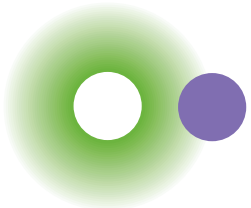
Automated Theorem Proving

Peter Baumgartner



Peter.Baumgartner@nicta.com.au

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

Slides partially based on material by Alexander Fuchs, Harald Ganzinger, Michael Norrish, John Slaney, Viorica Sofronie-Stockermans and Uwe Waldmann

Contents

Part 1: What is Automated Theorem Proving?

What is (Automated) Theorem Proving?

- An application-oriented subfield of logic in computer science and artificial intelligence
- About algorithms and their implementation on computer for reasoning with mathematical logic formulas
- Considers a variety of logics and reasoning tasks
- Applications in logics in computer science
  Program verification, dynamic properties of reactive systems, databases
- Applications in logic-based artificial intelligence
  Mathematical theorem proving, planning, diagnosis, knowledge representation (description logics), logic programming, constraint solving

Theorem Proving in Relation to ...

. . . Calculation: Compute function value at given point:

Problem: $2^2 = ?$ $3^2 = ?$ $4^2 = ?$

"Easy" (often polynomial)

. . . Constraint Solving: Given:

- Problem: $x^2 = a$ where $x \in [1 \ldots b]$
  (x variable, a, b parameters)
- Instance: $a = 16,\ b = 10$

Find values for variables such that problem instance is satisfied

"Difficult" (often exponential, but restriction to **finite** domains)

First-Order Theorem Proving: Given:

Problem: $\exists x\ (x^2 = a \wedge x \in [1 \ldots b])$

Is it satisfiable? unsatisfiable? valid? $\leadsto$ **Automated Logical Analysis!**

"Very difficult" (often undecidable)

Logical Analysis Example: N-Queens

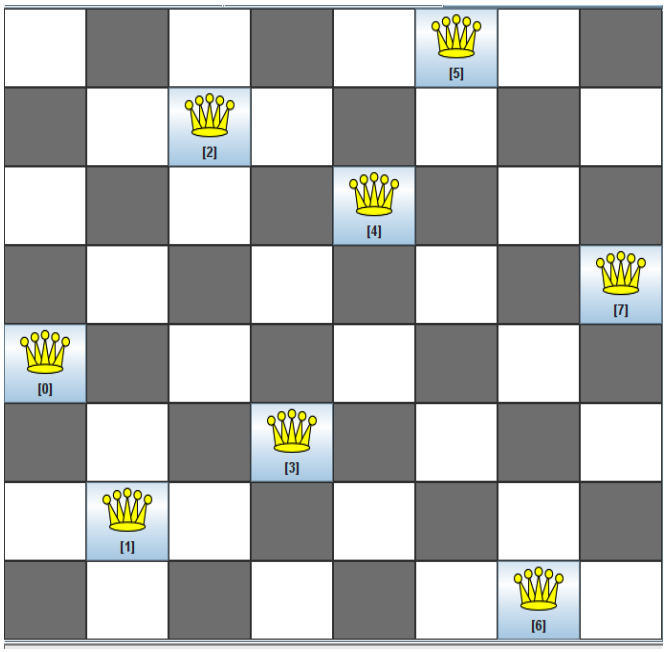
The n-queens problem:

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 e.g. a **constraint solver** to find a solution

Computing Solutions with a Constraint Solver

A **Zinc** model, ready to be "run":

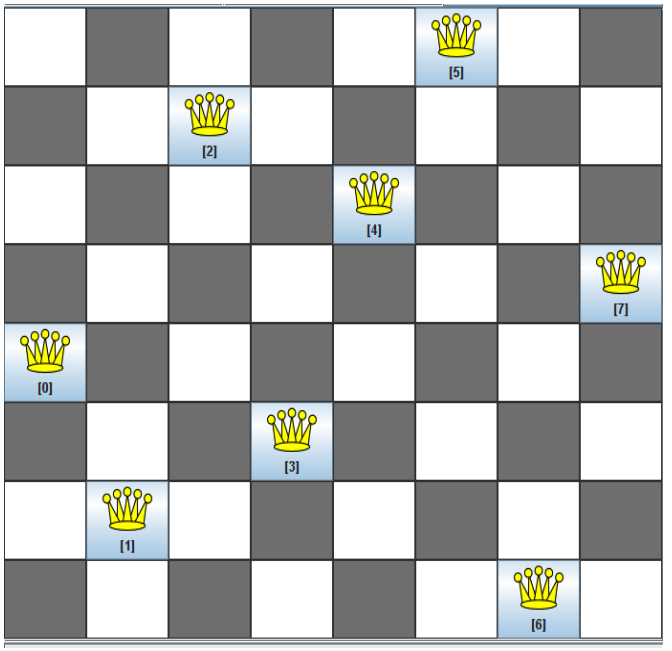
```
int: n = 8;
array [1..n] of var 1..n: p;

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"];
```

But, as said, constraint solving is not theorem proving.

What's the rôle of theorem proving here?

Logical Analysis Example: N-Queens

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



Number of solutions, depending on n:

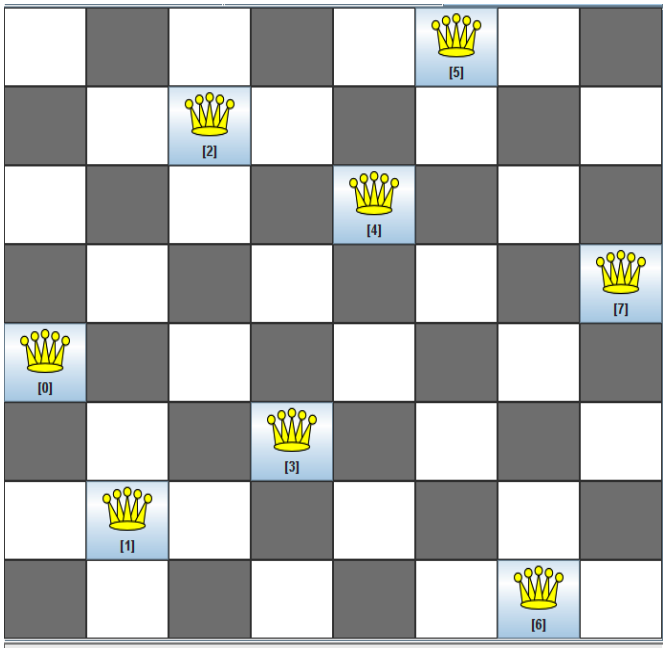
n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	..	24	25
unique:	1	0	0	1	2	1	6	12	46	92	341	1,787	9,233	45,752	..	28,439,272,956,934	275,986,683,743,434
distinct:	1	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	..	227,514,171,973,736	2,207,893,435,808,352

"unique" is **"distinct"** modulo reflection/rotation symmetry

For efficiency reasons better avoid searching symmetric solutions

Logical Analysis Example: N-Queens

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



- The n-queens has variable symmetry: mapping $p[i] \mapsto p[n+1-i]$ preserves solutions
- Therefore, it is justified to add (to the Zinc model) a constraint $p[1] < p[n]$, for search space pruning
- But how can one know, in general, that a problem has symmetries? Use a theorem prover!

Part 2: Methods for Automated Theorem Proving

How to Build a (First-Order) Theorem Prover

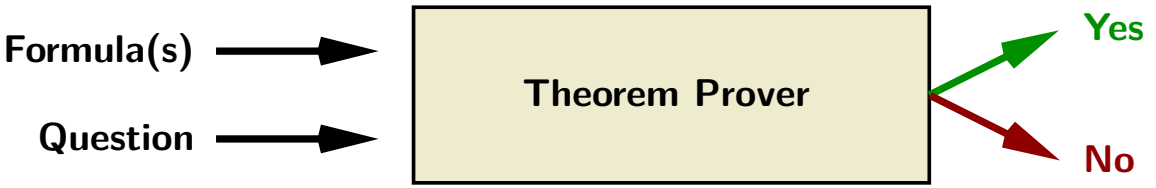
1. Fix an **input language** for formulas
2. Fix a **semantics** to define what the formulas mean
   Will be always "classical" here
3. Determine the desired **services** from the theorem prover
   (The questions we would like the prover be able to answer)
4. Design a **calculus** for the logic and the services
   Calculus: high-level description of the "logical analysis" algorithm
   This includes redundancy criteria for formulas and inferences
5. Prove the calculus is **correct** (sound and complete) wrt. the logic and the services, if possible
6. Design a **proof procedure** for the calculus
7. Implement the proof procedure (research topic of its own)

Go through the red issues in the rest of this part 2

How to Build a (First-Order) Theorem Prover

1. Fix an **input language** for formulas
2. Fix a **semantics** to define what the formulas mean
   Will be always "classical" here
3. Determine the desired **services** from the theorem prover
   (The questions we would like the prover be able to answer)
4. Design a **calculus** for the logic and the services
   Calculus: high-level description of the "logical analysis" algorithm
   This includes redundancy criteria for formulas and inferences
5. Prove the calculus is **correct** (sound and complete) wrt. the logic and the services, if possible
6. Design a **proof procedure** for the calculus
7. Implement the proof procedure (research topic of its own)

Languages and Services — Propositional SAT



Formula: Propositional logic formula ϕ

Question: Is ϕ satisfiable?

(Minimal model? Maximal consistent subsets?)

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

Issue: the formula ϕ can be **BIG**

DPLL as a Semantic Tree Method

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

⟨empty tree⟩

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

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

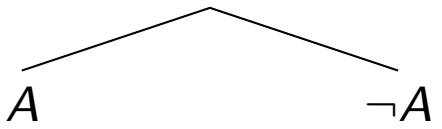
$\{\} \models D \vee \neg C \vee \neg A$

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

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



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

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

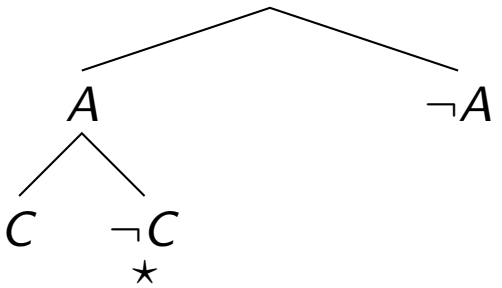
$\{A\} \models D \vee \neg C \vee \neg A$

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

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



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

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

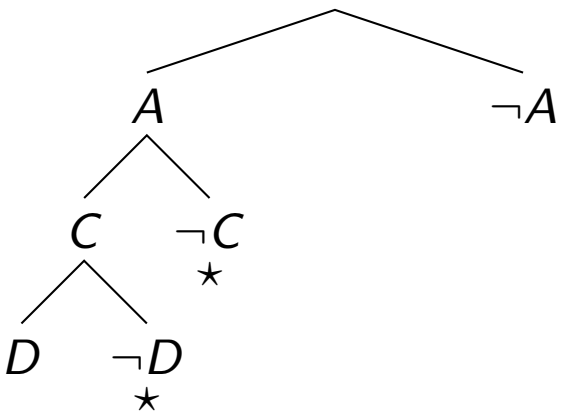
$\{A, C\} \not\models D \vee \neg C \vee \neg A$

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

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



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

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

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

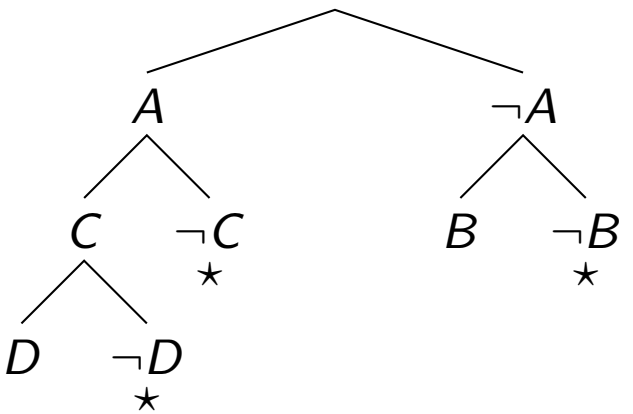
$\{A, C, D\} \models \neg D \vee \neg B$

