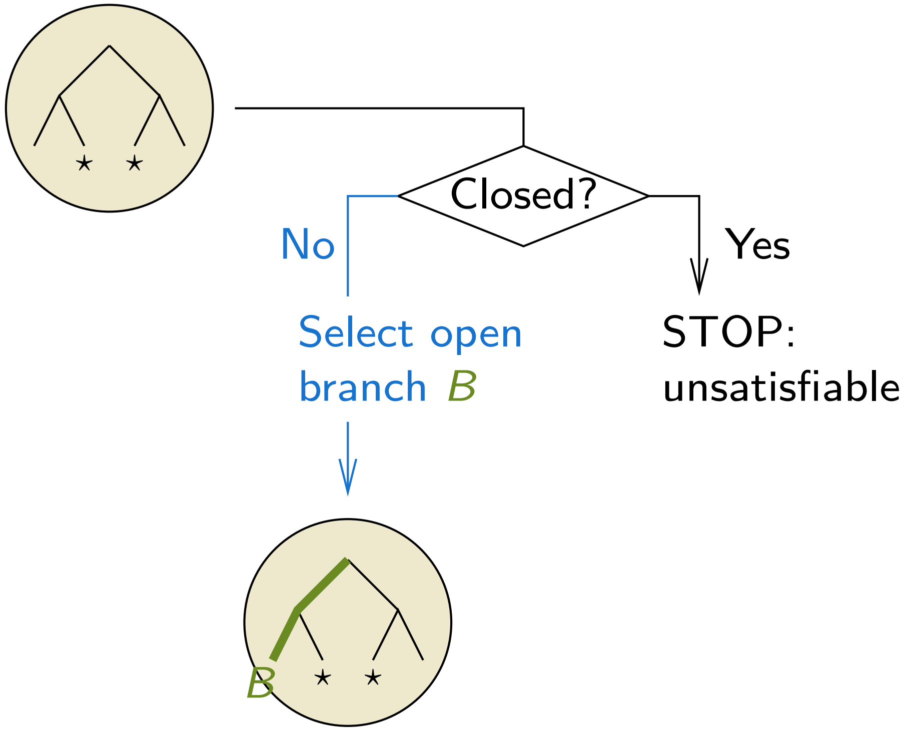


## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** "unsatisfiable" or "satisfiable" (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:

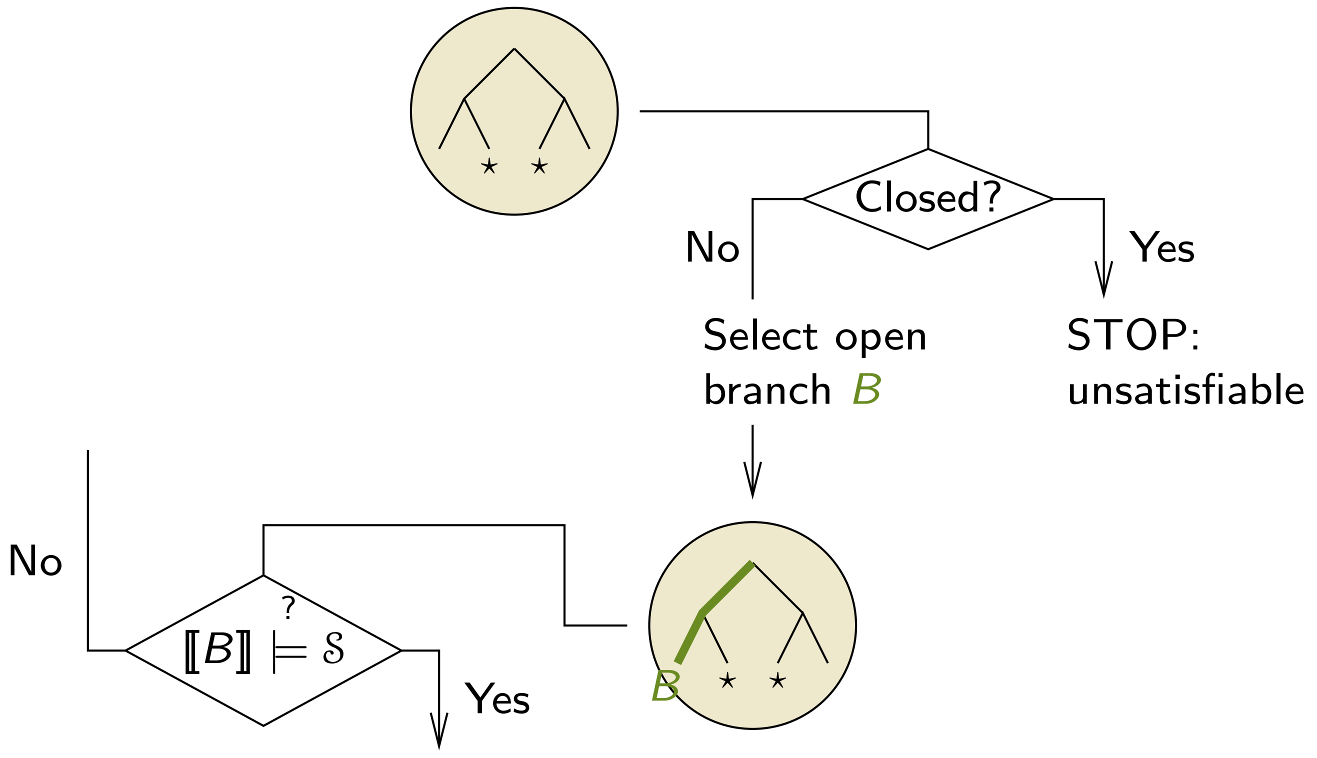


## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** "unsatisfiable" or "satisfiable" (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:

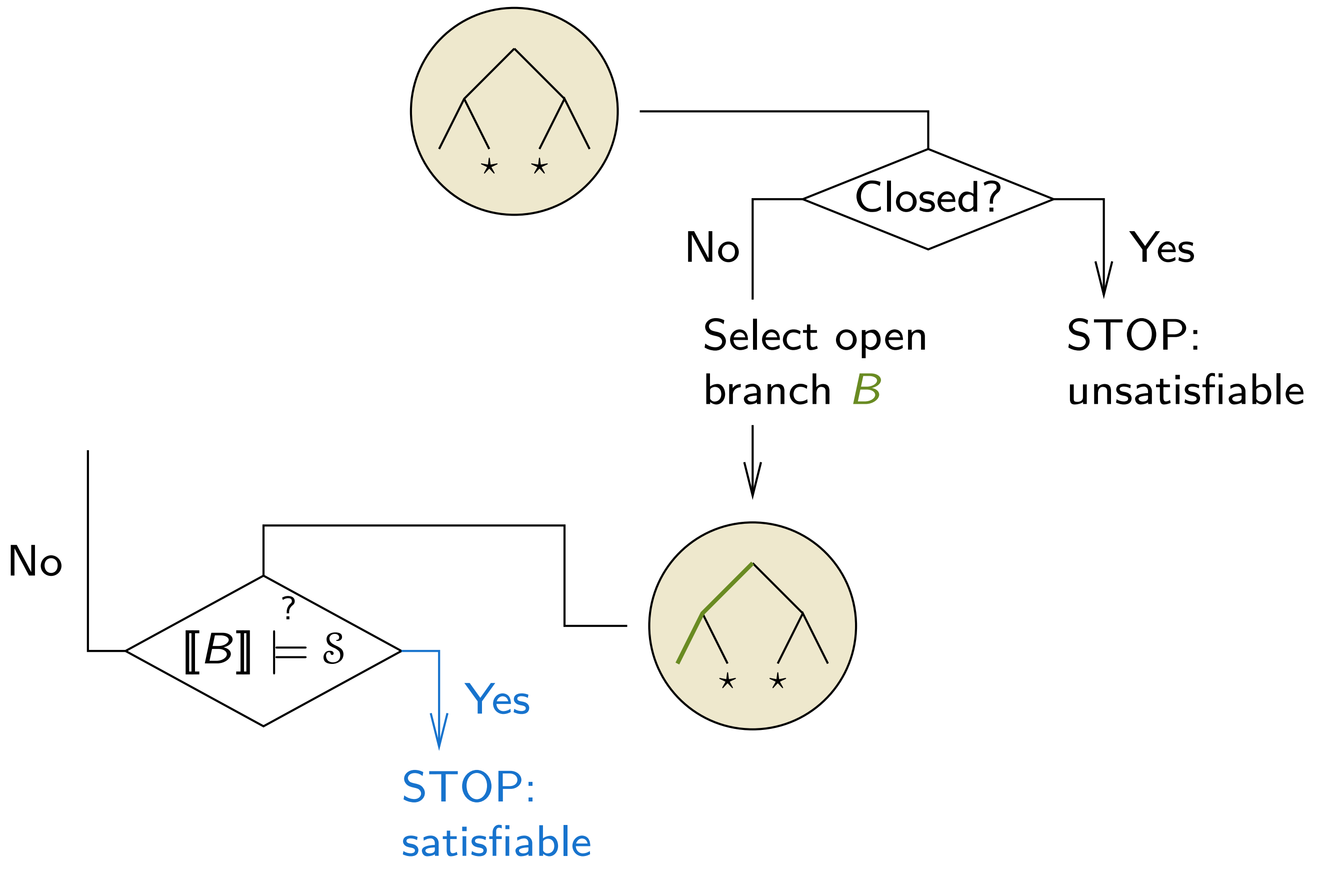


## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** “unsatisfiable” or “satisfiable” (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:

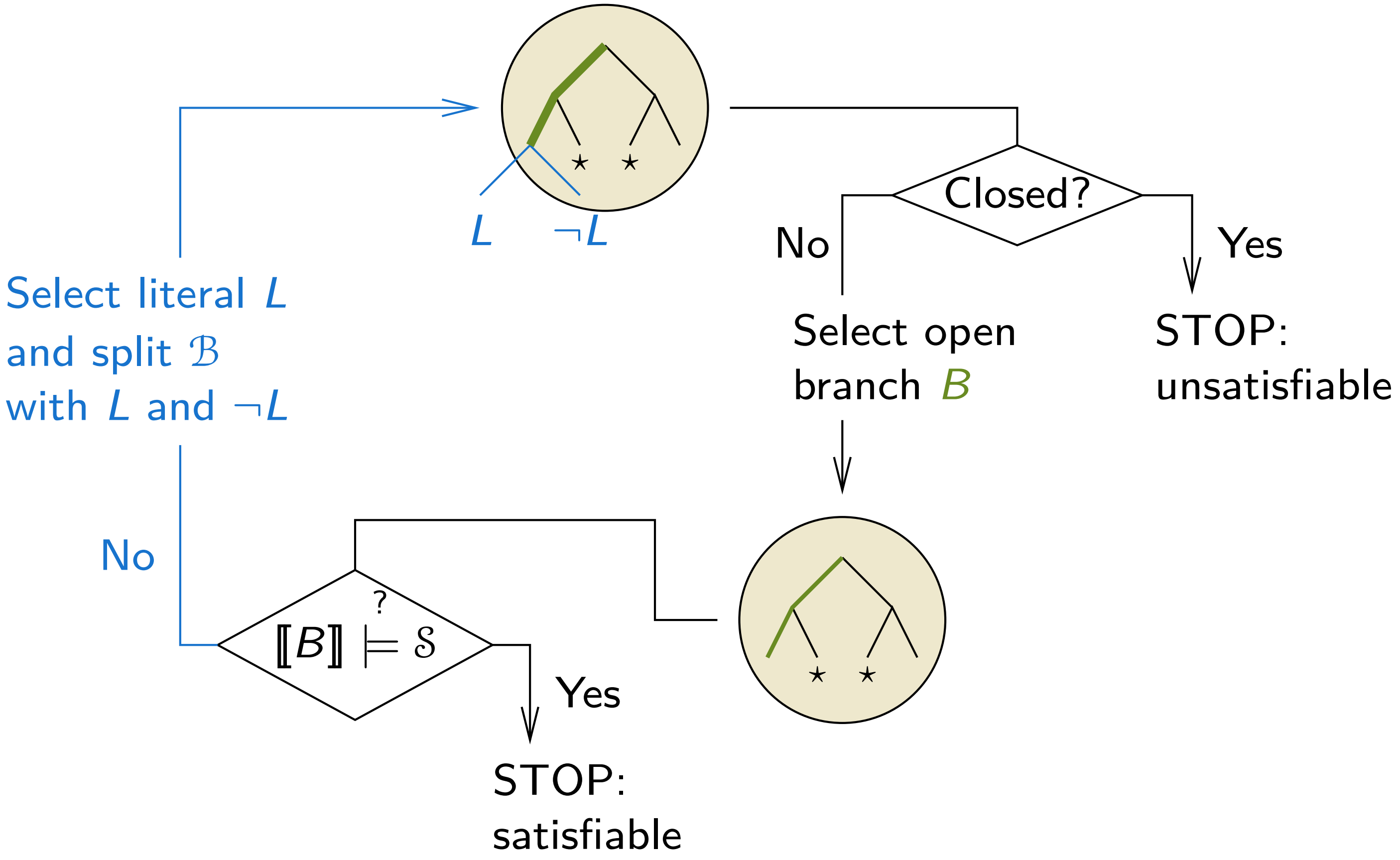


## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** “unsatisfiable” or “satisfiable” (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:



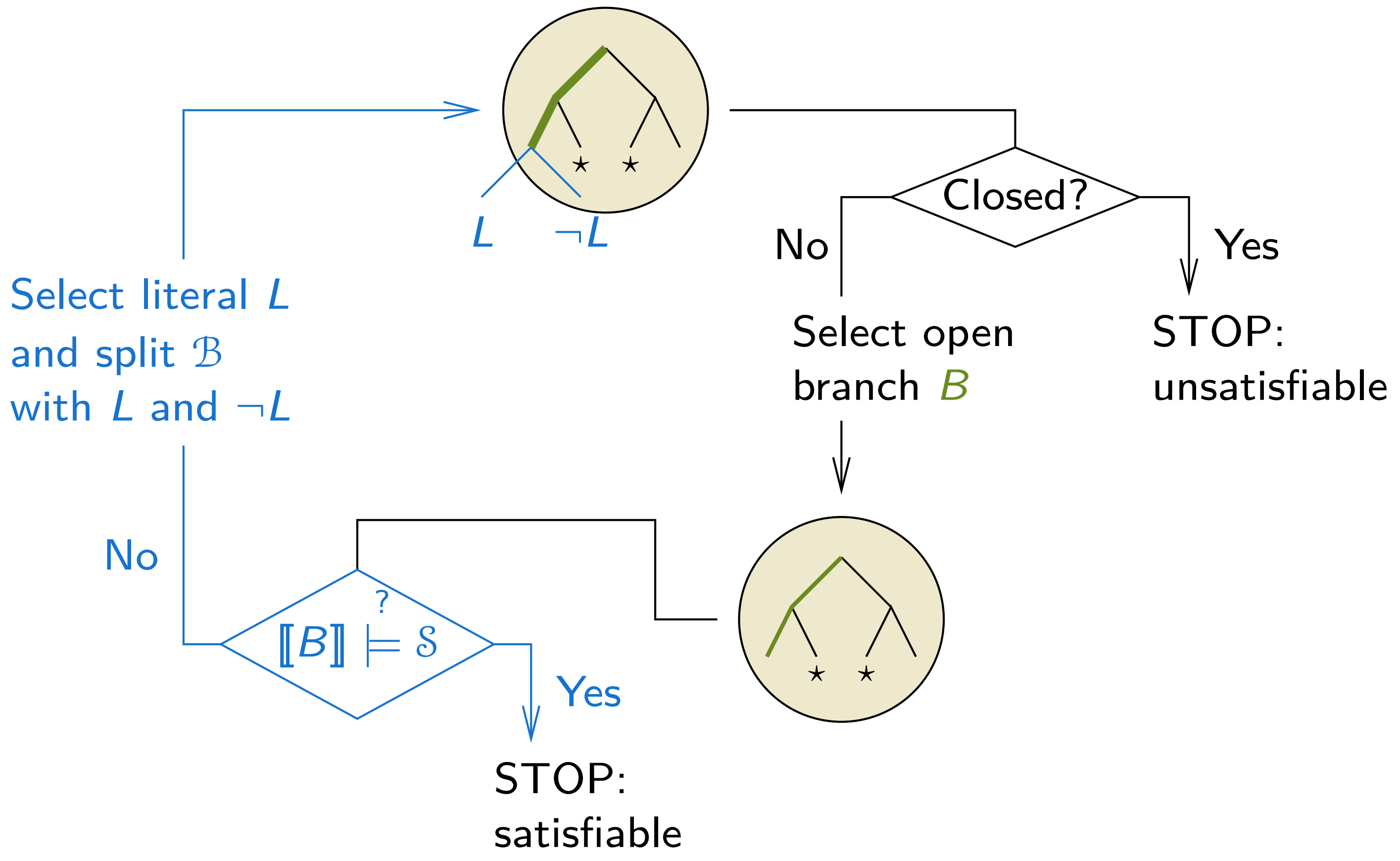
$L$
$\neg L$
Closed?
No
Yes
Select open branch $B$
STOP: unsatisfiable
Select literal $L$ and split $\mathcal{B}$ with $L$ and $\neg L$
No
$[\![B]\!] \overset{?}{\models} \mathcal{S}$
Yes
STOP: satisfiable

## FDPLL Calculus

**Input:** a clause set $\mathcal{S}$

**Output:** “unsatisfiable” or “satisfiable” (if terminates)

Note: Strategy much like in **inner** loop of propositional DPLL:

