

VL Logik (LVA-Nr. 342208)

Winter Semester 2015/2016

# Propositional Logic: Evaluating the Formulas

Version 2015.2

Armin Biere (biere@jku.at)

Martina Seidl (martina.seidl@jku.at)

# Satisfiability Checking

## Definition (Satisfiability Problem of Propositional Logic (SAT))

Given a formula  $\phi$ , is there an assignment  $\nu$  such that  $[\phi]_\nu = \mathbf{1}$ ?

- oldest **NP**-complete problem (see next slides)
  - checking a solution (is an assignment satisfying a formula?) is easy (polynomial effort)
  - finding a solution is difficult (probably exponential in the worst case, what is easy compared to satisfiability checking in other logics)
- many practical applications (used in industry)
- efficient SAT solvers (solving tools) are available
- other problems can be translated to SAT:

<b>problem</b>	<b>formulation in propositional logic</b>
$\phi$ is valid	$\neg\phi$ is unsatisfiable
$\phi$ is refutable	to $\neg\phi$ is satisfiable
$\phi \Leftrightarrow \psi$	to $\neg(\phi \leftrightarrow \psi)$ is unsatisfiable
$\phi_1, \dots, \phi_n \models \psi$	$\phi_1 \wedge \dots \wedge \phi_n \wedge \neg\psi$ is unsatisfiable

# A Glimpse of Complexity Theory

characterization of computational *hardness* of a problem

*Turing Machine*: machine model for abstract “run time” or “memory usage”  
allows more abstract versions of “run time”, “memory usage”  
the focus is on worst-case *asymptotic* time and space usage

## Definition

problem is in  $\mathcal{O}(f(n))$  iff exists constant  $c$  and an algorithm which needs  $c \cdot f(n)$  steps (in the worst case on a Turing machine) for an input of size  $n$

- logarithmic  $\mathcal{O}(\log n)$ , e.g. binary search on sorted array of size  $n$
- linear  $\mathcal{O}(n)$ , e.g. linear search in list with  $n$  elements
- quadratic  $\mathcal{O}(n^2)$ , e.g. generate list of pairs of  $n$  elements
- exponential  $\mathcal{O}(2^n)$ , e.g. produce all subsets of a set of  $n$  elements

## Definition

polynomial problems: exists fixed  $k$  such that worst-case run time is in  $\mathcal{O}(n^k)$   
the class of polynomial problems is called **P**

# SAT and the Complexity Class **NP**

## Definition

A decision problem asks whether an input belongs to a certain class.

**Prime**: decide whether a number given as input is prime.

**SAT**: decide whether formula given as input is satisfiable.

---

Basic idea of **NP** is to use a “guess” and “check” approach, where “guessing” is non-deterministic, e.g. just a “good” choice has to exist.

## Definition

The class **NP** contains all decision problems which can be decided by a “guessing” and “checking” algorithm in polynomial time in the input size.

Clearly both **Prime** and **SAT** belong to **NP**.

## Theorem (Cook'71)

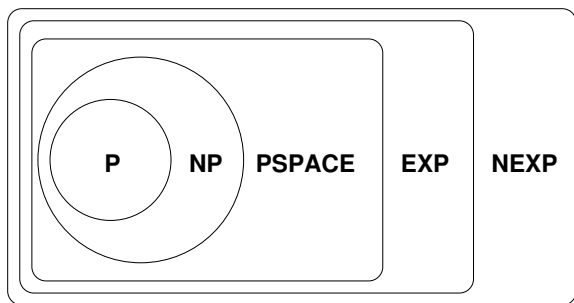
*Any decision problem in **NP** can be reduced (encoded) polynomially into **SAT**.*

Actually, **Prime** can also be solved polynomially (rather complicated).

More on this topic in the “complexity” course.

