

# **Automated Reasoning I**

**Uwe Waldmann**

**Winter Term 2023/2024**

# What is Automated Reasoning?

---

Automated reasoning:

Logical reasoning using a computer program,  
with little or no user interaction,  
using general methods, rather than approaches that work only for one  
specific problem.

Two examples:

Solving a sudoku.

Reasoning with equations.

# Introductory Example 1: Sudoku

---

	1	2	3	4	5	6	7	8	9
1								1	
2	4								
3		2							
4					5		4		7
5			8				3		
6			1		9				
7	3			4			2		
8		5		1					
9				8		6			

Goal:

Fill the empty fields with digits 1, ..., 9, so that each digit occurs exactly once in each row, column, and  $3 \times 3$  box.

# Introductory Example 1: Sudoku

---

	1	2	3	4	5	6	7	8	9
1								1	
2	4								
3		2							
4					5		4		7
5			8				3		
6			1		9				
7	3			4			2		
8		5		1					
9				8		6			

Idea:

Use boolean variables  $P_{i,j}^d$  with  $d, i, j \in \{1, \dots, 9\}$  to encode the problem:

$P_{i,j}^d = \text{true}$  iff the value of square  $i, j$  is  $d$ .

# Introductory Example 1: Sudoku

---

	1	2	3	4	5	6	7	8	9
1								1	
2	4								
3		2							
4					5		4		7
5			8				3		
6			1		9				
7	3			4			2		
8		5		1					
9				8		6			

Idea:

Use boolean variables  $P_{i,j}^d$  with  $d, i, j \in \{1, \dots, 9\}$  to encode the problem:

$P_{i,j}^d = \text{true}$  iff the value of square  $i, j$  is  $d$ .

For example:

$$P_{5,3}^8 = \text{true}.$$

$$P_{5,3}^7 = \text{false}.$$

# Coding Sudoku in Boolean Logic

---

- Concrete values result in formulas  $P_{i,j}^d$
- For every square  $(i,j)$  we generate  $P_{i,j}^1 \vee \dots \vee P_{i,j}^9$
- For every square  $(i,j)$  and pair of values  $d < d'$  we generate  $\neg P_{i,j}^d \vee \neg P_{i,j}^{d'}$
- For every value  $d$  and row  $i$  we generate  $P_{i,1}^d \vee \dots \vee P_{i,9}^d$   
(Analogously for columns and  $3 \times 3$  boxes)
- For every value  $d$ , row  $i$ , and pair of columns  $j < j'$   
we generate  $\neg P_{i,j}^d \vee \neg P_{i,j'}^d$   
(Analogously for columns and  $3 \times 3$  boxes)

# Coding Sudoku in Boolean Logic

---

Every assignment of boolean values to the variables  $P_{i,j}^d$  so that all formulas become true corresponds to a Sudoku solution (and vice versa).

# Coding Sudoku in Boolean Logic

---

Now use a SAT solver to check whether there is an assignment to the variables  $P_{i,j}^d$  so that all formulas become true:

Niklas Eén, Niklas Sörensson:

MiniSat (<http://minisat.se/>),

Beware:

The satisfiability problem is NP-complete.

Every known algorithm to solve it has an exponential time worst-case behaviour (or worse).



# Coding Sudoku in Boolean Logic

---

MiniSat solves the problem in a few milliseconds.

How? See part 2 of this lecture.

Does that contradict NP-completeness? No!

NP-completeness implies that there are really hard problem instances, it does not imply that all practically interesting problem instances are hard (for a well-written SAT solver).

# SAT Solvers in Practice

---

Some real-life applications of modern SAT solvers:

- hardware verification (model checking)

- with extensions:

  - software verification, hybrid system verification, ...

- checking software package dependencies

- solving combinatorial problems

- “The Largest Math Proof Ever” (Marijn Heule)

- ...

## Introductory Example 2: Equations

---

Task:

Prove:  $\frac{a}{a+1} = 1 + \frac{-1}{a+1}$ .

## Introductory Example 2: Equations

---

$$\frac{a}{a+1}$$

$$1 + \frac{-1}{a+1}$$

## Introductory Example 2: Equations

---

$$\frac{a}{a+1} = \frac{a+0}{a+1}$$

$$1 + \frac{-1}{a+1}$$

$$x + 0 = x \quad (1)$$

## Introductory Example 2: Equations

---

$$\frac{a}{a+1} = \frac{a+0}{a+1}$$

$$= \frac{a + (1 + (-1))}{a+1}$$

$$1 + \frac{-1}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

## Introductory Example 2: Equations

---

$$\frac{a}{a+1} = \frac{a+0}{a+1}$$

$$= \frac{a + (1 + (-1))}{a+1}$$

$$= \frac{(a+1) + (-1)}{a+1}$$

$$1 + \frac{-1}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

## Introductory Example 2: Equations

---

$$\frac{a}{a+1} = \frac{a+0}{a+1}$$

$$= \frac{a + (1 + (-1))}{a+1}$$

$$= \frac{(a+1) + (-1)}{a+1}$$

$$= \frac{a+1}{a+1} + \frac{-1}{a+1}$$

$$1 + \frac{-1}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$



## Introductory Example 2: Equations

---

$$\frac{a}{a+1} = \frac{a+0}{a+1}$$

$$= \frac{a + (1 + (-1))}{a+1}$$

$$= \frac{(a+1) + (-1)}{a+1}$$

$$= \frac{a+1}{a+1} + \frac{-1}{a+1}$$

$$= 1 + \frac{-1}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

How could we write a program that takes a set of equations and two terms and tests whether the terms can be connected via a chain of equalities?

It is easy to write a program that applies formulas *correctly*.

But: correct  $\neq$  useful.

## Introductory Example 2: Equations

---

$$\frac{a}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x + y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$\frac{a}{a+1} \longrightarrow \frac{a+0}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

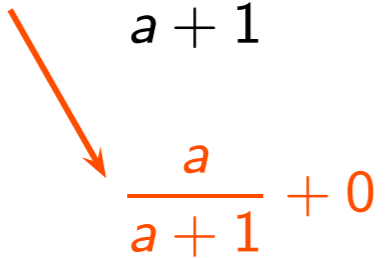
$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$\frac{a}{a+1} \xrightarrow{\quad} \frac{a+0}{a+1}$$

$$\frac{a}{a+1} + 0$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$\begin{array}{l} \frac{a}{a+1} \longrightarrow \frac{a+0}{a+1} \\ \quad \searrow \frac{a}{a+1} + 0 \\ \quad \searrow \frac{a}{a+(1+0)} \end{array}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$\begin{array}{l} \frac{a}{a+1} \longrightarrow \frac{a+0}{a+1} \\ \searrow \frac{a}{a+1} + 0 \\ \searrow \frac{a}{a+(1+0)} \\ \searrow \frac{a}{a + \frac{a+2}{a+2}} \end{array}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

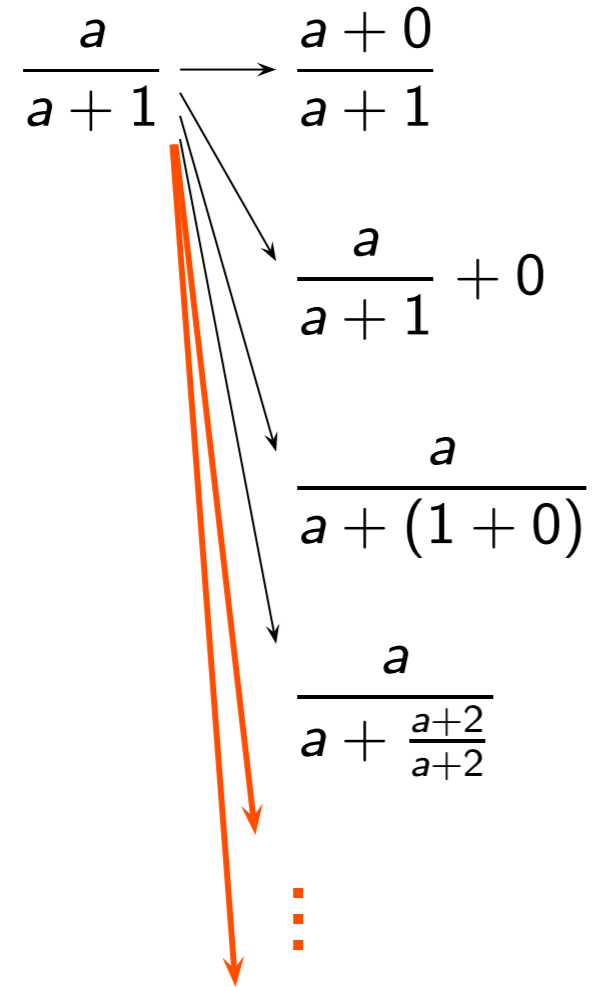
$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

# Introductory Example 2: Equations

---



$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$



## Introductory Example 2: Equations

---

$$1 + \frac{-1}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$1 + \frac{-1}{a+1} \longrightarrow \frac{a+1}{a+1} + \frac{-1}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$


$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$1 + \frac{-1}{a+1} \longrightarrow \frac{a+1}{a+1} + \frac{-1}{a+1}$$

$$\frac{a}{a} + \frac{-1}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$1 + \frac{-1}{a+1} \rightarrow \frac{a+1}{a+1} + \frac{-1}{a+1}$$
$$\frac{a}{a} + \frac{-1}{a+1}$$
$$1 + \frac{-1}{a + \frac{a}{a}}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

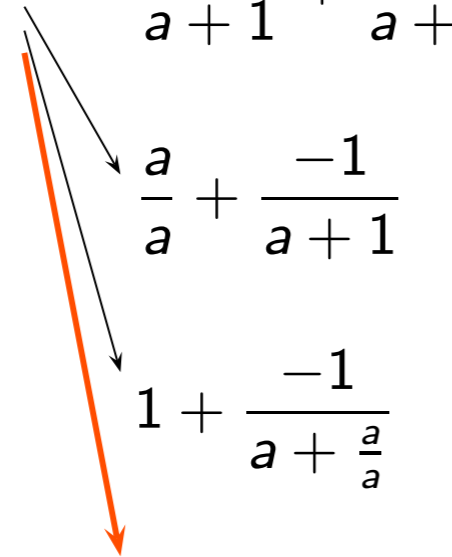
$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$1 + \frac{-1}{a+1} \rightarrow \frac{a+1}{a+1} + \frac{-1}{a+1}$$

$$\frac{a}{a} + \frac{-1}{a+1}$$
$$1 + \frac{-1}{a + \frac{a}{a}}$$
$$1 + \frac{-1+0}{a+1}$$

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

$$1 + \frac{-1}{a+1} \rightarrow \frac{a+1}{a+1} + \frac{-1}{a+1}$$
$$\frac{a}{a} + \frac{-1}{a+1}$$
$$1 + \frac{-1}{a + \frac{a}{a}}$$
$$1 + \frac{-1+0}{a+1}$$

⋮

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

Unrestricted application of equations leads to

- infinitely many equality chains,
- infinitely long equality chains.

⇒ The chance to reach the desired goal is very small.

In fact, the general problem is only recursively enumerable, but not decidable.

## Introductory Example 2: Equations

---

A better approach:

Apply equations in such a way that terms become “simpler”.

Start from both sides:

- 

-



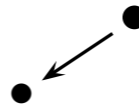
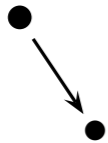
## Introductory Example 2: Equations

---

A better approach:

Apply equations in such a way that terms become “simpler”.

Start from both sides:



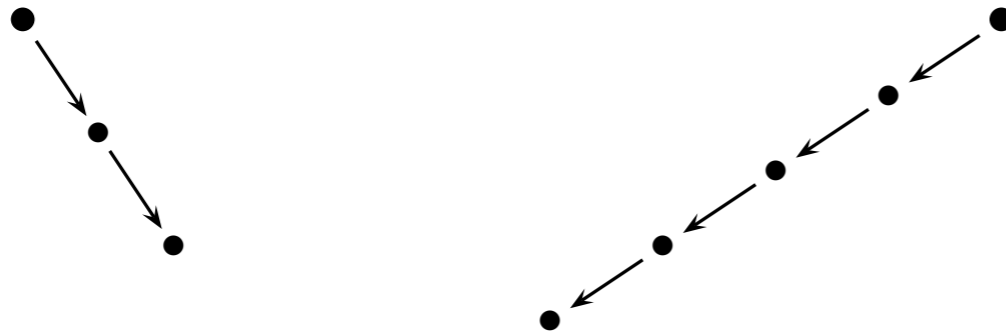
## Introductory Example 2: Equations

---

A better approach:

Apply equations in such a way that terms become “simpler”.

Start from both sides:



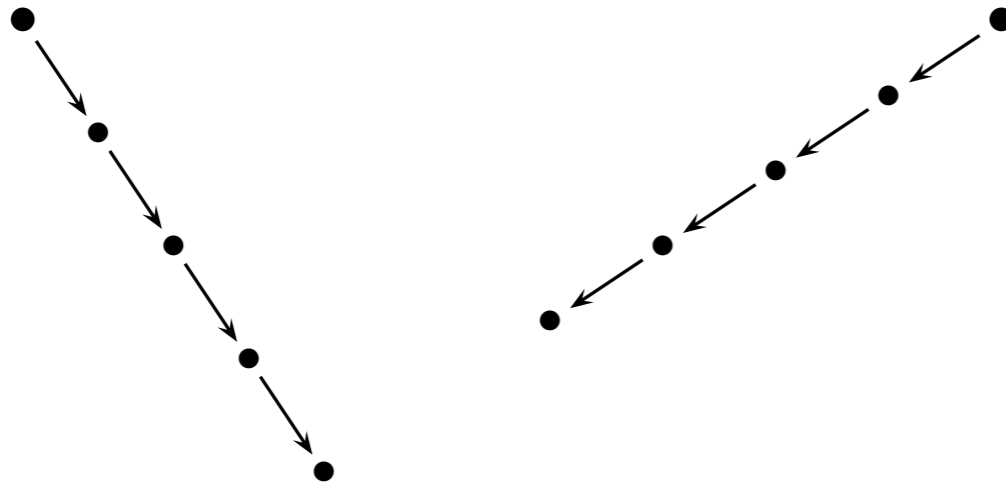
## Introductory Example 2: Equations

---

A better approach:

Apply equations in such a way that terms become “simpler”.

Start from both sides:



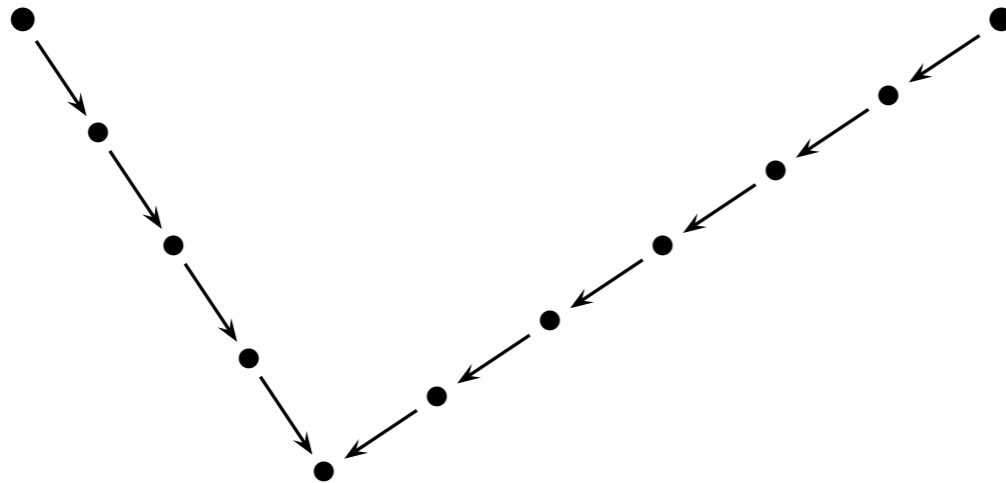
## Introductory Example 2: Equations

---

A better approach:

Apply equations in such a way that terms become “simpler”.

Start from both sides:



The terms are equal, if both derivations meet.

## Introductory Example 2: Equations

---

$$x + 0 = x \quad (1)$$

$$x + (-x) = 0 \quad (2)$$

$$x + (y + z) = (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x + y}{z} \quad (4)$$

$$\frac{x}{x} = 1 \quad (5)$$

## Introductory Example 2: Equations

---

Orient equations.

$$x + 0 \rightarrow x \quad (1)$$

$$x + (-x) \rightarrow 0 \quad (2)$$

$$x + (y + z) \rightarrow (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} \rightarrow \frac{x + y}{z} \quad (4)$$

$$\frac{x}{x} \rightarrow 1 \quad (5)$$

## Introductory Example 2: Equations

---

Orient equations.

Advantage:

Now there are only finitely many  
and finitely long derivations.

$$x + 0 \rightarrow x \quad (1)$$

$$x + (-x) \rightarrow 0 \quad (2)$$

$$x + (y + z) \rightarrow (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} \rightarrow \frac{x + y}{z} \quad (4)$$

$$\frac{x}{x} \rightarrow 1 \quad (5)$$

## Introductory Example 2: Equations

---

Orient equations.

But:

Now none of the equations is applicable to one of the terms

$$\frac{a}{a+1}, \quad 1 + \frac{-1}{a+1}$$

$$x + 0 \rightarrow x \quad (1)$$

$$x + (-x) \rightarrow 0 \quad (2)$$

$$x + (y + z) \rightarrow (x + y) + z \quad (3)$$

$$\frac{x}{z} + \frac{y}{z} \rightarrow \frac{x + y}{z} \quad (4)$$

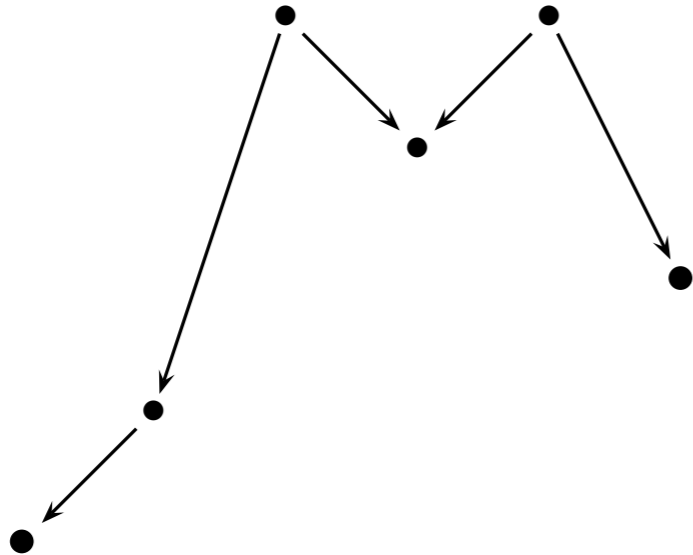
$$\frac{x}{x} \rightarrow 1 \quad (5)$$



## Introductory Example 2: Equations

---

The chain of equalities that we considered at the beginning looks roughly like this:

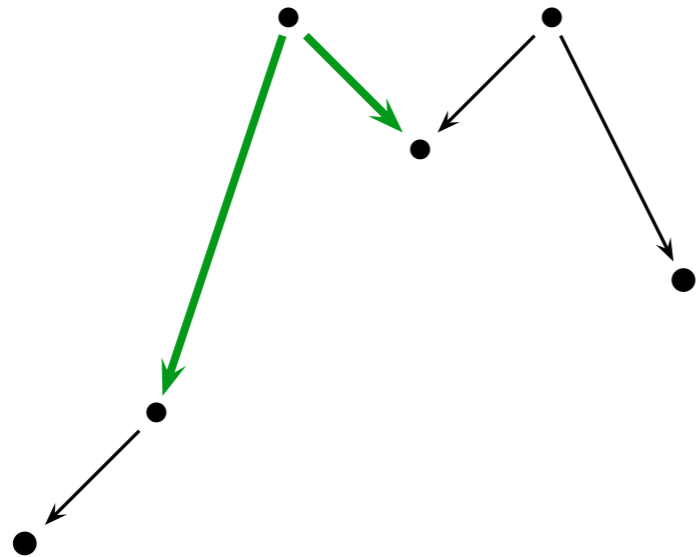


# Introductory Example 2: Equations

---

Idea:

Derive new equations that enable “shortcuts”.

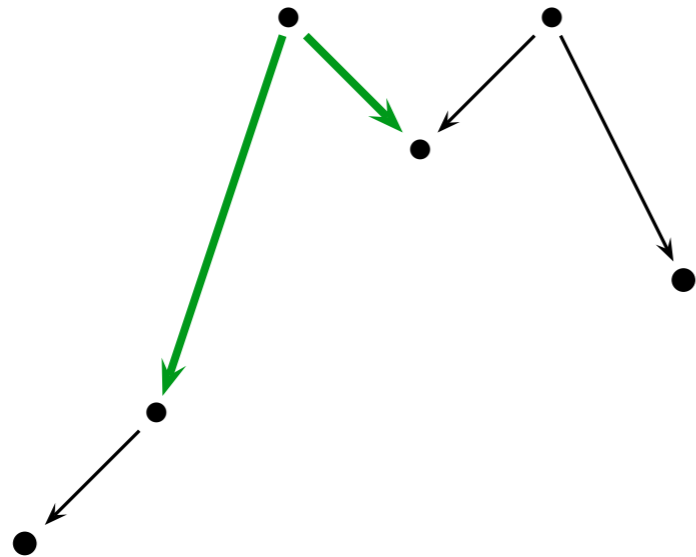


# Introductory Example 2: Equations

---

Idea:

Derive new equations that enable “shortcuts”.



From

$$x + (-x) \rightarrow 0 \quad (2)$$

$$x + (y + z) \rightarrow (x + y) + z \quad (3)$$

we derive

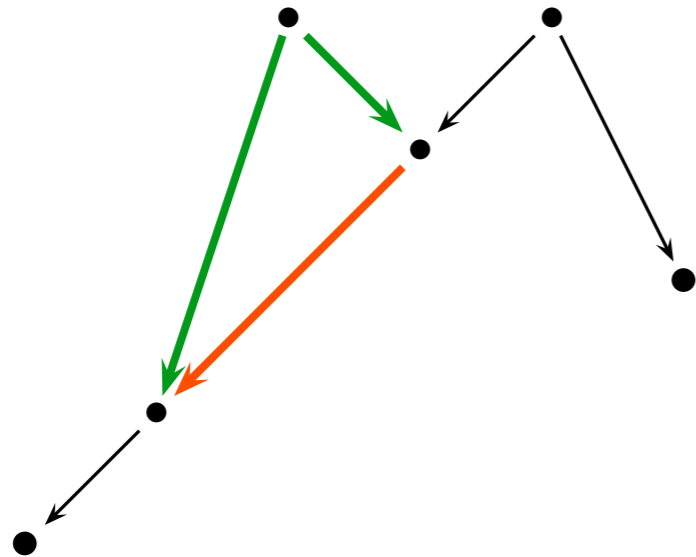
$$(x + y) + (-y) \rightarrow x + 0 \quad (6)$$

# Introductory Example 2: Equations

---

Idea:

Derive new equations that enable “shortcuts”.



From

$$x + (-x) \rightarrow 0 \quad (2)$$

$$x + (y + z) \rightarrow (x + y) + z \quad (3)$$

we derive

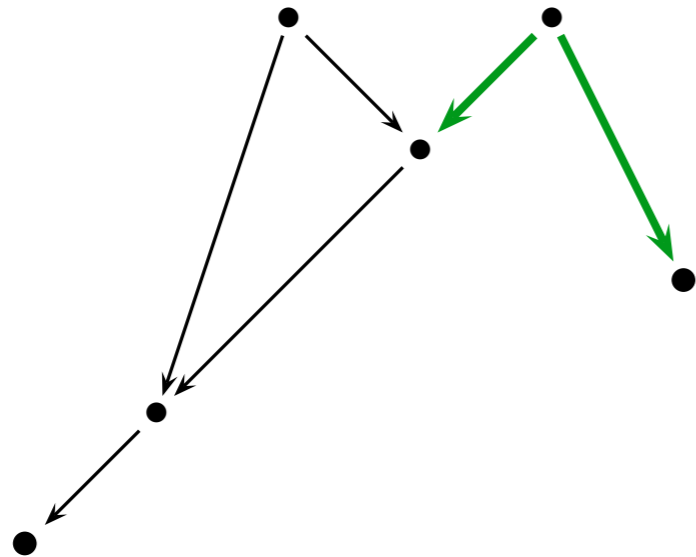
$$(x + y) + (-y) \rightarrow x + 0 \quad (6)$$

# Introductory Example 2: Equations

---

Idea:

Derive new equations that enable “shortcuts”.



From

$$\frac{x}{z} + \frac{y}{z} \rightarrow \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} \rightarrow 1 \quad (5)$$

we derive

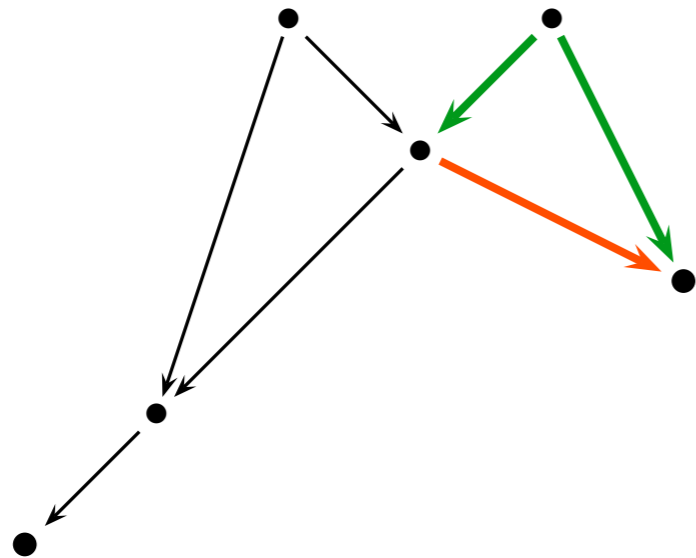
$$\frac{x+y}{x} \rightarrow 1 + \frac{y}{x} \quad (7)$$

# Introductory Example 2: Equations

---

Idea:

Derive new equations that enable “shortcuts”.



From

$$\frac{x}{z} + \frac{y}{z} \rightarrow \frac{x+y}{z} \quad (4)$$

$$\frac{x}{x} \rightarrow 1 \quad (5)$$

we derive

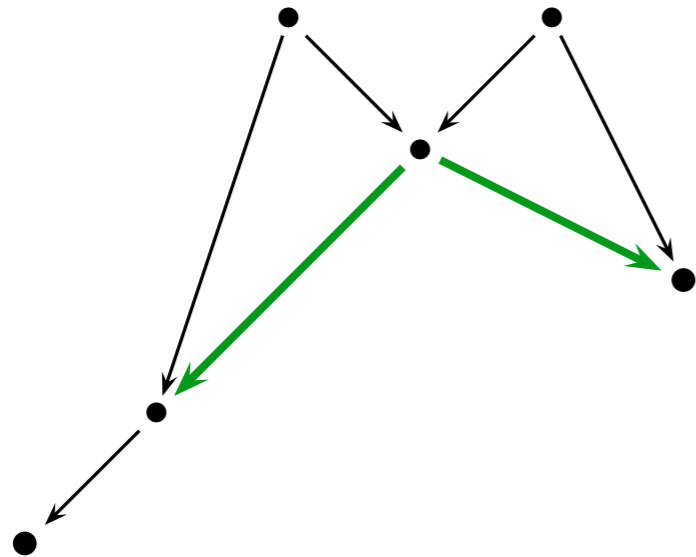
$$\frac{x+y}{x} \rightarrow 1 + \frac{y}{x} \quad (7)$$

# Introductory Example 2: Equations

---

Idea:

Derive new equations that enable “shortcuts”.



From

$$(x + y) + (-y) \rightarrow x + 0 \quad (6)$$

$$\frac{x + y}{x} \rightarrow 1 + \frac{y}{x} \quad (7)$$

we derive

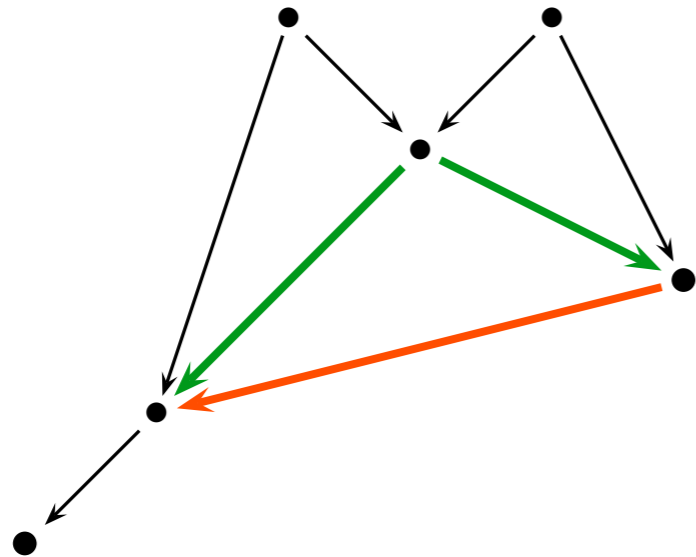
$$1 + \frac{-y}{x + y} \rightarrow \frac{x + 0}{x + y} \quad (8)$$

# Introductory Example 2: Equations

---

Idea:

Derive new equations that enable “shortcuts”.



From

$$(x + y) + (-y) \rightarrow x + 0 \quad (6)$$

$$\frac{x + y}{x} \rightarrow 1 + \frac{y}{x} \quad (7)$$

we derive

$$1 + \frac{-y}{x + y} \rightarrow \frac{x + 0}{x + y} \quad (8)$$

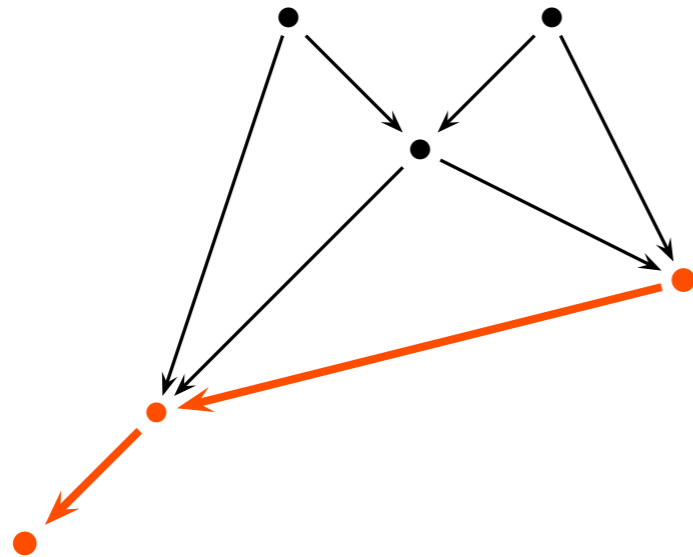


## Introductory Example 2: Equations

---

Idea:

Derive new equations that enable “shortcuts”.



Using these equations we can get a  
chain of equalities of the desired form.

## Introductory Example 2: Equations

---

In fact, it is not necessary to know some equational proof for the problem in advance.

We can derive these shortcut equations just by looking at the existing equation set.

How? See part 4 of this lecture.

## Result

---

Thomas Hiltenbrand's Waldmeister prover solves the problem in a few milliseconds.

## Result

---

But it's not the solution that we wanted to get!

We have to be more careful in formulating our axioms:

⇒ Exclude division by zero.

Then we get in fact a “real” proof.

## Result

---

So it works, but it looks like a lot of effort for a problem that one can solve with a little bit of highschool mathematics.

Reason: Pupils learn not only axioms, but also recipes to work efficiently with these axioms.

## Result

---

It makes a huge difference whether we work with well-known axioms

$$x + 0 = x$$

$$x + (-x) = 0$$

or with “new” unknown ones

$\forall Agent \ \forall Message \ \forall Key.$

$knows(Agent, crypt(Message, Key))$

$\wedge knows(Agent, Key)$

$\rightarrow knows(Agent, Message).$

# Result

---

This difference is also important for automated reasoning:

- For axioms that are well-known and frequently used, we can develop optimal specialized methods.
  - ⇒ Computer Algebra
  - ⇒ Automated Reasoning II (next semester)
- For new axioms, we have to develop methods that do “something reasonable” for arbitrary formulas.
  - ⇒ this lecture
- Combining the two approaches
  - ⇒ Automated Reasoning II

# First-order Provers in Practice

---

Real-life application:

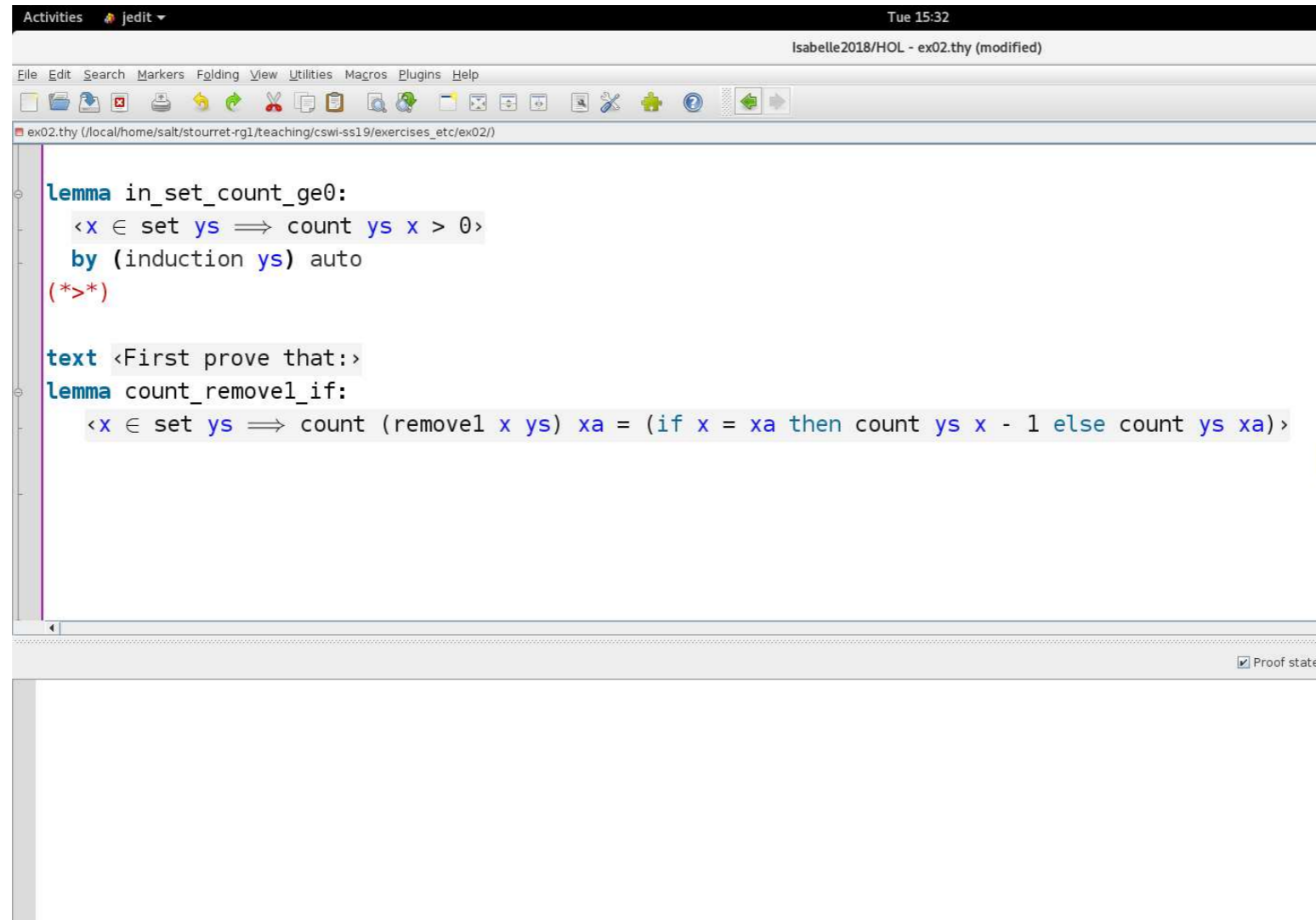
Use general-purpose provers to make interactive proof assistants more automatic:

Isabelle tactic “Sledgehammer”.



# First-order Provers in Practice

---



The screenshot shows a jedit editor window with the following content:

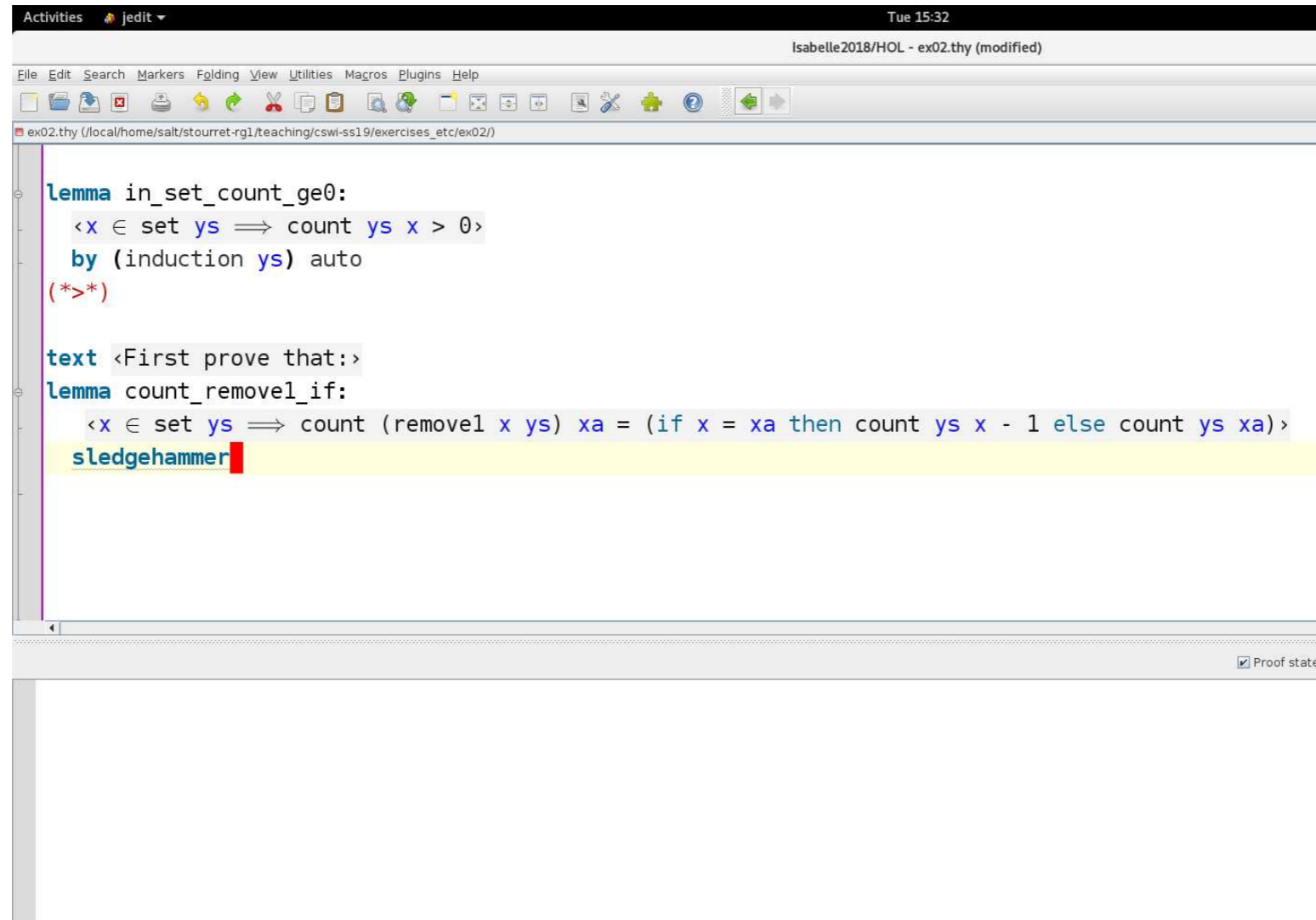
```
Activities  jedit  Tue 15:32
Isabelle2018/HOL - ex02.thy (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
ex02.thy (/local/home/salt/stouret-rg1/teaching/cswi-ss19/exercises_etc/ex02/)

lemma in_set_count_ge0:
  <x ∈ set ys ⇒ count ys x > 0>
  by (induction ys) auto
  (*>*)

text <First prove that:>
lemma count_remove1_if:
  <x ∈ set ys ⇒ count (remove1 x ys) xa = (if x = xa then count ys x - 1 else count ys xa)>
```

At the bottom right of the editor window, there is a checkbox labeled "Proof state" which is checked.

# First-order Provers in Practice



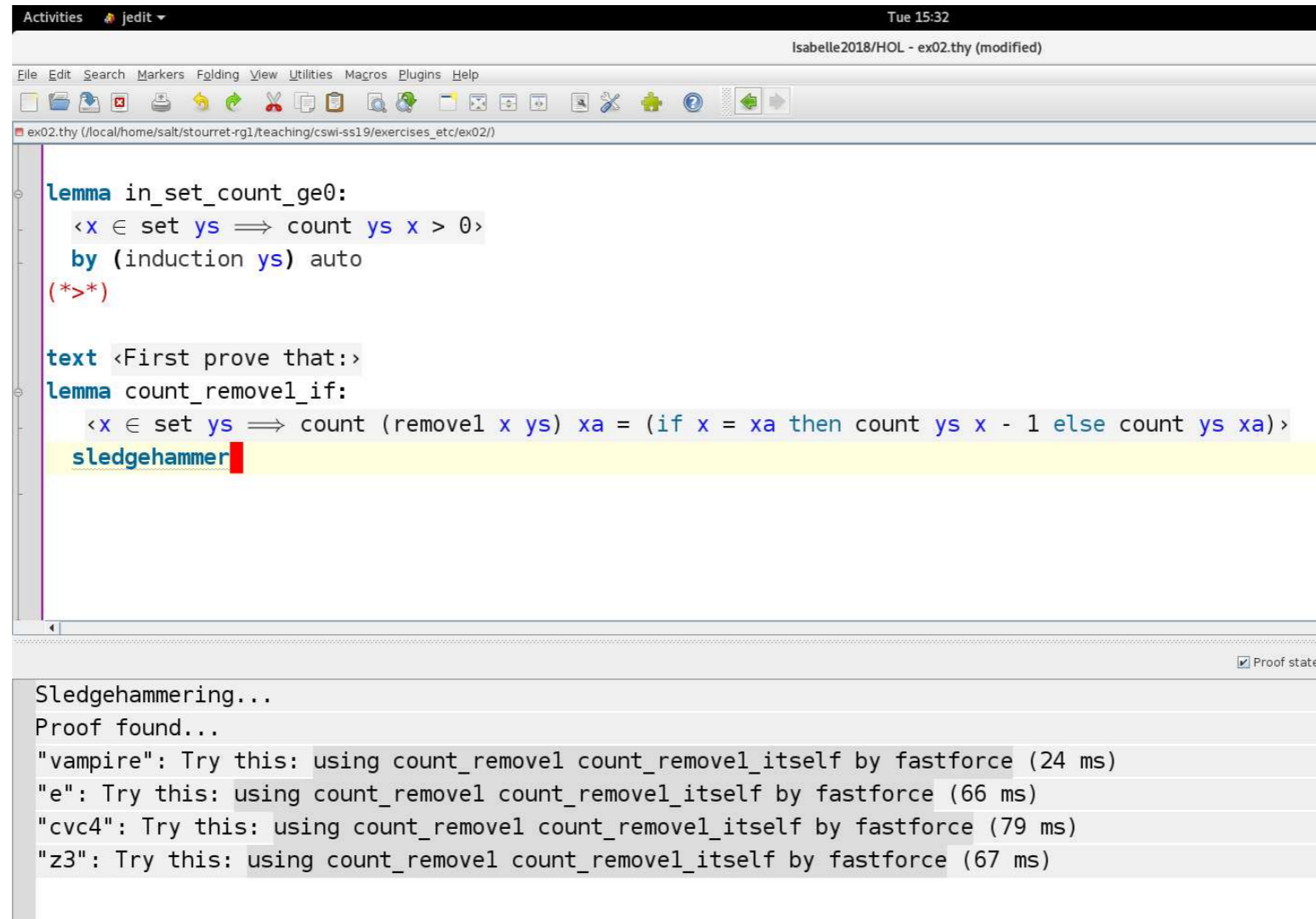
```
Activities jedit Tue 15:32
Isabelle2018/HOL - ex02.thy (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
ex02.thy (/local/home/salt/stouret-rg1/teaching/cswi-ss19/exercises_etc/ex02/)

lemma in_set_count_ge0:
  <x ∈ set ys ⇒ count ys x > 0>
  by (induction ys) auto
  (*>*)

text <First prove that:>
lemma count_remove1_if:
  <x ∈ set ys ⇒ count (remove1 x ys) xa = (if x = xa then count ys x - 1 else count ys xa)>
  sledgehammer
```

Proof state

# First-order Provers in Practice



The screenshot shows a theorem prover interface with a menu bar (File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins, Help) and a toolbar. The main window displays the following code:

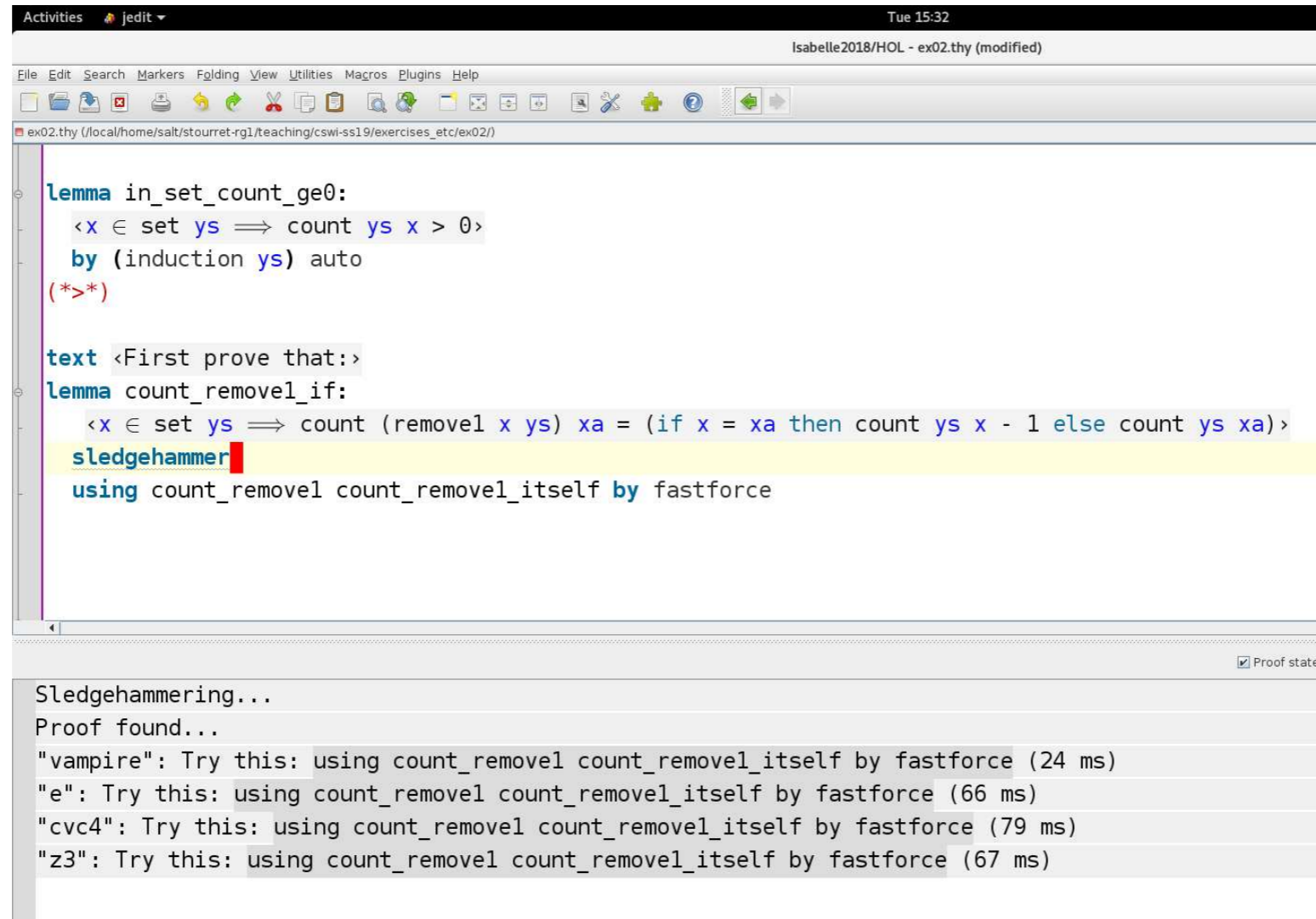
```
lemma in_set_count_ge0:
  <x ∈ set ys ⇒ count ys x > 0>
  by (induction ys) auto
  (*>*)

text <First prove that:>
lemma count_remove_if:
  <x ∈ set ys ⇒ count (remove x ys) xa = (if x = xa then count ys x - 1 else count ys xa)>
  sledgehammer
```

The output pane at the bottom shows the following text:

```
Proof state
Sledgehammering...
Proof found...
"vampire": Try this: using count_remove count_remove_itself by fastforce (24 ms)
"e": Try this: using count_remove count_remove_itself by fastforce (66 ms)
"cvc4": Try this: using count_remove count_remove_itself by fastforce (79 ms)
"z3": Try this: using count_remove count_remove_itself by fastforce (67 ms)
```

# First-order Provers in Practice



```
Activities jedit Tue 15:32
Isabelle2018/HOL - ex02.thy (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
ex02.thy (/local/home/salt/stouret-rg1/teaching/cswi-ss19/exercises_etc/ex02/)

lemma in_set_count_ge0:
  <x ∈ set ys ⇒ count ys x > 0>
  by (induction ys) auto
  (*>*)

text <First prove that:>
lemma count_remove_if:
  <x ∈ set ys ⇒ count (remove x ys) xa = (if x = xa then count ys x - 1 else count ys xa)>
  sledgehammer
  using count_remove count_remove_itself by fastforce

Sledgehammering...
Proof found...
"vampire": Try this: using count_remove count_remove_itself by fastforce (24 ms)
"e": Try this: using count_remove count_remove_itself by fastforce (66 ms)
"cvc4": Try this: using count_remove count_remove_itself by fastforce (79 ms)
"z3": Try this: using count_remove count_remove_itself by fastforce (67 ms)
```

# First-order Provers in Practice

---

Real-life application:

Use general-purpose provers to make interactive proof assistants more automatic:

Isabelle tactic “Sledgehammer”.

