Logic

(Métodos Formais em Engenharia de Software)

Maria João Frade

Departmento de Informática
Universidade do Minho

2011/2012

What is a (formal) logic?

Logic is defined as the study of the principles of reasoning. One of its branches is symbolic logic, that studies formal logic.

- A formal logic is a language equipped with rules for deducing the truth of one sentence from that of another.
- A logic consists of
  - A *logical language* in which (well-formed) sentences are expressed.
  - A *semantics* that defines the intended interpretation of the symbols and expressions of the logical language.
  - A *proof system* that is a framework of rules for deriving valid judgments.
- Examples: propositional logic, first-order logic, higher-order logic, modal logics, ...

What is a logical language?

A logical language consists of

- *logical symbols* whose interpretations are fixed
- *non-logical symbols* whose interpretations vary

These symbols are combined together to form *well-formed formulas*.

Logic and computer science

- Logic and computer science share a symbiotic relationship
  - Logic provides language and methods for the study of theoretical computer science.
  - Computers provide a concrete setting for the implementation of logic.
- Formal logic makes it possible to calculate consequences at the symbolic level, so computers can be used to automate such symbolic calculations.
- Moreover, logic can be used to model the situations we encounter as computer science professionals, in such a way that we can reason about them formally.

Motivation

- Many applications of formal methods rely on generating formulas of a logical system and investigate about their validity or satisfiablility.
- Constraint-satisfaction problems arise in diverse application areas, such as
  - software and hardware verification
  - static program analysis
  - test-case generation
  - scheduling and planning
  - ...
- These problems can be encoded by logical formulas. Solvers for such formulations (SAT solvers and SMT solvers) play a crucial rule in their resolution.

Motivation

- Increased attention has led to enormous progress in this area in the last decade. Modern SAT procedures can check formulas with hundreds of thousands of variables. Similar progress has been observed for SMT solvers for more commonly occurring theories.
- SMT solvers are the core engine of many tools for program analysis, testing and verification.
- Modern SMT solvers integrate specialized theory solvers with propositional satisfiability search techniques.

Goals of this course

- Review the basic concepts of Propositional Logic and First-Order Logic.
- Address the issues of decidability of logical systems.
- Talk about the algorithms that underlie a large number of automatic proof tools.
- Illustrate the use of automatic theorem provers and proof assistants.

Course overview

- **Classical Propositional Logic**
  - syntax; semantics; validity; satisfiability; decidability; complexity
  - normal forms; DPLL procedure; SAT solvers; modeling with PL
- **Classical First-Order Logic**
  - syntax; semantics; validity; satisfiability; modeling with FOL
  - normal forms; Herbrandization; Skolemization; Herbrand's theorem; semi-decidability; decidable fragments
  - FOL with equality; many-sorted FOL
- **First-Order Theories**
  - basic definitions; decidability issues; several theories: equality, integers, linear arithmetic, reals, arrays; combining theories
  - satisfiability modulo theories; SMT solvers; SMT-LIB; applications
- **Natural Deduction**
  - natural deduction proof system for propositional and predicate logic; forward and backward reasoning
  - soundness; completeness; compactness
  - proof assistants; the Coq system

Bibliography

Books

- [Huth&Ryan 2004] *Logic in Computer Science: Modelling and Reasoning About Systems.* Michael Huth & Mark Ryan. Cambridge University Press; 2nd edition (2004).
- [Bradley&Manna 2007] *The Calculus of Computation: Decision Procedures with Applications to Verification.* Aaron R. Bradley & Zohar Manna. Springer (2007).
- [RSD 2011] *Rigorous Software Development: An Introduction to Program Verification.* J.B. Almeida & M.J. Frade & J.S. Pinto & S.M. de Sousa. Springer (2011)
- [Bertot&Castéran 2004] *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* Yves Bertot & Pierre Castéran. Springer (2004)

Bibliography

Papers

- *Satisfiability Modulo Theories: An Appetizer.* Leonardo de Moura & Nikolaj Bjoener. Invited paper to SBMF 2009, Gramado, Brazil.
- *Satisfiability Modulo Theories: Introduction and Applications.* Leonardo de Moura & Nikolaj Bjorner. Communications of the ACM, September 2011.
- *Automated Deduction for Verification.* Natarajan Shankar, ACM Computing Surveys, Vol. 41, No. 4, Article 20, October 2009.
- *Coq in a Hurry*. Yves Bertot. February 2010.

(Classical) Propositional Logic

Roadmap

- **Classical Propositional Logic**
  - syntax; semantics; validity; satisfiability; decidability; complexity
  - normal forms; DPLL procedure; SAT solvers; modeling with PL
- Classical First-Order Logic
- First-Order Theories
- Natural Deduction

Introduction

- The language of propositional logic is based on propositions, or declarative sentences which one can, in principle, argue as being "true" or "false".
  "The capital of Portugal is Braga."
  "D. Afonso Herriques was the first king of Portugal."
- Propositional symbols are the atomic formulas of the language. More complex sentences are constructed using logical connectives.
- In classical propositional logic (PL) each sentence is either true or false.
- In fact, the content of the propositions is not relevant to PL. PL is not the study of truth, but of the relationship between the truth of one statement and that of another.

