

Automated Reasoning, 2IW15

prof dr Hans Zantema
HG room 6.73
tel 040 - 2472749
email: h.zantema@tue.nl

Information:

www.win.tue.nl/~hzantema/ar.html

Literature:

S. N. Burris: *Logic for Mathematics and Computer Science*

Prentice Hall, 1998, ISBN 0-13-285974-2

C. Meinel and T. Theobald: *Algorithms and Data Structures in VLSI Design*

Springer, 1998, ISBN 3-540-64486-5

Organization

- Course + examination
 - Practical assignment
 - Each 50 %, each has to be at least 5
 - Practical assignment to be done in groups of at most 2
 - Deadline for first part of practical assignment:
November 14, 2011
 - Deadline for second part of practical assignment:
January 9, 2012
 - Playing around with the tools `yices` and `bddsolve` may start now
-

Example: (free after Lewis Carroll)

1. Good-natured tenured professors are dynamic
 2. Grumpy student advisors play slot machines
 3. Smokers wearing a cap are phlegmatic
 4. Comical student advisors are professors
 5. Smoking untenured members are nervous
 6. Phlegmatic tenured members wearing caps are comical
 7. Student advisors who are not stock market players are scholars
 8. Relaxed student advisors are creative
 9. Creative scholars who do not play slot machines wear caps
 10. Nervous smokers play slot machines
 11. Student advisors who play slot machines do not smoke
 12. Creative good-natured stock market players wear caps
 13. Therefore no student advisor is smoking
-

Is it true that claim 13 can be concluded from statements 1 until 12?

We want to determine this fully automatically:

- Translate all statements and the desired conclusion to a formal description
- Apply some computer program with this formal description as input that decides fully automatically whether the conclusion is valid

This is a step further then verifying a given human reasoning as will be studied in the course *Proving with computer assistance*

The first step is giving names to every notion to be formalized

Next all claims (premisses and desired conclusion) should be transformed to formulas containing these names

5

name	meaning	opposite
<i>A</i>	good-natured	grumpy
<i>B</i>	tenured	
<i>C</i>	professor	
<i>D</i>	dynamic	phlegmatic
<i>E</i>	wearing a cap	
<i>F</i>	smoke	
<i>G</i>	comical	
<i>H</i>	relaxed	nervous
<i>I</i>	play stock market	
<i>J</i>	scholar	
<i>K</i>	creative	
<i>L</i>	plays slot machine	
<i>M</i>	student advisor	

Using these names all claims can be transformed to a **proposition** over the letters *A* to *M*, i.e., a expression composed from these letters and the boolean connectives \neg , \vee , \wedge , \rightarrow and \leftrightarrow

6

name	meaning	opposite
<i>A</i>	good-natured	grumpy
<i>B</i>	tenured	
<i>C</i>	professor	
<i>D</i>	dynamic	phlegmatic
	...	

The claim

Good-natured tenured professors are dynamic

yields:

$$(A \wedge B \wedge C) \rightarrow D$$

7

1. $(A \wedge B \wedge C) \rightarrow D$
2. $(\neg A \wedge M) \rightarrow L$
3. $(F \wedge E) \rightarrow \neg D$
4. $(G \wedge M) \rightarrow C$
5. $(F \wedge \neg B) \rightarrow \neg H$
6. $(\neg D \wedge B \wedge E) \rightarrow G$
7. $(\neg I \wedge M) \rightarrow J$
8. $(H \wedge M) \rightarrow K$
9. $(K \wedge J \wedge \neg L) \rightarrow E$
10. $(\neg H \wedge F) \rightarrow L$
11. $(L \wedge M) \rightarrow \neg F$
12. $(K \wedge A \wedge I) \rightarrow E$
13. Then $\neg(M \wedge F)$

8

Hence we want to prove automatically that

$$\begin{aligned}
 &(((A \wedge B \wedge C) \rightarrow D) \wedge \\
 &((\neg A \wedge M) \rightarrow L) \wedge \\
 &((F \wedge E) \rightarrow \neg D) \wedge \\
 &((G \wedge M) \rightarrow C) \wedge \\
 &((F \wedge \neg B) \rightarrow \neg H) \wedge \\
 &((\neg D \wedge B \wedge E) \rightarrow G) \wedge \\
 &((\neg I \wedge M) \rightarrow J) \wedge \\
 &((H \wedge M) \rightarrow K) \wedge \\
 &((K \wedge J \wedge \neg L) \rightarrow E) \wedge \\
 &((\neg H \wedge F) \rightarrow L) \wedge \\
 &((L \wedge M) \rightarrow \neg F) \wedge \\
 &((K \wedge A \wedge I) \rightarrow E)) \rightarrow \neg(M \wedge F)
 \end{aligned}$$

is a tautology, meaning that for every valuation of *A* to *M* the value of this formula is *true*

How can this be treated?

Simple method:

Truth tables, that is: compute the result for all 2^n valuations and check whether it is *true*

Here n is the number of variables

In the example we have $n = 13$, hence $2^n = 8192$, by which this approach is feasible

However, in many applications we have $n > 1000$ and need different methods

By putting \neg in front of the formula checking whether a formula yields *false* for all valuations is as difficult as checking whether a formula yields *true* for all valuations

A formula is called **satisfiable** if it does not yield *false* for all valuations

A formula composed from boolean variables and the boolean connectives $\neg, \vee, \wedge, \rightarrow$ and \leftrightarrow is called a **propositional formula**

The problem to determine whether a given propositional formula is satisfiable is called

SAT(isfiability)

So the method of truth tables is a method for SAT

However, the complexity of this method is always exponential in the number of variables, by which the method is unsuitable for formulas over many variables

Many practical problems can be expressed as SAT-problems over hundreds or thousands of variables

Hence we need other methods than truth tables

Example: verification of a microprocessor

Here we want that

$\neg(\text{specified behavior} \leftrightarrow \text{actual behavior})$

is not satisfiable

This can be expressed as a propositional formula, and proving that this formula is unsatisfiable implies that the microprocessor is correct with respect to its specification

We will see methods for SAT that may be used for huge formulas over many variables, like these

However, all known methods are worst case exponential

Now we give an example of a formula (the **pigeon hole formula**) that can be concluded to be unsatisfiable, hence is logically equivalent *false*, but for which it is hard to conclude this directly from the formula itself

Choose an integer number $n > 0$ and $n(n+1)$ boolean variables P_{ij} for $i = 1, \dots, n+1$ and $j = 1, \dots, n$

Define

$$C_n = \bigwedge_{i=1}^{n+1} \left(\bigvee_{j=1}^n P_{ij} \right)$$

$$R_n = \bigwedge_{j=1, \dots, n, 1 \leq i < k \leq n+1} (\neg P_{ij} \vee \neg P_{kj})$$

$$PF_n = C_n \wedge R_n$$

Here

$$\bigwedge_{i=1}^n A_i$$

is used as an abbreviation for

$$A_1 \wedge A_2 \wedge A_3 \wedge \dots \wedge A_n$$

and so on

For the report of your practical assignment the use of these notations is strongly recommended

14

Put the variables in a matrix as follows

$$\begin{array}{cccc} P_{11} & P_{21} & \cdots & P_{n+1,1} \\ P_{12} & P_{22} & \cdots & P_{n+1,2} \\ \vdots & \vdots & & \vdots \\ P_{1n} & P_{2n} & \cdots & P_{n+1,n} \end{array}$$

$$C_n = \bigwedge_{i=1}^{n+1} \left(\bigvee_{j=1}^n P_{ij} \right)$$

Validity of C_n means that in every column at least one variable is true

Hence if C_n holds then at least $n + 1$ variables are true

15

$$\begin{array}{cccc} P_{11} & P_{21} & \cdots & P_{n+1,1} \\ P_{12} & P_{22} & \cdots & P_{n+1,2} \\ \vdots & \vdots & & \vdots \\ P_{1n} & P_{2n} & \cdots & P_{n+1,n} \end{array}$$

$$R_n = \bigwedge_{j=1, \dots, n, 1 \leq i < k \leq n+1} (\neg P_{ij} \vee \neg P_{kj})$$

Validity of R_n means that in every row at most one variable is true:

for any two P 's in the row at least one is false

=

there are no two P 's in the row that are both true

Hence if R_n holds then at most n variables are true

Hence C_n and R_n cannot be valid both, hence $PF_n = C_n \wedge R_n$ is unsatisfiable

16

This counting argument is closely related to the **pigeon hole principle**:

if $n + 1$ pigeons fly out of a cage having n holes, then there is at least one hole through which at least two pigeons fly

The formula PF_n is called the **pigeon hole formula** for n

The formula is a conjunction of

$$(n + 1) + n \times \frac{n(n + 1)}{2}$$

disjunctions

The disjunctions are of the shape $\bigvee_{j=1}^n P_{ij}$ and $\neg P_{ij} \vee \neg P_{kj}$

If one arbitrary disjunction is removed from the big conjunction, then the resulting formula is always satisfiable

17

Summarizing pigeon hole formula:

PF_n is an artificial unsatisfiable formula of size polynomial in n

Minor modifications of PF_n are satisfiable

For most methods proving unsatisfiability of PF_n automatically is hard: it can be done, but for most methods the number of steps is exponential in n

PF_n and modifications are a good testcase for implementations of methods for SAT

Next we consider a different type of problem for which automated reasoning is desired too

18

Consider 12 processes over boolean variables A, \dots, J :

1. if D and E then $D, E := \text{false}, \text{false}$
2. if F and I then $F, I := \text{false}, \text{false}$
3. if A and E then $A, E := \text{false}, \text{false}$
4. if C and D then $C, D := \text{false}, \text{false}$
5. if D and G then $D, G := \text{false}, \text{false}$
6. if B and H then $B, H := \text{false}, \text{false}$
7. if B and I then $B, I := \text{false}, \text{false}$
8. if A and G then $A, G := \text{false}, \text{false}$
9. if D and H then $D, H := \text{false}, \text{false}$
10. if B and C then $B, C := \text{false}, \text{false}$
11. if B and J then $B, J := \text{false}, \text{false}$
12. if F and J then $F, J := \text{false}, \text{false}$

Claim:

From the initial state in which all variables have value *true* never the final state can be reached in which all variables have value *false*

19

It is possible to find an invariant proving this claim (try!)

However, in this course we want to treat this kind of questions fully automatically, without human cleverness of choosing an invariant

In this example having 1024 states it is feasible to compute all reachable states, but we want to do this for cases having 10^{10} or 10^{100} reachable states

More general:

- finite state space
- simple description of the set of initial states

- simple description of all possible state transitions
- some property has to be proved, for instance about all possible reachable states

20

Examples:

- Railways:
 - state transition: if a semaphore shows green then a train may enter the corresponding track
 - property to be proved: trains do not collide
- Telephone network:
 - state transitions:
 - * A enters a request for a connection with B
 - * a requested connection is made
 - * a connection is finished
 - property to be proved: no deadlock occurs
- CAN bus
- ...

21

The description of initial state and state transitions comprises the **model**, checking whether in such a model (e.g. in all reachable states) some property holds is called **model checking**

If state spaces and state transitions are described in a symbolic way (e.g. as formulas in propositional logic) then this is called **symbolic model checking**

Crucial is an efficient representation of formulas in propositional logic

This is a step further than SAT: we want efficient representations of all formulas, not only recognizing whether a formula is equivalent to *false*

Basic technique for SAT: resolution

Basic technique for symbolic model checking: B(inary) D(ecision) D(iagrams)

22

Not all kind of desired automated reasoning can be described in propositional logic

Other forms:

- Predicate logic: reasoning with \forall and \exists

all men are mortal

Socrates is a man

hence Socrates is mortal

- Equational logic: reasoning with equalities

given: $0 + x = x$ and

$(x + 1) + y = (x + y) + 1$

conclude:

$(0 + 1 + 1) + (0 + 1 + 1) = 0 + 1 + 1 + 1 + 1$

- Modal logic / temporal logic: reasoning involving a notion of time

every message sent will eventually be received

23

First we will concentrate on propositional logic

To be able to discuss the hardness of SAT we start by some remarks on **complexity** of algorithms

- (time) complexity: the number of steps required for executing an algorithm

- space complexity: the amount of memory required for executing an algorithm

Basic observation:

(time) complexity \geq space complexity

In order to abstract from details we consider orders of magnitude:

$f(n) = O(g(n))$:

f does not grow faster than g

$f(n) = \Omega(g(n))$:

f does not grow slower than g

24

More precisely:

$$f(n) = O(g(n))$$

means: there exist $c, n_0 \in \mathbf{N}$ such that

$$f(n) \leq c * g(n)$$

for every $n \geq n_0$

$$f(n) = \Omega(g(n))$$

means: there exist $n_0 \in \mathbf{N}, c > 0$ such that

$$f(n) \geq c * g(n)$$

for every $n \geq n_0$

25

Often $f(n)$ is the complexity of an algorithm depending on a number n , and g is a well-known function, like

- $g(n) = n$ (linear)
- $g(n) = n^2$ (quadratic)
- $g(n) = n^k$ for some k (polynomial)
- $g(n) = a^n$ for some $a > 1$ (exponential)

Roughly speaking:

- polynomial: still feasible for reasonably big values of n
- exponential: only feasible for very small values of n

26

No polynomial algorithm is known for SAT

More precisely: there is no algorithm

- receiving an arbitrary propositional formula as input
- that decides in all cases by execution whether this formula is satisfiable
- that requires no more than $c * n^k$ steps if $n > n_0$, where c, k, n_0 are values independent of the input and n is the size of the input

Even not if huge values are chosen for c, k, n_0

27

This can mean two different things:

- such an algorithm exists, but it has not yet been found
- existence of such an algorithm is impossible

Although it has not been proven, the latter is most likely

P is the class of decision problems admitting a polynomial algorithm

So we conjecture that SAT is not in **P**

SAT is in **NP**, i.e.,

there is a notion of **certificate** such that

- if the correct result for SAT is ‘no’, then no certificate exists

- if the correct result for SAT is ‘yes’, then a certificate exists, and there is a polynomial algorithm that can decide whether a candidate for a certificate is really a certificate

28

For SAT a certificate having these properties is a satisfying sequence of boolean values for the variables

NP is the abbreviation of non-deterministically polynomial

Basic observation: $\mathbf{P} \subseteq \mathbf{NP}$

Conjecture: $\mathbf{P} \neq \mathbf{NP}$

This is one of the main open problems in theoretical computer science

A decision problem \mathcal{A} in **NP** is called **NP-complete** if from the assumption $\mathcal{A} \in \mathbf{P}$ can be concluded that $\mathbf{P} = \mathbf{NP}$, i.e., every other decision problem in **NP** is in **P** too

So for NP-complete problems the existence of a polynomial algorithm is very unlikely

29

It has been proven that SAT is NP-complete (1970), in fact this was the starting point of NP-completeness results

Other NP-complete problems:

- Given a distance table between a set of places and a number n

Is there a path containing all of these places having a total length $\leq n$?

(closely related to Traveling Salesman)

- Given an undirected graph

Is there a cyclic path (hamiltonian circuit) containing every node exactly once?

- Given a number of packages, each having a weight
Can you divide these packages into two groups each having the same total weight?

- $c_0 = 0$ and $c_n = 0$, expressed as a formula:

$$\neg c_0 \wedge \neg c_n$$

30

Arithmetic in proposition logic

Binary representation

$$a_1 a_2 \cdots a_n$$

of a number a means that $a_i \in \{0, 1\}$ and

$$a = \sum_{i=1}^n a_i * 2^{n-i}$$

Addition

Given a and b , find d satisfying $a + b = d$

We need **carries** c_0, c_2, \dots, c_n

Example: $7 + 21 = 28$:

$$\begin{array}{rcccccc} c \rightarrow & & 0 & 0 & 1 & 1 & 1 & 0 \\ a = 7 \rightarrow & & 0 & 0 & 1 & 1 & 1 & \\ b = 21 \rightarrow & & 1 & 0 & 1 & 0 & 1 & \\ \hline d = 28 \rightarrow & & 1 & 1 & 1 & 0 & 0 & \end{array}$$

31

Requirements by which this is a correct computation:

- $d_i = a_i + b_i + c_i \pmod 2$, for $i = 1, \dots, n$, expressed as a formula:

$$a_i \leftrightarrow b_i \leftrightarrow c_i \leftrightarrow d_i$$

- $c_{i-1} = 1$ if and only if $a_i + b_i + c_i > 1$, for $i = 1, \dots, n$, expressed as a formula:

$$c_{i-1} \leftrightarrow ((a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i))$$

32

Let ϕ be the conjunction of all of these formulas

Then computation $d = a + b$ follows from the unique satisfying assignment for

$$\phi \wedge \bigwedge_{i=1}^n [\neg] a_i \wedge \bigwedge_{i=1}^n [\neg] b_i$$

In case d does not fit in n digits then c_0 would be forced to be 1, by which no satisfying assignment exists, to be solved by adding a leading 0 to a and b

Similarly subtraction: computing $b = d - a$ follows from satisfiability of

$$\phi \wedge \bigwedge_{i=1}^n [\neg] a_i \wedge \bigwedge_{i=1}^n [\neg] d_i$$

33

Multiplication

A fast way to do multiplication, essentially being the base school algorithm:

```
r := 0;
for i := 1 to n do
  if b_i then r := 2r + a else r := 2r
```

Invariant: $r = [b_1 \cdots b_i] * a$, so at the end $r = b * a$

For representing $\vec{b} = 2\vec{a}$ we introduce

$$\text{dup}(\vec{a}, \vec{b}) = \neg a_1 \wedge \neg b_n \wedge \bigwedge_{i=1}^{n-1} (a_{i+1} \leftrightarrow b_i)$$

34

Introduce extra boolean variables r_{ij}, s_{ij} for $i, j = 1, \dots, n$

Write $\vec{r}_i = (r_{i1}, \dots, r_{in}), \vec{s}_i = (s_{i1}, \dots, s_{in}),$

Using \vec{r}_{n+1} for the result, the requirement

$$\vec{a} * \vec{b} = \vec{r}_{n+1}$$

is described by the formula

$$\text{mul}(\vec{a}, \vec{b}, \vec{r}_{n+1}) =$$

$$\bigwedge_{j=1}^n \neg r_{1j} \wedge$$

$$\bigwedge_{i=1}^n [\text{dup}(\vec{r}_i, \vec{s}_i) \wedge (b_i \rightarrow \text{plus}(\vec{a}, \vec{s}_i, \vec{r}_{i+1}))$$

$$\wedge (\neg b_i \rightarrow \bigwedge_{j=1}^n (s_{ij} \leftrightarrow r_{i+1,j}))]$$

