Logic in Action

–Draft, March 12, 2010–

Johan van Benthem, Hans van Ditmarsch, Jan van Eijck, Jan Jaspars

Contents

Implementations 15-1

Appendices A-1

Chapter 1

What is Logic?

This chapter is a light introduction to the scope and content of this course. It raises many issues in a light manner, and these will then be developed more formally later on.

1.1 Reasoning in Daily Life

Logic can be seen in action all around us:

> In a restaurant, your Father has ordered Fish, your Mother ordered Vegetarian, and you ordered Meat. Out of the kitchen comes some new person carrying the three plates. What will happen?

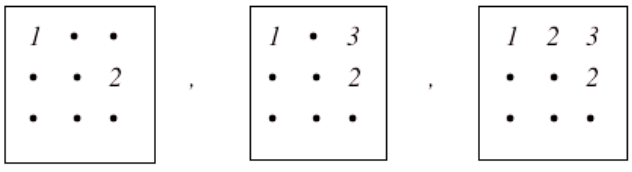
We all know this. The waiter first asks a question, say "Who has the meat?", and puts that plate. Then he asks a second question "Who has the fish?", and puts that plate. And then, without asking further, he puts the remaining plate. What has happened here?

Starting at the end, when the waiter puts the third glass without asking, you see a major logical act 'in broad daylight': the waiter draws a conclusion. The information in the two answers received allows the waiter to infer automatically where the third dish must go. We represent this in an *inference schema* with some self-explanatory notation:

$$F \text{ or } V \text{ or } M, \text{not } M, \text{not } F \Longrightarrow V \tag{1.1}$$

This formal view has many benefits: one schema stands for a wide range of inferences.

The latter often come to the surface especially vividly in puzzles, where we exercise our logical abilities just for the fun of it. Think of successive stages in the solution of a 3×3 Sudoku puzzle, produced by applying the two basic rules that each of the 9 positions must have a digit, but no digit occurs twice on a row or column:

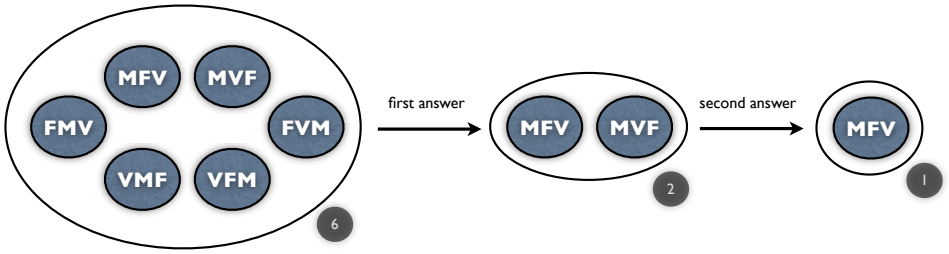


(1.2)

Each successive diagram displays a bit more explicit information about the solution, which is already implicitly determined by the initial placement of the two digits 1, 2. And the driving mechanism for these steps is exactly our Restaurant inference.

But much more information flows in the Restaurant. Before the final inference, the waiter first actively sought enough facts by another typical information-producing act, viz. asking a question. And the answers to his two questions are crucial, too.

The essence of this second process is a form of computation on information states. During a conversation, information states of people — singly, and in groups — change over time, triggered by communicative events. The Restaurant scenario starts with an initial information state consisting of six options, all the ways in which three plates can be distributed over three people $(MFV, MVF, ...)$. The answer to the first question then reduces this to two (the remaining orders FV, VF), and the answer to the second question reduces this to one, zooming in on just the actual situation (for convenience, assume it is MFV). This may be pictured as a diagram ('video') of successive updates:



(1.3)

But there is still more to the scenario, and the essence of information flow. Questions are typically 'social' actions involving more than one person, and indeed, much of our daily reasoning has to do with others, just as basic physics is about interaction between many objects, not just lonely planets. This social aspect of reasoning about, and often also together with, others is high-lighted by another type of restaurant puzzle, that recently circulated on the internet. It is a bit contrived, but it contains some interesting points:

> Three guests are sitting at a table. The waitress asks: "Does everyone want coffee". The first guest says: "I don't know". The second guest now says: "I don't know". Then the third guest says: "No, not everyone wants coffee". The waitress comes back and gives the right people their coffees. How?