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 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(φ)   %% φ is a set of clauses, connected by "and"
  while φ contains a unit clause {L}
    φ := simplify(φ, L);
  if φ = {} then return true;
  if {} ∈ φ then return false;
  L := choose-literal(φ);
  if DPLL(simplify(φ, L)) then return true;
  else return DPLL(simplify(φ, ¬L));

function simplify(φ, L)
  remove all clauses from φ that contain L;
  delete ¬L from all remaining clauses;
  return the resulting clause set;
```

Lemma Learning in DPLL

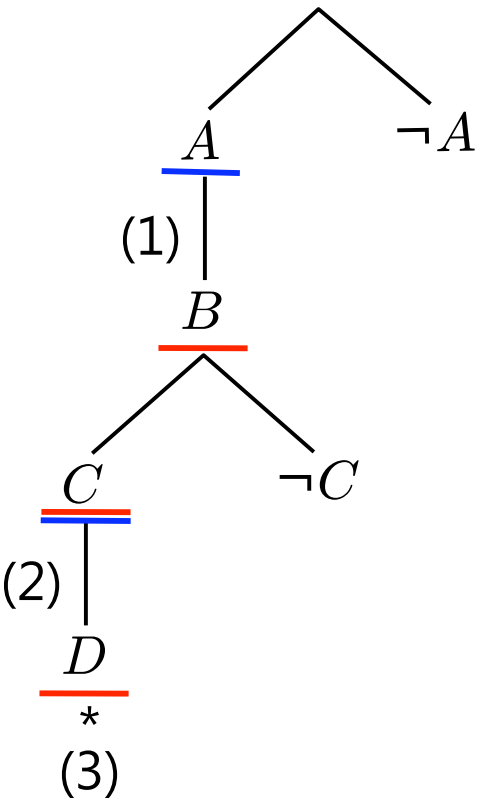
"Avoid making the same mistake twice"

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

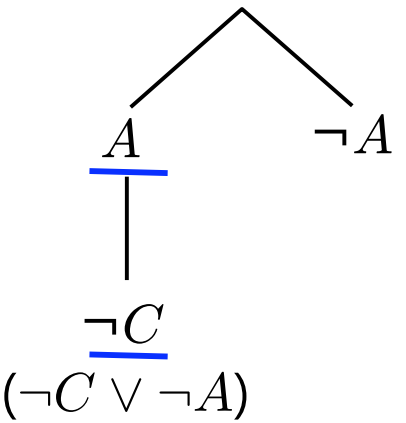
Lemma Candidates by Resolution:

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

w/o Lemma



With Lemma



Making DPLL Fast

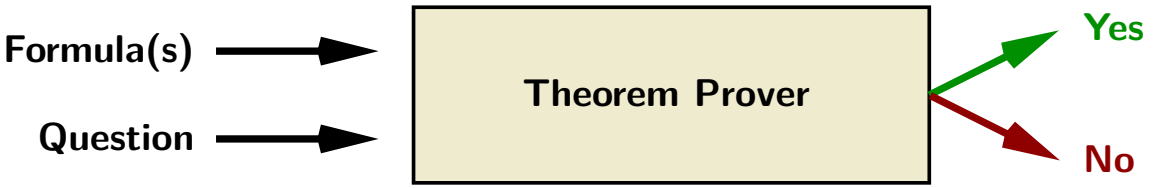
Key ingredients

- Lemma learning
- **plus** (randomized) restarts
- Variable selection heuristics (what literal to split on)
- Make unit-propagation fast (2-watched literal technique)

N.B: modern SAT solvers don't do "split"

- "left split" literal A is marked as a "decision literal" instead
- "right split" literal $\neg A$ can be obtained by unit-propagation into a learned clause $\{\ldots, \neg A\}$

Languages and Services — Description Logics



Formula: Description Logic TBox + ABox (restricted FOL)

TBox: Terminology

ABox: Assertions

$\text{Professor} \sqcap \exists \text{ supervises} \,.\, \text{Student} \sqsubseteq \text{BusyPerson}$

$p : \text{Professor} \quad (p, s) : \text{supervises}$

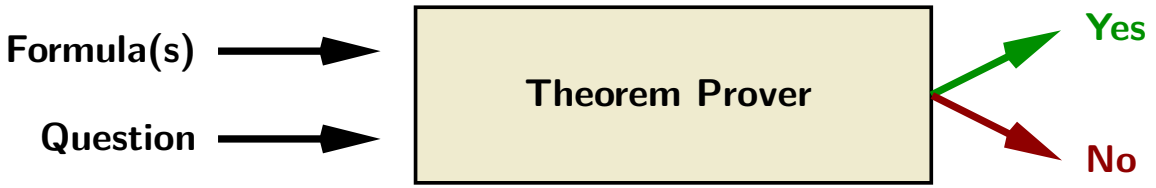
Question: Is TBox + ABox satisfiable?

(Does C subsume D?, Concept hierarchy?)

Theorem Prover: Tableaux algorithms (predominantly)

Issue: Push expressivity of DLs while preserving decidability

Languages and Services — Satisfiability Modulo Theories (SM



Formula: Usually **variable-free** first-order logic formula ϕ

Equality $\doteq$, combination of theories, free symbols

Question: Is ϕ valid? (satisfiable? entailed by another formula?)

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

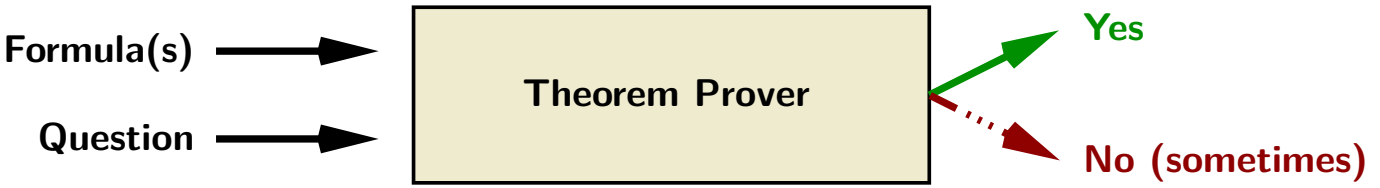
Theorem Prover: DPLL(T), translation into SAT, first-order provers

Issue: essentially undecidable for non-variable free fragment

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

Design a "good" prover anyways (ongoing research)

Languages and Services — "Full" First-Order Logic



Formula: First-order logic formula ϕ (e.g. the three-coloring spec above)
Usually with equality $\doteq$

Question: Is ϕ formula valid? (satisfiable?, entailed by another formula?)

Theorem Prover: Superposition (Resolution), Instance-based methods

Issues

- Efficient treatment of equality
- Decision procedure for sub-languages or useful reductions?
  Can do e.g. DL reasoning? Model checking? Logic programming?
- Built-in inference rules for arrays, lists, arithmetics (still open research)

How to Build a (First-Order) Theorem Prover

1. Fix an **input language** for formulas
2. Fix a **semantics** to define what the formulas mean
   Will be always "classical" here
3. Determine the desired **services** from the theorem prover
   (The questions we would like the prover be able to answer)
4. Design a **calculus** for the logic and the services
   Calculus: high-level description of the "logical analysis" algorithm
   This includes redundancy criteria for formulas and inferences
5. Prove the calculus is **correct** (sound and complete) wrt. the logic and the services, if possible
6. Design a **proof procedure** for the calculus
7. Implement the proof procedure (research topic of its own)

Semantics

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

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

Underlying Language

Variables ε, a, δ, 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 $\wedge$, $\vee$, $\rightarrow$, $\neg$

Quantifiers $\forall$, $\exists$

The function symbols and predicate symbols comprise a signature Σ

Semantics

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

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

"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 - Σ-Algebra Example

Let Σ_{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** Σ_{PA}-interpretations

Semantics - Σ-Algebra Example

Evaluation of terms and formulas

Under the interpretation $\mathbb{N}$ and the assignment $\beta : x \mapsto 1, y \mapsto 3$ 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 ϕ 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. $\mathbb{N} \models \forall x \exists y\ x < y$

Standard reasoning services can now be expressed semantically

Services Semantically

E.g. "entailment":

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

Services

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

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

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

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

Solve(I,ϕ): find an assignment β such that $I, \beta \models \phi$

Solve(ϕ): find an interpretation and assignment β such that $I, \beta \models \phi$

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

