Version September, 2021

1

Automated Reasoning, 2IMF25 / IMC009

prof dr Hans Zantema

TU/e, Metaforum room 6.078

RU, Mercator 1, 1.18 (only on Tuesday)

email: h.zantema@tue.nl

Information:

www.win.tue.nl/~hzantema/ar.html
(for TU Eindhoven)

www.win.tue.nl/~hzantema/arn.html
(for RU Nijmegen)

2

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

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein: Introduction to Algorithms
MIT Press, 2009, ISBN 978-0-262-53305-8, only Chapter 29

3

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
- Deadlines for practical assignment:

  September 29, 2021, and October 27, 2021, for TU/e students (Eindhoven)

  November 2, 2021, and January 5, 2022, for RU students (Nijmegen)

4

In 2021 the course is offered in a hybrid format: partly on-campus and partly online

MOOCs (Massive Open Online Course) available for free at https://www.coursera.org/learn/automated-reasoning-sat and https://www.coursera.org/learn/automated-reasoning-symbolic-model-checking

This roughly covers first eight lectures

When appropriate we will refer to these MOOCs

5

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

6

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 than 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

---

7

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

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

8

name	meaning	opposite
A	good-natured	grumpy
B	tenured	
C	professor	
D	dynamic	phlegmatic
	$\cdots$	

The claim

> Good-natured tenured professors are dynamic

yields:

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

9

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 $\neg(M \wedge F)$

10

Hence we want to prove automatically that

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

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

How can this be treated?

11

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 yields *true* for some valuations, so is not equivalent to *false*

12

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

13

Example: verification of a microprocessor

Here we want that

$\neg$(specified behavior $\leftrightarrow$ 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

14

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 $i = 1, \ldots, n$ and $j = 1, \ldots, n+1$

Define

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

$$R_n = \bigwedge_{i=1,\ldots,n, 1 \le j < k \le n+1} (\neg P_{ij} \vee \neg P_{ik})$$

$$PF_n = C_n \wedge R_n$$

15

Here

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

is used as an abbreviation for

$$A_1 \wedge A_2 \wedge A_3 \wedge \cdots \wedge A_n$$

and

$$\bigwedge_{P(i)} A_i$$

denotes the conjunction of A_i for all i that satisfy $P(i)$

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

---

16

Put the variables in a matrix as follows

$$\begin{array}{cccc} P_{11} & P_{12} & \cdots & P_{1,n+1} \\ P_{21} & P_{22} & \cdots & P_{2,n+1} \\ \vdots & \vdots & & \vdots \\ P_{n1} & P_{n2} & \cdots & P_{n,n+1} \end{array}$$

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

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

17

$$\begin{array}{cccc} P_{11} & P_{12} & \cdots & P_{1,n+1} \\ P_{21} & P_{22} & \cdots & P_{2,n+1} \\ \vdots & \vdots & & \vdots \\ P_{n1} & P_{n2} & \cdots & P_{n,n+1} \end{array}$$

$$R_n = \bigwedge_{i=1,\ldots,n, 1 \le j < k \le n+1} (\neg P_{ij} \vee \neg P_{ik})$$

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

for any two P's in the same 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

---

18

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 $\bigvee_{i=1}^{n} P_{ij}$ and $\neg P_{ij} \vee \neg P_{ik}$

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

---

19

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 type of problem different from satisfiability for which automated reasoning is desired too

---

20

Consider 12 processes over boolean variables $A, \ldots, J$:

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

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*

21

It is possible to find an invariant proving this claim

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: **model checking**

- finite state space
- simple description of the set of initial states
- simple description of all possible state transitions
- some property has to be proved, typically expressed in some modal logic like CTL

22

Examples:

- Program verification:
  - state transition: the program steps, may be non-deterministic / distributed
  - property to be proved: desired end state, or no deadlock
- Railways:
  - state transition: if a semaphore shows green then a train may enter the corresponding track
  - property to be proved: never two trains in the same track, so trains do not collide
- Solving puzzles
- ...

23

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, exploited in tools like `Z3` and `Yices`

Basic technique for symbolic model checking: B(inary) D(ecision) D(iagrams), exploited in tools like `NuSMV`

24

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

In this course we will also consider other forms / extensions

We will consider **SMT**: satisfiability modulo theories, in particular, the theory of linear inequalities, in which apart from boolean variables also integer or real values may occur, and linear inequalities like $3a + 4b - c \leq 17$

25

Other forms / extensions:

- 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

  We only present some of CTL: Computation Tree Logic; more on this in the course **Program Verification Techniques**

26

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

27

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 $n \geq n_0$

28

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

29

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

30

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

---

31

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

---

32

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?

33

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

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

So a_i are boolean variables; the following slides describe how to express addition, subtraction and multiplication of binary numbers in these boolean variables

The same is found in the last two lectures of Week 1 of the MOOC on satisfiability, in slightly more detail

---

34

Addition

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

We need **carries** $c_0, c_1, \ldots, c_n$

Example: $7 + 21 = 28$:

$c \rightarrow$	0	0	1	1	1	0
$a = 7 \rightarrow$		0	0	1	1	1
$b = 21 \rightarrow$		1	0	1	0	1
$d = 28 \rightarrow$		1	1	1	0	0

35

Requirements by which this is a correct computation:

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

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

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

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

36

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 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

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

37

Multiplication

A fast way to do multiplication, closely related to 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 $r = b * a$

As a, b are numbers, represented as boolean vectors, we will write $\vec{a} = (a_1, \ldots, a_n)$, and so on

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

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

38

Introduce extra boolean variables r_{ij}, s_{ij} for $i = 0, \ldots, n$ and $j = 1, \ldots, n$

where $\vec{r_i} = (r_{i1}, \ldots, r_{in})$ represents the value of r after i steps, and

$\vec{s_i} = (s_{i1}, \ldots, s_{in})$ represents the value of $s = 2r$ after i steps

Then $\vec{r}_n$ will represent the result, the requirement

$$\vec{a} * \vec{b} = \vec{r}_n$$

is described by the formula

39

$$\mathrm{mul}(\vec{a}, \vec{b}, \vec{r}_n) \ = \ \bigwedge_{j=1}^{n} \neg r_{0j} \ \wedge$$

$$\bigwedge_{i=0}^{n-1} [\mathrm{dup}(\vec{r}_i, \vec{s}_i) \wedge (b_{i+1} \rightarrow \mathrm{plus}(\vec{a}, \vec{s}_i, \vec{r}_{i+1})) \wedge (\neg b_{i+1} \rightarrow \bigwedge_{j=1}^{n} (s_{ij} \leftrightarrow r_{i+1,j}))]$$

40

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 $\iff$ fac(r) is unsatisfiable

If satisfiable, then a, b represent factors

fac(1234567891) is unsatisfiable, so 1234567891 is prime

Found by `minisat` or `Z3` within 1 minute

fac(1234567897) is satisfiable, yielding

$$1234567897 = 1241 \times 994817$$

found by `minisat` or `Z3` within 1 second

41

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 $\cdots$

  introduce $m+1$ copies $a_0, \ldots, a_m$ 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 $\neq a$

42

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

$$P_0 \;\wedge\; \neg Q_m$$

to the formula expressing the semantics of the program and prove that it is unsatisfiable

---

43

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 jth iteration:

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

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

---

44

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

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

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

$$\neg\text{mul}(\vec{m}, \vec{k}, \vec{a}_m)$$

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

45

if b then S_1 else S_2

in step i can be expressed as

$$(b_{i-1} \rightarrow F_1) \wedge (\neg b_{i-1} \rightarrow F_2)$$

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

In this way verification of imperative programs having a fixed number of steps can be expressed in SAT

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

---

46

The unfolding of a fixed number of steps to a SAT problem is the basis of **bounded model checking**

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

$$a < 3 * b + c$$

directly, supported by tools like `Z3`

Please also watch the last lecture of Week 2 of the MOOC on satisfiability, on bounded model checking

---

47

Resolution

This is a 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

A **literal** is either a variable or the negation of a variable

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	$\bot$	$(7, 9, r)$

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 $\cdots$

- 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 $q \vee \neg q \vee \cdots$ 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 proof 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 is done in the **second lecture of Week 3 of the MOOC** on satisfiability

---

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

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

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:

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

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

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

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

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:

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

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:

$$t, \quad \neg t$$

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

---

64

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

---

65

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 $\cdots$

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

---

66

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 $q \vee \neg q$

67

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

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

68

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

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 (and often does in practice $\cdots$)
- 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 $\qquad r = \mathit{true}$

$\downarrow r$

$\{\}$

find value for the last variable that was removed

72

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

$\downarrow p$

$q \vee r,\ \neg q,\ \neg q \vee r$ $\qquad q = \mathit{false}$

$\downarrow q$ $\qquad \uparrow$

r $\qquad r = \mathit{true}$

$\downarrow r$

$\{\}$

evaluate in formula, find next value

73

$p \vee q,\ \neg p \vee \neg q,\ p \vee \neg q,\ \neg p \vee r$ $\qquad p = \mathit{true}$

$\downarrow p$ $\qquad \uparrow$

$q \vee r,\ \neg q,\ \neg q \vee r$ $\qquad q = \mathit{false}$

$\downarrow q$ $\qquad \uparrow$

r $\qquad r = \mathit{true}$

$\downarrow r$

$\{\}$

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):
  $X :=$ unit-resol(X)
  if $X = \emptyset$ then return(satisfiable)
  if $\bot \notin X$ 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)
$\bot$	$\bot$

Both branches yield $\bot$, so original CNF is unsatisfiable

77

Example:

Consider the CNF consisting of the following eight clauses

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

No unit resolution possible: choose variable p

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

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

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; it is still the basis of current strong SAT solvers

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 $\bot$, having the same size

This is elaborated in the **4th lecture of Week 3 of the MOOC** on satisfiability

---

79

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

Two derivations from $V \wedge p$ to $\bot$ and from $V \wedge \neg p$ to $\bot$ then transform to derivations from V to both p and $\neg p$, yielding $\bot$ 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	$\bot$	$(2, 10)$

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

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

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	$\bot$	$(11, 13)$

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 mimick 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:

$$M \Longrightarrow M\ell$$

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

Decide:

$$M \Longrightarrow M\ell^d$$

if ℓ is undefined in M

Backtrack:

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

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

Fail:

$$M \Longrightarrow \text{fail}$$

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 (Minisat, Z3, 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:

$\emptyset$	$\Longrightarrow$	Decide
p^d	$\Longrightarrow$	UnitPropagate, clause 3
$p^d\ r$	$\Longrightarrow$	Backtrack, clause 4
$\neg p$	$\Longrightarrow$	UnitPropagate, clause 1
$\neg p\ q$	$\Longrightarrow$	Fail, clause 2

hence unsatisfiable

89

Optimizations

Optimizations are presented in the **last lecture of Week 3 of the MOOC** on satisfiability

---

90

Example:

Let the CNF contain the four clauses

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

$\emptyset$	$\Longrightarrow$	Decide
p^d	$\Longrightarrow$	UnitPropagate, clause 1
$p^d\ q$	$\Longrightarrow$	Decide
$p^d\ q\ r^d$	$\Longrightarrow$	UnitPropagate, clause 2
$p^d\ q\ r^d\ s$	$\Longrightarrow$	Decide
$p^d\ q\ r^d\ s\ t^d$	$\Longrightarrow$	UnitPropagate, clause 4
$p^d\ q\ r^d\ s\ t^d\ u$	$\Longrightarrow$	Backtrack, clause 3 ????
$p^d\ q\ r^d\ s\ \neg t$	$\Longrightarrow$	$\cdots$

91

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

---

92

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:

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

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

93

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

---

94

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'

95

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

---

96

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

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

Combined technology is often called **Conflict-Driven Clause Learning**

Good heuristics for choosing literals for Decide remain crucial

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

Learning clauses and then restart may cause better choice for decision literals

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

---

97

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

(often optional)

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

Example:

Calling

`yices-sat -m test.d` (or any other SAT solver)

on the file `test.d` containing

```
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 0
```

indeed showing that a satisfying assignment is obtained by making 2 true and 1 and 3 false

---

99

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 to establish 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

---

100

Example:

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

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)

101

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

$\Longrightarrow$ unacceptably large

See also the first lecture of **Week 4 of the MOOC** on satisfiability

---

102

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

This will result in the **Tseitin transformation**, see also the second lecture of **Week 4 of the MOOC** on satisfiability

---

103

For every formule 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:

$$\begin{aligned} cnf(p \leftrightarrow \neg q) &= (p \vee q) \\ &\quad \wedge(\neg p \vee \neg q) \end{aligned}$$

$$\begin{aligned} cnf(p \leftrightarrow (q \wedge r)) &= (p \vee \neg q \vee \neg r) \\ &\quad \wedge(\neg p \vee q) \\ &\quad \wedge(\neg p \vee r) \end{aligned}$$

$$\begin{aligned} cnf(p \leftrightarrow (q \vee r)) &= (\neg p \vee q \vee r) \\ &\quad \wedge(p \vee \neg q) \\ &\quad \wedge(p \vee \neg r) \end{aligned}$$

$$\begin{aligned} cnf(p \leftrightarrow (q \leftrightarrow r)) &= (p \vee q \vee r) \\ &\quad \wedge(p \vee \neg q \vee \neg r) \\ &\quad \wedge(\neg p \vee q \vee \neg r) \\ &\quad \wedge(\neg p \vee \neg q \vee r) \end{aligned}$$

104

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

105

Example: $A : \underbrace{(\underbrace{\neg s \wedge p}_{B}) \leftrightarrow (\underbrace{(\underbrace{q \rightarrow r}_{D}) \vee \neg p}_{C})}$

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

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

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

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

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

106

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

107

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 $T(A)$

108

There is no need to use a separate tool for transforming an arbitrary propositional formula A to $T(A)$ in dimacs format: several modern SAT solvers like `Z3` 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 `z3 test` for the file `test` containing

109

```
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(assert
(and
(iff A (and D B))
(implies C B)
(not (or A B (not D)))
(or (and (not A) C) D)
))
(check-sat)
(get-model)
```

110

yields

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

indeed describing a satisfying assignment

`declare-const` declares a variable

`assert` gives the formula

`check-sat` checks for satisfiability

`get-model` gives the model if satisfiable

111

Always **prefix notation**: first '`(`', then name of operator, then the operands, then '`)`'

Note that `and` and `or` may have any number of arguments

One can give the specification of a logic

The simplest logic is `:logic QF_UF`: quantifier free and uninterpreted functions, so not using integers or reals

In `Z3` the logic may be left implicit

112

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

113

Conclusions (continued)

- Modern CNF based SAT solvers, like
  - Minisat
  - Lingeling
  - Yices
  - Z3

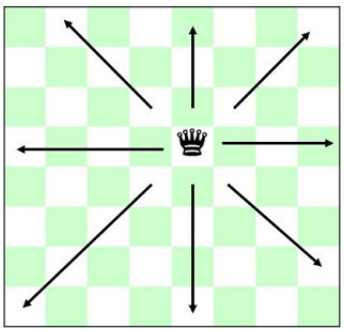
  essentially use DPLL, extended by Backjump, Learn, Forget and Restart:

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

---

114

Eight Queens Problem



Can we put 8 queens on the chess board in such a way that no two may hit each other?

See third lecture of **Week 1 of MOOC** on satisfiability

115

As usual in SAT/SMT: don't think about how to solve it, but only **specify** the problem

Here it can be done in pure SAT: only boolean variables, no numbers, no inequalities

For every position (i, j) on the board: boolean variable p_{ij} expresses whether there is a queen or not

---

116

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

117

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

p_{ij} and $p_{i'j'}$ in same row: $i = i'$

118

Row i:

$$p_{i1}, p_{i2}, p_{i3}, p_{i4}, p_{i5}, p_{i6}, p_{i7}, p_{i8}$$

At **least** one queen on row i:

$$p_{i1} \vee p_{i2} \vee p_{i3} \vee p_{i4} \vee p_{i5} \vee p_{i6} \vee p_{i7} \vee p_{i8}$$

Shorthand:

$$\bigvee_{j=1}^{8} p_{ij}$$

119

Row i:

$$p_{i1}, p_{i2}, p_{i3}, p_{i4}, p_{i5}, p_{i6}, p_{i7}, p_{i8}$$

At **most** one queen on row i?

