Automated Reasoning I

Uwe Waldmann

Winter Term 2017/2018

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 3×3 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 $P^d_{i,j}$ with $d, i, j \in \{1, \ldots, 9\}$ to encode the problem:

$P^d_{i,j}$=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 $P^d_{i,j}$ with $d, i, j \in \{1, \ldots, 9\}$ to encode the problem:

$P^d_{i,j}$=true iff the value of square i, j is d

For example:

$P^8_{5,3} = true$

Coding Sudoku in Boolean Logic

- Concrete values result in formulas $P^d_{i,j}$
- For every square (i,j) we generate $P^1_{i,j} \lor \ldots \lor P^9_{i,j}$
- For every square (i,j) and pair of values $d < d'$ we generate $\neg P^d_{i,j} \lor \neg P^{d'}_{i,j}$
- For every value d and row i we generate $P^d_{i,1} \lor \ldots \lor P^d_{i,9}$
  (Analogously for columns and 3×3 boxes)
- For every value d, row i, and pair of columns $j < j'$
  we generate $\neg P^d_{i,j} \lor \neg P^d_{i,j'}$
  (Analogously for columns and 3×3 boxes)

Coding Sudoku in Boolean Logic

Every assignment to the variables $P^d_{i,j}$
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).

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

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

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

$$x + 0 = x \tag{1}$$

Introductory Example 2: Equations

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

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

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

Introductory Example 2: Equations

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

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

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

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

Introductory Example 2: Equations

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

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

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

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

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

Introductory Example 2: Equations

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

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

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

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

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y+z) = (x+y)+z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

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

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x + y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

$$\begin{tikzcd}
\frac{a}{a+1} \arrow[r] \arrow[dr] & \frac{a+0}{a+1} \\
& \frac{a}{a+1}+0
\end{tikzcd}$$

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x + y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

$$\begin{tikzcd} \frac{a}{a+1} \arrow[r] \arrow[dr] \arrow[ddr] & \frac{a+0}{a+1} \\ & \frac{a}{a+1}+0 \\ & \frac{a}{a+(1+0)} \end{tikzcd}$$

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

$$
\begin{tikzcd}
\frac{a}{a+1} \arrow[r] \arrow[dr] \arrow[ddr] \arrow[dddr] & \frac{a+0}{a+1} \\
& \frac{a}{a+1}+0 \\
& \frac{a}{a+(1+0)} \\
& \frac{a}{a+\frac{a+2}{a+2}}
\end{tikzcd}
$$

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

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

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

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

$$\frac{a}{a+\frac{a+2}{a+2}}$$

$$\vdots$$

$$x+0=x \tag{1}$$

$$x+(-x)=0 \tag{2}$$

$$x+(y+z)=(x+y)+z \tag{3}$$

$$\frac{x}{z}+\frac{y}{z}=\frac{x+y}{z} \tag{4}$$

$$\frac{x}{x}=1 \tag{5}$$

Introductory Example 2: Equations

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

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

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

$$\begin{tikzcd} 1+\frac{-1}{a+1} \arrow[r] \arrow[dr] \arrow[ddr] & \frac{a+1}{a+1}+\frac{-1}{a+1} \\ & \frac{a}{a}+\frac{-1}{a+1} \\ & 1+\frac{-1}{a+\frac{a}{a}} \end{tikzcd}$$

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x + y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

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

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

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

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

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

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

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

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

$$\vdots$$

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

Unrestricted application of equations leads to

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

$\Rightarrow$ 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”.

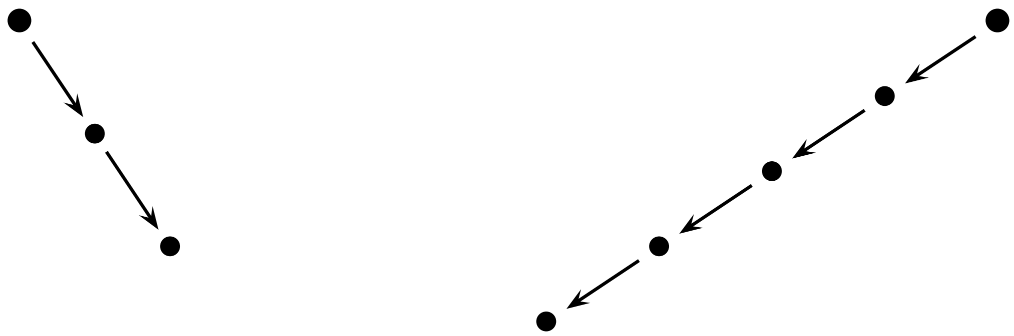
Start from both sides:

Introductory Example 2: Equations

A better approach:

Apply equations in such a way that terms become "simpler".

Start from both sides:

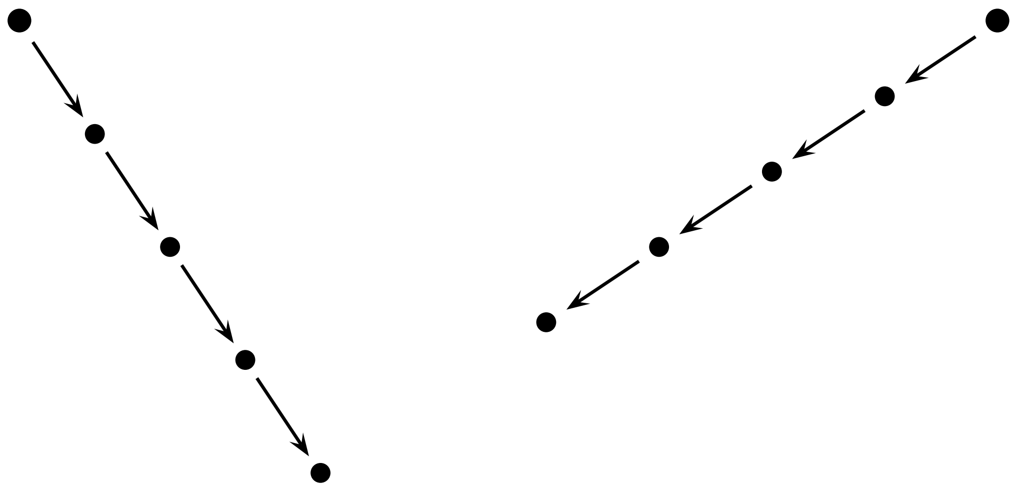


Introductory Example 2: Equations

A better approach:

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

Start from both sides:

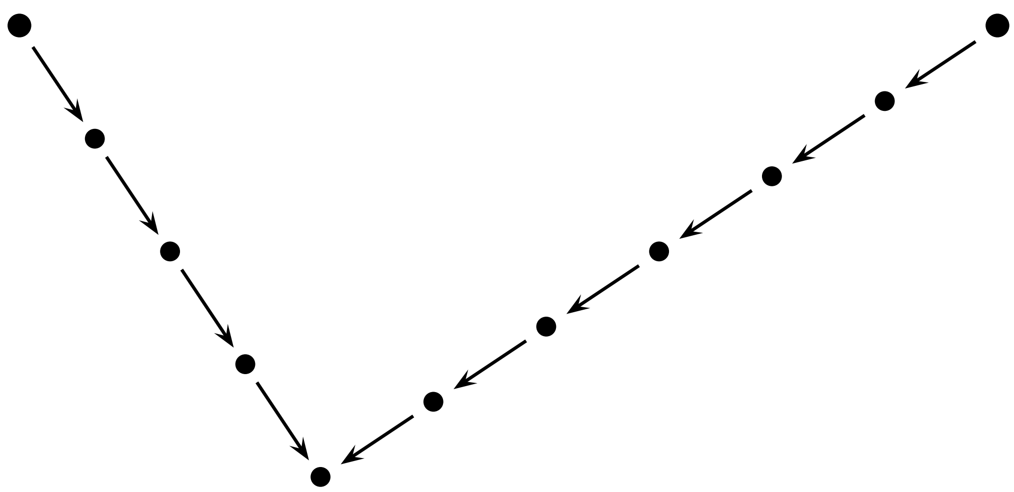


Introductory Example 2: Equations

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

$$x + 0 = x \tag{1}$$

$$x + (-x) = 0 \tag{2}$$

$$x + (y + z) = (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} = \frac{x + y}{z} \tag{4}$$

$$\frac{x}{x} = 1 \tag{5}$$

Introductory Example 2: Equations

Orient equations.

$$x + 0 \to x \tag{1}$$

$$x + (-x) \to 0 \tag{2}$$

$$x + (y + z) \to (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} \to \frac{x + y}{z} \tag{4}$$

$$\frac{x}{x} \to 1 \tag{5}$$

Introductory Example 2: Equations

Orient equations.

Advantage:
Now there are only finitely many and finitely long derivations.

$$x + 0 \to x \tag{1}$$

$$x + (-x) \to 0 \tag{2}$$

$$x + (y + z) \to (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} \to \frac{x + y}{z} \tag{4}$$

$$\frac{x}{x} \to 1 \tag{5}$$

Introductory Example 2: Equations

Orient equations.

But:

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

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

$$x + 0 \to x \tag{1}$$

$$x + (-x) \to 0 \tag{2}$$

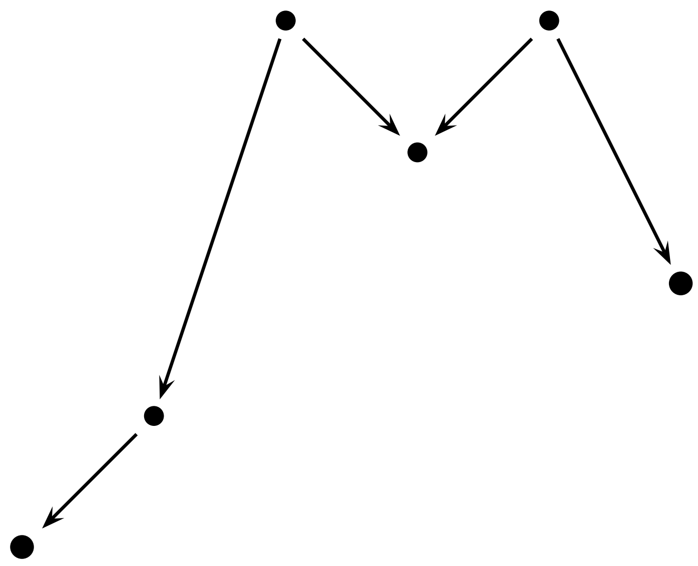
$$x + (y + z) \to (x + y) + z \tag{3}$$

$$\frac{x}{z} + \frac{y}{z} \to \frac{x+y}{z} \tag{4}$$

$$\frac{x}{x} \to 1 \tag{5}$$

Introductory Example 2: Equations

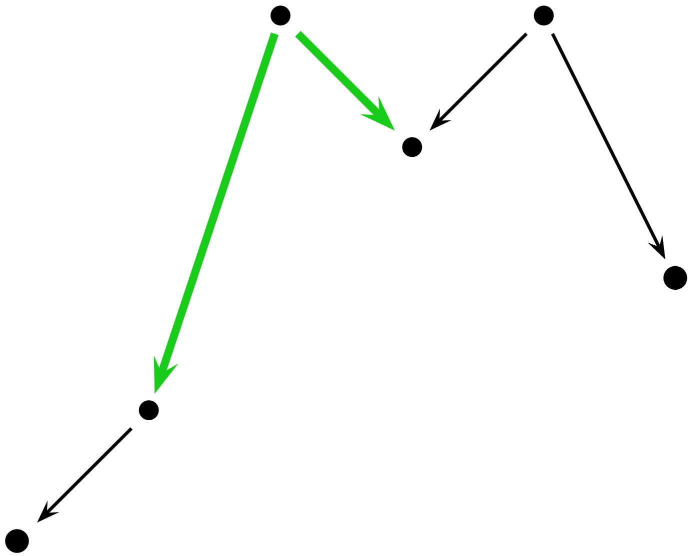
The chain of equalities that we considered at the beginning looks roughly like this:



Introductory Example 2: Equations

Idea:

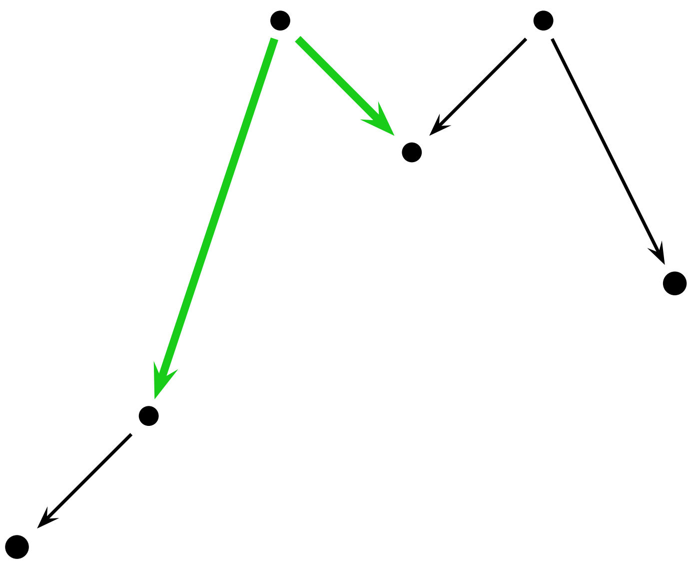
Derive new equations that enable “shortcuts”.



Introductory Example 2: Equations

Idea:

Derive new equations that enable “shortcuts”.



From

$$x + (-x) \to 0 \qquad (2)$$

$$x + (y + z) \to (x + y) + z \qquad (3)$$

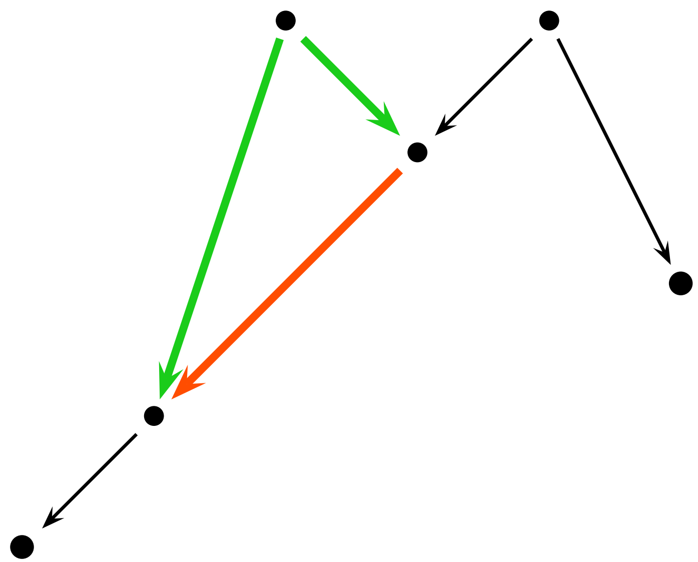
we derive

$$(x + y) + (-y) \to x + 0 \qquad (6)$$

Introductory Example 2: Equations

Idea:

Derive new equations that enable "shortcuts".



From

$$x + (-x) \to 0 \qquad (2)$$

$$x + (y + z) \to (x + y) + z \qquad (3)$$

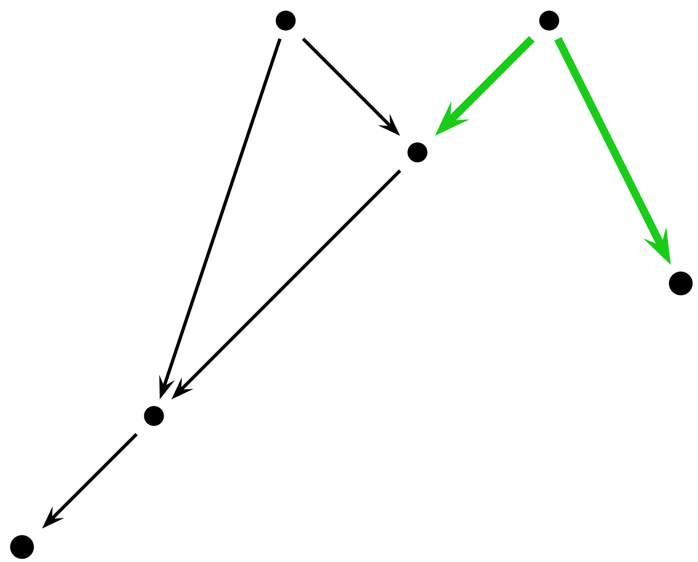
we derive

$$(x + y) + (-y) \to x + 0 \qquad (6)$$

Introductory Example 2: Equations

Idea:

Derive new equations that enable “shortcuts”.



From

$$\frac{x}{z} + \frac{y}{z} \to \frac{x+y}{z} \qquad (4)$$

$$\frac{x}{x} \to 1 \qquad (5)$$

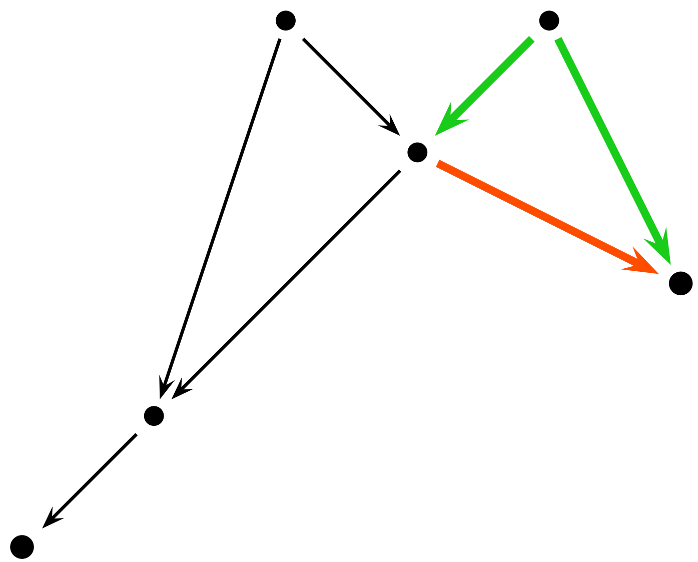
we derive

$$\frac{x+y}{x} \to 1 + \frac{y}{x} \qquad (7)$$

Introductory Example 2: Equations

Idea:

Derive new equations that enable “shortcuts”.



From

$$\frac{x}{z} + \frac{y}{z} \to \frac{x+y}{z} \qquad (4)$$

$$\frac{x}{x} \to 1 \qquad (5)$$

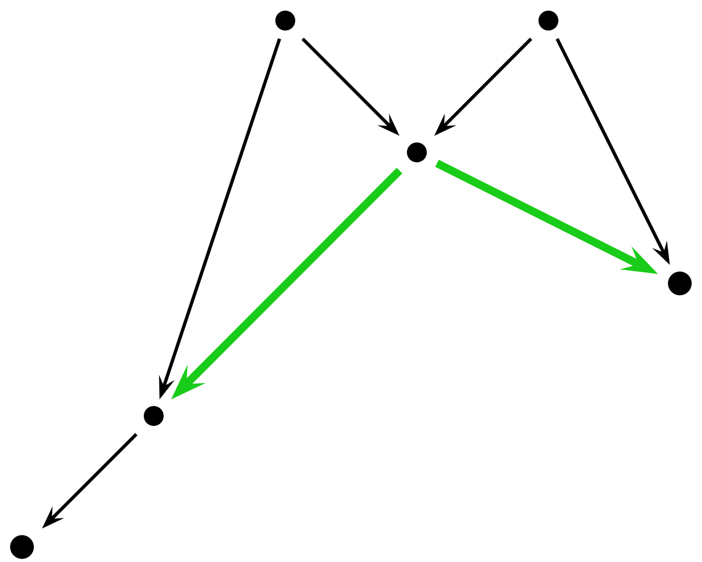
we derive

$$\frac{x+y}{x} \to 1 + \frac{y}{x} \qquad (7)$$

Introductory Example 2: Equations

Idea:

Derive new equations that enable “shortcuts”.



From

$$(x+y)+(-y) \to x+0 \qquad (6)$$

$$\frac{x+y}{x} \to 1+\frac{y}{x} \qquad (7)$$

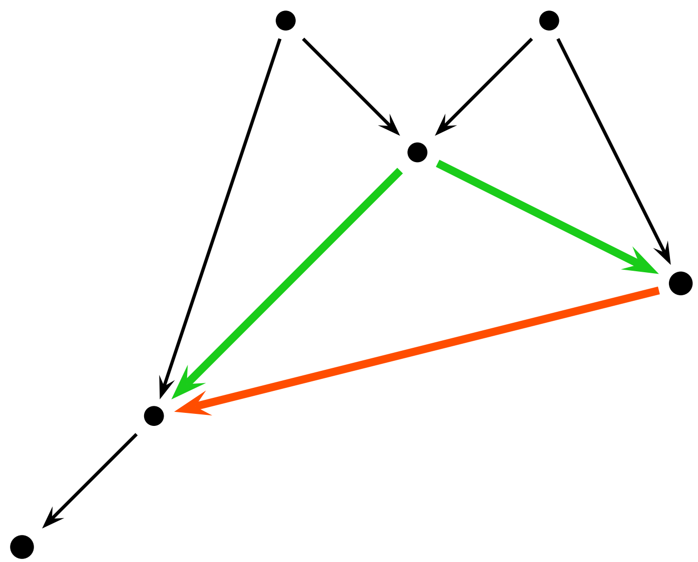
we derive

$$1+\frac{-y}{x+y} \to \frac{x+0}{x+y} \qquad (8)$$

Introductory Example 2: Equations

Idea:

Derive new equations that enable "shortcuts".



From

$$(x + y) + (-y) \to x + 0 \qquad (6)$$

$$\frac{x + y}{x} \to 1 + \frac{y}{x} \qquad (7)$$

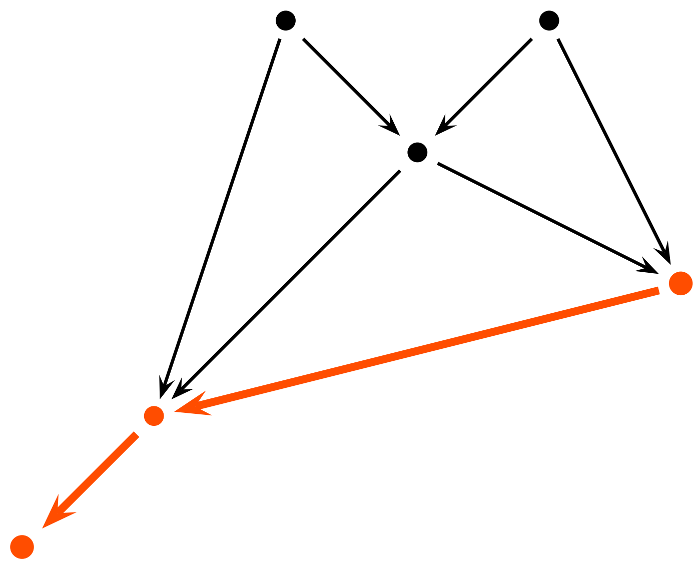
we derive

$$1 + \frac{-y}{x + y} \to \frac{x + 0}{x + y} \qquad (8)$$

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

Waldmeister

(Thomas Hilenbrand,

`http://www.mpi-inf.mpg.de/~hillen/waldmeister/`)

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:

$\Rightarrow$ 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

$$\begin{aligned} &\forall Agent\ \forall Message\ \forall Key. \\ &\qquad knows(Agent, crypt(Message, Key)) \\ &\qquad \wedge\ knows(Agent, Key) \\ &\qquad \rightarrow\ knows(Agent, Message). \end{aligned}$$

Result

This difference is also important for automated reasoning:

- For axioms that are well-known and frequently used,
  we can develop optimal specialized methods.
  $\Rightarrow$ Computer Algebra
  $\Rightarrow$ Automated Reasoning II (next semester)

- For new axioms, we have to develop methods that
  do "something reasonable" for arbitrary formulas.
  $\Rightarrow$ this lecture

- Combining the two approaches
  $\Rightarrow$ Automated Reasoning II

Topics of the Course

Preliminaries

abstract reduction systems

well-founded orderings

Propositional logic

syntax, semantics

calculi: CDCL-procedure, ...

implementation: 2-watched literals, clause learning

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, complexity, implementation.

Part 1: Preliminaries

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

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

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

Relations

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

For two n-ary relations R, Q over some set M, their union ($\cup$) or intersection ($\cap$) is again an n-ary relation, where

$$R \cup Q := \{ (m_1, \ldots, m_n) \in M^n \mid (m_1, \ldots, m_n) \in R \\ \text{or } (m_1, \ldots, m_n) \in Q \}$$

$$R \cap Q := \{ (m_1, \ldots, m_n) \in M^n \mid (m_1, \ldots, m_n) \in R \\ \text{and } (m_1, \ldots, m_n) \in Q \}.$$

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

Relations

We often use predicate notation for relations:

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

For binary relations, we often use infix notation, so
$(m, m') \in < \quad \Leftrightarrow \quad <(m, m') \quad \Leftrightarrow \quad m < m'$.

Words

Given a non-empty alphabet Σ, the set Σ^* of finite words over Σ is defined inductively by

(i) the empty word ε is in Σ^*,

(ii) if $u \in \Sigma^*$ and $a \in \Sigma$ then ua is in Σ^*.

The set of non-empty finite words Σ^+ 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

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

Througout 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 $(A, \to)$, where

A is a non-empty set,

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

The relation $\to$ is usually written in infix notation, i. e., $a \to b$ instead of $(a, b) \in \to$.

Abstract Reduction Systems

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

$$a \,(\to' \circ \to'')\, c \quad \text{if and only if}$$
$$a \to' b \text{ and } b \to'' c \text{ for some } b \in B.$$

Abstract Reduction Systems

$\to^0$	$= \{ (a, a) \mid a \in A \}$	identity
$\to^{i+1}$	$= \to^i \circ \to$	$i+1$-fold composition
$\to^+$	$= \bigcup_{i>0} \to^i$	transitive closure
$\to^*$	$= \bigcup_{i\geq 0} \to^i \; = \; \to^+ \cup \to^0$	reflexive transitive closure
$\to^=$	$= \to \cup \to^0$	reflexive closure
$\leftarrow$	$= \to^{-1} \; = \; \{ (b, c) \mid c \to b \}$	inverse
$\leftrightarrow$	$= \to \cup \leftarrow$	symmetric closure
$\leftrightarrow^+$	$= (\leftrightarrow)^+$	transitive symmetric closure
$\leftrightarrow^*$	$= (\leftrightarrow)^*$	refl. trans. 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 (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 $b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow \ldots$.

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 $x \in M$,

irreflexivity, 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 $\succ^{-1}$

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

Orderings

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

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

Analogously:

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

An $a \in M$ is called largest, 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
$a_0 \succ a_1 \succ a_2 \succ \ldots$ with $a_i \in M$.