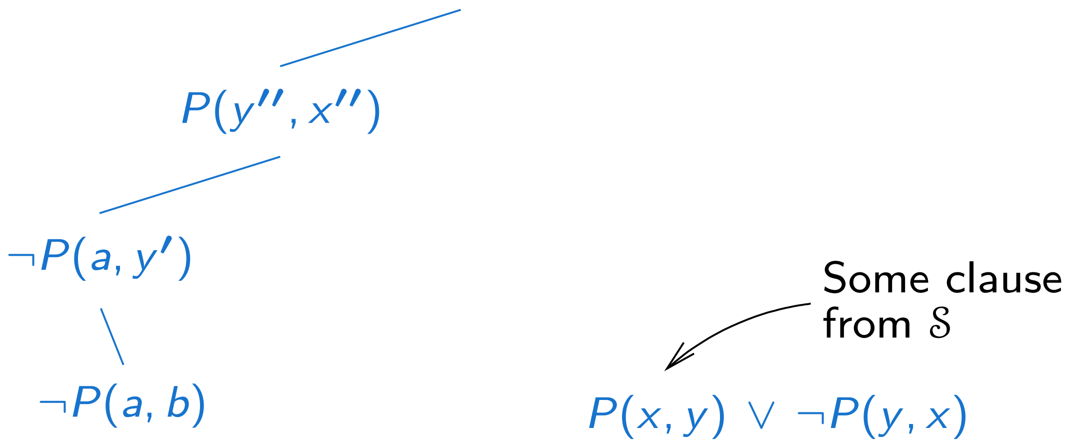


$L$
$\neg L$
Closed?
No
Yes
Select open branch $B$
STOP: unsatisfiable
Select literal $L$ and split $\mathcal{B}$ with $L$ and $\neg L$
No
$[\![B]\!] \overset{?}{\models} \mathcal{S}$
Yes
STOP: satisfiable

**Next:** Testing $[\![B]\!] \models \mathcal{S}$ and splitting

## Calculus: The Splitting Rule

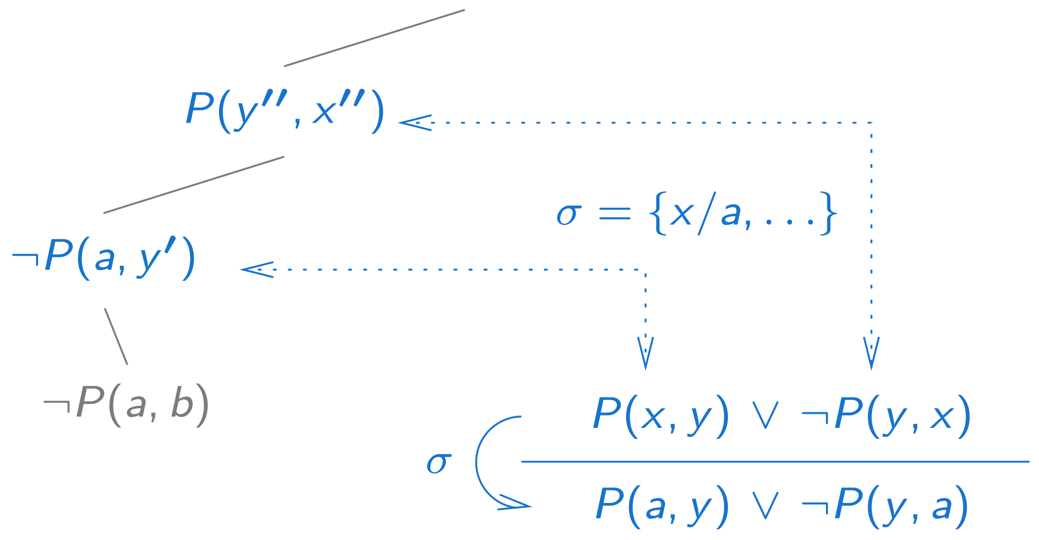
**Purpose:** Satisfy a clause that is currently “false”



1.
2.
3.

## Calculus: The Splitting Rule

**Purpose:** Satisfy a clause that is currently “false”

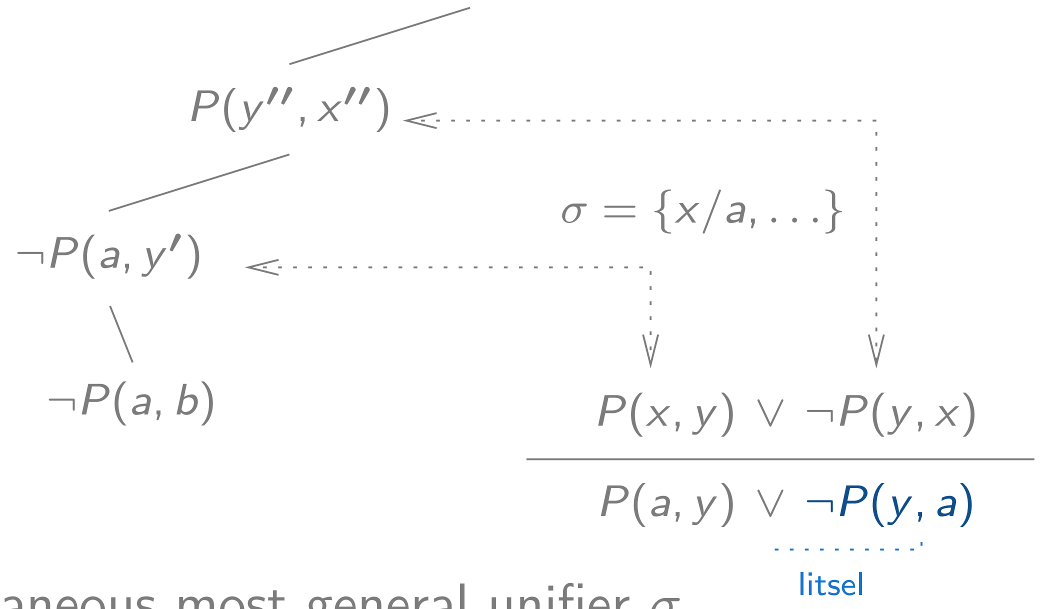


$P(y'', x'')$
$\neg P(a, y')$
$\neg P(a, b)$
$\sigma = \{x/a, \ldots\}$
$\sigma$
$P(x, y) \vee \neg P(y, x)$
$P(a, y) \vee \neg P(y, a)$

1. Compute simultaneous most general unifier $\sigma$
2.
3.

## Calculus: The Splitting Rule

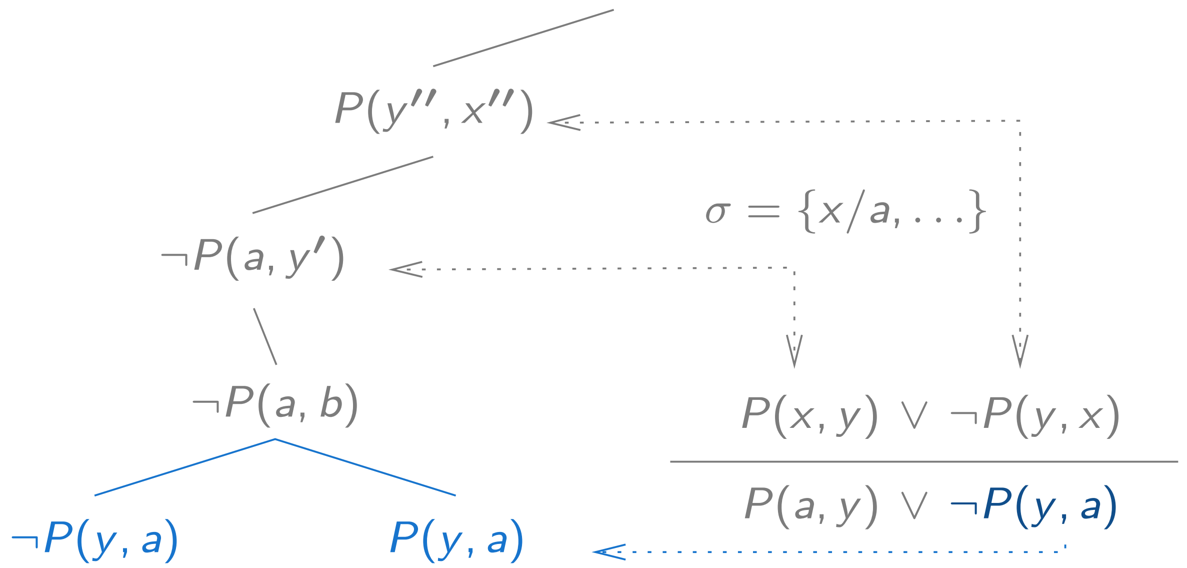
**Purpose:** Satisfy a clause that is currently "false"


$P(y'', x'')$
$\neg P(a, y')$
$\neg P(a, b)$
$\sigma = \{x/a, \ldots\}$
$P(x, y) \vee \neg P(y, x)$
$P(a, y) \vee \neg P(y, a)$
litsel

1. Compute simultaneous most general unifier $\sigma$
2. Select from clause instance a literal not on branch
3.

## Calculus: The Splitting Rule

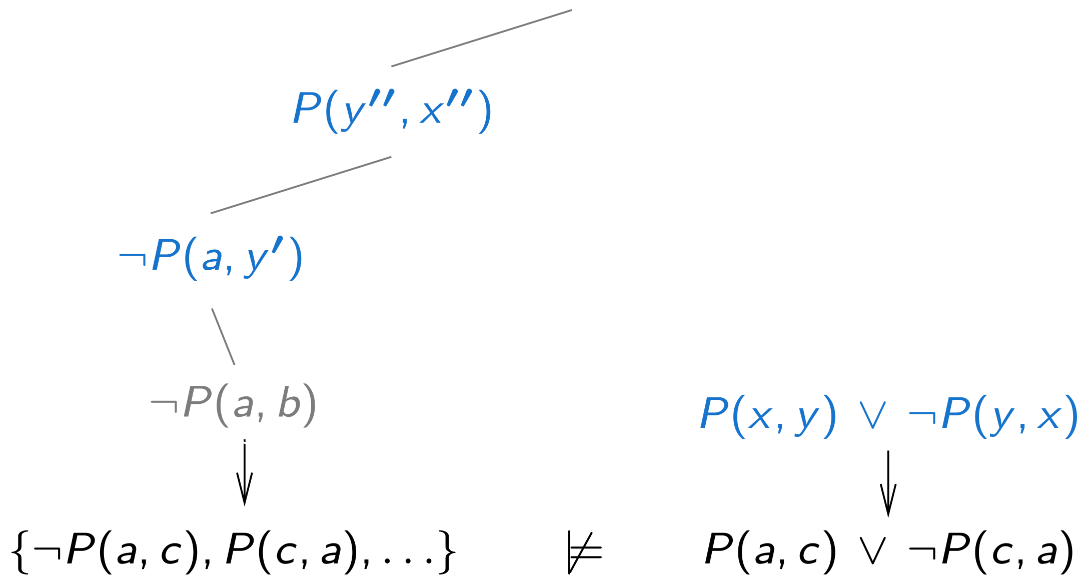
**Purpose:** Satisfy a clause that is currently “false”


$P(y'', x'')$
$\neg P(a, y')$
$\neg P(a, b)$
$\neg P(y, a)$
$P(y, a)$
$\sigma = \{x/a, \ldots\}$
$P(x, y) \vee \neg P(y, x)$
$P(a, y) \vee \neg P(y, a)$

1. Compute simultaneous most general unifier $\sigma$
2. Select from clause instance a literal not on branch
3. Split with this literal

## Calculus: The Splitting Rule

**Purpose:** Satisfy a clause that is currently “false”



$P(y'', x'')$
$\neg P(a, y')$
$\neg P(a, b)$
$P(x, y) \vee \neg P(y, x)$
$\{\neg P(a, c), P(c, a), \ldots\}$
$\not\models$
$P(a, c) \vee \neg P(c, a)$

1. Compute simultaneous most general unifier $\sigma$
2. Select from clause instance a literal not on branch
3. Split with this literal

**This split was really necessary!**

## Calculus: The Splitting Rule

**Purpose:** Satisfy a clause that is currently “false”

$$\begin{tikzcd}
 & & {} \arrow[dl, dash] & \\
 & P(y'', x'') \arrow[dl, dash] & & \\
\neg P(a, y') \arrow[d, dash] & & & \\
\neg P(a, b) \arrow[d] & & & P(x, y) \lor \neg P(y, x) \arrow[d] \\
\{\neg P(a, c), P(c, a), \ldots\} \arrow[rrr, phantom, "\not\models"] & & & P(a, c) \lor \neg P(c, a)
\end{tikzcd}$$

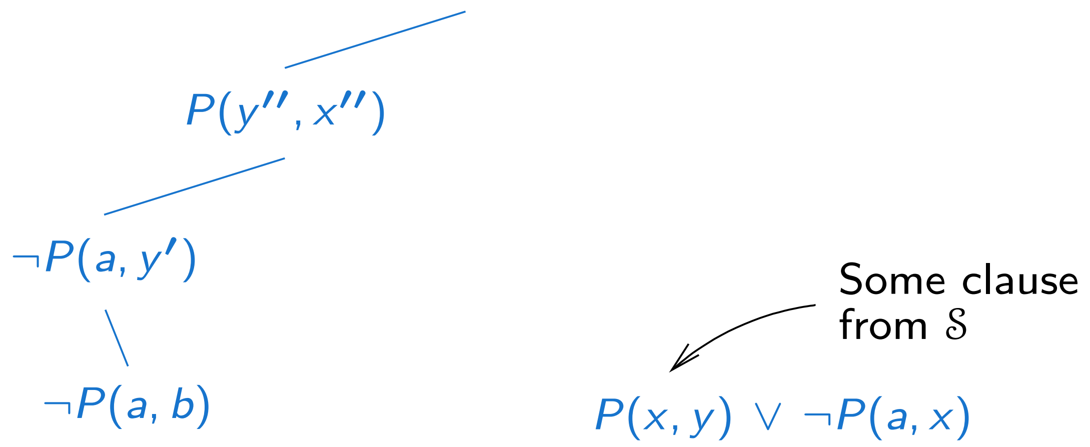
1. Compute simultaneous most general unifier $\sigma$
2. Select from clause instance a literal not on branch
3. Split with this literal

**This split was really necessary!**

**Proposition:** If $[\![\mathcal{B}]\!] \not\models \mathcal{S}$, then split is applicable to some clause from $\mathcal{S}$

## Calculus: The Splitting Rule – Another Example

**Purpose:** Satisfy a clause that is currently "false"

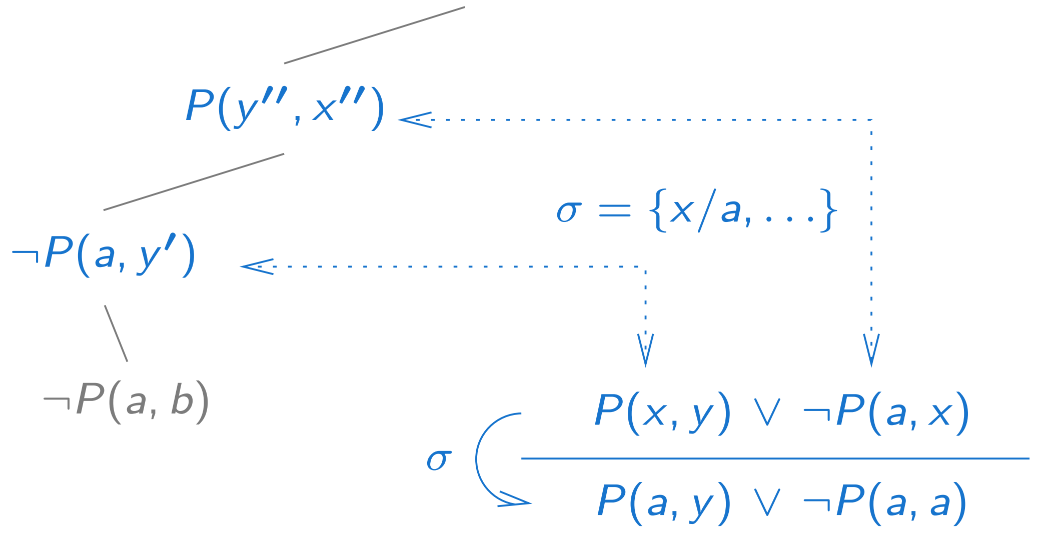


1.

2.

