Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2021

Martin Steffen

Contents

2 Logics **1**
2.1 Introduction . . . 1
2.2 Propositional logic . . . 5
2.3 Signatures, terms, and substitutions . . . 12
2.3.1 Signatures . . . 13
2.3.2 Terms . . . 15
2.3.3 Substitutions, in particular term substitutions . . . 16
2.3.4 First-order signature (with relations) . . . 17
2.3.5 Further side issues and elaborations . . . 17
2.4 First-order logic . . . 19
2.4.1 Syntax . . . 19
2.4.2 The meaning of first-order logic formulas (semantics, interpretation, models . . .) . . . 20
2.4.3 Giving meaning to variables . . . 22
2.4.4 Proof theory . . . 25
2.5 Modal logics . . . 31
2.5.1 Introduction . . . 32
2.5.2 Syntax . . . 33
2.5.3 Semantics . . . 34
2.5.4 Proof theory and axiomatic systems . . . 42
2.5.5 Exercises . . . 44
2.6 Dynamic logics . . . 46
2.6.1 Introduction . . . 46
2.6.2 Multi-modal logic . . . 47
2.6.3 Dynamic logics . . . 48
2.6.4 Semantics of PDL . . . 52
2.6.5 Exercises . . . 56

Chapter 2 Logics

What is it about?

Learning Targets of this Chapter

The chapter gives some basic information about "standard" logics, namely propositional logics and (classical) first-order logics (and maybe more).

Contents

2.1 Introduction 1
2.2 Propositional logic 5
2.3 Signatures, terms, and substitutions 12
2.4 First-order logic 19
2.5 Modal logics 31
2.6 Dynamic logics 46

2.1 Introduction

Logics

What's logic?

As discussed in the introductory part, we are concerned with formal methods, verification and analysis of systems etc., and that is done relative to a *specification* of a system. The specification lays down (the) desired properties of a system and can be used to judge whether a system is correct or not. The requirements or properties can be given in many different forms, including informal ones. We are dealing with *formal* specifications. Formal for us means, it has not just a precise meaning, that meaning is also fixed in a mathematical form for instance, in the form of a "model"[1] We will mostly not deal with informal specifications nor with formal specifications that are unrelated to the *behavior* in a broad sense of a system.

For example, a specification like

the system should cost 100 000$ or less, incl. VAT

[1]The notion of model will be variously discussed later resp. given a more precise meaning in the lecture. Actually, it will be given a precise mathematical meaning in different technical ways, depending on which framework, logics, etc. we are dealing with; the rough idea remains the "same", though.

could be seens as being formal and precise. In practice, such a statement is probably not precise enough for a legally binding contract (what's the exchange rate, if it's for Norwegian usage? Which date is taken to fix the exchange rate, the date of signing the contract, the scheduled delivery date, or the actual delivery date? What's the "system" anyway, the installation? The binary? Training? etc.) All of that would be "formalized" in a legal contract readable not even for mathematicians, but only for lawyers, but that's not the kind of formalization we are dealing with.

For us, properties are expressed in "logics". That is a very broad term, as well, and we will encounter various different logics and "classes" of logics.

This course is not about fundamentals of logics or philosophical questions, like "is logic as discipline a subfield of math, or is it the other way around", like "math is about drawing conclusions about some abstract notions and proving things about those, and in order to draw conclusions in a rigourous manner, one should use logical systems (as opposed to hand-waving ...)". We are also mostly not much concerned with fundamental questions of logical *meta-theory*. If one has agreed on a logic (including notation and meaning), one can use that to fix some "theory" which is expressed *inside* the logic. For example, if one is interested in formally deriving things about natural numbers, one could first choose first-order logic as general framework, then select symbols proper for the task at hand (getting some grip on the natural numbers), and then try to axiomatize them and formally derive theorems inside the chosen logical system. As the name implies meta-theory is *not* about things like that, it's about what can be said *about* the chosen logic itself: Is my logic decidable? How efficient can it be used for making arguments? How does its expressivity compares to that of other logics? ... Such questions will pop up from time to time, but are not at the center of the course. For us, logic is more of a tool for validating programs, and for different kind of properties or systemd, we will see what kind of logics fits.

Still, we will introduce basic vocabulary and terminology needed when talking *about* a logic (on the meta-level, so to say). That will include notions like formulas, satisfaction, validity, correctness, completeness, consistency, substitution ..., or at least a subset of those notions.

When talking about "math" and "logics" and what there relationship is: some may have the impression that math as discipline is a formal enterprise and formal methods is kind of like an invasion of math into computer science or programming. It's probably fair to say, however, that for the working mathematician, math is *not* a formal discipline in the sense the formal methods people or computer scientists do their business. Sure, math is about drawing conclusions and doing proofs. But most mathematicians would balk at the question "what's the logical axioms you use in your arguments?" or "what exact syntax do you use?". That only bothers mathematicans (to some extent) who prove things *about* logical systems, i.e., who take logics as object of their study. But even those will probably not write their arguments *about* a formally defined logic *inside* a(nother?) logical system. That formal-method people are more obsessed with such nit-picking questions has perhaps two reasons. For one is that they want not just clear, elegant and convincing arguments, they want that the *computer* makes the argument or at least assist in making the argument. To have a computer program do that, one needs to be 100% explicit what the syntax of a formal system is and what it means, how to draw arguments or check satisfaction of a formula etc.

Another reason is that the objects of study for formal-method people are, mathematically seen, “dirty stuff”. One tries to argue for the corrrectness of a program, an algorithm, maybe even an implementation. That often means one does not deal with any elegant mathematical structure but some specific artifact. It’s not about “in principle, the idea of the algorithm is correct”; whether the code is correct or not not depends also on special corner cases, uncovered conditions, or other circumstances. There is no such argument like “the remaining cases work analogously. . . ”: A mathematician might get away with that, but a formalistic argument covering really all cases would not.

Additionally, when making proofs about software, it’s often not about “the remaining five analogous cases”. Especially, in concurrent program or algorithms, one has to cover a huge amount of possible *interleavings* (combinations of orderings of executions), and a incorrectness, like a race condition, may occur only in some very seldom specific interleavings. So it’s more like there are “5 fantastillion more cases”. . . Proving that a few exemplary interleavings are correct (or test a few) will simply not do the job. That’s indeed one problem with *testing* concurrent systems. Even if one tests a huge amount of interleavings (and coaching an implementation to do check particular interleavings as opposed to rely on chance is also non-trivial), this normally pales in comparison to the astronomical number of possible interleavings. For sequential systems or a sequential, deterministic procedures, the possible behaviors is also astronomical, because typically there are many possible inputs, conceptually infinitely many often. However, being deterministic, at least the same input should lead to the same reaction. That’s a big help, in particular when it comes to reproducability. Furthermore, there are techniques that allow to tailor the input in such a way that *corner cases* are taken, for instance, feeding input that for all branches both the positive as well as the negative branch is executed by at least one test. That’s known as a form of test *coverage*.

For concurrent systems, what happens in one particular run is *non-deterministic*. In particular there is no part of the program code responsible that this or that decision is take (as is the case for a conditional statement in the sequential case). The different interleavings are done by the scheduler or the timings of the parallel processes.

General aspects of logics

- truth vs. provability
  - when does a formula *hold*, is *true*, is *satisfied*
  - valid
  - satisfiable
- syntax vs. semantics/models
- model theory vs. proof theory

We will encounter different logics. They differ in their syntax and their semantics (i.e., the way particular formulas are given meaning), but they share some commonalities. Actually, the fact that one distinguishes between the *syntax* of a logics and a way to fix the meaning of formulas is common to all the encountered approaches. The term “formula” refers in in general to a syntactic logical expression (details depend on the particular logic, of course, and sometimes there are alternative or more finegrained terminology, like *proposition*, or *predicate* or *sentence* or *statements*, or even in related contexts names like *assertion* or *constraint*). For the time being, we just generically speak about "formulas" here and leave terminological differentiations for later. Anyway, when it comes to the semantics, i.e. the meaning, it's the question of whether it's true or not (at least in classical settings…). Alternative and equivalent formulations is whether it *holds* or not and whether its *satisfied* or not.

That's only a rough approximation, insofar that, given a formula, one seldomly can stipulate unconditionally that it holds or not. That, generally, has to do with the fact that formulas typically has fixed parts and "movable" parts, i.e., parts for which an "interpretation" has to be chosen before one can judge the truth-ness of the formula. What exactly is fixed and what is to be chosen depends on the logic, but also on the setting or approach.

To make it more concrete in two logics one may be familiar with (but the lecture will cover them to some extent). For the rather basic *boolean logic* (or propositional logic), one deals with formulas of the form $p_1 \wedge p_2$, where $\wedge$ is a logical connective and the p's here are atomic propositions (or propositional variables, propositional constants, or propositional symbols, depending on your preferred terminology). No matter how it's called, the $\wedge$ part is fixed (it's always "and"), the two p's is the "movable part" (it's for good reasons why they are sometimes called propositional *variables*…). Anyway, it should be clear that asking whether $p_1 \wedge p_2$ is true or holds cannot be asked *per se*, if one does not know about p_1 and p_2, the truth or falsehood is relative to the choice of truth or falsehood of the propositional variables: choosing both p_1 and p_2 as "true" makes $p_1 \wedge p_2$ true.

There are then different ways of notationally write that. Let's abbreviate the mapping $[p_1 \mapsto \top, p_2 \mapsto \top]$ as σ, then all of the formulations (and notations) are equivalent

- $\sigma \models \varphi$ (or $\models_\sigma \varphi$):
  - σ satisfies φ
  - σ models φ (σ is a model of φ)
- $[\![\varphi]\!]^\sigma = \top$:
  - with σ as propositional variable assignment, φ is true or φ holds
  - the semantics of φ under σ is $\top$ ("true")

Of course, there are formulas whose truth-ness does *not* depend on particular choices, being unconditionally true (or other unconditionally false). They deserve a particular name like (propositional) "tautology" (or "contradiction" in the negative case).

Another name for a generally true formula or a formula which is true under all circumstances is to say it's *valid*. For propositional logic, the two notions (valid formula and tautology) coincide.

If we got to more complex logics like first-order logics, things get more subtle (and the same for modal logics later). In those cases, there are more "ingredients" in the logic that are potentially "non-fixed", but "movable". For example, in first-order logic, one can distinguish two "movable parts". First-order logic is defined relative to a so-called signature (to distinguish them from other forms of signatures, it's sometimes called first-order signature). It's the "alphabet" one agrees upon to work with. It contains functional and relational symbols (with fixed arity or sorts). Those operators define the "domain(s) of interest" one intends to talk about and their syntactic operators. For example, one could fix a signature containing operators `zero`, `succ`, and `plus` on a single domain (a single-sorted setting) where the chosen names indicate that one plans to interpret the single domain as natural numbers. We use in the discussion here `typewriter` font to remind that the signature and their operators are intended as *syntax*, not as the semantical interpretation (presumably representing the known mathematical entities 0, the successor function, and +, i.e., addition). There are also syntactic operators which constitute the logic itself (like the binary operator $\wedge$, or maybe we should write `and`...), which are treated as really and absolutely fixed (once one has agreed on doing classical first-order logic or similar).

The meaning of the symbols of a chosen signature, however, are generally *not* a priori fixed, at least when doing "logic" and meta-arguments about logics. On the other hand, when doing program verification, typically one is not bothered about that, one assumes a fixed interpretation of a given signature. Anyway, the elements of the signature are not typically thought of as *variables*, but choosing a semantics for them is one of the non-fixed, variable parts when talking about the semantics of a first-order formula. That part, fixing the functional and relational symbols of a given signature is called often an *interpretation*. There is, however, a second level of "non-fixed" syntax in a first-order formula, on which the truthness of a formula depends: those are (free) *variables*. For instance, assuming that we have fixed the interpretation of `succ`, `zero`, `leq` (for less-or-equal) and so on, by the standard meaning implied by their name, the truth of the formula `leq(succ x, y)` depends on the choices for the free variables `x` and `y`.

To judge, whether a formula with free variables holds or not, one this needs to fix two parts, the interpretation of the symbols of the alphabet (often called the interpretation), as well as the choice of values for the free variables. Now that the situation is more involved, with two levels of choices, the terminolgy becomes also a bit non-uniform (depending on the text-book, one might encouter slightly contradicting use of words).

One common interpretation is to call the choice of symbols the *interpretation* or also *model*. To make a distinction one sometimes say, the *model* (let's call it M) is the mathematic structure to part ... [Something's missing]

2.2 Propositional logic

A very basic form of logic is known as *propositional* or also *boolean* logic (in honor of George Boole).[2] Other names are statement logic or sentential logic. It's also underlying binary hardware, binary meaning "two-valued". The two-valuedness is the core of *classical* logics in general, the assumption that there is some *truth* which is either the case or else not (true or else false, nothing in between or "tertium-non-datur"). This is embodied the classical propositional logic.

In the following, we introduce the three ingredients of a mathematical logics, its *syntax*, its *semantics* (or notion of models, its semantics, its interpretation) and its *proof theory*. We don't go too deep into any of those, especially not *proof theory*.

[2]Like later for first-order logic and other logics, there are variations of that, not only syntactical, some also essential. We are dealing with *classical* propositional logics without without being too dogmatic which selection of operators we use. One can also study intuitionistic versions. One of such logic is known as *minimal* intuitionistic logic, that has implication $\rightarrow$ as the only constructor.

Model theory is concerned with the question of when formulas are "true", what structure satisfies a formula (its model). Proof theory, on the other hand, is about when formulas are *provable* by a formal procedure or derivation system. Those questions are not independent. A provable formula should ideally be semantically true and conversely a formula (a question of *soundness*) and vice versa: all formulas which are actually "true" should ideally be provably true as well (a question of *completeness*). Notationally, one often uses the symbol $\vdash$ when referring to proof-theoretical notions and $\models$ for model-theoretical, mathematical ones. $\vdash \varphi$ thus would represent φ is derivable or provable, and $\models \varphi$ for the formula being "true" (or valid etc.) referring to its semantics.

Syntax: propositions as formulas of propositional logic

Logics express themselves syntactically by so-called *formulas*. Different logics used different formulas. Propositional formulas, the formulas of propositional logics, are commonly also called just *propositions*. One can stumble also over texts which call them *sentences* (hence the word sentential logic as alternative name for propositional logic). Later we we also look at formulas of first-order logics and of other logics. The syntax of propositions we use is given as follows:

$$\begin{array}{rcll}
\varphi & ::= & & \text{propositions} \\
& & p \mid \top \mid \bot & \text{atomic propositions} \\
& \mid & \varphi \land \varphi \mid \neg\varphi \mid \varphi \rightarrow \varphi \mid \ldots & \text{compound propositions}
\end{array} \tag{2.1}$$

One can distinguish between atomic propositions and compound ones. There are two "constant" atomic propositions, called $\top$ and $\bot$. Constant in the sense that their meaning is fixed, resp. will be fixed in the obvious manner, indicate by the names of the symbols, as soon as we come to the semantics of formulas. Not fixed is the interpretation of p, representing *propositional variables.*[3] For propositional variable, we use p_1, p', q ...as typical elements.

Compound propositions are constructed using logical operators or connectives. We assume that they are a familiar. They included binary connectives like conjunction and disjunction $\land$ and $\lor$, negation $\neg$ as unary one, implication $\rightarrow$ as another one.

One could include more, like "exclusive or" $\oplus$ or equivalence or bi-implication $\leftrightarrow$, etc, but we are here not overly obsessed here by fixing the exact selection of logical operators. The grammar from equation (2.1) contains "...". One may also find presentations leaving out syntax for operators that can be expressed by others, so that the syntax becomes *minimal*: no operator can be removed without crippling the expressiveness of the logic (thus, being no longer classical propositional logic). Out shown syntax is not minimal. For example $\land$ can be expressed using $\neg$ and $\land$, and actually also the atomic propositions $\top$ and $\bot$ are not strictly needed and could be expressed likewise by $\neg$ and $\land$, for instance.

[3]One can stumble also over texts, where the p's are called propositional constants, or also propositional symbols. That's just terminology and does not change a thing. Even if called propositional constants, they are less constant than $\top$ and $\bot$, insofar that the truth value of $\top$ and $\bot$ is absolutely fixed, whereas for the p's, the truth-status needs to chosen. To call the p's propositional constants can also be justified, it's not unreasonable, but as said, it's just terminology and let's not loose sleep over that.

Especially in HW, for logical circuits, one may base the hardware one just *one* logical operator, "NAND" ("NOR" works likewise). That has advantages for the manufacturing process, and NAND can express all other operators, including the propositional constants $\top$ and $\bot$. For the purpose, we consider that as minor details resp. we rely on that the reader knows such basic things and can handle it. As long as we have prositional constants and operators like the ones from equation (2.1) and their standard interpretation as semantics), it's propositional logic.

As common, at least in computer science, the syntax is given by a grammar. More precisely here, the context-free grammar from equation (2.1) uses BNF-notation. It's a common compact and precise format to describe (abstract) syntax, basically syntax trees. The word "abstract" refers to the fact that we are not really interested in details of actual concrete syntax for text files used in a computer. There, more details would have to be fixed to lay down a presise computer-parseable format. Also things like associativity of the operators and their relative precedences and other specifics would have to be clarified.

But that is "noise" for the purpose of a written text and human communication. A context-free grammar *is* precise, if understood as describing *trees*, and following standard convention we allow parentheses to disambiguate formulas if necessary or helpful. That allows to write $p_1 \land (p_2 \lor p_3)$, even if parentheses are not mentioned in the grammar. Also we sometimes rely on a common understanding of precedences, for instance writing $p_1 \land p_2 \lor p_3$ instead of $(p_1 \land p_2) \lor p_3$, relying on the convention that $\land$ binds stronger than $\lor$. We are not overly obsessed with syntactic details, we treat logic *formal* and *precise* but not *formalistic*. Tools like theorem provers or model-checkers would rely on more explicit conventions and *concrete* syntax.

Semantics: the meaning of propositions

So far we have fixed only the notation of formulas, their syntax. If one has been in contact with some form of formal logics at all, one will probably have an grip on what propositions are supposed to represent. Like that $\land$ represents "and" and $\lor$ represents "or". But it may be less clear, especially when studying such things for the very first time, though I assume that for participants of the course this is not the case. For example, in an English sentence "do you want tea *or* coffee?", the colloqual "or" is typically not meant as the disjunctive operator of propositional logics, written $\lor$. That sentence is probably meant as given a choice between tea or coffee but not both, and asking "do you want either tea or else coffee?" sounds borderline impolite and "do you want tea xor coffee?" incomprehensible...