⇒ 70% of all subgoals are solved automatically.

# Topics of the Course

---

## Preliminaries

- abstract reduction systems
- well-founded orderings

## Propositional logic

- syntax, semantics
- calculi: CDCL-procedure, OBDDs
- implementation: Two watched literals

# Topics of the Course

---

## First-order predicate logic

syntax, semantics, model theory, . . .

calculi: resolution, tableaux

implementation: sharing, indexing

## First-order predicate logic with equality

term rewriting systems

calculi: Knuth-Bendix completion, dependency pairs

# Topics of the Course

---

Emphasis on:

logics and their properties,

proof systems for these logics and their properties:

soundness, completeness, implementation



# Part 1: Preliminaries

---

Literature:

Franz Baader and Tobias Nipkow: *Term rewriting and all that*,  
Cambridge Univ. Press, 1998, Chapter 2.

Before we start with the main subjects of the lecture, we repeat some prerequisites from mathematics and computer science and introduce some tools that we will need throughout the lecture.

# 1.1 Mathematical Prerequisites

---

$\mathbb{N} = \{0, 1, 2, \dots\}$  is the set of natural numbers (including 0).

$\mathbb{Z}$ ,  $\mathbb{Q}$ ,  $\mathbb{R}$  denote the integers, rational numbers and the real numbers, respectively.

$\emptyset$  is the empty set.

If  $M$  and  $M'$  are sets, then  $M \cap M'$ ,  $M \cup M'$ , and  $M \setminus M'$  denote the intersection, union, and set difference of  $M$  and  $M'$ .

The subset relation is denoted by  $\subseteq$ . The strict subset relation is denoted by  $\subset$  (i. e.,  $M \subset M'$  if and only if  $M \subseteq M'$  and  $M \neq M'$ ).

# Relations

---

Let  $M$  be a set, let  $n \geq 2$ .

We write  $M^n$  for the  $n$ -fold cartesian product  $M \times \cdots \times M$ .

In order to handle the cases  $n \geq 2$ ,  $n = 1$ , and  $n = 0$  simultaneously, we also define  $M^1 = M$  and  $M^0 = \{()\}$ .

(We do not distinguish between an element  $m$  of  $M$  and a 1-tuple  $(m)$  of an element of  $M$ .)

# Relations

---

An  $n$ -ary **relation**  $R$  over some set  $M$  is a subset of  $M^n$ :  $R \subseteq M^n$ .

We often use predicate notation for relations:

Instead of  $(m_1, \dots, m_n) \in R$  we write  $R(m_1, \dots, m_n)$ ,  
and say that  $R(m_1, \dots, m_n)$  holds or is true.

For binary relations, we often use infix notation, so

$$(m, m') \in < \Leftrightarrow <(m, m') \Leftrightarrow m < m'.$$

# Relations

---

Since relations are sets, we can use the usual set operations for them.

Example:

Let  $R = \{(0, 2), (1, 2), (2, 2), (3, 2)\} \subseteq \mathbb{N} \times \mathbb{N}$ .

Then  $R \cap < = R \cap \{(n, m) \in \mathbb{N} \times \mathbb{N} \mid n < m\}$   
 $= \{(0, 2), (1, 2)\}$ .

A relation  $Q$  is a **subrelation** of a relation  $R$  if  $Q \subseteq R$ .

# Words

---

Given a non-empty set (also called **alphabet**)  $\Sigma$ ,  
the set  $\Sigma^*$  of **finite words** over  $\Sigma$  is defined inductively by

- (i) the empty word  $\varepsilon$  is in  $\Sigma^*$ ,
- (ii) if  $u \in \Sigma^*$  and  $a \in \Sigma$  then  $ua$  is in  $\Sigma^*$ .

The set of **non-empty finite words**  $\Sigma^+$  is  $\Sigma^* \setminus \{\varepsilon\}$ .

The **concatenation** of two words  $u, v \in \Sigma^*$  is denoted by  $uv$ .

# Words

---

The length  $|u|$  of a word  $u \in \Sigma^*$  is defined by

(i)  $|\varepsilon| := 0,$

(ii)  $|ua| := |u| + 1$  for any  $u \in \Sigma^*$  and  $a \in \Sigma.$

## 1.2 Abstract Reduction Systems

---

Throughout the lecture, we will have to work with reduction systems, on the object level, in particular in the section on equality, and on the meta level, i. e., to describe deduction calculi.



# Abstract Reduction Systems

---

An **abstract reduction system** is a pair  $(A, \rightarrow)$ , where

$A$  is a non-empty set,

$\rightarrow \subseteq A \times A$  is a binary relation on  $A$ .

The relation  $\rightarrow$  is usually written in infix notation, i. e.,

$a \rightarrow b$  instead of  $(a, b) \in \rightarrow$ .

# Abstract Reduction Systems

---

Let  $\rightarrow' \subseteq A \times A$  and  $\rightarrow'' \subseteq A \times A$  be two binary relations. Then the **composition of  $\rightarrow'$  and  $\rightarrow''$**  is the binary relation  $(\rightarrow' \circ \rightarrow'') \subseteq A \times A$  defined by

$a (\rightarrow' \circ \rightarrow'') c$  if and only if

there exists some  $b \in A$  such that  $a \rightarrow' b$  and  $b \rightarrow'' c$ .

# Abstract Reduction Systems

---

$\rightarrow^0$	$= \{ (a, a) \mid a \in A \}$	identity
$\rightarrow^{i+1}$	$= \rightarrow^i \circ \rightarrow$	$i + 1$ -fold composition
$\rightarrow^+$	$= \bigcup_{i>0} \rightarrow^i$	transitive closure
$\rightarrow^*$	$= \bigcup_{i \geq 0} \rightarrow^i = \rightarrow^+ \cup \rightarrow^0$	reflexive transitive closure
$\rightarrow^=$	$= \rightarrow \cup \rightarrow^0$	reflexive closure
$\leftarrow$	$= \rightarrow^{-1} = \{ (b, c) \mid c \rightarrow b \}$	inverse
$\leftrightarrow$	$= \rightarrow \cup \leftarrow$	symmetric closure
$\leftrightarrow^+$	$= (\leftrightarrow)^+$	transitive symmetric closure
$\leftrightarrow^*$	$= (\leftrightarrow)^*$	reflexive transitive symmetric closure or equivalence closure

# Abstract Reduction Systems

---

$b \in A$  is **reducible**, if there is a  $c$  such that  $b \rightarrow c$ .

$b$  is **in normal form** (or **irreducible**), if it is not reducible.

$c$  is a **normal form of  $b$** , if  $b \rightarrow^* c$  and  $c$  is in normal form.

Notation:  $c = b \downarrow$  (if the normal form of  $b$  is unique).

# Abstract Reduction Systems

---

A relation  $\rightarrow$  is called

**terminating**, if there is no infinite descending chain

$$b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow \dots$$

**normalizing**, if every  $b \in A$  has a normal form.

# Abstract Reduction Systems

---

Lemma 1.1:

If  $\rightarrow$  is terminating, then it is normalizing.

Note: The reverse implication does not hold.

## 1.3 Orderings

---

Important properties of binary relations:

Let  $M \neq \emptyset$ . A binary relation  $R \subseteq M \times M$  is called

**reflexive**, if  $R(x, x)$  for all  $x \in M$ ,

**irreflexive**, if  $\neg R(x, x)$  for all  $x \in M$ ,

**antisymmetric**, if  $R(x, y)$  and  $R(y, x)$  imply  $x = y$   
for all  $x, y \in M$ ,

**transitive**, if  $R(x, y)$  and  $R(y, z)$  imply  $R(x, z)$   
for all  $x, y, z \in M$ ,

**total**, if  $R(x, y)$  or  $R(y, x)$  or  $x = y$  for all  $x, y \in M$ .

# Orderings

---

A **strict partial ordering**  $\succ$  on a set  $M \neq \emptyset$  is a transitive and irreflexive binary relation on  $M$ .

Notation:

$\prec$  for the inverse relation  $\succ^{-1}$

$\preceq$  for the reflexive closure  $(\succ \cup =)$  of  $\succ$



# Orderings

---

Let  $\succ$  be a strict partial ordering on  $M$ ; let  $M' \subseteq M$ .

$a \in M'$  is called **minimal in  $M'$** , if there is no  $b \in M'$  with  $a \succ b$ .

$a \in M'$  is called **smallest in  $M'$** , if  $b \succ a$  for all  $b \in M' \setminus \{a\}$ .

Analogously:

$a \in M'$  is called **maximal in  $M'$** , if there is no  $b \in M'$  with  $a \prec b$ .

$a \in M'$  is called **largest in  $M'$** , if  $b \prec a$  for all  $b \in M' \setminus \{a\}$ .

# Orderings

---

Notation:

$$M^{\prec x} = \{ y \in M \mid y \prec x \},$$

$$M^{\preceq x} = \{ y \in M \mid y \preceq x \}.$$

A subset  $M' \subseteq M$  is called **downward closed**, if  $x \in M'$  and  $x \succ y$  implies  $y \in M'$ .

# Well-Foundedness

---

Termination of reduction systems is strongly related to the concept of well-founded orderings.

A strict partial ordering  $\succ$  on  $M$  is called **well-founded (or Noetherian)**, if there is no infinite descending chain

$a_0 \succ a_1 \succ a_2 \succ \dots$  with  $a_i \in M$ .

# Well-Foundedness and Termination

---

Lemma 1.2:

If  $\succ$  is a well-founded partial ordering and  $\rightarrow \subseteq \succ$ ,  
then  $\rightarrow$  is terminating.

Lemma 1.3:

If  $\rightarrow$  is a terminating binary relation over  $A$ ,  
then  $\rightarrow^+$  is a well-founded partial ordering.

# Well-Founded Orderings: Examples

---

**Natural numbers:**  $(\mathbb{N}, >)$

**Lexicographic orderings:** Let  $(M_1, \succ_1), (M_2, \succ_2)$  be well-founded orderings. Define their **lexicographic combination**

$$\succ = (\succ_1, \succ_2)_{\text{lex}}$$

on  $M_1 \times M_2$  by

$$(a_1, a_2) \succ (b_1, b_2) \quad :\Leftrightarrow \quad a_1 \succ_1 b_1 \text{ or } (a_1 = b_1 \text{ and } a_2 \succ_2 b_2)$$

(analogously for more than two orderings). This again yields a well-founded ordering (proof below).

# Well-Founded Orderings: Examples

---

**Length-based ordering on words:** For alphabets  $\Sigma$  with a well-founded ordering  $>_{\Sigma}$ , the relation  $\succ$  defined as

$$w \succ w' \quad :\Leftrightarrow \quad |w| > |w'| \text{ or } (|w| = |w'| \text{ and } w >_{\Sigma, \text{lex}} w')$$

is a well-founded ordering on the set  $\Sigma^*$  of finite words over the alphabet  $\Sigma$  (Exercise).

**Counterexamples:**

$(\mathbb{Z}, >)$

$(\mathbb{N}, <)$

the lexicographic ordering on  $\Sigma^*$

# Basic Properties of Well-Founded Orderings

---

Lemma 1.4:

$(M, \succ)$  is well-founded if and only if every non-empty  $M' \subseteq M$  has a minimal element.

Lemma 1.5:

$(M_1, \succ_1)$  and  $(M_2, \succ_2)$  are well-founded if and only if  $(M_1 \times M_2, \succ)$  with  $\succ = (\succ_1, \succ_2)_{\text{lex}}$  is well-founded.

# Monotone Mappings

---

Let  $(M, \succ)$  and  $(M', \succ')$  be strict partial orderings.

A mapping  $\varphi : M \rightarrow M'$  is called **monotone**,  
if  $a \succ b$  implies  $\varphi(a) \succ' \varphi(b)$  for all  $a, b \in M$ .

Lemma 1.6:

If  $\varphi$  is a monotone mapping from  $(M, \succ)$  to  $(M', \succ')$   
and  $(M', \succ')$  is well-founded, then  $(M, \succ)$  is well-founded.



# Well-founded Induction

---

Theorem 1.7 (Well-founded (or Noetherian) Induction):

Let  $(M, \succ)$  be a well-founded ordering, let  $Q$  be a property of elements of  $M$ .

If for all  $m \in M$  the implication

if  $Q(m')$  for all  $m' \in M$  such that  $m \succ m'$ ,<sup>a</sup>  
then  $Q(m)$ .<sup>b</sup>

is satisfied, then the property  $Q(m)$  holds for all  $m \in M$ .

---

<sup>a</sup>induction hypothesis

<sup>b</sup>induction step

# Well-founded Recursion

---

Let  $M$  and  $S$  be sets, let  $N \subseteq M$ , and let  $f : M \rightarrow S$  be a function. Then the **restriction** of  $f$  to  $N$ , denoted by  $f|_N$ , is a function from  $N$  to  $S$  with  $f|_N(x) = f(x)$  for all  $x \in N$ .

Theorem 1.8 (**Well-founded (or Noetherian) Recursion**):

Let  $(M, \succ)$  be a well-founded ordering, let  $S$  be a set. Let  $\phi$  be a binary function that takes two arguments  $x$  and  $g$  and maps them to an element of  $S$ , where  $x \in M$  and  $g$  is a function from  $M^{\prec x}$  to  $S$ .

Then there exists exactly one function  $f : M \rightarrow S$  such that for all  $x \in M$

$$f(x) = \phi(x, f|_{M^{\prec x}})$$

# Well-founded Recursion

---

The well-founded recursion scheme generalizes terminating recursive programs.

Note that functions defined by well-founded recursion need *not* be computable, in particular since for many well-founded orderings the sets  $M^{\prec^x}$  may be infinite.

## 1.4 Multisets

---

Let  $M$  be a set. A **multiset**  $S$  over  $M$  is a mapping  $S : M \rightarrow \mathbb{N}$ . We interpret  $S(m)$  as the number of occurrences of elements  $m$  of the base set  $M$  within the multiset  $S$ .

*Example.*  $S = \{a, a, a, b, b\}$  is a multiset over  $\{a, b, c\}$ , where  $S(a) = 3$ ,  $S(b) = 2$ ,  $S(c) = 0$ .

We say that  $m$  is an **element** of  $S$ , if  $S(m) > 0$ .

# Multisets

---

We use set notation ( $\in$ ,  $\subseteq$ ,  $\cup$ ,  $\cap$ , etc.) with analogous meaning also for multisets, e. g.,

$$m \in S \quad :\Leftrightarrow \quad S(m) > 0$$

$$(S_1 \cup S_2)(m) \quad := \quad S_1(m) + S_2(m)$$

$$(S_1 \cap S_2)(m) \quad := \quad \min\{S_1(m), S_2(m)\}$$

$$(S_1 - S_2)(m) \quad := \quad \begin{cases} S_1(m) - S_2(m) & \text{if } S_1(m) \geq S_2(m) \\ 0 & \text{otherwise} \end{cases}$$

$$S_1 \subseteq S_2 \quad :\Leftrightarrow \quad S_1(m) \leq S_2(m) \text{ for all } m \in M$$

# Multisets

---

A multiset  $S$  is called **finite**, if the set

$$\{ m \in M \mid S(m) > 0 \}$$

is finite.

*From now on we only consider finite multisets.*

# Multiset Orderings

---

Let  $(M, \succ)$  be an abstract reduction system. The **multiset extension** of  $\succ$  to multisets over  $M$  is defined by

$S_1 \succ_{\text{mul}} S_2$  if and only if

there exist multisets  $X$  and  $Y$  over  $M$  such that

$$\emptyset \neq X \subseteq S_1,$$

$$S_2 = (S_1 - X) \cup Y,$$

$$\forall y \in Y \exists x \in X: x \succ y$$

# Multiset Orderings

---

Lemma 1.9 (König's Lemma):

Every finitely branching tree with infinitely many nodes contains an infinite path.

Theorem 1.10:

- (a) If  $\succ$  is transitive, then  $\succ_{mul}$  is transitive.
- (b) If  $\succ$  is irreflexive and transitive, then  $\succ_{mul}$  is irreflexive.
- (c) If  $\succ$  is a well-founded ordering, then  $\succ_{mul}$  is a well-founded ordering.
- (d) If  $\succ$  is a strict total ordering, then  $\succ_{mul}$  is a strict total ordering.



# Multiset Orderings

---

The multiset extension as defined above is due to Dershowitz and Manna (1979).

There are several other ways to characterize the multiset extension of a binary relation. The following one is due to Huet and Oppen (1980):

# Multiset Orderings

---

Let  $(M, \succ)$  be an abstract reduction system. The (Huet/Oppen) multiset extension of  $\succ$  to multisets over  $M$  is defined by

$S_1 \succ_{\text{mul}}^{\text{HO}} S_2$  if and only if

$S_1 \neq S_2$  and

$\forall m \in M: (S_2(m) > S_1(m))$

$\Rightarrow \exists m' \in M: m' \succ m \text{ and } S_1(m') > S_2(m')$

# Multiset Orderings

---

A third way to characterize the multiset extension of a binary relation  $\succ$  is to define it as the transitive closure of the relation  $\succ_{\text{mul}}^1$  given by

$S_1 \succ_{\text{mul}}^1 S_2$  if and only if

there exists  $x \in S_1$  and a multiset  $Y$  over  $M$  such that

$$S_2 = (S_1 - \{x\}) \cup Y,$$

$$\forall y \in Y: x \succ y$$

# Multiset Orderings

---

For strict partial orderings all three characterizations of  $\succ_{\text{mul}}$  are equivalent:

Theorem 1.11:

If  $\succ$  is a strict partial ordering, then

(a)  $\succ_{\text{mul}} = \succ_{\text{mul}}^{\text{HO}}$ ,

(b)  $\succ_{\text{mul}} = (\succ_{\text{mul}}^1)^+$ .

Note, however, that for an arbitrary binary relation  $\succ$  all three relations  $\succ_{\text{mul}}$ ,  $\succ_{\text{mul}}^{\text{HO}}$ , and  $(\succ_{\text{mul}}^1)^+$  may be different.

## 1.5 Complexity Theory Prerequisites

---

A **decision problem** is a subset  $L \subseteq \Sigma^*$  for some fixed finite alphabet  $\Sigma$ .

The function  $\text{chr}(L, x)$  denotes the **characteristic function** for some decision problem  $L$  and is defined by  $\text{chr}(L, u) = 1$  if  $u \in L$  and  $\text{chr}(L, u) = 0$  otherwise.

# P and NP

---

A decision problem is called **solvable in polynomial time** if its characteristic function can be computed in polynomial time.

The class of all polynomial-time decision problems is denoted by **P**.

We say that a decision problem  $L$  is in **NP** if there is a predicate  $Q(x, y)$  and a polynomial  $p(n)$  such that for all  $u \in \Sigma^*$  we have

- (i)  $u \in L$  if and only if there is a  $v \in \Sigma^*$  with  $|v| \leq p(|u|)$  and  $Q(u, v)$  holds, and
- (ii) the predicate  $Q$  is in **P**.

# Reducibility, NP-Hardness, NP-Completeness

---

A decision problem  $L$  is **polynomial-time reducible** to a decision problem  $L'$  if there is a function  $g$  computable in polynomial time such that for all  $u \in \Sigma^*$  we have  $u \in L$  iff  $g(u) \in L'$ .

For example, if  $L$  is polynomial-time reducible to  $L'$  and  $L' \in P$  then  $L \in P$ .

A decision problem is **NP-hard** if every problem in NP is polynomial-time reducible to it.

A decision problem is **NP-complete** if it is NP-hard and in NP.

# Reducibility, NP-Hardness, NP-Completeness

The following properties are equivalent:

- (i) There exists some NP-complete problem that is in P.
- (ii)  $P = NP$ .

The question whether P equals NP or not is probably the most famous unsolved problem in theoretical computer science.

All known algorithms for NP-complete problems have an exponential time complexity in the worst case.



## Part 2: Propositional Logic

---

### Propositional logic

- logic of truth values,
- decidable (but NP-complete),
- can be used to describe functions over a finite domain,
- industry standard for many analysis/verification tasks (e. g., model checking).

## 2.1 Syntax

---

When we define a logic, we must define

how formulas of the logic look like (syntax),  
and what they mean (semantics).

We start with the syntax.

Propositional formulas are built from

- propositional variables,
- logical connectives (e. g.,  $\wedge$ ,  $\vee$ ).

# Propositional Variables

---

Let  $\Pi$  be a set of propositional variables.

We use letters  $P, Q, R, S$ , to denote propositional variables.

# Propositional Formulas

---

$F_{\Pi}$  is the set of propositional formulas over  $\Pi$  defined inductively as follows:

$F, G$	$::=$	$\perp$	(falsum)
		$\top$	(verum)
		$P, P \in \Pi$	(atomic formula)
		$(\neg F)$	(negation)
		$(F \wedge G)$	(conjunction)
		$(F \vee G)$	(disjunction)
		$(F \rightarrow G)$	(implication)
		$(F \leftrightarrow G)$	(equivalence)

# Propositional Formulas

---

Sometimes further connectives are used, for instance

$(F \leftarrow G)$  (reverse implication)

$(F \oplus G)$  (exclusive or)

(if  $F$  then  $G_1$  else  $G_0$ ) (if-then-else)

## Notational Conventions

---

As a notational convention we assume that  $\neg$  binds strongest, and we remove outermost parentheses, so  $\neg P \vee Q$  is actually a shorthand for  $((\neg P) \vee Q)$ .

Instead of  $((P \wedge Q) \wedge R)$  we simply write  $P \wedge Q \wedge R$  (analogously for  $\vee$ ).

For all other logical connectives we will use parentheses when needed.

# Formula Manipulation

---

Automated reasoning is very much formula manipulation.

We perform syntactic operations on formulas in order to show semantic properties of formulas.

# Formula Manipulation

---

To precisely describe the manipulation of a formula, we introduce positions.

A **position** is a word over  $\mathbb{N}$ .

The set of positions of a formula  $F$  is inductively defined by

$$\text{pos}(F) := \{\varepsilon\} \text{ if } F \in \{\top, \perp\} \text{ or } F \in \Pi$$

$$\text{pos}(\neg F) := \{\varepsilon\} \cup \{1p \mid p \in \text{pos}(F)\}$$

$$\text{pos}(F \circ G) := \{\varepsilon\} \cup \{1p \mid p \in \text{pos}(F)\} \cup \{2p \mid p \in \text{pos}(G)\}$$

$$\text{where } \circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}.$$



# Formula Manipulation

---

The prefix order  $\leq$  on positions is defined by  $p \leq q$  if there is some  $p'$  such that  $pp' = q$ .

Note that the prefix order is partial, e. g., the positions 12 and 21 are not comparable, they are “parallel”, see below.

By  $<$  we denote the strict part of  $\leq$ , that is,  $p < q$  if  $p \leq q$  but not  $q \leq p$ .

By  $\parallel$  we denote incomparable positions, that is,  $p \parallel q$  if neither  $p \leq q$  nor  $q \leq p$ .

We say that  $p$  is **above**  $q$  if  $p \leq q$ ,  $p$  is **strictly above**  $q$  if  $p < q$ , and  $p$  and  $q$  are **parallel** if  $p \parallel q$ .

# Formula Manipulation

---

The **size** of a formula  $F$  is given by the cardinality of  $\text{pos}(F)$ :  $|F| := |\text{pos}(F)|$ .

The **subformula** of  $F$  at position  $p \in \text{pos}(F)$  is recursively defined by

$$\begin{aligned} F|_{\varepsilon} &:= F \\ (\neg F)|_{1p} &:= F|_p \\ (F_1 \circ F_2)|_{ip} &:= F_i|_p \quad \text{where } i \in \{1, 2\} \\ &\quad \text{and } \circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}. \end{aligned}$$

# Formula Manipulation

---

Finally, the **replacement** of a subformula at position  $p \in \text{pos}(F)$  by a formula  $G$  is recursively defined by

$$F[G]_\varepsilon := G$$

$$(\neg F)[G]_{1p} := \neg(F[G]_p)$$

$$(F_1 \circ F_2)[G]_{1p} := (F_1[G]_p \circ F_2)$$

$$(F_1 \circ F_2)[G]_{2p} := (F_1 \circ F_2[G]_p)$$

where  $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ .

# Formula Manipulation

---

Example 2.1:

The set of positions for the formula  $F = (P \rightarrow Q) \rightarrow (P \wedge \neg R)$  is  $\text{pos}(F) = \{\varepsilon, 1, 11, 12, 2, 21, 22, 221\}$ .

The subformula at position 22 is  $F|_{22} = \neg R$  and replacing this formula by  $P \leftrightarrow Q$  results in  $F[P \leftrightarrow Q]_{22} = (P \rightarrow Q) \rightarrow (P \wedge (P \leftrightarrow Q))$ .

## 2.2 Semantics

---

In **classical logic** (dating back to Aristotle) there are “only” two truth values “true” and “false” which we shall denote, respectively, by 1 and 0.

There are **multi-valued logics** having more than two truth values.

# Valuations

---

A propositional variable has no intrinsic meaning. The meaning of a propositional variable has to be defined by a valuation.

A  $\Pi$ -valuation is a function

$$\mathcal{A} : \Pi \rightarrow \{0, 1\}$$

where  $\{0, 1\}$  is the set of truth values.

## Truth Value of a Formula in $\mathcal{A}$

---

Given a  $\Pi$ -valuation  $\mathcal{A} : \Pi \rightarrow \{0, 1\}$ , its extension to formulas  $\mathcal{A}^* : F_{\Pi} \rightarrow \{0, 1\}$  is defined inductively as follows:

$$\mathcal{A}^*(\perp) = 0$$

$$\mathcal{A}^*(\top) = 1$$

$$\mathcal{A}^*(P) = \mathcal{A}(P)$$

$$\mathcal{A}^*(\neg F) = 1 - \mathcal{A}^*(F)$$

$$\mathcal{A}^*(F \wedge G) = \min(\mathcal{A}^*(F), \mathcal{A}^*(G))$$

$$\mathcal{A}^*(F \vee G) = \max(\mathcal{A}^*(F), \mathcal{A}^*(G))$$

$$\mathcal{A}^*(F \rightarrow G) = \max(1 - \mathcal{A}^*(F), \mathcal{A}^*(G))$$

$$\mathcal{A}^*(F \leftrightarrow G) = \text{if } \mathcal{A}^*(F) = \mathcal{A}^*(G) \text{ then } 1 \text{ else } 0$$

## Truth Value of a Formula in $\mathcal{A}$

---

For simplicity, the extension  $\mathcal{A}^*$  of  $\mathcal{A}$  is usually also denoted by  $\mathcal{A}$ .

Note that formulas and truth values are disjoint classes of objects. Statements like  $P = 1$  or  $F \wedge G = 0$  that equate formulas and truth values are non-sensical.

A formula is never *equal to* a truth value, but it *has* a truth value in some valuation  $\mathcal{A}$ .



## 2.3 Models, Validity, and Satisfiability

---

Let  $F$  be a  $\Pi$ -formula.

We say that  $F$  is **true** in  $\mathcal{A}$  ( $\mathcal{A}$  is a **model** of  $F$ ;  $F$  is **valid** in  $\mathcal{A}$ ;  $F$  **holds** in  $\mathcal{A}$ ), written  $\mathcal{A} \models F$ , if  $\mathcal{A}(F) = 1$ .

We say that  $F$  is **valid** or that  $F$  is a **tautology**, written  $\models F$ , if  $\mathcal{A} \models F$  for all  $\Pi$ -valuations  $\mathcal{A}$ .

$F$  is called **satisfiable** if there exists an  $\mathcal{A}$  such that  $\mathcal{A} \models F$ .  
Otherwise  $F$  is called **unsatisfiable** (or **contradictory**).

# Entailment and Equivalence

---

$F$  entails (implies)  $G$  (or  $G$  is a consequence of  $F$ ), written  $F \models G$ , if for all  $\Pi$ -valuations  $\mathcal{A}$  we have

$$\text{if } \mathcal{A} \models F \text{ then } \mathcal{A} \models G,$$

or equivalently

$$\mathcal{A}(F) \leq \mathcal{A}(G).$$

$F$  and  $G$  are called **equivalent**, written  $F \equiv G$ , if for all  $\Pi$ -valuations  $\mathcal{A}$  we have

$$\mathcal{A} \models F \text{ if and only if } \mathcal{A} \models G,$$

or equivalently

$$\mathcal{A}(F) = \mathcal{A}(G).$$

# Entailment and Equivalence

---

$F$  and  $G$  are called **equisatisfiable**,  
if either both  $F$  and  $G$  are satisfiable, or both  $F$  and  $G$  are unsatisfiable.

# Entailment and Equivalence

---

The notions defined above for formulas, such as satisfiability, validity, or entailment, are extended to sets of formulas  $N$  by treating sets of formulas analogously to conjunctions of formulas, e. g.:

$\mathcal{A} \models N$  if  $\mathcal{A} \models G$  for all  $G \in N$ .

$N \models F$  if for all  $\Pi$ -valuations  $\mathcal{A}$ : if  $\mathcal{A} \models N$ , then  $\mathcal{A} \models F$ .

Note: Formulas are always finite objects; but sets of formulas may be infinite. Therefore, it is in general not possible to replace a set of formulas by the conjunction of its elements.

# Entailment and Equivalence

---

Proposition 2.2:

$F \models G$  if and only if  $\models (F \rightarrow G)$ .

Proposition 2.3:

$F \models\!\!\models G$  if and only if  $\models (F \leftrightarrow G)$ .

# Validity vs. Unsatisfiability

---

Validity and unsatisfiability of formulas are just two sides of the same medal as explained by the following proposition.

Proposition 2.4:

$F$  is valid if and only if  $\neg F$  is unsatisfiable.

Hence in order to design a theorem prover (validity checker) it is sufficient to design a checker for unsatisfiability.

# Validity vs. Unsatisfiability

---

In a similar way, entailment can be reduced to unsatisfiability and vice versa:

Proposition 2.5:

$G \models F$  if and only if  $G \wedge \neg F$  is unsatisfiable.

$N \models F$  if and only if  $N \cup \{\neg F\}$  is unsatisfiable.

Proposition 2.6:

$G \models \perp$  if and only if  $G$  is unsatisfiable.

$N \models \perp$  if and only if  $N$  is unsatisfiable.

## Checking Unsatisfiability

---

Every formula  $F$  contains only finitely many propositional variables. Obviously,  $\mathcal{A}(F)$  depends only on the values of those finitely many variables in  $F$  in  $\mathcal{A}$ .

If  $F$  contains  $n$  distinct propositional variables, then it is sufficient to check  $2^n$  valuations to see whether  $F$  is satisfiable or not  $\Rightarrow$  truth table.

So the satisfiability problem is clearly decidable (but, by Cook's Theorem, NP-complete).

Nevertheless, in practice, there are (much) better methods than truth tables to check the satisfiability of a formula.



# Replacement Theorem

---

Proposition 2.7:

Let  $\mathcal{A}$  be a valuation, let  $F$  and  $G$  be formulas, and let  $H = H[F]_p$  be a formula in which  $F$  occurs as a subformula at position  $p$ .

If  $\mathcal{A}(F) = \mathcal{A}(G)$ , then  $\mathcal{A}(H[F]_p) = \mathcal{A}(H[G]_p)$ .

Theorem 2.8:

Let  $F$  and  $G$  be equivalent formulas, let  $H = H[F]_p$  be a formula in which  $F$  occurs as a subformula at position  $p$ .

Then  $H[F]_p$  is equivalent to  $H[G]_p$ .

# Some Important Equivalences

---

Proposition 2.9:

The following equivalences hold for all formulas  $F, G, H$ :

$$(F \wedge F) \models F$$

$$(F \vee F) \models F$$

(Idempotency)

$$(F \wedge G) \models (G \wedge F)$$

$$(F \vee G) \models (G \vee F)$$

(Commutativity)

$$(F \wedge (G \wedge H)) \models ((F \wedge G) \wedge H)$$

$$(F \vee (G \vee H)) \models ((F \vee G) \vee H)$$

(Associativity)

$$(F \wedge (G \vee H)) \models ((F \wedge G) \vee (F \wedge H))$$

$$(F \vee (G \wedge H)) \models ((F \vee G) \wedge (F \vee H))$$

(Distributivity)

# Some Important Equivalences

---

The following equivalences hold for all formulas  $F, G, H$ :

$$(F \wedge (F \vee G)) \models F$$

$$(F \vee (F \wedge G)) \models F$$

(Absorption)

$$(\neg\neg F) \models F$$

(Double Negation)

$$\neg(F \wedge G) \models (\neg F \vee \neg G)$$

$$\neg(F \vee G) \models (\neg F \wedge \neg G)$$

(De Morgan's Laws)

$$(F \wedge G) \models F, \text{ if } G \text{ is a tautology}$$

$$(F \vee G) \models \top, \text{ if } G \text{ is a tautology}$$

$$(F \wedge G) \models \perp, \text{ if } G \text{ is unsatisfiable}$$

$$(F \vee G) \models F, \text{ if } G \text{ is unsatisfiable}$$

(Tautology Laws)

## Some Important Equivalences

---

The following equivalences hold for all formulas  $F, G, H$ :

$$(F \leftrightarrow G) \models ((F \rightarrow G) \wedge (G \rightarrow F))$$

$$(F \leftrightarrow G) \models ((F \wedge G) \vee (\neg F \wedge \neg G)) \quad (\text{Equivalence})$$

$$(F \rightarrow G) \models (\neg F \vee G) \quad (\text{Implication})$$

## An Important Entailment

---

Proposition 2.10:

The following entailment holds for all formulas  $F, G, H$ :

$$(F \vee H) \wedge (G \vee \neg H) \models F \vee G \quad (\text{Generalized Resolution})$$

## 2.4 Normal Forms

---

Many theorem proving calculi do not operate on arbitrary formulas, but only on some restricted class of formulas.

# Normal Forms

---

We define **conjunctions** of formulas as follows:

$$\bigwedge_{i=1}^0 F_i = \top.$$

$$\bigwedge_{i=1}^1 F_i = F_1.$$

$$\bigwedge_{i=1}^{n+1} F_i = \bigwedge_{i=1}^n F_i \wedge F_{n+1}.$$

and analogously **disjunctions**:

$$\bigvee_{i=1}^0 F_i = \perp.$$

$$\bigvee_{i=1}^1 F_i = F_1.$$

$$\bigvee_{i=1}^{n+1} F_i = \bigvee_{i=1}^n F_i \vee F_{n+1}.$$

# Literals and Clauses

---

A **literal** is either a propositional variable  $P$  or a negated propositional variable  $\neg P$ .

A **clause** is a (possibly empty) disjunction of literals.



# CNF and DNF

---

A formula is in **conjunctive normal form (CNF, clause normal form)**, if it is a conjunction of disjunctions of literals (or in other words, a conjunction of clauses).

A formula is in **disjunctive normal form (DNF)**, if it is a disjunction of conjunctions of literals.

Warning: definitions in the literature differ:

- are complementary literals permitted?

- are duplicated literals permitted?

- are empty disjunctions/conjunctions permitted?

## CNF and DNF

---

Checking the validity of CNF formulas or the unsatisfiability of DNF formulas is easy:

A formula in CNF is valid, if and only if each of its disjunctions contains a pair of complementary literals  $P$  and  $\neg P$ .

Conversely, a formula in DNF is unsatisfiable, if and only if each of its conjunctions contains a pair of complementary literals  $P$  and  $\neg P$ .

On the other hand, checking the unsatisfiability of CNF formulas or the validity of DNF formulas is known to be coNP-complete.

# Conversion to CNF/DNF

---

Proposition 2.11:

For every formula there is an equivalent formula in CNF (and also an equivalent formula in DNF).

Proof:

We describe a (naive) algorithm to convert a formula to CNF.

Apply the following rules as long as possible (modulo commutativity of  $\wedge$  and  $\vee$ ):

Step 1: Eliminate equivalences:

$$H[F \leftrightarrow G]_p \Rightarrow_{\text{CNF}} H[(F \rightarrow G) \wedge (G \rightarrow F)]_p$$

# Conversion to CNF/DNF

---

Step 2: Eliminate implications:

$$H[F \rightarrow G]_p \Rightarrow_{\text{CNF}} H[\neg F \vee G]_p$$

Step 3: Push negations downward:

$$H[\neg(F \vee G)]_p \Rightarrow_{\text{CNF}} H[\neg F \wedge \neg G]_p$$

$$H[\neg(F \wedge G)]_p \Rightarrow_{\text{CNF}} H[\neg F \vee \neg G]_p$$

Step 4: Eliminate multiple negations:

$$H[\neg\neg F]_p \Rightarrow_{\text{CNF}} H[F]_p$$

# Conversion to CNF/DNF

---

Step 5: Push disjunctions downward:

$$H[(F \wedge F') \vee G]_p \Rightarrow_{\text{CNF}} H[(F \vee G) \wedge (F' \vee G)]_p$$

Step 6: Eliminate  $\top$  and  $\perp$ :

$$H[F \wedge \top]_p \Rightarrow_{\text{CNF}} H[F]_p$$

$$H[F \wedge \perp]_p \Rightarrow_{\text{CNF}} H[\perp]_p$$

$$H[F \vee \top]_p \Rightarrow_{\text{CNF}} H[\top]_p$$

$$H[F \vee \perp]_p \Rightarrow_{\text{CNF}} H[F]_p$$

$$H[\neg \perp]_p \Rightarrow_{\text{CNF}} H[\top]_p$$

$$H[\neg \top]_p \Rightarrow_{\text{CNF}} H[\perp]_p$$

## Conversion to CNF/DNF

---

Proving termination is easy for steps 2, 4, and 6; steps 1, 3, and 5 are a bit more complicated.

The resulting formula is equivalent to the original one and in CNF.

The conversion of a formula to DNF works in the same way, except that conjunctions have to be pushed downward in step 5. □

## Negation Normal Form (NNF)

---

The formula after application of Step 4 is said to be in **Negation Normal Form**, i.e., it contains neither  $\rightarrow$  nor  $\leftrightarrow$  and negation symbols only occur in front of propositional variables (atoms).

# Complexity

---

Conversion to CNF (or DNF) may produce a formula whose size is **exponential** in the size of the original one.



## 2.5 Improving the CNF Transformation

---

The goal

“Given a formula  $F$ ,  
find an *equivalent* formula  $G$  in CNF”

is unpractical.

But if we relax the requirement to

“Given a formula  $F$ ,  
find an *equisatisfiable* formula  $G$  in CNF”

we can get an efficient transformation.

# Improving the CNF Transformation

---

Literature:

Andreas Nonnengart and Christoph Weidenbach: Computing small clause normal forms, in *Handbook of Automated Reasoning*, pages 335-367. Elsevier, 2001.

Christoph Weidenbach: Automated Reasoning (Chapter 2). Textbook draft, available for registered participants in the lecture Nextcloud (same link as for the online session recordings), 2021.

# Tseitin Transformation

---

Proposition 2.12:

A formula  $H[F]_p$  is satisfiable if and only if  $H[Q]_p \wedge (Q \leftrightarrow F)$  is satisfiable, where  $Q$  is a new propositional variable that works as an abbreviation for  $F$ .

# Tseitin Transformation

---

Satisfiability-preserving CNF transformation (Tseitin 1970):

Apply Prop. 2.12 recursively bottom-up to all subformulas  $F$  in the original formula (except  $\perp$ ,  $\top$ , and literals).

This introduces a linear number of new propositional variables  $Q$  and definitions  $Q \leftrightarrow F$ .

Convert the resulting conjunction to CNF.

This increases the size only by an additional factor, since each formula  $Q \leftrightarrow F$  yields at most four clauses in the CNF.

# Polarity-based CNF Transformation

---

A further improvement is possible by taking the **polarity** of the subformula  $F$  into account (Plaisted and Greenbaum 1986):

Intuitively, if  $G$  occurs in  $F$  at the position  $p$ , then the polarity of  $G$  determines the number of “negations” starting from  $F$  down to  $G$ .

It is 1 for an even number,  $-1$  for an odd number and 0 if there is at least one equivalence connective along the path.

# Polarity-based CNF Transformation

---

The **polarity** of a subformula  $G = F|_p$  at position  $p$  is  $\text{pol}(F, p)$ , where  $\text{pol}$  is recursively defined by

$$\text{pol}(F, \varepsilon) := 1$$

$$\text{pol}(\neg F, 1p) := -\text{pol}(F, p)$$

$$\text{pol}(F_1 \circ F_2, ip) := \text{pol}(F_i, p) \text{ if } \circ \in \{\wedge, \vee\}$$

$$\text{pol}(F_1 \rightarrow F_2, 1p) := -\text{pol}(F_1, p)$$

$$\text{pol}(F_1 \rightarrow F_2, 2p) := \text{pol}(F_2, p)$$

$$\text{pol}(F_1 \leftrightarrow F_2, ip) := 0$$

# Polarity-based CNF Transformation

---

Example 2.13:

Let  $F = (P \rightarrow Q) \rightarrow (P \wedge \neg R)$ .

Then  $\text{pol}(F, 1) = \text{pol}(F, 12) = \text{pol}(F, 221) = -1$

and  $\text{pol}(F, \varepsilon) = \text{pol}(F, 11) = \text{pol}(F, 2) = \text{pol}(F, 21) = \text{pol}(F, 22) = 1$ .

Let  $F' = (P \wedge Q) \leftrightarrow (P \vee Q)$ .

Then  $\text{pol}(F', \varepsilon) = 1$  and  $\text{pol}(F', p) = 0$  for all  $p \in \text{pos}(F')$  different from  $\varepsilon$ .

# Polarity-based CNF Transformation

---

Proposition 2.14:

Let  $\mathcal{A}$  be a valuation, let  $F$  and  $G$  be formulas, and let  $H = H[F]_p$  be a formula in which  $F$  occurs as a subformula at position  $p$ .

If  $\text{pol}(H, p) = 1$  and  $\mathcal{A}(F) \leq \mathcal{A}(G)$ , then  $\mathcal{A}(H[F]_p) \leq \mathcal{A}(H[G]_p)$ .

If  $\text{pol}(H, p) = -1$  and  $\mathcal{A}(F) \geq \mathcal{A}(G)$ , then  $\mathcal{A}(H[F]_p) \leq \mathcal{A}(H[G]_p)$ .



# Polarity-based CNF Transformation

---

Let  $Q$  be a propositional variable not occurring in  $H[F]_p$ .

Define the formula  $\text{def}(H, p, Q, F)$  by

- $(Q \rightarrow F)$ , if  $\text{pol}(H, p) = 1$ ,
- $(F \rightarrow Q)$ , if  $\text{pol}(H, p) = -1$ ,
- $(Q \leftrightarrow F)$ , if  $\text{pol}(H, p) = 0$ .

Proposition 2.15:

Let  $Q$  be a propositional variable not occurring in  $H[F]_p$ .

Then  $H[F]_p$  is satisfiable if and only if  $H[Q]_p \wedge \text{def}(H, p, Q, F)$  is satisfiable.

# Optimized CNF

---

Not every introduction of a definition for a subformula leads to a smaller CNF.

The number of potentially generated clauses is a good indicator for useful CNF transformations.

The functions  $\nu(F)$  and  $\bar{\nu}(F)$  on the next slide give us upper bounds for the number of clauses in  $\text{cnf}(F)$  and  $\text{cnf}(\neg F)$  using a naive CNF transformation.

# Optimized CNF

---

$G$	$\nu(G)$	$\bar{\nu}(G)$
$P, \top, \perp$	1	1
$F_1 \wedge F_2$	$\nu(F_1) + \nu(F_2)$	$\bar{\nu}(F_1)\bar{\nu}(F_2)$
$F_1 \vee F_2$	$\nu(F_1)\nu(F_2)$	$\bar{\nu}(F_1) + \bar{\nu}(F_2)$
$\neg F_1$	$\bar{\nu}(F_1)$	$\nu(F_1)$
$F_1 \rightarrow F_2$	$\bar{\nu}(F_1)\nu(F_2)$	$\nu(F_1) + \bar{\nu}(F_2)$
$F_1 \leftrightarrow F_2$	$\nu(F_1)\bar{\nu}(F_2) + \bar{\nu}(F_1)\nu(F_2)$	$\nu(F_1)\nu(F_2) + \bar{\nu}(F_1)\bar{\nu}(F_2)$

# Optimized CNF

---

A better CNF transformation (Nonnengart and Weidenbach 2001):

Step 1: Exhaustively apply modulo commutativity of  $\leftrightarrow$  and associativity/commutativity of  $\wedge$ ,  $\vee$ :

$$H[(F \wedge \top)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \vee \perp)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \leftrightarrow \perp)]_p \Rightarrow_{\text{OCNF}} H[\neg F]_p$$

$$H[(F \leftrightarrow \top)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \vee \top)]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

$$H[(F \wedge \perp)]_p \Rightarrow_{\text{OCNF}} H[\perp]_p$$

# Optimized CNF

---

$$H[(F \wedge F)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \vee F)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \wedge (F \vee G))]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \vee (F \wedge G))]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \wedge \neg F)]_p \Rightarrow_{\text{OCNF}} H[\perp]_p$$

$$H[(F \vee \neg F)]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

$$H[\neg \top]_p \Rightarrow_{\text{OCNF}} H[\perp]_p$$

$$H[\neg \perp]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

## Optimized CNF

---

$$H[(F \rightarrow \perp)]_p \Rightarrow_{\text{OCNF}} H[\neg F]_p$$

$$H[(F \rightarrow \top)]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

$$H[(\perp \rightarrow F)]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

$$H[(\top \rightarrow F)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

Note: Applying the absorption laws exhaustively modulo associativity/commutativity of  $\wedge$  and  $\vee$  is expensive. In practice, it is sufficient to apply them only in those cases that are easy to detect.

# Optimized CNF

---

Step 2: Introduce top-down fresh variables for beneficial subformulas:

$$H[F]_p \Rightarrow_{\text{OCNF}} H[Q]_p \wedge \text{def}(H, p, Q, F)$$

where  $Q$  is new to  $H[F]_p$

and  $\nu(H[F]_p) > \nu(H[Q]_p \wedge \text{def}(H, p, Q, F))$ .

Remark: Although computing  $\nu$  is not practical in general, the test  $\nu(H[F]_p) > \nu(H[Q]_p \wedge \text{def}(H, p, Q, F))$  can be computed in constant time.

# Optimized CNF

---

Step 3: Eliminate equivalences dependent on their polarity:

$$H[F \leftrightarrow G]_p \Rightarrow_{\text{OCNF}} H[(F \rightarrow G) \wedge (G \rightarrow F)]_p$$

if  $\text{pol}(H, p) = 1$  or  $\text{pol}(H, p) = 0$ .

$$H[F \leftrightarrow G]_p \Rightarrow_{\text{OCNF}} H[(F \wedge G) \vee (\neg F \wedge \neg G)]_p$$

if  $\text{pol}(H, p) = -1$ .



# Optimized CNF

---

Step 4: Apply steps 2, 3, 4, 5 of  $\Rightarrow_{\text{CNF}}$

Remark: The  $\Rightarrow_{\text{OCNF}}$  algorithm is already close to a state of the art algorithm, but some additional redundancy tests and simplification mechanisms are missing.

## 2.6 The DPLL Procedure

---

Goal:

Given a propositional formula in CNF (or alternatively, a finite set  $N$  of clauses), check whether it is satisfiable (and optionally: output *one* solution, if it is satisfiable).

