Resume : Formal Methods

Sylvain Julmy

January 30, 2018

Chapter 1

Hoare Logic

1.1 Hoare Triple

Logical formulas can be used to express information about program states. We called $\{P\}S\{Q\}$ a "Hoare Triple" :

- A precondition P what can be assumed to be true before executing a sequence of statements S.
- A postcondition Q states what will be true after the execution of S.

We write $\{P\}S\{Q\}$ to indicate : if P is true, then executing S will make Q true.
We also use the notation x' to denote the variable x after the execution of S. For example

$$\{true\}\ x = x + 1;\ \{x' = x + 1\}$$

is a valid Hoare Triple in which x' is the value of x after the execution of S. We can write any predicate in between any two lines of code (an assertion), we assume that such a predicate is the postcondition of the previous line and the precondition of the following line.

1.2 Correctness of Hoare Triple

We translate our program S into a formula ϕ_S and then, we check

$$P \wedge \phi_S \rightarrow Q$$

or, equivalently

$$\phi_S \rightarrow (P \rightarrow Q)$$

So, for example, we turn the following Hoare Triple

$$\{x \neq 0\}\ x = 1/x;\ x = 1/x;\ \{x' = x\}$$

into the following formula (also using the primed notation) :

$$\underbrace{x \neq 0}_{P} \wedge \underbrace{x'' = 1/x \wedge x' = 1/x''}_{S} \rightarrow \underbrace{x' = x}_{Q}$$

Which is true by elementary arithmetic.

1.2.1 If Clauses

We turn

$$\{P\}\ if(condition)\ \{prog1\}\ else\ \{prog2\};\ \{Q\}$$

into

$$\{P \wedge condition\}\ prog1\ \{Q\}$$

and

$$\{P \wedge \neg condition\}\ prog2\ \{Q\}$$

Both of those Hoare triple must be true for the if clause to be correct.

1.2.2 Loops Clauses

In order to check the total correctness of a program with loops, we have to check the partial correctness and the termination of the program. Such a program is in the following form :

$$\{P\}\ initialisation;\ while\ (condition)\ \{loop\ body\};\ \{Q\}$$

Partial correctness

A loop invariant is a logical formula that is true

- before the loop,
- before each execution of the loop body,
- after each execution of the loop body,
- after the loop.

Then, from

$$\{P\}\ initialisation;\ while\ (condition)\ \{loop\ body\};\ \{Q\}$$

we get

$$\{P\}\ initialisation;\ \{inv\}$$
$$\{inv \wedge condition\}\ loop\ body;\ \{inv\}$$
$$\{inv \wedge \neg condition\}\ skip;\ \{Q\}$$

Termination

A loop variant is an integer-valued expression that

- is decreased at least by 1 in each execution of the loop body,
- cannot go below 0.

Then, from

$$\{P\}\ initialisation;\ while\ (condition)\ \{loop\ body\};\ \{Q\}$$

we get

$$\{int\ var \wedge var > 0\}\ loop\ body;\ \{var > var' \geq 0\}$$

Example

$$\begin{array}{ll} \{n > 0 \wedge x = 1\} & sum = 1; \\ & while\ (x < n)\ \{ \\ & x = x + 1; \\ & sum = sum + x; \\ & \}; \{sum = n(n+1)/2\} \end{array}$$

For the invariant, we try $sum = x(x+1)/2$ and we get the following Hoare Triples :

$$\{n > 0 \wedge x = 1\}\ sum = 1;\ \{sum = x(x+1)/2\}$$
$$\{sum = x(x+1)/2 \wedge x < n\}\ x = x + 1;\ sum = sum + x\ \{sum = x(x+1)/2\}$$
$$\{sum = x(x+1)/2 \wedge \neg(x < n)\}\ skip;\ \{sum = n(n+1)/2\}$$

(1) : We obtain the following formula to prove :

$$n > 0 \wedge x = 1 \wedge sum = 1 \rightarrow sum = \frac{x(x+1)}{2}$$

$1 = \frac{1(1+1)}{2} = \frac{2}{2} = 1$

(2) : We obtain the following formula to prove :

$$sum = x(x+1)/2 \wedge x < n \wedge x' = x + 1 \wedge sum' = sum + x' \rightarrow sum' = x'(x'+1)/2$$

$$sum = \frac{x(x+1)}{2}$$
$$x < n$$
$$x' = x + 1$$
$$sum' = sum + x' = sum + x + 1 = \frac{x(x+1)}{2} + x + 1$$
$$sum' = \frac{x'(x'+1)}{2}$$
$$\frac{x(x+1)}{2} + x + 1 = \frac{(x+1)(x+1+1)}{2}$$
$$\frac{x(x+1)}{2} + x + 1 = \frac{x^2+x}{2} + x + 1 = 0.5x^2 + 1.5x^2 + 1$$
$$\frac{(x+1)(x+1+1)}{2} = \frac{x^2+2+2x+x}{2} = \frac{x^2+3x+2}{2} = 0.5x^2 + 1.5x + 1$$

(3) : We obtain the following formula to prove :

$$sum = \frac{x(x+1)}{2} \wedge x \geq n \rightarrow sum = \frac{n(n+1)}{2}$$

Which is true because if $x = n$ then both side of the equation are equivalent.

termination : we try $n - x$ for the variant and we got the following formula :

$$
\begin{aligned}
&int\ var \wedge \\
&var > 0 \wedge \\
&x = 1 \wedge \\
&n > 0 \wedge \\
&var = n - x \wedge \\
&x' = x + 1 \wedge \\
&sum' = sum + x' \wedge \\
&var' = n - x' \ \rightarrow \\
&var > var' \geq 0
\end{aligned}
$$

$$var' = n - x' = n - x + 1$$

$$n - x > n - x - 1 \geq 0$$

Termination is proved.

1.3 Weakness and Strength of Predicates

P is weaker than $Q \leftrightarrow Q \rightarrow P$ ($\leftrightarrow$ stand for if and only if). true is the weakest predicate and false is the strongest one.
If P is weaker than P' ($P' \rightarrow P$), then proving $\{P\}\ S\ \{Q\}$ guarantees the truth of $\{P'\}\ S\ \{Q\}$.
If Q is stronger than Q' ($Q \rightarrow Q'$), then proving $\{P\}\ S\ \{Q\}$ guarantees the truth of $\{P\}\ S\ \{Q'\}$.