## Calculus: The Splitting Rule – Another Example

**Purpose:** Satisfy a clause that is currently “false”

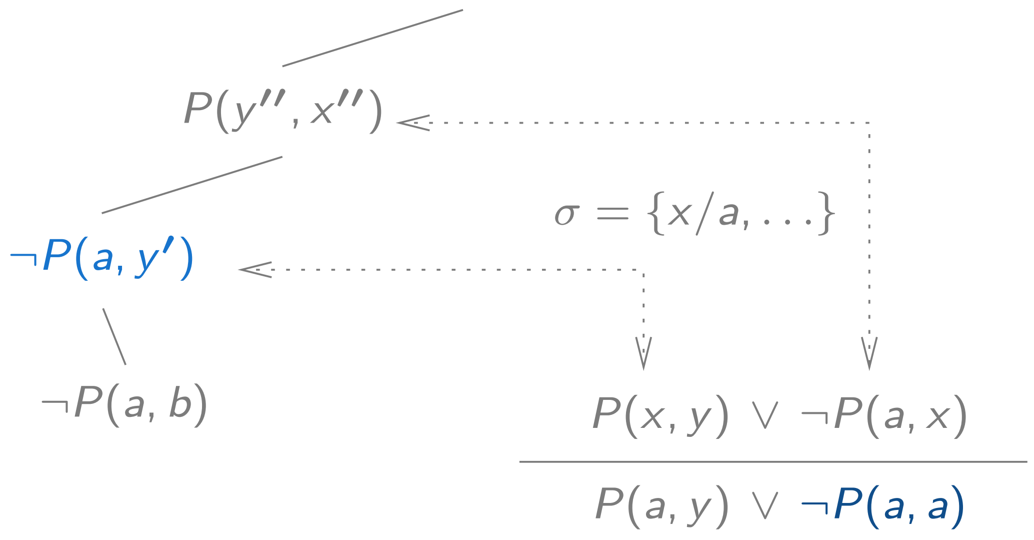


$P(y'', x'')$
$\neg P(a, y')$
$\neg P(a, b)$
$\sigma = \{x/a, \ldots\}$
$\sigma$
$P(x, y) \vee \neg P(a, x)$
$P(a, y) \vee \neg P(a, a)$

1. Compute MGU $\sigma$ of clause against branch literals
2.

## Calculus: The Splitting Rule – Another Example

**Purpose:** Satisfy a clause that is currently "false"

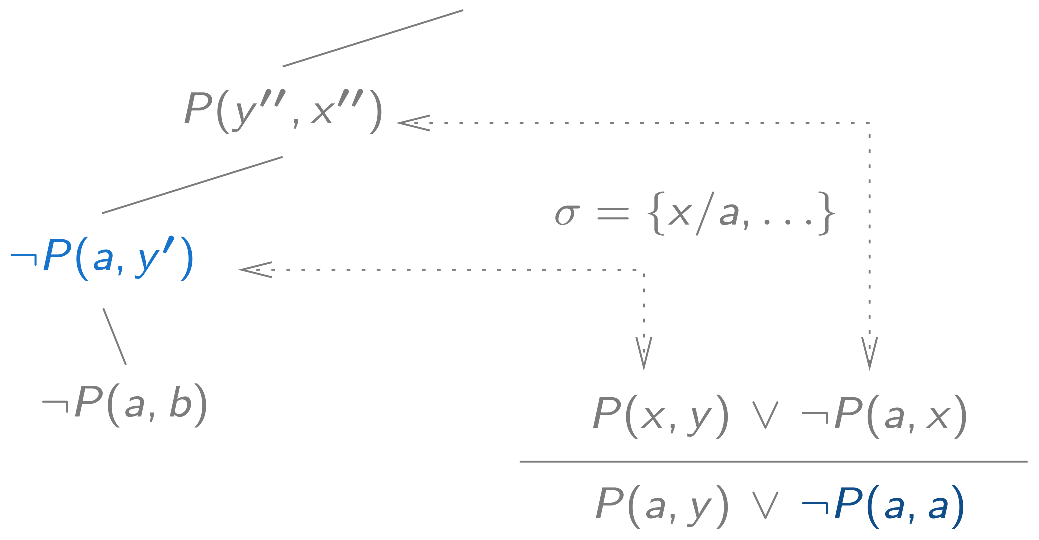


$P(y'', x'')$
$\neg P(a, y')$
$\neg P(a, b)$
$\sigma = \{x/a, \ldots\}$
$P(x, y) \vee \neg P(a, x)$
$P(a, y) \vee \neg P(a, a)$

1. Compute MGU $\sigma$ of clause against branch literals
2. If clause contains "true" literal, then split is not applicable

## Calculus: The Splitting Rule – Another Example

**Purpose:** Satisfy a clause that is currently “false”



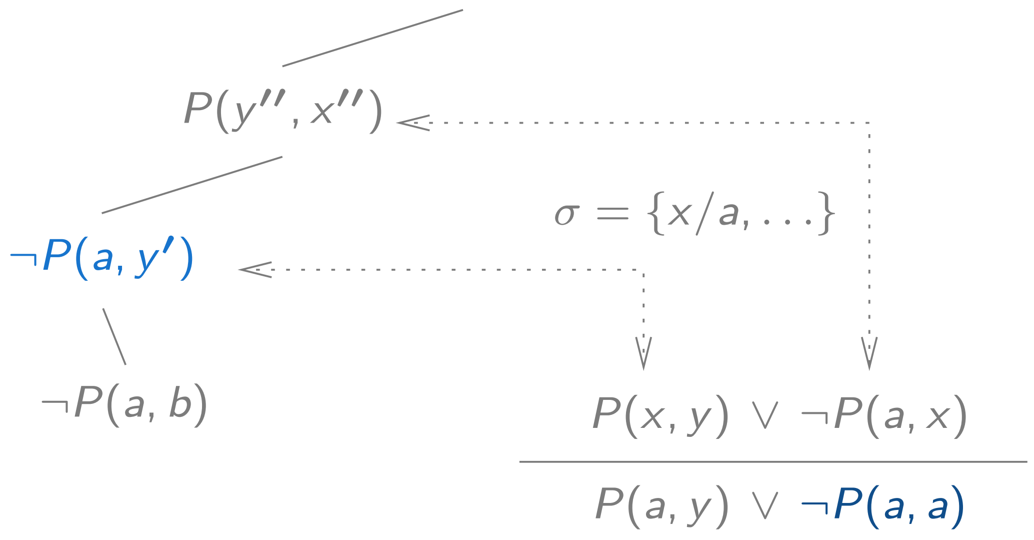
$P(y'', x'')$
$\neg P(a, y')$
$\neg P(a, b)$
$\sigma = \{x/a, \ldots\}$
$P(x, y) \lor \neg P(a, x)$
$P(a, y) \lor \neg P(a, a)$

1. Compute MGU $\sigma$ of clause against branch literals
2. If clause contains “true” literal, then split is not applicable

**Non-applicability is a redundancy test**

## Calculus: The Splitting Rule – Another Example

**Purpose:** Satisfy a clause that is currently "false"



$P(y'', x'')$
$\neg P(a, y')$
$\neg P(a, b)$
$\sigma = \{x/a, \ldots\}$
$P(x, y) \vee \neg P(a, x)$
$P(a, y) \vee \neg P(a, a)$

1. Compute MGU $\sigma$ of clause against branch literals
2. If clause contains "true" literal, then split is not applicable

**Non-applicability is a redundancy test**

**Proposition:** If for no clause split is applicable, $[\![\mathcal{B}]\!] \models \mathcal{S}$ holds

## FDPLL Complete Example

```
(1)  train(X,Y) ; flight(X,Y).          %% train from X to Y or flight from X to
(2)  -flight(koblenz,X).                %% no flight from koblenz to anywhere.
(3)  flight(X,Y) :- flight(Y,X).        %% flight is symmetric.
(4)  connect(X,Y) :- flight(X,Y).       %% a flight is a connection.
(5)  connect(X,Y) :- train(X,Y).        %% a train is a connection.
(6)  connect(X,Z) :- connect(X,Y),      %% connection is a transitive relation.
                     connect(Y,Z).
```

## FDPLL Complete Example

```
(1)  train(X,Y) ; flight(X,Y).        %% train from X to Y or flight from X to
(2)  -flight(koblenz,X).              %% no flight from koblenz to anywhere.
(3)  flight(X,Y) :- flight(Y,X).      %% flight is symmetric.
(4)  connect(X,Y) :- flight(X,Y).     %% a flight is a connection.
(5)  connect(X,Y) :- train(X,Y).      %% a train is a connection.
(6)  connect(X,Z) :- connect(X,Y),    %% connection is a transitive relation.
                     connect(Y,Z).
```

### Computed Model (as output by implementation)

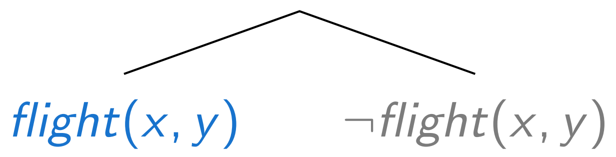
```
(1)    + flight(X, Y)
(2)    - flight(koblenz, X)
(3)    - flight(X, koblenz)
(4)    + train(koblenz, Y)
(5)    + train(Y, koblenz)
(6)    + connect(X, Y)
```

## FDPLL Model Computation Example - Derivation

$\langle$empyty tree$\rangle$

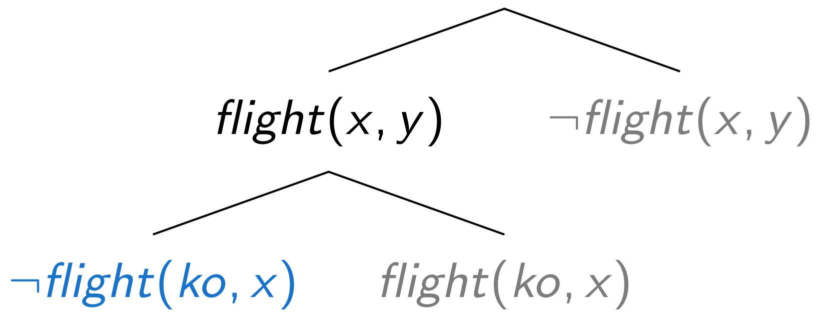
Clause instance used in inference: $\mathit{train}(x, y) \vee \mathit{flight}(x, y)$

## FDPLL Model Computation Example - Derivation



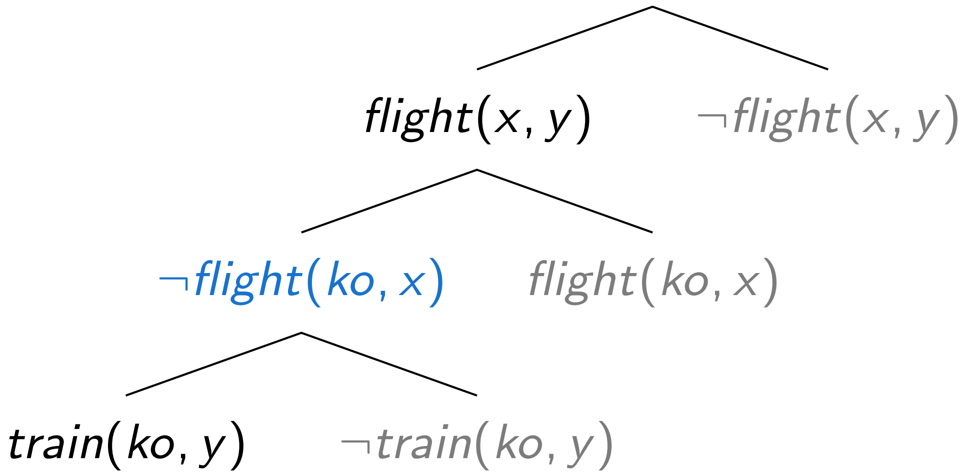
Clause instance used in inference: $\neg flight(ko, x)$

## FDPLL Model Computation Example - Derivation



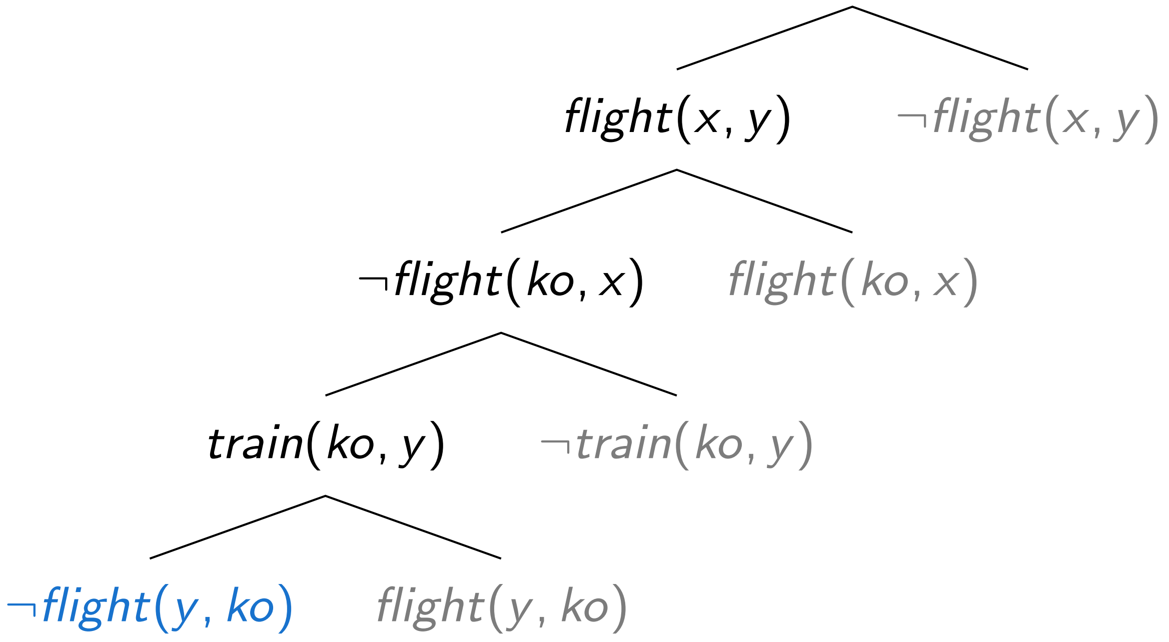
Clause instance used in inference: $train(ko, y) \vee flight(ko, y)$

## FDPLL Model Computation Example - Derivation



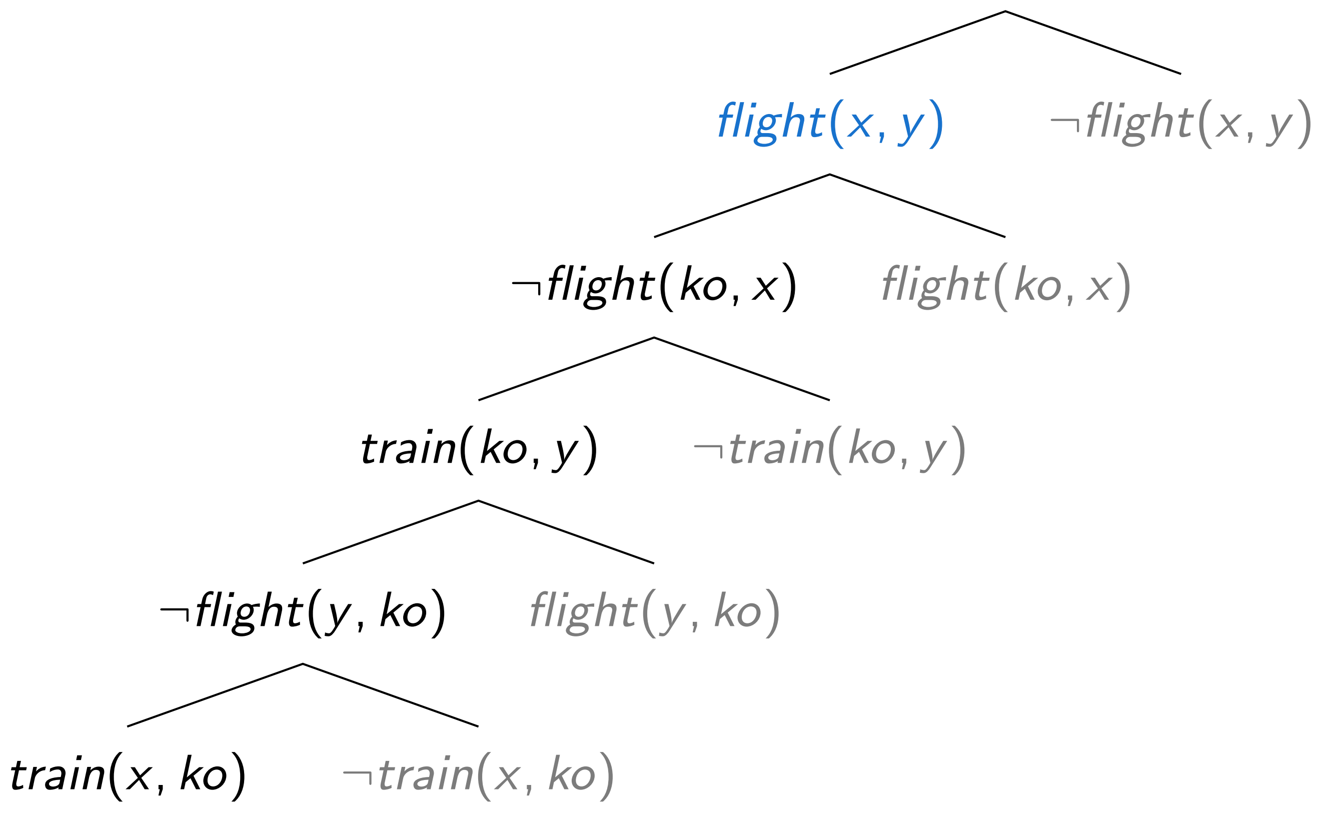
Clause instance used in inference: $flight(ko, y) \lor \neg flight(y, ko)$