# Preliminaries

---

Recall:

$\mathcal{A} \models N$  if and only if  $\mathcal{A} \models C$  for all clauses  $C$  in  $N$ .

$\mathcal{A} \models C$  if and only if  $\mathcal{A} \models L$  for some literal  $L \in C$ .

# Preliminaries

---

Assumptions:

Clauses contain neither duplicated literals nor complementary literals.

The order of literals in a clause is irrelevant.

⇒ Clauses behave like *sets* of literals.

Notation:

We use the notation  $C \vee L$  to denote a clause with some literal  $L$  and a clause rest  $C$ . Here  $L$  need *not* be the last literal of the clause and  $C$  may be empty.

$\bar{L}$  is the complementary literal of  $L$ , i. e.,  $\bar{P} = \neg P$  and  $\overline{\neg P} = P$ .

# Partial Valuations

---

Since we will construct satisfying valuations incrementally, we consider **partial valuations** (that is, partial mappings  $\mathcal{A} : \Pi \rightarrow \{0, 1\}$ ).

Every partial valuation  $\mathcal{A}$  corresponds to a set  $M$  of literals that does not contain complementary literals, and vice versa:

$\mathcal{A}(L)$  is true, if  $L \in M$ .

$\mathcal{A}(L)$  is false, if  $\bar{L} \in M$ .

$\mathcal{A}(L)$  is undefined, if neither  $L \in M$  nor  $\bar{L} \in M$ .

We will use  $\mathcal{A}$  and  $M$  interchangeably.

# Partial Valuations

---

A clause is true in a partial valuation  $\mathcal{A}$  (or in a set  $M$  of literals) if one of its literals is true;  
it is false (or “conflicting”) if all its literals are false;  
otherwise it is undefined (or “unresolved”).

# Unit Clauses

---

Observation:

Let  $\mathcal{A}$  be a partial valuation. If the set  $N$  contains a clause  $C$ , such that all literals but one in  $C$  are false in  $\mathcal{A}$ , then the following properties are equivalent:

- there is a valuation that is a model of  $N$  and extends  $\mathcal{A}$ .
- there is a valuation that is a model of  $N$  and extends  $\mathcal{A}$  and makes the remaining literal  $L$  of  $C$  true.

$C$  is called a **unit clause**;  $L$  is called a **unit literal**.

## Pure Literals

---

One more observation:

Let  $\mathcal{A}$  be a partial valuation and  $P$  a variable that is undefined in  $\mathcal{A}$ . If  $P$  occurs only positively (or only negatively) in the unresolved clauses in  $N$ , then the following properties are equivalent:

- there is a valuation that is a model of  $N$  and extends  $\mathcal{A}$ .
- there is a valuation that is a model of  $N$  and extends  $\mathcal{A}$  and assigns 1 (0) to  $P$ .

$P$  is called a **pure literal**.



# The Davis-Putnam-Logemann-Loveland Procedure

---

```
boolean DPLL(literal set  $M$ , clause set  $N$ ) {
  if (all clauses in  $N$  are true in  $M$ ) return true;
  elif (some clause in  $N$  is false in  $M$ ) return false;
  elif ( $N$  contains unit literal  $P$ ) return DPLL( $M \cup \{P\}$ ,  $N$ );
  elif ( $N$  contains unit literal  $\neg P$ ) return DPLL( $M \cup \{\neg P\}$ ,  $N$ );
  elif ( $N$  contains pure literal  $P$ ) return DPLL( $M \cup \{P\}$ ,  $N$ );
  elif ( $N$  contains pure literal  $\neg P$ ) return DPLL( $M \cup \{\neg P\}$ ,  $N$ );
  else {
    let  $P$  be some undefined variable in  $N$ ;
    if (DPLL( $M \cup \{\neg P\}$ ,  $N$ )) return true;
    else return DPLL( $M \cup \{P\}$ ,  $N$ );
  }
}
```

# The Davis-Putnam-Logemann-Loveland Procedure

Initially, DPLL is called with an empty literal set and the clause set  $N$ .

## 2.7 From DPLL to CDCL

---

The DPLL procedure can be improved significantly:

The pure literal check is only done while preprocessing (otherwise is too expensive).

If a conflict is detected, information is reused by conflict analysis and learning.

The algorithm is implemented iteratively  
⇒ the backtrack stack is managed explicitly  
(it may be possible and useful to backtrack more than one level).

The branching variable is not chosen randomly.

Under certain circumstances, the procedure is restarted.

## From DPLL to CDCL

---

Literature:

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: Solving SAT and SAT Modulo Theories – From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T), pp. 937–977, Journal of the ACM, 53(6), 2006.

# From DPLL to CDCL

---

Literature:

Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsh (eds.):  
Handbook of Satisfiability, IOS Press, 2009

Daniel Le Berre's slides at VTSA'09:

<http://www.mpi-inf.mpg.de/vtsa09/>.

# Conflict Analysis and Learning

---

Conflict analysis serves a dual purpose:

**Backjumping (non-chronological backtracking):**

If we detect that a conflict is independent of some earlier branch, we can skip over that backtrack level.

**Learning:**

By deriving a new clause from a conflict that is added to the current set of clauses, we can reuse information that is obtained in one branch in further branches.

(Note: This may produce a large number of new clauses; therefore it may become necessary to delete some of them afterwards to save space.)

# Conflict Analysis and Learning

---

These ideas are implemented in all modern SAT solvers.

Because of the importance of clause learning the algorithm is now called CDCL: Conflict Driven Clause Learning.

# Formalizing CDCL

---

We model the improved DPLL procedure by a transition relation  $\Rightarrow_{\text{CDCL}}$  on a set of states.

States:

- *fail*
- $M \parallel N$ ,

where  $M$  is a *list of annotated literals* (“trail”) and  $N$  is a set of clauses.



# Formalizing CDCL

---

Annotated literals:

- $L^C$ : deduced literal, due to unit propagation using clause  $C$ .
- $L^d$ : decision literal (guessed literal).

Alternative notation:  $L^{C_i} \equiv \frac{L}{C_i} \equiv \frac{L}{i}$

# Formalizing CDCL

---

Unit Propagate:

$$M \parallel N \cup \{C \vee L\} \Rightarrow_{\text{CDCL}} M L^{C \vee L} \parallel N \cup \{C \vee L\}$$

if  $C$  is false in  $M$  and  $L$  is undefined in  $M$ .

Decide:

$$M \parallel N \Rightarrow_{\text{CDCL}} M L^d \parallel N$$

if  $L$  is undefined in  $M$  and contained in  $N$ .

Fail:

$$M \parallel N \cup \{C\} \Rightarrow_{\text{CDCL}} \textit{fail}$$

if  $C$  is false in  $M$  and  $M$  contains no decision literals.

# Formalizing CDCL

---

Backjump:

$$M' \text{ } K^d \text{ } M'' \parallel N \Rightarrow_{\text{CDCL}} M' \text{ } L^{C \vee L} \parallel N$$

if some clause  $D \in N$  is false in  $M' \text{ } K^d \text{ } M''$

and if there is some “backjump clause”  $C \vee L$  such that

$$N \models C \vee L,$$

$C$  is false in  $M'$ , and

$L$  is undefined in  $M'$ .

## Formalizing CDCL

---

We will see later that the Backjump rule is always applicable, if the list of literals  $M$  contains at least one decision literal and some clause in  $N$  is false in  $M$ .

There are many possible backjump clauses.

One candidate is  $\overline{L_1} \vee \dots \vee \overline{L_n}$ , where the  $L_i$  are all the decision literals in  $M' \ L^d \ M''$ .

With this backjump clause, CDCL simulates DPLL.  
(But usually there are better choices.)

# Formalizing CDCL

---

Reasonable strategy:

- Use “Fail”, if applicable.
- Otherwise use “Backjump”, if applicable.  
Choose  $M'$  as short as possible. ( $\Rightarrow$  Go back to the earliest position where guessing can be replaced by knowledge.)
- Otherwise use “Unit Propagate”, if applicable.
- Otherwise use “Decide”.

## Formalizing CDCL

---

A trail  $M$  defines an ordering  $\succ_M$  on literals:  $L \succ_M K$  if  $L$  or  $\bar{L}$  occurs in  $M$  after (i. e., right of)  $K$  or  $\bar{K}$  (with any annotation  $d$  or  $C$ ).

# Formalizing CDCL

---

Lemma 2.16:

If  $\varepsilon \parallel N \Rightarrow_{\text{CDCL}}^* M \parallel N$ , then:

- (1)  $M$  contains neither duplicated nor complementary literals (regardless of their annotations).
- (2) If  $L^C$  is a deduced literal in  $M$ , then  $C$  has the form  $C' \vee L$ .
- (3) If  $L^{C' \vee L}$  is a deduced literal in  $M$ , then  $L \succ_M K$  for every literal  $K$  of  $C'$ .
- (4) If  $L^{C' \vee L}$  is a deduced literal in  $M$ , then  $C'$  is false in  $M$ .
- (5) Every literal  $L$  in  $M$  follows from  $N$  and decision literals in  $M$  that are smaller than or equal to  $L$  w. r. t.  $\succ_M$ .

# Formalizing CDCL

---

Lemma 2.17:

Every derivation starting from  $\varepsilon \parallel N$  terminates.



# Formalizing CDCL

---

Lemma 2.18:

Suppose that we reach a state  $M \parallel N$  starting from  $\varepsilon \parallel N$  such that some clause  $D \in N$  is false in  $M$ . Then:

- (1) If  $M$  does not contain any decision literal, then “Fail” is applicable.
- (2) Otherwise, “Backjump” is applicable.

# Formalizing CDCL

---

Theorem 2.19:

Suppose that we reach a final state starting from  $\varepsilon \parallel N$ .

- (1) If the final state is  $M \parallel N$ , then  $N$  is satisfiable and  $M$  is a model of  $N$ .
- (2) If the final state is *fail*, then  $N$  is unsatisfiable.

## Getting Better Backjump Clauses

---

Lemma 2.20:

Suppose that  $\varepsilon \parallel N \Rightarrow_{\text{CDCL}}^* M \parallel N$ .

Let  $D$  is a clause such that  $N \models D$  and  $D$  is false in  $M$ .

Let  $\bar{L}$  be the largest literal of  $D$  w. r. t.  $\succ_M$  and let  $D = D' \vee \bar{L}$ .

Suppose that  $L^{C \vee L}$  is a deduced literal in  $M$ .

Let  $D_0 = C \vee D'$ .

Then  $N \models D_0$ ;  $D_0$  is false in  $M$ ; and all literals of  $D_0$  are smaller than  $\bar{L}$ .

## Getting Better Backjump Clauses

---

The clause  $C \vee D'$  is called a *resolvent* of  $C \vee L$  and  $D' \vee \bar{L}$ ; this is denoted by

$$\frac{C \vee L \quad D' \vee \bar{L}}{C \vee D'}$$

Note that the resolvent  $C \vee D'$  is again entailed by  $N$  and false in  $M$ . If its largest literal is the complement of a deduced literal, we can therefore repeat the process with  $C \vee D'$ .

# Getting Better Backjump Clauses

---

Lemma 2.21:

Suppose that  $\varepsilon \parallel N \Rightarrow_{\text{CDCL}}^* M \parallel N$ .

Let  $D$  is a clause such that  $N \models D$  and  $D$  is false in  $M$ .

If  $D$  has the form  $D = D' \vee \bar{L}$ ,

where  $L$  is larger than or equal to the largest decision literal of  $M$  and all literals in  $D'$  are smaller than the largest decision literal of  $M$ ,

then  $M$  can be written as  $M' K^d M''$

such that  $D'$  is false in  $M'$  and  $L$  is undefined in  $M'$

(that is,  $D$  is a backjump clause).

## Getting Better Backjump Clauses

---

Suppose that  $\varepsilon \parallel N \Rightarrow_{\text{CDCL}}^* M \parallel N$ ,

where  $L^d$  is the largest decision literal in  $M$  and some  $D \in N$  is false in  $M$ .

If we have used a reasonable strategy, then  $D$  must contain two literals whose complements are larger than or equal to  $L$ .

By repeated resolution steps as described in Lemma 2.20, we must eventually reach a clause that satisfies Lemma 2.21.

This clause is a backjump clause

$\Rightarrow$  1UIP (first unique implication point) strategy.

# Learning Clauses

---

Backjump clauses are good candidates for learning.

To model learning, the CDCL system is extended by the following two rules:

Learn:

$$M \parallel N \Rightarrow_{\text{CDCL}} M \parallel N \cup \{C\}$$

if  $N \models C$ .

Forget:

$$M \parallel N \cup \{C\} \Rightarrow_{\text{CDCL}} M \parallel N$$

if  $N \models C$ .

# Learning Clauses

---

If we ensure that no clause is learned infinitely often, then termination is guaranteed.

The other properties of the basic CDCL system hold also for the extended system.



# Restart

---

Runtimes of CDCL-style procedures depend extremely on the choice of branching variables.

If no solution is found within a certain time limit, it can be useful to **restart** from scratch with an adapted variable selection heuristics.

Learned clauses, however, are kept.

In addition, it is useful to restart after a unit clause has been learned.

# Restart

---

The restart rule is typically applied after a certain number of clauses have been learned or a unit is derived:

Restart:

$$M \parallel N \Rightarrow_{\text{CDCL}} \varepsilon \parallel N$$

If Restart is only applied finitely often, termination is guaranteed.

## 2.8 Implementing CDCL

---

The formalization of CDCL that we have seen so far leaves many aspects unspecified.

To get a fast solver, we must use good heuristics, for instance to choose the next undefined variable, and we must implement basic operations efficiently.

## Variable Order Heuristic

---

Choosing the right undefined variable to branch is important for efficiency, but the branching heuristics may be expensive itself.

State of the art: Use branching heuristics that need not be recomputed too frequently.

In general: Choose variables that occur frequently; after a restart prefer variables from recent conflicts.

# Variable Order Heuristic

---

The VSIDS (Variable State Independent Decaying Sum) heuristic:

- We associate a positive **score** to every propositional variable  $P_i$ . At the start,  $k_i$  is the number of occurrences of  $P_i$  in  $N$ .
- The variable order is then the descending ordering of the  $P_i$  according to the  $k_i$ .

# Variable Order Heuristic

---

The scores  $k_i$  are adjusted during a CDCL run.

- Every time a learned clause is computed after a conflict, the propositional variables in the learned clause obtain a bonus  $b$ , i.e.,  $k_i := k_i + b$ .
- Periodically, the scores are leveled:  $k_i := k_i / l$  for some  $l$ .
- After each restart, the variable order is recomputed, using the new scores.

The purpose of these mechanisms is to keep the search focused.

The parameter  $b$  directs the search around the conflict,

# Variable Order Heuristic

---

Further refinements:

- Add the bonus to all literals in the clauses that occur in the resolution steps to generate a backjump clause.
- If the score of a variable reaches a certain limit, all scores are rescaled by a constant.
- Occasionally (with low probability) choose a variable at random, otherwise choose the undefined variable with the highest score.

## Implementing Unit Propagation Efficiently

---

For applying the unit rule, we need to know the number of literals in a clause that are not false.

Maintaining this number is expensive, however.



# Implementing Unit Propagation Efficiently

---

Better approach: “Two watched literals”:

In each clause, select two (currently undefined) “watched” literals.

For each variable  $P$ , keep a list of all clauses in which  $P$  is watched and a list of all clauses in which  $\neg P$  is watched.

If an undefined variable is set to 0 (or to 1), check all clauses in which  $P$  (or  $\neg P$ ) is watched and watch another literal (that is true or undefined) in this clause if possible.

Watched literal information need not be restored upon backtracking.

## 2.9 Preprocessing and Inprocessing

---

Some operations are only needed once at the beginning of the CDCL run.

- (i) Deletion of tautologies
- (ii) Deletion of duplicated literals

# Preprocessing and Inprocessing

---

Some operations are useful, but expensive.

They are performed only initially and after restarts (before computation of the variable order heuristics), possibly with time limits.

Note: Some of these operations are only satisfiability-preserving; they do not yield equivalent clause sets.

# Preprocessing and Inprocessing

---

Literature:

Matti Järvisalo, Marijn J. H. Heule, and Armin Biere: Inprocessing Rules, Proc. IJCAR 2012, LNAI 7364, pp. 355–370, Springer, 2012

# Preprocessing and Inprocessing

---

Examples:

(i) Subsumption

$$N \cup \{C\} \cup \{D\} \Rightarrow N \cup \{C\}$$

if  $C \subseteq D$  considering  $C, D$  as multisets of literals.

# Preprocessing and Inprocessing

---

## (ii) Purity deletion

Delete all clauses containing a literal  $L$  where  $\bar{L}$  does not occur in the clause set.

## (iii) Merging replacement resolution

$$N \cup \{C \vee L\} \cup \{D \vee \bar{L}\} \Rightarrow N \cup \{C \vee L\} \cup \{D\}$$

if  $C \subseteq D$  considering  $C, D$  as multisets of literals.

# Preprocessing and Inprocessing

---

## (iv) Bounded variable elimination

Compute all possible resolution steps

$$\frac{C \vee L \quad D \vee \bar{L}}{C \vee D}$$

on a literal  $L$  with premises in  $N$ ;

add all non-tautological conclusions to  $N$ ;

then throw away all clauses containing  $L$  or  $\bar{L}$ ;

repeat this as long as  $|N|$  does not grow.

# Preprocessing and Inprocessing

---

(v) RAT (“Resolution asymmetric tautologies”)

$C$  is called an **asymmetric tautology** w. r. t.  $N$ , if its negation can be refuted by unit propagation using clauses in  $N$ .

$C$  has the **RAT property** w. r. t.  $N$ , if it is an asymmetric tautology w. r. t.  $N$ , or if there is a literal  $L$  in  $C$  such that  $C = C' \vee L$  and all clauses  $D' \vee C'$  for  $D' \vee \bar{L} \in N$  are asymmetric tautologies w. r. t.  $N$ .

RAT elimination:

$$N \cup \{C\} \Rightarrow N$$

if  $C$  has the RAT property w. r. t.  $N$ .



## 2.10 OBDDs

---

Goal:

Efficient manipulation of (equivalence classes of) propositional formulas.

Method: Minimized graph representation of decision trees,  
based on a fixed ordering on propositional variables.

⇒ Canonical representation of formulas.

⇒ Satisfiability checking as a side effect.

# OBDDs

---

Literature:

Randal E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers, 35(8):677-691, 1986.

Randal E. Bryant: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, ACM Computing Surveys, 24(3), September 1992, pp. 293-318.

Michael Huth and Mark Ryan: *Logic in Computer Science: Modelling and Reasoning about Systems*, Chapter 6.1/6.2; Cambridge Univ. Press, 2000.

# BDDs

---

BDD (Binary decision diagram):

Labelled DAG (directed acyclic graph).

Leaf nodes:

labelled with a truth value (0 or 1).

Non-leaf nodes (interior nodes):

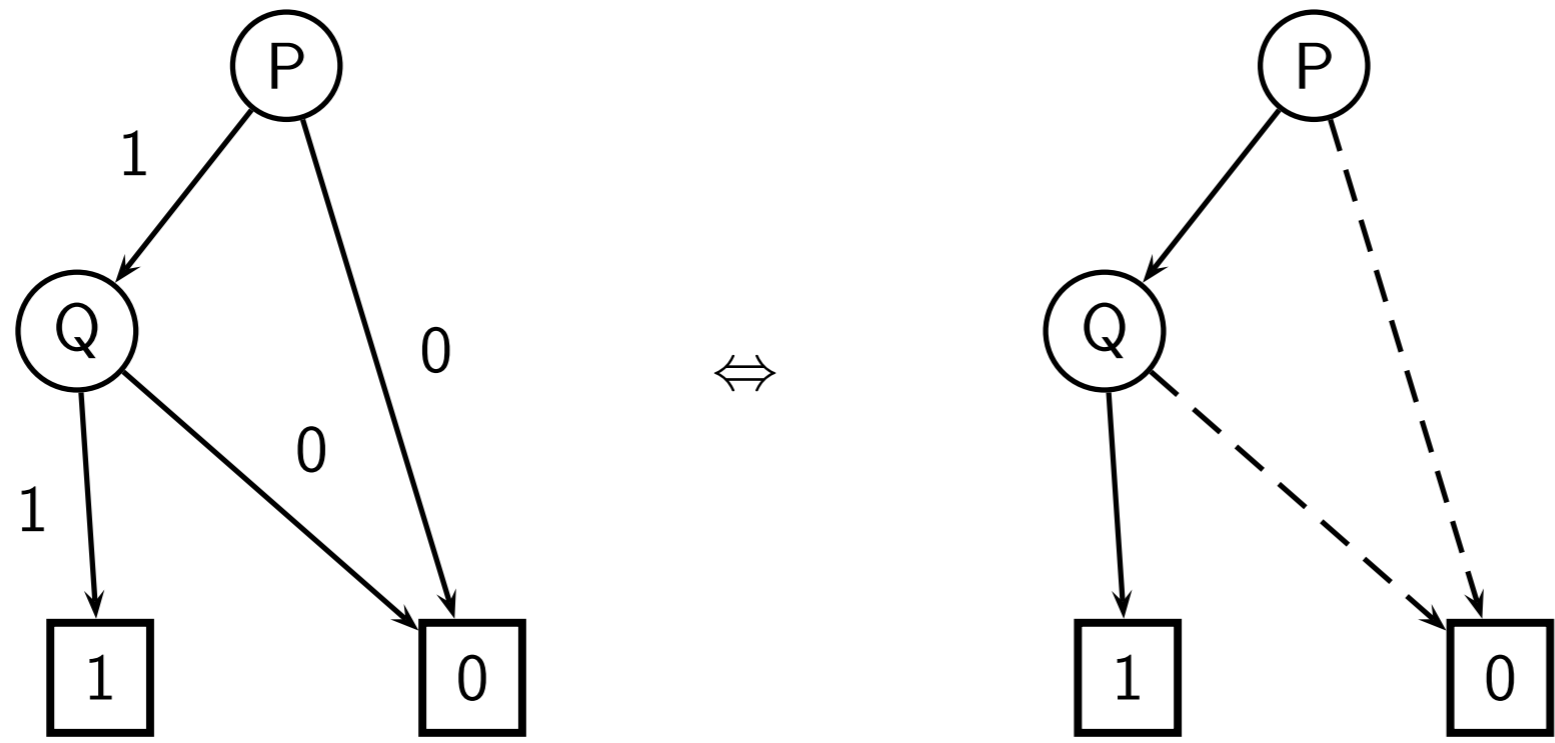
labelled with a propositional variable,

exactly two outgoing edges,

labelled with 0 (  $\dashrightarrow$  ) and 1 (  $\longrightarrow$  )

# BDDs

---



# BDDs

---

Every BDD node can be interpreted as a mapping from valuations to truth values:

Traverse the BDD from the given node to a leaf node; for any node labelled with  $P$  take the 0-edge or 1-edge depending on whether  $\mathcal{A}(P)$  is 0 or 1.

⇒ Compact representation of truth tables.

# OBDDs

---

OBDD (Ordered BDD):

Let  $<$  be a total ordering of the propositional variables.

An OBDD w. r. t.  $<$  is a BDD where every edge from a non-leaf node leads either to a leaf node or to a non-leaf node with a strictly larger label w. r. t.  $<$ .

# OBDDs

---

OBDDs and formulas:

A leaf node  $\boxed{0}$  represents  $\perp$  (or any unsatisfiable formula).

A leaf node  $\boxed{1}$  represents  $\top$  (or any valid formula).

If a non-leaf node  $v$  has the label  $P$ ,  
and its 0-edge leads to a node representing the formula  $F_0$ ,  
and its 1-edge leads to a node representing the formula  $F_1$ ,  
then  $v$  represents the formula

$$\begin{aligned} F &\models \text{if } P \text{ then } F_1 \text{ else } F_0 \\ &\models (P \wedge F_1) \vee (\neg P \wedge F_0) \\ &\models (P \rightarrow F_1) \wedge (\neg P \rightarrow F_0) \end{aligned}$$

# OBDDs

---

Conversely:

Define  $F\{P \mapsto H\}$  as the formula obtained from  $F$  by replacing every occurrence of  $P$  in  $F$  by  $H$ .

For every formula  $F$  and propositional variable  $P$ :

$$F \models (P \wedge F\{P \mapsto \top\}) \vee (\neg P \wedge F\{P \mapsto \perp\})$$

(*Shannon expansion* of  $F$ , originally due to Boole).

Consequence: Every formula  $F$  can be represented by an OBDD.

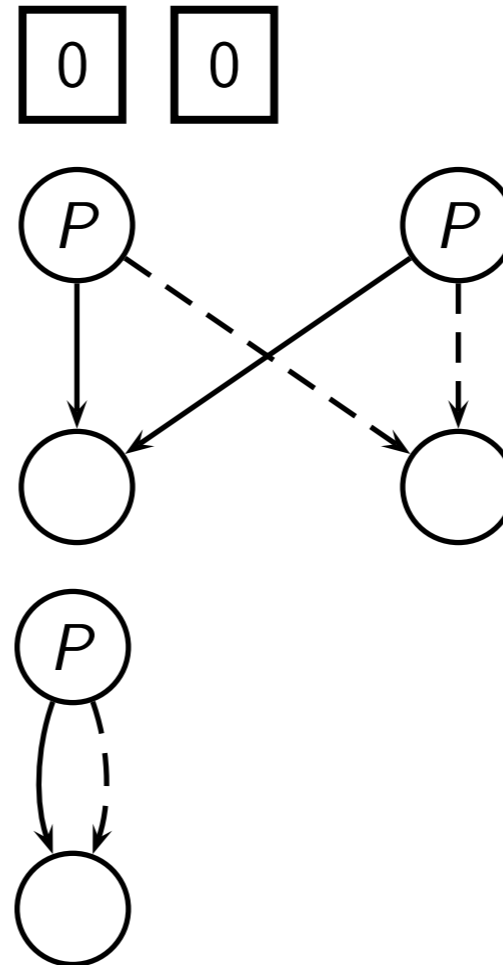


# Reduced OBDDs

---

An OBDD is called *reduced*, if it has

- no duplicated leaf nodes
- no duplicated interior nodes
- no redundant tests



# Reduced OBDDs

---

Theorem 2.22 (Bryant 1986):

Every OBDD can be converted into an equivalent reduced OBDD.

Assumptions from now on:

One fixed ordering  $>$ .

We consider only reduced OBDDs.

All OBDDs are sub-OBDDs of a single OBDD.

# Reduced OBDDs

---

Implementation:

Bottom-up construction of reduced OBDDs is possible using a hash table.

Keys and values are triples  $(PropVar, Ptr_0, Ptr_1)$ ,

where  $Ptr_0$  and  $Ptr_1$  are pointers to the 0-successor and 1-successor hash table entry.

## Reduced OBDDs

---

Theorem 2.23 (Bryant 1986):

If  $v$  and  $v'$  are two different nodes in a reduced OBDD, then they represent non-equivalent formulas.

## Reduced OBDDs

---

Corollary 2.24:

$F$  is valid, if and only if it is represented by  $\boxed{1}$ .

$F$  is unsatisfiable, if and only if it is represented by  $\boxed{0}$ .

# Operations on OBDDs

---

Example:

Let  $\circ$  be a binary connective. Let  $P$  be the smallest propositional variable that occurs in  $F$  or  $G$  or both.

$$F \circ G \models (P \wedge (F \circ G)\{P \mapsto \top\}) \vee (\neg P \wedge (F \circ G)\{P \mapsto \perp\})$$

$$\begin{aligned} &\models (P \wedge (F\{P \mapsto \top\} \circ G\{P \mapsto \top\}) \\ &\quad \vee (\neg P \wedge (F\{P \mapsto \perp\} \circ G\{P \mapsto \perp\}))) \end{aligned}$$

Note:  $F\{P \mapsto \top\}$  is either represented by the same node as  $F$  (if  $P$  does not occur in  $F$ ), or by its 1-successor (otherwise).

$\Rightarrow$  Obvious recursive function on OBDD nodes  
(needs memoizing for efficient implementation).

# Operations on OBDDs

---

OBDD operations are not restricted to the connectives of propositional logic.

We can also compute operations of *quantified boolean formulas*

$$\forall P. F \models (F\{P \mapsto \top\}) \wedge (F\{P \mapsto \perp\})$$

$$\exists P. F \models (F\{P \mapsto \top\}) \vee (F\{P \mapsto \perp\})$$

and images or preimages of propositional formulas w. r. t. boolean relations (needed for typical verification tasks).

## Operations on OBDDs

---

The size of the OBDD for  $F \circ G$  is bounded by  $mn$ , where  $F$  has size  $m$  and  $G$  has size  $n$ . (Size = number of nodes)

With memoization, the time for computing  $F \circ G$  is also at most  $O(mn)$ .



# Operations on OBDDs

---

The size of an OBDD for a given formula depends crucially on the chosen ordering of the propositional variables:

$$\text{Let } F = (P_1 \wedge P_2) \vee (P_3 \wedge P_4) \vee \cdots \vee (P_{2n-1} \wedge P_{2n}).$$

$$P_1 < P_2 < P_3 < P_4 < \cdots < P_{2n-1} < P_{2n}: 2n + 2 \text{ nodes.}$$

$$P_1 < P_3 < \cdots < P_{2n-1} < P_2 < P_4 < \cdots < P_{2n}: 2^{n+1} \text{ nodes.}$$

## Operations on OBDDs

---

Even worse: There are (practically relevant!) formulas for which the OBDD has exponential size *for every ordering* of the propositional variables.

Example: middle bit of binary multiplication.

## 2.11 FRAIGs

---

Goal:

Efficient manipulation of (equivalence classes of) propositional formulas.

Smaller representation than OBDDs.

Method: Minimized graph representation of boolean circuits.

# FRAIGs

---

FRAIG (Functionally Reduced And-Inverter Graph):

Labelled DAG (directed acyclic graph).

Leaf nodes:

labelled with propositional variables.

Non-leaf nodes (interior nodes):

labelled with  $\wedge$  (two outgoing edges) or  $\neg$  (one outgoing edge).

# FRAIGs

---

Reducedness (i. e., no two different nodes represent equivalent formulas) must be established explicitly, using

structural hashing,

simulation vectors,

CDCL,

OBDDs.

⇒ Semi-canonical representation of formulas.

# FRAIGs

---

Literature:

A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton: FRAIGs: A unifying representation for logic synthesis and verification, ERL Technical Report, EECS Dept., UC Berkeley, March 2005.

## 2.12 Other Calculi

---

Ordered resolution

Tableau calculus

Hilbert calculus

Sequent calculus

Natural deduction

see next chapter

## Part 3: First-Order Logic

---

### First-order logic

- is expressive:
  - can be used to formalize mathematical concepts,
  - can be used to encode Turing machines,
  - but cannot axiomatize natural numbers or uncountable sets,
- has important decidable fragments,
- has interesting logical properties (model and proof theory).

First-order logic is also called (first-order) **predicate logic**.



## 3.1 Syntax

---

Syntax:

- non-logical symbols (domain-specific)  
⇒ terms, atomic formulas
- logical connectives (domain-independent)  
⇒ Boolean combinations, quantifiers

# Signatures

---

A signature  $\Sigma = (\Omega, \Pi)$  fixes an alphabet of non-logical symbols, where

- $\Omega$  is a set of **function symbols**  $f$  with **arity**  $n \geq 0$ ,  
written  $\text{arity}(f) = n$ ,
- $\Pi$  is a set of **predicate symbols**  $P$  with **arity**  $m \geq 0$ ,  
written  $\text{arity}(P) = m$ .

Function symbols are also called **operator symbols**.

If  $n = 0$  then  $f$  is also called a **constant (symbol)**.

If  $m = 0$  then  $P$  is also called a **propositional variable**.

# Signatures

---

We will usually use

$b, c, d$  for constant symbols,

$f, g, h$  for non-constant function symbols,

$P, Q, R, S$  for predicate symbols.

Convention: We will usually write  $f/n \in \Omega$  instead of  $f \in \Omega$ ,  $\text{arity}(f) = n$  (analogously for predicate symbols).

# Signatures

---

Refined concept for practical applications:

*many-sorted* signatures (corresponds to simple type systems in programming languages);

no big change from a logical point of view.

# Variables

---

Predicate logic admits the formulation of abstract, schematic assertions.  
(Object) variables are the technical tool for schematization.

We assume that  $X$  is a given countably infinite set of symbols which we use to denote **variables**.

# Terms

---

**Terms** over  $\Sigma$  and  $X$  ( $\Sigma$ -terms) are formed according to these syntactic rules:

$$\begin{aligned} s, t, u, v & ::= x, x \in X && \text{(variable)} \\ & | f(s_1, \dots, s_n), f/n \in \Omega && \text{(functional term)} \end{aligned}$$

By  $T_\Sigma(X)$  we denote the set of  $\Sigma$ -terms (over  $X$ ).

A term not containing any variable is called a **ground term**.

By  $T_\Sigma$  we denote the set of  $\Sigma$ -ground terms.

# Atoms

---

**Atoms** (also called atomic formulas) over  $\Sigma$  are formed according to this syntax:

$$A, B ::= P(s_1, \dots, s_m) \text{ , } P/m \in \Pi \quad (\text{non-equational atom}) \\ \left[ \mid (s \approx t) \quad \quad \quad (\text{equation}) \right]$$

Whenever we admit equations as atomic formulas we are in the realm of **first-order logic with equality**. Admitting equality does not really increase the expressiveness of first-order logic (see next chapter). But deductive systems where equality is treated specifically are much more efficient.

# Literals

---

$L ::= A$  (positive literal)  
|  $\neg A$  (negative literal)



# Clauses

---

$C, D ::= \perp$  (empty clause)  
|  $L_1 \vee \dots \vee L_k, k \geq 1$  (non-empty clause)

# General First-Order Formulas

---

$F_{\Sigma}(X)$  is the set of **first-order formulas** over  $\Sigma$  defined as follows:

$F, G, H$	$::=$	$\perp$	(falsum)
		$\top$	(verum)
		$A$	(atomic formula)
		$\neg F$	(negation)
		$(F \wedge G)$	(conjunction)
		$(F \vee G)$	(disjunction)
		$(F \rightarrow G)$	(implication)
		$(F \leftrightarrow G)$	(equivalence)
		$\forall x F$	(universal quantification)
		$\exists x F$	(existential quantification)

# Notational Conventions

---

We omit parentheses according to the conventions for propositional logic.

$\forall x_1, \dots, x_n F$  and  $\exists x_1, \dots, x_n F$  abbreviate

$\forall x_1 \dots \forall x_n F$  and  $\exists x_1 \dots \exists x_n F$ .

# Notational Conventions

---

We use infix-, prefix-, postfix-, or mixfix-notation with the usual operator precedences.

Examples:

$$s + t * u \quad \text{for} \quad +(s, *(t, u))$$

$$s * u \leq t + v \quad \text{for} \quad \leq (*(s, u), +(t, v))$$

$$-s \quad \text{for} \quad -(s)$$

$$s! \quad \text{for} \quad !(s)$$

$$|s| \quad \text{for} \quad |-(s)|$$

$$0 \quad \text{for} \quad 0()$$

## Example: Peano Arithmetic

---

$$\Sigma_{\text{PA}} = (\Omega_{\text{PA}}, \Pi_{\text{PA}})$$

$$\Omega_{\text{PA}} = \{0/0, +/2, */2, s/1\}$$

$$\Pi_{\text{PA}} = \{</2\}$$

Examples of formulas over this signature are:

$$\forall x, y ((x < y \vee x \approx y) \leftrightarrow \exists z (x + z \approx y))$$

$$\exists x \forall y (x + y \approx y)$$

$$\forall x, y (x * s(y) \approx x * y + x)$$

$$\forall x, y (s(x) \approx s(y) \rightarrow x \approx y)$$

$$\forall x \exists y (x < y \wedge \neg \exists z (x < z \wedge z < y))$$

# Positions in Terms and Formulas

---

The set of positions is extended from propositional logic to first-order logic:

The **positions** of a term  $s$  (formula  $F$ ):

$$\text{pos}(x) = \{\varepsilon\},$$

$$\text{pos}(f(s_1, \dots, s_n)) = \{\varepsilon\} \cup \bigcup_{i=1}^n \{i p \mid p \in \text{pos}(s_i)\},$$

$$\text{pos}(P(t_1, \dots, t_n)) = \{\varepsilon\} \cup \bigcup_{i=1}^n \{i p \mid p \in \text{pos}(t_i)\},$$

$$\text{pos}(\forall x F) = \{\varepsilon\} \cup \{1 p \mid p \in \text{pos}(F)\},$$

$$\text{pos}(\exists x F) = \{\varepsilon\} \cup \{1 p \mid p \in \text{pos}(F)\}.$$

## Positions in Terms and Formulas

---

The prefix order  $\leq$ , the subformula (subterm) operator, the formula (term) replacement operator and the size operator are extended accordingly.

See the definitions in Sect. 2.

# Variables

---

The **set of variables** occurring in a term  $t$  is denoted by  $\text{var}(t)$  (and analogously for atoms, literals, clauses, and formulas).



# Bound and Free Variables

---

In  $Qx F$ ,  $Q \in \{\exists, \forall\}$ , we call  $F$  the **scope** of the quantifier  $Qx$ .

An *occurrence* of a variable  $x$  is called **bound**, if it is inside the scope of a quantifier  $Qx$ .

Any other occurrence of a variable is called **free**.

Formulas without free variables are called **closed formulas** (or **sentential forms**).

Formulas without variables are called **ground**.

# Bound and Free Variables

---

Example:

$$\forall y \left( \left( \forall x \left( P(x) \right) \right) \rightarrow R(x, y) \right)$$

The diagram shows the scope of the quantifiers. A bracket above the entire expression is labeled "scope of  $\forall y$ ". A smaller bracket above the inner expression  $(\forall x (P(x)))$  is labeled "scope of  $\forall x$ ".

The occurrence of  $y$  is bound, as is the first occurrence of  $x$ . The second occurrence of  $x$  is a free occurrence.

# Substitutions

---

Substitution is a fundamental operation on terms and formulas that occurs in all inference systems for first-order logic.

Substitutions are mappings

$$\sigma : X \rightarrow T_{\Sigma}(X)$$

such that the **domain** of  $\sigma$ , that is, the set

$$\text{dom}(\sigma) = \{x \in X \mid \sigma(x) \neq x\},$$

is finite. The set of variables **introduced** by  $\sigma$ , that is, the set of variables occurring in one of the terms  $\sigma(x)$ , with  $x \in \text{dom}(\sigma)$ , is denoted by **codom**( $\sigma$ ).

# Substitutions

---

Substitutions are often written as  $\{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\}$ , with  $x_i$  pairwise distinct, and then denote the mapping

$$\{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\}(y) = \begin{cases} s_i, & \text{if } y = x_i \\ y, & \text{otherwise} \end{cases}$$

We also write  $x\sigma$  for  $\sigma(x)$ .

The **modification** of a substitution  $\sigma$  at  $x$  is defined as follows:

$$\sigma[x \mapsto t](y) = \begin{cases} t, & \text{if } y = x \\ \sigma(y), & \text{otherwise} \end{cases}$$

# Why Substitution is Complicated

---

We define the application of a substitution  $\sigma$  to a term  $t$  or formula  $F$  by structural induction over the syntactic structure of  $t$  or  $F$  by the equations on the next slide.

In the presence of quantification it is surprisingly complex:

We must not only ensure that bound variables are not replaced by  $\sigma$ .

We must also make sure that the (free) variables in the codomain of  $\sigma$  are not *captured* upon placing them into the scope of a quantifier  $Qy$ .

Hence the bound variable must be renamed into a “fresh”, that is, previously unused, variable  $z$ .

# Application of a Substitution

---

“Homomorphic” extension of  $\sigma$  to terms and formulas:

$$f(s_1, \dots, s_n)\sigma = f(s_1\sigma, \dots, s_n\sigma)$$

$$\perp\sigma = \perp$$

$$\top\sigma = \top$$

$$P(s_1, \dots, s_n)\sigma = P(s_1\sigma, \dots, s_n\sigma)$$

$$(u \approx v)\sigma = (u\sigma \approx v\sigma)$$

$$\neg F\sigma = \neg(F\sigma)$$

$$(F \circ G)\sigma = (F\sigma \circ G\sigma) \quad \text{for each binary connective } \circ$$

$$(Qx F)\sigma = Qz (F\sigma[x \mapsto z]) \quad \text{with } z \text{ a fresh variable}$$

# Application of a Substitution

---

If  $s = t\sigma$  for some substitution  $\sigma$ ,  
we call the term  $s$  an **instance** of the term  $t$ ,  
and we call  $t$  a **generalization** of  $s$  (analogously for formulas).

## 3.2 Semantics

---

To give semantics to a logical system means to define a notion of truth for the formulas. The concept of truth that we will now define for first-order logic goes back to Tarski.

As in the propositional case, we use a two-valued logic with truth values “true” and “false” denoted by 1 and 0, respectively.



# Algebras

---

A  $\Sigma$ -algebra (also called  $\Sigma$ -interpretation or  $\Sigma$ -structure) is a triple

$$\mathcal{A} = (U_{\mathcal{A}}, (f_{\mathcal{A}} : U_{\mathcal{A}}^n \rightarrow U_{\mathcal{A}})_{f/n \in \Omega}, (P_{\mathcal{A}} \subseteq U_{\mathcal{A}}^m)_{P/m \in \Pi})$$

where  $U_{\mathcal{A}} \neq \emptyset$  is a set, called the **universe** of  $\mathcal{A}$ .

By  $\Sigma\text{-Alg}$  we denote the class of all  $\Sigma$ -algebras.

$\Sigma$ -algebras generalize the valuations from propositional logic.

# Assignments

---

A variable has no intrinsic meaning. The meaning of a variable has to be defined externally (explicitly or implicitly in a given context) by an assignment.

A (variable) assignment (over a given  $\Sigma$ -algebra  $\mathcal{A}$ ), is a function  $\beta : X \rightarrow U_{\mathcal{A}}$ .

Variable assignments are the semantic counterparts of substitutions.

## Value of a Term in $\mathcal{A}$ with respect to $\beta$

---

By structural induction we define

$$\mathcal{A}(\beta) : T_{\Sigma}(X) \rightarrow U_{\mathcal{A}}$$

as follows:

$$\begin{aligned} \mathcal{A}(\beta)(x) &= \beta(x), & x \in X \\ \mathcal{A}(\beta)(f(s_1, \dots, s_n)) &= f_{\mathcal{A}}(\mathcal{A}(\beta)(s_1), \dots, \mathcal{A}(\beta)(s_n)), & f/n \in \Omega \end{aligned}$$

## Value of a Term in $\mathcal{A}$ with respect to $\beta$

---

In the scope of a quantifier we need to evaluate terms with respect to modified assignments. To that end, let  $\beta[x \mapsto a] : X \rightarrow U_{\mathcal{A}}$ , for  $x \in X$  and  $a \in U_{\mathcal{A}}$ , denote the assignment

$$\beta[x \mapsto a](y) = \begin{cases} a & \text{if } x = y \\ \beta(y) & \text{otherwise} \end{cases}$$

## Truth Value of a Formula in $\mathcal{A}$ with respect to $\beta$

---

$\mathcal{A}(\beta) : F_{\Sigma}(X) \rightarrow \{0, 1\}$  is defined inductively as follows:

$$\mathcal{A}(\beta)(\perp) = 0$$

$$\mathcal{A}(\beta)(\top) = 1$$

$$\mathcal{A}(\beta)(P(s_1, \dots, s_n)) = \begin{array}{l} \text{if } (\mathcal{A}(\beta)(s_1), \dots, \mathcal{A}(\beta)(s_n)) \in P_{\mathcal{A}} \\ \text{then 1 else 0} \end{array}$$

$$\mathcal{A}(\beta)(s \approx t) = \begin{array}{l} \text{if } \mathcal{A}(\beta)(s) = \mathcal{A}(\beta)(t) \text{ then 1 else 0} \end{array}$$

## Truth Value of a Formula in $\mathcal{A}$ with respect to $\beta$

---

$\mathcal{A}(\beta) : F_{\Sigma}(X) \rightarrow \{0, 1\}$  is defined inductively as follows:

$$\mathcal{A}(\beta)(\neg F) = 1 - \mathcal{A}(\beta)(F)$$

$$\mathcal{A}(\beta)(F \wedge G) = \min(\mathcal{A}(\beta)(F), \mathcal{A}(\beta)(G))$$

$$\mathcal{A}(\beta)(F \vee G) = \max(\mathcal{A}(\beta)(F), \mathcal{A}(\beta)(G))$$

$$\mathcal{A}(\beta)(F \rightarrow G) = \max(1 - \mathcal{A}(\beta)(F), \mathcal{A}(\beta)(G))$$

$$\mathcal{A}(\beta)(F \leftrightarrow G) = \text{if } \mathcal{A}(\beta)(F) = \mathcal{A}(\beta)(G) \text{ then } 1 \text{ else } 0$$

$$\mathcal{A}(\beta)(\forall x F) = \min_{a \in U_{\mathcal{A}}} \{ \mathcal{A}(\beta[x \mapsto a])(F) \}$$

$$\mathcal{A}(\beta)(\exists x F) = \max_{a \in U_{\mathcal{A}}} \{ \mathcal{A}(\beta[x \mapsto a])(F) \}$$

## Example

---

The “Standard” interpretation for Peano arithmetic:

$$U_{\mathbb{N}} = \{0, 1, 2, \dots\}$$

$$0_{\mathbb{N}} = 0$$

$$s_{\mathbb{N}} : n \mapsto n + 1$$

$$+_{\mathbb{N}} : (n, m) \mapsto n + m$$

$$*_{\mathbb{N}} : (n, m) \mapsto n * m$$

$$<_{\mathbb{N}} = \{ (n, m) \mid n \text{ less than } m \}$$

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

## Example

---

Values over  $\mathbb{N}$  for sample terms and formulas:

Under the assignment  $\beta : x \mapsto 1, y \mapsto 3$  we obtain

$$\mathbb{N}(\beta)(s(x) + s(0)) = 3$$

$$\mathbb{N}(\beta)(x + y \approx s(y)) = 1$$

$$\mathbb{N}(\beta)(\forall x, y (x + y \approx y + x)) = 1$$

$$\mathbb{N}(\beta)(\forall z (z < y)) = 0$$

$$\mathbb{N}(\beta)(\forall x \exists y (x < y)) = 1$$



## Ground Terms and Closed Formulas

---

If  $t$  is a ground term, then  $\mathcal{A}(\beta)(t)$  does not depend on  $\beta$ , that is,  $\mathcal{A}(\beta)(t) = \mathcal{A}(\beta')(t)$  for every  $\beta$  and  $\beta'$ .

Analogously, if  $F$  is a closed formula, then  $\mathcal{A}(\beta)(F)$  does not depend on  $\beta$ , that is,  $\mathcal{A}(\beta)(F) = \mathcal{A}(\beta')(F)$  for every  $\beta$  and  $\beta'$ .

## Ground Terms and Closed Formulas

---

An element  $a \in U_{\mathcal{A}}$  is called **term-generated**, if  $a = \mathcal{A}(\beta)(t)$  for some ground term  $t$ .

In general, not every element of an algebra is term-generated.

### 3.3 Models, Validity, and Satisfiability

---

$F$  is **true** in  $\mathcal{A}$  under assignment  $\beta$ :

$$\mathcal{A}, \beta \models F \quad :\Leftrightarrow \quad \mathcal{A}(\beta)(F) = 1$$

$F$  is **true** in  $\mathcal{A}$  ( $\mathcal{A}$  is a **model** of  $F$ ;  $F$  is **valid** in  $\mathcal{A}$ ):

$$\mathcal{A} \models F \quad :\Leftrightarrow \quad \mathcal{A}, \beta \models F \quad \text{for all } \beta \in X \rightarrow U_{\mathcal{A}}$$

$F$  is **valid** (or is a **tautology**):

$$\models F \quad :\Leftrightarrow \quad \mathcal{A} \models F \quad \text{for all } \mathcal{A} \in \Sigma\text{-Alg}$$

$F$  is called **satisfiable** if there exist  $\mathcal{A}$  and  $\beta$  such that  $\mathcal{A}, \beta \models F$ .  
Otherwise  $F$  is called **unsatisfiable**.

# Entailment and Equivalence

---

$F$  entails (implies)  $G$  (or  $G$  is a consequence of  $F$ ), written  $F \models G$ , if for all  $\mathcal{A} \in \Sigma\text{-Alg}$  and  $\beta \in X \rightarrow U_{\mathcal{A}}$ , we have

$$\mathcal{A}, \beta \models F \quad \Rightarrow \quad \mathcal{A}, \beta \models G$$

$F$  and  $G$  are called **equivalent**, written  $F \equiv G$ , if for all  $\mathcal{A} \in \Sigma\text{-Alg}$  and  $\beta \in X \rightarrow U_{\mathcal{A}}$  we have

$$\mathcal{A}, \beta \models F \quad \Leftrightarrow \quad \mathcal{A}, \beta \models G$$

# Entailment and Equivalence

---

Proposition 3.1:

$F \models G$  if and only if  $(F \rightarrow G)$  is valid

Proposition 3.2:

$F \models\!\!\models G$  if and only if  $(F \leftrightarrow G)$  is valid.