35

In this way we can do all kinds of arithmetic by SAT, for instance factorize a number

Define

$$\text{fac}(r) = \text{mul}(a, b, r) \wedge a > 1 \wedge b > 1$$

r is prime \iff $\text{fac}(r)$ is unsatisfiable

If satisfiable, then a, b represent factors

$\text{fac}(1234567891)$ is unsatisfiable, so 1234567891

is prime

Found by `minisat` or `yices` within 1 minute

$\text{fac}(1234567897)$ is satisfiable, yielding

$$1234567897 = 1241 \times 994817$$

found by `minisat` or `yices` within 1 second

36

Program correctness by SAT

Basic idea:

- express all integer variables by sequences of boolean variables, in binary notation

- for

for $j := 1$ to m do \dots

introduce $m+1$ copies a_0, \dots, a_m for every boolean variable a , where a_i means: the value of a after i steps

- $a := b$ in step i can be expressed as

$$(a_{i+1} \leftrightarrow b_i) \wedge \bigwedge_c (c_{i+1} \leftrightarrow c_i)$$

where c runs over all variables $\neq a$

37

Required property to be proved = specification of the program

Typically given by a **Hoare triple**:

$$\{P\}S\{Q\}$$

Here

- S is the program
- P is the **precondition**: the property assumed to hold initially
- Q is the **postcondition**: the property that should hold after the program has finished

For proving $\{P\}S\{Q\}$ add the formula

$$P_0 \wedge \neg Q_m$$

and prove that the resulting formula is unsatisfiable

38

Simple example: boolean array $a[1..m]$

CLAIM: After doing

for $j := 1$ to $m - 1$ do $a[j + 1] := a[j]$

we have

$$\underbrace{a[1] = a[m]}_{\text{postcondition}}$$

Precondition = true, may be ignored

a_{ij} represents value $a[i]$ after j iterations

Semantics of j th iteration:

$$\bigwedge_{i \in \{1, \dots, m\}, i \neq j+1} (a_{ij} \leftrightarrow a_{i,j-1}) \wedge (a_{j+1,j} \leftrightarrow a_{j,j-1})$$

Negation of postcondition:
 $\neg(a_{1,m-1} \leftrightarrow a_{m,m-1})$

Prove by a SAT solver that conjunction of all of these claims is unsatisfiable

39

Same approach applies for more complicated programs, where for $+$ and $*$ we can use $\text{plus}(\dots)$ and $\text{mul}(\dots)$

Example

CLAIM: After doing

$a := 0;$
 for $i := 1$ to m do $a := a + k$

we have $a = m * k$

For fixed m and number n of bits this is proved by proving unsatisfiability of

$$\bigwedge_{j=1}^n \neg a_{0,j} \wedge \bigwedge_{i=0}^{m-1} \text{plus}(\vec{a}_i, \vec{k}, \vec{a}_{i+1}) \wedge \neg \text{mul}(\vec{m}, \vec{k}, \vec{a}_m)$$

where \vec{m} is the binary encoding of number m

40

if b then S_1 else S_2

in step i can be expressed as

$$(b_i \rightarrow F_1) \wedge (\neg b_i \rightarrow F_2)$$

where formulas F_1, F_2 express S_1, S_2 in step i

In this way verification of a rich class of imperative programs can be expressed in SAT

In this way for integers we have to fix a number of bits, and encode all arithmetical operations ourselves

Later we will see **SMT: satisfiability modulo theories**, by which we can express (in)equalities on linear expressions like

$$a < 3 * b + c$$

directly

This is supported by the tool **yices**

41

Semantic tableaux

Idea:

- Start from a formula
- Build a tree step by step
- Every node contains a formula
- For every root symbol there is a rule decomposing a formula keeping the following **invariant**:

$$\bigvee_{\text{paths}} \left(\bigwedge_{A \text{ on path}} A \right)$$

is equivalent to the original formula

A path (from root to leaf) is **closed** if it contains both A and $\neg A$ for some formula A

If every path is closed then it follows from the invariant that the original formula is equivalent to *false*

42

Rules of the game

- To a path containing $A \wedge B$ you may add both A and B
 (correct since $(A \wedge B) \wedge A \wedge B \equiv A \wedge B$)

- To a path containing $A \vee B$ you may add branching: one branch containing A and the other containing B

(correct since $(A \wedge C) \vee (B \wedge C) \equiv C$ where $C = (A \vee B) \wedge \dots$)

- To a path containing $\neg\neg A$ you may add A

(correct since $\neg\neg A \wedge A \equiv A$)

- For $\neg(A \wedge B)$ you may apply the rule for $(\neg A) \vee (\neg B)$

(since $\neg(A \wedge B) \equiv \neg A \vee \neg B$)

- For $\neg(A \vee B)$ you may apply the rule for $(\neg A) \wedge (\neg B)$

(since $\neg(A \vee B) \equiv \neg A \wedge \neg B$)

43

- For $A \rightarrow B$ you may apply the rule for $(\neg A) \vee B$

(since $A \rightarrow B \equiv \neg A \vee B$)

- For $\neg(A \rightarrow B)$ you may apply the rule for $A \wedge \neg B$

(since $\neg(A \rightarrow B) \equiv A \wedge \neg B$)

- For $A \leftrightarrow B$ you may apply the rule for $(A \wedge B) \vee (\neg A \wedge \neg B)$

(since $A \leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$)

- For $\neg(A \leftrightarrow B)$ you may apply the rule for $(\neg A \wedge B) \vee (A \wedge \neg B)$

(since $\neg(A \leftrightarrow B) \equiv (\neg A \wedge B) \vee (A \wedge \neg B)$)

In this way for every formula composed from variables and $\neg, \vee, \wedge, \rightarrow$ and \leftrightarrow , and not of the shape p or $\neg p$, a rule is applicable

44

Such an expression p or $\neg p$ where p is a variable is called a **literal**

Extra rule:

Applying twice the same rule in the same path in the same formula is not allowed

Now the procedure will terminate since all newly generated formulas are smaller than the formula they replace

(give \leftrightarrow a higher weight than \wedge)

Hence the procedure will always end in one of the two cases:

45

- All paths are closed

In this case the formula is unsatisfiable due to the invariant

- There is a path that is not closed, and on all non-literal nodes the corresponding rule has been applied

In this case a satisfying assignment for the formula is found by making true all literals on such a non-closed path, by which the formula is satisfiable

Hence this methods of semantic tableaux is a complete method for SAT: applying it to an arbitrary formula always yields either

- a proof of unsatisfiability, or
- a satisfying assignment for the formula

after finitely many steps

46

Heuristics for building such a tree:

- Postpone branching rules as long as possible
- Give preference to steps yielding closed paths

Important:

This method only works for SAT, so if a formula has to be proven to be a tautology or two formulas have to be proven equivalent then first the problem has to be transformed to a satisfiability problem

Applying this method to the non-smoking student advisors yields a tree in which all 31 paths are closed

Unfortunately the method is very inefficient for many formulas

47

Resolution

This is another method for SAT, being the basis of the best current SAT-solvers

Resolution is only applicable to formulas of a particular shape, namely CNF

A **conjunctive normal form (CNF)** is a conjunction of clauses

A **clause** is a disjunction of literals

Hence a CNF is of the shape

$$\bigwedge_i (\bigvee_j \ell_{ij})$$

where ℓ_{ij} are literals

For example, the pigeon hole formula PF_n is a CNF

48

Arbitrary formulas can be transformed to CNFs in a clever way maintaining satisfiability

This makes resolution applicable to arbitrary formulas

Basic idea of resolution:

Add new clauses in such a way that the conjunction of all clauses remains equivalent to the original CNF

The empty clause \perp is equivalent to *false*

If the clause \perp is created in the resolution process then the conjunction of all clauses is equivalent to *false*, and hence the same holds for the original CNF

49

Intuitively:

Clauses are properties that you know to be true

From these clauses you derive new clauses, trying to derive a contradiction: the empty clause

Surprisingly here we need only one rule, the **resolution rule**

This rule states that if there are clauses of the shape $V \vee p$ and $W \vee \neg p$, then the new clause $V \vee W$ may be added

This is correct since

$$(V \vee p) \wedge (W \vee \neg p) \Rightarrow (V \vee W)$$

(apply case analysis p and $\neg p$)

50

Order of literals in a clause does not play a role

Double occurrences of literals will be removed

Think of a clause as a **set** of literals

Think of a CNF as a **set** of clauses

Example

We prove that

$$(p \vee q) \wedge (\neg r \vee s) \wedge (\neg q \vee r) \wedge (\neg r \vee \neg s) \wedge (\neg p \vee r)$$

is unsatisfiable

51

1	$p \vee q$	
2	$\neg r \vee s$	
3	$\neg q \vee r$	
4	$\neg r \vee \neg s$	
5	$\neg p \vee r$	
6	$p \vee r$	(1, 3, q)
7	r	(5, 6, p)
8	s	(2, 7, r)
9	$\neg r$	(4, 8, s)
10	\perp	(7, 9, r)

52

Remarks:

- Lot of freedom in choice
Other first steps in the example could have been (3, 4, r) or (2, 4, s) or (1, 5, p) or ...
- Resolution steps in which V contains q and W contains $\neg q$ for some q (or conversely) are allowed but useless
In that case the new clause $V \vee W$ is of the shape $q \vee \neg q \vee \dots$ and hence equivalent to *true*, not containing fruitful information
- If a clause consists of a single literal ℓ then by the resolution rule the literal $\neg \ell$ may be removed from every clause containing $\neg \ell$
This is called **unit resolution**

53

This proofs system is **sound** due to the correctness of the resolution rule:

if the empty clause can be derived then the original formula is equivalent to *false*

Conversely we will prove that this proof system is **complete**:

if any formula is equivalent to *false* then it is possible to derive the empty clause only by using the resolution rule

Hence it should be possible to solve the problem of the non-smoking student advisor in this way, which we will do now

54

Due to $(P \wedge Q) \rightarrow R \equiv \neg P \vee \neg Q \vee R$ this problem is easily transformed to CNF:

1. $\neg A \vee \neg B \vee \neg C \vee D$
2. $A \vee \neg M \vee L$
3. $\neg F \vee \neg E \vee \neg D$
4. $\neg G \vee \neg M \vee C$
5. $\neg F \vee B \vee \neg H$
6. $D \vee \neg B \vee \neg E \vee G$
7. $I \vee \neg M \vee J$
8. $\neg H \vee \neg M \vee K$
9. $\neg K \vee \neg J \vee L \vee E$
10. $H \vee \neg F \vee L$
11. $\neg L \vee \neg M \vee \neg F$
12. $\neg K \vee \neg A \vee \neg I \vee E$
13. M
14. F

55

Apply unit resolution on M, F : remove every $\neg M, \neg F$

1. $\neg A \vee \neg B \vee \neg C \vee D$
2. $A \vee (\neg M) \vee L$
3. $(\neg F) \vee \neg E \vee \neg D$
4. $\neg G \vee (\neg M) \vee C$
5. $(\neg F) \vee B \vee \neg H$
6. $D \vee \neg B \vee \neg E \vee G$
7. $I \vee (\neg M) \vee J$
8. $\neg H \vee (\neg M) \vee K$
9. $\neg K \vee \neg J \vee L \vee E$
10. $H \vee (\neg F) \vee L$
11. $\neg L \vee (\neg M) \vee (\neg F)$
12. $\neg K \vee \neg A \vee \neg I \vee E$
13. M
14. F

56

Apply unit resolution on $\neg L$: remove every L

1. $\neg A \vee \neg B \vee \neg C \vee D$
2. A
3. $\neg E \vee \neg D$
4. $\neg G \vee C$
5. $B \vee \neg H$
6. $D \vee \neg B \vee \neg E \vee G$
7. $I \vee J$
8. $\neg H \vee K$
9. $\neg K \vee \neg J \vee E$
10. H
11. $\neg K \vee \neg A \vee \neg I \vee E$

57

Apply unit resolution on A, H : remove every $\neg A, \neg H$

1. $\neg B \vee \neg C \vee D$
2. $\neg E \vee \neg D$
3. $\neg G \vee C$
4. B
5. $D \vee \neg B \vee \neg E \vee G$
6. $I \vee J$
7. K
8. $\neg K \vee \neg J \vee E$
9. $\neg K \vee \neg I \vee E$

58

Apply unit resolution on B, K : remove every $\neg B, \neg K$

1. $\neg C \vee D$
2. $\neg E \vee \neg D$
3. $\neg G \vee C$
4. $D \vee \neg E \vee G$
5. $I \vee J$
6. $\neg J \vee E$
7. $\neg I \vee E$

No unit resolution possible any more

Resolution on $I, 5$ and 7 yields $J \vee E$

Resolution on $J, 6$ and $J \vee E$ yields unit clause E

59

Apply unit resolution on E : remove every $\neg E$

1. $\neg C \vee D$
2. $\neg D$
3. $\neg G \vee C$
4. $D \vee G$

Unit resolution on $\neg D$ yields $\neg C$ and G , and remaining clause

$$\neg G \vee C$$

Unit resolution on $\neg C$ and G yields empty clause, hence we have proved that the formula is unsatisfiable

60

Preferring unit resolution is a good strategy: unit resolution can not cause increase of CNF size

Another good strategy is **subsumption**: ignore or remove clauses V for which a smaller clause W occurs satisfying $W \subset V$

For the rest there is a lot of remaining choice for doing resolution

61

A strategy that can always be applied is

Davis-Putnam's procedure (1960):

Repeat until either no clauses are left or the empty clause has been derived:

Choose a variable p

Apply resolution on every pair of clauses for which the one contains p and the other contains $\neg p$

Remove all clauses containing both q and $\neg q$ for some q

Remove all clauses containing either p or $\neg p$

62

Example:

Consider the CNF consisting of the following nine clauses

$$\begin{array}{lll} \neg p \vee \neg s & p \vee r & \neg s \vee t \\ \neg p \vee \neg r & s \vee p & q \vee s \\ \neg q \vee \neg t & r \vee t & \neg r \vee q \end{array}$$

First do all resolution steps w.r.t. p

After removing all clauses containing p , $\neg p$ or the shape $A \vee \neg A$ the following clauses remain:

$$r \vee \neg s, \underbrace{\neg r \vee s}_{\text{new}}, \neg s \vee t, q \vee s, \neg q \vee \neg t, r \vee t, \neg r \vee q$$

Next do all resolution steps w.r.t. q

After removing all clauses containing q , $\neg q$ the following clauses remain:

$$r \vee \neg s, \neg r \vee s, \neg s \vee t, r \vee t, \underbrace{s \vee \neg t, \neg r \vee \neg t}_{\text{new}}$$

63

Next do all resolution steps w.r.t. r

After removing all clauses containing r , $\neg r$ or the shape $A \vee \neg A$ the following clauses remain:

$$\underbrace{\neg t \vee \neg s, s \vee t}_{\text{new}}, \neg s \vee t, s \vee \neg t$$

Next do all resolution steps w.r.t. s

After removing all clauses containing s , $\neg s$ or the shape $A \vee \neg A$ the following clauses remain:

$$t, \neg t$$

Finally we do resolution on t and obtain the empty clause, proving that the original CNF is unsatisfiable

Theorem:

Davis-Putnam's procedure ends in an empty clause if and only if the original CNF is unsatisfiable

A direct consequence of this theorem is completeness of resolution

Now we will prove the theorem

Soundness of resolution we observed before; we have to prove that

if the CNF is unsatisfiable then
Davis-Putnam's procedure will end
in an empty clause

We assume that Davis-Putnam's procedure does not end in an empty clause; we have to prove that the original CNF is satisfiable

Due to the assumption and the structure of the procedure

Repeat until either no clauses are left or the empty clause has been derived ...

the process ends in the empty set of clauses which is trivially satisfiable

The required satisfiability of the original CNF follows from the following property:

If a CNF X is transformed to X' by a Davis-Putnam step and X' is satisfiable, then X is satisfiable too

To prove: If a CNF X is transformed to X' by a Davis-Putnam step and X' is satisfiable, then X is satisfiable too

Let U be the set of clauses in X in which p does not occur

Let V be the set of clauses C such that $C \vee p$ occurs in X

Let W be the set of clauses C such that $C \vee \neg p$ occurs in X

Then

$$X : \bigwedge_{C \in U} C \wedge \bigwedge_{C \in V} (C \vee p) \wedge \bigwedge_{C \in W} (C \vee \neg p)$$

and

$$X' : \bigwedge_{C \in U} C \wedge \bigwedge_{C \in V, D \in W} (C \vee D)$$

up to clauses containing the pattern $q \vee \neg q$

Assume that X' is satisfiable

Consider its part

$$\bigwedge_{C \in V, D \in W} (C \vee D)$$

\equiv

$$\bigwedge_{C \in V} (\bigwedge_{D \in W} (C \vee D))$$

\equiv (distributivity)

$$\bigwedge_{C \in V} (C \vee \bigwedge_{D \in W} D)$$

\equiv (distributivity)

$$(\bigwedge_{C \in V} C) \vee (\bigwedge_{D \in W} D)$$

Hence

$$X' \equiv \bigwedge_{C \in U} C \wedge ((\bigwedge_{C \in V} C) \vee (\bigwedge_{D \in W} D))$$

$$X' \equiv \bigwedge_{C \in U} C \wedge ((\bigwedge_{C \in V} C) \vee (\bigwedge_{D \in W} D))$$

We assume that this is satisfiable

In the corresponding satisfying assignment either $\bigwedge_{C \in V} C$ or $\bigwedge_{C \in W} C$ is true

If $\bigwedge_{C \in V} C$ is true, then we assign the value *false* to the fresh variable p and keep the same assignment for the other variables, by which

$$\bigwedge_{C \in V} (C \vee p) \wedge \bigwedge_{C \in W} (C \vee \neg p)$$

has the value *true*

69

Either $\bigwedge_{C \in V} C$ or $\bigwedge_{C \in W} C$ is true

If $\bigwedge_{C \in W} C$ is true, then we assign the value *true* to the variable p and again keep the same assignment for the other variables, by which

$$\bigwedge_{C \in V} (C \vee p) \wedge \bigwedge_{C \in W} (C \vee \neg p)$$

has the value *true*

For both cases we have a satisfying assignment for

$$X : \bigwedge_{C \in U} C \wedge \bigwedge_{C \in V} (C \vee p) \wedge \bigwedge_{C \in W} (C \vee \neg p)$$

End of proof

70

Summarizing Davis-Putnam's procedure:

- Procedure to establish satisfiability of any CNF
- Complete: it always ends and always gives the right answer
- One long repeat loop doing at most n steps if there are n variables
- In every step of the loop clauses are added and removed
- Worst case exponential: intermediate CNF may blow up exponentially

- By keeping intermediate CNFs in case of satisfiability a satisfying assignment can be constructed from the run of the procedure, as we show by an example

71

$$p \vee q, \neg p \vee \neg q, p \vee \neg q, \neg p \vee r$$

$\downarrow p$

$$q \vee r, \neg q, \neg q \vee r$$

$\downarrow q$

r

$r = \text{true}$

$\downarrow r$

$\{\}$

find value for the last variable that was removed

72

$$p \vee q, \neg p \vee \neg q, p \vee \neg q, \neg p \vee r$$

$\downarrow p$

$$q \vee r, \neg q, \neg q \vee r$$

$q = \text{false}$

$\downarrow q$

\uparrow

r

$r = \text{true}$

$\downarrow r$

$\{\}$

evaluate in formula, find next value

73

$p \vee q, \neg p \vee \neg q, p \vee \neg q, \neg p \vee r$	$p = true$
$\downarrow p$	\uparrow
$q \vee r, \neg q, \neg q \vee r$	$q = false$
$\downarrow q$	\uparrow
r	$r = true$
$\downarrow r$	
$\{\}$	

evaluate all values in formula, find next value, until finished

74

The DPLL method

(Davis, Putnam, Logemann, Loveland, 1962)

Recursive procedure for SAT on CNF = set of clauses

DPLL(X):

```

 $X := \text{unit-resol}(X)$ 
if  $X = \emptyset$  then return(satisfiable)
if  $\perp \notin X$  then
  choose variable  $p$  in  $X$ 
  DPLL( $X \cup \{p\}$ )
  DPLL( $X \cup \{\neg p\}$ )

```

unit-resol means: apply unit resolution as long as possible, more precisely:

As long as a clause occurs consisting of one literal ℓ , remove all clauses containing ℓ and remove $\neg\ell$ from all clauses containing $\neg\ell$

75

Execution of DPLL(X) always terminates

(in every recursive call the number of occurring variables is strictly less)

Idea of DPLL:

- First try unit resolution as long as possible
- If you can not proceed by unit resolution or trivial observations then choose a variable p , introduce the cases p and $\neg p$, and for both cases go on recursively
- If all cases in this process end in a contradiction (the empty clause) then the original formula is unsatisfiable
- If one case is found arriving in the empty set of clauses, then a satisfying assignment for the original formula is found by making the consecutive choices for the variables for which case analysis has been done

76

Example:

Consider the CNF consisting of the following nine clauses

$\neg p \vee \neg s$	$p \vee r$	$\neg s \vee t$
$\neg p \vee \neg r$	$s \vee p$	$q \vee s$
$\neg q \vee \neg t$	$r \vee t$	$\neg r \vee q$

No unit resolution possible: choose variable p

Add p	Add $\neg p$
Unit resolution:	Unit resolution:
$\neg s$	r
$\neg r$	s
q (use $\neg s$)	q (use r)
t (use $\neg r$)	t (use s)
$\neg t$ (use q)	$\neg t$ (use q)
\perp	\perp

Both branches yield \perp , so original CNF is unsatisfiable

77

Example:

Consider the CNF consisting of the following eight clauses

$$\begin{array}{lll} \neg p \vee \neg s & p \vee r & \neg s \vee t \\ \neg p \vee \neg r & s \vee p & q \vee s \\ \neg q \vee \neg t & r \vee t & \end{array}$$

No unit resolution possible: choose variable p

Add p

Unit resolution:

$\neg s$

$\neg r$

q (use $\neg s$)

t (use $\neg r$)

$\neg t$ (use q)

\perp

Yields satisfying assignment $p = q = \text{false}$,
 $r = s = t = \text{true}$

78

In this way DPLL is a complete method for SAT, just like classical Davis-Putnam (DP)

Just like DP, DPLL admits freedom of choice, and this choice may strongly influence the efficiency of the algorithm

Usually DPLL is more efficient than DP

Lot of research is done on heuristics for good choices of p in DPLL

Although DPLL is not pure resolution due to the addition of p and $\neg p$, there is a strong relationship between a DPLL proof and resolution:

A proof of unsatisfiability of a CNF by DPLL can be transformed directly to a resolution proof of the same CNF ending in \perp , having the same size

79

The key idea is that any resolution derivation from $V \wedge \ell$ to \perp can be transformed to a resolution derivation from V to $\neg \ell$ (or \perp), simply by ignoring unit resolution steps on ℓ

Two derivations from $V \wedge p$ to \perp and from $V \wedge \neg p$ to \perp then transform to derivations from V to both p and $\neg p$, yielding \perp in one extra step

Example

Applying unit resolution to the non-smoking student advisors yields

1. $\neg C \vee D$
2. $\neg E \vee \neg D$
3. $\neg G \vee C$
4. $D \vee \neg E \vee G$
5. $I \vee J$
6. $\neg J \vee E$
7. $\neg I \vee E$

80

Let us choose p to be E

By adding the clause E unit resolution yields

2. $\neg D$
4. $D \vee G$
8. $\neg C$ (1, 2)
9. $\neg G$ (3, 8)
10. D (4, 9)
11. \perp (2, 10)

By adding the clause $\neg E$ unit resolution yields

6. $\neg J$
7. $\neg I$
12. I (5, 6)
13. \perp (7, 12)

81

These two derivations combine to

8	$\neg C \vee \neg E$	(1, 2)
9	$\neg G \vee \neg E$	(3, 8)
10	$D \vee \neg E$	(4, 9)
11	$\neg E$	(2, 10)
12	$E \vee I$	(5, 6)
13	E	(7, 12)
14	\perp	(11, 13)

The same approach applies for more complicated nested examples

Hence:

DPLL indeed may be (and is) considered as a resolution technique

82

Weak point of this version of DPLL:

The (typically very large) CNF is modified by unit resolution in every recursive call

More efficient: keep the same original CNF everywhere, and only remember the list M of literals = unit clauses that are chosen or derived

An original clause C yields a contradiction after a number of unit resolution steps in the DPLL process if and only if it only consists of negations of literals occurring in this list

Notation: $M \models \neg C$

Remember: this means that C conflicts with M

83

Now we will reformulate the DPLL process building and modifying a list M of literals and checking for $M \models \neg C$, rather than doing recursion and unit resolution

In this way we mimic the original DPLL program = traversal through the DPLL tree

At every moment the CNF in the original DPLL program corresponds to the combination of

- the literals in M , and
- the original CNF from which all negations of literals from M have been stripped away

84

During the process M is a list of literals, where every literal in M may or may not be marked to be a **decision** literal, notation ℓ^d

Idea:

these decision literals originate from a choice in the DPLL algorithm, the other literals in M are derived by unit resolution = unit propagation, or are the negation of a literal that was a decision literal before

Why?

For mimicking backtracking it is essential to recognize these particular decision literals:

backtracking = go back to last chosen decision literal and continue with its negation

85

Starting by M being empty, there are four rules that together mimic the DPLL process:

- **UnitPropagate:** mimicks the generation of a new unit clause
- **Decide:** mimicks the choice p in the DPLL process, only if no unitpropagate is possible
- **Backtrack:** mimicks backtracking to the negation of the last decision in case a branch is unsatisfiable
- **Fail:** mimicks the end of the DPLL process if every branch, and hence the CNF, is unsatisfiable

We say that ℓ is **undefined** in M if neither ℓ nor $\neg\ell$ occurs in M

86

The four rules

UnitPropagate:

$$M \implies M\ell$$

if ℓ is undefined in M and the CNF contains a clause $C \vee \ell$ satisfying $M \models \neg C$

Decide:

$$M \implies M\ell^d$$

if ℓ is undefined in M

Backtrack:

$$M\ell^d N \implies M\neg\ell$$

if $M\ell^d N \models \neg C$ for a clause C in the CNF and N contains no decision literals

Fail:

$$M \implies \text{fail}$$

if $M \models \neg C$ for a clause C in the CNF and M contains no decision literals

87

Observations:

Start with M being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from M) always ends in either

- fail, proving that the CNF is unsatisfiable since the derivation of Fail can be interpreted as a case analysis yielding a contradiction in all cases, or
- a list M yielding a satisfying assignment

In case **UnitPropagate** always gets priority, and **Decide** is only used for ℓ being positive, then such derivations mimick original DPLL computations

However, this derivational framework is much more suitable for describing optimizations as they are used in modern powerful SAT solvers (SATzilla, Picosat, Rsat, Minisat, March, Yices), and we will present

88

Example

Let the CNF consist of the four clauses

1. $p \vee q$
2. $p \vee \neg q$
3. $\neg p \vee r$
4. $\neg p \vee \neg r$

We get the following derivation proving unsatisfiability:

$$\begin{array}{ll} \emptyset & \implies \text{Decide} \\ p^d & \implies \text{UnitPropagate, clause 3} \\ p^d r & \implies \text{Backtrack, clause 4} \\ \neg p & \implies \text{UnitPropagate, clause 1} \\ \neg p q & \implies \text{Fail, clause 2} \\ \text{fail} & \end{array}$$

89

Optimization: Backjump

Example:

Let the CNF consist of the four clauses

1. $\neg p \vee q$
2. $\neg r \vee s$
3. $\neg t \vee \neg u$
4. $\neg q \vee \neg t \vee u$

$$\begin{array}{ll} \emptyset & \implies \text{Decide} \\ p^d & \implies \text{UnitPropagate, clause 1} \\ p^d q & \implies \text{Decide} \\ p^d q r^d & \implies \text{UnitPropagate, clause 2} \\ p^d q r^d s & \implies \text{Decide} \\ p^d q r^d s t^d & \implies \text{UnitPropagate, clause 4} \\ p^d q r^d s t^d u & \implies \text{Backtrack, clause 3} \text{ ???} \\ p^d q r^d s \neg t & \implies \dots \end{array}$$

For the found contradiction between t^d , u , and clause 3, the decision r^d and the derivation of s does not play a role

If instead of r^d the decision t^d was made directly, we would have had a better backtrack step to $p^d q \neg t$

However, in large CNFs it is hard to know in advance what will be such a good choice

So we keep the sequence of decisions as it is, but allow this step to $p^d q \neg t$

Since it does not negate the last decision as in Backtrack, but negates a decision of several steps back, this is called **Backjump**

In order to use this idea in general at every UnitPropagate step we store the corresponding clause and at every contradiction found we investigate which generated literals and corresponding clauses played a role

A clause conflicting these relevant literals can be obtained by resolution from the original CNF: the **backjump** clause

In our example u was derived using clause 4, and a contradiction was found using clause 3, yielding by resolution the backjump clause $\neg q \vee \neg t$

This clause is of the shape $C' \vee \ell'$, where C' conflicts with the list of literals and ℓ' typically is the negation of the last decision literal

More precisely, the new rule is

Backjump:

$$M\ell^d N \implies M\ell'$$

if $M\ell^d N \models \neg C$ for a clause C in the CNF and there is a clause $C' \vee \ell'$ derivable from the CNF such that $M \models \neg C'$ and ℓ' is undefined in M

Note that this general formulation is a generalization of Backtrack; the improvement is obtained when M is as small as possible

Correctness of this backjump rule is by construction

Implementation not clear by general formulation: how to find the backjump clause $C' \vee \ell'$ derivable from the CNF?

In implementation the choice for this backjump clause is guided by the decisions and unit propagation steps leading to the contradiction causing the backtrack step

More optimizations: **Learn** and **Forget**

The idea is to modify the CNF during the process of SAT solving:

- **Learn:** new clauses that follow from the original CNF may be added
- **Forget:** clauses that follow from the other clauses may be removed

In modern SAT solvers all backjump clauses that are learned are directly added to the CNF: they may be helpful later on

Subsumption: if the formula contains two clauses $C \subseteq C'$, then one may forget = remove C'

For all variants the main property remains:

Start with M being empty and apply the rules as long as possible always ends in either

- fail, proving that the CNF is unsatisfiable, or
- a list M yielding a satisfying assignment

Choices made in this process (e.g., which literal to choose for Decide) only influence efficiency of obtaining the result, **not** the validity of the result

95

Sometimes a **Restart** $M \implies \emptyset$ after learning some clauses is fruitful

Looks counterintuitive, but sometimes it may yield optimal profit of the clauses that have been learned

Good heuristics for choosing literals for Decide are crucial

Adding backjump clauses and then restart may cause better choice for decision literals

A basic heuristic takes a literal that occurs most often in the relevant clauses

Lively area of research: every one/two year(s) there is a SAT competition, and at every competition strong improvements show up

```
p cnf 3 5
1 2 3 0
1 -3 0
-1 2 0
-1 -3 0
2 -3 0
```

yields

```
sat
-1 2 -3
```

96

Standard format for SAT competition: **dimacs**

Boolean variables are numbered from 1 to n

For k clauses the file has to start by

```
p cnf n k
```

followed by k lines each containing a clause

- variable i is denoted by i
- the negation of variable i is denoted by $-i$
- these literals are separated by spaces
- every clause is ended by 0

98

Until now we only considered CNFs, and we are not yet able to apply resolution to arbitrary propositions

We want to be able to apply resolution for determination of satisfiability of arbitrary propositions by first transforming the proposition to CNF

Straightforward approach:

Transform the proposition to a logically equivalent CNF

This is always possible:

every 0 in the truth table yields a clause

proposition \equiv conjunction of these clauses

Often it can be done more efficient

Unfortunately:

It often happens that every CNF logically equivalent to a given proposition is unacceptably big

97

Example:

Calling

```
yices -e -d test.d
```

on the file `test.d` containing

99

Example:

$$A : (\dots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \dots \leftrightarrow p_n)$$

This formula yields true if and only if an even number of p_i 's has the value *false*

Claim:

Let B be a CNF satisfying $A \equiv B$
 Then every clause C in B contains exactly n literals

Proof:

Assume not, then some p_i does not occur in a clause C of B

Then you can give values to the remaining variables such that C is not true, hence neither B , independent of the value for p_i

Then you can give a value to p_i such that A yields *true*

Contradiction to $A \equiv B$ (end of proof of claim)

100

The truth table of A contains exactly 2^{n-1} zeroes: half of all entries

Every clause containing n literals yields exactly one zero in the truth table

According to the claim all clauses of B are of this shape

Hence B consists of 2^{n-1} clauses

Conclusion:

Every CNF B equivalent to A has size exponential in the size of A

\implies unacceptably large

101

Hence we are looking for a way to transform any arbitrary propositional formula A to a CNF B such that

- A is satisfiable if and only if B is satisfiable
- the size of B is linear (or at least polynomial) in the size of A

Note that we weaken the restriction that A and B are equivalent

Such a construction is possible if we allow that B contains a number of fresh variables

102

For every formula D on at most 3 variables there is a CNF $cnf(D)$ such that $cnf(D) \equiv D$ and $cnf(D)$ contains at most 4 clauses:

$$cnf(p \leftrightarrow \neg q) = (p \vee q) \wedge (\neg p \vee \neg q)$$

$$cnf(p \leftrightarrow (q \wedge r)) = (p \vee \neg q \vee \neg r) \wedge (\neg p \vee q) \wedge (\neg p \vee r)$$

$$cnf(p \leftrightarrow (q \vee r)) = (\neg p \vee q \vee r) \wedge (p \vee \neg q) \wedge (p \vee \neg r)$$

$$cnf(p \leftrightarrow (q \leftrightarrow r)) = (p \vee q \vee r) \wedge (p \vee \neg q \vee \neg r) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee r)$$

103

Introduce a new variable for every non-literal subformula of A (including A itself), the **name** of the subformula

For a subformula D of A we define

- $n_D = D$ if D is a literal
- $n_D =$ the name of D , otherwise

The CNF $T(A)$, the **Tseitin transformation** of A , is defined to be the CNF consisting of the clauses:

- n_A
- the clauses of $cnf(q \leftrightarrow \neg n_D)$ for every non-literal subformula of the shape $\neg D$ having name q

- the clauses of $\text{cnf}(q \leftrightarrow (n_D \diamond n_E))$ for every subformula of the shape $D \diamond E$ having name q , for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

104

Example: $A : \underbrace{(\neg s \wedge p)}_B \leftrightarrow \underbrace{((q \rightarrow r) \vee \neg p)}_C$

yields $T(A)$ consisting of the clauses A and

$$\left. \begin{array}{l} A \vee B \vee C \\ A \vee \neg B \vee \neg C \\ \neg A \vee \neg B \vee C \\ \neg A \vee B \vee \neg C \end{array} \right\} \text{cnf}(A \leftrightarrow (B \leftrightarrow C))$$

$$\left. \begin{array}{l} B \vee s \vee \neg p \\ \neg B \vee \neg s \\ \neg B \vee p \end{array} \right\} \text{cnf}(B \leftrightarrow (\neg s \wedge p))$$

$$\left. \begin{array}{l} \neg C \vee D \vee \neg p \\ C \vee \neg D \\ C \vee p \end{array} \right\} \text{cnf}(C \leftrightarrow (D \vee \neg p))$$

$$\left. \begin{array}{l} \neg D \vee r \vee \neg q \\ D \vee \neg r \\ D \vee q \end{array} \right\} \text{cnf}(D \leftrightarrow (q \rightarrow r))$$

105

Theorem:

For every propositional formula A we have:

A is satisfiable if and only if $T(A)$ is satisfiable

Proof sketch:

- a satisfying assignment for $T(A)$ restricting to the variables from A yields a satisfying assignment for A
- a satisfying assignment for A is extended to a satisfying assignment for $T(A)$ by giving n_D the value of D obtained from the satisfying assignment for A

106

Summary of Tseitin transformation:

For every propositional formula A we have:

- A is satisfiable if and only if $T(A)$ is satisfiable
- $T(A)$ contains two types of variables: variables occurring in A and variables representing names of subformulas of A
- The size of $T(A)$ is linear in the size of A
- Every clause in $T(A)$ contains at most 3 literals: $T(A)$ is a **3-CNF**
- A fruitful approach to investigate satisfiability of A is applying a modern CNF based SAT solver on $T(A)$

107

There is no need to use a separate tool for transforming an arbitrary propositional formula A to $T(A)$ is dimacs format:

several modern SAT solvers like **yices** also accept SMT format (satisfiability modulo theories) of which the most basic instance coincides with arbitrary propositional formulas

Internally then first the Tseitin transformation is applied, and then the same approach is followed as by entering dimacs format