What if theorem prover's native service is only "Is ϕ unsatisfiable?" ?

Semantics - 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

How to Build a (First-Order) Theorem Prover

1. Fix an **input language** for formulas
2. Fix a **semantics** to define what the formulas mean
   Will be always "classical" here
3. Determine the desired **services** from the theorem prover
   (The questions we would like the prover be able to answer)
4. Design a **calculus** for the logic and the services
   Calculus: high-level description of the "logical analysis" algorithm
   This includes redundancy criteria for formulas and inferences
5. Prove the calculus is **correct** (sound and complete) wrt. the logic and the services, if possible
6. Design a **proof procedure** for the calculus
7. Implement the proof procedure (research topic of its own)

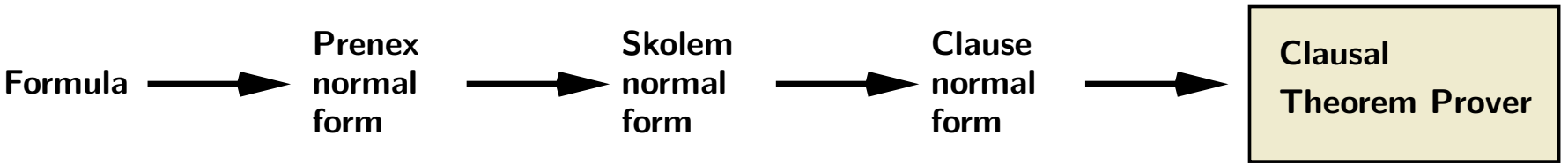
Calculus - Normal Forms

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



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 $Q_i \in \{\forall, \exists\}$

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

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

$$\Rightarrow_P$$

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

$$\Rightarrow_P$$

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

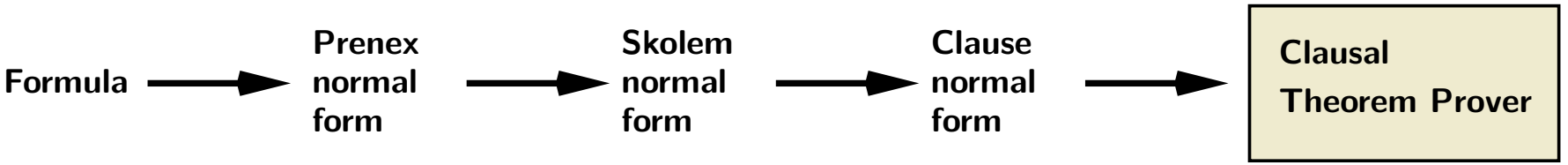
$$\Rightarrow_P$$

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

$$\Rightarrow_P$$

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

Skolem Normal Form



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 associativity and commutativity of $\wedge$ and $\vee$

In the Example

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

$$\Rightarrow_K$$

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

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

The Complete Picture

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

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

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

$N = \{C_1, \ldots, C_k\}$ 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

Herbrand Interpretations

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

- The universe is the set T_Σ of ground terms over Σ
  (a **ground term** is a term without any variables):

$$U_\mathcal{A} = T_\Sigma$$

- Every function symbol from Σ 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 Σ_{Pres}

$$I = \{\quad 0 \leq 0,\ 0 \leq s(0),\ 0 \leq s(s(0)),\ \ldots,$$
$$0 + 0 \leq 0,\ 0 + 0 \leq s(0),\ \ldots,$$
$$\ldots,\ (s(0) + 0) + s(0) \leq s(0) + (s(0) + s(0)), \ldots \quad \}$$

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:

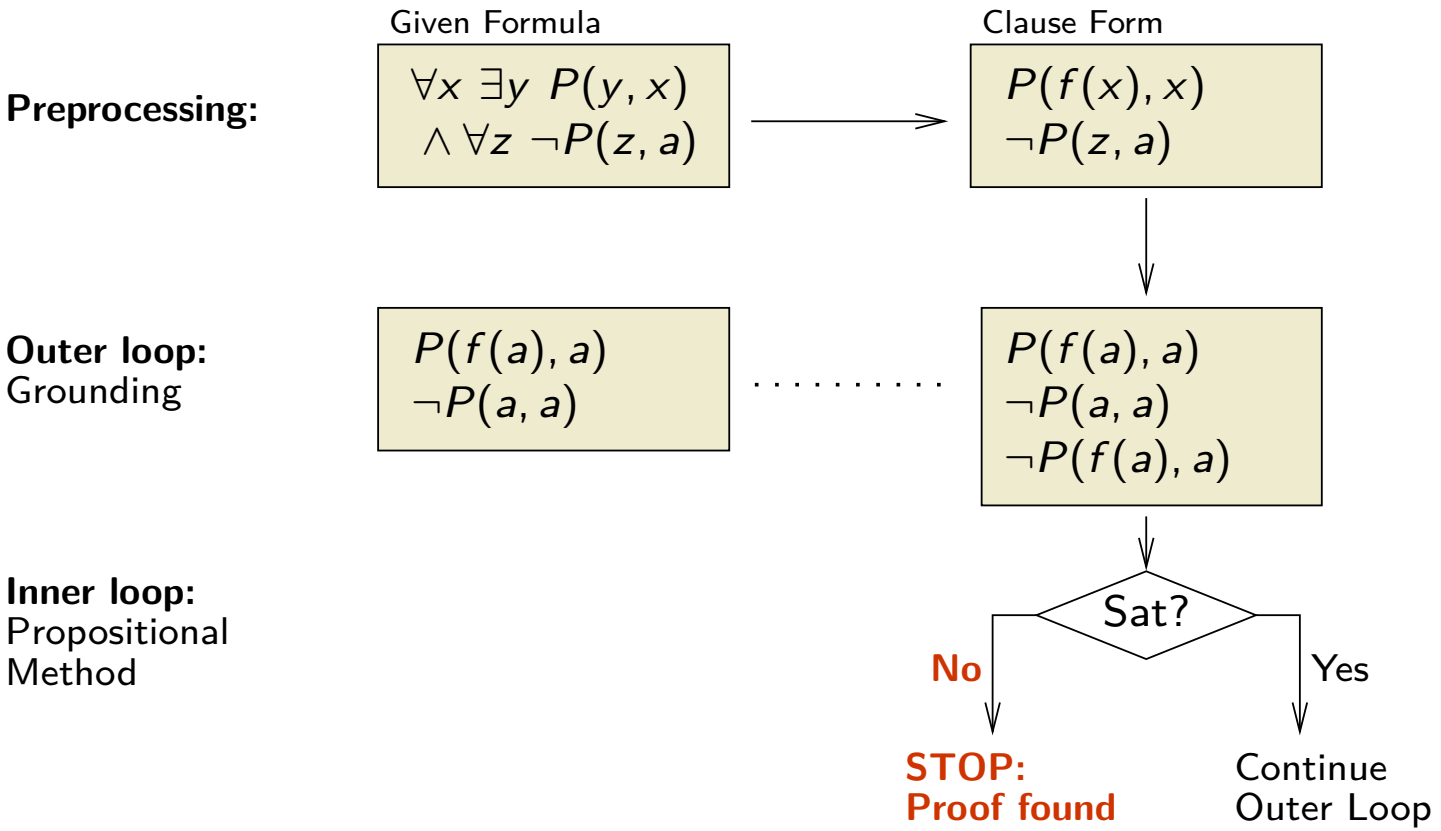
Theorem (Skolem-Herbrand-Theorem)

A set N of Σ-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

5



Calculi for First-Order Logic Theorem Proving

- 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.
  - **Guidance:** Instance-Based Methods are similar to Gilmore's method but generate ground instances in a guided way
  - **Avoidance:** Resolution calculi need not generate the ground instances at all

    Resolution inferences operate directly on clauses, not on their ground instances

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.	$\neg A \vee \neg A \vee B$	(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.	$\bot$	(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: $I \models C \vee A \vee A \Rightarrow I \models C \vee A$

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

- That is, if a propositional 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 $\bot$
- 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

Lifting Propositional Resolution to First-Order Resolution

Propositional resolution

Clauses	Ground instances
$P(f(x), y)$	$\{P(f(a), a), \ldots, P(f(f(a)), f(f(a))), \ldots\}$
$\neg P(z, z)$	$\{\neg P(a), \ldots, \neg P(f(f(a)), f(f(a))), \ldots\}$

Only common instances of $P(f(x), y)$ and $P(z, z)$ give rise to inference:

$$\frac{P(f(f(a)), f(f(a))) \qquad \neg P(f(f(a)), f(f(a)))}{\bot}$$

Unification

All common instances of $P(f(x), y)$ and $P(z, z)$ are instances of $P(f(x), f(x))$

$P(f(x), f(x))$ is computed deterministically by **unification**

First-order resolution

$$\frac{P(f(x), y) \qquad \neg P(z, z)}{\bot}$$

Justified by existence of $P(f(x), f(x))$

Can represent infinitely many propositional resolution inferences

Substitutions and Unifiers

- A **substitution** σ 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 σ is from above: $P(f(x), y)\sigma = P(f(x), f(x))$

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

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

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

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).

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**

- 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 $\bot$
- Perhaps easiest proof: Herbrand Theorem + completeness of propositional resolution + Lifting Theorem (see blackboard)

  Lifting Theorem: the conclusion of any propositional inference on ground instances of first-order clauses can be obtained by instantiating the conclusion of a first-order inference on the first-order clauses
