Logic for Computer Science

A Practical Introduction for Computer Science Students

Lecture Notes

Adrian Crăciun

January 24, 2017

Contents

1 Problem Solving Based on Reasoning **1**
1.1 Problem solving . . . 1
1.2 Problem Solving by Reasoning . . . 4
1.3 Language as Support for Reasoning . . . 6
1.4 Logic . . . 8
1.5 Acknowledgement . . . 10

2 Propositional Logic **11**
2.1 The Syntax of Propositional Logic . . . 12
2.2 Semantics of Propositional Logic . . . 15
2.2.1 Truth Valuations, Interpretation . . . 15
2.2.2 Validity, Satisfiability, Inconsistency . . . 16
2.3 Propositional Equivalence . . . 18
2.3.1 Logical Equivalence . . . 18
2.3.2 A Catalog of Equivalent Formulae . . . 19
2.4 Normal Forms . . . 22
2.5 Applications of Propositional Logic: Digital Circuit Design . . . 28
2.5.1 Propositional Logic and Boolean Functions . . . 28
2.5.2 Complete Set of Boolean Operators . . . 29
2.5.3 Digital Circuit Design . . . 29
2.6 Logical Consequence . . . 31
2.7 The Resolution Method in Propositional Logic . . . 34
2.7.1 Clause Form of Propositional Formulae . . . 34
2.7.2 Propositional Resolution . . . 35
2.7.3 Improvements of Propositional Resolution . . . 38

3 Predicate Logic as a Working Language **41**
3.1 The Importance of Predicate Logic 41
3.2 Syntax First Order Predicate Logic 42
3.3 Semantics of First Order Predicate Logic 46
3.3.1 Substitution and Semantics 48
3.4 Syntax (Revisited) 49
3.4.1 Syntactic Sugar 49
3.5 Back to Semantics: Validity, Satisfiability, Unsatisfiability 50
3.6 Proving. Proof techniques. Definitions. Theories. 51
3.6.1 Reasoning in Predicate Logic 51
3.6.2 Proof Situations 51
3.6.3 Basic Approaches to Proving 52
3.6.4 Definitions in Predicate Logic 55
3.7 Theories 57
3.8 Equality 59
3.9 Induction 60

CHAPTER 1

Problem Solving Based on Reasoning

1.1 Problem solving

Definition 1 (Problem, solution, context)**.** A *problem* is a situation where a certain object or state are wanted, but missing. A *solution* to a problem means making the desired object or state available. Problems do not appear by themselves, but rather in a *context*, *universe of discourse*, *world*. *Problem solving* is the process by which the problem is solved, i.e. the world with problem is transformed in the world with solution. See Figure 1.

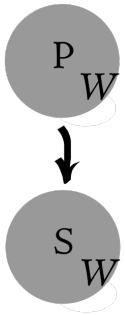


Figure 1: Problem solving: transforming a world (context, universe of discourse) *with problem* into a world (context, universe of discourse) *with solution*.

The definition above should agree with our everyday understanding of the

word "problem". Problems show up in different contexts, universes of discourse, "worlds". We are used to problems, we solve problems everyday, applying different methods and tools that are available to us. Let us consider some examples of problems and possible solutions.

Example 2 (River crossing (ancient)). Imagine that sometimes at the dawn of humanity a person stands by a river observing that on the other side a tree full of fruit. The problem is how to get there. There are many solutions to this problem, and the early humnans reached those solutions accumulating experience from failed attepmts, observing their environment, learning from e.g. animals how to dam the river, how to find fords in the river, how to bridge the river by cutting down a tree long enough to support them, and then refining these solutions by developing tools to build more permanent and reliable bridges, boats, etc.

These solutions are part of the succes of humanity as a race. However, note that this will be of little confort to that one individual at the dawn of humanity, trying to cross that river and reach the fruit, for whom these solutions might have taken a bit long to achieve. That is because this method of directly trying to solve the problem, essentially by trial-and-error, could be:

- *slow*: coming up with a successful solution can take a long time (generations),
- *expensive*: many resources may be needed in order to reach the solution,
- *exhausing*: reaching the solution could also require sustained effort,
- *potentially irreversible*: if the person gets badly injuried, or drowns attempting to reach the other side, then no other potential solution may be attempted.

Of course, this direct approach should not be discarded. It is a successful approach (it goes by the name of *evolution*). While it may not have served the person in our story trying to get on the other side, humanity has evolved to be able to solve these problems. But is there a more efficient way to solve problems?

Example 3 (Black box problem solving (modern)). In modern life there is a big pressure to solve problems quickly, and as directly as possible. To do so, we are used to call upon different tools, services, *black boxes*: there is an app for anything, a swipe on the screen of a mobile and as if by magic, we have the solution. Hardly anyone has patience if the solution is not available instantly. This technology-based method is:

- *fast*: complying with the demand of the modern world,
- *cheap*: more and more we expect the black boxes to be available, but note that this may be a position of privilege, and in general such black boxes are not widely available, one could even argue that where they are available their availability is at the expense of others (but this discussion is beyond the scope of these notes),

- *opaque*: solutions are achieved as if by magic, we don't care how, moreover we don't understand how, we have no control over the process. but

So the question stands: is there a more efficient way to solve problems while having control over the process?

1.2 Problem Solving by Reasoning

It turns out there is an efficient way for solving problems in an efficient manner, while maintaining control over the process, that is in principle available to everyone. This is problem solving based on *reasoning* and it works as follows, see Figure 2:

1. *Observation*: using the senses, refined with tools (microscopes, telescopes, rulers, etc.), a *model* of the world containing the problem is constructed. This is essentially an intellectual picture of the world containing the problem.

2. *Reasoning*: the model containing the problem is transformed into a model containing a solution, by transforming it through reasoning. The support of this step is the mind, refined by tools (pen and paper, computers, etc).

3. *Action*: once the solution is available in the model, it is implemented in the real world by using the hand (refined with tools and instruments: hammer, robots, bulldozers, etc).

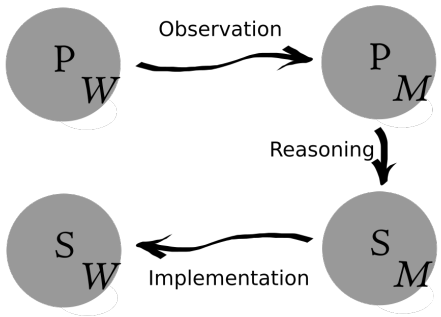


Figure 2: Problem solving by reasoning: by observation, construct *a model of the world where the problem is also expressed*, which is transformed by *reasoning* into a model with solution and then this solution is *implemented* obtaining a solution in the initial world.

This scheme for problem solving also provides an image of how scientific domains relate to the different stages of problem solving. The observation stage corresponds to *natural sciences*, the reasoning stage is the realm of *mathematics*, while the action stage corresponds to *engineering sciences*. So in this sense, mathematics is a method of problem solving. A very efficient method, as we will see.

Note that the scheme corresponds to more than just "thinking about" how to solve the problem (it is of course important to use common sense to filter out some of the bad direct solutions), or some sort of "magic" step in which

the model with the problem is transformed in the problem with solution. While these approaches would involve a similar scheme, there are some specific properties that define problem solving by reasoning. These will be discussed in the following.

Properties of Reasoning

Reasoning should be:

1. *Abstract:* Reasoning works in intellectual models. After the model is constructed, there is no longer a connection with the world described by the model. This provides:
   - *the power of reasoning*: it is reversible, fast, but also
   - *the weakness of reasoning*: the models constructed could be inadequate descriptions of the world and the problem, focusing only on some aspects but ignoring others, which can lead to unpredictable consequences (and this has, in fact, led to a series of crises faced by the world: ecological, technological, economical, social).
2. *Verifiable:*

   Reasoning should proceed in *small steps*, that everybody can perform, and they should be "deterministic" in nature. A reasoning step applied to a certain situation, will every time produce the same changes in the model, no matter who applies the step. These reasoning steps will be called *reasoning rules*, and the set of reasoning rules used to solve a problem will form a *reasoning system* (together with a description of what a solution is).
3. *Correct:*

   Reasoning should tranport truth. When in the model we consider a situation or object that have an image in the universe of discourse (in the world), then the changes carried out in the model by the application of reasoning steps should also have images in the universe of discourse. When we start from something "true", by the application of reasoning steps we should get something "true" as well.
4. *General:*

   Reasoning should be applicable to whole classes of problems, potentially infinitely many.

This method for problem solving illustrated in Figure 2 provides a framework for extremely effective problem solving. In fact this sits at the heart of the technological revolution that led to the development of Western societies. However, note that this is a tool for problem solving, one which is suitable for certain situations, but there are other tools for problem solving. Also, this is not a replacement for ethics, nor does it replace being.

1.3 Language as Support for Reasoning

Think back to our Example 2. What if the person standing by the river solves (by reasoning) the problem of crossing the river? Then they will want to tell the rest of the tribe what this solution is, so they can implement it together. But the solution has to be transmitted. This can be done through *language*. Therefore all the components of the problem solving have to be expressed in a language, so they can be communicated. By consequence *the support of reasoning is language*:

- intellectual models are built from language expressions,
- reasoning rules take expressions from the model and produce new expressions.

Definition 4 (Language)**.** A *language* (to be used to construct models and reason about problems) is definded by:

- the *symbols* of the language (i.e. its *vocabulary*),
- the *syntax* of the language, i.e. how are expressions in the language formed, and
- the *semantics* of the language, i.e. what is the meaning of the expressions of the language (in the universe that the language describes).
- the *reasoning system*, i.e. the rules for reasoning and a description of how to organize the reasoning process and how to

Whenever we want to solve a problem by reasoning we need a language that can describe the problem and its context (universe of discourse). We need to define the syntax, semantics, then specify the reasoning system (what are the rules for reasoning, do they respect the properties of reasoning, what do solutions look like?).

While there are many types of languages, some more appropriate for certain types of problems, others less so, the above structure is essentially the same.

Let us consider how languages can be classified according to different criteria.

Descriptive vs. Algorithmic Languages

Descriptive languages are used to describe objects, situations, knowledge. The language of "traditional" or "pure" mathematics is an example of a descriptive language. *Algorithmic languages* are used to describe actions, steps for solving problems. Programming languages are examples of algorithmic languages. Note, however that these languages should not be seen as separate languages. Using both the descriptive and algorithmic aspects in a language enhances the power and expressivity of the language. These aspects are in fact two sides of the same coin.

Universal vs. Special Languages

Universal languages are languages that are powerful enough to describe any universe of discourse, and to express any solutions. Such a language is the language of predicate logic, which is universal from a practical point of view – it can express all mathematics – and from a theoretical point of view – if a problem that can be expressed in the language has a solution, then this solution can be found using the reasoning system of predicate logic.

Special languages are languages that are relatively simple, therefore they can only describe special universes. However, for such languages the reasoning systems are simpler, and they are likely to have better properties: simpler solutions or for instance for each problem we can say whether it has a solution (and find it) or whether it has no solution. Such properties may not be available in more powerful (universal) languages. Examples of special languages are propositional logic, the language of switching circuits, constructive solid geometry.

Natural vs. Formal Languages

Natural languages are languages learned in context (i.e. by being among people who use a language, one starts to work out the syntax, semantics and the reasoning system - it is an evolutionary process). For *formal languages*, their vocabulary, syntax, semantics and reasoning systems are descrinbed precisely, and then they can be used.

For example, we can claim that in a certain sense, people learn mathematics in context, and only later does it become a formal language.

Metalanguage vs. Object Language

A *metalanguage* is a language that is used in order to define a language, and reason about the properties of the defined language, while the language defined is called *the object language*.

Note that these are not absolute terms. The same language (e.g. a natural, informal version of the language of sets) can play the role of metalanguage, when it is used to define a language (e.g. the language of predicate logic), while later it can be defined formally (thus playing the role of object language).

1.4 Logic

Logic (mathematical logic, symbolic logic, metamathematics) is a scientific domain concerned with *reasoning about reasoning*:

- defining languages: their vocabulary, syntax and semantics,
- their reasoning systems, and establishing the properties of these reasoning systems.

We will call a *logic* the language, its syntax and semantics, together with the reasoning system corresponding to the language. So within the scientific field of logic we can talk about many logics: propositional logic, predicate logic, fuzzy logic, etc.

Some of the properties of logics that are studied in logic include:

Correctness (soundness) of the reasoning rules: Are the reasoning rules correct? Two approaches are possible:

- *evolutionary*: where many people experience in many settings the correctness of certain inference steps, until everyone is convinced they are correct,
- *formal*: where we use a metalanguage and a meta reasoning system to establish the correctness of the inference rules.

Completeness of the reasoning rules: Can the inference system solve every problem that has a solution, is it powerful enough? For predicate logic it was Gödel who showed that this is the case, in [Gödel, 1930].

Consistency of the model: Is the model (i.e. the description of the universe of discourse) consistent, free from contradictions? Can the consistency be proved from within the logic? This question is also related to the next one.

Completeness of the model and reasoning system: NO, Gödel has shown in [Gödel, 1931] that in first order logic, if the model contains a description of arithmetic, then one cannot show that the inference rules are complete and the model is consistent. If the model is consistent, then there is some true statement that cannot be proved by the reasoning system.

Decidability: Can every problem be solved? For every problem can we say that it has a solution or not? This has been shown to be true for several situations: e.g. euclidian geometry, propositional logic, real closed fields, polynomial ideal teory, but Church and Turing have shown independently in 1936 that predicate logic is undecidable (i.e. there is no general mechanical method to establish the truth or falsity).

Categoricity of the model: Does the model describe just the indended universe of discourse, or are there other domains that are described by the model, but that behave essentially different?

Logic - An Essential Tool for Mathematics and Computer Science

Not only does logic provide a framework for problem solving that can be useful for any situation, but it is hardly conceivable that mathematics or computer science can "work" without logic. In fact logic played an essential rôle in

Russel's Paradox

Logic proved essential in solving the "proof crisis" that became apparent at the end of XIX century.

Example 5 (Russel's paradox). - an axiom for "sets":

Intuition:

For all "properties" E, there exist the "set" of all objects x that possess the property E.

More exactly, for every property E:

"there exists the set M such that for all x ($x \in M$ iff E)",

where M is a variable.

Now, consider the property $x \notin x$:

"there exists the set M such that for all x ($x \in M$ iff $x \notin x$)"

We take an M (since it exists):

"for all x ($x \in M$ iff $x \notin x$)"

Since the statement holds for all x, it holds also for $x := M$:

"($M \in M$ iff $M \notin M$)",

which is a contradiction.

In fact the axiom can be "patched" to avoid the paradox: For every property E with free variable x and every variable B not occurring free in E:

for all B there exists M such that for all x ($x \in M$ iff ($x \in B$ and E)).

But does the restricted form introduce anymore contradictions?

The Rôle of Logic in Computer Science

Why should a computer scientist know logic?

The traditional scope of CS was the automation of operation in numerical models. In fact, reasoning can be seen as a kind of "computation", thus in principle it can be automated.

"Logic plays a fundamental role in computer science, similar to that played by calculus in physics and traditional engineering. A knowledge of logic is becoming a practical necessity for the computer professional. " - Manna and Waldinger, see [Manna and Waldinger, 1985].

Logic-based Techniques in Computer Science

For raising the level of the abstract machine: boolean operations (logical gates), symbolic addresses (assembler), recursion (stacks), dynamic data structures (garbage collection), functional programming (programs as data), abstract data programming, OO, relational databases, expert systems, logic programming, CLP.

For bridging the gap between problem and program (algorithmic solution): program synthesis, program transformation, program verification.

1.5 Acknowledgement

The author has learned about logic and how it works from his advisor, Bruno Buchberger. In particular this introduction follows the introduction to logic from Buchberger's unpublished lecture notes, see [Buchberger, 1991].

CHAPTER 2

Propositional Logic

Propositional logic is a language of simple statements, called *atomic propositions*, and *compound statements*, formed by combining simpler ones, with the help of *propositional connectives* (*not*, *and*, *or*, *implication*, *equivalence*).

The atomic sentences that can be either **true** or **false**, but not both:

- "Today is Monday";
- "Snow is white";
- "Sugar is a carbohydrate";

Whether a compound proposition is **true** or **false** can be computed starting from the values of atomic propositions from which it is composed:

- "Today is Monday or today is Tuesday";
- "Today is Monday or today is not Monday";
- "If children are mean then birds fly westwards";
- "Children are mean and children are not mean".

The problems we want to solve in propositional logic are related to the meaning of propositions: the assignment of truth values to the atomic propositions that form a compound expression represents its *interpretation*. Statements are **true** for any interpretation, are called *tautologies* or *valid*. Statements that are **true** for some interpretations are *satisfiable*. Statements that are **false** for any interpretation are called *unsatisfiable*. Statements that are both **true** or both **false** under the same interpretations are *logically equivalent*. Statements that

are **true** whenever a given set of statements are all **true** are *logical consequences* of the respective set.