That is: for every $j < k$ not both p_{ij} and p_{ik} are true

So $\neg p_{ij} \vee \neg p_{ik}$ for all $j < k$

$$\bigwedge_{0<j<k\leq 8} (\neg p_{ij} \vee \neg p_{ik})$$

120

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

Column requirements for p_{ij}: same as for rows with i, j swapped

---

121

Requirements until now:

At least one queen on every row:

$$\bigwedge_{i=1}^{8} \bigvee_{j=1}^{8} p_{ij}$$

At most one queen on every row:

$$\bigwedge_{i=1}^{8} \bigwedge_{0<j<k\leq 8} (\neg p_{ij} \vee \neg p_{ik})$$

122

And similar for the columns:

At least one queen on every column:

$$\bigwedge_{j=1}^{8} \bigvee_{i=1}^{8} p_{ij}$$

At most one queen on every column:

$$\bigwedge_{j=1}^{8} \bigwedge_{0<i<k\leq 8} (\neg p_{ij} \vee \neg p_{kj})$$

123

Remains to express:

At most one queen on every diagonal

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

p_{ij} and $p_{i'j'}$ on such a diagonal

$$\iff$$

$$i + j = i' + j'$$

124

Diagonal in other direction:

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

p_{ij} and $p_{i'j'}$ on such a diagonal

$$\iff$$

$$i - j = i' - j'$$

125

So for all i, j, i', j' with $(i, j) \neq (i'j')$ satisfying $i + j = i' + j'$ or $i - j = i' - j'$:

$$\neg p_{ij} \vee \neg p_{i'j'}$$

stating that on (i, j) and $(i'j')$ being two distinct positions on a diagonal, no two queens are allowed

We may restrict to $i < i'$, yielding

$$\bigwedge_{0<i<i'\leq 8} \Big(\bigwedge_{j,j' : i+j=i'+j' \vee i-j=i'-j'} \neg p_{ij} \vee \neg p_{i'j'} \Big)$$

126

Total formula:

$$\bigwedge_{i=1}^{8} \bigvee_{j=1}^{8} p_{ij} \wedge$$

$$\bigwedge_{i=1}^{8} \bigwedge_{0<j<k\leq 8} (\neg p_{ij} \vee \neg p_{ik}) \wedge$$

$$\bigwedge_{j=1}^{8} \bigvee_{i=1}^{8} p_{ij} \wedge$$

$$\bigwedge_{j=1}^{8} \bigwedge_{0<i<k\leq 8} (\neg p_{ij} \vee \neg p_{kj}) \wedge$$

$$\bigwedge_{0<i<i'\leq 8} \Big(\bigwedge_{j,j':i+j=i'+j' \vee i-j=i'-j'} \neg p_{ij} \vee \neg p_{i'j'} \Big)$$

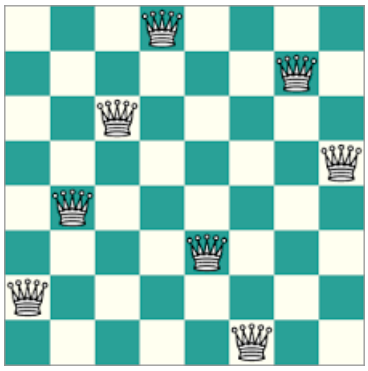
127

The resulting formula

- looks complicated, but is easily generated in SAT/SMT syntax by some *for* loops
- consists of 740 requirements
- is easily solved by current SAT solver; resulting true values easily give the queen positions

---

128



129

Trick to find all 92 solutions:

Add negation of found solution to formula, and repeat this until formula is unsatisfiable

Generalizes to n queens on $n \times n$ board, e.g., for $n = 100$ Z3 finds a satisfying assignment of the 50Mb formula within 10 seconds

Same approach applicable to extending a partially filled board, which is an NP-complete problem

---

130

To conclude: we expressed a chess board problem in a pure SAT problem

Generate this formula in the appropriate syntax by a small program

Then a SAT solver immediately finds a solution

131

Amazing example:

Prove termination of the system of three rules

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

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

132

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 $x_1, \ldots, x_n$, 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**

133

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

Here we will present the powerful **Simplex method** for linear optimization (following chapter 29 of the algorithms book by Cormen, Leiserson, Rivest and Stein)

It is also presented in the last four lectures of **Week 4 of the MOOC** on satisfiability

This can be combined with techniques for SAT solving, to solve propositional formulas over linear inequalities:

Satisfiability Modulo Theories (SMT)

134

Dealing with **linear inequalities**

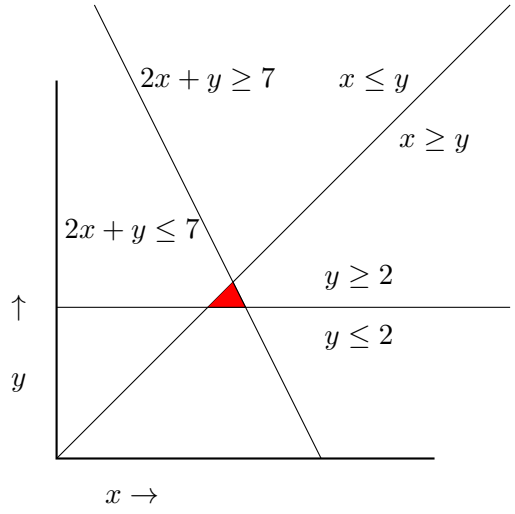
Do real numbers x, y exist such that

$$x \geq y$$

$$y \geq 2$$

$$2x + y \leq 7$$

135



136

So indeed the **red** part describes the values x, y satisfying

$$x \geq y$$

$$y \geq 2$$

$$2x + y \leq 7$$

For > 2 variables no such pictures: we want to do this for thousands of inequalities / variables

To do so we present the **Simplex method** for

linear optimization = linear programming

Not only determines existence of solution, but finds an **optimal** one, that is, one for which a given linear expression has the highest possible value

---

137

The Simplex method

Among all real values $x_1, \ldots, x_n \geq 0$ we want to find the **maximal** value of a linear **goal function**

$$f(x_1, \ldots, x_n) = v + c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

satisfying k linear constraints

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i$$

for $i = 1, 2, \ldots, k$

Here v, a_{ij}, c_i and b_i are arbitrary given real values, satisfying $b_i \geq 0$ for $i = 1, 2, \ldots, k$

Choosing $x_i = 0$ for all i satisfies all constraints and yields the value v for the goal function, but probably this is not the maximal value

---

138

More general linear optimization problems can be transformed to this format, for instance

- in an inequality '$\geq$' multiply both sides by -1:

  $x_1 - 2x_2 + 3x_3 \geq -5 \quad \equiv \quad -x_1 + 2x_2 - 3x_3 \leq 5$

- if one wants to minimize, multiply goal function by -1

- if a variable x runs over all reals (positive and negative), replace it by $x_1 - x_2$ for fresh variables x_1, x_2 satisfying $x_i \geq 0$ for $i = 1, 2$

- later we will see how to deal with the situation if not $b_i \geq 0$

139

Slack form

For every linear inequality

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i$$

a fresh variable $y_i \geq 0$ is introduced

The linear inequality is replaced by the equality

$$y_i = b_i - a_{i1}x_1 - a_{i2}x_2 - \cdots - a_{in}x_n$$

Together with $y_i \geq 0$ this is equivalent to the original inequality

This format with equalities is called the **slack form**

140

Some terminology on a slack form with equations

$$y_i = b_i - a_{i1}x_1 - a_{i2}x_2 - \cdots - a_{in}x_n$$

for $i = 1, 2, \ldots, k$

- The solution $y_i = b_i$ for $i = 1, 2, \ldots, k$ and $x_j = 0$ for $j = 1, 2, \ldots, n$ is called the **basic solution**

- The variables y_i for $i = 1, 2, \ldots, k$ are called **basic**

- The variables x_j for $j = 1, 2, \ldots, n$ are called **non-basic**

- The simplex algorithm consists of a repetition of **pivot**s, in which

- a pivot chooses a basic variable and a non-basic variable, swaps the roles of these two variables, and brings the result in a set of equations in slack form that is equivalent to the original one

We will illustrate this on an example

141

Maximize $z = 3 + x_1 + x_3$ satisfying the constraints

$$\begin{aligned} -x_1 + x_2 - x_3 &\leq 2 \\ x_1 + x_3 &\leq 3 \\ 2x_1 - x_2 &\leq 4 \end{aligned}$$

Slack form:

$$\begin{array}{lcrrrr} y_1 & = & 2 & +x_1 & -x_2 & +x_3 \\ y_2 & = & 3 & -x_1 & & -x_3 \\ y_3 & = & 4 & -2x_1 & +x_2 & \end{array}$$

Goal: maximize $z = 3 + x_1 + x_3$

Observation: if we increase x_1, and the other non-basic variables remain 0, then the goal function z will increase

---

142

We want to increase x_1 as much as possible, keeping $x_2 = x_3 = 0$, while in the equations

$$\begin{array}{lcrrrr} y_1 & = & 2 & +x_1 & -x_2 & +x_3 \\ y_2 & = & 3 & -x_1 & & -x_3 \\ y_3 & = & 4 & -2x_1 & +x_2 & \end{array}$$

all variables should be ≥ 0

$y_1 = 2 + x_1 \geq 0$: OK if x_1 increases

$y_2 = 3 - x_1 \geq 0$: only OK if $x_1 \leq 3$

$y_3 = 4 - 2x_1 \geq 0$: only OK if $x_1 \leq 2$

So $y_i \geq 0$ only holds for all i if $x_1 \leq 2$

The highest allowed value for x_1 is 2, and then y_3 will get the value 0

pivot: swap x_1 and y_3: the basic variable y_3 will become non-basic, and the non-basic variable x_1 will become basic

---

143

Swap x_1 and y_3 in

$$\begin{array}{lcrrrr} y_1 & = & 2 & +x_1 & -x_2 & +x_3 \\ y_2 & = & 3 & -x_1 & & -x_3 \\ y_3 & = & 4 & -2x_1 & +x_2 & \end{array}$$

In the equation $y_3 = 4 - 2x_1 + x_2$ move x_1 to the left and y_3 to the right, yielding the equivalent equation

$$x_1 = 2 + \frac{1}{2}x_2 - \frac{1}{2}y_3$$

Next replace every x_1 by $2 + \frac{1}{2}x_2 - \frac{1}{2}y_3$ in the equations for z, y_1, y_2, yielding the following slack form:

maximize $z = 5 + \frac{1}{2}x_2 + x_3 - \frac{1}{2}y_3$ satisfying

$$\begin{array}{lcrrrr} x_1 & = & 2 & +\frac{1}{2}x_2 & & -\frac{1}{2}y_3 \\ y_1 & = & 4 & -\frac{1}{2}x_2 & +x_3 & -\frac{1}{2}y_3 \\ y_2 & = & 1 & -\frac{1}{2}x_2 & -x_3 & +\frac{1}{2}y_3 \end{array}$$

144

This is the end of the first pivot

Note that

- all equations are replaced by equivalent equations, so the new optimization problem is equivalent to the original one
- Now the basic variables are x_1, y_1, y_2 and the non-basic variables are x_2, x_3, y_3
- By construction again we have a slack form with a basic solution in which all non-basic variables are 0, and all basic variables are ≥ 0
- In this new basic solution the goal function

  $$z = 5 + \frac{1}{2}x_2 + x_3 - \frac{1}{2}y_3$$

  has the value 5, which is an improvement with respect to the original value 3

145

In this goal function

$$z = 5 + \frac{1}{2}x_2 + x_3 - \frac{1}{2}y_3$$

the non-basic variable x_2 has a positive factor, so increasing x_2 would cause a further increase: the starting point for the next pivot

Similar as in the first pivot we try to increase this non-basic variable x_2, while the other non-basic variables remain 0

146

$$\begin{array}{lcrcrcr} x_1 & = & 2 & +\frac{1}{2}x_2 & & & -\frac{1}{2}y_3 \\ y_1 & = & 4 & -\frac{1}{2}x_2 & +x_3 & & -\frac{1}{2}y_3 \\ y_2 & = & 1 & -\frac{1}{2}x_2 & -x_3 & & +\frac{1}{2}y_3 \end{array}$$

yields

$x_1 = 2 + \frac{1}{2}x_2 \geq 0$, OK if x_2 increases

$y_1 = 4 - \frac{1}{2}x_2 \geq 0$, only OK if $x_2 \leq 8$

$y_2 = 1 - \frac{1}{2}x_2 \geq 0$, only OK if $x_2 \leq 2$

So the maximal allowed value for x_2 is 2, in which case y_2 will get the value 0

So we will do a pivot swapping x_2 and y_2:
$y_2 = 1 - \frac{1}{2}x_2 - x_3 + \frac{1}{2}y_3$ yields
$x_2 = 2 - 2x_3 - 2y_2 + y_3$, so in the goal function and in the other equations we replace x_2 by $2 - 2x_3 - 2y_2 + y_3$, yielding

---

147

Maximize $z = 6 - y_2$
satisfying

$$\begin{array}{lcrcrcr} x_1 & = & 3 & -x_3 & -y_2 & & \\ x_2 & = & 2 & -2x_3 & -2y_2 & +y_3 & \\ y_1 & = & 3 & +2x_3 & +y_2 & -y_3 & \end{array}$$

Still this optimization problem is equivalent to the original one

Observation: since in

$$z = 6 - y_2$$

y_2 should be ≥ 0, the value of z will always be ≤ 6

On the other hand, in the basic solution with $x_3 = y_2 = y_3 = 0$ and $x_1 = 3$, $x_2 = 2$ and $y_1 = 3$ we obtain $z = 6$, so this basic solution yields the maximal value for z

---

148

Summarizing, starting from a linear optimization problem having a basic solution

- Bring the problem in slack form, that is, maximize $z = v + \sum_{j=1}^{n} c_j x_j$ under a set of constraints of the shape

  $$y_i = b_i + \sum_{j=1}^{n} a_{ij} x_j$$

  with $b_i \geq 0$, for $i = 1, \ldots, k$

- As long there exists j such that $c_j > 0$ do a **pivot**, that is
  - find the highest value for x_j for which $b_i + a_{ij}x_j \geq 0$ for all i, and $b_i + a_{ij}x_j = 0$ for one particular i
  - swap x_j and y_i and bring the result in slack form

At the end $c_j \leq 0$ for all j, from which can be concluded that the basic solution of this last slack form yields the maximal value for z

---

149

General remarks

- Optimization problems may be unbounded, for instance, there is no maximal value for $3 + x$ satisfying the constraint $x \geq 0$; in this mechanism this will be encountered if all $a_{ij} \geq 0$ for all i for the chosen j, so no equation yields an upper bound on x_j

- A pivot only requires some basic linear algebra of complexity $O(kn)$, still feasible for high values of k, n

- In case $c_j > 0$ for more than one value of j, then the procedure is non-deterministic

- If one always chooses the smallest j with $c_j > 0$, the repetition of pivots can be proved to terminate

- In worst case the number of pivots may be exponential, but in practice this number of pivots is limited, and the simplex method is very efficient

---

150

The method described until now only finds an optimal value when starting by a basic solution

However, in our intended application to SMT the situation is opposite: one is not interested in an optimal solution, only in the question whether there exists a solution

A set of constraints (or a problem) is called **feasible** if it admits a solution

Fortunately, the simplex method presented so far for finding an optimal solution of a feasible set of inequalities, can be applied for deciding feasibility of any given set of inequalities, as we will see now

---

151

For a set of linear inequalities

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i$$

for $i = 1, \ldots, k$, and $x_j \geq 0$ for $j = 1, \ldots, n$, we introduce a fresh variable $z \geq 0$, and extend the set of inequalities to

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - z \leq b_i$$

for $i = 1, \ldots, k$, and $x_j \geq 0$ for $j = 1, \ldots, n$

This extended problem is always feasible, even if b_i may be negative: choose z very large

---

152

original: $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i$

extended: $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - z \leq b_i$

Theorem

The original problem is feasible if and only if in the extended problem the maximal value of $-z$ is 0

Proof

If there is a solution of the extended problem with $-z = 0$, then it is a solution of the original problem, showing feasibility

Conversely, if the original problem is feasible, then the extended one admits a solution with $-z = 0$

Since we required $z \geq 0$, this is the maximal possible value of $-z$

End of Proof

153

In case $b_i \geq 0$ for all i, the original problem is trivially feasible, so assume $b_i < 0$ for some i

Extended problem in slack form:

maximize $-z$ satisfying

$$y_i = b_i - a_{i1}x_1 - a_{i2}x_2 - \cdots - a_{in}x_n + z$$

for $i = 1, \ldots, k$

Since $b_i < 0$ for some i this does not have a basic solution

As the first pivot swap the non-basic variable z with the basic variable y_i for i for which b_i is the most negative

Then after this pivot the resulting equivalent problem in slack form has a basic solution, as is easily proved

Proceed as before; if at the end the value for $-z$ is 0, then the original problem is feasible, otherwise it is not

---

154

Example

Find values $x, y \geq 0$ satisfying

$$x + 3y \geq 12 \ \wedge \ x + y \leq 10 \ \wedge \ x - y \geq 7$$

First step: make '$\leq$':

$$\begin{array}{rrcr} -x & -3y & \leq & -12 \\ x & +y & \leq & 10 \\ -x & +y & \leq & -7 \end{array}$$

Introduce z for maximizing $-z$:

$$\begin{array}{rrrcr} -x & -3y & -z & \leq & -12 \\ x & +y & -z & \leq & 10 \\ -x & +y & -z & \leq & -7 \end{array}$$