Example : assume that we have proved

$$
\begin{aligned}
&\{x > 0\} \\
&x = 1/x; \\
&\{x' > 0\} \\
&\{x \neq 0\} \\
&\text{if } (x < 0) \text{then } x = -x \text{ else } x = x; \\
&\{x' > 0\}
\end{aligned}
$$

How to prove

$$
\begin{aligned}
&\{x > 0\} \\
&x = 1/x; \\
&\text{if } (x < 0) \text{then } x = -x \text{ else } x = x; \\
&\{x' > 0\}
\end{aligned}
$$

We can assume that both of the assertion present in the first one are just replaced by the assertion $\{true\}$. Then, we would have $\{x > 0\}\ x = 1/x\ \{true\}$ where $Q' = \{true\}$. From

the first one, we can extract the Hoare Triple $\{x > 0\}\ x = 1/x\ \{x' > 0 \wedge x \neq 0\}$, where $Q = \{x' > 0 \wedge x \neq 0\}$. Q is stronger that Q' and $\{P\}\ S\ \{Q\}$ is proved, then $\{P\}\ S\ \{Q'\}$ is proved too.

Chapter 2

Propositional Logic

A formula in propositional logic can be constructed as follows :

- $\top$ (true), $\bot$ (false) and propositional variables $(q, p, m, m, \dots)$ are formulas of propositional logic.
- if F and G are formulas of propositional logic, then so are :
  - (F)
  - $\neg F$
  - $F \wedge G$
  - $F \vee G$
  - $F \to G$
  - $F \leftrightarrow G$

Definition. *If P is a variable, then P and $\neg P$ are called **literals**.*

Definition. *An **interpretation** I is a truth-value assignment to propositional variables $P, Q, \dots$*

$$I : \{P \mapsto \top, Q \mapsto \bot, \dots\}$$

Definition. *The truth value of a variable P under an interpretation I is denoted by $I[P]$. For example, we would have*

$$I[P] = \top, I[Q] = \bot, \dots$$

Definition. *For an interpretation I, we write*

$$I \models F$$

if and only if propositional formula F is true under the interpretation I.

Definition. *We have the following for F and G being a propositional variables :*

- $I \models \top$
- $I \not\models \bot$
- $I \models F$ *iff* $I[F] = \top$
- $I \not\models F$ *iff* $I[F] = \bot$

- $I \models (F)$ *iff* $I \models F$
- $I \models \neg F$ *iff* $I \not\models F$
- $I \models F \wedge G$ *iff* $I \models F$ *and* $I \models G$
- $I \models F \vee G$ *iff* $I \models F$ *or* $I \models G$ *or both*
- $I \models F \to G$ *iff* $I \not\models F$ *or* $I \models G$ *or both*
- $I \models F \leftrightarrow G$ *iff* $I \models F \to G$ *and* $I \models G \to F$

2.1 Proof rules

$$\frac{I \models \neg F}{I \not\models F} \qquad \frac{I \models \neg F}{I \models F} \qquad \frac{I \models F \wedge G}{\begin{array}{c} I \models F \\ I \models G \end{array}} \qquad \frac{I \not\models F \wedge G}{I \not\models F \qquad I \not\models G} \qquad \frac{I \models F \vee G}{I \models F \qquad I \models G}$$

$$\frac{I \not\models F \vee G}{\begin{array}{c} I \not\models F \\ I \not\models G \end{array}} \qquad \frac{I \models F \to G}{I \not\models F \qquad I \models G} \qquad \frac{I \not\models F \to G}{\begin{array}{c} I \models F \\ I \not\models G \end{array}} \qquad \frac{I \models F \leftrightarrow G}{I \models F \wedge G \qquad I \not\models F \vee G}$$

$$\frac{I \not\models F \leftrightarrow G}{I \models \neg F \wedge G \qquad I \models F \wedge \neg G} \qquad \frac{\begin{array}{c} I \models F \\ I \not\models F \end{array}}{I \models \top}$$

Definition. *A propositional formula F is* ***satisfiable*** *if and only if there exists an interpretation I such that $I \models F$.*

Definition. *A propositional formula F is* ***valid*** *if and only if for all interpretations I, we have $I \models F$.*

Definition. *F is* ***valid*** *if and only if $\neg F$ is not satisfiable.*

Definition. *F is* ***satisfiable*** *if and only if $\neg F$ is not valid.*

Example : Prove the validity of $(P \wedge (P \to Q)) \to Q$:
We prove the satisfiability of $\neg((P \wedge (P \to Q)) \to Q)$ to prove the validity of $(P \wedge (P \to Q)) \to Q$.

1. $I \models \neg((P \wedge (P \to Q)) \to Q)$
2. $I \not\models (P \wedge (P \to Q)) \to Q$
3. $I \models P \wedge (P \to Q)$ and $I \not\models Q$
4. $I \models P$ and $I \models P \to Q$ and $I \not\models Q$

5. $I \models P$ and $I \not\models Q$ and $I \not\models P$ | $I \models P$ and $I \not\models Q$ and $I \models Q$
6. $I \models \bot$ | $I \models \bot$

$\neg((P \wedge (P \to Q)) \to Q)$ is not satisfiable, then $(P \wedge (P \to Q)) \to Q$ is valid.

2.2 Equivalence and Implication of Formulas

Definition. *Two formulas F and G are **equivalent** (written : $F \Leftrightarrow G$) if and only if $F \leftrightarrow G$ is a valid formula.*

Definition. *Formula F **implies** formula G (written $F \Rightarrow G$) if and only if $F \rightarrow G$ is a valid formula.*

Example : Does $P \wedge (P \rightarrow Q)$ imply Q ?
We just prove that $(P \wedge (P \rightarrow Q)) \rightarrow Q$ is a valid formula, then $P \wedge (P \rightarrow Q) \Rightarrow Q$.

2.3 Substitutions

Definition. *A substitution σ is a mapping from formulas to formulas :*

$$\{F_1 \mapsto G_1, F_2 \mapsto G_2, \ldots, F_n \mapsto G_n\}$$

*We denote σ's **domain** by $domain(\sigma) = \{F_1, F_2, \ldots, F_n\}$ and σ's **range** by $range(\sigma) = \{G_1, G_2, \ldots, G_n\}$.*
*If the domain of σ consists solely of single variables, then σ is valled a **variable substitution**.*

If we apply a substitution $\sigma = \{F_1 \mapsto G_1, F_2 \mapsto G_2, \ldots, F_n \mapsto G_n\}$ to formula F, written as F_σ, then each occurence of sub-formula F_i in F replaced by G_i.
If F contains sub-formulas F_i and F_j is larger than F_i, then we substitute F_j first.
If σ is a variable substitution and F is valid, then F_σ is valid.
If $F_i \Leftrightarrow G_i$ for all i, then $F \Leftrightarrow F_\sigma$.

Example : Prove the validity of

$$((P \leftrightarrow \neg R) \wedge ((P \leftrightarrow \neg R) \rightarrow (P \leftrightarrow \neg(R \wedge \neg Q)))) \rightarrow (P \leftrightarrow \neg(R \wedge \neg Q))$$

We use the following substitution

$$\sigma = \{P \leftrightarrow \neg R \mapsto P, P \leftrightarrow \neg(R \wedge \neg Q) \mapsto Q\}$$

Then, we have

$$F_\sigma = (P \wedge (P \rightarrow Q)) \rightarrow Q$$

which is a tautology.