At any rate, we are better off with fixing the meaning in some unambigous way. To ask what a proposition means, is to ask "is it true or else false"? So ultimately, a proposition is mapped to one of exactly two possible value, the two *truth values* or Boolean values. Let's use $\top$ for true and $\bot$ for false, and call the two-valued Boolean domain by the notation $\mathbb{B} = \{\top, \bot\}$. To rub it in: we use $\top$ and $\bot$ as propositional constants in the *syntax*, and the values $\top$ and $\bot$ as ingredient of the semantics or meaning, and unsurpisingly, two case of fixing the semantics of propositions is that $\top$ is interpreted as $\top$ and $\bot$ as $\bot$. Actually it will be an inductive definition over the syntax of propositions, and the two cases will be *base cases* in that inductive definition, dealing with two *atomic* propositions.

The definition for compound proposition will be done by, obviously, the *inductive cases* of the definition.

Besides $\top$ and $\bot$, there is one base case which we have not covered, namely how to treat propositional variables p. Since we assume that there is a reservoir of propositional variables P, one should more correctly say, there's one base case per variable.

Anyway, since propositions will generally contain propositional variables the truth status of the whole proposition depends on which truth value are assumed or chosen for the propositional variables.

Choosing those is a mapping or function from propositional variables to trues values. We can call such a function a *(propositional) variable assignment* and let's use the σ for those, i.e. a variable assignment is of the following type:

$$\sigma : P \to \mathbb{B} \ . \tag{2.2}$$

With that in place, we can finally define the semantics of a proposition, namely relative to a propositional assignment. One typical notation for the semantic function are the "semantic brackets" $[\![]\!]$; the notation is also used for semantic functions in other context. Anyway, given a variable assignment σ from $P \to \mathbb{B}$ and a proposition φ from Φ, $[\![\varphi]\!]^\sigma$ is a boolean value from $\mathbb{B}$. That makes here $[\![\_]\!]^-$ a function of the following type (assuming, for no particular reason, the variable assignment as the first argument):

$$[\![\_]\!]^- : (P \to \mathbb{B}) \to \Phi \to \mathbb{B} \ . \tag{2.3}$$

When the semantic functions gives back the truth value $\top$, i.e., if

$$[\![\varphi]\!]^\sigma = \top \ , \tag{2.4}$$

we say, proposition φ is true under the variable σ, or φ holds under said variable assignment. One hears also the formulation that the variable assignment σ *satisfies* φ. Notationaly, the latter formulation would often rather written as

$$\sigma \models \varphi \tag{2.5}$$

instead of the formulation from equation (2.4).

$\models$ in the latter equation is the the (semantic) *satisfaction relation*, in this case the satifaction relation for propositional logic. Other logics have other semantic functions resp. other satisfaction relation.

Both formulaic representations and wordings are, of course, completely equivalent, and which to use is a matter of preference. We will use both, presumably.

Equation (2.5) as well as (2.4) cover the "positive" case: the formula or prosition is true, it holds, the assignment satisfies the proposition etc. The negative case means $[\![\varphi]\!]^\sigma = \bot$, resp. $\sigma \not\models \varphi$.

That there is such a clear-cut split in satisfaction or else non-satisfaction, in a proposition being true or els not, in which case the proposition is false, that is characteristic for *classical* propositional logic (or more generally also for first-order logic or classical higher-order logics). We don't cover non-classical propositional or first-order logics. The most famous and important alternatives would be *intuitionistic* variants. One thing that changes for instance for propositional logics is that is no longer the case that a proposition is false if and only if it's negation is true and vice versa. Classically, a proposition (given an choice for the propositional variables) is true or else false, there is no other thing beside true or else false. In Latin, that principle is called "tertium non datur", there is no third thing besides true or false. For intuitionistic logics, that's dropped, resp. is not the case. Without going into details that complicates the way the semantics of propositions is given.

Indeed, for our classical propositional, we have not *actually* given the semantics. We have just discussed different symbols that are typically used for it ($[\![\_]\!]^{-}$ and $\models$) and we have given types, in equation (2.3). We leave the exact definition as an easy exercise to the reader. We have mentioned that it ultimately it's a induction (or "recursive") definition that needs to fix the base cases of the syntax from equation (2.1). The inductive cases have to cover all the non-atomic logical connectors, defining the truth value of a compound proposition in terms of the truth values of the sub-propositions. For example, $\varphi_1 \wedge \varphi_2$ is true of both φ_1 and φ_2 are true, and false otherwise. A definition like that would correspond to a straightforward recursive procedure.

A compact way to show all the neccessary cases for all connective is to arrange them in tabular form, a so-called *truth table*. We show the truth table for conjunction as one example in Table 2.1.

		$\wedge$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Table 2.1: Truth table for conjuction

Conjuction is a binary connective, i.e., it has two "inputs". The table contains this four lines, covering the for combinatins of $\top$ and $\bot$ for the pairs of input. The last columns shows the output of the conjunction, thus fixing its semantics. As said, the remaining constructors are left out here.

Proof theory

In the previous section we fixed the meaning of propositions by definining the semantic function resp. defining $\sigma \models \varphi$. We called σ a variable assignment, associating truth values to propositional variables. One could also call it a *model* and use for $\sigma \models \varphi$ the words "σ models φ" instead of "σ satisfies φ". We will later encounter the notion of "model" also for first-order logic (first-order model) and for other logics. There, not surprisingly, the notion of model becomes more complex and interesting. For propositional logic, it may sound a a bit too extravagant to speak of a model of a proposition. But its correct terminology.

We mention it, because its related to "model theory". That's dealing with questions concerning the models for a logic. What are the models? If a formula has an infinite model, is there also a finite one? And more questions like that. It's likewise not a coincidence that the word model is mentioned in *model checking*. Model checking is not about general questions like the ones mentioned, it's more concretely about "given a model on the one hand and a formula on the other, does the model satisfies the formula?". For propositional logic, to check

$$\sigma \models^{?} \varphi \tag{2.6}$$

is trivial and can be done efficiently; as indicated, the semantic function $[\![\_]\!]$ is a recursive procedure, so one just have to plug in the arguments, calculate $[\![\varphi]\!]^{\sigma}$ and look at the outcome. So model checking is a non-problem for propositional logic and actually, the more general question of what can be said about models *in general* ("model theory") is boring, as well. Model theory get's more hairy for first-order logic or logics more expressive than propositional logic, but we will not cover that. We will also don't do model checking for first-order logic, but later for modal logics.

In connection with propositional logic, another question is of relevance and is a challenge, that's the question if there *exists* a model or variable assignmnt that satisfies a proposition. One could formulaically represent that as

$$? \models \varphi \tag{2.7}$$

and that's a question of *satisfiability*. For propositional logic, it's the famous SAT problem and corresponding tools are sat-solvers or more generally constraint solvers. Pure sat-solving means constraint solving for Boolean constraints (and Boolean constraint is just another name for a proposition in propositional logic intended to be checked for satisfiability).

Back to the real issue in the section, which is neither model theory nor model checking, but proof theory. Proof theory is kind of like the opposite of model theory. Model theory is about what can be said about the semantics (the models), proof theory is about what can be proven in a logic in principle. That's a theoretical question, a more more practical one is how to prove things efficiently. So one may paraphrase that by saying that model theory is concerned with when something is "true" and proof theory is about when something "provable". These questions of course hang together; the whole purpose of doing a proof is to establish truth. Ideally, of course, all true things should be provably so, and likewise all false things should be demomstrably false as well (refutable). In such a situations truth and provability coincide. Howerer, that's rarely the case. That has to do with liminations of computability; performing a proof is ultimately a computation process and already for first-order logic, one runs into undecidability.

Proof theory (same for model theory) is concerned with *meta* theory, it's *about* a given logic and its proof procedures but it's not formulated *inside* the logic at hand. For instance, when doing a statement about what can be proven in propositional logic, that's not a propositional formula, it's perhaps an English sentence, mentioning a particular proof procedure, which likewise is not given as propositional formula. Propositional logic would anyway be too weak to capture any of that.

Let's get more concrete: how to do proofs for propositional logic? The question is impresise, it depends on what one intends to prove. But ultimately all plausible things one could be interested in are *decidable*.

As mentioned, the "model checking" question from equation (2.6) is efficiently decidable (and uninteresting). Finding a satisfying variable assignment (equation (2.7)) is likewise decidable. We simply have to choose $\top$ and $\bot$ for all propositional variabes inside the given proposition and check the result. If we find a satisfying σ, φ is satisfiably, and, after checking all combinations, we have found not, it's not satisfiable. It's like making a giant truth table. Since there are only finitely many relevant σ to check, since the proposition contains only finitle many variables, SAT is decidable. The problem with that *brute force* method is the combinatorial explosion of checking different combinations of truth values. Unfortunately, theoretically speaking, from the perspective of complexity theory, we have to live with that. There is not much one can do, Boolean satisfiability or SAT is known to be *NP-complete* (actually, it's the prototypical NP-complete problem. When claiming that not much can be done, that's from a theoretical point of view. Practically, many thing can and have been done to make SAT-solvers and other forms of constraint solvers to tackle larger and larger problems.

Another thing one could be interested in is not satisfiability, but *validity*: is a proposition true for *all* variable assignments. Such a *valid* propositional formula is also called a propositional *tautology*. One can also solve that by checking out all possible variable assignment σ for the involved propositional variables.

Just filling out a giant truth table in a brute force manner is a proof procedure, only not a very elegant one. So, doing proofs for boolean satisfiabilty or validity can be done smarter, looking for specific strategies, or finding "short-cuts". What I mean by short-cut is pretty common sense, like if on processes a conjunction $\varphi_1 \wedge \varphi_2$ and the evaluation strategy has determined that φ_1 is $\bot$, then there is no purpose to find out all different evaluations for φ_2.

Working hard on evaluation strategies has led to different techniques and proof methods (David-Putnam and variations, resolution and on and on) and while in general, SAT is still NP-hard, such strategies (and smart data representations) have lead to quite powerful SAT solvers or other boolean prover implementations.

To round off this section, let's have a look at *satisfiability* and *validity*. Those two notions in classical forms of logics, are two sides of a coin: a formula is valid iff its negation is unsatisfiable. Formulated contra-positively, in terms of non-validity and satisfiability, it means

$$\not\models \varphi \quad \text{iff} \quad \sigma \models \neg\varphi \quad \text{for some } \sigma$$

The model σ for the negation $\neg\varphi$, refuting validity of φ, is also called *counter-example*. Model-checking is not doing satisfiability or validity, but we will encounter also there the concept of counter-examples, which is analogous to the one discussed here. It's a valuable feature of a program verification method to be able to give back counter examples. The information that a program does not satisfy its specification and requirements is not very helpful. Such a failure needs to be accompagnied with information what and where the problem is, and that's called the *counter example*. It's a model or example that shows that the negation of the property of interest is satified. Since model checking is not validity or satisfiability checking, that idea needs some porting to work with, for instance, temporal logic model checking, but the concept is similar.

2.3 Signatures, terms, and substitutions

As far as logics are concerned, we dealt so far, in Section 2.2 with a most simple form of logic, classical propositional logic. It's a bare form of logics, insofar it does not speak *about* anything. It is a logic in most bare form, regulating truthness and falseness, without referring to anything particular which is being true or false. So, in that sense, it is a very sterile form of logic.

First-order logic, covered later in Section 2.4, allows to speak and reason about "things". In propositional logic, one could have a propositional variable *even*, but that's just a name, like p or q. Those propositional variables, which count among atomic propositions are either true or false in classical propositional logic, but there is no "evenness" about the variable or something that is being even or not. In first-order logic or other more expressive logic, one could formulate things like $even(x)$, where *even* now is called a *predicate*. First-order logic is also called more explicitly first-order predicate logic. The predicate *even* is still just a name and in that sense there is also no "evenness" about it as scuh, but now it speaks *about* something (and one can try to axiomatize things about that name so that it captures "evenness"). In $even(x)$, x is meant as variable, plausible representing natural numbers. And depending on which natural number x represents, the even-predicate is intended to be true or false. When one intends to speak about numbers, one also need syntax for that (not just variables). I.e., one would need syntax for natural number (like $0, 1, \ldots$), and operations on them (like addition and multiplication two be able to write expressions like $2 \times (x + 1)$. In combination with the even-predicate, one would like that $even(1 + 1)$ evaluates to $\top$, $even(2 \times 5 + 1)$ to $\bot$, and the truth status of $even(x + 1)$ dependent on the choice of x.

Without further arrangements, as mentioned, there is no evenness about the even-predicate $even(x + 1)$, it's a symbol, in the same way as the proposition called *even* would be in propositional logics. The "arrangement" could be *fixing* the interpretation of *even* and of $+$ etc. That would be a semantics, in a way a model-theoretic approach. Or one could try to specify rules one wishes to hold for a predicate called *even* in combination with other symbols for natural numbers, for instance stipulating as axiom $\forall x.even(2 \times x)$ or $\forall x.even(x + 1) \rightarrow odd(x)$ etc. That could provide an axiomatization. And then one could ask; does the axiomatization, the proof-theoretic approach, capture exacly the standard natural numbrs (the model-theoretic way). The answer would be *no*.

But this section is not about how to give meaning to the symbols (either by attaching a model like the natural numbers, nor by trying to axiomatize it). That will be dealt with in the subsequent Section 2.4 about first-order logic.

In this section we are concerned with the symbols themselves, the syntactic aspects of first-order logic, bare any meaning. The domain specific concept of syntactical material for first-order logic is called (first-order) signature. It consist of a choice of *functional* and *relational* symbols, each with a given *arity* (or more generally with given sorts). For instance, when working with natural numbers, the symbol $+$ represents a *binary* operation, i.e., is of arity 2, and a relation or predicate like $\leq$ is likewise of arity 2.

2.3.1 Signatures

The signature is domain-specific, since first-order logic has also logical syntax, including the operators from propositional logics like conjunction, negation, etc.

A *signature* is

Definition 2.3.1 (Signature (single-sorted))**.** A *signature* $\Sigma = (\Sigma_f, \Sigma_{rel}, ar)$ consists of two finite, disjoint sets of functional and relational symbols together with a function $ar : \Sigma_f + \Sigma_{rel} \to \mathbb{N}$, fixing the *arity* for each symbol.

A signature with functional symbols only is called **algebraic** signature.

Abstractly, one could write $\Sigma_f = \{f_1^{(2)}, f_2^{(0)}, g^{(1)}, \ldots\}$ and $\Sigma_{rel} = \{R_1^{(1)}, R_2^{(2)}, Q^{(2)}, \ldots\}$, indicating the arity as superscript.

In this lecture, we often leave the signature implicit or in English, and don't operate with Definition 2.3.1 in its full glory. For instance, when using natural numbers, we might use (and have used) symbols like $+$ and $\leq$, both of assumed arity 2, without writing $\leq^{(2)}$ or $+^{(2)}$ or explicating ar as function.

Functional symbols of arity 0 are also called *constant* symbols. For the natural numbers 0 is an example of a constant symbol.

Relational symbols of arity 0 are like the propositional symbols in propositional logic (we called them mostly propositional variables). Indeed, a degenerated signature with no functional symbols and only 0-arity relational symbols corresponds to propositional logic.

The concept of a signature from Definition 2.3.1 is a bit limited, at least in practice. An arity of a symbol is the number of arguments the symbol takes. That implies that all arguments are of the same type. A type in our context is commonly not called type, but sort. Definition 2.3.1 assumes that there is only one sort (left implicit), and that's why it's called a single-sorted signature. When working with such signatures, for instance in connection with first-order logics, that's too restrictive to be useful in many cases.

One wants the possibility to work and reason about different data structures at the same time. That then leads to *many-sorted* signatures. If multi-sortedness is what's needed in practice, why does one bother with single-sorted signatures (and single-sorted first-order logics) at all and why do many texts focus on the single-sorted case? Well, it's mostly a theoretician's thing. When studying, say, first-order logics, like studying its proof theory or model theory, basically all interesting, fundamental results work the same in both settings. The many-sorted case only leads to a slightly notational overhead ("given sorts $s_1, s_2, \ldots s_n$

with $n \geq 1$, blablabla"), which one is happy do do without when exploring properties of the logics, working on the logic's meta-theory, perhaps stating that the results *carry over* straightforwardly to the many-sorted case.

In practice, of course, specifying and verifying properties of a system or a program, one wants to be able to work with natural numbers and lists and whatever is relevant in the concrete setting.

So sorts are symbols representing different domain, like `nat` or `bool` or whatever we need. We don't use (for here) the notation $\mathbb{N}$ or $\mathbb{B}$, since for now we stress that sorts like `nat` and `bool` are *syntax*, symbols form the signature, whereas $\mathbb{B}$ and $\mathbb{N}$ are the semantics, i.e., the boolean values resp. the natural numbers. The intention is of course that `nat` is interpreted by the domain $\mathbb{N}$ etc., but associating meaning to syntax is the task of the semantics.

One could then work with integers and integer list and use a symbols like `cons` in the functional part of the signature with the type $\mathtt{int} \times \mathtt{List}_{\mathtt{int}} \to \mathtt{List}_{\mathtt{int}}$. We don't give a many-sorted version of the concept of signature from Definition 2.3.1. We hope it's clear enough how it works and invoke the excuse for not doing it, namely everything "just carries over".

In a many-sorted setting we then have some form of type system. Actually, already in the single-sorted case we have one. A term $\mathtt{cons}\ (0, 0)$ is equally ill-typed as is $+0$ in a single-sorted setting, where $+$ needs two arguments, not one. At any rate, even when working with a large number of different sorts, the type discipline is fairly trivial. Therefore we won't bother to give a formal definition of well-sorted terms. We simply assume that we only work with well-typed expressions. For a programming language, one does not waste time to think what an ill-typed program means, one cannot run it anyway, and we won't waste time here to consider formulas of ill-sorted terms.