- Closure can be achieved by 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`.

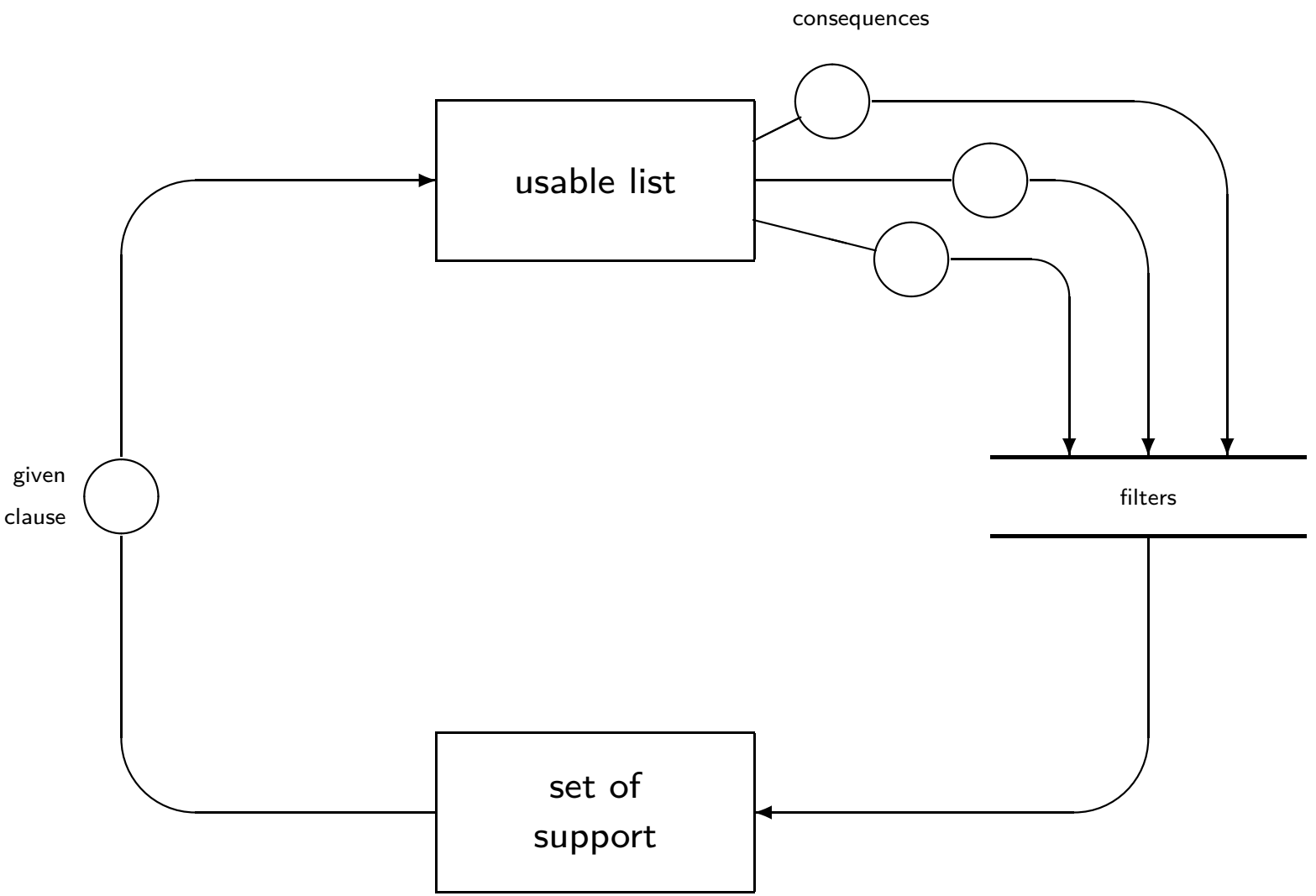
Initialize `sos` with the input clauses, `usable` empty.

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



Calculi for First-Order Logic Theorem Proving

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.
  - **Guidance:** Instance-Based Methods are similar to Gilmore's method but generate ground instances in a guided way
  - **Avoidance:** Resolution calculi need not generate the ground instances at all

    Resolution inferences operate directly on clauses, not on their ground instances

There are alternatives to resolution

Families of First-Order Logic Calculi

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

Families of First-Order Logic Calculi

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)

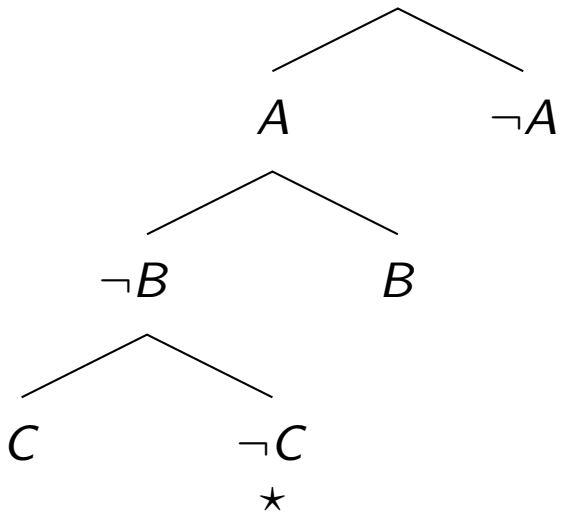
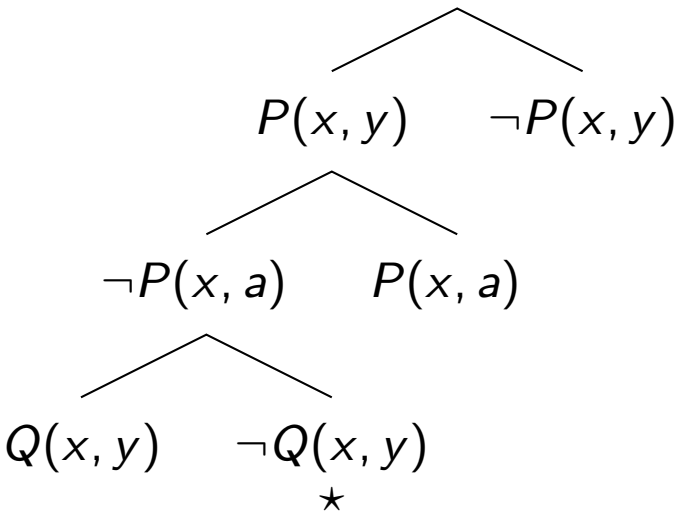
Clauses do not become longer (cf. Resolution)

May delete variant clauses (cf. Rigid Variable Approach)

Next: FDPLL as an example of a simple instance-based method

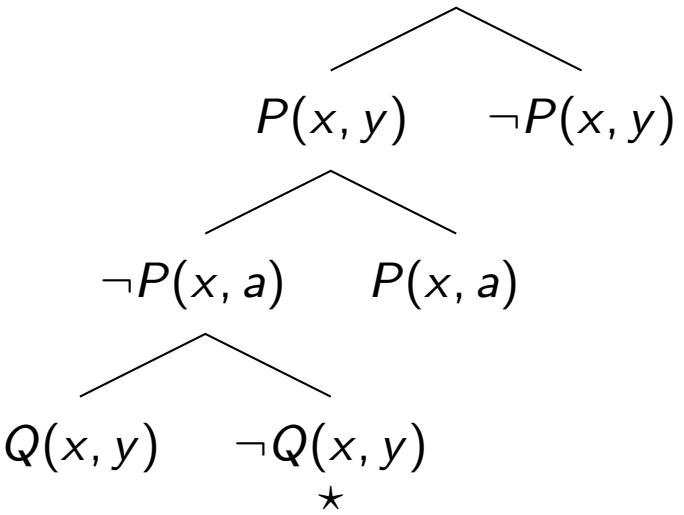
Instance-Based Method – FDPLL

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
- How are variables treated?
  (a) **Universal**, as in Resolution?, (b) **Rigid**, as in Tableaux? (c) **Schema!**
- How to extract an interpretation from a branch?
- When is a branch closed?
- How to construct such trees (calculus)?

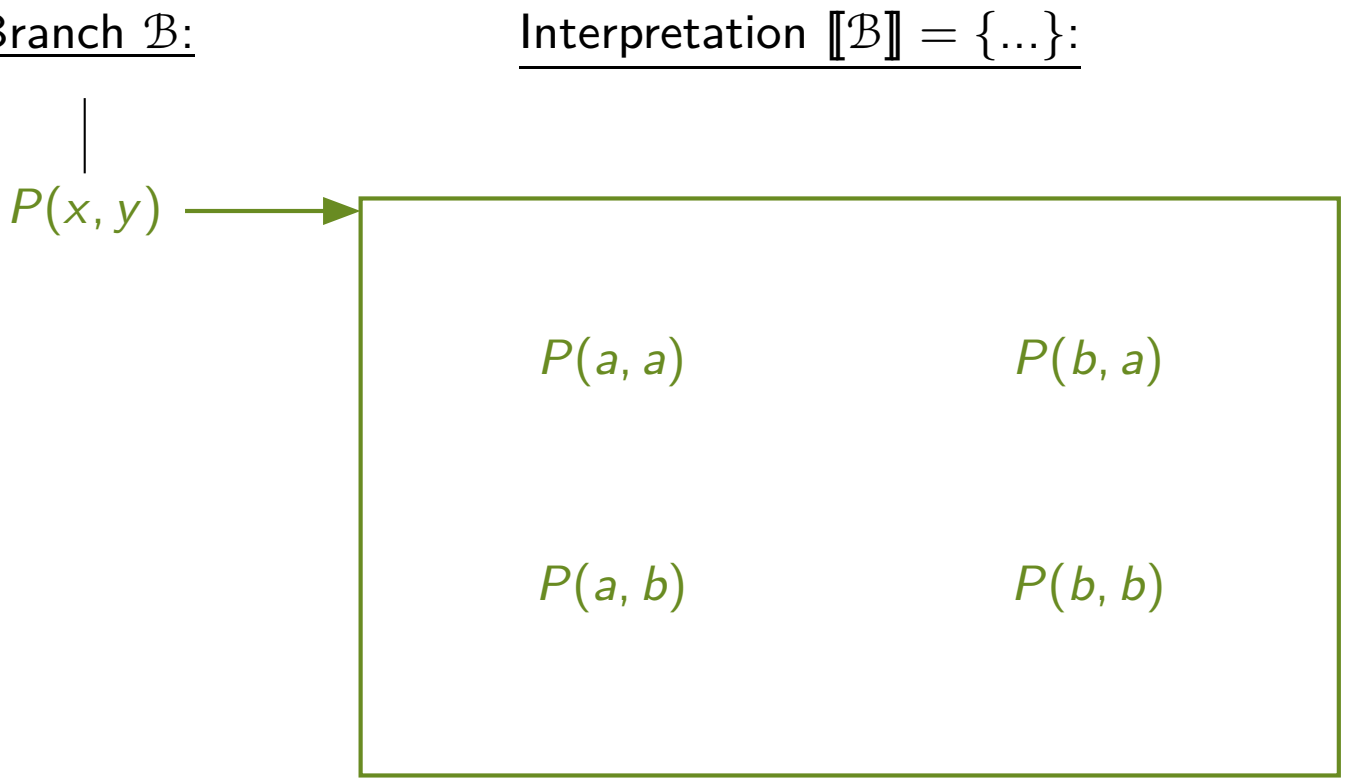
Extracting an Interpretation from a Branch

Branch $\mathcal{B}$:

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

- 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.

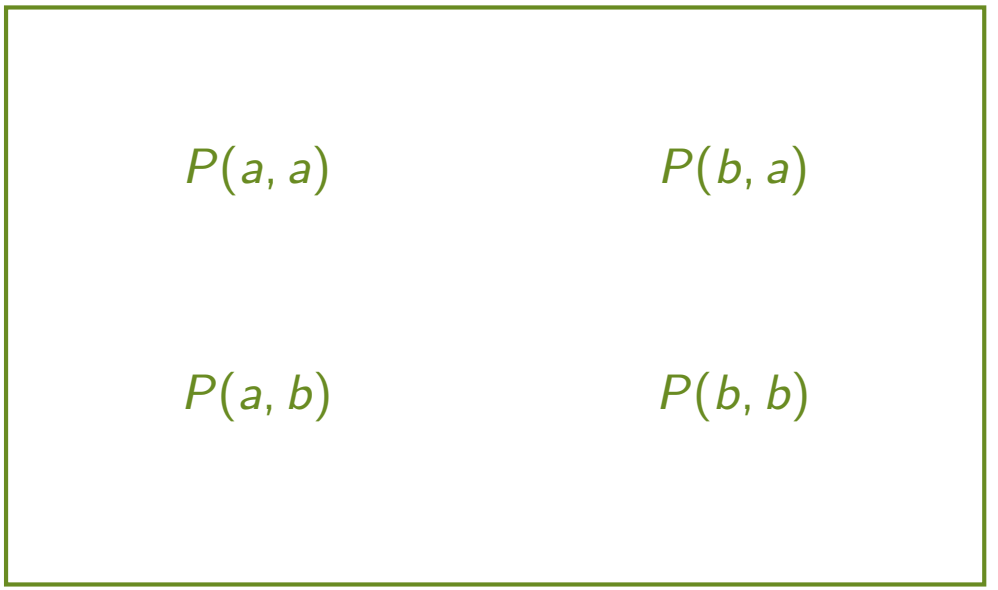
Extracting an Interpretation from a Branch

Branch $\mathcal{B}$:

$P(x, y)$

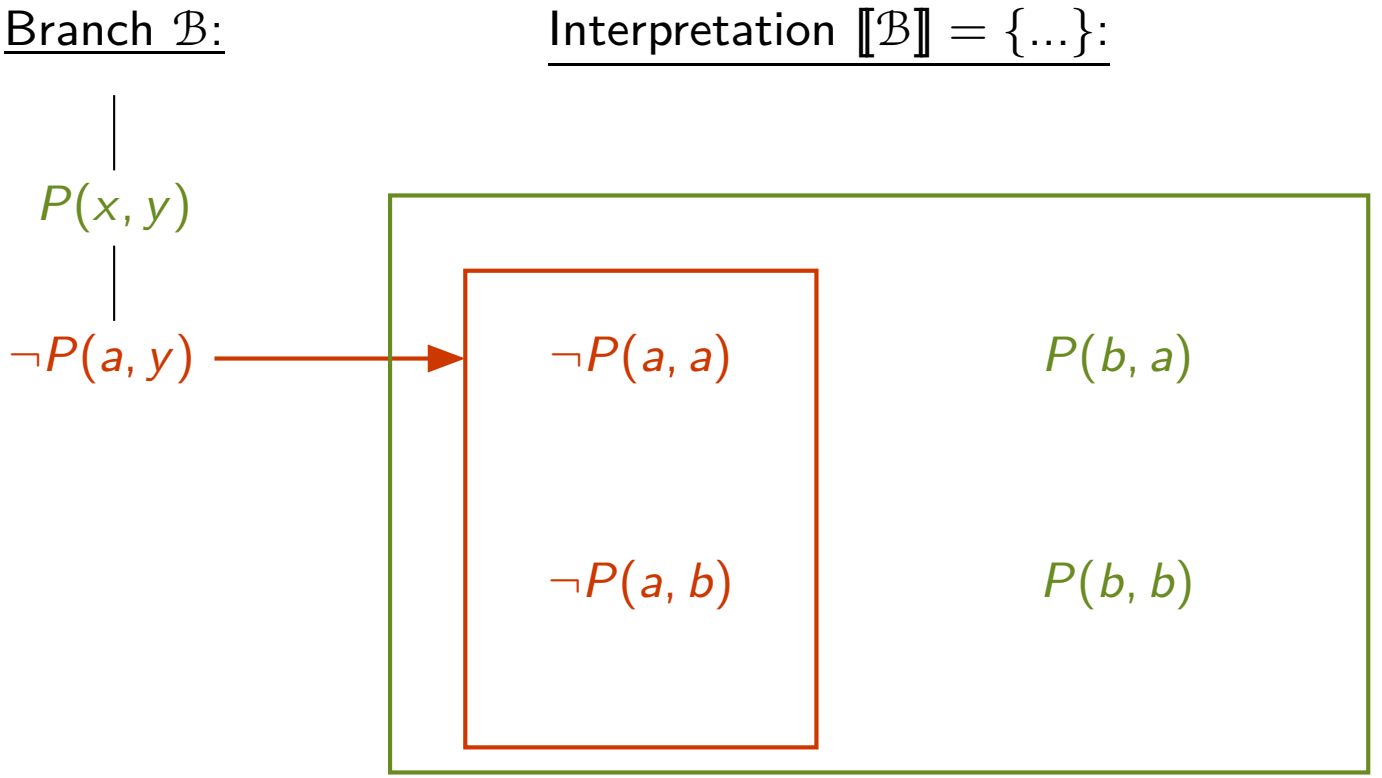
$\neg P(a, y)$

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



- 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

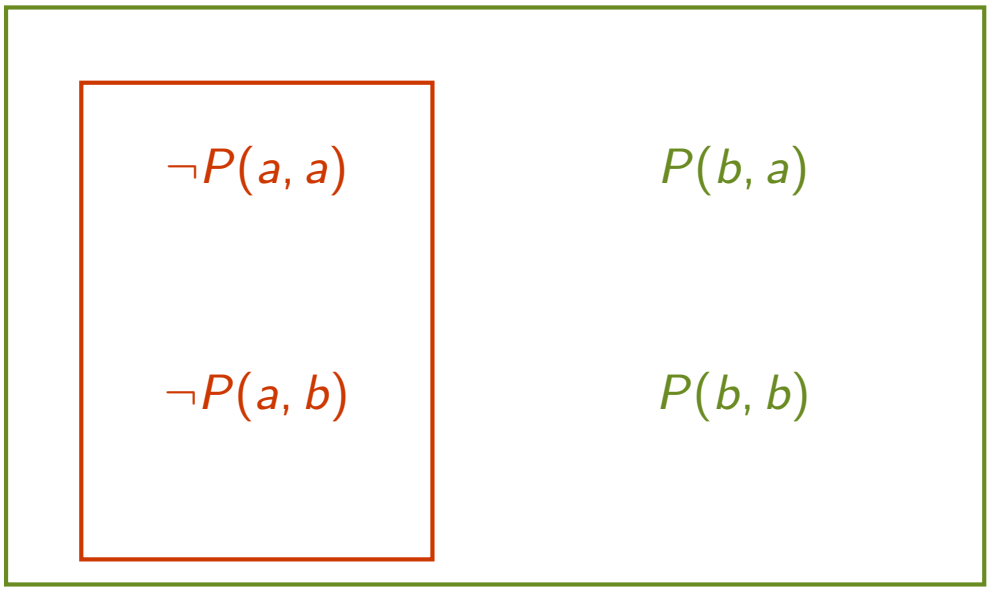
Branch $\mathcal{B}$:

$P(x, y)$

$\neg P(a, y)$

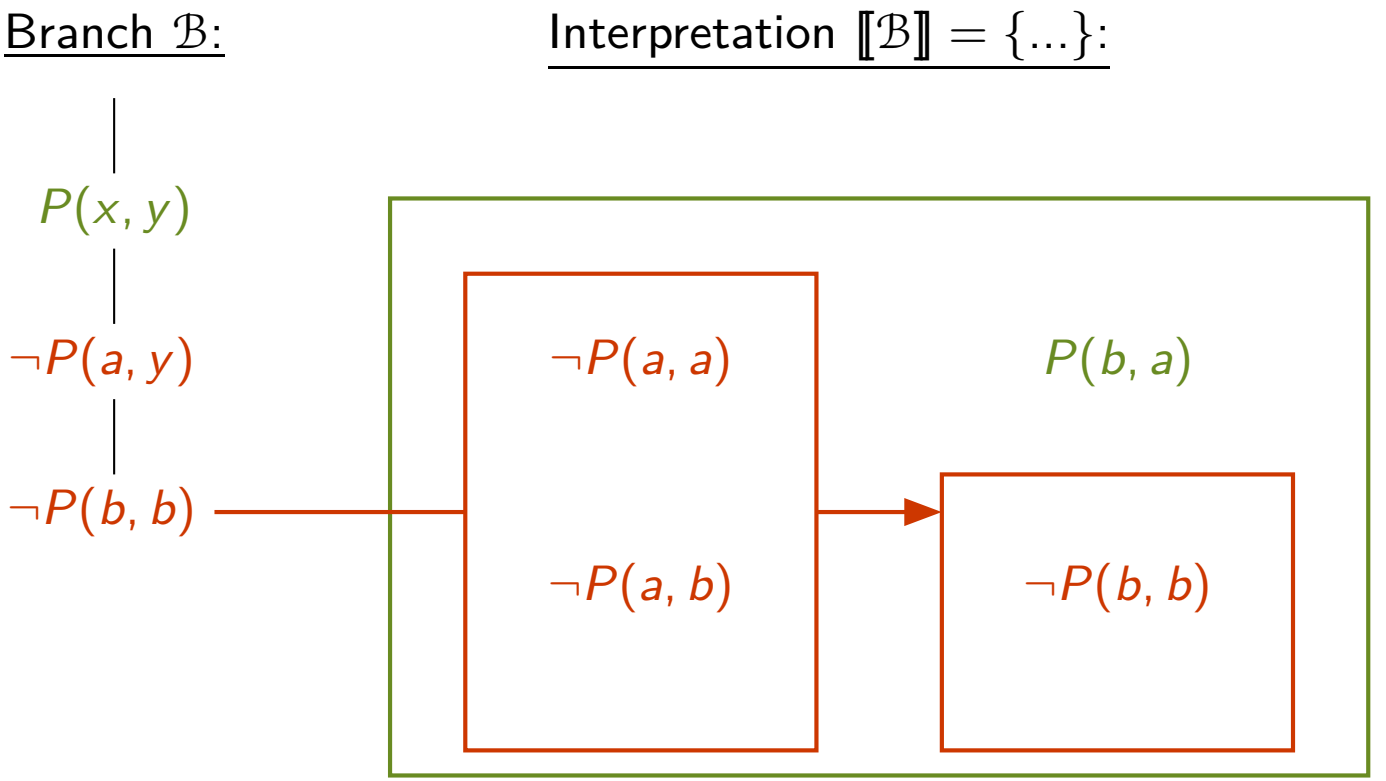
$\neg P(b, b)$

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



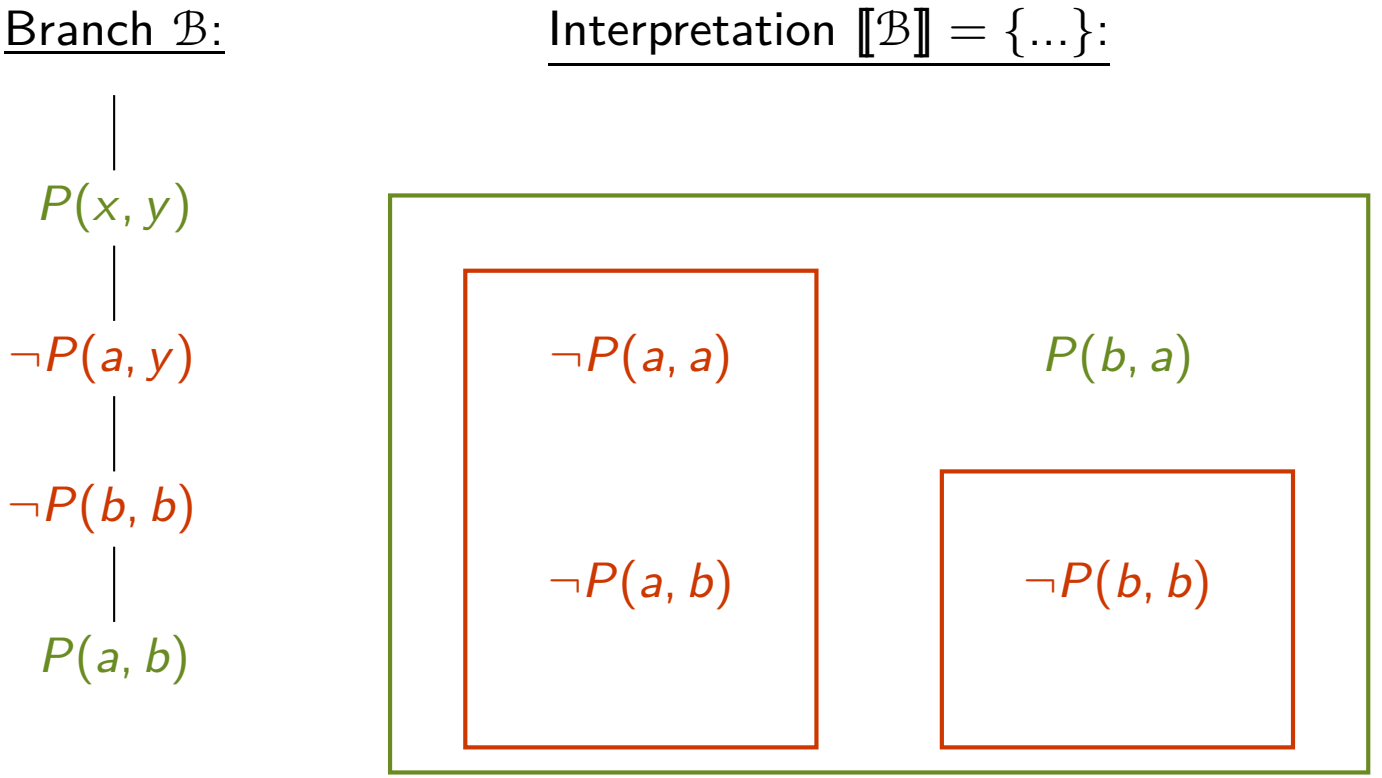
- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

Extracting an Interpretation from a Branch



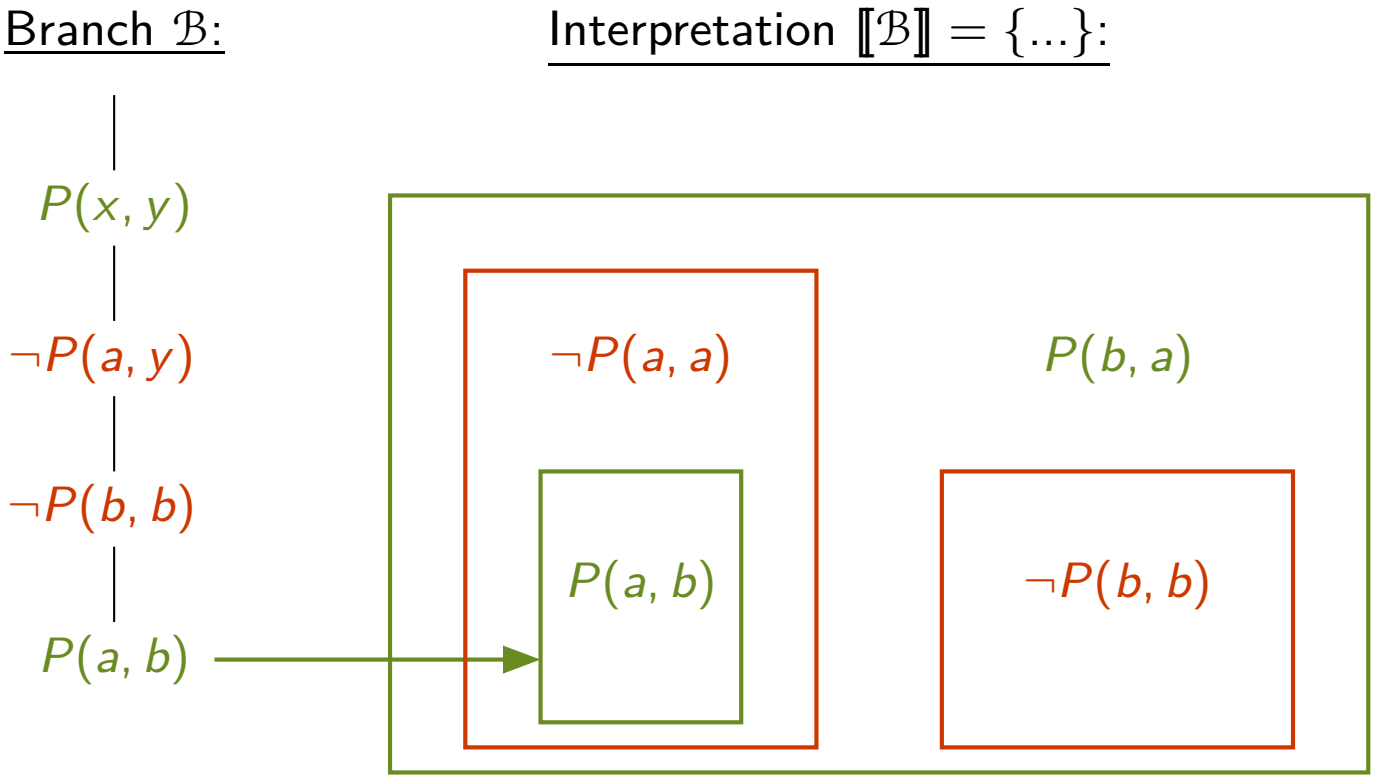
- A branch literal specifies the truth values for all its ground instances, unless there is a more specific literal specifying opposite truth values.

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 $\mathcal{B}$:

$P(x, y)$

$\neg P(a, y)$

$\neg P(b, b)$

$P(a, b)$

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

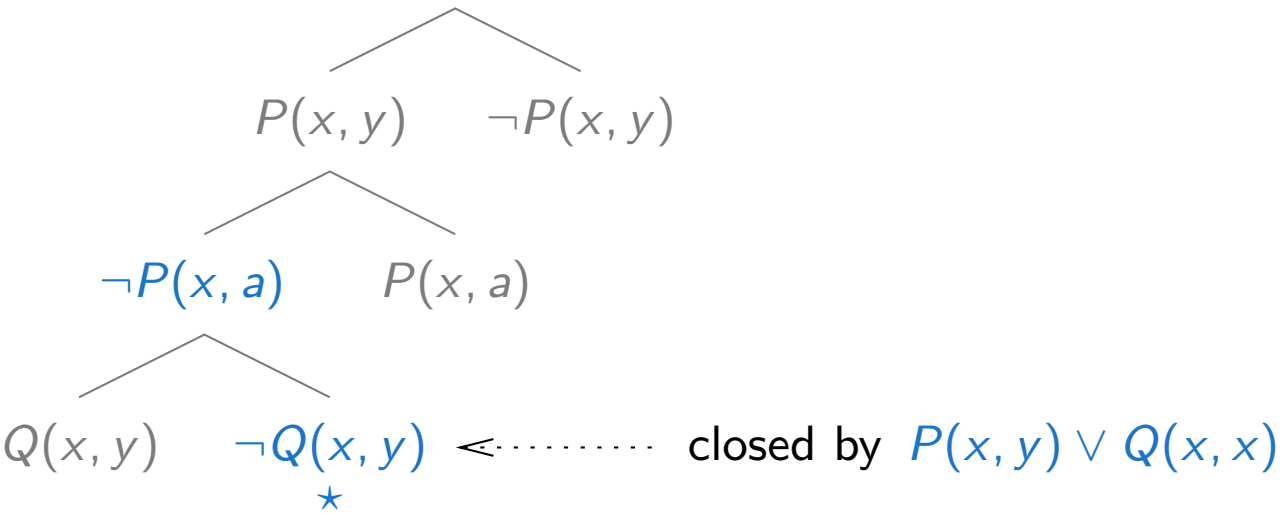
$\{ \quad \neg P(a, a) \quad , \quad P(b, a) \quad , \quad P(a, b) \quad , \quad \neg P(b, b) \quad \}$

- 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.

2First-Order case: 5



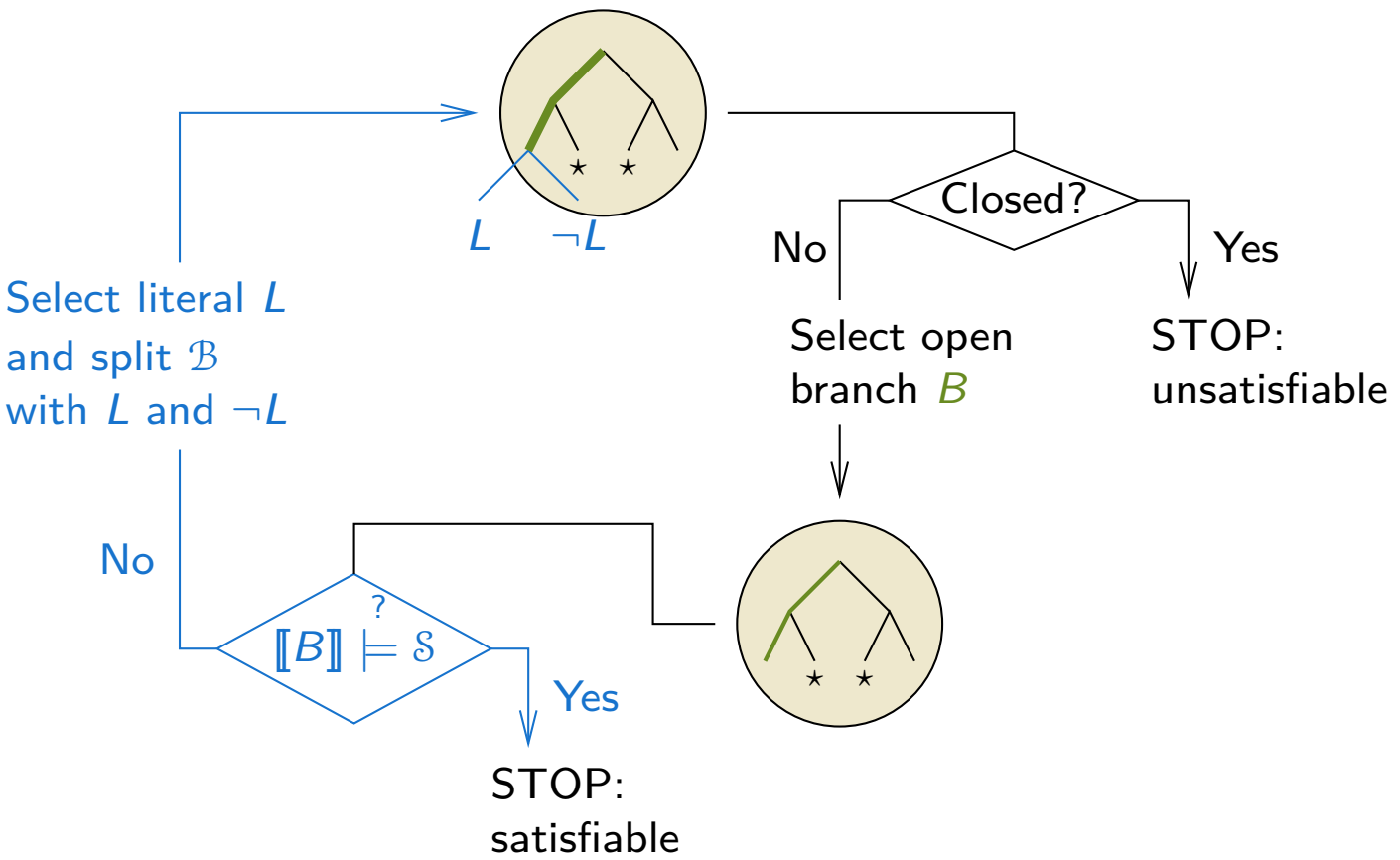
1. 4Replace all variables in tree by a constant \$. Gives propositional tree
2. 5Compute matcher γ to propositionally close branch
3. 5Mark branch as closed ($\star$)

FDPLL Calculus

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

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

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

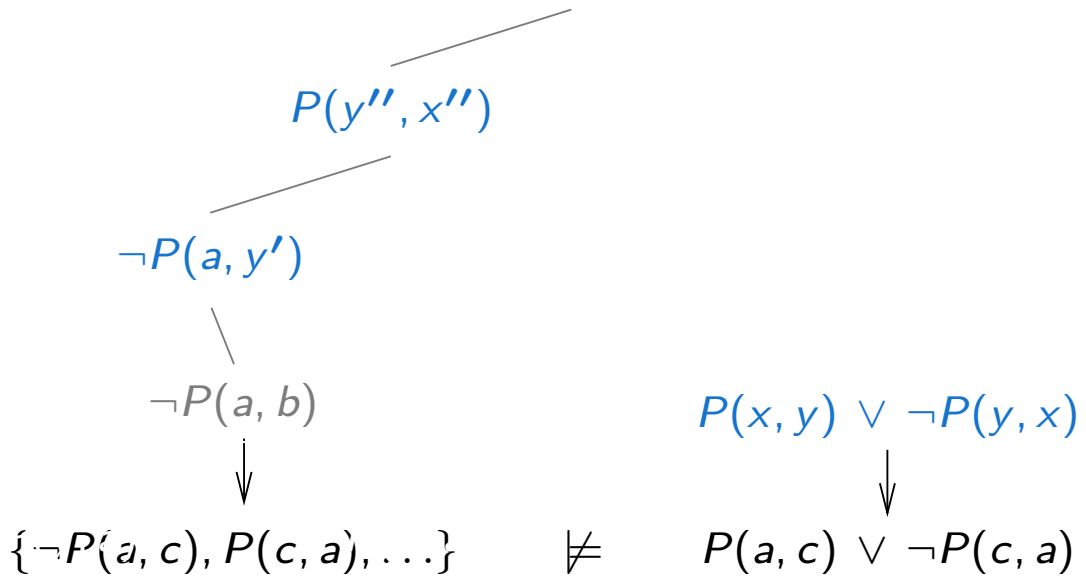


Next: Testing $[\![B]\!] \models \mathcal{S}$ and splitting

Calculus: The Splitting Rule

Purpose: Satisfy a clause that is currently "false"

5



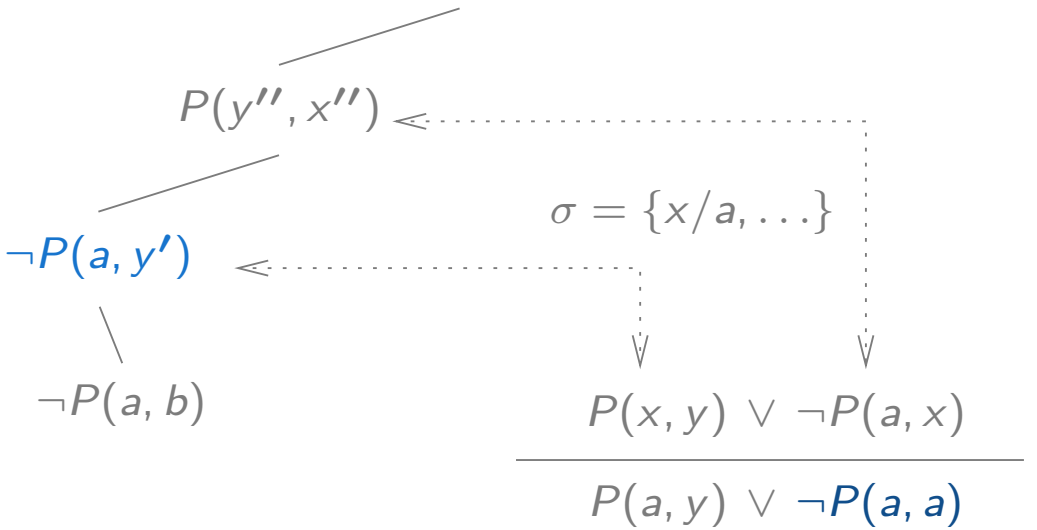
1. 3Compute simultaneous most general unifier σ
2. 4Select from clause instance a literal not on branch
3. 5Split with this literal

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"
3



1. 3Compute MGU σ of clause against branch literals
2. 4If 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

Calculus: 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

Model Generation

Scenario: no "theorem" to prove, or disprove a "theorem"

A model provides further information then

Why compute models?

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.
Finite models of quasigroups, (MGTP/G). [Fujita et al, IJCAI 93]

Model Generation

Why compute models (cont'd)?

Natural Language Processing:

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

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

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

$$\textit{BG-Knowledge} \cup \textit{Discourse\_so\_far} \not\models \neg \textit{New\_utterance}$$

iff $\textit{BG-Knowledge} \cup \textit{Discourse\_so\_far} \cup \textit{New\_utterance}$ is **satisfiable**

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

$$\textit{BG-Knowledge} \cup \textit{Discourse\_so\_far} \not\models \textit{New\_utterance}$$

iff $\textit{BG-Knowledge} \cup \textit{Discourse\_so\_far} \cup \neg \textit{New\_utterance}$ is **satisfiable**

Example - Group Theory

The following axioms specify a group

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

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: $\{1, 2, 3, 4, 5, 6\}$

$e : 1$

$i :$

1	2	3	4	5	6
1	2	3	5	4	6

$*:$

	1	2	3	4	5	6
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

Finite Model Finders - Idea