# Complexity Hierarchy

---



**P** polynomial time

**NP** non-deterministic polynomial time

**PSPACE** polynomial space

**EXP** exponential time

**NEXP** non-deterministic exponential time

except for **P**  $\neq$  **EXP** and **NP**  $\neq$  **NEXP** nothing is known about strict inclusion

# One Simple Algorithm for Satisfiability Checking

---

1 **Algorithm:** evaluate

**Data:** formula  $\phi$

**Result:** 1 iff  $\phi$  is satisfiable

2 **if**  $\phi$  contains a variable  $x$  **then**

3 | pick  $v \in \{\top, \perp\}$

4 | /\* replace  $x$  by truth constant  $v$ , evaluate resulting formula \*/

5 | **if**  $evaluate(\phi[x|v])$  **then return 1;**

6 | **else return**  $evaluate(\phi[x|\bar{v}])$  ;

7 **else**

8 | **switch**  $\phi$  **do**

9 |     **case**  $\top$  **return 1;**

10 |     **case**  $\perp$  **return 0;**

11 |     **case**  $\neg\psi$  **return**  $! evaluate(\psi)$                    /\* true iff  $\psi$  is false \*/ ;

12 |     **case**  $\psi' \wedge \psi''$

13 |     |     **return**  $evaluate(\psi') \ \&\& \ evaluate(\psi'')$      /\* true iff both  $\psi'$  and  $\psi''$  are true \*/

14 |     **case**  $\psi' \vee \psi''$

15 |     |     **return**  $evaluate(\psi') \ || \ evaluate(\psi'')$      /\* true iff  $\psi'$  or  $\psi''$  is true \*/

# Reasoning with (Propositional) Calculi

---

- *goal*: automatically reason about (propositional) formulas, i.e., mechanically show validity/unsatisfiability
- *basic idea*: use syntactical manipulations to prove/refute a formula
- *elements of a calculus*:
  - *axioms*: trivial truths/trivial contradictions
  - *rules*: inference of new formulas
- *approach*: construct a *proof/refutation*, i.e., apply the rules of the calculus until only axioms are inferred. If this is not possible, then the formula is not valid/unsatisfiable.
- *examples of calculi*:
  - sequence calculus: shows validity
  - resolution calculus: shows unsatisfiability

# Sequent Calculus: Sequents

## Definition

A *sequent* is an expression of the form

$$\phi_1, \dots, \phi_n \vdash \psi$$

where  $\phi_1, \dots, \phi_n, \psi$  are propositional formulas.

The formulas  $\phi_1, \dots, \phi_n$  are called *assumptions*,  $\psi$  is called *goal*.

*remarks:*

- *intuitively*  $\boxed{\phi_1, \dots, \phi_n \vdash \psi}$  means goal  $\psi$  follows from  $\{\phi_1, \dots, \phi_n\}$
- *special case*  $n = 0$ :
  - written as  $\boxed{\vdash \psi}$
  - meaning: we have to prove that  $\psi$  is valid
- *notation*: for sequent  $\phi_1, \dots, \phi_n \vdash \psi$ , we write  $K \dots \phi_i \vdash \psi$  if we are only interested in assumption  $\phi_i$
- the assumptions are *orderless* not ordered



# Sequent Calculus: Axiom and Structural Rules

---

- axiom "*goal in assumption*":

If the goal is among the assumptions, the goal can be proved.

$$\text{GoalAssum} \frac{}{K \dots, \psi \vdash \psi}$$

- axiom "*contradiction in assumptions*":

If the assumptions are contradicting, anything can be proved.

$$\text{ContrAssum} \frac{}{K \dots, \phi, \neg\phi \vdash \psi}$$

- rule "*add valid assumption*":

$$\text{ValidAssum} \frac{K \dots, \phi \vdash \psi}{K \dots \vdash \psi} \text{ if } \phi \text{ is valid}$$

# Sequent Calculus: Negation Rules

---

- rules "*contradiction*":

$$A-\neg \frac{K \dots \neg \psi \vdash \perp}{K \dots \vdash \psi}$$

$$P-\neg \frac{K \dots \vdash \neg \phi}{K \dots, \phi \vdash \perp}$$

- rules "*elimination of double negation*":

$$P-\neg_d \frac{K \dots \vdash \psi}{K \dots \vdash \neg \neg \psi}$$

$$A-\neg_d \frac{K \dots, \phi \vdash \psi}{K \dots, \neg \neg \phi \vdash \psi}$$