Many-sorted signatures have many *sorts*, like `nat`, or $\mathtt{List}_{\mathtt{int}}$. Why are those things are not called types, but sorts? Well, here and there, one finds also the word type for those. And for all intent and purposes, they are types. Still, more common is the sort-terminology for algebraic and first-order signatures. Perhaps that has historical reasons. It has also to do with the fact that, as far as type systems are concerned, the discipline is very restricted. People dealing with type systems would laugh at it and feel ashamed (and feel better when pointing out that it's just a sort system, not a grown-up type system...). The restriction is that one has a number of sorts, which corresponds to atomic base types, like the mentioned `nat` and $\mathtt{List}_{\mathtt{int}}$ and the operators like $\mathtt{cons}\colon \mathtt{nat} \times \mathtt{List}_{\mathtt{int}} \to \mathtt{List}_{\mathtt{int}}$, where $\mathtt{nat} \times \mathtt{List}_{\mathtt{int}} \to \mathtt{List}_{\mathtt{int}}$ can reasonably be called the type of `cons`. But for function symbols of the signature, only types of the following form are allowed

$$s_1 \times \ldots s_2 \to s$$

If we call that a type, theb s_i and s are *not* also types, but something else (namely sorts). This restricting disallows for instance that `cons` is of type $\mathtt{int} \to (\mathtt{List}_{\mathtt{int}} \to \mathtt{List}_{\mathtt{int}})$. It some sense, that type is equivalent to $(\mathtt{int} \times \mathtt{List}_{\mathtt{int}}) \to \mathtt{List}_{\mathtt{int}}$, but it's simple not within the framework of algebraic or first-order signature. Likewise on can not have a symbol for a map-like function of type $((\mathtt{int} \to \mathtt{bool}) \times \mathtt{List}_{\mathtt{int}}) \to \mathtt{List}_{\mathtt{bool}})$. A map-function is a *higher-order* function; it takes another function as argument. That's not doable in algebraic signature, and there are good reasons for that. Algebraic signatures are not just play a role for first-order logics, but the field of (general) algebras uses them. And the core theory and central results of (general) algebra simply rests on that syntactic restriction. Without it's no longer (general) algebra.

For the lecture, we will look not too deep at first-order logic, but don't explore "algebra" (which can be seen as a quite restricted form of first-order logic with only a functional signature and considering *equations* only. So it's a form of equational logic and in that sense, there is one and only one relational symbol, namely "$=$", but since that's fixed, one does not bother to introduce a Σ_{rel}-signature just for that.

When pointing out that higher-order functions like map are not allowed in algebraic and first-order signatures, we should avoid a misconception that could suggest itself at this point. Namely that higher-order functions would be covered by higher-order logic, which goes beyond first-order logic. That's a misconception. Higher-or-not-order-ness of predicate logic refers to about what can be quantified over. For instance, second-order predicate logic is allowed to quantify over predicates, first-order logic cannot. Third order predicate logic can quanify over predicates over predicates or sets of setc. etc. That's not the same as the question what the predicates ultimately speak over. In logics with algebraic signatures of the form introduced here, predicates range over elements from the domains of the given sort, and they don't range over functions.

Of course there are logics that can deal with higher-order function and are higher-order wrt. to the quantification over predicates. Often that are logics in connection with typed λ-calculi and phrased often as (intuitinistic) dependent type theory. There are also quite a number of theorem provers based on dependent type theories, like Isabelle/Hol, Coq, and others. Anyway, higher-order aspects won't be covered in our lecture, mostl probably.

2.3.2 Terms

Closely related to (algebraic) signatures is the notion of *terms*. That's nothing else than expressions formed with the syntactic material from a given signature, i.e., constructed using the given function symbols. Additonally, one has variables available, i.e., a given countably infinite setn X (and we assume typical elements like $x, y', \ldots$ as variables). For terms, the relational part of a first-order signature is not used, only the functional part Σ_f (also called an algebraic signature).

Definition 2.3.2 (Terms (single-sorted))**.** Given an algebraic signature Σ and a (countably infinite) set of *variables* X, then the set of *terms* over Σ and X, written $T_\Sigma(X)$ is given by the following grammar.

$$\begin{array}{rcll} t & ::= & x & \text{variable} \\ & \mid & f(t_1, \ldots, t_n) & f \text{ of arity } n \end{array} \tag{2.8}$$

The set of *ground* terms over Σ is given as $T_\Sigma(\emptyset)$ (also written as T_Σ).

The terms from Definition 2.3.2 are *single-sorted* and based on the single-sorted version of signatures from Definition 2.3.1. The definition requires that terms must be "well-typed" or "well-sorted" in that a function symbol that expects a certain number of arguments (as fixed by the signature in the form of the symbol's arity) must be a applied on exactly that number of arguments. The number n can be 0, in which case the function symbol is also called a *constant* symbol. In the straightforward many-sorted generalizations, terms, by definition would likewise be required to respect the sorts. As mentioned earlier, the multi-sorted setting is not really different, it does not pose fundamentally more complex challenges (neither syntactically nor also what proof theory or models or other questions are concerned).

As a simple example: with the standard interpretation in mind, a symbol `zero` would be of arity 0, i.e., represents a constant, `succ` would be of arity 1 and `plus` of arity 2. For clarity we used here (at least for a short while) `typewriter` font to refer to the symbols of the signature, i.e., the syntax, to distinguish them from their semantic meaning. Often, as in textbooks, one might relax that, and just $+$ and 0 for the *symbols* as well.

In practical situations (i.e., tools), one could allow *overloading*, or other "type-related" complications (sub-sorts, for example) for the sake of convenience. Also, in concrete syntax supported by tools, there might be questions of *associativity* or *precedence* or whether one uses *infix* or *prefix* notations. For us, we are more interested in other questions, and allow ourselves notations like x `plus` y or $x + y$ instead and similar things, even if the grammar seems to indicate that it should be `plus x y`. Basically, we understand the grammars as *abstract syntax* (i.e., as describing trees) an assume that educated readers know what is meant if we use more conventional concrete notations.

2.3.3 Substitutions, in particular term substitutions

A central notion in connection with terms is the concept of *substitution*. Actually, it's not just important for terms, but plays a role in many situations which involve syntactic constructions containing variables. But let's focus for now on terms.

To substitute means to replace something. Substitutions are meant to *replace* variables occurring in a term by terms. At its core, a substitution is defined as mapping from variables to terms, expressing said replacment.

Definition 2.3.3 (Term substitution (single-sorted)). Given terms $T_\Sigma(X)$ over a signature Σ and using variables from X, a *substitution* θ is a mapping of type

$$X \to T_\Sigma(X) \ . \tag{2.9}$$

In abuse of notation we use substitutions also on terms, i.e. as mapping of type $T_\Sigma(X) \to T_\Sigma(X)$.

We write θt or just $\theta(t)$ for applying the substitution θ in term t. In the literature, the post-application notation like $t\theta$ is also very common. Note that Definition 2.4.1 does not spell out how to *lift* the substitution function on variables from Equation (2.9) to work on terms. In abuse of notation (as is common), given a mapping θ on variables, we use the same symbol also for terms. In programming-language terms we make use of *overloading*.

Substitution, especially in practice, are *finite* functions in that they replace only finitely many variables. Terms contain finitely many variables, if any. So applying a subtitution only affects the variable occuring in the terms under consideration, so from that point of view, there is no real use for substitutions affecting initiely variables. For notation for finite substitution, we use $[t/x]$ for replaceing x by t, or $[t_1, t_2/x_1, x_2]$ for replacing x_1 and x_2, etc. As an example, $[1 + z/x]((x + (y * x)))$ gives $(1 + z) + (y * (1 + z))$.

That's enough for the moment concerning substitutions, it should be sufficiently clear and/or sufficiently known. We have defined substitutions on terms. We will later use substitution also on first-order formulas (actually, the concept of substitution makes sense everywhere if one has "syntactic expression" with "variables"): formulas will contain, besides logical constructs and relational symbols also variables and terms. The substitution will work the same as here, with one technical thing to watch out for (which is not covered right now): Later, variables can occur *bound* by quantifiers. That will have two consequences: the substitution will apply only to not-bound occurrences of variables (also called *free* occurrences). Secondly, one has to be careful: a naive replacement could suffer from so-called *variable-capture*, which is to be avoided (but it's easy enough anyway).

2.3.4 First-order signature (with relations)

So far we have focused *algebraic* signatures, the part Σ_f of the signature from Definition 2.3.1 used for terms. In first-order logic, the signature, besides function symbols, contains also *relational* symbols Σ_{rel}. Those are intended to be interpreted "logically". For instance, in a single -sorted case, if one plans to deal with natural numbers, one needs relational symbols on natural numbers, like the binary relation `leq` (less-or-equal, representing $\leq$) or the unary relation `even`. One can call those relations also **predicates** and they form later then the *atomic* formulas of the first-order logic (also called (first-order) predicate logic). When unspecific and talking generally, we use letters like P, Q' etc. as typical elements of Σ_{rel}. standard binary symbol: $\doteq$ (equality)

2.3.5 Further side issues and elaborations

Multi-sorted case and a sort for booleans The above presentation is for the single-sorted case again. The multi-sorted one, as mentioned, does not make fundamental trouble.

In the hope of not being confusing, I would like to point out the following in that context. If we assumed a many-sorted case (maybe again for illustration dealing with natural numbers and a sort **`nat`**), one can of course add a second sort intended to represent the booleans, let's call it **`bool`**. Easy enough. Also one could then think of relations as boolean valued function. I.e., instead of thinking of `leq` as relation-symbol, one could attempt to think of it as a *function symbol* namely of sort **`nat`** $\times$ **`nat`** $\rightarrow$ **`bool`**. Nothing wrong with that, but one has to be careful not confuse oneself. In that case, `leq` is a function symbol, and `leq(5,7)` (or `5 leq 8`) is a term of type **`bool`**, presumably interpreted same as term `true`, but it's not a predicate as far as the logic is concerned. One has chosen to use the surrounding logic (FOL) to speak about a domain intended to represent booleans. One can also add operator like `and` and `or` on the so-defined booleans, but those are *internal* inside the formalization, they are *not* the logical operators $\wedge$ and $\vee$ that part part of the logic itself.

0-arity relation symbols In principle, in the same way that one can have 0-arity function symbols (which are understood as constants), one can have 0-arity relation symbols or predicates. When later, we attach meaning to the symbols, like attaching the meaning $\leq$ to `leq`, then there are basically only two possible interpretations for 0-arity relation symbols: either "to be the case" i.e., true or else not, i.e., false. And actually there's no need for 0-arity relations, one has fixed syntax for those to cases, namely "true" and "false" or similar which are reserved words for the two only such trivial "relations" and their interpretation is fixed as well (so there is no need to add more alternative such symbols in the signature).

Anyway, that discussion shows how one can think of propositional logic as a special case of first-order logic. However, in boolean logic we assume many propositional symbols, which then are treated as *propositional variables* (with values true an false). In first order logics, the relational symbols are not thought of as variables, but fixed by choosing an interpretation, and the variable part are the variables inside the term as members of the underlying domain (or domains in the multi-sorted case).

Equality symbol The equality symbol (we use $\doteq$) plays a special role (in general in math, in logics, and also here). One could say (and some do) that the equality symbol is one particular binary *symbol*. Being *intended* as equality, it may be captured by certain laws or axioms, for instance, along the following lines: similar like requiring `x leq x` and with the intention that `leq` represents $\leq$, this relation is *reflexive*, one could do the same thing for equality, stating among other things `x eq x` with `eq` intended to represent equality. Fair enough, but equality is *so central* that, no matter what one tries to capture by a theory, equality is at least *also part of the theory*: if one cannot even state that two things are equal (or not equal), one cannot express anything at all. Since one likes to have equality anyway (and since it's not even so easy/possible to axiomatise it in that it's really the identity and not just some equivalence), one simply says, a special binary symbol is "reserved" for equality and not only that: it's agreed upon that it's *interpreted* semantically as equality. In the same way that one always interprets the logical $\wedge$ on predicates as conjuction, one always interprets the $\doteq$ as equality.

As a side remark: the status of equality, identity, equivalence etc is challenging from the standpoint of *foundational* logic or math. For us, those questions are not really important. We typically are not even interested in alternative interpretations of other stuff like `plus`. When "working with" logics using them for specifications, as opposed to investigate meta-properties of a logic like its general expressivity, we just work in a framework where the symbol `plus` is interpreted as $+$, end of story. Logicians may ponder the question, whether first-order logic is expressive enough that one can write axioms in such a way that the only possible interpretation of the symbols correspond to the "real" natural numbers and plus thereby is really $+$. Can one get an axiomatization that characterizes the natural numbers as the *only* model (the answer is: *no*) but we don't care much about questions like that.

2.4 First-order logic

In this section, we briefly cover first-order logics. It's not a in-depth discussion, since the lecture is mostly concerned with other forms of logics, like temporal logic. Still, like propositional logics, predicate logic is kind of like general basic knowledge, so we at least cover a possible syntax an discuss how that logic is interpreted, i.e., speak shortly about semantics and a bit of proof theory as well.

As said, we are mostly interested temporal logics or other logic. But that's actually a bit orthogonal. Temporal logic will have *temporal* connectives, but also standard, non-temporal logical operators, like conjunction or implication. Taking propositional logic as core, adding temporal or modal operators, leads to *propositional* temporal logic. But one can add them also to first-order logic, and then one gets first-order temporal logics etc. In that sense, can't hurt to have a short look at or get a short reminder of first-order logics.

2.4.1 Syntax

Definition 2.4.1 (Formulas of first-order logics)**.** Given a signature Σ, the formulas of first-order logic are given by the following grammar.

$$\begin{array}{rcll} \varphi & ::= & P(t,\ldots,t) \mid \top \mid \bot & \text{atomic formulas} \\ & \mid & \varphi \land \varphi \mid \neg\varphi \mid \varphi \to \varphi \mid \ldots & \text{formulas} \\ & \mid & \forall x.\varphi \mid \exists x.\varphi & \end{array} \tag{2.10}$$

We use φ for formulas of first-order logic, the same symbol we used for propositions from equation (2.1); propositions are the formulas of propositional logics, and we also will use the same symbol for formulas of later logics. The grammar shows the syntax for first-order logic. As before for propitional logic, we silently assume proper priorities and associativities (for instance, $\neg$ binds by convention stronger than $\land$, which in turn binds stronger than $\lor$ etc.) In case of need or convenience, we use parentheses for disambiguation.

The grammar, choice of symbols, and presentation (even terminology) exists in variations, depending on the textbook. It's often convenient to work with a sorted or typed variant, using for example syntax like $\forall x{:}Nat.\varphi$, which does not change much from a logical point of view.

Minimal representation and syntactic variations The above presentation, as we did in the propositional case, is a bit generous wrt. the offered syntax. One can be more economic in that one restricts oneself to a *minimal* selection of constructs (there are different possible choices for that). For instance, in the presence of (classical) negation, one does not need both $\land$ and $\lor$ (and also $\to$ can be defined as syntactic sugar). Likewise, one would need only one of the two quantification operators, not both. Of course, in the presence of negation, $\top$ can be defined using $\bot$, and vice versa. In the case of the boolean constants $\top$ and $\bot$, one could even go a step further and define them as $P \lor \neg P$ and $P \land \neg P$ (but actually it seems less forced to have at least one as native construct). One could also explain $\top$ and $\bot$ as propositions or relations with arity 0 and a fixed interpretation. All of that representation can be found here and there, but they are inessential for the nature of first-order logic and as a master-level course we are not loosing sleep over representational questions like that. Of course, if one had to interact with a tool that supports, for instance first-order logics or a fragment or extension thereof, (like a theorem prover or constraint solver) or if one wanted to implement such a tool oneself, syntactial questions would of course matter and one would have to adhere to stricter standards of that particular tool.

2.4.2 The meaning of first-order logic formulas (semantics, interpretation, models . . .).

After fixing the syntax of the logic, we have to address what formulas means. It's analogous to what we did for propositional logic (and later for other logics). Conceptually, the way the semantics is defined is analogous than for propositional logics. The syntax is given by a grammar, and the semantics is a mapping of syntax trees to a semantical "domain". In the propositional setting, the mapping was given by assigning truth values to the propositional syntax, and that mapping was lifted to propositions.

For first-order logics, the situation gets a bit more involved. There are two syntactic levels. There is the level of functional and relational symbols, with which one can express terms over a given signature and relations between terms or predicates on terms. Besides that level, there's the logical level, which needs to be interpreted as well. Most of that part is the same as for the propositional level (for instance, the symbol $\land$ is conjuction, as before etc.) The quantifiers are new of course, compared to propositional logics.

In the following we assume a given signature Σ, and we focus on the single-sorted case, with the excuse that, as mentioned, it does not make a difference, in theory. A first-order structure, in the single-sorted case, is simple some set of elements together with functions and relations on it.

For instance, let's take the familiar natural numbers. They set of natural numbers $\mathbb{N}$. The set in itself does not qualify as structure, for that, one needs additional constants, functions, and predicates or relations on that set or domain of the structure. A typical selection could be the following:

$$(\mathbb{N}; 0, \lambda x.x + 1, +, \times, \leq, \geq) \tag{2.11}$$

Here, the domain $\mathbb{N}$ is equipped with 4 functions (one of which is 0, which is of zero arity and this called usually a constant rather than function) and two binary relations $\leq$ and $\geq$. Never mind the λ-notation, it's just meant as a function that takes one argument from the domain and returns its successor.

Structures like that can be used to give meaning to first-order signatures. That's done by associating to each *function symbol* of a given arity a mathematical *function* of the same arity, and analogously, for each *relational symbol* or predicate symbol of a given arity. a matheamtical relation of fitting arity. The association is a mapping for which we use I, the interpretation function or interpretation for short, and a model is the pair consisisting of a domain A together with the interpretation function (and implicitly the signature, assumed given).

Definition 2.4.2 (First-order Σ-structure)**.** Assume a first-order signature Σ. A first-order *structure* M for Σ is a tuple

$$M = (A, I) \tag{2.12}$$

where A is a set (the domain) and I (the interpretation), maps functional symbols function f such that $I(f) : A^n \to A$ for each $f \in \Sigma_f^{(n)}$ and relational symbols P such that $I(P) \subseteq A^n$ (for each $P \in \Sigma_{rel}^{(n)}$).

Instead of $I(f)$ and $I(P)$, we mostly write $[\![f]\!]^I$ and $[\![P]\!]^I$ or f^I and P^I. Another name for Σ-structure is Σ-model. As a side remark: The special case of first-order signatures containing no relational symbols, but only function symbols is also called *algebraic* signature (see Definition 2.3.1). In line with that, the corresponding structures (without relations or predicates) are called Σ-algebras.

As before, we introduce the concept for a single-sorted signature and consequently, the Σ-structure has one single domain. The many-sorted case is the obvious generalization. In particular, the interpretation function is required in that case not respect the sorts of the symbols (not just their arity).

Definition 2.4.2 makes as strict separation between syntax on the one hand, and the mathematical structure or model on the other. So, for the illustrative example of natural numbers from equation (2.11), which is the semantical level, one has also a syntactical layer, the signature. So, to make the split more visible, would could write the coresponding signature Σ with one sort say `Nat` and function symbols `zero`, `succ`, `plus` and `times` and relational symbols `leq` and `geq`. So, the `times` is meant as symbol, and $\times$ as the well-know mathematical function of multiplication. The interpretation function would fix $[\![\texttt{times}]\!]^I = \times$, or $[\![\texttt{leq}]\!]^I = \leq$ for one of the relations.