This Part deals with propositional logic. We will define the *syntax* of propositional logic,define the *semantics* of propositional logic, and the problems that can be expressed in propositional logic: satisfiability, validity, logical consequence. We will show that the logic is decidable, and there is a direct (but prohibitive) solution for every problem that can be expressed in the logic. Therefore we need a reasoning system that hopefully is more efficient, and we present such a reasoning system, which is correct and complete. We also present some important applications for propositional logic (digital circuit design and transformation).

2.1 The Syntax of Propositional Logic

The Vocabulary of Propositional Logic

Definition 6 (Set of propositional variables)**.** $\mathcal{V}$ is a *set of propositional variables* iff $\mathcal{V}$ is a countable set of words consisting of English letters (uppercase) possibly followed by numerical indices.

Example 7. A, B, P_1, Q_5 are examples of propositional variables.

Definition 8 (Special symbols of propositional logic)**.** The *special symbols* of propositional logic are:

1. variables from the set of propositional variables,
2. symbols denoting the logical connectives (logical operators): $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$,
3. parentheses: (,).

Well-formed Formulae

Remark. Propositional variables can be used to denote *atoms*, i.e. simple propositions, not involving propositional connectives, but also to stand in or rename propositional formulae. The particular use for propositional variables will be specified each time.

Definition 9 (Well-formed formulae of propositional logic)**.** The *well-formed formulae* (formulae, WFF's) of propositional logic are defined recursively as follows:

1. If A is an *atom*, then A is a formula.
2. If P is a formula, then $(\neg P)$ is a formula.
3. If P and Q are formulae, then $(P \wedge Q)$, $(P \vee Q)$, $(P \rightarrow Q)$ and $(P \leftrightarrow Q)$ are formulae.

4. All formulae are generated following the above rules.

Example 10.

1. A is a formula;
2. $B \rightarrow)$ is not a formula;
3. $A \rightarrow B$
   - is not a formula, according to the definition,
   - but why not allow it (for practical reasons)?

Example 11 (Why use parentheses?).

- $P \rightarrow Q \wedge R$ is ambiguous, it could stand for:
  - $((P \rightarrow Q) \wedge R)$,
  - $(P \rightarrow (Q \wedge R))$.

Relaxing the Syntax

In practice, we allow a relaxed syntax, in that some of the parentheses are dropped. However, formulae should be nonambiguous, even in this relaxed syntax.The way to avoid ambiguity is to define *priorities (precedence) for propositional connectives*:

Example 12 (Priorities for propositional connectives (decreasing)).

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

REMARK. The above is just an example of precedence. Different authors may use different precedences. When using a relaxed syntax, always specify the precedence. *When in doubt, use parentheses!!!*

Example 13 (Precedences revisited). Using the precedence above,

$$P \rightarrow Q \wedge R$$

is, in fact

$$(P \rightarrow (Q \wedge R)).$$

REMARK. From now on, if not otherwise specified, we allow the use of the relaxed syntax for propositional logic, with the precedence given above.

Associativity of Propositional Connectives

Consider $P \rightarrow Q \rightarrow R$. This is ambiguous:

- is it $P \rightarrow (Q \rightarrow R)$,
- or is it $(P \rightarrow Q) \rightarrow R$?

It depends on the *associativity* of the connective. In general, propositional connectives tend to be right associative, so the first variant above will be considered. *When in doubt, use parentheses!!!*

Notation for Propositional Connectives:

Various notations are used in literature:

- $\leftrightarrow$, $\rightarrow$, $\vee$, $\wedge$, $\neg$;
- iff, $if...then...$, or, and, not;
- $\equiv$, $\supset$, $\vee$, $\wedge$, $\neg$.

2.2 Semantics of Propositional Logic

2.2.1 Truth Valuations, Interpretation

Definition 14 (Domain of truth values)**.** We choose 2 distinct values $\mathbb{T}$, $\mathbb{F}$. *The set of truth values* is the set $\{\mathbb{T}, \mathbb{F}\}$. The set of truth values is (totally) ordered, $\mathbb{F} < \mathbb{T}$.

Definition 15 (Truth functions)**.** The following functions are called *truth functions*:

$$\mathcal{B}_\neg : \{\mathbb{T}, \mathbb{F}\} \to \{\mathbb{T}, \mathbb{F}\},$$
$$\mathcal{B}_\wedge, \mathcal{B}_\vee, \mathcal{B}_\to, \mathcal{B}_\leftrightarrow : \{\mathbb{T}, \mathbb{F}\} \times \{\mathbb{T}, \mathbb{F}\} \to \{\mathbb{T}, \mathbb{F}\} ,$$

v	w	$\mathcal{B}_\neg(v)$	$\mathcal{B}_\wedge(v,w)$	$\mathcal{B}_\vee(v,w)$	$\mathcal{B}_\to(v,w)$	$\mathcal{B}_\leftrightarrow(v,w)$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{F}$		$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{F}$		$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$

Remark. The truth functions reflect the intuitive meaning of the propositional connectives.

Definition 16 (Truth valuation/Interpretation)**.** $\mathcal{T}$ is a *truth valuation (interpretation)* iff $\mathcal{T}$ is a mapping from $\mathcal{V}$ to the set $\{\mathbb{T}, \mathbb{F}\}$ of truth values.

Example 17 (Truth valuation). Let $G \doteq (P \wedge Q) \to (R \leftrightarrow (\neg S))$.
A truth valuation (interpretation) for the formula G is
$\mathcal{T}(P) = \mathbb{T}$, $\mathcal{T}(Q) = \mathbb{F}$, $\mathcal{T}(R) = \mathbb{T}$, $\mathcal{T}(S) = \mathbb{T}$.

Remark. The truth valuation assigns to each propositional variable that occurs in the formula either one of the values $\mathbb{T}, \mathbb{F}$.

Definition 18 (Truth values under truth valuations/interpretations of propositional formulae)**.** For all propositional formulae B, we denote $\upsilon_\mathcal{T}(B)$ - *"the truth value of B under the truth valuation/ interpretation $\mathcal{T}$"*. It is defined inductively as follows:

- if V is an atom, $\upsilon_\mathcal{T}(V) = \mathcal{T}(V)$;
- if B is a formula $\upsilon_\mathcal{T}(\neg B) = \mathcal{B}_\neg(\upsilon_\mathcal{T}(B))$;
- if B_1, B_2 are formulae,
  - $\upsilon_\mathcal{T}((B_1 \wedge B_2)) = \mathcal{B}_\wedge(\upsilon_\mathcal{T}(B_1), \upsilon_\mathcal{T}(B_2))$,
  - $\upsilon_\mathcal{T}((B_1 \vee B_2)) = \mathcal{B}_\vee(\upsilon_\mathcal{T}(B_1), \upsilon_\mathcal{T}(B_2))$,
  - $\upsilon_\mathcal{T}((B_1 \to B_2)) = \mathcal{B}_\to(\upsilon_\mathcal{T}(B_1), \upsilon_\mathcal{T}(B_2))$,
  - $\upsilon_\mathcal{T}((B_1 \leftrightarrow B_2)) = \mathcal{B}_\leftrightarrow(\upsilon_\mathcal{T}(B_1), \upsilon_\mathcal{T}(B_2))$.

Example 19 (Interpretation revisited). Now let's compute the truth value of $G \doteq ((P \wedge Q) \to (R \leftrightarrow (\neg S)))$, from the example above, under the interpretation $\mathcal{T}$ ($\mathcal{T}(P) = \mathbb{T}$, $\mathcal{T}(Q) = \mathbb{F}$, $\mathcal{T}(R) = \mathbb{T}$, $\mathcal{T}(S) = \mathbb{T}$):

$$
\begin{aligned}
&\upsilon_{\mathcal{T}}(G)\\
&= \upsilon_{\mathcal{T}}(((P \wedge Q) \rightarrow (R \leftrightarrow (\neg S))))\\
&= \mathcal{B}_{\rightarrow}(\upsilon_{\mathcal{T}}((P \wedge Q)), \upsilon_{\mathcal{T}}((R \leftrightarrow (\neg S))))\\
&= \mathcal{B}_{\rightarrow}(\mathcal{B}_{\wedge}(\upsilon_{\mathcal{T}}(P), \upsilon_{\mathcal{T}}(Q)), \mathcal{B}_{\leftrightarrow}(\upsilon_{\mathcal{T}}(R), \upsilon_{\mathcal{T}}((\neg S))))\\
&= \mathcal{B}_{\rightarrow}(\mathcal{B}_{\wedge}(\mathbb{T}, \mathbb{F}), \mathcal{B}_{\leftrightarrow}(\mathbb{T}, \mathcal{B}_{\neg}(\upsilon_{\mathcal{T}}(S))))\\
&= \mathcal{B}_{\rightarrow}(\mathbb{F}, \mathcal{B}_{\leftrightarrow}(\mathbb{T}, \mathbb{F}))\\
&= \mathcal{B}_{\rightarrow}(\mathbb{F}, \mathbb{F})\\
&= \mathbb{T}.
\end{aligned}
$$

Definition 20 (True proposition)**.** A propositional formula B is said to be *true under an interpretation* if it is evaluated to $\mathbb{T}$ in the interpretation. Otherwise B is said to be *false*.

REMARK.(Number of possible interpretations)
If there are n distinct atoms in a formula, then there will be 2^n distinct interpretations for the formula.
NOTATION.(Representation of interpretation)
For a given formula, assume $\{A_1, \ldots, A_n\}$ is the set of propositional variables. Then a particular interpretation can be represented as the set $\{m_1, \ldots, m_n\}$ where m_i is either A_i or $\neg A_i$, according to whether $\mathcal{T}(A_i) = \mathbb{T}$ or $\mathcal{T}(A_i) = \mathbb{F}$, respectively.

Example 21. Consider $((P \rightarrow Q) \vee R)$.

$$\{P, \neg Q, R\} \text{ stands for } \mathcal{T}(P) = \mathbb{T},\ \mathcal{T}(Q) = \mathbb{F},\ \mathcal{T}(R) = \mathbb{T}.$$

2.2.2 Validity, Satisfiability, Inconsistency

Definition 22 (Propositional Tautologies)**.** The formula B is *valid* in propositional logic (or B is a propositional tautology) iff, for all truth valuations $\mathcal{T}$, $\upsilon_{\mathcal{T}}(B) = \mathbb{T}$ (i.e. the proposition is true under all possible interpretations). The formula is *invalid* iff it is not valid.

Definition 23 (Satisfiability)**.** A propositional formula B is said to be *satisfiable* in propositional logic iff for some truth valuation $\mathcal{T}$, $\upsilon_{\mathcal{T}}(B) = \mathbb{T}$ (i.e. there exists some interpretation for which the formula is evaluated to true).

A propositional formula B is said to be *unsatisfiable (inconsistent)* iff it is false under all truth valuations (all interpretations).

Exercises
Using the definitions, show that:

1. A formula is a tautology (valid) iff its negation is unsatisfiable (inconsistent).
2. A formula is unsatisfiable (inconsistent) iff its negation is valid.
3. A formula is invalid iff there is at least one interpretation under which the formula is false.
4. If a formula is valid, then it is satisfiable, but not vice-versa.
5. If a formula is unsatisfiable (inconsistent), then it is invalid, but not vice-versa.

Definition 24. If a formula B is true (evaluated to $\mathbb{T}$) under some truth valuation (interpretation) $\mathcal{T}$, we say that $\mathcal{T}$ *satisfies* B (B *is satisfied by* $\mathcal{T}$).

On the other hand, if a formula B is false (evaluated to $\mathbb{F}$) under $\mathcal{T}$, we say that $\mathcal{T}$ *falsifies* B (B *is falsified by* $\mathcal{T}$).

When an interpretation (truth valuation) $\mathcal{T}$ satisfies a formula B, $\mathcal{T}$ is called a *model* of B.

Truth Tables

Definition 25 (Truth table (informal))**.** A *truth table* corresponding to a propositional formula is the table constructed in the following manner:

- on the first row, list all propositional variables occurring in the formula, then,
- list all subformulae of the formula tree, in a bottom-up manner, the last one being the formula itself,
- the columns under the propositional variables contain the truth values under all possible truth valuations,
- the columns under each subformula contain on each position the value under the truth valuation taken from the corresponding row,
- note that using the bottom-up approach each computation amounts to one application of a truth function.

Example 26 (Truth table). Consider the propositional formula $(P \to Q) \vee R$.

P	Q	R	$P \to Q$	$(P \to Q) \vee R$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$

2.3 Propositional Equivalence

2.3.1 Logical Equivalence

Definition 27 (Equivalence of Propositional Formulae)**.** Two propositional formulae F and G are *(propositionally) equivalent* (and we write $F \sim G$) iff for all truth valuations (interpretations) $\mathcal{T}$, $v_{\mathcal{T}}(F) = v_{\mathcal{T}}(G)$.

Example 28. Consider the formulae $(P \to Q)$ and $(\neg P \vee Q)$.

P	Q	$(P \to Q)$	$\neg P$	$(\neg P \vee Q)$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$

From the truth table we see that the two formulae are equivalent.

(Propositional) formulae can have very different structure, but the same values under the same interpretations, for all possible interpretations.

The question is how to detect the that two formulae are equivalent? Truth tables work, but are very expensive. Can we do better? Can we try (something like) simplification?

Exercise
How much time does a truth table take?

Consider the following table, taken from [Kleinberg and Tardos, 2006]:

	n	$nlog_2n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} yrs
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 yrs	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12892 yrs	10^{17} yrs	very long
$n = 1000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100000$	< 1 sec	2 sec	3 hours	32 yrs	very long	very long	very long
$n = 1000000$	1 sec	20 sec	12 days	31710 yrs	very long	very long	very long

- running times are estimated for a processor capable of performing 1 million high level instructions per second (MIPS),
- "very long" is considered to be running time that exceeds 10^{25} years,
- for comparison, currently (2016) available processors are capable of $\sim$ 10.000 - 238,310 MIPS
  (see, for example, estimates at `http://en.wikipedia.org/wiki/Instructions_per_second`).
- expensive is still **very very** bad.

Simplification

To decide if 2 expressions are equivalent (in some sense), *"reduce" ("simplify")* each of them to their (unique) *normal form*, by applying finitely many equivalence preserving steps. If these normal forms are (basically) the same, then the initial expressions are equivalent. Even if the normal form is not unique, it can still be useful, in that it contains information which could lead to the solving of the problem. This is illustrated in Figure 3.

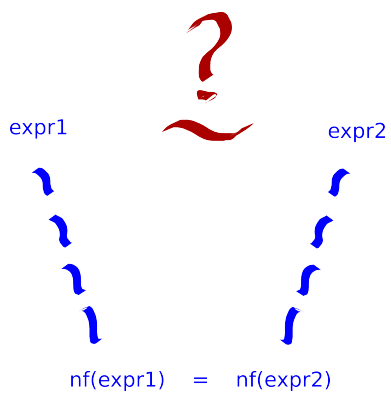


Figure 3: Simplification - transforming expressions to normal forms by equivalence rewriting. Expressions are equivalent if they can be transformed to the same normal form.

2.3.2 A Catalog of Equivalent Formulae

Extending the Language of Propositional Logic

We extend the language with two symbols, $\bot$, $\top$:

- $\bot$ denote formulae that are always false (unsatisfiable, inconsistent), i.e. for all truth valuations $\mathcal{T}$, $\upsilon_{\mathcal{T}}(\bot) = \mathbb{F}$;
- $\top$ denote formulae that are always true (valid), i.e. for all truth valuations $\mathcal{T}$, $\upsilon_{\mathcal{T}}(\top) = \mathbb{T}$;

Remark.

- $F \sim \bot$ iff F is unsatisfiable (inconsistent);
- $F \sim \top$ iff F is valid.

In the following, F, G, H are propositional formulae.

- Reduction Laws:

$$\begin{array}{ll} \text{(a)} & (F \leftrightarrow G) \sim (F \rightarrow G) \wedge (G \rightarrow F), \\ \text{(b)} & (F \rightarrow G) \sim (\neg F \vee G). \end{array}$$

- Commutative Laws:

$$\begin{array}{ll} \text{(a)} & F \vee G \sim G \vee F, \\ \text{(b)} & F \wedge G \sim G \wedge F, \end{array}$$

$$\text{(c)} \quad F \leftrightarrow G \sim G \leftrightarrow F.$$

- Associative Laws:

  (a) $(F \vee G) \vee H \sim F \vee (G \vee H)$,
  (b) $(F \wedge G) \wedge H \sim F \wedge (G \wedge H)$,
  (c) $(F \leftrightarrow G) \leftrightarrow H \sim F \leftrightarrow (G \leftrightarrow H)$.

- Distributive Laws:

  (a) $F \vee (G \wedge H) \sim (F \vee G) \wedge (F \vee H)$,
  (b) $F \wedge (G \vee H) \sim (F \wedge G) \vee (F \wedge H)$,

  (c) $(F \vee G) \to H \sim (F \to H) \wedge (G \to H)$,
  (d) $(F \wedge G) \to H \sim (F \to H) \vee (G \to H)$,
  (e) $F \to (G \vee H) \sim (F \to G) \vee (F \to H)$,
  (f) $F \to (G \wedge H) \sim (F \to G) \wedge (F \to H)$,
  (g) $(F \wedge G) \to H \sim F \to (G \to H)$.