Well-Foundedness and Termination

Lemma 1.2:

If $>$ is a well-founded partial ordering and $\rightarrow \subseteq >$,
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. Then let their lexicographic combination

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

on $M_1 \times M_2$ be defined as

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

(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 Σ with a well-founded ordering $>_\Sigma$, the relation $\succ$ defined as

$$w \succ w' \quad :\Leftrightarrow$$

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

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

Counterexamples:

$(\mathbb{Z}, >)$

$(\mathbb{N}, <)$

the lexicographic ordering on Σ^*

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 $\succ = (\succ_1, \succ_2)_{lex}$ is well-founded.

Monotone Mappings

Let $(M_1, >_1)$ and $(M_2, >_2)$ be strict partial orderings.
A mapping $\varphi : M_1 \to M_2$ is called monotone,
if $a >_1 b$ implies $\varphi(a) >_2 \varphi(b)$ for all $a, b \in M_1$.

Lemma 1.6:
If φ is a monotone mapping from $(M_1, >_1)$ to $(M_2, >_2)$
and $(M_2, >_2)$ is well-founded, then $(M_1, >_1)$ 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 $m \succ m'$,[a]
> > then $Q(m)$.[b]

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

---

[a] induction hypothesis
[b] induction step

Well-founded Recursion

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

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

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 $S : M \to \mathbb{N}$. 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.,

$$
\begin{aligned}
m \in S \quad &:\Leftrightarrow \quad S(m) > 0 \\
(S_1 \cup S_2)(m) \quad &:= \quad S_1(m) + S_2(m) \\
(S_1 \cap S_2)(m) \quad &:= \quad \min\{S_1(m), S_2(m)\} \\
(S_1 - S_2)(m) \quad &:= \quad \begin{cases} S_1(m) - S_2(m) & \text{if } S_1(m) \geq S_2(m) \\ 0 & \text{otherwise} \end{cases} \\
S_1 \subseteq S_2 \quad &:\Leftrightarrow \quad S_1(m) \leq S_2(m) \text{ for all } m \in M
\end{aligned}
$$

Multisets

A multiset S is called finite, if

$$|\{\, m \in M \mid S(m) > 0 \,\}| < \infty.$$

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

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

there exist multisets X and Y over M such that

$$\emptyset \neq X \subseteq S_1,$$
$$S_2 = (S_1 - X) \cup Y,$$
$$\forall y \in Y\ \exists x \in X\colon\ x \succ y$$

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

$$
\begin{aligned}
& S_1 \succ_{\text{mul}}^{\text{HO}} S_2 \text{ if and only if} \\
& \quad S_1 \neq S_2 \text{ and} \\
& \quad\quad \forall m \in M\colon \big(S_2(m) > S_1(m) \\
& \quad\quad\quad\quad \Rightarrow \ \exists m' \in M\colon m' \succ m \text{ and } S_1(m') > S_2(m')\big)
\end{aligned}
$$

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 $\succ^1_{\mathrm{mul}}$ given by

$S_1 \succ^1_{\mathrm{mul}} S_2$ if and only if

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

$$S_2 = (S_1 - \{x\}) \cup Y,$$
$$\forall y \in Y\colon x \succ y$$

Multiset Orderings

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

Theorem 1.11:
If $\succ$ is a strict partial ordering, then
(a) $\succ_{\mathsf{mul}} = \succ^{\mathsf{HO}}_{\mathsf{mul}}$,
(b) $\succ_{\mathsf{mul}} = (\succ^{1}_{\mathsf{mul}})^{+}$.

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

1.5 Complexity Theory Prerequisites

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

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 P denotes all polynomial-time decision problems.

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 $L' \in \mathrm{P}$ then $L \in \mathrm{P}$.

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.

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),
- growing importance for discrete optimization problems

2.1 Syntax

- propositional variables
- logical connectives
  $\Rightarrow$ Boolean combinations

Propositional Variables

Let Π be a set of propositional variables.

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

Propositional Formulas

F_Π is the set of propositional formulas over Π defined inductively as follows:

$$
\begin{array}{llll}
F, G & ::= & \bot & \text{(falsum)} \\
 & \mid & \top & \text{(verum)} \\
 & \mid & P, \quad P \in \Pi & \text{(atomic formula)} \\
 & \mid & (\neg F) & \text{(negation)} \\
 & \mid & (F \wedge G) & \text{(conjunction)} \\
 & \mid & (F \vee G) & \text{(disjunction)} \\
 & \mid & (F \rightarrow G) & \text{(implication)} \\
 & \mid & (F \leftrightarrow G) & \text{(equivalence)}
\end{array}
$$

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 $P \wedge Q \wedge R$ (and analogously for $\vee$).

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

Formula Manipulation

Automated reasoning is very much formula manipulation.
In order to precisely represent 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

$$
\begin{aligned}
\mathrm{pos}(F) &:= \{\varepsilon\} \text{ if } F \in \{\top, \bot\} \text{ or } F \in \Pi \\
\mathrm{pos}(\neg F) &:= \{\varepsilon\} \cup \{\, 1p \mid p \in \mathrm{pos}(F)\,\} \\
\mathrm{pos}(F \circ G) &:= \{\varepsilon\} \cup \{\, 1p \mid p \in \mathrm{pos}(F)\,\} \cup \{\, 2p \mid p \in \mathrm{pos}(G)\,\} \\
&\qquad \text{where } \circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}.
\end{aligned}
$$

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 $\|$ we denote incomparable positions, that is,
$p \| q$ 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 $p \| q$.

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

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

Formula Manipulation

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

$$\begin{aligned}
F[G]_\varepsilon &:= G \\
(\neg F)[G]_{1p} &:= \neg(F[G]_p) \\
(F_1 \circ F_2)[G]_{1p} &:= (F_1[G]_p \circ F_2) \\
(F_1 \circ F_2)[G]_{2p} &:= (F_1 \circ F_2[G]_p)
\end{aligned}$$

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

Formula Manipulation

Example 2.1:
The set of positions for the formula $F = (P \rightarrow Q) \rightarrow (P \wedge \neg Q)$ is $\mathrm{pos}(F) = \{\varepsilon, 1, 11, 12, 2, 21, 22, 221\}$.

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

Polarities

A further prerequisite for efficient formula manipulation is the polarity of a subformula G of F.

The polarity 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.

Polarities

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

$$\begin{aligned}
\mathrm{pol}(F, \varepsilon) &:= 1 \\
\mathrm{pol}(\neg F, 1p) &:= -\mathrm{pol}(F, p) \\
\mathrm{pol}(F_1 \circ F_2, ip) &:= \mathrm{pol}(F_i, p) \text{ if } \circ \in \{\wedge, \vee\} \\
\mathrm{pol}(F_1 \rightarrow F_2, 1p) &:= -\mathrm{pol}(F_1, p) \\
\mathrm{pol}(F_1 \rightarrow F_2, 2p) &:= \mathrm{pol}(F_2, p) \\
\mathrm{pol}(F_1 \leftrightarrow F_2, ip) &:= 0
\end{aligned}$$

Polarities

Example 2.2:

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

Then $\mathrm{pol}(F, 1) = \mathrm{pol}(F, 12) = \mathrm{pol}(F, 221) = -1$ and $\mathrm{pol}(F, \varepsilon) = \mathrm{pol}(F, 11) = \mathrm{pol}(F, 2) = \mathrm{pol}(F, 21) = \mathrm{pol}(F, 22) = 1$.

For the formula $F' = (P \wedge Q) \leftrightarrow (P \vee Q)$ we get $\mathrm{pol}(F', \varepsilon) = 1$ and $\mathrm{pol}(F', p) = 0$ for all $p \in \mathrm{pos}(F')$ different from ε.

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 Π-valuation is a map

$$\mathcal{A} : \Pi \to \{0, 1\}.$$

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

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

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

$$
\begin{aligned}
\mathcal{A}^*(\bot) &= 0 \\
\mathcal{A}^*(\top) &= 1 \\
\mathcal{A}^*(P) &= \mathcal{A}(P) \\
\mathcal{A}^*(\neg F) &= 1 - \mathcal{A}^*(F) \\
\mathcal{A}^*(F \wedge G) &= \mathrm{min}(\mathcal{A}^*(F), \mathcal{A}^*(G)) \\
\mathcal{A}^*(F \vee G) &= \mathrm{max}(\mathcal{A}^*(F), \mathcal{A}^*(G)) \\
\mathcal{A}^*(F \to G) &= \mathrm{max}(1 - \mathcal{A}^*(F), \mathcal{A}^*(G)) \\
\mathcal{A}^*(F \leftrightarrow G) &= \text{if } \mathcal{A}^*(F) = \mathcal{A}^*(G) \text{ then } 1 \text{ else } 0
\end{aligned}
$$

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

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

2.3 Models, Validity, and Satisfiability

Let F be a Π-formula.

We say that F is true under $\mathcal{A}$ ($\mathcal{A}$ is a model of F;
F is valid in $\mathcal{A}$; F holds under $\mathcal{A}$), 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 Π-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 Π-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 $F \models\mid G$, if for all Π-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

Proposition 2.3:
$F \models G$ if and only if $\models (F \rightarrow G)$.

Proposition 2.4:
$F \models\!\mid G$ if and only if $\models (F \leftrightarrow G)$.

Entailment and Equivalence

Entailment is extended to sets of formulas N in the "natural way":

$N \models F$ if for all Π-valuations $\mathcal{A}$:
if $\mathcal{A} \models G$ for all $G \in N$, 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.

Validity vs. Unsatisfiability

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

Proposition 2.5:
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 2.6:
$N \models F$ if and only if $N \cup \{\neg F\}$ is unsatisfiable.

Proposition 2.7:
$N \models \bot$ 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 under $\mathcal{A}$.

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. (later more)

Substitution Theorem

Proposition 2.8:
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 2.9:
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 2.10:

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

$$
\begin{array}{rcll}
(F \wedge F) & \models\mid & F & \\
(F \vee F) & \models\mid & F & \text{(Idempotency)} \\
(F \wedge G) & \models\mid & (G \wedge F) & \\
(F \vee G) & \models\mid & (G \vee F) & \text{(Commutativity)} \\
(F \wedge (G \wedge H)) & \models\mid & ((F \wedge G) \wedge H) & \\
(F \vee (G \vee H)) & \models\mid & ((F \vee G) \vee H) & \text{(Associativity)} \\
(F \wedge (G \vee H)) & \models\mid & ((F \wedge G) \vee (F \wedge H)) & \\
(F \vee (G \wedge H)) & \models\mid & ((F \vee G) \wedge (F \vee H)) & \text{(Distributivity)}
\end{array}
$$

Some Important Equivalences

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

$$(F \wedge (F \vee G)) \;\models\!\mid\; F$$

$$(F \vee (F \wedge G)) \;\models\!\mid\; F \qquad \text{(Absorption)}$$

$$(\neg\neg F) \;\models\!\mid\; F \qquad \text{(Double Negation)}$$

$$\neg(F \wedge G) \;\models\!\mid\; (\neg F \vee \neg G)$$

$$\neg(F \vee G) \;\models\!\mid\; (\neg F \wedge \neg G) \qquad \text{(De Morgan's Laws)}$$

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

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

$$(F \wedge G) \;\models\!\mid\; \bot, \text{ if } G \text{ is unsatisfiable}$$

$$(F \vee G) \;\models\!\mid\; F, \text{ if } G \text{ is unsatisfiable} \qquad \text{(Tautology Laws)}$$

Some Important Equivalences

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

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

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

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

An Important Entailment

Proposition 2.11:

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

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

2.4 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:

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

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

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

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 2.12:
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:

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

Conversion to CNF/DNF

Step 2: Eliminate implications:

$$H[F \to G]_p \;\Rightarrow_{\mathsf{CNF}}\; H[\neg F \vee G]_p$$

Step 3: Push negations downward:

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

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

Step 4: Eliminate multiple negations:

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

Conversion to CNF/DNF

Step 5: Push disjunctions downward:

$$H[(F \land F') \lor G]_p \;\Rightarrow_{\mathrm{CNF}}\; H[(F \lor G) \land (F' \lor G)]_p$$

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

$$\begin{aligned}
H[F \land \top]_p &\;\Rightarrow_{\mathrm{CNF}}\; H[F]_p \\
H[F \land \bot]_p &\;\Rightarrow_{\mathrm{CNF}}\; H[\bot]_p \\
H[F \lor \top]_p &\;\Rightarrow_{\mathrm{CNF}}\; H[\top]_p \\
H[F \lor \bot]_p &\;\Rightarrow_{\mathrm{CNF}}\; H[F]_p \\
H[\neg\bot]_p &\;\Rightarrow_{\mathrm{CNF}}\; H[\top]_p \\
H[\neg\top]_p &\;\Rightarrow_{\mathrm{CNF}}\; H[\bot]_p
\end{aligned}$$

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.

$\square$

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

"find a formula G in CNF such that $F \models\mid G$"

is unpractical.

But if we relax the requirement to

"find a formula G in CNF such that $F \models \bot \Leftrightarrow G \models \bot$"

we can get an efficient transformation.

Tseitin Transformation

Proposition 2.13:
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.

Satisfiability-preserving CNF transformation (Tseitin 1970):

Use the rule above recursively for all subformulas in the original formula (this introduces a linear number of new propositional variables Q and definitions $Q \leftrightarrow F$).

Convert of 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.

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

If $\mathrm{pol}(H, p) = -1$ 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 $\mathrm{def}(H, p, Q, F)$ by

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

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 $H[Q]_p \wedge \mathrm{def}(H, p, Q, F)$ is satisfiable.

Optimized CNF

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

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

The functions ν and $\bar{\nu}$ on the next slide give us an overapproximation for the number of clauses generated by a formula that occurs positively/negatively.

Optimized CNF

G	$\nu(G)$	$\bar{\nu}(G)$
$P, \top, \bot$	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:

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

$$
\begin{aligned}
H[(F \wedge \top)]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \vee \bot)]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \leftrightarrow \bot)]_p &\Rightarrow_{\mathrm{OCNF}} H[\neg F]_p \\
H[(F \leftrightarrow \top)]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \vee \top)]_p &\Rightarrow_{\mathrm{OCNF}} H[\top]_p \\
H[(F \wedge \bot)]_p &\Rightarrow_{\mathrm{OCNF}} H[\bot]_p
\end{aligned}
$$

Optimized CNF

$$
\begin{aligned}
H[(F \wedge F)]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \vee F)]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \wedge (F \vee G))]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \vee (F \wedge G))]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \wedge \neg F)]_p &\Rightarrow_{\mathrm{OCNF}} H[\bot]_p \\
H[(F \vee \neg F)]_p &\Rightarrow_{\mathrm{OCNF}} H[\top]_p \\
H[\neg \top]_p &\Rightarrow_{\mathrm{OCNF}} H[\bot]_p \\
H[\neg \bot]_p &\Rightarrow_{\mathrm{OCNF}} H[\top]_p
\end{aligned}
$$

Optimized CNF

$$
\begin{aligned}
H[(F \rightarrow \bot)]_p &\Rightarrow_{\text{OCNF}} H[\neg F]_p \\
H[(F \rightarrow \top)]_p &\Rightarrow_{\text{OCNF}} H[\top]_p \\
H[(\bot \rightarrow F)]_p &\Rightarrow_{\text{OCNF}} H[\top]_p \\
H[(\top \rightarrow F)]_p &\Rightarrow_{\text{OCNF}} H[F]_p
\end{aligned}
$$

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:

$$H[F]_p \;\Rightarrow_{\mathrm{OCNF}}\; H[P]_p \land \mathrm{def}(H, p, P, F)$$

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

and $\nu(H[F]_p) > \nu(H[P]_p \land \mathrm{def}(H, p, P, F))$.

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

Optimized CNF

Step 3: Eliminate equivalences dependent on their polarity:

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

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

$$H[F \leftrightarrow G]_p \;\Rightarrow_{\text{OCNF}}\; H[(F \land G) \lor (\neg F \land \neg G)]_p$$

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

Optimized CNF

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

Remark: The $\Rightarrow_{\mathrm{OCNF}}$ 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.

$\Rightarrow$ 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.

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

Partial Valuations

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

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 $\overline{L} \in M$.

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

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

Partial Valuations

A clause is true under a partial valuation $\mathcal{A}$ (or under 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 under $\mathcal{A}$, 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 under $\mathcal{A}$. 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 Proc.

```
boolean DPLL(literal set M, clause set N) {
    if (all clauses in N are true under M) return true;
    elsif (some clause in N is false under M) return false;
    elsif (N contains unit literal P) return DPLL(M ∪ {P}, N);
    elsif (N contains unit literal ¬P) return DPLL(M ∪ {¬P}, N);
    elsif (N contains pure literal P) return DPLL(M ∪ {P}, N);
    elsif (N contains pure literal ¬P) return DPLL(M ∪ {¬P}, N);
    else {
        let P be some undefined variable in N;
        if (DPLL(M ∪ {P}, N)) return true;
        else return DPLL(M ∪ {¬P}, N);
    }
}
```

The Davis-Putnam-Logemann-Loveland Proc.

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

2.7 From DPLL to CDCL

In practice, there are several changes to the procedure:

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

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

Information is reused by conflict analysis and learning.

The branching variable is not chosen randomly.

Under certain circumstances, the procedure is restarted.

Conflict Analysis and Learning

Conflict analysis serves a dual purpose:

Backjumping (non-chronological backtracking): If we detect that the conflict is independent of some earlier branch, we can skip over that backtrack level.

Learning: By deriving a new clause from the 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.

Annotated literal:

- L: deduced literal, due to unit propagation.
- L^{d}: decision literal (guessed literal).

Formalizing CDCL

Unit Propagate:

$$M \parallel N \cup \{C \vee L\} \;\Rightarrow_{\mathrm{CDCL}}\; M\, L \parallel N \cup \{C \vee L\}$$

if C is false under M and L is undefined under M.

Decide:

$$M \parallel N \;\Rightarrow_{\mathrm{CDCL}}\; M\, L^{\mathrm{d}} \parallel N$$

if L is undefined under M and contained in N.

Fail:

$$M \parallel N \cup \{C\} \;\Rightarrow_{\mathrm{CDCL}}\; \mathit{fail}$$

if C is false under M and M contains no decision literals.

Formalizing CDCL

Backjump:

$$M' \, L^{\mathrm{d}} \, M'' \parallel N \;\Rightarrow_{\mathrm{CDCL}}\; M' \, L' \parallel N$$

if there is some "backjump clause" $C \vee L'$ such that

$N \models C \vee L'$,

C is false under M', and

L' is undefined under 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 under M.

There are many possible backjump clauses.
One candidate: $\overline{L_1} \vee \ldots \vee \overline{L_n}$,
where the L_i are all the decision literals in $M' \; L^{\mathrm{d}} \; M''$.
(But usually there are better choices.)

Formalizing CDCL

Lemma 2.16:

If we reach a state $M \parallel N$ starting from $\varepsilon \parallel N$, then:

(1) M does not contain complementary literals.

(2) Every deduced literal L in M follows from N and decision literals occurring before L in 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 under 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

Suppose that we have reached a state $M \parallel N$ such that some clause $C \in N$ (or following from N) is false under M.

Consequently, every literal of C is the complement of some literal in M.

(1) If every literal in C is the complement of a decision literal of M, then C is a backjump clause.

(2) Otherwise, $C = C' \vee \overline{L}$, such that L is a deduced literal.

For every deduced literal L, there is a clause $D \vee L$, such that $N \models D \vee L$ and D is false under M.

Then $N \models D \vee C'$ and $D \vee C'$ is also false under M. ($D \vee C'$ is a resolvent of $C' \vee \overline{L}$ and $D \vee L$.)

Getting Better Backjump Clauses

By repeating this process, we will eventually obtain a clause that satisfies the requirements of a backjump clause (or the empty clause).

Usually, one resolves the literals in the reverse order in which they were added to M and stops as soon as one obtains a clause in which all but one literal are complements of literals occurring in M before the last decision literal.
$\Rightarrow$ 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:

$$M \parallel N \;\Rightarrow_{\mathsf{CDCL}}\; \varepsilon \parallel N$$

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.

Preprocessing

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

(i) Deletion of tautologies

(ii) Deletion of duplicated literals

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.

Inprocessing

Examples:

(i) Subsumption

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

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

Inprocessing

(ii) Purity deletion

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

(iii) Merging replacement resolution

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

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

Inprocessing

(vi) Bounded variable elimination

Compute all possible resolution steps

$$\frac{C \lor L \qquad D \lor \overline{L}}{C \lor D}$$

on a literal L with premises in N;
add all non-tautological conclusions to N;
then throw away all clauses containing L or $\overline{L}$;
repeat this as long as $|N|$ does not grow.

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 $D' \vee \overline{L} \in N$ are asymmetric tautologies w. r. t. N.

RAT elimination:

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

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

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.

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/`.

2.9 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.

$\Rightarrow$ Canonical representation of formulas.

$\Rightarrow$ Satisfiability checking as a side effect.

BDDs

BDD (Binary decision diagram):

Labelled DAG (directed acyclic graph).

Leaf nodes:

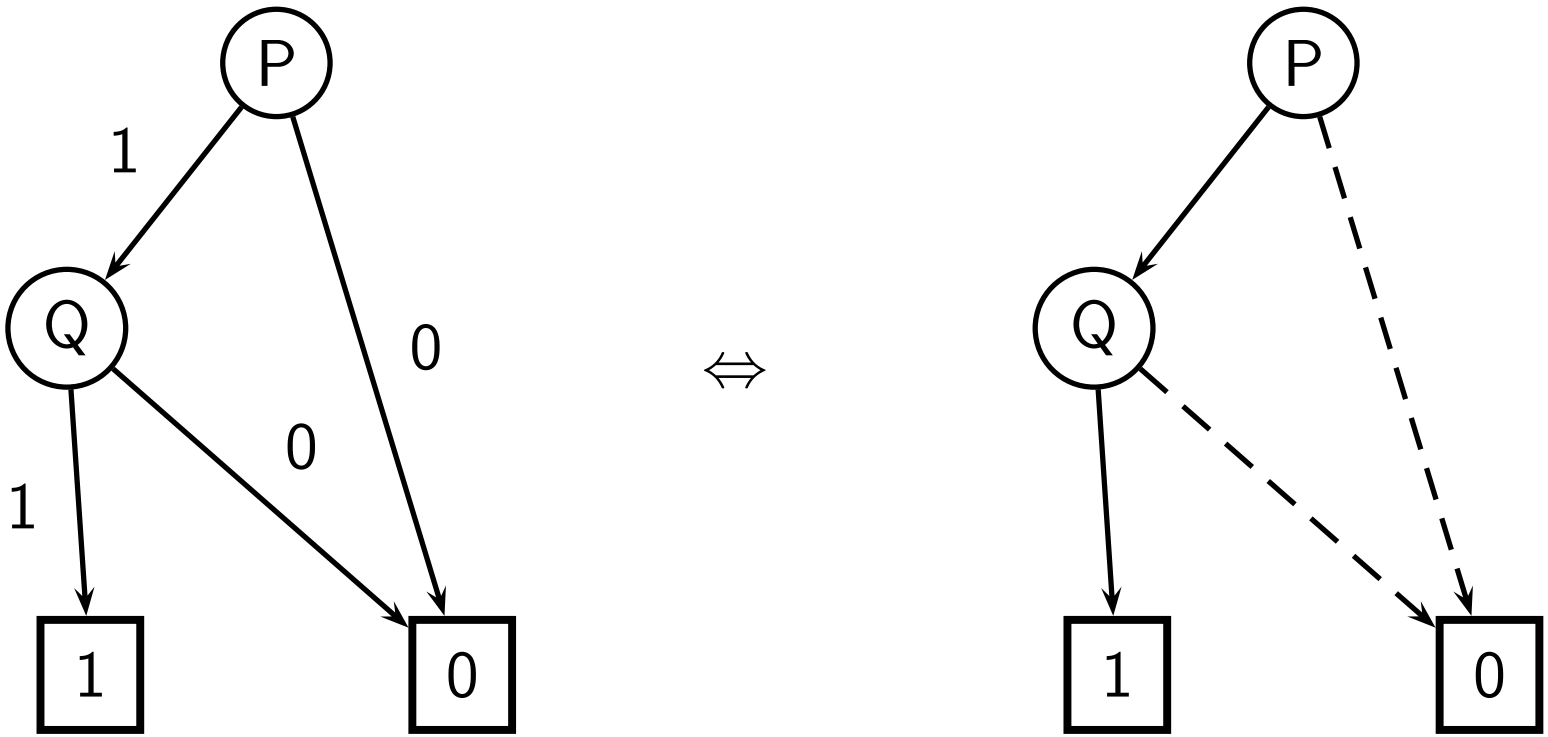
labelled with a truth value (0 or 1).

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.

$\Rightarrow$ 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 $\bot$ (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

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

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:

$$F \;\;|\!\!=\!\!|\;\; (P \wedge F\{P \mapsto \top\}) \vee (\neg P \wedge F\{P \mapsto \bot\})$$

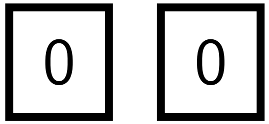
(*Shannon expansion* of F, originally due to Boole).

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

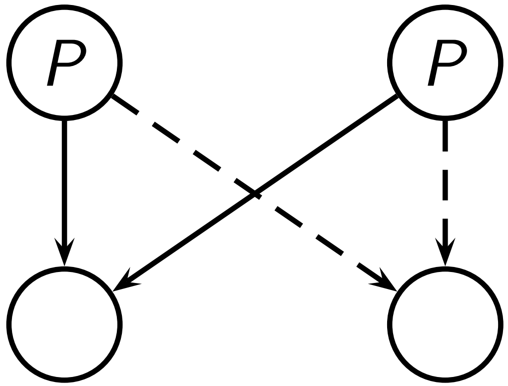
Reduced OBDDs

An OBDD is called *reduced*, if it has

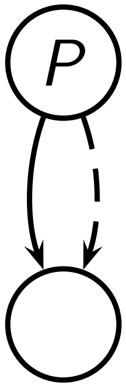
- no duplicated leaf nodes



- no duplicated interior nodes



- no redundant tests



Reduced OBDDs

Theorem 2.20 (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 $(\mathit{PropVar}, \mathit{Ptr}_0, \mathit{Ptr}_1)$,

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

Reduced OBDDs

Theorem 2.21 (Bryant 1986):
If v and v' are two different nodes in a reduced OBDD, then they represent non-equivalent formulas.