## FDPLL Model Computation Example - Derivation



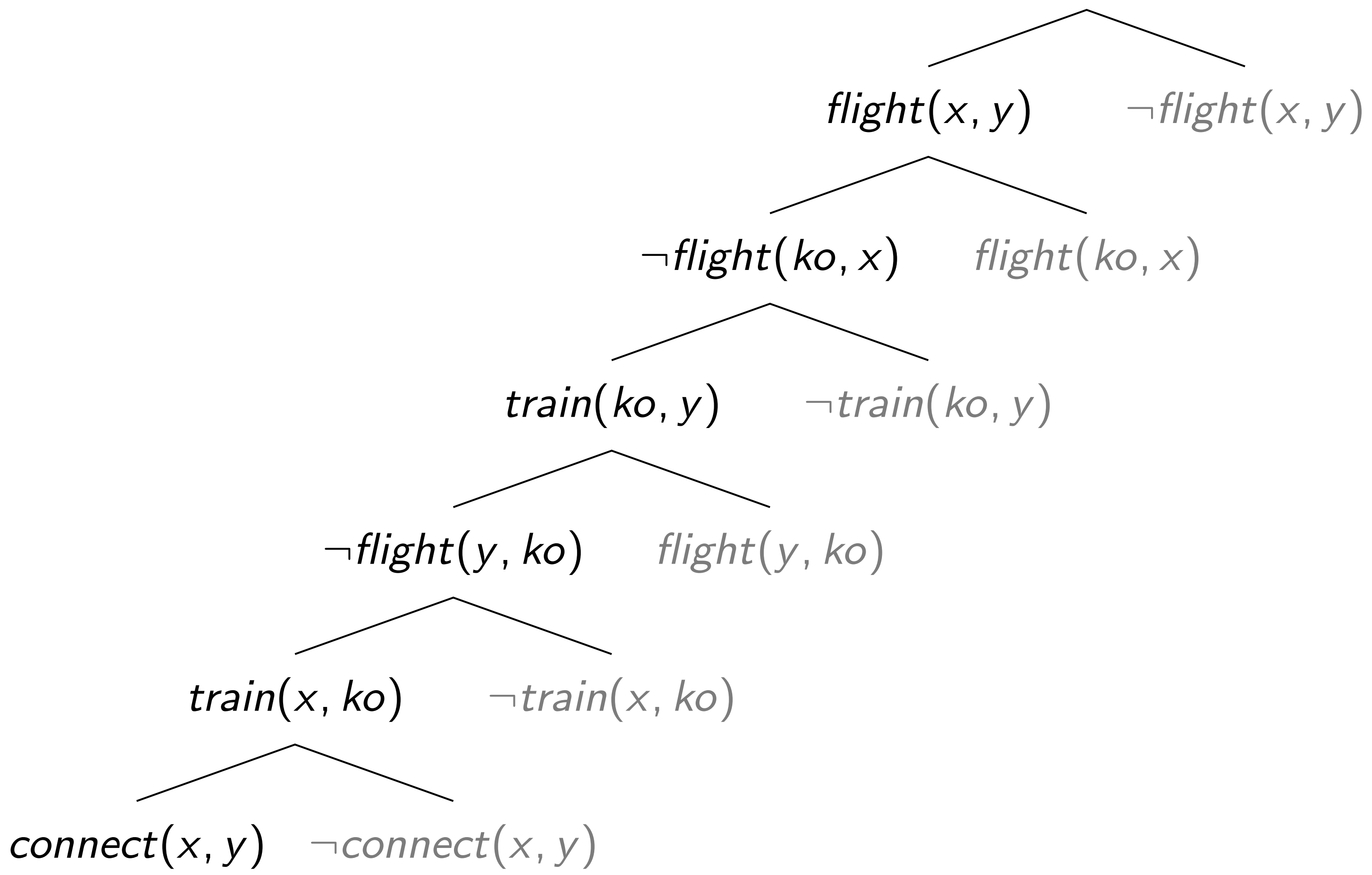
Clause instance used in inference: $train(x, ko) \vee flight(x, ko)$

## FDPLL Model Computation Example - Derivation



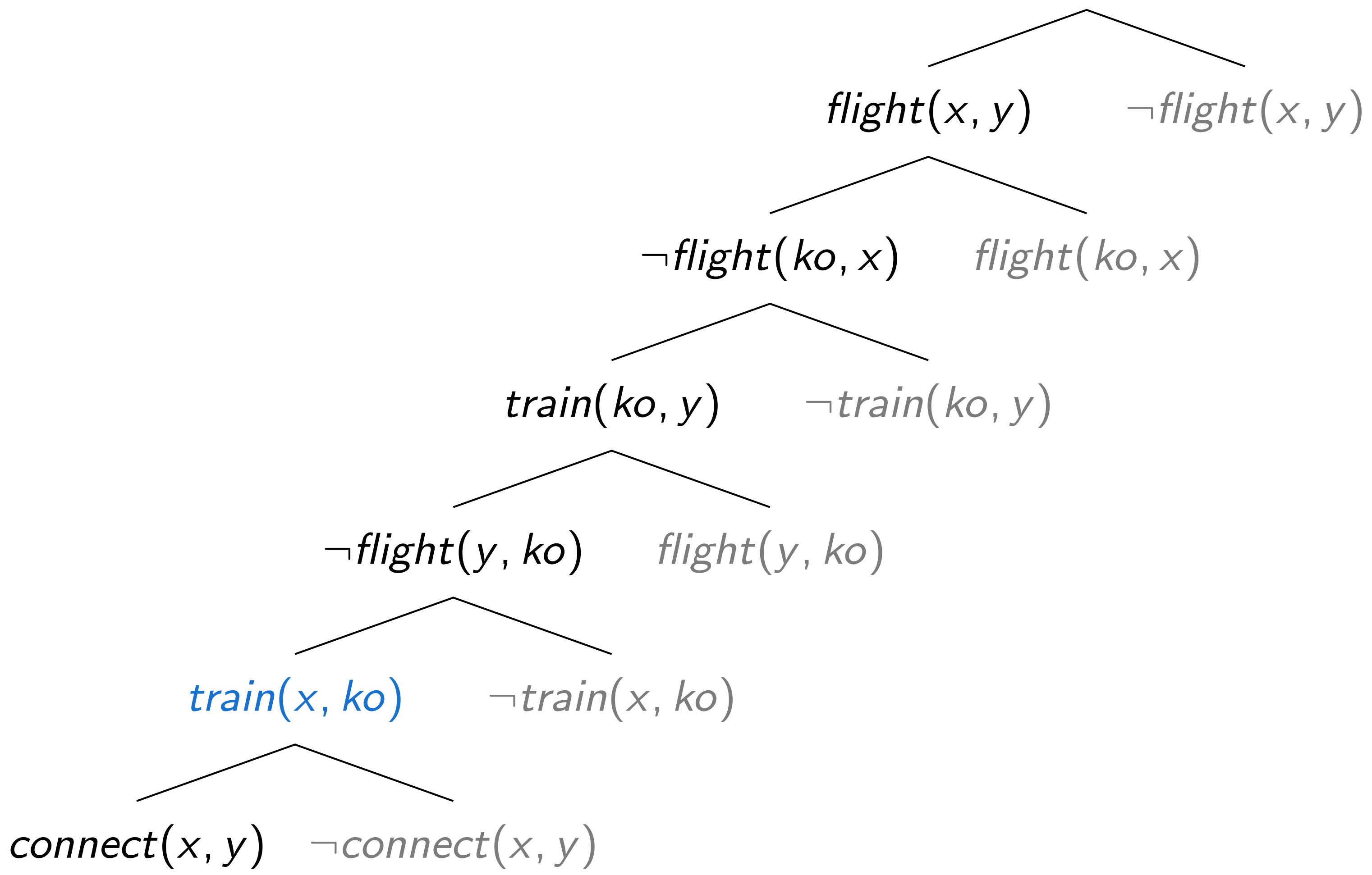
Clause instance used in inference: $connect(x, y) \vee \neg flight(x, y).$

## FDPLL Model Computation Example - Derivation



**Done.** Return “satisfiable with model $\{flight(x, y), \ldots, connect(x, y)\}$”

## FDPLL Model Computation Example - Derivation



**Done.** Return “satisfiable with model $\{flight(x,y),\ldots,connect(x,y)\}$”

**Redundancy:** Instance **not** used in inference: $connect(x,ko) \vee \neg train(x,ko)$

## Summary / Properties

### Summary

- DPLL data structure lifted to first-order logic level
- Two simple inference rules, controlled by unification
- Computes with interpretations/models
- Semantical redundancy criterion

## Summary / Properties

### Summary

- DPLL data structure lifted to first-order logic level
- Two simple inference rules, controlled by unification
- Computes with interpretations/models
- Semantical redundancy criterion

### Properties

- Soundness and completeness (with fair strategy).
- Extension: More efficient reasoning with **unit clauses** (e.g. $\forall x\ P(x, a)$)
- Proof convergence (avoids backtracking the semantics trees)
- Decides function-free clause logic (Bernays-Schönfinkel class)
  Covers e.g. Basic modal logic, Description logic, DataLog
  Returns model in satisfiable case
- Can be combined with Resolution, equality inference rules

## Calculi in Comparison

Consider a transitivity clause $P(x, z) \leftarrow P(x, y) \wedge P(y, z)$.

## Calculi in Comparison

Consider a transitivity clause $P(x,z) \leftarrow P(x,y) \wedge P(y,z)$.

[Bachmair & Ganzinger, Handbook AR 2001], [Fermüller et. al., Handbook AR 2001]

Resolution:

$$P(x,z') \leftarrow P(x,y) \wedge P(y,z) \wedge P(z,z')$$
$$P(x,z'') \leftarrow P(x,y) \wedge P(y,z) \wedge P(z,z') \wedge P(z',z'')$$

## Calculi in Comparison

Consider a transitivity clause $P(x,z) \leftarrow P(x,y) \wedge P(y,z)$.

[Bachmair & Ganzinger, Handbook AR 2001], [Fermüller et. al., Handbook AR 2001]

Resolution:

$$P(x,z') \leftarrow P(x,y) \wedge P(y,z) \wedge P(z,z')$$
$$P(x,z'') \leftarrow P(x,y) \wedge P(y,z) \wedge P(z,z') \wedge P(z',z'')$$

**Does not terminate for function-free clause sets**

**Complicated to extract model**

Very good on other classes, Equality

## Calculi in Comparison

[Bachmair & Ganzinger, Handbook AR 2001], [Fermüller et. al., Handbook AR 2001]

Consider a transitivity clause $P(x, z) \leftarrow P(x, y) \wedge P(y, z)$.

Resolution:

$$P(x, z') \leftarrow P(x, y) \wedge P(y, z) \wedge P(z, z')$$
$$P(x, z'') \leftarrow P(x, y) \wedge P(y, z) \wedge P(z, z') \wedge P(z', z'')$$

**Does not terminate for function-free clause sets**

**Complicated to extract model**

Very good on other classes, Equality

Rigid Variable Approaches:

$$P(x', z') \leftarrow P(x', y') \wedge P(y', z')$$
$$P(x'', z'') \leftarrow P(x'', y'') \wedge P(y'', z'')$$

**Tableaux and Connection Methods**

## Calculi in Comparison

Consider a transitivity clause $P(x,z) \leftarrow P(x,y) \wedge P(y,z)$.

[Bachmair & Ganzinger, Handbook AR 2001], [Fermüller et. al., Handbook AR 2001]

Resolution:

$$P(x,z') \leftarrow P(x,y) \wedge P(y,z) \wedge P(z,z')$$
$$P(x,z'') \leftarrow P(x,y) \wedge P(y,z) \wedge P(z,z') \wedge P(z',z'')$$

**Does not terminate for function-free clause sets**

**Complicated to extract model**

Very good on other classes, Equality

Rigid Variable Approaches:

$$P(x',z') \leftarrow P(x',y') \wedge P(y',z')$$
$$P(x'',z'') \leftarrow P(x'',y'') \wedge P(y'',z'')$$

**Tableaux and Connection Methods**

**Unpredictable number of variants, weak redundancy test**

**Difficult to avoid unnecessary (!) backtracking**

**Difficult to extract model**

## Calculi in Comparison

Consider a transitivity clause $P(x,z) \leftarrow P(x,y) \land P(y,z)$.

## Calculi in Comparison

Consider a transitivity clause $P(x,z) \leftarrow P(x,y) \wedge P(y,z)$.

Instance Based Methods:

$$P(x,z) \leftarrow P(x,y) \wedge P(y,z)$$
$$P(a,z) \leftarrow P(a,y) \wedge P(y,b)$$

FDPLL, Model Evolution,
Inst-Gen, Disconnection Tableaux,
Overview paper on my web page

## Calculi in Comparison

Consider a transitivity clause $P(x,z) \leftarrow P(x,y) \wedge P(y,z)$.

Instance Based Methods:

$$P(x,z) \leftarrow P(x,y) \wedge P(y,z)$$
$$P(a,z) \leftarrow P(a,y) \wedge P(y,b)$$

FDPLL, Model Evolution,
Inst-Gen, Disconnection Tableaux,
Overview paper on my web page

**Weak redundancy criterion (no subsumption)**

**Need to keep clause instances (memory problem)**

## Calculi in Comparison

Consider a transitivity clause $P(x, z) \leftarrow P(x, y) \wedge P(y, z)$.

Instance Based Methods:

$$P(x, z) \leftarrow P(x, y) \wedge P(y, z)$$
$$P(a, z) \leftarrow P(a, y) \wedge P(y, b)$$

FDPLL, Model Evolution,
Inst-Gen, Disconnection Tableaux,
Overview paper on my web page

**Weak redundancy criterion (no subsumption)**

**Need to keep clause instances (memory problem)**

Clauses do not become longer (cf. Resolution)

May delete variant clauses (cf. Rigid Variable Approach)

## Contents

## Model Generation

For every FOL formula $F$ exactly one of these three cases applies:

1. $F$ is unsatisfiable

   (Complete) theorem prover will detect this eventually (in theory)

2. $F$ is satisfiable with only infinite models

   Example: $nat(0)$ $\qquad$ $lt(x, succ(N)) \leftarrow nat(x)$

   $nat(succ(x)) \leftarrow nat(x)$ $\qquad$ $lt(x, z) \leftarrow lt(x, y) \wedge lt(y, z)$

   $\neg lt(x, x)$

   Sometimes resolution refinements help to detect such cases

3. $F$ is satisfiable with a finite model

   A **finite model-finder** will detect this eventually (in theory)

The rest of this section is concerned with computing finite models.

## Model Generation

Two main applications:

- To disprove a “false” theorem by means of a counterexample, i.e., a “countermodel”
- A model provides the expected answer to the user, as in the three coloring example

### Some applications

Planning: Can be formalised as propositional satisfiability problem.
[Kautz& Selman, AAAI96; Dimopolous et al, ECP97]

Diagnosis: Minimal models of *abnormal* literals (circumscription). [Reiter, AI87]

Databases: View materialisation, View Updates, Integrity Constraints.

Nonmonotonic reasoning: Various semantics (GCWA, Well-founded, Perfect, Stable,. . . ), all based on minimal models. [Inoue et al, CADE 92]

Software Verification: Counterexamples to conjectured theorems.

Theorem proving: Counterexamples to conjectured theorems.

## Example - Discourse Representation

Natural Language Processing:

- Maintain models $\mathcal{I}_1, \ldots, \mathcal{I}_n$ as different readings of discourses:

$$\mathcal{I}_i \models \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far}$$

## Example - Discourse Representation

Natural Language Processing:

- Maintain models $\mathcal{I}_1, \ldots, \mathcal{I}_n$ as different readings of discourses:

$$\mathcal{I}_i \models \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far}$$

- Consistency checks ("Mia's husband loves Sally. She is not married.")

$$\begin{array}{ll} & \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far} \not\models \neg \mathit{New\_utterance} \\ \mathit{iff} & \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far} \cup \mathit{New\_utterance} \text{ is } \textbf{satisfiable} \end{array}$$

## Example - Discourse Representation

Natural Language Processing:

- Maintain models $\mathcal{I}_1, \ldots, \mathcal{I}_n$ as different readings of discourses:

$$\mathcal{I}_i \models \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far}$$

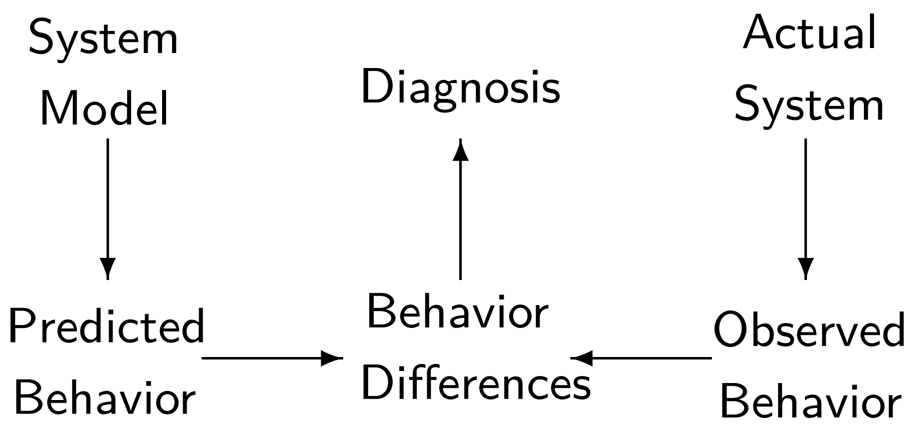
- Consistency checks ("Mia's husband loves Sally. She is not married.")