- Assume a fixed domain size n.
- Use a tool to decide if there exists a model with domain size n for a given problem.
- Do this starting with $n = 1$ with increasing n until a model is found.
- Note: domain of size n will consist of $\{1, \ldots, n\}$.

1. Approach: SEM-style

- Tools: SEM, Finder, Mace4
- Specialized constraint solvers.
- For a given domain generate all ground instances of the clause.
- Example: For domain size 2 and clause $p(a, g(x))$ the instances are $p(a, g(1))$ and $p(a, g(2))$.

1. Approach: SEM-style

- Set up multiplication tables for all symbols with the whole domain as cell values.
- Example: For domain size 2 and function symbol g with arity 1 the cells are $g(1) = \{1, 2\}$ and $g(2) = \{1, 2\}$.
- Try to restrict each cell to exactly 1 value.
- The clauses are the constraints guiding the search and propagation.
- Example: if the cell of a contains $\{1\}$, the clause $a = b$ forces the cell of b to be $\{1\}$ as well.

2. Approach: Mace-style

- Tools: Mace2, Paradox
- For given domain size n transform first-order clause set into equisatisfiable propositional clause set.
- Original problem has a model of domain size n iff the transformed problem is satisfiable.
- Run SAT solver on transformed problem and translate model back.

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

Part 3: Theory Reasoning

Theory Reasoning

Let T be a first-order theory of signature Σ

Let L be a class of Σ-formulas

The T-validity Problem

- Given ϕ in L, is it the case that $T \models \phi$? More accurately:
- Given ϕ 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 $\models \phi$ (ϕ 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 decidably only for restricted L and T

Approaches to Theory Reasoning

Theory-Reasoning in Automated First-Order Theorem Proving

- Semi-decide the T-validity problem, $T \models \phi$?
- ϕ arbitrary first-order formula, T universal theory
- Generality is strength and weakness at the same time
- Really successful only for specific instance:
  T = equality, inference rules like paramodulation

Satisfiability Modulo Theories (SMT)

- Decide the T-validity problem, $T \models \phi$?
- Usual restriction: ϕ is quantifier-free, i.e. all variables implicitly universally quantified
- Applications in particular to formal verification

  Trivial example:

  "arrays+integers" $\models m \geq 0 \wedge a[i] \geq 0 \wedge a'[i] = a[i] + m \rightarrow a'[i] \geq 0$

Checking Satisfiability Modulo Theories

Given: A quantifier-free formula ϕ (implicitly existentially quantified)
Task: Decide whether ϕ 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
- 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

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
  Σ: 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 Σ_1, Σ_2
- $\Sigma_1 \cap \Sigma_2 = \emptyset$
- ϕ 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*:

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

Nelson-Oppen Combination Method

Variable abstraction + *equality propagation*:

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

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*:

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

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

Nelson-Oppen Combination Method

Variable abstraction + *equality propagation*:

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

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

Nelson-Oppen Combination Method

Variable abstraction + *equality propagation*:

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

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

Nelson-Oppen Combination Method

Variable abstraction + *equality propagation*:

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

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

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*:

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

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

Linear Arithmetic Decision Problems

(Slides by Michael Norrish)

- If the language is rich enough (has multiplication, has quantifiers), deciding the validity of arbitrary mathmatical formulas (over $\mathbb{Z}$ or $\mathbb{N}$) is impossible.
- With a more impoverished language, a theory may be decidable.
- Historically, this research was part of the attempt to determine the limits of decidability.
- In the present, techniques similar to these are used to solve real-world problems, in a huge variety of systems.

Linear Arithmetic Formulas

$$
\begin{array}{lll}
\mathit{formula} & ::= & \mathit{formula} \wedge \mathit{formula} \mid \mathit{formula} \vee \mathit{formula} \mid \\
 & & \neg \mathit{formula} \mid \exists \mathit{var}.\, \mathit{formula} \mid \forall \mathit{var}.\, \mathit{formula} \mid \\
 & & \mathit{term}\ \mathit{relop}\ \mathit{term} \\
\mathit{term} & ::= & \mathit{numeral} \mid \mathit{term} + \mathit{term} \mid -\mathit{term} \mid \\
 & & \mathit{numeral} * \mathit{term} \mid \mathit{var} \\
\mathit{relop} & ::= & < \mid \leq \mid = \mid \geq \mid > \\
\mathit{var} & ::= & x \mid y \mid z \ldots \\
\mathit{numeral} & ::= & 0 \mid 1 \mid 2 \ldots
\end{array}
$$