155

Slack form:

$$\begin{array}{rcrrrr} y_1 & = & -12 & +x & +3y & +z \\ y_2 & = & 10 & -x & -y & +z \\ y_3 & = & -7 & +x & -y & +z \end{array}$$

Indeed $x = y = z = 0$ does not yield a basic solution, since then $y_1 = -12$ and $y_3 = -7$ do not satisfy $y_i \geq 0$

Most negative b_i is $b_1 = -12$, so do pivot on z and y_1

replace every z by $z = 12 - x - 3y + y_1$:

Maximize $-12 + x + 3y - y_1$ satisfying

$$\begin{array}{rcrrrr} z & = & 12 & -x & -3y & +y_1 \\ y_2 & = & 22 & -2x & -4y & +y_1 \\ y_3 & = & 5 & & -4y & +y_1 \end{array}$$

156

$$\begin{array}{rcrrrr} z & = & 12 & -x & -3y & +y_1 \\ y_2 & = & 22 & -2x & -4y & +y_1 \\ y_3 & = & 5 & & -4y & +y_1 \end{array}$$

Indeed now we have a basic solution $x = y = y_1 = 0$, $z = 12$, $y_2 = 22$ and $y_3 = 5$, all ≥ 0

This is not by accident, this is always the case

Proceed by simplex algorithm as before

Maximize $-12 + x + 3y - y_1$, so swap x (or y) with z, y_2 or y_3

z: $12 - x \geq 0$, so $x \leq 12$
y_2: $22 - 2x \geq 0$, so $x \leq 11$
y_3: no bound on x

So next pivot: swap x with y_2

157

Replace x by $x = 11 - 2y + \frac{1}{2}y_1 - \frac{1}{2}y_2$ in

Maximize $-12 + x + 3y - y_1$ satisfying

$$\begin{array}{rcrrrr} z & = & 12 & -x & -3y & +y_1 \\ y_2 & = & 22 & -2x & -4y & +y_1 \\ y_3 & = & 5 & & -4y & +y_1 \end{array}$$

yields:
Maximize $-1 + y - \frac{1}{2}y_1 - \frac{1}{2}y_2$ satisfying

$$\begin{array}{rcrrrr} x & = & 11 & -2y & +\frac{1}{2}y_1 & -\frac{1}{2}y_2 \\ z & = & 1 & -y & +\frac{1}{2}y_1 & +\frac{1}{2}y_2 \\ y_3 & = & 5 & -4y & +y_1 & \end{array}$$

Next pivot: swap y and z

158

Replace y by $y = 1 - z + \frac{1}{2}y_1 + \frac{1}{2}y_2$ yields:

Maximize $-z$ satisfying

$$\begin{array}{rcrrrr} x & = & 9 & \cdots z & \cdots y_1 & \cdots y_2 \\ y & = & 1 & -z & +\frac{1}{2}y_1 & +\frac{1}{2}y_2 \\ y_3 & = & 1 & \cdots z & \cdots y_1 & \cdots y_2 \end{array}$$

Since the maximization function $-z$ has no positive factors any more, the resulting basic solution

$$z = y_1 = y_2 = 0, \ x = 9, \ y = 1, \ y_3 = 1$$

yields the optimal value $-z = 0$

Since $-z = 0$, the original set of inequalities

$$x + 3y \geq 12 \ \wedge \ x + y \leq 10 \ \wedge \ x - y \geq 7$$

is satisfiable: we found the solution $x = 9$, $y = 1$

159

Observations

- Doing this by hand this looks quite elaborative
- However, every step is completely systematic, and easy to implement very efficiently, even for thousands of variables
- Although worst case exponential, in practice this simplex algorithm is the most efficient way to decide whether a conjunction of linear inequalities is satisfiable or not
- The full simplex algorithm serves for deciding whether a solution exists, and if so, find an optimal one
- Linear optimization is also called **linear programming**
- There exists a more complicated **ellipsoid algorithm** for linear programming that is worst case polynomial, but in practice not better than the simplex algorithm

---

160

In linear programming one looks for an optimal solution satisfying a **conjunction** of linear inequalities

In **Satisfiability Modulo Theories** (SMT) this is generalized: one looks for a solution satisfying any propositional formula in which the building blocks may be both boolean variables and linear inequalities

So in SMT we may also use negations and disjunctions

A typical example that is beyond linear programming and essentially needs disjunctions is **Rectangle fitting**

It is presented in the first lecture of **Week 2 of the MOOC** on satisfiability

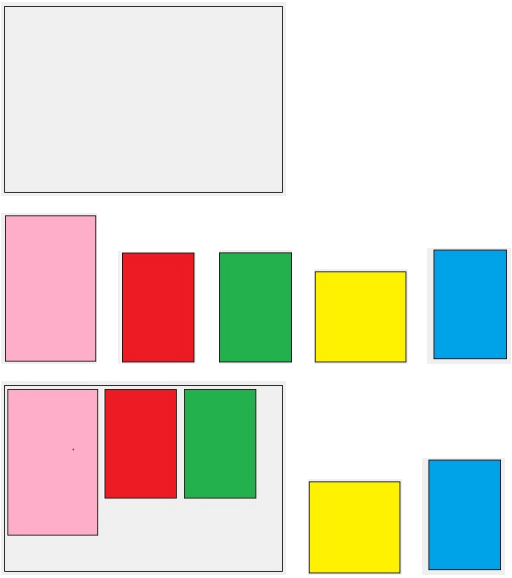
---

161

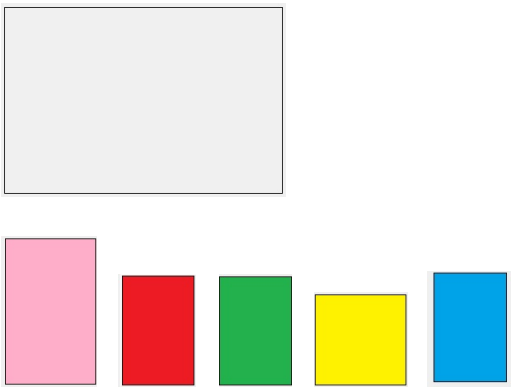
Rectangle fitting

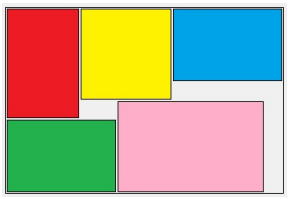
Given a big rectangle and a number of small rectangles, can you fit the small rectangles in the big one such that no two overlap?

162



163





164

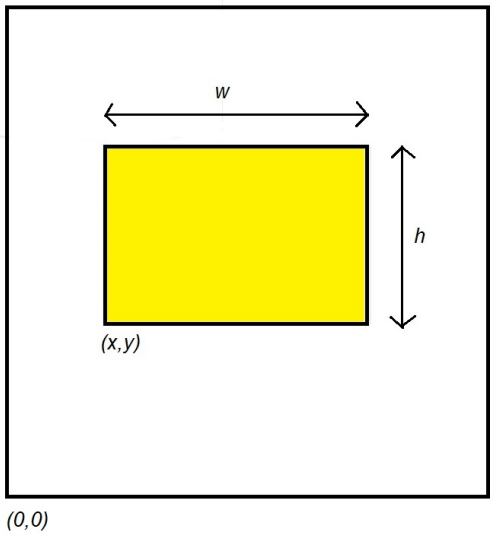
How to specify this problem?

Number rectangles from 1 to n

for $i = 1, \ldots, n$ introduce variables:

- w_i is the width of rectangle i
- h_i is the height of rectangle i
- x_i is the x-coordinate of the left lower corner of rectangle i
- y_i is the y-coordinate of the left lower corner of rectangle i

165



166

Collect all requirements in a formula

width / height requirements:

First rectangle has width 4 and height 6:

$$(w_1 = 4 \wedge h_1 = 6) \vee (w_1 = 6 \wedge h_1 = 4)$$

Similar for all $i = 1, \ldots, n$

167

Fit in big rectangle:

Let

- $(0, 0)$ = lower left corner of big rectangle
- W = width of big rectangle
- H = height of big rectangle

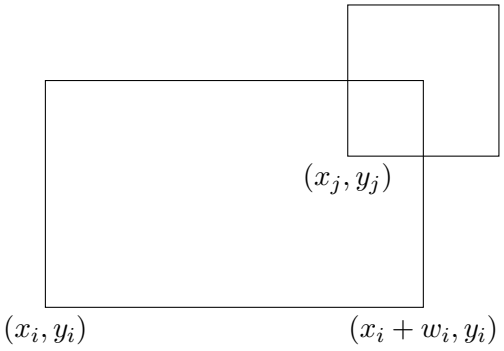
Requirements:

$$x_i \geq 0 \ \wedge \ x_i + w_i \leq W$$

and $y_i \geq 0 \ \wedge \ y_i + h_i \leq H$
for all $i = 1, \ldots, n$

168

No overlap?



If overlap, then $x_i + w_i > x_j$

169

Overlap:

Rectangles i and j overlap if

$$x_i + w_i > x_j$$

(right side of rectangle i is right from left side of rectangle j)

and $x_i < x_j + w_j$ (left side of rectangle i is left from right side of rectangle j)

and $y_i + h_i > y_j$ (top of rectangle i is above bottom of rectangle j)

and $y_i < y_j + h_j$ (bottom of rectangle i is below top of rectangle j)

170

So for all $i, j = 1, \ldots, n$, $i < j$, we should add the negation of this overlappingness:

$$\neg(x_i + w_i > x_j \ \wedge \ x_i < x_j + w_j \ \wedge$$

$$y_i + h_i > y_j \ \wedge \ y_i < y_j + h_j)$$

or, equivalently

$$x_i + w_i \leq x_j \ \vee \ x_j + w_j \leq x_i \ \vee$$

$$y_i + h_i \leq y_j \ \vee \ y_j + h_j \leq y_i$$

171

Summarizing, the full formula reads

$$\bigwedge_{i=1}^{n}((w_i = W_i \wedge h_i = H_i) \vee (w_i = H_i \wedge h_i = W_i))$$

$$\wedge \bigwedge_{i=1}^{n}(x_i \geq 0 \wedge x_i + w_i \leq W \wedge y_i \geq 0 \wedge y_i + h_i \leq H)$$

$$\wedge \bigwedge_{1 \leq i < j \leq n} (x_i + w_i \leq x_j \ \vee \ x_j + w_j \leq x_i \ \vee$$

$$y_i + h_i \leq y_j \ \vee \ y_j + h_j \leq y_i)$$

172

The resulting formula

- exactly describes all requirements of our rectangle fitting problem
- is composed from linear inequalities and propositional operations $\neg, \wedge, \vee$
- hence is perfectly suitable for SMT solvers

The formula is satisfiable **if and only if** our fitting problem has a solution

---

173

If the formula is satisfiable, then the SMT solver yields a **satisfying assignment**, that is, the corresponding values of x_i, y_i, w_i, h_i

From these values the intended solution is easily obtained

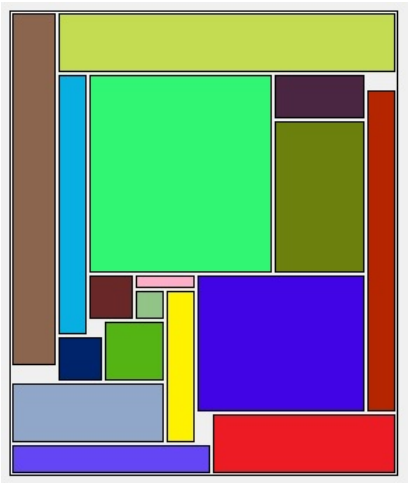
Applying a standard SMT solver like Z3, Yices, or CVC4:
feasible for rectangle fitting problems up to 20 or 25 rectangles

Is used in practice, for printing posters, see MOOC

For generating the formula in SMT syntax (typically hundreds of lines):

better generate by a script/program than edit yourself

174



175

How is satisfiability of such an SMT formula established?

The SAT solver is extended in such a way that it can deal with the inequalities as atoms, just like boolean variables

So we consider CNFs as before, but now the literals are linear inequalities or their negations, which are inequalities again

In the derivation rules Decide, UnitPropagate, Backtrack and Fail, the only 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

Such a conjunction of literals is a conjunction of inequalities, for which the simplex algorithm is called to check whether it is contradictory

176

Example

Consider the CNF of the following three clauses

$$\begin{array}{ll} (1) & x \geq y+1 \vee z \geq y+1 \\ (2) & y \geq z \\ (3) & z \geq x+1 \end{array}$$

$(y \geq z)$ (unit propagate on (2))

$(y \geq z)$ $(x \geq y+1)$ (unit propagate on (1) since $y \geq z \wedge z \geq y+1$ is unsatisfiable by simplex)

(fail on (3) since $y \geq z \wedge x \geq y+1 \wedge z \geq x+1$ is unsatisfiable by simplex)

This proves that the given CNF is unsatisfiable

177

So SAT solvers are extended to SMT solvers 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 typically a basic formula is a linear inequality, but it also works for other theories

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

For linear inequalities the simplex method serves for this goal

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

178

Reals or integers?

The simplex method works on linear inequalities over real numbers (LP: linear programming)

Surprisingly, solving linear inequalities over integers (ILP: integer linear programming) is much harder, this is even NP-complete since satisfiability of any propositional CNF can be expressed as an ILP problem

179

Typically, a set of inequalities may have a solution in reals, but not in integers

Fortunately, often by adding properties of integers (like $x \leq 3 \vee x \geq 4$ after a solution for x between 3 and 4 has been found) the resulting formula may be unsatisfiable over the

reals (to be proved by simplex based SMT), by which also unsatisfiability of the ILP problem is proved quickly

Conversely, if a satisfying assignment is found by simplex for which the solution is integer, then also an ILP solution has been found

---

180

This has been implemented in SMT solvers like `Z3`

The logic to be used is `:logic QF_UFLIA`

That is: Linear Integer Arithmetic (still quantifier free and over uninterpreted functions)

181

Example:

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

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

In an SMT solver like `Z3` one can directly enter and solve

```
(declare-const A Int)
(declare-const B Int)
(declare-const C Int)
(declare-const D Int)
(assert
(and
(> (* 2 A) (+ B C))
(> (* 2 B) (+ C D))
(> (* 2 C) (* 3 D))
(> (* 3 D) (+ A C))
))
(check-sat)
(get-model)
```

182

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

```
z3 test.smt
```

yields the output

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

indeed giving a solution

183

No bound on number size:

For similarly declared integers `A B C` and

```
(assert (and
(= A 98798798987987987987987923423879)
(= B 763429999988888888888736457864587)
(= (+ (* 87 A) (* 93 B)) (+ C C))
))
```

yields the output

```
sat
(= A 98798798987987987987987923423879)
(= B 763429999988888888888736457864587)
(= C 784769725592581081080371995964203 2)
```

184

One can also use **functions**:

```
(declare-fun F (Int) Int)
(assert
(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)))
))
(check-sat)
(get-model)
```

yielding

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

185

A convenient operation having boolean arguments that may result in a number is **if-then-else**, abbreviated to `ite`

It has always three arguments:

- the first is the **condition**, which is boolean,
- the second is the result in case this condition is true, and
- the third is the result in case this condition is false

Example:
`(ite (< a b) 13 (* 3 a))`
yields the value 13 in case $a < b$, otherwise it yields $3a$

Example:
`(+ (ite a 1 0) (ite b 1 0) (ite c 1 0))`
yields the number of variables that are true among the boolean variables a, b, c

186

Some SMT solvers allow **quantifications**
`forall` and `exists`, but typically perform quite weak when ranging over infinite domains like integers or reals

As long as quantifications are over a finite domain they can be expanded by `and` and `or`

For instance, $\forall i = 1 \cdots 8 : P(i)$ is expressed by
```
(and (P 1) (P 2) (P 3) (P 4)
(P 5) (P 6) (P 7) (P 8))
```

In this way formulas get much larger, but do not contain `forall` and `exists` at all

Typically, such formulas are not edited by hand, but are generated by a script, and for solvers it is no problem if the formulas get large

187

It is also possible to define your own finite data type, like a data type `A` consisting of three elements `a b c` by

```
(declare-datatypes () ((A a b c)))
```

Then $\forall x \in A : P$ is expressed by

```
(forall ((x A)) P)
```

This performs efficiently; probably internally it is just expanded to a big 'and', but in this way you may keep your formulas small

188

Model Checking

An automatic way to verify properties of programs

Where by **testing** only some cases are verified, model checking yields the claim to be verified for **all cases**

In our lectures on satisfiability we already saw **bounded model checking**, that is, for a program doing a fixed number of steps, say n, for every variable a we introduce a_i for $i = 0, 1, \ldots, n$, meaning the value of a after i steps, and prove by SAT/SMT that a formula is unsat, expressing the meaning of the program and the negation of the postcondition

189

Where in our bounded model check approach we encoded the SAT/SMT formula ourselves, now we present a language to describe the program and the property to be verified

It is a fragment of the language of the model checker **NuSMV** that is freely available

The program may be **non-deterministic**, so allowing choice

We roughly follow the second MOOC, on

symbolic model checking, and start by the first video (after the general introduction) of Week 1

190

A **marble game**

Starting from a single marble we do steps in which either

- five marbles are added, or
- the number of marbles is doubled

all until a maximum of 100 marbles

Typical questions to be answered by the model checker:

Is it possible to reach exactly 98 marbles?

or 99?

or 100?