- Laws of "True" and "False":

  (a) $\neg\top \sim \bot$,
  (b) $\neg\bot \sim \top$,
  (c) $F \vee \bot \sim F$,
  (d) $F \wedge \top \sim F$,
  (e) $F \vee \top \sim \top$,
  (f) $F \wedge \bot \sim \bot$,
  (g) $\bot \to F \sim \top$,
  (h) $F \to \top \sim \top$.

- Idempocy rules:

  (a) $F \wedge F \sim F$,
  (b) $F \vee F \sim F$.

- Absorbtion Laws:

  (a) $F \vee (F \wedge G) \sim F$,
  (b) $F \wedge (F \vee G) \sim F$.

- "Annihilation" Laws:

  (a) $F \vee \neg F \sim \top$, ("tertium non datur")
  (b) $F \wedge \neg F \sim \bot$,

  (c) $F \to F \sim \top$.

- Negation Laws:

  (a) $\neg(\neg F) \sim F$, ("double negation")
  (b) $\neg(F \vee G) \sim \neg F \wedge \neg G$, ("De Morgan")
  (c) $\neg(F \wedge G) \sim \neg F \vee \neg G$, ("De Morgan")

  (d) $\neg(F \to G) \sim F \wedge (\neg G)$,
  (e) $\neg(F \leftrightarrow G) \sim F \leftrightarrow (\neg G)$.

- Other Laws:
$$(a) \quad F \to G \sim F \leftrightarrow (F \wedge G),$$
$$(b) \quad F \to G \sim G \leftrightarrow (F \vee G).$$

Conjunctions and Disjunctions of Formulae

REMARK. Because of the associative laws, parantheses can be dropped, e.g. in $(F \vee G) \vee H$ or in $F \vee (G \vee H)$.

Definition 29 (Conjunction of formulae)**.** Let $F_1, F_2, \ldots, F_n$ be formulae. The formula $F_1 \wedge \ldots \wedge F_n$, $n \geq 1$, is called *the conjunction of* $F_1, \ldots, F_n$, and its value under a truth valuation $\mathcal{T}$ is:

$$\mathcal{B}_\wedge(v_\mathcal{T}(F_1), \ldots, v_\mathcal{T}(F_n)) = \begin{cases} \mathbb{T}, & \text{if } v_\mathcal{T}(F_1) = \ldots = v_\mathcal{T}(F_n) = \mathbb{T} \\ \mathbb{F}, & \text{otherwise} \end{cases}.$$

Definition 30 (Disjunction of formulae)**.** Let $F_1, F_2, \ldots, F_n$ be formulae. The formula $F_1 \vee \ldots \vee F_n$, $n \geq 1$, is called *the disjunction of* $F_1, \ldots, F_n$, and its value under a truth valuation $\mathcal{T}$ is:

$$\mathcal{B}_\vee(v_\mathcal{T}(F_1), \ldots, v_\mathcal{T}(F_n)) = \begin{cases} \mathbb{F}, & \text{if } v_\mathcal{T}(F_1) = \ldots = v_\mathcal{T}(F_n) = \mathbb{F} \\ \mathbb{T}, & \text{otherwise} \end{cases}.$$

Principle of Duality

De Morgan's laws ensure us of the following fact:

- starting from a tautology involving only $\top, \bot, \wedge, \vee, \neg$,
- *the principle of duality* states that one can swap the '$\wedge$'s and '$\vee$'s, swap the '$\top$'s and the '$\bot$'s, and then negate the result,
- and get another tautology.

Example 31. Start with $F \vee \neg F$, swap, obtaining $F \wedge \neg F$, and then negate, obtaining $\neg(F \wedge \neg F)$ (the principle of noncontradiction).

2.4 Normal Forms

Negation Normal Form

Definition 32 (Literal)**.** A *literal* is an atom or the negation of an atom. Atoms are called positive literals, negations of atoms are called negative literals.

Definition 33 (Negation Normal Form)**.** A formula F is said in *negation normal form (NNF)* , iff:

- F is $\top$ or F is $\bot$;
- F is constructed from literals, using only the binary connectives '$\wedge$' and '$\vee$'.

Example 34. Let P, Q, R, S be atoms. The following are in NNF:

$$\top,$$

$$P,$$

$$P \wedge (Q \wedge (\neg R \vee S)),$$

however,

$$\neg(\neg P),$$

$$\neg(P \vee \neg Q) \wedge R,$$

are not.

Transformation to NNF

Remark. Any propositional logic formula can be transformed into an equivalent formula in NNF.

Transformation to NNF

1. use the reduction laws (given in the catalog of equivalent formulae) to eliminate '$\leftrightarrow$', '$\rightarrow$';
2. repeatedly use the double negation and De Morgan's laws to eliminate '$\neg\neg$'s and '$\neg(\ldots)$'s.
3. at each of the steps above, it is useful to perform simplifications of $\top$ and $\bot$ (using the laws of true and false, annihilation, idempocy).

Disjunctive Normal Form, Conjunctive Normal Form

Definition 35 (Disjunctive Normal Form)**.** A formula F is said to be in *disjunctive normal form (DNF)* iff F has the form $F \doteq F_1 \vee \ldots \vee F_n$, $n \geq 1$, and each of $F_1, \ldots, F_n$ is a conjunction of literals.

Example 36 (Formulae in DNF). Let P, Q, R be atoms,

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

P,

$P \wedge Q$,

$P \vee Q \vee R$,

are in DNF.

Remark. Any formula which is in DNF is also in NNF.

Definition 37 (Conjunctive Normal Form)**.** A formula F is said to be in *conjunctive normal form (CNF)* iff F has the form $F \doteq F_1 \wedge \ldots \wedge F_n$, $n \geq 1$, and each of $F_1, \ldots, F_n$ is a disjunction of literals.

Example 38 (Formulae in CNF). Let P, Q, R be atoms,

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

P,

$P \wedge Q$,

$P \vee Q \vee R$,

are in CNF.

Remark. Any formula which is in CNF is also in NNF.

Normal Form Transformations

So far, we have seen that any formula can be transformed into its NNF equivalent, then defined DNF, CNF. The question now is how to transform the formula from NNF into DNF (or CNF) and if we can do that, what does that tell us? For once, if two formulae have the same normal form, they are equivalent, but, does it help in detecting tautologies, satisfiable formulae, unsatisfiable formulae? And is it better than truth tables?

DNF Transformations

DNF Transformation via Truth Tables

Given a formula, one method to obtain a DNF equivalent formula is through truth tables, by:

- selecting all the rows that evaluate to $\mathbb{T}$,

- for each such row, constuct a conjunct of literals in the following way: positive literals for corresponding $\mathbb{T}$ values for the propositional variable, and negative literals otherwise.
- the DNF is the disjunctions of all these conjunctions.

Example 39. DNF from truth tables

P	Q	$P \to Q$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$

The corresponding equivalent DNF formula:
$(P \wedge Q) \vee (\neg P \wedge Q) \vee (\neg P \wedge \neg Q)$.

REMARK.

- In the above example, the DNF equivalent form is quite different from the well-known equivalent $(\neg P \vee Q)$, i.e. it is not the simplest DNF (also DNF's are not unique).
- In general, getting the DNF from the truth table is expensive: one has to construct the truth tables.

DNF Transformation

Transformation to DNF

An arbitrary propositional formula can be transformed in an equivalent DNF by carrying out the following steps:

- bring the formula into NNF;
- repeatedly apply the tautologies:

$$F \wedge (G \vee H) \sim (F \wedge G) \vee (F \wedge H),$$

$$(F \vee G) \wedge H \sim (F \wedge H) \vee (G \wedge H),$$

  starting with the outermost '$\wedge$', until the normal form is reached.

Example 40 (DNF transformation). Transform the following formula into its DNF:

$$(F \wedge (G \vee H)) \wedge (\neg F \vee \neg G).$$

The formula is already in NNF.

$$\begin{aligned}
&(F \wedge (G \vee H)) \underset{distr}{\wedge} (\neg F \vee \neg G) \sim \\
&\quad (((F \wedge (G \vee H)) \underset{distr}{\wedge} \neg F) \vee ((F \wedge (G \vee H)) \underset{distr}{\wedge} \neg G)) \sim \\
&\quad (((F \wedge G) \vee (F \wedge H)) \underset{distr}{\wedge} \neg F) \vee ((F \wedge G) \vee (F \wedge H)) \underset{distr}{\wedge} \neg G)) \sim \\
&\quad (((F \underset{ann}{\wedge} G \wedge \neg F) \vee (F \underset{ann}{\wedge} H \wedge \neg F)) \vee ((F \underset{ann}{\wedge} G \wedge \neg G) \vee (F \wedge H \wedge \neg G))) \sim \\
&\quad (\bot \vee \bot \vee \bot (F \wedge H \wedge \neg G)) \sim \\
&\quad (F \wedge H \wedge \neg G).
\end{aligned}$$

REMARK. In the above, *distr*, *ann* indicate the place where the distributive law (of '$\wedge$' over '$\vee$') and the annihilation law (for '$\wedge$') were applied, respectively, at each step.

When transforming a formula into its DNF equivalent, it is a good idea to perform, after each transformation, simplification steps:

- *contradictions* within conjunctions:
  - if both a subformula and its negation show up in the same conjunction, then the conjunction is a contradiction (i.e. equivalent to $\bot$),
  - since $P \vee \bot \sim P$, contradictions can be discarded;
- *subsumption*:
  - since $(F \wedge G) \vee F \sim (F \vee F) \wedge (G \vee F) \sim F \wedge (G \vee F)$, we see that if F is true then the initial formula is true, and if F is false then the initial formula is false (read evaluated to false under some interpretation), i.e. the truth value of G plays no part in the result of the evaluation,
  - therefore, in fact $(F \wedge G) \vee F \sim F$,
  - therefore, $(F \wedge G)$ can be safely removed from the disjunct (we say that $(F \wedge G)$ *is subsumed* by F),
  - due to the associativity of '$\wedge$' this can be generalized (i.e. F and G can stand in for conjunctions of literals).

Example 41 (DNF transformation, with contradiction checking.). Transform the following formula into its DNF:

$$(F \wedge (G \vee H)) \wedge (\neg F \vee \neg G).$$

$$\begin{aligned}
&(F \wedge (G \vee H)) \underset{distr}{\wedge} (\neg F \vee \neg G) \sim \\
&\quad (((F \wedge (G \vee H)) \wedge \neg F) \vee ((F \wedge (G \vee H)) \underset{distr}{\wedge} \neg G)) \sim \\
&\quad ((F \wedge \overset{assoc \text{ and } ann}{G}) \vee (F \wedge H)) \underset{distr}{\wedge} \neg G)) \sim \\
&\quad ((F \wedge G \wedge \neg G) \vee (F \wedge H \wedge \neg G))) \sim \\
&\quad (F \wedge \overset{ann}{H} \wedge \neg G).
\end{aligned}$$

REMARK. Note that applying after each transformation a check for contradiction (i.e. looking to apply the annihilation law for '$\wedge$') can lead to much shorter derivations.

DNF and satisfiability

- Given a formula in DNF, the formula is satisfiable precisely if one of its disjuncts is satisfiable,
- a DNF disjunct (which is a conjunction) is satisfiable precisely if it does not contain complementary literals,

- therefore, transformation into DNF represents a method to establish satisfiablity of propositional formulae,
- however, in general, DNF does not offer a simple way to establish validity.

CNF Transformation

Transformation to CNF
An arbitrary propositional formula can be transformed in an equivalent DNF by carrying out the following steps:

- bring the formula into NNF;
- repeatedly apply the tautologies:

$$F \vee (G \wedge H) \sim (F \vee G) \wedge (F \vee H),$$

$$(F \wedge G) \vee H \sim (F \vee H) \wedge (G \vee H),$$

  starting with the outermost '$\vee$', until the normal form is reached.

When transforming into CNF, one should check at each step for *valid conjuncts*, i.e. disjunctions that contain complementary subformulae.

- *validity* within conjunctions:
  - if both a subformula and its negation show up in the same disjunction, then the disjunction is a valid (i.e. equivalent to $\top$),
  - since $P \vee \top \sim P$, valid conjuncts can be discarded;
- *absorbtion*:
  - $F \wedge (F \vee G) \sim F$.

Remark. DNF and CNF transformations are similar.

Example 42 (CNF transformation). Transform the following formula into its CNF:

$$(F \vee (G \wedge H)) \vee (\neg F \wedge \neg G).$$

The formula is already in NNF.

$$\begin{aligned}
&(F \vee (G \wedge H)) \underset{distr}{\vee} (\neg F \wedge \neg G) \sim \\
&\quad (((F \vee (G \wedge H)) \vee \neg F) \wedge ((F \vee (G \wedge H)) \underset{distr}{\vee} \neg G)) \sim \\
&\quad ((F \vee G) \wedge (F \vee H)) \underset{distr}{\vee} \neg G)) \sim \\
&\quad ((F \vee G \vee \neg G) \wedge (F \vee H \vee \neg G))) \sim \\
&\quad (F \vee H \vee \neg G).
\end{aligned}$$

$\underset{assoc \text{ and } ann}{\sim}$

CNF and validity

- Given a formula in CNF, the formula is valid iff each conjunct is valid,
- i.e. each disjunction contains complementary literals.
- If discarding of valid conjuncts is performed, then a formula is valid iff its CNF equivalent is $\top$.
- However, in general, it is not easy to establish satisfiability, given a CNF formula.
- In fact, the SAT problem (boolean satisfiability problem) is formulated as the decision whether a CNF formula is satisfiable,
- and SAT is the first problem proved to be NP complete (see [H.Gallier, 2003], pp. 50–54 for a discussion on NP and SAT).

DNF and CNF: Sharing Transformations

- due to the De Morgan laws, the CNF of a formula can be obtained directly from the DNF of its negation, by switching '$\wedge$'s for '$\vee$'s, and complementing the literals (i.e. positive literals become negative, and vice-versa).
- this allows sharing code for the normal form transformations, if these are implemented on a computer.
- Start with a formula, and its negation,
- bring them both to equivalent NNF,
- then transform them both to DNF,
- and finally, flip the '$\wedge$'s and '$\vee$'s in the DNF of the negation, and then complement the literals.

2.5 Applications of Propositional Logic: Digital Circuit Design

2.5.1 Propositional Logic and Boolean Functions

Consider the boolean domain $\{\mathbb{T}, \mathbb{F}\}$, let $\mathcal{A}_1, \mathcal{A}_2 \ldots$ range over $\{\mathbb{T}, \mathbb{F}\}$, and $A_1, A_2 \ldots$ be propositional variables (atoms). Propositional formulae describe boolean functions in a natural way:

Definition 43 (Boolean Functions described by Formulae)**.** Let F be a propositional formula. Let all the propositional variables (atoms) that occur in F be among $A_1, \ldots, A_n$. Then $[F]_n$ (*the n-ary boolean function described by* F) is defined as follows:

$$\begin{aligned}&[F]_n : \{\mathbb{T}, \mathbb{F}\}^n \longrightarrow \{\mathbb{T}, \mathbb{F}\},\\ &\quad [F]_n(\mathcal{A}_1, \ldots, \mathcal{A}_n) = v_{\mathcal{T}}(F),\\ &\quad \text{for all } \mathcal{T}, \text{ truth valuations s.t. } \mathcal{T}(A_i) = \mathcal{A}_i, \text{ for all } 1 \leq i \leq n.\end{aligned}$$

In fact, the truth table corresponding to a propositional formula provides the corresponding boolean function.

Lemma 44. *For every n-ary boolean function f there exists a propositional formula F such that $f = v_{\mathcal{T}}(F\rangle$.*

Proof. Let f be an n-ary boolean function , and let

$$({A_1}^1, \ldots, {A_n}^1), ({A_1}^2, \ldots, {A_n}^2), \ldots, ({A_1}^m, \ldots, {A_n}^m)$$

be the distinct n-tuples in $\{\mathbb{T}, \mathbb{F}\}^n$ for which f has the value $\mathbb{T}$. Then f can be described by the following propositional formula (in DNF):

$$F = C_{({\mathcal{A}_1}^1, \ldots, {\mathcal{A}_n}^1)} \vee C_{({\mathcal{A}_1}^2, \ldots, {\mathcal{A}_n}^2)} \vee \ldots \vee C_{({\mathcal{A}_1}^l, \ldots, {\mathcal{A}_n}^l)},$$

where $C_{({\mathcal{A}_1}^1, \ldots, {\mathcal{A}_n}^1)} = {A_1}^{\mathcal{A}_1} \wedge \ldots \wedge {A_n}^{\mathcal{A}_n}$, and ${A_i}^{\mathcal{A}} = \begin{cases} A_i & \text{if } \mathcal{A} = \mathbb{T} \\ \neg A_i & \text{otherwise} \end{cases}$. □

Example 45 (Majority function). Consider the boolean ternary function defined by the following table:

$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$f(\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$

The formula corresponding to f is:

$$F = (A_1 \wedge A_2 \wedge A_3) \vee (A_1 \wedge A_2 \wedge \neg A_3) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge A_2 \wedge A_3).$$

2.5.2 Complete Set of Boolean Operators

DNF, CNF show us that we can transform any propositional formula into equivalent forms expressed only in terms of $\{\neg, \wedge, \vee\}$. In fact, due to De Morgan's rules, we can do better: any formula can be expressed only in terms of $\{\neg, \wedge\}$, or only in terms of $\{\neg, \vee\}$. These are called *complete sets of boolean operators* (since they are sufficient to express any formula). Another example of complete set of boolean operators is $\{\rightarrow, \bot\}$ (exercise!).