Extension to sets of formulas  $N$  as in propositional logic, e. g.:

$N \models F \quad :\Leftrightarrow \quad$  for all  $\mathcal{A} \in \Sigma\text{-Alg}$  and  $\beta \in X \rightarrow U_{\mathcal{A}}$ :  
if  $\mathcal{A}, \beta \models G$  for all  $G \in N$ , then  $\mathcal{A}, \beta \models F$ .

# Validity vs. Unsatisfiability

---

Validity and unsatisfiability are just two sides of the same medal as explained by the following proposition.

Proposition 3.3:

Let  $F$  and  $G$  be formulas, let  $N$  be a set of formulas. Then

- (i)  $F$  is valid if and only if  $\neg F$  is unsatisfiable.
- (ii)  $F \models G$  if and only if  $F \wedge \neg G$  is unsatisfiable.
- (iii)  $N \models G$  if and only if  $N \cup \{\neg G\}$  is unsatisfiable.

Hence in order to design a theorem prover (validity checker) it is sufficient to design a checker for unsatisfiability.

# Substitution Lemma

---

Lemma 3.4:

Let  $\mathcal{A}$  be a  $\Sigma$ -algebra, let  $\beta$  be an assignment, let  $\sigma$  be a substitution. Then for any  $\Sigma$ -term  $t$

$$\mathcal{A}(\beta)(t\sigma) = \mathcal{A}(\beta \circ \sigma)(t),$$

where  $\beta \circ \sigma : X \rightarrow U_{\mathcal{A}}$  is the assignment  $\beta \circ \sigma(x) = \mathcal{A}(\beta)(x\sigma)$ .

Proposition 3.5:

Let  $\mathcal{A}$  be a  $\Sigma$ -algebra, let  $\beta$  be an assignment, let  $\sigma$  be a substitution. Then for every  $\Sigma$ -formula  $F$

$$\mathcal{A}(\beta)(F\sigma) = \mathcal{A}(\beta \circ \sigma)(F).$$

# Substitution Lemma

---

Corollary 3.6:

$$\mathcal{A}, \beta \models F\sigma \iff \mathcal{A}, \beta \circ \sigma \models F$$

These theorems basically express that the syntactic concept of substitution corresponds to the semantic concept of an assignment.



## Two Lemmas

---

Lemma 3.7:

Let  $\mathcal{A}$  be a  $\Sigma$ -algebra. Let  $F$  be a  $\Sigma$ -formula with free variables  $x_1, \dots, x_n$ .

Then

$$\mathcal{A} \models \forall x_1, \dots, x_n F \text{ if and only if } \mathcal{A} \models F.$$

Note that it is not possible to replace  $\mathcal{A} \models \dots$  by  $\mathcal{A}, \beta \models \dots$  in Lemma 3.7.

## Two Lemmas

---

Lemma 3.8:

Let  $\mathcal{A}$  be a  $\Sigma$ -algebra.

Let  $F$  be a  $\Sigma$ -formula with free variables  $x_1, \dots, x_n$ .

Let  $\sigma$  be a substitution and let  $y_1, \dots, y_m$  be the free variables of  $F\sigma$ . Then

$$\mathcal{A} \models \forall x_1, \dots, x_n F \text{ implies } \mathcal{A} \models \forall y_1, \dots, y_m F\sigma .$$

## 3.4 Algorithmic Problems

---

**Validity( $F$ ):**  $\models F$  ?

**Satisfiability( $F$ ):**  $F$  satisfiable?

**Entailment( $F, G$ ):** does  $F$  entail  $G$ ?

**Model( $\mathcal{A}, F$ ):**  $\mathcal{A} \models F$ ?

**Solve( $\mathcal{A}, F$ ):** find an assignment  $\beta$  such that  $\mathcal{A}, \beta \models F$ .

**Solve( $F$ ):** find a substitution  $\sigma$  such that  $\models F\sigma$ .

**Abduce( $F$ ):** find  $G$  with “certain properties” such that  $G \models F$ .

# Theory of an Algebra

---

Let  $\mathcal{A} \in \Sigma\text{-Alg}$ . The (first-order) theory of  $\mathcal{A}$  is defined as

$$\text{Th}(\mathcal{A}) = \{ G \in F_{\Sigma}(X) \mid \mathcal{A} \models G \}$$

Problem of axiomatizability:

Given an algebra  $\mathcal{A}$  (or a class of algebras) can one axiomatize  $\text{Th}(\mathcal{A})$ , that is, can one write down a formula  $F$  (or a recursively enumerable set  $F$  of formulas) such that

$$\text{Th}(\mathcal{A}) = \{ G \mid F \models G \}?$$

## Two Interesting Theories

---

Let  $\Sigma_{\text{Pres}} = (\{0/0, s/1, +/2\}, \{<\})$  and  $\mathbb{N}_+ = (\mathbb{N}, 0, s, +, <)$  its standard interpretation on the natural numbers.

$\text{Th}(\mathbb{N}_+)$  is called **Presburger arithmetic** (M. Presburger, 1929).

(There is no essential difference when one, instead of  $\mathbb{N}$ , considers the integer numbers  $\mathbb{Z}$  as standard interpretation.)

Presburger arithmetic is decidable in 3EXPTIME (D. Oppen, JCSS, 16(3):323–332, 1978), and in 2EXPSPACE, using automata-theoretic methods (and there is a constant  $c \geq 0$  such that  $\text{Th}(\mathbb{Z}_+) \notin \text{NTIME}(2^{2^{cn}})$ ).

## Two Interesting Theories

---

However,  $\mathbb{N}_* = (\mathbb{N}, 0, s, +, *, <)$ , the standard interpretation of  $\Sigma_{PA} = (\{0/0, s/1, +/2, */2\}, \{<\})$ , has as theory the so-called **Peano arithmetic** which is undecidable and not even recursively enumerable.

# (Non-)Computability Results

1. For most signatures  $\Sigma$ , validity is undecidable for  $\Sigma$ -formulas.  
(One can easily encode Turing machines in most signatures.)
2. Gödel's completeness theorem:  
For each signature  $\Sigma$ , the set of valid  $\Sigma$ -formulas is recursively enumerable.  
(We will prove this by giving complete deduction systems.)
3. Gödel's incompleteness theorem:  
For  $\Sigma = \Sigma_{PA}$  and  $\mathbb{N}_* = (\mathbb{N}, 0, s, +, *, <)$ , the theory  $\text{Th}(\mathbb{N}_*)$  is not recursively enumerable.

These complexity results motivate the study of subclasses of formulas (**fragments**) of first-order logic

# Some Decidable Fragments

---

Some decidable fragments:

- **Monadic class**: no function symbols, all predicates unary; validity is NEXPTIME-complete.
- Variable-free formulas without equality: satisfiability is NP-complete. (why?)
- Variable-free Horn clauses (clauses with at most one positive atom): entailment is decidable in linear time.
- Finite model checking is decidable in exponential time and PSPACE-complete.



## 3.5 Normal Forms and Skolemization

---

Study of normal forms motivated by

- reduction of logical concepts,
- efficient data structures for theorem proving.

The main problem in first-order logic is the treatment of quantifiers. The subsequent normal form transformations are intended to eliminate many of them.

# Prenex Normal Form (Traditional)

---

Prenex formulas have the form

$$Q_1x_1 \dots Q_nx_n F,$$

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

we call  $Q_1x_1 \dots Q_nx_n$  the **quantifier prefix** and  $F$  the **matrix** of the formula.

# Prenex Normal Form (Traditional)

---

Computing prenex normal form by the reduction system  $\Rightarrow_P$ :

$$H[(F \leftrightarrow G)]_P \Rightarrow_P H[(F \rightarrow G) \wedge (G \rightarrow F)]_P$$

$$H[\neg Qx F]_P \Rightarrow_P H[\bar{Q}x \neg F]_P$$

$$H[((Qx F) \circ G)]_P \Rightarrow_P H[Qy (F\{x \mapsto y\} \circ G)]_P,$$
$$\circ \in \{\wedge, \vee\}$$

$$H[((Qx F) \rightarrow G)]_P \Rightarrow_P H[\bar{Q}y (F\{x \mapsto y\} \rightarrow G)]_P,$$

$$H[(F \circ (Qx G))]_P \Rightarrow_P H[Qy (F \circ G\{x \mapsto y\})]_P,$$
$$\circ \in \{\wedge, \vee, \rightarrow\}$$

Here  $y$  is always assumed to be some fresh variable and  $\bar{Q}$  denotes the quantifier **dual** to  $Q$ , i. e.,  $\bar{\forall} = \exists$  and  $\bar{\exists} = \forall$ .

# Skolemization

---

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

Transformation  $\Rightarrow_S$

(to be applied outermost, *not* in subformulas):

$$\forall x_1, \dots, x_n \exists y F \Rightarrow_S \forall x_1, \dots, x_n F\{y \mapsto f(x_1, \dots, x_n)\}$$

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

# Skolemization

---

**Together:**  $F \Rightarrow_P^* \underbrace{G}_{\text{prenex}} \Rightarrow_S^* \underbrace{H}_{\text{prenex, no } \exists}$

Theorem 3.9:

Let  $F$ ,  $G$ , and  $H$  as defined above and closed. Then

- (i)  $F$  and  $G$  are equivalent.
- (ii)  $H \models G$  but the converse is not true in general.
- (iii)  $G$  satisfiable (w. r. t.  $\Sigma$ -Alg)  $\Leftrightarrow H$  satisfiable (w. r. t.  $\Sigma'$ -Alg)  
where  $\Sigma' = (\Omega \cup SKF, \Pi)$  if  $\Sigma = (\Omega, \Pi)$ .

# The Complete Picture

---

$$F \Rightarrow_P^* Q_1 y_1 \dots Q_n y_n G \quad (G \text{ quantifier-free})$$

$$\Rightarrow_S^* \forall x_1, \dots, x_m H \quad (m \leq n, H \text{ quantifier-free})$$

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

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

Note: The variables in the clauses are implicitly universally quantified.

# The Complete Picture

---

Theorem 3.10:

Let  $F$  be closed. Then  $F' \models F$ .

(The converse is not true in general.)

Theorem 3.11:

Let  $F$  be closed. Then  $F$  is satisfiable if and only if  $F'$  is satisfiable if and only if  $N$  is satisfiable

# Optimization

---

The normal form algorithm described so far leaves lots of room for optimization. Note that we only can preserve satisfiability anyway due to Skolemization.

- the size of the CNF is exponential when done naively; the transformations we introduced already for propositional logic avoid this exponential growth;
- we want to preserve the original formula structure;
- we want small arity of Skolem functions (see next section).



## 3.6 Getting Skolem Functions with Small Arity

---

A clause set that is better suited for automated theorem proving can be obtained using the following steps:

- eliminate trivial subformulas
- replace beneficial subformulas
- produce a negation normal form (NNF)
- apply miniscoping
- rename all variables
- Skolemize
- push quantifiers upward
- apply distributivity

We start with a closed formula.

# Elimination of Trivial Subformulas

---

Eliminate subformulas  $\top$  and  $\perp$  essentially as in the propositional case modulo associativity/commutativity of  $\wedge$ ,  $\vee$ :

$$H[(F \wedge \top)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \vee \perp)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \leftrightarrow \perp)]_p \Rightarrow_{\text{OCNF}} H[\neg F]_p$$

$$H[(F \leftrightarrow \top)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[(F \vee \top)]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

$$H[(F \wedge \perp)]_p \Rightarrow_{\text{OCNF}} H[\perp]_p$$

$$H[\neg \top]_p \Rightarrow_{\text{OCNF}} H[\perp]_p$$

$$H[\neg \perp]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

# Elimination of Trivial Subformulas

---

Eliminate subformulas  $\top$  and  $\perp$  essentially as in the propositional case modulo associativity/commutativity of  $\wedge$ ,  $\vee$ :

$$H[(F \rightarrow \perp)]_p \Rightarrow_{\text{OCNF}} H[\neg F]_p$$

$$H[(F \rightarrow \top)]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

$$H[(\perp \rightarrow F)]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

$$H[(\top \rightarrow F)]_p \Rightarrow_{\text{OCNF}} H[F]_p$$

$$H[Qx \top]_p \Rightarrow_{\text{OCNF}} H[\top]_p$$

$$H[Qx \perp]_p \Rightarrow_{\text{OCNF}} H[\perp]_p$$

## Replacement of Beneficial Subformulas

---

The functions  $\nu$  and  $\bar{\nu}$  that give us an overapproximation for the number of clauses generated by a formula are extended to quantified formulas by

$$\nu(\forall x F) = \nu(\exists x F) = \nu(F),$$

$$\bar{\nu}(\forall x F) = \bar{\nu}(\exists x F) = \bar{\nu}(F).$$

The other cases are defined as for propositional formulas.

# Replacement of Beneficial Subformulas

---

Introduce top-down fresh predicates for beneficial subformulas:

$$H[F]_p \Rightarrow_{\text{OCNF}} H[P(x_1, \dots, x_n)]_p \wedge \text{def}(H, p, P, F)$$

if  $\nu(H[F]_p) > \nu(H[P(\dots)]_p \wedge \text{def}(H, p, P, F))$ ,

where  $\{x_1, \dots, x_n\}$  are the free variables in  $F$ ,

$P/n$  is a predicate new to  $H[F]_p$ ,

$\text{def}(H, p, P, F)$  is defined by

$$\begin{aligned} &\forall x_1, \dots, x_n (P(x_1, \dots, x_n) \rightarrow F), \text{ if } \text{pol}(H, p) = 1, \\ &\forall x_1, \dots, x_n (F \rightarrow P(x_1, \dots, x_n)), \text{ if } \text{pol}(H, p) = -1, \\ &\forall x_1, \dots, x_n (P(x_1, \dots, x_n) \leftrightarrow F), \text{ if } \text{pol}(H, p) = 0. \end{aligned}$$

## Replacement of Beneficial Subformulas

---

As in the propositional case, one can test  $\nu(H[F]_p) > \nu(H[P]_p \wedge \text{def}(H, p, P, F))$  in constant time without actually computing  $\nu$ .

# Negation Normal Form (NNF)

---

Apply the reduction system  $\Rightarrow_{\text{NNF}}$ :

$$H[F \leftrightarrow G]_p \Rightarrow_{\text{NNF}} H[(F \rightarrow G) \wedge (G \rightarrow F)]_p$$

if  $\text{pol}(H, p) = 1$  or  $\text{pol}(H, p) = 0$ .

$$H[F \leftrightarrow G]_p \Rightarrow_{\text{NNF}} H[(F \wedge G) \vee (\neg G \wedge \neg F)]_p$$

if  $\text{pol}(H, p) = -1$ .

$$H[F \rightarrow G]_p \Rightarrow_{\text{NNF}} H[\neg F \vee G]_p$$

# Negation Normal Form (NNF)

---

$$\begin{aligned}H[\neg\neg F]_p &\Rightarrow_{\text{NNF}} H[F]_p \\H[\neg(F \vee G)]_p &\Rightarrow_{\text{NNF}} H[\neg F \wedge \neg G]_p \\H[\neg(F \wedge G)]_p &\Rightarrow_{\text{NNF}} H[\neg F \vee \neg G]_p \\H[\neg Qx F]_p &\Rightarrow_{\text{NNF}} H[\bar{Q}x \neg F]_p\end{aligned}$$



# Miniscoping

---

Apply the reduction system  $\Rightarrow_{MS}$  modulo associativity and commutativity of  $\wedge, \vee$ . For the rules below we assume that  $x$  occurs freely in  $F, F'$ , but  $x$  does not occur freely in  $G$ :

$$H[Qx (F \wedge G)]_p \Rightarrow_{MS} H[(Qx F) \wedge G]_p$$

$$H[Qx (F \vee G)]_p \Rightarrow_{MS} H[(Qx F) \vee G]_p$$

$$H[\forall x (F \wedge F')]_p \Rightarrow_{MS} H[(\forall x F) \wedge (\forall x F')]_p$$

$$H[\exists x (F \vee F')]_p \Rightarrow_{MS} H[(\exists x F) \vee (\exists x F')]_p$$

$$H[Qx G]_p \Rightarrow_{MS} H[G]_p$$

## Variable Renaming

---

Rename all variables in  $H$  such that there are no two different positions  $p, q$  with  $H|_p = Q \times F$  and  $H|_q = Q' \times G$ .

# Standard Skolemization

---

Apply the reduction system:

$$H[\exists x F]_p \Rightarrow_{\text{SK}} H[F\{x \mapsto f(y_1, \dots, y_n)\}]_p$$

where  $p$  has minimal length,

$\{y_1, \dots, y_n\}$  are the free variables in  $\exists x F$ ,  
and  $f/n$  is a new function symbol to  $H$ .

## Final Steps

---

Apply the reduction system modulo commutativity of  $\wedge$ ,  $\vee$  to push  $\forall$  upward:

$$H[(\forall x F) \wedge G]_p \Rightarrow_{\text{OCNF}} H[\forall x (F \wedge G)]_p$$

$$H[(\forall x F) \vee G]_p \Rightarrow_{\text{OCNF}} H[\forall x (F \vee G)]_p$$

Note that variable renaming ensures that  $x$  does not occur in  $G$ .

## Final Steps

---

Apply the reduction system modulo commutativity of  $\wedge$ ,  $\vee$  to push disjunctions downward:

$$H[(F \wedge F') \vee G]_p \Rightarrow_{\text{CNF}} H[(F \vee G) \wedge (F' \vee G)]_p$$

## 3.7 Herbrand Interpretations

---

From now on we shall consider FOL without equality.

We assume that  $\Omega$  contains at least one constant symbol.

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

- $U_{\mathcal{A}} = T_{\Sigma}$  (= the set of ground terms over  $\Sigma$ )
- $f_{\mathcal{A}} : (s_1, \dots, s_n) \mapsto f(s_1, \dots, s_n)$ ,  $f/n \in \Omega$

In other words, *values are fixed* to be ground terms and *functions are fixed* to be the **term constructors**. Only predicate symbols  $P/m \in \Pi$  may be freely interpreted as relations  $P_{\mathcal{A}} \subseteq T_{\Sigma}^m$ .

# Herbrand Interpretations

---

Proposition 3.12:

Every set of ground atoms  $I$  uniquely determines an Herbrand interpretation  $\mathcal{A}$  via

$$(s_1, \dots, s_n) \in P_{\mathcal{A}} \text{ if and only if } P(s_1, \dots, s_n) \in I$$

Thus we shall identify Herbrand interpretations (over  $\Sigma$ ) with sets of  $\Sigma$ -ground atoms.

# Existence of Herbrand Models

---

An Herbrand interpretation  $I$  is called an **Herbrand model** of  $F$ , if  $I \models F$ .

The importance of Herbrand models lies in the following theorem, which we will prove later in this lecture:

Let  $N$  be a set of (universally quantified)  $\Sigma$ -clauses. Then the following properties are equivalent:

- (1)  $N$  has a model.
- (2)  $N$  has an Herbrand model (over  $\Sigma$ ).
- (3)  $G_\Sigma(N)$  has an Herbrand model (over  $\Sigma$ ).

where  $G_\Sigma(N) = \{ C\sigma \text{ ground clause} \mid (\forall \vec{x} C) \in N, \sigma : X \rightarrow T_\Sigma \}$  is the set of **ground instances** of  $N$ .



## 3.8 Inference Systems and Proofs

---

Inference systems  $\Gamma$  (proof calculi) are sets of tuples

$$(F_1, \dots, F_n, F_{n+1}), \quad n \geq 0,$$

called *inferences*, and written

$$\frac{\overbrace{F_1 \dots F_n}^{\text{premises}}}{\underbrace{F_{n+1}}_{\text{conclusion}}} \quad \textit{side condition}$$

*Clausal inference system*: premises and conclusions are clauses. One also considers inference systems over other data structures.

# Inference Systems

---

Inference systems  $\Gamma$  are shorthands for reduction systems over sets of formulas. If  $N$  is a set of formulas, then

$$\frac{\overbrace{F_1 \dots F_n}^{\text{premises}}}{\underbrace{F_{n+1}}_{\text{conclusion}}} \quad \textit{side condition}$$

is a shorthand for

$$N \cup \{F_1, \dots, F_n\} \Rightarrow_{\Gamma} N \cup \{F_1, \dots, F_n\} \cup \{F_{n+1}\}$$

*if side condition*

# Proofs

---

A **proof** in  $\Gamma$  of a formula  $F$  from a set of formulas  $N$  (called **assumptions**) is a sequence  $F_1, \dots, F_k$  of formulas where

(i)  $F_k = F$ ,

(ii) for all  $1 \leq i \leq k$ :  $F_i \in N$  or there exists an inference

$$\frac{F_{m_1} \dots F_{m_n}}{F_i}$$

in  $\Gamma$ , such that  $0 \leq m_j < i$ , for  $1 \leq j \leq n$ .

# Soundness and Completeness

---

Provability  $\vdash_{\Gamma}$  of  $F$  from  $N$  in  $\Gamma$ :

$N \vdash_{\Gamma} F$  if there exists a proof in  $\Gamma$  of  $F$  from  $N$ .

$\Gamma$  is called **sound**, if

$$\frac{F_1 \dots F_n}{F} \in \Gamma \text{ implies } F_1, \dots, F_n \models F$$

$\Gamma$  is called **complete**, if

$$N \models F \text{ implies } N \vdash_{\Gamma} F$$

$\Gamma$  is called **refutationally complete**, if

$$N \models \perp \text{ implies } N \vdash_{\Gamma} \perp$$

# Soundness and Completeness

---

Proposition 3.13:

- (i) Let  $\Gamma$  be sound. Then  $N \vdash_{\Gamma} F \Rightarrow N \models F$
- (ii) If  $N \vdash_{\Gamma} F$  then there exist finitely many  $F_1, \dots, F_n \in N$  such that  $F_1, \dots, F_n \vdash_{\Gamma} F$

## Reduced Proofs

---

The definition of a proof of  $F$  given above admits sequences  $F_1, \dots, F_k$  of formulas where some  $F_i$  are not ancestors of  $F_k = F$  (i. e., some  $F_i$  are not actually used to derive  $F$ ).

A proof is called **reduced**, if every  $F_i$  with  $i < k$  is an ancestor of  $F_k$ .

We obtain a reduced proof from a proof by marking first  $F_k$  and then recursively all the premises used to derive a marked conclusion, and by deleting all non-marked formulas in the end.



# Mandatory vs. Admissible Inferences

---

It is useful to distinguish between two kinds of inferences:

- Mandatory (required) inferences:
  - Must be performed to ensure refutational completeness.
  - The less, the better.
- Optional (admissible) inferences:
  - May be performed, if useful.

We will first consider only mandatory inferences.



## 3.9 Ground (or propositional) Resolution

---

We observe that propositional clauses and ground clauses are essentially the same, as long as we do not consider equational atoms.

In this section we only deal with ground clauses.

Unlike in Section 2 we admit duplicated literals in clauses, i. e., we treat clauses like multisets of literals, not like sets.

# The Resolution Calculus *Res*

---

Resolution inference rule:

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

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

(Positive) factorization inference rule:

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

## The Resolution Calculus *Res*

---

These are **schematic inference rules**; for each substitution of the **schematic variables**  $C$ ,  $D$ , and  $A$ , by ground clauses and ground atoms, respectively, we obtain an inference.

We treat “ $\vee$ ” as associative and commutative, hence  $A$  and  $\neg A$  can occur anywhere in the clauses; moreover, when we write  $C \vee A$ , etc., this includes unit clauses, that is,  $C = \perp$ .

## Sample Refutation

---

1	$\neg P(f(c)) \vee \neg P(f(c)) \vee Q(b)$	(given)
2	$P(f(c)) \vee Q(b)$	(given)
3	$\neg P(g(b, c)) \vee \neg Q(b)$	(given)
4	$P(g(b, c))$	(given)
5	$\neg P(f(c)) \vee Q(b) \vee Q(b)$	(Res. 2 into 1)
6	$\neg P(f(c)) \vee Q(b)$	(Fact. 5)
7	$Q(b) \vee Q(b)$	(Res. 2 into 6)
8	$Q(b)$	(Fact. 7)
9	$\neg P(g(b, c))$	(Res. 8 into 3)
10	$\perp$	(Res. 4 into 9)

# Soundness of Resolution

---

Theorem 3.14:

Ground first-order resolution is sound.

Note: In ground first-order logic we have (like in propositional logic):

1.  $\mathcal{B} \models L_1 \vee \dots \vee L_n$  if and only if there exists  $i$ :  $\mathcal{B} \models L_i$ .
2.  $\mathcal{B} \models A$  or  $\mathcal{B} \models \neg A$ .

This does not hold for formulas with variables!

## 3.10 Refutational Completeness of Resolution

---

How to show refutational completeness of ground resolution:

- We have to show:  $N \models \perp \Rightarrow N \vdash_{Res} \perp$ ,  
or equivalently: If  $N \not\vdash_{Res} \perp$ , then  $N$  has a model.
- Idea: Suppose that we have computed sufficiently many inferences (and not derived  $\perp$ ).
- Now order the clauses in  $N$  according to some appropriate ordering, inspect the clauses in ascending order, and construct a series of Herbrand interpretations.
- The limit interpretation can be shown to be a model of  $N$ .

## Closure of Clause Sets under $Res$

---

$$Res(N) = \{ C \mid C \text{ is conclusion of an inference in } Res \\ \text{with premises in } N \}$$

$$Res^0(N) = N$$

$$Res^{n+1}(N) = Res(Res^n(N)) \cup Res^n(N), \text{ for } n \geq 0$$

$$Res^*(N) = \bigcup_{n \geq 0} Res^n(N)$$

$N$  is called **saturated** (w. r. t. resolution), if  $Res(N) \subseteq N$ .

# Closure of Clause Sets under $Res$

---

Proposition 3.15:

- (i)  $Res^*(N)$  is saturated.
- (ii)  $Res$  is refutationally complete, if and only if for each set  $N$  of ground clauses:

$$N \models \perp \text{ implies } \perp \in Res^*(N)$$



# Clause Orderings

---

1. We assume that  $\succ$  is any fixed ordering on ground atoms that is *total* and *well-founded*. (There exist many such orderings, e. g., the length-based ordering on atoms when these are viewed as words over a suitable alphabet.)

2. Extend  $\succ$  to an ordering  $\succ_L$  on ground literals:

$$\begin{aligned} [\neg]A \succ_L [\neg]B & \text{ , if } A \succ B \\ \neg A \succ_L A \end{aligned}$$

3. Extend  $\succ_L$  to an ordering  $\succ_C$  on ground clauses:

$\succ_C = (\succ_L)_{\text{mul}}$ , the multiset extension of  $\succ_L$ .

*Notation:*  $\succ$  also for  $\succ_L$  and  $\succ_C$ .

## Example

---

Suppose  $A_5 \succ A_4 \succ A_3 \succ A_2 \succ A_1 \succ A_0$ . Then:

$$\begin{aligned} & A_1 \vee \neg A_5 \\ \succ & A_3 \vee \neg A_4 \\ \succ & \neg A_1 \vee A_3 \vee A_4 \\ \succ & A_1 \vee \neg A_2 \\ \succ & \neg A_1 \vee A_2 \\ \succ & A_1 \vee A_1 \vee A_2 \\ \succ & A_0 \vee A_1 \end{aligned}$$

# Properties of the Clause Ordering

---

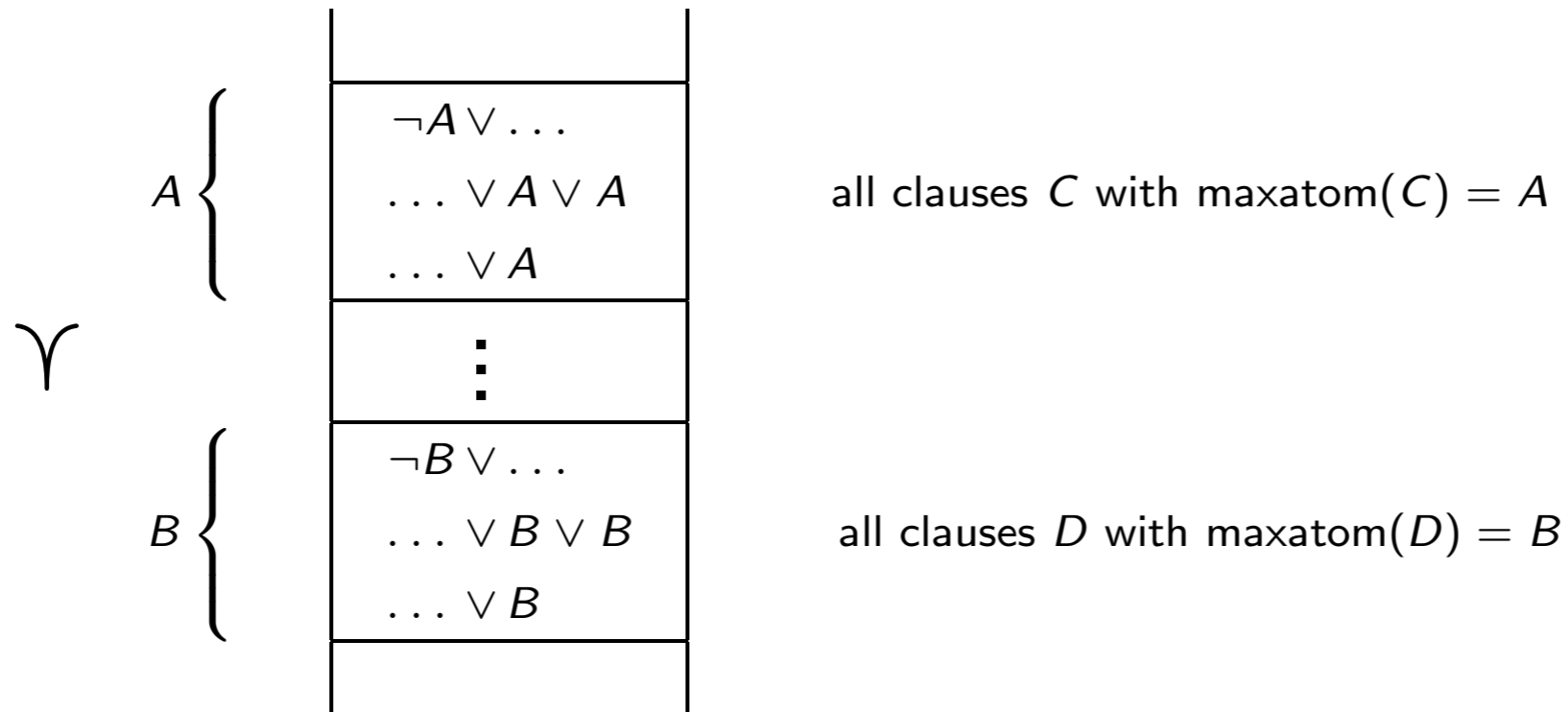
Proposition 3.16:

1. The orderings on literals and clauses are total and well-founded.
2. Let  $C$  and  $D$  be clauses with  
 $A = \text{maxatom}(C)$ ,  $B = \text{maxatom}(D)$ ,  
where  $\text{maxatom}(C)$  denotes the maximal atom in  $C$ .
  - (i) If  $A \succ B$  then  $C \succ D$ .
  - (ii) If  $A = B$ ,  $A$  occurs negatively in  $C$  but only positively in  $D$ ,  
then  $C \succ D$ .

# Stratified Structure of Clause Sets

---

Let  $A \succ B$ . Clause sets are then stratified in this form:



# Construction of Interpretations

---

Given: set  $N$  of ground clauses, atom ordering  $\succ$ .

Wanted: Herbrand interpretation  $I$  such that

$$I \models N \quad \text{if } N \text{ is saturated and } \perp \notin N$$

Construction according to  $\succ$ , starting with the smallest clause.

## Main Ideas of the Construction

---

- Clauses are considered in the order given by  $\succ$ .
- When considering  $C$ , one already has an interpretation so far available ( $I_C$ ). Initially  $I_C = \emptyset$ .
- If  $C$  is true in this interpretation, nothing needs to be changed.
- Otherwise, one would like to change the interpretation such that  $C$  becomes true.

## Main Ideas of the Construction

---

- Changes should, however, be *monotone*. One never deletes atoms from the interpretation, and the truth value of clauses smaller than  $C$  should not change from true to false.
- Hence, one adds  $\Delta_C = \{A\}$ , if and only if  $C$  is false in  $I_C$ , if  $A$  occurs positively in  $C$  (*adding  $A$  will make  $C$  become true*) and if this occurrence in  $C$  is strictly maximal in the ordering on literals (*changing the truth value of  $A$  has no effect on smaller clauses*). Otherwise,  $\Delta_C = \emptyset$ .

## Main Ideas of the Construction

---

- We say that the construction fails for a clause  $C$ , if  $C$  is false in  $I_C$  and  $\Delta_C = \emptyset$ .
- We will show: If there are clauses for which the construction fails, then some inference with the smallest such clause (the so-called “minimal counterexample”) has not been computed. Otherwise, the limit interpretation is a model of all clauses.



# Construction of Candidate Interpretations

---

Let  $N, \succ$  be given. We define sets  $I_C$  and  $\Delta_C$  for all ground clauses  $C$  over the given signature inductively over  $\succ$ :

$$I_C := \bigcup_{C \succ D} \Delta_D$$

$$\Delta_C := \begin{cases} \{A\}, & \text{if } C \in N, C = C' \vee A, A \succ C', I_C \not\models C \\ \emptyset, & \text{otherwise} \end{cases}$$

We say that  $C$  **produces**  $A$ , if  $\Delta_C = \{A\}$ .

Note that the definitions satisfy the conditions of Thm. 1.8; so they are well-defined even if  $\{D \mid C \succ D\}$  is infinite.

## Construction of Candidate Interpretations

---

The **candidate interpretation** for  $N$  (w. r. t.  $\succ$ ) is given as  $I_N^\succ := \bigcup_C \Delta_C$ .  
(We also simply write  $I_N$  or  $I$  for  $I_N^\succ$  if  $\succ$  is either irrelevant or known from the context.)

## Example

Let  $A_5 \succ A_4 \succ A_3 \succ A_2 \succ A_1 \succ A_0$  (max. literals in red)

	clauses $C$	$I_C$	$\Delta_C$	Remarks
7	$\neg A_1 \vee A_5$	$\{A_1, A_2, A_4\}$	$\{A_5\}$	
6	$\neg A_1 \vee A_3 \vee \neg A_4$	$\{A_1, A_2, A_4\}$	$\emptyset$	max. lit. $\neg A_4$ neg.; <i>min. counter-ex.</i>
5	$A_0 \vee \neg A_1 \vee A_3 \vee A_4$	$\{A_1, A_2\}$	$\{A_4\}$	$A_4$ maximal
4	$\neg A_1 \vee A_2$	$\{A_1\}$	$\{A_2\}$	$A_2$ maximal
3	$A_1 \vee A_2$	$\{A_1\}$	$\emptyset$	true in $I_C$
2	$A_0 \vee A_1$	$\emptyset$	$\{A_1\}$	$A_1$ maximal
1	$\neg A_0$	$\emptyset$	$\emptyset$	true in $I_C$

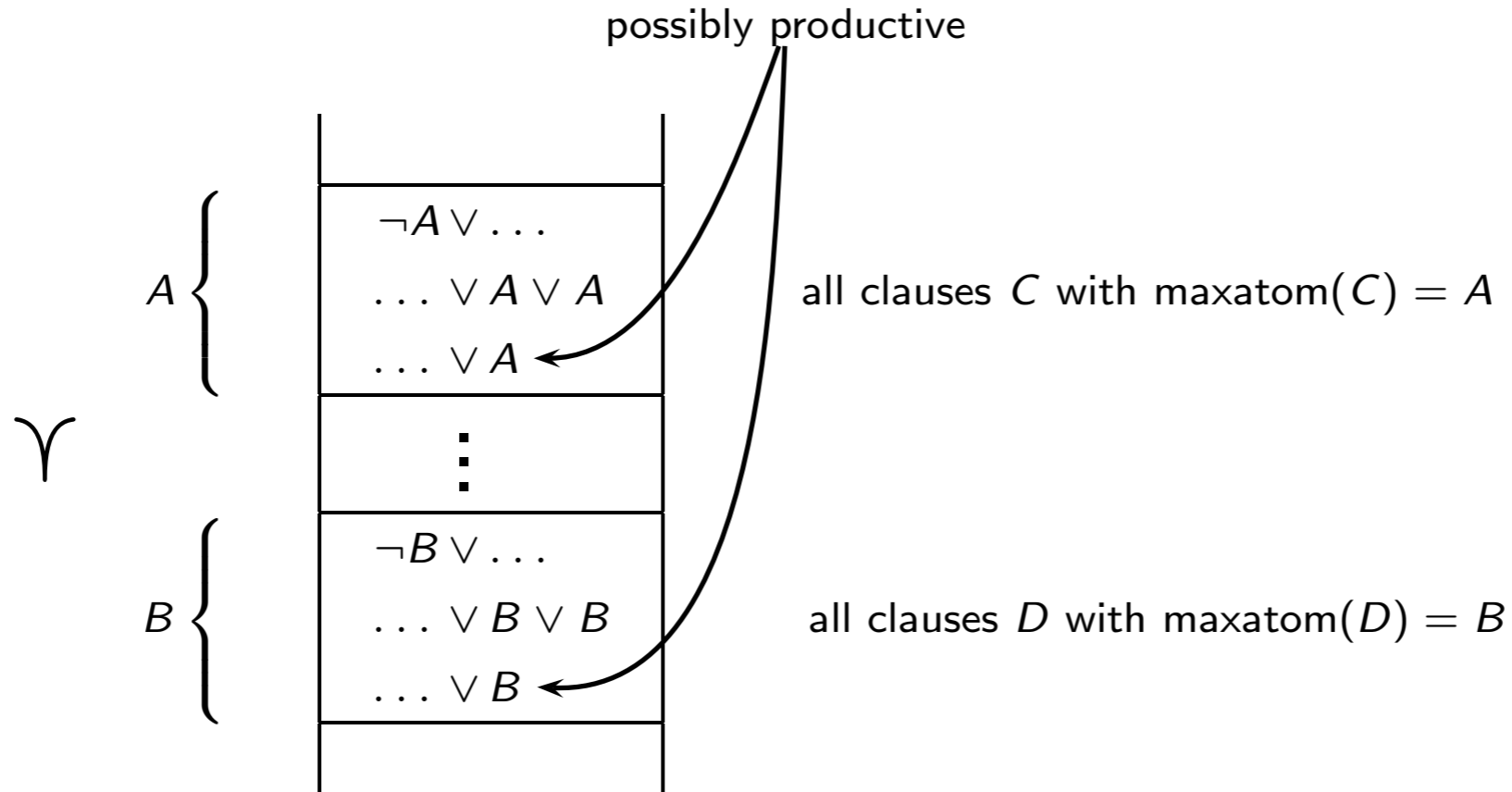
$I = \{A_1, A_2, A_4, A_5\}$  is not a model of the clause set

$\Rightarrow$  there exists a **counterexample**.

# Structure of $N, \succ$

---

Let  $A \succ B$ . Note that producing a new atom does not change the truth value of smaller clauses.



## Some Properties of the Construction

---

Proposition 3.17:

- (i) If  $D = D' \vee \neg A$ , then no  $C \succeq D$  produces  $A$ .
- (ii) If  $I_D \models D$ , then  $I_C \models D$  for every  $C \succeq D$  and  $I_N^\lambda \models D$ .
- (iii) If  $D = D' \vee A$  produces  $A$ ,  
then  $I_C \models D$  for every  $C \succ D$  and  $I_N^\lambda \models D$ .
- (iv) If  $D = D' \vee A$  produces  $A$ ,  
then  $I_C \not\models D'$  for every  $C \succeq D$  and  $I_N^\lambda \not\models D'$ .
- (v) If for every clause  $C \in N$ ,  $C$  is productive or  $I_C \models C$ , then  $I_N^\lambda \models N$ .

# Model Existence Theorem

---

Proposition 3.18:

Let  $\succ$  be a clause ordering.

If  $N$  is saturated w. r. t.  $Res$  and  $\perp \notin N$ ,

then for every clause  $C \in N$ ,  $C$  is productive or  $I_C \models C$ .

Theorem 3.19 (Bachmair & Ganzinger 1990):

Let  $\succ$  be a clause ordering.

If  $N$  is saturated w. r. t.  $Res$  and  $\perp \notin N$ , then  $I_N^\succ \models N$ .

Corollary 3.20:

Let  $N$  be saturated w. r. t.  $Res$ .

Then  $N \models \perp$  if and only if  $\perp \in N$ .

# Compactness of Propositional Logic

---

Lemma 3.21:

Let  $N$  be a set of propositional (or first-order ground) clauses.

Then  $N$  is unsatisfiable, if and only if some finite subset  $N' \subseteq N$  is unsatisfiable.

Theorem 3.22 (**Compactness for Propositional Formulas**):

Let  $S$  be a set of propositional (or first-order ground) formulas.

Then  $S$  is unsatisfiable, if and only if some finite subset  $S' \subseteq S$  is unsatisfiable.

## 3.11 General Resolution

---

Propositional (ground) resolution:

- refutationally complete,

- in its most naive version:

- not guaranteed to terminate for satisfiable sets of clauses,

- (improved versions do terminate, however)

- inferior to the CDCL procedure.

But: in contrast to the CDCL procedure, resolution can be easily extended to non-ground clauses.



## Observation

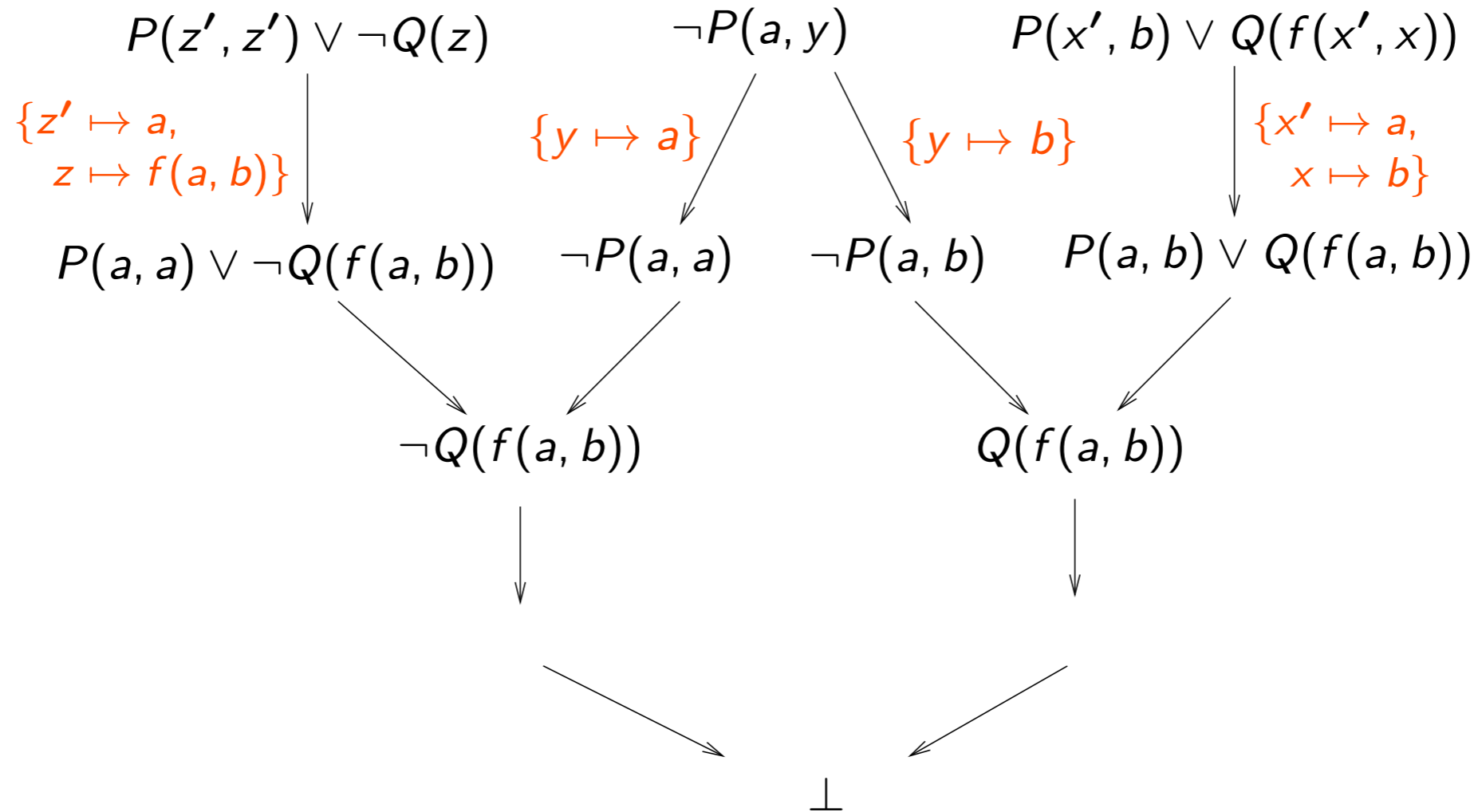
---

If  $\mathcal{A}$  is a model of an (implicitly universally quantified) clause  $C$ , then by Lemma 3.8 it is also a model of all (implicitly universally quantified) instances  $C\sigma$  of  $C$ .

Consequently, if we show that some instances of clauses in a set  $N$  are unsatisfiable, then we have also shown that  $N$  itself is unsatisfiable.

# General Resolution through Instantiation

Idea: instantiate clauses appropriately:



# General Resolution through Instantiation

---

Early approaches (Gilmore 1960, Davis and Putnam 1960):

- Generate ground instances of clauses.

- Try to refute the set of ground instances by resolution.

- If no contradiction is found, generate more ground instances.

Problems:

- More than one instance of a clause can participate in a proof.

- Even worse: There are infinitely many possible instances.

# General Resolution through Instantiation

---

Observation:

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

# General Resolution through Instantiation

---

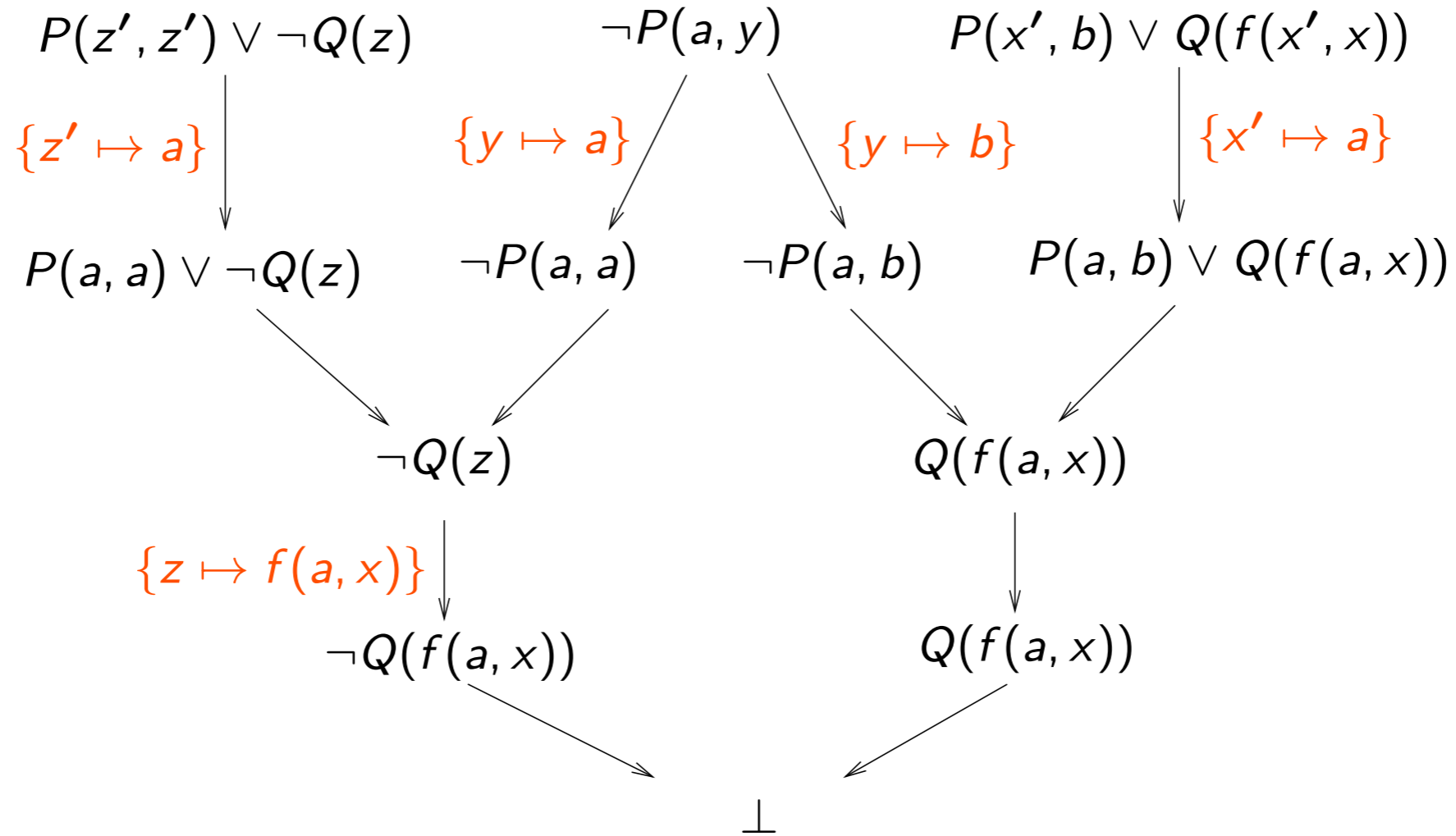
Idea (Robinson 1965):

Do not instantiate more than necessary to get complementary literals  
⇒ most general unifiers (mgu).