Reduced OBDDs

Corollary 2.22:

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.

$$
\begin{aligned}
F \circ G \;&\models\mid\; (P \wedge (F \circ G)\{P \mapsto \top\}) \vee (\neg P \wedge (F \circ G)\{P \mapsto \bot\}) \\
&\models\mid\; (P \wedge (F\{P \mapsto \top\} \circ G\{P \mapsto \top\}) \\
&\qquad \vee (\neg P \wedge (F\{P \mapsto \bot\} \circ G\{P \mapsto \bot\})))
\end{aligned}
$$

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*

$$\forall P.\, F \;\;\models\!\mid\;\; (F\{P \mapsto \top\}) \land (F\{P \mapsto \bot\})$$

$$\exists P.\, F \;\;\models\!\mid\;\; (F\{P \mapsto \top\}) \lor (F\{P \mapsto \bot\})$$

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:

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

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

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

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.

2.10 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.

$\Rightarrow$ Semi-canonical representation of formulas.

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.

2.11 Other Calculi

Ordered resolution

Tableau calculus

Hilbert calculus

Sequent calculus

Natural deduction

see next chapter

Part 3: First-Order Logic

First-order logic

- formalizes fundamental mathematical concepts
- is expressive (Turing-complete)
- is not too expressive
  (e. g. not axiomatizable: natural numbers, uncountable sets)
- has a rich structure of decidable fragments
- has a rich model and proof theory

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

3.1 Syntax

Syntax:

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

Signatures

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

- Ω is a set of function symbols f with arity $n \geq 0$, written $\mathrm{arity}(f) = n$,
- Π is a set of predicate symbols P with arity $m \geq 0$, written $\mathrm{arity}(P) = m$.

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 Σ and X (Σ-terms) are formed according to these syntactic rules:

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

By $\mathrm{T}_\Sigma(X)$ we denote the set of Σ-terms (over X).

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

By T_Σ we denote the set of Σ-ground terms.

Terms

In other words, terms are formal expressions with well-balanced parentheses which we may also view as marked, ordered trees.
The markings are function symbols or variables.
The nodes correspond to the subterms of the term.
A node v that is marked with a function symbol f of arity n has exactly n subtrees representing the n immediate subterms of v.

Atoms

Atoms (also called atomic formulas) over Σ are formed according to this syntax:

$$
\begin{array}{lcll}
A, B & ::= & P(s_1, \ldots, s_m) \quad , \; P/m \in \Pi & \text{(non-equational atom)} \\
 & \Big[\; \mid & (s \approx t) & \text{(equation)} \; \Big]
\end{array}
$$

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

$$
\begin{array}{lcll}
L & ::= & A & \text{(positive literal)} \\
 & | & \neg A & \text{(negative literal)}
\end{array}
$$

Clauses

$$
\begin{array}{llll}
C, D & ::= & \bot & \text{(empty clause)} \\
 & \mid & L_1 \vee \ldots \vee L_k, \;\; k \geq 1 & \text{(non-empty clause)}
\end{array}
$$

General First-Order Formulas

$\mathsf{F}_\Sigma(X)$ is the set of first-order formulas over Σ defined as follows:

$$
\begin{array}{rcll}
F, G, H & ::= & \bot & \text{(falsum)} \\
& \mid & \top & \text{(verum)} \\
& \mid & A & \text{(atomic formula)} \\
& \mid & \neg F & \text{(negation)} \\
& \mid & (F \land G) & \text{(conjunction)} \\
& \mid & (F \lor G) & \text{(disjunction)} \\
& \mid & (F \rightarrow G) & \text{(implication)} \\
& \mid & (F \leftrightarrow G) & \text{(equivalence)} \\
& \mid & \forall x\, F & \text{(universal quantification)} \\
& \mid & \exists x\, F & \text{(existential quantification)}
\end{array}
$$

Notational Conventions

We omit parentheses according to the conventions for propositional logic.

$\forall x_1, \ldots, x_n\, F$ and $\exists x_1, \ldots, x_n\, F$ abbreviate
$\forall x_1 \ldots \forall x_n\, F$ and $\exists x_1 \ldots \exists x_n\, F$.

Notational Conventions

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

Examples:

$s + t * u$	for	$+(s, *(t, u))$
$s * u \leq t + v$	for	$\leq (*(s, u), +(t, v))$
$-s$	for	$-(s)$
$s!$	for	$!(s)$
$\|s\|$	for	$\|\_\|(s)$
0	for	$0()$

Example: Peano Arithmetic

$$
\begin{aligned}
\Sigma_{\mathsf{PA}} &= (\Omega_{\mathsf{PA}},\ \Pi_{\mathsf{PA}}) \\
\Omega_{\mathsf{PA}} &= \{0/0,\ +/2,\ */2,\ s/1\} \\
\Pi_{\mathsf{PA}} &= \{</2\}
\end{aligned}
$$

Examples of formulas over this signature are:

$$\forall x, y\,((x < y \lor x \approx y) \leftrightarrow \exists z\,(x + z \approx y))$$
$$\exists x \forall y\,(x + y \approx y)$$
$$\forall x, y\,(x * s(y) \approx x * y + x)$$
$$\forall x, y\,(s(x) \approx s(y) \rightarrow x \approx y)$$
$$\forall x \exists y\,(x < y \land \neg \exists z (x < z \land z < y))$$

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):

$$\mathrm{pos}(x) = \{\varepsilon\},$$
$$\mathrm{pos}(f(s_1, \dots, s_n)) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{\, i\,p \mid p \in \mathrm{pos}(s_i)\,\},$$
$$\mathrm{pos}(P(t_1, \dots, t_n)) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{\, i\,p \mid p \in \mathrm{pos}(t_i)\,\},$$
$$\mathrm{pos}(\forall x\, F) = \{\varepsilon\} \cup \{\, 1\,p \mid p \in \mathrm{pos}(F)\,\},$$
$$\mathrm{pos}(\exists x\, F) = \{\varepsilon\} \cup \{\, 1\,p \mid p \in \mathrm{pos}(F)\,\}.$$

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.

Bound and Free Variables

In $\mathsf{Q}x\, F$, $\mathsf{Q} \in \{\exists, \forall\}$, we call F the scope of the quantifier $\mathsf{Q}x$.
An *occurrence* of a variable x is called bound, if it is inside the scope of a quantifier $\mathsf{Q}x$.
Any other occurrence of a variable is called free.

Formulas without free variables are also called closed formulas or sentential forms.

Formulas without variables are called ground.

Bound and Free Variables

Example:

$$\forall y \quad \overbrace{((\forall x \; \overbrace{P(x)}^{\text{scope of } x}\;) \quad \rightarrow \quad R(x, y))}^{\text{scope of } y}$$

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

$$\sigma : X \to \mathsf{T}_\Sigma(X)$$

such that the domain of σ, that is, the set

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

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

Substitutions

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

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

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

The modification of a substitution σ 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 σ to a term t or formula F by structural induction over the syntactic structure of t or F by the equations depicted on the next page.

In the presence of quantification it is surprisingly complex:
We need to make sure that the (free) variables in the codomain of σ 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 σ to terms and formulas:

$$
\begin{aligned}
f(s_1,\ldots,s_n)\sigma &= f(s_1\sigma,\ldots,s_n\sigma)\\
\bot\sigma &= \bot\\
\top\sigma &= \top\\
P(s_1,\ldots,s_n)\sigma &= P(s_1\sigma,\ldots,s_n\sigma)\\
(u\approx v)\sigma &= (u\sigma\approx v\sigma)\\
\neg F\sigma &= \neg(F\sigma)\\
(F\circ G)\sigma &= (F\sigma\ \circ\ G\sigma)\ ;\quad \text{for each binary connective } \circ\\
(\mathsf{Q}x\,F)\sigma &= \mathsf{Q}z\,(F\,\sigma[x\mapsto z])\ ;\quad \text{with } z \text{ a fresh variable}
\end{aligned}
$$

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 Σ-algebra (also called Σ-interpretation or Σ-structure) is a triple

$$\mathcal{A} = (U_{\mathcal{A}},\ (f_{\mathcal{A}} : U_{\mathcal{A}}^n \to U_{\mathcal{A}})_{f/n \in \Omega},\ (P_{\mathcal{A}} \subseteq U_{\mathcal{A}}^m)_{P/m \in \Pi})$$

where $U_{\mathcal{A}} \neq \emptyset$ is a set, called the universe of $\mathcal{A}$.

By Σ-Alg we denote the class of all Σ-algebras.

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, also called a valuation (over a given Σ-algebra $\mathcal{A}$), is a map $\beta : X \to U_{\mathcal{A}}$.

Variable assignments are the semantic counterparts of substitutions.

Value of a Term in $\mathcal{A}$ with Respect to β

By structural induction we define

$$\mathcal{A}(\beta) : \mathsf{T}_\Sigma(X) \to U_\mathcal{A}$$

as follows:

$$\begin{aligned} \mathcal{A}(\beta)(x) &= \beta(x), && x \in X \\ \mathcal{A}(\beta)(f(s_1, \ldots, s_n)) &= f_\mathcal{A}(\mathcal{A}(\beta)(s_1), \ldots, \mathcal{A}(\beta)(s_n)), && f/n \in \Omega \end{aligned}$$

Value of a Term in $\mathcal{A}$ with Respect to β

In the scope of a quantifier we need to evaluate terms with respect to modified assignments. To that end, let $\beta[x \mapsto a] : X \to U_{\mathcal{A}}$, 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 β

$\mathcal{A}(\beta) : \mathrm{F}_\Sigma(X) \to \{0, 1\}$ is defined inductively as follows:

$$
\begin{aligned}
\mathcal{A}(\beta)(\bot) &= 0 \\
\mathcal{A}(\beta)(\top) &= 1 \\
\mathcal{A}(\beta)(P(s_1, \ldots, s_n)) &= \text{if } (\mathcal{A}(\beta)(s_1), \ldots, \mathcal{A}(\beta)(s_n)) \in P_\mathcal{A} \\
& \text{ then 1 else 0} \\
\mathcal{A}(\beta)(s \approx t) &= \text{if } \mathcal{A}(\beta)(s) = \mathcal{A}(\beta)(t) \text{ then 1 else 0}
\end{aligned}
$$

Truth Value of a Formula in $\mathcal{A}$ with Respect to β

$\mathcal{A}(\beta) : \mathsf{F}_\Sigma(X) \to \{0, 1\}$ is defined inductively as follows:

$$
\begin{aligned}
\mathcal{A}(\beta)(\neg F) &= 1 - \mathcal{A}(\beta)(F) \\
\mathcal{A}(\beta)(F \wedge G) &= \min(\mathcal{A}(\beta)(F), \mathcal{A}(\beta)(G)) \\
\mathcal{A}(\beta)(F \vee G) &= \max(\mathcal{A}(\beta)(F), \mathcal{A}(\beta)(G)) \\
\mathcal{A}(\beta)(F \rightarrow G) &= \max(1 - \mathcal{A}(\beta)(F), \mathcal{A}(\beta)(G)) \\
\mathcal{A}(\beta)(F \leftrightarrow G) &= \text{if } \mathcal{A}(\beta)(F) = \mathcal{A}(\beta)(G) \text{ then } 1 \text{ else } 0 \\
\mathcal{A}(\beta)(\forall x\, F) &= \min_{a \in U_\mathcal{A}} \{\mathcal{A}(\beta[x \mapsto a])(F)\} \\
\mathcal{A}(\beta)(\exists x\, F) &= \max_{a \in U_\mathcal{A}} \{\mathcal{A}(\beta[x \mapsto a])(F)\}
\end{aligned}
$$

Example

The "Standard" Interpretation for Peano Arithmetic:

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

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

Example

Values over $\mathbb{N}$ for sample terms and formulas:

Under the assignment $\beta : x \mapsto 1, y \mapsto 3$ we obtain

$$
\begin{array}{lcl}
\mathbb{N}(\beta)(s(x) + s(0)) & = & 3 \\
\mathbb{N}(\beta)(x + y \approx s(y)) & = & 1 \\
\mathbb{N}(\beta)(\forall x, y\,(x + y \approx y + x)) & = & 1 \\
\mathbb{N}(\beta)(\forall z\,(z < y)) & = & 0 \\
\mathbb{N}(\beta)(\forall x \exists y\,(x < y)) & = & 1
\end{array}
$$

Ground Terms and Closed Formulas

If t is a ground term, then $\mathcal{A}(\beta)(t)$ does not depend on β:

$$\mathcal{A}(\beta)(t) = \mathcal{A}(\beta')(t)$$

for every β and β'.

Analogously, if F is a closed formula, then $\mathcal{A}(\beta)(F)$ does not depend on β:

$$\mathcal{A}(\beta)(F) = \mathcal{A}(\beta')(F)$$

for every β and β'.

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 β:

$$\mathcal{A}, \beta \models F \quad :\Leftrightarrow \quad \mathcal{A}(\beta)(F) = 1$$

F is true in $\mathcal{A}$ ($\mathcal{A}$ is a model of F; F is valid in $\mathcal{A}$):

$$\mathcal{A} \models F \quad :\Leftrightarrow \quad \mathcal{A}, \beta \models F \ \text{ for all } \beta \in X \to U_{\mathcal{A}}$$

F is valid (or is a tautology):

$$\models F \quad :\Leftrightarrow \quad \mathcal{A} \models F \ \text{ for all } \mathcal{A} \in \Sigma\text{-Alg}$$

F is called satisfiable iff there exist $\mathcal{A}$ and β such that $\mathcal{A}, \beta \models F$. Otherwise F is called unsatisfiable.

Substitution Lemma

The following propositions, to be proved by structural induction, hold for all Σ-algebras $\mathcal{A}$, assignments β, and substitutions σ.

Lemma 3.1:
For any Σ-term t

$$\mathcal{A}(\beta)(t\sigma) = \mathcal{A}(\beta \circ \sigma)(t),$$

where $\beta \circ \sigma : X \to U_{\mathcal{A}}$ is the assignment $\beta \circ \sigma(x) = \mathcal{A}(\beta)(x\sigma)$.

Proposition 3.2:
For any Σ-formula F, $\mathcal{A}(\beta)(F\sigma) = \mathcal{A}(\beta \circ \sigma)(F)$.

Substitution Lemma

Corollary 3.3:

$\mathcal{A}, \beta \models F\sigma \quad \Leftrightarrow \quad \mathcal{A}, \beta \circ \sigma \models F$

These theorems basically express that the syntactic concept of substitution corresponds to the semantic concept of an assignment.

Entailment and Equivalence

F entails (implies) G (or G is a consequence of F), written $F \models G$, if for all $\mathcal{A} \in \Sigma$-Alg and $\beta \in X \to U_{\mathcal{A}}$, whenever $\mathcal{A}, \beta \models F$, then $\mathcal{A}, \beta \models G$.

F and G are called equivalent, written $F \models\!\!\!\mid G$, if for all $\mathcal{A} \in \Sigma$-Alg and $\beta \in X \to U_{\mathcal{A}}$ we have
$\mathcal{A}, \beta \models F \quad \Leftrightarrow \quad \mathcal{A}, \beta \models G$.

Entailment and Equivalence

Proposition 3.4:
F entails G iff $(F \rightarrow G)$ is valid

Proposition 3.5:
F and G are equivalent iff $(F \leftrightarrow G)$ is valid.

Extension to sets of formulas N in the "natural way", e. g.,
$N \models F$

$:\Leftrightarrow$ for all $\mathcal{A} \in \Sigma$-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 3.6:
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.

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 β such that $\mathcal{A}, \beta \models F$.

Solve(F): find a substitution σ such that $\models F\sigma$.

Abduce(F): find G with "certain properties" such that $G \models F$.

Theory of an Algebra

Let $\mathcal{A} \in \Sigma$-Alg. The (first-order) theory of $\mathcal{A}$ is defined as

$$\mathrm{Th}(\mathcal{A}) = \{\, G \in \mathrm{F}_\Sigma(X) \mid \mathcal{A} \models G \,\}$$

Problem of axiomatizability:

For which algebras $\mathcal{A}$ can one axiomatize $\mathrm{Th}(\mathcal{A})$, that is, can one write down a formula F (or a recursively enumerable set F of formulas) such that

$$\mathrm{Th}(\mathcal{A}) = \{\, G \mid F \models G \,\}?$$

(analogously for classes of algebras).

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.
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 Th($\mathbb{Z}_+$) $\notin$ NTIME($2^{2^{cn}}$)).

Two Interesting Theories

However, $\mathbb{N}_* = (\mathbb{N}, 0, s, +, *, <)$, the standard interpretation of $\Sigma_{\mathsf{PA}} = (\{0/0, s/1, +/2, */2\}, \{<\})$, has as theory the so-called Peano arithmetic which is undecidable and not even recursively enumerable.

(Non-)Computability Results

1. For most signatures Σ, validity is undecidable for Σ-formulas.
   (One can easily encode Turing machines in most signatures.)

2. Gödel's completeness theorem:
   For each signature Σ, the set of valid Σ-formulas is recursively enumerable.
   (We will prove this by giving complete deduction systems.)

3. Gödel's incompleteness theorem:
   For $\Sigma = \Sigma_{\mathsf{PA}}$ and $\mathbb{N}_* = (\mathbb{N}, 0, s, +, *, <)$, the theory $\mathrm{Th}(\mathbb{N}_*)$ 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

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

where F is quantifier-free and $Q_i \in \{\forall, \exists\}$;
we call $Q_1 x_1 \ldots Q_n x_n$ the quantifier prefix and F the matrix of the formula.

Prenex Normal Form (Traditional)

Computing prenex normal form by the reduction system $\Rightarrow_P$:

$$
\begin{array}{rcl}
H[(F \leftrightarrow G)]_p & \Rightarrow_P & H[(F \rightarrow G) \wedge (G \rightarrow F)]_p \\
H[\neg \mathrm{Q}x\, F]_p & \Rightarrow_P & H[\overline{\mathrm{Q}}x\, \neg F]_p \\
H[((\mathrm{Q}x\, F) \;\circ\; G)]_p & \Rightarrow_P & H[\mathrm{Q}y\,(F\{x \mapsto y\} \;\circ\; G)]_p, \\
 & & \quad \circ \in \{\wedge, \vee\} \\
H[((\mathrm{Q}x\, F) \rightarrow G)]_p & \Rightarrow_P & H[\overline{\mathrm{Q}}y\,(F\{x \mapsto y\} \rightarrow G)]_p, \\
H[(F \;\circ\; (\mathrm{Q}x\, G))]_p & \Rightarrow_P & H[\mathrm{Q}y\,(F \;\circ\; G\{x \mapsto y\})]_p, \\
 & & \quad \circ \in \{\wedge, \vee, \rightarrow\}
\end{array}
$$

Here y is always assumed to be some fresh variable and $\overline{\mathrm{Q}}$ denotes the quantifier dual to Q, i. e., $\overline{\forall} = \exists$ and $\overline{\exists} = \forall$.

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):

$$\forall x_1, \ldots, x_n \exists y\, F \quad \Rightarrow_S \quad \forall x_1, \ldots, x_n\, F\{y \mapsto f(x_1, \ldots, x_n)\}$$

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 3.7:

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. Σ-Alg) $\Leftrightarrow$ H satisfiable (w. r. t. Σ'-Alg) where $\Sigma' = (\Omega \cup SKF, \Pi)$ if $\Sigma = (\Omega, \Pi)$.

The Complete Picture

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

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

Note: The variables in the clauses are implicitly universally quantified.

The Complete Picture

Theorem 3.8:

Let F be closed. Then $F' \models F$.

(The converse is not true in general.)

Theorem 3.9:

Let F be closed. Then F is satisfiable iff F' is satisfiable iff 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 $\bot$ essentially as in the propositional case modulo associativity/commutativity of $\wedge$, $\vee$:

$$
\begin{aligned}
H[(F \wedge \top)]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \vee \bot)]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \leftrightarrow \bot)]_p &\Rightarrow_{\mathrm{OCNF}} H[\neg F]_p \\
H[(F \leftrightarrow \top)]_p &\Rightarrow_{\mathrm{OCNF}} H[F]_p \\
H[(F \vee \top)]_p &\Rightarrow_{\mathrm{OCNF}} H[\top]_p \\
H[(F \wedge \bot)]_p &\Rightarrow_{\mathrm{OCNF}} H[\bot]_p \\
H[\neg\top]_p &\Rightarrow_{\mathrm{OCNF}} H[\bot]_p \\
H[\neg\bot]_p &\Rightarrow_{\mathrm{OCNF}} H[\top]_p
\end{aligned}
$$

Elimination of Trivial Subformulas

Eliminate subformulas $\top$ and $\bot$ essentially as in the propositional case modulo associativity/commutativity of $\wedge$, $\vee$:

$$\begin{aligned}
H[(F \rightarrow \bot)]_p &\Rightarrow_{\text{OCNF}} H[\neg F]_p \\
H[(F \rightarrow \top)]_p &\Rightarrow_{\text{OCNF}} H[\top]_p \\
H[(\bot \rightarrow F)]_p &\Rightarrow_{\text{OCNF}} H[\top]_p \\
H[(\top \rightarrow F)]_p &\Rightarrow_{\text{OCNF}} H[F]_p \\
H[\mathrm{Q}x\, \top]_p &\Rightarrow_{\text{OCNF}} H[\top]_p \\
H[\mathrm{Q}x\, \bot]_p &\Rightarrow_{\text{OCNF}} H[\bot]_p
\end{aligned}$$

Replacement of Beneficial Subformulas

The functions ν 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

$$\begin{array}{ll}
\forall x_1,\dots,x_n\,(P(x_1,\dots,x_n) \rightarrow F), & \text{if } \text{pol}(H,p)=1,\\
\forall x_1,\dots,x_n\,(F \rightarrow P(x_1,\dots,x_n)), & \text{if } \text{pol}(H,p)=-1,\\
\forall x_1,\dots,x_n\,(P(x_1,\dots,x_n) \leftrightarrow F), & \text{if } \text{pol}(H,p)=0.
\end{array}$$

Replacement of Beneficial Subformulas

As in the propositional case, one can test $\nu(H[F]_p) > \nu(H[P]_p \wedge \mathrm{def}(H, p, P, F))$ in constant time without actually computing ν.

Negation Normal Form (NNF)

Apply the reduction system $\Rightarrow_{\mathsf{NNF}}$:

$$H[F \leftrightarrow G]_p \;\Rightarrow_{\mathsf{NNF}}\; H[(F \rightarrow G) \wedge (G \rightarrow F)]_p$$

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

$$H[F \leftrightarrow G]_p \;\Rightarrow_{\mathsf{NNF}}\; H[(F \wedge G) \vee (\neg G \wedge \neg F)]_p$$

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

$$H[F \rightarrow G]_p \;\Rightarrow_{\mathsf{NNF}}\; H[\neg F \vee G]_p$$

Negation Normal Form (NNF)

$$
\begin{aligned}
H[\neg\neg F]_p &\Rightarrow_{\mathrm{NNF}} H[F]_p \\
H[\neg(F \vee G)]_p &\Rightarrow_{\mathrm{NNF}} H[\neg F \wedge \neg G]_p \\
H[\neg(F \wedge G)]_p &\Rightarrow_{\mathrm{NNF}} H[\neg F \vee \neg G]_p \\
H[\neg \mathrm{Q}x\ F]_p &\Rightarrow_{\mathrm{NNF}} H[\overline{\mathrm{Q}}x\ \neg F]_p
\end{aligned}
$$

Miniscoping

Apply the reduction system $\Rightarrow_{\mathrm{MS}}$ 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:

$$
\begin{array}{lcl}
H[\mathrm{Q}x\,(F \wedge G)]_p & \Rightarrow_{\mathrm{MS}} & H[(\mathrm{Q}x\,F) \wedge G]_p \\
H[\mathrm{Q}x\,(F \vee G)]_p & \Rightarrow_{\mathrm{MS}} & H[(\mathrm{Q}x\,F) \vee G]_p \\
H[\forall x\,(F \wedge F')]_p & \Rightarrow_{\mathrm{MS}} & H[(\forall x\,F) \wedge (\forall x\,F')]_p \\
H[\exists x\,(F \vee F')]_p & \Rightarrow_{\mathrm{MS}} & H[(\exists x\,F) \vee (\exists x\,F')]_p \\
H[\mathrm{Q}x\,G]_p & \Rightarrow_{\mathrm{MS}} & H[G]_p
\end{array}
$$

Variable Renaming

Rename all variables in H such that there are no two different positions p, q with $H|_p = \mathsf{Q}x\, F$ and $H|_q = \mathsf{Q}'x\, G$.

Standard Skolemization

Apply the reduction system:

$$H[\exists x\, F]_p \Rightarrow_{\mathrm{SK}} H[F\{x \mapsto f(y_1, \ldots, y_n)\}]_p$$

where p has minimal length,

$\{y_1, \ldots, y_n\}$ 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:

$$
\begin{aligned}
H[(\forall x\, F) \wedge G]_p \ &\Rightarrow_{\mathrm{OCNF}}\ H[\forall x\,(F \wedge G)]_p \\
H[(\forall x\, F) \vee G]_p \ &\Rightarrow_{\mathrm{OCNF}}\ H[\forall x\,(F \vee G)]_p
\end{aligned}
$$

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:

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

3.7 Herbrand Interpretations

From now on we shall consider FOL without equality.
We assume that Ω contains at least one constant symbol.

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

- $U_{\mathcal{A}} = \mathsf{T}_\Sigma$ (= the set of ground terms over Σ)
- $f_{\mathcal{A}} : (s_1, \ldots, s_n) \mapsto f(s_1, \ldots, s_n)$, $f/n \in \Omega$

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 $P_{\mathcal{A}} \subseteq \mathsf{T}_\Sigma^m$.

Herbrand Interpretations

Proposition 3.10:
Every set of ground atoms I uniquely determines a Herbrand interpretation $\mathcal{A}$ via

$$(s_1, \ldots, s_n) \in P_{\mathcal{A}} \;\text{ iff }\; P(s_1, \ldots, s_n) \in I$$

Thus we shall identify Herbrand interpretations (over Σ) with sets of Σ-ground atoms.

Existence of Herbrand Models

A Herbrand interpretation I is called a Herbrand model of F, if $I \models F$.

Theorem 3.11 (Herbrand):
Let N be a set of (universally quantified) Σ-clauses.

$$
\begin{aligned}
N \text{ satisfiable} \quad &\Leftrightarrow \quad N \text{ has a Herbrand model (over } \Sigma) \\
&\Leftrightarrow \quad G_\Sigma(N) \text{ has a Herbrand model (over } \Sigma)
\end{aligned}
$$

where $G_\Sigma(N) = \{\, C\sigma \text{ ground clause} \mid (\forall \vec{x}\, C) \in N,\ \sigma : X \to \mathrm{T}_\Sigma \,\}$ is the set of ground instances of N.