Can we do better? Yes! The following are complete sets of boolean connectives: $\{|\}$ (NAND), $\{\triangledown\}$ (NOR), where, for all formulae F, G:

$$F|G = \neg(F \wedge G),$$
$$F \triangledown G = \neg(F \vee G).$$

2.5.3 Digital Circuit Design

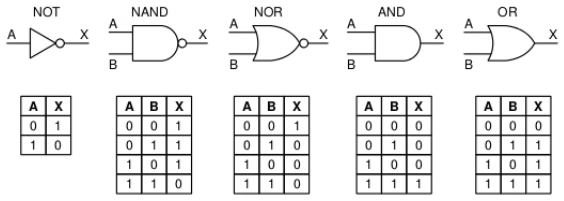
Definition 46 (Digital circuits)**.** *Digital circuits* are elctronic circuits whose inputs and outputs are electric signals (different voltage levels, usually 2). *Combinatorial digital circuits* are digital circuits whose output is determined solely by the values of their inputs.

Examples of combinatorial digital circuits: arithmetic-logical operations, memory addressing.

We can use boolean values to express these voltage levels (e.g. $\mathbb{T}$ - high voltage, $\mathbb{F}$ - low voltage). Therefore, digital circuits can be represented as boolean functions; A circuit is given by a *circuit diagram* which is described as a collection of *gates* and their interconnections. Each gate is a boolean function of its inputs. Each circuit (diagram) can be represented as a propositional formula. In a circuit diagram, input wires correspond to propositional atoms, internal wires correspond to subformulae.

Note.
The fact that digital circuits can be modelled by boolean functions was observed by Claude E. Shannon, and described in his 1937 master's thesis ("possibly the most important, and also the most famous, master's thesis of the century"), published later as [Shannon, 1938], and it had a tremendous impact on the subsequent development of computers.

Logical gates are the basic devices from which one can (conceptually) build up digital circuits. They can be described bypropositional connectives.



A	X
0	1
1	0

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

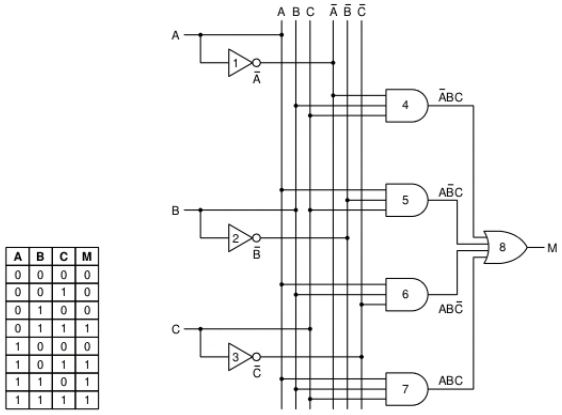
Figure 4: Logical Gates.

Example 47 (Logical Gates). In Figure 47 (source: teaching resources associated to [Tanenbaum and Austin, 2013]) 0 and 1 are used for $\mathbb{F}$ and $\mathbb{T}$, respectively.

Digital circuits can be designed using propositional logic. A digital circuit will have inputs and outputs. Each output can be seen as a boolean function of the inputs (therefore it has a corresponding boolean formula). They can be constructed in the following manner:

- start with the specification for the circuit, i.e. the desired behaviour of the circuit (a boolean function), represented by a (truth) table;
- from the truth table, construct DNF formula determined by the function represented in the table;
- the corresponding digital circuit can immediately be constructed (and it involves NOT, AND and OR gates).

Example 48 (Majority, revisited). The majority ternary function returns $\mathbb{T}$ if most of its inputs are $\mathbb{T}$, and $\mathbb{F}$ otherwise. See the design in Figure 48.



A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 5: Majority Circuit

Source: teaching resources associated to [Tanenbaum and Austin, 2013]. 0 and 1 are used for $\mathbb{F}$ and $\mathbb{T}$, respectively. $\bar{A}$ represents $\neg A$, ABC represents $A \wedge B \wedge C$.

Circuit Design Issues

Usually, the DNF form or a circuit is not suitable for practical implementation:

- AND, OR gates with a variable number of inputs are not practical,
- therefore, these will be replaced with 2-input AND, OR gates (based on the associativity of $\wedge, \vee$),
- moreover, for industrial implementation, it is more practical to have only one type of gate (even if the number of gates grows), therefore AND, OR, NOT gates will be replaced by one of NAND or NOR gates,

The number of gates should be minimized:

Example 49. Circuit simplification (reducing the number of gates, Figure 49). Source: teaching resources associated to [Tanenbaum and Austin, 2013].

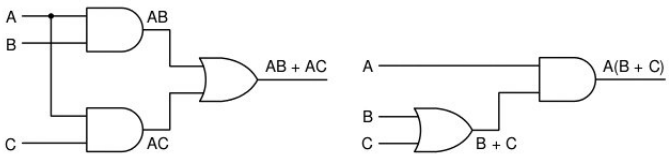


Figure 6: Circuit simplification.

Propositional logic and its machinery (equivalence transformations, reasoning by resolution, DP, DPLL) is an essential tool for logic circuit design:

- For circuit simplification (equivalence transformations),
- For circuit transformation (equivalence transformations): start with the DNF form from the specification, then transform the circuit to one using only one type of gate.
- For correctness of circuits (show that a "fast" circuit, e.g. with significantly fewer gates, is equivalent to one that is known to be correct, e.g. one that is obtained from a DNF directly from specification - use resolution, DP, DPLL);
- Circuits today are integrated on chips on a very large scale (hundreds of millions of transistors on a modern CPU, i.e. same class of magnitude for the number of gates), therefore efficient implementations of propositional logic reasoning methods are essential.

2.6 Logical Consequence

We are interested in showing whether some statement follows from some other statements.

Definition 50 (Propositional Logical Consequence)**.** Let $F_1, \ldots, F_n, G$ be propositional formuale. G is a *(propositional) logical consequence* of $F_1, \ldots, F_n$ iff, for all truth valuations (interpretations) $\mathcal{T}$,
if $\upsilon_{\mathcal{T}}(F_1) = \ldots = \upsilon_{\mathcal{T}}(F_n) = \mathbb{T}$, then $\upsilon_{\mathcal{T}}(G) = \mathbb{T}$.

Remark (Notation). To denote that G is a logical consequence of $F_1, \ldots, F_n$, write $F_1, \ldots, F_n \models G$.

Example 51. Let F and G be propositional formulae. Then G is a propositional consequence of F and $F \to G$.

F	G	$F \to G$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$

Deduction Theorem

Theorem 52 (deduction theorem for propositional logic)**.** *Let $F_1, \ldots, F_n, G$ be propositional formulae.*
G is a propositional consequence of $F_1, \ldots, F_n$ iff $((F_1 \wedge \ldots \wedge F_n) \to G)$ is valid.

Proof."$\Rightarrow$"(the direct implication).

To prove the implication,

assume G is a logical consequence of $F_1, \ldots, F_n$, and

show $((F_1 \wedge \ldots \wedge F_n) \rightarrow G)$ is valid, i.e. show that for every truth valuation $\mathcal{T}$,

$$\upsilon_{\mathcal{T}}(((F_1 \wedge \ldots \wedge F_n) \rightarrow G)) = \mathbb{T}.$$

For this, take $\mathcal{T}_0$ arbitrary but fixed truth valuation, and show

$$\upsilon_{\mathcal{T}_0}(((F_1 \wedge \ldots \wedge F_n) \rightarrow G)) = \mathbb{T}.$$

By the definition of truth value under truth valuation,

$$\begin{array}{l} \upsilon_{\mathcal{T}_0}(((F_1 \wedge \ldots \wedge F_n) \rightarrow G)) = \\ \quad \mathcal{B}_{\rightarrow}(\upsilon_{\mathcal{T}_0}((F_1 \wedge \ldots \wedge F_n)), \upsilon_{\mathcal{T}_0}(G)) = \\ \quad \mathcal{B}_{\rightarrow}(\mathcal{B}_{\wedge}(\upsilon_{\mathcal{T}}(F_1), \ldots, \upsilon_{\mathcal{T}_0}(F_n)), \upsilon_{\mathcal{T}_0}(G)), \end{array}$$

i.e. we have to show

$$\mathcal{B}_{\rightarrow}(\mathcal{B}_{\wedge}(\upsilon_{\mathcal{T}_0}(F_1), \ldots, \upsilon_{\mathcal{T}_0}(F_n)), \upsilon_{\mathcal{T}_0}(G)) = \mathbb{T}. \ (\star)$$

Two cases are possible:

Case: $\upsilon_{\mathcal{T}_0}(F_1) = \ldots = \upsilon_{\mathcal{T}_0}(F_n) = \mathbb{T}$.
Since G is a logical consequence of $F_1, \ldots, F_n$, $\upsilon_{\mathcal{T}_0}(G) = \mathbb{T}$, and

$\mathcal{B}_{\wedge}(\upsilon_{\mathcal{T}_0}(F_1), \ldots, \upsilon_{\mathcal{T}_0}(F_n)) = \mathbb{T}$, therefore $(\star)$ holds $(\mathcal{B}_{\rightarrow}(\mathbb{T}, \mathbb{T}) = \mathbb{T})$.

Case: not $(\upsilon_{\mathcal{T}_0}(F_1) = \ldots = \upsilon_{\mathcal{T}_0}(F_n) = \mathbb{T})$.
$\mathcal{B}_{\wedge}(\upsilon_{\mathcal{T}_0}(F_1), \ldots, \upsilon_{\mathcal{T}_0}(F_n)) = \mathbb{F}$, therefore $(\star)$ holds $(\mathcal{B}_{\rightarrow}(\mathbb{F}, \upsilon_{\mathcal{T}_0}(G)) = \mathbb{T})$.

□

Proof."$\Leftarrow$"(the inverse implication).
To prove the implication,

assume $((F_1 \wedge \ldots \wedge F_n) \rightarrow G)$ is valid, and
show G is a propositional logical consequence of $F_1, \ldots, F_n$.
For this, take an arbitrary but fixed truth valuation $\mathcal{T}_0$,
assume $\upsilon_{\mathcal{T}_0}(F_1) = \ldots = \upsilon_{\mathcal{T}_0}(F_n) = \mathbb{T}$, and
show $\upsilon_{\mathcal{T}_0}(G) = \mathbb{T}$.
From the assumption, we have

$$\upsilon_{\mathcal{T}_0}(((F_1 \wedge \ldots \wedge F_n) \rightarrow G)) = \mathbb{T}.$$

But

$$\begin{array}{l} \upsilon_{\mathcal{T}_0}(((F_1 \wedge \ldots \wedge F_n) \rightarrow G)) = \\ \quad \mathcal{B}_{\rightarrow}(\upsilon_{\mathcal{T}_0}((F_1 \wedge \ldots \wedge F_n)), \upsilon_{\mathcal{T}_0}(G)) = \\ \quad \mathcal{B}_{\rightarrow}(\mathcal{B}_{\wedge}(\upsilon_{\mathcal{T}_0}(F_1), \ldots, \upsilon_{\mathcal{T}_0}(F_n)), \upsilon_{\mathcal{T}_0}(G)) = \\ \quad \mathcal{B}_{\rightarrow}(\mathcal{B}_{\wedge}(\mathbb{T}, \ldots, \mathbb{T}), \upsilon_{\mathcal{T}_0}(G)) = \\ \quad \mathcal{B}_{\rightarrow}(\mathbb{T}, \upsilon_{\mathcal{T}_0}(G)), \end{array}$$

therefore, $\mathcal{B}_{\rightarrow}(\mathbb{T}, v_{\mathcal{T}_0}(G)) = \mathbb{T}$.

By the definition of $\mathcal{B}_{\rightarrow}$ this means $v_{\mathcal{T}_0}(G) = \mathbb{T}$.

□

Theorem 53 (alternative characterization of propositional consequences)**.**
Let $F_1, \ldots, F_n, G$ be propositional formulae.
G is a propositional consequence of $F_1, \ldots, F_n$ iff $(F_1 \wedge \ldots \wedge F_n \wedge \neg G)$ is unsatisfiable.

Proof. Exercise.

□

The significance of the Deduction Theorem

The deduction theorem is significant, in that it tells:

- that any logic equivalence in propositional logic can be decided by deciding the validity of a formula, for example by using reasoning (i.e. manipulation of formulae)
- if a method for reasoning establishes the validity of a certain formula, then we have logical consequence.

The alternative characterization tells us that deciding logical consequence can also be done by establishing unsatisfiability,e.g. by reasoning (i.e. manipulation of formulae). All we need is a method to do reasoning in propositional logic.

2.7 The Resolution Method in Propositional Logic

2.7.1 Clause Form of Propositional Formulae

Propositional resolution is a method for deciding the satisfiability of propositions or whether a proposition is a logical consequence of a set of others.

Propositional resolution works on representation of formulae, called "clause forms", which are equivalent to CNF.

Definition 54 (Propositional Clause Set)**.**

$\mathcal{K}$ is a *propositional clause set* iff $\mathcal{K}$ is a finite set of propositional clauses.

$\mathcal{C}$ is a *propositional clause* iff $\mathcal{C}$ is a finite set of propositional literals.

L is a *propositional literal* iff L is an atom, or the negation of an atom.

Example 55 (Propositional Clause Set). $\mathcal{K} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$, where

$$\mathcal{C}_1 = \{F_1, F_2\}, \mathcal{C}_2 = \{\neg F_2\}, \mathcal{C}_3 = \{\neg F_1, F_2, \neg F_3\}.$$

NOTATION. Let $\mathcal{K}, \mathcal{C}$ be a propositional clause set, propositional clause, respectively. We denote $\mathcal{K}^*, \mathcal{C}^*$, the propositional formula detedmined by $\mathcal{K}, \mathcal{C}$ respectively.

Definition 56 (Formula Determined by a Clause Set)**.**

If $\mathcal{C} = \{L_1, \ldots, L_m\}$ is a propositional clause, then $\mathcal{C}^* = L_1 \vee \ldots \vee L_m$.

If $\mathcal{K} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ is a propositional clause set then $\mathcal{K}^* = {\mathcal{C}_1}^* \wedge \ldots \wedge {\mathcal{C}_n}^*$.

REMARK. The definition above introduces formulae that are not uniquely defined (because elements in sets are not ordered). However, these are unique modulo the equivalence of formulae. We want to identify clauses that contain the same literals, in possibly different order, and possibly different number of occurences.

Example 57 (Formula determined by a clause set).
Let $\mathcal{K} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$, where $\mathcal{C}_1 = \{F_1, F_2\}$, $\mathcal{C}_2 = \{\neg F_2\}$,$\mathcal{C}_3 = \{\neg F_1, F_2, F3\}$.

$$\mathcal{K}^* = {\mathcal{C}_1}^* \wedge {\mathcal{C}_2}^* \wedge {\mathcal{C}_3}^* = (F_1 \vee F_2) \wedge (\neg F_2) \wedge (\neg F_1 \vee F_2 \vee F_3).$$

Example 58 (Particular clause sets and their corresponding formulae).

$\{\}$ i.e. the empty clause set $\varnothing$ is a clause set (the set containing a finite number of clauses, 0),

$\{\{\}\}$, i.e. $\{\varnothing\}$ is a clause set (the set containing one empty clause).

Since no propositional variable occurs in these clause sets, the corresponding formulae should be chosen from those propositions that do not contain any variables, i.e. $\top, \bot$. The question is how to make the assignment (see below, propositional resolution method).

From any formula, we can obtain, in a natural way, its clause set form, by transforming it into CNF, and reading the clauses directly from the disjuncts. In particular, the clause set corresponding to $\top$ is the **empty clause set** $\{\}$, the clause set corresponding to $\bot$ is (following simplification) **the clause set containing the empty clause** $\{\{\}\}$. Also, for $\bot$, the corresponding clause is the empty clause.

ATTENTION! Note the distinction between the empty clause set (the clause set which is empty) and the empty clause.

The clause set determined by a set of formulae can be obtained by union of the clause sets corresponding to each formula.

Properties of propositions extend in a natural way to their clause set form: a clause set is satisfiable (valid, unsatisfiable), iff the proposition determined by the clause set is satisfiable (valid, unsatisfiable).