Calculus works with non-ground clauses;  
inferences with non-ground clauses represent infinite sets  
of ground inferences which are computed simultaneously  
⇒ lifting principle.

Computation of instances becomes a by-product of boolean reasoning.

# General Resolution through Instantiation



# Unification

---

Let  $E = \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}$  ( $s_i, t_i$  terms or atoms) be a multiset 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**.

# Unification

---

A substitution  $\sigma$  is called **more general** than a substitution  $\tau$ , denoted by  $\sigma \leq \tau$ , if there exists a substitution  $\rho$  such that  $\rho \circ \sigma = \tau$ , where  $(\rho \circ \sigma)(x) := (x\sigma)\rho$  is the composition of  $\sigma$  and  $\rho$  as mappings.  
(Note that  $\rho \circ \sigma$  has a finite domain as required for a substitution.)

If a unifier of  $E$  is more general than any other unifier of  $E$ , then we speak of a **most general unifier** of  $E$ , denoted by  $\text{mgu}(E)$ .



# Unification

---

Proposition 3.23:

- (i)  $\leq$  is a quasi-ordering on substitutions, and  $\circ$  is associative.
- (ii) If  $\sigma \leq \tau$  and  $\tau \leq \sigma$  (we write  $\sigma \sim \tau$  in this case), then  $x\sigma$  and  $x\tau$  are equal up to (bijective) variable renaming, for any  $x$  in  $X$ .

A substitution  $\sigma$  is called **idempotent**, if  $\sigma \circ \sigma = \sigma$ .

Proposition 3.24:

$\sigma$  is idempotent if and only if  $\text{dom}(\sigma) \cap \text{codom}(\sigma) = \emptyset$ .

# Rule-Based Naive Standard Unification

---

$$\begin{array}{l} t \doteq t, E \Rightarrow_{SU} E \\ f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n), E \Rightarrow_{SU} s_1 \doteq t_1, \dots, s_n \doteq t_n, E \\ f(\dots) \doteq g(\dots), E \Rightarrow_{SU} \perp \\ \quad \text{if } f \neq g \\ x \doteq t, E \Rightarrow_{SU} x \doteq t, E\{x \mapsto t\} \\ \quad \text{if } x \in \text{var}(E), x \notin \text{var}(t) \\ x \doteq t, E \Rightarrow_{SU} \perp \\ \quad \text{if } x \neq t, x \in \text{var}(t) \\ t \doteq x, E \Rightarrow_{SU} x \doteq t, E \\ \quad \text{if } t \notin X \end{array}$$

## SU: Main Properties

---

If  $E = \{x_1 \doteq u_1, \dots, x_k \doteq u_k\}$ , with  $x_i$  pairwise distinct,  $x_i \notin \text{var}(u_j)$ , then  $E$  is called an (equational problem in) **solved form** representing the solution

$$\sigma_E = \{x_1 \mapsto u_1, \dots, x_k \mapsto u_k\}.$$

Proposition 3.25:

If  $E$  is a solved form then  $\sigma_E$  is an mgu of  $E$ .

# SU: Main Properties

---

Theorem 3.26:

1. If  $E \Rightarrow_{SU} E'$  then  $\sigma$  is a unifier of  $E$  if and only if  $\sigma$  is a unifier of  $E'$
2. If  $E \Rightarrow_{SU}^* \perp$  then  $E$  is not unifiable.
3. If  $E \Rightarrow_{SU}^* E'$  with  $E'$  in solved form, then  $\sigma_{E'}$  is an mgu of  $E$ .

# Main Unification Theorem

---

Theorem 3.27:

$E$  is unifiable if and only if there is a most general unifier  $\sigma$  of  $E$ , such that  $\sigma$  is idempotent and  $\text{dom}(\sigma) \cup \text{codom}(\sigma) \subseteq \text{var}(E)$ .

# Rule-Based Polynomial Unification

---

Problem: Using  $\Rightarrow_{SU}$ , an *exponential growth* of terms is possible.

The following unification algorithm avoids this problem, at least if the final solved form is represented as a DAG.

# Rule-Based Polynomial Unification

---

$$t \doteq t, E \Rightarrow_{PU} E$$

$$f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n), E \Rightarrow_{PU} s_1 \doteq t_1, \dots, s_n \doteq t_n, E$$

$$f(\dots) \doteq g(\dots), E \Rightarrow_{PU} \perp$$

if  $f \neq g$

$$x \doteq y, E \Rightarrow_{PU} x \doteq y, E\{x \mapsto y\}$$

if  $x \in \text{var}(E), x \neq y$

$$x_1 \doteq t_1, \dots, x_n \doteq t_n, E \Rightarrow_{PU} \perp$$

if there are positions  $p_i$  with  
 $t_i|_{p_i} = x_{i+1}, t_n|_{p_n} = x_1$   
and some  $p_i \neq \varepsilon$

# Rule-Based Polynomial Unification

---

$$x \doteq t, E \Rightarrow_{PU} \perp$$

if  $x \neq t, x \in \text{var}(t)$

$$t \doteq x, E \Rightarrow_{PU} x \doteq t, E$$

if  $t \notin X$

$$x \doteq t, x \doteq s, E \Rightarrow_{PU} x \doteq t, t \doteq s, E$$

if  $t, s \notin X$  and  $|t| \leq |s|$



# Properties of PU

---

Theorem 3.28:

1. If  $E \Rightarrow_{PU} E'$  then  $\sigma$  is a unifier of  $E$  if and only if  $\sigma$  is a unifier of  $E'$
2. If  $E \Rightarrow_{PU}^* \perp$  then  $E$  is not unifiable.
3. If  $E \Rightarrow_{PU}^* E'$  with  $E'$  in solved form, then  $\sigma_{E'}$  is an mgu of  $E$ .

Note: The solved form of  $\Rightarrow_{PU}$  is different from the solved form obtained from  $\Rightarrow_{SU}$ . In order to obtain the unifier  $\sigma_{E'}$ , we have to sort the list of equality problems  $x_i \doteq t_i$  in such a way that  $x_i$  does not occur in  $t_j$  for  $j < i$ , and then we have to compose the substitutions  $\{x_1 \mapsto t_1\} \circ \cdots \circ \{x_k \mapsto t_k\}$ .

# Resolution for General Clauses

---

We obtain the resolution inference rules for non-ground clauses from the inference rules for ground clauses by replacing equality by unifiability:

**General resolution *Res*:**

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

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

## Resolution for General Clauses

---

For inferences with more than one premise, we assume that the variables in the premises are (bijectively) renamed such that they become different to any variable in the other premises.

We do not formalize this. Which names one uses for variables is otherwise irrelevant.

# Lifting Lemma

---

Lemma 3.29:

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

$$\frac{\begin{array}{ccc} D & & C \\ \downarrow \theta_1 & & \downarrow \theta_2 \\ D\theta_1 & & C\theta_2 \end{array}}{C'} \quad [\text{ground resolution}]$$

then there exists a substitution  $\rho$  such that

$$\frac{D \quad C}{C''} \quad [\text{general resolution}]$$
$$\downarrow \rho$$
$$C' = C''\rho$$

# Lifting Lemma

---

An analogous lifting lemma holds for factorization.

# Saturation of Sets of General Clauses

---

Corollary 3.30:

Let  $N$  be a set of general clauses saturated under  $Res$ , i. e.,  $Res(N) \subseteq N$ .

Then also  $G_{\Sigma}(N)$  is saturated, that is,

$$Res(G_{\Sigma}(N)) \subseteq G_{\Sigma}(N).$$

# Soundness for General Clauses

---

Proposition 3.31:

The general resolution calculus is sound.

# Herbrand's Theorem

---

Lemma 3.32:

Let  $N$  be a set of  $\Sigma$ -clauses, let  $\mathcal{A}$  be an interpretation.

Then  $\mathcal{A} \models N$  implies  $\mathcal{A} \models G_{\Sigma}(N)$ .

Lemma 3.33:

Let  $N$  be a set of  $\Sigma$ -clauses, let  $\mathcal{A}$  be an *Herbrand* interpretation.

Then  $\mathcal{A} \models G_{\Sigma}(N)$  implies  $\mathcal{A} \models N$ .



# Herbrand's Theorem

---

Theorem 3.34 (Herbrand):

A set  $N$  of  $\Sigma$ -clauses is satisfiable if and only if it has an Herbrand model over  $\Sigma$ .

Corollary 3.35:

A set  $N$  of  $\Sigma$ -clauses is satisfiable if and only if its set of ground instances  $G_{\Sigma}(N)$  is satisfiable.

# Refutational Completeness of General Resolution

---

Theorem 3.36:

Let  $N$  be a set of general clauses that is saturated w. r. t.  $Res$ .

Then  $N \models \perp$  if and only if  $\perp \in N$ .

## 3.12 Theoretical Consequences

---

We get some classical results on properties of first-order logic as easy corollaries.

# The Theorem of Löwenheim-Skolem

---

Theorem 3.37 (Löwenheim–Skolem):

Let  $\Sigma$  be a countable signature and let  $S$  be a set of closed  $\Sigma$ -formulas. Then  $S$  is satisfiable if and only if  $S$  has a model over a countable universe.

There exist more refined versions of this theorem. For instance, one can show that, if  $S$  has some infinite model, then  $S$  has a model with a universe of cardinality  $\kappa$  for every  $\kappa$  that is larger than or equal to the cardinality of the signature  $\Sigma$ .

# Compactness of Predicate Logic

---

Theorem 3.38 (Compactness Theorem for First-Order Logic):

Let  $S$  be a set of closed first-order formulas.

$S$  is unsatisfiable  $\Leftrightarrow$  some finite subset  $S' \subseteq S$  is unsatisfiable.

## 3.13 Ordered Resolution with Selection

---

Motivation: Search space for *Res* very large.

Ideas for improvement:

1. In the completeness proof (Model Existence Theorem 3.19) one only needs to resolve and factor maximal atoms  
⇒ if the calculus is restricted to inferences involving maximal atoms, the proof remains correct  
⇒ *ordering restrictions*
2. In the proof, it does not really matter with which negative literal an inference is performed  
⇒ choose a negative literal don't-care-nondeterministically  
⇒ *selection*

# Ordering Restrictions

---

In the completeness proof one only needs to resolve and factor maximal atoms. Therefore the proof remains correct, if we impose ordering restrictions on ground inferences.

(Ground) Ordered Resolution:

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

if  $A \succ L$  for all  $L$  in  $D$  and  $\neg A \succeq L$  for all  $L$  in  $C$ .

(Ground) Ordered Factorization:

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

if  $A \succeq L$  for all  $L$  in  $C$ .

# Ordering Restrictions

---

Problem: How to extend this to non-ground inferences?

In the completeness proof, we talk about (strictly) maximal literals of *ground* clauses.

In the non-ground calculus, we have to consider those literals that correspond to (strictly) maximal literals of ground instances.



# Ordering Restrictions

---

An ordering  $\succ$  on atoms (or terms) is called **stable under substitutions**, if  $A \succ B$  implies  $A\sigma \succ B\sigma$ .

Note:

- We can not require that  $A \succ B$  if and only if  $A\sigma \succ B\sigma$ .
- We can not require that  $\succ$  is total on non-ground atoms.

Consequence:

In the ordering restrictions for non-ground inferences, we have to replace  $\succ$  by  $\not\succeq$  and  $\succeq$  by  $\not\succ$ .

# Ordering Restrictions

---

Ordered Resolution:

$$\frac{D \vee B \quad C \vee \neg A}{(D \vee C)\sigma}$$

if  $\sigma = \text{mgu}(A, B)$  and  $B\sigma \not\prec L\sigma$  for all  $L$  in  $D$   
and  $\neg A\sigma \not\prec L\sigma$  for all  $L$  in  $C$ .

Ordered Factorization:

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

if  $\sigma = \text{mgu}(A, B)$  and  $A\sigma \not\prec L\sigma$  for all  $L$  in  $C$ .

# Selection Functions

---

Selection functions can be used to override ordering restrictions for individual clauses.

A **selection function** is a mapping

$$\text{sel} : C \mapsto \text{set of occurrences of } \textit{negative} \text{ literals in } C$$

Example of selection with selected literals indicated as  $\boxed{X}$ :

$$\boxed{\neg A} \vee \neg A \vee B$$

$$\boxed{\neg B_0} \vee \boxed{\neg B_1} \vee A$$

# Selection Functions

---

Intuition:

- If a clause has at least one selected literal, compute only inferences that involve a selected literal.
- If a clause has no selected literals, compute only inferences that involve a maximal literal.

# Resolution Calculus $Res_{sel}^{\succ}$

---

The resolution calculus  $Res_{sel}^{\succ}$  is parameterized by

- a selection function  $sel$
- and a well-founded ordering  $\succ$  on atoms that is total on ground atoms and stable under substitutions.

# Resolution Calculus $Res_{sel}^>$

---

(Ground) Ordered Resolution with Selection:

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

if the following conditions are satisfied:

- (i)  $A \succ L$  for all  $L$  in  $D$ ;
- (ii) nothing is selected in  $D \vee A$  by  $sel$ ;
- (iii)  $\neg A$  is selected in  $C \vee \neg A$ ,  
or nothing is selected in  $C \vee \neg A$  and  $\neg A \succeq L$  for all  $L$  in  $C$ .

# Resolution Calculus $Res_{sel}^>$

---

(Ground) Ordered Factorization with Selection:

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

if the following conditions are satisfied:

- (i)  $A \succeq L$  for all  $L$  in  $C$ ;
- (ii) nothing is selected in  $C \vee A \vee A$  by  $sel$ .

## Resolution Calculus $Res_{sel}^{\succ}$

---

The extension from ground inferences to non-ground inferences is analogous to ordered resolution (replace  $\succ$  by  $\preceq$  and  $\succeq$  by  $\succ$ ). Again we assume that  $\succ$  is stable under substitutions.



# Resolution Calculus $Res_{sel}^>$

---

Ordered Resolution with Selection:

$$\frac{D \vee B \quad C \vee \neg A}{(D \vee C)\sigma}$$

if the following conditions are satisfied:

- (i)  $\sigma = \text{mgu}(A, B)$ ;
- (ii)  $B\sigma \not\leq L\sigma$  for all  $L$  in  $D$ ;
- (iii) nothing is selected in  $D \vee B$  by  $\text{sel}$ ;
- (iv)  $\neg A$  is selected in  $C \vee \neg A$ ,  
or nothing is selected in  $C \vee \neg A$  and  $\neg A\sigma \not\leq L\sigma$  for all  $L$  in  $C$ .

# Resolution Calculus $Res_{sel}^>$

---

Ordered Factorization with Selection:

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

if the following conditions are satisfied:

- (i)  $\sigma = \text{mgu}(A, B)$ ;
- (ii)  $A\sigma \not\prec L\sigma$  for all  $L$  in  $C$ .
- (iii) nothing is selected in  $C \vee A \vee B$  by  $sel$ .

# Lifting Lemma for $Res_{sel}^\succ$

---

Lemma 3.39:

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

$$\frac{\begin{array}{c} D \\ \downarrow \theta_1 \\ D\theta_1 \end{array} \quad \begin{array}{c} C \\ \downarrow \theta_2 \\ C\theta_2 \end{array}}{C'} \quad [\text{ground inference in } Res_{sel}^\succ]$$

and if  $\text{sel}(D\theta_1) \simeq \text{sel}(D)$ ,  $\text{sel}(C\theta_2) \simeq \text{sel}(C)$  (that is, “corresponding” literals are selected), then there exists a substitution  $\rho$  such that

$$\frac{\begin{array}{c} D \quad C \\ \hline C'' \end{array}}{\downarrow \rho} \quad [\text{inference in } Res_{sel}^\succ]$$

$$C' = C''\rho$$

## Lifting Lemma for $Res_{sel}^{\succ}$

---

An analogous lifting lemma holds for factorization.

## Saturation of Sets of General Clauses

---

Corollary 3.40:

Let  $N$  be a set of general clauses saturated under  $Res_{sel}^>$ , i. e.,  $Res_{sel}^>(N) \subseteq N$ .

Then there exists a selection function  $sel'$  such that  $sel|_N = sel'|_N$  and  $G_\Sigma(N)$  is also saturated, i. e.,

$$Res_{sel'}^>(G_\Sigma(N)) \subseteq G_\Sigma(N).$$

# Soundness and Refutational Completeness

---

Theorem 3.41:

Let  $\succ$  be an atom ordering and  $\text{sel}$  a selection function such that  $\text{Res}_{\text{sel}}^{\succ}(N) \subseteq N$ . Then  $N \models \perp \Leftrightarrow \perp \in N$

Proof:

( $\Leftarrow$ ): trivial.

( $\Rightarrow$ ): Consider first the propositional level:

Construct a candidate interpretation  $I_N$  as for unrestricted resolution, except that clauses  $C$  in  $N$  that have selected literals are never productive (even if they are false in  $I_C$  and if their maximal atom occurs only once and is positive).

The result for general clauses follows using Corollary 3.40. □

# What Do We Gain?

---

Search spaces become smaller:

- |   |                              |          |
|---|------------------------------|----------|
| 1 | $P \vee Q$                   |          |
| 2 | $P \vee \boxed{\neg Q}$      |          |
| 3 | $\neg P \vee Q$              |          |
| 4 | $\neg P \vee \boxed{\neg Q}$ |          |
| 5 | $Q \vee Q$                   | Res 1, 3 |
| 6 | $Q$                          | Fact 5   |
| 7 | $\neg P$                     | Res 6, 4 |
| 8 | $P$                          | Res 6, 2 |
| 9 | $\perp$                      | Res 8, 7 |

we assume  $P \succ Q$  and sel as indicated by  $\boxed{X}$ . The maximal literal in a clause is depicted in red.

In this example, the ordering and selection function even ensure that the refutation proceeds strictly deterministically.

## What Do We Gain?

---

Rotation redundancy can be avoided:

From

$$\frac{\frac{C_1 \vee A \quad C_2 \vee \neg A \vee B}{C_1 \vee C_2 \vee B} \quad C_3 \vee \neg B}{C_1 \vee C_2 \vee C_3}$$

we can obtain by **rotation**

$$\frac{C_1 \vee A \quad \frac{C_2 \vee \neg A \vee B \quad C_3 \vee \neg B}{C_2 \vee \neg A \vee C_3}}{C_1 \vee C_2 \vee C_3}$$

another proof of the same clause. In large proofs many rotations are possible. However, if  $A \succ B$ , then the second proof does not fulfill the ordering restrictions.



# Craig-Interpolation

---

Theorem 3.42 (Craig 1957):

Let  $F$  and  $G$  be two propositional formulas such that  $F \models G$ . Then there exists a formula  $H$  (called the **interpolant** for  $F \models G$ ), such that  $H$  contains only propositional variables occurring both in  $F$  and in  $G$ , and such that  $F \models H$  and  $H \models G$ .

The theorem also holds for first-order formulas, but in the general case, a proof based on resolution technology is complicated because of Skolemization.

## 3.14 Redundancy

---

So far: local restrictions of the resolution inference rules using orderings and selection functions.

Is it also possible to delete clauses altogether?

Under which circumstances are clauses unnecessary?

(e. g., if they are tautologies)

Intuition: If a clause is guaranteed to be neither a minimal counterexample nor productive, then we do not need it.

## A Formal Notion of Redundancy

---

Let  $N$  be a set of ground clauses and  $C$  a ground clause (not necessarily in  $N$ ).  $C$  is called **redundant** w. r. t.  $N$ , if there exist  $C_1, \dots, C_n \in N$ ,  $n \geq 0$ , such that  $C_j \prec C$  and  $C_1, \dots, C_n \models C$ .

Redundancy for general clauses:

$C$  is called **redundant** w. r. t.  $N$ , if all ground instances  $C\sigma$  of  $C$  are redundant w. r. t.  $G_\Sigma(N)$ .

Intuition: If a ground clause  $C$  is redundant and all clauses smaller than  $C$  hold in  $I_C$ , then  $C$  holds in  $I_C$   
(so  $C$  is neither a minimal counterexample nor productive).

# A Formal Notion of Redundancy

---

Note: The same ordering  $\succ$  is used for ordering restrictions and for redundancy (and for the completeness proof).

## Examples of Redundancy

---

In general, redundancy is undecidable. Decidable approximations are sufficient for us, however.

Proposition 3.43:

Some redundancy criteria:

- $C$  tautology (i. e.,  $\models C$ )  $\Rightarrow$   $C$  redundant w. r. t. any set  $N$ .
- $C\sigma \subset D \Rightarrow D$  redundant w. r. t.  $N \cup \{C\}$ .

(Under certain conditions one may also use non-strict subsumption, but this requires a slightly more complicated definition of redundancy.)

## Saturation up to Redundancy

---

$N$  is called **saturated up to redundancy** (w. r. t.  $Res_{sel}^>$ ) if

$$Res_{sel}^>(N \setminus Red(N)) \subseteq N \cup Red(N)$$

Theorem 3.44:

Let  $N$  be saturated up to redundancy. Then

$$N \models \perp \Leftrightarrow \perp \in N$$

# Monotonicity Properties of Redundancy

---

When we want to delete redundant clauses during a derivation, we have to ensure that redundant clauses *remain redundant* in the rest of the derivation.

Theorem 3.45:

$$(i) \quad N \subseteq M \Rightarrow Red(N) \subseteq Red(M)$$

$$(ii) \quad M \subseteq Red(N) \Rightarrow Red(N) \subseteq Red(N \setminus M)$$

Recall that  $Red(N)$  may include clauses that are not in  $N$ .

# Computing Saturated Sets

---

Redundancy is preserved when, during a theorem proving derivation one adds new clauses or deletes redundant clauses. This motivates the following definitions:

A **run** of the resolution calculus is a sequence

$N_0 \vdash N_1 \vdash N_2 \vdash \dots$ , such that

(i)  $N_i \models N_{i+1}$ , and

(ii) all clauses in  $N_i \setminus N_{i+1}$  are redundant w. r. t.  $N_{i+1}$ .

In other words, during a run we may add a new clause if it follows from the old ones, and we may delete a clause, if it is redundant w. r. t. the remaining ones.



# Computing Saturated Sets

---

For a run, we define  $N_\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} N_j$ .

The set  $N_\infty$  of all **persistent** clauses is called the **limit** of the run.

*This notation differs significantly from the 2021/2022 lecture. Keep that in mind when you use online lecture recordings or read exercises or exam questions from previous years.*

# Computing Saturated Sets

---

Lemma 3.46:

Let  $N_0 \vdash N_1 \vdash N_2 \vdash \dots$  be a run.

Then  $Red(N_i) \subseteq Red(\bigcup_{i \geq 0} N_i)$  and  $Red(N_i) \subseteq Red(N_\infty)$  for every  $i$ .

Proof:

Exercise. □

# Computing Saturated Sets

---

Corollary 3.47:

$N_i \subseteq N_\infty \cup \text{Red}(N_\infty)$  for every  $i$ .

Proof:

If  $C \in N_i \setminus N_\infty$ , then there is a  $k \geq i$  such that  $C \in N_k \setminus N_{k+1}$ , so  $C$  must be redundant w. r. t.  $N_{k+1}$ .

Consequently,  $C$  is redundant w. r. t.  $N_\infty$ . □

## Computing Saturated Sets

---

Even if a set  $N$  is inconsistent, it could happen that  $\perp$  is never derived, because some required inference is never computed.

The following definition rules out such runs:

A run is called **fair**, if the conclusion of every inference from clauses in  $N_\infty \setminus Red(N_\infty)$  is contained in some  $N_i \cup Red(N_i)$ .

Lemma 3.48:

If a run is fair, then its limit is saturated up to redundancy.

# Computing Saturated Sets

---

Theorem 3.49 (Refutational Completeness: Dynamic View):

Let  $N_0 \vdash N_1 \vdash N_2 \vdash \dots$  be a fair run, let  $N_\infty$  be its limit.

Then  $N_0$  has a model if and only if  $\perp \notin N_\infty$ .

Proof:

( $\Leftarrow$ ): By fairness,  $N_\infty$  is saturated up to redundancy.

If  $\perp \notin N_\infty$ , then it has an Herbrand model.

Since every clause in  $N_0$  is contained in  $N_\infty$  or redundant w. r. t.  $N_\infty$ , this model is also a model of  $G_\Sigma(N_0)$  and therefore a model of  $N_0$ .

( $\Rightarrow$ ): Obvious, since  $N_0 \models N_\infty$ . □

# Simplifications

---

In theory, the definition of a run permits to add arbitrary clauses that are entailed by the current ones.

# Simplifications

---

In practice, we restrict to two cases:

- We add conclusions of  $Res_{sel}^>$ -inferences from non-redundant premises.  
 $\rightsquigarrow$  necessary to guarantee fairness
- We add clauses that are entailed by the current ones if this *makes* other clauses redundant:

$$N \cup \{C\} \vdash N \cup \{C, D\} \vdash N \cup \{D\}$$

$$\text{if } N \cup \{C\} \models D \text{ and } C \in Red(N \cup \{D\}).$$

Net effect:  $C$  is *simplified* to  $D$

$\rightsquigarrow$  useful to get easier/smaller clause sets

# Simplifications

---

Notation for simplification rules:

$$\frac{C_1 \dots C_n}{D_1 \dots D_m}$$

means

$$N \cup \{C_1, \dots, C_n\} \vdash N \cup \{D_1, \dots, D_m\}$$



# Simplifications

---

Examples of simplification techniques:

- Deletion of duplicated literals:

$$\frac{C \vee L \vee L}{C \vee L}$$

- Subsumption resolution:

$$\frac{D \vee L \quad C \vee D\sigma \vee \bar{L}\sigma}{D \vee L \quad C \vee D\sigma}$$

## 3.15 Hyperresolution

---

There are *many* variants of resolution.

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

Assume that several negative literals are selected in a clause  $C$ .

If we perform an inference with  $C$ , then one of the selected literals is eliminated.

Suppose that the remaining selected literals of  $C$  are again selected in the conclusion. Then we must eliminate the remaining selected literals one by one by further resolution steps.

# Hyperresolution

---

Hyperresolution replaces these successive steps by a single inference.

As for  $Res_{sel}^{\succ}$ , the calculus is parameterized by an atom ordering  $\succ$  and a selection function  $sel$ .

# Hyperresolution

---

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

with  $\sigma = \text{mgu}(A_1 \doteq B_1, \dots, A_n \doteq B_n)$ , if

- (i)  $B_i\sigma$  strictly maximal in  $D_i\sigma$ ,  $1 \leq i \leq n$ ;
- (ii) nothing is selected in  $D_i$ ;
- (iii) the indicated occurrences of the  $\neg A_i$  are exactly the ones selected by sel, or nothing is selected in the right premise and  $n = 1$  and  $\neg A_1\sigma$  is maximal in  $C\sigma$ .

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

# Hyperresolution

---

As we have seen, hyperresolution can be simulated by iterated binary resolution.

However this yields intermediate clauses which HR might not derive, and many of them might not be extendable into a full HR inference.

## 3.16 Implementing Resolution: The Main Loop

---

Standard approach:

Select one clause (“Given clause”).

Find many partner clauses that can be used in inferences together with the “given clause” using an appropriate index data structure.

Compute the conclusions of these inferences; add them to the set of clauses.

# Implementing Resolution: The Main Loop

---

The set of clauses is split into two subsets:

- $WO$  = “Worked-off” (or “active”) clauses:  
Have already been selected as “given clause”.
- $U$  = “Usable” (or “passive”) clauses:  
Have not yet been selected as “given clause”.

# Implementing Resolution: The Main Loop

---

During each iteration of the main loop:

Select a new given clause  $C$  from  $U$ ;

$U := U \setminus \{C\}$ .

Find partner clauses  $D_i$  from  $WO$ ;

$New :=$  Conclusions of inferences from  $\{D_i \mid i \in I\} \cup C$

where one premise is  $C$ ;

$U := U \cup New$ ;

$WO := WO \cup \{C\}$

$\Rightarrow$  At any time, all inferences between clauses in  $WO$  have been computed.

$\Rightarrow$  The procedure is fair, if no clause remains in  $U$  forever.



# Implementing Resolution: The Main Loop

---

Additionally:

Try to simplify  $C$  using  $WO$ .

(Skip the remainder of the iteration, if  $C$  can be eliminated.)

Try to simplify (or even eliminate) clauses from  $WO$  using  $C$ .

# Implementing Resolution: The Main Loop

---

Design decision: should one also simplify  $U$  using  $C$ ?

yes  $\rightsquigarrow$  “Otter loop”:

Advantage: simplifications of  $U$  may be useful to derive the empty clause.

no  $\rightsquigarrow$  “Discount loop”:

Advantage: clauses in  $U$  are really passive;

only clauses in  $WO$  have to be kept in index data structure.

(Hence: can use index data structure for which retrieval is faster, even if update is slower and space consumption is higher.)

## 3.17 Summary: Resolution Theorem Proving

---

- Resolution is a machine-oriented calculus.
- Using unification, the enumeration of instances becomes a by-product of inference computation.
- Parameters: atom ordering  $\succ$  and selection function  $\text{sel}$ .  
On the non-ground level, ordering constraints can (only) be solved approximatively.
- Completeness proof by constructing candidate interpretations from productive clauses  $C \vee A, A \succ C$ .

# Summary: Resolution Theorem Proving

---

- *Local* restrictions of inferences via  $\succ$  and sel  
⇒ fewer proof variants.
- *Global* restrictions of the search space via redundancy  
⇒ computing with “smaller” / “easier” clause sets.  
(In practice: simplification and detection of redundant clauses uses 90% of the prover runtime.)
- Termination on many decidable fragments.
- However: not good enough for dealing with orderings, equality and more specific algebraic theories (lattices, abelian groups, rings, fields)  
⇒ further specialization of inference systems required.

## 3.18 Semantic Tableaux

---

Literature:

M. Fitting: *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, New York, 1996, chapters 3, 6, 7.

R. M. Smullyan: *First-Order Logic*, Dover Publ., New York, 1968, revised 1995.

Like resolution, semantic tableaux were developed in the sixties, independently by Zbigniew Lis and Raymond Smullyan on the basis of work by Gentzen in the 30s and of Beth in the 50s.

# Idea

---

Idea (for the propositional case):

A set  $\{F \wedge G\} \cup N$  of formulas has a model if and only if  $\{F \wedge G, F, G\} \cup N$  has a model.

A set  $\{F \vee G\} \cup N$  of formulas has a model if and only if  $\{F \vee G, F\} \cup N$  or  $\{F \vee G, G\} \cup N$  has a model.

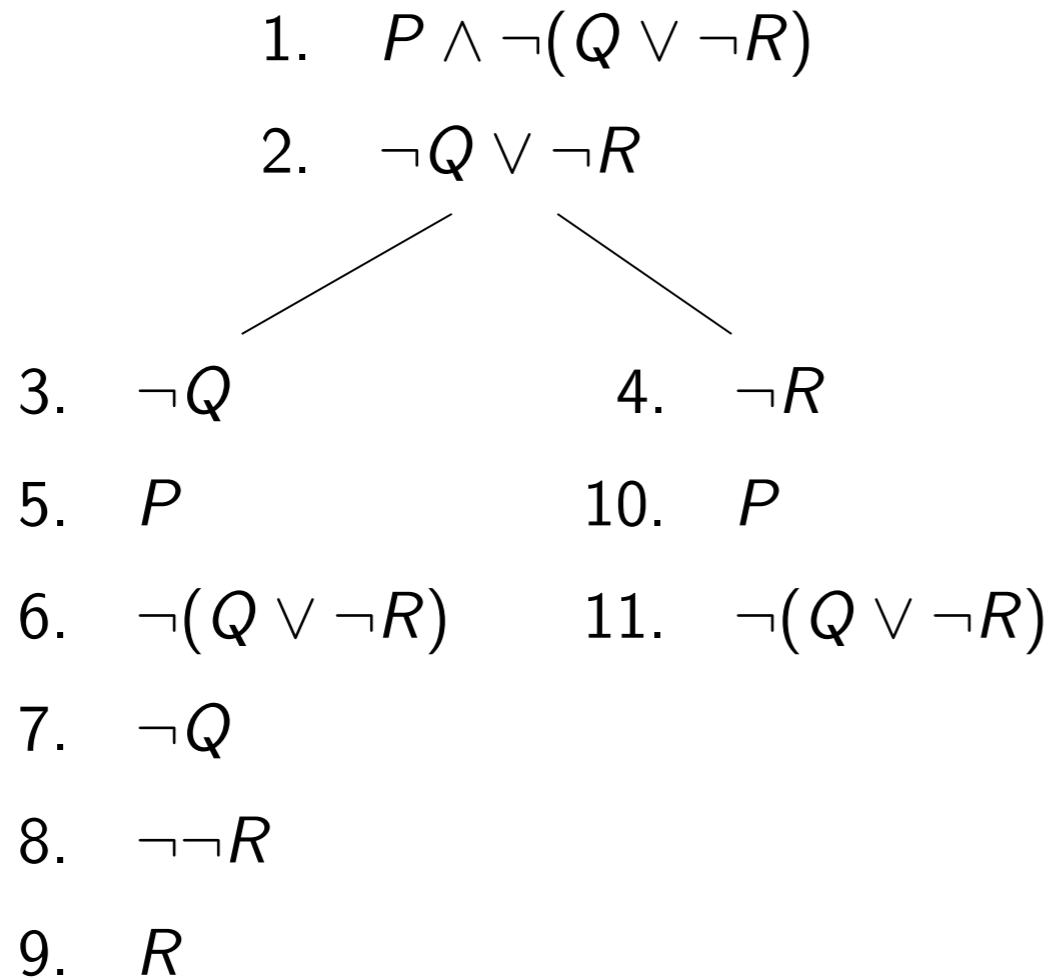
(and similarly for other connectives).

To avoid duplication, represent sets as paths of a tree.

Continue splitting until two complementary formulas are found  $\Rightarrow$  inconsistency detected.

## A Tableau for $\{P \wedge \neg(Q \vee \neg R), \neg Q \vee \neg R\}$

---



This tableau is not “maximal”, however the first “path” is. This path is not “closed”, hence the set  $\{1, 2\}$  is satisfiable. (These notions will all be defined below.)

# Properties

---

Properties of tableau calculi:

analytic: inferences correspond closely to the logical meaning of the symbols.

goal oriented: inferences operate directly on the goal to be proved.

global: some inferences affect the entire proof state (set of formulas), as we will see later.



# Propositional Expansion Rules

---

Expansion rules are applied to the formulas in a tableau and expand the tableau at a leaf. We append the conclusions of a rule (horizontally or vertically) at a *leaf*, whenever the premise of the expansion rule matches a formula appearing *anywhere* on the path from the root to that leaf.

## Negation Elimination

$$\frac{\neg\neg F}{F}$$

$$\frac{\neg T}{\perp}$$

$$\frac{\neg\perp}{T}$$

# Propositional Expansion Rules

---

## $\alpha$ -Expansion

(for formulas that are essentially conjunctions: append subformulas  $\alpha_1$  and  $\alpha_2$  one on top of the other)

$$\frac{\alpha}{\alpha_1 \alpha_2}$$

## $\beta$ -Expansion

(for formulas that are essentially disjunctions: append  $\beta_1$  and  $\beta_2$  horizontally, i. e., branch into  $\beta_1$  and  $\beta_2$ )

$$\frac{\beta}{\beta_1 \mid \beta_2}$$

# Classification of Formulas

---

conjunctive			disjunctive		
$\alpha$	$\alpha_1$	$\alpha_2$	$\beta$	$\beta_1$	$\beta_2$
$F \wedge G$	$F$	$G$	$\neg(F \wedge G)$	$\neg F$	$\neg G$
$\neg(F \vee G)$	$\neg F$	$\neg G$	$F \vee G$	$F$	$G$
$\neg(F \rightarrow G)$	$F$	$\neg G$	$F \rightarrow G$	$\neg F$	$G$

We assume that the binary connective  $\leftrightarrow$  has been eliminated in advance.

## Tableaux: Notions

---

A **semantic tableau** is a marked (by formulas), finite, unordered tree and inductively defined as follows: Let  $\{F_1, \dots, F_n\}$  be a set of formulas.

(i) The tree consisting of a single path

$$\begin{array}{c} F_1 \\ \vdots \\ F_n \end{array}$$

is a tableau for  $\{F_1, \dots, F_n\}$ .

(We do not draw edges if nodes have only one successor.)

## Tableaux: Notions

---

- (ii) If  $T$  is a tableau for  $\{F_1, \dots, F_n\}$  and if  $T'$  results from  $T$  by applying an expansion rule then  $T'$  is also a tableau for  $\{F_1, \dots, F_n\}$ .

Note: We may also consider the *limit tableau* of a tableau expansion; this can be an *infinite* tree.

## Tableaux: Notions

---

A **path** (from the root to a leaf) in a tableau is called **closed**, if it either contains  $\perp$ , or else it contains both some formula  $F$  and its negation  $\neg F$ . Otherwise the path is called **open**.

A tableau is called **closed**, if all paths are closed.

A **tableau proof** for  $F$  is a closed tableau for  $\{\neg F\}$ .

## Tableaux: Notions

---

A path  $\pi$  in a tableau is called **maximal**, if for each formula  $F$  on  $\pi$  that is neither a literal nor  $\perp$  nor  $\top$  there exists a node in  $\pi$  at which the expansion rule for  $F$  has been applied.

In that case, if  $F$  is a formula on  $\pi$ ,  $\pi$  also contains:

- (i)  $\alpha_1$  and  $\alpha_2$ , if  $F$  is a  $\alpha$ -formula,
- (ii)  $\beta_1$  or  $\beta_2$ , if  $F$  is a  $\beta$ -formula, and
- (iii)  $F'$ , if  $F$  is a negation formula, and  $F'$  the conclusion of the corresponding elimination rule.

A tableau is called **maximal**, if each path is closed or maximal.

## Tableaux: Notions

---

A tableau is called **strict**, if for each formula the corresponding expansion rule has been applied at most once on each path containing that formula.

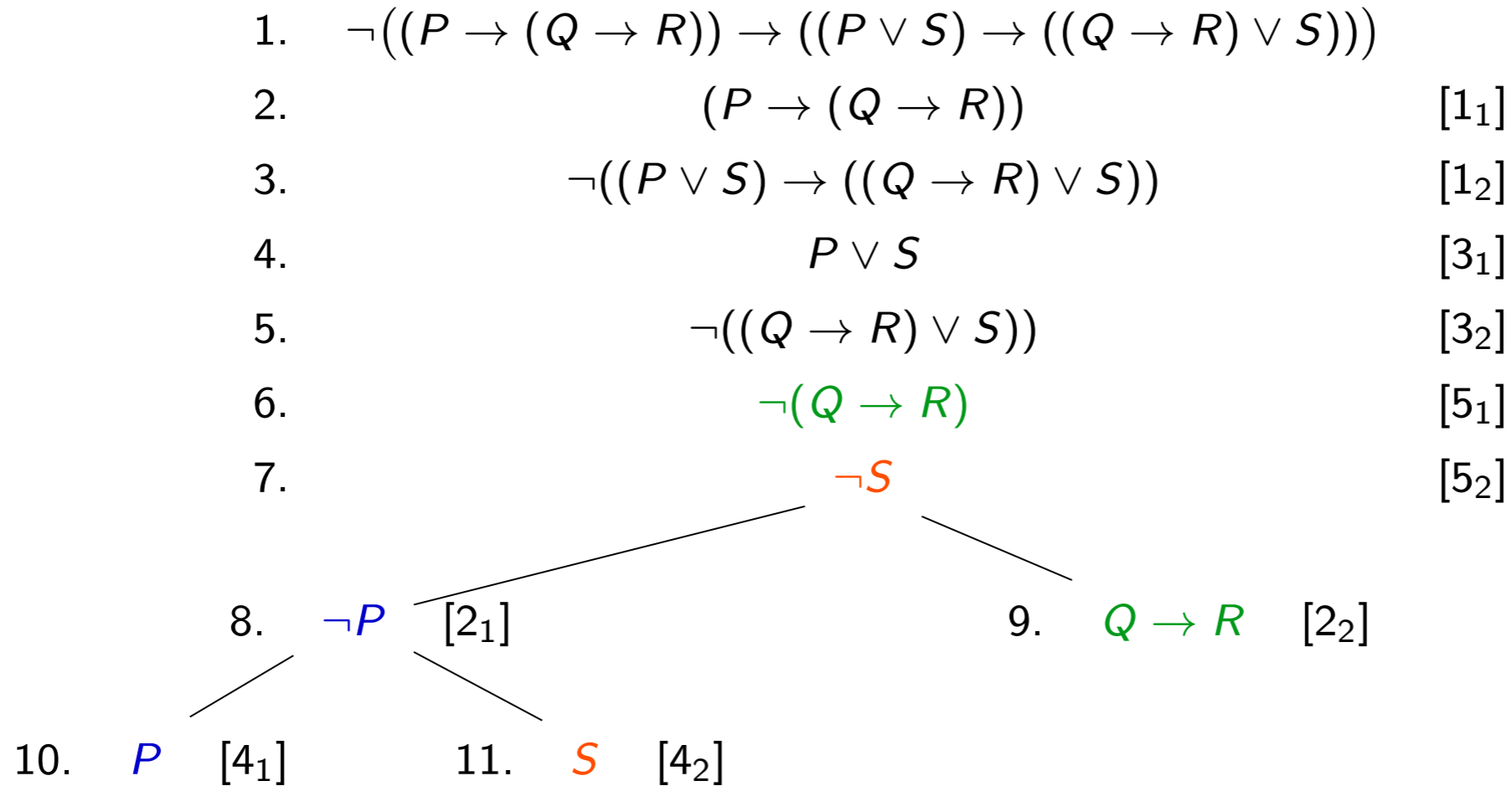
A tableau is called **clausal**, if each of its formulas is a clause.



# A Sample Proof

---

One starts out from the negation of the formula to be proved.



There are three paths, each of them closed.

# Properties of Propositional Tableaux

---

We assume that  $T$  is a tableau for  $\{F_1, \dots, F_n\}$ .

Theorem 3.50:

$\{F_1, \dots, F_n\}$  satisfiable  $\Leftrightarrow$  some path (i. e., the set of its formulas) in  $T$  is satisfiable.

Proof:

( $\Leftarrow$ ) Trivial, since every path contains in particular  $F_1, \dots, F_n$ .

( $\Rightarrow$ ) By induction over the structure of  $T$ . □

Corollary 3.51:

$T$  closed  $\Rightarrow \{F_1, \dots, F_n\}$  unsatisfiable

# Properties of Propositional Tableaux

---

Theorem 3.52:

Every strict propositional tableau expansion is finite.

Proof:

New formulas resulting from expansion are either  $\perp$ ,  $\top$  or subformulas of the expanded formula (modulo de Morgan's law), so the number of formulas that can occur is finite. By strictness, on each path a formula can be expanded at most once. Therefore, each path is finite, and a finitely branching tree with finite paths is finite by Lemma 1.9.  $\square$

Conclusion: Strict and maximal tableaux can be effectively constructed.

# Refutational Completeness

---

A set  $\mathcal{H}$  of propositional formulas is called a Hintikka set, if

- (1) there is no  $P \in \Pi$  with  $P \in \mathcal{H}$  and  $\neg P \in \mathcal{H}$ ;
- (2)  $\perp \notin \mathcal{H}$ ,  $\neg\top \notin \mathcal{H}$ ;
- (3) if  $\neg\neg F \in \mathcal{H}$ , then  $F \in \mathcal{H}$ ;
- (4) if  $\alpha \in \mathcal{H}$ , then  $\alpha_1 \in \mathcal{H}$  and  $\alpha_2 \in \mathcal{H}$ ;
- (5) if  $\beta \in \mathcal{H}$ , then  $\beta_1 \in \mathcal{H}$  or  $\beta_2 \in \mathcal{H}$ .

# Refutational Completeness

---

Lemma 3.53 (Hintikka's Lemma):

Every Hintikka set is satisfiable.

Proof:

Let  $\mathcal{H}$  be a Hintikka set. Define a valuation  $\mathcal{A}$  by  $\mathcal{A}(P) = 1$  if  $P \in \mathcal{H}$  and  $\mathcal{A}(P) = 0$  otherwise. Then show that  $\mathcal{A}(F) = 1$  for all  $F \in \mathcal{H}$  by induction over the size of formulas. □

# Refutational Completeness

---

Theorem 3.54:

Let  $\pi$  be a maximal open path in a tableau. Then the set of formulas on  $\pi$  is satisfiable.

Proof:

We show that set of formulas on  $\pi$  is a Hintikka set: Conditions (3), (4), (5) follow from the fact that  $\pi$  is maximal; conditions (1) and (2) follow from the fact that  $\pi$  is open and from maximality for the second negation elimination rule. □

Note: The theorem holds also for infinite trees that are obtained as the limit of a tableau expansion.

# Refutational Completeness

---

Theorem 3.55:

$\{F_1, \dots, F_n\}$  satisfiable  $\Leftrightarrow$  there exists no closed strict tableau for  $\{F_1, \dots, F_n\}$ .

Proof:

( $\Rightarrow$ ) Clear by Cor. 3.51.

( $\Leftarrow$ ) Let  $T$  be a strict maximal tableau for  $\{F_1, \dots, F_n\}$  and let  $\pi$  be an open path in  $T$ . By the previous theorem, the set of formulas on  $\pi$  is satisfiable, and hence by Theorem 3.50 the set  $\{F_1, \dots, F_n\}$ , is satisfiable.

□

# Consequences

---

The validity of a propositional formula  $F$  can be established by constructing a strict maximal tableau for  $\{\neg F\}$ :

- $T$  closed  $\Leftrightarrow F$  valid.
- It suffices to test complementarity of paths w. r. t. atomic formulas (cf. reasoning in the proof of Theorem 3.54).
- Which of the potentially many strict maximal tableaux one computes does not matter. In other words, tableau expansion rules can be applied don't-care non-deterministically (“**proof confluence**”).
- The expansion strategy, however, can have a dramatic impact on the tableau size.



## A Variant of the $\beta$ -Rule

---

Since  $F \vee G \models F \vee (G \wedge \neg F)$ , the  $\beta$  expansion rule

$$\frac{\beta}{\beta_1 \mid \beta_2}$$

can be replaced by the following variant:

$$\frac{\beta}{\beta_1 \mid \begin{array}{l} \beta_2 \\ \neg\beta_1 \end{array}}$$

## A Variant of the $\beta$ -Rule

---

The variant  $\beta$ -rule can lead to much shorter proofs, but it is not always beneficial.

In general, it is most helpful if  $\neg\beta_1$  can be at most (iteratively)  $\alpha$ -expanded.

## 3.19 Semantic Tableaux for First-Order Logic

---

There are two ways to extend the tableau calculus to quantified formulas:

- using ground instantiation,
- using free variables.

# Tableaux with Ground Instantiation

---

Classification of quantified formulas:

universal		existential	
$\gamma$	$\gamma(t)$	$\delta$	$\delta(t)$
$\forall xF$	$F\{x \mapsto t\}$	$\exists xF$	$F\{x \mapsto t\}$
$\neg\exists xF$	$\neg F\{x \mapsto t\}$	$\neg\forall xF$	$\neg F\{x \mapsto t\}$

# Tableaux with Ground Instantiation

---

Idea:

Replace universally quantified formulas by appropriate ground instances.

**$\gamma$ -expansion**

$$\frac{\gamma}{\gamma(t)} \quad \text{where } t \text{ is some ground term}$$

**$\delta$ -expansion**

$$\frac{\delta}{\delta(c)} \quad \text{where } c \text{ is a new Skolem constant}$$

## Tableaux with Ground Instantiation

---

Skolemization becomes part of the calculus and needs not necessarily be applied in a preprocessing step. Of course, one could do Skolemization beforehand, and then the  $\delta$ -rule would not be needed.

Note:

Skolem *constants* are sufficient:

In a  $\delta$ -formula  $\exists x F$ ,  $\exists$  is the outermost quantifier and  $x$  is the only free variable in  $F$ .

# Tableaux with Ground Instantiation

---

Problems:

Having to guess ground terms is impractical.