191

In general, we have

- A notion of **state**s, described by the values of variables, in our case only the number of marbles
- A notion of **step**s, describing jumps from one state to another

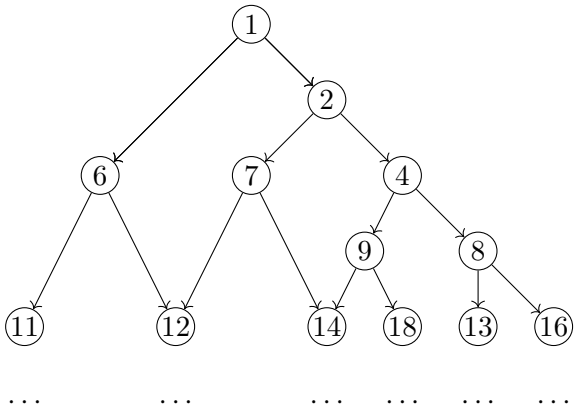
  The combination of states and steps is called a **transition system**, in which the steps are described by a **transition relation**
- A set of **initial** states, in our case consisting of the single state of one marble
- A property to check, in our case: exactly 98 marbles can be obtained, or 99, or 100

The transition system can be drawn as a graph, in which the nodes are states and the edges are steps

---

192

The graph for our marble game:



193

First we present how to specify such a transition system in our fragment of the NuSMV language

The property to be checked which will be presented next

First we declare the variables

In our case there is only one variable `a` representing the number of marbles:

```
VAR
a :  1..100;
```

194

Initially, the number of marbles is 1, stated by

```
INIT
a = 1
```

In the **transition relation**, denoted by

```
TRANS
```

we describe how the new value of `a`, described by

`next(a)`

depends on its old value `a`

195

The first case is adding 5, but this is only allowed if `a <= 95`

This is done by a `case` statement, being an if-then-else in which in the else branch the value of `a` remains unchanged:

```
case a <= 95 :  next(a) = a + 5;
TRUE : next(a) = a; esac
```

Similarly, the second case for doubling reads

```
case a <= 50 :  next(a) = 2*a;
TRUE : next(a) = a; esac
```

These are separated by a disjunction $\vee$, denoted by a vertical bar `|`

The full program reads

196

```
MODULE main
VAR
a :  1..100;
INIT
a = 1
TRANS
case a <= 95 :  next(a) = a + 5;
TRUE : next(a) = a; esac
|
case a <= 50 :  next(a) = 2*a;
TRUE : next(a) = a; esac
```

197

Here `MODULE main` is the start

`VAR` contains the **declarations** of the variables

`INIT` contains a **logical formula** over the declared variables, representing the set of **initial states**

`TRANS` contains a **logical formula** over `x` and `next(x)` where `x` runs over the declared variables, and represents the **transition relation**

198

In `TRANS` the fragment

```
case P : Q;
TRUE : R; esac
```

is just a notation for

$$(P \to Q) \wedge (\neg P \to R)$$

`case` can also be used with more arguments: always the first valid condition is chosen

---

199

Summarizing,

We described a fragment of the NuSMV language to

- declare variables,
- define the set of initial states, and
- define the transition relation

describing programs

Next we will present the logic CTL to give properties of such programs that can be proved or disproved automatically

---

200

CTL

= Computation Tree Logic

CTL is a logic to describe properties of a **transition system**, to be checked by model checking

We follow the MOOC on **symbolic model checking**, the third video of Week 1 on CTL

201

Variables $v_1, \ldots, v_n$ from sets $V_1, \ldots, V_n$ imply the state space

$$S = V_1 \times \cdots \times V_n$$

We concentrate on $V_1, \ldots, V_n$ being finite, hence S is finite too, but may be very large

A **transition system** is a **relation** $\rightarrow$ on S describing **step**s

So for $s, s' \in S$, by $s \rightarrow s'$ we denote that there is a step from s to s'

202

Assumption:

For every state $s \in S$ there exists $s' \in S$ such that $s \rightarrow s'$

Convention: for steps that require a condition we have $s \rightarrow s$ if s does not satisfy the condition

A **path** is an infinite sequence $s_1, s_2, s_3, \ldots \in S$ such that $s_i \rightarrow s_{i+1}$ for all i

Main idea of CTL: describe properties of the transition system as properties of paths

---

203

For instance, a state satisfying property ϕ is **reachable** if and only if

there **exists** a path $s_1, s_2, s_3, \ldots$
and there **exists** i such that s_i satisfies ϕ

CTL uses E (exists) and A (all) for quantification over paths,
and X (next), F (future), G (globally) and U (until) for properties over a single path

ϕ reachable: $EF\phi$

204

CTL syntax

Basic properties on variables using $=$, $<$, $\geq \cdots$ are called **atomic propositions**

CTL formulas are defined inductively by:

- false, true and atomic propositions are CTL formulas
- If ϕ and ψ are CTL formulas, then so are
  - $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\cdots$
  - $EX\phi$, $AX\phi$
  - $EF\phi$, $AF\phi$
  - $EG\phi$, $AG\phi$
  - $E[\phi U \psi]$, $A[\phi U \psi]$

205

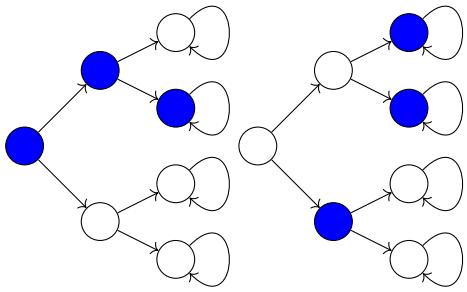
CTL semantics

- false, true, atomic propositions, $\neg$, $\wedge$, $\vee$ have there usual meaning
- $EX\phi$ holds if there exists a path $s_1, s_2, \ldots$ such that s_2 satisfies ϕ
- $EF\phi$ holds if there exists a path $s_1, s_2, \ldots$ such that s_i satisfies ϕ for **some** i
- $EG\phi$ holds if there exists a path $s_1, s_2, \ldots$ such that s_i satisfies ϕ for **all** i
- $E[\phi U \psi]$ holds if there exists a path $s_1, s_2, \ldots$ such that

  s_i satisfies ψ for **some** i, and s_j satisfies ϕ for **all** $j < i$
- AX, AF, AG, AU: same with 'exists a path' replaced by 'for all paths'

206

Examples



EG blue AF blue

207

Redundancy in CTL building blocks

By definition we have

$$AX\phi \equiv \neg EX\neg\phi$$

$$AG\phi \equiv \neg EF\neg\phi$$

$$AF\phi \equiv \neg EG\neg\phi$$

$$EF\phi \equiv E[\text{true}U\phi]$$

Slightly harder, but straightforward to check is

$$A[\phi U\psi] \equiv AF\psi \wedge \neg E[\neg\psi U\neg(\phi \vee \psi)]$$

So proposition logic and the operators EX, EG and EU fully generate CTL

208

Concluding,

- CTL is **computation tree logic**, and serves for describing properties of transition systems
- A transition system in which the states are labeled by the set of valid atomic propositions is called a **Kripke structure**, and provides a framework for formally defining the semantics of CTL
- Properties are described as quantifications A or E over **path**s, using X, G, F or U internally
- Only allowed in these combinations
- EX, EG and EU fully generate CTL

Next we will consider algorithms to verify CTL properties, following the MOOC on **symbolic model checking**, the fourth video of Week 1

209

How to check a CTL property?

That is, we have a transition system $(S, \to)$, where the **state space**

$$S = V_1 \times \cdots \times V_n$$

is implied by the variables $v_1, \ldots, v_n$ from **finite** sets $V_1, \ldots, V_n$

We want to check whether a CTL formula Φ holds for **all** states $s \in I$ for a given set of initial states $I \subseteq S$

210

Basic idea:

Compute the set S_Φ consisting of all states that satisfy Φ

Here a state $s \in S$ satisfies Φ if the set of all paths starting in s satisfies Φ

Then the property to check is $I \subseteq S_\Phi$

Since Φ is composed from CTL building blocks, we will describe how to compute S_Φ for all of these building blocks

---

211

$S_{\text{false}} = \emptyset$

$S_{\text{true}} = S$

For an atomic proposition p:

$S_p = \{s \in S \mid p(s)\}$

$S_{\neg\Phi} = \{s \in S \mid s \notin S_\Phi\}$

$S_{\Phi\vee\Psi} = S_\Phi \cup S_\Psi$

$S_{\Phi\wedge\Psi} = S_\Phi \cap S_\Psi$

It remains to describe S_Φ for Φ containing CTL operators EX, EG, EU

212

A state s satisfies $EX\Phi$ if there **exists** a path starting in s such that Φ holds in the **next** state of that path

So:

$$S_{EX\Phi} = \{s \in S \mid \exists t \in S_\Phi : s \to t\}$$

EG and EU are harder: they deal with properties of paths beyond a fixed finite part of the path

Idea:

Consider the first n steps for increasing n, until the
corresponding set does not change any more

In a more abstract way this can be presented as a **fixed point** construction; here we focus on the resulting algorithm

---

213

Computing $S_{EG\Phi}$

$EG\Phi$ means: there **exists** a path $s_0 \to s_1 \to s_2 \cdots$ on which Φ **globally** holds, that is, ϕ holds in s_i for all i

For $n = 0, 1, 2, \ldots$, let
T_n = set of states s_0 for which there exists a path $s_0 \to s_1 \to s_2 \cdots$ on which Φ holds for all $i \leq n$

Then $T_0 = S_\Phi$, and for all $n = 0, 1, \ldots$ we have

$$T_{n+1} = T_n \cap \{s \in S_\Phi \mid \exists t \in T_n : s \to t\}$$

These properties yield the following algorithm

214

$T_0 := S_\Phi$; $n := 0$;

repeat
$T_{n+1} := T_n \cap$
$\{s \in S_\Phi \mid \exists t \in T_n : s \to t\}$;
$n := n + 1$
until $T_n = T_{n-1}$

After finishing this algorithm we have $T_n = T_{n-1}$, yielding $T_n = T_i$ for all $i \geq n$

So then T_n states that for every i there is a path of which the first i states satisfy Φ

S finite $\Rightarrow$ this implies a path on which Φ globally holds (take $i = |S|$)

So after running this algorithm we have $S_{EG\Phi} = T_n$

The loop terminates since all sets are finite and $|T_n|$ decreases in every step

---

215

Computing $S_{E[\Phi U\Psi]}$

s_0 satisfies $E[\Phi U\Psi]$ means: there **exists** n such that P_n holds, for

P_n = there **exists** a path $s_0 \to s_1 \to s_2 \cdots$ on which Ψ holds in s_n, and Φ holds in s_i for all $i < n$

For $n = 0, 1, 2, \ldots$, let
U_n = set of states s_0 for which P_i holds for some $i \leq n$

Then $U_0 = S_\Psi$, and for all $n = 0, 1, \ldots$ we have

$$U_{n+1} = U_n \cup \{s \in S_\Phi \mid \exists t \in U_n : s \to t\}$$

These properties yield the following algorithm

216

$U_0 := S_\Psi$; $n := 0$;

repeat
$U_{n+1} := U_n \cup$
$\{s \in S_\Phi \mid \exists t \in U_n : s \to t\}$;
$n := n + 1$
until $U_n = U_{n-1}$

After finishing this algorithm we have $U_n = U_{n-1}$, yielding $U_n = U_i$ for all $i \geq n$

So after running this algorithm we have $S_{E[\Phi U \Psi]} = U_n$

The loop terminates since all sets are finite and $|U_n|$ increases in every step

---

217

Concluding,

- For an arbitrary CTL formula Φ we saw how to compute the set S_Φ, being the set of states that satisfy Φ
- In this computation we only needed the computation of the sets $T \cup U$, $T \cap U$ and

$$\{s \in T \mid \exists t \in U : s \to t\}$$

for given sets T, U

218

- In **explicit state based** model checking the complexity of this algorithm will be at least the order of the size of the state space S
- In **BDD based symbolic** model checking the same algorithm is feasible for much bigger state spaces, being the main next topic of this course

Next we will see how this algorithm applies for our marble puzzle, as in the last video of Week 1 of the MOOC on **symbolic model checking**

219

Recall the **marble puzzle**

Starting from a single marble we do steps in which either

- five marbles are added, or
- the number of marbles is doubled

Is it possible to reach exactly 98 marbles?

220

We already specified this transition system, using a variable a indicating the number of marbles

The question is whether

$$a = 98$$

is reachable from the initial state $a = 1$

Expressed as a CTL formula:

$EF(a = 98)$

221

In the CTL algorithm, any formula is first transformed to operators

$$EX,\ EG,\ EU$$

that generate full CTL

For our formula $EF(a = 98)$ this yields

$$E[\text{true} U (a = 98)]$$

So the algorithm for EU will be applied

222

So for $\Phi \equiv$ true and $\Psi \equiv (a = 98)$ we should compute

$U_0 := S_\Psi;\ n := 0;$

repeat
$U_{n+1} := U_n \cup$

$\{s \in S_\Phi \mid \exists t \in U_n : s \to t\}$;
$n := n + 1$
until $U_n = U_{n-1}$

Writing a state by the value of a we obtain

$$U_0 = \{98\}$$

$$U_1 = \{49, 93, 98\}$$

$$U_2 = \{44, 49, 88, 93, 98\}$$

223

$$U_3 = \{22, 39, 44, 49, 83, 88, 93, 98\}$$

$$\ldots$$

$$U_{17} = \{1, 2, 3, 4, 6, 7, 9, 11, \ldots$$

$$\ldots, 58, 63, 68, 73, 78, 83, 88, 93, 98\}$$

$$U_{18} = \{1, 2, 3, 4, 6, 7, 8, 9, 11, \ldots$$

$$\ldots, 58, 63, 68, 73, 78, 83, 88, 93, 98\}$$

$$U_{19} = U_{18}$$

So $S_{E[\Phi U \Psi]} = U_{18}$, and for $I = \{1\}$ we have $I \subseteq U_{18}$, so indeed
exactly 98 marbles can be obtained

224

Indeed, applying NuSMV on

```
MODULE main
VAR
a :  0..100;
INIT
a = 1
TRANS
case a<=95 :  next(a) = a + 5;
TRUE : next(a)=a; esac |
case a<=50 :  next(a) = 2*a;
TRUE : next(a)=a; esac
CTLSPEC EF (a=98)
```

yields as output

```
-- specification EF a = 98 is true
```

225

Way to find a witness: negate the goal, so try to prove

```
CTLSPEC !EF (a=98)
```

Here `!` stands for $\neg$

Then NuSMV finds false and produces a counterexample:

```
-> State:  1.1 <- a = 1
-> State:  1.2 <- a = 6
-> State:  1.3 <- a = 11
-> State:  1.4 <- a = 22
-> State:  1.5 <- a = 44
-> State:  1.6 <- a = 49
-> State:  1.7 <- a = 98
```

226

Summarizing,

- We saw how our marble puzzle can be expressed in a transition system and a CTL formula
- We saw how it is solved by the CTL algorithm and by NuSMV
- The sets of states were presented by showing their elements; internally NuSMV uses a BDD representation that can deal with state spaces with over 10^{100} states

Next we present a toy example that is also presented in the MOOC on **symbolic model checking**, in the second video of Week 4

227

Example: foxes and rabbits

Three foxes and three rabbits have to cross a river

There is only one boat that can carry at most two animals

When the boat is on the river, at each of the sides the number of foxes should be $\leq$ the number of rabbits, otherwise the rabbits will be eaten

How to solve this?