2.7.2 Propositional Resolution

Lemma 59 (Correctness of a propositional resolution step)**.** *Let B_1, B_2, C be propositional formulae. Then $B_1 \vee B_2$ is a propositional consequence of $C \vee B_1$ and $\neg C \vee B_2$.*

Proof. Let $\mathcal{T}$ be an arbitrary thruth valuation such that

$$\upsilon_{\mathcal{T}}(C \vee B_1) = \mathbb{T}, \quad (1)$$
$$\upsilon_{\mathcal{T}}(\neg C \vee B_2) = \mathbb{T}. \quad (2)$$

Case $\upsilon_{\mathcal{T}}(B_1) = \mathbb{T}$, then $\upsilon_{\mathcal{T}}(B_1 \vee B_2) = \mathbb{T}$.

Case $\upsilon_{\mathcal{T}}(B_1) = \mathbb{F}$, then, from (1), $\upsilon_{\mathcal{T}}(C) = \mathbb{T}$, i.e. $\upsilon_{\mathcal{T}}(\neg C) = \mathbb{F}$, i.e. $\upsilon_{\mathcal{T}}(B_2) = \mathbb{T}$ (from (2)), i.e. $\upsilon_{\mathcal{T}}(B_1 \vee B_2) = \mathbb{T}$.

□

Definition 60 (Complementary literals)**.** Two propositional literals L_1, L_2 are *complementary* iff L_1 is an atom and L_2 is its negation (or viceversa).

Definition 61 (Propositional resolvent)**.** Let $\mathcal{C}' = \{L_1', \ldots, L_{m'}{}'\}, \mathcal{C}'' = \{L_1'', \ldots, L_{m''}{}''\}$ be two propositional clauses such that L_1', L_1'' are complementary literals. Then $\mathcal{C}$ is a *propositional resolvent* of $\mathcal{C}'$ and $\mathcal{C}''$ iff

$$\mathcal{C} = \{L_2', \ldots, L_{m'}{}', L_2'', \ldots, L_{m''}{}''\}.$$

Example 62 (Propositional resolvent). $\{F_2, F_3\}$ is a propositional resolvent (in fact the only propositional resolvent) of $\{F_1, F_2\}$ and $\{\neg F_1, F_3\}$.

Theorem 63 (Propositional Resolution Method)**.** *Consider the following problem:*

- *Given: $\mathcal{K}$, a propositional clause set,*
- *Question: Is $\mathcal{K}^*$ satisfiable?*

This problem can be solved by the following algorithm:

Propositional resolution

$\mathcal{K}' := \mathcal{K}$

while exists $\mathcal{C}$ *such that*

$\mathcal{C}$ *is a propositional resolvent of two clauses in* $\mathcal{K}'$ *and* $\mathcal{C} \notin \mathcal{K}'$

do

if $\mathcal{C} = \varnothing$ *then answer: "Not Satisfiable"*

$\underline{else}\ \mathcal{K}' := \mathcal{K}' \cup \{\mathcal{C}\}$

answer: "Satisfiable".

Proof Sketch. We have to show:

1. The algorithm terminates;
2. If the algorithm terminates with the answer "Not Satisfiable", then $\mathcal{K}^*$ is not satisfiable;
3. If the algorithm terminates with the answer "Satisfiable", then $\mathcal{K}^*$ is satisfiable.

1. Termination

The algorithm terminates because from finitely many literals occuring in the initial clause $\mathcal{K}$ one can form only finitely many clauses.

(If there are n distinct variables in $\mathcal{K}$, then at most 4^n many clauses - including many redundant ones - can be formed (why?)). □

2. Answer "Not Satisfiable".

This answer is produced only when the clause $\varnothing$ is generated by a resolvent.

This clause can only be generated by clauses of the form $\{L'\}, \{L''\}$, where the literals L', L'' are complementary.

By the lemma of the correctness of a propositional resolution step, each clause produced by the application of the algorithm is a propositional consequence of $\mathcal{K}$.

Hence $\{L'\}, \{L''\}$ are propositional consequences of $\mathcal{K}$. Therefore, if a truth valuation $\mathcal{T}$ satisfied $\mathcal{K}$, then it would satisfy both L', L''. This is a contradiction to the fact that L', L'' are complementary.

□

3. Answer "Satisfiable".

Idea: If the answer is "Satisfiable", one can construct a satisfying truth valuation for $\mathcal{K}$ from the final clause set: start with the shortest clauses, construct for them a partial satisfying truth valuation, then "propagate" the truth valuation towards $\mathcal{K}$. (see Examples, satisfiable case).

□

Example 64. Decide by resolution whether the following set of formulae is satisfiable, and if it is, construct a satisfying truth valuation:

(a)

(1) $F_1 \vee F_2$,
(2) $F_1 \vee \neg F_3$,
(3) $\neg F_1 \vee F_3$,
(4) $\neg F_1 \vee \neg F_2$,
(5) $F_3 \vee \neg F_2$,
(6) $\neg F_3 \vee F_2$.

Solution: See [Buchberger, 1991], pag. 66, Example 2.84. (Note that there is no transformation into clause form there. However the transformation makes a potential implementation easier: it is easier to operate on sets than to operate on formulae).

(b)

(1) $F_1 \vee F_2$,
(2) $\neg F_2$,
(3) $\neg F_1 \vee F_2 \vee \neg F_3$.

Solution: See [Buchberger, 1991], pag. 67, Example 2.85. (Again no transformation. However, the corresponding clause set formulation should not be hard to see).

(c)

(1) $F_1 \vee \neg F_2$,
(2) $F_1 \vee F_3$,
(3) $\neg F_2 \vee F_3$,
(4) $\neg F_1 \vee F_2$,
(5) $F_2 \vee \neg F_3$,
(6) $\neg F_1 \vee \neg F_3$.

Solution:

– Transform the problem in the clausal form:

(1) $\{F_1, \neg F_2\}$,
(2) $\{F_1, F_3\}$,
(3) $\{\neg F_2, F_3\}$,
(4) $\{\neg F_1, F_2\}$,
(5) $\{F_2, \neg F_3\}$,
(6) $\{\neg F_1, \neg F_3\}$.

– Apply resolution:

~~(7)~~	$\{\neg F_2, F_2\}$	~~from (1) and (4)~~,
(8)	$\{F_1, \neg F_3\}$	from (1) and (5),
(9)	$\{\neg F_2, \neg F_3\}$	from (1) and (6),
(10)	$\{F_1\}$	from (2) and (8),
(11)	$\{\neg F_3\}$	from (10) and (6),
(12)	$\{F_2\}$	from (10) and (4),
(13)	$\{F_3\}$	from (12) and (3),
(14)	$\varnothing$	from (11) and (13).

Answer: "Not Satisfiable".

REMARK. The clause (7) can be eliminated, as it plays no role in the evaluation of the clause set. The clause corresponds to the formula $\neg F_2 \vee F_2$, which is always true (i.e. valid, tautology). Tautology elimination should always be performed, to avoid unnecessary resolution steps.

Example 65. Decide, by resolution, whether $\neg F_1 \wedge F_2$ is a logical consequence of $(F_1 \vee F_2) \to F_3$ and $\neg F_3$.

Solution: See [Buchberger, 1991], pag. 67, Example 2.86.

Example 66. (c)

(1) $F_1 \vee \neg F_2$,
(2) $F_1 \vee F_3$,
(3) $\neg F_2 \vee F_3$,
(4) $\neg F_1 \vee F_2$,
(5) $F_2 \vee \neg F_3$,
(6) $\neg F_1 \vee \neg F_3$.

Example 67. Decide, by resolution, whether $\neg F_1 \wedge F_2$ is a logical consequence of $(F_1 \vee F_2) \rightarrow F_3$ and $\neg F_3$.

2.7.3 Improvements of Propositional Resolution

Davis-Putnam method - DP(1960)
The Davis-Putnam method for deciding the satisfiability of a clause set $\mathcal{K}$ consists in the application of 3 steps:

I. *the 1-literal rule (unit propagation):*
   - if a single literal L appears in a clause set, remove any instances of $\neg L$ from the other clauses of $\mathcal{K}$;
   - remove any instances of clauses containing L, including the clause itself,

II. *the pure literal rule:* If a literal occurs only positively or negatively in the clause set, delete all clauses containing it;

III. *resolution:* on the remaining clauses, apply propositional resolution.

DP returns the answer "Satisfiable" when none of the rules can be applied, and "Not Satisfiable" when the empty clause $\varnothing$ is generated.

Example 68 (revisited). Solution by DP:

- The clause set:

  (1) $\{F_1, \neg F_2\}$,
  (2) $\{F_1, F_3\}$,
  (3) $\{\neg F_2, F_3\}$,
  (4) $\{\neg F_1, F_2\}$,
  (5) $\{F_2, \neg F_3\}$,
  (6) $\{\neg F_1, \neg F_3\}$.

- Rule I and II are not applicable, do resolution, and check after each step whether I, II are applicable

~~(7)~~	$\{\neg F_2, F_2\}$	~~from (1) and (4)~~, tautology!
(8)	$\{F_1, \neg F_3\}$	from (1) and (5), I, II not applicable
(9)	$\{\neg F_2, \neg F_3\}$	from (1) and (6), I, II not applicable
(10)	$\{F_1\}$	from (2) and (8), I. applicable!!!

- Apply I (unit propagation) with $\{F_1\}$:
  - erase $\neg F_1$ from (4),(6);
  - delete (1), (2), (8), (10).

  The clause set becomes:

  (3) $\{\neg F_2, F_3\}$,
  (4') $\{F_2\}$,
  (5) $\{F_2, \neg F_3\}$,
  (6') $\{\neg F_3\}$,
  (9) $\{\neg F_2, \neg F_3\}$.

- Rule I (unit propagation) is applicable with $\{F_2\}$:

- erase $\neg F_2$ from (3), (9);
- delete (4'), (5),

The clause set becomes:

$$\begin{array}{ll} (3') & \{F_3\}, \\ (6') & \{\neg F_3\}, \\ (9') & \{\neg F_3\}. \end{array}$$

- Rule I (unit propagation) is applicable with $\{F_3\}$:
  - erase $\neg F_3$ from (6'), (9') STOP, these both become $\{\}$, i.e. $\varnothing$,

Answer: "Not Satisfiable".

Davis Putnam Logemann Loveland method - DPLL (1962)
DP could (and it used to) run out of memory. DPLL replaces resolution with the splitting rule

I. *the 1-literal rule (unit propagation):*
   - if a single literal L appears in a clause set, remove any instances of $\neg L$ from the other clauses of $\mathcal{K}$;
   - remove any instances of clauses containing L, including the clause itself,

II. *the pure literal rule:* If a literal occurs only positively or negatively in the clause set, delete all clauses containing it;

III. *splitting:* The satisfiability of $\mathcal{K}'$ (the current set of clauses, when I. and II. are no longer applicable) is reduced to the satisfiability of $\mathcal{K}' \cup \{\{L\}\}, \mathcal{K}' \cup \{\{\neg L\}\}$, where L is a literal from $\mathcal{K}'$ ($\mathcal{K}'$ is satisfiable exactly if one of the two is).

DPLL returns the answer "Not Satisfiable" when for **all** subproblems generated by applications of the splitting rule, the **empty clause** $\varnothing$ is generated. It returns the answer "Satisfiable" if at least for **one** of the subproblems generated by the splitting rule no rules are applicable. This, in fact, amounts to the situation when the **empty clause set** is generated, i.e. no clauses are left. This always happens, as the splitting rule can be applied as long as there are clauses left in the clause set. Therefore: ATTENTION!!!. The empty clause set gives the answer "Satisfiable", whereas the empty clause gives the answer "Unsatisfiable" (if it is generated for all subproblems).

Example 69 (revisited, again). Solution by DPLL:

- The clause set:

$$\begin{array}{ll} (1) & \{F_1, \neg F_2\}, \\ (2) & \{F_1, F_3\}, \\ (3) & \{\neg F_2, F_3\}, \\ (4) & \{\neg F_1, F_2\}, \\ (5) & \{F_2, \neg F_3\}, \\ (6) & \{\neg F_1, \neg F_3\}. \end{array}$$

- Since none of the rules I and II are applicable, we apply III (splitting) using F_1:
  - **split on the positive literal:** adding (7) $\{F_1\}$ to the clause set:

* unit propagation (I) is applicable with F_1: delete $\neg F1$ from (4), (6), then delete clauses (1), (2), (7): the clause set becomes :

$$\begin{array}{ll} (3) & \{\neg F_2, F_3\}, \\ (4') & \{F_2\}, \\ (5) & \{F_2, \neg F_3\}, \\ (6') & \{\neg F_3\}, \end{array}$$

* apply (I) with F_2, delete $\neg F_2$ from (3), delete clauses (4'),(5), yieding

$$\begin{array}{ll} (3') & \{F_3\}, \\ (6') & \{\neg F_3\}. \end{array}$$

* a last application of (I) yields $\varnothing$. Answer: "Not Satisfiable".

– **split on the negative literal:** adding (7) $\{\neg F_1\}$ to the clause set: Exercise (it will also yield the empty clause ($\varnothing$) not surprisinly). But note that *both* splits have to be carried out to establish unsatisfiability.

Comparing DP, DPLL to resolution

- All the methods return the answer "Not Satisfiable" when an empty clause ($\varnothing$) is generated.
- The empty clause ($\varnothing$) is generated in the case of DP, DPLL by the 1-literal rule.
- As opposed to the method presented for resolution, DP and DPLL do not keep the initial clause set, rules I (1-literal), II (pure literal) have as an effect the deletion of clauses.
- DP, DPLL return the answer "Satisfiable" iff none of the rules are applicable. (In particular, DPLL will be applied as long as there are literals).
- For each of the methods, ***choice*** plays an important role: for resolution, which clauses to be resolved, for DP, DPLL which literal clause should be propagated. A good choice will make the algorithm produce an answer quicly. Bad choices will lead to very lengthy executions (and possibly memory consuming, for instance for DP).

CHAPTER 3

Predicate Logic as a Working Language

3.1 The Importance of Predicate Logic

1. General enough to be a formal frame for all of mathematics.
2. It can serve as a formal frame for computer science.
3. It is where most of the foundational research effort was concentrated in the past decades (so it is very well understood).

A Frame for Doing Mathematics

- By no means trivial.
- Took thousands of years to achieve these results.

  1879 Frege - first complete syntactical presentation,

  1930 Gödel - completeness of first order predicate logic,

  1936 Church, Turing - undecidability.

- In fact, even a restricted form, first order predicate logic (together with set theory), is sufficient for this.

A Frame for Algorithmic Problem Solving

- Predicate logic is a general frame for proving mathematical concepts and facts, therefore it can also be used as a universal frame for specifying problems.
- Proving is computing (Robinson 1965 - resolution).
- Logic programming (Prolog).
- Unifying potential and practical power in data design and analysis (abstract data types, relational databases).

3.2 Syntax First Order Predicate Logic

The Language Symbols

Definition 70 (Vocabulary of first order predicate logic)**.** The *vocabulary of first order predicate logic* consists of

- The (countable) set $\mathcal{V}$ *of variables*(*individual variables*, *object variables*): x, y, z, that range over arbitrary elements in a "universe of discourse".
- The set $\mathcal{F}$ *of function symbols* (*function names*, *function constants*): represent operations, processes in the universe of discourse:
  - examples: $+, \cdot$ (usually known as "addition" and "multiplication"),
  - functions have *arities* (number of arguments),
  - some functions are 0-ary, i.e. they have no arguments - they are called *constants*("individual constants"),
  - the 0-ary functions are in 1-to-1 correspondence with the elements of the universe of discourse, and are often identified with them,
  - for practical reasons, we can (and will) distinguish between function symbols and constants (the set of constants will be denoted by $\mathcal{C}$).
- The set $\mathcal{P}$ *of predicate symbols* (*relation symbols*) denote attribute of / relations between objects in the universe of discourse,
  - examples: $<, |$,
  - predicate symbols also have arities.

Remark. The sets of symbols used to denote variables, function symbols, predicate symbols and constants are disjoint.

Terms

Definition 71 (Terms)**.** *Terms of first order predicate logic* are defined inductively as follows:

- if x is a variable then x is a term,
- if f is a function symbol of arity n, and $t_1, \ldots, t_n$ are terms, then so is $f(t_1, \ldots, t_n)$,
- in particular, if c is a constant, then c is also a term.

Parsing terms in predicate logic

Some remarks and clarifications about the way to write and parse expressions are in order:

- **position:** function symbols can be used *prefix*: $f(x)$, *infix*: $x + y, x * y$, *postfix* $x!$ (factorial), or mixed $|x|$ (absolute value), $\sqrt[n]{x}$.
- **associativity**: $8/2/2$ can be parsed as $8/(2/2)$, i.e. 8 or $(8/2)/2$, i.e. 2, depending whether / (division) is right associative or left associative (arithmetic functions are usually considered left associative, but this has to be specified),
- **precedence**: $x + yz$ can be parsed as $(x + y)z$ or $x + (yz)$ (this is the usual use) according to the precedence of each of the function symbols.