# Sequent Calculus: Binary Connective Rules

---

- rules "*conjunction*":

$$A-\wedge \frac{K \dots, \phi_1, \phi_2 \vdash \psi}{K \dots, \phi_1 \wedge \phi_2 \vdash \psi}$$

$$P-\wedge \frac{K \dots \vdash \psi_1 \quad K \dots \vdash \psi_2}{K \dots \vdash \psi_1 \wedge \psi_2}$$

- rules "*disjunction*":

$$P-\vee \frac{K \dots, \neg\psi_1 \vdash \psi_2}{K \dots \vdash \psi_1 \vee \psi_2}$$

$$A-\vee \frac{K \dots, \phi_1 \vdash \psi \quad K \dots, \phi_2 \vdash \psi}{K \dots, \phi_1 \vee \phi_2 \vdash \psi}$$

Rules for other connectives like implication " $\rightarrow$ " and equivalence " $\leftrightarrow$ " are constructed accordingly.

# Some Remarks on Sequent Calculus

---

- *premises* of a rule: sequent(s) above the line
- *conclusion* of a rule: sequent below the line
- *axiom*: rule without premises
- *non-deterministic rule*:  $P\text{-}\vee$
- *further non-determinism*: decision which rule to apply next
- *rules with case split*:  $P\text{-}\wedge$ ,  $A\text{-}\vee$
- *proof of formula  $\psi$* 
  1. start with  $\vdash \psi$
  2. apply rules from bottom to top as long as possible, i.e., for given conclusion, find suitable premise(s)
  3. if finally all sequents are axioms then  $\psi$  is valid
- note: there are many variants of the sequent calculus