Definition. *A formula F is in **negation formal-form**, if and only if negations $\neg$ appear only directly in front of variables, i.e. they appear only in literals, and the formulas contain only disjunctions and conjunctions.*
We can use the following equivalences :

$$
\begin{aligned}
\neg\neg F &\Leftrightarrow F \\
\neg\top &\Leftrightarrow \bot \\
\neg\bot &\Leftrightarrow \top \\
\neg(F \wedge G) &\Leftrightarrow (\neg F \vee \neg G) \\
\neg(F \vee G) &\Leftrightarrow (\neg F \wedge \neg F) \\
F \to G &\Leftrightarrow \neg F \vee G \\
F \leftrightarrow G &\Leftrightarrow (F \to G) \wedge (G \to F)
\end{aligned}
$$

Definition. *A formula F is in **disjunctive normal-form** if and only if it is in negation normal-form and it is the disjunction ($\vee$) of conjunctions ($\wedge$) :*

$$F = \bigvee_i \bigwedge_j L_{ij}$$

We can use the following additional equivalences :

$$
\begin{aligned}
(F \vee G) \wedge H &\Leftrightarrow (F \wedge H) \vee (G \wedge H) \\
F \wedge (G \vee H) &\Leftrightarrow (F \wedge G) \vee (F \wedge H)
\end{aligned}
$$

Definition. *A formula F is in **conjunctive normal-form** if and only if it is in negation normal-form and it is the conjunction ($\wedge$) of disjunctions ($\vee$) :*

$$F = \bigwedge_i (\bigvee_j L_{ij})$$

We can use the following additional equivalences :

$$
\begin{aligned}
(F \wedge G) \vee H &\Leftrightarrow (F \vee H) \wedge (G \vee H) \\
F \vee (G \wedge H) &\Leftrightarrow (F \vee G) \wedge (F \vee H)
\end{aligned}
$$

Example : Turn $(P \wedge (P \to Q)) \to Q$ into CNF :

$$
\begin{aligned}
(P \wedge (P \to Q)) \to Q &= \neg(P \wedge (P \to Q)) \vee Q \\
&= (\neg P \vee \neg(P \to Q)) \vee Q \\
&= (\neg P \vee \neg(\neg P \vee Q)) \vee Q \\
&= (\neg P \vee (\neg\neg P \vee \neg Q)) \vee Q \\
&= (\neg P \vee (P \vee \neg Q)) \vee Q \\
&= \neg Q \vee Q \\
&= \top
\end{aligned}
$$

2.4 Deciding satisfiability

If formula F contains n propositional variables, then we can try out all value assignments to the variables to see whether or not F is satisfiable. The truth table of such a construction will have size $O(2^n)$. However, we can check this more space efficiently :

function SAT(F)
  if $F = \top$ **then**
    return *true*
  else if $F = \bot$ **then**
    return *false*
  else
    $P = choose\_vars(F)$
    return $SAT(F\{P \mapsto \top\}) \vee SAT(F\{P \mapsto \bot\})$
  end if
end function

However, the SAT method requires that the formula is in conjunctive normal-form, but turning a formula of the form

$$\bigvee_{i=1}^{n} (P_i \wedge Q_i)$$

is exponential.
Instead of turning the formula F into an equivalent formula in CNF, we turn it into an equisatisfiable formula F' in CNF. An equisatisfiable formula F' to F is satisfiable if and only if F is satisfiable (this is not equivalence).

Definition. *Propositional variables $rep(F)$ that represent formula F :*

$$\begin{aligned}
rep(\top) &= \top \\
rep(\bot) &= \bot \\
rep(P) &= P \text{ for variables } P \\
rep(F) &= P_F \text{ for all other formulas } F
\end{aligned}$$

Definition. *Formulas $enc(F)$ that make sure that $rep(F)$ represents formulas F :*

$$\begin{aligned}
enc(\top) &= \top \\
enc(\bot) &= \top \\
enc(P) &= \top \\
enc(\neg F) &= (\neg P_{\neg F} \vee \neg rep(F)) \wedge (P_{\neg F} \vee rep(F)) \\
enc(F \wedge G) &= (\neg P_{F \wedge G} \vee rep(F)) \wedge (\neg P_{F \wedge G} \vee rep(G)) \wedge (P_{F \wedge G} \vee \neg rep(F) \vee \neg rep(G)) \\
enc(F \vee G) &= (P_{F \vee G} \vee \neg rep(F)) \wedge (P_{F \vee G} \vee \neg rep(G)) \wedge (\neg P_{F \vee G} \vee rep(F) \vee rep(G)) \\
enc(F \to G) &= (P_{F \to G} \vee rep(F)) \wedge (P_{F \to G} \vee \neg rep(G)) \wedge (\neg P_{F \to G} \vee \neg rep(F) \vee rep(G)) \\
enc(F \leftrightarrow G) &= (\neg P_{F \leftrightarrow G} \vee \neg rep(F) \vee rep(G)) \\
&\quad \wedge (\neg P_{F \leftrightarrow G} \vee rep(F) \vee \neg rep(G)) \\
&\quad \wedge (P_{F \leftrightarrow G} \vee rep(F) \vee rep(G)) \\
&\quad \wedge (P_{F \leftrightarrow G} \vee \neg rep(F) \vee \neg rep(G))
\end{aligned}$$

Finally, we replace formula F with the equisatisfiable formula

$$rep(F) \wedge \bigwedge_{G \in sub\_formulas(F)} enc(G)$$

where $sub\_formulas(F)$ is the set of all sub-formulas of F including F itself.

Definition. *Let F be a formula in CNF, i.e. it is the conjunction of clauses (disjunction of literals). Let C_1 and C_2 be two clauses in F both containing, but disagreeing on variable P. Assume (without loss of generality) that C_1 contains P in its positive and C_2 contains P in its negative form. We write $C_1[P]$ and $C_2[\neg P]$.*
Then we get the following rule, denote ***resolution step*** *:*

$$\frac{C_1 \qquad C_2}{C_1\{P \mapsto \bot\} \vee C_2\{\neg P \mapsto \bot\}}$$

$C_1\{P \mapsto \bot\} \vee C_2\{\neg P \mapsto \bot\}$ is called the resolvent of C_1 and C_2.

To check the satisfiability of a formula F

- As long as we can, we apply the resolution step.
- In case we ever derive $\bot$ in a resolution step, F is not satisfiable.
- Otherwise, if no more resolution steps can be executed, F is satisfiable.