Call

```
yices -e -smt test.smt
```

where `test.smt` contains the formula

108

Example: (and and or have any number of arguments)

```
(benchmark test.smt
:extrapreds ((A) (B) (C) (D))
:formula (and
(iff A (and D B))
(implies C B))
```

```
(not (or A B (not D)))
(or (and (not A) C) D)
))
```

yields

```
sat
(= A false)
(= B false)
(= D true)
(= C false)
```

109

Conclusions on resolution for proposition logic:

- Technique to establish satisfiability of CNF
- Tseitin transformation efficiently transforms every propositional formula to 3-CNF, maintaining satisfiability
In this way resolution is applicable for every propositional formula
- Both DP and DPLL are complete methods for SAT based on resolution, both having freedom of choice
Often DPLL is more efficient than DP

110

Conclusions (continued)

- Modern CNF based SAT solvers, like
 - SATzilla
 - Picosat
 - Rsat
 - Minisat
 - Yices

essentially use DPLL, extended by Back-jump, Learn, Forget and Restart

- Resolution hardly recognizable in actual algorithms
- Extremely powerful approach for a wide range of problems that seem unrelated to SAT solving

111

Amazing example:

Prove termination of the system of three rules

$$aa \rightarrow bc, \quad bb \rightarrow ac, \quad cc \rightarrow ab$$

Solution: interpret a, b, c by matrices over natural numbers in such a way that by doing rewrite steps a particular entry in matrices always strictly decreases

Since this entry is a natural number, this cannot go on forever, proving termination

Restricting to binary encoded numbers of fixed size, and fixing matrix dimension (both ≈ 4) this was encoded in a SAT problem, and the obtained satisfying assignment was transformed to a formal proof of the above shape

Until now all known proofs of this problem are variants of this idea, all found by SAT solving

No human intuition available

112

Extension of SAT solving

We have seen how several problems involving e.g. arithmetic or program correctness can be encoded as a SAT problem

Typically, a program is written in which an instance of a problem is entered, and a corresponding SAT problem is produced, after which a plain SAT solver is applied to solve the problem

Apart from SAT solving there are several other formats of **constraint problems** where any solution or an optimal solution has to be found

For instance: **linear optimization** given n real valued variables x_1, \dots, x_n , find the highest (or lowest) value of a linear combination $\sum_{i=1}^n a_i x_i$ satisfying a given number of constraints all of the shape $\sum_{i=1}^n b_i x_i \leq c$

If the variables are integer valued, this is called **integer optimization**

113

For these problems **linear optimization** and **integer optimization** extremely powerful techniques are available, unrelated to SAT solving

In particular these techniques can be used to establish whether a conjunction of inequalities has a solution, and if so, find one, rather than finding an optimal solution

An important technique is the **Simplex method**, in which sets of inequalities are reduced by **Gauss elimination**

Here we will not present these techniques, we only consider how they can be joined with techniques for SAT solving, to solve propositional formulas over such inequalities:

Satisfiability Modulo Theories

114

Satisfiability Modulo Theories (SMT)

Example:

Find positive integer values a, b, c, d such that

$2a > b + c, 2b > c + d, 2c > 3d$ and $3d > a + c$

Approach 1:

Choose n boolean variables for each of the numbers $a, b, c, d, 2a, b+c, 2b, c+d, 2c, 2d, 3d, a+c$ representing their binary encodings, and express the constraints with '+' and '>' in the standard way for expressing binary arithmetic, using several extra boolean variables for carries

Then apply a SAT solver on the resulting formula

The formula will be satisfiable; transform the satisfying assignment to the desired numbers a, b, c, d

Depends on n ; $n = 7$ gives a solution

115

Approach 2:

Extend the SAT solver in such a way that it can deal with the inequalities directly, rather than only on boolean variables

In the mechanism with the derivation rules, the central access to the formula is checking whether

$$M \models \neg C$$

for both M being a list of literals and C being a clause

That is, we have to check whether for every literal ℓ in C , the conjunction of ℓ and all literals in M gives rise to a contradiction

For basic SAT this means that $\neg \ell$ occurs in M

The machinery is also correct if we have another mechanism to check whether a conjunction of literals is contradictory

For instance, $x > y + 1 \wedge y > z \wedge z > x + 2$ is contradictory

116

So SAT solvers can be extended to deal with establishing satisfiability of CNFs defined by

- A CNF is a conjunction of clauses
- A clause is a disjunction of literals
- A literal is a basic formula in some theory or its negation

Here for the theory there should be an efficient implementation to check whether the conjunction of a given set of literals is satisfiable or not

For efficiency this implementation has to be incremental

For linear inequalities over reals or integers there are such implementations

117

By applying the Tseitin transformation this approach works for every propositional formula over basic formulas in such a theory

In an SMT solver like **yices** one can directly enter and solve

```
(benchmark test.smt
:extrafuns ((A Int) (B Int) (C Int)
(D Int))
:formula (and
(> (* 2 A) (+ B C))
(> (* 2 B) (+ C D))
(> (* 2 C) (* 3 D))
(> (* 3 D) (+ A C))
))
```

118

More precisely, if this formula is in file `test.smt`, then calling

```
yices -e -smt test.smt
```

yields the output

```
sat
(= A 30)
(= B 27)
(= C 32)
(= D 21)
```

119

Note that we do not need to specify an upperbound on the numbers:

```
(benchmark test.smt
:extrafuns ((A Int) (B Int) (C Int))
:formula (and
(= A 98798798987987987987987923423879)
(= B 76342999998888888888736457864587)
(= (+ (* 87 A) (* 93 B)) (+ C C))
))
```

yields the output

```
sat
(= A 98798798987987987987987923423879)
(= B 76342999998888888888736457864587)
(= C 7847697255925810810803719959642032)
```

120

One can also use **functions**:

```
(benchmark test.smt
:extrafuns ((F Int Int))
:formula (and
(> (* 2 (F 1)) (+ (F 2) (F 3)))
(> (* 2 (F 2)) (+ (F 3) (F 4)))
(> (* 2 (F 3)) (* 3 (F 4)))
(> (* 3 (F 4)) (+ (F 1) (F 3)))
))
```

yielding

```
sat
(= (F 1) 30)
(= (F 2) 27)
(= (F 3) 32)
(= (F 4) 21)
```

121

One can even use **quantifications**:

```
(benchmark test.smt
:extrafuns ((F Int Int))
:formula (and
(forall (?i Int) (= (F ?i) (F (+ ?i
1))))
(= (F 1) 1)
(= (F 1000) 2)
))
```

yielding **unsat**

When quantifying over integers instead of `sat` one typically gets **unknown**: one can not expect that the tool can check `(forall (?i Int))...` for all integers

The required theory (integer/real, functions?, quantifications?) can be specified

If it is omitted as we did, it is detected from the formula

122

Pseudo boolean constraints

Here all variables are boolean

These boolean variables are identified with the numbers:

$$\text{false} \equiv 0, \quad \text{true} \equiv 1$$

A **pseudo boolean constraint** is a constraint of the shape

$$a_1x_1 + a_2x_2 + a_3x_3 + \dots + a_nx_n \leq c$$

for a_i, c all being given integer values, and x_i are boolean variables

So this is SMT for the theory of linear inequalities over de integers, with the restriction that the values of the variables only may be 0 or 1

123

A fruitful method for solving pseudo boolean constraints transforms the problem to propositional SAT and then applies a SAT solver, so a different approach then by using an SMT solver

A tool doing so is `minisat+`

Constraints with $=$ or \geq are easily transformed to constraints with \leq :

$$\sum_{i=1}^n a_i x_i \geq c \iff \sum_{i=1}^n -a_i x_i \leq -c$$

$$E = c \iff E \leq c \wedge E \geq c$$

124

Many problems are much easier expressed by pseudo boolean constraints than as a SAT problem

For instance, for expressing that exactly one of x_1, \dots, x_n is true, instead of a quadratic number of clauses we only need one constraint

$$x_1 + x_2 + \dots + x_n = 1$$

This holds even more for

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = c$$

for given numbers c, a_1, \dots, a_n

This kind of constraints occurs a lot in **scheduling problems**

125

For instance if x_i means that a particular course of size a_i will be followed, then the constraint that a series of courses has to be followed with a total size c can be expressed as

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = c$$

Later in the course we will see how pseudo boolean constraints can be transformed to SAT clauses

These problems can also be expressed in SMT for the theory of linear inequalities over the integers: simply add the restriction that the values of the variables only may be 0 or 1

However, for pseudo boolean constraints often `minisat+` is much faster than applying a general SMT solver

126

Unique representation

for boolean functions

In "normal" propositional notation equivalent formulas (even in CNF) may appear quite different:

$$(p \vee q) \wedge (q \vee r) \wedge (p \vee \neg q)$$

$$\equiv$$

$$p \wedge (\neg p \vee r \vee q) \wedge (p \vee \neg q)$$

We look for a way to describe boolean functions, in particular given by propositional formulas, such that:

- it is a **unique representation** for boolean functions: equivalent formulas yield the same representation
- for many formulas this representation is efficient

127

Why not for **all** formulas?

On n variables a truth table consists of 2^n lines

Hence on n variables there are 2^{2^n} distinct boolean functions

Indeed, there are 2^{64} distinct boolean functions on six variables

If all of these 2^{2^n} distinct boolean functions should have a distinct representation, then **on average** at least 2^n bits are needed for that

Hence for every representation it holds that if some boolean functions have a representation of much less than 2^n bits, then for many others 2^n bits or more are required

This information theoretic argument shows that it is **unavoidable** that most of the boolean functions have **untractable** representation

128

A computable unique representation immediately implies a method for SAT:

For any formula compute its unique representation

If this representation is equal to the representation of *false*, then the formula is unsatisfiable

Otherwise, the formula is satisfiable

129

Unique representation due to Herbrand, 1929

Every boolean function can uniquely be expressed as an exclusive or (\oplus) of conjunctions of variables, in which both \oplus and \wedge run over sets, i.e.,

- commutative, associative
- double occurrence is not allowed
- *true* is conjunction over empty set
- *false* is \oplus over empty set

In this way negation is not required as a building block:

$$\neg x \equiv x \oplus \text{true}$$

130

The unique representation of a propositional formula can be computed by the rules

$$\begin{aligned} x \leftrightarrow y &\rightarrow \neg(x \oplus y) \\ x \rightarrow y &\rightarrow (\neg x) \vee y \\ x \vee y &\rightarrow (x \wedge y) \oplus x \oplus y \\ \neg x &\rightarrow x \oplus \text{true} \\ x \oplus \text{false} &\rightarrow x \\ x \oplus x &\rightarrow \text{false} \\ x \wedge \text{false} &\rightarrow \text{false} \\ x \wedge \text{true} &\rightarrow x \\ x \wedge x &\rightarrow x \\ x \wedge (y \oplus z) &\rightarrow (x \wedge y) \oplus (x \wedge z) \end{aligned}$$

131

Example:

$\neg a \vee b$ and $\neg(a \wedge \neg b)$ are equivalent since they yield the same representation $a \wedge b \oplus a \oplus \text{true}$:

$$\begin{aligned}
\neg a \vee b &\rightarrow (a \oplus \text{true}) \vee b \\
&\rightarrow ((a \oplus \text{true}) \wedge b) \oplus a \oplus \text{true} \oplus b \\
&\rightarrow (a \wedge b) \oplus (\text{true} \wedge b) \oplus a \oplus \text{true} \oplus b \\
&\rightarrow (a \wedge b) \oplus b \oplus a \oplus \text{true} \oplus b \\
&\rightarrow (a \wedge b) \oplus a \oplus \text{true} \oplus \text{false} \\
&\rightarrow (a \wedge b) \oplus a \oplus \text{true}
\end{aligned}$$

$$\begin{aligned}
\neg(a \wedge \neg b) &\rightarrow \neg(a \wedge (b \oplus \text{true})) \\
&\rightarrow \neg((a \wedge b) \oplus (a \wedge \text{true})) \\
&\rightarrow \neg((a \wedge b) \oplus a) \\
&\rightarrow (a \wedge b) \oplus a \oplus \text{true}
\end{aligned}$$

132

Although this Herbrand representation is a unique representation it is not of practical use since it is not acceptably efficient for larger formulas

Before introducing B(inary) D(ecision) D(iagrams) as a more efficient unique representation, first we present an important application, to get a feeling for the kind of operations that are required to be efficient

We express the **Reachability problem**

Given a set of initial states, a transition relation and a set of final states, establish whether any of these final states is reachable from a given initial state

in propositional logic and show how to treat it by manipulation of unique representation

133

Example: Apply the following statements in any order:

1. if D and E then $D, E := \text{false}, \text{false}$
2. if F and I then $F, I := \text{false}, \text{false}$
3. if A and E then $A, E := \text{false}, \text{false}$
4. if C and D then $C, D := \text{false}, \text{false}$

5. if D and G then $D, G := \text{false}, \text{false}$
6. if B and H then $B, H := \text{false}, \text{false}$
7. if B and I then $B, I := \text{false}, \text{false}$
8. if A and G then $A, G := \text{false}, \text{false}$
9. if D and H then $D, H := \text{false}, \text{false}$
10. if B and C then $B, C := \text{false}, \text{false}$
11. if B and J then $B, J := \text{false}, \text{false}$
12. if F and J then $F, J := \text{false}, \text{false}$

Prove that from the initial state in which all variables have value *true* never the final state can be reached in which all variables have value *false*

134

Each of the ingredients should be represented by a formula

For the set I of initial states find a formula Φ_I such that Φ_I yields *true* on a state if and only if it is in I

More precisely, a state is a map π from variables to $\{\text{false}, \text{true}\}$ and it should hold that $\pi \in I$ if and only if evaluating Φ_I in π yields *true*

For example, in our example we may choose $\Phi_I = A \wedge B \wedge \dots \wedge J$ expressing the single state in which all variables are true

Similar for the set F of final states, in our example $\Phi_F = \neg A \wedge \neg B \wedge \dots \wedge \neg J$ expressing the single state in which all variables are false

135

Describing the transition relation is slightly more involved

A transition relation is not a set of states, but a set of state transitions, where a state transition is a pair of states

In representing this by a formula two kinds of variables occur: the starting point of such a transition and the end point of the transition

Distinguish these kinds by a suffix:

- for the start states the variable names are followed by the symbol 0
- for the end states the variable names are followed by the symbol 1

136

A formula Φ_T in this double set of variables describes the following set of transitions:

there is a transition from π_0 to π_1
if and only if the formula Φ_T yields *true* if for every i_0 the value $\pi_0(i_0)$ is evaluated, and for every i_1 the value $\pi_1(i_1)$ is evaluated

As an example, consider

if a then $b := c$

over the four variables a, b, c, d

The transition relation corresponding to this program fragment can be represented in this way by

$$(a_0 \leftrightarrow a_1) \wedge (c_0 \leftrightarrow c_1) \wedge (d_0 \leftrightarrow d_1) \wedge (a_0 \rightarrow (b_1 \leftrightarrow c_0)) \wedge (\neg a_0 \rightarrow (b_0 \leftrightarrow b_1))$$

137

Once we have expressed the initial states and the transition relation in this way we want to compute

$B_n =$ set of states reachable in $\leq n$ steps from I

for $n = 0, 1, \dots$, until $B_n = B_{n-1}$ or until $B_n \cap F \neq \emptyset$

This can be done by

$$B_0 := I$$

$$B_{i+1} := B_i \cup \{t \mid \exists s : s \in B_i \wedge sTt\}$$

We should be able to decompose this full computation into small computable elements

138

Operations \cup and \cap on sets correspond to \vee and \wedge in propositional notation

In $\exists s : s \in B_i \wedge sTt$ we use variables labelled by 0 for the part $s \in B_i$, and simply use \wedge to compute

$$s \in B_i \wedge sTt$$

This is a boolean function in variables both labelled by 0 and by 1

By $\exists s$ all variables labelled by 0 should be eliminated; this can be done one by one for every variable x :

$$\exists x : \phi \equiv \underbrace{\phi[x := false] \vee \phi[x := true]}_{\text{computable}}$$

Finally in the resulting representation over variables labelled by 1, every label 1 should be replaced by 0

139

Summarizing, the following operations should be efficient:

- Operations \cup and \cap on sets correspond to \vee and \wedge in propositional notation
- Checking $B_n = B_{n-1}$; for this we need a unique representation
- Compute $\phi[x := false]$ and $\phi[x := true]$ from ϕ

- Rename variables (label 1 by label 0)

B(inary) **D**(ecision) **D**(iagrams) are a representation in which all of these operations can be done efficiently

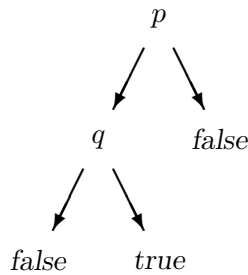
The above scheme is followed in the implementation of `bddsolve` for reachability problems

140

Decision trees

A **decision tree** is a binary tree in which

- nodes are labelled by boolean variables
- leaves are labelled by *false* or *true*



A decision tree describes a boolean function by starting at the root and consider every node as an if-then-else-

141

More precisely, if every variable has a boolean value then the corresponding function value is obtained is follows:

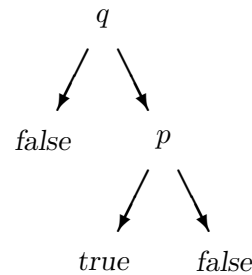
- Start at the root
- For a node proceed by its left branch if the corresponding variable is *true*
- For a node proceed by its right branch if the corresponding variable is *false*
- Repeat until a leaf has been reached
- The label of the leaf is the resulting function value

So the example describes the same boolean function as $p \wedge \neg q$

Sometimes the left (*true*) branch is written solid and the right (*false*) branch is written dashed

142

Unfortunately this representation is not yet unique: the same boolean function is described by the decision tree



We may disallow this second representation if we require that below every variable in a tree only **greater** variables are allowed, with respect to some total order $<$ on the variables

Such a decision tree is called **ordered** with respect to $<$

In our example: $p < q$

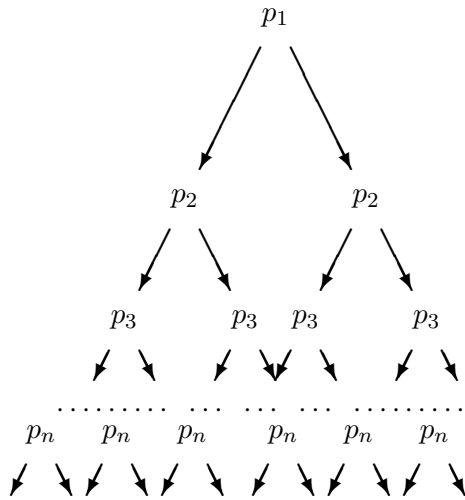
143

Every boolean function on a finite number of variables can be represented as an ordered decision tree:

simply transform its truth table

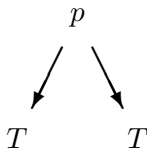
144

For the order $p_1 < p_2 < p_3 < \dots < p_n$ we obtain:



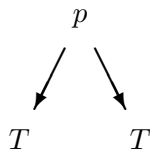
145

Still the representation is not yet unique, since for every decision tree T and every variable p



describes the same boolean function as T itself

Replacing



by T is called **elimination**

If no elimination is applicable on T then T is called **reduced**

146

Now we do have a unique representation:

Every boolean function can be expressed in exactly one way by a reduced ordered decision tree

Ingredients for the proof:

- Induction on the number of variables
- The smallest variable either occurs only at the top or does not occur at all

Exercise:

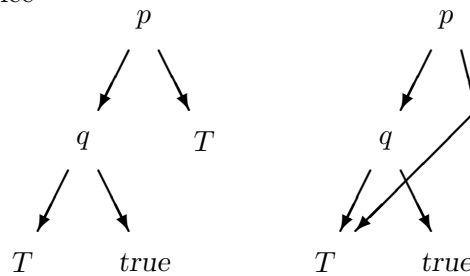
Determine the reduced ordered decision trees for the boolean functions described by the following formulas, with respect to the order $p < q < r$

1. $p \vee q \vee r$
2. $r \rightarrow (q \vee (p \wedge q))$
3. $p \leftrightarrow (q \leftrightarrow r)$

147

For efficient storage of such decision trees we prefer the following rule

If a subtree of a decision tree has multiple occurrences, we want to store this subtree only once

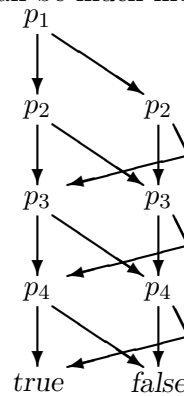


The right representation is preferred

This is a **D(irected) A(cyclic) G(raph)** rather than a tree

148

This can be much more efficient



This DAG has 7 nodes, while the tree has 15

Generalized to p_1, \dots, p_n : the DAG has $2n - 1$ nodes, while the tree has $2^n - 1$: a ratio of exponential size

149

For the implementation it hardly makes any difference

In the usual implementation for decision trees (or other binary trees) for every node its label and the pointers to its children are stored

Due to the tree structure to every node there is exactly one pointer

For DAGs the same data structure can be used, only now there may be more pointers to one node

A **B**(inary) **D**(ecision) **D**(iagram) is a decision tree in DAG representation

An **O**(rdered) **BDD** is an ordered decision tree in DAG representation

150

Both for uniqueness and for efficiency we not only want to admit common reference to the same node, we even want to **force** it

In the example we want to disallow the left representation

The pointer describing the left branch of a node is called the *true*-successor

The pointer describing the right branch of a node is called the *false*-successor

We require that no two nodes in the DAG have the following three properties:

- both are labelled by the same variable
- both have the same *true*-successor
- both have the same *false*-successor

151

Every decision tree can be transformed to a DAG satisfying this requirement:

As long there are two such nodes, remove one of them and redirect every reference to the removed one to the remaining one

This is called **merging**

So both elimination and merging are a way to decrease the size of a DAG without affecting its meaning: it is a kind of **reduction**

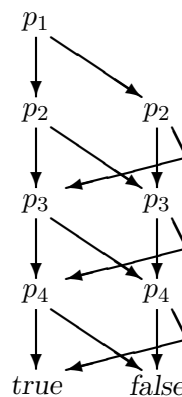
A **R**(educed) **OBDD** is an OBDD on which no elimination and no merging is possible

Main theorem:

For every order on the variables every boolean function has exactly one representation as a ROBDD

152

Example:



is the unique ROBDD of the boolean function described by the formula

$$((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \leftrightarrow p_4$$

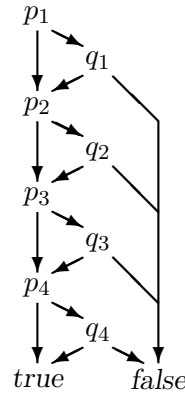
and the order $p_1 < p_2 < p_3 < p_4$

This formula yields *true* if and only if an even number of the variables has the value *true*

153

Proof of the main theorem:

- At least one ROBDD representation:
 - Start by the full ordered decision tree representing the truth table
 - Apply elimination and merge on this decision tree as long as possible
 - This ends since by every step the size decreases
 - The resulting BDD is by construction an ROBDD representing the given boolean function



However, if $p_1 < p_2 < \dots < p_n < q_1 < q_2 < \dots < q_n$ then the ROBDD of the same formula has over 2^n nodes

- At most one ROBDD representation:
 - Assume ROBDDs T and U represent the same boolean function
 - Unwind T to a tree T' and U to a tree U'
 - Then T' and U' are reduced ordered decision trees representing the same boolean function
 - Earlier result $\implies T' = U'$
 - Lemma $\implies T = U$

156

Heuristics for choosing a good order: variables close together in the formula, influencing each other, should be close in the order

Choosing an appropriate order then for many applications the unique ROBDD representation is feasible, even for $n > 1000$

154

Lemma: Every tree has a unique representation as a DAG with maximal sharing, i.e., a DAG on which no merge can be applied

End of proof of main theorem

Since there are 2^{2^n} distinct boolean functions many of them have a very big representation as a ROBDD

The size of the ROBDD strongly depends on the chosen order on the variables

If $p_1 < q_1 < p_2 < q_2 < \dots < p_n < q_n$ then the ROBDD of

$$\phi = (p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge \dots \wedge (p_n \vee q_n)$$

only has $2n$ nodes, and shows up for $n = 4$ as follows:

155

157

We still need an algorithm to compute the ROBDD for a given propositional formula and an order on the variables

Such an algorithm can be used for SAT:

Apply the algorithm to the given formula

If the resulting ROBDD is *false*, then the formula is unsatisfiable, and otherwise it is satisfiable

If the formula is satisfiable, then

- the ROBDD is not equal to *false* and at least one leaf is *true*, otherwise elimination can be applied
- a satisfying assignment is obtained from the ROBDD by following a path from the root to this leaf labelled by *true*

This is how the tool `bddsolve` for the practical assignment works, when using for SAT

158

Algorithm to determine the ROBDD of a formula

Every formula is of the shape:

- p for a variable p , or
- $\neg\phi$, or
- $\phi \diamond \psi$ for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

The ROBDD $ROBDD(\phi)$ of a formula ϕ will be constructed recursively according this recursive structure of the formulas:

- $ROBDD(false) = false, ROBDD(true) = true,$
- $ROBDD(p) = p(true, false)$
- $ROBDD(\neg\phi) = ROBDD(\phi \rightarrow false)$
- $ROBDD(\phi \diamond \psi) = apply(ROBDD(\phi), ROBDD(\psi), \diamond)$

159

So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

Notation:

- $p(T, U)$ is the BDD having root p for which the left branch is T and the right branch is U
- On OBDDs we define

$$p(T, U)(p := true) = T$$

$$p(T, U)(p := false) = U$$

- $T(p := true) = T(p := false) = T$ if p does not occur in T

As the basis of the recursion we define

$apply(T, U, \diamond) =$ value according the truth table of \diamond if $T, U \in \{false, true\}$

160

If T, U not both in $\{false, true\}$, hence T and/or U contains at least one variable, then we define

$$apply(T, U, \diamond) = p(apply(T(p := true), U(p := true), \diamond), apply(T(p := false), U(p := false), \diamond))$$

where p is the smallest variable occurring in T or U

Writing shortly $\diamond(T, U)$ for $apply(T, U, \diamond)$ this means

$$\diamond(p(T_1, T_2), p(U_1, U_2)) = p(\diamond(T_1, U_1), \diamond(T_2, U_2))$$

$$\diamond(p(T_1, T_2), U) = p(\diamond(T_1, U), \diamond(T_2, U))$$

if p does not occur in U

$$\diamond(T, p(U_1, U_2)) = p(\diamond(T, U_1), \diamond(T, U_2))$$

if p does not occur in T

161

So for two BDDs T, U computing

$$apply(T, U, \diamond) = \diamond(T, U)$$

is done by pushing \diamond downwards, meanwhile combining T and U , until \diamond applied to *true/false* has to be computed, which is replaced by its value according to the truth table

In this recursion only calls $apply(T', U', \diamond)$ are done where T' is a node of T and U' is a node of U

This is implemented in such a way that the same call $apply(T', U', \diamond)$ is never executed twice

Keep track for which pairs T', U' this has already been executed, using a hash table

162

Hence this algorithm $apply(T, U, \diamond)$ has complexity

$$O(\#T * \#U),$$

where $\#$ is the number of nodes of a BDD

Is the resulting BDD $apply(T, U, \diamond)$ always reduced?

NO

This can be repaired by applying elimination and merging in this process for every newly created node

This can be done in such a way that the complexity remains $O(\#T * \#U)$

The resulting algorithm $ROBDD$ is essentially the algorithm as it is used in practice (and in `bddsolve`)

163

Exercise

Compute the ROBDD of

$$(p \rightarrow r) \wedge (q \leftrightarrow (r \vee p))$$

with respect to the order $p < q < r$ using this algorithm

The algorithm $apply$ is polynomial in the size of its input (even quadratic)

However, in general the full algorithm $ROBDD$ is not polynomial: intermediate results may have exponential size

164

Example:

If $p_1 < p_2 < \dots < p_n < q_1 < q_2 < \dots < q_n$ then the ROBDD of

$$\phi = (p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge \dots \wedge (p_n \vee q_n)$$

has more than 2^n nodes

The algorithm $ROBDD$ applied on $p \wedge (\phi \wedge \neg p)$ will compute this big ROBDD as an intermediate result, and hence will have exponential complexity, although the final result is simply *false*

165

If in a special case all intermediate results are of polynomial size then the full algorithm is polynomial

since the full algorithm consists of a linear number of $apply$ calls, each having polynomial complexity

Every formula purely constructed from \neg, \leftrightarrow and variables has an ROBDD of which the size is linear in the size of the formula

Hence for such formulas the algorithm $ROBDD$ is always polynomial

166

Using resolution the behaviour may be just opposite:

Using only unit resolution it can be established in linear time that any formula of the shape $p \wedge (\phi \wedge \neg p)$ is unsatisfiable

Conversely, there are unsatisfiable formulas purely constructed from \neg, \leftrightarrow and variables of which it can be proved that every resolution proof requires an exponential number of steps

Conclusion:

Resolution and the algorithm $ROBDD$ are essentially incomparable

Both techniques are successfully applied to huge formulas

Typical situation in hardware verification

Two chip designs have to be proven to behave equivalent

More precisely, they should have the same input/output behaviour:

- both have k input nodes, n output nodes and a great number of internal nodes, all representing boolean values, connected by ports
- both chip designs compute the values of all n output nodes whenever the values of the k input nodes are set to boolean values
- if the input nodes are set to the same values for both chip designs, then the resulting values of the output nodes should be the same too, for all possible inputs

Example:

For a 32-bit multiplier we have $k = 64$ and $n = 32$

Let

- B_1, \dots, B_k express input nodes of both chip designs
- A_1, \dots, A_p express internal and output nodes of one chip design, the first $p - n$ being internal and the last n being output
- C_1, \dots, C_q express internal and output nodes of the other chip design, the first $q - n$ being internal and the last n being output

Typically, k and n are reasonably small and p and q are very big

We assume both chip designs have no circular behaviour:

the nodes are numbered in such a way that every A_i only depends on

$$B_1, \dots, B_k, A_1, \dots, A_{i-1},$$

Similarly for C_i

Let ϕ_1, \dots, ϕ_p be the simple formulas reflecting the behaviour of the ports, where ϕ_i describes how A_i depends on

$$B_1, \dots, B_k, A_1, \dots, A_{i-1}$$

So $A_i \leftrightarrow \phi_i$ describes how the value of A_i is computed

Similarly ψ_1, \dots, ψ_q describe C_1, \dots, C_q

To be proven:

the last n nodes A_{p-n+1}, \dots, A_p from the one chip design have the same behaviour as the last n nodes C_{q-n+1}, \dots, C_q from the other chip design

Expressed in a formula:

$$\left(\bigwedge_{i=1}^p (A_i \leftrightarrow \phi_i) \right) \wedge \left(\bigwedge_{i=1}^q (C_i \leftrightarrow \psi_i) \right)$$

implies

$$\bigwedge_{i=1}^n A_{p-n+i} \leftrightarrow C_{q-n+i}$$

Using resolution these are formulas in $k + p + q$ variables plus auxiliary variables due to Tseitin transformation

Using BDDs this can be done much more efficient:

- only consider B_1, \dots, B_k as variables
- subsequently compute ROBDDs for A_1, \dots, A_p using ϕ_1, \dots, ϕ_p
- similarly for C_1, \dots, C_q
- check whether the ROBDDs of A_{p-n+i} and C_{q-n+i} coincide for $i = 1, \dots, n$

In this way in practice BDD technology is used a lot in hardware verification

172

BDDs for pseudo boolean constraints

Some time ago we claimed that pseudo boolean constraints, i.e., constraints of the shape

$$\sum_{i=1}^n c_i x_i \leq a$$

for given integers c_i, a and boolean variables x_i can be transformed to CNF

One way to do so makes use of BDDs

The constraint

$$\sum_{i=1}^n c_i x_i \leq a$$

can be seen as a boolean function in x_1, \dots, x_n , yielding true if the constraint holds and yielding false otherwise

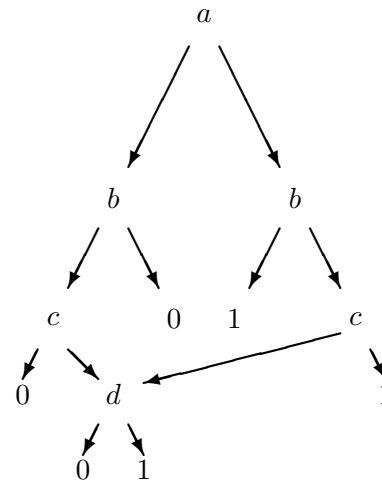
Construct the ROBDD of this boolean function

173

Heuristic: choose variable order in such a way that variables with high absolute coefficients (being influential) are low in the order, so will be tested first

Example:

$$3a - 2b + c + d \leq 1$$



174

How to build such an ROBDD?

- Evaluate values a, b, \dots in the inequality

E.g., in the example:

$$a = 1 \text{ yields } -2b + c + d \leq -2$$

$$a = 1, b = 1 \text{ yields } c + d \leq 0$$

- As soon as an inequality is false, put 0, if it is true, put 1

- Keep track of all inequalities so far

Every inequality corresponds to a node

If an inequality is found equivalent to an inequality found before, then point to the corresponding node

In this way sharing is introduced

175

The next step is to transform the ROBDD to a CNF

This is done via the Tseitin transformation

For propositional formulas composed from

$$\neg, \vee, \wedge, \rightarrow, \leftrightarrow$$

we discussed the Tseitin transformation to CNF by giving a fresh name to every subformula

Here we consider the nodes of a BDD as an if-then-else, which can be seen as propositional operation with three arguments:

$$\text{if } p \text{ then } q \text{ else } r = ((p \rightarrow q) \wedge (\neg p \rightarrow r))$$

Instead of giving a fresh name to every subformula we give a fresh name to every node in the BDD

This is the same idea as before, with the extra facility of sharing

176

The **Tseitin transformation** consists of

- A unit clause consisting of the name of the root
- The CNF

$$\neg A \vee \neg p \vee B$$

$$\neg A \vee p \vee C$$

$$A \vee p \vee \neg C$$

$$A \vee \neg p \vee \neg B$$

$$A \vee \neg B \vee \neg C$$

of $A \leftrightarrow ((p \rightarrow B) \wedge (\neg p \rightarrow C))$ for every node A labelled by p where the *true*-branch points to B and the *false*-branch to C

- Here B, C are either names of nodes or *false* or *true*

In this way the size of the Tseitin transformation is linear in the size of the BDD

177

In our example we get the unit clause A together with the CNFs of the following 6 formulas:

$$A \leftrightarrow ((a \rightarrow B_1) \wedge (\neg a \rightarrow B_2))$$

$$B_1 \leftrightarrow ((b \rightarrow C_1) \wedge (\neg b \rightarrow \text{false}))$$

$$B_2 \leftrightarrow ((b \rightarrow \text{true}) \wedge (\neg b \rightarrow C_2))$$

$$C_1 \leftrightarrow ((c \rightarrow \text{false}) \wedge (\neg c \rightarrow D))$$

$$C_2 \leftrightarrow ((c \rightarrow D) \wedge (\neg c \rightarrow \text{true}))$$

$$D \leftrightarrow ((d \rightarrow \text{false}) \wedge (\neg d \rightarrow \text{true}))$$

Note the sharing of D

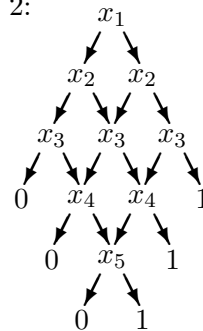
178

Sharing really helps: for

$$\sum_{i=1}^{2n+1} x_i \leq n$$

the BDD size is $(n + 1)^2$, while unshared it would have been exponential in n

For $n = 2$:



179

This BDD based technique is one of the three techniques used by **minisat+** to transform pseudo boolean constraints to CNF, and hence solve pseudo boolean constraint problems by means of a SAT solver

180

Predicate logic

Until now we only reasoned in the world of the domain $\{\text{true}, \text{false}\}$, together with the usual operations

Now we want to do automated reasoning in an arbitrary bigger domain D in which we can quantify using \forall and \exists , and where we may have **relations** and **functions**

(just like SMT, abstracting from the integers)

This is called **Predicate logic**

Example

- There is a student that is awake during all lectures
- During all boring lectures no student keeps awake
- Then there are no boring lectures

181

How can we prove this?

Just like proposition logic: take the conjunction of all given statements and the negation of the conclusion, and try to prove that the resulting formula is unsatisfiable, i.e., equivalent to *false*

In order to express the example by a formula we define relations S , L , B and A :

- $S(x)$: x is a student
- $L(x)$: x is a lecture
- $B(x)$: x is boring
- $A(x, y)$: x is awake during y

182

Hence we want to prove that

$$(\exists x(S(x) \wedge \forall y(L(y) \rightarrow A(x, y)))) \wedge (\forall x((L(x) \wedge B(x)) \rightarrow \neg \exists y(S(y) \wedge A(y, x)))) \wedge \exists x(L(x) \wedge B(x))$$

is unsatisfiable = equivalent to *false*

What does this mean exactly?

