1 Model Checking problem

input:

- formula ϕ
- structure S defining a semantic for the tokens of the formular

output:
does the formula hold over ϕ

validity satifiability
input:

- formula $\phi \rightarrow$ checker \rightarrow yes no
  output:
  yes/no does ϕ hold over all possible S
  ϕ needs to hold over all S and is more strict than

logical equivalence:
input:
formula ϕ_1
formula ϕ_n
output:
yes: if ϕ_1 till ϕ_n hold over all S

usefull for:
if ϕ_1 is very complex and ϕ_2 is super trivial. If it turns out that they are logically equivalent we can use ϕ_2 instead of ϕ_1 and save a lot of calculations.

equivalence reduces to validity satifiability

> **Note**
>
> $\models$ is equivalent to $\Rightarrow$
> and means from that the Righthand side is the logical consequence of the Lefthand side

1.1 definability between logics

input:

- formula ϕ_1
- Logic L_1

Question:
can you rewrite ϕ_1 valid in L_1 to another formula ϕ_2 in L_2 for example because maybe L_2 has better algorithmical complexity.

2 Propositional Logic

vocabulary:

- $\sum = \{p, q, r\}$ are boolean variables
- $\vee, \wedge, \neg, \implies, \iff$ are boolean connectives

syntax
grammar: what is a valid formula
examples : $p|q|, |\phi \wedge \phi|\neg|\phi \implies \phi|$

all formulas can also be represented by a syntactic tree
example $p \wedge (\neg q \vee r)$ is

semantics:
requires a structure $S : \sum \implies \{true, false\}$
describes when ϕ holds on S (denoted as $S \models \phi$)

examples for the application of $\models$

$$S \models p \iff S(p) = True$$

$$S \models \phi_1 \wedge \phi_2 \iff S \models \phi_1 \text{ and } S \models \phi_2$$

$$S \models \phi_1 \vee \phi_2 \iff S \models \phi_1 \text{ or } S \models \phi_2$$

3 Model Checking

We write a program that check if the above statements are fulfilled

```
def Model-check(phi,S):
        if phi = p:
                return S(p)
        elif phi = phi_1 or phi_2:
                return Model-check(phi_1,S)
                                or Model-check(phi_2,s)
        elif phi = phi_1 and phi_2:
                return Model-check(phi_1,S)
                                and Model-check(phi_2,s)
        elif ...
```

What is the complexity of this algorithm is polynomial:
size of the formula -> number of rows of the syntactic tree
times the size of the structure -> polynomial

Name EVAL and is p-complete

4 Satisfiability

Short name: SAT
We look for one Structure of S where the

```
def Satisfiable(phi):
        let p1,p2,... = all propositional variables in phi

        for v1 = true,false do
                for v2 = true,false do
                ...
                        S=[p1:v1,p2:v2]
                        if Model-check(phi,S):
                                return true
        return false
```

The complexity is exponential time because you have to try every possible configuration of variables.

One can guess the inputs vx to make it non-deterministic. Then the complexity is not exponential.

#Validity
We look if the formula ϕ is valid in all Structures S

```
def Valid(phi):
        let p1,p2,... = all propositional variables in phi

        for v1 = true,false do
                for v2 = true,false do
                ...
                        S=[p1:v1,p2:v2]
                        if not Model-check(phi,S):
                                return false
        return true
```

The complexity is exponential time because you have to try every possible configuration of variables.

One can guess the inputs vx to make it non deterministic.

5 Economy of Syntax

Lemma 1:
Every formula is equivalent to one without $\iff$ and $\implies$

prove:
$\phi_1 \implies \phi_2$ is equivalent to $(\neg\phi_1) \vee \phi_2$

$\phi_1 \iff \phi_2$ is equivalent to

$$(\phi_1 \wedge \phi_2) \wedge (\neg\phi_1 \wedge \neg\phi_2))$$

Lemma 2:
Every formula is equivalent to one in Negation Normal Form.
The Negation Normal Form can be computed efficiently if there is no $\iff$.

$$\neg(\phi_1 \vee \phi_2)$$

proof:

- $\neg(\phi_1 \vee \phi_2)$ becomes $(\neg\phi_1 \wedge \neg\phi_2)$
- $\neg(\phi_1 \wedge \phi_2)$ becomes $(\neg\phi_1 \vee \neg\phi_2)$

see: Morgans Laws

Lemma 3:
Every formula is equi-sartisfiable to one in Conjunctive Normal Form. The Conjunctive Normal Form can be computed efficiently if there is no $\iff$

proof: exercise

Corollary 1:
CNF-SAT is NP-complete

6 Tableaux

We want to check Satisfiability
Motto:instead of guessing the structure we guess pieces of the formula that are easy to satisfy.

A Tableaux is usually a tree.

Another example:

Rules:
One can only break up on conjunction or disjunction every level.
The choice changes the tableaux but the result is always the same.