Even worse, we may have to guess *several* ground instances, as strictness for  $\gamma$  is incomplete. For instance, constructing a closed tableau for

$$\{\forall x (P(x) \rightarrow P(f(x))), P(b), \neg P(f(f(b)))\}$$

is impossible without applying  $\gamma$ -expansion twice on one path.

# Free-Variable Tableaux

---

An alternative approach:

Delay the instantiation of universally quantified variables.

Replace universally quantified variables by new free variables.

Intuitively, the free variables are universally quantified *outside* of the entire tableau.



# Free-Variable Tableaux

---

$\gamma$ -expansion

$$\frac{\gamma}{\gamma(x)} \quad \text{where } x \text{ is a new free variable}$$

$\delta$ -expansion

$$\frac{\delta}{\delta(f(x_1, \dots, x_n))}$$

where  $f$  is a new Skolem function, and the  $x_i$  are the free variables in  $\delta$

## Free-Variable Tableaux

---

Application of expansion rules has to be supplemented by a **substitution rule**:

- (iii) If  $T$  is a tableau for  $\{F_1, \dots, F_n\}$  and if  $\sigma$  is a substitution, then  $T\sigma$  is also a tableau for  $\{F_1, \dots, F_n\}$ .

The substitution rule may, potentially, modify all the formulas of a tableau. This feature is what makes the tableau method a *global proof method*. (Resolution, by comparison, is a local method.)

## Free-Variable Tableaux

---

One can show that it is sufficient to consider substitutions  $\sigma$  for which there is a path in  $T$  containing two *literals*  $\neg A$  and  $B$  such that  $\sigma = \text{mgu}(A, B)$ . Such tableaux are called **AMGU-Tableaux**.

## Example

---

1.  $\neg(\exists w \forall x P(x, w, f(x, w))) \rightarrow \exists w \forall x \exists y P(x, w, y)$
2.  $\exists w \forall x P(x, w, f(x, w))$  1<sub>1</sub> [ $\alpha$ ]
3.  $\neg \exists w \forall x \exists y P(x, w, y)$  1<sub>2</sub> [ $\alpha$ ]
4.  $\forall x P(x, c, f(x, c))$  2( $c$ ) [ $\delta$ ]
5.  $\neg \forall x \exists y P(x, v_1, y)$  3( $v_1$ ) [ $\gamma$ ]
6.  $\neg \exists y P(b(v_1), v_1, y)$  5( $b(v_1)$ ) [ $\delta$ ]
7.  $P(v_2, c, f(v_2, c))$  4( $v_2$ ) [ $\gamma$ ]
8.  $\neg P(b(v_1), v_1, v_3)$  6( $v_3$ ) [ $\gamma$ ]

7. and 8. are complementary (modulo unification):

$$\{v_2 \doteq b(v_1), c \doteq v_1, f(v_2, c) \doteq v_3\}$$

is solvable with an mgu  $\sigma = \{v_1 \mapsto c, v_2 \mapsto b(c), v_3 \mapsto f(b(c), c)\}$ ,  
and hence,  $T\sigma$  is a closed (linear) tableau for the formula in 1.

## Example

---

Problem:

Strictness for  $\gamma$  is still incomplete.

For instance, constructing a closed tableau for

$$\{\forall x (P(x) \rightarrow P(f(x))), P(b), \neg P(f(f(b)))\}$$

is impossible without applying  $\gamma$ -expansion twice on one path.

# Semantic Tableaux vs. Resolution

---

- Tableaux: global, goal-oriented, “backward” .
- Resolution: local, “forward” .
- Goal-orientation is a clear advantage if only a small subset of a large set of formulas is necessary for a proof.  
(Note that resolution provers saturate also those parts of the clause set that are irrelevant for proving the goal.)

# Semantic Tableaux vs. Resolution

---

- Resolution can be combined with more powerful redundancy elimination methods; because of its global nature this is more difficult for the tableau method.
- Resolution can be refined to work well with equality; for tableaux this seems to be impossible.
- On the other hand tableau calculi can be easily extended to other logics; in particular tableau provers are very successful in modal and description logics.

## 3.20 Other Deductive Systems

---

- Instantiation-based methods
  - Resolution-based instance generation
  - Disconnection calculus
  - ...
- Natural deduction
- Sequent calculus/Gentzen calculus
- Hilbert calculus



# Instantiation-Based Methods for FOL

---

Idea:

Overlaps of complementary literals produce instantiations  
(as in resolution);

However, contrary to resolution, clauses are not recombined.

Instead: treat remaining variables as constant and use efficient propositional proof methods, such as CDCL.

# Instantiation-Based Methods for FOL

---

There are both saturation-based variants, such as partial instantiation (Hooker et al. 2002) or resolution-based instance generation (Inst-Gen) (Ganzinger and Korovin 2003), and tableau-style variants, such as the disconnection calculus (Billon 1996; Letz and Stenz 2001).

Successful in practice for problems that are “almost propositional” (i. e., no non-constant function symbols, no equality).

# Natural Deduction

---

Idea:

Model the concept of proofs from assumptions as humans do it.

To prove  $F \rightarrow G$ , assume  $F$  and try to derive  $G$ .

Initial ideas: Jaśkowski (1934), Gentzen (1934); extended by Prawitz (1965).

Popular in interactive proof systems.

# Sequent Calculus

---

Idea:

Assumptions internalized into the data structure of sequents

$$F_1, \dots, F_m \vdash G_1, \dots, G_k$$

meaning

$$F_1 \wedge \dots \wedge F_m \rightarrow G_1 \vee \dots \vee G_k$$

# Sequent Calculus

---

Inferences rules, e. g.:

$$\frac{\Gamma \vdash \Delta}{\Gamma, F \vdash \Delta} \quad (WL)$$

$$\frac{\Gamma, F \vdash \Delta \quad \Sigma, G \vdash \Pi}{\Gamma, \Sigma, F \vee G \vdash \Delta, \Pi} \quad (\vee L)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash F, \Delta} \quad (WR)$$

$$\frac{\Gamma \vdash F, \Delta \quad \Sigma \vdash G, \Pi}{\Gamma, \Sigma \vdash F \wedge G, \Delta, \Pi} \quad (\wedge R)$$

# Sequent Calculus

---

Initial idea: Gentzen 1934.

Perfect symmetry between the handling of assumptions and their consequences; interesting for proof theory.

Can be used both backwards and forwards.

Allows to simulate both natural deduction and semantic tableaux.

# Hilbert Calculus

---

Idea:

Direct proof method (proves a theorem from axioms, rather than refuting its negation)

Axiom schemes, e. g.,

$$F \rightarrow (G \rightarrow F)$$

$$(F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$$

plus Modus ponens:

$$\frac{F \quad F \rightarrow G}{G}$$

# Hilbert Calculus

---

Unsuitable for finding or reading proofs,  
but sometimes used for *specifying* (e. g., modal) logics.



## Part 4: First-Order Logic with Equality

---

Equality is the most important relation in mathematics and functional programming.

In principle, problems in first-order logic with equality can be handled by any prover for first-order logic without equality:

## 4.1 Handling Equality Naively

---

Proposition 4.1:

Let  $F$  be a closed first-order formula with equality. Let  $\sim \notin \Pi$  be a new predicate symbol. The set  $Eq(\Sigma)$  contains the formulas

$$\begin{aligned} & \forall x (x \sim x) \\ & \forall x, y (x \sim y \rightarrow y \sim x) \\ & \forall x, y, z (x \sim y \wedge y \sim z \rightarrow x \sim z) \\ & \forall \vec{x}, \vec{y} (x_1 \sim y_1 \wedge \dots \wedge x_n \sim y_n \rightarrow f(x_1, \dots, x_n) \sim f(y_1, \dots, y_n)) \\ & \forall \vec{x}, \vec{y} (x_1 \sim y_1 \wedge \dots \wedge x_m \sim y_m \wedge P(x_1, \dots, x_m) \rightarrow P(y_1, \dots, y_m)) \end{aligned}$$

for every  $f/n \in \Omega$  and  $P/m \in \Pi$ . Let  $\tilde{F}$  be the formula that one obtains from  $F$  if every occurrence of  $\approx$  is replaced by  $\sim$ . Then  $F$  is satisfiable if and only if  $Eq(\Sigma) \cup \{\tilde{F}\}$  is satisfiable.

## Handling Equality Naively

---

An analogous proposition holds for *sets* of closed first-order formulas with equality.

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

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

# Handling Equality Naively

---

Equality is theoretically difficult:

First-order functional programming is Turing-complete.

But: resolution theorem provers cannot even solve equational problems that are intuitively easy.

Consequence: to handle equality efficiently, knowledge must be integrated into the theorem prover.

# Roadmap

---

How to proceed:

- This semester: Equations (unit clauses with equality).

Term rewrite systems.

Expressing semantic consequence syntactically.

Knuth-Bendix-Completion.

Entailment for equations.

- Next semester: Equational clauses.

Combining resolution and KB-completion.

→ Superposition.

Entailment for clauses with equality.

## 4.2 Rewrite Systems

---

Let  $E$  be a set of (implicitly universally quantified) equations.

The **rewrite relation**  $\rightarrow_E \subseteq T_\Sigma(X) \times T_\Sigma(X)$  is defined by

$$\begin{aligned} s \rightarrow_E t \quad \text{if and only if} \quad & \text{there exist } (l \approx r) \in E, p \in \text{pos}(s), \\ & \text{and } \sigma : X \rightarrow T_\Sigma(X), \\ & \text{such that } s|_p = l\sigma \text{ and } t = s[r\sigma]_p. \end{aligned}$$

An instance of the lhs (left-hand side) of an equation is called a **redex** (reducible expression).

**Contracting** a redex means replacing it with the corresponding instance of the rhs (right-hand side) of the rule.

# Rewrite Systems

---

An equation  $l \approx r$  is also called a **rewrite rule**, if  $l$  is not a variable and  $\text{var}(l) \supseteq \text{var}(r)$ .

Notation:  $l \rightarrow r$ .

A set of rewrite rules is called a **term rewrite system (TRS)**.

# Rewrite Systems

---

We say that a set of equations  $E$  or a TRS  $R$  is terminating, if the rewrite relation  $\rightarrow_E$  or  $\rightarrow_R$  has this property.

(Analogously for other properties of abstract reduction systems).

Note: If  $E$  is terminating, then it is a TRS.



# E-Algebras

---

Let  $E$  be a set of universally quantified equations.

A model of  $E$  is also called an  $E$ -algebra.

If  $E \models \forall \vec{x}(s \approx t)$ , i. e.,  $\forall \vec{x}(s \approx t)$  is valid in all  $E$ -algebras, we write this also as  $s \approx_E t$ .

Goal:

Use the rewrite relation  $\rightarrow_E$  to express the semantic consequence relation syntactically:

$$s \approx_E t \text{ if and only if } s \leftrightarrow_E^* t.$$

## E-Algebras

---

Let  $E$  be a set of equations over  $T_{\Sigma}(X)$ . The following inference system allows to derive consequences of  $E$ :

# E-Algebras

---

$$E \vdash t \approx t$$

(Reflexivity)

for every  $t \in T_{\Sigma}(X)$

$$\frac{E \vdash t \approx t'}{E \vdash t' \approx t}$$

(Symmetry)

$$\frac{E \vdash t \approx t' \quad E \vdash t' \approx t''}{E \vdash t \approx t''}$$

(Transitivity)

$$\frac{E \vdash t_1 \approx t'_1 \quad \dots \quad E \vdash t_n \approx t'_n}{E \vdash f(t_1, \dots, t_n) \approx f(t'_1, \dots, t'_n)}$$

(Congruence)

$$E \vdash t\sigma \approx t'\sigma$$

(Instance)

if  $(t \approx t') \in E$  and  $\sigma : X \rightarrow T_{\Sigma}(X)$

# E-Algebras

---

Lemma 4.2:

The following properties are equivalent:

(i)  $s \leftrightarrow_E^* t$

(ii)  $E \vdash s \approx t$  is derivable.

# E-Algebras

---

Constructing a **quotient algebra**:

Let  $X$  be a set of variables.

For  $t \in T_\Sigma(X)$  let  $[t] = \{ t' \in T_\Sigma(X) \mid E \vdash t \approx t' \}$  be the **congruence class** of  $t$ .

Define a  $\Sigma$ -algebra  $T_\Sigma(X)/E$  (abbreviated by  $\mathcal{T}$ ) as follows:

$$U_{\mathcal{T}} = \{ [t] \mid t \in T_\Sigma(X) \}.$$

$$f_{\mathcal{T}}([t_1], \dots, [t_n]) = [f(t_1, \dots, t_n)] \text{ for } f/n \in \Omega.$$

# E-Algebras

---

Lemma 4.3:

$f_{\mathcal{T}}$  is well-defined:

If  $[t_i] = [t'_i]$ , then  $[f(t_1, \dots, t_n)] = [f(t'_1, \dots, t'_n)]$ .

Lemma 4.4:

$\mathcal{T} = T_{\Sigma}(X)/E$  is an  $E$ -algebra.

Lemma 4.5:

Let  $X$  be a countably infinite set of variables; let  $s, t \in T_{\Sigma}(Y)$ .

If  $T_{\Sigma}(X)/E \models \forall \vec{x}(s \approx t)$ , then  $E \vdash s \approx t$  is derivable.

# E-Algebras

---

Theorem 4.6 (“Birkhoff’s Theorem”):

Let  $X$  be a countably infinite set of variables, let  $E$  be a set of (universally quantified) equations. Then the following properties are equivalent for all  $s, t \in T_{\Sigma}(X)$ :

(i)  $s \leftrightarrow_E^* t$ .

(ii)  $E \vdash s \approx t$  is derivable.

(iii)  $s \approx_E t$ , i. e.,  $E \models \forall \vec{x}(s \approx t)$ .

(iv)  $T_{\Sigma}(X)/E \models \forall \vec{x}(s \approx t)$ .

# Universal Algebra

---

$T_{\Sigma}(X)/E = T_{\Sigma}(X)/\approx_E = T_{\Sigma}(X)/\leftrightarrow_E^*$  is called the **free  $E$ -algebra** with generating set  $X/\approx_E = \{ [x] \mid x \in X \}$ :

Every mapping  $\varphi : X/\approx_E \rightarrow \mathcal{B}$  for some  $E$ -algebra  $\mathcal{B}$  can be extended to a homomorphism  $\hat{\varphi} : T_{\Sigma}(X)/E \rightarrow \mathcal{B}$ .

$T_{\Sigma}(\emptyset)/E = T_{\Sigma}(\emptyset)/\approx_E = T_{\Sigma}(\emptyset)/\leftrightarrow_E^*$  is called the **initial  $E$ -algebra**.



# Universal Algebra

---

$\approx_E = \{ (s, t) \mid E \models s \approx t \}$  is called the **equational theory** of  $E$ .

$\approx_E^I = \{ (s, t) \mid T_\Sigma(\emptyset)/E \models s \approx t \}$  is called the **inductive theory** of  $E$ .

Example:

Let  $E = \{ \forall x(x + 0 \approx x), \forall x \forall y(x + s(y) \approx s(x + y)) \}$ .

Then  $x + y \approx_E^I y + x$ , but  $x + y \not\approx_E y + x$ .

## 4.3 Confluence

---

Let  $(A, \rightarrow)$  be an abstract reduction system.

$b$  and  $c \in A$  are **joinable**, if there is a  $a$  such that  $b \rightarrow^* a \leftarrow^* c$ .

Notation:  $b \downarrow c$ .

The relation  $\rightarrow$  is called

**Church-Rosser**, if  $b \leftrightarrow^* c$  implies  $b \downarrow c$ .

**confluent**, if  $b \leftarrow^* a \rightarrow^* c$  implies  $b \downarrow c$ .

**locally confluent**, if  $b \leftarrow a \rightarrow c$  implies  $b \downarrow c$ .

**convergent**, if it is confluent and terminating.

# Confluence

---

Theorem 4.7:

The following properties are equivalent:

- (i)  $\rightarrow$  has the Church-Rosser property.
- (ii)  $\rightarrow$  is confluent.

# Confluence

---

Lemma 4.8:

If  $\rightarrow$  is confluent, then every element has at most one normal form.

Corollary 4.9:

If  $\rightarrow$  is normalizing and confluent, then every element  $b$  has a unique normal form.

Proposition 4.10:

If  $\rightarrow$  is normalizing and confluent, then  $b \leftrightarrow^* c$  if and only if  $b\downarrow = c\downarrow$ .

# Confluence and Local Confluence

---

Theorem 4.11 (“Newman’s Lemma”):

If a terminating relation  $\rightarrow$  is locally confluent, then it is confluent.

# Rewrite Relations

---

Corollary 4.12:

If  $E$  is convergent (i. e., terminating and confluent),  
then  $s \approx_E t$  if and only if  $s \leftrightarrow_E^* t$  if and only if  $s \downarrow_E = t \downarrow_E$ .

Corollary 4.13:

If  $E$  is finite and convergent, then  $\approx_E$  is decidable.

Reminder:

If  $E$  is terminating, then it is confluent if and only if it is locally confluent.

# Rewrite Relations

---

Problems:

Show local confluence of  $E$ .

Show termination of  $E$ .

Transform  $E$  into an equivalent set of equations that is locally confluent and terminating.

## 4.4 Critical Pairs

---

Showing local confluence (Sketch):

Problem: If  $t_1 \leftarrow_E t_0 \rightarrow_E t_2$ , does there exist a term  $s$  such that  $t_1 \rightarrow_E^* s \leftarrow_E^* t_2$ ?

If the two rewrite steps happen in different subtrees (disjoint redexes):  
yes.

If the two rewrite steps happen below each other (overlap at or below a variable position): yes.

If the left-hand sides of the two rules overlap at a non-variable position:  
needs further investigation.



# Critical Pairs

---

Showing local confluence (Sketch):

Question:

Are there rewrite rules  $l_1 \rightarrow r_1$  and  $l_2 \rightarrow r_2$  such that some subterm  $l_1|_p$  and  $l_2$  have a common instance  $(l_1|_p)\sigma_1 = l_2\sigma_2$ ?

Observation:

If we assume w.l.o.g. that the two rewrite rules do not have common variables, then only a single substitution is necessary:  $(l_1|_p)\sigma = l_2\sigma$ .

Further observation:

The mgu of  $l_1|_p$  and  $l_2$  subsumes all unifiers  $\sigma$  of  $l_1|_p$  and  $l_2$ .

# Critical Pairs

---

Let  $l_i \rightarrow r_i$  ( $i = 1, 2$ ) be two rewrite rules in a TRS  $R$  whose variables have been renamed such that  $\text{var}(l_1) \cap \text{var}(l_2) = \emptyset$ . (Remember that  $\text{var}(l_i) \supseteq \text{var}(r_i)$ .)

Let  $p \in \text{pos}(l_1)$  be a position such that  $l_1|_p$  is not a variable and  $\sigma$  is an mgu of  $l_1|_p$  and  $l_2$ .

Then  $r_1\sigma \leftarrow l_1\sigma \rightarrow (l_1\sigma)[r_2\sigma]_p$ .

$\langle r_1\sigma, (l_1\sigma)[r_2\sigma]_p \rangle$  is called a **critical pair** of  $R$ .

The critical pair is **joinable** (or: converges), if  $r_1\sigma \downarrow_R (l_1\sigma)[r_2\sigma]_p$ .

# Critical Pairs

---

Theorem 4.14 (“Critical Pair Theorem”):

A TRS  $R$  is locally confluent if and only if all its critical pairs are joinable.

Proof:

“only if”: obvious, since joinability of a critical pair is a special case of local confluence.

## Critical Pairs

---

“if” : Suppose  $s$  rewrites to  $t_1$  and  $t_2$  using rewrite rules  $l_i \rightarrow r_i \in R$  at positions  $p_i \in \text{pos}(s)$ , where  $i = 1, 2$ .

Without loss of generality, we can assume that the two rules are variable disjoint, hence  $s|_{p_i} = l_i\theta$  and  $t_i = s[r_i\theta]_{p_i}$ .

We distinguish between two cases: Either  $p_1$  and  $p_2$  are in disjoint subtrees ( $p_1 \parallel p_2$ ), or one is a prefix of the other (w.l.o.g.,  $p_1 \leq p_2$ ).

## Critical Pairs

---

Case 1:  $p_1 \parallel p_2$ .

Then  $s = s[l_1\theta]_{p_1}[l_2\theta]_{p_2}$ ,

and therefore  $t_1 = s[r_1\theta]_{p_1}[l_2\theta]_{p_2}$  and  $t_2 = s[l_1\theta]_{p_1}[r_2\theta]_{p_2}$ .

Let  $t_0 = s[r_1\theta]_{p_1}[r_2\theta]_{p_2}$ .

Then clearly  $t_1 \rightarrow_R t_0$  using  $l_2 \rightarrow r_2$  and  $t_2 \rightarrow_R t_0$  using  $l_1 \rightarrow r_1$ .

## Critical Pairs

---

Case 2:  $p_1 \leq p_2$ .

Case 2.1:  $p_2 = p_1 q_1 q_2$ , where  $l_1|_{q_1}$  is some variable  $x$ .

In other words, the second rewrite step takes place at or below a variable in the first rule. Suppose that  $x$  occurs  $m$  times in  $l_1$  and  $n$  times in  $r_1$  (where  $m \geq 1$  and  $n \geq 0$ ).

Then  $t_1 \rightarrow_R^* t_0$  by applying  $l_2 \rightarrow r_2$  at all positions  $p_1 q' q_2$ , where  $q'$  is a position of  $x$  in  $r_1$ .

Conversely,  $t_2 \rightarrow_R^* t_0$  by applying  $l_2 \rightarrow r_2$  at all positions  $p_1 q q_2$ , where  $q$  is a position of  $x$  in  $l_1$  different from  $q_1$ , and by applying  $l_1 \rightarrow r_1$  at  $p_1$  with the substitution  $\theta'$ , where  $\theta' = \theta[x \mapsto (x\theta)[r_2\theta]_{q_2}]$ .

## Critical Pairs

---

Case 2.2:  $p_2 = p_1 p$ , where  $p$  is a non-variable position of  $l_1$ .

Then  $s|_{p_2} = l_2\theta$  and  $s|_{p_2} = (s|_{p_1})|_p = (l_1\theta)|_p = (l_1|_p)\theta$ ,  
so  $\theta$  is a unifier of  $l_2$  and  $l_1|_p$ .

Let  $\sigma$  be the mgu of  $l_2$  and  $l_1|_p$ ,

then  $\theta = \tau \circ \sigma$  and  $\langle r_1\sigma, (l_1\sigma)[r_2\sigma]_p \rangle$  is a critical pair.

By assumption, it is joinable, so  $r_1\sigma \rightarrow_R^* v \leftarrow_R^* (l_1\sigma)[r_2\sigma]_p$ .

Consequently,  $t_1 = s[r_1\theta]_{p_1} = s[r_1\sigma\tau]_{p_1} \rightarrow_R^* s[v\tau]_{p_1}$  and  $t_2 = s[r_2\theta]_{p_2} =$   
 $s[(l_1\theta)[r_2\theta]_p]_{p_1} = s[(l_1\sigma\tau)[r_2\sigma\tau]_p]_{p_1} = s[((l_1\sigma)[r_2\sigma]_p)\tau]_{p_1} \rightarrow_R^* s[v\tau]_{p_1}$ .

This completes the proof of the Critical Pair Theorem. □

# Critical Pairs

---

Note: Critical pairs between a rule and (a renamed variant of) itself must be considered – except if the overlap is at the root (i. e.,  $p = \varepsilon$ ).



# Critical Pairs

---

Corollary 4.15:

A terminating TRS  $R$  is confluent if and only if all its critical pairs are joinable.

Corollary 4.16:

For a finite terminating TRS, confluence is decidable.

## 4.5 Termination

---

Termination problems:

Given a finite TRS  $R$  and a term  $t$ , are all  $R$ -reductions starting from  $t$  terminating?

Given a finite TRS  $R$ , are all  $R$ -reductions terminating?

# Termination

---

Proposition 4.17:

Both termination problems for TRSs are undecidable in general.

Proof:

Encode Turing machines using rewrite rules and reduce the (uniform) halting problems for TMs to the termination problems for TRSs. □

Consequence:

Decidable criteria for termination are not complete.

## Two Different Scenarios

---

Depending on the application, the TRS whose termination we want to show can be

- (i) fixed and known in advance, or
- (ii) evolving (e. g., generated by some saturation process).

Methods for case (ii) are also usable for case (i).

Many methods for case (i) are not usable for case (ii).

We will first consider case (ii);

additional techniques for case (i) will be considered later.

# Reduction Orderings

---

Goal:

Given a finite TRS  $R$ , show termination of  $R$  by looking at finitely many rules  $l \rightarrow r \in R$ , rather than at infinitely many possible replacement steps  $s \rightarrow_R s'$ .

# Reduction Orderings

---

A binary relation  $\sqsupseteq$  over  $T_\Sigma(X)$  is called **compatible with  $\Sigma$ -operations**, if  $s \sqsupseteq s'$  implies  $f(t_1, \dots, s, \dots, t_n) \sqsupseteq f(t_1, \dots, s', \dots, t_n)$  for all  $f \in \Omega$  and  $s, s', t_i \in T_\Sigma(X)$ .

Lemma 4.18:

The relation  $\sqsupseteq$  is compatible with  $\Sigma$ -operations, if and only if  $s \sqsupseteq s'$  implies  $t[s]_p \sqsupseteq t[s']_p$  for all  $s, s', t \in T_\Sigma(X)$  and  $p \in \text{pos}(t)$ .

Note: **compatible with  $\Sigma$ -operations** = **compatible with contexts**.

# Reduction Orderings

---

A binary relation  $\sqsupset$  over  $T_\Sigma(X)$  is called **stable under substitutions**, if  $s \sqsupset s'$  implies  $s\sigma \sqsupset s'\sigma$  for all  $s, s' \in T_\Sigma(X)$  and substitutions  $\sigma$ .

# Reduction Orderings

---

A binary relation  $\sqsupset$  is called a **rewrite relation**, if it is compatible with  $\Sigma$ -operations and stable under substitutions.

Example: If  $R$  is a TRS, then  $\rightarrow_R$  is a rewrite relation.

A strict partial ordering over  $T_\Sigma(X)$  that is a rewrite relation is called **rewrite ordering**.

A well-founded rewrite ordering is called **reduction ordering**.



# Reduction Orderings

---

Theorem 4.19:

A TRS  $R$  terminates if and only if there exists a reduction ordering  $\succ$  such that  $l \succ r$  for every rule  $l \rightarrow r \in R$ .

# The Interpretation Method

---

Proving termination by interpretation:

Let  $\mathcal{A}$  be a  $\Sigma$ -algebra;

let  $\succ$  be a well-founded strict partial ordering on its universe.

Define the ordering  $\succ_{\mathcal{A}}$  over  $T_{\Sigma}(X)$  by  $s \succ_{\mathcal{A}} t$  if and only if  $\mathcal{A}(\beta)(s) \succ \mathcal{A}(\beta)(t)$  for all assignments  $\beta : X \rightarrow U_{\mathcal{A}}$ .

Is  $\succ_{\mathcal{A}}$  a reduction ordering?

# The Interpretation Method

---

Lemma 4.20:

$\succ_{\mathcal{A}}$  is stable under substitutions.

# The Interpretation Method

---

A function  $\phi : U_{\mathcal{A}}^n \rightarrow U_{\mathcal{A}}$  is called **monotone** (w. r. t.  $\succ$ ),  
if  $a \succ a'$  implies  $\phi(b_1, \dots, a, \dots, b_n) \succ \phi(b_1, \dots, a', \dots, b_n)$   
for all  $a, a', b_i \in U_{\mathcal{A}}$ .

Lemma 4.21:

If the interpretation  $f_{\mathcal{A}}$  of every function symbol  $f$  is monotone w. r. t.  $\succ$ ,  
then  $\succ_{\mathcal{A}}$  is compatible with  $\Sigma$ -operations.

Theorem 4.22:

If the interpretation  $f_{\mathcal{A}}$  of every function symbol  $f$  is monotone w. r. t.  $\succ$ ,  
then  $\succ_{\mathcal{A}}$  is a reduction ordering.

# Polynomial Orderings

---

## Polynomial orderings:

Instance of the interpretation method:

The carrier set  $U_{\mathcal{A}}$  is  $\mathbb{N}$  or some subset of  $\mathbb{N}$ .

To every function symbol  $f/n$  we associate a polynomial  $P_f(X_1, \dots, X_n) \in \mathbb{N}[X_1, \dots, X_n]$  with coefficients in  $\mathbb{N}$  and indeterminates  $X_1, \dots, X_n$ .

Then we define  $f_{\mathcal{A}}(a_1, \dots, a_n) = P_f(a_1, \dots, a_n)$  for  $a_i \in U_{\mathcal{A}}$ .

# Polynomial Orderings

---

Requirement 1:

If  $a_1, \dots, a_n \in U_{\mathcal{A}}$ , then  $f_{\mathcal{A}}(a_1, \dots, a_n) \in U_{\mathcal{A}}$ .  
(Otherwise,  $\mathcal{A}$  would not be a  $\Sigma$ -algebra.)

# Polynomial Orderings

---

Requirement 2:

$f_{\mathcal{A}}$  must be monotone (w. r. t.  $\succ$ ).

From now on:

$$U_{\mathcal{A}} = \{ n \in \mathbb{N} \mid n \geq 1 \}.$$

If  $\text{arity}(f) = 0$ , then  $P_f$  is a constant  $\geq 1$ .

If  $\text{arity}(f) = n \geq 1$ , then  $P_f$  is a polynomial  $P(X_1, \dots, X_n)$ , such that every  $X_i$  occurs in some monomial  $m \cdot X_1^{j_1} \dots X_k^{j_k}$  with exponent at least 1 and non-zero coefficient  $m \in \mathbb{N}$ .

$\Rightarrow$  Requirements 1 and 2 are satisfied.

# Polynomial Orderings

---

The mapping from function symbols to polynomials can be extended to terms:

A term  $t$  containing the variables  $x_1, \dots, x_n$  yields a polynomial  $P_t$  with indeterminates  $X_1, \dots, X_n$  (where  $X_i$  corresponds to  $\beta(x_i)$ ).

Example:

$$\Omega = \{b/0, f/1, g/3\}$$

$$P_b = 3, \quad P_f(X_1) = X_1^2, \quad P_g(X_1, X_2, X_3) = X_1 + X_2X_3.$$

$$\text{Let } t = g(f(b), f(x), y), \text{ then } P_t(X, Y) = 9 + X^2Y.$$



# Polynomial Orderings

---

If  $P, Q$  are polynomials in  $\mathbb{N}[X_1, \dots, X_n]$ , we write  $P > Q$  if  $P(a_1, \dots, a_n) > Q(a_1, \dots, a_n)$  for all  $a_1, \dots, a_n \in U_{\mathcal{A}}$ .

Clearly,  $s \succ_{\mathcal{A}} t$  if and only if  $P_s > P_t$  if and only if  $P_s - P_t > 0$ .

Question: Can we check  $P_s - P_t > 0$  automatically?

# Polynomial Orderings

---

## Hilbert's 10th Problem:

Given a polynomial  $P \in \mathbb{Z}[X_1, \dots, X_n]$  with integer coefficients, is  $P = 0$  for some  $n$ -tuple of natural numbers?

Theorem 4.23:

Hilbert's 10th Problem is undecidable.

Proposition 4.24:

Given a polynomial interpretation and two terms  $s, t$ , it is undecidable whether  $P_s > P_t$ .

Proof:

By reduction of Hilbert's 10th Problem. □

# Polynomial Orderings

---

One easy case:

If we restrict to linear polynomials, deciding whether  $P_s - P_t > 0$  is trivial:

$\sum k_i a_i + k > 0$  for all  $a_1, \dots, a_n \geq 1$  if and only if

$$k_i \geq 0 \text{ for all } i \in \{1, \dots, n\},$$

$$\text{and } \sum k_i + k > 0$$

# Polynomial Orderings

---

Another possible solution:

Test whether  $P_s(a_1, \dots, a_n) > P_t(a_1, \dots, a_n)$   
for all  $a_1, \dots, a_n \in \{x \in \mathbb{R} \mid x \geq 1\}$ .

This is decidable (but hard).

Since  $U_{\mathcal{A}} \subseteq \{x \in \mathbb{R} \mid x \geq 1\}$ , it implies  $P_s > P_t$ .

Alternatively:

Use fast overapproximations.

# Simplification Orderings

---

The **proper subterm ordering**  $\triangleright$  is defined by  $s \triangleright t$  if and only if  $s|_p = t$  for some position  $p \neq \varepsilon$  of  $s$ .

# Simplification Orderings

---

A rewrite ordering  $\succ$  over  $T_\Sigma(X)$  is called **simplification ordering**, if it has the **subterm property**:

$s \triangleright t$  implies  $s \succ t$  for all  $s, t \in T_\Sigma(X)$ .

Example:

Let  $R_{\text{emb}}$  be the rewrite system

$$R_{\text{emb}} = \{ f(x_1, \dots, x_n) \rightarrow x_i \mid f/n \in \Omega, 1 \leq i \leq n \}.$$

Define  $\triangleright_{\text{emb}} = \rightarrow_{R_{\text{emb}}}^+$  and  $\triangleleft_{\text{emb}} = \rightarrow_{R_{\text{emb}}}^*$   
(“homeomorphic embedding relation”).

$\triangleright_{\text{emb}}$  is a simplification ordering.

# Simplification Orderings

---

Lemma 4.25:

If  $\succ$  is a simplification ordering, then  $s \triangleright_{\text{emb}} t$  implies  $s \succ t$

and  $s \triangleleft_{\text{emb}} t$  implies  $s \succ t$ .

# Simplification Orderings

---

Goal:

Show that every simplification ordering is well-founded  
(and therefore a reduction ordering).

Note: This works only for **finite** signatures!

To fix this for infinite signatures, the definition of simplification orderings  
and the definition of embedding have to be modified.



# Simplification Orderings

---

Theorem 4.26 (“Kruskal’s Theorem”):

Let  $\Sigma$  be a finite signature, let  $X$  be a finite set of variables. Then for every infinite sequence  $t_1, t_2, t_3, \dots$  there are indices  $j > i$  such that  $t_j \succeq_{\text{emb}} t_i$ . ( $\succeq_{\text{emb}}$  is called a **well-partial-ordering (wpo)**.)

Proof:

See Baader and Nipkow, page 113–115.

□

# Simplification Orderings

---

Theorem 4.27 (Dershowitz):

If  $\Sigma$  is a finite signature, then every simplification ordering  $\succ$  on  $T_{\Sigma}(X)$  is well-founded (and therefore a reduction ordering).

# Simplification Orderings

---

There are reduction orderings that are not simplification orderings and terminating TRSs that are not contained in any simplification ordering.

Example:

Let  $R = \{f(f(x)) \rightarrow f(g(f(x)))\}$ .

$R$  terminates and  $\rightarrow_R^+$  is therefore a reduction ordering.

Assume that  $\rightarrow_R$  were contained in a simplification ordering  $\succ$ .

Then  $f(f(x)) \rightarrow_R f(g(f(x)))$  implies  $f(f(x)) \succ f(g(f(x)))$ , and

$f(g(f(x))) \sqsupseteq_{\text{emb}} f(f(x))$  implies  $f(g(f(x))) \succeq f(f(x))$ , hence

$f(f(x)) \succ f(f(x))$ .

# Path Orderings

---

Let  $\Sigma = (\Omega, \Pi)$  be a finite signature, let  $\succ$  be a strict partial ordering (“precedence”) on  $\Omega$ .

The **lexicographic path ordering**  $\succ_{\text{lpo}}$  on  $T_{\Sigma}(X)$  induced by  $\succ$  is defined by:

$s \succ_{\text{lpo}} t$  if

(1)  $t \in \text{var}(s)$  and  $t \neq s$ , or

(2)  $s = f(s_1, \dots, s_m)$ ,  $t = g(t_1, \dots, t_n)$ , and

(a)  $s_i \succeq_{\text{lpo}} t$  for some  $i$ , or

(b)  $f \succ g$  and  $s \succ_{\text{lpo}} t_j$  for all  $j$ , or

(c)  $f = g$ ,  $s \succ_{\text{lpo}} t_j$  for all  $j$ , and  $(s_1, \dots, s_m) (\succ_{\text{lpo}})_{\text{lex}} (t_1, \dots, t_n)$ .

# Path Orderings

---

Lemma 4.28:

$s \succ_{\text{lpo}} t$  implies  $\text{var}(s) \supseteq \text{var}(t)$ .

Theorem 4.29:

$\succ_{\text{lpo}}$  is a simplification ordering on  $T_{\Sigma}(X)$ .

Theorem 4.30:

If the precedence  $\succ$  is total, then the lexicographic path ordering  $\succ_{\text{lpo}}$  is total on ground terms, i. e., for all  $s, t \in T_{\Sigma}(\emptyset)$ :

$s \succ_{\text{lpo}} t \vee t \succ_{\text{lpo}} s \vee s = t$ .

# Path Orderings

---

Recapitulation:

Let  $\Sigma = (\Omega, \Pi)$  be a finite signature, let  $\succ$  be a strict partial ordering (“precedence”) on  $\Omega$ . The **lexicographic path ordering**  $\succ_{\text{lpo}}$  on  $T_{\Sigma}(X)$  induced by  $\succ$  is defined by:  $s \succ_{\text{lpo}} t$  if

- (1)  $t \in \text{var}(s)$  and  $t \neq s$ , or
- (2)  $s = f(s_1, \dots, s_m)$ ,  $t = g(t_1, \dots, t_n)$ , and
  - (a)  $s_i \succeq_{\text{lpo}} t$  for some  $i$ , or
  - (b)  $f \succ g$  and  $s \succ_{\text{lpo}} t_j$  for all  $j$ , or
  - (c)  $f = g$ ,  $s \succ_{\text{lpo}} t_j$  for all  $j$ , and  $(s_1, \dots, s_m) (\succ_{\text{lpo}})_{\text{lex}} (t_1, \dots, t_n)$ .

# Path Orderings

---

There are several possibilities to compare subterms in (2)(c):

- compare list of subterms lexicographically left-to-right (“lexicographic path ordering (lpo)”, Kamin and Lévy)
- compare list of subterms lexicographically right-to-left (or according to some permutation  $\pi$ )
- compare multiset of subterms using the multiset extension (“multiset path ordering (mpo)”, Dershowitz)
- to each function symbol  $f/n \in \Omega$  with  $n \geq 1$  associate a status  $\in \{\text{mul}\} \cup \{\text{lex}_\pi \mid \pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}\}$  and compare according to that status (“recursive path ordering (rpo) with status”)

# The Knuth-Bendix Ordering

---

Let  $\Sigma = (\Omega, \Pi)$  be a finite signature,

let  $\succ$  be a strict partial ordering (“precedence”) on  $\Omega$ ,

let  $w : \Omega \cup X \rightarrow \mathbb{R}_0^+$  be a **weight function**,

such that the following admissibility conditions are satisfied:

$w(x) = w_0 \in \mathbb{R}^+$  for all variables  $x \in X$ ;

$w(c) \geq w_0$  for all constants  $c \in \Omega$ .

If  $w(f) = 0$  for some  $f/1 \in \Omega$ , then  $f \succ g$  for all  $g/n \in \Omega$  with  $f \neq g$ .



# The Knuth-Bendix Ordering

---

The weight function  $w$  can be extended to terms recursively:

$$w(f(t_1, \dots, t_n)) = w(f) + \sum_{1 \leq i \leq n} w(t_i)$$

or alternatively

$$w(t) = \sum_{x \in \text{var}(t)} w(x) \cdot \#(x, t) + \sum_{f \in \Omega} w(f) \cdot \#(f, t).$$

where  $\#(a, t)$  is the number of occurrences of  $a$  in  $t$ .

# The Knuth-Bendix Ordering

---

The **Knuth-Bendix ordering**  $\succ_{\text{kbo}}$  on  $T_{\Sigma}(X)$  induced by  $\succ$  and  $w$  is defined by:  $s \succ_{\text{kbo}} t$  if

- (1)  $\#(x, s) \geq \#(x, t)$  for all variables  $x$  and  $w(s) > w(t)$ , or
- (2)  $\#(x, s) \geq \#(x, t)$  for all variables  $x$ ,  $w(s) = w(t)$ , and
  - (a)  $t = x$ ,  $s = f^n(x)$  for some  $n \geq 1$ , or
  - (b)  $s = f(s_1, \dots, s_m)$ ,  $t = g(t_1, \dots, t_n)$ , and  $f \succ g$ , or
  - (c)  $s = f(s_1, \dots, s_m)$ ,  $t = f(t_1, \dots, t_m)$ , and  $(s_1, \dots, s_m) (\succ_{\text{kbo}})_{\text{lex}} (t_1, \dots, t_m)$ .

# The Knuth-Bendix Ordering

---

Theorem 4.31:

The Knuth-Bendix ordering induced by  $\succ$  and  $w$  is a simplification ordering on  $T_{\Sigma}(X)$ .

Proof:

Baader and Nipkow, pages 125–129.

□

## Remark

---

If  $\Pi \neq \emptyset$ , then all the term orderings described in this section can also be used to compare non-equational atoms by treating predicate symbols like function symbols.

## 4.6 Knuth-Bendix Completion

---

### Completion:

Goal: Given a set  $E$  of equations, transform  $E$  into an equivalent convergent set  $R$  of rewrite rules.

(If  $R$  is finite: decision procedure for  $E$ .)

# Knuth-Bendix Completion: Idea

---

How to ensure termination?

Fix a reduction ordering  $\succ$  and construct  $R$  in such a way that  $\rightarrow_R \subseteq \succ$  (i. e.,  $l \succ r$  for every  $l \rightarrow r \in R$ ).

How to ensure confluence?

Check that all critical pairs are joinable.

Note: Every critical pair  $\langle s, t \rangle$  can be *made* joinable by adding  $s \rightarrow t$  or  $t \rightarrow s$  to  $R$ .

(Actually, we first add  $s \approx t$  to  $E$  and later try to turn it into a rule that is contained in  $\succ$ ; this gives us some additional degree of freedom.)

# Knuth-Bendix Completion: Inference Rules

---

The completion procedure is presented as a set of inference rules working on a set of equations  $E$  and a set of rules  $R$ :

$$E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \dots$$

At the beginning,  $E = E_0$  is the input set and  $R = R_0$  is empty.

At the end,  $E$  should be empty; then  $R$  is the result.

For each step  $E, R \vdash E', R'$ , the equational theories of  $E \cup R$  and  $E' \cup R'$  agree:  $\approx_{E \cup R} = \approx_{E' \cup R'}$ .

# Knuth-Bendix Completion: Inference Rules

---

Notations:

The formula  $s \overset{\cdot}{\approx} t$  denotes either  $s \approx t$  or  $t \approx s$ .

$CP(R)$  denotes the set of all critical pairs between rules in  $R$ .



# Knuth-Bendix Completion: Inference Rules

---

*Orient:*

$$\frac{E \cup \{s \approx t\}, R}{E, R \cup \{s \rightarrow t\}} \quad \text{if } s \succ t$$

Note: There are equations  $s \approx t$  that cannot be oriented, i. e., neither  $s \succ t$  nor  $t \succ s$ .

# Knuth-Bendix Completion: Inference Rules

---

Trivial equations cannot be oriented – but we don't need them anyway:

*Delete:*

$$\frac{E \cup \{s \approx s\}, R}{E, R}$$

# Knuth-Bendix Completion: Inference Rules

---

Critical pairs between rules in  $R$  are turned into additional equations:

*Deduce:*

$$\frac{E, R}{E \cup \{s \approx t\}, R} \quad \text{if } \langle s, t \rangle \in \text{CP}(R).$$

Note: If  $\langle s, t \rangle \in \text{CP}(R)$  then  $s \leftarrow_R u \rightarrow_R t$  and hence  $R \models s \approx t$ .

# Knuth-Bendix Completion: Inference Rules

---

The following inference rules are not absolutely necessary, but very useful (e. g., to get rid of joinable critical pairs and to deal with equations that cannot be oriented):

*Simplify-Eq:*

$$\frac{E \cup \{s \approx t\}, R}{E \cup \{u \approx t\}, R} \quad \text{if } s \rightarrow_R u.$$

# Knuth-Bendix Completion: Inference Rules

---

Simplification of the right-hand side of a rule is unproblematic:

*R-Simplify-Rule:*

$$\frac{E, R \cup \{s \rightarrow t\}}{E, R \cup \{s \rightarrow u\}} \quad \text{if } t \rightarrow_R u.$$

Simplification of the left-hand side may influence orientability and orientation. Therefore, it yields an *equation*:

*L-Simplify-Rule:*

$$\frac{E, R \cup \{s \rightarrow t\}}{E \cup \{u \approx t\}, R} \quad \text{if } s \rightarrow_R u \text{ using a rule } l \rightarrow r \in R \\ \text{such that } s \sqsupset l \text{ (see next slide).}$$

# Knuth-Bendix Completion: Inference Rules

---

For technical reasons, the lhs of  $s \rightarrow t$  may only be simplified using a rule  $l \rightarrow r$ , if  $l \rightarrow r$  cannot be simplified using  $s \rightarrow t$ , that is, if  $s \sqsupset l$ , where the **encompassment quasi-ordering**  $\sqsupseteq$  is defined by

$$s \sqsupseteq l \text{ if } s|_p = l\sigma \text{ for some } p \text{ and } \sigma$$

and  $\sqsupset = \sqsupseteq \setminus \sqsubseteq$  is the strict part of  $\sqsupseteq$ .

Lemma 4.32:

$\sqsupset$  is a well-founded strict partial ordering.

# Knuth-Bendix Completion: Inference Rules

---

Lemma 4.33:

If  $E, R \vdash E', R'$ , then  $\approx_{E \cup R} = \approx_{E' \cup R'}$ .

Lemma 4.34:

If  $E, R \vdash E', R'$  and  $\rightarrow_R \subseteq \succ$ , then  $\rightarrow_{R'} \subseteq \succ$ .

# Knuth-Bendix Completion: Inference Rules

---

Note: Like in ordered resolution, simplification should be preferred to deduction:

- Simplify/delete whenever possible.
- Otherwise, orient an equation, if possible.
- Last resort: compute critical pairs.



# Knuth-Bendix Completion: Correctness Proof

---

What can happen if we run the completion procedure on a set  $E$  of equations?

- (1) We reach a state where no more inference rules are applicable and  $E$  is not empty.  
⇒ Failure (try again with another ordering?)
- (2) We reach a state where  $E$  is empty and all critical pairs between the rules in the current  $R$  have been checked.
- (3) The procedure runs forever.

In order to treat these cases simultaneously, we need some definitions.

# Knuth-Bendix Completion: Correctness Proof

---

A (finite or infinite sequence)  $E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \dots$  with  $R_0 = \emptyset$  is called a **run** of the completion procedure with input  $E_0$  and  $\succ$ .

For a run,  $E_U = \bigcup_{i \geq 0} E_i$  and  $R_U = \bigcup_{i \geq 0} R_i$ .

The sets of **persistent equations or rules** of the run are  $E_\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} E_j$  and  $R_\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} R_j$ .

Note: If the run is finite and ends with  $E_n, R_n$ , then  $E_\infty = E_n$  and  $R_\infty = R_n$ .

*This notation differs significantly from the 2021/2022 lecture. Keep that in mind when you use online lecture recordings or read exercises or exam questions from previous years.*

# Knuth-Bendix Completion: Correctness Proof

---

A run is called **fair**, if  $CP(R_\infty) \subseteq E_U$

(i. e., if every critical pair between persisting rules is computed at some step of the derivation).

Goal:

Show: If a run is fair and  $E_\infty$  is empty,  
then  $R_\infty$  is convergent and equivalent to  $E_0$ .

In particular: If a run is fair and  $E_\infty$  is empty,  
then  $\approx_{E_0} = \approx_{E_U \cup R_U} = \leftrightarrow_{E_U \cup R_U}^* = \downarrow R_\infty$ .

# Knuth-Bendix Completion: Correctness Proof

---

General assumptions from now on:

$E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \dots$  is a fair run.

$R_0$  and  $E_\infty$  are empty.

# Knuth-Bendix Completion: Correctness Proof

---

A **proof** of  $s \approx t$  in  $E_U \cup R_U$  is a finite sequence  $(s_0, \dots, s_n)$  such that  $s = s_0$ ,  $t = s_n$ , and for all  $i \in \{1, \dots, n\}$ :

(1)  $s_{i-1} \leftrightarrow_{E_U} s_i$ , or

(2)  $s_{i-1} \rightarrow_{R_U} s_i$ , or

(3)  $s_{i-1} \leftarrow_{R_U} s_i$ .

The pairs  $(s_{i-1}, s_i)$  are called **proof steps**.