Syntax

The alphabet of a first-order language is organised into the following categories.

- *Propositional variables*: $P, Q, R, \ldots \in \mathcal{V}_{\text{Prop}}$ (a countably infinite set)
- *Logical connectives*: $\bot$ (*false*) ,$\top$ (*true*), $\neg$ (*not*), $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implies*)
- *Auxiliary symbols*: "(" and ")".

The set **Form** of *formulas* of propositional logic is given by the abstract syntax

$$\textbf{Form} \ni A, B \ ::= \ P \mid \bot \mid \top \mid (\neg A) \mid (A \wedge B) \mid (A \vee B) \mid (A \rightarrow B)$$

We let $A, B, C, F, G, H, \ldots$ range over **Form**.

Outermost parenthesis are usually dropped. In absence of parentheses, we adopt the following convention about precedence. Ranging from the highest precedence to the lowest, we have respectively: $\neg$, $\wedge$, $\vee$ and $\rightarrow$. All binary connectives are right-associative.

Semantics

The semantics of a logic provides its meaning. What exactly is meaning?
In propositional logic, meaning is given by the truth values *true* and *false*, where true $\neq$ false. We will represent true by 1 and false by 0.

An *assignment* is a function $\mathcal{A} : \mathcal{V}_{\text{Prop}} \rightarrow \{0, 1\}$, that assigns to every propositional variable a truth value.
An assignment $\mathcal{A}$ naturally extends to all formulas, $\mathcal{A}$: **Form** $\rightarrow \{0, 1\}$.
The truth value of a formula is computed using *truth tables*:

F	A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$\bot$	$\top$
$\mathcal{A}_1(F)$	0	1	1	0	1	1	0	1
$\mathcal{A}_2(F)$	0	0	1	0	0	1	0	1
$\mathcal{A}_3(F)$	1	1	0	1	1	1	0	1
$\mathcal{A}_4(F)$	1	0	0	0	1	0	0	1

Semantics

Let $\mathcal{A}$ be an assignment and let F be a formula.
If $\mathcal{A}(F) = 1$, then we say F *holds* under assignment $\mathcal{A}$, or $\mathcal{A}$ *models* F.
We write $\mathcal{A} \models F$ iff $\mathcal{A}(F) = 1$, and $\mathcal{A} \not\models F$ iff $\mathcal{A}(F) = 0$.

An alternative (inductive) definition of $\mathcal{A} \models F$ is

$$\begin{array}{lll}
\mathcal{A} \models \top & & \\
\mathcal{A} \not\models \bot & & \\
\mathcal{A} \models P & \text{iff} & \mathcal{A}(P) = 1 \\
\mathcal{A} \models \neg A & \text{iff} & \mathcal{A} \not\models A \\
\mathcal{A} \models A \wedge B & \text{iff} & \mathcal{A} \models A \text{ and } \mathcal{A} \models B \\
\mathcal{A} \models A \vee B & \text{iff} & \mathcal{A} \models A \text{ or } \mathcal{A} \models B \\
\mathcal{A} \models A \rightarrow B & \text{iff} & \mathcal{A} \not\models A \text{ or } \mathcal{A} \models B
\end{array}$$

Validity, satisfiability, and contradiction

A formula F is

valid	iff	it holds under every assignment. We write $\models F$. A valid formula is called a *tautology*.
satisfiable	iff	it holds under some assignment.
unsatisfiable	iff	it holds under no assignment. An unsatisfiable formula is called a *contradiction*.
refutable	iff	it is not valid.

Proposition

F is valid iff $\neg F$ is a contradiction

$(A \wedge (A \to B)) \to B$ is valid. $A \to B$ is satisfiable and refutable.
$A \wedge \neg A$ is a contradiction.

Consequence and equivalence

- $F \models G$ iff for every assignment $\mathcal{A}$, if $\mathcal{A} \models F$ then $\mathcal{A} \models G$. We say G is a *consequence* of F.
- $F \equiv G$ iff $F \models G$ and $G \models F$. We say F and G are *equivalent*.
- Let $\Gamma = \{F_1, F_2, F_3, \dots\}$ be a set of formulas.
  $\mathcal{A} \models \Gamma$ iff $\mathcal{A} \models F_i$ for each formula F_i in Γ. We say $\mathcal{A}$ *models* Γ.
  $\Gamma \models G$ iff $\mathcal{A} \models \Gamma$ implies $\mathcal{A} \models G$ for every assignment $\mathcal{A}$. We say G is a *consequence* of Γ.