Example : Show that $\phi = (P \wedge (P \to Q)) \to Q$ is satisfiable by using the resolution steps. First, we turn ϕ into an equisatisfiable formula by using its represent :

$$\begin{aligned} rep((P \wedge (P \to Q)) \to Q) &= P_1 \\ rep(P \wedge (P \to Q)) &= P_2 \\ rep(P \to Q) &= P_3 \\ rep(Q) &= Q \\ rep(P) &= P \end{aligned}$$

and its encoding :

$$\begin{aligned} enc(\phi) &= rep(\phi) \wedge enc(P_1) \wedge enc(P_2) \wedge enc(P_3) \\ rep(\phi) &= P_1 \\ enc(P_1) &= (P_1 \vee P_2) \wedge (P_1 \vee \neg Q) \wedge (\neg P_1 \vee \neg P_2 \vee Q) \\ enc(P_2) &= (\neg P_2 \vee P) \wedge (\neg P_2 \vee P_3) \wedge (P_2 \vee \neg P \vee \neg P_3) \\ enc(P_3) &= (P3 \vee P) \wedge (P_3 \vee \neg Q) \wedge (\neg P_3 \vee \neg P \vee Q) \\ enc(\phi) &= P_1 \\ &\quad \wedge (P_1 \vee P_2) \wedge (P_1 \vee \neg Q) \wedge (\neg P_1 \vee \neg P_2 \vee Q) \\ &\quad \wedge (\neg P_2 \vee P) \wedge (\neg P_2 \vee P_3) \wedge (P_2 \vee \neg P \vee \neg P_3) \\ &\quad \wedge (P3 \vee P) \wedge (P_3 \vee \neg Q) \wedge (\neg P_3 \vee \neg P \vee Q) \end{aligned}$$

Then we apply the resolution steps on :

$$P_1 \wedge (P_1 \vee P_2) \wedge (P_1 \vee \neg Q) \wedge (\neg P_1 \vee \neg P_2 \vee Q) \wedge (\neg P_2 \vee P) \wedge (\neg P_2 \vee P_3) \wedge (P_2 \vee \neg P \vee \neg P_3) \wedge$$
$$(P_3 \vee P) \wedge (P_3 \vee \neg Q) \wedge (\neg P_3 \vee \neg P \vee Q)$$

Using P_1

$$\dfrac{\dfrac{P_1 \qquad P_1 \vee P_2 \qquad P_1 \vee \neg Q \qquad \neg P_1 \vee \neg P_2 \vee Q}{(\neg P_2 \vee Q) \wedge (P_2 \vee \neg P_2 \vee Q) \wedge (\neg P_2 \vee Q \vee \neg Q)}}{(\neg P_2 \vee Q)}$$

Using P_2

$$\dfrac{\dfrac{\neg P_2 \vee Q \qquad \neg P_2 \vee P \qquad \neg P_2 \vee P_3 \qquad P_2 \vee \neg P \vee \neg P_3}{(\neg P \vee \neg P_3 \vee Q) \wedge (\neg P \vee \neg P_3 \vee P) \wedge (\neg P \vee \neg P_3 \vee P_3)}}{(\neg P \vee \neg P_3 \vee Q)}$$

Using P_3

$$\dfrac{\dfrac{\neg P \vee \neg P_3 \vee Q \qquad P_3 \vee P \qquad P_3 \vee \neg Q}{(\neg P \vee P \vee Q) \wedge (\neg P \vee \neg Q \vee Q)}}{\top}$$

Definition. *The Boolean Constraint Propagation (BCP) on a formula F uses* ***unit resolution*** *(resolution against literals). Let P be a propositional variable, then we get the rules :*

$$\dfrac{P \qquad C[\neg P]}{C\{\neg P \mapsto \bot\}} \qquad \dfrac{\neg P \qquad C[P]}{C\{P \mapsto \bot\}}$$

Then we obtain the DPLL algorithm used in practice :

function DPLL(F)
  $F' = BCP(F)$
  if $F' = \top$ **then**
    return $true$
  else if $F' = \bot$ **then**
    return $false$
  else
    $P = choose\_vars(F')$
    return $SAT(F'\{P \mapsto \top\}) \vee SAT(F'\{P \mapsto \bot\})$
  end if
end function

Example : Show that $(P \wedge (P \to Q)) \to Q$ is satisfiable by using DPLL.

$$\mathbf{DPLL}(P_1 \wedge (P_1 \vee P_2) \wedge (P_1 \vee \neg Q) \wedge (\neg P_1 \vee \neg P_2 \vee Q) \wedge (\neg P_2 \vee P) \wedge (\neg P_2 \vee P_3) \wedge (P_2 \vee \neg P \vee$$

$$\neg P_3) \wedge (P_3 \vee P) \wedge (P_3 \vee \neg Q) \wedge (\neg P_3 \vee \neg P \vee Q)) = \mathbf{BCP}(\dots) =$$

$$(\neg P_2 \vee Q) \wedge (P_1 \vee P_2) \wedge (P_1 \vee \neg Q) \wedge (\neg P_2 \vee P) \wedge (\neg P_2 \vee P_3) \wedge (P_2 \vee \neg P \vee \neg P_3) \wedge (P_3 \vee P) \wedge$$

$$(P_3 \vee \neg Q) \wedge (\neg P_3 \vee \neg P \vee Q)$$

$$\mathbf{DPLL}(\dots [P_2 \mapsto \top]) = \mathbf{BCP}(Q \wedge P \wedge P_3 \wedge (P_1 \vee \neg Q) \wedge (P_3 \vee P) \wedge (P_3 \vee \neg Q) \wedge (\neg P_3 \vee \neg P \vee Q)) =$$

$$(\text{w.r.t. } Q) P \wedge P_3 \wedge P_1 \wedge (P_3 \vee P) \wedge (\neg P_3 \vee \neg P) =$$

$$(\text{w.r.t. } P_3) P \wedge P_1 \wedge (P_3 \wedge P) \wedge (\neg P \vee Q) =$$

$$(\text{w.r.t. } P) P_1 \wedge (P_3 \vee P) \wedge Q =$$

$$\mathbf{DPLL}(\dots [P_1 \mapsto \top]) =$$

$$\mathbf{DPLL}(\dots [Q \mapsto \top]) =$$

$$\mathbf{DPLL}(\dots [P_3 \mapsto \top]) =$$

$$\mathbf{DPLL}(\dots [P \mapsto \top])$$

$(P \wedge (P \to Q)) \to Q$ is satisfiable with the interpretation

$$I = \{P \mapsto \top, Q \mapsto \top\}$$

Chapter 3

Computability

Decidability and undecidability are synonymous to computability and non-computability in the case of decision problems.
In general, it is impossible to decide wheter or not a program will write "Hello, World". That does not mean that we cannot decide the property for all programs but it means that we can't decide the property for any program.