Associativity and precedence simplify the writing of expressions in predicate logic, *but when in doubt, use parantheses!!!*

Formulae

Definition 72 (Formulae of first order logic)**.** *Formulae of first order predicate logic* are defined inductively as follows:

- *Atomic formulae* are formed from a predicate symbol and several terms, i.e. if p is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an atomic formula.
- Given the formulae A, B, use the propositional connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence) to form *compound formulae*:

$(\neg A)$	"not A"
$(A \wedge B)$	"A and B"
$(A \vee B)$	"A or B"
$(A \Rightarrow B)$	"A implies B", "if A then B"
$(A \Leftrightarrow B)$	"A iff B" (if and only if)

- Given a variable x and a formula A, one can form *quantified formulae*:

  $(\forall x A)$ "*for all* x A" - *universally quantified formula*,

  $(\exists x A)$ "*there exists* x *such that* A", "*for some* x, A" - *existential formula*,

  - *for all* $(\forall)$ is called the *universal quantifier*
  - *exists* $(\exists)$ is called the *existential quantifier*
  - the variable x becomes *bound* by the quantifiers.
- Remark. Attention, also in the case of atomic formulae, these can be written prefix $(odd(x))$, infix $(x < y)$, etc.

Free and bound variables

Definition 73 (Free and bound variables)**.**

- The variable x becomes *bound* by the quantifiers $\forall, \exists$ in $\forall x A$, $\exists x A$.
- Variables that are not bound (by a quantifier), are *free*.
- In terms (as defined above): variables are free (since terms do not contain quantifiers).
- In formulae (as defined above):
  - variables are free in atomic formulae,
  - variable x is free in $(\neg A)$ if it is free in A,
  - variable x is free in $(A \Box B)$ if it is free in A or in B, where $\Box$ is one of $\wedge, \vee \Rightarrow, \Leftrightarrow$,
  - variable x is free in $\forall y A$ if x is free in A,
  - variable x is free in $\exists y A$ if x is free in A.
- A formula with no free variables is called a *closed formula*.

Example 74.

- In the formula $\forall x(x + y = 2x + 4)$, x is bound and y is free.
- In the formula:

$$\forall \varepsilon \exists \delta \forall y(\mid y - x \mid < \delta) \Rightarrow \mid f \odot y - f \odot x \mid < \varepsilon,$$

  variables x and f are free, the others (ε, δ, y) are bound.
- In the formula $\forall x((x > y) \wedge (\exists y y > 5))$, x is bound, but y is both bound and free.

Substitution

- *Substitution of terms for variables* is an elementary process for deriving new formulae from given ones.
- Notation: if E is an expression (term or formula), v is a variable and t a term, then $E_{\{v \leftarrow t\}}$ is the expression obtained by substituting the term t for variable v in E.
- More generally, for $i = 1 \dots n$, if v_i are variables and t_i are terms, then $E_{\{v_1 \leftarrow t_1, \dots, v_n \leftarrow t_n\}}$ is the expression obtained by substituting the terms t_i for the variables x_i in E (at the same time).

Definition 75 (Substitution in first order logic)**.** For $i = 1 \dots n$, let x_i be variables and t_i be terms.

- We call $\{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$ a substitution of terms for variables.
- We will usually use Greek letters to denote substitutions ($\sigma, \lambda, \mu, \theta$).
- In particular, we will denote the empty substitution by ϵ.
- If E is an expression and σ a substitution, then we call E_σ, the expression obtained by substitution, *an instance* of E.
- Now let $\sigma = \{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$.
- *Substitution in terms* is defined inductively on the structure of terms as:
  - If $v \in \mathcal{V}$ (i.e. v is a variable),
    $$v_\sigma = \begin{cases} t_i & \text{if } v = x_i, \text{ for some } i \\ v & \text{otherwise} \end{cases}.$$
  - If $f \in \mathcal{F}$, is an m-ary function symbol and $s_1, \dots, s_m$ are terms, $f(s_1, \dots, s_m)_\sigma = f((s_1)_\sigma, \dots, (s_m)_\sigma)$.
- *Substitution in formulae* is defined inductively on the structure of formulae as:
  - If $p \in \mathcal{P}$ is an m-ary predicate symbol and $s_1, \dots, s_m$ are terms, $p(s_1, \dots, s_m)_\sigma = p((s_1)_\sigma, \dots, (s_m)_\sigma)$,
  - If A, B are formulae, $(\neg A)_\sigma = \neg(A_\sigma)$, $(A \Box B)_\sigma = (A_\sigma) \Box (B_\sigma)$, where $\Box$ is any of $\wedge, \vee, \Rightarrow, \Leftrightarrow$,
  - If A is a formula, v a variable,
    $$(QvA)_\sigma = \begin{cases} Qv(A_{(\sigma \setminus \{x_i \leftarrow t_i\})}) & \text{if } v = x_i \\ Qv(A_\sigma) & \text{otherwise} \end{cases},$$
    where Q is any of $\forall, \exists$. Intuitively, the bound variable is "protected" from substitution. If the bound variable appears in the substitution, it is ignored, and the rest of the substitution is applied on the quantified formula.

Example 76 (Substitution). $x + y + z_{\{x \leftarrow y, y \leftarrow 3, z \leftarrow x\}}$ is

- $y + 3 + x$

  and not
- $y + y + z_{\{y \leftarrow 3, z \leftarrow x\}}$, etc.

i.e. all substitutions happen at once, in parallel.

New substitutions can be obtained from existing ones by composition.

Definition 77 (Composition of substitutions)**.** Let $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ and $\sigma = \{y_1 \leftarrow s_1, \ldots, y_n \leftarrow s_k\}$ be substitutions, X, Y be the set of variables in these substitutions, respectively. $\theta\sigma$, *the composition of* θ and σ is the substitution

$$\theta\sigma = \{x_i \leftarrow t_i\sigma | x_i \in X, x_i \neq t_i\sigma\} \cup \{y_j \leftarrow s_j | y_j \in Y, y_j \notin X\}.$$

I.e. apply the substitution σ to the terms t_i of θ (unless they would collapse to $x_i \leftarrow x_i$), then append the fragment of σ whose variables are not already in θ.

Example 78. (Composition of substitutions)

- Example: let
$$\theta = \{x \leftarrow f(y), y \leftarrow f(a), z \leftarrow u\},$$
$$\sigma = \{y \leftarrow g(a), u \leftarrow z, v \leftarrow f(f(a))\},$$
then:
$$\theta\sigma = \{x \leftarrow f(g(a)), y \leftarrow f(a), u \leftarrow z, v \leftarrow f(f(a))\}.$$

Some properties of composition of substitutions

- Let E be an expression and θ, σ substitutions. Then $E(\theta\sigma) = (E\theta)\sigma$.
- Let θ, σ, λ be substitutions. Then $\theta(\sigma\lambda) = (\theta\sigma)\lambda$.

3.3 Semantics of First Order Predicate Logic

Interpretations

Definition 79 (Interpretation)**.** Let $\mathcal{L} = \{\mathcal{F}, \mathcal{P}, \mathcal{C}\}$ be the set of function, predicate and constant symbols of the language $\mathcal{L}$. Let D be a nonempty domain.

- An *interpretation* $\mathcal{I}$ of the language $\mathcal{L}$ into the domain D is a mapping that assigns:
  - to every function symbol $f \in \mathcal{F}$ a function of corresponding arity on D,
  - to every predicate symbol $p \in \mathcal{P}$ a predicate of corresponding arity on D,
  - to every constant symbol $c \in \mathcal{C}$ a constant of the domain D.

Example 80 (Interpretations).

Consider the formula $\forall x p(a, x \odot b)$. The following are its interpretations:

- $\mathcal{I}_1 : \mathcal{I}_1(D) := \mathbb{N}, \mathcal{I}_1(p) := \leq, \mathcal{I}_1(\odot) := +, \mathcal{I}_1(a) := 0, \mathcal{I}_1(b) := 1$.
- $\mathcal{I}_2 : \mathcal{I}_2(D) := \mathbb{Z}, \mathcal{I}_2(p) := <, \mathcal{I}_2(\odot) := -, \mathcal{I}_2(a) := 3, \mathcal{I}_2(b) := 5$.
- $\mathcal{I}_3 : \mathcal{I}_3(D) :=$ strings, $\mathcal{I}_3(p) :=$ substring, $\mathcal{I}_3(\odot) :=$ string concatenation, $\mathcal{I}_3(a) :=$ "a special example ofsome string", $\mathcal{I}_3(b) :=$ "some string".

Definition 81 (Variable assignment)**.** Let $\mathcal{I}$ be an interpretation. An *asignment for variables under interpretation* $\mathcal{I}$

$$\sigma_{\mathcal{I}} : \mathcal{V} \to D$$

which assigns to every variable an element from the domain D.

The assignment $\sigma_{\mathcal{I}}[x \leftarrow d]$ is the same as $\sigma_{\mathcal{I}}$ except that x is mapped to the element d, $d \in D$.

Definition 82 (Value of terms under assignment)**.** Let $\mathcal{I}$ be an interpretation, $\sigma_{\mathcal{I}}$ an assignment. The *value of a term under* $\upsilon_{\sigma_{\mathcal{I}}}$ is defined as follows:

- If $v \in \mathcal{V}$, $\upsilon_{\sigma_{\mathcal{I}}}(v) = \sigma_{\mathcal{I}}(v)$.
- If $c \in \mathcal{C}$, $\upsilon_{\sigma_{\mathcal{I}}}(c) = \mathcal{I}(c)$.
- If $t_1, \ldots t_n$ are terms, $f \in \mathcal{F}$, $\upsilon_{\sigma_{\mathcal{I}}}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\upsilon_{\sigma_{\mathcal{I}}}(t_1), \ldots, \upsilon_{\sigma_{\mathcal{I}}}(t_n))$.

Definition 83 (Domain of truth values)**.** We choose 2 distinct values $\mathbb{T}$, $\mathbb{F}$. The set of truth values is the set $\{\mathbb{T}, \mathbb{F}\}$. The set of truth values is (totally) ordered, $\mathbb{F} < \mathbb{T}$.

Definition 84 (Value of formulae under assignment)**.** Let $\mathcal{I}$ be an interpretation, $\sigma_{\mathcal{I}}$ an assignment. The *value of a formula under* $\upsilon_{\sigma_{\mathcal{I}}}$ is defined as follows:

- If $p \in \mathcal{P}$ and $t_1, \ldots, t_n$ are terms, then $\upsilon_{\sigma_{\mathcal{I}}}(p(t_1, \ldots, t_n)) = \mathbb{T}$ iff

$$\mathcal{I}(p)(\upsilon_{\sigma_{\mathcal{I}}}(t_1), \ldots, \upsilon_{\sigma_{\mathcal{I}}}(t_n)).$$

- Let A, B be formulae. Then:
  - $\upsilon_{\sigma_{\mathcal{I}}}(\neg A) = \mathbb{T}$ iff $\upsilon_{\sigma_{\mathcal{I}}}(A) = \mathbb{F}$.
  - $\upsilon_{\sigma_{\mathcal{I}}}(A \wedge B) = \mathbb{T}$ iff $\upsilon_{\sigma_{\mathcal{I}}}(A) = \mathbb{T}$ and $\upsilon_{\sigma_{\mathcal{I}}}(B) = \mathbb{T}$.
  - $\upsilon_{\sigma_{\mathcal{I}}}(A \vee B) = \mathbb{T}$ iff $\upsilon_{\sigma_{\mathcal{I}}}(A) = \mathbb{T}$ or $\upsilon_{\sigma_{\mathcal{I}}}(B) = \mathbb{T}$.
  - $\upsilon_{\sigma_{\mathcal{I}}}(A \Rightarrow B) = \mathbb{F}$ iff $\upsilon_{\sigma_{\mathcal{I}}}(A) = \mathbb{T}$ and $\upsilon_{\sigma_{\mathcal{I}}}(B) = \mathbb{F}$.

- $\upsilon_{\sigma_\mathcal{I}}(A \Leftrightarrow B) = \mathbb{T}$ iff $\upsilon_{\sigma_\mathcal{I}}(A) = \upsilon_{\sigma_\mathcal{I}}(B)$.

- Let $x \in \mathcal{V}$, A be a formula:
  - $\upsilon_{\sigma_\mathcal{I}}(\forall x A) = \mathbb{T}$ iff for all $d \in D$, $\upsilon_{\sigma_\mathcal{I}[x \leftarrow d]}(A) = \mathbb{T}$.
  - $\upsilon_{\sigma_\mathcal{I}}(\exists x A) = \mathbb{T}$ iff for some $d \in D$, $\upsilon_{\sigma_\mathcal{I}[x \leftarrow d]}(A) = \mathbb{T}$.

Theorem 85. *Let A be a closed formula. Then $\upsilon_{\sigma_\mathcal{I}}(A)$ does not depend on $\sigma_\mathcal{I}$.*

Proof idea. If all variables are bound, then the assignment of a domain value to a variable does not change the value of the formula. □

> By this result, we can now talk about the value of a closed formula under interpretation $\upsilon_\mathcal{I}$, without being concerned with the variable assignments.

Example 86 (Values of formulae under variable assignment and interpretation). Consider the interpretation

$$\mathcal{I}_2 : \mathcal{I}_2(D) := \mathbb{Z}, \mathcal{I}_2(p) := <, \mathcal{I}_2(\odot) := -, \mathcal{I}_2(a) := 3, \mathcal{I}_2(b) := 5$$

from Example 80. Also consider the formulae:

- $A : p(a, x \odot b)$,
- $B : \forall x p(a, x \odot b)$,
- $C : \forall x p(y, x \odot z)$,

and the variable assignment for the interpretation $\mathcal{I}_2$:

- $\sigma_{\mathcal{I}_2} = \{x \leftarrow 0, y \leftarrow -2, z \leftarrow -9\}$.

Then:

- $\upsilon_{\sigma_{\mathcal{I}_2}}(A)$ is evaluated according to
  $\underbrace{\mathcal{I}_2(p)}_{\text{the interpretation of } p}(\underbrace{\mathcal{I}_2(a)}_{\text{the intepretation of } a}, \underbrace{\sigma_{\mathcal{I}_2}(x)}_{\text{the assignment of } x}\underbrace{\mathcal{I}_2(\odot)}_{\text{the interpretation of } \odot}\underbrace{\mathcal{I}_2(b)}_{\text{the intepretation of } b})$
  i.e. $(< (3, 0 - 5))$,
  i.e. $(3 < (0 - 5))$,
  i.e. $(3 < -5)$,
  which is $\mathbb{F}$ on the integers $\mathbb{Z}$,
  i.e. $\upsilon_{\sigma_{\mathcal{I}_2}}(A) = \mathbb{F}$.

- $\upsilon_{\sigma_{\mathcal{I}_2}}(B) = \mathbb{T}$ iff
  for all assignments of elements of the domain$\mathbb{Z}$ to x, $(3 < x - 5)$,
  but the formula considered above, A, is a counterexample to this
  i.e. $\upsilon_{\sigma_{\mathcal{I}_2}}(B) = \mathbb{F}$.

- $\upsilon_{\sigma_{\mathcal{I}_2}}(C) = \mathbb{T}$ iff
  $(-2 < 0 - (-9))$,
  which is true in $\mathbb{Z}$, therefore
  $\upsilon_{\sigma_{\mathcal{I}_2}}(C) = \mathbb{T}$.

Try a similar exercise for $\mathcal{I}_1, \mathcal{I}_3$.

Summary: Semantics of Predicate logic

To summarize, the meaning of expressions in predicate logic is "computed" according to the *interpretations* (i.e. assigned meanings) of its subexpressions.

- the meaning of terms or formulae with free variables can be determined only after assigning values to free variables,
- the meaning of terms $f(t_1, \ldots, t_n)$ is given by the *interpretation* of f applied to the *interpretations* of $t_1, \ldots, t_n$,
- the meaning of formulae with no free variables is either "true" of "false",
- the meaning of $\neg A, A \vee B, A \wedge B, A \Rightarrow B, A \Leftrightarrow B$ is given by first interpreting A, B, then computing the meaning of the formulae according to the rules for $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$,
- $\neg A$ is "true" if A is "false", $A \vee B$ is "true" if at least one of A, B is "true", $A \wedge B$ is "true" if both A, B are "true", $A \Rightarrow B$ is "false" only when A is "true" and B is "false", $A \Leftrightarrow B$ is "true" when A, B have the same meaning.
- the meaning of $\forall x A$ is "true", if A is "true" for all possible value assignments of x, and "false" otherwise,
- the meaning of $\exists x A$ is "true" if A is "true" for some value assignment of x (i.e. there exists an assignment of x that makes A "true"), and "false" otherwise.

3.3.1 Substitution and Semantics

- Intuition about substitution: "$p_{\{v \leftarrow t\}}$ says the same thing about the individual denoted by t as p says about the individual denoted by v".

Example 87. (Example taken from [Buchberger, 1991])
The statement about natural numbers P : $\exists y(x = 2y)$ (under the usual interpretations of the function symbols) says that the individual denoted by x is even.
However, $P_{\{x \leftarrow y+1\}}$: $\exists y(y + 1 = 2y)$ says that $y = 1$ (why?) and is no longer saying that something is even.
This is because the variable y in the term substituted for x was free, but now becomes bound. *Substitutions should not be carried out in any conditions.*

Definition 88 (Substitutable terms)**.** The term t *is substitutable for* v *in a formula* A is defined as:

- t is substitutable for v in A where A is an atomic formula,
- t is substitutable for v in $\neg A$ ($A \square B$) iff t is substitutable for v in A (A and B), where $\square$ is one of $\wedge, \vee, \Rightarrow, \Leftrightarrow$,
- t is substitutable for v in QvA, where Q is one of $\forall, \exists$,
- t is substitutable for v in QwA iff not both v is free in A and w is free in t, and t is substitutable for v in A.