[The proof will be given below in the context of the completeness proof for general resolution.]

3.8 Inference Systems and Proofs

Inference systems Γ (proof calculi) are sets of tuples

$$(F_1, \ldots, F_n, F_{n+1}), \;\; n \geq 0,$$

called inferences, and written

$$\frac{\overbrace{F_1 \;\; \ldots \;\; F_n}^{\text{premises}}}{\underbrace{F_{n+1}}_{\text{conclusion}}} \; .$$

Clausal inference system: premises and conclusions are clauses.

One also considers inference systems over other data structures.

Inference Systems

Inference systems Γ are shorthands for reduction systems over sets of formulas. If N is a set of formulas, then

$$\frac{\overbrace{F_1 \quad \dots \quad F_n}^{\text{premises}}}{\underbrace{F_{n+1}}_{\text{conclusion}}} \quad \textit{side condition}$$

is a shorthand for

$$N \cup \{F_1, \dots, F_n\} \quad \Rightarrow_\Gamma \quad N \cup \{F_1, \dots, F_n\} \cup \{F_{n+1}\}$$

if *side condition*

Proofs

A proof in Γ 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

$$\frac{F_{m_1} \quad \dots \quad F_{m_n}}{F_i}$$

in Γ, such that $0 \leq m_j < i$, for $1 \leq j \leq n$.

Soundness and Completeness

Provability $\vdash_\Gamma$ of F from N in Γ:

$$N \vdash_\Gamma F \ \text{ if there exists a proof in } \Gamma \text{ of } F \text{ from } N.$$

Γ is called sound, if

$$\frac{F_1 \ \ldots \ F_n}{F} \in \Gamma \ \text{ implies } F_1, \ldots, F_n \models F$$

Γ is called complete, if

$$N \models F \ \text{ implies } N \vdash_\Gamma F$$

Γ is called refutationally complete, if

$$N \models \bot \ \text{ implies } N \vdash_\Gamma \bot$$

Soundness and Completeness

Proposition 3.12:

(i) Let Γ be sound. Then $N \vdash_\Gamma F \Rightarrow N \models F$

(ii) If $N \vdash_\Gamma F$ then there exist finitely many $F_1, \ldots, F_n \in N$ such that $F_1, \ldots, F_n \vdash_\Gamma F$

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.

Reduced Proofs as Trees

markings	$\widehat{=}$	formulas
leaves	$\widehat{=}$	assumptions and axioms
other nodes	$\widehat{=}$	inferences: conclusion $\widehat{=}$ parent node
		premises $\widehat{=}$ child nodes

$$
\dfrac{P(f(c)) \qquad \dfrac{\dfrac{\dfrac{P(f(c)) \vee Q(b) \qquad \dfrac{\dfrac{P(f(c)) \vee Q(b) \quad \neg P(f(c)) \vee \neg P(f(c)) \vee Q(b)}{\neg P(f(c)) \vee Q(b) \vee Q(b)}}{\neg P(f(c)) \vee Q(b)}}{Q(b) \vee Q(b)}}{Q(b)} \qquad \neg P(f(c)) \vee \neg Q(b)}{\neg P(f(c))}}{\bot}
$$

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.

Note that 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:

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

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, $C = \bot$.

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 3.13:
Propositional resolution is sound.

Note: In ground first-order logic we have (like in propositional logic):

1. $\mathcal{B} \models L_1 \vee \ldots \vee L_n$ 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: $N \models \bot \;\Rightarrow\; N \vdash_{Res} \bot$,
  or equivalently: If $N \not\vdash_{Res} \bot$, then N has a model.

- Idea: Suppose that we have computed sufficiently many inferences (and not derived $\bot$).

- 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.

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 lenght-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:

$$\begin{array}{lcll} [\neg]A & \succ_L & [\neg]B & \text{, if } A \succ B \\ \neg A & \succ_L & A & \end{array}$$

3. Extend $\succ_L$ to an ordering $\succ_C$ on ground clauses:
   $\succ_C = (\succ_L)_{\mathrm{mul}}$, 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:

$$
\begin{array}{rc}
 & A_1 \lor \neg A_5 \\
\succ & A_3 \lor \neg A_4 \\
\succ & \neg A_1 \lor A_3 \lor A_4 \\
\succ & A_1 \lor \neg A_2 \\
\succ & \neg A_1 \lor A_2 \\
\succ & A_1 \lor A_1 \lor A_2 \\
\succ & A_0 \lor A_1
\end{array}
$$

Properties of the Clause Ordering

Proposition 3.14:

1. The orderings on literals and clauses are total and well-founded.

2. Let C and D be clauses with $A = \max(C)$, $B = \max(D)$, where $\max(C)$ 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 $B \succ A$. Clause sets are then stratified in this form:

B	$\neg B \vee \ldots$ $\ldots \vee B \vee B$ $\ldots \vee B$	all clauses D with $\max(D) = B$
$\succ$	$\vdots$	
A	$\neg A \vee \ldots$ $\ldots \vee A \vee A$ $\ldots \vee A$	all clauses C with $\max(C) = A$

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, iff for each set N of ground clauses:

$$N \models \bot \text{ implies } \bot \in Res^*(N)$$

Construction of Interpretations

Given: set N of ground clauses, atom ordering $\succ$.

Wanted: Herbrand interpretation I such that

- "many" clauses from N are valid in I;
- $I \models N$, *if* N *is saturated and* $\bot \notin N$.

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 a partial interpretation I_C (initially $I_C = \emptyset$) available.
- If C is true in the partial interpretation I_C, nothing is done. $(\Delta_C = \emptyset)$.
- If C is false, one would like to change I_C such that C becomes true.

Main Ideas of the Construction

- Changes should, however, be *monotone*. One never deletes anything from I_C and the truth value of clauses smaller than C should remain as it was in I_C.

- Hence, one chooses $\Delta_C = \{A\}$ 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*).

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 Δ_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)

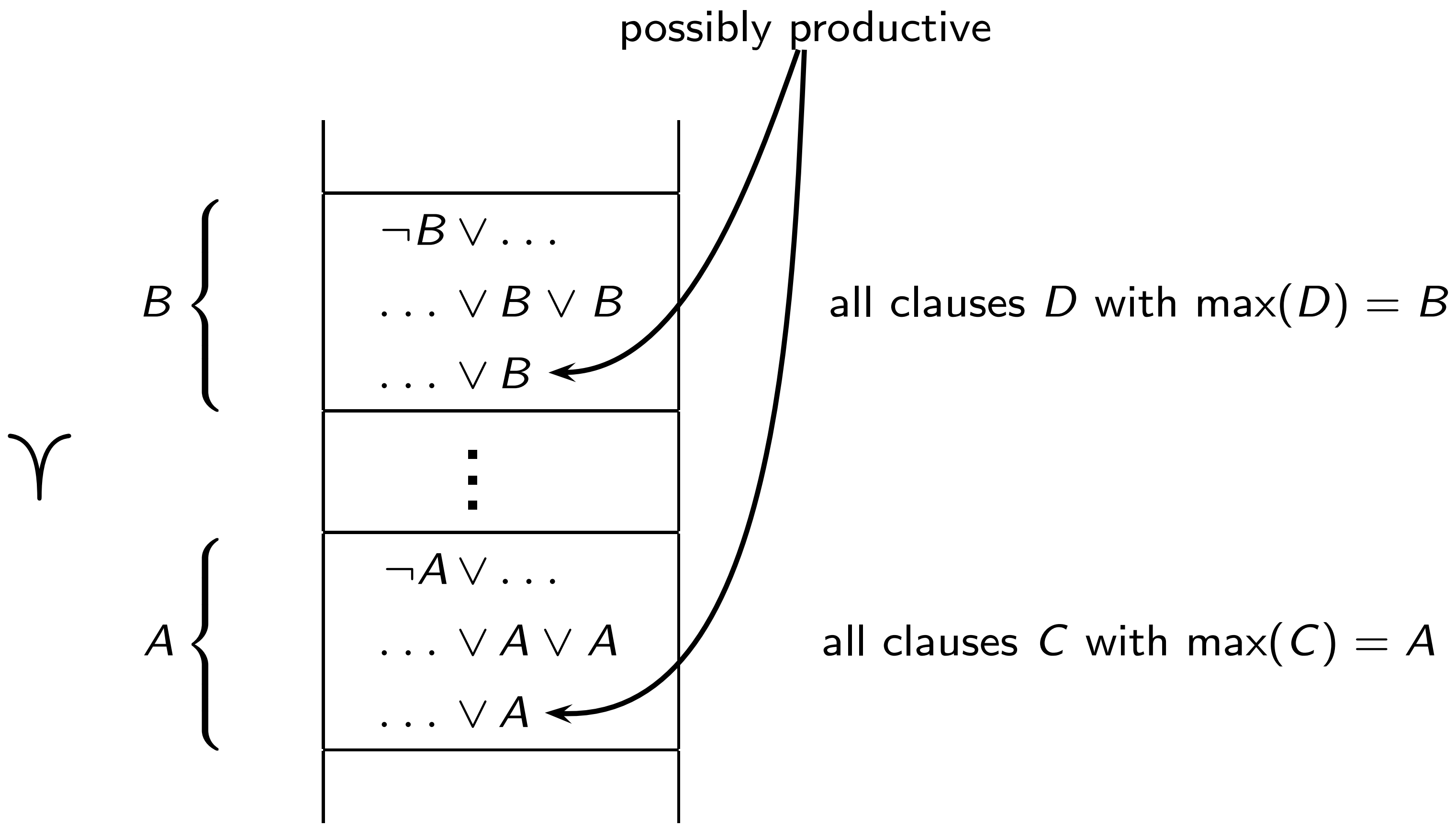
	clauses C	I_C	Δ_C	Remarks
7	$\neg A_1 \lor A_5$	$\{A_1, A_2, A_4\}$	$\{A_5\}$	
6	$\neg A_1 \lor A_3 \lor \neg A_4$	$\{A_1, A_2, A_4\}$	$\emptyset$	max. lit. $\neg A_4$ neg.; *min. counter-ex.*
5	$A_0 \lor \neg A_1 \lor A_3 \lor A_4$	$\{A_1, A_2\}$	$\{A_4\}$	A_4 maximal
4	$\neg A_1 \lor A_2$	$\{A_1\}$	$\{A_2\}$	A_2 maximal
3	$A_1 \lor A_2$	$\{A_1\}$	$\emptyset$	true in I_C
2	$A_0 \lor A_1$	$\emptyset$	$\{A_1\}$	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 $B \succ A$. Note that producing a new atom does not change the truth value of smaller clauses.



Some Properties of the Construction

Proposition 3.16:

(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 $I_N^{\succ} \models D$.

(iii) If $D = D' \vee A$ produces A,
then $I_C \models D$ for every $C \succ D$ and $I_N^{\succ} \models D$.

Some Properties of the Construction

(iv) If $D = D' \vee A$ produces A,
then $I_C \not\models D'$ for every $C \succeq D$ and $I_N^{\succ} \not\models D'$.

(v) If for every clause $C \in N$, C is productive or $I_C \models C$,
then $I_N^{\succ} \models N$.

Model Existence Theorem

Proposition 3.17:

Let $\succ$ be a clause ordering.

If N is saturated w. r. t. *Res* and $\bot \notin N$,

then for every clause $C \in N$, C is productive or $I_C \models C$.

Theorem 3.18 (Bachmair & Ganzinger 1990):

Let $\succ$ be a clause ordering.

If N is saturated w. r. t. *Res* and $\bot \notin N$, then $I_N^{\succ} \models N$.

Corollary 3.19:

Let N be saturated w. r. t. *Res*.

Then $N \models \bot$ if and only if $\bot \in N$.

Compactness of Propositional Logic

Lemma 3.20:
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 3.21 (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.

Two Lemmas

Lemma 3.22:
Let $\mathcal{A}$ be a Σ-algebra and let F be a Σ-formula with free variables $x_1, \ldots, x_n$. Then

$$\mathcal{A} \models \forall x_1, \ldots, x_n\, F \quad \text{if and only if} \quad \mathcal{A} \models F$$

Lemma 3.23:
Let F be a Σ-formula with free variables $x_1, \ldots, x_n$,
let σ be a substitution,
and let $y_1, \ldots, y_m$ be the free variables of $F\sigma$. Then

$$\mathcal{A} \models \forall x_1, \ldots, x_n\, F \quad \text{implies} \quad \mathcal{A} \models \forall y_1, \ldots, y_m\, F\sigma$$

Two Lemmas

In particular, if $\mathcal{A}$ is a model of an (implicitly universally quantified) clause C, then 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:

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

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
$\Rightarrow$ 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
$\Rightarrow$ lifting principle.

Computation of instances becomes a by-product of boolean reasoning.

General Resolution through Instantiation

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

Unification

Let $E = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ (s_i, t_i terms or atoms) be a multiset of equality problems. A substitution σ 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 σ is called more general than a substitution τ, denoted by $\sigma \leq \tau$, if there exists a substitution ρ such that $\rho \circ \sigma = \tau$, where $(\rho \circ \sigma)(x) := (x\sigma)\rho$ is the composition of σ and ρ 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 $\mathrm{mgu}(E)$.

Unification

Proposition 3.24:

(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 σ is called idempotent, if $\sigma \circ \sigma = \sigma$.

Proposition 3.25:
σ is idempotent iff $\mathrm{dom}(\sigma) \cap \mathrm{codom}(\sigma) = \emptyset$.

Rule-Based Naive Standard Unification

$$
\begin{array}{rll}
t \doteq t, E & \Rightarrow_{SU} & E \\[1em]
f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n), E & \Rightarrow_{SU} & s_1 \doteq t_1, \ldots, s_n \doteq t_n, E \\[1em]
f(\ldots) \doteq g(\ldots), E & \Rightarrow_{SU} & \bot \\
& & \text{if } f \neq g \\[1em]
x \doteq t, E & \Rightarrow_{SU} & x \doteq t, E\{x \mapsto t\} \\
& & \text{if } x \in \mathrm{var}(E), x \notin \mathrm{var}(t) \\[1em]
x \doteq t, E & \Rightarrow_{SU} & \bot \\
& & \text{if } x \neq t, x \in \mathrm{var}(t) \\[1em]
t \doteq x, E & \Rightarrow_{SU} & x \doteq t, E \\
& & \text{if } t \notin X
\end{array}
$$

SU: Main Properties

If $E = \{x_1 \doteq u_1, \ldots, x_k \doteq u_k\}$, with x_i pairwise distinct, $x_i \notin \mathrm{var}(u_j)$, then E is called an (equational problem in) solved form representing the solution $\sigma_E = \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k\}$.

Proposition 3.26:
If E is a solved form then σ_E is an mgu of E.

SU: Main Properties

Theorem 3.27:

1. If $E \Rightarrow_{SU} E'$ then σ is a unifier of E iff σ 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 3.28:
E is unifiable if and only if there is a most general unifier σ of E, such that σ is idempotent and $\mathrm{dom}(\sigma) \cup \mathrm{codom}(\sigma) \subseteq \mathrm{var}(E)$.

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

$$
\begin{array}{rcl}
t \doteq t, E & \Rightarrow_{PU} & E \\[1em]
f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n), E & \Rightarrow_{PU} & s_1 \doteq t_1, \ldots, s_n \doteq t_n, E \\[1em]
f(\ldots) \doteq g(\ldots), E & \Rightarrow_{PU} & \bot \\
& & \text{if } f \neq g \\[1em]
x \doteq y, E & \Rightarrow_{PU} & x \doteq y, E\{x \mapsto y\} \\
& & \text{if } x \in \operatorname{var}(E), x \neq y \\[1em]
x_1 \doteq t_1, \ldots, x_n \doteq t_n, E & \Rightarrow_{PU} & \bot \\
& & \text{if there are positions } p_i \text{ with} \\
& & t_i|_{p_i} = x_{i+1}, t_n|_{p_n} = x_1 \\
& & \text{and some } p_i \neq \varepsilon
\end{array}
$$

Rule-Based Polynomial Unification

$$
\begin{array}{rcl}
x \doteq t, E & \Rightarrow_{PU} & \bot \\
 & & \text{if } x \neq t, x \in \mathrm{var}(t) \\
t \doteq x, E & \Rightarrow_{PU} & x \doteq t, E \\
 & & \text{if } t \notin X \\
x \doteq t, x \doteq s, E & \Rightarrow_{PU} & x \doteq t, t \doteq s, E \\
 & & \text{if } t, s \notin X \text{ and } |t| \leq |s|
\end{array}
$$

Properties of PU

Theorem 3.29:

1. If $E \Rightarrow_{PU} E'$ then σ is a unifier of E iff σ 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 unifiabilty:

General resolution ***Res*****:**

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

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

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 3.30:

Let C and D be variable-disjoint clauses. If

$$\begin{array}{ccc} D & & C \\ \big\downarrow \sigma & & \big\downarrow \rho \\ \end{array}$$

$$\frac{D\sigma \qquad C\rho}{C'} \qquad \text{[ground resolution]}$$

then there exists a substitution τ such that

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

$$\big\downarrow \tau$$

$$C' = C''\tau$$

Lifting Lemma

An analogous lifting lemma holds for factorization.

Saturation of Sets of General Clauses

Corollary 3.31:

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 3.32:

The general resolution calculus is sound.

Herbrand's Theorem

Lemma 3.33:

Let N be a set of Σ-clauses, let $\mathcal{A}$ be an interpretation.

Then $\mathcal{A} \models N$ implies $\mathcal{A} \models G_\Sigma(N)$.

Lemma 3.34:

Let N be a set of Σ-clauses, let $\mathcal{A}$ be a *Herbrand* interpretation.

Then $\mathcal{A} \models G_\Sigma(N)$ implies $\mathcal{A} \models N$.

Herbrand's Theorem

Theorem 3.35 (Herbrand):
A set N of Σ-clauses is satisfiable if and only if it has a Herbrand model over Σ.

Corollary 3.36:
A set N of Σ-clauses is satisfiable if and only if its set of ground instances $G_\Sigma(N)$ is satisfiable.

Refutational Completeness of General Resolution

Theorem 3.37:

Let N be a set of general clauses that is saturated w. r. t. *Res*. Then $N \models \bot$ if and only if $\bot \in N$.

Proof:

The "$\Leftarrow$" part is trivial. For the "$\Rightarrow$" part assume that N is saturated, that is, $Res(N) \subseteq N$. By Corollary 3.31, $G_\Sigma(N)$ is saturated as well, i. e., $Res(G_\Sigma(N)) \subseteq G_\Sigma(N)$. By Cor. 3.36, $N \models \bot$ implies $G_\Sigma(N) \models \bot$. By the refutational completeness of ground resolution, $G_\Sigma(N) \models \bot$ implies $\bot \in G_\Sigma(N)$, so $\bot \in N$.

$\square$

3.12 Theoretical Consequences

We get some classical results on properties of first-order logic as easy corollaries.

The Theorem of Löwenheim-Skolem

Theorem 3.38 (Löwenheim–Skolem):
Let Σ be a countable signature and let S be a set of closed Σ-formulas. Then S is satisfiable iff 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 κ for every κ that is larger than or equal to the cardinalty of the signature Σ.

Compactness of Predicate Logic

Theorem 3.39 (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 3.18) one only needs to resolve and factor maximal atoms
   $\Rightarrow$ if the calculus is restricted to inferences involving maximal atoms, the proof remains correct
   $\Rightarrow$ *ordering restrictions*

2. In the proof, it does not really matter with which negative literal an inference is performed
   $\Rightarrow$ choose a negative literal don't-care-nondeterministically
   $\Rightarrow$ *selection*

Ordering Restrictions

In the completeness proof one only needs to resolve and factor maximal atoms $\Rightarrow$ If we impose ordering restrictions on ground inferences, the proof remains correct:

(Ground) Ordered Resolution:

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

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 $A \succ B$ iff $A\sigma \succ B\sigma$.
- 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 $\not\preceq$ and $\succeq$ by $\not\prec$.

Ordering Restrictions

Ordered Resolution:

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

if $\sigma = \text{mgu}(A, B)$ and $B\sigma \not\preceq L\sigma$ 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

$$\text{sel} : C \quad \mapsto \quad \text{set of occurrences of } \textit{negative} \text{ literals in } C$$

Example of selection with selected literals indicated as $\boxed{X}$:

$$\boxed{\neg A} \lor \neg A \lor B$$

$$\boxed{\neg B_0} \lor \boxed{\neg B_1} \lor A$$

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 $Res^{\succ}_{\mathrm{sel}}$

The resolution calculus $Res^{\succ}_{\mathrm{sel}}$ 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 $Res^{\succ}_{\text{sel}}$

(Ground) Ordered Resolution with Selection:

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

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 $Res^{\succ}_{\text{sel}}$

(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 $Res^{\succ}_{\mathrm{sel}}$

The extension from ground inferences to non-ground inferences is analogous to ordered resolution (replace $\succ$ by $\not\preceq$ and $\succeq$ by $\not\prec$). Again we assume that $\succ$ is stable under substitutions.

Resolution Calculus $Res_{\text{sel}}^{\succ}$

Ordered Resolution with Selection:

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

if the following conditions are satisfied:

(i) $\sigma = \text{mgu}(A, B)$;

(ii) $B\sigma \not\preceq L\sigma$ for all L in D;

(iii) nothing is selected in $D \vee B$ by sel;

(iv) $\neg A$ is selected in $C \vee \neg A$,
or nothing is selected in $C \vee \neg A$ and
$\neg A\sigma \not\prec L\sigma$ for all L in C.

Resolution Calculus $Res^{\succ}_{\mathrm{sel}}$

Ordered Factorization with Selection:

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

if the following conditions are satisfied:

(i) $\sigma = \mathrm{mgu}(A, B)$;

(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 $Res^{\succ}_{\text{sel}}$

Lemma 3.40:

Let D and C be variable-disjoint clauses. If

$$\begin{array}{ccc} D & & C \\ \downarrow \sigma & & \downarrow \rho \\ \end{array}$$

$$\frac{D\sigma \qquad\qquad C\rho}{C'} \qquad [\text{ground inference in } Res^{\succ}_{\text{sel}}]$$

and if $\text{sel}(D\sigma) \simeq \text{sel}(D)$, $\text{sel}(C\rho) \simeq \text{sel}(C)$ (that is, “corresponding” literals are selected), then there exists a substitution τ such that

$$\frac{D \qquad\qquad C}{C''} \qquad [\text{inference in } Res^{\succ}_{\text{sel}}]$$

$$\downarrow \tau$$

$$C' = C''\tau$$

Lifting Lemma for $Res^{\succ}_{\mathsf{sel}}$

An analogous lifting lemma holds for factorization.

Saturation of Sets of General Clauses

Corollary 3.41:
Let N be a set of general clauses saturated under $\mathit{Res}^{\succ}_{\mathrm{sel}}$, i. e., $\mathit{Res}^{\succ}_{\mathrm{sel}}(N) \subseteq N$. Then there exists a selection function sel' such that $\mathrm{sel}|_N = \mathrm{sel}'|_N$ and $G_\Sigma(N)$ is also saturated, i. e.,

$$\mathit{Res}^{\succ}_{\mathrm{sel}'}(G_\Sigma(N)) \subseteq G_\Sigma(N).$$

Soundness and Refutational Completeness

Theorem 3.42:

Let $\succ$ be an atom ordering and sel a selection function such that $Res^{\succ}_{\text{sel}}(N) \subseteq N$. Then

$$N \models \bot \Leftrightarrow \bot \in N$$

Proof:

The "$\Leftarrow$" part is trivial. For the "$\Rightarrow$" part 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 not 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 3.41. $\square$

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

$$\dfrac{\dfrac{C_1 \vee A \qquad C_2 \vee \neg A \vee B}{C_1 \vee C_2 \vee B} \qquad C_3 \vee \neg B}{C_1 \vee C_2 \vee C_3}$$

we can obtain by rotation

$$\dfrac{C_1 \vee A \qquad \dfrac{C_2 \vee \neg A \vee B \qquad C_3 \vee \neg B}{C_2 \vee \neg A \vee C_3}}{C_1 \vee C_2 \vee C_3}$$

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 3.43 (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 $C_i \prec C$ 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 3.44:
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. $Res^{\succ}_{\text{sel}}$) if

$$Res^{\succ}_{\text{sel}}(N \setminus Red(N)) \subseteq N \cup Red(N)$$

Theorem 3.45:

Let N be saturated up to redundancy. Then

$$N \models \bot \Leftrightarrow \bot \in N$$

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 3.46:

(i) $N \subseteq M \Rightarrow \mathit{Red}(N) \subseteq \mathit{Red}(M)$

(ii) $M \subseteq \mathit{Red}(N) \Rightarrow \mathit{Red}(N) \subseteq \mathit{Red}(N \setminus M)$

Recall that $\mathit{Red}(N)$ 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
$N_0 \vdash N_1 \vdash N_2 \vdash \ldots$, 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 $N_\infty = \bigcup_{i\geq 0} N_i$ and $N_* = \bigcup_{i\geq 0} \bigcap_{j\geq i} N_j$.

The set N_* of all persistent clauses is called the limit of the run.

Computing Saturated Sets

Lemma 3.47:

Let $N_0 \vdash N_1 \vdash N_2 \vdash \ldots$ be a run.

Then $Red(N_i) \subseteq Red(N_\infty)$ and $Red(N_i) \subseteq Red(N_*)$ for every i.

Proof:

Exercise. $\square$

Computing Saturated Sets

Corollary 3.48:
$N_i \subseteq N_* \cup \mathit{Red}(N_*)$ for every i.

Proof:
If $C \in N_i \setminus N_*$, 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_*. □

Computing Saturated Sets

Even if a set N is inconsistent, it could happen that $\bot$ 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 $N_* \setminus \mathit{Red}(N_*)$ is contained in some $N_i \cup \mathit{Red}(N_i)$.

Lemma 3.49:
If a run is fair, then its limit is saturated up to redundancy.

Computing Saturated Sets

Theorem 3.50 (Refutational Completeness: Dynamic View):
Let $N_0 \vdash N_1 \vdash N_2 \vdash \ldots$ be a fair run, let N_* be its limit.
Then N_0 has a model if and only if $\bot \notin N_*$.

Proof:
($\Leftarrow$): By fairness, N_* is saturated up to redundancy.
If $\bot \notin N_*$, then it has a Herbrand model.
Since every clause in N_0 is contained in N_* or redundant w. r. t. N_*, this model is also a model of $G_\Sigma(N_0)$ and therefore a model of N_0.