One could also alternatively fix $[\![\texttt{times}]\!]^{I'} = +$, or $[\![\texttt{leq}]\!]^{I'} = \geq$. Sortwise or typewise that would be ok, but of course would be an non-recommended interpretation of symbols like `times` and `leq`.

For most people including mathematicians (excepts perhaps logicians doing model theory or semanticists) all of that is a bit schizophrenic or over-the-top hair-splitting. Interpreting the symbol `times` by $\times$, ok, but isn't $\times$ also a symbol? Sure, in a way yes, still talking about models, interpetations, semantics, etc. there is this split, one symbol (say `times`) is is meant as syntax, and associated with that is a mathematical function (but one needs to say which function it is, or define it, and in this case one could use the symbol $+$ and appeals to the knowledge of the reader to have an understanding what $+$ does, or defining it perhaps semantically, maybe using induction, if one feels the need).

As said, for most it's a bit over the top, and would be content with just working with a structure as in equation (2.11), not bothering to make the split in the two levels explicit.

That may sound nitpicking, but probably it's due to the fact that when dealing with "foundational" questions like model theory, etc. one should be clear what a *model* actually is (at least at the beginning). But also practically, one should not forget that the illustration here, the natural numbers, may be deceivingly simple. If one deals with more mundane stuff, like capturing real world things as for instance is done in ontologies, there may be hundreds or thousands of symbols, predicates, functions etc. and one should be clear about what means what. Ontologies are related to "semantics techniques" that try to capture and describe things and then query about it (which basically means, asking questions and draw conclusions from the "data base" of collected knowledge) and the underlying language is often (some fragment of) first-order logic.

2.4.3 Giving meaning to variables

After having introduced the notion of Σ-structure, giving an interpretation for the syntactic material of a first-order signature, are we now ready to fix the semantics of first order formulas? Formulas contain of course also logic connectives like $\wedge$ or $\neg$, but that will work analogously to the propositional case. One piece missing is that we have to deal with *variables*. Formulas can contain occurrences of (free) variables, i.e., variables which are not in the scope of a quantifier. The logical status of such an open formula obviously may depend on which values are chosen for the free variables For example, assume an atomic formula like (x `minus one`) `geq zero` and assume the obvious interpretation of the involved symbols on the domain of natural numbers, i.e., the formula represents $(x - 1) \geq 0$. For values of x larger or equal 1, that represents a truth about natural numbers, but for a value of 0 for x, it's false. That should be clear enough, so let's nail it down.

Let's call a mapping that assigns values to variables plausibly a *variable assignment*.

Definition 2.4.3 (Variable assignment)**.** A *variable assignment* for variables from X and a domain A is a mapping σ of the following type:

$$\sigma : X \to A \tag{2.13}$$

In Section 2.2, we introduced already variable assignments, there for propositional logics, see equation (2.2). We also used the same symbol for it and conceptually, it's similar, anyway: giving values to variables, on propositional logics, propositional variables or symbols, here variables representing values from a semantic domain of values. Note that what what we called propositional variables p ... in propositional logic correspond in first-order logics to relational symbol or predicate symbols of arity 0. If one has such 0-arity symbols in the signature, they are not considered variables and they are given their meaning by the *interpretation*, either as $\top$ or $\bot$. That actually means there is no much need for such "constant" predicate symbols or proposition symbols, two are enugh, one interpreted as $\top$ and one as $\bot$ (and even those could be *defined*, for example the one representing false-hood as negation of the one representing truth-ness and truthness as $\varphi \vee \neg\varphi$, choosing some arbitrary formula φ. Our syntax from Definition 2.4.1 provided two keywords $\top$ and $\bot$ (which are like 0-arity predicates) whose interpretation will be fixed to the obvious truth values $\top$ and $\bot$ accordingly.

A variable assignment is sometimes also called *valuation* or also *state*. The latter namely is not so much used when dealing with (first-order) logics as such, but when using such a logic for program verification. There, some variables play the dual role. The program being verified containts typical program variables. Th logic *speaks* about the program, using perhaps first-order logic or a fragment thereof. For instance, first-order formulas could express at a point in a program expectations about the values of some program variables, like *asserting* that, at *that* point, a particular variable is non-negative. This would be an example of a *assertion*. Many languages, for instance Java, offer assertions of that form (but not quantifications, that would be too expressive for the intended purposes of run-time assertions). Properties of program variables are captured by *free* variables in formulas. In a logical setting, values of variables are fixed by variable assignments or valuation, when speaking about variables in a program, we think of that assignment as the current *state* of the program. In imperative programs, a stretch of code can of course change the values of variables by assignments, i.e., there will be state changes, so the assertion before a stretch of code will be different from the assertion afterwards. The assertion before is typically called *pre-condition* and the assertion afterwards the *post-condition* of that code piece (but of course the post-condition of a particular piece of code is the pre-condition for what follows, if anything).

Be it as it may, the discussion just explains that mappings of the form from equation (2.13) are also known as *state* in particular in the context of verification of imperative programs which deal with states and state changes. Logics like first-order logics in isolation are typically seen as *declarative*, not dealing with variable changes; thus the terminology talks about valuations rather than states (though it's the same thing). Later we will talk about modal and temporal logics, those are logics which (rather generally speaking) reason about changes, for instance changes over "time" during an execution of a system.

Now we have fixed the interpretation of function symbols and know how to assign values to variable. So everything is in place to give meaning to terms containing variables. Given an interpretation for, it's done by straightforwardly lifting σ to terms for which we write

$$[\![t]\!]^I_\sigma$$

Even if straighforward, for completeness sake, here is the definition by induction on the structure of terms:

Definition 2.4.4 (Interpretation of terms)**.** Given a signature Σ and a corresponding model $M = (A, I)$, the value of term t from $T_\Sigma(X)$, with variable assignment $\sigma : X \to A$ written $[\![t]\!]^I_\sigma$) is given inductively as follow:

$$\begin{array}{rcl} [\![x]\!]^I_\sigma & = & \sigma(x) \\ [\![f(t_1, \ldots, t_n)]\!]^I_\sigma & = & [\![f]\!]^I([\![t_1]\!]^I_\sigma, \ldots, [\![t_n]\!]^I_\sigma), \end{array} \tag{2.14}$$

Variables are given their meaning by σ, function symbols are given their meaning by I (in the equation we write $[\![f]\!]^I$) and the rest is straightforward induction.

Before we tackle also the semantics of formulas, we have get one aspect concerning variables out of the way. It has to do with the quantifiers in the formulas. The quantifiers $\forall$ and $\exists$ range over variables and *bind* them. For instance, (free) occurrences of x in φ or *bound* by the quantification in $\forall x.\varphi$, so no longer free. The quantification introduces a *scope* for the bound variable, the variable can be seen as "local" to that scope.

Note that it's possible that a variable is at the same time free in some part of a formula and bound in another part, like variable x in $(\exists x.x = y + 15) \wedge y = 2 \times z$. Thus one speaks not just about free or bound variables, but more precisely of variable *occurrences*:

x *occurs* bound in the left part of the conjunction and occurs free in the right-hand part. The other variables y and z occur free, only.

In the presence of variable binders, the notion of substitution needs some adaptation resp. some words of caution. Definition 2.4.1 defined substitution for terms over an algebraic signature. That implied there had been no quantifiers around, and everything was straightforward. Now, we want to make use of substitutions also for formulas. Substitutions are, as before, (see equation (2.9))... [careful with substitution, other binding, scoping mechanisms]

...

$$\varphi = \exists x.x + 1 \doteq y \qquad \theta = [x/y] \tag{2.15}$$

The satisaction relation

Without further ado, we can now fix the meaning of first-order formulas. As mentioned in connection with propositional logics, one can do that in the form of a semantic function $[\![\_]\!]$ (like we did for terms as part, which are part of formulas)) or with a satisfaction relation $\models$ (see equation (2.16). One finds both notations, it's a matter of taste, and we will use both.

The semantics combines the interpretation of functional symbols, variables and relational symbol with that of the logical connectives, the latter defined analogously to what's been done for propositional case (except for the quantifiers, which are new, of course).

Given a signature Σ as fixed, one says that variable assignment σ makes a formula φ true in a model M, or φ *holds* in a model M and under an assignment σ, or M and σ *satisfy* φ, or similar. Notationally written as

$$M, \sigma \models \varphi \quad \text{or} \quad \sigma \models_M \varphi\ . \tag{2.16}$$

Alternatively, as said, on can base the definition on a functional formulation, for instance writing $[\![\varphi]\!]^I_\sigma$ for the truth value of φ in a given model M and under a variable assignment σ.

Definition 2.4.5 (Satisfaction relation)**.**

$$\begin{array}{lllll}
M, \sigma \models \top & & & & \\
M, \sigma \not\models \bot & & & & \\
M, \sigma \models P(t_1, \ldots, t_n) & \text{iff} & P^I([\![t_1]\!]^I_\sigma, \ldots, [\![t_n]\!]^I_\sigma) & & \\
M, \sigma \models \neg\varphi & \textit{iff} & M, \sigma \not\models \varphi & & \\
M, \sigma \models \varphi_1 \wedge \varphi_2 & \textit{iff} & M, \sigma \models \varphi_1 & \text{and} & M, \sigma \models \varphi_2 \\
M, \sigma \models \varphi_1 \vee \varphi_2 & \textit{iff} & M, \sigma \models \varphi_1 & \text{or} & M, \sigma \models \varphi_2 \\
M, \sigma \models \varphi_1 \rightarrow \varphi_2 & \textit{iff} & M, \sigma \not\models \varphi_1 & \text{and} & M, \sigma \models \varphi_2 \\
M, \sigma \models \forall x.\varphi & \textit{iff} & M, \sigma' \models \varphi & \text{for all } x\text{-variants } \sigma' \text{ of } \sigma & \\
M, \sigma \models \exists x.\varphi & \textit{iff} & M, \sigma' \models \varphi & \text{for some } x\text{-variants } \sigma' \text{ of } \sigma &
\end{array} \tag{2.17}$$

2.4.4 Proof theory

We have fixed the semantics or model and interpretation of first-order logic formulas, we have defined validity and satisfiability. What is missing is how to *prove* that a formula or sentence is valid or satisfiable. Notions like truth and satifaction etc. are defined in reference to an external mathematical "reality", e.g., in reference to a Σ-structure. That's all very standard, though the sceptics may ask about what kind of reality are we talking about, like how real are, for instance, the natural numbers or other structures, are they somehow more fundamental than first-order or other logics that justifies that we use mathematical structures to give meaning to logical formulas? Shouldn't logics rather be the foundation of mathematics?

We don't loose sleep over such philosophical questions. But it's a concern, of course, having clarified validity and satisfaction etc., how to **establish it** as a fact or disprove it. So it's a question of mechanical procedure or algorithmic approach to **prove** or **disprove** valid formulas or derive logical consequences from a given set of sentences. That's what proof theory is concerned with.

Using the semantic definition of validity directly, one would have to check for *all* models that each of them make the formula true. There are infinitely many models and one cannot check them all. If one is after satifiabilty, not validity, one could try one model after the other to find one that satisfies the formula. That likewise would be hopelessly unpractical. There are not only infinitely many models, the models themselves may be infinite, so even after having decided on a model, checking if a formula holds in the model or not may not be possible.

All that attempts would not qualify as proof theory anyway. We are interested how to establish the "truth-ness" of a formula or refute it. Model-theory asks, is this formula "true" or valid, proof-theory asks is a formula provable. More generaly, it's about how to devise proof systems for that task, and asking about limits of what can be proven in a given logic. Implementations of such calculi are called *theorem provers*. That's because a formula derivable or inferrable in a proof systems are also called a *theorem* of the calculus. There are also proof systems trying to establish if a formula is satisfiability, corresponding implementation are rather called satisfiability checkers or constraint solvers.

Above we mentioned limits of what can be proven. It's a well-known fact, that validity of first-order logic is *undecidable*. Which means there cannot be an algorithm or a proof system that can be used to *decide* that. For first-order logic, the best we can hope for is a system that is able to derive as theorems all *valid* formulas, and non-valid formulas cannot be derived as theorem. Isn't that deciding validity? No, of course not, because when a formula is not valid, non-derivability can mean that inference process does not terminate, so one never knows. *Soundness* and *completeness* is as good as it gets for first-order logics, but the logic is undecidable. Actually, it does not take much to obtain logics where also completeness is out of the window, for instance second-order logics or doing first-order logics and fixing the model to be the natural numbers with the usual operations.

Since we are not too deeply interested in first-order logic in this lecture and also not so much in validity and theoremhood, but rather in model checking, we don't dig deeper here. We just say a few gegeral words about the shape of proof systems.

Proof systems

There are *many, many* flavors of proof systems for first-order logic. We don't explore them or their differences. We just mention some aspects largely common to such systems, also for different logics.

Let's focus on validity. There are also refutation systems, basically focusing on the dual question of (non-)satisfiabilty. They are also of practical relevance, but as said, let's focus just one flavor, validity.

The purpose of such a proof system is to allow to derive all valid formulas. Those are infinitely many, so one cannot just list them all, of course. So, it's a (specification of) a way to derive or infer them. Often, the proof system is given in a non-deterimistic way, leaving out which derivation steps are supposed to be explored first. Of course in an implementation, derivation strategies need to be fixed, and heuristisc of how concretely make use of the proof system in terms of strategy or scheduling is very important. For the specification of a proof system, it is often simpler to leave out, at least to some extent, such questions that have to do with efficience, and focus of whether the system is sound and complete (if possible).

The way that such inference systems are arranged, very generally, is that there is a pool of a priori given formulas together with a way to generate or derive new formulas from those and those already derivered previousy. The given formulas are often called **axioms** and the generation process is specified by derivation **rules**.

To use a standard notation, a rule looks as follows

$$\frac{\varphi_1 \quad \dots \quad \varphi_n}{\psi} \tag{2.18}$$

The formulas $\varphi_1, \dots, \varphi_n$ on top of the rule are called the **premises** and the one ψ below the **conclusion** of the rule. The rule is intended to express that if all of the premises have been established, either by virtue of being an axiom or having been derived earler in the process, the the conclusion ψ can be added as (another) theorem to the derived formulas.

If the axioms are semantically valid, and if the rules like the one shown allow only to generate valid formulas when using prior valid formulas in a derivation step, then obviously all derivable formulas are valid. In other words, one has a **sound** derivation system for the given logic. Typically, that's easy to achieve and easy to see. **Completeness** is quite a different story, that no valid formula is missed in the genration process.

The derivation of new formulas from old ones can be seen as a *sequence*, adding new formulas to older ones. Alternatively, in many presentations, it's rather a tree. Be that as it may, the derivation is also called a **proof** of the "last" formula at the end. Since the rule system does typically not specify which rules to apply when, proving a formula corresponds to **proof search**, and the choice of search strategy has a huge impact in the efficiency of a implementation of a proof system.

The way the story here is presented, deriving valid formulas is a process that generates valid formulas from axioms using rules. Also that is not the way that proving works. In practical verification tasks, one has most probably a property in mind one is instereted in to establish or check if it holds. In other words, simply generating valid formulas one after other so see if the one interesting one shows up in the process is not a useful strategy. So the derivation rules in a search are better seen *backwards*. A *conclusion* of the rule is the *goal* one tries to establish and the hypotheses of a rule are the *subgoals*.

Given a proof system one writes $\vdash \varphi$, of the formula φ can be *derived* in the system. One also says synonymously, φ is a *theorem* of the given derivation system. One writes $\nvdash \varphi$ if that's not the case. Those are proof theoretic statements about the formula and the proof system. There model theoretic counterparts are $\models \varphi$ and $\not\models \varphi$: the formula is *valid* resp. is not valid. A theory is a set of formulas, and one can also there distinguish between a proof theoretic theory: all the formulas such that $\vdash \varphi$ in a given derivation system. Or the semantic or model theoretic theory, all valid formulas $\models \varphi$.

Ideally, one has a sound and complete proof systems for the models one wants to cover. In this case, semantical concepts like validity or truth coincide with the proof theoretic ones like derivability or theoremhood. In notation

$$\models \varphi \quad \text{iff} \quad \vdash \varphi$$

Often, one is also interested not just in validity, but in "consequences". Like $\Gamma \models \varphi$ meaning that φ is a *semantical consequence* of the set of formulas Γ. It's a consequence in that every model of (all the formulas of) Γ is also a model of φ. Proof-theoretically one would write $\Gamma \vdash \varphi$. That's meant to represent: In the given proof system, φ can be derived using the given axiom plus the formulas from Γ (as so to say additional axioms). Again, in a favorable situation with soundness and completeness, one has $\Gamma \models \varphi$ iff $\Gamma \vdash \varphi$.

In some way, implication $\rightarrow$ represents a form of conseqence (as do inference rules): something follows from or is implied by something else. Thus $\Gamma \models \varphi$ may seem the same $\models \bigwedge \Gamma \rightarrow \varphi$: the conjunction of all formulas in Γ implies φ. For some logics, that's indeed the case, but one has to be careful, depending on the logic, and the exact interpretation of what $\models$ actually is supposed to mean. For instance, in first order logics when having formulas containting free variable. At any rate, that's fineprint for our interest and we don't dig deeper.

A proof system for propositional logic

As said, we don't venture into introducing, comparing, and investigating different proof systems. But let's have at least one or two simple examples for concreteness sake. We don't show a proof system for first-order logic, to keep it really simple, show one for propositional logic. A first-order logic system could contain the propositional rules as shown, but would need more to deal with quantification.

The system has three axioms and one inference rule. The name DN of the third axiom stands for *double negation*. The reason for that is, that negation can be defined or explained

$$\frac{}{\varphi \to (\psi \to \varphi)}\ \mathrm{Ax}_1$$

$$\frac{}{(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))}\ \mathrm{Ax}_2$$

$$\frac{}{((\varphi \to \bot) \to \bot) \to \varphi}\ \mathrm{DN}$$

$$\frac{\varphi \qquad \varphi \to \psi}{\psi}\ \mathrm{MP}$$

Table 2.2: Axioms and rules for a fragment of propositional logic

in terms of $\bot$ and implication by having $\neg\varphi$ defined as

$$\varphi \to \bot\ .$$

Actually, we were a bit sloppy when saying there are three axioms. We should more correctly say, that there are *3 axiom schemas*. What that means is that each axiom scheme represents infinitely many concrete axioms as *instantiation*. The "formulas" φ, ψ, and χ mentioned in the rule system are meta-variables representing formulas, but are not formulas themselves, strictly speaking.

The rule system from Table 2.2 is in a form called *Hilbert style*. Hilbert systems are an early and influential form of proof systems. It's however mostly of historical interest, as there are better ways to automate reasoning. What exactly is a good design depends also on what one wants to achieve (and for what kind of domain), for instance whether it's for *interactive* theorem proving or fully automated deduction. Hilbert-style systems are not good for much practical things. . .