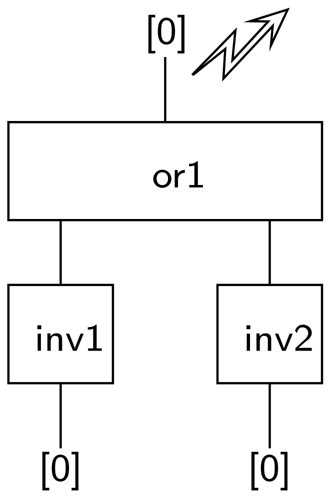
$$\begin{array}{ll} & \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far} \not\models \neg \mathit{New\_utterance} \\ \mathit{iff} & \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far} \cup \mathit{New\_utterance} \text{ is } \textbf{satisfiable} \end{array}$$

- Informativity checks ("Mia's husband loves Sally. She is married.")

$$\begin{array}{ll} & \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far} \not\models \mathit{New\_utterance} \\ \mathit{iff} & \mathit{BG\text{-}Knowledge} \cup \mathit{Discourse\_so\_far} \cup \neg \mathit{New\_utterance} \text{ is } \textbf{satisfiable} \end{array}$$

## Example - Model-Based Diagnosis [Reiter 87]





**Formal Treatment:**

$COMP$ = Components

$SD$ = System description, components are allowed to perform "abnormal"

$OBS$ = Observations

Def. **Diagnosis**: Some minimal $\Delta \subseteq COMP$ such that

$$SD \cup OBS \cup \{ab(\Delta)\} \cup \{\neg ab(COMP - \Delta)\} \text{ is consistent}$$

## Formal Treatment

System Description $SD =$

| | | | |
|---|---|---|---|
| OR1: | $\neg(ab(or1))$ | $\rightarrow$ | $high(or1,o) \leftrightarrow (high(or1,i1) \vee high(or1,i2))$ |
| INV1: | $\neg(ab(inv1))$ | $\rightarrow$ | $high(inv1,o) \leftrightarrow \neg(high(inv1,i))$ |
| INV2: | $\neg(ab(inv2))$ | $\rightarrow$ | $high(inv2,o) \leftrightarrow \neg(high(inv2,i))$ |
| CONN1: | | | $high(inv1,o) \leftrightarrow high(or1,i1)$ |
| CONN2: | | | $high(inv2,o) \leftrightarrow high(or1,i2)$ |

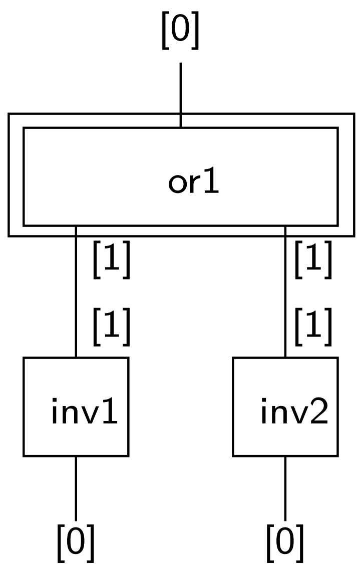
Observations $OBS =$

| | |
|---|---|
| LOW_INV1_I: | $\neg(high(inv1,i))$ |
| LOW_INV1_I: | $\neg(high(inv2,i))$ |
| LOW_OR1_O: | $\neg(high(or1,o))$ |

**Task:** Find minimal $\Delta \subseteq \{ab(or1),\ ab(inv1),\ ab(inv2)\}$ such that

$$SD \cup OBS \cup \Delta \cup \neg\overline{\Delta} \text{ is consistent}$$

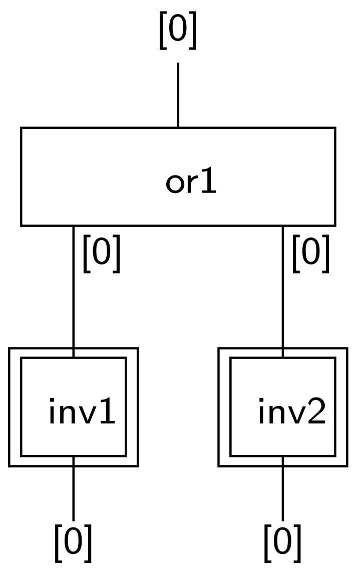
## Cardinality of Diagnosis



Single Fault Diagnosis $\Delta 1 = \{ab(or1)\}$

$$1F = \begin{cases} \leftarrow ab(inv1) \land ab(inv2) \\ \leftarrow ab(inv1) \land ab(or1) \\ \leftarrow ab(inv2) \land ab(or1) \end{cases}$$

$SD \cup OBS \cup \Delta 1 \cup \neg\overline{\Delta 1} \cup 1F$ is consistent



2-Fault Diagnosis $\Delta 2 = \{ab(inv1),\ ab(inv2)\}$

$$2F = \left\{ \leftarrow ab(inv1) \land ab(inv2) \land ab(or1) \right.$$

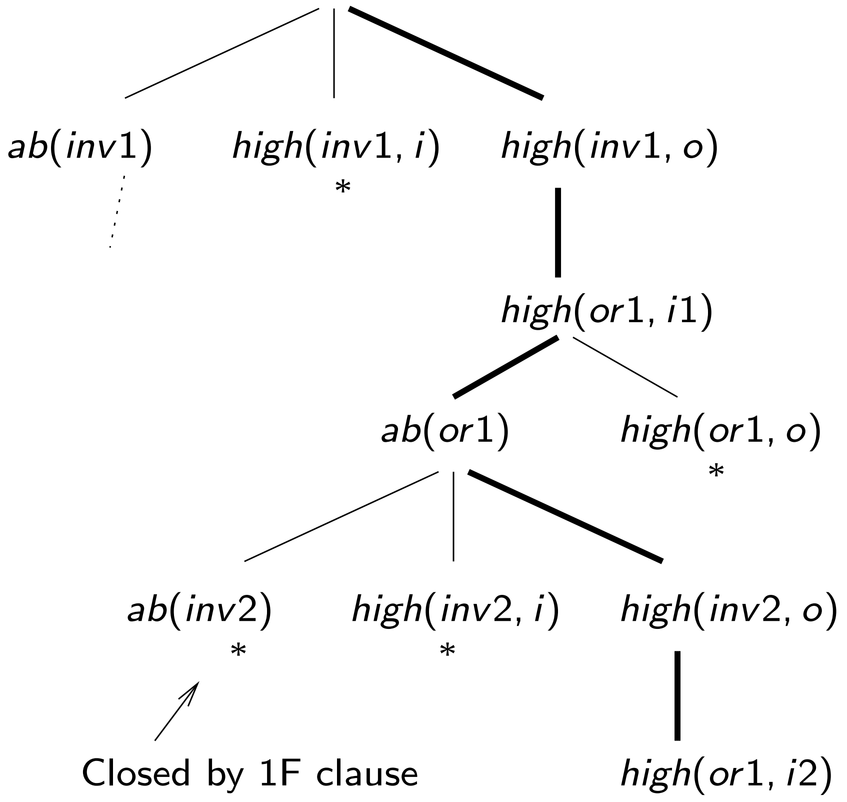
$SD \cup OBS \cup \Delta 2 \cup \neg\overline{\Delta 2} \cup 1F$ is **in**consistent

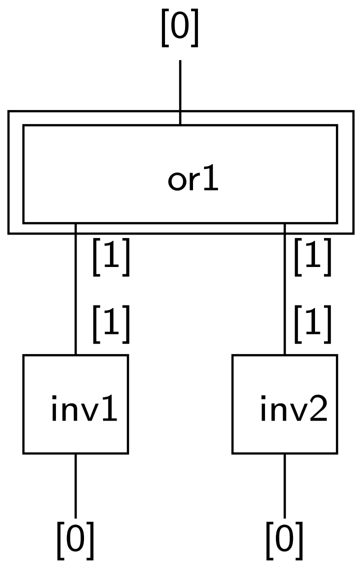
$SD \cup OBS \cup \Delta 2 \cup \neg\overline{\Delta 2} \cup 2F$ is consistent

How to compute a (all) diagnosis $\Delta 1$, $\Delta 2$, ...?

## Tableaux Construction to Determine Models

The following tree uses the clause normal form of $SD \cup OBS \cup 1F$ to compute a (all) single fault diagnosis $\Delta 1$

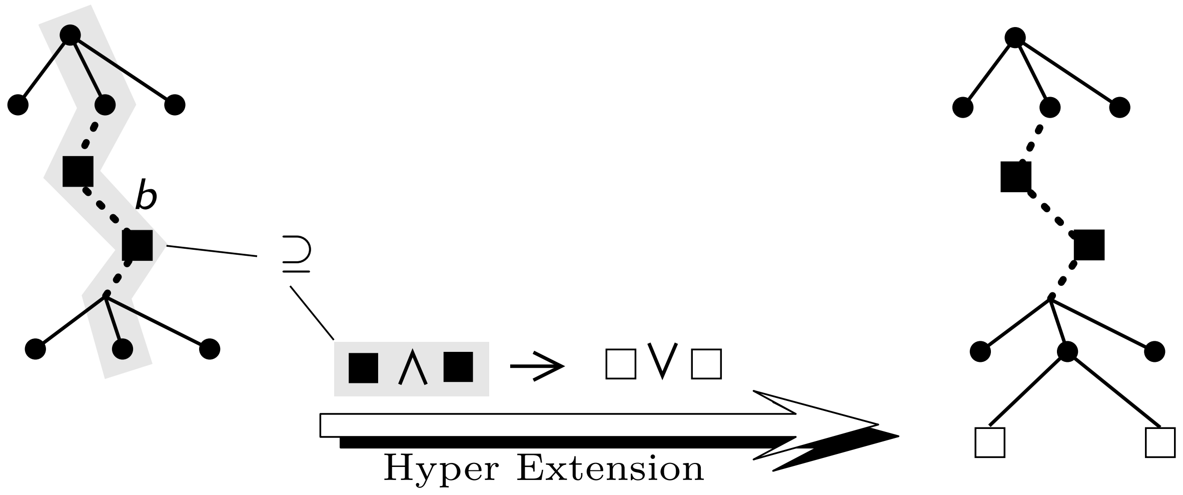




Minimality of diagnosis: add lemma $\bigvee_{L \in \Delta i} \overline{L}$ for each diagnosis $\Delta i$ as soon as it is computed, for $i = 1, 2, \ldots$.

What is the calculus behind?

## (Ground) Hyper Tableaux ($\approx$ Hyperresolution + Splitting)



- All nodes contain **positive** literals, current branch $b$ selected arbitrarily.
- Model construction: $b \mapsto$ interpretation $[\![b]\!]$
  - Redundancy: $[\![b]\!] \models ■ \wedge ■ \rightarrow □ \vee □$
  - Finished: branch $b$ is **finished** iff$_{Def.}$ every clause is redundant in $[\![b]\!]$.
  - Completeness: every open finished branch $b$ contains a model $[\![b]\!]$.
    **Here:** Minimal Model Completeness: For every minimal model $\mathcal{I}$ there is an open finished branch $b$ with $\mathcal{I} = [\![b]\!]$.
  - Minimal Diagnosis $\approx$
    Minimal Model wrt. *ab*-literals.

## Method: SATCHMO [Manthey/Bry 1988]

A very lean implementation of a bottum-up model generation method on top of Prolog.

1. Convert clauses to range-restricted form:

   $q(x) \vee p(x,y) \leftarrow q(x) \quad \leadsto$ `q(X) ; p(X,Y) <- q(X), dom(Y)`

2. `assert` range-restricted clauses and `dom` clauses in Prolog database.

3. Call `satisfiable`:

```
satisfiable :-                              assume(X) :- asserta(X).
        (Head <- Body),                     assume(X) :- retract(X), !, fail.
        Body, not Head, !,
        component(HLit, Head),              component(E, (E ; _)).
        assume(HLit),                       component(E, (_ ; R)) :-
        not false,                                !, component(E, R).
        satisfiable.                        component(E, E).
satisfiable.
```

Similar to hyperresolution + splitting and hyper tableaux (but there are differences).

Termination guaranteed for function-free clause sets (“Datalog”).

## Example - Group Theory

The following axioms specify a group

$$\begin{array}{llcll}
\forall x, y, z & : & (x * y) * z & = & x * (y * z) & \text{(associativity)} \\
\forall x & : & e * x & = & x & (\text{left} - \text{identity}) \\
\forall x & : & i(x) * x & = & e & (\text{left} - \text{inverse})
\end{array}$$

Does

$$\forall x, y \quad : \quad x * y \quad = \quad y * x \qquad \text{(commutat.)}$$

follow?

## Example - Group Theory

The following axioms specify a group

$$\begin{array}{llcll}
\forall x,y,z & : & (x*y)*z & = & x*(y*z) & \text{(associativity)} \\
\forall x & : & e*x & = & x & \text{(left} - \text{identity)} \\
\forall x & : & i(x)*x & = & e & \text{(left} - \text{inverse)}
\end{array}$$

Does

$$\forall x,y \quad : \quad x*y \quad = \quad y*x \qquad \text{(commutat.)}$$

follow?

**No, it does not**

## Example - Group Theory

Counterexample: a group with finite domain of size 6, where the elements 2 and 3 are not commutative: Domain: $\{1, 2, 3, 4, 5, 6\}$

$e : 1$

$$i : \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 2 & 3 & 5 & 4 & 6 \end{array}$$

$$* : \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 2 & 1 & 4 & 3 & 6 & 5 \\ 3 & 3 & 5 & 1 & 6 & 2 & 4 \\ 4 & 4 & 6 & 2 & 5 & 1 & 3 \\ 5 & 5 & 3 & 6 & 1 & 4 & 2 \\ 6 & 6 & 4 & 5 & 2 & 3 & 1 \end{array}$$

## Finite Model Finding

Def: A formula $F$ has the **finite model property** iff $F$ has a model with a finite domain. (The finite model property is undecidable.)

Question here: how to compute ("efficiently") finite models?

Today's finite model finders all follow a generate-and-test approach:

- Given a formula $F$ in clause normal form.
- For each **domain size** $n = 1, 2, \ldots$ transform $F$ into a clause set $G(F, n)$ such that $G(F, n)$ is satisfiable iff $F$ is satisfiable with the domain $D = \{1, 2, \ldots, n\}$

  For each $n$, use a theorem prover to determine if $G(F, n)$ is satisfiable.

  If so, stop and report the model. Otherwise continue.

## Finite Model Finding

Obviously, the theorem prover must be able to report the correct result (satsfiable/unsatisfiable) terminate on $G(F, n)$.

Candidate language fragments for $G(F, n)$:

- Propositional logic: use a SAT solver (the “Maze approach”).
- Function-free clause logic: use an instance-based method.

Alternative: SEM approach, e.g., Slaney’s Finder: works directly on $F$ and systematically checks all interpretations with domain $\{1, 2, \ldots, n\}$ as candidate models.

In the following: Plaisted’s propositional logic encoding.

## Plaisted's Encoding

In the following let $D = \{1, 2, \ldots, n\}$.

For given variables $X = \{x_1, \ldots, x_m\}$ a **domain substitution** $\gamma$ is a substitution of the form

$$\gamma = \{x_1 \mapsto b_1, \ldots, x_m \mapsto b_m\} \qquad \text{where } \{b_1, \ldots, b_m\} \subseteq D$$

### Definition of $G(F, n)$, 5 Steps

Initially, $G(F, n) = \emptyset$

**Step 1:** For each clause $C \in F$ and every domain substitution $\gamma$ for the variables in $C$, add $C\gamma$ to $G(F, n)$.

Example: for $C = P(f(x, y), z) \vee \neg Q(c, g(z))$ and $\gamma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ obtain $P(f(1, 2), 1) \vee \neg Q(c, g(1))$.

Intuitively, variables in clauses are universally quantified, so we need to consider all assignments over the domain. Because the domain is represented as constants in the logic itself, we need to build all ground instances.

## Plaisted's Encoding

We need to say that each function symbol $f$ is mapped to a total function with range $D = \{1, 2, \ldots, n\}$. Let $\approx$ be a new predicate symbol ("equality"), written infix.

For each $k$-ary function symbol $f$ that occurs in $F$ let

$$T_f = f(x_1, \ldots, x_k) \approx 1 \vee \cdots \vee f(x_1, \ldots, x_k) \approx n$$

be the **totality axiom (for $f$)**.

**Step 2:** For each totality axiom $T_f$ and every domain substitution $\gamma$ for its variables add $T_f\gamma$ to $G(F, n)$.

Example: Let $n = 2$ and $T_f = f(x, y) \approx 1 \vee f(x, y) \approx 2$.
Then $G(F, 2)$ contains

$$\begin{array}{ll} f(1,1) \approx 1 \vee f(1,1) \approx 2 & \quad f(2,1) \approx 1 \vee f(2,1) \approx 2 \\ f(1,2) \approx 1 \vee f(1,2) \approx 2 & \quad f(2,2) \approx 1 \vee f(2,2) \approx 2 \end{array}$$

## Plaisted's Encoding

Replacing "equals by equals" preserves the truth value of atoms:

For each (ground) atomic formula $A$ of the form $P(t_1, \dots, t_k)$ that occurs in $G(F, n)$ let

$$S_A = t_1 \approx x_1 \wedge \cdots \wedge t_k \approx x_k \rightarrow (P(t_1, \dots, t_k) \leftrightarrow P(x_1, \dots, x_k))$$

be the **substitution axiom (for $A$)**.

**Step 3:** For each atom $A$ that occurs in $G(F, n)$ and every domain substitution $\gamma$ for $S_A$ add the clausal form of $S_A\gamma$ to $G(F, n)$.

Example: Let $n = 2$, $A = P(f(1, 2), 1)$ and
$S_A = f(1, 2) \approx x_1 \wedge 1 \approx x_2 \rightarrow (P(f(1, 2), 1) \leftrightarrow P(x_1, x_2))$.
Then $G(F, 2)$ contains, among others (take $\gamma = \{x_1 \mapsto 1, x_2 \mapsto 2\}$):

$$\begin{array}{lrl} f(1,2) \approx 1 \ \wedge 1 \approx 2 \wedge & P(f(1,2),1) & \rightarrow P(1,2) \\ f(1,2) \approx 1 \ \wedge 1 \approx 2 \wedge & P(1,2) & \rightarrow P(f(1,2),1) \end{array}$$

## Plaisted's Encoding

Replacing "equals by equals" preserves the value of function applications:

For each (ground) term $t$ of the form $f(t_1, \ldots, t_k)$ that occurs in $G(F, n)$ let

$$S_t = f(x_1, \ldots, x_k) \approx x \wedge t_1 \approx x_1 \wedge \cdots \wedge t_k \approx x_k \to f(t_1, \ldots, t_k) \approx x$$

be the **substitution axiom (for $t$)**.

**Step 4:** For each term $t$ that occurs in $G(F, n)$ and every domain substitution $\gamma$ for $S_t$ add $S_t\gamma$ to $G(F, n)$.

Example: Let $n = 2$ and $t = f(a)$.
Then $G(F, 2)$ contains

$$f(1) \approx 1 \wedge a \approx 1 \to f(a) \approx 1 \qquad f(1) \approx 2 \wedge a \approx 1 \to f(a) \approx 2$$

$$f(2) \approx 1 \wedge a \approx 2 \to f(a) \approx 1 \qquad f(2) \approx 2 \wedge a \approx 2 \to f(a) \approx 2$$

## Plaisted's Encoding

It remains to define equality.

**Step 5:**

For each $b \in \{1, 2, \ldots n\}$ add $b \approx b$ to $G(F, n)$.