Example 89 (Substitutivity).
- Is $y + 1$ substitutable for x in $\exists y(x = 2y)$?

NO! x is free in the existential formula, the result of substitution would become bound.

- Is $y+1$ substitutable for y in $\exists y(x = 2y)$?
  YES! (why?)
- Is vw substitutable for x in $\exists y(x < vx \Rightarrow (\exists w(w < v)))$?
  YES! x is free in the formula after the quantifier, but y is not free in vw, and vw is substitutable for x in the formula after the quantifier.
- Is vw substitutable for v in $\exists y(x < vx \Rightarrow (\exists w(w < v)))$?
  NO! (why?)
- Is vw substitutable for w in $\exists y(x < vx \Rightarrow (\exists w(w < v)))$?
  YES! (why?)

3.4 Syntax (Revisited)

3.4.1 Syntactic Sugar

- Notation: for the rest of the lecture, instead of $\forall xA$ ($\exists xA$) we will write $\underset{x}{\forall}A$ ($\underset{x}{\exists}A$).
- The common (and often informal) use of predicate logic contains many syntactical "shortcuts".
- Their role is to make expressions shorter (*syntactic sugar*).
- Below we review some of the more common notation variants.
- *Negation of binary predicates:* $x \not\lhd y$ stands for $\neg(x \lhd y)$, where $\lhd$ is a binary predicate.
- *Associativity of propositional connectives:* $P \wedge Q \wedge R$ stands for $P \wedge (Q \wedge R)$, $P \vee Q \vee R$ stands for $P \vee (Q \vee R)$, $P \Rightarrow Q \Rightarrow R$ stands for $P \Rightarrow (Q \Rightarrow R)$.
- *If ... then ... { else ... }:* "If P then Q" stands for $P \Rightarrow Q$, "If P then Q, else R" stands for
$$(P \Rightarrow Q) \wedge (\neg P \Rightarrow R).$$
- *Conjunctions of formulae:* P, Q, R stands for $P \wedge Q \wedge R$.
- *Conjunctions of atomic formulae involving infix binary relations:*
  - $x < y < z < 1$ stand for $x < y \wedge y < z \wedge z < 1$,
  - $x, y, z < 1$ stands for $x < 1 \wedge y < 1 \wedge z < 1$.
- *Conjunctions of formulae involving quantifiers*
  - $\underset{x,y}{\forall} A$ ($\underset{x,y}{\exists} A$) stands for $\underset{x\,y}{\forall\forall} A$ ($\underset{x\,y}{\exists\exists} A$),
  - $\underset{P(x)}{\forall} A$ stands for $\underset{x}{\forall}(P(x) \Rightarrow A)$,
  - $\underset{P(x)}{\exists} A$ stands for $\underset{x}{\exists}(P(x) \wedge A)$.

The predicate logic language: usage

Using the term (first order) predicate language:

- *the language of (first order) predicate logic* = the language built from variables and arbitrary (all conceivable) functions and predicate symbols,
- *a (first order) predicate logic language* = a language characterized by a few function and predicate symbols,
  i.e. a language of (first order) predicate logic is characterized by its function and predicate symbols (non-logical symbols).

3.5 Back to Semantics: Validity, Satisfiability, Unsatisfiability

In general we are interested in the following questions (concerning truth of formulae):

- Is a formula F *valid*, i.e. does it evaluate to $\mathbb{T}$ under *all* variable assignments and *all* interpretations, in *all* domains (infinitely many)?
- Is a formula F *satisfiable*, i.e. does it evaluate to $\mathbb{T}$ under *some* variable assignments and *some* interpretations, in *some* domain (out of infinitely many)?
- Is a formula F *unsatisfiable*, i.e. does it evaluate to $\mathbb{F}$ under *some* variable assignments and *some* interpretations, in *some* domain (out of infinitely many)?
- Is a formula F *a logical consequence* of a set of formulae KB (denoted $KB \models F$), i.e. does F evaluate to $\mathbb{T}$ under *all* variable assignments and *all* interpretations, in *all* domains where the formulae KB evaluate to $\mathbb{T}$ (out of infinitely many)?
- The difficulty of finding the answer to such questions shows the limitation of the direct method (interpretation of first order logic).
- Can we do better?

3.6 Proving. Proof techniques. Definitions. Theories.

3.6.1 Reasoning in Predicate Logic

- The good news is that reasoning (as defined Chapter 1)
  ‘ is possible in predicate logic.
- In fact several *calculi* for reasoning in (first order) predicate logic were proposed.
- A *reasoning calculus* represents a way to organize reasoning:
  - specifies how a *proof* should be organized,
  - specifies the *inference rules* (i.e. rules for producing proof steps).
- Intuitively *proof* is a trace of reasoning.
- Notation: If KB is a set of closed formulae and G can be proved (derived) from KB by the application of inference rules from a given calculus, we denote this as $KB \vdash G$.
- For reasoning in predicate logic (natural deduction, i.e a calculus that models the human reasoning as done by e.g. mathematicians) a remarkable result was proved by Gödel

  Theorem 90 (Soundness and completeness of reasoning in first order predicate logic)**.**

  $$KB \models G \textit{ iff } KB \vdash G.$$

- We will present here **informally** a particular calculus for reasoning in first order logic.

3.6.2 Proof Situations

- *Proving* is stepwise arrangement of *“proof situations”*.
- A *proof situation* is characterized by the current *knowledge base* (i.e. sentences known to be true, which can be used), and the current *goal* (i.e. sentence which should be proved).
- A proof situation is *trivial* if the goal occurs in the knowledge base.
- *Proof steps* (proof techniques, proof rules, inference rules) describe how a proof situation is transformed in (one or more) “simpler” proof situations.
- New proof situations are “simpler” in the sense that either the goal has a simpler structure, or more sentences are added to the knowledge base (i.e. we know more).
- A *proof* describes the necessary inference steps that transform the initial proof situation into (one or more) trivial proof situations.
- (AND-OR) trees can be used to represent proofs.
- Only a few proof situations are possible.
- Each determined by the structure of its sentences.
- and proof steps can be chosen according to the “outermost construct” in the sencence considered.

- Below are described such typical situatons and the proof steps that can be applied.
- This should be considered as a guideline for human proving (but note that this can be formalized to "natural deduction", and even automated theorem proving).

3.6.3 Basic Approaches to Proving

For proving the sentence A (when not knowing whether A is true):

- Try to prove A. If successful, :) (be happy). Otherwise:
- Assume $\neg A$ and try to derive a contradiction. If successful, :) (be happy, A is proved). Otherwise:
- Try to prove $\neg A$. If successful, :) (be happy). Otherwise:
- Assume A and try to derive a contradiction. If successful, :) (be happy, $\neg A$ is proved). Otherwise:
- Start again with the attempt to prove A. (By this time you have much more insight).

Proof Rules

Notation: In the following, A, B, C are formulae, s, t are terms, P is a predicate, f is a function constant, x is a variable. $A[x]$ is a formula where x is free, $A[t]$ is $A_{x \leftarrow t}$ (when $A[x]$ is changed into $A[t]$), $A[C]$ is a formula where C is a subformula.

$\forall x A[x]$

- Prove $\forall x A[x]$:
  - For proving $\forall x A[x]$,
  - show $A[x_0]$, where x_0 is a new constant (which did not occur so far).
  - Announcement in proofs:
    "Let x_0 be arbitrary but fixed (constant). We show $A[x_0]$."
- Use $\forall x A[x]$:
  - If $\forall x A[x]$ is known,
  - then one may conclude $A[t]$ where t is an arbitrary term.
  - Announcement in proofs: "Since we know that $\forall x A[x]$, we know that, in particular, $A[t]$".
  - Note that the choice of t is probably going to be suggested by the goal statement.

$\exists x A[x]$

- Prove $\exists x A[x]$:
  - For proving $\exists x A[x]$,
  - try to find a term t for which $A[t]$ can be shown,

  - Finding such a term t is many times nontrivial step, which needs creativity.
- Use $\exists x A[x]$:
  - If $\exists x A[x]$ is known, and B has to be proved,
  - one may assume $A[x_0]$ where x_0 is a new constant, and try to prove B.
  - Announcement in proofs: "Since we know that $\exists x A[x]$, let x_0 be such that $A[x_0]$. We have to prove B".

$A \wedge B$

- Prove $A \wedge B$:
  - For proving $A \wedge B$
  - prove A and
  - prove B.
- Use $A \wedge B$:
  - If $A \wedge B$ is known, then
  - A is known and
  - B is known.

$A \vee B$

- Prove $A \vee B$:
  - For proving $A \vee B$
  - assume $\neg A$ and
  - prove B
  - (or assume $\neg B$ and
  - prove A).
- Use $A \vee B$:
  - If $A \vee B$ is known, and C has to be proved then
  - assume A and prove C and
  - assume B and prove C.
  - Announcement in proofs: "We prove by cases: Case A: We prove C. Case B: We prove C."

$A \Rightarrow B$

- Prove $A \Rightarrow B$:
  - For proving $A \Rightarrow B$
  - assume A and
  - prove B.
- Use $A \Rightarrow B$:

  - If B has to be proved
  - then look for $A \Rightarrow B$ (or something that "matches" it) and
  - prove A.
  - Announcement in proofs: "To prove B, since we know $A \Rightarrow B$, it suffices to prove A".

    Also ("modus ponens")...
  - If A and $A \Rightarrow B$ are known,
  - then B can be added to the knowledge.
  - Announcement in proofs: "From $A \Rightarrow B$ and A, by modus ponens, we get B."

$A \Leftrightarrow B$

- Prove $A \Leftrightarrow B$:
  - To prove $A \Leftrightarrow B$,
  - assume A and
  - prove B, then
  - assume B and
  - prove A.
- Use $A \Leftrightarrow B$:
  - If $C[A]$ has to be proved, and
  - $A \Leftrightarrow B$ is known, then
  - try to prove $C[B]$.

$\neg A$

- Prove $\neg A$:
  - To prove $\neg A$,
  - assume A,
  - and derive a *contradiction*, i.e.
  - prove $\neg C$,
  - where C is in the knowledge base.

$P(t_1, \ldots, t_n)$

- Prove $P(t_1, \ldots, t_n)$:
  - To prove $P(t_1, \ldots, t_n)$
  - look for an "explicit definition" $\underset{x_1,\ldots,x_n}{\forall} P(x_1, \ldots, x_n) \Leftrightarrow A[x_1, \ldots, x_n]$
  - and prove $A[t_1, \ldots, t_n]$.
- Use $P(t_1, \ldots, t_n)$:
  - If $P(t_1, \ldots, t_n)$ is known and the definition:
  - $\underset{x_1,\ldots,x_n}{\forall} P(x_1, \ldots, x_n) \Leftrightarrow A[x_1, \ldots, x_n]$ is in the knowledge base,
  - then $A[t_1, \ldots, t_n]$ can be added to the knowledge base.

$A[f(t_1, \ldots, t_n)]$

- Prove $A[f(t_1, \ldots, t_n)]$:
  - To prove $A[f(t_1, \ldots, t_n)]$
  (a) "explicit definition" case:
    * where the "explicit definition" of f:
    $$\underset{x_1,\ldots,x_n}{\forall} f(x_1, \ldots, x_n) = s[x_1, \ldots, x_n] \text{ is in the knowledge base,}$$
    * prove $A[s[t_1, \ldots, t_n]]$.
  (b) "implicit definition" case:
    * where the "implicit definition" of f:
    $$\underset{x_1,\ldots,x_n}{\forall} f(x_1, \ldots, x_n) = \text{such a } y \text{ that } B[x_1, \ldots, x_n, y]$$
    is in the knowledge base,
    * prove $\underset{y}{\forall} B[t_1, \ldots, t_n, y] \Rightarrow A[y]$.
- Use $A[f(t_1, \ldots, t_n)]$:
  - If $A[f(t_1, \ldots, t_n)]$ is known:
  (a) "explicit definition" case:
    * and the "explicit definition" of f:
    $$\underset{x_1,\ldots,x_n}{\forall} f(x_1, \ldots, x_n) = s[x_1, \ldots, x_n] \text{ is in the knowledge base,}$$
    * then $A[s[t_1, \ldots, t_n]]$ can be added to the knowledge base.
  (b) "implicit definition" case:
    * and the "implicit definition" of f:
    $$\underset{x_1,\ldots,x_n}{\forall} f(x_1, \ldots, x_n) = \text{such a } y \text{ that } B[x_1, \ldots, x_n, y]$$
    is in the knowledge base,
    * then $\underset{y}{\exists}(B[t_1, \ldots, t_n, y] \wedge A[y])$ can be added to the knowledge base.
    * see below, the explanation about "such a ... that ..."

3.6.4 Definitions in Predicate Logic

- *Definitions* allow the introduction of *new concepts* in terms of existing ones.
- They provide a facility for concise formalization of mathematics, *but they are not essential, i.e. they can be eliminated.*
- However, they allow formulae to be shorter,
  and help structuring the knowledge.
- Practical mathematics would hardly be conceivable without this facility.

Example 91 (Irreducible natural numbers).

$$\underset{\textit{is-nat}(i)}{\forall}(\underbrace{\textit{is-irreducible}(i)}_{\text{"definiendum"(lat. to be defined)}} \Leftrightarrow \underbrace{\underset{\textit{is-nat}(n)}{\forall}(n|i \Rightarrow (n = 1 \vee n = i))}_{\text{"definiens"(lat. the defining)}})$$

Types of definitions

- (Explicit) definitions of predicate symbols. See Example 91.
- Explicit definitions of function symbols.

  Example 92 (The determinant of a 2x2 matrix.). Let $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$. Then $|A| = a_{11}a_{22} - a_{12}a_{21}$.
- Implicit non–unique definitions of function symbols.

  Example 93 (Implicit non–unique definition of $\sqrt{}$). The squareroot function can be defined (e.g. in the complex numbers) by:

  $$\underset{x}{\forall} \sqrt{x} = \text{such a } y \text{ that } y^2 = x.$$

  Note that this (such a ... that ...) is syntactic sugar for: $\underset{x,y}{\forall} \left(\sqrt{x} = y \Rightarrow y^2 = x\right)$.
- Implicit unique definitions of function symbols.

  Example 94 (Implicit unique definition of $\sqrt{}$). The squareroot function can be defined (e.g. in the real numbers) by:

  $$\underset{x}{\forall} \left(\sqrt{x} = \text{ the } y \text{ such that } \left((x \geq 0 \Rightarrow (y \geq 0 \wedge y^2 = x)) \wedge (x < 0 \Rightarrow y = 0)\right) \right).$$

  Note that this (the ...such that ...) is syntactic sugar for:

  $$\underset{x,y}{\forall} \left(\sqrt{x} = y \Leftrightarrow \left((x \geq 0 \Rightarrow (y \geq 0 \wedge y^2 = x)) \wedge (x < 0 \Rightarrow y = 0)\right)\right).$$

Properties of definitions

- Definitions:
  - are axioms,
  - can always be "eliminated",
  - do not bring anything new to the power of the theory (conservative extensions),
  - do not introduce contradictions.
- Correct definitions (watchlist):
  - *no extra variables* in the definiens.

    Example 95 (Extra variables). Consider the "definition": *is-factor*$(f) \Leftrightarrow f|x$. This leads to a contradiction: *is-factor*(f), because $3|6$ and $\neg$*is-factor*(f) because $3 \nmid 5$.
  - definitions of terms, *uniqueness, pairing functions.*

Example 96 (Definition of terms (wrong)). . The "definition"

$$\textit{is-nice-sum}(x+y) \Leftrightarrow x = 2y$$

introduces a contradiction: $6+3$ is a nice sum, because $6 = 2*3$, but $5+4$ is not a nice sum $5 \neq 2*4$. So 9 is both a nice sum and not a nice sum.

Example 97 (Definition of terms (with unique pairing function)). The definition $\textit{pair-of-primes}((x,y)) \Leftrightarrow (\textit{is-prime}(x) \wedge \textit{is-prime}(y))$ is correct, because the pairing function (x,y) which yields the pair of x, y has the uniqueness property:

$$(x,y) = (x_1,y_1) \Rightarrow (x = x_1 \wedge y = y_1).$$

3.7 Theories

Structure of a Theory

- A (mathematical) theory $\mathcal{T}$ is described by its:
  - *language (symbols)*: $\mathcal{L}_{\mathcal{T}} = \langle \mathcal{F}_{\mathcal{T}}, \mathcal{P}_{\mathcal{T}}, \mathcal{C}_{\mathcal{T}} \rangle$,
    i.e. its function, predicate and constant symbols respectively. Note that, for any theory, $id, =$ (the identity function and the equality) can always be included in the language;
  - *knowledge base* $\mathcal{KB}_{\mathcal{T}}$, which contains facts (formulae over the language), i.e. axioms and propositions (theorems, lemmata, corollaries, etc.).
  - *inference rules*, $\mathcal{IR}_{\mathcal{T}}$. The rules for predicate logic, as well as rules for equality proving (rewriting) may be included in any theory. Additional rules may be included (depending of the nature of objects in the theory), which make the theory *more powerful*, in the sense that more facts may be proved using these additional rules. These cannot be applied outside of the theory.