To get at least a taste of the proof system (and a taste why it's clumsy), let's look at a very small example, deriving a rather trivial property.

Example 2.4.6. The following derivation establises $\vdash \varphi \to \varphi$.

$$(\varphi \to ((\varphi \to \varphi) \to \varphi)) \to ((\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi)) \qquad \mathrm{Ax}_2 \tag{2.19}$$

$$\varphi \to ((\varphi \to \varphi) \to \varphi) \qquad \mathrm{Ax}_1 \tag{2.20}$$

$$(\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi) \qquad \text{MP on (2.19) and (2.20)} \tag{2.21}$$

$$\varphi \to (\varphi \to \varphi) \qquad \mathrm{Ax}_1 \tag{2.22}$$

$$\varphi \to \varphi \qquad \text{MP on (2.21) and (2.22)} \tag{2.23}$$

□

The example also illustrates the concept of axiom schemas. The axioms from the derivation rules are not used "as-is" but they are instantiated. For instance, in the first two lines, it's a rather complicated specific case of the second axiom. It's problematic, since if we we proceed line by line, generating new theorems from previous until we get what we want, how to come up with the specific instances of the axioms that ultimately lead to success. And that's just one aspect why Hilbert systems are no practically useful representation. Another one is that it seems weird that in order to prove something pretty obvious like $\varphi \to \varphi$ one is forced to use some ridicously convoluted formula like $(\varphi \to ((\varphi \to \varphi) \to \varphi)) \to ((\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi))$ as part of the argument. That's not how it should be.

Natural deduction style proof systems

Better alternatives are known as *sequent calculi* or systems of *natural deduction.* All those systems are about deriving valid formulas. There are also so-called *refutation systems*, on the other hand, which do something else, namely they h try to refute a formula by checking if it's negation is *satisfiable*, among them various resolution methods and tableaux method.

We don't look here into those refutation alternatives, but we at least mention in which way sequent calculi and natural deduction systems differ from the Hilbert formulation. At at very high level, also sequent calculi and natural deduction systems are of the form described, there is a pool of axioms (resp. axioms schemas) and there are derivation rules. The systems typically work with derivation *trees* as proofs, not with sequences of formulas, but that's more a presentational issue. One can easily use a tree like presentation for proofs in the Hilbert rules of Table 2.2: the axioms are at the leaves and the inner nodes are instances of the modus ponens rule MP. Maybe also for historical reasons, Hilbert-style proofs are mostly presented as sequences, not as trees, but that's not the real difference.

Hilbert-style formulations were "criticized" as unnatural, in that it was perceived that using those axioms and rules does not really reflect in a natural way, that logical arguments are done. The above derivation in Example 2.4.6 should have given a feeling of that

The proof was pretty convoluted, in particular given the fact that this is really the most trivial valid thing that can be said about $\to$, and implication somehow lies the heart of logics. Logics is not just about describing things with formulas, it's about *drawing conclusions*, figuring out consequences from facts (as for instance in automated reasoning). Now that a formula φ is implied by itself seems the single most obvious thing about implication and it does not feel right that it requires so many steps to derive.

One could say, why not add that as axiom to the other ones if it's so central? One could of course do so. Of course then one has more axioms than one needs since, as the derivation shows, the "self-implication" formula can be derived.

Trying to get an axiomatization with it as axiom would also miss the point. The problem is not that it's hard to derive φ as being implied by φ as the most immediate consequence. One need a better way to draw conclusions.

Gentzen in particular suggested that a more natural way to arrange logical arguments is reasoning from hypotheses. Like.

Assuming φ as hypothesis, it follows that φ holds.

One could write $\varphi \rhd \varphi$ for that and use that as axiom. An as a rule one could use

$$\frac{\varphi_1 \rhd \varphi_2}{\rhd \varphi_1 \to \varphi_2} \to\mathrm{I} \tag{2.24}$$

Also that expresses a simple fact about $\to$. If one assumes φ_1 as hypothesis and that allows to prove φ_2, as stipulated in the premise of the rule, then surely one can derive the implication $\varphi_1 \to \varphi_2$ in the conclusion, and that implication is derived *without* assuming φ_1 as hypothesis on the left of $\rhd$ (the hypothesis has been discharged in the derivation step).

Such kind of argumentation working with hypotheses is characteristic for natural deduction system and sequent calculi. Another general distinguishing feature of such systems, in comparison to Hilbert's formulation is that they have *more rules and less axioms.*

In the discussion here, we focus on one connective, namely $\to$. Of course, there may be more built into the logics (as opposed to be explained as macros of others), like $\land$, $\neg$ etc. All of them need to be covered by a proof system. Hilbert's standpoint would be: there is exactly *one derivation rule, namely modus ponens* and that's it. At least in propositional logic, in first-order logic one would add also some rule(s) dealing with quantification. Anyway, all the propositional connectives are covered by an approriate selection of axioms, and propositional deriviation relies solely on applying modus ponens as the one and only rule.

Natural deduction, in contrast, would cover each logical connective with a few rules, characteristic for the connective. One particular connective, say $\land$ is characterised by so-called *introduction* and *elimination* rules. The terminology is easily explained. An intruduction rule, for instance for $\land$ is a rule, where the premises don't mention $\land$, but the conclusion does, so applying the rule introduces that construct. An elimination rule does the converse. At least one premise mentions $\land$, but the conclusion does not.

Let's have a look then at the rules for conjunction for illustration. Before doing that, and looking back at equation (2.24) for implication, we realize the the format of the rules of the derivation system gets more complex compared to the ones we started out with in equation (2.18). The the premises and the conclusions don't just consist of formulas. In the example from equation (2.24), the only premise is of the form $\varphi_1 \rhd \varphi_2$. That's slightly simplified, insofar that left of the symbol $\rhd$, in general there is a *set* of formulas, not just one as in the discussion from above. So-called sequent calculi often also work with sets of formulas on the *right* of $\rhd$. Let's stick however, to a formulation with one formula on the right. Thus, the rules operate with pairs of the form $\Gamma \rhd \varphi$, with Γ a set of formulas, the hypotheses. Such a *judgement* or sequent is intended to capture that, assuming *all* formulas from Γ, φ holds.

Concerning concretely the rules for $\land$, see Table 2.3. For $\land$, there is one introduction rule and two elimnation rules. It's characteristic for so called natural deduction system, that each connective of the syntax is covered by appropriate introduction and elimination *rules.* The so-called sequent calculi work similar, however focusing on elimination rules. As mentioned, the pair $\Gamma \rhd \varphi$ is often called a sequent. Still, the rules from Table 2.3 are natural deduction rules, namely that of a natural deduction system in sequent formulation, as it's called.

$$\frac{\Gamma \rhd \varphi_1 \qquad \Gamma \rhd \varphi_2}{\Gamma \rhd \varphi_1 \land \varphi_2}\ \land\text{-I} \qquad \frac{\Gamma \rhd \varphi_1 \land \varphi_2}{\Gamma \rhd \varphi_1}\ \land\text{-E}_1 \qquad \frac{\Gamma \rhd \varphi_1 \land \varphi_2}{\Gamma \rhd \varphi_2}\ \land\text{-E}_2$$

Table 2.3: Introduction and elimination rules for conjunction

A Hilbert style system would capture $\land$ not by rules, but by axioms like the ones from Table 2.3.

$$\varphi_1 \to \varphi_2 \to (\varphi_1 \land \varphi_2) \quad \land\text{-I} \qquad \varphi_1 \land \varphi_2 \to \varphi_1 \quad \land\text{-E}_1 \qquad \varphi_1 \land \varphi_2 \to \varphi_2 \quad \land\text{-E}_2$$

Table 2.4: Axioms for conjunction

We have said, natural deduction systems work with introduction and elimination rules. That also applies to implication $\to$. The rule modus ponens MP 2.2 is nothing else than the elimination rule for $\to$. For completeness sake, Table 2.5 shows the corresponding rules.

$$\frac{\Gamma, \varphi_1 \rhd \varphi_2}{\Gamma \rhd \varphi_1 \to \varphi_2}\ \to\text{-I} \qquad \frac{\Gamma \rhd \varphi_1 \to \varphi_2 \qquad \Gamma \rhd \varphi_1}{\Gamma \rhd \varphi_2}\ \to\text{-E}$$

Table 2.5: Introduction and elimination rules for implication

The introduction rule $\to$-I is the rule that shows how natural deduction systems (and likewise sequent calculi) work explicitly with a set of hypotheses and the set of hypotheses in the rule changes from Γ, φ in the premise to Γ in the conclusion, discharging φ. We had discussed that already in connection with the special situation for $\varphi \to \varphi$ in equation (2.24).

2.5 Modal logics

This section gives a taste of so-called modal logics. It covers some general aspects from the perspective of logics. In the verification part, we will deal with modal logics, but more from the perspective of how to do model checking and we cover a few specific modal and in particular temporal logics of interest in computer science and program verification. Still, some warm-up about modal logics in general can't hurt. After some general remarks in Section 2.5.1, we follow the same path as we did for propositional logics and first-order logics. We fix some syntax in Section 2.5.2, we address semantics, models, etc., and say a few words about proof systems in Sections 2.5.3 and 2.5.4.

Actually, there is no such thing as “the” modal logics. To some extent that is true also for propositional logics and first-order logics. Sure, one can select a few fundamental connectives or add more syntactic sugar to the syntax, there are classical versions or inuitionistic ones. And then there are many different ways how to design a proof system for the logics. But mostly, when saying “propositional logics”, everyone means more or less the same thing, the rest is details. Similar for first-order logics.

Modal logics are differentiated also by the fact “modality” can mean quite different things (like time, belief, knowledge and other things). All that can be captured by modalities, leading to different modal logics.

At the face of it, logics covering beliefs, time, knowledge have not much in common, at least there seems no reason why one would expect so. It turns out, they have a lot in common, namely what conceptually constitutes a **model** for such logics. The central section for us is thus Section 2.5.3, which introduces the idea of such models. They will be called **Kripke models** and in computer science, they basically can be seen as **transition systems** (a form of graphs). The section about the syntax of modal logic will be less interesting for us, since later we will work with *specific* syntax for specific logics, in particular temporal logics not with a generic syntax with modal operators. The section about proof system is less relevant, as we are concerned often with model checking, not deriving valid formulas. But the idea of Kripke structure will be important, as this is the notion of “model” when doing model(!) checking (for temporal logics).

2.5.1 Introduction

The roots of logics date back very long, and those of modal logic not less so. Aristotle not only wrote about syllogisms etc., like modus ponens, he also had his fingers in the origins of modal logic and discussed some kind of “paradoxes” in that context that gave food for thought for future generations, puzzling about modalities.

Very generally, a logic of that kind is concerned not with *absolute* statements (which are true or else false) but *qualified* statements, i.e., statements that “depend on” something. An example for a modal statement would be “tomorrow it rains”. It’s difficult to say in which way that sentence is true or false, only time will tell... It’s an example of a statement depending on “time”, and *tomorrow* is an example of a *temporal* modality. But there are other modalities, as well (referring to *knowledge* or *belief* like “I know it rains”, or “I believe it rains”) or similar qualifications of absolute truth.

Statements like “tomorrow it rains” or others were long debated, often with philosphical and/or even religous connotions like: is the future deterministic (and pre-determined by God’s providence), do persons have a free will, etc. Those questions will not enter the lecture. Nonetheless: determinism vs. non-determinism is meaningful distinction when dealing with program behavior, and we will also encounter temporal logics that view time as *linear* which kind of means, there is only *one* possible future, or *branching*, which means there are many. It’s however, not meant as a fundamental statement about the “nature of nature”, it’s just a distinction of how we want to treat the system. If we want to check individual runs, which are sequences of actions, then we are committing ourselves to a *linear* picture (even when dealing with non-deterministic programs). But there are branching alternatives to that view as well, which lead to branching temporal logics.

Different flavors of modal logic lead to different axioms. Let's write $\Box$ for the basic modal operator (whatever its interpretation), and consider

$$\Box\varphi \to \Box\Box\varphi \; , \tag{2.25}$$

with φ being some ordinary statement (in propositional logic perhaps, or first-order logic). If we are dealing with a logic of belief, should the following ne a valid formula: If I believe that something is true, do I believe that I believe that the thing is true? What about "I believe that you believe that something is true"? Do I believe it myself? Not necessarily so.

As a side-remark: the latter can be seen as a formulation in *multi-modal* logic: it's not about one single modality (being believed), but "person p believes", i.e., there's one belief-modality per person; thus, it's a *multi-modal* logic. We start in the presentation with a "non-multi" modal logic, where there is only *one* basic modality (say $\Box$). Technically, the syntax may feature two modalities $\Box$ and $\Diamond$, but to be clear: that does *not* yet earn the logic the qualification of "multi": the $\Diamond$-modality is in all considered cases expressible by $\Box$, and vice versa. It's analogous to the fact that (in most logics), $\forall$ and negation allows to express $\exists$ and vice versa.

Now, coming back to various interpretations of equation (2.25): if we take a "temporal" standpoint and interpret $\Box$ as *"tomorrow"*, then certainly the implication should not be valid. If we interpret $\Box$ differently, but still temporally, as *"in the future"* then again the interpretation seems valid.

If we take an "epistemologic" interpretation of $\Box$ as "knowledge", the left-hand of the implication would express (if we take a multi-modal view): "I know that you know that φ". Now we may ponder whether that means that then I *also* know that φ? A question like that may lead to philosophical reflections about what it means to "know" (maybe in contrast with "believe" or "believe to know", etc.).

The lecture will not dwell much on philospophical questions. The question whether equation (2.25) will be treated as *mathematical question*, more precisely a question of the assumed underlying *models* or *semantics*.

It's historically perhaps interesting: modal logic has attracted long interest, but the question of "what's the mathematics of those kind of logics" was long unclear. Long in the sense that classical logics, in the first half of the 20th century had already been super-thoroughly investigated and formalized from all angles with elaborate results concerning *model theory* and proposed as "foundations" for math etc. But no one had yet come up with a convincing, universally accepted answer for the question: "what the heck is a model for modal logics?". Until a teenager and undergrad came along and provided the now accepted answer, his name is *Saul Kripke*. And models of modal logics are now called *Kripke-structures* (it's basically transition systems).

2.5.2 Syntax

Modal logics comes equipped with connectives to express the modalities the logics is interested in. One typicallly finds two modal operators, called **"necessity"** and **"possibility"**.

Formulas $\Box\varphi$ and $\Diamond\varphi$ are read as "necessarily φ" and "possibibly φ" or simply "box φ" or "diamond φ".

Different flavors of modal logics interpret the modal operators differently. In a temporal setting $\Box\varphi$ can be interpreted as "always φ. And there are many other modal logics.

logics	$\Box\varphi$
temporal	always φ
doxastic	I believe φ.
epistemic	I know φ.
intuitionistic	φ is provable.
deontic	It ought to be the case that φ.

One finds logics with more operators, but those two are the classical ones. One can also work with combinations, like trying to capture the temporal evolution of knowlege, which would be a temporal-epistemic logic. We will restrict here the modal operators to $\Box$ and $\Diamond$, and in the later part of the lecture mostly work with a temporal mind-set.

Definition 2.5.1 shows a syntax for some vanilla modal logics. It is formulated as extension of propositional logics, so it's propositional modal logics. One can also use first-order logic as underlying logic, that does not change the modal part of the story.

Definition 2.5.1 (Formulas of modal logic)**.** The formulas of (propositional) modal logic are given by the following grammar.

$$\begin{array}{rcll} \varphi & ::= & \top \mid \bot & \text{atomic propositional formulass} \\ & \mid & \varphi \land \varphi \mid \neg\varphi \mid \varphi \to \varphi \mid \ldots & \text{propositional formulas} \\ & \mid & \Box\varphi \mid \Diamond\varphi & \end{array} \tag{2.26}$$

2.5.3 Semantics

Now to the core of the modal logic section, clarifying the semantics and the notion of models for such logics.

Kripke structures

The definition below makes use of the "classical" terminology for modal logics. It's actually quite simple, based on a relation, here written R, on some set. The relation is also called *accessibility relation* and the "points" connected by that relation are called *worlds*. So: a modal model is thus just a relation, or we also could call it a *graph*, but traditionally it's called a *frame* (a Kripke frame). That kind of semantics is also called *possible world semantics* (but not graph semantics or transition system semantics, even if that would be an ok name as well).

The Kripke frame in itself is not a model yet, in that it does not contain information to determine if a modal formula holds or not. The general picture is as follows: the elements of the underlying set W are called “worlds”: on some world some formulas holds, and in a different world, different ones. “Embedded” in the modal logic is an “underlying”logic. As said we mostly assume *propositional* modal logic, but one might as well consider an extension of first-logic with modal operators. For instance, in connection with *runtime verification* will make use of a first-order variant of LTL, called QTL, quantified temporal logic, but that will be later. In the propositional setting, what then is needed for giving meaning to model formulas is an interpretation of the propositional variables, and that has to be done **per world**.

In the section of propopositional logic, we introduced “propositional variable assignments” $\sigma : P \to \mathbb{B}$, giving a boolean value to each propositional variable from σ. What we now call a *valuation* does the same *for each world* which we can model as a function of type

$$W \to P \to \mathbb{B} \ .$$

Alternatively one often finds also the “representation” to have valuations of type $W \to 2^P$: for each world, the valuation gives the set of atomic propositions which “hold” in that word. Both views are, of course equivalent in being *isomorphic*.

Labelling The valuation function V associates a propositional value to each propositional value in eeach world. As mentioned, a Kripke frame may also be called a graph or also transition system. In the latter case, the worlds may be called less pompously just *states* and the accessibility relation is called transition relation. The individual edges in the graph are seen as *transitions* from one state to another. That terminolgy is perhaps more familiar in computer science. The valuation function can also be seen to **label** the states with propositional information. A transition system with attached information is also called *labelled transition system*.

But one has to be careful a bit with terminology. When it comes to *labelled* transition systems, additional information can be attached to transitions or states (or both). Often labelled transition systems, especially for some areas of model checking and modelling, are silently understood as *transition-labelled*. For such models, an edge between two states does not just expresse that one can go from one state to the other. It states that one can go from one state to the other *by doing such-and-such* as expressed by the label of the transition. In an abstract setting, the transitions may be labelled just with letters from some alphabet.

As we will see later, going from a transition system with unlabelled *transitions* to one with transition labels correspond to a generalization from “simple” modal logic to multi-modal logic. But independent on whether one considers transitions as labelled or not, there is a “state-labelling” at least, namely the valuation that is needed to interpret the propsitions per world or state.