For each $b_1, b_2 \in \{1, 2, \ldots n\}$ with $b1 \neq b2$ add $\neg(b_1 \approx b_2)$ to $G(F, n)$.

Example: Let $n = 2$.
Then $G(F, 2)$ contains $1 \approx 1$, $2 \approx 2$, $\neg(1 \approx 2)$, $\neg(2 \approx 1)$.

(End of Plaisted's encoding)

- It can be shown that Plaisted's encoding is correct. Notice that for every equation $t \approx c$ in the encoding, $c$ is always a domain element.
- Complexity: $O(n^k)$ ground clauses, where $k$ is maximal arity of a symbol in $F$.
- The perhaps most advanced propositional encoding is the one of the Paradox model finder. Equality may occur in the input formula. See next slide for an example.

## Paradox - Example

| | |
|---|---|
| Domain: | $\{1, 2\}$ |
| Clauses: | $\{p(a) \vee f(x) = a\}$ |
| Flattened: | $p(y) \vee f(x) = y \vee a \neq y$ |
| Instances: | $p(1) \vee f(1) = 1 \vee a \neq 1$ |
| | $p(2) \vee f(1) = 1 \vee a \neq 2$ |
| | $p(1) \vee f(2) = 1 \vee a \neq 1$ |
| | $p(2) \vee f(2) = 1 \vee a \neq 2$ |
| Totality: | $a = 1 \vee a = 2$ |
| | $f(1) = 1 \vee f(1) = 2$ |
| | $f(2) = 1 \vee f(2) = 2$ |
| Functionality: | $a \neq 1 \vee a \neq 2$ |
| | $f(1) \neq 1 \vee f(1) \neq 2$ |
| | $f(2) \neq 1 \vee f(2) \neq 2$ |

A model is obtained by setting the blue literals true

## Difficult Example

- Consider the clause set consisting of the $n \cdot (n-1)/2 + 1$ unit clauses:

$$P(c_1, \ldots, c_n)$$
$$\neg P(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{j-1}, x, x_{j+1}, \ldots, x_n)$$

- The model must contain (at least) $n$ domain elements.
- Example for $n = 3$:

| *Clauses* | *Model* |
|---|---|
| $p(c_1, c_2, c_3)$ | $c_1 = 1$ |
| $\neg p(x_1, x_1, x_3)$ | $c_2 = 2$ |
| $\neg p(x_1, x_2, x_1)$ | $c_3 = 3$ |
| $\neg p(x_1, x_2, x_2)$ | $p(1, 2, 3)$ |

- Guess: For which $n$ do propositional model finders give up?

## Difficult Example

- Answer: $n = 8$.
- There are $n^{n-1}$ instances of the clause $\neg p(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{j-1}, x, x_{j+1}, \ldots, x_n)$.
- Memory consumption is the main bottleneck.
- Encoding in function-free clause logic requires only quadratically many clauses (savings are on not having to apply the domain substitutions $\gamma$). But instance-based methods are not the solution either. Scalability remains the issue.

## Contents

## Theory Reasoning

Let $T$ be a first-order theory of signature $\Sigma$ and $L$ be a class of $\Sigma$-formulas.

- $T$ can be given as a set of axioms (e.g., the theory of groups), or
- $T$ can be given as a class of interpretations (e.g., the standard model of peano arithmetic)

### The $T$-validity Problem

- Given $\phi$ in $L$, is it the case that $T \models \phi$ ? More accurately:
- Given $\phi$ in $L$, is it the case that $T \models \forall\ \phi$ ?

### Examples

- "$0/0,\ s/1,\ +/2,\ =/2,\ \leq/2$" $\models \exists y.y > x$
- The theory of equality E $\models \phi$ ($\phi$ arbitrary formula)
- "An equational theory" $\models \exists\ s_1 = t_1 \wedge \cdots \wedge s_n = t_n$ (E-Unification problem)
- "Some group theory" $\models s = t$ (Word problem)

The $T$-validity problem is decidable (even semi-decidable) only for restricted $L$ and $T$

## Approaches to Theory Reasoning

### Theory-Reasoning in Automated First-Order Theorem Proving

- Semi-decide the $T$-validity problem, $T \models \phi$ ?
- $\phi$ arbitrary first-order formula, $T$ set of formulas (axioms for $T$)
- Generality is strength and weakness at the same time
- Really successful only for specific instance:
  $T =$ equality, inference rules like paramodulation (see below)

### Satisfiability Modulo Theories (SMT)

- Decide the $T$-validity problem, $T \models \phi$ ?
- Usual restriction: $\phi$ is quantifier-free, i.e. all variables implicitly universally quantified
- Applications in particular to formal verification
  Simple example:
  "arrays+integers" $\models m \geq 0 \land a[i] \geq 0 \land a'[i] = a[i] + m \rightarrow a'[i] \geq 0$

## Equality

Reserve a binary predicate symbol $\approx$ ("equality").

Intuitively, we expect that from the clauses

$$P(a) \qquad a \approx b \qquad b \approx c \qquad f(x) \approx x \qquad f(x) \approx g(x)$$

it follows, e.g.,

$$P(g(f(c)))$$

This requires to fix the meaning of $\approx$. Two options:

- Semantically: define $\approx \quad = \quad \{(d, d) \mid d \in U\}$
  (Recall that predicate symbols are interpreted as relations, $U$ is the universe)
- Syntactically: add **equality axioms** to the given clause set

The semantic approach cannot be used in conjunction with Herbrand models, but the syntactic approach can.

## Handling Equality Naively - Equality Axioms

Let $F$ be a first-order clause set with equality. The clause set $EqAx(F)$ consists of the clauses

$$x \approx x$$

$$x \approx y \rightarrow y \approx x$$

$$x \approx y \wedge y \approx z \rightarrow x \approx z$$

$$x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n \rightarrow f(x_1, \ldots, x_n) \approx f(y_1, \ldots, y_n)$$

$$x_1 \approx y_1 \wedge \cdots \wedge x_m \approx y_m \wedge P(x_1, \ldots, x_m) \rightarrow P(y_1, \ldots, y_m)$$

for every $n$-ary function symbol $f$ occurring in $F$ and every $m$-ary predicate symbol $P$ occurring in $F$.

$EqAx(F)$ are the axioms of a congruence relation on terms and atoms.

It holds: $F$ is satisfiable, where $\approx$ is defined semantically as in the previous slide, if and only if $F \cup EqAx(\Sigma)$ is satisfiable, where $\approx$ is left undefined.

## Handling Equality Naively - Equality Axioms

By giving the equality axioms explicitly, first-order problems with equality can in principle be solved by a standard resolution prover or instance-based method.

But this is unfortunately not efficient (mainly due to the transitivity and congruence axioms).

Modern systems "build-in" equality by dedicated inference rules, which are (restricted) versions of the **Paramodulation** inference rule.

## Recapitulation: Resolution

Resolution: inference rules:

Ground case: | Non-ground case:

**Resolution:**

$$\frac{D' \vee A \qquad C' \vee \neg A}{D' \vee C'} \qquad \frac{D' \vee A \qquad C' \vee \neg A'}{(D' \vee C')\sigma}$$

where $\sigma = \text{mgu}(A, A')$.

**Factoring:**

$$\frac{C' \vee A \vee A}{C' \vee A} \qquad \frac{C' \vee A \vee A'}{(C' \vee A)\sigma}$$

where $\sigma = \text{mgu}(A, A')$.

## Paramodulation

Ground inference rules:

**Paramodulation:**
$$\frac{D' \vee t \approx t' \qquad C' \vee L[t]}{D' \vee C' \vee L[t']}$$

**Equality Resolution:**
$$\frac{C' \vee s \not\approx s}{C'}$$

In the Paramodulation rule, $L[t]$ means that the literal $L$ contains the term $t$, and $L[t']$ means that one occurrence of $t$ in $L$ has been replaced by $t'$.

## Paramodulation

First-order inference rules:

**Paramodulation:**

$$\frac{D' \lor t \approx t' \qquad C' \lor L[u]}{(D' \lor C' \lor L[t'])\sigma}$$

where $\sigma = \mathrm{mgu}(t, u)$ and
$u$ is not a variable.

**Equality Resolution:**

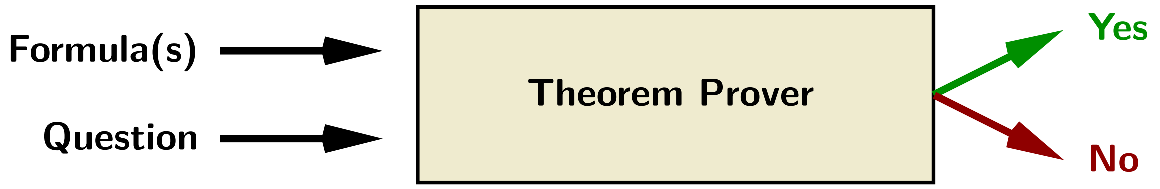
$$\frac{C' \lor s \not\approx s'}{C'\sigma}$$

where $\sigma = \mathrm{mgu}(s, s')$.

These are the main inference rules for equality reasoning. Together with the Resolution and Factoring inference rules, and an additional inference rule (not shown here), one obtains a refutationally complete and sound calculus.

The calculus can still be considerably improved by means of ordering restrictions.

## Satisfiability Modulo Theories (SMT)



Formula: first-order logic formula $\phi$, over equality and other theories

Question: Is $\phi$ valid? (satisfiable? entailed by another formula?)

$$\models_{\mathbb{N}\cup\mathbb{L}} \forall l\ (c = 5 \rightarrow \mathrm{car}(\mathrm{cons}(3 + c, l)) \doteq 8)$$

Theorem Prover: DPLL(T), translation into SAT, first-order provers

**Issue:** essentially undecidable for non-variable free fragment ($\forall$-quantifier left of $\models$):

$$P(0) \wedge (\forall x\ P(x) \rightarrow P(x+1)) \models_{\mathbb{N}} \forall x\ P(x)$$

Design a "good" prover anyways (ongoing research)

## Checking Satisfiability Modulo Theories

**Given:** A quantifier-free formula $\phi$ (implicitly existentially quantified)
**Task:** Decide whether $\phi$ is T-satisfiable
($T$-validity via "$T \models \forall\ \phi$" iff "$\exists\ \neg\phi$ is not $T$-satisfiable")

### Approach: eager translation into SAT

- Encode problem into a $T$-equisatisfiable propositional formula
- Feed formula to a SAT-solver
- Example: $T$ = equality (Ackermann encoding)

### Approach: lazy translation into SAT

- Couple a SAT solver with a given decision procedure for T-satisfiability of ground literals, "DPLL(T)"
- For instance if $T$ is "equality" then the Nelson-Oppen congruence closure method can be used
- If $T$ is "linear arithmetic", a quantifier elimination method (see below)

## Lazy Translation into SAT

$$g(a) = c \quad \wedge \quad f(g(a)) \neq f(c) \vee g(a) = d \quad \wedge \quad c \neq d$$

**Theory: Equality**

## Lazy Translation into SAT

$$\underbrace{g(a) = c}_{1} \quad \wedge \quad \underbrace{f(g(a)) \neq f(c)}_{\overline{2}} \vee \underbrace{g(a) = d}_{3} \quad \wedge \quad \underbrace{c \neq d}_{\overline{4}}$$

## Lazy Translation into SAT

$$\underbrace{g(a) = c}_{1} \quad \wedge \quad \underbrace{f(g(a)) \neq f(c)}_{\overline{2}} \vee \underbrace{g(a) = d}_{3} \quad \wedge \quad \underbrace{c \neq d}_{\overline{4}}$$

- Send $\{1,\ \overline{2} \vee 3,\ \overline{4}\}$ to SAT solver.

## Lazy Translation into SAT

$$\underbrace{g(a) = c}_{1} \quad \wedge \quad \underbrace{f(g(a)) \neq f(c)}_{\overline{2}} \vee \underbrace{g(a) = d}_{3} \quad \wedge \quad \underbrace{c \neq d}_{\overline{4}}$$

- Send $\{1, \overline{2} \vee 3, \overline{4}\}$ to SAT solver.
- SAT solver returns model $\{1, \overline{2}, \overline{4}\}$.
  Theory solver finds $\{1, \overline{2}\}$ $E$-unsatisfiable.

## Lazy Translation into SAT

$$\underbrace{g(a) = c}_{1} \quad \wedge \quad \underbrace{f(g(a)) \neq f(c)}_{\overline{2}} \vee \underbrace{g(a) = d}_{3} \quad \wedge \quad \underbrace{c \neq d}_{\overline{4}}$$

- Send $\{1,\ \overline{2} \vee 3,\ \overline{4}\}$ to SAT solver.
- SAT solver returns model $\{1,\ \overline{2},\ \overline{4}\}$.
  Theory solver finds $\{1,\ \overline{2}\}$ $E$-unsatisfiable.
- Send $\{1,\ \overline{2} \vee 3,\ \overline{4},\ \overline{1} \vee 2\}$ to SAT solver.

## Lazy Translation into SAT

$$\underbrace{g(a) = c}_{1} \quad \wedge \quad \underbrace{f(g(a)) \neq f(c)}_{\overline{2}} \vee \underbrace{g(a) = d}_{3} \quad \wedge \quad \underbrace{c \neq d}_{\overline{4}}$$

- Send $\{1,\ \overline{2} \vee 3,\ \overline{4}\}$ to SAT solver.
- SAT solver returns model $\{1,\ \overline{2},\ \overline{4}\}$.
  Theory solver finds $\{1,\ \overline{2}\}$ $E$-unsatisfiable.
- Send $\{1,\ \overline{2} \vee 3,\ \overline{4},\ \overline{1} \vee 2\}$ to SAT solver.
- SAT solver returns model $\{1,\ 2,\ 3,\ \overline{4}\}$.
  Theory solver finds $\{1,\ 3,\ \overline{4}\}$ $E$-unsatisfiable.

## Lazy Translation into SAT

$$\underbrace{g(a) = c}_{1} \quad \wedge \quad \underbrace{f(g(a)) \neq f(c)}_{\overline{2}} \vee \underbrace{g(a) = d}_{3} \quad \wedge \quad \underbrace{c \neq d}_{\overline{4}}$$

- Send $\{1,\ \overline{2} \vee 3,\ \overline{4}\}$ to SAT solver.
- SAT solver returns model $\{1,\ \overline{2},\ \overline{4}\}$.
  Theory solver finds $\{1,\ \overline{2}\}$ $E$-unsatisfiable.
- Send $\{1,\ \overline{2} \vee 3,\ \overline{4},\ \overline{1} \vee 2\}$ to SAT solver.
- SAT solver returns model $\{1,\ 2,\ 3,\ \overline{4}\}$.
  Theory solver finds $\{1,\ 3,\ \overline{4}\}$ $E$-unsatisfiable.
- Send $\{1,\ \overline{2} \vee 3,\ \overline{4},\ \overline{1} \vee 2,\ \overline{1} \vee \overline{3} \vee 4\}$ to SAT solver.
  SAT solver finds $\{1,\ \overline{2} \vee 3,\ \overline{4},\ \overline{1} \vee 2,\ \overline{1} \vee \overline{3} \vee 4\}$ unsatisfiable.

## Lazy Translation into SAT: Summary

- Abstract $T$-atoms as propositional variables
- SAT solver computes a model, i.e. satisfying boolean assignment for propositional abstraction (or fails)
- Solution from SAT solver may not be a $T$-model. If so,
  - Refine (strengthen) propositional formula by incorporating reason for false solution
  - Start again with computing a model

## Optimizations

### Theory Consequences

- The theory solver may return consequences (typically literals) to guide the SAT solver

### Online SAT solving

- The SAT solver continues its search after accepting additional clauses (rather than restarting from scratch)

### Preprocessing atoms

- Atoms are rewritten into normal form, using theory-specific atoms (e.g. associativity, commutativity)

### Several layers of decision procedures

- “Cheaper” ones are applied first

## Example Theory: Linear Arithmetic Semantics

The **$\Sigma_{\mathsf{LA}}$-algebra** (also called $\Sigma_{\mathsf{LA}}$-interpretation or $\Sigma_{\mathsf{LA}}$-structure) is the triple

$$\mathcal{A}_{\mathsf{LA}} = (\mathbb{Q},\ (+_{\mathcal{A}_{\mathsf{LA}}}, -_{\mathcal{A}_{\mathsf{LA}}}, *_{\mathcal{A}_{\mathsf{LA}}}),\ (\leq_{\mathcal{A}_{\mathsf{LA}}}, \geq_{\mathcal{A}_{\mathsf{LA}}}, <_{\mathcal{A}_{\mathsf{LA}}}, >_{\mathcal{A}_{\mathsf{LA}}}))$$

where $+_{\mathcal{A}_{\mathsf{LA}}}, -_{\mathcal{A}_{\mathsf{LA}}}, *_{\mathcal{A}_{\mathsf{LA}}}, \leq_{\mathcal{A}_{\mathsf{LA}}}, \geq_{\mathcal{A}_{\mathsf{LA}}}, <_{\mathcal{A}_{\mathsf{LA}}}, >_{\mathcal{A}_{\mathsf{LA}}}$ are the “standard” intepretations of $+, -, *, \leq, \geq, <, >$, respectively.

## On Quantification

Linear arithmetic can also be considered with respect to quantification. The quantifiers are $\exists$ meaning "there exists" and $\forall$ meaning "for all". For example, $\exists x\,(x \geq 0)$ is valid (or true) in $\mathcal{A}_{\mathrm{LA}}$, $\forall x\,(x \geq 0)$ is unsatisfiable (or false) and $\forall x\,(x \geq 0 \vee x < 0)$ is again valid.

Note that a quantifier free formula is satisfiable iff the existential closure of the formula is valid. If we introduce new free constants $c_i$ for the variables $x_i$ of a quantifier free formula, where $\mathcal{A}_{\mathrm{LA}}(c_i) = q_i$ for some $q_i \in \mathbb{Q}$, then a quantifier free formula is satisfiable iff the same formula where variables are replaced by new free constants is satisfiable.