In such an expression we may have

- **variables** (here x, y)
- **function symbols** (not here)
- **relation symbols** (here S, L, B, A), also called **predicate symbols**

Function symbols and relation symbols have an **arity** = 0, 1, 2, 3, ...

A function symbol of arity 0 is also called a **constant**

183

We inductively define:

- A **term** is
 - a variable from a set \mathcal{X} , or
 - a function symbol of arity n applied on n terms
- A **predicate (formula)** is
 - a relation symbol of arity n applied on n terms, or
 - $\forall x(P)$ for a variable $x \in \mathcal{X}$ and a predicate P , or
 - $\exists x(P)$ for a variable $x \in \mathcal{X}$ and a predicate P , or
 - $\neg P$ for a predicate P , or
 - $P \diamond Q$ for predicates P and Q and $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

184

If we give meaning to variables, function symbols and relation symbols in a **model**, then a predicate has a boolean value

More precisely, a **model** is a non-empty set M together with

- $[f] : M^n \rightarrow M$ for every function symbol f of arity n
- $[R] : M^n \rightarrow \{false, true\}$ for every relation symbol R of arity n

For $\alpha : \mathcal{X} \rightarrow M$ we define inductively

- $[x, \alpha] = \alpha(x)$ for $x \in \mathcal{X}$
- $[f(t_1, \dots, t_n), \alpha] = [f]([t_1, \alpha], \dots, [t_n, \alpha])$ for every function symbol f of arity n and terms t_1, \dots, t_n
- $[R(t_1, \dots, t_n), \alpha] = [R]([t_1, \alpha], \dots, [t_n, \alpha])$ for every function symbol f of arity n and terms t_1, \dots, t_n

So $[f(t_1, \dots, t_n), \alpha] \in M$ and $[R(t_1, \dots, t_n), \alpha] \in \{false, true\}$

185

In order to define \forall and \exists we need to modify the valuation α

If $\alpha : \mathcal{X} \rightarrow M$, $x \in \mathcal{X}$ and $m \in M$ then we define $\alpha\langle x := m \rangle : \mathcal{X} \rightarrow M$ by

$$\alpha\langle x := m \rangle(x) = m$$

$$\alpha\langle x := m \rangle(y) = \alpha(y)$$

for all $y \in \mathcal{X}$, $y \neq x$

We define

$$[\forall x(P), \alpha] = \bigwedge_{m \in M} [P, \alpha\langle x := m \rangle]$$

$$[\exists x(P), \alpha] = \bigvee_{m \in M} [P, \alpha\langle x := m \rangle]$$

Avoid problems by disallowing $\forall x$ or $\exists x$ to occur inside P for same x : choose fresh x for every quantification

186

We define

$$[\neg P, \alpha] = \neg[P, \alpha]$$

and

$$[P \diamond Q, \alpha] = [P, \alpha] \diamond [Q, \alpha]$$

for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

In this way $[P, \alpha] \in \{false, true\}$ has been defined for every predicate P and every $\alpha : \mathcal{X} \rightarrow M$

A predicate P is **satisfiable** if there is a model M and $\alpha : \mathcal{X} \rightarrow M$ such that $[P, \alpha] = true$

All these definitions are motivated by common knowledge of the usual notions of \forall and \exists

187

Proposition logic can be seen as a special case of predicate logic in which

- there are no variables,
- there are no function symbols, and
- all relation symbols (corresponding to propositional variables) have arity 0

Predicate logic can be expressed in SMT, but tools like `yices` are very weak in quantification, e.g.

```
(benchmark test.smt
:extrapreds ((P Int))
:extrafuns ((a Int))
:formula
(and
(forall (?x Int) (P ?x))
(forall (?x Int) (not (P ?x))))
))
yields unknown
```

188

Semantic Tableaux

This is a technique to prove unsatisfiability of a proposition or a predicate by building a tree

in which every node represents case analysis and in which every branch contains a contradiction, being a combination of P and $\neg P$ for some P

For

- $P \wedge Q$
- $P \vee Q$
- $P \rightarrow Q$
- $P \leftrightarrow Q$
- $\neg\neg P$
- $\neg(P \wedge Q)$
- $\neg(P \vee Q)$
- $\neg(P \rightarrow Q)$
- $\neg(P \leftrightarrow Q)$

we keep the same rules as we had before

189

We add new rules:

- Universal instantiation:
below $\forall x(A)$ the new predicate $A[x := t]$ may be added for every term t
- Existential instantiation:
below $\exists x(A)$ the new predicate $A[x := a]$ may be added for a fresh constant a

'Fresh' means that it does not occur somewhere else; the fresh constant a is called **witness**
- DeMorgan rules:
 - below $\neg(\forall x(A))$ the new predicate $\exists x(\neg A)$ may be added
 - below $\neg(\exists x(A))$ the new predicate $\forall x(\neg A)$ may be added

190

We keep the heuristics of postponing case analysis, as obtained by

- $P \vee Q$
- $P \rightarrow Q$
- $P \leftrightarrow Q$
- $\neg(P \wedge Q)$
- $\neg(P \leftrightarrow Q)$

For universal instantiation arbitrary terms may be chosen, note that this usually allows infinitely many correct possible steps

It is unclear which one to choose

We apply the heuristic to postpone this as long as possible

191

In the boring lecture example we instantiate

$$\exists x(S(x) \wedge \forall y(L(y) \rightarrow A(x, y)))$$

by a and

$$\exists x(L(x) \wedge B(x))$$

by b , resulting in

$$S(a) \wedge \forall y(L(y) \rightarrow A(a, y))$$

and

$$L(b) \wedge B(b)$$

By instantiating both $\forall y(L(y) \rightarrow A(a, y))$ and

$$\forall x((L(x) \wedge B(x)) \rightarrow \neg \exists y(S(y) \wedge A(y, x)))$$

by b , and next instantiate

$$\forall y(\neg(S(y) \wedge A(y, b)))$$

by a all branches will be closed

192

In case of occurrence of function symbols there are infinitely many terms given unbounded choice for universal instantiation

This makes this method hard to implement

Now we will extend the **resolution method** to be applicable for predicates

As before resolution is only defined for conjunctive normal forms (CNF), where

- a CNF is a conjunction of clauses,
- a clause is a disjunction of literals, and
- a literal is an atomic formula or its negation

Here an **atomic formula** is an expression of the shape $P(t_1, \dots, t_n)$ where P is a relation symbol of arity n and t_1, \dots, t_n are terms

A clause will be interpreted as being universally quantified over all occurring variables

193

As before the only thing to be done by resolution is proving that a CNF is unsatisfiable by deriving the empty clause

As before this will be extended to arbitrary formulas by giving a transformation from an arbitrary formula to CNF

Due to terms and variables occurring in atomic formulas and implicit universal quantification of clauses the resolution rule

$$\frac{P \vee V, \quad \neg P \vee W}{V \vee W}$$

will be slightly more complicated now

194

Example:

A special case of the clause

$$P(f(x), y) \vee Q(x, y)$$

is $P(f(x), g(y)) \vee Q(x, g(y))$

A special case of the clause

$$\neg P(x, g(y)) \vee R(x, y)$$

is $\neg P(f(x), g(y)) \vee R(f(x), y)$

On both 'special cases' now resolution yields the new clause $Q(x, g(y)) \vee R(f(x), y)$

Now we want to see

$$\frac{P(f(x), y) \vee Q(x, y), \quad \neg P(x, g(y)) \vee R(x, y)}{Q(x, g(y)) \vee R(f(x), y)}$$

as a valid resolution step

195

A **substitution** is a map from variables to terms

A substitution σ can be extended to arbitrary terms and atomic formulas by inductively defining:

$$x\sigma = \sigma(x)$$

for every variable x and

$$F(t_1, \dots, t_n)\sigma = F(t_1\sigma, \dots, t_n\sigma)$$

for every function/relation symbol F

So $t\sigma$ is obtained from t by replacing every variable x in t by $\sigma(x)$

For instance, if $\sigma(x) = y$ and $\sigma(y) = g(x)$ then

$$P(f(x), y)\sigma = P(f(y), g(x))$$

196

Let X be the set of variables and $T(X)$ the set of terms, then in this way the given substitution

$$\sigma : X \rightarrow T(X)$$

is extended to

$$\sigma : T(X) \rightarrow T(X)$$

The latter σ is written in postfix notation

If both σ and τ are substitutions we can define the **composition** $\sigma\tau$:

$$x(\sigma\tau) = (x\sigma)\tau$$

for all $x \in X$, by which we have

$$t(\sigma\tau) = (t\sigma)\tau$$

for all $t \in T(X)$

$\sigma\tau$ means: first apply σ , then τ

Here we do not have the usual confusion of composition in prefix notation, where $f \circ g$ means: first apply g , then f

197

For a clause $V = P_1 \vee P_2 \vee \dots \vee P_n$ and a substitution σ we write

$$V\sigma = P_1\sigma \vee P_2\sigma \vee \dots \vee P_n\sigma$$

For every substitution σ we want to consider the clause $V\sigma$ as a special case of the clause V

Now the general version of the **resolution rule** for predicates reads:

$$\frac{P \vee V, \neg Q \vee W}{V\sigma \vee W\tau}$$

for all substitutions σ, τ satisfying

$$P\sigma = Q\tau$$

198

In order to be able to use this rule we need an algorithm to determine whether substitutions σ, τ exist such that $P\sigma = Q\tau$ for given atomic formulas P and Q , and if so, to find them

This is called **unification**

Examples:

- $P(f(x), y)$ and $P(x, g(y))$ unify by choosing

$$\sigma(x) = x, \sigma(y) = g(y),$$

$$\tau(x) = f(x), \tau(y) = y$$

- $P(f(x), y)$ and $Q(x, g(y))$ do not unify

- $P(f(x), f(y))$ and $P(x, g(y))$ do not unify

- $P(f(x), x)$ and $P(x, x)$ do not unify

199

For unification there is no principal difference between terms and atomic formulas, neither between function symbols and relation symbols

Moreover by renaming of variables we can force that P and Q have no variables in common

Then the behavior of the two substitutions σ, τ can be expressed by one single substitution

Now the general unification problem reads:

Given two terms P and Q , is there a substitution σ such that $P\sigma = Q\sigma$?

If so, find it

The resulting substitution σ is called a **unifier**

200

Simple observation:

If σ is a unifier for (P, Q) and τ is an arbitrary substitution, then $\sigma\tau$ is a unifier too for (P, Q)

Definition:

A unifier σ_0 is called a **most general unifier (mgu)** for (P, Q) if for every unifier σ

for (P, Q) a substitution τ exists such that $\sigma = \sigma_0\tau$

If both σ_0 and σ_1 are an mgu for (P, Q) , then by definition τ_0, τ_1 exist such that

$$\sigma_1 = \sigma_0\tau_0 \quad \text{and} \quad \sigma_0 = \sigma_1\tau_1$$

From this property one can conclude that σ_0 and σ_1 are equal up to renaming, hence an mgu is unique up to renaming

201

Hence we will speak about **the** mgu rather than **an** mgu

We will give an algorithm with two terms as input, and as output:

- whether the terms unify or not, and
- the mgu in case they unify

The existence of an mgu in case two terms unify is a consequence of this property of the algorithm

202

Write $v(t)$ for the test whether t is a variable

Write $in(x, t)$ for the test whether the variable x occurs in t , this is called **occur check**

The unification algorithm has the following invariant:

$$\begin{aligned} \exists\sigma(P\sigma = Q\sigma) &\equiv \exists\sigma(\forall(t, u) \in S(t\sigma = u\sigma)) \\ &\wedge (\exists\sigma(P\sigma = Q\sigma) \rightarrow un) \end{aligned}$$

where P, Q are the original terms to be unified

If $S = \emptyset$ then it follows that P and Q unify

If $\neg un$ then it follows that P and Q do not unify

Inspired by the invariant and these observations we arrive at the following unification algorithm:

203

$S := \{(P, Q)\};$

$un := true;$

while $(un \wedge S \neq \emptyset)$ do {

 choose $(t, u) \in S;$

$S := S \setminus \{(t, u)\};$

 if $t \neq u$ then

 if $v(t) \wedge in(t, u)$ then $un := false$

 else if $v(t) \wedge \neg(in(t, u))$ then

$S := S[t := u]$

 else if $v(u) \wedge in(u, t)$ then

$un := false$

 else if $v(u) \wedge \neg(in(u, t))$ then

$S := S[u := t]$

 else if $t = f(t_1, \dots, t_n) \wedge u = f(u_1, \dots, u_n)$

 then

$S := S \cup \{(t_1, u_1), \dots, (t_n, u_n)\}$

 else if $t = f(\dots) \wedge u = g(\dots) \wedge f \neq g$

 then

$un := false$

}

204

Basic idea of the algorithm:

- S is inspected and decomposed
- un is set to *false* if a reason is found that there is no unification, then the algorithm stops
- if such a reason is not found and the list S of unification requirements is empty, then there is a unifier

This unification algorithm always terminates, since in every step either

- the total number of variables occurring in S strictly decreases, or
- the total number of variables occurring in S remains the same and the total size of S decreases

Here total size of $\{(t_1, u_1), \dots, (t_n, u_n)\}$ is defined to be

$$\sum_{i=1}^n (|t_i| + |u_i|)$$

where $|t|$ is the size of the term t

205

After termination the following holds:

- $un = false$ and P and Q are not unifiable, or
- $un = true$ and P and Q are unifiable

If P and Q are unifiable, what about the unifier?

We extend the program by building up the unifier in a variable mgu

As only steps are done that are really forced, the resulting unifier mgu will be a most general unifier of P and Q

We write id for the substitution mapping every variable on itself

For a variable x and a term P we write $[x := P]$ for the substitution mapping x on P and every other variable on itself

This notation will be used for pairs of terms and sets of pairs of terms, meaning that the substitution is applied to every occurring term

206

```

S := {(P, Q)};
un := true;
mgu := id;
while (un ∧ S ≠ ∅) do {
  choose (t, u) ∈ S;
  S := S \ {(t, u)};
  if t ≠ u then
    if v(t) ∧ in(t, u) then un := false
    else if v(t) ∧ ¬(in(t, u)) then
      {S := S[t := u];
       mgu := mgu[t := u]}
    else if v(u) ∧ in(u, t) then

```

```

un := false
else if v(u) ∧ ¬(in(u, t)) then
  {S := S[u := t];
   mgu := mgu[u := t]}
else if t = f(t1, ..., tn) ∧ u = f(u1, ..., un)
then
  S := S ∪ {(t1, u1), ..., (tn, un)}
  else if t = f(⋯) ∧ u = g(⋯) ∧ f ≠ g
then
  un := false
}

```

207

Example:

Unify $P(f(x), y)$ and $P(z, g(w))$

Start:

$S = \{(P(f(x), y), P(z, g(w)))\}$, $mgu = id$

After 1 step:

$S = \{(f(x), z), (y, g(w))\}$, $mgu = id$

After 2 steps:

$S = \{(y, g(w))\}$, $mgu = [z := f(x)]$

After 3 steps:

$S = \emptyset$, $mgu = [z := f(x)][y := g(w)]$

So the resulting most general unifier σ is given by

$$x\sigma = x, y\sigma = g(w), z\sigma = f(x), w\sigma = w,$$

208

Example:

Unify $P(f(x), x)$ and $P(y, g(y))$

Start:

$S = \{(P(f(x), x), P(y, g(y)))\}$, $mgu = id$

After 1 step:

$S = \{(f(x), y), (x, g(y))\}$, $mgu = id$

After 2 steps:

$S = \{(x, g(f(x)))\}$, $mgu = [y := f(x)]$

After 3 steps:

x occurs in $g(f(x))$, hence no unification

Remarks:

- If two terms unify, then they have an mgu which is unique up to renaming of variables
- Occur check can be expensive: search for a particular variable in a big term
- There are optimizations of this unification algorithm that are linear
- The unifier can have a size that is exponential in the size of the terms to be unified
- Efficient algorithms use DAG representation for terms

Example:

Unification of

$$P(x_1, f(x_2, x_2), x_2, f(x_3, x_3), x_3)$$

and

$$P(f(y_1, y_1), y_1, f(y_2, y_2), y_2, f(y_3, y_3))$$

yields an mgu σ in which $x_1\sigma$ is a term containing 32 copies of the same variable and 31 f -symbols

Back to resolution

$$\frac{P \vee V, \neg Q \vee W}{V\sigma \vee W\tau}$$

for substitutions σ, τ satisfying

$$P\sigma = Q\tau$$

Here we can rename variables by which $P \vee V$ and $\neg Q \vee W$ have no variables in common

Now we restrict this general version of resolution:

instead of allowing the rule for all (infinitely many) unifiers of P and Q we **only** allow the mgu

So the resolution step can be described as follows:

- Take two (possibly equal) non-empty clauses
- Rename variables such that they do not have variables in common
- Choose a positive literal P in one of the clauses and a negative literal $\neg Q$ in the other
- Unify P and Q
- if they do not unify a corresponding resolution step is not possible
- if they unify then the clause

$$(V \vee W)\sigma$$

can be concluded, where

- $P \vee V$ is the one clause,
- $\neg Q \vee W$ is the other clause,
- σ is the most general unifier of P and Q

As a consequence, given a CNF, there are only finitely many possibilities of doing a resolution step, and these are computable

Theorem

(completeness of resolution)

A predicate in CNF is equivalent to *false* if and only if there is a sequence of resolution steps ending in the empty clause

Theorem

(undecidability of predicate logic)

No algorithm exists that can establish in all cases whether a given predicate in CNF is equivalent to *false*

214

These two theorems look contradictory, but they are not:

After extensive but unsuccessful search for a resolution sequence ending in the empty clause by only using completeness you are not able to conclude that such a resolution sequence does not exist

(compare to halting problem)

We do not prove these important theorems

215

Examples of resolution sequences:

$$\begin{array}{ll}
1 & P(x, f(y)) \vee \neg P(x, y) \\
2 & \neg P(x, f(f(y))) \\
3 & P(a, g(y)) \\
\hline
4 & P(a, f(g(y))) \quad (1, 3) \\
5 & P(a, f(f(g(y)))) \quad (1, 4) \\
& \perp \quad (2, 5)
\end{array}$$

216

$$\begin{array}{ll}
1 & P(x, f(y)) \vee \neg P(x, y) \\
2 & \neg P(x, f(f(y))) \\
3 & P(a, g(y)) \\
\hline
4 & \neg P(x, f(y)) \quad (1, 2) \\
5 & \neg P(x, y) \quad (1, 4) \\
& \perp \quad (3, 5)
\end{array}$$

In searching for such a resolution proof there is a lot of choice

This choice is more restricted if every clause contains at most one positive literal, making search for resolution much simpler

217

A **Horn clause** is a clause containing at most one positive literal

Usually a Horn clause $C \vee \neg C_1 \vee \dots \vee \neg C_n$ is written as

$$C \leftarrow C_1, \dots, C_n$$

or as

$$C :- C_1, \dots, C_n.$$

The positive literal C is called the **head**

A clause without a head is called a **goal**

A clause consisting only of a head is called a **fact**, it is written as C . instead of $C :-$.