Let `f` be the number of foxes at the side where the boat is, and let `fb` be the number of foxes that goes into the boat, and similar `r` and `rb`

Let `b` be a boolean expressing where the boat is

The problem is solved by applying `NuSMV` on the following

228

```
MODULE main
VAR
r :  0..3;
rb :  0..2;
f :  0..3;
fb :  0..2;
b :  boolean;
INIT
b & f = 3 & r = 3
TRANS
next(b) = !b &
fb + rb <= 2 &
fb + rb >= 1 &
f - fb <= r - rb &
next(f) = 3 - f + fb &
next(r) = 3 - r + rb
CTLSPEC !EF(!b & f = 3 & r = 3)
```

229

Hardware verification

After these toy examples we want to scale up to real applications, as they are used in hardware verification

We could use more features of NuSMV, but everything can be expressed in our current tiny fragment of NuSMV, typically generating the NuSMV code by some program, expanding issues like 'all other variables remain unchanged'

A typical issue in hardware verification is **deadlock detection**

In real applications the underlying semantics is often implicit/undefined, that's why we precisely define a slightly simplified framework, still keeping the flavor of real applications

This is also presented in the MOOC on **symbolic model checking**, in the last two videos of Week 4

230

A **packet switching network** consists of

- a finite set N of *nodes*
- a finite set C of *channels* c, each having a *source* $\mathsf{source}(c) \in N$ and a *target* $\mathsf{target}(c) \in N$, such that every node $n \in N$ is the source of at least one channel in C
- a *routing function* $\mathsf{rout} : N \times N \to C$ for which $\mathsf{source}(\mathsf{rout}(n, m)) = n$ for all $n, m \in N$ to decide at which channel to proceed when a message with destination m arrives in node n

---

231

For $m, n \in N$ write
$\mathsf{next}_m(n) = \mathsf{target}(\mathsf{rout}(n, m))$

Note that $\mathsf{next}_m : N \to N$

A switching network is **correct** if for every $m, m' \in N, m \neq m'$, there exists $k > 0$ such that $\mathsf{next}_m^k(m') = m$

So if from node m' a message will be send to m, this message will arrive at its destination m in k steps

Typically, the routing function will be chosen in such a way that this k will be as small as possible

---

232

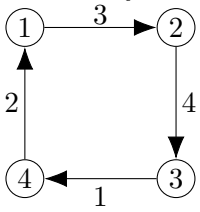
A channel may be free, or be occupied by a message

In the latter case it may block access to that channel for other messages

A state in which no (real) step can be executed is called a **deadlock**

In this case this means that no progress is possible due to blocked / occupied channels

Example: All four channels are occupied and labeled by the destination of its message



233

We restrict to the simplest setting being **asynchronous**

That is, at every moment steps can be done without a central control by a clock

For investigating deadlocks the contents of the data package in the message does not play a role: the content of a channel is identified by the destination $m \in N$ of the corresponding message

In a **state** of the system for every channel $c \in C$ we have a bit stating whether it is free, and if it is not free it contains a message destination $n \in N$

234

For a set $M \subseteq N$ of nodes that may send and receive, we consider the following steps:

- (send) For $m, m' \in M$ if channel $\mathsf{rout}(m, m')$ is free, then the channel may be filled by m'

  This expresses that m sends a message with destination m'

- (receive) If $c \in C$ contains destination $m \in N$, and $\mathsf{target}(c) = m$, then the message may reach its destination by which c is made free

- (process) If $c \in C$ contains destination $m \in N$, and $\mathsf{target}(c) \neq m$, then the message may proceed to $c' = \mathsf{rout}(\mathsf{target}(c), m)$

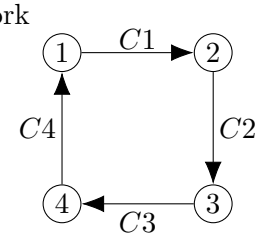
  This is only possible if this channel c' is free, and as a result this channel c' is not free any more, but occupied by m, while c is made free

---

235

Now for a given correct packet switching network (N, C, rout) and a given set $M \subseteq N$ of nodes that may send and receive, we wonder whether a deadlock is reachable from the initial state in which all channels are free

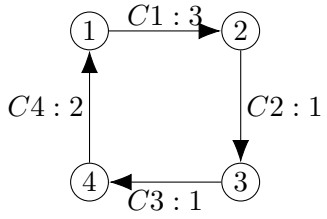
For instance, for $M = \{1, 2, 3\}$ in the network



by the five steps
send(3,2): $C3 := 2$
process $C3$: $C4 := 2$; $C3 :=$ free
send(3,1): $C3 := 1$
send(1,3): $C1 := 3$
send(2,1): $C2 := 1$

236

the following deadlock is reached



For more complicated networks doing this by hand is intractable, but this NuSMV approach serves well to establish whether deadlocks are reachable or not

See practical assignment

237

Features of `NuSMV`

- **CTL** = computational tree logic, we have seen
- **LTL** = linear temporal logic, also based on X = next, U = until, G = global and F = future, but not having A, E; incomparable with CTL, but reachability is covered: for checking whether Φ is reachable check

  `G !`(Φ)
- Default underlying machinery based on BDDs; by calling verbose option `NuSMV -v 3` information on underlying BDDs is shown

238

Features of `NuSMV`

- **Bounded model checking** `-bmc`: fixing a maximal number k of steps, introduce a_i for every variable a and every $0 \leq i \leq k$ representing the value of a after i steps, and express and solve the problem by SAT
- Bounded model checking only works for LTL
- Newer version NuXMV of NuSMV also supports bounded model checking for real and integer variables, expressing and solving the problem by SMT

239

SMT example

Let the file `test` contain:

```
MODULE main
VAR a :  real;
INIT a = 1
TRANS next(a) = a / 2
LTLSPEC G (a > 0.01)
```

and the file `comm` contain:

```
go_msat
msat_check_ltlspec_bmc -k 20
quit
```

then `nuxmv -source comm test` produces the counterexample that `a` reaches 1/128 in 7 steps

240

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

241

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

---

242

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

243

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:

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

244

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

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

245

Example:

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

$$\begin{array}{rcl}
\neg a \vee b & \rightarrow & (a \oplus \mathit{true}) \vee b \\
& \rightarrow & ((a \oplus \mathit{true}) \wedge b) \oplus a \oplus \mathit{true} \oplus b \\
& \rightarrow & (a \wedge b) \oplus (\mathit{true} \wedge b) \oplus a \oplus \mathit{true} \oplus b \\
& \rightarrow & (a \wedge b) \oplus b \oplus a \oplus \mathit{true} \oplus b \\
& \rightarrow & (a \wedge b) \oplus a \oplus \mathit{true} \oplus \mathit{false} \\
& \rightarrow & (a \wedge b) \oplus a \oplus \mathit{true}
\end{array}$$

$$\begin{array}{rcl} \neg(a \wedge \neg b) & \rightarrow & \neg(a \wedge (b \oplus \mathit{true})) \\ & \rightarrow & \neg((a \wedge b) \oplus (a \wedge \mathit{true})) \\ & \rightarrow & \neg((a \wedge b) \oplus a) \\ & \rightarrow & (a \wedge b) \oplus a \oplus \mathit{true} \end{array}$$

246

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

---

247

We want a unique representation in which not only $\neg$, $\vee$ and $\wedge$ can be computed efficiently, but also applies for CTL model checking over booleans, to be extended to integers of a finite range by binary representation

Here sets of states are represented by formulas, where a state is in the set if and only if the formula yields true on the state

Then the set operations complement, $\cup$ and $\cap$ correspond to $\neg$, $\vee$ and $\wedge$ on formula level

Moreover, for the transition relation $\rightarrow$ for given sets T, U we should be able to efficiently compute

$$\{ s \in T \mid \exists t \in U : s \rightarrow t \}$$

We saw that then we can compute EX, EG and EU, where for EG and EU the algorithms do a repeat until two sets are equal

The operators $\neg$, $\vee$, $\wedge$, EX, EG and EU generate full CTL

248

How to compute the set V defined by

$$V = \{ s \in T \mid \exists t \in U : s \rightarrow t \}$$

Here T and U are given sets represented by boolean functions on the variables $a_1, \ldots, a_n$

The transition relation $\rightarrow$ is given by a boolean function on a double set of variables $a_1, \ldots, a_n, \mathrm{next}(a_1), \ldots, \mathrm{next}(a_n)$

249

Write

$$V = \{(a_1, \ldots, a_n) \mid$$

$$\exists(\mathrm{next}(a_1), \ldots, \mathrm{next}(a_n)) :$$

$$P(a_1, \ldots, a_n, \mathrm{next}(a_1), \ldots, \mathrm{next}(a_n))\}$$

in which

$$P(a_1, \ldots, a_n, \mathrm{next}(a_1), \ldots, \mathrm{next}(a_n)) \iff$$

$$(a_1, \ldots, a_n) \in T \ \wedge$$

$$(a_1, \ldots, a_n) \rightarrow (\mathrm{next}(a_1), \ldots, \mathrm{next}(a_n)) \ \wedge$$

$$(\mathrm{next}(a_1), \ldots, \mathrm{next}(a_n)) \in U$$

So $P(\cdots)$ can be computed

250

The big '$\exists$' can be replaced by

$$\exists \mathrm{next}(a_1) \exists \mathrm{next}(a_2) \cdots \exists \mathrm{next}(a_n)$$

Observe that for every boolean variable x:

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

Applying this n times, for $x = \mathrm{next}(a_1), \ldots, \mathrm{next}(a_n)$, all '$\exists$' are eliminated, by which V is computed

---

251

Summarizing, the following operations should be efficient:

- Operations complement, $\cup$ and $\cap$ on sets, corresponding to $\neg$, $\vee$ and $\wedge$ in propositional notation

- Checking equality of sets; for this we use uniqueness of representation
- Compute $\phi[x := \mathit{false}]$ and $\phi[x := \mathit{true}]$ from ϕ

B(inary) D(ecision) D(iagrams) are a representation in which all of these operations can be done efficiently, and this is the underlying data structure in NuSMV for CTL model checking

---

252

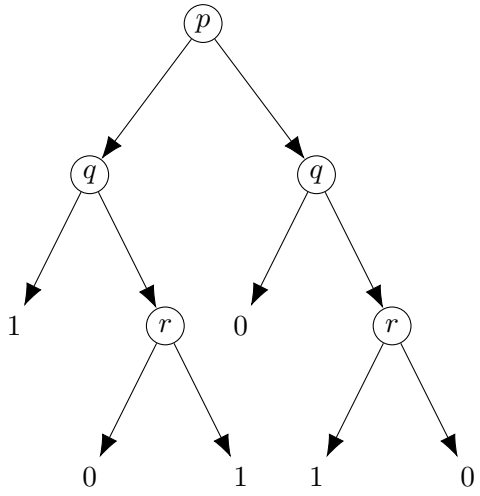
Binary Decisions Diagrams (BDDs) are based on **decision trees**, and that we will consider now, and are also presented in the second and third video of Week 2 of the **MOOC on symbolic model checking**

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

- Every node is labeled by a boolean variable
- Every leaf is labeled by 1 or 0, representing true and false respectively

---

253



254

A decision tree describes a boolean function:

If every variable has a boolean value then the corresponding function value is obtained by

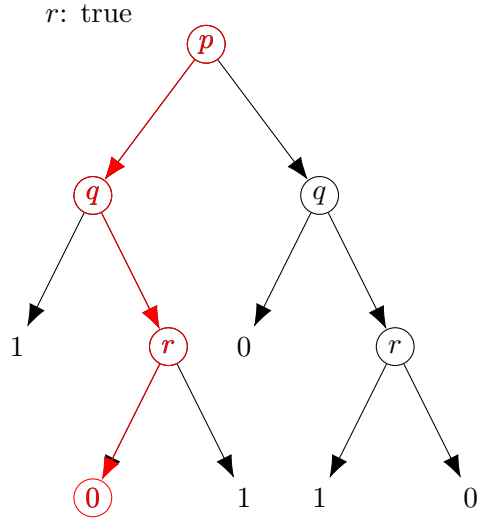
- Start at the root
- For any node: go to the **left** if the corresponding variable is true, otherwise go to the **right**
- Repeat until a leaf has been reached
- If the leaf is 1 then the result is true, otherwise false

255

Hence every node is interpreted as an if–then–else

Consider our example for the values

p: true
q: false
r: true

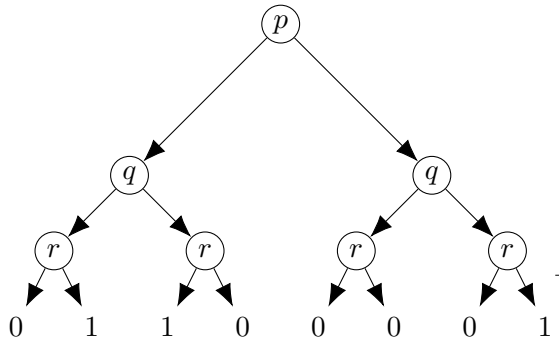


Hence the result is false

256

Does every boolean function on finitely many boolean variables have a representation as decision tree?

YES: it can be defined by a **truth table**, and any truth table on n variables having 2^n lines can be represented as a decision tree with 2^n leaves

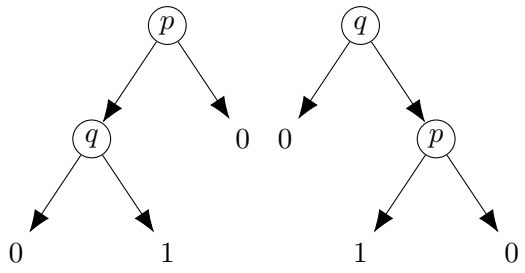


257

Does every boolean function on finitely many boolean variables have a **unique** representation as decision tree?

NO

The following two decision trees both represent the boolean function on p, q that yields true in case p is true and q is false, and false otherwise



258

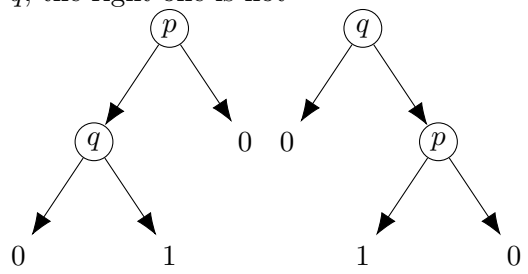
Observe that in one case p is on top of q, while in the other case q is on top of p

Fix an order $<$ on the boolean variables, like $p < q$

An **ordered decision tree** with respect to $<$ is a decision tree such that if node n is on top of node n', then

$$\mathsf{label}(n) \ < \ \mathsf{label}(n')$$

So the left one is ordered with respect to $p < q$, the right one is not



259

Concluding, we saw that

- Every decision tree represents a boolean function
- Every boolean function on n boolean variables can be represented as an **ordered** decision tree of depth n by its truth table
- The requirement of being ordered was motivated by the requirement of **uniqueness** of the representation
- Next we will see that we will need one more requirement for uniqueness

---

260

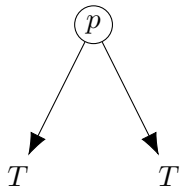
Uniqueness of decision trees

Fixing the order $<$ on the boolean variables, does every boolean function have a unique representation as an ordered decision tree with respect to $<$?

No

Let T be any ordered decision tree, and let p be a variable less than the variables in T

Then T and

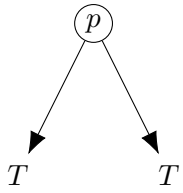


are two ordered decision trees representing the same boolean function

---

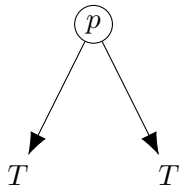
261

We are looking for a **small** representation, hence among these two we prefer T and want to exclude



262

Replacing



by T is called **elimination**

An ordered decision tree on which no elimination is possible is called a

reduced ordered decision tree

263

Theorem

For a fixed order $<$ on boolean variables, every boolean function has a **unique** representation as a reduced ordered decision tree

Proof sketch

Existence: Start by the ordered decision tree reflecting the truth table, and apply elimination anywhere in the decision tree as long a possible

Elimination strictly decreases the size, so cannot go on forever

During elimination orderedness is maintained

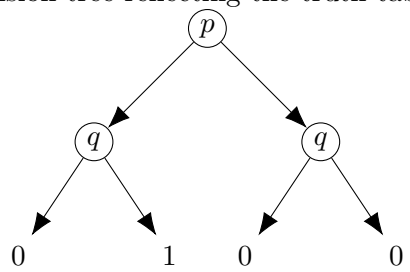
So at the end we have a reduced ordered decision tree

representing the given boolean function

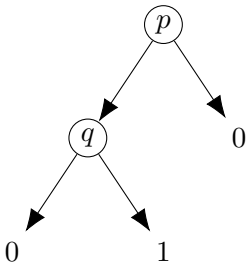
264

An example

For the boolean function defined by the formula $p \wedge \neg q$, for the order $p < q$ the ordered decision tree reflecting the truth table is



Applying elimination on
the right q yields



Now no elimination is possible any more, so this is a reduced ordered decision tree

---

265

It remains to prove **uniqueness**, that is:

if T and U are reduced ordered decision trees representing the same boolean function, denoted by $T \simeq U$, then $T = U$

This is proved by **induction** on the number of variables

If this number is zero, then T and U both

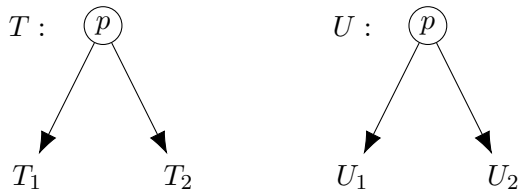
consist of the single leaf 0 or 1, so OK

This is the base case of the induction; the induction step splits into two cases

---

266

Case 1: T and U have the same root, so



Since $T \simeq U$, we get $T_1 \simeq U_1$ and $T_2 \simeq U_2$

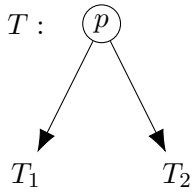
So by the induction hypothesis we obtain $T_1 = U_1$ and $T_2 = U_2$

So $T = U$

267

Case 2: T and U have distinct roots, say, the root p of T is the smaller

Then



and p does not occur in U

Since $T \simeq U$, we get $T_1 \simeq U$ and $T_2 \simeq U$

So by the induction hypothesis we obtain $T_1 = U = T_2$

But then elimination can be applied on T, contradicting

the assumption that T is reduced

End of proof

268

Binary Decisions Diagrams (BDDs)

are decision trees stored more efficiently, also presented in the last video of Week 2 of the **MOOC on symbolic model checking**

How to store (decision) trees efficiently?