## Some Important LA Equivalences

The following equivalences are valid for all LA terms $s, t$:

$$\neg s \geq t \leftrightarrow s < t$$

$$\neg s \leq t \leftrightarrow s > t \qquad \text{(Negation)}$$

$$(s = t) \leftrightarrow (s \leq t \wedge s \geq t) \qquad \text{(Equality)}$$

$$s \geq t \leftrightarrow t \leq s$$

$$s > t \leftrightarrow t < s \qquad \text{(Swap)}$$

With $\lesssim$ we abbreviate $<$ or $\leq$.

## The Fourier-Motzkin Procedure

```
boolean FM(Set $N$ of LA atoms) {
    if ($N = \emptyset$) return true;
    elsif ($N$ is ground) return $\mathcal{A}_{\mathrm{LA}}(N)$;
    else {
        select a variable $x$ from $N$;
        transform all atoms in $N$ containing $x$ into $s_i \lesssim x$, $x \lesssim t_j$
        and the subset $N'$ of atoms not containing $x$;
        compute $N^* := \{s_i \lesssim_{i,j} t_j \mid s_i \lesssim_i x \in N,\ x \lesssim_j t_j \in N \text{ for all } i, j\}$
        where $\lesssim_{i,j}$ is strict iff at least one of $\lesssim_i$, $\lesssim_j$ is strict
        return FM($N' \cup N^*$);
    }
}
```

## Properties of the Fourier-Motzkin Procedure

- Any ground set $N$ of linear arithmetic atoms can be easily decided.
- FM($N$) terminates on any $N$ as in recursive calls $N$ has strictly less variables.
- The set $N' \cup N^*$ is worst case of size $O(|N|^2)$.
- FM($N$)=true iff $N$ is satisfiable in $\mathcal{A}_{\mathrm{LA}}$.
- The procedure was invented by Fourier (1826), forgotten, and then rediscovered by Dines (1919) and Motzkin (1936).
- There are more efficient methods known, e.g., the simplex algorithm.
- As said, the Fourier-Motzkin Procedure decides the satisfiability of a set (conjunction) of linear arithmetic atoms, which is what is needed to build a sound and complete DPLL(T)-solver.

## Combining Theories

Theories:

- $\mathcal{R}$: theory of rationals
  $\Sigma_{\mathcal{R}} = \{\leq, +, -, 0, 1\}$
- $\mathcal{L}$: theory of lists
  $\Sigma_{\mathcal{L}} = \{=, \mathrm{hd}, \mathrm{tl}, \mathrm{nil}, \mathrm{cons}\}$
- $\mathcal{E}$: theory of equality
  $\Sigma$: free function and predicate symbols

Problem: Is

$$x \leq y \wedge y \leq x + \mathrm{hd}(\mathrm{cons}(0, \mathrm{nil})) \wedge P(h(x) - h(y)) \wedge \neg P(0)$$

satisfiable in $\mathcal{R} \cup \mathcal{L} \cup \mathcal{E}$?

## Nelson-Oppen Combination Method

G. Nelson and D.C. Oppen: *Simplification by cooperating decision procedures*, ACM Trans. on Programming Languages and Systems, 1(2):245-257, 1979.

Given:

- $\mathcal{T}_1$, $\mathcal{T}_2$ first-order theories with signatures $\Sigma_1$, $\Sigma_2$
- $\Sigma_1 \cap \Sigma_2 = \emptyset$
- $\phi$ quantifier-free formula over $\Sigma_1 \cup \Sigma_2$

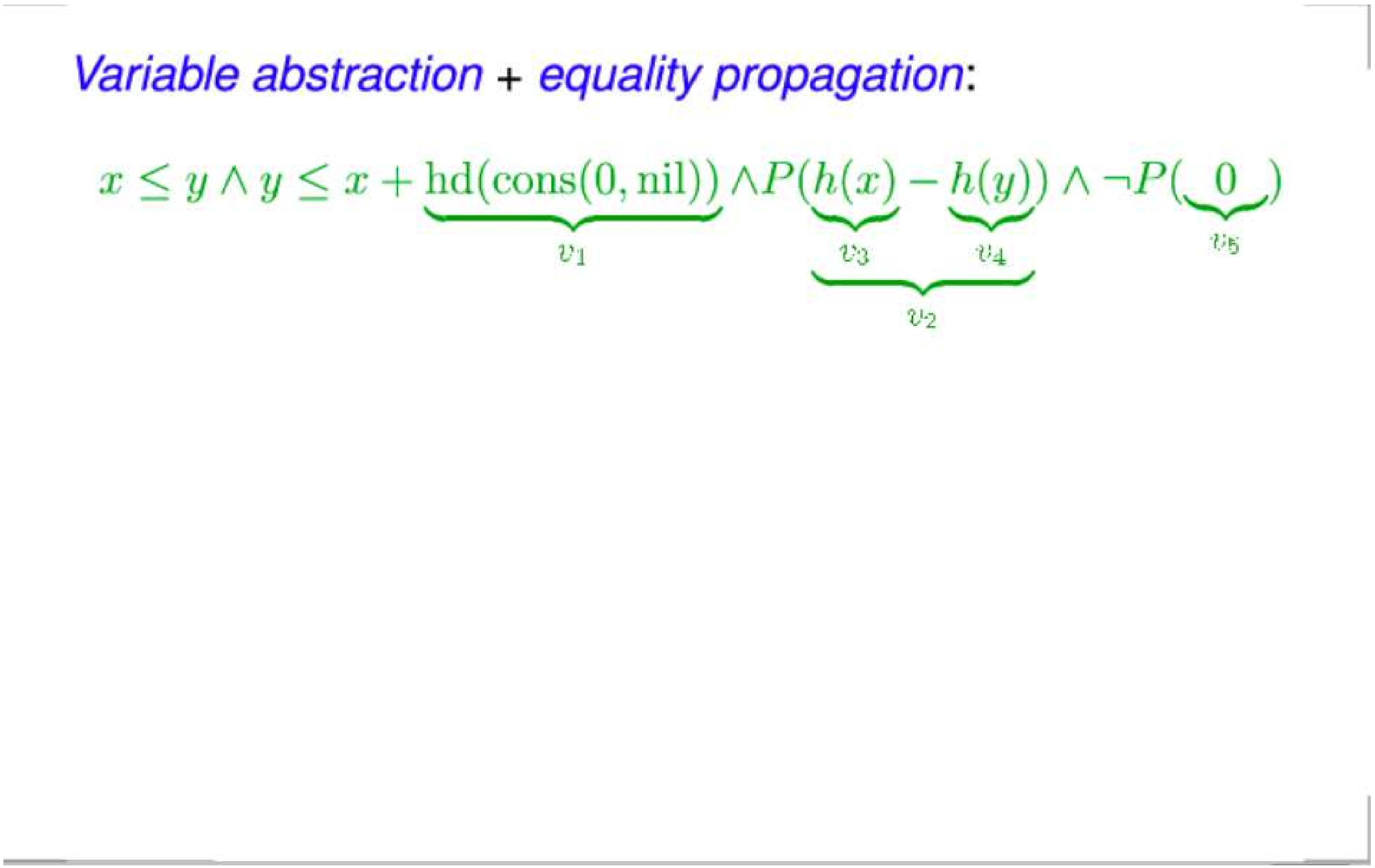
Obtain a decision procedure for satisfiability in $\mathcal{T}_1 \cup \mathcal{T}_2$ from decision procedures for satisfiability in $\mathcal{T}_1$ and $\mathcal{T}_2$.

## Nelson-Oppen Combination Method

*Variable abstraction* + *equality propagation*:

$$x \leq y \land y \leq x + \mathrm{hd}(\mathrm{cons}(0, \mathrm{nil})) \land P(h(x) - h(y)) \land \neg P(0)$$

## Nelson-Oppen Combination Method



*Variable abstraction* + *equality propagation*:

$$x \le y \wedge y \le x + \underbrace{\mathrm{hd}(\mathrm{cons}(0, \mathrm{nil}))}_{v_1} \wedge P(\underbrace{\underbrace{h(x)}_{v_3} - \underbrace{h(y)}_{v_4}}_{v_2}) \wedge \neg P(\underbrace{0}_{v_5})$$

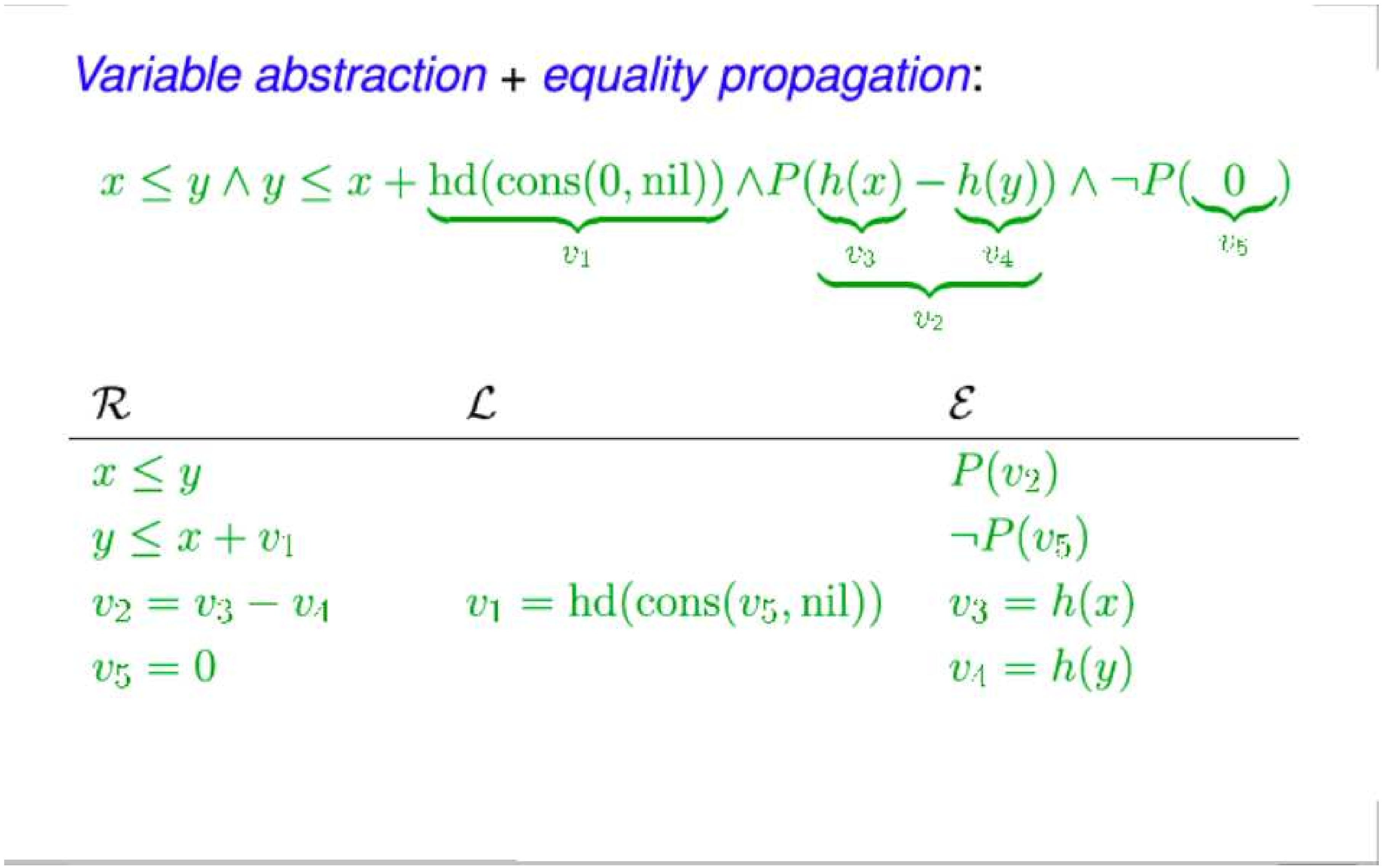
## Nelson-Oppen Combination Method

*Variable abstraction* + *equality propagation*:

$$x \leq y \wedge y \leq x + \underbrace{\mathrm{hd}(\mathrm{cons}(0, \mathrm{nil}))}_{v_1} \wedge P(\underbrace{\underbrace{h(x)}_{v_3} - \underbrace{h(y)}_{v_4}}_{v_2}) \wedge \neg P(\underbrace{0}_{v_5})$$

| $\mathcal{R}$ | $\mathcal{L}$ | $\mathcal{E}$ |
|---|---|---|
| $x \leq y$ | | $P(v_2)$ |
| $y \leq x + v_1$ | | $\neg P(v_5)$ |

# Nelson-Oppen Combination Method

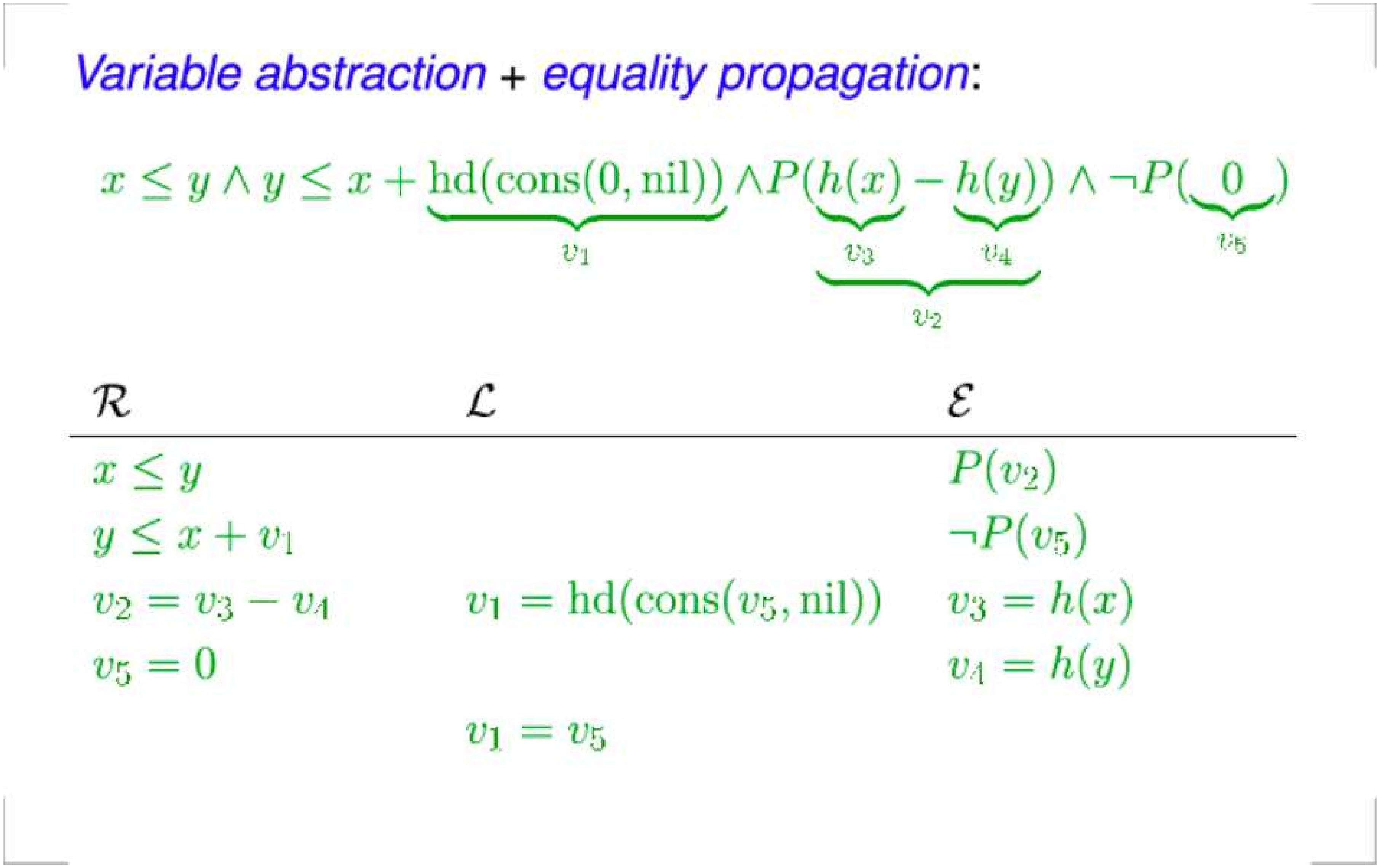


*Variable abstraction* + *equality propagation*:

$$x \le y \wedge y \le x + \underbrace{\mathrm{hd}(\mathrm{cons}(0, \mathrm{nil}))}_{v_1} \wedge P(\underbrace{\underbrace{h(x)}_{v_3} - \underbrace{h(y)}_{v_4}}_{v_2}) \wedge \neg P(\underbrace{0}_{v_5})$$

| $\mathcal{R}$ | $\mathcal{L}$ | $\mathcal{E}$ |
|---|---|---|
| $x \le y$ | | $P(v_2)$ |
| $y \le x + v_1$ | | $\neg P(v_5)$ |
| $v_2 = v_3 - v_4$ | $v_1 = \mathrm{hd}(\mathrm{cons}(v_5, \mathrm{nil}))$ | $v_3 = h(x)$ |
| $v_5 = 0$ | | $v_4 = h(y)$ |