1. ϕ needs to be in the Negation Normal Form -> to keep the $\neg$ next to the variables
2. start with a singleton tree $\{\phi\}$ which in other representation is $\{\alpha_0\}$ with one single leave F_0
   for each leave that is not *unblocked*:
3. Choose a part of the formula α_n (a sub-formula)
4. expand the tree under F_n by adding
   - if α_n has the form of $\beta \wedge \beta'$
     - $F_n + 1$ = $\{F \text{ ohne } \alpha, \beta, \beta'\}$
   - if α_n has the form of $\beta \vee \beta'$ add two children
     - F_{n+1} = $\{F \text{ ohne } \alpha, \beta\}$
     - F_{n+1} = $\{F \text{ ohne } \alpha, \beta'\}$
     The formula is not
     The formula is **invalid** if one branch contains a contradiction for instance $\{p, \neg p\}$
     The formula is **valid** if no branch contains a contradiction

6.1 Example/ Application:

Consider a device whose internal state is encoded by k bits $\underline{p} = p_1, \ldots, p_n$

- initial state is described by a propositional formula $\phi_{init}(\underline{p})$
- target state is described by a propositional formula $\phi_{target}(\underline{p})$
- Transitional formulas are described by a propositional formula $\phi_{step}(\underline{p}, \underline{p}')$

Question: can we go from $\phi_{init}(\underline{p})$ to $\phi_{target}(\underline{p})$ by only applying $\phi_{step}(\underline{p}, \underline{p}')$

Maximum length of steps is 2^k as there are ony 2^k possible variations when having k bits.

The formula describes it:

$$\phi_{reach}(\underline{p_1}, \dots, \underline{p_n}) = \phi_{init}(\underline{p_1}) \wedge \phi_{step}(\underline{p_1}, \underline{p_2}) \wedge \ \dots \ \wedge \phi_{step}(\underline{p_{n-2}}, \underline{p_{n-1}}) \wedge \phi_{target}(\underline{p_n})$$

If this formula is satisfiable there is a way from *init* to *target*

7 QBF Quantified Boolean Formulas

Vocabulary:

- Boolean variables $\sum = \{p, q, r \dots\}$
- Boolean connectives $\wedge, \vee, \neg, \iff, \implies$
- $\forall, \exists$ Quantifiers

Syntax:

$$\phi = p|q|\dots|\phi \wedge \phi|\phi \vee \phi|\neg\phi|\phi \iff \phi|\phi \implies \phi|\exists p\phi|\forall p\phi$$

semantics :
describes when $S \models \phi$ for $S : \sum \implies \{\text{True,False}\}$

$S \models p$	$\iff$	$S(p) = True$
$S \models \phi_1 \wedge \phi_2$	$\iff$	$S \models \phi_1 \text{ or } S \models \phi_2$
...		
$S \models \exists p\phi$	$\iff$	$S' \models \phi \text{ for some } S' \in \{S[p := \text{true}], S[p := \text{false}]\}$
$S \models \exists p\phi$	$\iff$	$S' \models \phi \text{ for every } S' \in \{S[p := \text{true}], S[p := \text{false}]\}$

The $\forall$ and $\exists$ means that we fix one of the variables to either true or false and reevaluate it with this fixed value. If S is valid for one of the set values one uses $\exists$ if the structure holds for all one uses $\forall$.

> **Note**
>
> $\exists p\phi$ is logically equivalente to $\neg\forall p\neg\phi$

Example:
$S \overset{?}{\models} \forall p\exists q(p \vee \neg q) \wedge (\neg p \vee q)$

$S_1 = [p := true] \models \forall p\exists q(p \vee \neg q) \wedge (\neg p \vee q)$
two possibilities for S_1 of which one but be satisfied:
$S_1' = [p = true, q = false]$ result with this input is not satisfied
$S_1'' = [p = true, q = true]$ result with this input is satisfied => True

$S_2 = [p := false] \models \forall p\exists q(p \vee \neg q) \wedge (\neg p \vee q)$
Same here, one of the two options need to be satisfieed
$S_2' = [p = false, q = false]$ result with this input is satisfied =>true
$S_2'' = [p = false, q = true]$ result with this input is not satisfied=>false

That means that means that in case S'' there does not exist one q that makes the equation valid that means the whole statement is wrong

Bound and free variables

- a variable p is considered bound when it appears under a scope of a quantor i.e. $\exists p$ or $\forall p$
- a variable is considered free when it is not in the a scope under quantor. i.e. in all other cases

Example:
the first occurence of p is free. The third occurence of p ($\neg p$) is bound to the $\exists p$.
$\phi = p.\ldots.\exists p.\ldots(\neg p)$

When we write $\phi(p_1, p_2, \ldots, p_n)$ we mean that the free variables of ϕ are $p_1, p_2, \ldots, p_n$

When we write $\phi[p/\alpha]$ we denote a formula where we replace all occurences of p by α. Another Notation could be when we first write $\phi(p)$ that to denote that we exchange p by α we write :$\phi[\alpha]$. Take care square brackets!

Example:
$\phi = \neg p \vee \exists p(p \wedge q)$
When we write $\phi[p/\alpha]$ we only replace the free occurences of p
i.e:
$\phi = \neg \alpha \vee \exists p(p \wedge q)$

Lemma 4 (renaming)

$\exists p \phi$ is equivalent to $\exists q \phi[p/q]$ if q does not appear free in ϕ