A proof is called a **rewrite proof in  $R_\infty$** ,

if there is a  $k \in \{0, \dots, n\}$  such that  $s_{i-1} \rightarrow_{R_\infty} s_i$  for  $1 \leq i \leq k$

and  $s_{i-1} \leftarrow_{R_\infty} s_i$  for  $k + 1 \leq i \leq n$

# Knuth-Bendix Completion: Correctness Proof

---

Idea (Bachmair, Dershowitz, Hsiang):

Define a well-founded ordering on proofs, such that for every proof that is not a rewrite proof in  $R_\infty$  there is an equivalent smaller proof.

Consequence: For every proof there is an equivalent rewrite proof in  $R_\infty$ .

# Knuth-Bendix Completion: Correctness Proof

---

We associate a **cost**  $c(s_{i-1}, s_i)$  with every proof step as follows:

- (1) If  $s_{i-1} \leftrightarrow_{E \cup} s_i$ , then  $c(s_{i-1}, s_i) = (\{s_{i-1}, s_i\}, -, -)$ ,  
where the first component is a multiset of terms and  $-$  denotes an arbitrary (irrelevant) term.
- (2) If  $s_{i-1} \rightarrow_{R \cup} s_i$  using  $l \rightarrow r$ , then  $c(s_{i-1}, s_i) = (\{s_{i-1}\}, l, s_i)$ .
- (3) If  $s_{i-1} \leftarrow_{R \cup} s_i$  using  $l \rightarrow r$ , then  $c(s_{i-1}, s_i) = (\{s_i\}, l, s_{i-1})$ .

Proof steps are compared using the lexicographic combination of the multiset extension of the reduction ordering  $\succ$ , the encompassment ordering  $\sqsupseteq$ , and the reduction ordering  $\succ$ .

# Knuth-Bendix Completion: Correctness Proof

---

The cost  $c(P)$  of a proof  $P$  is the multiset of the costs of its proof steps.

The **proof ordering**  $\succ_C$  compares the costs of proofs using the multiset extension of the proof step ordering.

Lemma 4.35:

$\succ_C$  is a well-founded ordering.



# Knuth-Bendix Completion: Correctness Proof

---

Lemma 4.36:

Let  $P$  be a proof in  $E_{\cup} \cup R_{\cup}$ . If  $P$  is not a rewrite proof in  $R_{\infty}$ , then there exists an equivalent proof  $P'$  in  $E_{\cup} \cup R_{\cup}$  such that  $P \succ_C P'$ .

Proof:

If  $P$  is not a rewrite proof in  $R_{\infty}$ , then it contains

- (a) a proof step that is in  $E_{\cup}$ , or
- (b) a proof step that is in  $R_{\cup} \setminus R_{\infty}$ , or
- (c) a subproof  $s_{i-1} \leftarrow_{R_{\infty}} s_i \rightarrow_{R_{\infty}} s_{i+1}$  (peak).

We show that in all three cases the proof step or subproof can be replaced by a smaller subproof:

# Knuth-Bendix Completion: Correctness Proof

---

Case (a): A proof step using an equation  $s \dot{\approx} t$  is in  $E_U$ .  
This equation must be deleted during the run.

If  $s \dot{\approx} t$  is deleted using *Orient*:

$$\dots S_{i-1} \leftrightarrow_{E_U} S_i \dots \implies \dots S_{i-1} \rightarrow_{R_U} S_i \dots$$

If  $s \dot{\approx} t$  is deleted using *Delete*:

$$\dots S_{i-1} \leftrightarrow_{E_U} S_{i-1} \dots \implies \dots S_{i-1} \dots$$

If  $s \dot{\approx} t$  is deleted using *Simplify-Eq*:

$$\dots S_{i-1} \leftrightarrow_{E_U} S_i \dots \implies \dots S_{i-1} \rightarrow_{R_U} S' \leftrightarrow_{E_U} S_i \dots$$

# Knuth-Bendix Completion: Correctness Proof

---

Case (b): A proof step using a rule  $s \rightarrow t$  is in  $R_U \setminus R_\infty$ .

This rule must be deleted during the run.

If  $s \rightarrow t$  is deleted using *R-Simplify-Rule*:

$$\dots S_{i-1} \rightarrow_{R_U} S_i \dots \implies \dots S_{i-1} \rightarrow_{R_U} S' \leftarrow_{R_U} S_i \dots$$

If  $s \rightarrow t$  is deleted using *L-Simplify-Rule*:

$$\dots S_{i-1} \rightarrow_{R_U} S_i \dots \implies \dots S_{i-1} \rightarrow_{R_U} S' \leftrightarrow_{E_U} S_i \dots$$

# Knuth-Bendix Completion: Correctness Proof

---

Case (c): A subproof has the form  $s_{i-1} \leftarrow_{R_\infty} s_i \rightarrow_{R_\infty} s_{i+1}$ .

If there is no overlap or a non-critical overlap:

$$\dots s_{i-1} \leftarrow_{R_\infty} s_i \rightarrow_{R_\infty} s_{i+1} \dots \implies \dots s_{i-1} \rightarrow_{R_\infty}^* s' \leftarrow_{R_\infty}^* s_{i+1} \dots$$

If there is a critical pair that has been added using *Deduce*:

$$\dots s_{i-1} \leftarrow_{R_\infty} s_i \rightarrow_{R_\infty} s_{i+1} \dots \implies \dots s_{i-1} \leftrightarrow_{E_U} s_{i+1} \dots$$

In all cases, checking that the replacement subproof is smaller than the replaced subproof is routine. □

# Knuth-Bendix Completion: Correctness Proof

---

Theorem 4.37:

Let  $E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \dots$  be a fair run and let  $R_0$  and  $E_\infty$  be empty. Then

- (1) every proof in  $E_\cup \cup R_\cup$  is equivalent to a rewrite proof in  $R_\infty$ ,
- (2)  $R_\infty$  is equivalent to  $E_0$ , and
- (3)  $R_\infty$  is convergent.

# Knuth-Bendix Completion: Correctness Proof

---

Proof:

(1) By well-founded induction on  $\succ_C$  using the previous lemma.

(2) Clearly  $\approx_{E \cup R_U} = \approx_{E_0}$ .

Since  $R_\infty \subseteq R_U$ , we get  $\approx_{R_\infty} \subseteq \approx_{E \cup R_U}$ .

On the other hand, by (1),  $\approx_{E \cup R_U} \subseteq \approx_{R_\infty}$ .

(3) Since  $\rightarrow_{R_\infty} \subseteq \succ$ ,  $R_\infty$  is terminating.

By (1),  $R_\infty$  is confluent. □

## 4.7 Unfailing Completion

---

Classical completion:

Try to transform a set  $E$  of equations into an equivalent convergent TRS.

Fail, if an equation can neither be oriented nor deleted.

Unfailing completion (Bachmair, Dershowitz and Plaisted):

If an equation cannot be oriented, we can still use *orientable instances* for rewriting.

Note: If  $\succ$  is total on ground terms, then every *ground instance* of an equation is trivial or can be oriented.

Goal: Derive a *ground convergent* set of equations.

# Unfailing Completion

---

Let  $E$  be a set of equations, let  $\succ$  be a reduction ordering.

We define the relation  $\rightarrow_{E\succ}$  by

$$s \rightarrow_{E\succ} t \quad \text{if} \quad \begin{array}{l} \text{there exist } (u \approx v) \in E \text{ or } (v \approx u) \in E, \\ p \in \text{pos}(s), \text{ and } \sigma : X \rightarrow T_{\Sigma}(X), \\ \text{such that } s|_p = u\sigma \text{ and } t = s[v\sigma]_p \\ \text{and } u\sigma \succ v\sigma. \end{array}$$

Note:  $\rightarrow_{E\succ}$  is terminating by construction.



# Unfailing Completion

---

From now on let  $\succ$  be a reduction ordering that is total on ground terms.

$E$  is called ground convergent w. r. t.  $\succ$ , if for all ground terms  $s$  and  $t$  with  $s \leftrightarrow_E^* t$  there exists a ground term  $v$  such that  $s \rightarrow_{E \succ}^* v \leftarrow_{E \succ}^* t$ .

(Analogously for  $E \cup R$ .)

# Unfailing Completion

---

As for standard completion, we establish ground convergence by computing critical pairs.

However, the ordering  $\succ$  is not total on non-ground terms.

Since  $s\theta \succ t\theta$  implies  $s \not\prec t$ , we approximate  $\succ$  on ground terms by  $\not\prec$  on arbitrary terms.

# Unfailing Completion

---

Let  $u_i \dot{\approx} v_i$  ( $i = 1, 2$ ) be equations in  $E$  whose variables have been renamed such that  $\text{var}(u_1 \dot{\approx} v_1) \cap \text{var}(u_2 \dot{\approx} v_2) = \emptyset$ .

Let  $p \in \text{pos}(u_1)$  be a position such that  $u_1|_p$  is not a variable,  $\sigma$  is an mgu of  $u_1|_p$  and  $u_2$ , and  $u_i\sigma \not\dot{\approx} v_i\sigma$  ( $i = 1, 2$ ).

Then  $\langle v_1\sigma, (u_1\sigma)[v_2\sigma]_p \rangle$  is called a **semi-critical pair** of  $E$  with respect to  $\succ$ .

The set of all semi-critical pairs of  $E$  is denoted by  $\text{SP}_{\succ}(E)$ .

Semi-critical pairs of  $E \cup R$  are defined analogously.

If  $\rightarrow_R \subseteq \succ$ , then  $\text{CP}(R)$  and  $\text{SP}_{\succ}(R)$  agree.

# Unfailing Completion

---

Note: In contrast to critical pairs, it may be necessary to consider overlaps of an equation with itself at the top.

For instance, if  $E = \{f(x) \approx g(y)\}$ , then  $\langle g(y), g(y') \rangle$  is a non-trivial semi-critical pair.

# Unfailing Completion

---

The *Deduce* rule takes now the following form:

*Deduce:*

$$\frac{E, R}{E \cup \{s \approx t\}, R} \quad \text{if } \langle s, t \rangle \in \text{SP}_{\succ}(E \cup R).$$

Moreover, the fairness criterion for runs is replaced by

$$\text{SP}_{\succ}(E_{\infty} \cup R_{\infty}) \subseteq E_{\cup}$$

(i. e., if every semi-critical pair between persisting rules or equations is computed at some step of the derivation).

# Unfailing Completion

---

Analogously to Thm. 4.37 we obtain now the following theorem:

Theorem 4.38:

Let  $E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \dots$  be a fair run; let  $R_0 = \emptyset$ . Then

- (1)  $E_\infty \cup R_\infty$  is equivalent to  $E_0$ , and
- (2)  $E_\infty \cup R_\infty$  is ground convergent.

## Unfailing Completion

---

Moreover one can show that, whenever there exists a *reduced* convergent  $R$  such that  $\approx_{E_0} = \downarrow_R$  and  $\rightarrow_R \in \succ$ , then for every fair *and simplifying* run  $E_\infty = \emptyset$  and  $R_\infty = R$  up to variable renaming.

Here  $R$  is called reduced, if for every  $l \rightarrow r \in R$ , both  $l$  and  $r$  are irreducible w. r. t.  $R \setminus \{l \rightarrow r\}$ .

A run is called simplifying, if  $R_\infty$  is reduced, and for all equations  $u \approx v \in E_\infty$ ,  $u$  and  $v$  are incomparable w. r. t.  $\succ$  and irreducible w. r. t.  $R_\infty$ .

# Unfailing Completion

---

Unfailing completion is refutationally complete for equational theories:

Theorem 4.39:

Let  $E$  be a set of equations, let  $\succ$  be a reduction ordering that is total on ground terms.

For any two terms  $s$  and  $t$ , let  $\hat{s}$  and  $\hat{t}$  be the terms obtained from  $s$  and  $t$  by replacing all variables by Skolem constants.

Let  $eq/2$ ,  $true/0$  and  $false/0$  be new operator symbols, such that  $true$  and  $false$  are smaller than all other terms.

Let  $E_0 = E \cup \{eq(\hat{s}, \hat{t}) \approx true, eq(x, x) \approx false\}$ .

If  $E_0, \emptyset \vdash E_1, R_1 \vdash E_2, R_2 \vdash \dots$  be a fair run of unfailing completion, then  $s \approx_E t$  if and only if some  $E_i \cup R_i$  contains  $true \approx false$ .



# Unfailing Completion

---

Outlook:

Combine ordered resolution and unfailing completion to get a calculus for equational clauses:

compute inferences between (strictly) maximal literals as in ordered resolution,

compute overlaps between maximal sides of equations as in unfailing completion

⇒ Superposition calculus.

## Part 5: Termination Revisited

---

So far: Termination as a subordinate task for entailment checking.

TRS is generated by some saturation process; ordering must be chosen before the saturation starts.

Now: Termination as a main task (e. g., for program analysis).

TRS is fixed and known in advance.

# Termination Revisited

---

Literature:

Nao Hirokawa and Aart Middeldorp: Dependency Pairs Revisited, RTA 2004, pp. 249-268 (in particular Sect. 1–4).

Thomas Arts and Jürgen Giesl: Termination of Term Rewriting Using Dependency Pairs, Theoretical Computer Science, 236:133-178, 2000.

## 5.1 Dependency Pairs

---

Invented by T. Arts and J. Giesl in 1996,  
many refinements since then.

Given: finite TRS  $R$  over  $\Sigma = (\Omega, \emptyset)$ .

$T_0 := \{ t \in T_\Sigma(X) \mid \exists \text{ infinite deriv. } t \rightarrow_R t_1 \rightarrow_R t_2 \rightarrow_R \dots \}$ .

$T_\infty := \{ t \in T_0 \mid \forall p > \varepsilon : t|_p \notin T_0 \}$   
= minimal elements of  $T_0$  w. r. t.  $\triangleright$ .

$t \in T_0 \Rightarrow$  there exists a  $t' \in T_\infty$  such that  $t \triangleright t'$ .

$R$  is non-terminating if and only if  $T_0 \neq \emptyset$  if and only if  $T_\infty \neq \emptyset$ .

## Dependency Pairs

---

Assume that  $T_\infty \neq \emptyset$  and consider some non-terminating derivation starting from  $t \in T_\infty$ .

Since all subterms of  $t$  allow only finite derivations, at some point a rule  $l \rightarrow r \in R$  must be applied at the root of  $t$  (possibly preceded by rewrite steps below the root):

$$t = f(t_1, \dots, t_n) \xrightarrow{>\varepsilon}_R^* f(s_1, \dots, s_n) = l\sigma \xrightarrow{\varepsilon}_R r\sigma.$$

In particular,  $\text{root}(t) = \text{root}(l)$ , so we see that the root symbol of any term in  $T_\infty$  must be contained in

$$D := \{ \text{root}(l) \mid l \rightarrow r \in R \}.$$

$D$  is called the set of **defined symbols** of  $R$ ;

$C := \Omega \setminus D$  is called the set of **constructor symbols** of  $R$ .

## Dependency Pairs

---

The term  $r\sigma$  is contained in  $T_0$ , so there exists a  $v \in T_\infty$  such that  $r\sigma \triangleright v$ .

If  $v$  occurred in  $r\sigma$  at or below a variable position of  $r$ , then  $x\sigma|_p = v$  for some  $x \in \text{var}(r) \subseteq \text{var}(l)$ , hence  $s_i \triangleright x\sigma$  and there would be an infinite derivation starting from some  $t_i$ .

This contradicts  $t \in T_\infty$ , though.

Therefore,  $v = u\sigma$  for some non-variable subterm  $u$  of  $r$ .

As  $v \in T_\infty$ , we see that  $\text{root}(u) = \text{root}(v) \in D$ .

Moreover,  $u$  cannot be a proper subterm of  $l$ , since otherwise again there would be an infinite derivation starting from some  $t_i$ .

## Dependency Pairs

---

Putting everything together, we obtain

$$t = f(t_1, \dots, t_n) \xrightarrow{>\varepsilon}_R^* f(s_1, \dots, s_n) = l\sigma \xrightarrow{\varepsilon}_R r\sigma \triangleright u\sigma$$

where  $r \triangleright u$ ,  $u$  is not a variable,  $\text{root}(u) \in D$ ,  $l \not\triangleright u$ .

Since  $u\sigma \in T_\infty$ , we can continue this process and obtain an infinite sequence.

# Dependency Pairs

---

If we define

$$S := \{ l \rightarrow u \mid l \rightarrow r \in R, r \triangleright u, u \notin X, \text{root}(u) \in D, l \not\triangleright u \},$$

we can combine the rewrite step at the root and the subterm step and obtain

$$t \xrightarrow{>\varepsilon}_R^* l\sigma \xrightarrow{\varepsilon}_S u\sigma.$$



## Dependency Pairs

---

To get rid of the superscripts  $\varepsilon$  and  $>\varepsilon$ , it turns out to be useful to introduce a new set of function symbols  $f^\#$  that are only used for the root symbols of this derivation:

$$\Omega^\# := \{ f^\# / n \mid f / n \in \Omega \}.$$

For a term  $t = f(t_1, \dots, t_n)$  we define  $t^\# := f^\#(t_1, \dots, t_n)$ ;  
for a set of terms  $T$  we define  $T^\# := \{ t^\# \mid t \in T \}$ .

The set of **dependency pairs** of a TRS  $R$  is then defined by

$$\text{DP}(R) := \{ l^\# \rightarrow u^\# \mid l \rightarrow r \in R, r \triangleright u, u \notin X, \text{root}(u) \in D, l \not\triangleright u \}.$$

## Dependency Pairs

---

For  $t \in T_\infty$ , the sequence using the  $S$ -rule corresponds now to

$$t^\# \rightarrow_R^* l^\# \sigma \rightarrow_{\text{DP}(R)} u^\# \sigma$$

where  $t^\# \in T_\infty^\#$  and  $u^\# \sigma \in T_\infty^\#$ .

(Note that rules in  $R$  do not contain symbols from  $\Omega^\#$ , whereas all roots of terms in  $\text{DP}(R)$  come from  $\Omega^\#$ , so rules from  $R$  can only be applied below the root and rules from  $\text{DP}(R)$  can only be applied at the root.)

## Dependency Pairs

---

Since  $u^\# \sigma$  is again in  $T_\infty^\#$ , we can continue the process in the same way. We obtain:  $R$  is non-terminating if and only if there is an infinite sequence

$$t_1 \xrightarrow*_R t_2 \xrightarrow{\text{DP}(R)} t_3 \xrightarrow*_R t_4 \xrightarrow{\text{DP}(R)} \dots$$

with  $t_i \in T_\infty^\#$  for all  $i$ .

Moreover, if there exists such an infinite sequence, then there exists an infinite sequence in which all DPs that are used are used infinitely often. (If some DP is used only finitely often, we can cut off the initial part of the sequence up to the last occurrence of that DP; the remainder is still an infinite sequence.)

# Dependency Graphs

---

Such infinite sequences correspond to “cycles” in the “dependency graph”:

Dependency graph  $DG(R)$  of a TRS  $R$ :

directed graph

nodes: dependency pairs  $s \rightarrow t \in DP(R)$

edges: from  $s \rightarrow t$  to  $u \rightarrow v$  if there are  $\sigma, \tau$   
such that  $t\sigma \rightarrow_R^* u\tau$ .

# Dependency Graphs

---

Intuitively, we draw an edge between two dependency pairs, if these two dependency pairs can be used after another in an infinite sequence (with some  $R$ -steps in between). While this relation is undecidable in general, there are reasonable overapproximations:

# Dependency Graphs

---

The functions  $\text{cap}$  and  $\text{ren}$  are defined by:

$$\begin{aligned}\text{cap}(x) &= x \\ \text{cap}(f(t_1, \dots, t_n)) &= \begin{cases} y & \text{if } f \in D \\ f(\text{cap}(t_1), \dots, \text{cap}(t_n)) & \text{if } f \in C \cup D^\# \end{cases}\end{aligned}$$

$$\text{ren}(x) = y, \quad y \text{ fresh}$$

$$\text{ren}(f(t_1, \dots, t_n)) = f(\text{ren}(t_1), \dots, \text{ren}(t_n))$$

The overapproximated dependency graph contains an edge from  $s \rightarrow t$  to  $u \rightarrow v$  if  $\text{ren}(\text{cap}(t))$  and  $u$  are unifiable.

# Dependency Graphs

---

A **cycle** in the dependency graph is a non-empty subset  $K \subseteq DP(R)$  such that there is a non-empty path in  $K$  from every DP in  $K$  to every DP in  $K$  (the two DPs may be identical).

Let  $K \subseteq DP(R)$ . An infinite rewrite sequence in  $R \cup K$  of the form

$$t_1 \rightarrow_R^* t_2 \rightarrow_K t_3 \rightarrow_R^* t_4 \rightarrow_K \dots$$

with  $t_i \in T_\infty^\#$  is called  $K$ -minimal, if all rules in  $K$  are used infinitely often.

$R$  is non-terminating if and only if there is a cycle  $K \subseteq DP(R)$  and a  $K$ -minimal infinite rewrite sequence.

## 5.2 Subterm Criterion

---

Our task is to show that there are no  $K$ -minimal infinite rewrite sequences.

Suppose that every dependency pair symbol  $f^\sharp$  in  $K$  has positive arity (i. e., no constants). A **simple projection**  $\pi$  is a mapping  $\pi : \Omega^\sharp \rightarrow \mathbb{N}$  such that  $\pi(f^\sharp) = i \in \{1, \dots, \text{arity}(f^\sharp)\}$ .

We define  $\pi(f^\sharp(t_1, \dots, t_n)) = t_{\pi(f^\sharp)}$ .



## Subterm Criterion

---

Theorem 5.1 (Hirokawa and Middeldorp):

Let  $K$  be a cycle in  $DG(R)$ . If there is a simple projection  $\pi$  for  $K$  such that  $\pi(l) \succeq \pi(r)$  for every  $l \rightarrow r \in K$  and  $\pi(l) \triangleright \pi(r)$  for some  $l \rightarrow r \in K$ , then there are no  $K$ -minimal sequences.

## Subterm Criterion

---

Problem: The number of cycles in  $DG(R)$  can be exponential.

Better method: Analyze strongly connected components (SCCs).

SCC of a graph: maximal subgraph in which there is a non-empty path from every node to every node. (The two nodes can be identical.)<sup>a</sup>

Important property: Every cycle is contained in some SCC.

---

<sup>a</sup>There are several definitions of SCCs that differ in the treatment of edges from a node to itself.

## Subterm Criterion

---

Idea: Search for a simple projection  $\pi$  such that  $\pi(l) \succeq \pi(r)$  for all DPs  $l \rightarrow r$  in the SCC. Delete all DPs in the SCC for which  $\pi(l) \triangleright \pi(r)$  (by the previous theorem, there cannot be any  $K$ -minimal infinite rewrite sequences using these DPs). Then re-compute SCCs for the remaining graph and re-start.

No SCCs left  $\Rightarrow$  no cycles left  $\Rightarrow R$  is terminating.

Example: See Ex. 13 from Hirokawa and Middeldorp.

## 5.3 Reduction Pairs and Argument Filterings

---

Goal: Show the non-existence of  $K$ -minimal infinite rewrite sequences

$$t_1 \rightarrow_R^* u_1 \rightarrow_K t_2 \rightarrow_R^* u_2 \rightarrow_K \dots$$

using well-founded orderings.

We observe that the requirements for the orderings used here are less restrictive than for reduction orderings:

$K$ -rules are only used at the top, so we need stability under substitutions, but compatibility with contexts is unnecessary.

While  $\rightarrow_K$ -steps should be decreasing, for  $\rightarrow_R$ -steps it would be sufficient to show that they are not increasing.

# Reduction Pairs and Argument Filterings

---

This motivates the following definitions:

Rewrite quasi-ordering  $\succsim$ :

reflexive and transitive binary relation, stable under substitutions, compatible with contexts.

Reduction pair  $(\succsim, \succ)$ :

$\succsim$  is a rewrite quasi-ordering.

$\succ$  is a well-founded ordering that is stable under substitutions.

$\succsim$  and  $\succ$  are compatible:  $\succsim \circ \succ \subseteq \succ$  or  $\succ \circ \succsim \subseteq \succ$ .

(In practice,  $\succ$  is almost always the strict part of the quasi-ordering  $\succsim$ .)

# Reduction Pairs and Argument Filterings

---

Clearly, for any reduction ordering  $\succ$ ,  $(\succeq, \succ)$  is a reduction pair. More general reduction pairs can be obtained using argument filterings:

Argument filtering  $\pi$ :

$$\pi : \Omega \cup \Omega^\# \rightarrow \mathbb{N} \cup \text{list of } \mathbb{N}$$

$$\pi(f) = \begin{cases} i \in \{1, \dots, \text{arity}(f)\}, \text{ or} \\ [i_1, \dots, i_k], \text{ where } 1 \leq i_1 < \dots < i_k \leq \text{arity}(f), \\ 0 \leq k \leq \text{arity}(f) \end{cases}$$

# Reduction Pairs and Argument Filterings

---

Extension to terms:

$$\pi(x) = x$$

$$\pi(f(t_1, \dots, t_n)) = \pi(t_i), \text{ if } \pi(f) = i$$

$$\pi(f(t_1, \dots, t_n)) = f'(\pi(t_{i_1}), \dots, \pi(t_{i_k})), \text{ if } \pi(f) = [i_1, \dots, i_k],$$

where  $f'/k$  is a new function symbol.

# Reduction Pairs and Argument Filterings

---

Let  $\succ$  be a reduction ordering, let  $\pi$  be an argument filtering. Define  $s \succ_{\pi} t$  if and only if  $\pi(s) \succ \pi(t)$  and  $s \sim_{\pi} t$  if and only if  $\pi(s) \succeq \pi(t)$ .

Lemma 5.2:

$(\sim_{\pi}, \succ_{\pi})$  is a reduction pair.



## Reduction Pairs and Argument Filterings

---

For interpretation-based orderings (such as polynomial orderings) the idea of “cutting out” certain subterms can be included directly in the definition of the ordering:

# Reduction Pairs and Argument Filterings

---

Reduction pairs by interpretation:

Let  $\mathcal{A}$  be a  $\Sigma$ -algebra;

let  $\succ$  be a well-founded strict partial ordering on its universe.

Assume that all interpretations  $f_{\mathcal{A}}$  of function symbols are **weakly monotone**, i. e.,  $a_i \succeq b_i$  implies  $f(a_1, \dots, a_n) \succeq f(b_1, \dots, b_n)$  for all  $a_i, b_i \in U_{\mathcal{A}}$ .

Define  $s \succsim_{\mathcal{A}} t$  if and only if  $\mathcal{A}(\beta)(s) \succeq \mathcal{A}(\beta)(t)$  for all assignments  $\beta : X \rightarrow U_{\mathcal{A}}$ ; define  $s \succ_{\mathcal{A}} t$  if and only if  $\mathcal{A}(\beta)(s) \succ \mathcal{A}(\beta)(t)$  for all assignments  $\beta : X \rightarrow U_{\mathcal{A}}$ .

Then  $(\succsim_{\mathcal{A}}, \succ_{\mathcal{A}})$  is a reduction pair.

## Reduction Pairs and Argument Filterings

---

For polynomial orderings, this definition permits interpretations of function symbols where some variable does not occur at all (e. g.,  $P_f(X_1, X_2) = 2X_1 + 1$  for a *binary* function symbol).

It is no longer required that *every* variable must occur with some positive coefficient.

# Reduction Pairs and Argument Filterings

---

Theorem 5.3 (Arts and Giesl):

Let  $K$  be a cycle in the dependency graph of the TRS  $R$ . If there is a reduction pair  $(\succsim, \succ)$  such that

- $l \succsim r$  for all  $l \rightarrow r \in R$ ,
- $l \succsim r$  or  $l \succ r$  for all  $l \rightarrow r \in K$ ,
- $l \succ r$  for at least one  $l \rightarrow r \in K$ ,

then there is no  $K$ -minimal infinite sequence.

## Reduction Pairs and Argument Filterings

---

The idea can be extended to SCCs in the same way as for the subterm criterion:

Search for a reduction pair  $(\succsim, \succ)$  such that  $l \succsim r$  for all  $l \rightarrow r \in R$  and  $l \succsim r$  or  $l \succ r$  for all DPs  $l \rightarrow r$  in the SCC. Delete all DPs in the SCC for which  $l \succ r$ .

Then re-compute SCCs for the remaining graph and re-start.

# Reduction Pairs and Argument Filterings

---

Example: Consider the following TRS  $R$  from [Arts and Giesl]:

$$\mathit{minus}(x, 0) \rightarrow x \quad (1)$$

$$\mathit{minus}(s(x), s(y)) \rightarrow \mathit{minus}(x, y) \quad (2)$$

$$\mathit{quot}(0, s(y)) \rightarrow 0 \quad (3)$$

$$\mathit{quot}(s(x), s(y)) \rightarrow s(\mathit{quot}(\mathit{minus}(x, y), s(y))) \quad (4)$$

( $R$  is not contained in any simplification ordering, since the left-hand side of rule (4) is embedded in the right-hand side after instantiating  $y$  by  $s(x)$ .)

# Reduction Pairs and Argument Filterings

---

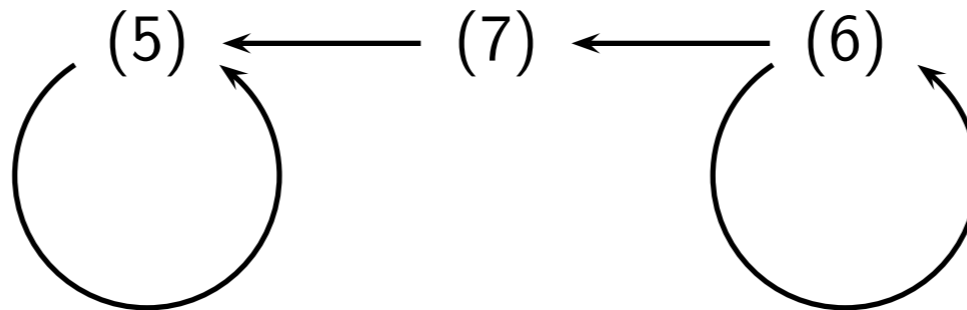
$R$  has three dependency pairs:

$$\mathit{minus}^\sharp(s(x), s(y)) \rightarrow \mathit{minus}^\sharp(x, y) \quad (5)$$

$$\mathit{quot}^\sharp(s(x), s(y)) \rightarrow \mathit{quot}^\sharp(\mathit{minus}(x, y), s(y)) \quad (6)$$

$$\mathit{quot}^\sharp(s(x), s(y)) \rightarrow \mathit{minus}^\sharp(x, y) \quad (7)$$

The dependency graph of  $R$  is



## Reduction Pairs and Argument Filterings

---

There are exactly two SCCs (and also two cycles).

The cycle at (5) can be handled using the subterm criterion with  $\pi(\mathit{minus}^\sharp) = 1$ .

For the cycle at (6) we can use an argument filtering  $\pi$  that maps  $\mathit{minus}$  to 1 and leaves all other function symbols unchanged (that is,  $\pi(g) = [1, \dots, \text{arity}(g)]$  for every  $g$  different from  $\mathit{minus}$ .) After applying the argument filtering, we compare left and right-hand sides using an LPO with precedence  $\mathit{quot} > s$  (the precedence of other symbols is irrelevant). We obtain  $l \succ r$  for (6) and  $l \approx r$  for (1), (2), (3), (4), so the previous theorem can be applied.



## Reduction Pairs and Argument Filterings

---

Alternatively, we can handle the cycle at (5) with a polynomial interpretation with  $P_{minus\#}(X_1, X_2) = X_1$ ,  $P_s(X_1) = X_1 + 1$ ,  $P_{minus}(X_1, X_2) = X_1$ ,  $P_{quot}(X_1, X_2) = X_1$ ,  $P_0 = 1$ . We obtain  $l \succ r$  for (5) and  $l \succsim r$  for (1), (2), (3), (4), so the previous theorem can be applied.

It remains to handle the cycle at (6). We choose a polynomial interpretation with  $P_{quot\#}(X_1, X_2) = X_1$ ,  $P_s(X_1) = X_1 + 1$ ,  $P_{minus}(X_1, X_2) = X_1$ ,  $P_{quot}(X_1, X_2) = X_1$ ,  $P_0 = 1$ . We obtain  $l \succ r$  for (6) and  $l \succsim r$  for (1), (2), (3), (4), so the previous theorem can be applied again.

# DP Processors

---

The methods described so far are particular cases of **DP processors**:

A DP processor

$$\frac{(G, R)}{(G_1, R_1), \dots, (G_n, R_n)}$$

takes a graph  $G$  and a TRS  $R$  as input and produces a set of pairs consisting of a graph and a TRS.

It is sound and complete if there are  $K$ -minimal infinite sequences for  $G$  and  $R$  if and only if there are  $K$ -minimal infinite sequences for at least one of the pairs  $(G_i, R_i)$ .

# DP Processors

---

Examples:

$$\frac{(G, R)}{(SCC_1, R), \dots, (SCC_n, R)}$$

where  $SCC_1, \dots, SCC_n$  are the strongly conn. components of  $G$ .

$$\frac{(G, R)}{(G \setminus N, R)}$$

if there is an SCC of  $G$  and a simple projection  $\pi$  such that  $\pi(l) \succeq \pi(r)$  for all DPs  $l \rightarrow r$  in the SCC, and  $N$  is the set of DPs of the SCC for which  $\pi(l) \triangleright \pi(r)$ .

(and analogously for reduction pairs)

# Innermost Termination

---

The dependency method can also be used for proving termination of **innermost rewriting**:  $s \xrightarrow{i} \rightarrow_R t$  if  $s \rightarrow_R t$  at position  $p$  and no rule of  $R$  can be applied at a position strictly below  $p$ .

(DP processors for innermost termination are more powerful than for ordinary termination, and for program analysis, innermost termination is usually sufficient.)

## Part 6: Implementing Saturation Procedures

---

Problem:

Refutational completeness is nice in theory, but . . .

. . . it guarantees only that proofs will be found eventually, not that they will be found quickly.

Even though orderings and selection functions reduce the number of possible inferences, the search space problem is enormous.

First-order provers “look for a needle in a haystack”:

It may be necessary to make some millions of inferences to find a proof that is only a few dozens of steps long.

# Coping with Large Sets of Formulas

---

Consequently:

- We must deal with large sets of formulas.
- We must use efficient techniques to find formulas that can be used as partners in an inference.
- We must simplify/eliminate as many formulas as possible.
- We must use efficient techniques to check whether a formula can be simplified/eliminated.

# Coping with Large Sets of Formulas

---

Note:

Often there are several competing implementation techniques.

Design decisions are not independent of each other.

Design decisions are not independent of the particular class of problems we want to solve.

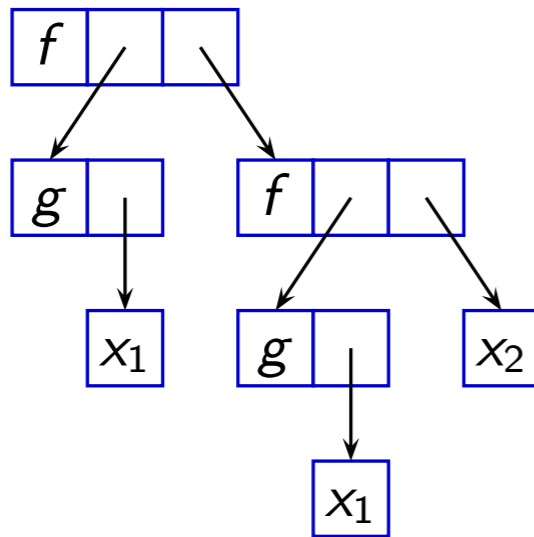
(FOL without equality/FOL with equality/unit equations,  
size of the signature,  
special algebraic properties like AC, etc.)

## 6.1 Term Representations

---

The obvious data structure for terms: Trees

$$f(g(x_1), f(g(x_1), x_2))$$



optionally: (full) sharing

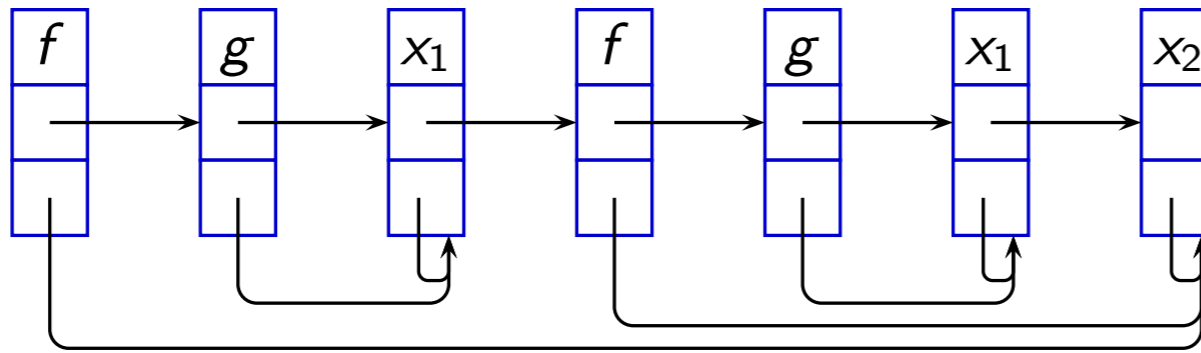


# Term Representations

---

An alternative: Flatterms

$$f(g(x_1), f(g(x_1), x_2))$$



need more memory;

but: better suited for preorder term traversal  
and easier memory management.

## 6.2 Index Data Structures

---

Problem:

For a term  $t$ , we want to find all terms  $s$  such that

- $s$  is an instance of  $t$ ,
- $s$  is a generalization of  $t$  (i. e.,  $t$  is an instance of  $s$ ),
- $s$  and  $t$  are unifiable,
- $s$  is a generalization of some subterm of  $t$ ,
- ...

# Index Data Structures

---

Requirements:

fast insertion,

fast deletion,

fast retrieval,

small memory consumption.

Note: In applications like functional or logic programming, the requirements are different (insertion and deletion are much less important).

# Index Data Structures

---

Many different approaches:

- Path indexing
- Discrimination trees
- Substitution trees
- Context trees
- Feature vector indexing
- ...

# Index Data Structures

---

Perfect filtering:

The indexing technique returns exactly those terms satisfying the query.

Imperfect filtering:

The indexing technique returns some superset of the set of all terms satisfying the query.

Retrieval operations must be followed by an additional check, but the index can often be implemented more efficiently.

Frequently: All occurrences of variables are treated as different variables.

# Path Indexing

---

Path indexing:

Paths of terms are encoded in a trie (“retrieval tree”).

A star \* represents arbitrary variables.

Example: Paths of  $f(g(*, b), *)$ :  $f.1.g.1.*$

$f.1.g.2.b$

$f.2.*$

Each leaf of the trie contains the set of (pointers to) all terms that contain the respective path.

## Path Indexing

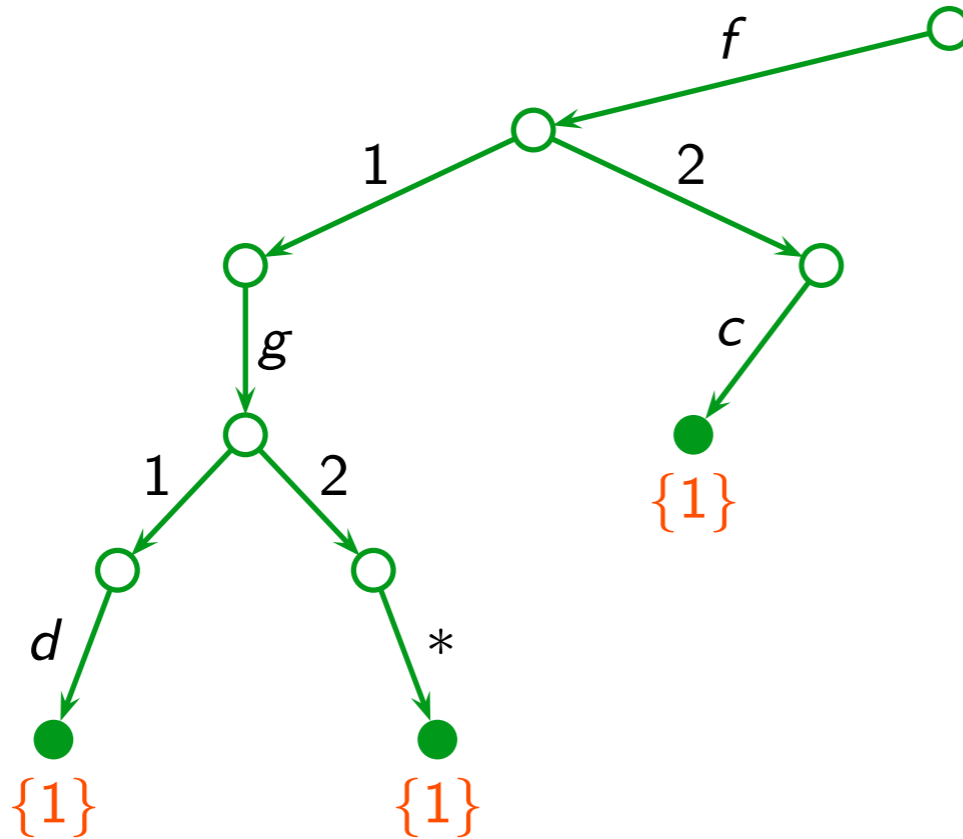
---

Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$

# Path Indexing

---

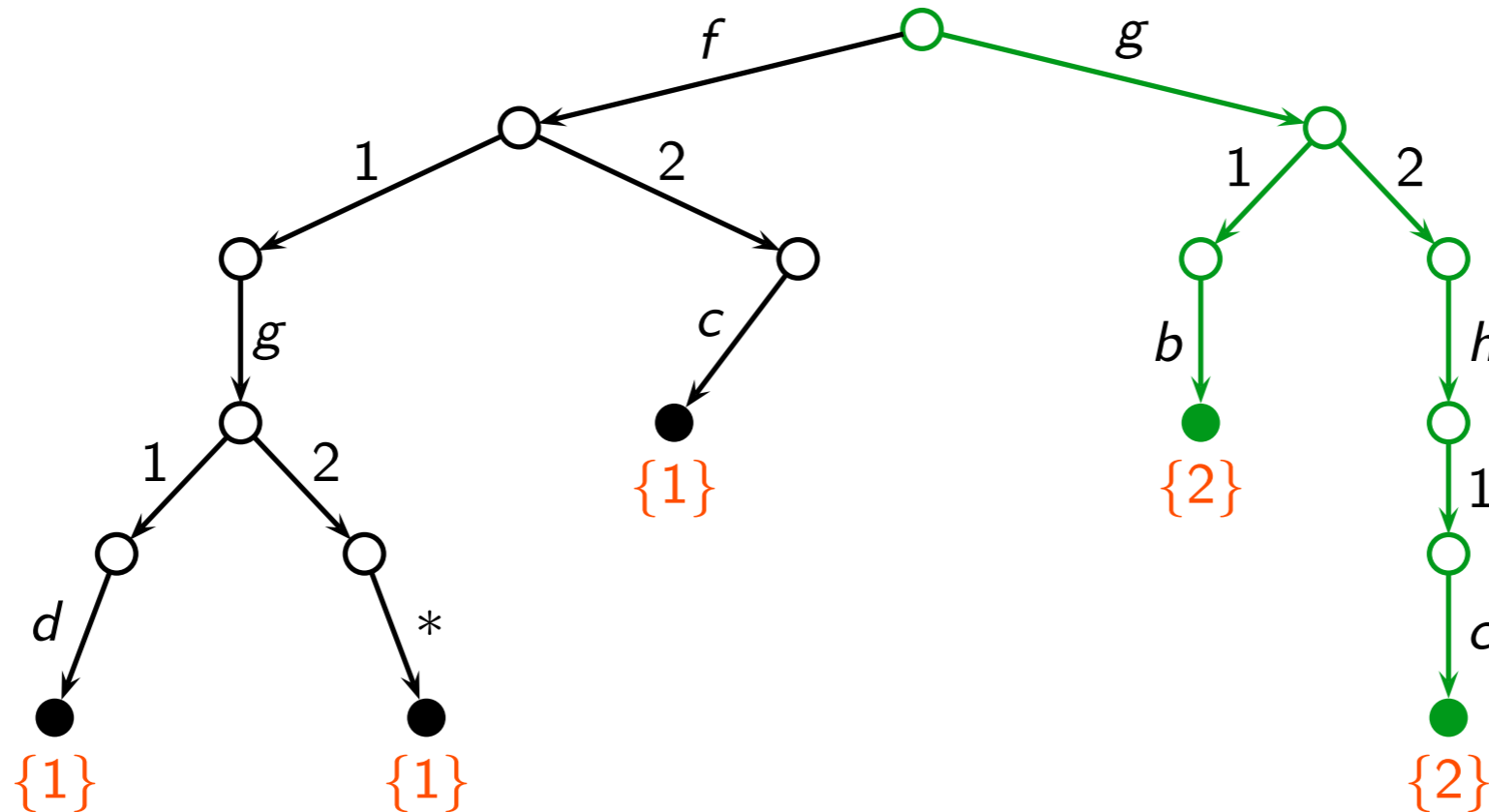
Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$