($\Rightarrow$): Obvious, since $N_0 \models N_*$. $\square$

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 $\mathit{Res}^{\succ}_{\mathrm{sel}}$-inferences from non-redundant premises.
  $\leadsto$ necessary to guarantee fairness

- We add clauses that are entailed by the current ones if this *makes* other clauses redundant:

$$N \cup \{C\} \;\vdash\; N \cup \{C, D\} \;\vdash\; N \cup \{D\}$$

$$\text{if } N \cup \{C\} \models D \text{ and } C \in \mathit{Red}(N \cup \{D\}).$$

  Net effect: C is *simplified* to D
  $\leadsto$ useful to get easier/smaller clause sets

Simplifications

Examples of simplification techniques:

- Deletion of duplicated literals:

$$N \cup \{C \vee L \vee L\} \;\vdash\; N \cup \{C \vee L\}$$

- Subsumption resolution:

$$N \cup \{D \vee L,\; C \vee D\sigma \vee \overline{L}\sigma\} \;\vdash\; N \cup \{D \vee L,\; C \vee D\sigma\}$$

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 $Res^{\succ}_{\mathrm{sel}}$, the calculus is parameterized by an atom ordering $\succ$ and a selection function sel.

Hyperresolution

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

with $\sigma = \mathrm{mgu}(A_1 \doteq B_1, \ldots, A_n \doteq B_n)$, 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 $\leadsto$ "Otter loop":
Advantage: simplifications of U may be useful to derive the empty clause.

no $\leadsto$ "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 calculus.
- Subtle interleaving of enumerating instances and proving inconsistency through the use of unification.
- Parameters: atom ordering $\succ$ and selection function 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
  $\Rightarrow$ fewer proof variants.

- *Global* restrictions of the search space via elimination of redundancy
  $\Rightarrow$ computing with "smaller" / "easier" clause sets;
  $\Rightarrow$ termination on many decidable fragments.

- However: not good enough for dealing with orderings, equality and more specific algebraic theories (lattices, abelian groups, rings, fields)
  $\Rightarrow$ 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 $\{P \wedge \neg(Q \vee \neg R),\ \neg Q \vee \neg R\}$

1. $P \wedge \neg(Q \vee \neg R)$

2. $\neg Q \vee \neg R$

3. $\neg Q$	4. $\neg R$
5. P	10. P
6. $\neg(Q \vee \neg R)$	11. $\neg(Q \vee \neg R)$
7. $\neg Q$	
8. $\neg\neg R$	
9. 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 (unlike, e. g., ordered resolution).

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

$$\frac{\neg\neg F}{F} \qquad \frac{\neg\top}{\bot} \qquad \frac{\neg\bot}{\top}$$

Propositional Expansion Rules

α-**Expansion**

(for formulas that are essentially conjunctions: append subformulas α_1 and α_2 one on top of the other)

$$\frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}}$$

β-**Expansion**

(for formulas that are essentially disjunctions:
append β_1 and β_2 horizontally, i. e., branch into β_1 and β_2)

$$\frac{\beta}{\beta_1 \mid \beta_2}$$

Classification of Formulas

conjunctive			disjunctive		
α	α_1	α_2	β	β_1	β_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 $\{F_1, \ldots, F_n\}$ 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 $\{F_1, \ldots, F_n\}$.
(We do not draw edges if nodes have only one successor.)

Tableaux: Notions

(ii) If T is a tableau for $\{F_1, \ldots, F_n\}$ and if T' results from T by applying an expansion rule then T' is also a tableau for $\{F_1, \ldots, F_n\}$.

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 π in a tableau is called maximal, if for each formula F on π that is neither a literal nor $\bot$ nor $\top$ there exists a node in π at which the expansion rule for F has been applied.

In that case, if F is a formula on π, π also contains:

(i) α_1 and α_2, if F is a α-formula,

(ii) β_1 or β_2, if F is a β-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.

1.	$\neg((P \to (Q \to R)) \to ((P \vee S) \to ((Q \to R) \vee S)))$	
2.	$(P \to (Q \to R))$	$[1_1]$
3.	$\neg((P \vee S) \to ((Q \to R) \vee S))$	$[1_2]$
4.	$P \vee S$	$[3_1]$
5.	$\neg((Q \to R) \vee S))$	$[3_2]$
6.	$\neg(Q \to R)$	$[5_1]$
7.	$\neg S$	$[5_2]$

8. $\neg P$ $[2_1]$ 9. $Q \to R$ $[2_2]$

10. P $[4_1]$ 11. S $[4_2]$

There are three paths, each of them closed.

Properties of Propositional Tableaux

We assume that T is a tableau for $\{F_1, \ldots, F_n\}$.

Theorem 3.51:
$\{F_1, \ldots, F_n\}$ satisfiable $\Leftrightarrow$ some path (i. e., the set of its formulas) in T is satisfiable.

Proof:
($\Leftarrow$) Trivial, since every path contains in particular $F_1, \ldots, F_n$.
($\Rightarrow$) By induction over the structure of T. $\square$

Corollary 3.52:
T closed $\Rightarrow \{F_1, \ldots, F_n\}$ unsatisfiable

Properties of Propositional Tableaux

Theorem 3.53:
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) $\bot \notin \mathcal{H}$, $\neg\top \notin \mathcal{H}$;

(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 3.54 (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. $\square$

Refutational Completeness

Theorem 3.55:
Let π be a maximal open path in a tableau. Then the set of formulas on π is satisfiable.

Proof:
We show that set of formulas on π is a Hintikka set: Conditions (3), (4), (5) follow from the fact that π is maximal; conditions (1) and (2) follow from the fact that π is open and from maximality for the second negation elimination rule. $\square$

Note: The theorem holds also for infinite trees that are obtained as the limit of a tableau expansion.

Refutational Completeness

Theorem 3.56:
$\{F_1, \ldots, F_n\}$ satisfiable $\Leftrightarrow$ there exists no closed strict tableau for $\{F_1, \ldots, F_n\}$.

Proof:
($\Rightarrow$) Clear by Cor. 3.52.
($\Leftarrow$) Let T be a strict maximal tableau for $\{F_1, \ldots, F_n\}$ and let π be an open path in T. By the previous theorem, the set of formulas on π is satisfiable, and hence by Theorem 3.51 the set $\{F_1, \ldots, F_n\}$, is satisfiable. $\square$

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 3.55).
- 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 β-Rule

Since $F \vee G \models\mathrel{\mkern-3mu}\dashv F \vee (G \wedge \neg F)$, the β expansion rule

$$\frac{\beta}{\beta_1 \mid \beta_2}$$

can be replaced by the following variant:

$$\frac{\beta}{\beta_1 \;\Bigg|\; \begin{array}{l} \beta_2 \\ \neg\beta_1 \end{array}}$$

A Variant of the β-Rule

The variant β-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) α-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	
γ	$\gamma(t)$	δ	$\delta(t)$
$\forall xF$	$F\{x \mapsto t\}$	$\exists xF$	$F\{x \mapsto t\}$
$\neg\exists xF$	$\neg F\{x \mapsto t\}$	$\neg\forall xF$	$\neg F\{x \mapsto t\}$

Tableaux with Ground Instantiation

Idea:

Replace universally quantified formulas by appropriate ground instances.

γ-**expansion**

$$\frac{\gamma}{\gamma(t)} \quad \text{where } t \text{ is some ground term}$$

δ-**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 δ-rule would not be needed.

Note:

Skolem *constants* are sufficient:
In a δ-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 γ 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 γ-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

γ-expansion

$$\frac{\gamma}{\gamma(x)} \qquad \text{where } x \text{ is a new free variable}$$

δ-expansion

$$\frac{\delta}{\delta(f(x_1, \ldots, x_n))}$$

where f is a new Skolem function, and the x_i are the free variables in δ

Free-Variable Tableaux

Application of expansion rules has to be supplemented by a substitution rule:

(iii) If T is a tableau for $\{F_1, \ldots, F_n\}$ and if σ is a substitution, then $T\sigma$ is also a tableau for $\{F_1, \ldots, F_n\}$.

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 σ 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.	$\neg\big(\exists w \forall x\ P(x, w, f(x, w)) \to \exists w \forall x \exists y\ P(x, w, y)\big)$	
2.	$\exists w \forall x\ P(x, w, f(x, w))$	1_1 $[\alpha]$
3.	$\neg \exists w \forall x \exists y\ P(x, w, y)$	1_2 $[\alpha]$
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 γ 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 γ-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 $F \to G$, 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

$$F_1, \ldots, F_m \vdash G_1, \ldots, G_k$$

meaning

$$F_1 \wedge \cdots \wedge F_m \rightarrow G_1 \vee \cdots \vee G_k$$

Sequent Calculus

Inferences rules, e.g.:

$$\frac{\Gamma \vdash \Delta}{\Gamma, F \vdash \Delta} \quad (WL) \qquad \frac{\Gamma, F \vdash \Delta \qquad \Sigma, G \vdash \Pi}{\Gamma, \Sigma, F \vee G \vdash \Delta, \Pi} \quad (\vee L)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash F, \Delta} \quad (WR) \qquad \frac{\Gamma \vdash F, \Delta \qquad \Sigma \vdash G, \Pi}{\Gamma, \Sigma \vdash F \wedge G, \Delta, \Pi} \quad (\wedge R)$$

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:

$$\frac{F \qquad F \rightarrow G}{G}$$

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

$$\begin{gathered}
\forall x\,(x \sim x)\\
\forall x, y\,(x \sim y \rightarrow y \sim x)\\
\forall x, y, z\,(x \sim y \wedge y \sim z \rightarrow x \sim z)\\
\forall \vec{x}, \vec{y}\,(x_1 \sim y_1 \wedge \cdots \wedge x_n \sim y_n \rightarrow f(x_1, \ldots, x_n) \sim f(y_1, \ldots, y_n))\\
\forall \vec{x}, \vec{y}\,(x_1 \sim y_1 \wedge \cdots \wedge x_m \sim y_m \wedge P(x_1, \ldots, x_m) \rightarrow P(y_1, \ldots, y_m))
\end{gathered}$$

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

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

  $\rightarrow$ Superposition

  Entailment for clauses with equality

4.2 Rewrite Systems

Let E be a set of (implicitly universally quantified) equations.

The rewrite relation $\rightarrow_E \subseteq \mathrm{T}_\Sigma(X) \times \mathrm{T}_\Sigma(X)$ is defined by

$$\begin{array}{lll} s \rightarrow_E t & \text{iff} & \text{there exist } (l \approx r) \in E,\ p \in \mathrm{pos}(s), \\ & & \text{and } \sigma : X \rightarrow \mathrm{T}_\Sigma(X), \\ & & \text{such that } s|_p = l\sigma \text{ and } t = s[r\sigma]_p. \end{array}$$

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 $\mathrm{var}(l) \supseteq \mathrm{var}(r)$.

Notation: $l \to r$.

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 $\to_E$ or $\to_R$ 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 $\to_E$ to express the semantic consequence relation syntactically:

$s \approx_E t$ if and only if $s \leftrightarrow_E^* t$.

E-Algebras

Let E be a set of equations over $\mathsf{T}_\Sigma(X)$. The following inference system allows to derive consequences of E:

E-Algebras

$E \vdash t \approx t$ (Reflexivity)

for every $t \in \mathrm{T}_\Sigma(X)$

$$\frac{E \vdash t \approx t'}{E \vdash t' \approx t}$$ (Symmetry)

$$\frac{E \vdash t \approx t' \qquad E \vdash t' \approx t''}{E \vdash t \approx t''}$$ (Transitivity)

$$\frac{E \vdash t_1 \approx t'_1 \quad \ldots \quad E \vdash t_n \approx t'_n}{E \vdash f(t_1, \ldots, t_n) \approx f(t'_1, \ldots, t'_n)}$$ (Congruence)

$E \vdash t\sigma \approx t'\sigma$ (Instance)

if $(t \approx t') \in E$ and $\sigma : X \to \mathrm{T}_\Sigma(X)$

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 $t \in \mathrm{T}_\Sigma(X)$ let $[t] = \{\, t' \in \mathrm{T}_\Sigma(X) \mid E \vdash t \approx t' \,\}$ be the congruence class of t.

Define a Σ-algebra $\mathrm{T}_\Sigma(X)/E$ (abbreviated by $\mathcal{T}$) as follows:

$U_\mathcal{T} = \{\, [t] \mid t \in \mathrm{T}_\Sigma(X) \,\}$.

$f_\mathcal{T}([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$ for $f/n \in \Omega$.

E-Algebras

Lemma 4.3:

$f_{\mathcal{T}}$ is well-defined:

If $[t_i] = [t_i']$, then $[f(t_1, \ldots, t_n)] = [f(t_1', \ldots, t_n')]$.

Lemma 4.4:

$\mathcal{T} = \mathsf{T}_\Sigma(X)/E$ is an E-algebra.

Lemma 4.5:

Let X be a countably infinite set of variables; let $s, t \in \mathsf{T}_\Sigma(Y)$.

If $\mathsf{T}_\Sigma(X)/E \models \forall\vec{x}(s \approx t)$, 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 $s, t \in \mathrm{T}_\Sigma(X)$:

(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) $\mathrm{T}_\Sigma(X)/E \models \forall \vec{x}(s \approx t)$.

Universal Algebra

$\mathsf{T}_\Sigma(X)/E = \mathsf{T}_\Sigma(X)/{\approx_E} = \mathsf{T}_\Sigma(X)/{\leftrightarrow_E^*}$ is called the free E-algebra with generating set $X/{\approx_E} = \{\, [x] \mid x \in X \,\}$:

Every mapping $\varphi : X/{\approx_E} \to \mathcal{B}$ for some E-algebra $\mathcal{B}$ can be extended to a homomorphism $\hat{\varphi} : \mathsf{T}_\Sigma(X)/E \to \mathcal{B}$.

$\mathsf{T}_\Sigma(\emptyset)/E = \mathsf{T}_\Sigma(\emptyset)/{\approx_E} = \mathsf{T}_\Sigma(\emptyset)/{\leftrightarrow_E^*}$ 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.

$\approx_E^I = \{ (s, t) \mid \mathrm{T}_\Sigma(\emptyset)/E \models s \approx t \}$
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 $(A, \to)$ be an abstract reduction system.

b and $c \in A$ are joinable, if there is a a such that $b \to^* a \leftarrow^* c$.
Notation: $b \downarrow c$.

The relation $\to$ is called

Church-Rosser, if $b \leftrightarrow^* c$ implies $b \downarrow c$.

confluent, if $b \leftarrow^* a \to^* c$ implies $b \downarrow c$.

locally confluent, if $b \leftarrow a \to c$ 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 $\to$ 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 $l_1 \to r_1$ and $l_2 \to r_2$ 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 σ of $l_1|_p$ and l_2.

Critical Pairs

Let $l_i \to r_i$ $(i = 1, 2)$ be two rewrite rules in a TRS R whose variables have been renamed such that $\mathrm{var}(l_1) \cap \mathrm{var}(l_2) = \emptyset$. (Remember that $\mathrm{var}(l_i) \supseteq \mathrm{var}(r_i)$.)

Let $p \in \mathrm{pos}(l_1)$ be a position such that $l_1|_p$ is not a variable and σ is an mgu of $l_1|_p$ and l_2.

Then $r_1\sigma \leftarrow l_1\sigma \to (l_1\sigma)[r_2\sigma]_p$.

$\langle r_1\sigma, (l_1\sigma)[r_2\sigma]_p\rangle$ 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 $l_i \to r_i \in R$ 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 $t_1 \to_R t_0$ using $l_2 \to r_2$ and $t_2 \to_R t_0$ using $l_1 \to r_1$.

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 $t_1 \to_R^* t_0$ by applying $l_2 \to r_2$ at all positions $p_1\, q'\, q_2$, where q' is a position of x in r_1.

Conversely, $t_2 \to_R^* t_0$ by applying $l_2 \to r_2$ 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 $l_1 \to r_1$ at p_1 with the substitution θ', 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 θ is a unifier of l_2 and $l_1|_p$.

Let σ 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. $\square$

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. $\square$

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 $l \to r \in R$, rather than at infinitely many possible replacement steps $s \to_R s'$.

Reduction Orderings

A binary relation $\sqsupset$ over $T_\Sigma(X)$ is called
compatible with Σ-operations,
if $s \sqsupset s'$ implies $f(t_1, \dots, s, \dots, t_n) \sqsupset f(t_1, \dots, s', \dots, t_n)$
for all $f \in \Omega$ and $s, s', t_i \in T_\Sigma(X)$.

Lemma 4.18:
The relation $\sqsupset$ is compatible with Σ-operations, if and only if
$s \sqsupset s'$ implies $t[s]_p \sqsupset t[s']_p$
for all $s, s', t \in T_\Sigma(X)$ and $p \in \mathrm{pos}(t)$.

Note: compatible with Σ-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 σ.

Reduction Orderings

A binary relation $\sqsupset$ is called a rewrite relation, if it is compatible with Σ-operations and stable under substitutions.

Example: If R is a TRS, then $\rightarrow_R$ is a rewrite relation.

A strict partial ordering over $\mathsf{T}_\Sigma(X)$ 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 Σ-algebra;
let $\succ$ be a well-founded strict partial ordering on its universe.

Define the ordering $\succ_{\mathcal{A}}$ over $\mathrm{T}_{\Sigma}(X)$ by $s \succ_{\mathcal{A}} t$ iff
$\mathcal{A}(\beta)(s) \succ \mathcal{A}(\beta)(t)$ for all assignments $\beta : X \to U_{\mathcal{A}}$.

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 $f : U_{\mathcal{A}}^n \to U_{\mathcal{A}}$ is called monotone (w. r. t. $\succ$), if $a \succ a'$ implies $f(b_1, \ldots, a, \ldots, b_n) \succ f(b_1, \ldots, a', \ldots, b_n)$ 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 Σ-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 $P_f(X_1, \ldots, X_n) \in \mathbb{N}[X_1, \ldots, X_n]$ with coefficients in $\mathbb{N}$ and indeterminates $X_1, \ldots, X_n$. Then we define $f_{\mathcal{A}}(a_1, \ldots, a_n) = P_f(a_1, \ldots, a_n)$ 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 Σ-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 arity$(f) = 0$, then P_f is a constant ≥ 1.

If 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 $m \cdot X_1^{j_1} \cdots X_k^{j_k}$ 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 $x_1, \ldots, x_n$
yields a polynomial P_t with indeterminates $X_1, \ldots, X_n$
(where X_i corresponds to $\beta(x_i)$).

Example:

$$\Omega = \{b/0,\ f/1,\ g/3\}$$
$$P_b = 3, \quad P_f(X_1) = X_1^2, \quad P_g(X_1, X_2, X_3) = X_1 + X_2X_3.$$

Let $t = g(f(b), f(x), y)$, then $P_t(X, Y) = 9 + X^2Y$.

Polynomial Orderings

If P, Q are polynomials in $\mathbb{N}[X_1, \ldots, X_n]$, we write $P > Q$ if $P(a_1, \ldots, a_n) > Q(a_1, \ldots, a_n)$ for all $a_1, \ldots, a_n \in U_{\mathcal{A}}$.

Clearly, $s \succ_{\mathcal{A}} t$ iff $P_s > P_t$ iff $P_s - P_t > 0$.

Question: Can we check $P_s - P_t > 0$ automatically?

Polynomial Orderings

Hilbert's 10th Problem:

> Given a polynomial $P \in \mathbb{Z}[X_1, \ldots, X_n]$ 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. $\square$

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

$k_i \geq 0$ for all $i \in \{1, \dots, n\}$,

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 $\rhd$ is defined by $s \rhd t$ if and only if $s|_p = t$ for some position $p \neq \varepsilon$ of s.

Simplification Orderings

A rewrite ordering $\succ$ over $\mathrm{T}_\Sigma(X)$ is called simplification ordering, if it has the subterm property:
$s \rhd t$ implies $s \succ t$ for all $s, t \in \mathrm{T}_\Sigma(X)$.

Example:

Let R_{emb} be the rewrite system
$R_{\text{emb}} = \{\, f(x_1, \dots, x_n) \to x_i \mid f/n \in \Omega,\ 1 \le i \le n \,\}$.

Define $\rhd_{\text{emb}} = \to^{+}_{R_{\text{emb}}}$ and $\trianglerighteq_{\text{emb}} = \to^{*}_{R_{\text{emb}}}$
("homeomorphic embedding relation").

$\rhd_{\text{emb}}$ is a simplification ordering.

Simplification Orderings

Lemma 4.25:

If $\succ$ is a simplification ordering, then $s \rhd_{\text{emb}} t$ implies $s \succ t$ and $s \trianglerighteq_{\text{emb}} t$ implies $s \succeq t$.

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 Σ be a finite signature, let X be a finite set of variables.
Then for every infinite sequence $t_1, t_2, t_3, \ldots$ there are indices $j > i$ such that $t_j \trianglerighteq_{\text{emb}} t_i$.
($\trianglerighteq_{\text{emb}}$ is called a well-partial-ordering (wpo).)

Proof:
See Baader and Nipkow, page 113–115. $\square$

Simplification Orderings

Theorem 4.27 (Dershowitz):

If Σ is a finite signature, then every simplification ordering $\succ$ on $\mathrm{T}_\Sigma(X)$ 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 $R = \{f(f(x)) \to f(g(f(x)))\}$.

R terminates and $\to_R^+$ is therefore a reduction ordering.

Assume that $\to_R$ were contained in a simplification ordering $\succ$. Then $f(f(x)) \to_R f(g(f(x)))$ implies $f(f(x)) \succ f(g(f(x)))$, and $f(g(f(x))) \trianglerighteq_{\text{emb}} f(f(x))$ 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 Ω.

The lexicographic path ordering $\succ_{\mathsf{lpo}}$ on $\mathrm{T}_\Sigma(X)$ induced by $\succ$ is defined by: $s \succ_{\mathsf{lpo}} t$ iff

(1) $t \in \mathrm{var}(s)$ and $t \neq s$, or

(2) $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, and

- (a) $s_i \succeq_{\mathsf{lpo}} t$ for some i, or
- (b) $f \succ g$ and $s \succ_{\mathsf{lpo}} t_j$ for all j, or
- (c) $f = g$, $s \succ_{\mathsf{lpo}} t_j$ for all j, and $(s_1, \ldots, s_m)\ (\succ_{\mathsf{lpo}})_{\mathsf{lex}}\ (t_1, \ldots, t_n)$.

Path Orderings

Lemma 4.28:
$s \succ_{\mathsf{lpo}} t$ implies $\mathrm{var}(s) \supseteq \mathrm{var}(t)$.

Theorem 4.29:
$\succ_{\mathsf{lpo}}$ is a simplification ordering on $\mathrm{T}_\Sigma(X)$.

Theorem 4.30:
If the precedence $\succ$ is total, then the lexicographic path ordering $\succ_{\mathsf{lpo}}$ is total on ground terms, i. e., for all $s, t \in \mathrm{T}_\Sigma(\emptyset)$:
$s \succ_{\mathsf{lpo}} t \vee t \succ_{\mathsf{lpo}} s \vee s = t$.

Path Orderings

Recapitulation:

Let $\Sigma = (\Omega, \Pi)$ be a finite signature, let $\succ$ be a strict partial ordering ("precedence") on Ω. The lexicographic path ordering $\succ_{\mathrm{lpo}}$ on $\mathrm{T}_\Sigma(X)$ induced by $\succ$ is defined by: $s \succ_{\mathrm{lpo}} t$ iff

(1) $t \in \mathrm{var}(s)$ and $t \neq s$, or

(2) $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, and

(a) $s_i \succeq_{\mathrm{lpo}} t$ for some i, or

(b) $f \succ g$ and $s \succ_{\mathrm{lpo}} t_j$ for all j, or

(c) $f = g$, $s \succ_{\mathrm{lpo}} t_j$ for all j, and $(s_1, \ldots, s_m)\ (\succ_{\mathrm{lpo}})_{\mathrm{lex}}\ (t_1, \ldots, t_n)$.

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 π)
- 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 $\in \{\mathit{mul}\} \cup \{\mathit{lex}_\pi \mid \pi : \{1, \dots, n\} \to \{1, \dots, n\}\}$
  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 Ω,
let $w : \Omega \cup X \to \mathbb{R}_0^+$ 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:

$$w(f(t_1, \dots, t_n)) = w(f) + \sum_{1 \le i \le n} w(t_i)$$

or alternatively

$$w(t) = \sum_{x \in \mathrm{var}(t)} w(x) \cdot \#(x, t) + \sum_{f \in \Omega} w(f) \cdot \#(f, t).$$

where $\#(a, t)$ is the number of occurrences of a in t.

The Knuth-Bendix Ordering

The Knuth-Bendix ordering $\succ_{\mathrm{kbo}}$ on $\mathrm{T}_\Sigma(X)$ induced by $\succ$ and w is defined by: $s \succ_{\mathrm{kbo}} t$ iff

(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) $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, and $f \succ g$, or

  (c) $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, and $(s_1, \ldots, s_m)\ (\succ_{\mathrm{kbo}})_{\mathrm{lex}}\ (t_1, \ldots, t_m)$.

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. $\square$

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:
$E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \ldots$

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 $s \mathrel{\dot{\approx}} t$ denotes either $s \approx t$ or $t \approx s$.

$\mathrm{CP}(R)$ denotes the set of all critical pairs between rules in R.

Knuth-Bendix Completion: Inference Rules

Orient:

$$\frac{E \cup \{s \mathrel{\dot{\approx}} t\}, \quad R}{E, \quad R \cup \{s \to t\}} \qquad \text{if } s \succ t$$

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:

$$\frac{E \cup \{s \approx s\}, \quad R}{E, \quad R}$$

Knuth-Bendix Completion: Inference Rules

Critical pairs between rules in R are turned into additional equations:

Deduce:

$$\frac{E, \quad R}{E \cup \{s \approx t\}, \quad R} \qquad \text{if } \langle s, t\rangle \in \mathrm{CP}(R).$$

Note: If $\langle s, t\rangle \in \mathrm{CP}(R)$ 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:

$$\frac{E \cup \{s \mathrel{\dot{\approx}} t\}, \quad R}{E \cup \{u \approx t\}, \quad R} \qquad \text{if } s \to_R u.$$

Knuth-Bendix Completion: Inference Rules

Simplification of the right-hand side of a rule is unproblematic:

R-Simplify-Rule:

$$\frac{E, \quad R \cup \{s \to t\}}{E, \quad R \cup \{s \to u\}} \qquad \text{if } t \to_R u.$$

Simplification of the left-hand side may influence orientability and orientation. Therefore, it yields an *equation*:

L-Simplify-Rule:

$$\frac{E, \quad R \cup \{s \to t\}}{E \cup \{u \approx t\}, \quad R} \qquad \begin{array}{l}\text{if } s \to_R u \text{ using a rule } l \to r \in R \\ \text{such that } s \sqsupset l \text{ (see next slide).}\end{array}$$

Knuth-Bendix Completion: Inference Rules

For technical reasons, the lhs of $s \to t$ may only be simplified using a rule $l \to r$, if $l \to r$ *cannot* be simplified using $s \to t$, that is, if $s \sqsupset l$, where the encompassment quasi-ordering $\sqsupset\!\!\!\sim$ is defined by

$$s \;\sqsupset\!\!\!\sim\; l \quad \text{if} \quad s|_p = l\sigma \text{ for some } p \text{ and } \sigma$$

and $\sqsupset \;=\; \sqsupset\!\!\!\sim \setminus \sqsubset\!\!\!\sim$ is the strict part of $\sqsupset\!\!\!\sim$.