Theorem 1. *Testing whether a program will always print "Hello, World" (or any other property) is undecidable, i.e. no computing device can perform the test.*

Proof. The proof is by contradiction. Assume there exists a program **tester** that gets as input a program P and an input I for the program P. Then, **tester**, when apply to P and I (denote $tester(P, I)$), will output "yes" or "no" depending on whether P on I outputs "Hello, World".
Then we define the function **tester2** and **tester3** as follow :

function TESTER2(P,I)
  if $tester(P, I) == '' yes``$ **then**
    $print(''yes``)$
  else
    $print(''Hello, world``)$
  end if
end function
function TESTER3(P)
  $print(tester2(P, P))$
end function

Finally we apply $tester3$ to itself :

$$tester3(tester3) = tester2(tester3, tester3) = tester(tester3, tester3)$$

If $tester3$ on input $tester3$ output "Hello, World", then $tester(tester3, tester3)$ will output "yes" and then $tester3(tester3)$ must output "yes".
If $tester3$ on input $tester3$ does not output "Hello, World", then $tester$ $tester(tester3, tester3)$ will output "no" and this $tester3$ must output "Hello, World". □

3.1 Reduction

Definition. *For two problems P_1 and P_2 where P_2 is undecidable, if we can show that if P_1 is decidable then P_2 is decidable too, then P_2 is undecidable.*
We say we ***reduced*** *P_2 to P_1.*

Example : The problem of wheter or not a program calls a particular function named foo is undecidable.

Proof. For any program, if it contains a function foo, rename that function.
Add an empty function foo and an array storing the first 12 letters of program output.
Modify the program in such a way that whenever anything is output, check if the array contains "Hello, World" and, if yes, call foo.
So foo is called if and only if the original program had output "Hello, World". So if calling foo were decidable, we would decide outputting "Hello, World", which we know is undecidable. □

3.2 Church-Turing Thesis

"Any general way to compute will allow us to compute what can be computed by a Turing machine".
A Turing machine uses

- a finite control memory (**states**) of a read-/write-head.
- a two-sidedly infinite **tape** divided into **cells** that can store finitely many different tape symbols.

Initially, a string of **input symbols** is stored on the tape (all cells not containing input contain the so-called **blank**), and the read-/write-head is positioned at the tape cell that contains the first symbol of the input string.
The Turing machine reads the content of the cell where its read-/write-head is positioned. Depending on its state and the symbol in the cell, it overwrites the cell content, changes the control state and moves the read-/write-head one cell to the left or to the right.
If the Turing machine reaches an accepting state, it stops accepting.
If the Turing machine cannot move being in a non-accepting state, it stops non-accepting.

Definition. *A **deterministic Turing machine** $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ consists of :*

- *a finite set Q of states,*
- *a finite set Σ of input symbols,*
- *a finite set $\Gamma \supseteq \Sigma$ of type symbols,*
- *a transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$,*
- *an initial state $q_0 \in Q$,*
- *a blank symbol $B \in \Gamma \setminus \Sigma$,*
- *a finite set $F \subseteq Q$ of accepting states.*

A Turing machine halts when it cannot move anymore or when it reaches an accepting state. Whenever the Turing machine halts in its accepting state it accepts its input and what is on the tape is the output that it computed.

3.3 Non-computability

There exists functions that cannot be computed, it means that no computer or human being can compute the function.
A **Busy Beaver** is a Turing machine with a two symbol alphabet $\{B, 1\}$.

- It starts on an empty tape, write a sequence of consecutive 1's and then stops.
- Its output is the number of 1's it writes on the tape.
- Takes as input a number n.
- $BB(n)$ is the maximal output a busy beaver with $n+1$ states can produce.

Theorem 2. *$BB(n)$ is not computable.*

Proof. The proof is by contradiction on $BB(\dots)$.

- Let $f(n)$ be a computable function.
- Let *compute* be a Turing machine computing $f(n)$ in unary format with a constant number of states.
- Let *write_m* be a Turing machine writing m-many 1's with $m+1$ states.
- Let *double* be a Turing machine doubling the number of 1's with a constant number of states.

Let's combine those Turing machines, we get a new Turing machine, having $m+c$ states, that is a busy beaver (it writes $f(2 \times m)$-many 1's). Hence, $BB(m+c) \geq f(2 \times m)$.
Now we assume that $m > c$, then $BB(2 \times m) > BB(m+c) \geq f(2 \times m)$, hence $BB(\dots)$ grows more quickly than $f(\dots)$.
$BB(\dots)$ grows more quickly than $f(\dots)$, where $f(\dots)$ is an arbitrary computable function, if $BB(\dots)$ were computable, it would grow more quickly than itself, thus $BB(\dots)$ is not computable. □

We can turn each computational problem into a decision problem by giving the TM (Turing Machine) two strings v and w and ask whether the TM on input v will produce output w.
To decide a decision problem, the TM gets some input and says "yes" or "no" whenever it stop in an accepting state, respectively in an non-accepting state.

3.4 Binary Coding of TMs

We can number the elements from the sets Q, Γ and $\{L, R\}$. Let $\delta(q_i, X_j) = (q_k, X_l, D_m)$, where the indices indicate the numbers representing the different objects used in the definition of δ. Then, we can encode $\delta(q_i, X_j) = (q_k, X_l, D_m)$ by $0^i10^j10^k10^l10^m$.
Each such code $C_1, \dots, C_n$ for all the entries for δ in the TM's transition table, can be separated by two consecutive 1's, giving us $C_111C_211\dots11C_n$ as a binary coding of the entire TM.
We assume that we are dealing, for the moment, only with TMs with $\Gamma = \{0, 1, B\}$.

- Not each binary string represents the coding of a TM.
- We define that these strings represent the TM with one state and no move (accepting nothing [the empty set]).

- That allows us to talk about the ith TM (we have an ordering on TMs).

Let L_D be the set of all strings w of 0's and 1's such that, if w is the code of a TM M, M does not accept w.
TMs and their input (0 : not accepted, 1 : accepted):

$$\begin{array}{c} \\ M_1 \\ M_2 \\ M_3 \\ M_4 \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccc} w_1 & w_2 & w_3 & w_4 & \dots \end{array} \\ \begin{pmatrix} \mathbf{1} & 0 & 0 & 1 & \dots \\ 0 & \mathbf{1} & 1 & 0 & \dots \\ 0 & 1 & \mathbf{0} & 0 & \dots \\ 1 & 1 & 1 & \mathbf{0} & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \end{array}$$

The big diagonal is the language of all TMs that accept their coding as input. So its complement is L_D, then we use L_D for the diagonal :

$$\begin{array}{c} \\ M_1 \\ M_2 \\ M_3 \\ M_4 \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccc} w_1 & w_2 & w_3 & w_4 & \dots \end{array} \\ \begin{pmatrix} \mathbf{0} & 0 & 0 & 1 & \dots \\ 0 & \mathbf{0} & 1 & 0 & \dots \\ 0 & 1 & \mathbf{1} & 0 & \dots \\ 1 & 1 & 1 & \mathbf{1} & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \end{array}$$

L_D is not in the list of TM accepted language (it differs with each line in at least one bit).