Proposition

- $F \models G$ iff $\models F \to G$
- $\Gamma \models G$ and Γ finite iff $\models \bigwedge \Gamma \to G$

Some basic equivalences

$$
\begin{array}{lcl}
A \vee A & \equiv & A \\
A \wedge A & \equiv & A \\
\\
A \vee B & \equiv & B \vee A \\
A \wedge B & \equiv & B \wedge A \\
\\
A \wedge (A \vee B) & \equiv & A \\
\\
A \wedge (B \vee C) & \equiv & (A \wedge B) \vee (A \wedge C) \\
A \vee (B \wedge C) & \equiv & (A \vee B) \wedge (A \vee C) \\
\\
\neg(A \vee B) & \equiv & \neg A \wedge \neg B \\
\neg(A \wedge B) & \equiv & \neg A \vee \neg B
\end{array}
$$

$$
\begin{array}{lcl}
A \wedge \neg A & \equiv & \bot \\
A \vee \neg A & \equiv & \top \\
\\
A \wedge \top & \equiv & A \\
A \vee \top & \equiv & \top \\
\\
A \wedge \bot & \equiv & \bot \\
A \vee \bot & \equiv & A \\
\\
\neg\neg A & \equiv & A \\
\\
A \to B & \equiv & \neg A \vee B
\end{array}
$$

Consistency

Let $\Gamma = \{F_1, F_2, F_3, \dots\}$ be a set of formulas.

- Γ is *consistent* or *satisfiable* iff there is an assignment that models Γ.
- We say that Γ is *inconsistent* iff it is not consistent and denote this by $\Gamma \models \bot$.

Proposition

- $\{F, \neg F\} \models \bot$
- If $\Gamma \models \bot$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \models \bot$.
- $\Gamma \models F$ iff $\Gamma, \neg F \models \bot$

Theories

A set of formulas $\mathcal{T}$ is *closed* under logical consequence iff for all formulas F, if $\mathcal{T} \models F$ then $F \in \mathcal{T}$.

$\mathcal{T}$ is a *theory* iff it is closed under logical consequence. The elements of $\mathcal{T}$ are called *theorems*.

Let Γ be a set of formulas.

$\mathcal{T}(\Gamma) = \{F \mid \Gamma \models F\}$ is called the *theory* of Γ.
The formulas of Γ are called *axioms* and the theory $\mathcal{T}(\Gamma)$ is *axiomatizable*.

Substitution

- Formula G is a *subformula* of formula F if it occurs syntactically within F.
- Formula G is a *strict subformula* of F if G is a subformula of F and $G \neq F$

Substitution theorem

Suppose $F \equiv G$. Let H be a formula that contains F as a subformula. Let H' be the formula obtained by replacing some occurrence of F in H with G. Then $H \equiv H'$.

Adquate sets of connectives for PL

There is some redundancy among the logical connectives.

Some smaller adquate sets of conectives for PL:

$\{\wedge, \neg\}$ $\quad \bot \equiv P \wedge \neg P, \quad \top \equiv \neg(P \wedge \neg P),$
$A \vee B \equiv \neg(\neg A \wedge \neg B), \quad A \rightarrow B \equiv \neg(A \wedge \neg B)$

$\{\vee, \neg\}$ $\quad \top \equiv A \vee \neg A, \quad \bot \equiv \neg(A \vee \neg A),$
$A \wedge B \equiv \neg(\neg A \vee \neg B), \quad A \rightarrow B \equiv A \vee B$

$\{\rightarrow, \neg\}$ $\quad \top \equiv A \rightarrow A, \quad \bot \equiv \neg(A \rightarrow A),$
$A \vee B \equiv \neg A \rightarrow B, \quad A \wedge B \equiv \neg(A \rightarrow \neg B)$

$\{\rightarrow, \bot\}$ $\quad \neg A \equiv A \rightarrow \bot, \quad \top \equiv A \rightarrow A,$
$A \vee B \equiv (A \rightarrow \bot) \rightarrow B), \quad A \wedge B \equiv (A \rightarrow B \rightarrow \bot) \rightarrow \bot$

Decidability

A *decision problem* is any problem that, given certain input, asks a question to be answered with a "yes" or a "no".

A *solution* to a decision problem is a program that takes problem instances as input and always terminates, producing a correct "yes" or "no" output. A decision problem is *decidable* if it has a solution.

Given formulas F and G as input, we may ask:

Decision problems

Validity problem:	"Is F valid ?"
Satisfiability problem:	"Is F satisfiable ?"
Consequence problem:	"Is G a consequence of F ?"
Equivalence problem:	"Are F and G equivalent ?"

All these problems are **decidable!**

Decidability

Any algorithm that works for one of these problems also works for all of these problems!

F is satisfiable	iff	$\neg F$ is not valid
$F \models G$	iff	$\neg(F \to G)$ is not satisfiable
$F \equiv G$	iff	$F \models G$ and $G \models F$
F is valid	iff	$F \equiv \top$

Truth-table method

For the satisfiability problem, we first compute a truth table for F and then check to see if its truth value is ever one.

This algorithm certainly works, but is very inefficient.
Its exponential-time! $\mathcal{O}(2^n)$

If F has n atomic formulas, then the truth table for F has 2^n rows.