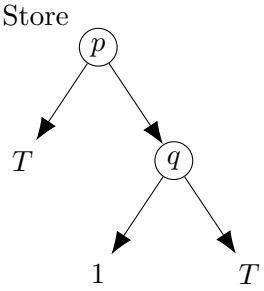
If a subtree occurs twice, we do not want to store it twice

Instead it will be stored only once, with two incoming pointers

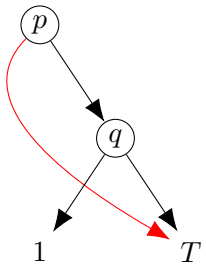
---

269

Store



as



270

Now it is a
D(irected) A(cyclic) G(raph)

rather than a tree, in which **sharing** occurs

In the usual implementation for 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

271

An **O(rdered) BDD** is an ordered decision tree in DAG representation, so if node n is on top of node n', then its label is smaller

For obtaining an efficient unique representation we do not only want to allow sharing, but want to **force** sharing as much as possible

The pointer describing the left (right) branch of a node is called the **true-(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

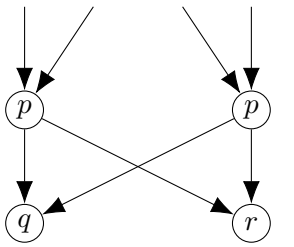
since then these two nodes can be shared

272

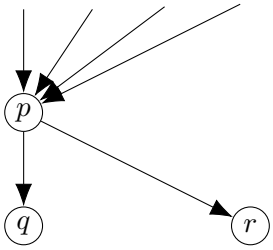
If two nodes have these properties, then we can do a **merge**, that is, remove one of them and redirect the incoming arrows to the other node

In a picture:

merge: replace



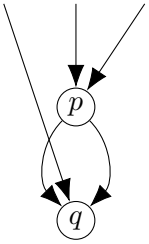
by



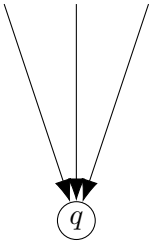
273

From decision trees we still have **elimination**:

replace



by



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

274

Theorem

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

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

275

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 $\implies$ $T = U$

276

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

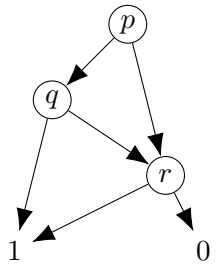
End of proof of main theorem

277

An example

The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

With respect to the order
$p < q < r$ its ROBDD is



278

In such a ROBDD every node represents a boolean function itself

The ROBBD of this function is the part of the original ROBDD of which the indicated node is the root

All nodes of a ROBDD represent **distinct** boolean functions, since if two would represent the same, then they can be shared by applying merge and elimination steps, contradicting being R(educed)

279

The operator $\leftrightarrow$

$p \leftrightarrow q$ yields true if and only if p and q are **equivalent**, that is, they are both true or both false

One can check that $\leftrightarrow$ is
associative, that is,

$$(p \leftrightarrow q) \leftrightarrow r$$

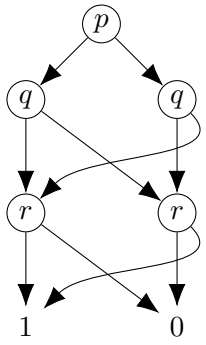
and

$$p \leftrightarrow (q \leftrightarrow r)$$

are equivalent, so can be
written as $p \leftrightarrow q \leftrightarrow r$

280

For $p < q < r$ the ROBDD of
$p \leftrightarrow q \leftrightarrow r$ is

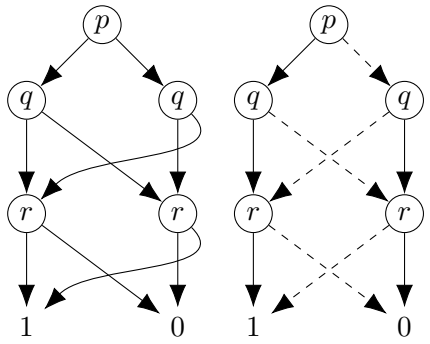


yields true if and only if the number of variables that is false, is **even**

281

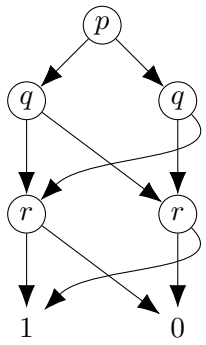
Alternative notation to avoid curved arrows: use **solid** arrows for true-branch and **dashed** arrows for false-branche

Instead of one denotes

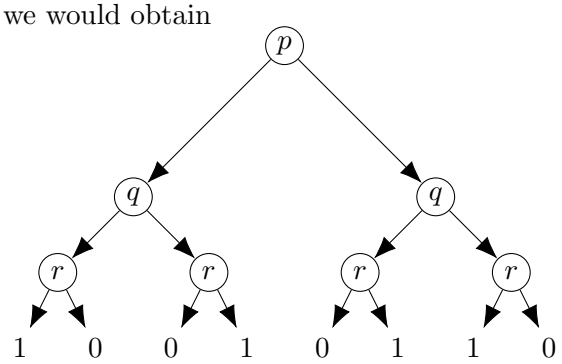


282

Sharing really helps: without sharing (so the unique reduced ordered decision tree) for $p \leftrightarrow q \leftrightarrow r$ instead of



we would obtain

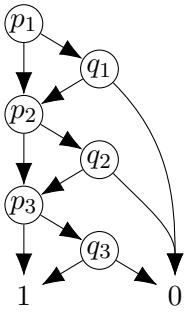


Doing the same for $p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n$ yields a ROBDD of $2n-1$ nodes, and a reduced ordered decision tree of $2^n - 1$ nodes: an **exponential** gap

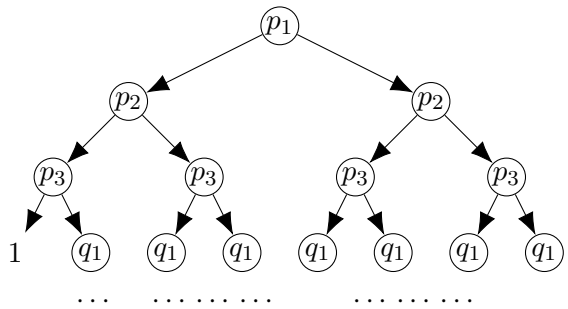
283

The ROBDD of $(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge (p_3 \vee q_3)$ with respect to

$p_1 < q_1 < p_2 < q_2 <$
$p_3 < q_3$:



$p_1 < p_2 < p_3 < q_1 < q_2 < q_3$:



where all q_1 nodes represent distinct boolean functions on q_1, q_2, q_3, so cannot be shared

284

More general, the ROBBD of

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge \cdots \wedge (p_n \vee q_n)$$

with respect to the order $p_1 < q_1 < p_2 < q_2 < \cdots < p_n < q_n$ has exactly $2n$ nodes

The ROBDD of the same boolean function with respect to the order $p_1 < p_2 < \cdots < p_n < q_1 < q_2 < \cdots < q_n$

has more than 2^n nodes, since only at the level of q_1 there are 2^n distinct nodes

So distinct orders may result in ROBDDs of sizes with an **exponential** gap in between

Heuristic: choose the order in such a way that variables close to each other in the formula are also close in the order

285

These examples were also presented in the first video of Week 3 of the **MOOC on symbolic model checking**

The rest of Week 3 of that MOOC is on the algorithm to compute the ROBDD for a given propositional formula and an order on the variables, as will be presented now

---

286

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*

287

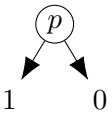
Algorithm to determine the ROBDD of a formula

Every formula is of the shape:

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

288

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

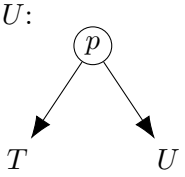
- $ROBDD(false) = 0$, $ROBDD(true) = 1$,
- $ROBDD(p) =$ 

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

289

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

We write $p(T, U)$ for the BDD having root p for which the left branch is T and the right branch is U:



290

Shortly write

$$\diamond(T, U)$$

instead of $apply(T, U, \diamond)$

As the basis of the recursion we define

$\diamond(T, U) =$

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

291

If T, U not both in $\{false, true\}$, let p be the **smallest** variable occurring in T and U

Three cases:

- p is on top of both T and U
- p is on top of T but does not occur in U
- p is on top of U but does not occur in T

For these three cases we define

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

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

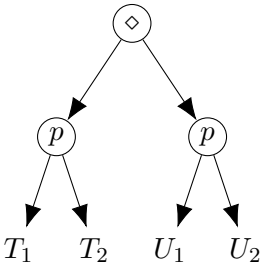
292

Intuitively: for two BDDS T, U computing

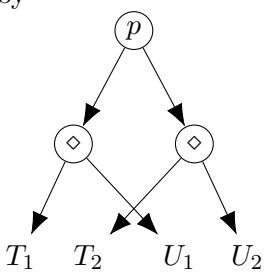
$$apply(T, U, \diamond) = \diamond(T, U)$$

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 a picture: Replace

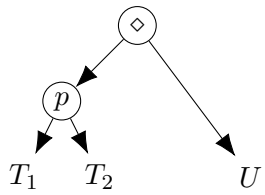


by

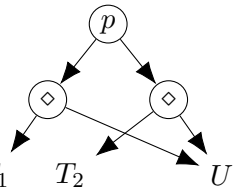


293

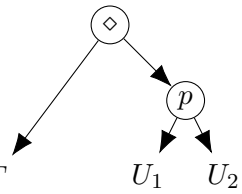
If p not in U, then replace



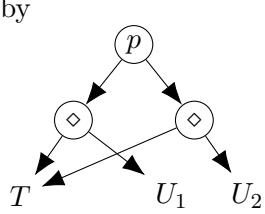
by



If p not in T, then replace



by



294

In this way for given ROBDDs T, U, the ROBDD of $T \diamond U$ is obtained by

- first put $\diamond$ on top of T, U
- then throw $\diamond$ downward step by step, preserving the order and the meaning of the boolean function
- whenever $\diamond$ reaches leaves 0, 1, it disappears and is replaced by the value from the truth table of $\diamond$

So at the end $\diamond$ has been disappeared completely, and the result is an ordered BDD representing the required boolean function

295

This approach is the key idea of the algorithm to compute the ROBDD of a propositional formula as it is used in practice, for instance in `NuSMV`

We did not yet worry about **complexity**

In this recursion only calls $\diamond(T', U')$ 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

296

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

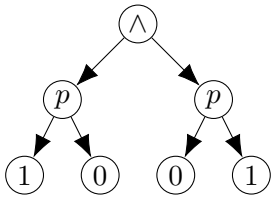
$$O(\#T * \#U),$$

where # is the number of nodes of a BDD

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

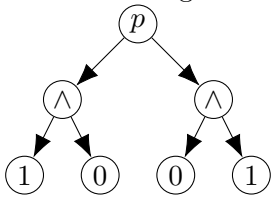
NO

For instance, computing $p \wedge \neg p$ starts in

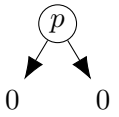


297

After throwing down $\wedge$ this yields



Since both $1 \wedge 0$ and $0 \wedge 1$ yield 0, this results in



This is **not** the required ROBDD, as **eliminate** has to be applied to yield the result 0

298

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, for instance in `NuSMV`

299

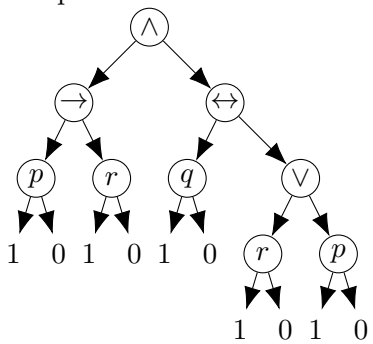
Example

We compute the ROBDD of

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

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

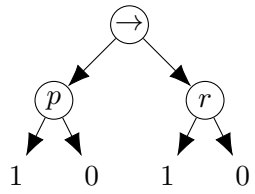
In a picture:



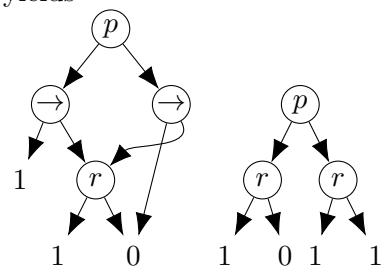
in recursion: start by computing ROBDDs of both arguments of top symbol $\wedge$

300

Left argument:

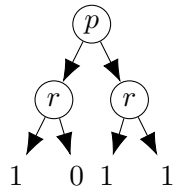


yields

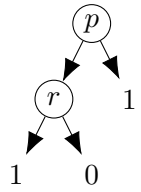


301

Applying **elimination** on



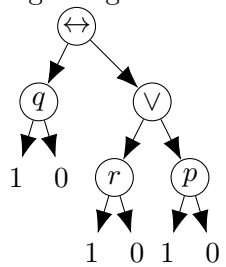
yields the result



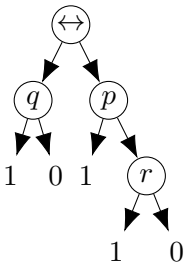
of the left argument of $\wedge$

302

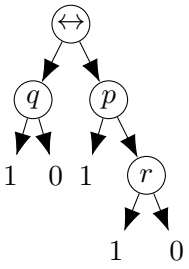
Right argument:

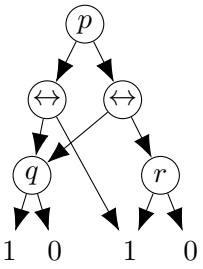


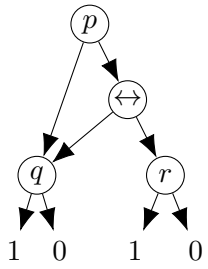
First process $\vee$:

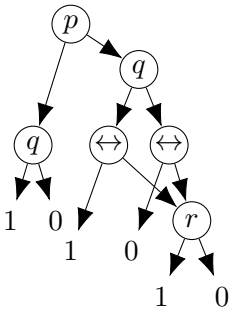


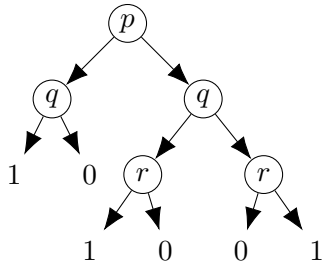
303







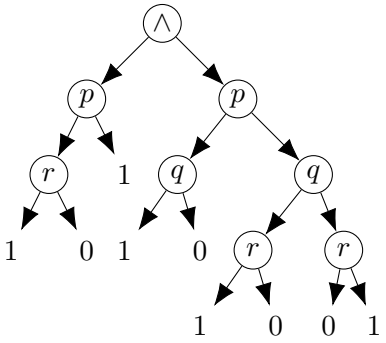


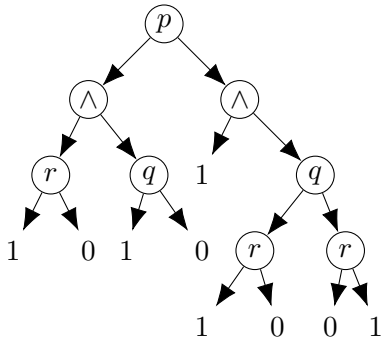


304

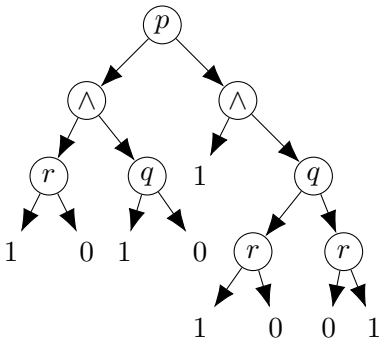
Now we have computed the ROBDDS of both arguments of $\wedge$ in the original formula,

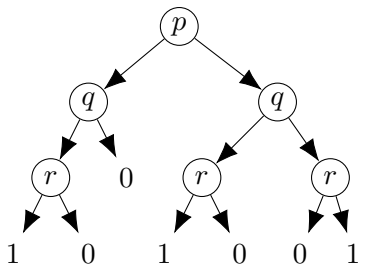
and it remains to apply $\wedge$ on these two



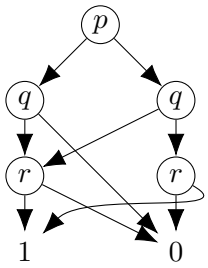


305





Apply **sharing**:



306

We computed the ROBDD of the formula

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

with respect to the order $p < q < r$, using our 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

---

307

Example:

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

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

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*

308

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

---

309

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

310

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

---

311

Example:

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

Let

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

---

312

We assume both chip designs have no circular behaviour:

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

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

Similarly for C_i

Let $\phi_1, \ldots, \phi_p$ be the simple formulas reflecting the behaviour of the ports, where ϕ_i describes how A_i depends on

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

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

Similarly $\psi_1, \ldots, \psi_q$ describe $C_1, \ldots, C_q$

313

To be proven:

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

Expressed in a formula:

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

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

---

314

Using BDDs this can be done much more efficient:

- only consider $B_1, \ldots, B_k$ as variables
- subsequently compute ROBDDs for $A_1, \ldots, A_p$ using $\phi_1, \ldots, \phi_p$
- similarly for $C_1, \ldots, C_q$
- check whether the ROBDDs of A_{p-n+i} and C_{q-n+i} coincide for $i = 1, \ldots, n$

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

---

315

BDDs for pseudo boolean constraints

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

$$c_1x_1 + c_2x_2 + c_3x_3 + \cdots + c_nx_n \leq a$$

where c_i, a are given integer values, and x_i are boolean variables

Such pseudo boolean constraints often occur in scheduling problems

Now we show how such a pseudo boolean constraint can be transformed to a CNF by using BDDs

The constraint

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

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

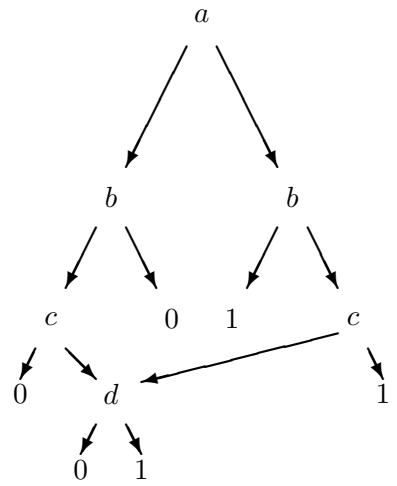
Construct the ROBDD of this boolean function

316

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:

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



317

How to build such an ROBDD?