## Nelson-Oppen Combination Method

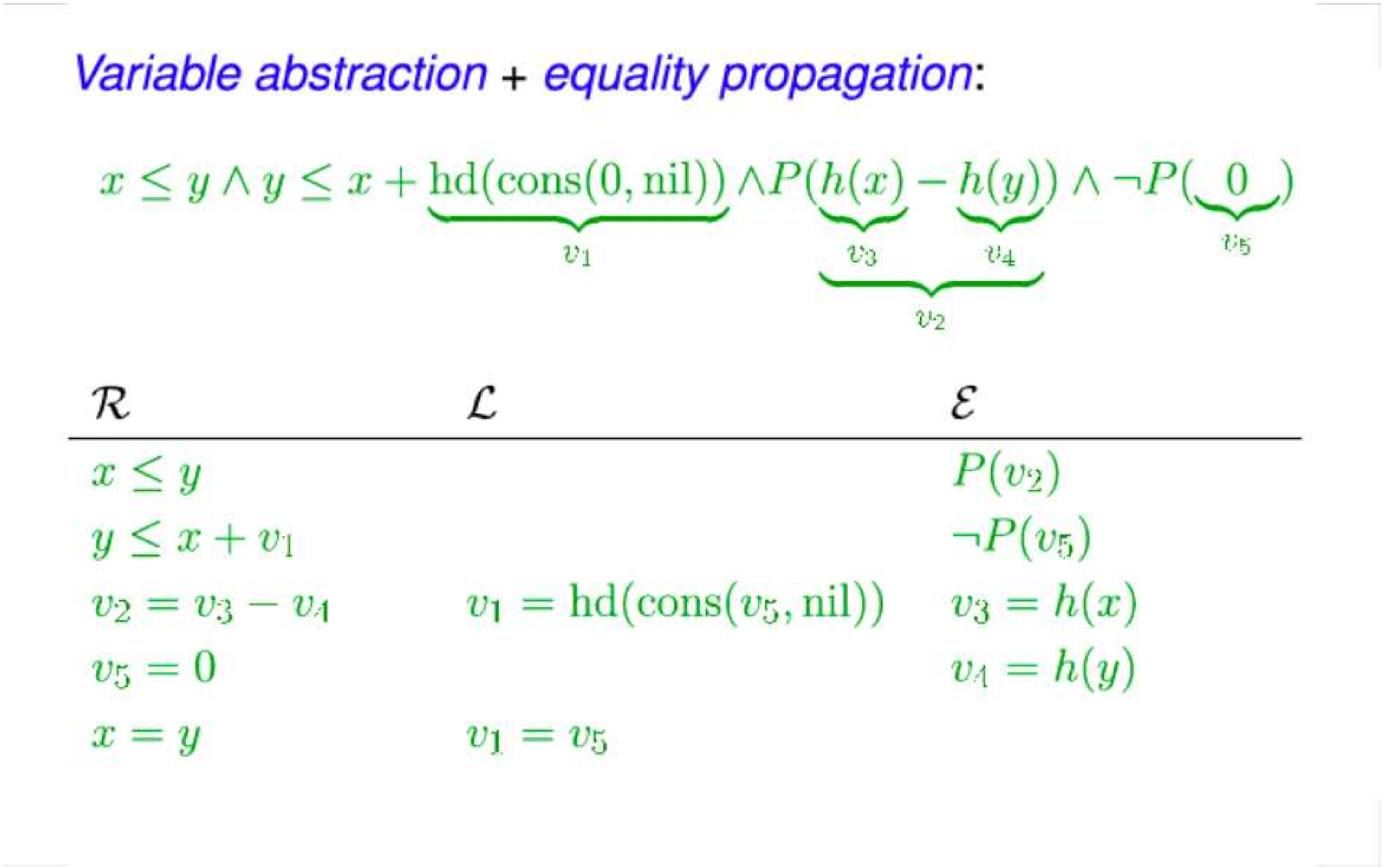


*Variable abstraction* + *equality propagation*:

$$x \le y \wedge y \le x + \underbrace{\mathrm{hd}(\mathrm{cons}(0, \mathrm{nil}))}_{v_1} \wedge P(\underbrace{\underbrace{h(x)}_{v_3} - \underbrace{h(y)}_{v_4}}_{v_2}) \wedge \neg P(\underbrace{0}_{v_5})$$

| $\mathcal{R}$ | $\mathcal{L}$ | $\mathcal{E}$ |
|---|---|---|
| $x \le y$ | | $P(v_2)$ |
| $y \le x + v_1$ | | $\neg P(v_5)$ |
| $v_2 = v_3 - v_4$ | $v_1 = \mathrm{hd}(\mathrm{cons}(v_5, \mathrm{nil}))$ | $v_3 = h(x)$ |
| $v_5 = 0$ | | $v_4 = h(y)$ |
| | $v_1 = v_5$ | |

## Nelson-Oppen Combination Method

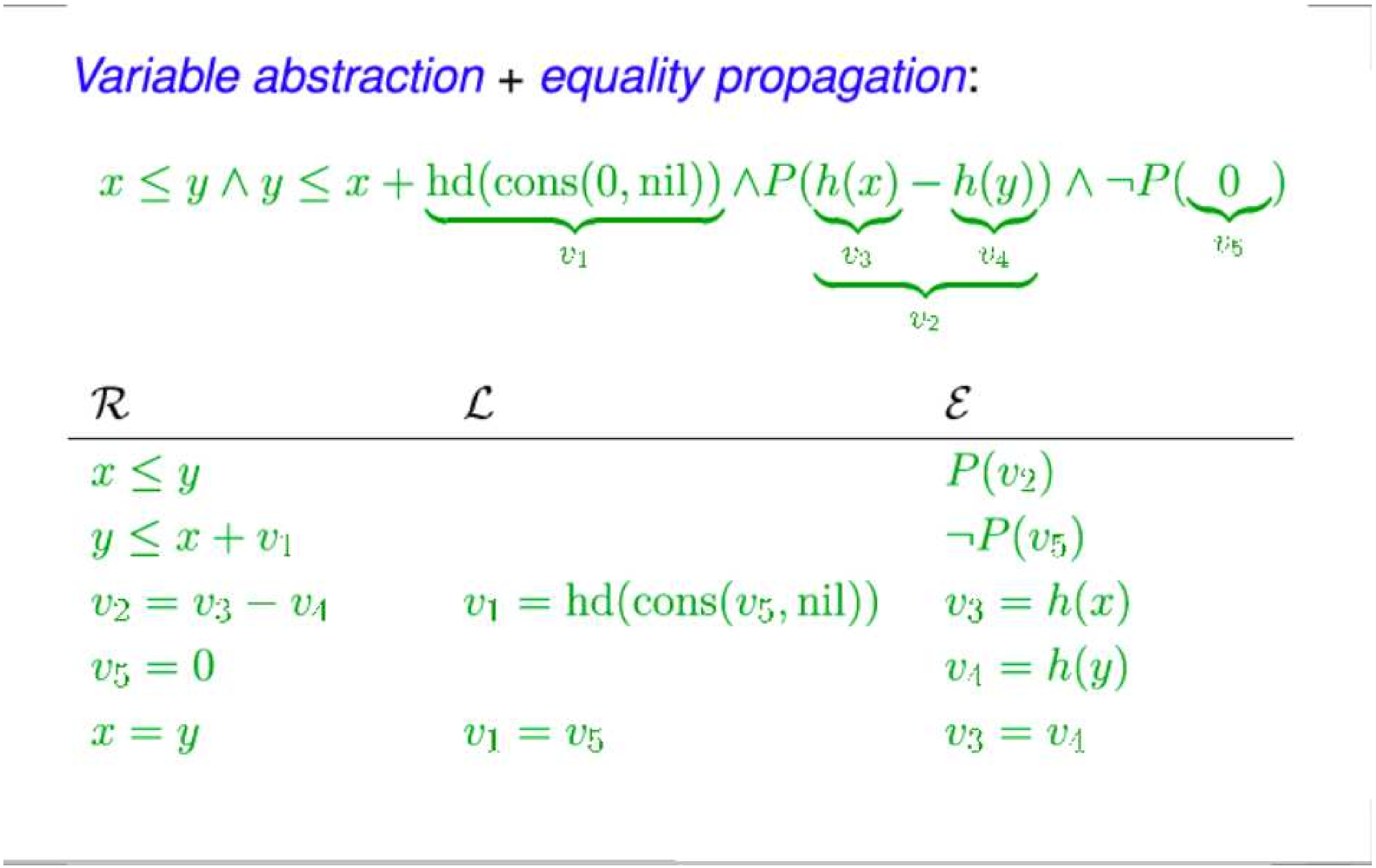


*Variable abstraction* + *equality propagation*:

$$x \le y \wedge y \le x + \underbrace{\mathrm{hd}(\mathrm{cons}(0, \mathrm{nil}))}_{v_1} \wedge P(\underbrace{\underbrace{h(x)}_{v_3} - \underbrace{h(y)}_{v_4}}_{v_2}) \wedge \neg P(\underbrace{0}_{v_5})$$

| $\mathcal{R}$ | $\mathcal{L}$ | $\mathcal{E}$ |
|---|---|---|
| $x \le y$ | | $P(v_2)$ |
| $y \le x + v_1$ | | $\neg P(v_5)$ |
| $v_2 = v_3 - v_4$ | $v_1 = \mathrm{hd}(\mathrm{cons}(v_5, \mathrm{nil}))$ | $v_3 = h(x)$ |
| $v_5 = 0$ | | $v_4 = h(y)$ |
| $x = y$ | $v_1 = v_5$ | |

# Nelson-Oppen Combination Method



*Variable abstraction* + *equality propagation*:

$$x \le y \wedge y \le x + \underbrace{\mathrm{hd}(\mathrm{cons}(0, \mathrm{nil}))}_{v_1} \wedge P(\underbrace{\underbrace{h(x)}_{v_3} - \underbrace{h(y)}_{v_4}}_{v_2}) \wedge \neg P(\underbrace{0}_{v_5})$$

| $\mathcal{R}$ | $\mathcal{L}$ | $\mathcal{E}$ |
|---|---|---|
| $x \le y$ | | $P(v_2)$ |
| $y \le x + v_1$ | | $\neg P(v_5)$ |
| $v_2 = v_3 - v_4$ | $v_1 = \mathrm{hd}(\mathrm{cons}(v_5, \mathrm{nil}))$ | $v_3 = h(x)$ |
| $v_5 = 0$ | | $v_4 = h(y)$ |
| $x = y$ | $v_1 = v_5$ | $v_3 = v_4$ |

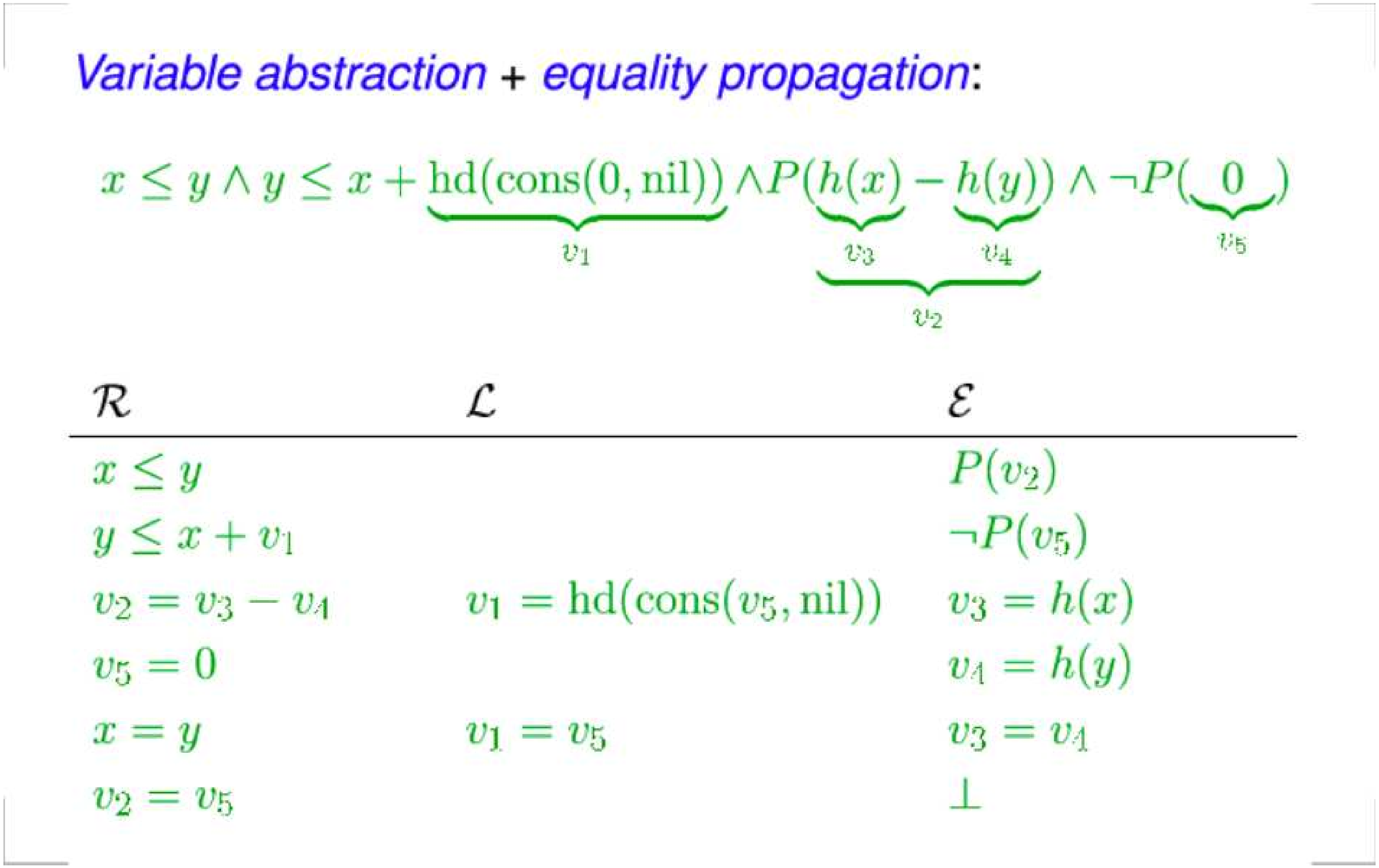
## Nelson-Oppen Combination Method

*Variable abstraction* + *equality propagation*:

$$x \leq y \wedge y \leq x + \underbrace{\mathrm{hd}(\mathrm{cons}(0, \mathrm{nil}))}_{v_1} \wedge P(\underbrace{\underbrace{h(x)}_{v_3} - \underbrace{h(y)}_{v_4}}_{v_2}) \wedge \neg P(\underbrace{0}_{v_5})$$

| $\mathcal{R}$ | $\mathcal{L}$ | $\mathcal{E}$ |
|---|---|---|
| $x \leq y$ | | $P(v_2)$ |
| $y \leq x + v_1$ | | $\neg P(v_5)$ |
| $v_2 = v_3 - v_4$ | $v_1 = \mathrm{hd}(\mathrm{cons}(v_5, \mathrm{nil}))$ | $v_3 = h(x)$ |
| $v_5 = 0$ | | $v_4 = h(y)$ |
| $x = y$ | $v_1 = v_5$ | $v_3 = v_4$ |
| $v_2 = v_5$ | | |

## Nelson-Oppen Combination Method



*Variable abstraction* + *equality propagation*:

$$x \leq y \wedge y \leq x + \underbrace{\mathrm{hd}(\mathrm{cons}(0, \mathrm{nil}))}_{v_1} \wedge P(\underbrace{\underbrace{h(x)}_{v_3} - \underbrace{h(y)}_{v_4}}_{v_2}) \wedge \neg P(\underbrace{\ 0\ }_{v_5})$$

| $\mathcal{R}$ | $\mathcal{L}$ | $\mathcal{E}$ |
|---|---|---|
| $x \leq y$ | | $P(v_2)$ |
| $y \leq x + v_1$ | | $\neg P(v_5)$ |
| $v_2 = v_3 - v_4$ | $v_1 = \mathrm{hd}(\mathrm{cons}(v_5, \mathrm{nil}))$ | $v_3 = h(x)$ |
| $v_5 = 0$ | | $v_4 = h(y)$ |
| $x = y$ | $v_1 = v_5$ | $v_3 = v_4$ |
| $v_2 = v_5$ | | $\perp$ |

## Conclusions

- Talked about the role of first-order theorem proving
- Talked about some standard techniques (Normal forms of formulas, Resolution calculus, unification, Instance-based method, Model computation)
- Talked about DPLL and Satisfiability Modulo Theories (SMT)

### Further Topics

- Redundancy elimination, efficient equality reasoning, adding arithmetics to first-order theorem provers
- FOTP methods as decision procedures in special cases
  E.g. reducing planning problems and temporal logic model checking problems to function-free clause logic and using an instance-based method as a decision procedure
- Implementation techniques
- Competition CASC and TPTP problem library
- Instance-based methods (a lot to do here, cf. my home page)
  Attractive because of complementary features to more established methods

## Further Reading

- Wikipedia article on **Automated Theorem Proving**
  `en.wikipedia.org/wiki/Automated_theorem_proving`
- Wikipedia article on **Boolean Satisfiability Problem** (propositional logic)
  `en.wikipedia.org/wiki/Boolean_satisfiability_problem`
- Wikipedia article on **Satisfiability Modulo Theories (SMT)**
  `en.wikipedia.org/wiki/Satisfiability_Modulo_Theories`
- A good, recent textbook with an emphasis on theory reasoning (arithmetic, arrays) for software verification:

  Aaron Bradley and Zohar Manna, The Calculus of Computation, Springer, 2007
- Another good one, on what the title says, comes with OCaml code:

  Handbook of Practical Logic and Automated Reasoning, Cambridge University Press, 2009

## Implemented Systems

- The TPTP (Thousands of Problems for Theorem Provers) is a library of test problems for automated theorem proving
  `www.tptp.org`
- The automated theorem prover **SPASS** is an implementation of the "modern" version of resolution with equality, the superposition calculus, and comes with a comprehensive set of examples and documentation. A good choice to start with.
  `www.spass-prover.org`
- `users.rsise.anu.edu.au/~baumgart/systems/`