# One Algorithm for Calculating with Sequent Calculus

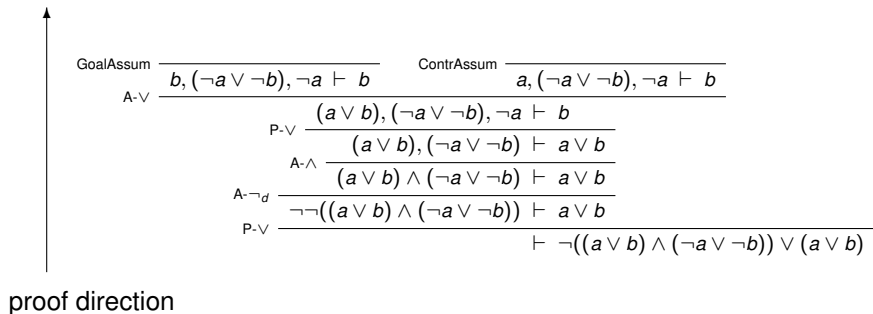
## 1 Algorithm: entails

**Data:** set of assumptions  $\mathcal{A}$ , formula  $\psi$

**Result:** 1 iff  $\mathcal{A}$  entails  $\psi$ , i.e.,  $\mathcal{A} \models \psi$

- 2 if  $\psi = \neg\neg\psi'$  then return **entails** ( $\mathcal{A}, \psi'$ );
- 3 if  $\neg\neg\phi \in \mathcal{A}$  then return **entails** ( $\mathcal{A} \setminus \{\neg\neg\phi\} \cup \{\phi\}, \psi$ );
- 4 if  $\phi_1 \wedge \phi_2 \in \mathcal{A}$  then return **entails** ( $\mathcal{A} \setminus \{\phi_1 \wedge \phi_2\} \cup \{\phi_1, \phi_2\}, \psi$ );
- 5 if  $(\psi \in \mathcal{A})$  or  $(\phi, \neg\phi \in \mathcal{A})$  then return **1**;
- 6 if  $\mathcal{A} \cup \{\psi\}$  contains only literals then return **0**;
- 7 switch  $\psi$  do
  - 8 case  $\perp$ 
    - 9 if  $\neg\phi \in \mathcal{A}$  then return **entails** ( $\mathcal{A} \setminus \{\neg\phi\}, \phi$ );
    - 10 if  $\phi_1 \vee \phi_2 \in \mathcal{A}$  then
      - 11 if **!entails** ( $\mathcal{A} \setminus \{\neg\phi_1 \vee \phi_2\} \cup \{\phi_1\}, \perp$ ) then return **0**;
      - 12 else return **entails** ( $\mathcal{A} \setminus \{\neg\phi_1 \vee \phi_2\} \cup \{\phi_2\}, \perp$ );
  - 13 case  $x$  where  $x$  is a variable return **entails** ( $\mathcal{A} \cup \{\neg x\}, \perp$ );
  - 14 case  $\neg\psi'$  return **entails** ( $\mathcal{A} \cup \{\psi'\}, \perp$ );
  - 15 case  $\psi_1 \vee \psi_2$  return **entails** ( $\mathcal{A} \cup \{\neg\psi_1\}, \psi_2$ );
  - 16 case  $\psi_1 \wedge \psi_2$  return **entails** ( $\mathcal{A}, \psi_1$ ) && **entails** ( $\mathcal{A}, \psi_2$ );

# Proving XOR stronger than OR with the Sequent Calculus



# Refuting XOR stronger than AND with the Sequent Calculus

$$\begin{array}{c}
 \text{GAss} \frac{}{a, (\neg a \vee \neg b) \vdash a} \qquad \text{CAss} \frac{}{b, \neg b \vdash a} \qquad \text{A-}\vee \frac{b, \neg a \vdash a}{b, (\neg a \vee \neg b) \vdash a} \qquad \vdots \qquad \vdots \\
 \text{A-}\vee \frac{a, (\neg a \vee \neg b) \vdash a}{(a \vee b), (\neg a \vee \neg b) \vdash a} \qquad \text{A-}\vee \frac{\vdots}{(a \vee b), (\neg a \vee \neg b) \vdash b} \\
 \text{P-}\wedge \frac{(a \vee b), (\neg a \vee \neg b) \vdash a}{(a \vee b), (\neg a \vee \neg b) \vdash a \wedge b} \\
 \text{A-}\wedge \frac{(a \vee b), (\neg a \vee \neg b) \vdash a \wedge b}{(a \vee b) \wedge (\neg a \vee \neg b) \vdash a \wedge b} \\
 \text{A-}\neg_d \frac{(a \vee b) \wedge (\neg a \vee \neg b) \vdash a \wedge b}{\neg \neg ((a \vee b) \wedge (\neg a \vee \neg b)) \vdash a \wedge b} \\
 \text{P-}\vee \frac{\neg \neg ((a \vee b) \wedge (\neg a \vee \neg b)) \vdash a \wedge b}{\vdash \neg ((a \vee b) \wedge (\neg a \vee \neg b)) \vee (a \wedge b)}
 \end{array}$$

*counter example to validity:*  $a = \perp, b = \top$

# Soundness and Completeness

---

For any calculus important properties are, first *soundness*, i.e. the question “Can only valid formulas be shown as valid?” and second *completeness*, i.e. the question “Is there a proof for every valid formula?”.

---

## *Soundness*

If a formula is shown to be valid in the Gentzen Calculus, then it is valid.

*Proof sketch:* consider each rule individually and show that from valid premises only valid conclusions can be drawn.

---

## *Completeness*

Every valid formula can be proven to be valid in the Gentzen Calculus.

*Proof sketch:* Show that the algorithm terminates and that there is at least one case where it returns false if the formula is not valid.