Theorem 3. *There does not exist a Turing machine accepting L_D.*

Proof. The proof is by contradiction to the assumption that M exists, where M is a Turing machine that accept L_D.
By definition, M would occur in our list of Turing machine at position i, it is M_i. Then, is $w_i \in L_D$?

- If $w_i \in L_D$, then M_i accepts w_i, because it accepts L_D. However, $w_i \in L_D$ indicates that M_i does not accept w_i.
- If $w_i \notin L_D$, then M_i does no accepts w_i, because it accepts L_D. However, $w_i \notin L_D$ indicates that M_i accept w_i.

We get a contradiction in both cases to the assumption that M exists. □

Definition. *A language L is called **recursively enumerable** (RE) if and only if there exists a TM M such that:*

- *whenever $w \in L$, then M will stop in an accepting state when started on input w.*
- *whenever $w \notin L$, then M will stop in a non-accepting state or run forever when started on input w.*

Definition. *A language L is called **recursive** if and only if there exists a TM M such that:*

- *whenever $w \in L$, then M will stop in an accepting state when started on input w .*
- *whenever $w \notin L$, then M will stop in a non-accepting state when started on input w .*

A problem is decidable when the language that encodes the problem is recursive.

3.5 Reductions

Definition. *A **reduction** of one problem $P_1 \subseteq \Sigma_1^*$ to another problem $P_2 \subseteq \Sigma_2^*$ is a computable transformation $r : \Sigma_1^* \to \Sigma_2^*$ such that*

$$\forall w \in \Sigma_1^*; w \in P_1 \iff r(w) \in P_2$$

We then say that P_1 reduces to P_2, written $P_1 \propto P_2$

Theorem 4. *Let $P_1 \propto P_2$. If P_2 is decidable, then P_1 is decidable.*

Proof. To decide whether or not $w \in P_1$, compute $r(w)$ where r is the reduction of P_1 to P_2. Using the fact that P_2 is decidable, we decide whether or not $r(w) \in P_2$, deciding immediately whether or not $w \in P_1$. □

Corollary 1. *Let $P_1 \propto P_2$. If P_1 is undecidable, then P_2 is undecidable.*

Proof. The corollary is just the counterpositive version of the theorem. □

3.6 The Universal Language L_u

Definition. *L_u is defined as follows :*

$$\{(M, w) | \, TM \; M \; accepts \; input \; w\}$$

The structure (M, w) represents, in fact, the word over $\{0, 1\}$ that stats with the encoding of M, followed by 111, followed by w.

Lemma 1. *L_u is recursively enumerable.*

Proof. The TM M_u that accepts L_u operates in a quite simple way. On input (M, w), M_u simulates M on input w.

- As soon as M reaches its accepting when operating on w, then M_u stops accepting.
- As soon as M stops without accepting w, then M_u stops without accepting.
- Otherwise M_u continues simulating M on w.

□

The complement of L_u in not RE.

Definition. *A **property** of the RE languages is just a set of RE languages (those that satisfy the property).*
An RE language possesses the property if and only if it is contained in the set representing the property.
*A property is **trivial** if it is either the empty set or the set of all RE languages.*
All other properties are nontrivial.
Sets of languages cannot be recognised as the languages themselves (they are normally infinite).
RE languages are recognised by TMs.
If P is a property of the RE languages, we define L_P to be the language containing (the codes) of the TMs recognising the languages in P.
The decidability of P is then defined as the decidability of L_P.

Theorem 5 (Rice's Theorem)**.** *Every nontrivial property of the RE languages is undecidable.*

Proof. The proof is by reducing L_u to L_P.
Let P be a nontrivial property of the RE languages. Assume for the moment that $\emptyset \notin P$. Then, there must be some nonempty language $L \in P$.
Let M_L be the TM recognising L. We transform (M, w) into the following TM M' :

- M' get input x.
- M' simulates M on w.
- If M accepts w, then M' simulates M_L on x.
- Otherwise M' stops without accepting.

If $\emptyset \in P$, then $\overline{P}$ is a nontrivial property not containing $\emptyset$.
Therefore $\overline{P}$ is not decidable, and consequently P is neither. □

Practically nothing is decidable for formal (and practical) systems that are equivalent to TMs. So proving that a formal system is Turing complete (i.e. equivalent to a TM) immediately proves that nothing can be decided about that systems.

Definition. *A* ***non-deterministic Turing machine*** $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ *consists of :*

- *a finite set* Q *of states,*
- *a finite set* Σ *of input symbols,*
- *a finite set* $\Gamma \supseteq \Sigma$ *of type symbols,*
- *a transition relation* $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$,
- *an initial state* $q_0 \in Q$,
- *a blank symbol* $B \in \Gamma \setminus \Sigma$,
- *a finite set* $F \subseteq Q$ *of accepting states.*

3.7 Undecidability of First-order Logic

Chapter 4

Complexity

A non-computable (undecidable) problem is a problem to which no algorithm solving the problem exists.
An **intractable** problem is one to which an algorithm solving the problem does exist, but it is so inefficient that it is considered as practically useless.

Definition. *The **running time** of an algorithm is a function $T(n)$ taking as its argument the length n of the input given to the algorithm. Whenever a TM makes at most $T(n)$ moves on an input of length n, we say that $T(n)$ is the **running time** (or **time complexity**) of the TM.*
*If $T(n)$ is a polynomial in n, we say that the algorithm has a **polynomial run-time**. Polynomials algorithms are perceived to be tractable.*
*If $T(n)$ is an exponential function in n, we say that the algorithm has an **exponential run-time**. Exponentials algorithms are perceived to be intractable.*

Definition. *If for a problem P there exists a deterministic TM M such that M solves P in polynomial time, then we say that P is in the complexity class $\mathcal{P}$.*

Definition. *If for a problem P there exists a non-deterministic TM M such that M solves P in polynomial time, then we say that P is in the complexity class $\mathcal{NP}$.*

Because a deterministic TM is a special case of a non-deterministic TM, we get that $\mathcal{P} \subseteq \mathcal{NP}$. However, it is strongly believed that $\mathcal{P}$ and $\mathcal{NP}$ are different, and therefore that $\mathcal{NP} \not\subseteq \mathcal{P}$.