Please try to see for yourself how the waitress solves this. Later on in this course, we will give you all the skills needed to deal systematically with examples like this.

While few restaurants will have conversations like the above, knowledge about others is actually quite realistic. When I ask you a normal question, I convey the information that I do not know the answer, and even, that I think that you may know the answer. Such repetitions express mutual expectations about what others know, and it is these, philosophers, economists, and psychologists tell us, that keep human interaction in place. It is our ability to reason in the presence of other reasoning agents that has made us historically so successful in debate, organization, and way back when: planning a mammoth hunt!

1.2 Logic as the Study of Informational Practices

Putting these observations together, you get the task for logic in this course. We need to study informational processes of inference and information update — and while we can start dealing with these for single agents, our theories must also work interactively when many agents exchange information, say, in a conversation or a debate. As we proceed, you will see many further aspects of this program, and you will learn about mathematical models for it, some quite recent, some already very old.

Reasoning as a specialized skill But first we want to remove a possible misunderstanding. While reasoning in daily life and solving practical tasks is important, many logical phenomena become more pronounced when we look at specialized areas, where our skills have been honed to a greater degree. Hunters with their basic needs satisfied become scientists.

To see the power of pure inference unleashed, think of mathematical proofs. Already in Greek Antiquity (and in parallel, in other cultures), logical inference provided a searchlight toward surprising new mathematical facts. In our later chapter on Proof, we will give examples, including the famous Pythagorean proof that $\sqrt{2}$ is not a rational number.The Holy Writ of this tradition are Euclid's *Elements* from around 300 BC with its formal set-up of axioms, definitions, and theorems for geometry.


EVCLIDIS GEOM. LIBER I. 71
PROBL. 14. PROPOS. 46.
A DATA recta linea quadratum describere.
THEOR. 33. PROPOS. 47.
IN rectangulis triangulis, quadratum quod a latere rectum angulum subtendente describitur, æquale est eis, quæ a lateribus rectum angulum continentibus.
SCHOLION.
EX PROCLO.
(1.4)

Indeed, mathematical methods have deeply influenced the development of logic. They did so in two ways. First, mathematical proof is about the purest form of inference that exists, so it is an excellent 'laboratory' for studying inference. But also, mathematics is about the clearest way that we have for modeling phenomena and studying their properties, and logical systems of any kind, even when dealing with daily life, use mathematical techniques.

Combinations of inference with other information sources drive the natural sciences, where experiments provide information that is just as crucial as mathematical proof. Observations about Nature made by scientists involves the same sort of information update

as in our Restaurant example. Seeing new facts removes uncertainty. And the art is to find the right mixtures of new evidence and deduction from what we have seen already.

The same skill actually occurs in other specialized practices. Conan Doyle's famous detective Sherlock Holmes is constantly thinking about what follows from what he has seen already, but he also uses his powers of deduction to pinpoint occasions where he needs new evidence. In a famous story, the dog did not bark at night-time (and so, the intruder must have been known to the dog), but this conclusion also directs attention toward making further observations, needed to see which of the various familiar persons committed the crime.

(1.5)

From crime it is only one step to lawyers and courts. Legal reasoning is another major tradition where logic is much in evidence, and we will return to this later. Many further specialized practices are driven by reasoning: think of philosophy, or theology. But you will have got the point by now.

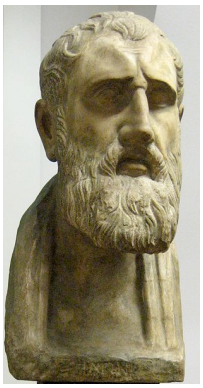
After this brief introduction, let us briefly go back in history, to see where it all started. After that, we give a first survey of some basic logical notions.

1.3 The Origins of Logic

Logic as a systematic discipline dates back two and a half millennia: younger than Mathematics or the Law, but much older than most current academic disciplines, social institutions, or for that matter, religions. Aristotle and the Stoic philosophers formulated explicit

systems of reasoning in Greek Antiquity around 300 BC.