As a side remark: classical automata can be seen as labelled transition systems, as well, with the transitions being labelled. There are also variations of such automata which deal with input *and* output (thereby called I/O automata). Two classical versions of that idea are used in describing hardware (which is a form of propositional logic as well. . .), both label the transitions for the input. However, one version labels the states with the output

(Moore-machines) whereas another one labels the transitions with the output (Mealy-machines), i.e., in the latter representation, transitions contain input as well as output information. Both correspond to different kinds of hardware circuitry (Moore roughly correspond to synchronous hardware, and Mealy to asynchronous one).

We will encounter automata later, as well, but in a form that fits to our particuar modal resp. temporal logic needs. In particular, we will look at Büchi-automata, which are like standard finite-state automata except that they can deal with infinite words (and not just finite ones). Those automata have connections, for instance, with LTL, a central temporal logic which we will cover.

Definition 2.5.2 (Kripke frame and Kripke model)**.**

- A *Kripke frame* is a structure (W, R) where
  - W is a non-empty set of *worlds*, and
  - $R \subseteq W \times W$ is called the *accessibility relation* between worlds.
- A *Kripke model* M is a structure (W, R, V) where
  - (W, R) is a frame, and
  - V a function of type $V : W \rightarrow (P \rightarrow \mathbb{B})$, called *valuation.*

The valuation function is isomorphic to a function $V : W \rightarrow 2^P$; we will make use of both representations.

Kripke models are also called *Kripke structures.* The standard textbook about model checking Baier and Katoen [1] does not even mention the word "Kripke structure" or "Kripke model", it basically uses *transition systems* instead of Kripke models with worlds called states (and the concept of Kripke frame is called *state graph* there). I say, it's "basically" the same insofar that there, they (sometimes) also care to consider labelled transitions, and furthermore, their transition systems are equipped with a set of *initial states.* Whether one has initial states as part of the graph does not make much of a difference.

Also the terminology concerning the set P varies a bit (we mentioned it also in the context of propositional logics). What we call here propositional variables, is also known as propositional constants, propositional atoms, symbols, *atomic propositions*, whatever.

Example 2.5.3 (Kripke model)*.* Assume $P = \{p, q\}$ as propositional variables. Figure 2.1 shows (the graphical representation of) a simple Kripke model. Formulaically $M = (W, R, V)$ is given by $W = \{w_1, w_2, w_3, w_4, w_5\}$, $R = \{(w_1, w_5), (w_1, w_4), (w_4, w_1), \ldots\}$, and $V = [w_1 \mapsto \emptyset, w_2 \mapsto \{p\}, w_3 \mapsto \{q\}, \ldots]$.

The example is slightly informal (and also later we allow ourselves these kind of "informalities", appealing to the intuitive understanding of the reader). There are five worlds, numbered for identification. In the Kripke model, they are referred to via $w_1, w_2, \ldots$ (not as $1, 2, \ldots$ as in the figure). Later, we often call corresponding entities states, not worlds, and then we tend to use $s_1, s_2, \ldots$ for typical states. For the valuation, we use a notation of the form [... ↦ ...] to denote a *finite* mapping.

In particular, we are dealing with finite mappings of type $W \rightarrow 2^P$, i.e., to subsets of the list of atomic propositions $P = \{p, q\}$. The sets are not explicitly noted in the graphical

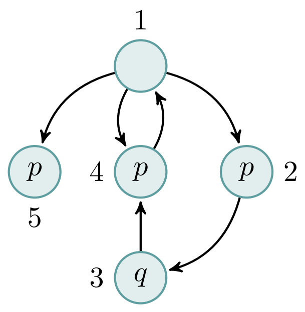


Figure 2.1: Example of a Kripke model

illustration, i.e., the set-braces $\{\dots\}$ are omitted. For instance, in world w_1, no propositional letter is mentioned, i.e., the valuation maps w_1 to the empty set $\emptyset$.

An isomporphic (i.e., equivalent) view on the valuation is, that it is a function of type $W \to (P \to \mathbb{B})$ which perhaps captures the intended interpretation better. Each propositional letter mentioned in a world or state is intended to evaluate to "true" in that world or state. Propositional letter not mentioned are intented to be evaluated to "false" in that world.

As a side remark: we used finite mappings inf the example and illustration, and in and many applications. The definition of Kripke structure, however, does *not* require that there is only a finite set of worlds, W in general is a *set*, finite or not.

Satisfaction relation

Now we come to the *semantics* of modal logic, i.e., how to interpret formulas of (propositional) modal formulas. That is done by defining the corresponding *satisfaction* relation, written as $\models$, as before. After the introduction and discussion of Kripke models or transition systems, the satisfaction relation should be fairly obvious to some extent, especially the part of the *underlying logic* (here propositional logic): the valuation V is made exactly so that it covers the base cases of atomic propositions, namely give meaning to the elements of P depending on the current world of the Kripke frame. The treatment of the propositional connectives $\wedge$, $\neg$, ... is identical to their treatment before. Remains the treatment of the real innovation of the logic, the modal operators $\Box$ and $\Diamond$.

Definition 2.5.4 (Satisfaction)**.** A modal formula φ is *true* (or it *holds*) in the world w of a model V, written $V, w \models \varphi$, if:

$$
\begin{array}{lll}
V, w \models p & \text{iff} & V(w)(p) = \top \\
\\
V, w \models \neg\varphi & \text{iff} & V, w \not\models \varphi \\
V, w \models \varphi_1 \vee \varphi_2 & \text{iff} & V, w \models \varphi_1 \text{ or } V, w \models \varphi_2 \\
\\
V, w \models \Box\varphi & \text{iff} & V, w' \models \varphi, \text{ for all } w' \text{ such that } w\ R\ w' \\
V, w \models \Diamond\varphi & \text{iff} & V, w' \models \varphi, \text{ for some } w' \text{ such that } w\ R\ w'
\end{array}
$$

As mentioned, we consider V to be of type $W \to (P \to \mathbb{B})$. If we equivalently assumed a type $W \to 2^P$, the base case of the definition would read $p \in V(w)$, instead.

For now, we prefer the former presentation for 2 reasons (but actually, it does not matter of course). One is, it seems to fit better with the presentation of propositional logic, generalizing directly the concept a boolean valuation. Secondly, the picture of "assigning boolean values to variables" fit better with seeing Kriple models more like transition systems, for instance capturing the behavior of computer programs. There, we are not so philosphically interested in speaking of "worlds" that are "accessible" via some accessibility relation R, it's more like states in a progam, and doing some action or step does a transition to another state, potentially changing the memory, i.e., the content of variable, which in the easiest case may be boolean variables. So the picture that one has a control-flow graph of a program and a couple of variables (propositional or Boolean variables here) whose values change while the control moves inside the graph seems rather straightforward and natural.

Sometimes, other notations or terminology is used, for instance $w \models_M \varphi$. Sometimes, the model M is fixed (for the time being), indicated by the words like. "Let in the following M be defined as …", in which case one finds also just $w \models \varphi$ standing for "state w satisfies φ", or "φ holds in state w" etc. but of course the interpretation of a modal formula requires that there is alway a transition system relative to which it is interpreted.

Often one finds also notations using the "semantic brackets" $[\![\_]\!]$. Here, the meaning (i.e., truth-ness of false-ness of a formula, depends on the Kripke model as well as the state, which means one could define a notation like $[\![\varphi]\!]^M_w$ as $\top$ or $\bot$ depending on wether $M, w \models \varphi$ or not. Remember that we had similar notation in first-order logic $[\![\varphi]\!]^I_\sigma$ We discussed (perhaps uneccessarily so) two isomorphic view of the valuation function V.

Even if not relevant for the lecture, it could be noted that a third "viewpoint" and terminology exists in the literature in that context. Instead of associating with each world or state the set of propositions intended to hold in that state, one can define a model also "the other way around": then one associates with each propositional variable the set of states in which the proposition is suppoed to hold, one would have a "valuation"

$$\tilde{V} : P \to 2^W \ .$$

That's of course also an equivalent and legitimate way of proceeding. It seems that this representation is not "popular" when doing Kripke models for the purpose of capturing systems and their transitions (as for model checking in the sense of our lecture), but for Kripke models of intuionistic logics. Kripke also propose "Kripke-models" for that kind of logics (for clarity, I am talking about intuitionistic propositional logics or intuitionistic first-order logice et.c, not (necessarily) intuitionistic *modal* logics). In that kind of setting, the accessibility relation has also special properties (being a partial order), and there are other side conditions to be aware of. As for terminology, in that context, one sometimes does not speak of "w satisfies φ (in a model), for which we write "$w \models p$", but says "world w *forces* a formula φ", for which sometimes the notation $w \Vdash \varphi$ is favored. But those are mainly different traditions for the same thing.

For us, we sometimes use notations like $[\![\varphi]\!]^M$ to represent the set of all states in M that satisfy φ, i.e.,

$$[\![\varphi]\!]^M = \{w \mid M, s \models \varphi\} .$$

In general (and also other logics), $\models$- and $[\![\_]\!]$-style notations are interchangable and interdefinable. And we will freely make use of both

"Box" and "diamond"

The pronounciation of $\Box\varphi$ as "necessarily φ" and $\Diamond\varphi$ as "possibly φ" are *generic*. When dealing with specific interpretations, they get more specific meanings and then be called likewise: "in all futures φ" or "I know that φ" etc. Related to the intended mindset, one imposes different restrictions in the accessibility relation R. In a temporal setting, if we interpret $\Box\varphi$ as "tomorrow φ", then it is clear that $\Box\Box\varphi$ ("φ holds in the day after tomorrow") is not equivalent to $\Box\varphi$. If, in a different temporal mind-set, we intend to mean $\Box\varphi$ to represent "now and in the future φ", then $\Box\Box\varphi$ and $\Box\varphi$ are equivalent. That reflects common sense and reflects what one might think about the concept of "time" and "days" and "future". Technically, and more precisely, it's a property of the assumed class of frames (i.e., of the relation R). If we assume that all models are built upon frames where R is *transitive*, then $\Box\varphi \to \Box\Box\varphi$ is generally true.

Validity and frame validity

We should be more explicit about what it means that a formula is "generally true". We have encountered the general terminology of a formula being "true" vs. being "valid" already. In the context of modal logic, the truth-ness requires a model and a state to judge the truth-ness: $M, w \models \varphi$. A model M is of the form (W, R, V), it's a frame (= "graph") together with a valuation V. A *propositional* formula is *valid* if it's true for all boolean valuations (and the notion coincided with being a propositional tautology).

Now the situation get's more finegrained (as was the case in first-order logics). A modal formula is *valid* if $M, w \models \varphi$ for all M and all w. For that one can write

$$\models \varphi \qquad (2.27)$$

So far so good. But then there is also a middle-ground, where one fixes the frame (or a class of frames), but the formula must be true *for all valuations* and all states. For that we can write

$$(W, R) \models \varphi \tag{2.28}$$

Let's abbreviate with F a frame, i.e., a tuple (W, R). We could call that notion *frame validity* and say for $F \models \varphi$ that "φ is valid in frame F". So, in other words, a formula is valid in a frame F if it holds in all models with F as underlying frame and for all states of the frame.

One uses that definition not just for a single frame; often the notion of frame-validity is applied to *sets* of frames, in that one says $F \models \varphi$ for all frames F such that …". For instance, all frames where the relatiton R is *transitive* or *reflexive* or whatever. Those restrictions of the allowed class of frames reflect then the intentions of the modal logic (temporal, epistemic …), and one could speak of a formula to be "transitivity-valid" for instance, i.e., for all frames with a transitive accessibility relation.

The latter would be an ok terminology, but it's not standard. There are (for historic reasons) more esoteric names for some standard classes, for instance, a formula could be S4-valid. That refers to one particular restriction on R which corresponds to a particular set of axioms traditionally known as S4. See below for some examples.

Further notational discussion and preview to LTL Coming back to the informal "temporal" interpretation of $\Box\varphi$ as either "tomorrow φ" vs. "now and in the future φ", where the accessibility relation refers to "tomorrow" or to "the future time, from now on". In the latter case, the accessibility relation would be reflexive and transitive. When thinking about such a temporal interpretation, there may also be another assumption on the frame, depending on how one sees the nature of time and future.

One way would be to see the time as *linear*. Points in time form a line, fittingly called a *timeline*, connecting the past and the future, with "now" in the middle, perhaps measured in years or seconds, or steps in an run of a program etc. With such a linear picture in mind, it's also clear that there is no difference between the modal operators $\Box$ and $\Diamond$.[4] In the informal interpretation of $\Box$ as "tomorrow", one should have been more explicit that "tomorrow" was meant "for all possible tomorrows" to distinguish it from $\Diamond$ that represent "there exist a possible tomorrow". In the linear timeline picture, there is only one tomorrow, we conventionally say "the next day" not "for all possible next days" or some such complications. Consequently, if one has such a linear picture in mind (resp. works only with such linear frames), one does not actually *need* two modal operators $\Box$ and $\Diamond$, one can collapse them into one. Conventionally, for that collapsed one, one writes $\bigcirc$. A formula $\bigcirc\varphi$ is interpreted as "in *the* next state or world, φ holds" and pronouced "next φ" for short. The $\bigcirc$ operator will be part of LTL (linear-time temporal logic), which is an important logic used for model checking and which will be covered later. When we (later) deal with LTL, the operator $\bigcirc$ corresponds to the modal operators $\Diamond$ and $\Box$ collapsed into one, as explained. Besides that, LTL will have *additional* operators written (perhaps confusingly) $\Box$ and $\Diamond$, with a *different interpretation* capturing "always" and "eventually"

[4] Characterize as an exercise what *exactly* (not just roughly) the condition the accessibility relation must have to make $\Box$ and $\Diamond$ identical.

Those are also temporal modalities, but their interpretation in LTL is different from the ones that we haved fixed for now, when discussing modal logics in general.

Restrictions on the frames, resp. the accessibility relation

As mentioned, different classes of model logics arise by imposing restrictions on which frames one considers. Restrictions on the binary relation $R \subseteq W$ include that it is reflexive, transitive, (right) Euclidian, total, that it's an order relation, and more. It's not the goal of the lecture to study those and compare different logics (and many variations have been studied indeed). Still, we at least mention some common restrictions.

Definition 2.5.5. A binary relation $R \subseteq W \times W$ is

- *reflexive* if every element in W is R-related to itself.
$$\forall a.\ a\ R\ a$$
- *transitive* if
$$\forall a\ b\ c.\quad a\ R\ b \wedge b\ R\ c \rightarrow a\ R\ c$$
- (right) *Euclidean* if
$$\forall a\ b\ c.\quad a\ R\ b \wedge a\ R\ c \rightarrow b\ R\ c$$
- *total* if
$$\forall a.\ \exists b.\quad a\ R\ b$$

The following remark may be obvious, but anyway: The quantifiers like $\forall$ and the other operators $\wedge$ and $\vee$ are not meant here to be vocabulary of some (first-order) logic, they are meant more as mathematical statements, which, when formulated in for instance English, would use sentences containing words like "for all" and "and" and "implies". One could see if one can formalize or characterize the defined concepts making use formally of a first-order logic, but that's not what is meant here. We use the logical connectives just as convenient shorthand for English words.

Example 2.5.6 shows some how some accessibiity relations are captured by different formulas, in that they are valid under the corresponding restriction.

Example 2.5.6.

- $(W, R) \models \Box\varphi \rightarrow \varphi$ iff R is reflexive.
- $(W, R) \models \Box\varphi \rightarrow \Diamond\varphi$ iff R is total.
- $(W, R) \models \Box\varphi \rightarrow \Box\Box\varphi$ iff R is transitive.
- $(W, R) \models \neg\Box\varphi \rightarrow \Box\neg\Box\varphi$ iff R is Euclidean.

Some exercises

Prove the double implications from the slide before!

Hints By "double implications", the iff's (if-and-only-if) are meant. In each case there are two directions to show.

- The forward implications are based on the fact that we quantify over *all* valuations and all states. More precisely; assume an arbitrary frame (W, R) which does *not* have the property (e.g., reflexive). Find a valuation and a state where the axiom does not hold. You have now the contradiction …
- For the backward implication take an arbitrary frame (W, R) which *has* the property (e.g., Euclidian). Take an arbitrary valuation and an arbitrary state on this frame. Show that the axiom holds in this state under this valuation. Sometimes one may need to use an inductive argument or to work with properties derived from the main property on R (e.g., if R is euclidian then $w_1 R w_2$ implies $w_2 R w_2$).

2.5.4 Proof theory and axiomatic systems

We only sketch proof theory of modal logic, as we are more interested in model checking as opposed to verify that a formula is valid. There are connections between these two questions, though. As explained earlier, proof theory is about formalizing the notion of proof. That's done by defining a formal system, a proof system, that allows to *derive* or *infer* formulas from others. Formulas a priori given are also called *axioms*, and rules allow new formulas to be derived from previously derived ones (or from axiom). One may also see axioms as special form of rule, namely one without *premises*.

The style of presenting the proof system here is the plain old Hilbert-style presentation. As mentioned, there are other styles of presentations, some better suited for interactive, manual proofs and some for automatic reasoning, and in general more elegant anyway. As also mentioned, one difference between Hilbert-style and the natural deduction style presentations is that Hilbert's presentation put's more weight on the axioms, whereas the alternative downplay the role of axioms and have more deduction rules (generally speaking). That suits us fine: As discussed, different classes of frames (transitive, reflexive …) correspond to axioms or selection of axioms, and we have seen examples for that in Example 2.5.6.

Since we intend (classical propositional) modal logics to encompass classical propositional logic not just syntactically but also conceptually/semantically, we have all propositional tautologies as derivable. Furthermore, we have the standard rule of derivation, already present in the propositional setting, namely *modus ponens*.

That so far took care of the propositional aspects (but note that MP can be applied to all formulas, of course, not just propositional ones). But we have not really taken care of the modal operators $\Box$ and $\Diamond$. Now, having lot of operators is nice for the user, but puts more burden when formulating a proof system (or implementing one) as we have to cover more case. So, we treat $\Diamond$ as *syntactic sugar*, as it can be expressed by $\Box$ and $\neg$. Note: "syntactic sugar" is a well-established technical term for such situations, mostly used in the context of programming languages and compilers. Anyway, we now need to cover only one modal operator, and conventionally, it's $\Box$, necessitation. The corresponding rule consequently is often called the rule of (modal) *necessitation*. The rule is below called Nec, sometimes also just N or also called G (maybe historically so).

$$\frac{\varphi \text{ is a propositional tautology}}{\varphi}\ \text{PL}$$

$$\frac{}{\Box(\varphi_1 \to \varphi_2) \to (\Box\varphi_1 \to \Box\varphi_2)}\ \text{K}$$

$$\frac{\varphi \to \psi \qquad \varphi}{\psi}\ \text{MP}$$

$$\frac{\varphi}{\Box\varphi}\ \textsc{Nec}$$

Table 2.6: Axioms for the base line propositional modal logics ("K")

Is that all? Remember that we touched upon the issue that one can consider special classes of frames, for instance those with *transitive* relation R or other special cases, that lead them to special *axioms* being added to the derivation system. Currently, we do *not* impose such restrictions, we want **general frame validity**. So does that mean, we are done? At the current state of discussion, we have the propositional part covered including the possibility do to propositional-style inference (with modus ponens), we have the plausible rule of necessitation, introducing the $\Box$-modality. Apart from that, the two aspects of the logic (the propositional part and modal part seem conceptually *separated*. Note: a formula $\Box p \to \Box p$ "counts" as (an instance of a) propositional tautology, even if $\Box$ is mentioned. A question therefore is: are the two parts of the logic somehow further connected, even if we don't assume anything about the set of underlying frames?