4.1 $\mathcal{P}$ versus $\mathcal{NP}$

We know $\mathcal{P} \subseteq \mathcal{NP}$. If we find a problem that is in $\mathcal{NP}$, but not in $\mathcal{P}$, then we found $\mathcal{P} \neq \mathcal{NP}$. However, if we take an arbitrary problem from $\mathcal{NP}$ and find that it is in $\mathcal{P}$, then this does not tell us anything, we could just have picked a problem that is in $\mathcal{P}$.
If $\mathcal{P} \neq \mathcal{NP}$, clearly $\mathcal{NP} \setminus \mathcal{P} \neq \emptyset$. Can we somehow characterize the problems that, if $\mathcal{P} \neq \mathcal{NP}$, would be in $\mathcal{NP} \setminus \mathcal{P}$?
If, for such a problem, we found that it were in $\mathcal{P}$, then we would know $\mathcal{P} = \mathcal{NP}$.

Definition. *We say that a problem P_1 is more difficult that a problem P_2, when P_1 cannot be solved in polynomial time whereas P_2 can.*

To prove that a problem is more difficult than another, we are doing a **polynomial-time reduction**, denote by $P_1 \propto_{poly} P_2$.
Note that the transformation will produce an output of a length that is polynomial in the length of the input it started with.
If $P_1 \propto_{poly} P_2$ and $P_2 \in \mathcal{P}$, then $P_1 \in \mathcal{P}$.

Definition. *Let $P_1 \propto_{poly} P_2$. Then if*

$$P_2 \in \mathcal{P} \Longrightarrow P_1 \in \mathcal{P}$$

or, conversely

$$P_1 \notin \mathcal{P} \Longrightarrow P_2 \notin \mathcal{P}$$

Then P_2 is at least as difficult as P_1.

Definition. *A problem P is said to be $\mathcal{NP}$-**complete** if and only if*

- $P \in \mathcal{NP}$,
- $\forall P' \in \mathcal{NP} : P' \propto_{poly} P$.

Note that $\mathcal{NP}$-completeproblems are believed to be intractable.

To show that a problem in $\mathcal{NP}$-complete, we need a first $\mathcal{NP}$-completeproblem and then we use reduction proofs.

4.2 Cook's Theorem

Definition. *Instantaneous Descriptions ID*
*Let $x_1x_2 \dots x_i \dots x_n$ be the content of the tape where x_1 is the leftmost **non-blank** symbol and x_n is the rightmost one.*
If the TM points in the tape cell that contains the ith symbol x_i, then the TM's ID is

$$x_1x_2 \dots x_{i-1}qx_ix_{i+1} \dots x_n$$

If the TM points in the tape cell that contains the jth blank symbol left of x_1, then the TM's ID is $qB^jx_1x_2 \dots x_n$
If the TM points in the tape cell that contains the jth blank symbol right of x_n, then the TM's ID is $x_1x_2 \dots x_nB^{j-1}qB$.

To represent a propositional formula to a TM :

- n truth variables and the symbol's set $\{(,), \wedge, \vee, \neg\}$.
- For a symbol, we just write into into the tape cells.
- For the variables $x_1, x_2, \dots, x_n$,we write x in a tape cell, followed by tape cells containing 0's and 1's that represent x's index in binary format.
- The tape alphabet is $\Gamma = \{x, 0, 1, (,), \wedge, \vee, \neg, B\}$.

The length of such a formula is $2m - 1$ where m is the number of different variable. The representation of the formula has length $m \log_2 m + m - 1$, which is $O(m \log_2 m)$, which is polynomial.

Theorem 6. *SAT is $\mathcal{NP}$-complete.*

Proof. We will prove that SAT is in $\mathcal{NP}$ and in $\mathcal{NP}$-hard.

SAT is in $\mathcal{NP}$: A non-deterministic TM can check all the truth assignments to the propositional variables in parallel, and then evaluate the formula in polynomial time.
So SAT is in $\mathcal{NP}$.

SAT is in $\mathcal{NP}$-hard: We will show $\forall P \in \mathcal{NP} : P \propto_{poly} SAT$.
If a problem P is in $\mathcal{NP}$, then there is an non-deterministic TM M that decides P in time $p(n)$ for input of size n, where $p(n)$ is a polynomial.
Making at most $p(m)$ moves, M cannot move right of its $p(n)$th tape cell.

- M's computations are ID sequences $ID_1 \dots ID_{p(n)+1}$.
- Each ID contains $p(n) + 1$ symbols.

So $(p(n) + 1) \times (p(n) + 1)$-many symbols suffice to represent a computation of M.

- We represent each symbol by $X_{i,j}$ with $o \leq i, j \leq p(n)$ representing the jth symbol of the ith ID.
- Each of these symbols is either a tape symbol or a state of M.
- There is a constant number c of tape symbols and states of M.

To each $X_{i,j}$ we create c-many propositional variables $y_{i,j,z}$ where Z is a tape symbol or a state of M. $y_{i,j,z}$ being true is supposed to represent the jth symbol of the ith ID being Z.
The number of propositional variables is $c \times (p(n) + 1) \times (p(n) + 1)$, which is polynomial.
Using the propositional variables $y_{i,j,z}$, we build a propositional formula that will be true if and only if we can assign to the $y_{i,j,z}$ truth values that correspond to representing an accepting computation of M.
The size of the formula will be a polynomial $q(\dots)$ in the number of propositional variables $(O(q(p^2(n))))$.
So the formula, that can be constructed in polynomial time, will be satisfiable if and only if M accepts the given input (the input is in P). In other word, P is reduced in polynomial time to SAT.

Constructing the Formula : The formula that we construct is $U \wedge S \wedge N \wedge F$, such that

- U represents that each symbol in the ID sequence is unique,
- S represents that the start ID is correct,
- N represents that the next ID is represented correctly,
- F represents that the sequence of IDs finishes correctly (i.e. an accepting ID is reached).

Formula U : U is the logical conjuntion (AND) of formulas

$$\neg(y_{i,j,Z_1} \wedge y_{y_i,j,Z_2})$$

with $Z_1 \neq Z_2$. The size of U is $O(p^2(n))$.

Formula S : Let q_0 be the initial state of M and let $a_1 a_2 \dots a_n$ be its input. Then S is the formula

$$y_{0,0,q_0} \wedge y_{0,1,a_1} \wedge y_{0,2,a_2} \wedge \dots \wedge y_{0,n,a_n} \wedge y_{0,n+1,B} \wedge \dots \wedge y_{0,p(n),B}$$

The size of S is $O(p(n))$.

Formula F : Let $q_1, \ldots, q_k$ be the accepting states of M and, for $0 \leq j \leq p(n)$, let F_j be the formula

$$y_{p(n),j,q_1} \wedge y_{p(n),j,q_2} \wedge \cdots \wedge y_{p(n),j,q_k}$$

Then F is the formula

$$F_0 \wedge F_1 \wedge \cdots \wedge F_{p(n)}.$$

The size of F is $O(p(n))$.