Complexity

An algorithm is *polynomial-time* if there exists a polynomial $p(x)$ such that given input of size n, the algorithm halts in fewer than $p(n)$ steps. The class of all decision problems that can be resolved by some polynomial-time algorithm is denoted by **P** (or PTIME).

It is not known whether the satisfiability problem (and the other three decision problems) is in **P**.

We do not know of a polynomial-time algorithm for satisfiability.

If it exists, then **P** = **NP** !

The Satisfiability problem for PL (PSAT) is **NP**-complete.

Complexity

- A *deterministic* algorithm is a step-by-step procedure. At any stage of the algorithm, the next step is completely determined.
- In contrast, a *non-deterministic* algorithm may have more than one possible "next step" at a given stage. That is, there may be more than one computation for a given input.

NP *(non-deterministic polynomial-time)* decision problems

Let PROB be an arbitrary decision problem. Given certain input, PROB produces an output of either "yes" or "no". Let Y be the set of all inputs for which PROB produces the output of "yes" and let N be the analogous set of inputs that produce output "no".

- If there exists a non-deterministic algorithm which, given input x, can produce the output "yes" in polynomial-time if and only if $x \in Y$, then PROB is in **NP**.
- If there exists a non-deterministic algorithm which, given input x, can produce the output "no" in polynomial-time if and only if $x \in N$, then PROB is in **coNP**.

Complexity

Essentially, a decision problem is in **NP** (**coNP**) if a "yes" ("no") answer can be obtained in polynomial-time by guessing.

Satisfiability problem is **NP**. Given a formula F compute *an* assignment $\mathcal{A}$ for F. If $\mathcal{A}(F) = 1$, then F is satisfiable.

Validity problem is **coNP**.

NP-complete

A decision problem Π is **NP**-*complete* if it is in **NP** and for every problem Π_1 in **NP**, Π_1 is *polynomially reducible* to Π ($\Pi_1 \propto \Pi$).

Cook's theorem (1971)

PSAT is **NP**-complete.

Decision procedures for satisfiability

We have seen the truth-table method for deciding satisfiability. However this method is very inefficient (exponential on the number of atomic propositions).

The are more efficient algorithms (although not polynomial-time) for the decision problems presented above:

- Semantic Tableaux
- Resolution
- Davis-Putnam-Logemann-Loveland algorithm (DPLL)

The most successful SAT solvers are based on the DPLL algoritm. This algoritm recives as input a formula in a specific syntatical format. So, one has first to transform the input formula to this specific format preserving satisfiability.

Normal forms

- A *literal* is a propositional variable or its negation.
- A formula A is in *negation normal form (NNF)*, if the only connectives used in A are $\neg$, $\wedge$ and $\vee$, and negation only appear in literals.
- A *clause* is a disjunction of literals.
- A formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses, i.e., it has the form

$$\bigwedge_i \left(\bigvee_j l_{ij} \right)$$

where l_{ij} is the j-th literal in the i-th clause.

Normalization

Transforming a formula F to equivalent formula F' in NNF can be computed by repeatedly replace any subformula that is an instance of the left-hand-side of one of the following equivalences by the corresponding right-hand-side

$$A \to B \equiv \neg A \vee B \qquad \neg\neg A \equiv A$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B \qquad \neg(A \vee B) \equiv \neg A \wedge \neg B$$

This algoritm is linear on the size of the formula.

Normalization

To transform a formula already in NNF into an equivalent CNF, apply recursively the following equivalences (left-to-right):

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C) \qquad (A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$$
$$A \wedge \bot \equiv \bot \qquad \bot \wedge A \equiv \bot \qquad A \wedge \top \equiv A \qquad \top \wedge A \equiv A$$
$$A \vee \bot \equiv A \qquad \bot \vee A \equiv A \qquad A \vee \top \equiv \top \qquad \top \vee A \equiv \top$$

This althoritm converts a NNF formula into an equivalente CNF, but its worst case is exponential on the size of the formula.

Example

Compute the CNF of $((P \to Q) \to P) \to P$

The first step is to compute its NNF by transforming implications into disjunctions and pushing negations to proposition symbols:

$$\begin{aligned} ((P \to Q) \to P) \to P &\equiv \neg((P \to Q) \to P) \lor P \\ &\equiv \neg(\neg(P \to Q) \lor P) \lor P \\ &\equiv \neg(\neg(\neg P \lor Q) \lor P) \lor P \\ &\equiv \neg((P \land \neg Q) \lor P) \lor P \\ &\equiv (\neg(P \land \neg Q) \land \neg P) \lor P \\ &\equiv ((\neg P \lor Q) \land \neg P) \lor P \end{aligned}$$

To reach a CNF, distributivity is then applied to pull the conjunction outside:

$$((\neg P \lor Q) \land \neg P) \lor P \equiv (\neg P \lor Q \lor P) \land (\neg P \lor P)$$