The answer is, **yes**, and that connection is captured by the *axiom* stating that $\Box$ *distributes* over $\to$. The axiom is known as distribution axiom or traditionally also as axiom K. In a way, the given rules are the standard *base line* for all classical modal logics. Modal logics with the propositional part covered plus necessitation and axiom K are also called *normal* modal logics.

As a side remark: there are also certain modal logics where K is dropped or replaced, which consequently are *no longer* normal logics. Note that it means they no longer have a Kripke-model interpretation either. Since our interest in Kripke-models is that we use transition systems as representing steps of programs, Kripke-style thinking is natural in the context of our course. Non-normal logics are more esoteric and "unconventional" and we don't go there.

The distribution axiom K is written as "rule" without premises. The system focuses on the "new" aspects, i.e., the modal part. It's not explicit about how the rules look like that allow to *derive* propositional tautologies (which would be easy enough to do, and includes MP anyway). We have seen those earlier anyway.

The sketched logic is is also known under the name **K** itself, so K is not just the name of the axiom. The presentation here is Hilbert-style, but could also present the same logics differently.

We show in Table 2.8 a few more axioms (with their traditional names, some of which are just numbers, like "axiom 4" or "axiom 5"). In the literature, then one considers and studies "combinations" of those axioms (like K + 5), and they are traditionally also known under special, not very transparent names like "S4" or "S5". See Table 2.8 for some better known ones.

$$\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi) \tag{K}$$
$$\Box\varphi \to \Diamond\varphi \tag{D}$$
$$\Box\varphi \to \varphi \tag{T}$$
$$\Box\varphi \to \Box\Box\varphi \tag{4}$$
$$\neg\Box\varphi \to \Box\neg\Box\varphi \tag{5}$$
$$\Box(\Box\varphi \to \psi) \to \Box(\Box\psi \to \varphi) \tag{3}$$
$$\Box(\Box(\varphi \to \Box\varphi) \to \varphi) \to (\Diamond\Box\varphi \to \varphi)) \tag{Dum}$$

Table 2.7: Various properties of R and their axioms

The first ones are pretty common and are connected to more or less straightforward frame conditions (except K which is, as said, generally the case for a frame-based Kripke-style interpretation). Observe that T implies D.

The are many more different axiom studied in the literature, how they are related and what not. The axiom called DUM is more esoteric ([3] calls it "[among the] most bizzare formulae that occur in the literature") and actually, there are even different versions of that (DUM$_1$, DUM$_2$ ...).

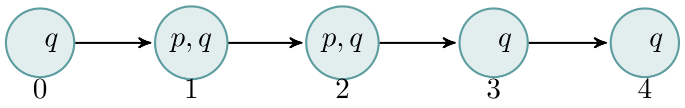
Logic	Axioms	Interpretation	Properties of R
D	K D	deontic	total
T	K T		reflexive
K45	K 4 5	doxastic	transitive/euclidean
S4	K T 4		reflexive/transitive
S5	K T 5	epistemic	reflexive/euclidean
			reflexive/symmetric/transitive
			(i.e. equivalence relation)

Table 2.8: Different flavors of modal logic

Concerning the terminology *doxastic* logic is about beliefs, *deontic* logic tries to capture obligations and similar concepts. Epistemic logic is about knowledge.

2.5.5 Exercises

Exercise 2.5.7 (Formulas holding in worlds of a model)**.** Consider the frame (W, R) with $W = \{0, 1, 2, 3, 4\}$ and $(i, i + 1) \in R$.



Let the "valuation" $\tilde{V}(p) = \{1, 2\}$ and $\tilde{V}(q) = \{0, 1, 2, 3, 4\}$ and let the model M be $M = (W, R, V)$. Which of the following statements are correct in M and why?

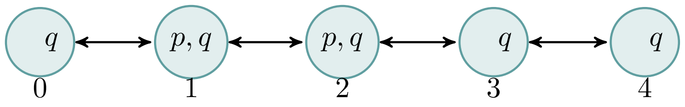
$$\begin{array}{rcl} M, 0 & \models & \Diamond\Box p \\ M, 0 & \models & \Diamond\Box p \rightarrow p \\ M, 2 & \models & \Diamond(q \land \neg p) \land \Box(q \land \neg p) \\ M, 0 & \models & q \land \Diamond(q \land \Diamond(q \land \Diamond(q \land \Diamond q))) \\ M & \models & \Box q \end{array}$$

□

Solution 2.5.8 (of Exercise 2.5.7). *The answers to the above questions are: yes, no, yes, yes, yes.*

Perhaps a remark concerning the status $\Diamond\Box p$ *in the first situation. The frame is* □

Exercise 2.5.9 (Bidirectional frame). A frame (W, R) is *bidirectional* iff $R = R_F + R_P$ s.t. $\forall w, w'.\ w\ R_F\ w' \leftrightarrow w'\ R_P\ w$.



Consider $M = (W, R, V)$ from before. Which of the following statements are correct in M and why?

$$\begin{array}{rcl} M, 0 & \models & \Diamond\Box p \\ M, 0 & \models & \Diamond\Box p \rightarrow p \\ M, 2 & \models & \Diamond(q \land \neg p) \land \Box(q \land \neg p) \\ M, 0 & \models & q \land \Diamond(q \land \Diamond(q \land \Diamond(q \land \Diamond q))) \\ M & \models & \Box q \\ M & \models & \Box q \rightarrow \Diamond\Diamond p \end{array}$$

Solution 2.5.10. *The solutions are: no, yes, no, yes, yes, no.*

The frame used in Exercise 2.5.9 is bi-directional. Effectively, the relation R in the example is *symmetric*, so actually the example would not need to mention the concept of being bi-directional. Later, when we present *temporal logics*, there are variants that can speak not just about the future, but also about the past. In such a temporal setting, one could interpret the formulas on bi-directional frames, the future operators being based on R_F and the operators taking about the past on R_P.

Exercise 2.5.11 (Validities)**.** Which of the following formulas are *valid* in modal logic. For those that are not, argue why and find a class of frames on which they become valid.

$$\begin{array}{l} \Box\bot \\ \Diamond p \to \Box p \\ p \to \Box\Diamond p \\ \Diamond\Box p \to \Box\Diamond p \end{array}$$

□

Solution 2.5.12. *1. $\Box\bot$: Valid on frames where $R = \emptyset$.*
2. $\Diamond p \to \Box p$: Valid on frames where R is a partial function.
3. $p \to \Box\Diamond p$: Valid on bidirectional frames.
4. $\Diamond\Box p \to \Box\Diamond p$: Valid on Euclidian frames.

□

As for further reading, [4] and [2] may be good reads.

2.6 Dynamic logics

2.6.1 Introduction

Dynamic logics is a so-called program logics and used in connection with a pogramming language. I.e., the formalism contains two levels of syntax, one the programming notation, and then the logics, both interwoven. A specified program consists of the program code plus logical annotations, sprinkled throughout the code. Hoare-logic is a typical example for that, where the code is annotated with assertions (in first-order logic or similar). Indeed, dynamic logics can be seen as a generalization of Hoare logics. Such a specification style, formulas being written as part of the code syntax, is a bit different from the way, for instance, LTL will be used. There, the logic is used *externally*, specifying the behavior of a program or process *from the outside*. It's a bit like the difference between a black box and white box view

We said, Hoare-logics can be seen as a special case of dynamic logics. Our exposition does not present it under that perspective, but a different one, namely as a form of multi-modal logic.

modal logic: gives us the power to talk about *changing of state*. Modal logics is natural when one is interested in systems that are essentially modeled as states and transitions between states. The slide calls first-order logic as very expressive, but all is relative. There are much more expressive logics and FOL has some serious restrictions, as far as expressivity is concerned.

We want to talk about programs, states of programs, and change of the state of the computer via executing programming instructions, like assignments, that change the system state. The (operational) semantics of a program is typically some form of transition system (or Kripke structure), and we know already, that logics that talk about transition systems are *modal logics*. There are various connections between FOL and modal logics, for instance, modal logic may be seen as FOL with one free variable, but seeing it like that we loose the "beauty" of modal logics (being tailor-made as logic speaking about transition systems). Such connections between other logics and modal logics are not important for the course.

2.6.2 Multi-modal logic

Dynamic logics later may be seen as a form of multi-modal logics. Thus we start with explaining what that is. Actually, it's pretty simple. Instead of one relation in the transition system, we handle many, instead of having possibility and necessity as modal operator over one relation, we have modal operators for each of them. That's basically it. We start by illustrating it with a Kripke frame with 2 relations R_a and R_b, both subsets of $W \times W$. An alternative and equivalent way of seeing it that one deals with a "labelled transition relation", i.e. one $R \subseteq W \times A \times W$, where A is the set of labels, in the illustrative "example" $A = \{a, b\}$.

Multi-modal logic

"Kripke frame" (W, R_a, R_b), where R_a and R_b are two relations over W.

Syntax (2 relations) *Multi-modal logic* has one modality for each relation:

$$\varphi ::= p \mid \bot \mid \varphi \to \varphi \mid \Diamond_a \varphi \mid \Diamond_b \varphi \tag{2.29}$$

where p is from a set of propositional constants (i.e., functional symbols of arity 0) and the other operators are *derived* as usual:

$$\varphi ::= \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg \varphi \mid \Box_a \varphi \mid \Box_b \varphi \tag{2.30}$$

Semantics: "natural" generalization of the "mono"-case

$$M, w \models \Diamond_a \varphi \text{ iff } \exists w' : w R_a w' \text{ and } M, w' \models \varphi \tag{2.31}$$

- analogously for modality $\Diamond_b$ and relation R_b

As *multi-modal* logic: obvious generalization of modal logic from before

1. The relations can overlap; i.e., their intersection need not be empty
2. of course: more than 2 relations possible, for each relation one modality.
3. There may be *infinitely* many relations and infinitely many modalities.

Infinitely many modalities are possible. One has to be slightly careful then, though. Infinitely many modalities may pose theoretical challenges (not just for the question how to deal with them computationally). We ignore issues concerning that in this lecture. As a further remark: later there will be PDL and maybe TLA (temporal logic of actions). In those kind of logics, the different relations R_a arise from "programs" or "actions". So, if one sees the a as taken from a set A, then the set is not taken as unstructured and containing uninterpreted letters from an alphabet. Instead, the "actions" may have a syntax in itself, representing programs or single steps in a program. Since there are many *actions* or *programs*, which lead to many corresponding "modalities".

2.6.3 Dynamic logics

Dynamic logics

- different variants
- can be seen as special case of multi-modal logics
- variant of Hoare-logics
- here: PDL on **regular** programs
- "P" stands for "propositional"

Regular programs

DL Dynamic logic is a multi-modal logic to talk about programs.

here: dynamic logic talks about **regular programs**

Regular programs are formed syntactically from:

- *atomic* programs $\Pi_0 = \{a, b, ...\}$, which are indivisible, single-step, basic programming constructs
- *sequential* composition $\alpha \cdot \beta$, which means that program α is executed/done first and then β.
- *nondeterministic choice* $\alpha + \beta$, which nondeterministically chooses one of α and β and executes it.
- *iteration* α^*, which executes α some nondeterministically chosen finite number of times.
- the special `skip` and `fail` programs (denoted $\mathbf{1}$ resp. $\mathbf{0}$)

Dynamic logics speaks about "programs", where a program is written in some "notation". in other words, programs written in some form of programming language. Obviously, there are a huge variety of notations and languages. Dynamic logic may have been used for real programming languages. For instance, the KeY verifier tool is a theorem prover for Java programs, based on dynamic logics. Thus, the underlying program "notation" is Java.

Here, we discuss dynamic logics using a rather more restricting programming notation, namely *regular programs*. As mentioned earlier, DL is a multi-modal logic where one adds "structure" to the set of labels A (if we see the multi-transitions relations as one labelled transition relation). Here, we add as structure on A basically *regular expressions*, which are seen as the underlying "programming languages". Of course, that's a very abstract way of representing programs, basically abstracting away from concrete data. One focuses on sequential composition, non-deterministic choices, and iteration. It's a level of abstraction that correponds to control-flow graphs (with the data-part abstracted away).

Regular programs and tests

Definition 2.6.1 (Regular programs). The syntax of *regular programs* $\alpha, \beta \in \Pi$ is given according to the grammar:

$$\alpha ::= a \in \Pi_0 \mid \mathbf{1} \mid \mathbf{0} \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^* \mid \varphi? \,. \tag{2.32}$$

The clause $\varphi?$ is called *test*.

Tests can be seen as special atomic programs which may have *logical* structure, but their execution properly **terminates** in the same state iff the test succeeds (is true), otherwise **fails** if the test is deemed false in the current state.

As for termination: test have two outcomes, a positive and a negative. If the test "succeeds", the test properly terminates. The qualification "properly" for this form of termination is a hint that there are also other forms of termination: When the test fails, it "terminates" in that it fails. That is sometimes called *improper termination*. In programming languages, a cause of improper termination can be raising an (uncaught) *exception*. Actually, the behavior of tests is connected to *assertions* as known from many high-level programming language. For example, in Java, one can use `assert(b)` as construct, where `b` is a Boolean expression. Basic programming hygiene mandates, that `b` has no side effects, so it's a side-effect free boolean expression over the variables used in the programs (like making sure that `x >= y` or similar). Indeed, assert-statements of that form corresponds to a simple form of tests as supported in DL. It's a simple form, in that only propositional formulas (= boolean expressions) over program variables are allowed, whereas in dynamic logic, more complex formulas are allowed. In particular, formulas in the multi-modal logic (here with regular programs as syntax) which is known as dynamic logic.

Note: it's about proper or improper termination of the *test*, not the program. In general a test is used in a program, for instance a test followed by the rest of the program. In regular programs that is written as $\varphi? \cdot \alpha$ (in other contexts and real programming languages, often semicolon **;** is used for sequential composition instead).

The difference between proper termination and failure is: in the first case, the test terminates without consequences (especially no side effects) and the program behaves as α. On the other hand, when the test fails, the whole program fails, i.e., terminates improperly as well. That corresponds to the situation, that an assertion `assert(b)` raises an exception which is uncaught and causes the whole program to terminate in a non-proper way.

Tests

- *simple* Boolean tests: $\varphi ::= \top \mid \bot \mid \varphi \rightarrow \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$
- *complex* tests: $\varphi?$ where φ is a logical formula in *dynamic logic*

Propositional Dynamic Logic: Syntax

Definition 2.6.2 (PDL syntax)**.** The formulas φ of *propositional dynamic logic* (PDL) over regular programs α are given as follows.

$$\begin{array}{lcl} \alpha & ::= & a \in \Pi_0 \mid \mathbf{1} \mid \mathbf{0} \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^* \mid \varphi? \\ \varphi & ::= & p, q \in \Phi_0 \mid \top \mid \bot \mid \varphi \to \varphi \mid [\alpha]\varphi \end{array} \tag{2.33}$$

where Φ_0 is a set of atomic propositions.

1. programs, which we denote $\alpha ... \in \Pi$
2. formulas, which we denote $\varphi ... \in \Phi$

Propositional Dynamic Logic (PDL): based on propositional logic, only

PDL: remarks

- Programs α interpreted as a relation R_α

$\Rightarrow$ multi-modal logic.
- $[\alpha]\varphi$ defines many modalities, one modality for each program, each interpreted over the relation defined by the program α.
- The relations of the basic programs are just given.
- Operations on/composition of programs are interpreted as operations on relations.
- ∞ many complex programs $\Rightarrow$ ∞ many relations/modalities
- $[..]\varphi$ is the universal one, with $\langle .. \rangle\varphi$ defined as usual.

Intiutive meaning/semantics of $[\alpha]\varphi$ "If program α is started in the current state, then, *if* it terminates, then in its final state, φ holds."

Actually, we basically "know" the interpretation of the two modalities already, as they are known from the modal logic part. The $[\alpha]$ corresponds to a *universal quantification* of all α-successors, the dual $\langle\alpha\rangle$ stipulates the *existance* of an α-successor.

There is, however, a subtlety now which was not much visible in the modal logical setting (including the multi-modal one), where the actions were uninterpreted symbols from an alphabet. Now, they *are* interpreted, namely interpreted as regular programs, and that includes the possibility of non-termination. So having a pair s_1 and s_2 in a relation R_α, i.e., $s_1 R_\alpha s_2$ (for which we also write $s_1 \xrightarrow{\alpha} s_2$), it's interpreted as that a program starts in state s_1 and when executing α, it reaches s_2 when α properly terminates. So, a transition $s_1 \xrightarrow{\alpha} s_2$ implies that α terminates properly. The absence of such a transition implies that α does not properly terminate. In our regular languages, that would be caused by a failed test. In real programming languages, they may be other reason for not terminating properly. For instance, divergence (like infinite looping).

For the modalities, $[\alpha]\varphi$, being interpreted as universal quantification over all α-successors stipulates for a state s_1, that, **if** the execution of α terminates starting from s_2, then the progam will be in a state, say s_2, for which φ holds. If the program, starting at state s_1 does not terminate, then $[\alpha]$ *vacuously* holds in s_1. Programs may be non-deterministic (including regular programs). So it may be the case, that some executions of α terminate and others don't. The formula $[\alpha]\varphi$ imposes a restriction (namely φ) on the post-states for all terminating executions of α. But does not require *anything* for the others. This is connected with the notion of *partial* correctness (as opposed to total correctness).

In contrast, the dual operator $\langle\alpha\rangle\varphi$ insists on that there *exists* an execution, where α terminates after which φ holds.

The terminology if partial and total correctness may become more plausible if one would consider the restricted setting where programs are *deterministic*. In the current setting with regular programs, that would not be the case, but dynamic logics, as stated, make sense also with other program notation, for instance, talking about sequential and deterministic programs. For example, the core of the Key-tool focuses on *sequential* Java programs (though also adaptations to multi-threaded Java exist).