Formula N : N is the formula

$$N_0 \wedge N_1 \wedge \cdots \wedge N_{p(n)-1}$$

The N_i are constructed in such a way that they ensure that the $(i+1)$st ID is one of the successor IDs of the ith ID.
N_i is of the form

$$(A_{i,0} \vee B_{i,0}) \wedge (A_{i,1} \vee B_{i,1}) \wedge \cdots \wedge (A_{i,p(n)} \vee B_{i,p(n)})$$

such that $A_{i,j}$ handles the case where the jth symbol in the ith ID is the state, and $B_{i,j}$ handles the case where the jth symbol in the ith ID is not the state.
The size of N is $O(p^2(n))$, because $A_{i,j}$ and $B_{i.j}$ are $O(1)$.

Formula $B_{i,j}$: Let $a_0 \ldots a_s$ be the tape symbols of M, then, for $0 < j < p(n)$, $B_{i,j}$ is

$$(y_{i,j,a_0} \vee \cdots \vee y_{i,j,a_s}) \wedge ((y_{i,j-1,a_0} \vee \cdots \vee y_{i,j-1,a_s}) \wedge (y_{i,j+1,a_0} \vee \cdots \vee y_{i,j+1,a_s}) \rightarrow$$

$$(y_{i,j,a_0} \wedge y_{i+1,j,a_0}) \vee \cdots \vee (y_{i,j,a_s} \wedge y_{i+1,j,a_s}))$$

which is equivalent to

$$(y_{i,j,a_0} \vee \cdots \vee y_{i,j,a_s}) \wedge (\neg(y_{i,j-1,a_0} \vee \cdots \vee y_{i,j-1,a_s}) \vee \neg(y_{i,j+1,a_0} \vee \cdots \vee y_{i,j+1,a_s}) \vee (y_{i,j,a_0} \wedge$$

$$y_{i+1,j,a_0}) \vee \cdots \vee (y_{i,j,a_s} \wedge y_{i+1,j,a_s}))$$

If $j = 0$, then $B_{i,j}$ reduces to

$$(y_{i,j,a_0} \vee \cdots \vee y_{i,j,a_s}) \wedge (\neg(y_{i,j+1,a_0} \vee \cdots \vee y_{i,j+1,a_s}) \vee (y_{i,j,a_0} \wedge y_{i+1,j,a_0}) \vee \cdots \vee (y_{i,j,a_s} \wedge y_{i+1,j,a_s}))$$

If $j = p(n)$, then $B_{i,j}$ reduces to

$$(y_{i,j,a_0} \vee \cdots \vee y_{i,j,a_s}) \wedge (\neg(y_{i,j-1,a_0} \vee \cdots \vee y_{i,j-1,a_s}) \vee (y_{i,j,a_0} \wedge y_{i+1,j,a_0}) \vee \cdots \vee (y_{i,j,a_s} \wedge y_{i+1,j,a_s}))$$

The size of $B_{i,j}$ is $O(1)$.

Formula $A_{i,j}$: We consider the case $0 < j < p(n)$. $A_{i,j}$ is the disjunction of sub-formulas we get in the following way :

- Let $(p, b, L) \in \delta(q, a)$ be a transition of M. Then $A_{i,j}$ contains, for all tape symbols c of M, the sub-formulas

$$y_{i,j-1,c} \wedge y_{i,j,q} \wedge y_{i,j+1,a} \wedge y_{i+1,j-1,p} \wedge y_{i+1,j,c} \wedge y_{i+1,j+1,b}$$

- Let $(p, b, R) \in \delta(q, a)$ be a transition of M. Then $A_{i,j}$ contains, for all tape symbols c of M, the sub-formulas

$$y_{i,j-1,c} \wedge y_{i,j,q} \wedge y_{i,j+1,a} \wedge y_{i+1,j-1,c} \wedge y_{i+1,j,b} \wedge y_{i+1,j+1,p}$$

- Let q be an accepting state of M. Then $A_{i,j}$ contains, for all tape symbols a, b of M, the sub-formulas

$$y_{i,j-1,a} \wedge y_{i,j,q} \wedge y_{i,j+1,b} \wedge y_{i+1,j-1,a} \wedge y_{i+1,j,q} \wedge y_{i+1,j+1,b}$$

For the case $j = 0$, $A_{i,0}$ is the disjunction of sub-formulas we get in the following way :

- Let $(p, b, R) \in \delta(q, a)$ be a transition of M. Then $A_{i,0}$ contains the sub-formulas

$$y_{i,0,q} \wedge y_{i,1,a} \wedge y_{i+1,0,b} \wedge y_{i+1,1,p}$$

- Let q be an accepting state of M. Then $A_{i,0}$ contains, for all tape symbols a, b of M, the sub-formulas

$$y_{i,0,q} \wedge y_{i,1,a} \wedge y_{i+1,0,q} \wedge y_{i+1,1,a}$$

The size of $A_{i,j}$ is $O(1)$.
For an arbitrary problem P that is in $\mathcal{NP}$, we take a polynomial-time non-deterministic Turing machine M that accepts P and an input w to M of length n, and construct a propositional formula ϕ of size $O(p^2(n))$ where $p(n)$ is the polynomial limiting M's run time on inputs of length n. Even if for each construction step, we would require a number of steps that is itself a polynomial in n, the construction will not take more than polynomial time.
ϕ will be satisfiable if and only if there is an accepting computation of M on w or, in other words, if and only if w is in P. Therefore our construction is a polynomial-time reduction from P to SAT. □

Chapter 5

Polynomial Time Reductions

CSAT is the problem of whether or not a given formula in CNF is satisfiable.

Lemma 2. *CSAT is $\mathcal{NP}$-complete.*

Proof. CSAT is in $\mathcal{NP}$, because SAT is in $\mathcal{NP}$. We can check the satisfiability of arbitrary propositional formulas in non-deterministic polynomial time, we clearly can do this for a subset of propositional formulas.
CSAT is in $\mathcal{NP}$-hard, we reduce SAT to CSAT in polynomial time. We just apply the construction of equisatisfiable formulas that we applied when dealing with resolution. □

3SAT is the problem of whether or not a given formula in CNF, such that each of the clauses consists of the disjunction of exactly 3 literals, is satisfiable.

Lemma 3. *3SAT is $\mathcal{NP}$-complete.*

Proof. 3SAT is in $\mathcal{NP}$, because SAT is in $\mathcal{NP}$.
3SAT is $\mathcal{NP}$-hard, we reduce CSAT to 3SAT in linear time by using the following construction :
The disjunction

$$(x_1 \vee x_2 \vee \dots x_n)$$

is satisfiable if and only if the conjunction

$$(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee \neg y_1 \vee y_2) \wedge (x_4 \vee \neg y_2 \vee y_3) \dots (x_{n-2} \vee \neg y_{n-4} \vee y_{n-3}) \wedge (x_{n-1} \vee x_n \vee \neg y_{n-3})$$

is satisfiable, where $y_1, y_2, \dots, y_{n-3}$ are new variables. □