Worst-case example

Compute the CNF of $(P_1 \land Q_1) \lor (P_2 \land Q_2) \lor \ldots \lor (P_n \land Q_n)$

$$\begin{aligned} & (P_1 \land Q_1) \lor (P_2 \land Q_2) \lor \ldots \lor (P_n \land Q_n) \\ \equiv\ & (P_1 \lor (P_2 \land Q_2) \lor \ldots \lor (P_n \land Q_n)) \land (Q_1 \lor (P_2 \land Q_2) \lor \ldots \lor (P_n \land Q_n)) \\ \equiv\ & \ldots \\ \equiv\ & (P_1 \lor \ldots \lor P_n) \land \\ & (P_1 \lor \ldots \lor P_{n-1} \lor Q_n) \land \\ & (P_1 \lor \ldots \lor P_{n-2} \lor Q_{n-1} \lor P_n) \land \\ & (P_1 \lor \ldots \lor P_{n-2} \lor Q_{n-1} \lor Q_n) \land \\ & \ldots \land \\ & (Q_1 \lor \ldots \lor Q_n) \end{aligned}$$

The original formula has $2n$ literals, while the equivalent CNF has 2^n clauses, each with n literals.
The size of the formula increases exponentially.

Definitional CNF

Equisatisfiability

Two formulas F and F' are *equisatisfiable* when F is satisfiable iff F' is satisfiable.

Any propositional formula can be transformed into a equisatisfiable CNF formula with only linear increase in the size of the formula.
The price to be paid is n new Boolean variables, where n is the number of logical conectives in the formula.
This transformation can be done via *Tseitin's encoding*.

This tranformation compute what is called the *definitional CNF* of a formula, because they rely on the introduction of new proposition symbols that act as names for subformulas of the original formula.

Tseitin's encoding

Tseitin transformation

1. Introduce a new fresh variable for each compound subformula.
2. Assign new variable to each subformula.
3. Encode local constraints as CNF.
4. Make conjunction of local constraints and the root variable.

- This transformation produces a formula that is equisatisfiable: the result is satisfiable iff and only the original formula is satisfiable.
- One can get a satisfying assignment for original formula by projecting the satisfying assignment onto the original variables.

There are various optimizations that can be performed in order to reduce the size of the resulting formula and the number of additional variables.

Tseitin's encoding: an example

Encode $P \rightarrow Q \wedge R$

1. $$\overbrace{P \rightarrow \underbrace{Q \wedge R}_{A_2}}^{A_1}$$
2. We need to satisfy A_1 together with the following equivalences
$$A_1 \leftrightarrow (P \rightarrow A_2) \qquad\qquad A_2 \leftrightarrow (Q \wedge R)$$
3. These equivalences can be rewritten in CNF as $(A_1 \vee P) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee \neg P \vee A_2)$ and $(\neg A_2 \vee Q) \wedge (\neg A_2 \vee R) \wedge (A_2 \vee \neg Q \vee \neg R)$, respectively.
4. The CNF which is equisatisfiable with $P \rightarrow (Q \wedge R)$ is
$$\begin{array}{lll} A_1 & \wedge & (A_1 \vee P) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee \neg P \vee A_2) \\ & \wedge & (\neg A_2 \vee Q) \wedge (\neg A_2 \vee R) \wedge (A_2 \vee \neg Q \vee \neg R) \end{array}$$

CNFs

- Recall that CNFs are formulas with the following shape (each l_{ij} denotes a literal):
$$(l_{11} \vee l_{12} \vee \ldots \vee l_{1k}) \wedge \ldots \wedge (l_{n1} \vee l_{n2} \vee \ldots \vee l_{nj})$$
- Associativity, commutativity and idempotence of both disjunction and conjunction allow us to treat each CNF as a set of sets of literals S
$$S = \{\{l_{11}, l_{12}, \ldots, l_{1k}\}, \ldots, \{l_{n1}, l_{n2}, \ldots, l_{nj}\}\}$$
- An empty inner set will be identified with $\perp$, and an empty outer set with $\top$.

CNFs validity

- The strict shape of CNFs make them particularly suited for checking validity problems.
  - A CNF is a tautology iff all of its clauses are tautologies.
  - A clause c is a tautology precisely when there exists a proposition symbol P such that $\{P, \neg P\} \subseteq c$ (such clauses said to be *closed*).
  - So, a CNF is a tautology iff all of its clauses are closed.
- However, the applicability of this simple criterion for validity is compromised by the potential exponential growth in the CNF transformation.
- This limitation is overcomed considering instead SAT, with satisfiability preserving CNFs (definitional CNF). Recall that
$$F \text{ is valid} \quad \text{iff} \quad \neg F \text{ is unsatisfiable}$$

CNFs satisfiability

- A formula F is satisfiable if it holds under some assignment (i.e. if one can find a model for it).
- *A CNF is satisfied* by an assignment if all its clauses are satisfied. And *a clause is satisfied* if at least one of its literals is satisfied.
- The ideia is to incrementally construct an assignment compatible with a CNF.
- Most current state-of-the-art SAT solvers are based on the *Davis-Putnam-Logemann-Loveland (DPLL)* framework: in this framework the tool can be thought of as traversing and backtracking on a binary tree, in which
  - internal nodes represent *partial assignments*
  - and leaves represent *full assignments*

DPLL procedure

- The DPLL algorithm is a complete, backtracking-based algorithm for deciding the satisfiability of propositional CNFs. It was introduced in the early 1960s.
- The DPLL algorithm progresses by making a decision about a variable an its value, propagates implications of this decision that are easy to detect, and backtracks in case a conflict is detected in the form of a falsified clause.