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

If we run the completion procedure on a set E of equations, different things can happen:

(1) We reach a state where no more inference rules are applicable and E is not empty.
$\Rightarrow$ 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) $E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \ldots$ with $R_0 = \emptyset$ is called a run of the completion procedure with input E_0 and $\succ$.

For a run, $E_\infty = \bigcup_{i \geq 0} E_i$ and $R_\infty = \bigcup_{i \geq 0} R_i$.

The sets of persistent equations or rules of the run are $E_* = \bigcup_{i \geq 0} \bigcap_{j \geq i} E_j$ and $R_* = \bigcup_{i \geq 0} \bigcap_{j \geq i} R_j$.

Note: If the run is finite and ends with E_n, R_n, then $E_* = E_n$ and $R_* = R_n$.

Knuth-Bendix Completion: Correctness Proof

A run is called fair, if $CP(R_*) \subseteq E_\infty$
(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_* is empty, then R_* is convergent and equivalent to E_0.

In particular: If a run is fair and E_* is empty, then $\approx_{E_0} = \approx_{E_\infty \cup R_\infty} = \leftrightarrow^*_{E_\infty \cup R_\infty} = \downarrow_{R_*}$.

Knuth-Bendix Completion: Correctness Proof

General assumptions from now on:

$E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \ldots$ is a fair run.

R_0 and E_* are empty.

Knuth-Bendix Completion: Correctness Proof

A proof of $s \approx t$ in $E_\infty \cup R_\infty$ 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) $s_{i-1} \leftrightarrow_{E_\infty} s_i$, or

(2) $s_{i-1} \rightarrow_{R_\infty} s_i$, or

(3) $s_{i-1} \leftarrow_{R_\infty} s_i$.

The pairs (s_{i-1}, s_i) are called proof steps.

A proof is called a rewrite proof in R_*,
if there is a $k \in \{0, \dots, n\}$ such that $s_{i-1} \rightarrow_{R_*} s_i$ for $1 \leq i \leq k$
and $s_{i-1} \leftarrow_{R_*} s_i$ 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_* there is an equivalent smaller proof.

Consequence: For every proof there is an equivalent rewrite proof in R_*.

Knuth-Bendix Completion: Correctness Proof

We associate a cost $c(s_{i-1}, s_i)$ with every proof step as follows:

(1) If $s_{i-1} \leftrightarrow_{E_\infty} s_i$, 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 $s_{i-1} \rightarrow_{R_\infty} s_i$ using $l \rightarrow r$, then $c(s_{i-1}, s_i) = (\{s_{i-1}\}, l, s_i)$.

(3) If $s_{i-1} \leftarrow_{R_\infty} s_i$ 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 $\sqsupset$, 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 $E_\infty \cup R_\infty$. If P is not a rewrite proof in R_*, then there exists an equivalent proof P' in $E_\infty \cup R_\infty$ such that $P \succ_C P'$.

Proof:

If P is not a rewrite proof in R_*, then it contains

(a) a proof step that is in E_∞, or

(b) a proof step that is in $R_\infty \setminus R_*$, or

(c) a subproof $s_{i-1} \leftarrow_{R_*} s_i \rightarrow_{R_*} s_{i+1}$ (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 $s \doteq t$ is in E_∞.
This equation must be deleted during the run.

If $s \doteq t$ is deleted using *Orient*:

$$\dots s_{i-1} \leftrightarrow_{E_\infty} s_i \dots \qquad \Longrightarrow \qquad \dots s_{i-1} \rightarrow_{R_\infty} s_i \dots$$

If $s \doteq t$ is deleted using *Delete*:

$$\dots s_{i-1} \leftrightarrow_{E_\infty} s_{i-1} \dots \qquad \Longrightarrow \qquad \dots s_{i-1} \dots$$

If $s \doteq t$ is deleted using *Simplify-Eq*:

$$\dots s_{i-1} \leftrightarrow_{E_\infty} s_i \dots \qquad \Longrightarrow \qquad \dots s_{i-1} \rightarrow_{R_\infty} s' \leftrightarrow_{E_\infty} s_i \dots$$

Knuth-Bendix Completion: Correctness Proof

Case (b): A proof step using a rule $s \to t$ is in $R_\infty \setminus R_*$.
This rule must be deleted during the run.

If $s \to t$ is deleted using *R-Simplify-Rule*:

$$\ldots s_{i-1} \to_{R_\infty} s_i \ldots \quad \Longrightarrow \quad \ldots s_{i-1} \to_{R_\infty} s' \leftarrow_{R_\infty} s_i \ldots$$

If $s \to t$ is deleted using *L-Simplify-Rule*:

$$\ldots s_{i-1} \to_{R_\infty} s_i \ldots \quad \Longrightarrow \quad \ldots s_{i-1} \to_{R_\infty} s' \leftrightarrow_{E_\infty} s_i \ldots$$

Knuth-Bendix Completion: Correctness Proof

Case (c): A subproof has the form $s_{i-1} \leftarrow_{R_*} s_i \rightarrow_{R_*} s_{i+1}$.

If there is no overlap or a non-critical overlap:

$$\dots s_{i-1} \leftarrow_{R_*} s_i \rightarrow_{R_*} s_{i+1} \dots \implies \dots s_{i-1} \rightarrow^*_{R_*} s' \leftarrow^*_{R_*} s_{i+1} \dots$$

If there is a critical pair that has been added using *Deduce*:

$$\dots s_{i-1} \leftarrow_{R_*} s_i \rightarrow_{R_*} s_{i+1} \dots \implies \qquad \dots s_{i-1} \leftrightarrow_{E_\infty} s_{i+1} \dots$$

In all cases, checking that the replacement subproof is smaller than the replaced subproof is routine. $\square$

Knuth-Bendix Completion: Correctness Proof

Theorem 4.37:

Let $E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \ldots$ be a fair run and let R_0 and E_* be empty. Then

(1) every proof in $E_\infty \cup R_\infty$ is equivalent to a rewrite proof in R_*,

(2) R_* is equivalent to E_0, and

(3) R_* is convergent.

Knuth-Bendix Completion: Correctness Proof

Proof:

(1) By well-founded induction on $\succ_C$ using the previous lemma.

(2) Clearly $\approx_{E_\infty \cup R_\infty} = \approx_{E_0}$.
Since $R_* \subseteq R_\infty$, we get $\approx_{R_*} \subseteq \approx_{E_\infty \cup R_\infty}$.
On the other hand, by (1), $\approx_{E_\infty \cup R_\infty} \subseteq \approx_{R_*}$.

(3) Since $\rightarrow_{R_*} \subseteq \succ$, R_* is terminating.
By (1), R_* is confluent. $\square$

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

$$s \rightarrow_{E^\succ} t \quad \text{iff} \quad \begin{array}{l} \text{there exist } (u \approx v) \in E \text{ or } (v \approx u) \in E, \\ p \in \text{pos}(s), \text{ and } \sigma : X \to \mathrm{T}_\Sigma(X), \\ \text{such that } s|_p = u\sigma \text{ and } t = s[v\sigma]_p \\ \text{and } u\sigma \succ v\sigma. \end{array}$$

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 $s \not\preceq t$, we approximate $\succ$ on ground terms by $\not\preceq$ on arbitrary terms.

Unfailing Completion

Let $u_i \mathrel{\dot{\approx}} v_i$ $(i = 1, 2)$ be equations in E whose variables have been renamed such that $\mathrm{var}(u_1 \mathrel{\dot{\approx}} v_1) \cap \mathrm{var}(u_2 \mathrel{\dot{\approx}} v_2) = \emptyset$.
Let $p \in \mathrm{pos}(u_1)$ be a position such that $u_1|_p$ is not a variable, σ is an mgu of $u_1|_p$ and u_2, and $u_i\sigma \not\preceq v_i\sigma$ $(i = 1, 2)$.
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 $\mathrm{SP}_\succ(E)$.

Semi-critical pairs of $E \cup R$ are defined analogously.
If $\rightarrow_R \subseteq \succ$, then $\mathrm{CP}(R)$ and $\mathrm{SP}_\succ(R)$ agree.

Unfailing Completion

Note: In contrast to critical pairs, it may be necessary to consider overlaps of a rule 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:

$$\frac{E, \quad R}{E \cup \{s \approx t\}, \quad R} \qquad \text{if } \langle s, t\rangle \in \mathrm{SP}_{\succ}(E \cup R).$$

Moreover, the fairness criterion for runs is replaced by

$$\mathrm{SP}_{\succ}(E_* \cup R_*) \subseteq E_\infty$$

(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 $E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \ldots$ be a fair run; let $R_0 = \emptyset$.
Then

(1) $E_* \cup R_*$ is equivalent to E_0, and

(2) $E_* \cup R_*$ 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 $E_* = \emptyset$ and $R_* = R$ 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_* is reduced, and for all equations $u \approx v \in E_*$, u and v are incomparable w. r. t. $\succ$ and irreducible w. r. t. R_*.

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 $E_0, \emptyset \vdash E_1, R_1 \vdash E_2, R_2 \vdash \ldots$ be a fair run of unfailing completion, then $s \approx_E t$ iff 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

$\Rightarrow$ 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 $\Sigma = (\Omega, \emptyset)$.

$T_0 := \{\, t \in \mathrm{T}_\Sigma(X) \mid \exists \text{ infinite deriv. } t \to_R t_1 \to_R t_2 \to_R \ldots \}$.

$$T_\infty := \{\, t \in T_0 \mid \forall p > \varepsilon : t|_p \notin T_0 \,\}$$
$$= \text{minimal elements of } T_0 \text{ w.r.t. } \rhd.$$

$t \in T_0 \Rightarrow$ there exists a $t' \in T_\infty$ such that $t \trianglerighteq t'$.

R is non-terminating iff $T_0 \neq \emptyset$ iff $T_\infty \neq \emptyset$.

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 $l \to r \in R$ 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, $\mathrm{root}(t) = \mathrm{root}(l)$, so we see that the root symbol of any term in T_∞ must be contained in $D := \{\, \mathrm{root}(l) \mid l \to r \in R \,\}$.
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 $r\sigma \trianglerighteq v$.

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 $s_i \trianglerighteq x\sigma$ 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

$$t = f(t_1, \ldots, t_n) \xrightarrow{>\varepsilon}{}^{*}_{R} f(s_1, \ldots, s_n) = l\sigma \xrightarrow{\varepsilon}_{R} r\sigma \trianglerighteq u\sigma$$

where $r \trianglerighteq u$, u is not a variable, $\mathrm{root}(u) \in D$, $l \not\triangleright u$.

Since $u\sigma \in T_\infty$, we can continue this process and obtain an infinite sequence.

Dependency Pairs

If we define

$$S := \{\, l \to u \mid l \to r \in R,\ r \trianglerighteq u,\ u \notin X,\ \mathrm{root}(u) \in D,\ l \not\triangleright u \,\},$$

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 ε and $>\varepsilon$, it turns out to be useful to introduce a new set of function symbols $f^\sharp$ that are only used for the root symbols of this derivation:

$$\Omega^\sharp := \{\, f^\sharp/n \mid f/n \in \Omega \,\}.$$

For a term $t = f(t_1, \dots, t_n)$ we define $t^\sharp := f^\sharp(t_1, \dots, t_n)$;
for a set of terms T we define $T^\sharp := \{\, t^\sharp \mid t \in T \,\}$.

The set of dependency pairs of a TRS R is then defined by

$$\mathrm{DP}(R) := \{\, l^\sharp \to u^\sharp \mid l \to r \in R,\ r \trianglerighteq u,\ u \notin X,\ \mathrm{root}(u) \in D,\ l \not\triangleright u \,\}.$$

Dependency Pairs

For $t \in T_\infty$, the sequence using the S-rule corresponds now to

$$t^\sharp \rightarrow_R^* l^\sharp\sigma \rightarrow_{\mathrm{DP}(R)} u^\sharp\sigma$$

where $t^\sharp \in T_\infty^\sharp$ and $u^\sharp\sigma \in T_\infty^\sharp$.

(Note that rules in R do not contain symbols from $\Omega^\sharp$, whereas all roots of terms in $\mathrm{DP}(R)$ come from $\Omega^\sharp$, so rules from R can only be applied below the root and rules from $\mathrm{DP}(R)$ can only be applied at the root.)

Dependency Pairs

Since $u^\sharp\sigma$ is again in $T^\sharp_\infty$, we can continue the process in the same way. We obtain: R is non-terminating iff there is an infinite sequence

$$t_1 \to_R^* t_2 \to_{\mathrm{DP}(R)} t_3 \to_R^* t_4 \to_{\mathrm{DP}(R)} \cdots$$

with $t_i \in T^\sharp_\infty$ 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 $s \to t \in \text{DP}(R)$

edges: from $s \to t$ to $u \to v$ if there are σ, τ such that $t\sigma \to_R^* u\tau$.

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 cap and ren are defined by:

$$\mathrm{cap}(x) = x$$

$$\mathrm{cap}(f(t_1, \ldots, t_n)) = \begin{cases} y & \text{if } f \in D \\ f(\mathrm{cap}(t_1), \ldots, \mathrm{cap}(t_n)) & \text{if } f \in C \cup D^\sharp \end{cases}$$

$$\mathrm{ren}(x) = y, \quad y \text{ fresh}$$

$$\mathrm{ren}(f(t_1, \ldots, t_n)) = f(\mathrm{ren}(t_1), \ldots, \mathrm{ren}(t_n))$$

The overapproximated dependency graph contains an edge from $s \to t$ to $u \to v$ if $\mathrm{ren}(\mathrm{cap}(t))$ and u are unifiable.

Dependency Graphs

A cycle in the dependency graph is a non-empty subset $K \subseteq \mathrm{DP}(R)$ 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 $K \subseteq \mathrm{DP}(R)$. 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 $t_i \in T_\infty^\sharp$ is called K-minimal, if all rules in K are used infinitely often.

R is non-terminating iff there is a cycle $K \subseteq \mathrm{DP}(R)$ 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 π is a mapping $\pi : \Omega^\sharp \to \mathbb{N}$ 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 $\mathsf{DG}(R)$. If there is a simple projection π for K such that $\pi(l) \trianglerighteq \pi(r)$ for every $l \to r \in K$ and $\pi(l) \triangleright \pi(r)$ for some $l \to r \in K$, 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.)[a]

Important property: Every cycle is contained in some SCC.

[a] 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 π such that $\pi(l) \trianglerighteq \pi(r)$ for all DPs $l \to r$ in the SCC. Delete all DPs in the SCC for which $\pi(l) \vartriangleright \pi(r)$ (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

$$t_1 \rightarrow_R^* u_1 \rightarrow_K t_2 \rightarrow_R^* u_2 \rightarrow_K \ldots$$

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 π:

$$\pi : \Omega \cup \Omega^\sharp \to \mathbb{N} \cup \text{list of } \mathbb{N}$$

$$\pi(f) = \begin{cases} i \in \{1, \ldots, \text{arity}(f)\}, \text{ or} \\ [i_1, \ldots, i_k], \text{ where } 1 \leq i_1 < \cdots < i_k \leq \text{arity}(f), \\ \qquad\qquad 0 \leq k \leq \text{arity}(f) \end{cases}$$

Reduction Pairs and Argument Filterings

Extension to terms:

$\pi(x) = x$

$\pi(f(t_1, \ldots, t_n)) = \pi(t_i)$, if $\pi(f) = i$

$\pi(f(t_1, \ldots, t_n)) = f'(\pi(t_{i_1}), \ldots, \pi(t_{i_k}))$, if $\pi(f) = [i_1, \ldots, i_k]$,
where f'/k is a new function symbol.

Reduction Pairs and Argument Filterings

Let $\succ$ be a reduction ordering, let π be an argument filtering.
Define $s \succ_\pi t$ iff $\pi(s) \succ \pi(t)$ and $s \succsim_\pi t$ iff $\pi(s) \succeq \pi(t)$.

Lemma 5.2:
$(\succsim_\pi, \succ_\pi)$ 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 Σ-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 $f(a_1, \ldots, , a_n) \succeq f(b_1, \ldots, b_n)$ for all $a_i, b_i \in U_{\mathcal{A}}$.

Define $s \succsim_{\mathcal{A}} t$ iff $\mathcal{A}(\beta)(s) \succeq \mathcal{A}(\beta)(t)$ for all assignments $\beta : X \to U_{\mathcal{A}}$; define $s \succ_{\mathcal{A}} t$ iff $\mathcal{A}(\beta)(s) \succ \mathcal{A}(\beta)(t)$ for all assignments $\beta : X \to U_{\mathcal{A}}$.

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 $l \to r \in R$,
- $l \succsim r$ or $l \succ r$ for all $l \to r \in K$,
- $l \succ r$ for at least one $l \to r \in K$,

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 $l \to r \in R$ and $l \succsim r$ or $l \succ r$ for all DPs $l \to r$ 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]:

$$minus(x, 0) \to x \tag{1}$$

$$minus(s(x), s(y)) \to minus(x, y) \tag{2}$$

$$quot(0, s(y)) \to 0 \tag{3}$$

$$quot(s(x), s(y)) \to s(quot(minus(x, y), s(y))) \tag{4}$$

(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:

$$minus^\sharp(s(x), s(y)) \to minus^\sharp(x, y) \tag{5}$$

$$quot^\sharp(s(x), s(y)) \to quot^\sharp(minus(x, y), s(y)) \tag{6}$$

$$quot^\sharp(s(x), s(y)) \to minus^\sharp(x, y) \tag{7}$$

The dependency graph of R is

$$\begin{tikzcd}
(5) \arrow[loop below] & (7) \arrow[l] & (6) \arrow[l] \arrow[loop below]
\end{tikzcd}$$

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 π that maps *minus* to 1 and leaves all other function symbols unchanged (that is, $\pi(g) = [1, \ldots, \text{arity}(g)]$ for every g different from *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 $l \succsim r$ for (1), (2), (3), (4), so the previous theorem can be applied.

DP Processors

The methods described so far are particular cases of DP processors:

A DP processor

$$\frac{(G, R)}{(G_1, R_1), \quad \ldots, \quad (G_n, R_n)}$$

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:

$$\frac{(G, R)}{(SCC_1, R), \quad \ldots, \quad (SCC_n, R)}$$

where $SCC_1, \ldots, SCC_n$ 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 π such that $\pi(l) \trianglerighteq \pi(r)$ for all DPs $l \to r$ in the SCC, and N is the set of DPs of the SCC for which $\pi(l) \vartriangleright \pi(r)$.

(and analogously for reduction pairs)

Innermost Termination

The dependency method can also be used for proving termination of innermost rewriting: $s \xrightarrow{\mathrm{i}}_R t$ if $s \to_R t$ 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.

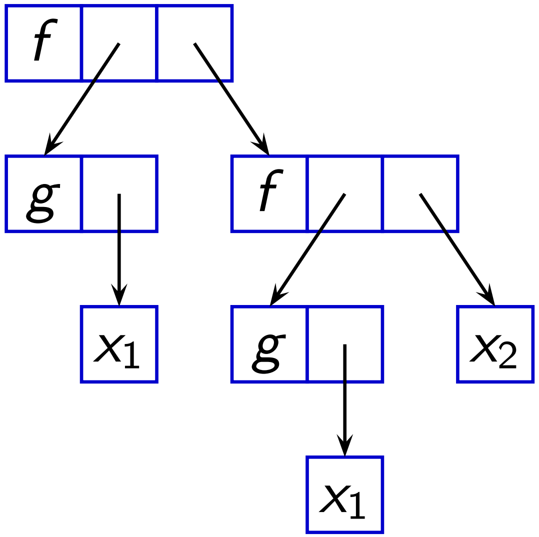
Design decisions are not independent of each other.

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

$$f(g(x_1), f(g(x_1), x_2))$$

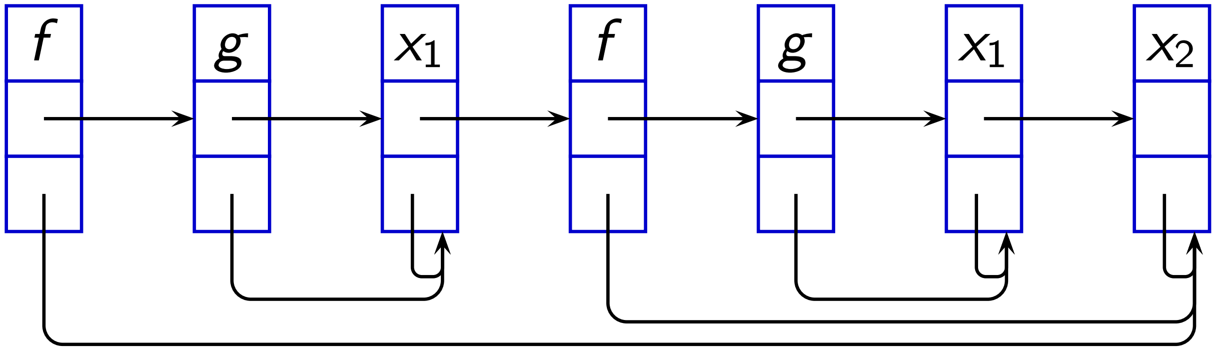


optionally: (full) sharing

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 $f(g(*, b), *)$: $f.1.g.1.*$

$f.1.g.2.b$

$f.2.*$

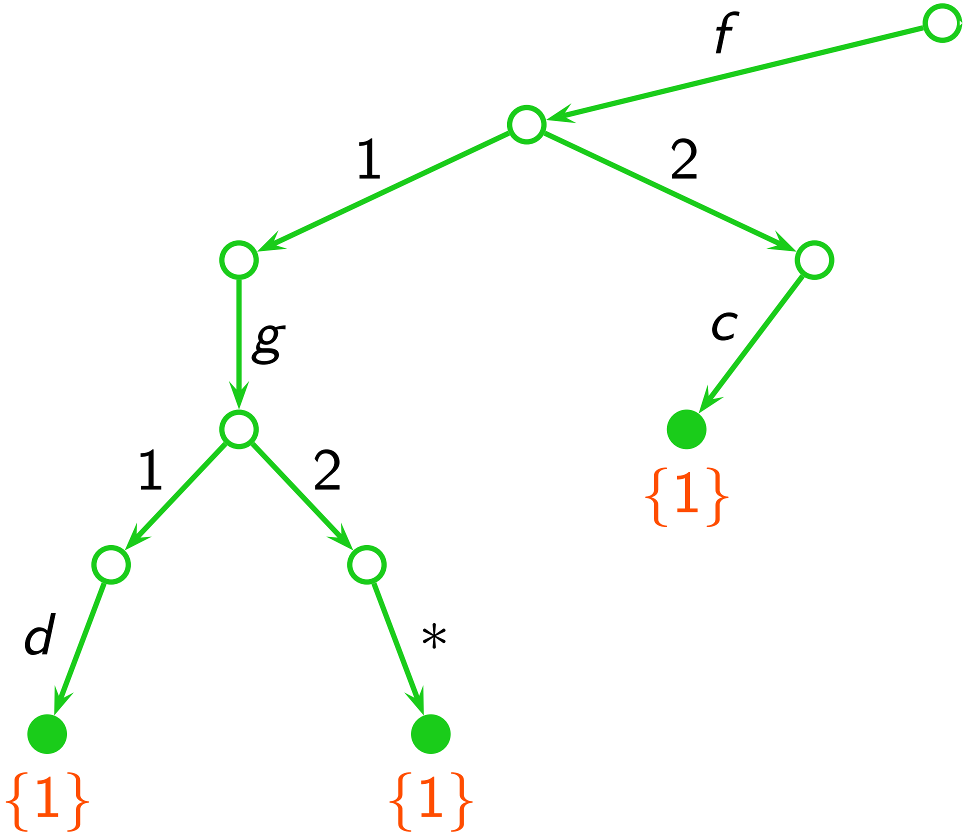
Each leaf of the trie contains the set of (pointers to) all terms that contain the respective path.

Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$
$f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$

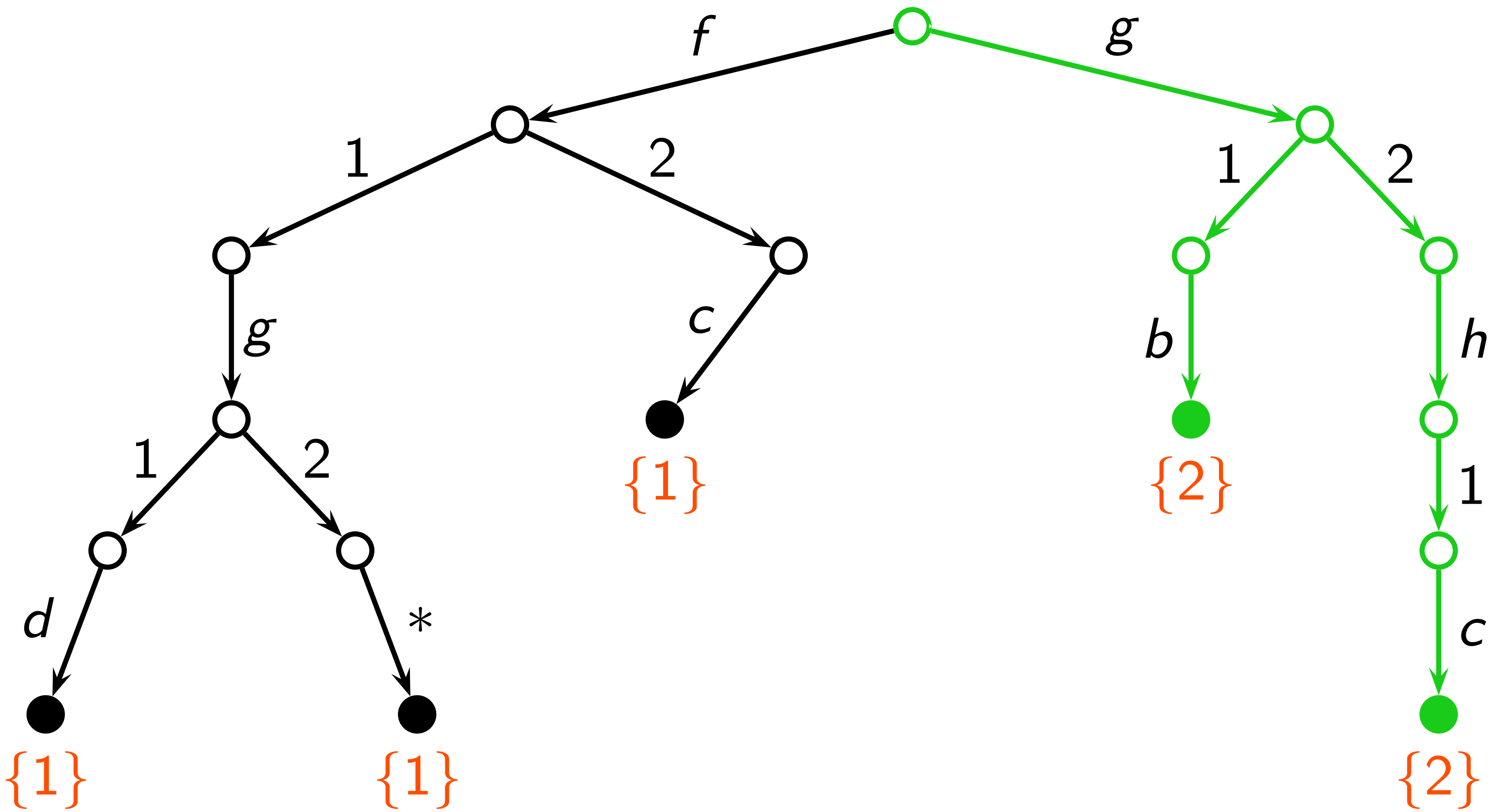
Path Indexing

Example: Path index for $\{f(g(d,*),c), g(b,h(c)), f(g(*,c),c), f(b,g(c,b)), f(b,g(*,b)), f(*,c), f(*,g(c,b))\}$



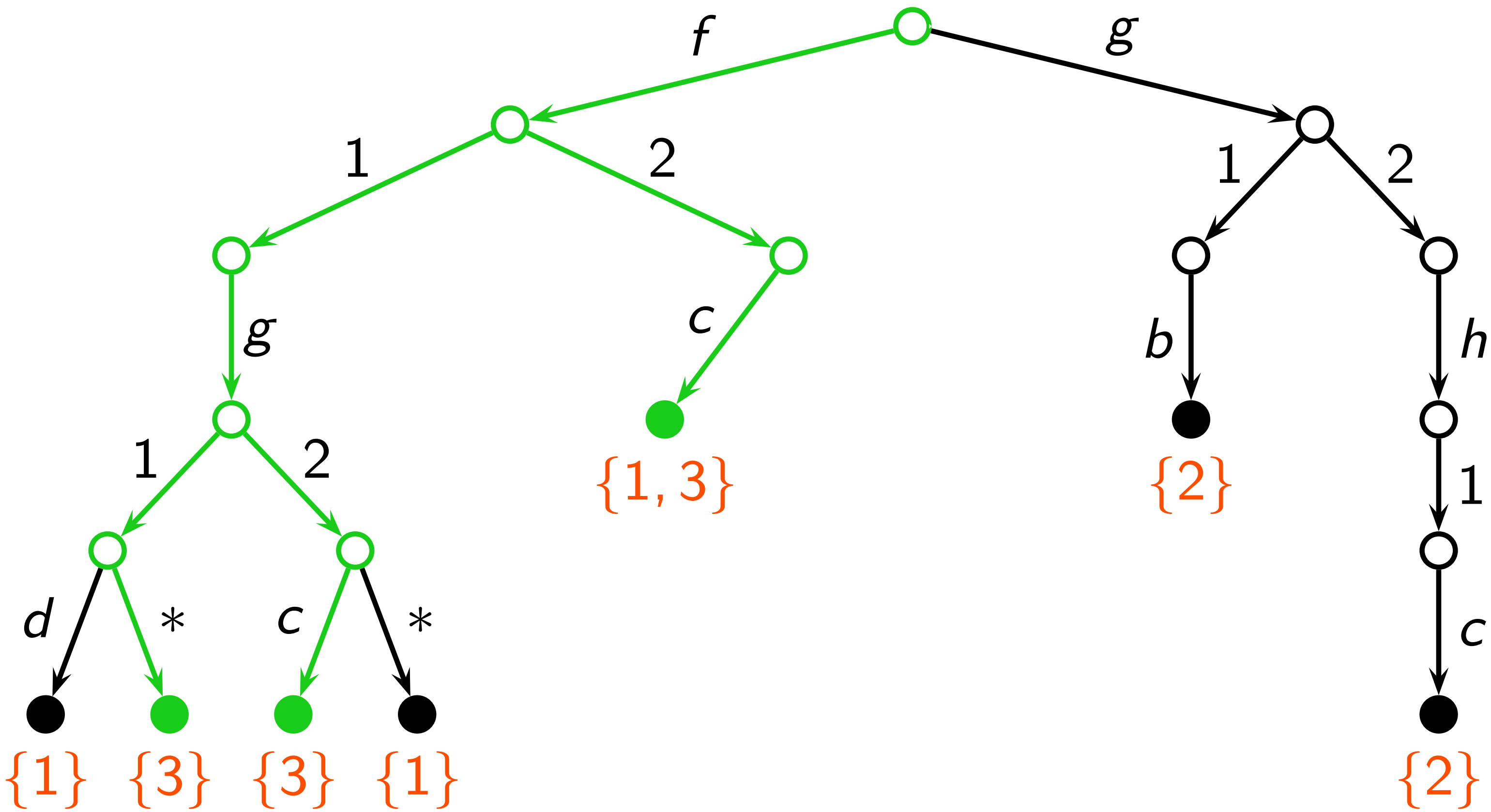
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



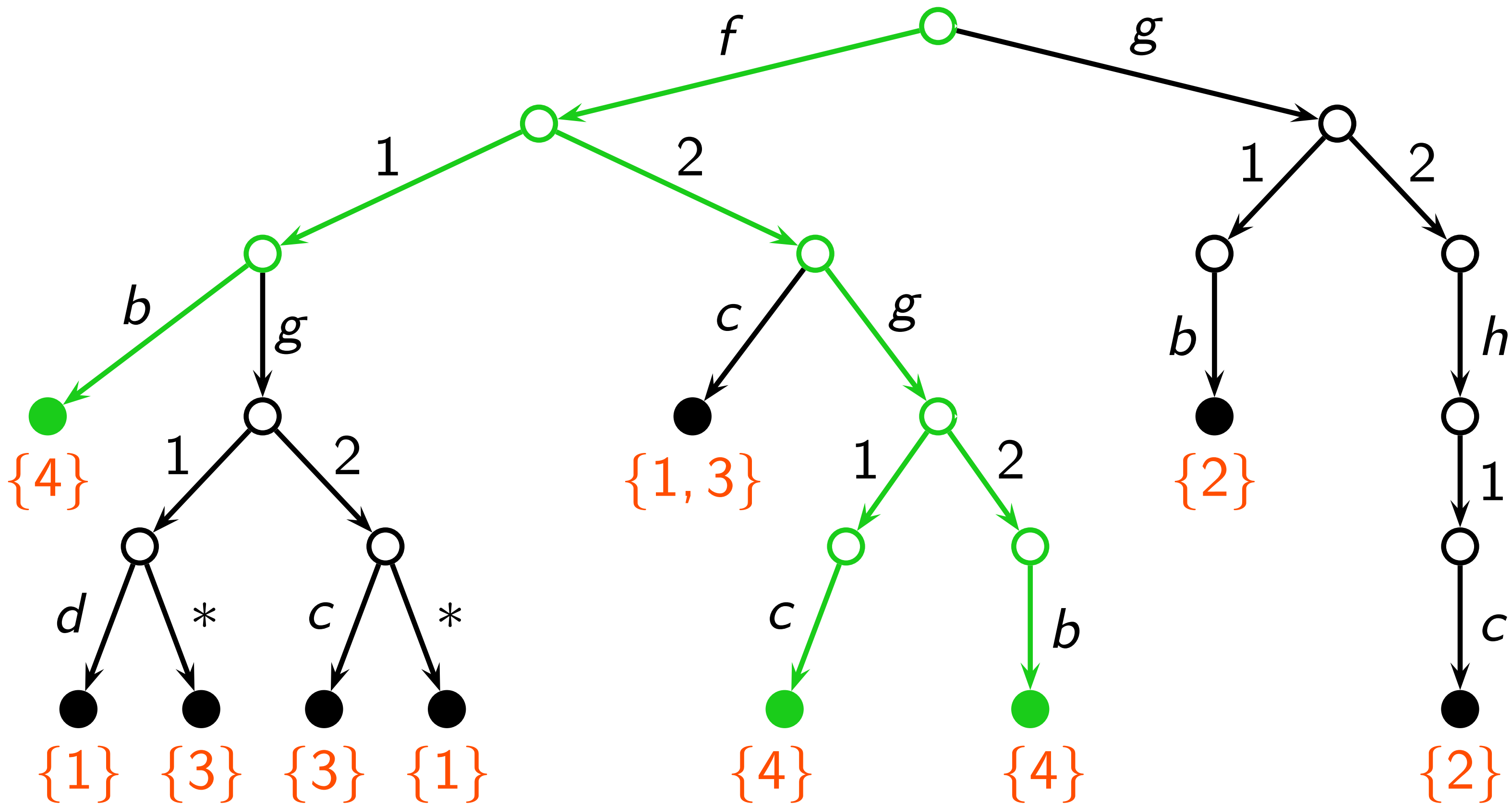
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$
$f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



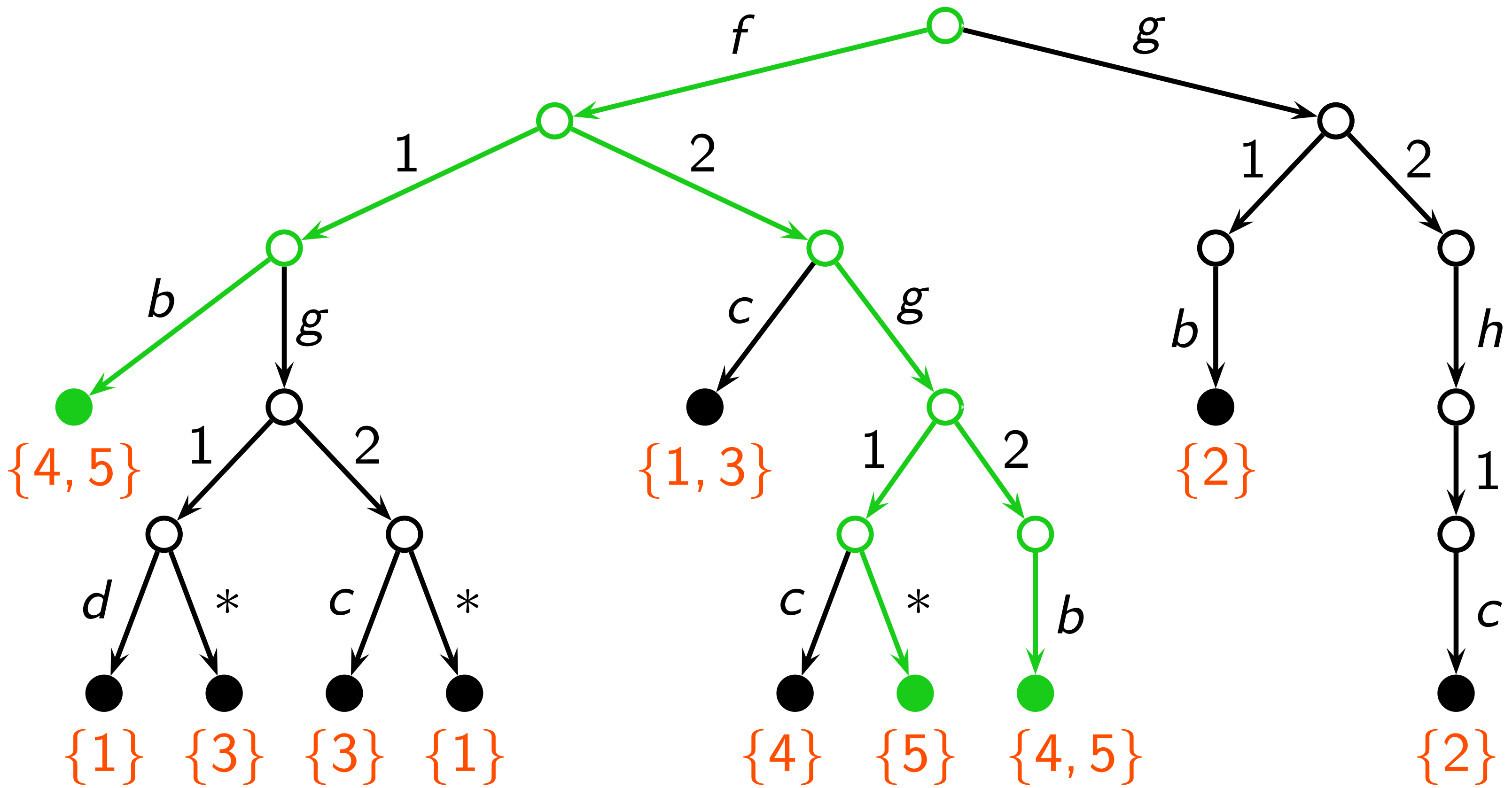
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



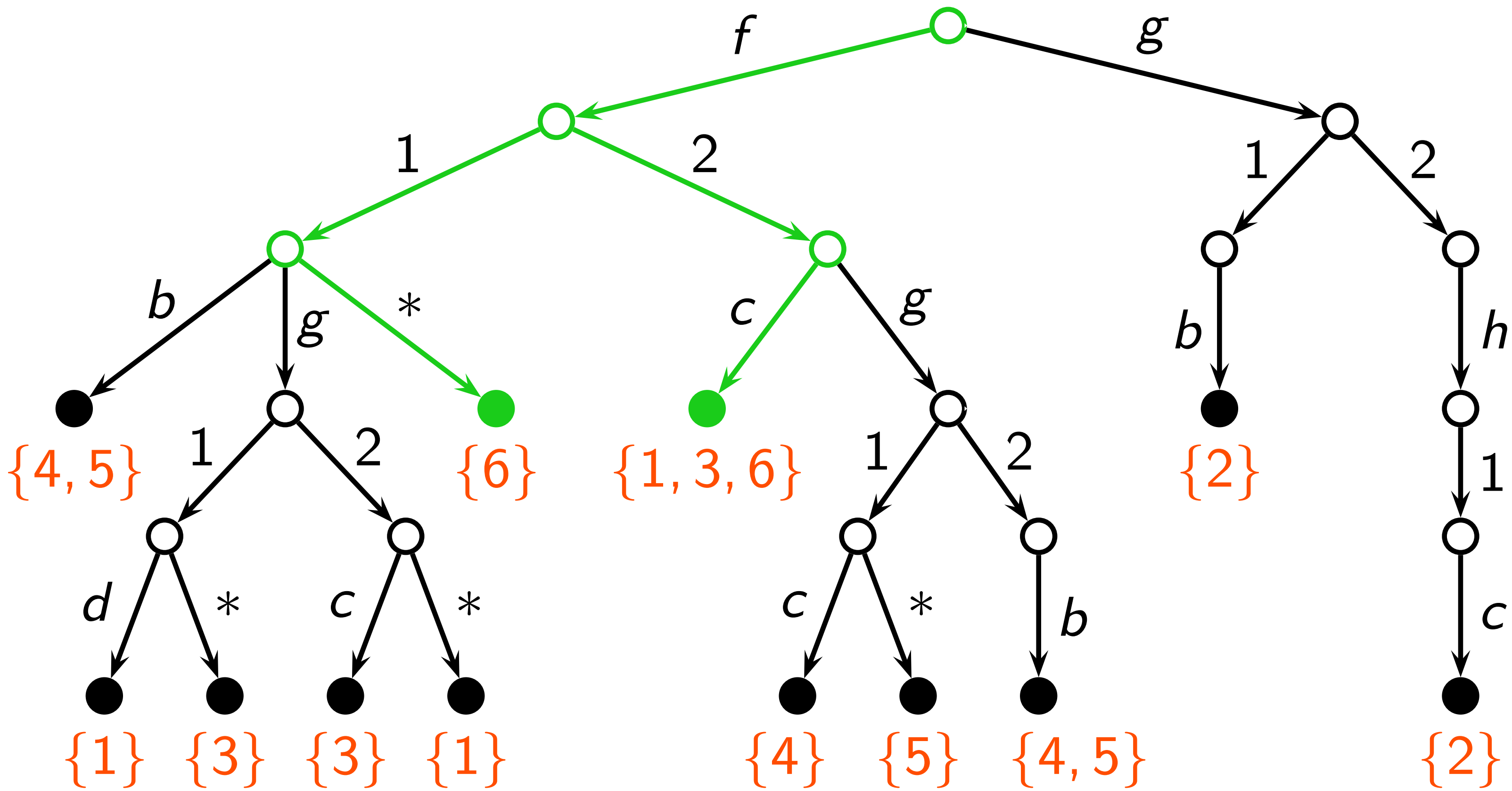
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



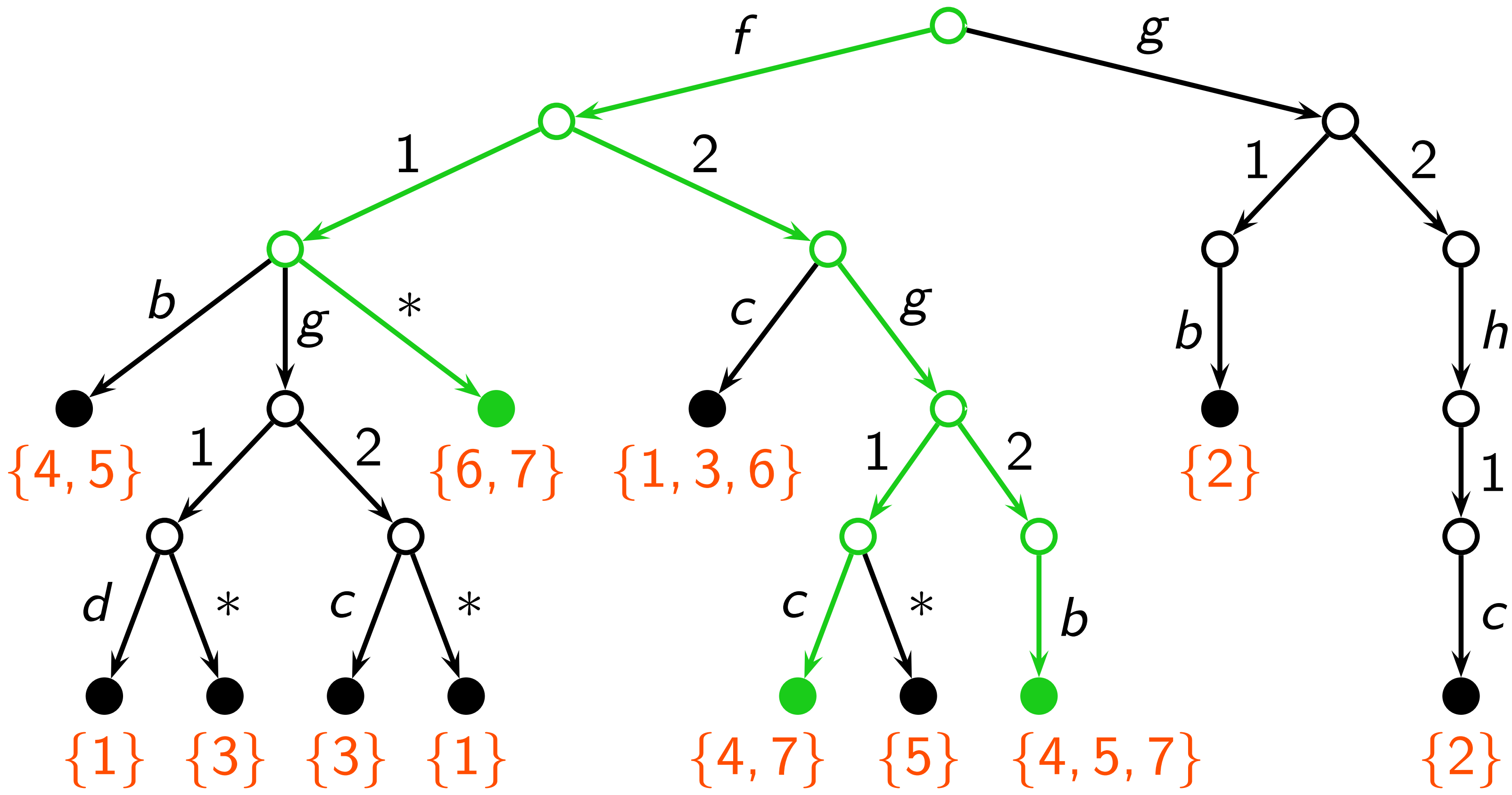
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



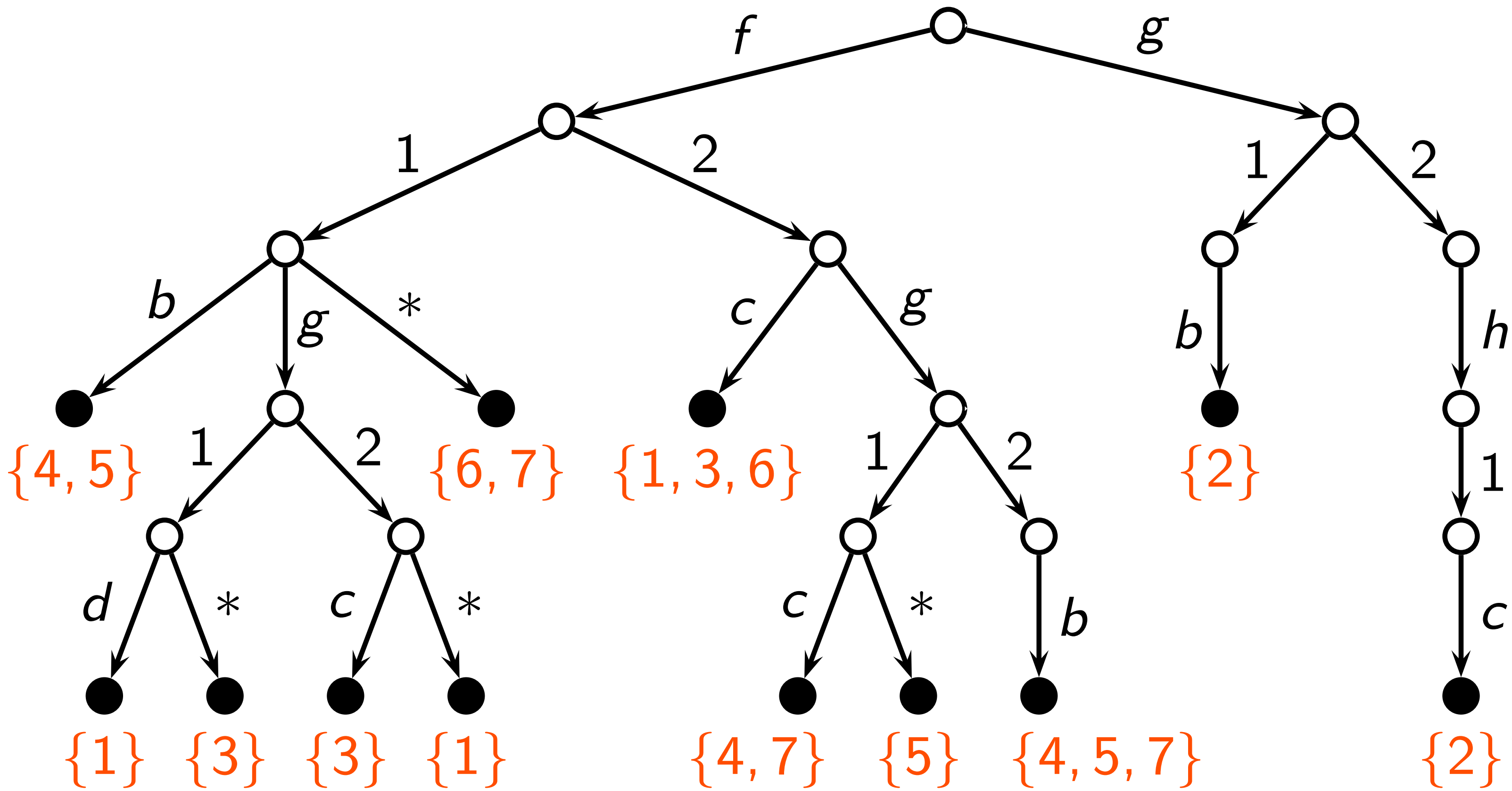
Path Indexing

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.

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 $f(g(*, b), *)$: $f.g.*.b.*$

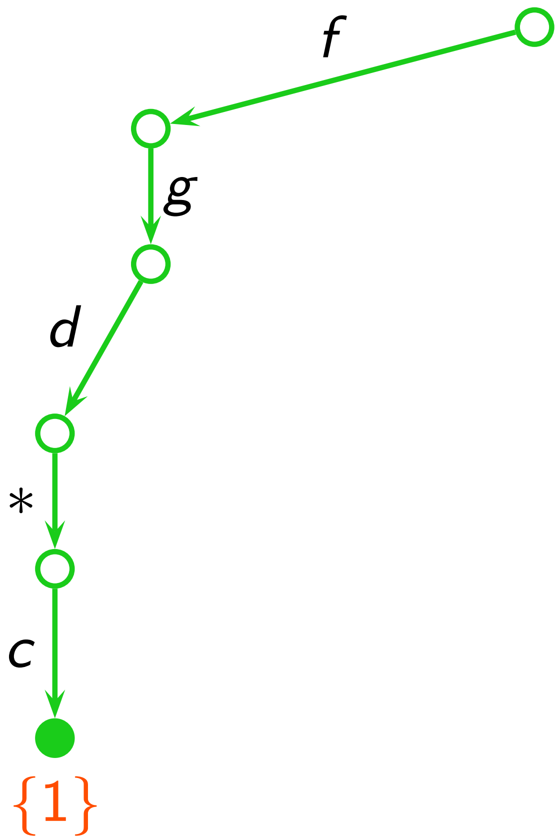
Each leaf of the trie contains (a pointer to) the term that is represented by the path.