Anway, if the programming language is happens to be deterministic, then for each s_1, there exists *at most one* s_2 such that $s_1 \xrightarrow{\alpha} s_2$: if α terminates, s_2 is *the* sucessor state, if not, there is no successor state. In that setting as before, $[\alpha]\varphi$ specifies that, if the program terminates, the successor state must satisfy φ, but if it does not terminate, it's fine too (= partial correctness). The dual operator $\langle\alpha\rangle\varphi$, on the other hand, *insists* that α terminates plus that φ holds afterwards (= *total* correctness).

The notions of partial and total correctness are often used in connection with Hoare logic. Indeed, Hoare logic can be seen as a special, restricted case of dynamic logic. There, one operates syntactically not with "modal operators", but one write specifications as so-called *triples*. Triples can be interpreted in a total or in a partial correctness way. Sometimes, one distiguished that notationally by writing $[\varphi_1]\ \alpha\ [\varphi_2]$ and $\{\ \varphi_1\ \}\ \alpha\ \{\ \varphi_2\ \}$, correspondingly. Note that, unlike here, the logical formulas φ_1 and φ_2 do not contain "program syntax": in dynamic logic, the two layers (programs and logic) are more intimately inverwoven (via the modalities and the tests), in Hoare-logics, the layers are more clearly separated (programs sprinkled with formulas in between). In that sense, Hoare-logic is a restricted form of dynamic logic.

Exercises: "programs"

Define the following programming constructs in PDL:

$$
\begin{array}{rcl}
\textbf{skip} & \triangleq & \top? \\
\textbf{fail} & \triangleq & \bot? \\
\textbf{if}\ \varphi\ \textbf{then}\ \alpha\ \textbf{else}\ \beta & \triangleq & (\varphi?\cdot\alpha) + (\neg\varphi?\cdot\beta) \\
\textbf{if}\ \varphi\ \textbf{then}\ \alpha & \triangleq & (\varphi?\cdot\alpha) + (\neg\varphi?\cdot\textbf{skip}) \\
\textbf{case}\ \varphi_1\ \textbf{then}\ \alpha_1;\ \ldots & \triangleq & (\varphi_1?\cdot\alpha_1) + \ldots + (\varphi_n?\cdot\alpha_n) \\
\textbf{case}\ \varphi_n\ \textbf{then}\ \alpha_n & & \\
\textbf{while}\ \varphi\ \textbf{do}\ \alpha & \triangleq & (\varphi?\cdot\alpha)^*\cdot\neg\varphi? \\
\textbf{repeat}\ \alpha\ \textbf{until}\ \varphi & \triangleq & \alpha\cdot(\neg\varphi?\cdot\alpha)^*\cdot\varphi? \\
\\
\textit{(General \textbf{while} loop)} & & \\
\textbf{while}\ \varphi_1\ \textbf{then}\ \alpha_1 \mid \cdots \mid \varphi_n\ \textbf{then}\ \alpha_n\ \textbf{od} & \triangleq & (\varphi_1?\cdot\alpha_1 + \ldots + \varphi_n?\cdot\alpha_n)^*\cdot \\
 & & \cdot(\neg\varphi_1 \wedge \ldots \neg \wedge \varphi_n)?
\end{array}
$$

2.6.4 Semantics of PDL

Making Kripke structures "multi-modal-prepared"

Definition 2.6.3 (Labeled Kripke structures)**.** Assume a set of labels Σ. A *labeled Kripke structure* is a tuple (W, R, Σ) where

$$R = \bigcup_{l \in \Sigma} R_l$$

is the disjoint union of the relations indexed by the labels of Σ.

for us (at leat now): The labels of Σ can be thought as programs

- Σ: aka alphabet,
- alternative: $R \subseteq W \times \Sigma \times W$
- labels $l, l_1 \ldots$ but also $a, b, \ldots$ or others
- often: $\xrightarrow{a}$, like $w_1 \xrightarrow{a} w_2$ or $s_1 \xrightarrow{a} s_2$

Regular Kripke structures

- "labels" now have "structure"
- remember: regular program syntax
- interpretation of certain programs/labels fixed,
  - $\mathbf{0}$: failing program
  - $\alpha_1 \cdot \alpha_2$: sequential composition
  - $\ldots$
- thus, relations like $\mathbf{0}$, $R_{\alpha_1 \cdot \alpha_2}$, . . . must obey side-conditions

leaving open the interpretation of the "atoms" a, we fix the interpretation/semantics of the constructs of regular programs

Regular Kripke structures

Definition 2.6.4 (Regular Kripke structures)**.** A *regular Kripke structure* is a Kripke structure labeled as follows. For all basic programs $a \in \Pi_0$, choose some relation R_a. For the remaining syntactic constructs (except tests), the corresponding relations are defined inductively as follows.

$$\begin{array}{lcl}
R_{\mathbf{1}} & = & Id \\
R_{\mathbf{0}} & = & \emptyset \\
R_{\alpha_1 \cdot \alpha_2} & = & R_{\alpha_1} \circ R_{\alpha_2} \\
R_{\alpha_1 + \alpha_2} & = & R_{\alpha_1} \cup R_{\alpha_2} \\
R_{\alpha^*} & = & \bigcup_{n \geq 0} R_\alpha^n
\end{array}$$

In the definition, Id represents the identity relation, $\circ$ relational composition, and R^n and the n-fold composition of R.

Kripke *models* and interpreting PDL formulas

Now: add *valutions* $\Rightarrow$ Kripke model

Definition 2.6.5 (Semantics)**.** A PDL formula φ is *true* in the world w of a regular Kripke model M, i.e., we have attached a valuation V also, written $M, w \models \varphi$, if:

$$\begin{array}{lll}
M, w \models p_i & \text{iff} & p_i \in V(w) \text{ for all propositional constants} \\
M, w \not\models \bot & \text{and} & M, w \models \top \\
M, w \models \varphi_1 \rightarrow \varphi_2 & \text{iff} & \text{whenever } M, w \models \varphi_1 \text{ then also } M, w \models \varphi_2 \\
M, w \models [\alpha]\varphi & \text{iff} & M, w' \models \varphi \text{ for all } w' \text{ such that } wR_\alpha w' \\
M, w \models \langle\alpha\rangle\varphi & \text{iff} & M, w' \models \varphi \text{ for some } w' \text{ such that } wR_\alpha w'
\end{array}$$

This part of the semantics should contain no surprises: it's the standard Kripke interpretation in a multi-modal setting. One ingredient is missing, though, that's the semantics for *tests* (coming next).

Semantics (cont'd)

- programs and formulas: mutually dependent
- *omitted* so far: what relationship corresponds to

$$\varphi?$$

- remember the intuitive meaning (semantics) of tests

Test programs

Tests interpreted as subsets of the identity relation.

$$R_{\varphi?} = \{(w, w) \mid w \models \varphi\} \subseteq \mathit{Id} \tag{2.34}$$

Some special cases:

- $R_{\top?} = \mathit{Id}$
- $R_{\bot?} = \emptyset$
- $R_{(\varphi_1 \wedge \varphi_2)?} = \{(w, w) \mid w \models \varphi_1 \text{ and } w \models \varphi_2\}$

- $[\alpha]\varphi$ is like looking into the *future* of the program and then deciding on the action to take. . .

The slides show some special cases of the definition from equation (2.34). One may also compare that with the "exercises" slide from earlier. As discussed earlier, the intuition of "executing" a test in a particular state is that it either succeeds (it terminates properly, by doing nothing) or it "fails" which blocks the program from continuing. In the informal discussion back then concerning proper and non-proper termination, we drew a parallel to raising exceptions in programming languages. That parallel is adequate in particular for deterministic programs; for non-determistic ones it only goes so far. So, in connection with the non-deterministic choice operator, a few more words may be in order. See below, after finishing the discussion of equation (2.34).

Testing $\top$ always, i.e., in each state succeeds. That means that $R_{\top ?}$ is the identity relation itself. It can also be interpreted that executing $\top$? corresponds to executing `skip` (later, in a different context like LTL and programs there, "do-nothing" steps are also called *stuttering*...). Testing for $\perp$ "fail" in each state when executed, which means $R_{\perp}$? is the empty relation. The corresponding program, the dual to `skip`, is also called `fail`. The standard Boolean operators are interpreted as usual. For instance the $\wedge$ as logical conjuction (or as *intersection* of R_{φ_1} and R_{φ_2}, which is the same).

Finally, as the most complex case, the interpretation for $[\alpha]\varphi ?$: what kind of relation does that corresponds to, resp. what does it mean to execute such a test? Well, we know what $[\alpha]\varphi$ means. It specifies the "post-condition" for all executions of α. If that evaluates to true, the test succeeds and the program can proceeds, alternatively, it fails. So, this allows to "look into the future" of an execution, for instance, if one would write $[\alpha]\varphi ? \cdot \alpha$ as regular program. So, the rest of the program is only executed, if it assured that all possible outcomes of α satisfy φ. Note that α in turn may contain further tests. Of course, one can speak about "other" programs than the rest $[\beta]\varphi ? \cdot \alpha$ where $\beta \neq \alpha$, but still the rest α is only executed if the execution of β would satify the given property φ. Referring to future outcomes of programs, be it α or β is a very powerful mechanism.

Earlier, we compared tests to assertions like `assert(b)` in standard programming languages. Those assertions are very much weaker, in particular one cannot use modal operators to speak about complex properties that may involve specifyng the future of program and make the execution of the program depend on that. The propositional assertions are more intended for being checked at run-time, without involving complex properties referring to future continuations (or other complications).

Non-deterministic choice, tests, and exceptions As mentioned, the parallel between `fail`, i.e., $\perp$? and exceptions works convincingly for *deterministic* programs, for non-determistic ones less so. That can be best illustrated, of course, when combining tests with non-deterministic choices. As in regular expressions, the non-deterministic choice operator may be read as "or" (and written + for regular programs, for regular expression | is more common). So, $\alpha_1 + \alpha_2$ may be read as "do α_1 or α_2". All fine and good, but let's combine that with tests, and consider

$$\varphi ? \cdot \alpha_1 + \neg\varphi ? \cdot \alpha_2 \tag{2.35}$$

For the discussion here, the form of the assertion φ does not matter, is it may be a simple predicate (like $x = 0$ resp $x \neq 0$ for the negate case). The semantics starts, make a choice between the left-hand or the right-hand side. The left-hand side will be execute if the test φ? succeeds. Alternatively choose the right hand side, which "then" will be executed if $\neg\varphi$? succeeds, otherwise not.

What is slightly dubious seem the explanation

> "first execute the choice and then check the test; if successful, continue, if non-successful, don't do anything".

It's in particular dubious, if we intuitively think that a failed test amounts to raising an exception. The program from equation (2.35) is to interpreted as "if-then-else": depending on which of the two tests evaluates to true, either the left-hand side or the right-hand side of the conditional is executed. But if we think that *first* a choice is made, and *afterwards*, the corresponding test is executed to see if one can continue, that is a different way of thinking about the program. In particular, when the test is interpreted as potential exception, that's a unplausible illustration. In a conventional program involving choices, the semantics makes a choice, and if that leads to an exception, then that's what happens. Here, the failing alternative is ignored: having a transition corresponding to a non-propertly terminating program is interpreted the same way as not having a transition at all (remember $R_{\bot?} = \emptyset$). It's unplausible to say, a program that throughs an exception (or will through one in all possible futures is the same as having no program at all).

Here, the interpretation is more, that a choice is made selecting among those alternatives that terminate properly, where branches that lead to failures (= non-proper termination) *are ignored* as if there were not even there. This form of choosing among "successful" alternatives while ignoring the ones that fail is called *angelic* choice and the form of non-determinism *angelic nondetermism*. It miracously picks an alternative that will turn out "positively" in that it terminates properly, if such an alternative exists. Note that the choice choses not just decision "right now" as in equation (2.35), which is used for illustration. Also in a station $\alpha_1 + \alpha_2$, where the two branches are not immediately "guarded" by a test, which can be used to make a decision now, the choice pics one of the "successful", i.e., properly terminating outcomes of α_1 or α_2. To have such an angelic interpretation of non-determinism is not part of a standard behavior of a programming language (neither is the dual, choising the worst outcome, which is known as *demonic* choice). These notions appear in connection with how to *interpret* the occurence of non-determistic choices in a program when it comes to verify properties about it. And, as it is, regular programs have an *angelic* intepretation of the choice operator.

Axiomatic System of PDL

Take all tautologies of propositional logic (i.e., the axiom system of PL from some earlier lecture) and add

Axioms:

$$[\alpha](\varphi_1 \to \varphi_2) \to ([\alpha]\varphi_1 \to [\alpha]\varphi_2) \tag{1}$$
$$[\alpha](\varphi_1 \wedge \varphi_2) \leftrightarrow [\alpha]\varphi_1 \wedge [\alpha]\varphi_2 \tag{2}$$
$$[\alpha + \beta]\varphi \leftrightarrow [\alpha]\varphi \wedge [\beta]\varphi \tag{3}$$
$$[\alpha \cdot \beta]\varphi \leftrightarrow [\alpha][\beta]\varphi \tag{4}$$
$$[\varphi?]\psi \leftrightarrow \varphi \to \psi \tag{5}$$
$$\varphi \wedge [\alpha][\alpha^*]\varphi \leftrightarrow [\alpha^*]\varphi \tag{6}$$
$$\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \to [\alpha^*]\varphi \tag{IND}$$

Rules: take the (MP) modus ponens and (G) generalization of modal logic.

Further reading

On dynamic logic, a book nicely written, with examples and easy presentation: David Harel, Dexter Kozen, and Jerzy Tiuryn: [4]. Chap. 3 for beginners, a general introduction to logic concepts. This lecture is based on Chap. 5 (which has some connections with Chap. 4 and is strongly based on mathematical notions which can be reviewed in Chap. 1)

2.6.5 Exercises

The exercises have been placed on a separate sheet.

Exercises: Play with binary relations

- Composition of relations distributes over union of relations.
  $R \circ (\bigcup_i Q_i) = \bigcup_i (R \circ Q_i) \qquad (\bigcup_i Q_i) \circ R = \bigcup_i (Q_i \circ R)$
- $R^* \triangleq I \cup R \cup R \circ R \cup \ldots \cup R^n \cup \ldots \triangleq \bigcup_{n \geq 0} R^n$

Show the following:

1. $R^n \circ R^m = R^{n+m}$ for $n, m \geq 0$
2. $R \circ R^* = R^* \circ R$
3. $R \circ (Q \circ R)^* = (R \circ Q)^* \circ R$
4. $(R \cup Q)^* = (R^* \circ Q)^* \circ Q^*$
5. $R^* = I \cup R \circ R^*$

Exercises: Play with programs in DL

- In DL we say that two programs α and β are equivalent iff they represent the same binary relation $R_\alpha = R = R_\beta$.

Show:

1. Two programs α and β are equivalent iff for some arbitrary propositional constant p the formula $\langle\alpha\rangle p \leftrightarrow \langle\beta\rangle p$.
2. The two programs below are equivalent:

```
while φ1 do                      if φ1 then
    α;                               α;
    while φ2 do β                    while φ1 ∨ φ2 do
                                         if φ2 then β else α
```

Hint: encode them in PDL and use (1) or work only with relations

Exercises: Play with programs in DL

Use a semantic argument to show that the following formula is valid:

$$p \wedge [a^*]((p \to [a]\neg p) \wedge (\neg p \to [a]p)) \leftrightarrow [(a \cdot a)^*]p \wedge [a \cdot (a \cdot a)^*]\neg p$$

What does the formula say (considering a as some atomic programming instruction)?

Bibliography

[1] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

[2] Blackburn, P., de Rijke, M., and Venema, Y. (2001). *Modal Logic*. Cambridge University Press.

[3] Goré, R., Heinle, W., and Heuerding, A. (1997). Relations between propositional normal modal logics: an overview. *Journal of Logic and Computation*, 7(5):649–658.

[4] Harel, D., Kozen, D., and Tiuryn, J. (2000). *Dynamic Logic*. Foundations of Computing. MIT Press.

Index

Σ, 13, 19
X, 15
$\doteq$, 18
I (interpretation), 21
λ-calculus, 15
$\Box\varphi$, 39
$\Diamond\varphi$, 39
$\models$, 4, 37
$[\![\_]\!]$, 39
θ, 16
$[t/x]$, 17
σ (variable assignment), 22

abstract syntax, 7, 19
accessibility relation, 36
algebraic signature, 17
arity, 13, 15, 18
atomic proposition, 6
automaton, 35

Büchi autmaton, 36
binder, 24
bound variable, 17

completeness, 6
concrete syntax, 7, 19
contradiction, 4
correctness
  partial, 51
  total, 51

domain, 21
dynamic logics, 46, 48

equality symbol, 18
exception, 49

finite mapping, 36
first-order logics, 5, 46
first-order model, 21
first-order signature, 4, 17, 19
first-order structure, 21, 22
forcing, 39
formal method, 1
formula, 4, 6
frame validity, 39, 40
free and bound variable occurrence, 24
free variable, 17

generalization (in modal logic), 56
ground term, 15

Hilbert-style proof system, 42
Hoare logic., 51

I/O automaton, 36
interpretation, 5, 21

Java, 48

KeY, 48
Kripke frame, 34, 36, 40
Kripke model, 34
Kripke structure, 33, 34, 52

labelled transition system, 35
logics
  syntax, 4
LTL, 40

many-sorted, 21
Mealy machine, 36
modal logic
  syntax, 34
modal logics, 32
modality, 32
model, 1, 12
model theory, 3
modus ponens, 42, 56
Moore machine, 36
multi-modal logic, 47
multi-modal logics, 33
multi-sorted signature, 17

normal modal logic, 43

ontology, 22

partial correctness, 51
PDL, 48
  syntax, 50
predicate, 17
predicate logic, 17

proof system, 42
proof theory, 3, 12
proposition, 6
  atomic, 6
propositional logic, 4

QTL, 35
quantifier, 17, 24

regular Kripke structure, 52

S4, 40
satisfaction relation, 37
satisfiability, 3
Saul Kripke, 33
scope, 24
semantics, 37
signature, 13, 15
  algebraic, 17
  first order, 17
  first-order, 19
  multi-sorted, 17
  signature, 4
  single-sorted, 17
single-sorted signature, 15, 17
sort, 15, 17
soundness, 6
state, 22
structure
  first-order, 22
substitution, 17, 24
subtitution, 16
syntactic sugar, 42
syntax
  abstract, 7
  concrete, 7

tautology, 4, 11
term, 15
  ground, 15
test, 53
TLA, 48
total correctness, 51
transition system, 36
  labelled, 35
truth, 3
truth table, 9

valid, 11
validity, 3
  fram, 40
  frame, 39
valuation, 22, 36, 53
variable, 5
variable assignment, 22