Opposite of a literal

The *opposite* of a literal l, written $-l$, is defined by

$$-l = \begin{cases} \neg P & \text{, if } l = P \\ P & \text{, if } l = \neg P \end{cases}$$

DPLL procedure

If we fix the assignment of a particular proposition symbol, we are able to simplify the corresponding CNF accordingly.

When we set a literal l to be true,

- any clause that has the literal l is now guaranteed to be satisfied, so we throw it away for the next part of the search.
- any clause that had the literal $-l$, on the other hand, must rely on one of the other literals in the clause, hence we throw out the literal $-l$ before going forward.

Simplification of S assuming l holds

$$S|_l = \{c \backslash \{-l\} \mid c \in S \text{ and } l \notin c\}$$

DPLL procedure

If a CNF S contains a clause that consists of a single literal (called *unit clause*), we know for certain that the literal must be set to true and S can be simplified. This is the premise behind the procedure called *unit propagation*.

Unit propagation

This procedure receives a CNF S and a partial assignment $\mathcal{A}$ (represented by a set of literals – where P denote that P is set to true, and $\neg P$ that P is set to false) and apply unit propagation while it is possible and worthwhile.

UNIT_PROPAGATE $(S, \mathcal{A})$ {
  while $\{\} \notin S$ and S has a unit clause l **do** {
    $S \leftarrow S|_l$;
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{l\}$
  }
}

DPLL procedure

Traditionally the DPLL algorithm is presented as a recursive procedure. The procedure DPLL is called with the CNF and a partial assignment.

1. First the formula is simplified by the unit propagation procedure.
2. If the simplified formula is $\{\}$, the original formula is satisfiable by the current assignment.
3. If the simplified formula contains the empty clause, this means that the current assignment does not satisfy the original formula.
4. Otherwise, algorithm runs by choosing a literal l, assigning a truth value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable; if this is the case, the original formula is satisfiable; otherwise, the same recursive check is done assuming the opposite truth value for l.

Unsatisfiability of the complete formula can only be detected after exhaustive search.

DPLL procedure

DPLL algorithm

This procedure is called with a CNF S and a partial assignment $\mathcal{A}$ (initially $\emptyset$).

$\mathrm{DPLL}(S, \mathcal{A})$ {
  $\textsc{unit\_propagate}(S, \mathcal{A})$;
  if $S = \{\}$ **then return** SAT;
  else if $\{\} \in S$ **then return** UNSAT;
  else { $l \leftarrow$ a literal of S ;
    if DPLL $(S|_l, \mathcal{A} \cup \{l\}) =$ SAT **then return** SAT;
    else return DPLL $(S|_{-l}, \mathcal{A} \cup \{-l\})$
  }
}

DPLL procedure

Is $(\neg P \vee Q) \wedge (\neg P \vee R) \wedge (Q \vee R) \wedge (\neg Q \vee \neg R)$ satisfiable?

	S	$\mathcal{A}$
DPLL	$\{\{\neg P, Q\}, \{\neg P, R\}, \{Q, R\}, \{\neg Q, \neg R\}\}$	$\emptyset$
UNIT_PROPAGATE		
	$\{\{\neg P, Q\}, \{\neg P, R\}, \{Q, R\}, \{\neg Q, \neg R\}\}$	$\emptyset$
choose $l = P$		
DPLL	$\{\{Q\}, \{R\}, \{Q, R\}, \{\neg Q, \neg R\}\}$	$\{P = 1\}$
UNIT_PROPAGATE		
	$\{\{\}\}$	$\{P = 1, Q = 1, R = 1\}$
$-l = \neg P$		
DPLL	$\{\{Q, R\}, \{\neg Q, \neg R\}\}$	$\{P = 0\}$
UNIT_PROPAGATE		
	$\{\{Q, R\}, \{\neg Q, \neg R\}\}$	$\{P = 0\}$
choose $l = Q$		
DPLL	$\{\{\neg R\}\}$	$\{P = 0, Q = 1\}$
UNIT_PROPAGATE		
	$\{\}$	$\{P = 0, Q = 1, R = 0\}$

DPLL procedure

- Probably the most important element in SAT solving is the strategy by which the literals are chosen. This strategy is called the *decision heuristic* of the SAT solver.
- Optimisations to the DPLL procedure are usually explored to avoid unnecessary branches during execution. Such optimizations include:
  - defining new data structures to make the algorithm faster;
  - defining variants of the basic backtracking algorithm, such as
    - *non-chronological backtracking* (during backtrack search, for each conflict backtrack to one of the causes of the conflict);
    - *conflict-driven backtracking* (during backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict).
  - defining more efficient decision heuristics for choosing the branching literals, such as
    - *Jeroslow-Wang* (selects literals that appear frequently in short clauses)
    - *DLIS: Dynamic Large Individual Sum* (selects the literal that appears most frequently in unresolved clauses)
    - *VSIDS: Variable State Independent Decaying Sum*, *Berkmin*, ...