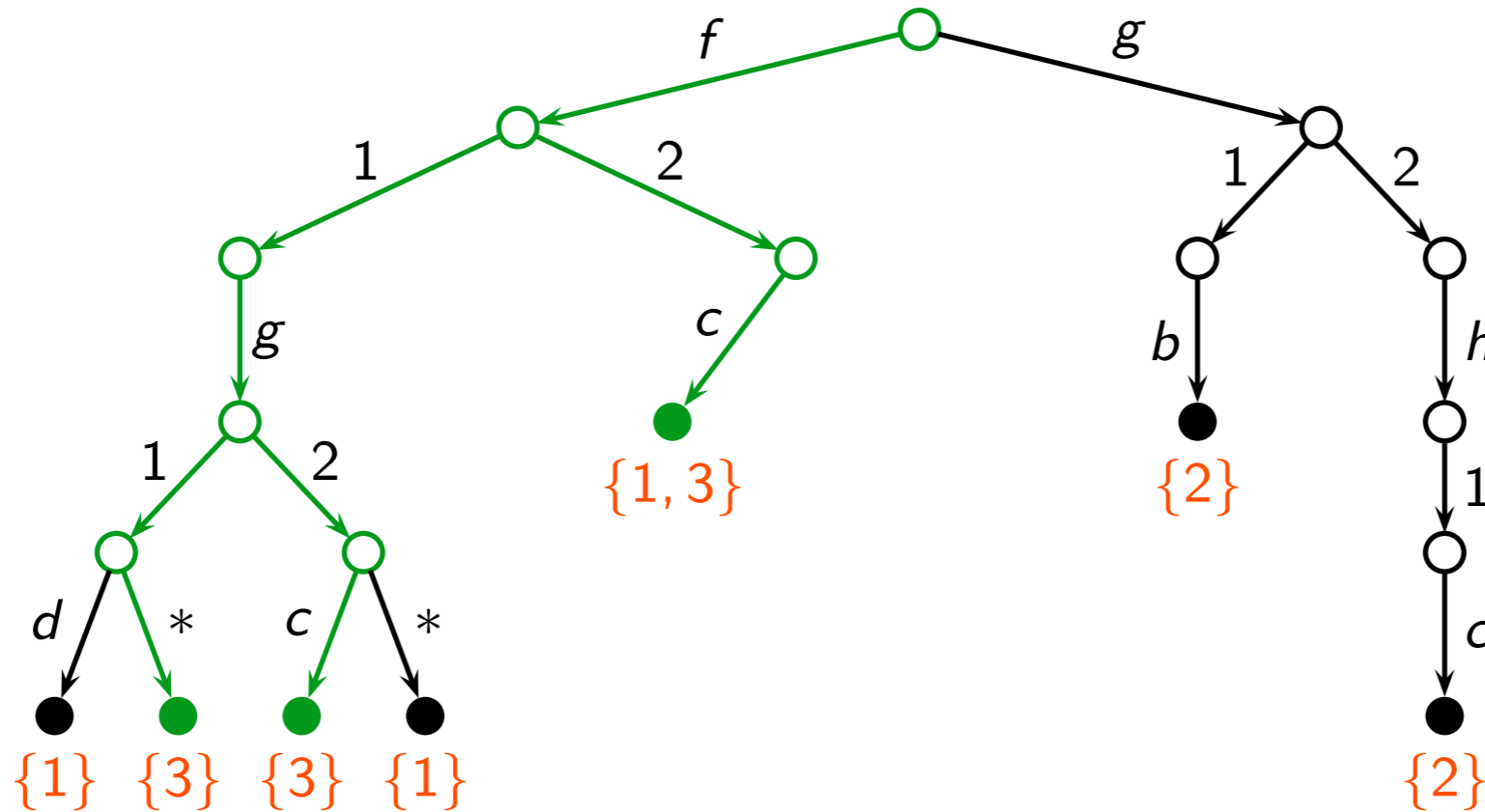
# Path Indexing

Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



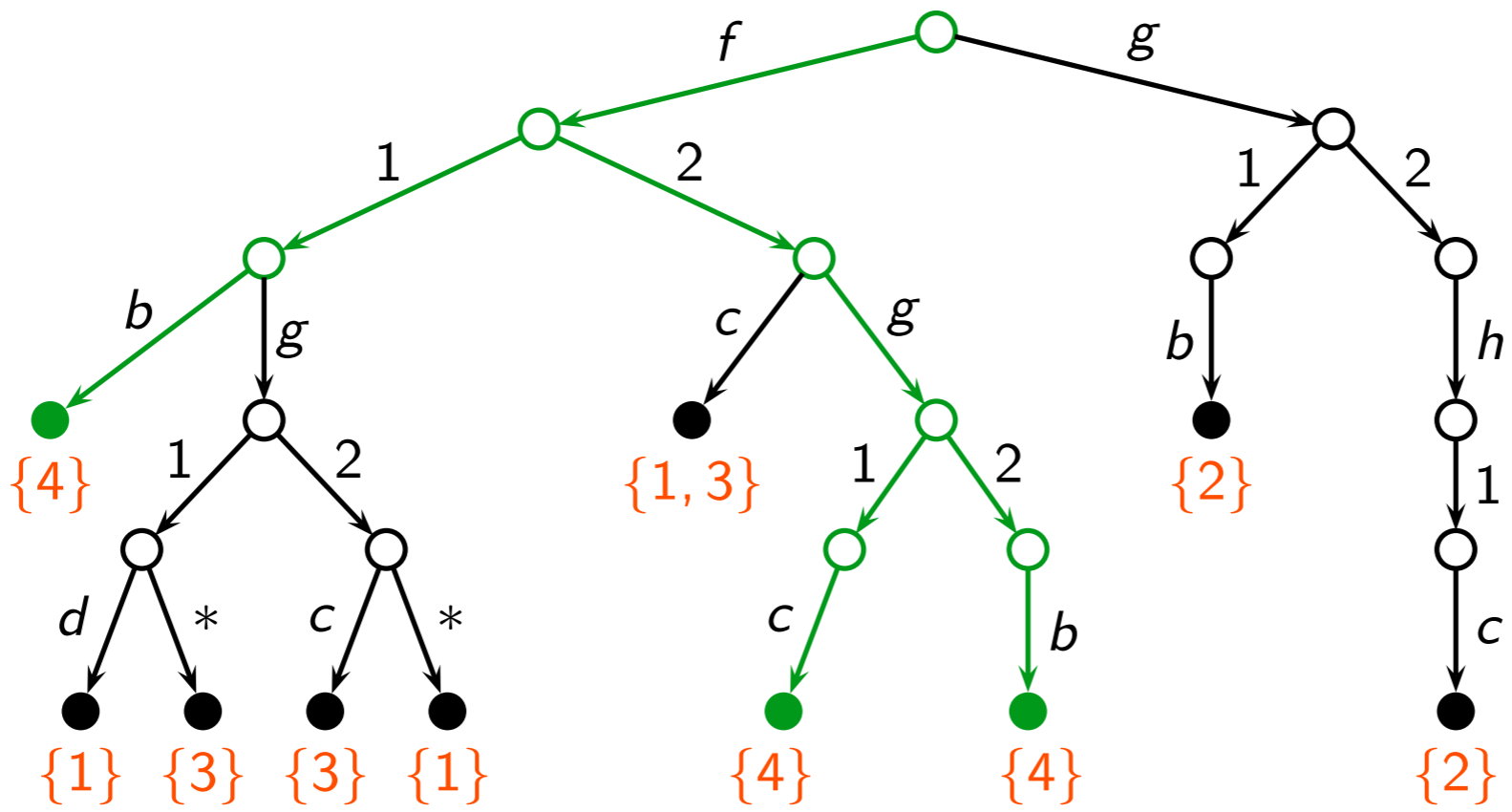
# Path Indexing

Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



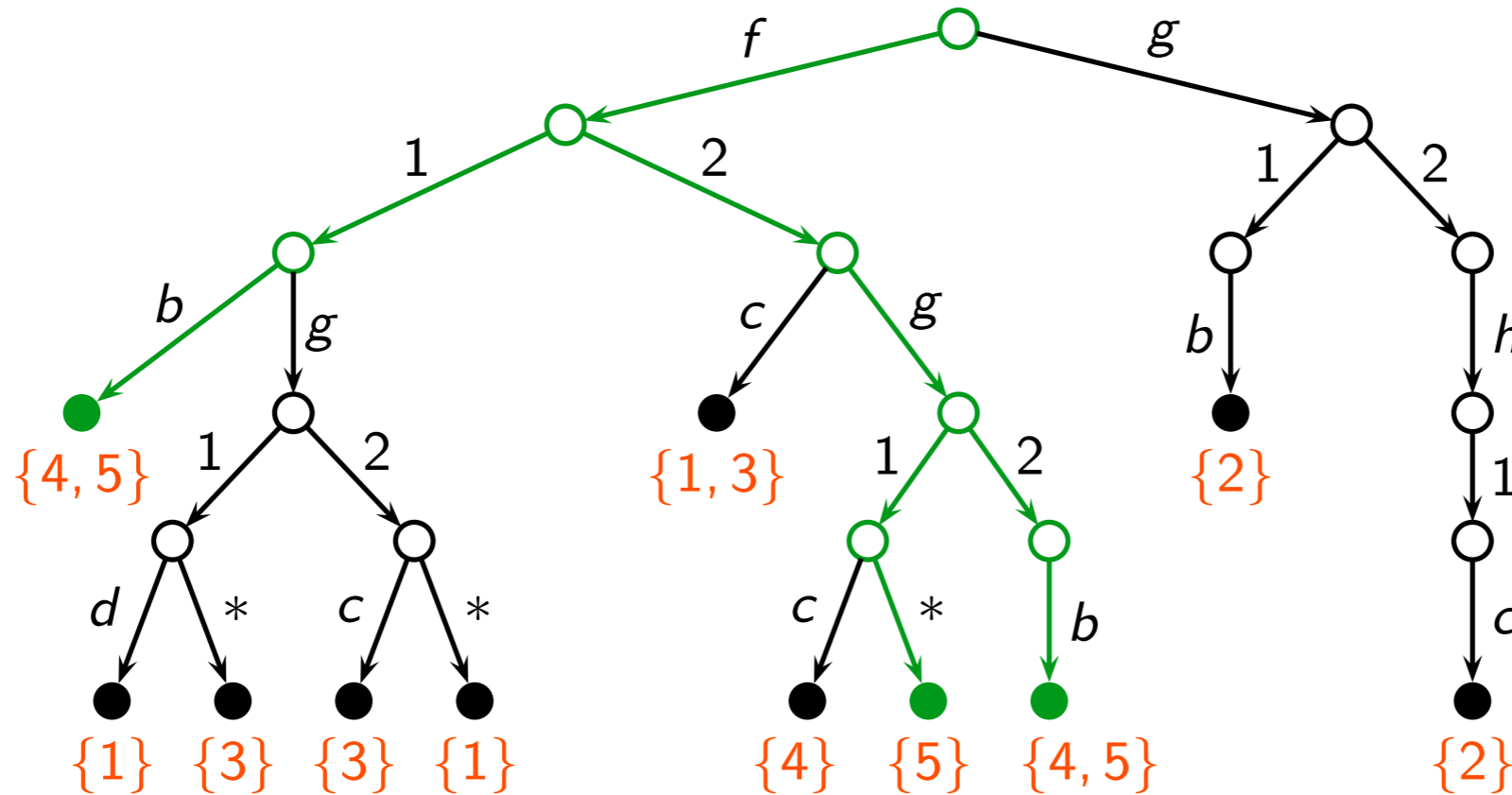
# Path Indexing

Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



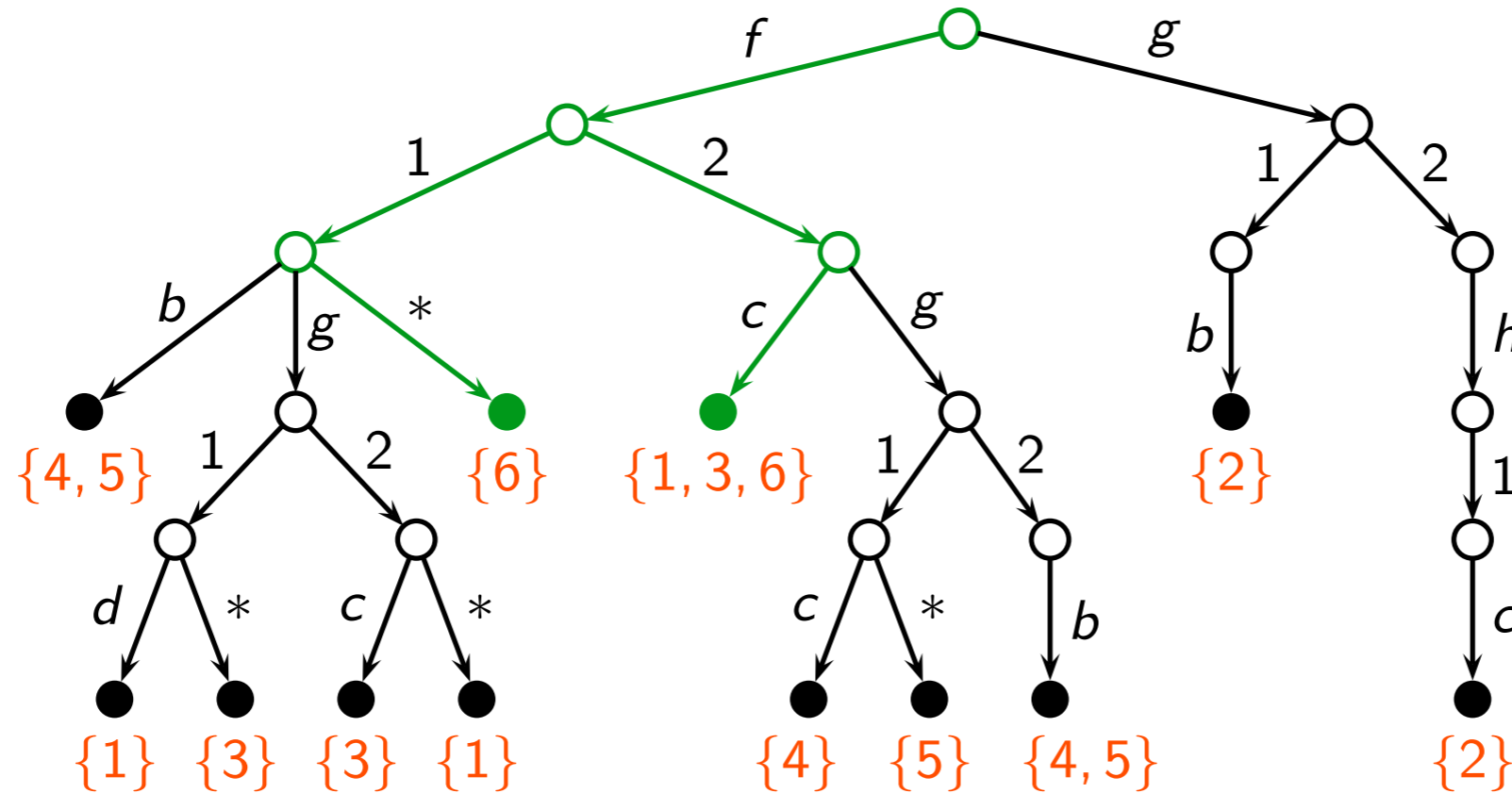
# Path Indexing

Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



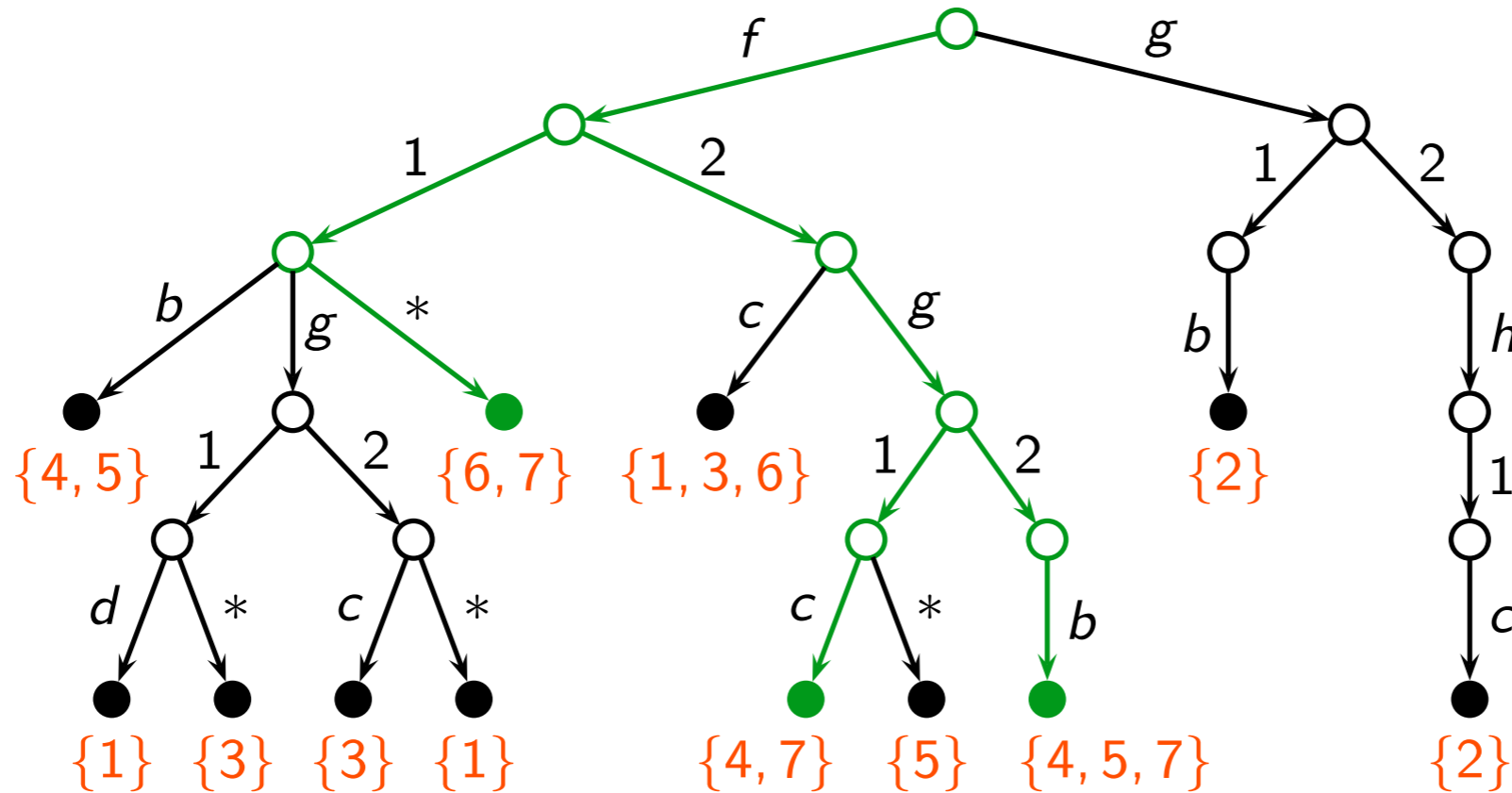
# Path Indexing

Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



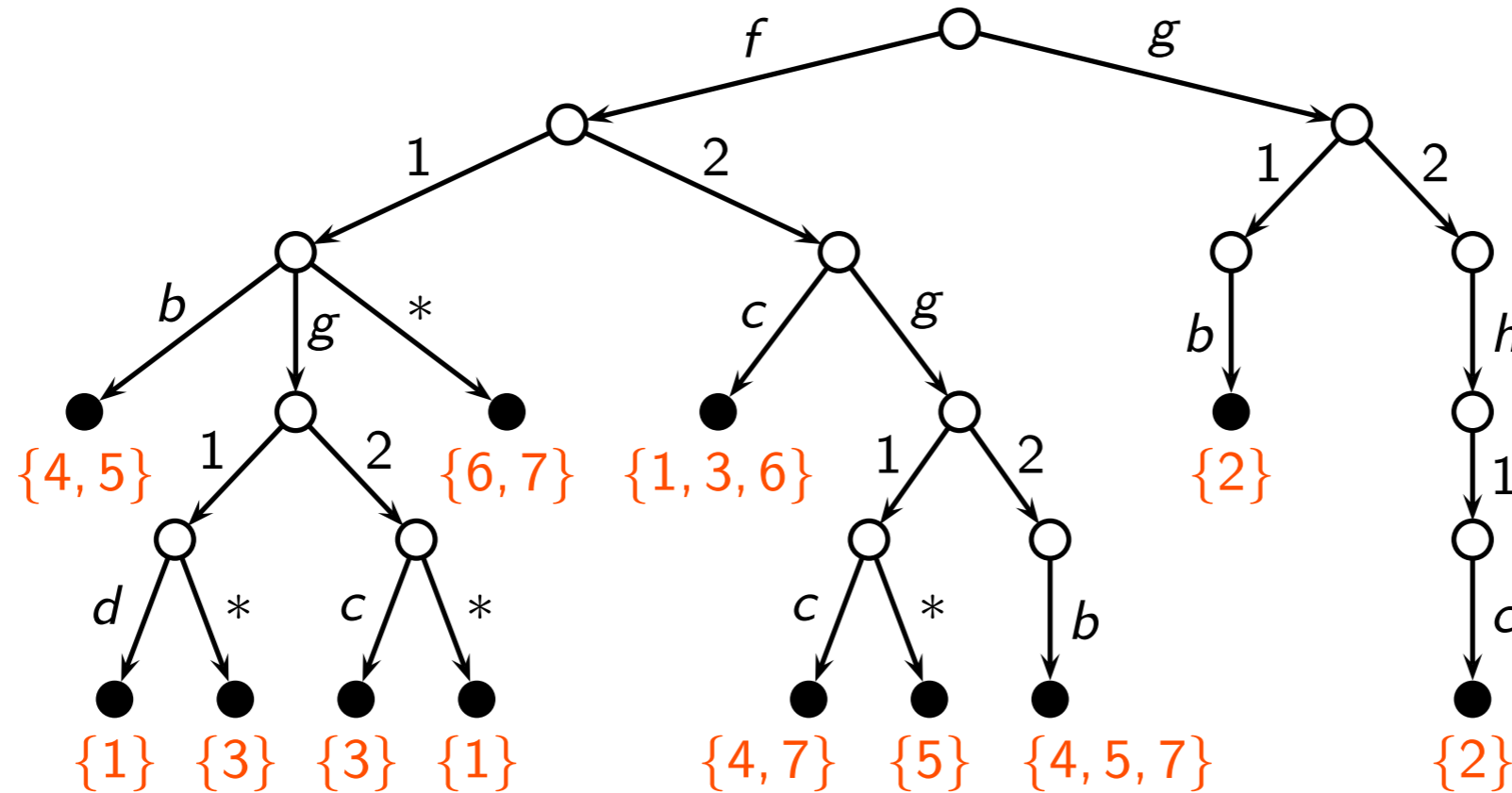
# Path Indexing

Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



# Path Indexing

Example: Path index for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



# Path Indexing

---

## Advantages:

Uses little space.

No backtracking for retrieval.

Efficient insertion and deletion.

Good for finding instances,  
also usable for finding generalizations.

## Disadvantages:

Retrieval requires combining intermediate results for all paths.



# Discrimination Trees

---

Discrimination trees:

Preorder traversals of terms are encoded in a trie.

A star  $*$  represents arbitrary variables.

Example: String of  $f(g(*, b), *)$ :  $f.g.*.b.*$

Each leaf of the trie contains (a pointer to) the term that is represented by the path.

# Discrimination Trees

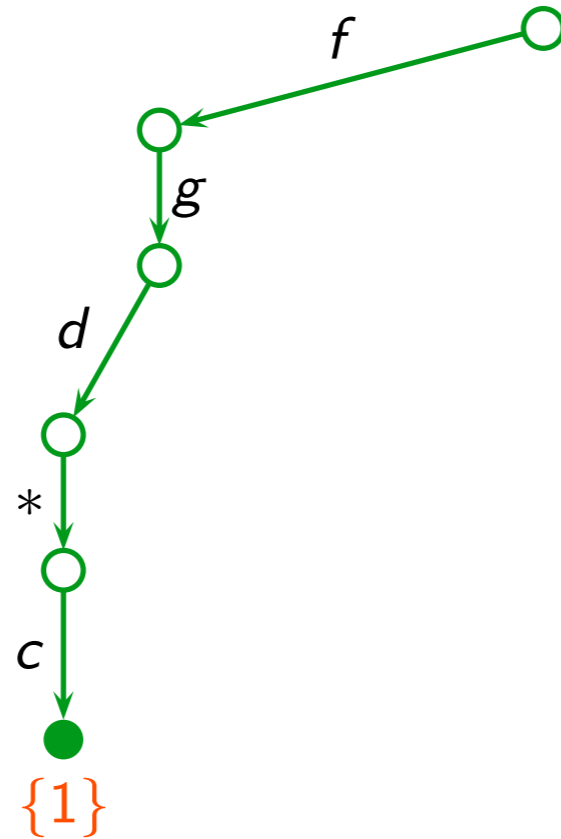
---

Example: Discrimination tree for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$

# Discrimination Trees

---

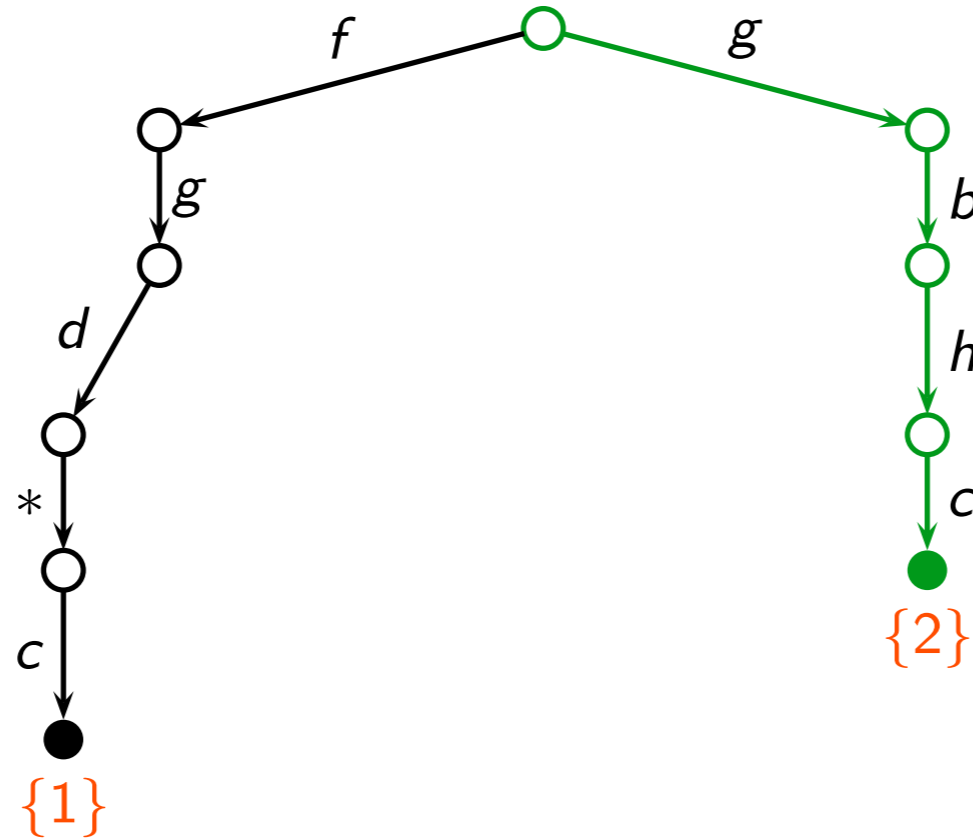
Example: Discrimination tree for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



# Discrimination Trees

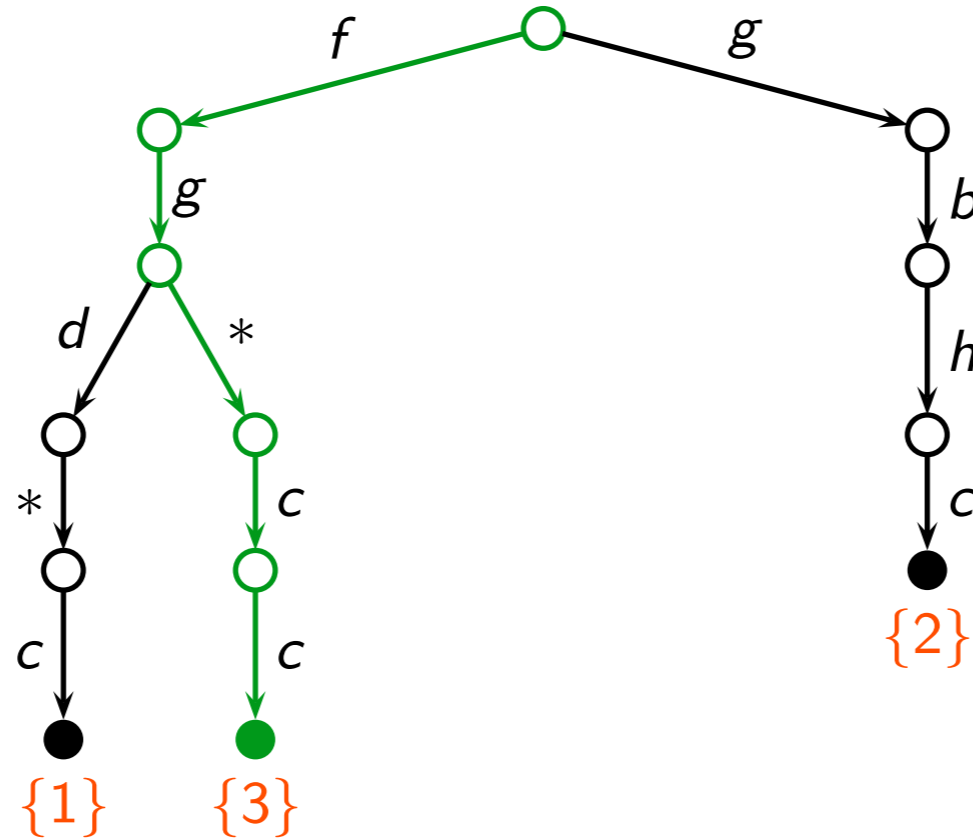
---

Example: Discrimination tree for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



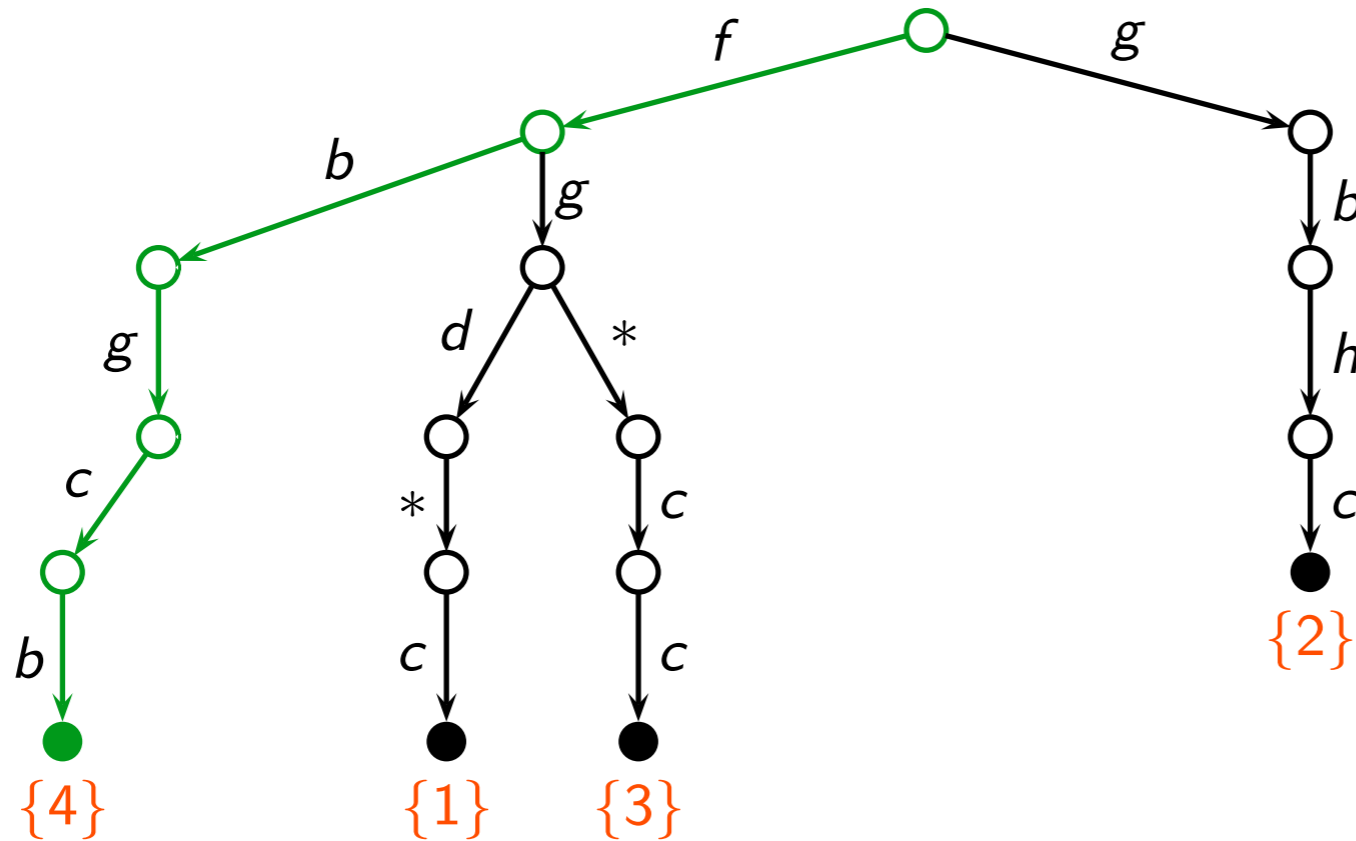
# Discrimination Trees

Example: Discrimination tree for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



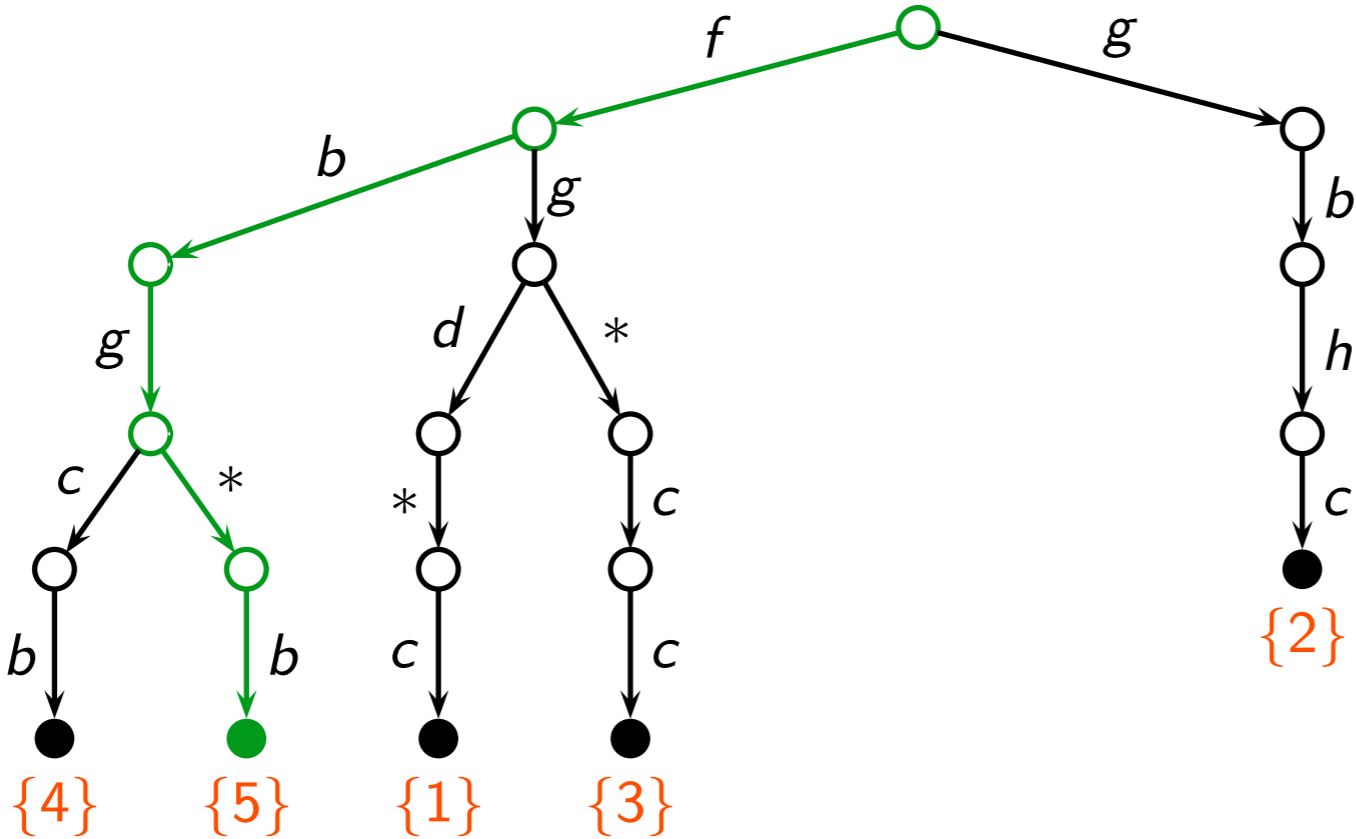
# Discrimination Trees

Example: Discrimination tree for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



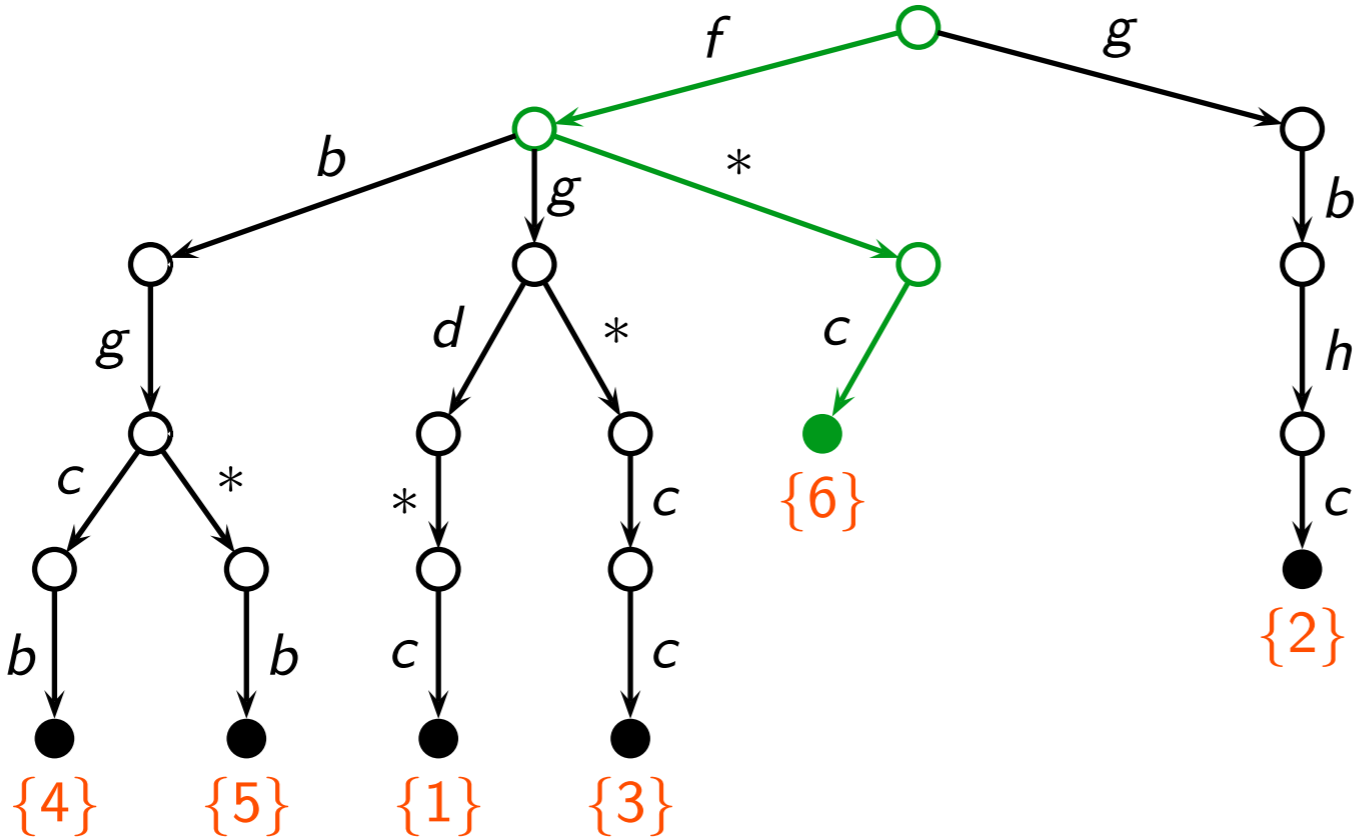
# Discrimination Trees

Example: Discrimination tree for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



# Discrimination Trees

Example: Discrimination tree for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$

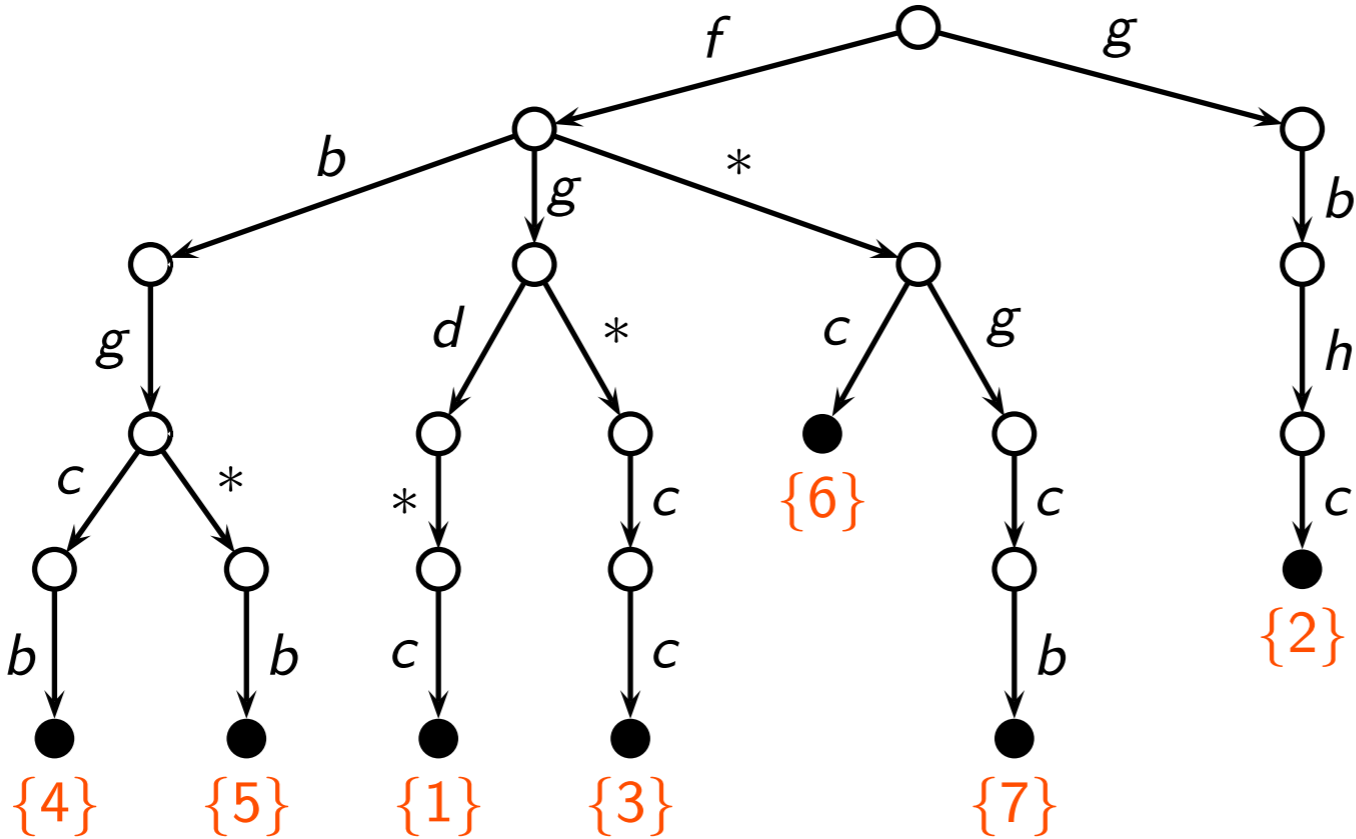






# Discrimination Trees

Example: Discrimination tree for  $\{f(g(d, *), c), g(b, h(c)), f(g(*, c), c), f(b, g(c, b)), f(b, g(*, b)), f(*, c), f(*, g(c, b))\}$



# Discrimination Trees

---

## Advantages:

Each leaf yields one term, hence retrieval does not require intersections of intermediate results for all paths.

Good for finding generalizations,  
not so good for finding instances.

## Disadvantages:

Uses more storage than path indexing (due to less sharing).

Uses still more storage, if jump lists are maintained to speed up the search for instances or unifiable terms.

# Feature Vector Indexing

---

Goal:

$C'$  is subsumed by  $C$  if  $C' = C\sigma \vee D$ .

Find all clauses  $C'$  for a given  $C$  or vice versa.

# Feature Vector Indexing

---

If  $C'$  is subsumed by  $C$ , then

- $C'$  contains at least as many literals as  $C$ .
- $C'$  contains at least as many positive literals as  $C$ .
- $C'$  contains at least as many negative literals as  $C$ .
- $C'$  contains at least as many function symbols as  $C$ .
- $C'$  contains at least as many occurrences of  $f$  as  $C$ .
- $C'$  contains at least as many occurrences of  $f$  in negative literals as  $C$ .
- the deepest occurrence of  $f$  in  $C'$  is at least as deep as in  $C$ .
- ...

# Feature Vector Indexing

---

Idea:

Select a list of these “features” .

Compute the “feature vector” (a list of natural numbers)  
for each clause and store it in a trie.

When searching for a subsuming clause:

Traverse the trie, check all clauses for which all features are smaller or equal. (Stop if a subsuming clause is found.)

When searching for subsumed clauses:

Traverse the trie, check all clauses for which all features are larger or equal.

# Feature Vector Indexing

---

## Advantages:

Works on the clause level, rather than on the term level.

Specialized for subsumption testing.

## Disadvantages:

Needs to be complemented by other index structure for other operations.

## Literature

---

R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov:

Term Indexing, Ch. 26 in Robinson and Voronkov (eds.), *Handbook of Automated Reasoning, Vol. II*, Elsevier, 2001.

Stephan Schulz:

Simple and Efficient Clause Subsumption with Feature Vector Indexing, in Bonacina and Stickel (eds.), *Automated Reasoning and Mathematics*, LNCS 7788, Springer, 2013.

Christoph Weidenbach:

Combining Superposition, Sorts and Splitting, Ch. 27 in Robinson and Voronkov (eds.), *Handbook of Automated Reasoning, Vol. II*, Elsevier, 2001.



## Part 7: Outlook

---

Further topics in automated reasoning.

## 7.1 Satisfiability Modulo Theories (SMT)

---

CDCL checks satisfiability of propositional formulas.

CDCL can also be used for ground first-order formulas without equality:

Ground first-order atoms are treated like propositional variables.

Truth values of  $P(a)$ ,  $Q(a)$ ,  $Q(f(a))$  are independent.

# Satisfiability Modulo Theories (SMT)

---

For ground formulas with equality, independence is lost:

If  $b \approx c$  is true, then  $f(b) \approx f(c)$  must also be true.

Similarly for other theories, e. g. linear arithmetic:

$b > 5$  implies  $b > 3$ .

We can still use CDCL, but we must combine it with a decision procedure for the theory part  $T$ :

$M \models_T C$ :  $M$  and the theory axioms  $T$  entail  $C$ .

# Satisfiability Modulo Theories (SMT)

---

New CDCL rules:

$T$ -Propagate:

$$M \parallel N \Rightarrow_{\text{CDCL}(T)} M \ L \parallel N$$

if  $M \models_T L$

where  $L$  is undefined in  $M$  and  $L$  or  $\bar{L}$  occurs in  $N$ .

$T$ -Learn:

$$M \parallel N \Rightarrow_{\text{CDCL}(T)} M \parallel N \cup \{C\}$$

if  $N \models_T C$  and each atom of  $C$  occurs in  $N$  or  $M$ .

# Satisfiability Modulo Theories (SMT)

---

$T$ -Backjump:

$$M \ L^d \ M' \parallel N \cup \{C\} \Rightarrow_{\text{CDCL}(T)} M \ L' \parallel N \cup \{C\}$$

if  $M \ L^d \ M' \models \neg C$

and there is some “backjump clause”  $C' \vee L'$  such that

$N \cup \{C\} \models_T C' \vee L'$  and  $M \models \neg C'$ ,

$L'$  is undefined in  $M$ , and

$L'$  or  $\overline{L'}$  occurs in  $N$  or in  $M \ L^d \ M'$ .

## 7.2 Sorted Logics

---

So far, we have considered only unsorted first-order logic.

In practice, one often considers many-sorted logics:

*read/2* becomes  $read : array \times nat \rightarrow data$ .

*write/3* becomes  $write : array \times nat \times data \rightarrow array$ .

Variables:  $x : data$

Only one declaration per function/predicate/variable symbol.

All terms, atoms, substitutions must be well-sorted.

# Sorted Logics

---

Algebras:

Instead of universe  $U_{\mathcal{A}}$ , one set per sort:  $array_{\mathcal{A}}$ ,  $nat_{\mathcal{A}}$ .

Interpretations of function and predicate symbols correspond to their declarations:

$$read_{\mathcal{A}} : array_{\mathcal{A}} \times nat_{\mathcal{A}} \rightarrow data_{\mathcal{A}}$$

# Sorted Logics

---

Proof theory, calculi, etc.:

Essentially as in the unsorted case.

More difficult:

Subsorts

Overloading



## 7.3 Splitting

---

Tableau-like rule within resolution to eliminate variable-disjoint (positive) disjunctions:

$$\frac{N \cup \{C_1 \vee C_2\}}{N \cup \{C_1\} \mid N \cup \{C_2\}}$$

if  $\text{var}(C_1) \cap \text{var}(C_2) = \emptyset$ .

Split clauses are smaller and more likely to be usable for simplification.

Splitting tree is explored using intelligent backtracking.

# Splitting

---

Improvement:

Use a CDCL solver to manage the selection of split clauses.

⇒ AVATAR.

## 7.4 Integrating Theories into Resolution

---

Certain kinds of axioms are  
important in practice,  
but difficult for theorem provers.

Most important case: equality

but also: orderings, (associativity and) commutativity, . . .

# Integrating Theories into Resolution

---

Idea: Combine ordered resolution and critical pair computation.

Superposition (ground case):

$$\frac{D' \vee t \approx t' \quad C' \vee s[t] \approx s'}{D' \vee C' \vee s[t'] \approx s'}$$

Superposition (non-ground case):

$$\frac{D' \vee t \approx t' \quad C' \vee s[u] \approx s'}{(D' \vee C' \vee s[t'] \approx s')\sigma}$$

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

# Integrating Theories into Resolution

---

Advantages:

No variable overlaps (as in KB-completion).

Stronger ordering restrictions:

Only overlaps of (strictly) maximal sides of (strictly) maximal literals are required.

Stronger redundancy criteria.

# Integrating Theories into Resolution

---

Similarly for orderings:

Ordered chaining:

$$\frac{D' \vee t' < t \quad C' \vee s < s'}{(D' \vee C' \vee t' < s')\sigma}$$

where  $\sigma$  is a most general unifier of  $t$  and  $s$ .

# Integrating Theories into Resolution

---

Integrating other theories:

Black box:

Use external decision procedure.

Easy, but works only under certain restrictions.

White box:

Integrate using specialized inference rules and theory unification.

Hard work.

Often: integrating more theory axioms is better.

## 7.5 Higher-Order Logics

---

What's new if we switch to higher-order logics?

Applied variables:  $x\ b$ .

Partially applied functions: *times*  $z$ .

Lambda-expressions with  $\alpha\beta\eta$ -conversion:

$$(\lambda x. f\ (x\ b)\ c)\ (\lambda y. d) = f\ d\ c.$$

Embedded booleans:  $(\lambda x. \text{if } x \text{ then } b \text{ else } c)\ (p \vee q)$



# Higher-Order Logics

---

Problems:

Orderings cannot have all desired compatibility properties.

⇒ additional inferences.

Most general unifiers need not exist anymore.

⇒ interleave enumeration of unifiers and computation of inferences.

CNF transformation by preprocessing is no longer sufficient.

⇒ need calculus with embedded clausification.

The End