- Evaluate values $a, b, \ldots$ in the inequality

  E.g., in the example:

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

  $a = 1, b = 1$ 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

318

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:

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

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

---

319

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

320

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

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

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

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

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

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

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

Note the sharing of D

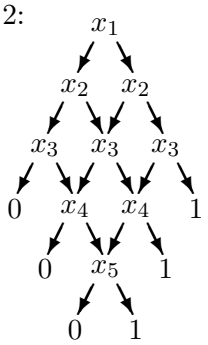
321

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 $n = 2$:



322

This BDD based technique has been implemented in `minisat+` to solve pseudo boolean constraint problems by means of the SAT solver `minisat`

323

Predicate logic

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

(except for the excursion to SMT)

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

324

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

325

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** $= 0, 1, 2, 3, \ldots$

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

326

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

327

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

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

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

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

328

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

---

329

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 $[P, \alpha] \in \{\mathit{false}, \mathit{true}\}$ 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 $\alpha : \mathcal{X} \to M$ such that $[P, \alpha] = \mathit{true}$

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

330

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` and `Z3` 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`

331

Now we extend the **resolution method** to be applicable for predicates; this is fully exploited in the tool `Prover9`, being the successor of `Otter`

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 $P(t_1, \ldots, t_n)$ where P is a relation symbol of arity n and $t_1, \ldots, t_n$ are terms

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

---

332

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

333

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

334

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

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

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

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

335

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

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

is extended to

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

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

336

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

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

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:

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

for all substitutions σ, τ satisfying

$$P\sigma = Q\tau$$

337

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

338

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**

339

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

340

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

341

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:

$$\exists\sigma(P\sigma = Q\sigma) \;\equiv\; \exists\sigma(\forall(t,u) \in S(t\sigma = u\sigma))$$

$$\wedge\, (\exists\sigma(P\sigma = Q\sigma) \;\rightarrow\; un)$$

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:

342

$S := \{(P, Q)\}$;
$un := \mathit{true}$;

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 $un := \mathit{false}$
    else if $v(t) \wedge \neg(in(t, u))$ then
      $S := S[t := u]$
    else if $v(u) \wedge in(u, t)$ then

$un := \mathit{false}$

else if $v(u) \wedge \neg(in(u,t))$ then
$S := S[u := t]$

else if $t = f(t_1, \ldots, t_n) \wedge u = f(u_1, \ldots, u_n)$ then
$S := S \cup \{(t_1, u_1), \ldots, (t_n, u_n)\}$

else if $t = f(\cdots) \wedge u = g(\cdots) \wedge f \neq g$ then
$un := \mathit{false}$

}

343

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 $\{(t_1, u_1), \ldots, (t_n, u_n)\}$ is defined to be

$$\sum_{i=1}^{n} (|t_i| + |u_i|)$$

where $|t|$ is the size of the term t

344

After termination the following holds:

- $un = \mathit{false}$ and P and Q are not unifiable, or
- $un = \mathit{true}$ 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

---

345

$S := \{(P, Q)\}$;
$un := \mathit{true}$;
$mgu := id$;
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 $un := \mathit{false}$
else if $v(t) \wedge \neg(in(t, u))$ then
$\{S := S[t := u]$;
$mgu := mgu[t := u]\}$
else if $v(u) \wedge in(u, t)$ then
$un := \mathit{false}$
else if $v(u) \wedge \neg(in(u, t))$ then
$\{S := S[u := t]$;
$mgu := mgu[u := t]\}$
else if $t = f(t_1, \ldots, t_n) \wedge u = f(u_1, \ldots, u_n)$ then
$S := S \cup \{(t_1, u_1), \ldots, (t_n, u_n)\}$
else if $t = f(\cdots) \wedge u = g(\cdots) \wedge f \neq g$ then
$un := \mathit{false}$
}

346

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

$$x\sigma = x,\ y\sigma = g(w),\ z\sigma = f(x),\ w\sigma = w,$$

347

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

348

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

349

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

---

350

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:

351

- 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

352

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*

353

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

354

Examples of resolution sequences:

1	$P(x, f(y)) \vee \neg P(x, y)$	
2	$\neg P(x, f(f(y)))$	
3	$P(a, g(y))$	
4	$P(a, f(g(y)))$	$(1, 3)$
5	$P(a, f(f(g(y))))$	$(1, 4)$
	$\bot$	$(2, 5)$

355

1	$P(x, f(y)) \vee \neg P(x, y)$	
2	$\neg P(x, f(f(y)))$	
3	$P(a, g(y))$	
4	$\neg P(x, f(y))$	$(1, 2)$
5	$\neg P(x, y)$	$(1, 4)$
	$\bot$	$(3, 5)$

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

356

A **Horn clause** is a clause containing at most one positive literal

Usually a Horn clause $C \vee \neg C_1 \vee \cdots \vee \neg C_n$ is written as

$$C \leftarrow C_1, \ldots, C_n$$

or as

$$C \texttt{ :- } C_1, \ldots, C_n.$$

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

357

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

```
:- path(a,d)
```

being the clause ¬`path(a,d)`

Systematical depth first search for a resolution proof starting from the goal yields:

358

1	$\texttt{arrow}(a, b)$	
2	$\texttt{arrow}(a, c)$	
3	$\texttt{arrow}(b, c)$	
4	$\texttt{arrow}(c, d)$	
5	$\texttt{path}(x, y) \vee \neg\texttt{arrow}(x, y)$	
6	$\texttt{path}(x, y) \vee \neg\texttt{arrow}(x, z) \vee \neg\texttt{path}(z, y)$	
7	$\neg\texttt{path}(a, d)$	
	$(5, 7)$ no result	
8	$\neg\texttt{arrow}(a, z) \vee \neg\texttt{path}(z, d)$	$(6, 7)$
9	$\neg\texttt{path}(b, d)$	$(1, 8)$
	$(5, 9)$ no result	
10	$\neg\texttt{arrow}(b, z) \vee \neg\texttt{path}(z, d)$	$(6, 9)$
11	$\neg\texttt{path}(c, d)$	$(3, 10)$
12	$\neg\texttt{arrow}(c, d)$	$(5, 11)$
	$\bot$	$(4, 12)$

359

This kind of search for a resolution proof is called **SLD-resolution**

The found resolution sequence ending in $\bot$ 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?'

360

This mechanism also applies for goals containing variables

For instance, the goal `:- path(a,X)` will yield a refutation, but the goal `:- 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

---

361

Back to general predicates $\cdots$

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

---

362

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
  - $\neg(A \wedge B) \equiv (\neg A) \vee (\neg B)$
  - $\neg(A \vee B) \equiv (\neg A) \wedge (\neg B)$
  - $\neg(\neg A) \equiv A$
  - $\neg(\forall x(A)) \equiv \exists x(\neg A)$
  - $\neg(\exists x(A)) \equiv \forall x(\neg A)$

363

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 quantors 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
  - $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
  - $(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$

364

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

$$\cdots \exists y \cdots (\cdots y \cdots)$$

by

$$\cdots\cdots (\cdots f(x_1, \ldots, x_n) \cdots)$$

where f is a fresh function symbol and $x_1, \ldots, x_n$ are the universally quantified variables left from y

---

365

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

$$\exists f \forall x (A(x, f(x))) \equiv \forall x \exists y (A(x, y))$$

366

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

---

367

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	$\neg A(c, d)$	$(1, 8)$
	$\bot$	$(6, 9)$

368

This is done automatically by the tool `Prover9`, by calling

```
./prover9 -f stud
```

where `stud` is a file containing

```
formulas(assumptions).
(exists x (S(x) & (L(y) -> A(x,y)))).
L(x) & B(x) ->
-(exists y (S(y) & A(y,x))).
end_of_list.
formulas(goals).
-(exists z (L(z) & B(z))).
end_of_list.
```

369

The Prolog graph example can also be done by `Prover9`:

```
formulas(assumptions).
arrow(a,b).
arrow(a,c).
arrow(b,c).
arrow(c,d).
arrow(x,y) -> path(x,y).
(arrow(x,z) & path(z,y)) -> path(x,y).
end_of_list.
formulas(goals).
path(a,d).
end_of_list.
```

By default, in `Prover9` x, y, z, u, v are universally quantified variables, and other names are constants, by which declarations may be omitted

---

370

Syntax:

negation: `-`
conjunction: `&`, disjunction: `|`
implication: `->`, bi-implication: `<->`
exists: `exists`, forall: `all`

Terms and variables represent elements of the (unknown) model, the only (but very useful) built-in predicate on this model is equality: `=`

```
formulas(assumptions).
f(x) = g(y).
h(x) = f(a).
end_of_list.
formulas(goals).
h(a) = f(b).
end_of_list.
```

371

In case `Prover9` fails, then may be the goal does not follow from the assumptions

One way to prove this is to find a model in which the assumptions hold but the goal does not hold

Automatically searching for a **finite** model with these properties may be done by the tool `Mace4`, accepting the same format

It can be proved that if the goal does not follow from the assumptions, then a model exists in which the assumptions hold but the goal does not hold, but not always a finite model exists

Conversely, it can be the case that the goal follows from the assumptions, but `Prover9` fails to prove this, so if both `Prover9` and `Mace4` fail then no conclusion can be drawn on validity

---

372

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:

$$0, s(0), s(s(0)), s(s(s(0))), \ldots$$

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

373

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 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:

$$0 + x = x$$

$$s(x) + y = s(x + y)$$

374

What is the meaning of such rules?

- For variables (here: x, y) arbitrary terms may be substituted
- These rules may be applied on any subterm 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)

375

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 $f(t_1, \ldots, t_n) \rightarrow_R f(u_1, \ldots, u_n)$

376

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:

$$\begin{aligned} s(s(0)) + s(s(0)) &\rightarrow_R s(\underbrace{s(0) + s(s(0))}) \\ &\rightarrow_R s(s(\underbrace{0 + s(s(0))})) \\ &\rightarrow_R s(s(s(s(0)))) \end{aligned}$$

377

Expressed in Prover9:

```
formulas(assumptions).
R(a(0,x),x).
R(a(s(x),y),s(a(x,y))).
R(x,y) -> R(a(x,z),a(y,z)).
R(x,y) -> R(a(z,x),a(z,y)).
R(x,y) -> R(s(x),s(y)).
RR(x,x).
(RR(x,y) & R(y,z)) -> RR(x,z).
end_of_list.
formulas(goals).
RR(a(s(s(0)),s(s(0))),s(s(s(s(0))))).
end_of_list.
```

`a` = plus operator
`R` = single rewrite step
`RR` = zero or more rewrite steps

378

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.

379

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

---

380

Example

$\text{rev}(\text{nil}) = \text{nil}$
$\text{rev}(a : x) = \text{conc}(\text{rev}(x), a : \text{nil})$
$\text{conc}(\text{nil}, x) = x$
$\text{conc}(a : x, y) = a : \text{conc}(x, y)$

Here a, x, y are variables, and $=$ corresponds to $\rightarrow$ in rewrite rules

Then we have a reduction to normal form
$\text{rev}(1{:}2{:}\text{nil}) \rightarrow$
$\text{conc}(\text{rev}(2{:}\text{nil}), 1{:}\text{nil}) \rightarrow$
$\text{conc}(\text{conc}(\text{rev}(\text{nil}), 2{:}\text{nil}), 1{:}\text{nil}) \rightarrow$
$\text{conc}(\text{conc}(\text{nil}, 2{:}\text{nil}), 1{:}\text{nil}) \rightarrow$
$\text{conc}(2{:}\text{nil}, 1{:}\text{nil}) \rightarrow$
$2{:}\text{conc}(\text{nil}, 1{:}\text{nil}) \rightarrow$
$2{:}1{:}\text{nil}$

381

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

---

382

More nice properties:

- R is **terminating** (= strongly normalizing, SN):

  no infinite sequence of terms $t_1, t_2, t_3, \ldots$ 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

---

383

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

$$a \to a \to a \to a \to \cdots$$

384

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 $u \to_R^* v$ and $u' \to_R^* v$

Since u, u' are normal forms we have $u = v = u'$

385

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 $\delta : Q \times S \to Q \times S \times \{L, R\}$

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

---

386

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

$$\cdots \square\square\square s'_m s'_{m-1} \cdots s'_1 s_1 s_2 \cdots s_{n-1} s_n \square\square\square \cdots$$

in which the Turing machine is in state q and reads symbol s_1 is represented by the term

$$q(s'_1(s'_2(\cdots(s'_m(b))\cdots)), s_1(s_2(\cdots(s_n(b))\cdots)))$$

387

For the Turing machine $M = (Q, S, \delta)$ the TRS $R(M)$ is defined to consist of the rules

$$q(x, s(y)) \to q'(s'(x), y)$$

for all $(q, s) \in Q \times S$ with $\delta(q, s) = (q', s', R)$, and

$$q(t(x), s(y)) \to q'(x, t(s'(y)))$$

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

388

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 $u \to_R v$

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

---

389

In our example

$$+(0, x) \to x$$

$$+(s(x), y) \to s(+(x, y))$$

we find such a weight function W by defining inductively

$W(0) = 1$
$W(s(t)) = W(t) + 1$
$W(t + u) = 2W(t) + W(u)$

390

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 $t \to_R u$, 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 $\ell \to r$ rather than for all rewrite steps

---

391

For every symbol f of arity n choose a **monotonic** function $[f] : \mathbf{N}^n \to \mathbf{N}$

Here **monotonic** means:

if for all $a_i, b_i \in \mathbf{N}$ for $i = 1, \ldots, n$ with $a_i > b_i$ for some i and $a_j = b_j$ for all $j \neq i$ then

$$[f](a_1, \ldots, a_n) \; > \; [f](b_1, \ldots, b_n)$$

392

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

393

For a map $\alpha : \mathcal{X} \to \mathbf{N}$ the weight function $[\cdot, \alpha] : T \to \mathbf{N}$ is defined inductively by

$$[x, \alpha] = \alpha(x),$$

$$[f(t_1, \ldots, t_n), \alpha] = [f]([t_1, \alpha], \ldots, [t_n, \alpha])$$

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 $\alpha : \mathcal{X} \to \mathbf{N}$ and every rule $\ell \to r$ in R

Then R is terminating

394

Example

For our TRS R consisting of the rules

$$+(0,x) \to x \quad +(s(x),y) \to s(+(x,y))$$

we choose monotonic functions

$$[0] = 1, \quad [s](x) = x + 1$$

$$[+](x,y) = 2x + y$$

Now indeed for every $\alpha : \mathcal{X} \to \mathbf{N}$ 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