$\mathit{numeral} * \mathit{term}$ isn't really multiplication; it's short-hand for $\mathit{term} + \mathit{term} + \cdots + \mathit{term}$.

Decision Procedures

- The aim is to produce an algorithm for determining whether or not a Presburger formula is valid with respect to the standard interpretation in arithmetic.
- Such an algorithm is a decision procedure if it is sure to correctly say "true" or "false" for all **closed** formulas.

- Will discuss algorithms for determining truth of formulas of Presburger arithmetic:
  - **Fourier-Motzkin** variable elimination (FMVE), when variables are from $\mathbb{R}$ (or $\mathbb{Q}$)
  - **Omega Test** when variables are from $\mathbb{Z}$ (or $\mathbb{N}$)
  - **Cooper's algorithm** for $\mathbb{Z}$ (or $\mathbb{N}$)

Quantifier Elimination

- All the methods we'll look at are **quantifier elimination** procedures.
- If a formula with no free variables has no quantifiers, then it is easy to determine its truth value, e.g., $10 > 11 \ \vee \ 3 + 4 < 5 \times 3 - 6$.
- Quantifier elimination works by taking input P with n quantifiers and turning it into equvalent formula P' with m quantifiers, and where $m < n$.
- So, eventually

  $$P \leftrightarrow P' \leftrightarrow ... \leftrightarrow Q$$

  and Q has no quantifiers.
- Q will be trivially true or false, and that's the decision

Normalisation

- Methods require input formulas to be normalised (e.g., collect coefficients, use only $<$ and $\leq$)
- Methods eliminate innermost **existential** quantifiers. Universal quantifiers are normalised with
  $(\forall x.\ P(x)) \leftrightarrow \neg(\exists x.\ \neg P(x))$
- In FMVE, the sub-formula under the innermost existential quantifier must be a conjunction of relations.
- This means the inner formula must be converted to **disjunctive normal form** (DNF):

$$(c_{11} \wedge c_{12} \wedge \cdots \wedge c_{1n_1}) \vee \cdots \vee (c_{m1} \wedge c_{m2} \wedge \cdots \wedge c_{mn_m})$$

Disjunctive Normal Form

Transform with equivalences

$$\begin{aligned} p \wedge (q \vee r) &\leftrightarrow (p \wedge q) \vee (p \wedge r) \\ (p \vee q) \wedge r &\leftrightarrow (p \wedge r) \vee (q \wedge r) \end{aligned}$$

Possibly exponential cost.

Must have also moved negations inwards, achieving **Negation Normal Form**, using

$$\begin{aligned} \neg(p \wedge q) &\leftrightarrow \neg p \vee \neg q \\ \neg(p \vee q) &\leftrightarrow \neg p \wedge \neg q \\ \neg\neg p &\leftrightarrow p \end{aligned}$$

Normalisation (cont.)

The formula under $\exists$ is in DNF.

Next, the $\exists$ must be moved inwards

- First over disjuncts, using

$$(\exists x.P \vee Q) \leftrightarrow (\exists x.\ P) \vee (\exists x.\ Q)$$

- Must then ensure every conjunct under the quantifier mentions the bound variable.

  Use

$$(\exists x.\ \ P(x)\ \wedge\ Q) \leftrightarrow (\exists x.\ \ P(x))\ \wedge\ Q$$

  For example

$$(\exists x.\ \ 3 < x\ \wedge\ x + 2y \leq 6\ \wedge\ y < 0) \longrightarrow$$
$$(\exists x.\ \ 3 < x\ \wedge\ x + 2y \leq 6)\ \wedge\ y < 0$$

Linear Real Number Arithmetic – Fourier-Motzkin theorems

The following simple facts are the basis for a very simple-minded quantifier elimination procedure.
Over $\mathbb{R}$ (or $\mathbb{Q}$), with $a, b > 0$:

$$
\begin{aligned}
(\exists x.\ c \leq ax \wedge bx \leq d) &\quad \leftrightarrow \quad bc \leq ad \\
(\exists x.\ c < ax \wedge bx \leq d) &\quad \leftrightarrow \quad bc < ad \\
(\exists x.\ c \leq ax \wedge bx < d) &\quad \leftrightarrow \quad bc < ad \\
(\exists x.\ c < ax \wedge bx < d) &\quad \leftrightarrow \quad bc < ad
\end{aligned}
$$

In all four, the right hand side is implied by the left because of transitivity (e.g., $x < y \wedge y \leq z \Rightarrow x < z$).

Fourier-Motzkin theorems (cont.)

In the other direction:

$$bc < ad \Rightarrow (\exists x.\ c < ax \wedge bx \leq d)$$

take x to be $\frac{d}{b}$: $c < a(\frac{d}{b})$, and $b(\frac{d}{b}) \leq d$.

For

$$bc < ad \Rightarrow (\exists x.\ c < ax \wedge bx < d)$$

take x to be $\frac{bc+ad}{2ab}$:

$$c < a\left(\frac{bc+ad}{2ab}\right) \leftrightarrow 2bc < bc + ad \leftrightarrow bc < ad$$

(and similarly for the other bound)

Extending to a full procedure

- So far: a quantifier elimination procedure for formulas where quantifiers only ever have scope over 1 upper bound, and 1 lower bound.
- The method needs to extend to cover cases with multiple constraints.
- No lower bound, many upper bounds:

$$(\exists x.\ \ b_1 x < d_1 \wedge b_2 x < d_2 \cdots \wedge b_n x < d_n)$$

  Verdict: **True!** (take $\min(\frac{d_i}{b_i}) - 1$ as witness for x)
- No upper bound, many lower bounds: obviously analogous.

Combining many constraints—I

Example:

$$(\exists x.\ \ c \leq ax \wedge b_1 x \leq d_1 \wedge b_2 x \leq d_2) \leftrightarrow b_1 c \leq a d_1 \wedge b_2 c \leq a d_2$$

- From left to right, result just depends on transitivity.
- From right to left, take x to be $\min(\frac{d_1}{b_1}, \frac{d_2}{b_2})$.

In general, with many constraints, combine all possible lower-upper bound pairs.

(Proof that this is possible is by induction on number of constraints.)

Combining many constraints—II

The core elimination formula is

$$\exists x.\ (\bigwedge_h c_h \leq a_h x) \wedge (\bigwedge_i c_i < a_i x) \wedge (\bigwedge_j b_j x \leq d_j) \wedge (\bigwedge_k b_k x < d_k)$$