Discrimination Trees

Example: Discrimination tree for $\{f(g(d, *), c),\ g(b, h(c)),$
$f(g(*, c), c),\ f(b, g(c, b)),\ f(b, g(*, b)),\ f(*, c),\ f(*, g(c, b))\}$

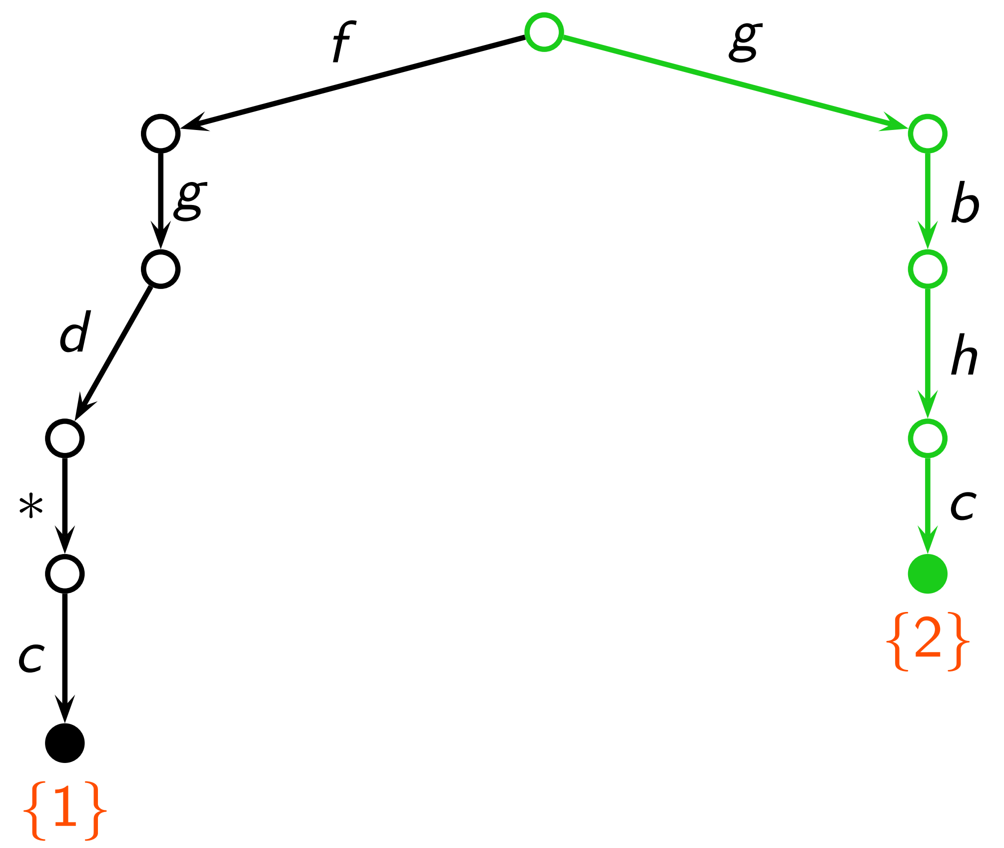
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),$
$f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



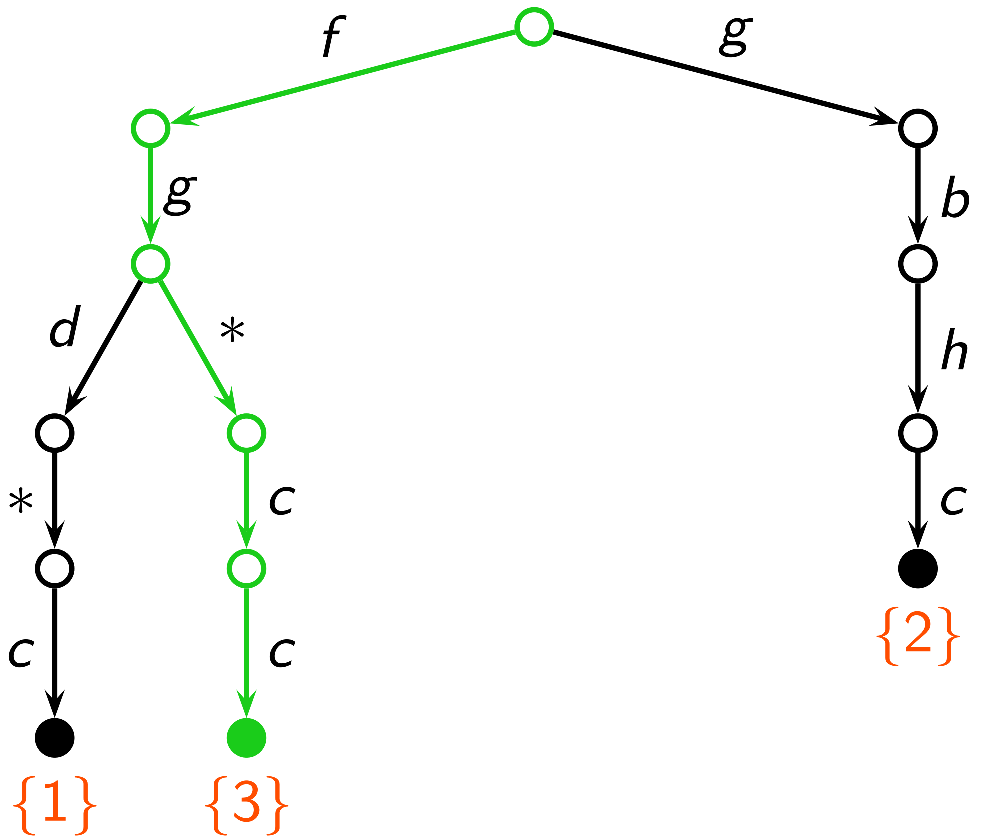
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),$
$f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



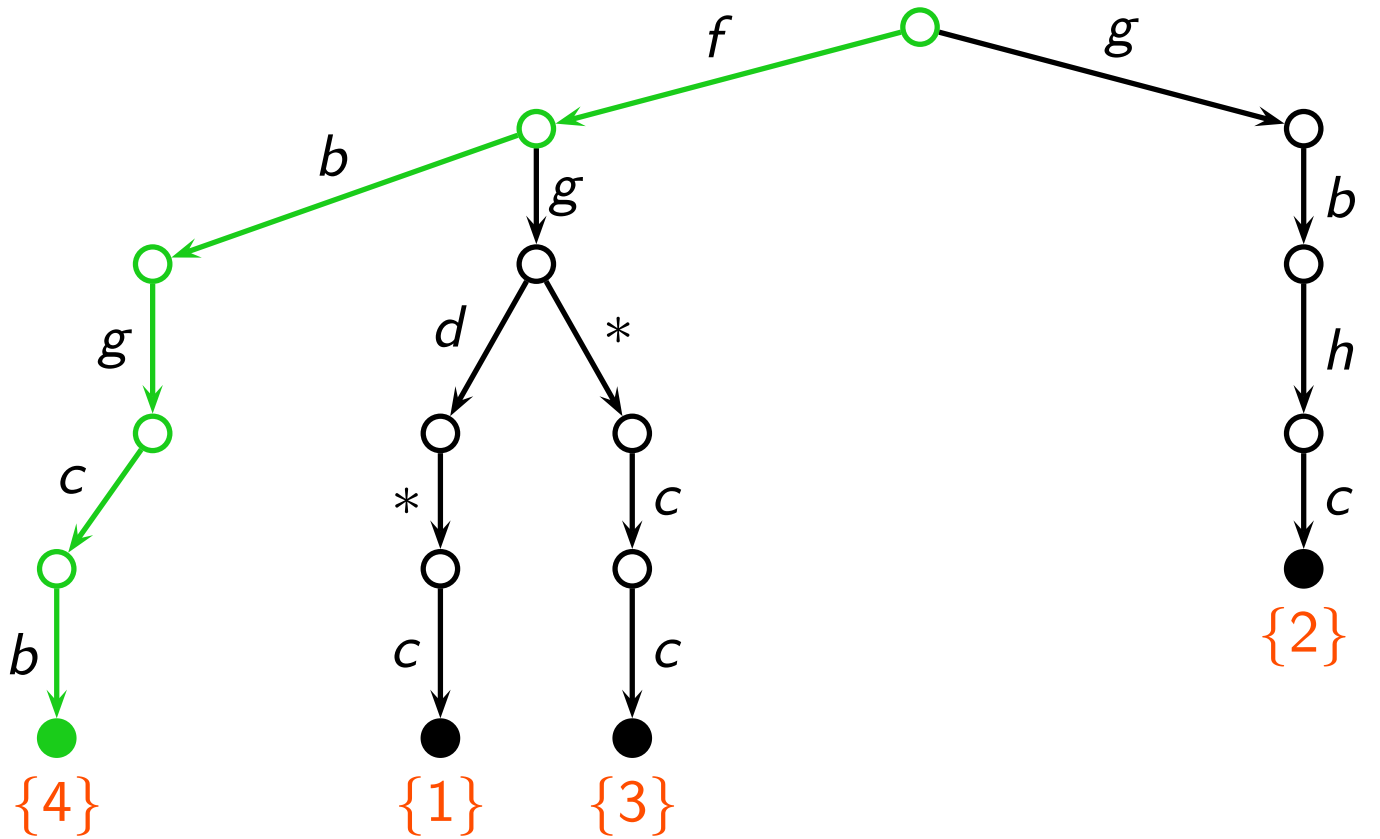
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),$
$f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



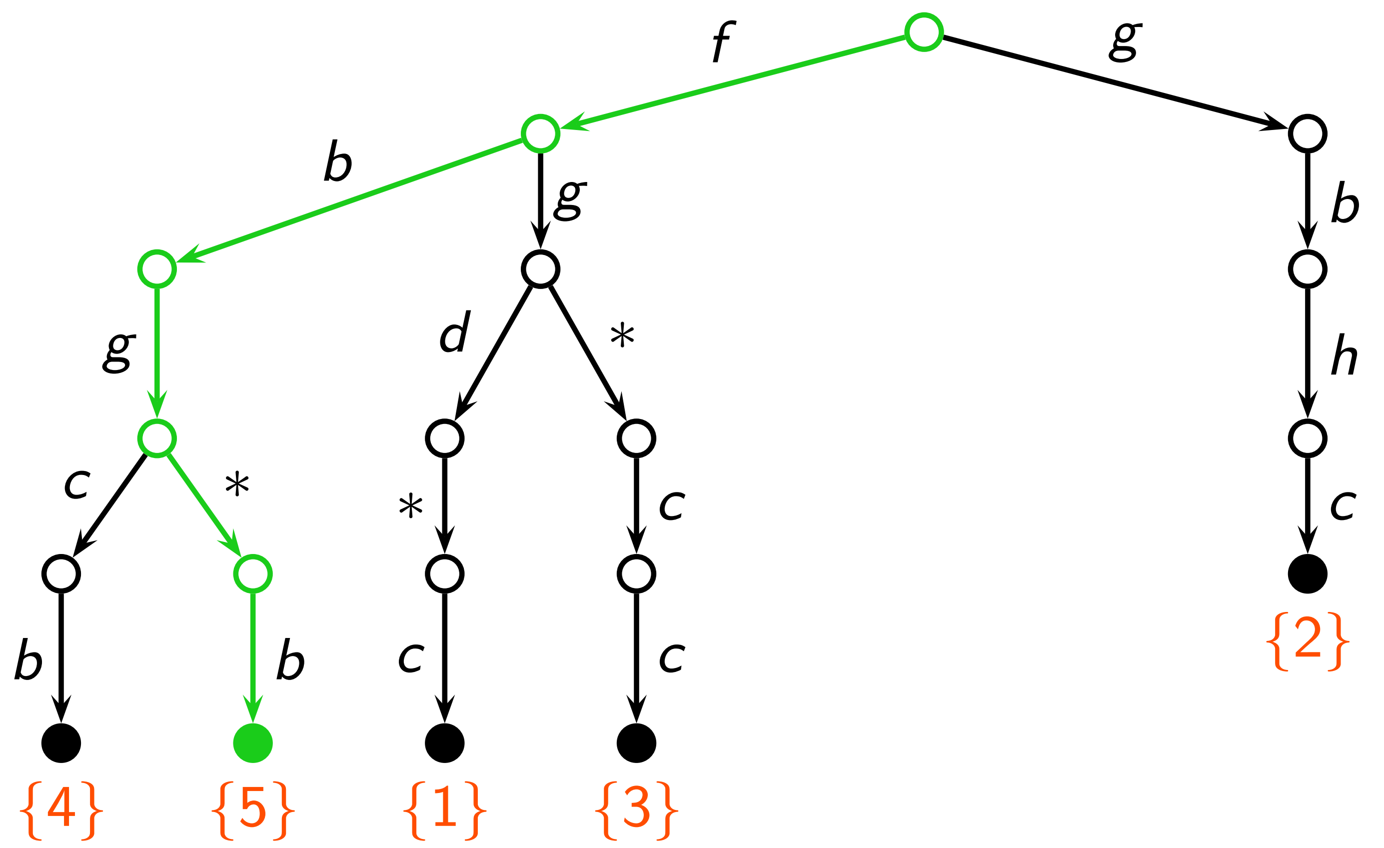
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),$ $f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



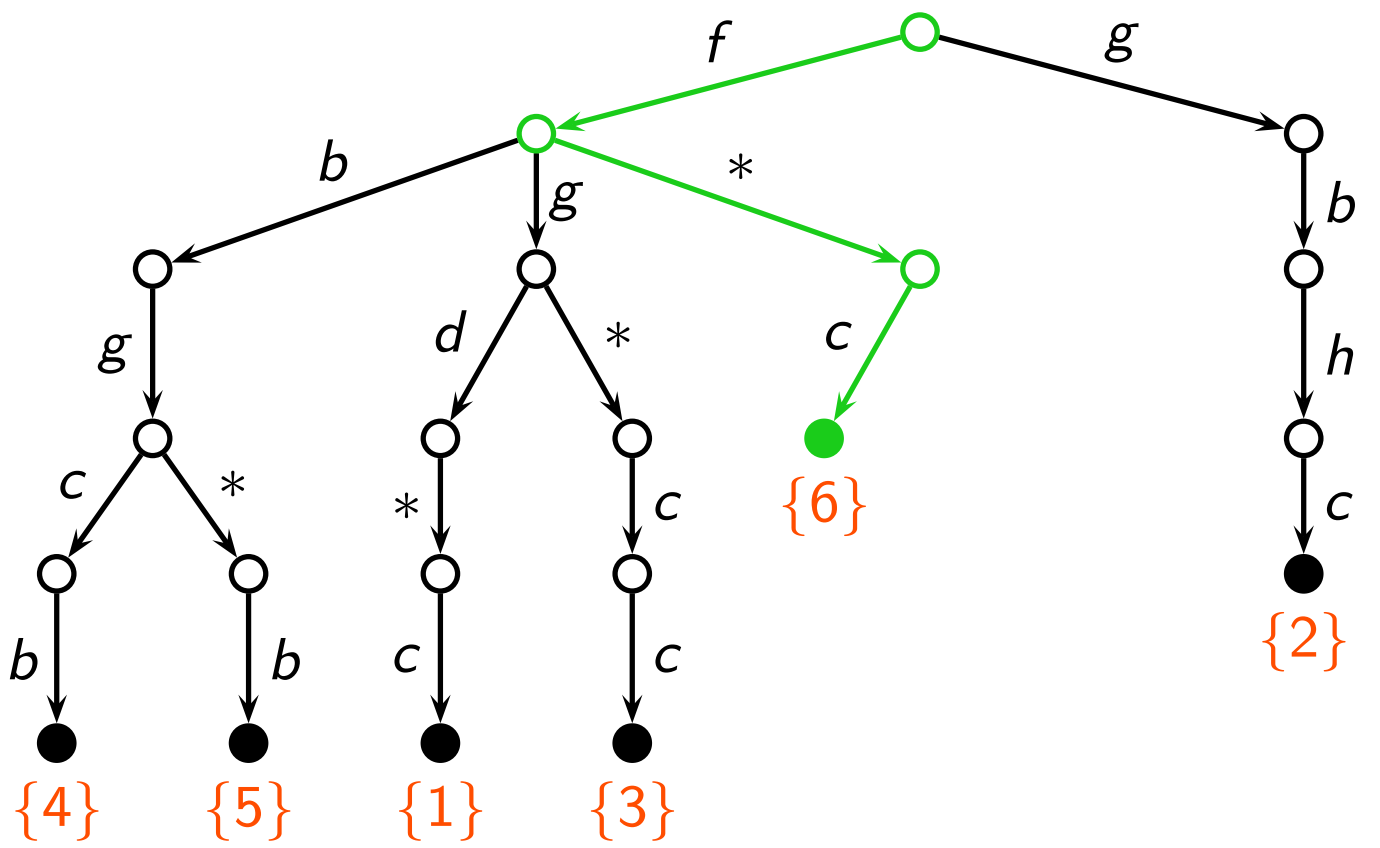
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),$
$f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



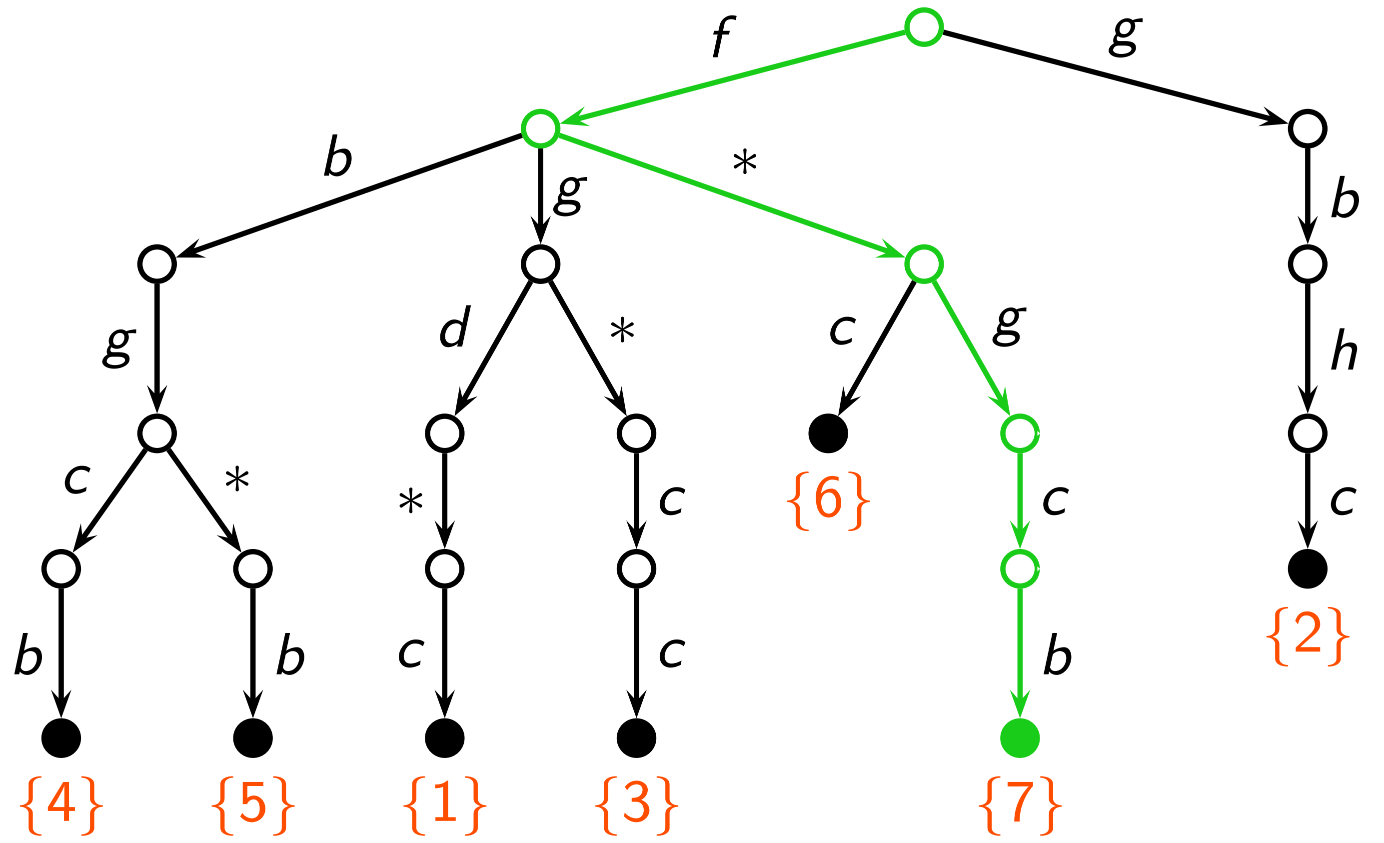
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),$
$f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



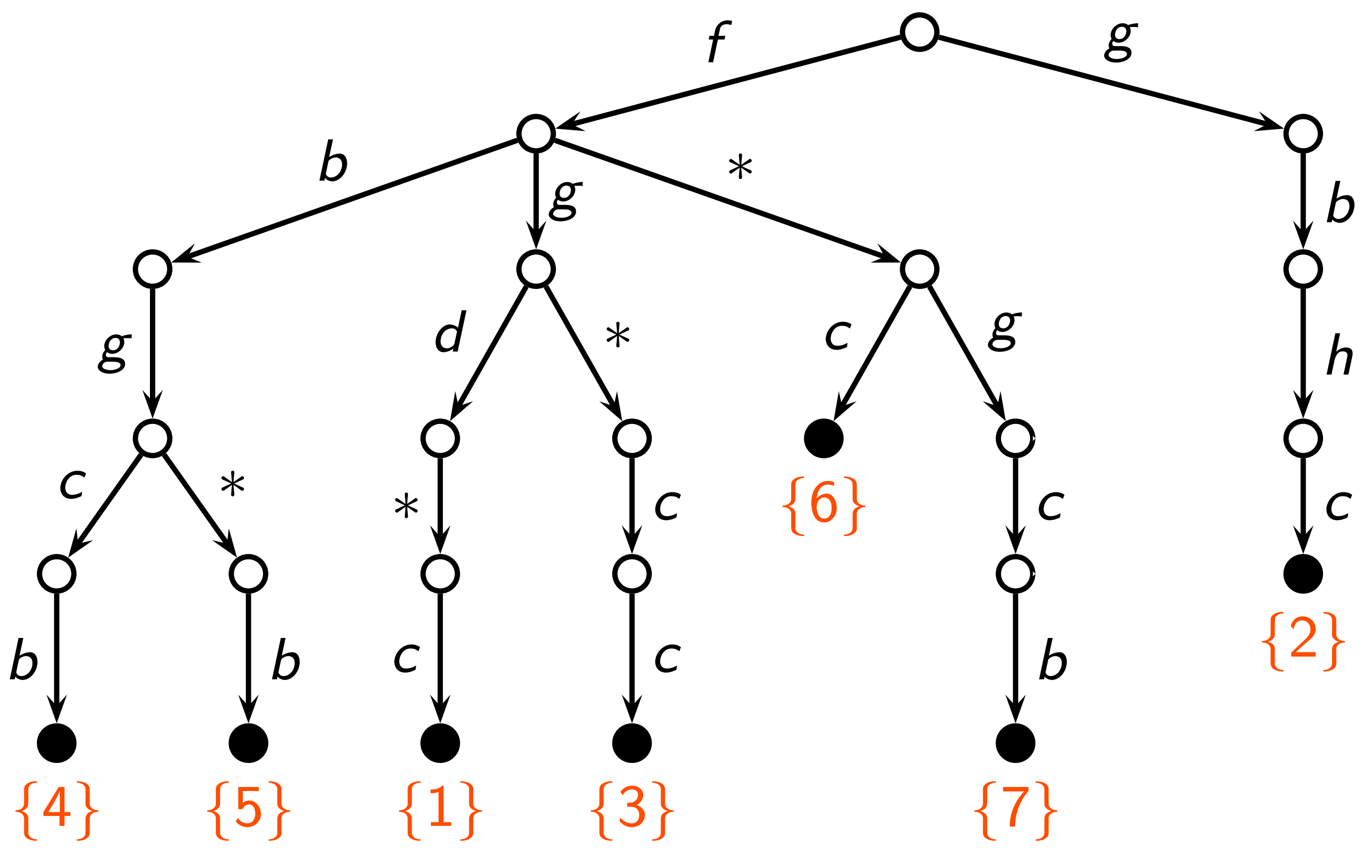
Discrimination Trees

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

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.

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:

$$M \parallel N \ \Rightarrow_{\mathrm{CDCL(T)}} \ M\, L \parallel N$$

if $M \models_T L$
where L is undefined in M and L or $\overline{L}$ occurs in N.

T-Learn:

$$M \parallel N \ \Rightarrow_{\mathrm{CDCL(T)}} \ M \parallel N \cup \{C\}$$

if $N \models_T C$ and each atom of C occurs in N or M.

Satisfiability Modulo Theories (SMT)

T-Backjump:

$$M\ L^{\mathrm{d}}\ M' \parallel N \cup \{C\} \ \Rightarrow_{\mathrm{CDCL(T)}}\ M\ L' \parallel N \cup \{C\}$$

if $M\ L^{\mathrm{d}}\ M' \models \neg C$
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 under M, and
L' or $\overline{L'}$ occurs in N or in $M\ L^{\mathrm{d}}\ M'$.

7.2 Sorted Logics

So far, we have considered only unsorted first-order logic.

In practice, one often considers many-sorted logics:

$\mathit{read}/2$ becomes $\mathit{read} : \mathit{array} \times \mathit{nat} \to \mathit{data}$.

$\mathit{write}/3$ becomes $\mathit{write} : \mathit{array} \times \mathit{nat} \times \mathit{data} \to \mathit{array}$.

Variables: $x : \mathit{data}$

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: $\mathit{array}_{\mathcal{A}}$, $\mathit{nat}_{\mathcal{A}}$.

Interpretations of function and predicate symbols correspond to their declarations:

$\mathit{read}_{\mathcal{A}} : \mathit{array}_{\mathcal{A}} \times \mathit{nat}_{\mathcal{A}} \rightarrow \mathit{data}_{\mathcal{A}}$

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:

$$\frac{N \cup \{C_1 \vee C_2\}}{N \cup \{C_1\} \quad | \quad N \cup \{C_2\}}$$

if $\mathrm{var}(C_1) \cap \mathrm{var}(C_2) = \emptyset$.

Split clauses are smaller and more likely to be usable for simplification.

Splitting tree is explored using intelligent backtracking.

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):

$$\frac{D' \vee t \approx t' \qquad C' \vee s[t] \approx s'}{D' \vee C' \vee s[t'] \approx s'}$$

Superposition (non-ground case):

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

where $\sigma = \mathrm{mgu}(t, u)$ 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:

$$\frac{D' \ \vee \ t' < t \qquad C' \ \vee \ s < s'}{(D' \ \vee \ C' \ \vee \ t' < s')\sigma}$$

where σ 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.