395

Example

For the TRS R consisting of the single rule

$$f(g(x)) \to g(g(f(x)))$$

we choose monotonic functions

$$[f](x) = 3x$$

$$[g](x) = x + 1$$

Now indeed for every $\alpha : \mathcal{X} \to \mathbf{N}$ we have

$$[f(g(x)),\alpha] = 3(\alpha(x)+1) >$$

$$3\alpha(x) + 1 + 1 = [g(g(f(x))),\alpha]$$

proving termination

396

Example

The single rule $f(x) \to g(f(x))$ is not terminating, but by choosing

$$[f](x) = x + 1, \quad [g](x) = 0$$

for every $\alpha : \mathcal{X} \to \mathbf{N}$ 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

397

Extension of the approach:

Theorem

Let $R' \subseteq R$ be a TRSs for which R' is terminating

Let $[f]$ be chosen such that

- $[f]$ is monotonic for every symbol f, and
- $[\ell,\alpha] > [r,\alpha]$ for every $\alpha : \mathcal{X} \to \mathbf{N}$ and every rule $\ell \to r$ in $R \setminus R'$
- $[\ell,\alpha] \geq [r,\alpha]$ for every $\alpha : \mathcal{X} \to \mathbf{N}$ and every rule $\ell \to r$ in R'

Then R is terminating

398

Another technique: **lexicographic path order**

Choose an order $>$ on the set of function symbols

Theorem

If $\ell >_{lpo} r$ for all $\ell \to r$ in R, then R is terminating

Before this makes sense we have to define / characterize $>_{lpo}$

$f(t_1,\ldots,t_n) >_{lpo} u \iff$

- $\exists i : t_i = u \vee t_i >_{lpo} u$, or
- $u = g(u_1,\ldots,u_m)$ and
  $\forall i : f(t_1,\ldots,t_n) >_{lpo} u_i$ and either
  - $f > g$, or

– $f = g$ and
$(t_1, \ldots, t_n) >_{lpo}^{lex} (u_1, \ldots, u_m)$

399

Lemma
If s is a proper subterm of t, then $t >_{lpo} s$

Easily follows from first bullet

Example
For the rule

$$+(0, x) \to x$$

we have $+(0, x) >_{lpo} x$ by this lemma

For the rule

$$+(s(x), y) \to s(+(x, y))$$

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 $s(x) >_{lpo} x$

400

Hence $\ell >_{lpo} r$ for all rules $\ell \to r$, proving termination

More interesting example: **Ackermann function**

$$A(m, n) =$$

$$\begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Looks simple, but $A(4, 2)$ turns out to be an integer of 19,729 decimal digits

It is the simplest function that is not **primitive recursive**, and grows much harder than any primitive recursive function

Expressed as TRS:

$$\begin{array}{rcl} A(0, x) & \to & s(x) \\ A(s(x), 0) & \to & A(x, s(0)) \\ A(s(x), s(y)) & \to & A(x, A(s(x), y)) \end{array}$$

401

$$\begin{array}{rcl} A(0, x) & \to & s(x) \\ A(s(x), 0) & \to & A(x, s(0)) \\ A(s(x), s(y)) & \to & A(x, A(s(x), y)) \end{array}$$

Termination can be proved by lexicographic path order, $A > s$

So termination theoretically holds, but the normal form of $A(s(s(s(0)))), s(s(0)))$ is a term containing N symbols s, for N being a number of 19,729 decimal digits, so the normal form does not fit in all computer memory of the world

---

402

Checking termination by lexicographic path order is easy to implement; do not choose $>$ in advance but collect requirements on $>$ during the process of proving $\ell >_{lpo} r$

Monotone interpretations and path orders are building blocks in proving termination, exploited in many other techniques

The most important of these is **Dependency pairs**

First we will investigate weakness of the techniques we considered until now

A TRS R is called **simply terminating** if $R \cup E$ is terminating, for E being the TRS consisting of the rules

$$f(x_1, \ldots, x_n) \to x_i$$

for all symbols f in R of arity ≥ 1

403

Example

For $R = \{f(f(x)) \to f(g(f(x)))\}$ we have $E = \{f(x) \to x,\ g(x) \to x\}$

R is not simply terminating since in $R \cup E$ we have

$$f(f(x)) \to_R f(g(f(x))) \to_E f(f(x)) \to \cdots$$

But R is terminating, since in every step the number of $f(f(\cdots))$ patterns strictly decreases

If R is not simply terminating, then both lexicographic path order and monotone interpretations fail to prove termination of R

How to prove termination of the above TRS R automatically?

404

Dependency pairs

For a TRS R a symbol f is called a **defined symbol** if f is the root symbol of a left hand side of a rule of R

Symbols f are written in lower case letters, for every defined symbol f a new capitalized symbol F is added having the same arity as f

If $f(s_1, \ldots, s_n) \to C[g(t_1, \ldots, t_m)]$ is a rule in R and g is a defined symbol of R, then

$$\langle F(s_1, \ldots, s_n), G(t_1, \ldots, t_m) \rangle$$

is called a **dependency pair** of R

An infinite sequence $(\langle s_i, t_i \rangle)_{i=1,2,3,\ldots}$ of dependency pairs of a TRS R is called an **infinite R-chain** if subsititutions σ_i exist such that $t_i^{\sigma_i} \to_R^* s_{i+1}^{\sigma_{i+1}}$ for every $i = 1, 2, 3, \ldots$

405

Theorem

A TRS R is terminating if and only if it does not admit an infinite R-chain

Example Consider the TRS R consisting of the rule:

$$f(f(x)) \ \to \ f(g(f(x)))$$

Here f is the only defined symbol, hence the signature is extended by one unary symbol F

In the right hand side two copies of the defined symbol f occur, hence there are two dependency pairs:

$$\langle F(f(x)), F(g(f(x)) \rangle \quad \text{and} \quad \langle F(f(x)), F(x) \rangle$$

406

Two dependency pairs:

$$\langle F(f(x)), F(g(f(x)) \rangle \quad \text{and} \quad \langle F(f(x)), F(x) \rangle$$

Assume that $(\langle s_i, t_i \rangle)_{i=1,2,3,\ldots}$ is an infinite R-chain

Since no substitutions σ, τ exist satisfying $F(g(f(x))^\sigma \to_R^* F(f(x))^\tau$, we conclude that $s_i = F(f(x))$ and $t_i = F(x)$ for all $i = 1, 2, 3, \ldots$

Now from $F(x)^{\sigma_i} \to_R^* F(f(x))^{\sigma_{i+1}}$ we conclude that x^{σ_i} contains one more f-symbol than $x^{\sigma_{i+1}}$ for every $i = 1, 2, 3, \ldots$, contradiction

Hence by the theorem we have proved that R is terminating

407

More general:

Dependency pairs can be seen as the nodes of a **dependency graph**, where an arrow from a dependency pair $\langle s, t \rangle$ to dependency pair $\langle u, v \rangle$ is drawn if substitutions σ, τ exist satisfying $t^\sigma \to_R^* u^\tau$

Precisely computing the arrows may be hard, but approximation is feasible

In our example the dependency graph consists of two nodes $\langle F(f(x)), F(g(f(x))) \rangle$ and $\langle F(f(x)), F(x) \rangle$, and two arrows: one from $\langle F(f(x)), F(x) \rangle$ to $\langle F(f(x)), F(g(f(x))) \rangle$ and one from $\langle F(f(x)), F(x) \rangle$ to itself

---

408

Any infinite chain gives rise to an infinite path in the dependency graph

Since the dependency graph is finite, an infinite chain exists if and only an infinite chain exists only involving dependency pairs that are on a cycle of the dependency graph

So the analysis restricts to **strongly connected components** of the dependency graph

That's why in our example the dependency pair $\langle F(f(x)), F(g(f(x))) \rangle$ may be ignored

---

409

In proving non-existence of infinite paths in these strongly connected components automatically, both path orders and monotone interpretations play a key role

Several tools have been developed by which termination of a TRS can be proved fully automatically: AProVE, TTT2, exploiting all of these techniques

Techniques for proving termination of TRSs also form the basis of several techniques for automatically proving termination of programs

---

410

Back to the other properties

Confluence is strictly stronger than local confluence:

$$a \to b, \quad b \to a, \quad a \to c, \quad b \to d$$

is locally confluent:

if $t \to_R u$ and $t \to_R v$ then either

- $t = a$, then choose $w = c$, or
- $t = b$, then choose $w = d$

In both cases we conclude $u \to_R^* w$ and $v \to_R^* w$

but not confluent:

for $t = a, u = c, v = d$ we have $t \to_R^* u$ and $t \to_R^* v$, but no w exists satisfying $u \to_R^* w$ and $v \to_R^* w$

411

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 SN($\to$), CR($\to$) and WCR($\to$) all can be defined for arbitrary binary relations $\to$, in which general setting we will prove Newman's lemma

So SN($\to$) simply means the non-existence of an infinite sequence $t_1 \to t_2 \to t_3 \to \cdots$

We write $\to^+$ for the transitive closure of $\to$: one or more steps

---

412

Principle of well-founded induction

Theorem

Let SN($\to$) and

$\forall t(\underbrace{\forall u(t \to^+ u \Rightarrow P(u))}_{\text{Induction Hypothesis}} \Rightarrow P(t))$

Then $P(t)$ holds for all t

(think of $t \to^+ u$ 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 $\neg\forall u(t \to^+ u \Rightarrow P(u))$, yielding u such that $t \to^+ u$ and $\neg P(u)$

Repeat the argument for u, yielding a v, and so on, so yielding an infinite sequence

$$t \to^+ u \to^+ v \to^+ \cdots$$

contradicting SN($\to$) (End of proof)

413

Proof of Newman's Lemma

Assume SN($\to$) and WCR($\to$), we have to prove CR($\to$)

We apply the principle of well-founded induction for $P(t)$ being

$$\forall u, v : \text{if } t \to^* u \wedge t \to^* v \text{ then}$$
$$\exists w : u \to^* w \wedge v \to^* w$$

So assume $t \to^* u$ and $t \to^* v$; we have to find w such that $u \to^* w$ and $v \to^* w$

If $t = u$ we may choose $w = v$

if $t = v$ we may choose $w = u$

In the remaining case we have $t \to^+ u$ and $t \to^+ v$

Write $t \to u_1 \to^* u$ and $t \to v_1 \to^* v$

414

Using WCR there exists w_1 such that $u_1 \to^* w_1$ and $v_1 \to^* w_1$

Using the induction hypothesis on u_1 there exists w_2 such that $w_1 \to^* w_2$ and $u \to^* w_2$

Now we have $v_1 \to^* w_2$ and $v_1 \to^* v$; using the induction hypothesis on v_1 there exists w such that $w_2 \to^* w$ and $v \to^* w$

t	$\to$	u_1	$\to^*$	u
$\downarrow$	WCR	$\downarrow_*$	IH	$\downarrow_*$
v_1	$\to^*$	w_1	$\to^*$	w_2
$\downarrow_*$		IH		$\downarrow_*$
v		$\to^*$		w

Since $u \to^* w_2$ we have $u \to^* w$, and we are done

415

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

---

416

Definition of critical pairs

Let $\ell_1 \to r_1$ and $\ell_2 \to r_2$ 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**

417

Example

Assume we have rules for arithmetic including

$$\begin{array}{lcl} x - x & \to & 0 \\ s(x) - y & \to & s(x - y) \end{array}$$

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

- $\ell_1 \to r_1$ to be the rule $z - z \to 0$
- $\ell_2 \to r_2$ to be the rule $s(x)-y \to s(x-y)$
- $t = \ell_2 = s(x) - y$

Indeed t, ℓ_1 unify, with mgu σ:

$$\sigma(x) = x, \;\; \sigma(y) = \sigma(z) = s(x)$$

418

Example

Let R consist of the single rule

$$f(f(x)) \to g(x)$$

By choosing

- $\ell_1 \to r_1$ to be the rule $f(f(x)) \to g(x)$
- $\ell_2 \to r_2$ to be the rule $f(f(y)) \to g(y)$
- $t = f(y)$

we see that t, ℓ_1 unify, with mgu σ:

$$\sigma(x) = x, \;\; \sigma(y) = f(x)$$

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 $t \to_R^* v$ and $u \to_R^* v$

419

Theorem

A TRS R is locally confluent if and only if all critical pairs converge

Example

The single rewrite rule $f(f(x)) \to g(x)$ is not locally confluent, so neither confluent, since for its critical pair $[f(g(x)), g(f(x))]$ no term v exists such that

$$f(g(x)) \to_R^* v \quad \text{and} \quad g(f(x)) \to_R^* v$$

This is immediate from the observation that both $f(g(x))$ and $g(f(x))$ are normal forms

---

420

For a term t and a TRS R define

$$S(t) = \{v \mid t \to_R^* v\}$$

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 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 WCR(R) does not hold
- If all of these sets are non-empty then WCR(R) holds

421

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:

$$\begin{array}{rcl} d(x,x) & \to & b \\ c(x) & \to & d(x, c(x)) \\ a & \to & c(a) \end{array}$$

has no overlap but is not confluent:

$$c(a) \to_R d(a, c(a)) \to_R d(c(a), c(a)) \to_R b$$

$$c(a) \to_R c(c(a)) \to_R^+ c(b)$$

while $[b, c(b)]$ does not converge

422

Write $\leftrightarrow_R^*$ for the reflexive symmetric transitive closure of $\to_R$, i.e., $t \leftrightarrow_R^* u$ holds if and only if terms $t_1, \ldots, t_n$ exist for $n \geq 1$ such that

- $t_1 = t$
- $t_n = u$
- For every $i = 1, \ldots, n-1$ either $t_i \to_R t_{i+1}$ or $t_{i+1} \to_R t_i$ 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

---

423

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 $t' = u'$

424

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 $t \to_R^* v$ and $u \to_R^* v$

Proof of the theorem:

If $t' = u'$ then $t \to_R^* t' = u' \leftarrow_R^* u$, hence $t \leftrightarrow_R^* u$

Conversely assume $t \leftrightarrow_R^* u$

Then $t' \leftarrow_R^* t \leftrightarrow_R^* u \to_R^* u'$, hence $t' \leftrightarrow_R^* u'$

According the lemma a term v exists such that $t' \to_R^* v$ and $u' \to_R^* v$

Since t', u' are normal forms we have $t' = v = u'$ End of proof

---

425

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 $t' = u'$

426

Example:

R consists of the rule $s(s(s(x))) \to x$

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**

427

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

428

Idea of Knuth-Bendix completion

Fix a well-founded order $>$ on terms, i.e., SN($>$), that has some closedness properties:

- if $t > u$ then $t\sigma > u\sigma$ for every substitution σ
- if $t > u$ then $f(\ldots, t, \ldots) > f(\ldots, u, \ldots)$ 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 $\ell \to r$ in R,
> then SN(R)

A typical example of a reduction order is a lexicographic path order

---

429

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 $t \to u$ to R if $t > u$
- add $u \to t$ to R if $u > t$
- give up otherwise

After adding any new rule $\ell \to r$ 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

430

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 $\ell \to r$ 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

---

431

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 $f(f(x)) \to g(x)$ 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

432

Since

$$f(g(x)) >_{lpo} g(f(x))$$

we add the rule $f(g(x)) \to g(f(x))$ to the TRS R

Together with the older rule $f(f(x)) \to g(x)$ we get the critical pair $[f(g(f(x))), g(g(x))]$

Since $g(g(x))$ is a normal form and

$$f(g(f(x))) \to_R g(f(f(x))) \to_R g(g(x))$$

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

$$f(f(x)) \to g(x), \quad f(g(x)) \to g(f(x))$$

having the same convertibility relation as the original equation $f(f(x)) = g(x)$

433

Example:

For decision trees we consider the set E of equations

$$\begin{array}{rcl} p(x,x) & = & x \\ p(q(x,y),q(z,w)) & = & q(p(x,z),p(y,w)) \\ p(p(x,y),z) & = & p(x,z) \\ p(x,p(y,z)) & = & p(x,z) \end{array}$$

where p, q runs over all boolean variables

It can be proved that two decision trees represent the same boolean function if and only 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

434

$$\begin{array}{rcll} p(x,x) & \to & x & \text{for all } p \\ p(p(x,y),z) & \to & p(x,z) & \text{for all } p \\ p(x,p(y,z)) & \to & p(x,z) & \text{for all } p \\ p(q(x,y),z) & \to & q(p(x,z),p(y,z)) & \text{for } p > q \\ p(x,q(y,z)) & \to & q(p(x,y),p(x,z)) & \text{for } p > q \end{array}$$

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

Overview of the course

- Proposition logic
  - SAT by resolution: DP and DPLL and CDCL
  - Tseitin transformation
  - (RO)BDDs
  - Symbolic model checking: `NuSMV`
- Extension to SMT
  - Tools: `Yices` and `Z3`
  - Underlying mechanism: linear programming, simplex algorithm, combined with CDCL
- Predicate logic
  - Resolution, unification
  - Prenex normal form, Skolemization
  - Prolog, `Prover9`
- Equational reasoning by term rewriting
  - termination by monotonic interpretation or lexicographic path order
  - (local) confluence by critical pair analysis
  - Knuth-Bendix completion