Aristotle appearing on two Greek postal stamps



The early Stoic Zeno of Citium

(1.6)

Independent traditions arose around that time in China and in India, which produced famous figures like the Buddhist logician Dignaga, or Gangesa, and this long tradition lives on in some philosophical schools today. Through translations of Aristotle, logic also reached the Islamic world. The work of the Persian logician Avicenna around 1000 AD was still taught in madrassa's by 1900. All these traditions have their special concerns and features, and there is a growing interest these days in bringing them closer together. We mention this point because the cross-cultural nature of logic is a social asset beyond its scientific agenda.

Mo Zi, founder of Mohism

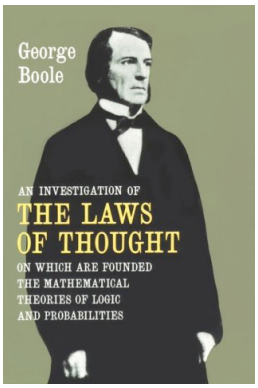
Dignaga, Indian Buddhist Logician

Avicenna, Persian Logician

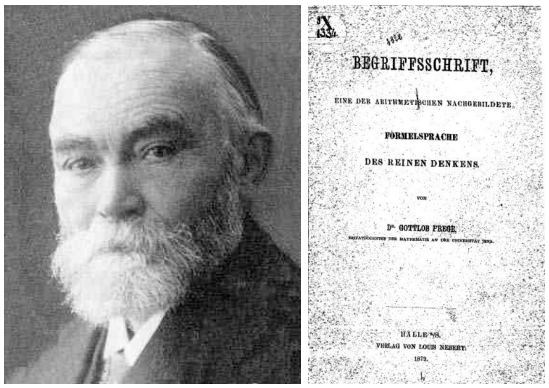
(1.7)

Still, with all due respect for this historical past that is slowly coming to light, it seems fair to say that logic made a truly major leap in the nineteenth century, and the modern logic that you will see in this course derives its basic mind-set largely from the resulting golden age of Boole, Frege, Gödel, and others: a bunch of European university professors, some

quite colourful, some much less so.



George Boole on the cover of
the 'Laws of Thought' (1847),
the book that created propositional logic,
the theme of the next chapter.



Gottlob Frege with on the right
the first page of his 'Begriffsschrift' (1879),
with the system of first-order predicate logic
that can analyze much of mathematics.

(1.8)

Even so, it remains an intriguing and unsolved historical question just how and why logic arose — and we will have more to say on this below. The standard story is that great thinkers like Aristotle suddenly realized that there is structure to the human reasoning that we see all around us. Some patterns are valid and reliable, while others are not. But it has also been suggested that an interest in logic arose out of philosophical, mathematical, juridical, or even political practice. Some 'moves' worked, others did not – and people became curious to see the general reasons why.

1.4 Inference Patterns, Validity, and Invalidity

Of the many informational processes mentioned earlier, we will now concentrate on inference. We will briefly mention other information sources later on in this chapter, and they will get ample attention as this course proceeds.

For a start, consider the following statement from your doctor:

If you take my medication, you will get better.
But you are not taking my medication.

So, you will not get better. (1.9)

Here the word 'so' (or 'therefore', 'thus', etc.) suggests the drawing of a conclusion from two pieces of information: traditionally called the 'premises'. We call this an act of inference. Now, as it happens, this particular inference is not compelling. The conclusion might be false even though the two premises are true. You might get better by taking that greatest medicine of all (but so hard to swallow for modern people): just wait. Relying

on a pattern like this might even be pretty dangerous in some scenarios:

$$\begin{array}{l}\text{If I resist, the enemy will kill me.}\\ \text{But I am not resisting.}\\ \hline \text{So, the enemy will not kill me.}\end{array} \tag{1.10}$$

Now contrast this with another pattern:

$$\begin{array}{l}\text{If you take my medication, you will get better.}\\ \text{But you are not getting better.}\\ \hline \text{So, you have not taken my medication.}\end{array} \tag{1.11}$$

This is valid (listen to your mind — or your heart). There is no way that the two stated premises can be true while the conclusion is false. It is time for a definition. Broadly speaking,

> we call an inference *valid* if there is 'transmission of truth': in every situation where all the premises are true, the conclusion is also true.