# Proving Formulas in Normal Form

---

- In practice, formulas of arbitrary structure are quite challenging to handle
  - tree structure
  - simplifications affect only subtrees
- We have seen that CNF and DNF are able to represent every formula
  - so why not use them as input for SAT?
- *Conjunctive Normal Form*
  - refutability is easy to show
  - CNF can be efficiently calculated (polynomial)
- *Disjunctive Normal Form*
  - satisfiability is easy to show
  - complexity is in getting the DNF
- CNF and DNF can be obtained from the *truth tables*
  - exponential many assignments have to be considered
- alternative approach
  - *structural rewritings* which are (satisfiability) equivalence preserving

# Transformation to Normal Form

1. Remove  $\leftrightarrow$ ,  $\rightarrow$ ,  $\oplus$  as follows:

$$\phi \leftrightarrow \psi \Leftrightarrow (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi), \phi \rightarrow \psi \Leftrightarrow \neg\phi \vee \psi,$$

$$\phi \oplus \psi \Leftrightarrow (\phi \vee \psi) \wedge (\neg\phi \vee \neg\psi)$$

2. Transform formula to negation normal form (NNF) by application of laws of De Morgan and elimination of double negation
3. Transform formula to CNF (DNF) by laws of distributivity

## Example

Transform  $\neg(a \leftrightarrow b) \rightarrow (\neg(c \wedge d) \wedge e)$  to an equivalent formula in CNF.

1. a) remove equivalences:  $\Leftrightarrow \neg((a \rightarrow b) \wedge (b \rightarrow a)) \rightarrow (\neg(c \wedge d) \wedge e)$   
b) remove implications:  $\Leftrightarrow \neg\neg((\neg a \vee b) \wedge (\neg b \vee a)) \vee (\neg(c \wedge d) \wedge e)$
2. NNF:  $\Leftrightarrow ((\neg a \vee b) \wedge (\neg b \vee a)) \vee ((\neg c \vee \neg d) \wedge e)$
3.  $\Leftrightarrow ((\neg a \vee b) \vee ((\neg c \vee \neg d) \wedge e)) \wedge ((\neg b \vee a) \vee ((\neg c \vee \neg d) \wedge e))$   
 $\Leftrightarrow (\neg a \vee b \vee \neg c \vee \neg d) \wedge (\neg a \vee b \vee e) \wedge (\neg b \vee a \vee \neg c \vee \neg d) \wedge (\neg b \vee a \vee e)$

## Some Remarks on Normal Forms

---

- The presented transformation to CNF/DNF is exponential in the worst case (e.g., transform  $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \dots \vee (a_n \wedge b_n)$  to CNF).
- For DNF transformation, there is probably no better algorithm.
- For CNF transformation, there are polynomial algorithms.
  - Basic idea: introduce labels for subformulas.
  - Also works for formulas with sharing (circuits).
  - Also known as “Tseitin Encoding”.
- CNF is usually not easier to solve, but easier to handle:
  - compact data structures: a CNF is simply a list of lists of literals.
- CNF very popular in practice: standard input format DIMACS
- For solving satisfiability of CNF formulas, there are many optimization techniques as well as dedicated algorithms.

# Resolution

- the *resolution calculus* consists of the single resolution rule

$$\frac{x \vee C \quad \neg x \vee D}{C \vee D}$$

- $C$  and  $D$  are (possibly empty) clauses
  - the clause  $C \vee D$  is called *resolvent*
  - variable  $x$  is called *pivot*
  - usually antecedent clauses  $x \vee C$  and  $\neg x \vee D$  are assumed not to be tautological, that is  $x \notin C$  and  $x \notin D$ .
- in other words:  
 $(\neg x \rightarrow C), (x \rightarrow D) \models C \vee D$
  - resolution is *sound* and *complete*.
  - the resolution calculus works only on formulas in CNF
  - if the empty clause can be derived then the formula is *unsatisfiable*
  - if no new clause can be generated by application of the resolution rule then the formula is *satisfiable*