$$\leftrightarrow$$

$$(\bigwedge_{h,j} b_j c_h \leq a_h d_j) \wedge (\bigwedge_{h,k} b_k c_h < a_h d_k) \wedge$$
$$(\bigwedge_{i,j} b_j c_i < a_i d_j) \wedge (\bigwedge_{i,k} b_k c_i < a_i d_k)$$

With n constraints initially, evenly divided between upper and lower bounds, this formula generates $\frac{n^2}{4}$ new constraints.

FMVE example

$$\forall x.\ 20 + x \leq 0 \quad \Rightarrow \quad \exists y.\ 3y + x \leq 10 \ \wedge \ 20 \leq y - x$$

(re-arrange)

$$\leftrightarrow \quad \forall x.\ 20 + x \leq 0 \quad \Rightarrow \quad \exists y.\ 20 + x \leq y \ \wedge \ 3y \leq 10 - x$$

(eliminate y)

$$\leftrightarrow \quad \forall x.\ 20 + x \leq 0 \quad \Rightarrow \quad 60 + 3x \leq 10 - x$$

(re-arrange)

$$\leftrightarrow \quad \forall x.\ 20 + x \leq 0 \quad \Rightarrow \quad 4x + 50 \leq 0$$

(normalise universal)

$$\leftrightarrow \quad \neg\exists x.\ 20 + x \leq 0 \ \wedge \ 0 < 4x + 50$$

(re-arrange)

$$\leftrightarrow \quad \neg\exists x.\ -50 < 4x \ \wedge \ x \leq -20$$

(eliminate x)

$$\leftrightarrow \quad \neg(-50 < -80) \qquad \leftrightarrow \ \top$$

Efficiency

- As before, when eliminating an existential over n constraints we may introduce $\frac{n^2}{4}$ new constraints.
- With k quantifiers to eliminate, we might end with

$$\frac{n^{2^k}}{4^k}$$

  constraints.
- If dealing with alternating quantifiers, repeated conversions to DNF may really hurt.

Expressivity

- Unique existence:

$$(\exists! x.\ P(x)) \leftrightarrow (\exists x.\ P(x) \wedge \forall y.\ P(y) \Rightarrow (y = x))$$

- Conditional expressions:
  - **if** $formula_1$ **then** $formula_2$ **else** $formula_3$ is the same as $(formula_1 \wedge formula_2) \vee (\neg formula_1 \wedge formula_3)$
  - **if-then-else** expressions over *term*, can be moved up and out to be over formulas:

$$(\textbf{if } x < y \textbf{ then } x \textbf{ else } y) < z$$

$$\leftrightarrow$$

$$\textbf{if } x < y \textbf{ then } x < z \textbf{ else } y < z$$

- Minimum, maximum, absolute value...

Constraint satisfaction, optimisation

- It's possible to make the algorithm return witnesses to purely existential problems.
- E.g.,

$$\exists x\, y.\ 3x + 4y = 18 \wedge 5x - y \leq 7$$

might return $\{(x, 2), (y, 3)\}$ (or $\{(x, \frac{2}{3}), (y, 4)\}$, or ...).

- Can also maximise (minimise) z in system $\exists \vec{x}\, z.\ P(\vec{x}, z)$:
  - First check $\exists \vec{x}\, z.\ P(\vec{x}, z)$
  - If it has a solution, check

$$\exists z.\ \ (\exists \vec{x}.\ \ P(\vec{x}, z)) \ \wedge\ (\forall \vec{x}\, z'.\ \ P(\vec{x}, z') \ \Rightarrow\ z' \leq z)$$

  - If there is a maximum solution for z, this will find it
    Note alternation of quantifiers!

Integer Decision Procedures – Expressivity—I

- Can't do primality

$$prime(x) \leftrightarrow \neg\exists y\, z.\ x = yz \wedge 1 < y < x$$

because of restriction on multiplication

- Can do divisibility by specific numerals:

$$2|e \quad \leftrightarrow \quad \exists x.\ 2x = e$$

and so (for example):

$$\forall x.\ \ 0 < x < 30 \ \Rightarrow\ \neg(2|x \wedge 3|x \wedge 5|x)$$

Expressivity over Integers—II

- Can do integer division and modulus, as long as divisor is constant
- Use one of the following results (similar for division)

$$P(x \bmod d) \leftrightarrow$$
$$\exists q\, r.\, (x = qd + r) \wedge (0 \leq r < d \vee d < r \leq 0) \wedge P(r)$$

$$P(x \bmod d) \leftrightarrow$$
$$\forall q\, r.\, (x = qd + r) \wedge (0 \leq r < d \vee d < r \leq 0) \Rightarrow P(r)$$

Any formula involving modulus or integer division by a constant can be translated to one without.

When d is known, one of the disjuncts will immediately simplify away to false.

Expressivity over Integers—III

- Any procedure for $\mathbb{Z}$ trivially extends to be one for $\mathbb{N}$ (or any mixture of $\mathbb{N}$ and $\mathbb{Z}$) too: add extra constraints stating that variables are ≥ 0
- Ignore non-Presburger sub-terms by trying to prove more general goals.

  For example,

  $$\forall x\, y.\ xy > 6 \Rightarrow 2xy > 13$$

  becomes

  $$\forall z.\ z > 6 \Rightarrow 2z > 13$$

One Nice Thing About the Integers

The relations $<$ and $\leq$ are inter-convertible:

$$x \leq y \quad \leftrightarrow \quad x < y + 1$$
$$x < y \quad \leftrightarrow \quad x + 1 \leq y$$

Decision procedures can normalise one relation into the other.

Fourier-Motzkin for Integers?

- Central theorem is false:

$$(\exists x : \mathbb{Z}.\ 3 \leq 2x \leq 3) \quad \not\Rightarrow \quad 6 \leq 6$$

- But one direction still works (thanks to transitivity):

$$(\exists x.\ c \leq ax \wedge bx \leq d) \ \Rightarrow \ bc \leq ad$$

- We can compute consequences of existentially quantified formulas

Fourier-Motzkin for Integers?

Have

$$(\exists x.\ c \leq ax \wedge bx \leq d) \ \Rightarrow \ bc \leq ad$$

Thus an incomplete procedure for universal formulas over $\mathbb{Z}$:

1. Compute negation: $(\forall x.\ P(x)) \leftrightarrow \neg(\exists x.\ \neg P(x))$
2. Compute consequences:

   if $(\exists x.\ \neg P(x)) \Rightarrow \bot$ then $(\exists x.\ \neg P(x)) \leftrightarrow \bot$

   and

   $(\forall x.\ P(x)) \leftrightarrow \top$

   (Repeat for all quantified variables.)

This is Phase 1 of the Omega Test (when there are no alternating quantifiers)