Stated differently but equivalently, an inference is valid if it has no 'counter-examples': that is, situations where the premises are all true while the conclusion is false. This is a crucial notion to understand, so we dwell on it a bit longer.

What validity really tells us While this definition makes intuitive sense, it is good to realize that it may be weaker than it looks a first sight. For instance, a valid inference with two premises

$$P_1, P_2, \text{ so } C \tag{1.12}$$

allows many combinations of truth and falsity. If any premise is false, nothing follows about the conclusion. In particular, in the second doctor example, the rule may hold (the first premise is true), but you are getting better (false second premise), and you did take the medication (false conclusion). Of all eight true-false combinations for three sentences, validity rules out 1 (true-true-false)! The most you can say for sure thanks to the validity can be stated in one of two ways:

$$\begin{array}{l}\text{(a) if all premises are true, then the conclusion is true}\\ \text{(b) if the conclusion is false, then at least one premise is false}\end{array} \tag{1.13}$$

The first version is how people often think of logic: adding more things that you have to acccept given what you have accepted already. But there is an equally important use of logic in *refuting* assertions, perhaps made by your opponents. You show that some false consequence follows, and then cast doubt on the original assertion. The second

formulation says exactly how this works. Logical inferences also help us see what things are false — or maybe more satisfyingly, refute someone else. But note the subtlety: a false concluion does not mean that all premises were false, just that at least one is. Detecting this bad apple in a basket may still take further effort.

To help you understand both aspects of validity, consider the tree below: representing a 'complex argument' consisting of individual inferences with capital letters for sentences, premises above the bar, and the conclusion below it. Each inference in the tree is valid:

$$\dfrac{\dfrac{\dfrac{A \quad B \quad C}{A \qquad D} \qquad \dfrac{E}{F} \qquad \dfrac{A}{B}}{E \qquad\qquad F \qquad\qquad B}}{G} \tag{1.14}$$

You are told reliably that sentence A is true and G is false. For which further sentences that occur in the tree can you now determine their truth and falsity? (The answer is that A, B, are true, C, D, E , G are false, while we cannot tell whether F is true or false.)

Inference patterns The next step in the birth of logic was the insight that the validity and invalidity here have to do with abstract patterns, the shapes of the inferences, rather than their specific content. (It is still being debated whether this insight occurred explicitly outside of the Greek tradition.) Clearly, the valid second argument would also be valid in the following concrete form, far removed from doctors and medicine:

$$\begin{array}{l} \text{If the enemy cuts the dikes, Holland will be inundated.} \\ \text{Holland is not inundated.} \\ \hline \text{So, the enemy has not cut the dikes.} \end{array} \tag{1.15}$$

This form has variable parts (we have replaced some sentences by others), but there are also constant parts, whose meaning must stay the same, if the inference is to be valid. For instance, if we also replace the negative word 'not' by the positive word 'indeed', then we get the clearly invalid inference:

$$\begin{array}{l} \text{If the enemy cuts the dikes, Holland will be inundated.} \\ \text{Holland is indeed inundated.} \\ \hline \text{So, the enemy has indeed cut the dikes.} \end{array} \tag{1.16}$$

For counter-examples: the inundation may be due to faulty water management, rain, etc.

To bring out the relevant shared form of inferences, we need a notation for both fixed and variable parts. We do this by using variable letters for expressions that can be replaced by others in their linguistic category, plus special notation (either words, or later

on mostly symbols) for key expressions that determine the inference, often called the logical constants. In this format, our first invalid inference looks like this:

$$\text{if } P \text{ then } C, \text{not } P, \text{so: not } C \tag{1.17}$$

And its valid cousin looked like this:

$$\text{if } P \text{ then } C, \text{not } C, \text{so: not } P \tag{1.18}$$

The virtues of abstract notation are clear. We see the essence that drives the inference. We can see relations between different reasoning patterns. And also, each abstract pattern finitely encodes infinitely many instances that arise by substituting concrete sentences for P and C, so we can see analogies between many different situations.