## Example

one application of resolution

$$\frac{x \vee y \vee \neg z \quad \neg x \vee y' \vee \neg z}{y \vee \neg z \vee y'}$$

derivation of empty clause:

$$\frac{y \quad \neg y}{\perp}$$

derivation of tautology:

$$\frac{x \vee a \quad \neg x \vee \neg a}{a \vee \neg a}$$

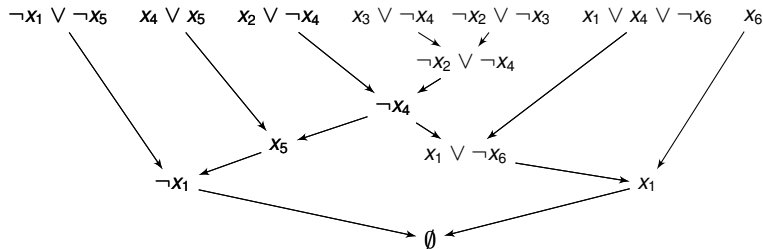
# Resolution Example

---

We *prove unsatisfiability* of

$$\{(\neg x_1 \vee \neg x_5), (x_4 \vee x_5), (x_2 \vee \neg x_4), (x_3 \vee \neg x_4), (\neg x_2 \vee \neg x_3), (x_1 \vee x_4 \vee \neg x_6), (x_6)\}$$

as follows:



The DPLL algorithm is ...

- *old* (invented 1962)
- *easy* (basic pseudo-code is less than 10 lines)
- *popular* (well investigated; also theoretical properties)
- usually realized for *formulas in CNF*
- using *binary constraint propagation (BCP)*
- in its modern form as *conflict drive clause learning (CDCL)*  
basis for state-of-the-art SAT solvers

# Binary Constraint Propagation

## Definition (Binary Constraint Propagation (BCP))

Let  $\phi$  be a formula in CNF containing a unit clause  $C$ , i.e.,  $\phi$  has a clause  $C = (l)$  which consists only of literal  $l$ . Then  $BCP(\phi, l)$  is obtained from  $\phi$  by

- removing all clauses with  $l$
  - removing all occurrences of  $\bar{l}$
- 
- one application of  $BCP$  can trigger other applications of  $BCP$
  - $BCP(\phi)$  denotes all possible applications of  $BCP(\phi, l)$  until fixpoint
  - if  $BCP(\phi)$  produces the empty clause, then the formula  $\phi$  is unsatisfiable
  - if  $BCP(\phi)$  produces the empty CNF, then the formula  $\phi$  is satisfiable

## Example

$\phi = \{(\neg a \vee b \vee \neg c), (a \vee b), (\neg a \vee \neg b), (a)\}$

1.  $\phi' = BCP(\phi, a) = \{(b \vee \neg c), (\neg b)\}$
2.  $\phi'' = BCP(\phi', \neg b) = \{(\neg c)\}$
3.  $\phi''' = BCP(\phi'', c) = \{\} = \top$

# DPLL Algorithm

---

1 **Algorithm:** evaluate

**Data:** formula  $\phi$  in CNF

**Result:** 1 iff  $\phi$  satisfiable

2 **while** 1 **do**

3      $\phi = \text{BCP}(\phi)$

4     **if**  $\phi == \top$  **then return** 1;

5     **if**  $\phi == \perp$  **then**

6         **if** *stack.isEmpty()* **then return** 0;

7          $(I, \phi) = \text{stack.pop}()$

8          $\phi = \phi \wedge I$

9     **else**

10         select literal  $I$  occurring in  $\phi$

11          $\text{stack.push}(\bar{I}, \phi)$

12          $\phi = \phi \wedge I$



## Some Remarks on DPLL

---

- DPLL is the basis for most state-of-the-art SAT solvers
  - like Lingeling <http://fmv.jku.at/lingeling>
  - simpler or more established solvers: MiniSAT, PicoSAT, Cleaneling, ...
- DPLL alone is not enough - powerful optimizations required for efficiency:
  - learning and non-chronological back-tracking (CDCL)
  - reset strategies and phase-saving
  - compact lazy data-structures
  - variable selection heuristics
  - usually combined with preprocessing before search
  - and inprocessing algorithms interleaved with search
- variants of DPLL are also used for other logics:
  - quantified propositional logic (QBF)
  - satisfiability modulo theories (SMT)
- challenge to parallelize
  - some successful attempts: ManySAT, Plingeling, Penelope, Treengeling, ...