Omega Phase 1—Example

$$\forall x\, y : \mathbb{Z}.\ 0 < x \wedge y < x \ \Rightarrow\ y + 1 < 2x$$

(normalise)

$$\leftrightarrow \quad \neg\exists x\, y.\ 1 \leq x\ \wedge\ y + 1 \leq x\ \wedge\ 2x \leq y + 1$$

$$\exists x\, y.\ 1 \leq x\ \wedge\ y + 1 \leq x\ \wedge\ 2x \leq y + 1$$

(eliminate y)

$$\Rightarrow \quad \exists x.\ 1 \leq x\ \wedge\ 2x \leq x$$

(normalise)

$$\Rightarrow \quad \exists x.\ 1 \leq x\ \wedge\ x \leq 0$$

(eliminate x)

$$\Rightarrow \quad 1 \leq 0 \quad (\leftrightarrow \bot)$$

Omega Phase 1 and the Interactive Theorem-Provers

The Omega Test's Phase 1 is used by systems like Coq, HOL4, HOL Light and Isabelle to decide arithmetic problems.

Against:

- it's incomplete
- it's inefficient
  - conversion to DNF
  - quadratic increase in numbers of constraints

For:

- it's easy to implement
- it's easy to adapt the procedures to create proofs that can be checked by other tools

FMVE can be extended to a complete method (see literature)

Cooper's Algorithm

A non-Fourier-Motzkin alternative:

- Cooper's algorithm is a decision procedure for (integer) Presburger arithmetic.
- It is also a quantifier elimination procedure, which also works from the inside out, eliminating existentials.
- Its **big** advantage is that it doesn't need to normalise input formulas to DNF.

Description is of simplest possible implementation: many tweaks are possible.

Cooper's Algorithm: outline

To eliminate the quantifier in $\exists x.\ P(x)$:

1. Normalise so that only operators are $<$, and divisibility $(c|e)$, and negations only occur around divisibility leaves.
2. Compute **least common multiple** of all coefficients of x, and multiply all leaves through by appropriate numbers so that every leaf features x multiplied by the same number c.
3. Now apply $(\exists x.\ P(cx)) \leftrightarrow (\exists x.\ P(x) \wedge c|x)$.

Cooper's Algorithm: normalisation

$$\forall x\, y : \mathbb{Z}.\ 0 < y \wedge x < y \Rightarrow x + 1 < 2y$$

(normalise)

$$\leftrightarrow \quad \neg\exists x\, y.\ 0 < y \wedge x < y \wedge 2y < x + 2$$

(transform y to $2y$ everywhere)

$$\leftrightarrow \quad \neg\exists x\, y.\ 0 < 2y \wedge 2x < 2y \wedge 2y < x + 2$$

(give y unit coefficient)

$$\leftrightarrow \quad \neg\exists x\, y.\ 0 < y \wedge 2x < y \wedge y < x + 2 \wedge 2 \,|\, y$$

Cooper's Algorithm: two cases

How might $\exists x.\ P(x)$ be true?

Either:

- there is a least x making P true; or
- there is no least x: however small you go, there will be a smaller x that still makes P true

Construct two formulas corresponding to both cases.

Cooper's Algorithm: infinitely many small solutions

The case when the values of x satisfying P "go all the way down".

Look at the leaf formulas in P, and think about their values when x has been made arbitrarily small:

- $x < e$: if x goes as small as we like, this will be **true**
- $e < x$: if x goes small, this will be **false**
- $c|x + e$: **unchanged**

This constructs $P_{-\infty}$, a formula where x only occurs in divisibility leaves.

Say δ is the **l.c.m.** of the constants involved in divisibility leaves. Need just test $P_{-\infty}$ on $1 \ldots \delta$.

Cooper's Algorithm: example

For

$$\exists y.\, 0 < y \;\wedge\; 2x < y \;\wedge\; y < x + 2 \;\wedge\; 2 \,|\, y$$

- $0 < y$ will become false as y gets small
- $2x < y$ also becomes false as y gets small
- $y < x + 2$ will be true as y gets small
- $2 \,|\, y$ doesn't change (it tests if y is even or not)

So in this case, $P_{-\infty}(y) \leftrightarrow (\bot \wedge \bot \wedge \top \wedge 2 \,|\, y) \leftrightarrow \bot$.

Cooper's Algorithm: least solution

The case when there is a least x satisfying P.

For there to be a least x satisfying P, it must be the case that one of the leaves $e < x$ is true, and that if x was any smaller the formula would become false.

Let $B = \{e \ : \ e < x \text{ is a leaf of } P\}$

Need just consider $P(b + j)$, where $b \in B$ and $j \in 1 \ldots \delta$.

Final elimination formula is:

$$(\exists x.\ P(x)) \leftrightarrow \bigvee_{j=1..\delta} P_{-\infty}(j) \ \vee \ \bigvee_{j=1..\delta} \ \bigvee_{b \in B} P(b + j)$$

Cooper's Algorithm: example continued

For

$$\exists y.\, 0 < y \ \wedge\ 2x < y \ \wedge\ y < x + 2 \ \wedge\ 2 \,|\, y$$

least solutions, if they exist, will be at $y = 1$, $y = 2$, $y = 2x + 1$, or $y = 2x + 2$.

The divisibility constraint eliminates two of these.

Original formula is equivalent to:

$$(2x < 2 \ \wedge\ 0 < x) \ \vee\ (0 < 2x + 2 \ \wedge\ x < 0)$$

(Which is unsatisfiable for x.)

Conclusion – Quantifier Elimination

- This just scratches the surface of a very big area.
- Fourier-Motzkin methods are very simple techniques for solving problems in $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, and $\mathbb{N}$.
- The correctness of the Omega Test and of Cooper's algorithm are alternative proofs of Presburger's 1929 result that Presburger arithmetic is decidable.
- Many other methods exist (particularly for purely existential problems, which is the field of **linear programming**).
- Though most interesting maths remains undecidable, these methods are extremely useful in practical situations.

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/`