A **Prolog program** is a set of Horn clauses in this notation

Prolog is a standard programming language in artificial intelligence

218

Example:

```

arrow(a,b).
arrow(a,c).
arrow(b,c).
arrow(c,d).
path(X,Y) :- arrow(X,Y).
path(X,Y) :- arrow(X,Z),path(Z,Y).

```

The first four clauses define a directed graph on four nodes

By the last two clauses the notion of a path in a graph is defined

If we wonder whether a path exists from a to d , then we add the goal

$:- \text{path}(a, d)$

being the clause $\neg\text{path}(a, d)$

Systematical depth first search for a resolution proof starting from the goal yields:

219

```

1  arrow(a, b)
2  arrow(a, c)
3  arrow(b, c)
4  arrow(c, d)
5  path(x, y) ∨ ¬arrow(x, y)
6  path(x, y) ∨ ¬arrow(x, z) ∨ ¬path(z, y)
7  ¬path(a, d)

```

(5, 7) no result

8 $\neg\text{arrow}(a, z) \vee \neg\text{path}(z, d)$ (6, 7)

9 $\neg\text{path}(b, d)$ (1, 8)

(5, 9) no result

10 $\neg\text{arrow}(b, z) \vee \neg\text{path}(z, d)$ (6, 9)

11 $\neg\text{path}(c, d)$ (3, 10)

12 $\neg\text{arrow}(c, d)$ (5, 11)

\perp (4, 12)

220

This kind of search for a resolution proof is called **SLD-resolution**

The found resolution sequence ending in \perp is called a **refutation**

In the example it was proved automatically that a path from a to d exists, while the input is nothing more than

- the definition of a graph
- the definition of the notion ‘path’
- the question: ‘is there a path from a to d ?’

221

This mechanism also applies for goals containing variables

For instance, the goal $:- \text{path}(a, X)$ will yield a refutation, but the goal $:- \text{path}(d, X)$ will not

This is quite subtle: sometimes search can go on for ever

Prolog is a **declarative programming language**, it is a kind of **logic programming**

In many Prolog implementations occur check is omitted for efficiency reasons, by which

$p(X, X)$.

$:- p(X, f(X))$.

yielding the CNF $p(X, X) \wedge \neg p(X, f(X))$ gives rise to an infinite computation

222

Back to general predicates ...

Now we shall see how a general predicate can be transformed to CNF maintaining satisfiability

This transformation consists from:

- **prenex normal form:** shift all quantifiers \forall and \exists to the front and write the body as a CNF

- **Skolemization:** removal of \exists by introducing fresh function symbols

223

Prenex normal form

Starting from an arbitrary predicate we apply the following steps:

- Remove \leftrightarrow by applying

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$$

- Remove \rightarrow by applying

$$A \rightarrow B \equiv (\neg A) \vee B$$

- Remove negations of non-atomic formulas by repetitively applying

$$- \neg(A \wedge B) \equiv (\neg A) \vee (\neg B)$$

$$- \neg(A \vee B) \equiv (\neg A) \wedge (\neg B)$$

$$- \neg(\neg A) \equiv A$$

$$- \neg(\forall x(A)) \equiv \exists x(\neg A)$$

$$- \neg(\exists x(A)) \equiv \forall x(\neg A)$$

224

The formula obtained so far is composed from $\vee, \wedge, \exists, \forall$ and literals

- Rename variables until over every variable there is at most one quantification
- Assuming non-empty domain now all quantifiers may be put at the front, for example

$$B \vee \forall x(A) \equiv \forall x(B \vee A)$$

where x does not occur in B

Now the formula consists of a quantor free body on which a number of quantors is applied

- Transform the body to CNF using

$$- A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

$$- (A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$$

225

The result is a prenex normal form, i.e., a CNF preceded by a number of quantors, being equivalent to the original predicate

If the result is too big then Tseitin's transformation can be applied

In order to reach the desired CNF format with implicit universal quantification it remains to eliminate the \exists -symbols

This is done by **Skolemization**, i.e., replace

$$\dots \exists y \dots (\dots y \dots)$$

by

$$\dots \dots (\dots f(x_1, \dots, x_n) \dots)$$

where f is a fresh function symbol and x_1, \dots, x_n are the universally quantified variables left from y

226

Example

Skolemization applied to

$$\exists z \forall x \exists y \forall u \forall v \exists w (A(x, y, z, u, v, w))$$

yields

$$\forall x \forall u \forall v (A(x, f(x), c, u, v, g(x, u, v)))$$

By Skolemization satisfiability is maintained:

Deriving a contradiction from $\forall x(A(x, f(x)))$ without any knowledge of f coincides with deriving a contradiction from

$$\exists f \forall x (A(x, f(x)))$$

According to the Axiom of Choice we have

$$\exists f \forall x (A(x, f(x))) \equiv \forall x \exists y (A(x, y))$$

227

By Skolemization all \exists -symbols are removed, introducing a fresh symbol for every removed \exists -symbol

The result is a CNF for which

- all variables are implicitly universally quantified, and

- satisfiability is equivalent to satisfiability of the original arbitrary predicate formula

228

The example of the boring lectures

$$(\exists x(S(x) \wedge \forall y(L(y) \rightarrow A(x, y)))) \wedge$$

$$(\forall x((L(x) \wedge B(x)) \rightarrow \neg \exists y(S(y) \wedge A(y, x)))) \wedge \exists x(L(x) \wedge B(x))$$

yields the CNF with refutation:

1	$S(c)$	
2	$\neg L(y) \vee A(c, y)$	
3	$\neg L(x) \vee \neg B(x) \vee \neg S(z) \vee \neg A(z, x)$	
4	$L(d)$	
5	$B(d)$	
6	$A(c, d)$	(2, 4)
7	$\neg B(d) \vee \neg S(z) \vee \neg A(z, d)$	(3, 4)
8	$\neg S(z) \vee \neg A(z, d)$	(5, 7)
9	$\neg A(c, d)$	(1, 8)
	\perp	(6, 9)

229

Equational reasoning

We will give a minimal description of natural numbers in which $2 + 2 = 4$ makes sense and can be proved automatically

Natural numbers:

$$0, s(0), s(s(0)), s(s(s(0))), \dots$$

These are the closed terms composed from the constant 0 and the unary symbol s

Here a term is called **closed** if it does not contain variables

We want to show that

$$s(s(0)) + s(s(0)) = s(s(s(s(0))))$$

Here $+$ is a binary operator written in infix notation

230

This claim only holds if we have some basic rules for $+$:

$+$ applied to natural numbers should yield a natural number after application of these basic rules

Here natural number numbers are defined to be closed terms composed from the constant 0 and the unary symbol s

Hence we need rules by which every closed term containing the symbol $+$ can be rewritten to a closed term not containing $+$

One way to do so is:

$$0 + x = x$$

$$s(x) + y = s(x + y)$$

231

What is the meaning of such rules?

- For variables (here: x, y) arbitrary terms may be substituted
- These rules may be applied on any sub-term of a term that has to be rewritten

In case the rules are only allowed to be applied from left to right we write an arrow \rightarrow instead of $=$

The rules are called **rewrite rules**

A set of such rewrite rules is called a **term rewrite system (TRS)**

232

More precisely:

A TRS R is a subset of $T \times T$, where T is the set of terms over a given set of function symbols and variables

An element $(\ell, r) \in R$ is called a **rule** and is usually written as $\ell \rightarrow r$ instead of (ℓ, r)

ℓ is called the left hand side and r is called the right hand side of the rule

The rewrite relation \rightarrow_R is defined to be the smallest relation $\rightarrow_R \subseteq T \times T$ satisfying:

- $\ell\sigma \rightarrow_R r\sigma$ for every $\ell \rightarrow r$ in R and every substitution σ
- if $t_j \rightarrow_R u_j$ and $t_i = u_i$ for every $i \neq j$, then $f(t_1, \dots, t_n) \rightarrow_R f(u_1, \dots, u_n)$

233

This last property causes that application of rules is allowed on subterms

For instance, we have

$$s(0) + (0 + s(0)) \rightarrow_R s(0) + s(0)$$

If the TRS R consists of the rules

$$0 + x \rightarrow x \quad s(x) + y \rightarrow s(x + y)$$

then indeed $2 + 2 = 4$ holds:

$$\begin{aligned} s(s(0)) + s(s(0)) &\rightarrow_R \underbrace{s(s(0) + s(s(0)))}_{\text{subterm}} \\ &\rightarrow_R \underbrace{s(s(0 + s(s(0))))}_{\text{subterm}} \\ &\rightarrow_R s(s(s(s(0)))) \end{aligned}$$

234

A term t is called a **normal form** if no u exists satisfying $t \rightarrow_R u$

Computation

=

rewrite to normal form

=

apply rewriting as long as possible

So in our example rewriting to normal form of the term $2 + 2$ represented by $(s(s(0)) + s(s(0)))$ yields the term 4 represented by $s(s(s(s(0))))$

A term t is called a **normal form of** a term u if t is a normal form and u rewrites to t in zero or more steps.

235

A rewriting sequence is also called a **reduction**; it can be infinite, unfinished, or end in a normal form

Rewriting to normal form is the basic formalism in several kinds of computation

In particular, it is the underlying formalism for both semantics and implementation of **functional programming**, in which the function definitions are interpreted as rewrite rules

236

Example

$$\begin{aligned} \text{rev}(\text{nil}) &= \text{nil} \\ \text{rev}(a : x) &= \text{conc}(\text{rev}(x), a : \text{nil}) \\ \text{conc}(\text{nil}, x) &= x \\ \text{conc}(a : x, y) &= a : \text{conc}(x, y) \end{aligned}$$

Here a, x, y are variables, and $=$ corresponds to \rightarrow in rewrite rules

Then we have a reduction to normal form

$$\begin{aligned} \text{rev}(1:2:\text{nil}) &\rightarrow \\ \text{conc}(\text{rev}(2:\text{nil}), 1:\text{nil}) &\rightarrow \\ \text{conc}(\text{conc}(\text{rev}(\text{nil}), 2:\text{nil}), 1:\text{nil}) &\rightarrow \\ \text{conc}(\text{conc}(\text{nil}, 2:\text{nil}), 1:\text{nil}) &\rightarrow \\ \text{conc}(2:\text{nil}, 1:\text{nil}) &\rightarrow \\ 2:\text{conc}(\text{nil}, 1:\text{nil}) &\rightarrow \\ 2:1:\text{nil} & \end{aligned}$$

237

Without extra requirements a term can have no normal form, or more than one normal

form

For instance, with respect to $f(x) \rightarrow f(x)$ the term $f(a)$ does not have a normal form

For instance, with respect to $f(f(x)) \rightarrow a$ the term $f(f(f(a)))$ has two normal forms a and $f(a)$

Now we investigate some nice properties forcing that every term has exactly one normal form

A TRS is called **weakly normalizing** (WN) if every term has at least one normal form

238

More nice properties:

- R is **terminating** (= strongly normalizing, SN):

no infinite sequence of terms t_1, t_2, t_3, \dots exists such that $t_i \rightarrow_R t_{i+1}$ for all i

- R is **confluent** (= Church-Rosser, CR):

if $t \rightarrow_R^* u$ and $t \rightarrow_R^* v$ then a term w exists satisfying $u \rightarrow_R^* w$ and $v \rightarrow_R^* w$

- R is **locally confluent** (= weak Church-Rosser, WCR):

if $t \rightarrow_R u$ and $t \rightarrow_R v$ then a term w exists satisfying $u \rightarrow_R^* w$ and $v \rightarrow_R^* w$

Here \rightarrow_R^* is the reflexive transitive closure of \rightarrow_R , i.e., $t \rightarrow_R^* u$ if and only if t can be rewritten to u in zero or more steps

239

Property

If a TRS is terminating, then every term has at least one normal form

Proof: rewriting as long as possible does not go on forever due to termination

So it ends in a normal form

The converse is not true: the TRS over the two constants a, b consisting of the two rules $a \rightarrow a$ and $a \rightarrow b$ is weakly normalizing since the two terms a and b both have b as a normal form, but it is not terminating due to

$$a \rightarrow a \rightarrow a \rightarrow a \rightarrow \dots$$

240

Property

If a TRS is confluent, then every term has at most one normal form

Proof: Assume t has two normal forms u, u'

Then by confluence there is a v such that $u \rightarrow_R^* v$ and $u' \rightarrow_R^* v$

Since u, u' are normal forms we have $u = v = u'$

241

Termination of term rewriting is undecidable, i.e., there is no algorithm that can decide for every finite TRS whether it is terminating

This can be proved by transforming an arbitrary Turing machine to a TRS and prove that the TRS is terminating if and only the Turing machine is halting from every initial configuration

A Turing machine (Q, S, δ) consists of

- a finite set Q of machine states
- a finite set S of tape symbols, including $\square \in S$ representing the blank symbol
- the transition function $\delta : Q \times S \rightarrow Q \times S \times \{L, R\}$

Here $\delta(q, s) = (q', s', L)$ means that if the machine is in state q and reads s , this s is replaced by s' , the machine shifts to the left, and the new machine state is q'

Similar for $\delta(q, s) = (q', s', R)$: then the machine shifts to the right

242

This Turing machine behaviour can be simulated by a TRS: for a Turing machine $M = (Q, S, \delta)$ we define a TRS $R(M)$ over $Q \cup S \cup \{b\}$ where

- symbols from Q are binary
- symbols from S are unary
- b is a constant representing an infinite sequence of blank symbols

The configuration with tape

$$\dots \square \square \square s'_m s'_{m-1} \dots s'_1 s_1 s_2 \dots s_{n-1} s_n \square \square \square \dots$$

in which the Turing machine is in state q and reads symbol s_1 is represented by the term

$$q(s'_1(s'_2(\dots(s'_m(b))\dots)), s_1(s_2(\dots(s_n(b))\dots)))$$

243

For the Turing machine $M = (Q, S, \delta)$ the TRS $R(M)$ is defined to consist of the rules

$$q(x, s(y)) \rightarrow q'(s'(x), y)$$

for all $(q, s) \in Q \times S$ with $\delta(q, s) = (q', s', R)$, and

$$q(t(x), s(y)) \rightarrow q'(x, t(s'(y)))$$

for all $(q, s) \in Q \times S$ with $\delta(q, s) = (q', s', L)$, for all $t \in S$

and some extra rules representing b to consist of blank symbols

Theorem

M halts on every configuration if and only if $R(M)$ is terminating

Consequence: TRS termination is undecidable

244

Although termination is undecidable, in many special cases termination of a TRS can be proved

General technique:

Find a weight function W from terms to natural numbers in such a way that $W(u) > W(v)$ for all terms u, v satisfying $u \rightarrow_R v$

If such a function W exists then R is terminating since an infinite rewriting sequence would give rise to an infinite decreasing sequence of natural numbers which does not exist

245

In our example

$$+(0, x) \rightarrow x$$

$$+(s(x), y) \rightarrow s(+ (x, y))$$

we find such a weight function W by defining inductively

$$W(0) = 1$$

$$W(s(t)) = W(t) + 1$$

$$W(t + u) = 2W(t) + W(u)$$

246

The general idea of weight functions is too general:

It allows arbitrary definitions of weight functions, and we have to prove that $W(t) > W(u)$ for **all** rewrite steps $t \rightarrow_R u$, while typically there are infinitely many of them

Now we work out a special case of this idea of weight functions in such a way that for finding a termination proof we only have to

- choose interpretations for the (finitely many) operation symbols rather than for all terms, and

- check $W(\ell) > W(r)$ for the (finitely many) rules $\ell \rightarrow r$ rather than for all rewrite steps

250

Example

For our TRS R consisting of the rules

$$+(0, x) \rightarrow x \quad +(s(x), y) \rightarrow s(+ (x, y))$$

we choose monotonic functions

$$[0] = 1, \quad [s](x) = x + 1$$

$$[+](x, y) = 2x + y$$

Now indeed for every $\alpha : \mathcal{X} \rightarrow \mathbf{N}$ we have

$$[+(0, x), \alpha] = 2 + \alpha(x) > \alpha(x) = [x, \alpha]$$

and

$$[+(s(x), y), \alpha] = 2(\alpha(x) + 1) + \alpha(y) >$$

$$(2\alpha(x) + \alpha(y)) + 1 = [s(+ (x, y)), \alpha]$$

proving termination

251

Example

For the TRS R consisting of the single rule

$$f(g(x)) \rightarrow g(g(f(x)))$$

we choose monotonic functions

$$[f](x) = 3x$$

$$[g](x) = x + 1$$

Now indeed for every $\alpha : \mathcal{X} \rightarrow \mathbf{N}$ we have

$$[f(g(x)), \alpha] = 3(\alpha(x) + 1) >$$

$$3\alpha(x) + 1 + 1 = [g(g(f(x))), \alpha]$$

proving termination

247

For every symbol f of arity n choose a **monotonic** function $[f] : \mathbf{N}^n \rightarrow \mathbf{N}$

Here **monotonic** means:

if for all $a_i, b_i \in \mathbf{N}$ for $i = 1, \dots, n$ with $a_i > b_i$ for some i and $a_j = b_j$ for all $j \neq i$ then

$$[f](a_1, \dots, a_n) > [f](b_1, \dots, b_n)$$

248

Examples

$$\lambda x \cdot x$$

$$\lambda x \cdot x + 1$$

$$\lambda x \cdot 2x$$

$$\lambda x, y \cdot x + y$$

$$\lambda x, y \cdot x + y + 1$$

$$\lambda x, y \cdot 2x + y$$

are monotonic

$$\lambda x \cdot 2$$

$$\lambda x, y \cdot x$$

are **not** monotonic

249

For a map $\alpha : \mathcal{X} \rightarrow \mathbf{N}$ the weight function $[\cdot, \alpha] : T \rightarrow \mathbf{N}$ is defined inductively by

$$[x, \alpha] = \alpha(x),$$

$$[f(t_1, \dots, t_n), \alpha] = [f]([t_1, \alpha], \dots, [t_n, \alpha])$$

Theorem

Let R be a TRS and let $[f]$ be chosen such that

- $[f]$ is monotonic for every symbol f , and
- $[\ell, \alpha] > [r, \alpha]$ for every $\alpha : \mathcal{X} \rightarrow \mathbf{N}$ and every rule $\ell \rightarrow r$ in R

Example

The single rule $f(x) \rightarrow g(f(x))$ is not terminating, but by choosing

$$[f](x) = x + 1, \quad [g](x) = 0$$

for every $\alpha : \mathcal{X} \rightarrow \mathbf{N}$ we have

$$[f(x), \alpha] = \alpha(x) + 1 > 0 = [g(f(x)), \alpha]$$

Where is the error?