Modern SAT solvers

- The majority of modern SAT solvers can be classified into two main categories:
  - SAT solvers based on the DPLL framework;
  - SAT solvers based on a *stochastic search*: the solver guesses a full assignment, and then, if the formula is evaluated to false under this assignment, starts to flip values of variables according to some (greedy) heuristic.
- DPLL-based SAT solvers, however, are considered better in most cases.
- DPLL-based SAT solvers also have the advantage that, unlike most stochastic search methods, they are complete (i.e., they always terminate with the correct answer).
- Modern SAT solvers can check formulas with hundreds of thousands variables and millions of clauses in a reasonable amount of time.

Modern SAT solvers

- In the last two decades, satisfiability procedures have undergone dramatic improvements in efficiency and expressiveness. Breakthrough systems like GRASP (1996), SATO (1997), Chaff (2001) and MiniSAT (2003) have introduced several enhancements to the efficiency of DPLL-based SAT solving.
- New SAT solvers are introduced every year.
  - The satisfiability library SATLIB[1] is an online resource that proposes, as a standard, a unified notation and a collection of benchmarks for performance evaluation and comparison of tools.
  - Such a uniform test-bed has been serving as a framework for regular tool competitions organised in the context of the regular SAT conferences.[2]

[1] http://www.satlib.org
[2] http://www.satcompetition.org

DIMACS CNF format

- DIMACS CNF format is a standard format for CNF used by most SAT solvers.
- Plain text file with following structure:

```
c <comments>
...
p cnf <num.of variables> <num.of clauses>
<clause> 0
<clause> 0
...
```

- Every number 1, 2, . . . corresponds to a variable (variable names have to be mapped to a variable).
- A negative number denote the negation of the corresponding variable.
- Every clause is a list of numbers, separated by spaces. (One or more lines per clause).

DIMACS CNF format

Example

$$A_1 \wedge (A_1 \vee P) \wedge (\neg A_1 \vee \neg P \vee A_2) \wedge (A_1 \vee \neg A_2)$$

- We have 3 variables and 4 clauses.
- CNF file:

```
p cnf 3 4
1 0
1 3 0
-1 -3 2 0
1 -2 0
```


Applications of SAT

- A large number of problems can be described in terms of satisfiability, including graph problems, planning, games, scheduling, software and hardware verification, extended static checking, optimization, test-case generation, among others.
- These problems can be encoded by propositional formulas and solved using SAT solvers.

problem $\mathcal{P}$ $\rightsquigarrow$ formula F $\longrightarrow$ $\boxed{\text{CNF converter}}$ $\longrightarrow$ $\boxed{\text{SAT solver}}$

SAT solver output: If F is satisfiable: **sat** + model
If F is unsatisfiable: **unsat** + proof

The satisfying assignments (models) of F are the solutions of $\mathcal{P}$.

- SAT solvers are core engines for other solvers (like SMT solvers).
- SAT solver may be integrated into theorem provers.

Modeling with PL

When can the meeting take place?

– Maria cannot meet on Wednesday
– Peter can only meet either on Monday, Wednesday or Thursday
– Anne cannot meet on Friday
– Mike cannot meet neither on Tuesday nor on Thursday

Encode into the following proposition:

$$\neg\text{Wed} \wedge (\text{Mon} \vee \text{Wed} \vee \text{Thu}) \wedge \neg\text{Fri} \wedge (\neg\text{Tue} \wedge \neg\text{Thu})$$

Modeling with PL

Graph coloring

Can one assign one of K colors to each of the vertices of graph $G = (V, E)$ such that adjacent vertices are assigned different colors?

- Create $|V| \times K$ variables: $x_{ij} = 1$ iff vertex i is assigned color j; 0 otherwise.
- For each edge (u, v), require different assigned colors to u and v:
  $$\text{for each } 1 \leq j \leq K, \quad (x_{uj} \to \neg x_{vj})$$
- Each vertex is assigned exactly one color.
  - *At least* one color to each vertex:
    $$\text{for each } 1 \leq i \leq |V|, \quad \bigvee_{j=1}^{K} x_{ij}$$
  - *At most* one color to each vertex:
    $$\text{for each } 1 \leq i \leq |V|, \quad \bigwedge_{a=1}^{K-1} (x_{ia} \to \bigwedge_{b=a+1}^{K} \neg x_{ib})$$

Modeling with PL

At least, at most, exactly one...