Logical form and abstraction stages In our definition of validity, we talked about 'different situations' in which the given sentences might be true or false. One way of thinking about this comes from the great Czech logician, mathematician and philosopher of religion Bernard Bolzano (*Wissenschaftslehre*, 1837),

(1.19)

who looked at the 'situations' as all possible instances that arise by substituting concrete expressions for variable parts. Or in the opposite direction, we can see logical form as arising through a process of successive abstraction: the more parts of an inference we make 'variable', the harder it is to remain valid. At final stages, we have a maximally general skeleton of logical key-words like 'not', 'if then' (you will see many others later in this course), plus variable parts (indicated by the letters).

This concrete process of successive abstraction can also work for other expressions. Here is an example to loosen up your ideas about the logical structure in the natural language that we all use. Consider the following ubiquitous inference: what makes it tick?

John is taller than Paul, Paul is taller than Mary. So, John is taller than Mary. (1.20)

If we fix the meaning of all these expressions, the only different 'situations' are varying contexts of people with these names. In all of these, truth of the premises implies that of the conclusion. This suggests a first abstraction step, replacing people by variables:

$$a \text{ is taller than } b, b \text{ is taller than } c. \text{ So, a is taller than } c. \tag{1.21}$$

But thinking about it, we can abstract out even more, and still have a valid inference:

$$a \text{ is } P\text{-er than } b, b \text{ is } P\text{-er than } c. \text{ So, a is } P\text{-er than } c. \tag{1.22}$$

This is still valid for any adjective P, by the meaning of the comparative '-er than'. Natural language is full of function words that can play a role in reasoning: comparatives, prepositions, and so on. If you doubt this, just use Google to find a frequency list for English expressions. You will see how logical expressions and function words are high up on the list.

Actually, many logicians like to say that this last comparative inference is invalid — but what they mean then is something different. Replace the whole predicate 'is taller than' in one step by a variable predicate Q:

$$aQb, bQc. \text{ So }, aQc. \tag{1.23}$$

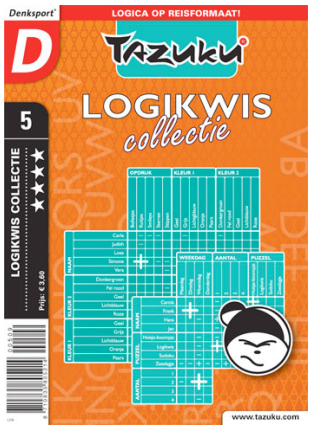
This still more general form is indeed no longer valid. For a counter-example, fill in for Q 'is the mother of'.

1.5 Uses of Inference

We have already seen how logical patterns direct our actions in daily life. The TV has gone dark. If it goes dark, this is due to the apparatus or the remote (or both). But the remote is working, so it must be the apparatus, and we must start repairs there. This pattern involves another logical key-word, the *inclusive disjunction* 'or':

$$A \text{ or } R, \text{ not } R. \text{ So: } A. \tag{1.24}$$

In pure form, we saw this at work in the earlier scenario of solving Sudoku puzzles. It is an interesting fact that people seem to enjoy exercising their logical abilities per sé. In fact, Holland has a great popular magazine with logic puzzles of many sorts:



(1.25)

Logic also helps create new Sudoku puzzles. Start with any complete nine-digit diagram. Then remove any digit that follows from others by a valid inference. Keep doing this until no longer possible. You have now generated a minimal puzzle, and since your steps are hidden, it may take readers quite a while to figure out the unique solution.

Back to practical abilities, cognitive scientists have suggested that the primary use of logic may have been in planning. Clearly, thinking about constraints and consequences of tasks beforehand is an immense evolutionary advantage. Here is a simple illustration.

Planning a party How can we send invitations given the following constraints?

(i) John comes if Mary or Ann comes.
(ii) Ann comes if Mary does not come. (1.26)
(iii) If Ann comes, John does not.

In the next chapter, you will learn simple techniques for solving this: for now, just try!

Legal reasoning We also said that daily skills can be optimized for special purposes. As we said already, inference is crucial to legal reasoning, and so is the earlier-mentioned multi-agent feature that different actors are involved: defendant, lawyer, prosecutor, judge. In this richer courtroom setting, both of the above-noted technical aspects of validity play together.