Example 98 (The theory of Strict partial orderings $\mathcal{PO}$.).

- The *symbols in the language*:
  - function symbols, $\mathcal{F}_{\mathcal{PO}} = \{\}$ (no function symbols),
  - predicate symbols,$\mathcal{P}_{\mathcal{PO}} = \{p\}$, where p is a binary symbol,
  - constants, $\mathcal{C}_{\mathcal{PO}} = \{\}$ (no constants either).
- the *knowledge base* $\mathcal{KB}_{PO}$:
  - Axioms describing strict partial orderings:

$$\begin{cases} \underset{x,y,z}{\forall}(p(x,y) \wedge p(y,z) \Rightarrow p(x,z)), & (transitivity) \\ \underset{x}{\forall} \neg p(x,x). & (irreflexivity) \end{cases}$$

- The *inference mechanism* $\mathcal{IR}_{PO}$ consists of the inference rules of predicate logic.

Development of a Theory

- Theories are not static, but are developed:
- from *initial descriptions* (such as in the previous example), where the knowledge base contains *axioms*,
- by adding new components:
  - *new formulae* to the knowledge base, i.e. consequences of the existing formulae, proved using the mechanisms available,
  - *new symbols* with their corresponding definitions (defining axioms, which are added to the knowledge base),
  - *new inference rules*, *a very subtle point - lifting knowledge to the level of inference*, as these have to be proved correct
- Questions that arise in the development of theories include:
  - how much of the theory is relevant? (not always all knowledge is needed, or even useful)
  - ...

3.8 Equality

- The binary predicate = (equality) plays a very important role in predicate logic: many theories contain equality.
- The language symbols (vocabulary) of a *theory of equality* (or containing equality) consists of = and an unspecified number of function, predicate and constant symbols.
- Any theory containing equality includes the axioms for equality:

$$\begin{array}{rl} \underset{x,y,z}{\forall} (x = y \wedge y = z \Rightarrow x = z)\,, & (transitivity) \\ \underset{x,y}{\forall} (x = y \Rightarrow y = x)\,. & (symmetry) \\ \underset{x}{\forall}(x = x) & (reflexivity) \end{array}$$

  Now let f be any function symbol of arity k, p be any predicate symbol of arity l:

$$\underset{\substack{x, y, z_1, \ldots, z_{i-1}, \\ z_{i+1}, \ldots, z_k}}{\forall} \left(\begin{array}{l} (x = y) \Rightarrow \\ \left(\begin{array}{l} f(z_1, \ldots, z_{i-1}, x, z_{i+1}, \ldots, z_k) = \\ f(z_1, \ldots, z_{i-1}, y, z_{i+1}, \ldots, z_k) \end{array} \right) \end{array} \right)$$

  (*functional substitutivity*)

  i.e. functions applied to arguments that are equal (x and y) yield equal results.
- (*functional substitutivity*) is in fact an *axiom scheme*, that can be applied to **any** functions of **any** arity.
- The corresponding proof technique is *equality rewriting*, i.e. chains of equalities from a complex (in some measure) to a simpler (simplest) term.
- E.g. two terms are equal if they can be rewriten to the same simpler (simplest) term.

$$\underset{\substack{x, y, z_1, \ldots, z_{i-1}, \\ z_{i+1}, \ldots, z_l}}{\forall} \left(\begin{array}{l} (x = y) \Rightarrow \\ \left(\begin{array}{l} p(z_1, \ldots, z_{i-1}, x, z_{i+1}, \ldots, z_l) \Leftrightarrow \\ p(z_1, \ldots, z_{i-1}, y, z_{i+1}, \ldots, z_k) \end{array} \right) \end{array} \right)$$

  (*predicate substitutivity*)

  i.e. predicates applied to arguments that are equal (x and y) are equivalent.
- (*predicate substitutivity*) is in fact an *axiom scheme*, that can be applied to **any** predicates of **any** arity.
- The corresponding proof technique is *equivalence rewriting*, i.e. chains of equivalences.
- E.g. two formulae are equivalent if there is a chain of equivalences between them.

3.9 Induction

- Induction is an inference rule used to prove (universal) properties of objects in domains that are *manageable*,
- i.e. every object in the domain can be "constructed" in a finitary way from "basic" objects, "smaller" objects,
- e.g. apply the successor function to a natural number to get the next one, and this is the only way to construct natural numbers (they are all successors of 0),
- all concepts in such (i.e. inductive) theories will be defined in a way which reflects the structure of the objects (inductive definitions, "recursive"),
- universal properties of inductive objects can be proved by corresponding induction inference rules, which again reflect the structure of the objects in the domain,
- induction is in general essential to prove in inductive theories, i.e. induction is stronger than predicate logic,
- examples of inductive domains: natural numbers, strings, lists, trees, sets, bags, formulae, proofs.

The theory of natural numbers

Example 99 (The theory of natural numbers $\mathbb{N}$).

- Natural numbers start with 0, and any other natural number is obtained from 0 by successive applications of the successor function $^+$
- the *symbols in the language*:
  - function symbols, $\mathcal{F}_{\mathbb{N}} = \{^+\}$ the unary successor function,
  - predicate symbols,$\mathcal{P}_{\mathbb{N}} = \{\mathit{is\text{-}nat}, =\}$, a unary predicate that detects natural numbers, and equality, respectively
  - constants, $\mathcal{C}_{\mathbb{N}} = \{0\}$.
- the *knowledge base* $\mathcal{KB}_{\mathbb{N}}$:
  - generation axioms:

$$\begin{array}{ll} \mathit{is\text{-}nat}(0) & \textit{(gen. zero)} \\ \underset{\mathit{is\text{-}nat}(x)}{\forall} \mathit{is\text{-}nat}(x^+) & \textit{(gen. succ.)} \end{array}$$

  - uniqueness axioms:

$$\begin{array}{ll} \underset{\mathit{is\text{-}nat}(x)}{\forall} x^+ \neq 0 & \textit{(zero)} \\ \underset{\mathit{is\text{-}nat}(x),\mathit{is\text{-}nat}(y)}{\forall} (x^+ = y^+) \Leftrightarrow (x = y) & \textit{(succ.)} \end{array}$$

  - induction axiom (scheme): for any formula $\mathfrak{F}$:

$$\left(\mathfrak{F}[0] \wedge \underset{\mathit{is\text{-}nat}(x)}{\forall} (\mathfrak{F}[x] \Rightarrow \mathfrak{F}[x^+])\right) \Rightarrow \underset{\mathit{is\text{-}nat}(x)}{\forall} \mathfrak{F}[x]$$

- the *inference mechanism* $\mathcal{IR}_{\mathbb{N}}$ consists of the inference rules of predicate logic and rewriting.

Induction as an inference rule

- A subtle point: the induction axiom scheme cannot be expressed in *first order predicate logic*, because first order logic only allows variables over objects and not over formulae (as is needed to express the induction scheme).
- One way to overcome this is to add, for each formula $\mathfrak{F}$ used an instantiation of the scheme. This may be cumbersome, and will result in very big knowledge bases, containing many similar formulae.
- However, we can add an inference rule (*the induction rule*) to $\mathcal{IR}_{\mathbb{N}}$:
  - To prove: $\underset{is\text{-}nat(x)}{\forall} \mathfrak{F}[x]$ (an universal property of natural numbers),
  - Prove $\mathfrak{F}[0]$ (the base case) and

    (Induction step):
  - Take x_0 arbitrary but fixed such that *is-nat*(x_0) and

    Assume $\mathfrak{F}[x_0]$ (the induction hypothesis) and

    Prove $\mathfrak{F}[x_0^+]$ (the induction conclusion).

Inductive definitions

- In inductive domains, definitions of new concepts will reflect the inductive structure of objects:
  - the notion has to be defined for (all) the simplest objects (the base case(s)).
  - the notion has to be defined for the complex objects, and it will be defined using the notion for the simpler objects, i.e. **recursion** will be used.

Example 100 (Semantics of expressions in predicate logic).

- Expressions of predicate logic are an inductive domain:
  - For terms:
    - variables and constants are (the simplest) terms,
    - function symbols applied to other (simpler) terms are terms (complex terms),
  - For formulae:
    - atomic formulae are the simplest formulae (but not the simplest expressions), predicates applied to terms,
    - compound formulae and quantified formulae are build from (simpler) formulae/terms are (complex) formulae.
- The notion of the semantics of terms and formulae is defined recursively:
  - For terms:
    - for variables it is given directly by the variable assignment, for constants, directly by the interpretation of the constant symbol,
    - for compound terms it is given **recursively**, in terms of the meaning of the smaller terms that make up the compound term.
  - For formulae:

  * for atomic formulae it is given directly by the interpretation of the function symbol applied to meaning of the terms,
  * for compound formulae it is given **recursively**, in terms of the meaning of the simpler formulae that make up the compound formula,
  * for quantified formulae it is given **recursively**, in terms of the meaning of the simpler formula that (but depending of the assignment of the quantified variable).

Example 101 (Recursive definitions in the theory of natural numbers).

- Consider the theory of natural numbers, as defined in Example 99.
- Now consider the following definition of a new concept (function symbol $+$):

$$\begin{aligned} \underset{is\text{-}nat(x)}{\forall} x + 0 &= x && \textit{(right zero)} \\ \underset{is\text{-}nat(x), is\text{-}nat(y)}{\forall} x + y^+ &= (x + y)^+ && \textit{(right succ.)} \end{aligned}$$

  Notice that for the simplest object (0) the value was given directly, and for a compound object (y^+), it was given in terms of the notion ($+$) for the simpler object (y).

- Here is a definition of a new concept (predicate symbol $\leq$):

$$\begin{aligned} \underset{is\text{-}nat(x)}{\forall} x \leq 0 &\Leftrightarrow x = 0 && \textit{(zero)} \\ \underset{is\text{-}nat(x), is\text{-}nat(y)}{\forall} x \leq y^+ &\Leftrightarrow (x = y^+ \vee x \leq y) && \textit{(succ.)} \end{aligned}$$

  Notice again the structure of the definition.

Example 102 (Proving by induction). The theory of natural numbers $\mathbb{N}$ is described by

- the *symbols in the language*:
  - function symbols, $\mathcal{F}_{\mathbb{N}} = \{^+\}$ the unary successor function,
  - predicate symbols,$\mathcal{P}_{\mathbb{N}} = \{is\text{-}nat, =\}$, a unary predicate that detects natural numbers, and equality, respectively
  - constants, $\mathcal{C}_{\mathbb{N}} = \{0\}$.
- the *knowledge base* $\mathcal{KB}_{\mathbb{N}}$:
  - generation axioms:

$$\begin{aligned} & is\text{-}nat(0) && \textit{(gen. zero)} \\ \underset{is\text{-}nat(x)}{\forall} & is\text{-}nat(x^+) && \textit{(gen. succ.)} \end{aligned}$$

  - uniqueness axioms:

$$\begin{aligned} \underset{is\text{-}nat(x)}{\forall} x^+ &\neq 0 && \textit{(zero)} \\ \underset{is\text{-}nat(x), is\text{-}nat(y)}{\forall} (x^+ = y^+) &\Leftrightarrow (x = y) && \textit{(succ.)} \end{aligned}$$

- the *inference mechanism*[1] $\mathcal{IR}_{\mathbb{N}}$ consists of the inference rules of predicate logic rewriting and the induction inference rule:

[1] By taking into account the remarks made in the lecture notes when we introduced the natural numbers, we move the induction axiom scheme to the level of inference

- To prove: $\underset{is\text{-}nat(x)}{\forall} \mathfrak{F}[x]$ (an universal property of natural numbers),
- Prove $\mathfrak{F}[0]$ (the base case) and
  (Induction step):
- Take x_0 arbitrary but fixed such that $is\text{-}nat(x_0)$ and
  Assume $\mathfrak{F}[x_0]$ (the induction hypothesis) and
  Prove $\mathfrak{F}[x_0^+]$ (the induction conclusion).

Prove

$$\underset{is\text{-}nat(x)}{\forall}(x \neq 0 \Rightarrow \underset{is\text{-}nat(y)}{\exists}(x = y^+)). \qquad (decomposition)$$

Now extend the theory with a new function symbol, $+$, defined as follows:

$$\underset{is\text{-}nat(x)}{\forall} x + 0 = x \qquad (right\ zero)$$
$$\underset{is\text{-}nat(x),is\text{-}nat(y)}{\forall} x + y^+ = (x + y)^+ \qquad (right\ succ.)$$

Prove:

$$\underset{is\text{-}nat(x),is\text{-}nat(y)}{\forall} is\text{-}nat(x + y) \qquad (closure)$$
$$\underset{is\text{-}nat(x)}{\forall} 0 + x = x \qquad (left\ zero)$$
$$\underset{is\text{-}nat(x),is\text{-}nat(y)}{\forall} x^+ + y = (x + y)^+ \qquad (left\ succ.)$$
$$\underset{is\text{-}nat(x),is\text{-}nat(y)}{\forall} x + y = y + x \qquad (commutativity)$$

Exercise

Consider the (inductive) theory of strings:

- the symbols of the language:
  - a binary function symbol $\bullet$, the *prefix* function,
  - a unary predicate symbol *is-char*, the *character* relation,
  - a unary predicate symbol *is-string*, the *string* relation,
  - a constant symbol, Λ, the *empty string*;
- the knowledge base:
  Generation axioms:

$$is\text{-}string(\Lambda), \qquad (generation\ empty)$$

$$\underset{is\text{-}char(u)}{\forall} is\text{-}string(u), \qquad (generation\ character)$$

$$\underset{is\text{-}char(u),is\text{-}string(x)}{\forall} is\text{-}string(u \bullet x), \qquad (generation\ prefix)$$

Uniqueness axioms:

$$\underset{is\text{-}char(u),is\text{-}string(x)}{\forall}(u \bullet x \neq \Lambda), \qquad (uniqueness\ empty)$$

$$\underset{\substack{is\text{-}char(u),\,is\text{-}char(v),\\ is\text{-}string(x),\,is\text{-}string(y)}}{\forall} (u \bullet x = v \bullet y \Rightarrow (u = v \wedge x = y)), \quad (uniqueness\ prefix)$$

Character equality axiom:

$$\underset{\textit{is-char}(u)}{\forall}(u \bullet \Lambda = u); \quad (\textit{character equality})$$

- the inference rules:
  - string induction: To prove a universally quantified formula over strings,

    $$\underset{\textit{is-string}(x)}{\forall}\mathcal{F}[x],$$

    * Base case: prove $\mathcal{F}[\Lambda]$ (i.e. the property holds for the empty string), then
    * Induction step,
      Take arbitrary but fixed u_0, x_0 such that *is-char*(u_0), *is-string*(x_0),
      Assume $\mathcal{F}[x_0]$ (the property holds for x_0) ,
      Show $\mathcal{F}[u_0 \bullet x_0]$.
  - predicate logic and equality.

Prove

$$\underset{\textit{is-string}(x)}{\forall}(x \neq \Lambda \Rightarrow \underset{\textit{is-char}(v),\textit{is-string}(y)}{\exists}(x = v \bullet y)). \quad (\textit{decomposition})$$

Try to write the recursive definition for the concatenation of two strings, i.e. a binary function that takes two strings (e.g. *abcdef* and *ghijkaa*) and returns the string built from putting together the argument strings (*abcdefghijkaa* in our case). Then prove the closure and associativity of concatenation in the theory of strings.

Bibliography

[Buchberger, 1991] Buchberger, B. (1991). Logic for computer science. Unpublished lecture notes, Copyright Bruno Buchberger.

[Gödel, 1930] Gödel, K. (1930). Die vollständigkeit der axiome des logischen funktionenkalküls. *Monatshefte für Mathematik*, 1(37):349–360.

[Gödel, 1931] Gödel, K. (1931). Über formal unentscheidbare sätze der principia mathematica und verwandter systeme, I. *Monatshefte für Mathematik und Physik*, (38):173–198.

[H.Gallier, 2003] H.Gallier, J. (2003). *Logic for Computer Science, Foundations of Automated Theorem Proving*. Copyright Jean H. Gallier.

[Kleinberg and Tardos, 2006] Kleinberg, J. and Tardos, E. (2006). *Algorithm Design*. Pearson International, Inc.

[Manna and Waldinger, 1985] Manna, Z. and Waldinger, R. (1985). *The Logical Basis for Computer Programming Volume I, Deductive Reasoning*. Addison-Wesley Publishing Company, Inc.

[Shannon, 1938] Shannon, C. E. (1938). A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57.

[Tanenbaum and Austin, 2013] Tanenbaum, A. and Austin, T. (2013). *Structured Computer Organization*. Pearson, 6th edition edition.