How to represent in CNF the following constraints

- At least one: $\sum_{j=1}^{N} x_j \geq 1$?
  Standard solution:
  $$\bigvee_{j=1}^{N} x_j$$
- At most one: $\sum_{j=1}^{N} x_j \leq 1$?
  Naive solution:
  $$\bigwedge_{a=1}^{N-1} \bigwedge_{b=a+1}^{N} (\neg x_a \vee \neg x_b)$$
  More compact solutions are possible.
- Exactly one: $\sum_{j=1}^{N} x_j = 1$?
  Standard solution: at least 1 and at most 1 constraints.

Modeling with PL

Placement of guests

We have three chairs in a row and we need to place Anne, Susan and Peter.

- Anne does not want to sit near Peter.
- Anne does not want to sit in the left chair.
- Susan does not want to sit to the right of Peter.

Can we satisfy these constrains?

- Denote: Anne $= 1$, Susan $= 2$, Peter $= 3$
- Introduce a propositional variable for each pair *(person, place)*
- $x_{ij} = 1$ iff person i is sited in place j; 0 otherwise

Modeling with PL

Placement of guests (cont.)

- Anne does not want to sit near Peter.
$$((x_{11} \vee x_{13}) \rightarrow \neg x_{32}) \wedge (x_{12} \rightarrow (\neg x_{31} \wedge \neg x_{33}))$$
- Anne does not want to sit in the left chair. $\neg x_{11}$
- Susan does not want to sit to the right of Peter.
$$(x_{31} \rightarrow \neg x_{22}) \wedge (x_{32} \rightarrow \neg x_{23})$$
- Each person is placed.
$$\bigwedge_{i=1}^{3} \bigvee_{j=1}^{3} x_{ij}$$
- No more than one person per chair.
$$\bigwedge_{i=1}^{3} \bigwedge_{a=1}^{2} \bigwedge_{b=a+1}^{3} (\neg x_{ia} \vee \neg x_{ib})$$

Modeling with PL

Equivalence checking of if-then-else chains

Original C code

```
if(!a && !b) h();
else if(!a) g();
else f();
```

Optimized C code

```
if(a) f();
else if(b) g();
else h();
```

Are these two programs equivalent?

1. Model the variables a and b and the procedures that are called using the Boolean variables a, b, f, g, and h.
2. Compile if-then-else chains into Boolean formulae
$$\text{compile}(\textbf{if } x \textbf{ then } y \textbf{ else } z) \equiv (x \wedge y) \vee (\neg x \wedge z)$$
3. Check the validity of the following formula
$$\text{compile}(\textit{original}) \leftrightarrow \text{compile}(\textit{optimized})$$
Reformulate it as a SAT problem: Is the Boolean formula
$$\neg\,(\text{compile}(\textit{original}) \leftrightarrow \text{compile}(\textit{optimized}))$$
satisfiable?

Proof system

- So far we have taken the *"semantic"* approach to logic, with the aim of characterising the semantic concept of model, from which validity, satisfiability and semantic entailment were derived.
- However, this is not the only possible point of view.
- Instead of adopting the view based on the notion of truth, we can think of logic as a codification of reasoning. This alternative approach to logic, called *"deductive"*, focuses directly on the deduction relation that is induced on formulas, i.e., on what formulas are logical consequences of other formulas.
- We will explore this perspective later in this course.

Exercises

- Encode into SAT a Sodoku puzzle.
  - 9×9 square divided into 9 sub-squares
  - General rules:
    - Values 1-9, one value per cell
    - No duplicates in rows
    - No duplicates in columns
    - No duplicates in sub-squares
  - A particular instance of the Sodoku puzzle has some known initial values.
- Use a SAT solver to show that the following two if-then-else expressions are equivalent.

```
!(a||b) ?  h :  !(a==b) ?  f :  g
!(!a||!b) ?  g :  (!a&&!b) ?  h :  f
```

Exercises

- Convert into an equivalent CNF the following formulas.
  - $A \vee (A \to B) \to A \vee \neg B$
  - $(A \to B \vee C) \wedge \neg(A \wedge \neg B \to C)$
  - $(\neg A \to \neg B) \to (\neg A \to B) \to A$
- Convert $P \wedge Q \vee (R \wedge P)$ into a equisatisfiable formula in CNF by using the Tseitin transformation.
- Run by hand the DPLL procedure to decide about the satisfiability of the formulas above.

Exercises

- Pick up a SAT solver.
- Play with simple examples.
- Use the SAT solver to test if each of the following formulas is satisfiable, valid, refutable or a contradition.
  - $A \vee (A \to B) \to A \vee \neg B$
  - $(A \to B \vee C) \wedge \neg(A \wedge \neg B \to C)$
  - $(\neg A \to \neg B) \to (\neg A \to B) \to A$

  Note that CNF equivalents of these formulas where already calculated.
- Search the web for “SAT benchmarks” and experiment.