The prosecutor has to prove that the defendant is guilty (G) on the basis of the available admissible evidence (E), i.e., she has to prove the implication 'if E then G'. But the usual 'presumption of innocence' means that the lawyer has another logical task: viz. making it plausible that G does not follow from E. This does not require her to demonstrate that her client is innocent: she just needs to paint one scenario consistent with the evidence E where G fails, whether it is the actual one or not.

Other logical key-words: quantifiers There are many more logical key-words driving patterns of inference than the ones we have seen so far. Expressions like 'not', 'and', 'or', 'if then' are sentence forming constructions that classify situations as a whole. But other expressions tell us more about the internal structure of these situations, in terms of objects and their properties and relations. Historically, the most important example are quantifiers, expressions of quantity such as 'all', 'some' or 'no'.

Aristotle's syllogisms listed the basic inference patterns with quantifiers, such as

All humans are animals, no animals are mortal. So, no humans are mortal. (1.27)

This is a valid inference. But the following is not valid:

Not all humans are animals, no animals are mortal. So, some humans are mortal. (1.28)

Syllogistic forms were long considered the essence of logical reasoning, and their format has been very influential until the 19th century. Today, they are still popular test cases for psychological experiments about human reasoning.

Quantifiers are essential to understanding both ordinary and scientific discourse. If you unpack standard mathematical assertions, you will find any amount of stacked quantifiers. For instance, think of saying that 7 is a *prime number*. This involves:

$$\begin{array}{l} \textit{All of } 7\text{'s divisors are either equal to 1 or to 7,} \\ \text{where } x \text{ divides } y \text{ if } \textit{for some } z\text{: } x \cdot z = y. \end{array} \tag{1.29}$$

Other examples with many quantifiers occur in Euclid's geometry and spatial reasoning in general.

We will devote even two chapters to the logic of the quantifiers 'all', 'some', given its central importance. Actually, natural language has many further quantifier expressions, such as 'three', 'most', 'few', 'almost all', or 'enough'. This broad repertoire raises many issues of its own about the expressive and communicative function of logic, but we sidestep these here.

Many further logical key-words will emerge further on in this course, including expressions for reasoning about knowledge and action.

Aside: using logic to understand logic We have to start somewhere. The very definition of validity that we gave in the above involved quantifiers over situations, and indeed, some valid reasoning about the quantifiers 'all', 'some' as well as propositional operators was involved in our subsequent discussion. We leave a discussion of this circularity (is it?) to you.

1.6 Richer Informational Practices

We are ready to leave our first exploration of inference. But it may be of interest to point out that the syllogism is just one format. E.g., one recurrent 'syllogism' in the Indian logic tradition runs as follows (simplified a bit):

> I am standing at the foot of the mountain, and cannot inspect directly what is going on there. But I can make observations in my current situation. A useful inference might then work as follows: "I see smoke right here. Seeing smoke here indicates fire on the mountain. So, there is fire on the mountain top."

Compare this with the Aristotelean syllogism, which is about one fixed situation. But here, logical inference typically crosses between situations. Given suitable information channels between situations, observations about one give information about another. Likewise, Chinese examples often involved transfer by analogy from one situation to another. Similar themes have only made Western mainstream logic in recent decades.

Connected with this, on the Indian view, inference is sometimes a sort of last resort, when other informational processes have failed. If I can see for myself what is happening in the relevant situation, that suffices. If I can ask some reliable person who knows, then that suffices as well. But if no direct or indirect observation is possible, we must resort to inference.

Again, we see the logical entanglement of different informational processes. Inference is important, but so is observation, and communication. Later on in this course, we will turn to the study of this wider set of issues, with the techniques that were first developed in the study of inference plus some further ideas from the study of computation.

For now, we just note another historical analogy. Interestingly, similar ideas occur in ancient Chinese logic. In particular, the Mohists had this wonderfully compact statement, that summarizes the perspective of this course:

知問說親 (1.30)

or in English: "Knowledge arises through: questions, demonstration, and experience."

1.7 Logical Systems and the Foundations of Mathematics

Next, let us look at another crucial feature of logic, that makes it a true scientific endeavour in a systematic sense. To many people taking a logic course, there is a startling case of 'self-reflection': turning human reasoning to itself as a subject of investigation. But things go even one step further. Logicians study reasoning practices by developing mathematical models for them – but then, they also make these systems themselves into a new object of investigation.