$[g]$ is not monotonic

So monotonicity is essential

 253

Another technique: **lexicographic path order**

Choose an order $>$ on the set of function symbols

Theorem

If $\ell >_{lpo} r$ for all $\ell \rightarrow r$ in R , then R is terminating

Before this makes sense we have to define / characterize $>_{lpo}$

$$f(t_1, \dots, t_n) >_{lpo} u \iff$$

- $\exists i : t_i = u \vee t_i >_{lpo} u$, or
- $u = g(u_1, \dots, u_m)$ and
 - $\forall i : f(t_1, \dots, t_n) >_{lpo} u_i$ and either
 - $f > g$, or
 - $f = g$ and
 - $(t_1, \dots, t_n) >_{lpo}^{lex} (u_1, \dots, u_m)$

 254
Lemma

If s is a proper subterm of t , then $t >_{lpo}^{lex} s$

Easily follows from first bullet

Example

For the rule

$$+(0, x) \rightarrow x$$

we have $+(0, x) >_{lpo} x$ by this lemma

For the rule

$$+(s(x), y) \rightarrow s(+ (x, y))$$

we choose $+ > s$, then by the second item it remains to prove

$$+(s(x), y) >_{lpo} + (x, y)$$

Again using the second item we have to prove

- $+(s(x), y) >_{lpo} x$, follows from lemma
- $+(s(x), y) >_{lpo} y$, follows from lemma
- $(s(x), y) >_{lpo}^{lex} (x, y)$, follows from $s(x) >_{lpo} x$

 255

Hence $\ell >_{lpo} r$ for all rules $\ell \rightarrow r$, proving termination

Checking termination by lexicographic path order is easy to implement; do not choose $>$ in advance but collect requirements on $>$ during the process of proving $\ell >_{lpo} r$

Several more techniques for proving termination have been developed

Several tools have been developed by which termination of a TRS can be proved fully automatically: AProVE, TTT2

 256

Back to the other properties

Confluence is strictly stronger than local confluence:

$$a \rightarrow b, \quad b \rightarrow a, \quad a \rightarrow c, \quad b \rightarrow d$$

is locally confluent:

if $t \rightarrow_R u$ and $t \rightarrow_R v$ then either

- $t = a$, then choose $w = c$, or
- $t = b$, then choose $w = d$

In both cases we conclude $u \rightarrow_R^* w$ and $v \rightarrow_R^* w$

but not confluent:

for $t = a, u = c, v = d$ we have $t \rightarrow_R^* u$ and $t \rightarrow_R^* v$, but no w exists satisfying $u \rightarrow_R^* w$ and $v \rightarrow_R^* w$

257

Newman's lemma (1942):

Theorem

For terminating TRSs the properties confluence and local confluence are equivalent

For the proof of Newman's lemma we will use the principle of well-founded induction

Note that $\text{SN}(\rightarrow)$, $\text{CR}(\rightarrow)$ and $\text{WCR}(\rightarrow)$ all can be defined for arbitrary binary relations \rightarrow , in which general setting we will prove Newman's lemma

So $\text{SN}(\rightarrow)$ simply means the non-existence of an infinite sequence $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \dots$

We write \rightarrow^+ for the transitive closure of \rightarrow : one or more steps

258

Principle of well-founded induction

Theorem

Let $\text{SN}(\rightarrow)$ and

$$\forall t(\underbrace{\forall u(t \rightarrow^+ u \Rightarrow P(u))}_{\text{Induction Hypothesis}} \Rightarrow P(t))$$

Then $P(t)$ holds for all t

(think of $t \rightarrow^+ u$ as $t > u$ as in well-known induction)

Proof of this principle

Assume there exists t such that $\neg P(t)$

Then the induction hypothesis does not hold for this t , so $\neg \forall u(t \rightarrow^+ u \Rightarrow P(u))$, yielding u such that $t \rightarrow^+ u$ and $\neg P(u)$

Repeat the argument for u , yielding a v , and so on, so yielding an infinite sequence

$$t \rightarrow^+ u \rightarrow^+ v \rightarrow^+ \dots$$

contradicting $\text{SN}(\rightarrow)$ (End of proof)

259

Proof of Newman's Lemma

Assume $\text{SN}(\rightarrow)$ and $\text{WCR}(\rightarrow)$, we have to prove $\text{CR}(\rightarrow)$

So assume $t \rightarrow^* u$ and $t \rightarrow^* v$; we have to find w such that $u \rightarrow^* w$ and $v \rightarrow^* w$

We apply the principle of well-founded induction

If $t = u$ we may choose $w = v$

if $t = v$ we may choose $w = u$

In the remaining case we have $t \rightarrow^+ u$ and $t \rightarrow^+ v$

Write $t \rightarrow u_1 \rightarrow^* u$ and $t \rightarrow v_1 \rightarrow^* v$

260

Using WCR there exists w_1 such that $u_1 \rightarrow^* w_1$ and $v_1 \rightarrow^* w_1$

Using the induction hypothesis on u_1 there exists w_2 such that $w_1 \rightarrow^* w_2$ and $u \rightarrow^* w_2$

Now we have $v_1 \rightarrow^* w_2$ and $v_1 \rightarrow^* v$; using the induction hypothesis on v_1 there exists w such that $w_2 \rightarrow^* w$ and $v \rightarrow^* w$

$$\begin{array}{ccccc} t & \rightarrow & u_1 & \rightarrow^* & u \\ \downarrow & \text{WCR} & \downarrow^* & \text{IH} & \downarrow^* \\ v_1 & \rightarrow^* & w_1 & \rightarrow^* & w_2 \\ \downarrow^* & & \text{IH} & & \downarrow^* \\ v & & \rightarrow^* & & w \end{array}$$

Since $u \rightarrow^* w_2$ we have $u \rightarrow^* w$, and we are done

Both confluence and local confluence are undecidable properties

However, for terminating TRSs there is a simple decision procedure for local confluence, and hence for confluence too

Idea:

analyze overlapping patterns in left hand sides of the rules, yielding **critical pairs**

In our example for addition of natural numbers there is no overlap, hence it is locally confluent

Since we observed it is terminating, by Newman's lemma it is confluent

Definition of critical pairs

Let $\ell_1 \rightarrow r_1$ and $\ell_2 \rightarrow r_2$ be two (possibly equal) rewrite rules

Rename variables such that ℓ_1, ℓ_2 have no variables in common

Let t be a subterm of ℓ_2 , possibly equal to ℓ_2 ; t is not a variable

Assume t, ℓ_1 unify, with mgu σ : $t\sigma = \ell_1\sigma$

Now $\ell_2\sigma$ can be rewritten in two ways:

- to $r_2\sigma$, and
- to a term u obtained by replacing its subterm $t\sigma = \ell_1\sigma$ to $r_1\sigma$

In the above situation the pair $[u, r_2\sigma]$ is called a **critical pair**

Example

Assume we have rules for arithmetic including

$$\begin{aligned} x - x &\rightarrow 0 \\ s(x) - y &\rightarrow s(x - y) \end{aligned}$$

Then $s(x) - s(x)$ can be rewritten in two ways:

- to 0 by the first rule
- to $s(x - s(x))$ by the second rule

Now $[0, s(x - s(x))]$ is a **critical pair**

More precisely, in the above notation we choose

- $\ell_1 \rightarrow r_1$ to be the rule $z - z \rightarrow 0$
- $\ell_2 \rightarrow r_2$ to be the rule $s(x) - y \rightarrow s(x - y)$
- $t = \ell_2 = s(x) - y$

Indeed t, ℓ_1 unify, with mgu σ :

$$\sigma(x) = x, \quad \sigma(y) = \sigma(z) = s(x)$$

Example

Let R consist of the single rule

$$f(f(x)) \rightarrow g(x)$$

By choosing

- $\ell_1 \rightarrow r_1$ to be the rule $f(f(x)) \rightarrow g(x)$
- $\ell_2 \rightarrow r_2$ to be the rule $f(f(y)) \rightarrow g(y)$
- $t = f(y)$

we see that t, ℓ_1 unify, with mgu σ :

$$\sigma(x) = x, \quad \sigma(y) = f(x)$$

yielding the critical pair $[f(g(x)), g(f(x))]$

A critical pair $[t, u]$ is said to **converge** if there is a term v such that $t \rightarrow_R^* v$ and $u \rightarrow_R^* v$

Theorem

A TRS R is locally confluent if and only if all critical pairs converge

Example

The single rewrite rule $f(f(x)) \rightarrow g(x)$ is not locally confluent, so neither confluent, since for its critical pair $[f(g(x)), g(f(x))]$ no term v exists such that

$$f(g(x)) \rightarrow_R^* v \quad \text{and} \quad g(f(x)) \rightarrow_R^* v$$

This is immediate from the observation that both $f(g(x))$ and $g(f(x))$ are normal forms

266

For a term t and a TRS R define

$$S(t) = \{v \mid t \rightarrow_R^* v\}$$

If R is finite and terminating then $S(t)$ is finite and computable

Using the theorem, for a finite terminating TRS R indeed we have an algorithm to decide whether $\text{WCR}(R)$ holds:

- Compute all critical pairs $[t, u]$

They are found by unification of left hand sides with subterms of left hand sides: there are finitely many of them

- For all critical pairs $[t, u]$ compute

$$S(t) \cap S(u)$$

- If one of these sets is empty then $\text{WCR}(R)$ does not hold
- If all of these sets are non-empty then $\text{WCR}(R)$ holds

267

A TRS is said to have **no overlap** if there are only **trivial** critical pairs, i.e., the critical

pairs obtained by unifying a left hand side with itself

A trivial critical pair always converges since it is of the shape $[t, t]$

As a consequence, every TRS having no overlap is locally confluent

It is not the case that every TRS having no overlap is confluent:

$$\begin{aligned} d(x, x) &\rightarrow b \\ c(x) &\rightarrow d(x, c(x)) \\ a &\rightarrow c(a) \end{aligned}$$

has no overlap but is not confluent:

$$\begin{aligned} c(a) &\rightarrow_R d(a, c(a)) \rightarrow_R d(c(a), c(a)) \rightarrow_R b \\ c(a) &\rightarrow_R c(c(a)) \rightarrow_R^+ c(b) \end{aligned}$$

while $[b, c(b)]$ does not converge

268

Write \leftrightarrow_R^* for the reflexive symmetric transitive closure of \rightarrow_R , i.e., $t \leftrightarrow_R^* u$ holds if and only if terms t_1, \dots, t_n exist for $n \geq 1$ such that

- $t_1 = t$
- $t_n = u$
- For every $i = 1, \dots, n - 1$ either $t_i \rightarrow_R t_{i+1}$ or $t_{i+1} \rightarrow_R t_i$ holds

A general question is: given R, t, u , does $t \leftrightarrow_R^* u$ hold?

This is called the **word problem**

In general the word problem is undecidable

However, in case R is terminating and confluent then the word problem is decidable and admits a simple algorithm

269

A terminating and confluent TRS is called **complete**

Now we give a decision procedure for the word problem for complete TRSs

Rewriting a term t in a terminating TRS as long as possible will always end in a normal form; the result is called a **normal form of t**

Theorem

If R is a complete TRS and t', u' are normal forms of t, u , then $t \leftrightarrow_R^* u$ if and only if $t' = u'$

270

For the proof we need a lemma that is easily proved by induction on the length of the path corresponding to $t \leftrightarrow_R^* u$:

Lemma:

If R is confluent and $t \leftrightarrow_R^* u$ then a term v exists such that $t \rightarrow_R^* v$ and $u \rightarrow_R^* v$

Proof of the theorem:

If $t' = u'$ then $t \rightarrow_R^* t' = u' \leftarrow_R^* u$, hence $t \leftrightarrow_R^* u$

Conversely assume $t \leftrightarrow_R^* u$

Then $t' \leftarrow_R^* t \leftrightarrow_R^* u \rightarrow_R^* u'$, hence $t' \leftrightarrow_R^* u'$

According the lemma a term v exists such that $t' \rightarrow_R^* v$ and $u' \rightarrow_R^* v$

Since t', u' are normal forms we have $t' = v = u'$ End of proof

271

The relation \leftrightarrow_R^* is an equivalence relation, and in a complete TRS the normal form is a unique representation for the corresponding equivalence class

According to the theorem there is a very simple decision procedure for the word problem for complete TRSs:

In order to decide whether $t \leftrightarrow_R^* u$, rewrite

- t to a normal form t' , en

- u to a normal form u' ,

Then $t \leftrightarrow_R^* u$ if and only if $t' = u'$

272

Example:

R consists of the rule $s(s(s(x))) \rightarrow x$

Does $s^{17}(0) \leftrightarrow_R^* s^{10}(0)$ hold?

We can establish fully automatically that this is not:

- check that R is terminating
- check that R is locally confluent
- compute the normal form $s(s(0))$ of $s^{17}(0)$
- compute the normal form $s(0)$ of $s^{10}(0)$
- these are different, hence the answer is **No**

273

Often a TRS R is not complete, but a complete TRS R' satisfying

$$\leftrightarrow_{R'}^* = \leftrightarrow_R^*$$

can be found in a systematic way

Finding such a complete TRS is called

(Knuth-Bendix) completion

The new complete TRS can be used for the word problem and unique representation of the original TRS

Often the original TRS is only a set of equations

274

Idea of Knuth-Bendix completion

Fix a well-founded order $>$ on terms, i.e., $\text{SN}(>)$, that has some closedness properties:

- if $t > u$ then $t\sigma > u\sigma$ for every substitution σ
- if $t > u$ then $f(\dots, t, \dots) > f(\dots, u, \dots)$ for every symbol f and every position for t

Such an order is called a **reduction order**, and has the property:

If $\ell > r$ for every rule $\ell \rightarrow r$ in R , then $\text{SN}(R)$

A typical example of a reduction order is a lexicographic path order

275

Starting with a set E of equations and an empty set R of rewrite rules, repeat the following until E is empty:

Remove an equation $t = u$ from E , and

- add $t \rightarrow u$ to R if $t > u$
- add $u \rightarrow t$ to R if $u > t$
- give up otherwise

After adding any new rule $\ell \rightarrow r$ to R compute all critical pairs between this new rule and existing rules of R , or between the new rule and itself

For every such critical pair $[t, u]$

- R -rewrite t to normal form t'
- R -rewrite u to normal form u'
- if $t' \neq u'$, then add $t' = u'$ as an equation to the set E

276

What can happen in this Knuth-Bendix procedure?

- it fails due to an equation $t = u$ in E for which neither $t > u$ nor $u > t$ holds
- it fails since the procedure goes on forever: E gets larger and is never empty
- it ends with E being empty

In the last case we really have success: then

- R is terminating since it only contains rule $\ell \rightarrow r$ satisfying $\ell > r$
- R is locally confluent since all critical pairs converge, so R is complete
- Convertibility \leftrightarrow_R^* of the resulting R is equivalent to convertibility of the original E since in the whole procedure $\leftrightarrow_{R \cup E}^*$ remains invariant

277

Example:

Let E consist of the single equation

$$f(f(x)) = g(x)$$

Choose the lexicographic path order defined by $f > g$

Since

$$f(f(x)) >_{lpo} g(x)$$

we add the rule $f(f(x)) \rightarrow g(x)$ to the empty TRS R

Now the critical pair $[f(g(x)), g(f(x))]$ gives rise to the new equation $f(g(x)) = g(f(x))$ in E

278

Since

$$f(g(x)) >_{lpo} g(f(x))$$

we add the rule $f(g(x)) \rightarrow g(f(x))$ to the TRS R

Together with the older rule $f(f(x)) \rightarrow g(x)$ we get the critical pair $[f(g(f(x))), g(g(x))]$

Since $g(g(x))$ is a normal form and

$$f(g(f(x))) \rightarrow_R g(f(f(x))) \rightarrow_R g(g(x))$$

no new equation is added to E , and E is empty

So we end up in the complete TRS R consisting of the two rules

$$f(f(x)) \rightarrow g(x), \quad f(g(x)) \rightarrow g(f(x))$$

having the same convertibility relation as the original equation $f(f(x)) = g(x)$

279

Example:

For decision trees we consider the set E of equations

$$\begin{aligned} p(x, x) &= x \\ p(q(x, y), q(z, w)) &= q(p(x, z), p(y, w)) \\ p(p(x, y), z) &= p(x, z) \\ p(x, p(y, z)) &= p(x, z) \end{aligned}$$

where p, q runs over all boolean variables

It can be proved that two decision trees represent the same boolean function if and only if they are equivalent with respect to \leftrightarrow_E^*

As a TRS E is not terminating and not confluent

Choose any order $>$ on the boolean variables

Completion yields the TRS R consisting of the rules

280

$$\begin{aligned} p(x, x) &\rightarrow x && \text{for all } p \\ p(p(x, y), z) &\rightarrow p(x, z) && \text{for all } p \\ p(x, p(y, z)) &\rightarrow p(x, z) && \text{for all } p \\ p(q(x, y), z) &\rightarrow q(p(x, z), p(y, z)) && \text{for } p > q \\ p(x, q(y, z)) &\rightarrow q(p(x, y), p(x, z)) && \text{for } p > q \end{aligned}$$

Now rewriting to normal form in R yields the unique representation as an ordered decision tree; unicity is a consequence of completeness of R'

Storing by sharing common subterms yields the ROBDD

Arbitrary formulas are easily transformed to (unordered) BDDs representing the same boolean function; R -rewriting now yields an alternative method for computing ROBDDs

Unfortunately this is not very efficient

281

Overview of the course

- Proposition logic
 - SAT by semantic tableaux
 - SAT by resolution: DP and DPLL and backjump
 - Tseitin transformation
 - (RO)BDDs
 - Extension to SMT, pseudo-boolean constraints

- Predicate logic
 - Semantic tableaux
 - Resolution, unification
 - Prenex normal form, Skolemization
 - Prolog

- Equational reasoning by term rewriting
 - termination by monotonic interpretation or lexicographic path order
 - (local) confluence by critical pair analysis
 - Knuth-Bendix completion