Logical systems Indeed, Aristotle already formulated explicit logical systems of inference in his Syllogistics, giving all valid rules for syllogistic quantifier patterns. Interestingly, Aristotle also started the study of grammar, language looking at language — and earlier than him, the famous Sanskrit grammarian Panini had used mathematical systems there, creating a system that is still highly sophisticated by modern standards:

(1.31)

This mathematical system building tradition has flourished over time, largely (but not exclusively) in the West. In the nineteenth century, George Boole gave a complete analysis of propositional logic for reasoning with sentential operators like 'not', 'and', 'or', that has become famous as the 'Boolean algebra' that underlies the switching circuits of your computer. Here is what such a system looks like, with variables for propositions that can be true (1) or false (0), and operations $-$ for 'not', $\cdot$ for 'and', and $+$ for 'or' in the sense of binary arithmetic (at this stage, don't even try to read all this in detail! We will discuss a few nice principles from this list in class, with their intuitive meaning):

$$\begin{array}{rclrcl}
x+(y+z) & = & (x+y)+z & x\cdot(y\cdot z) & = & (x\cdot y)\cdot z \\
x+y & = & y+x & x\cdot y & = & y\cdot x \\
x+x & = & x & x\cdot x & = & x \\
x+(y\cdot z) & = & (x+y)\cdot(x+z) & x\cdot(y+z) & = & (x\cdot y)+(x\cdot z) \\
x+(x\cdot y) & = & x & x\cdot(x+y) & = & x \\
-(x+y) & = & -x\cdot -y & -(x\cdot y) & = & -x+-y \\
x+0 & = & x & x\cdot 0 & = & 0 \\
x+1 & = & 1 & x\cdot 1 & = & x \\
x+-x & = & 1 & x\cdot -x & = & 0 \\
\multicolumn{6}{c}{--x \;=\; x}
\end{array} \tag{1.32}$$

All valid principles of propositional reasoning can be derived from this calculus, by purely algebraic manipulations. We will explain such properties later on in this course.

Subsequently, Frege gave formal systems for reasoning with quantifiers in ways that go far beyond Aristotle's Syllogistic. Over time, systems in this line have proved strong enough to formalize most of mathematics, including its foundational *set theory*.

Foundations of mathematics Through this process of scrutiny, mathematical and logical theories themselves become objects of investigation. And then, some startling discoveries were made. For instance, here is the so-called Russell Paradox from the foundations of set theory.

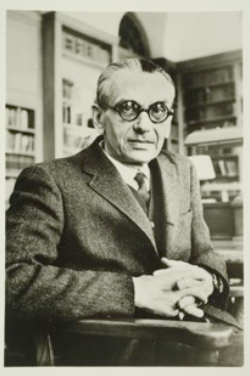
> Some sets contain themselves as a member (e.g., the set of all non-teaspoons). Others do not (for instance, the set of all teaspoons is not itself a teaspoon.) Now consider the set $R = \{x \mid x \notin x\}$ of all sets that do not have themselves as members. It is easy to see that $R \in R$ if and only if $R \notin R$: and that is a contradiction.

Behind this reasoning lies a law of quantifier reasoning that we just write for you here:

$$\neg\exists x\,\forall y(Rxy \leftrightarrow \neg Ryy) \tag{1.33}$$

This sort of quantifier reasoning will be taken up again in several later chapters.

This systems strand in the development of logic led to the so-called foundations of mathematics, which investigates formal properties of mathematical theories, and power and limits of proofs. A famous name here is Kurt Gödel, probably the greatest figure in the history of logic. His incompleteness theorems are fundamental insights into the scope and reliability of mathematics, that got him on the TIME 2001 list of most influential intellectuals of the twentieth century. But in Amsterdam, we also cite our own L.E.J. Brouwer, the father of 'intuitionistic logic', an important program in the foundations of mathematics and computation. These mathematical theoretical aspects of logic belong more properly to an advanced course, but we will give you some feeling for this theme further on in this book.



Kurt Gödel

Brouwer on a Dutch post stamp

(1.34)

1.8 Logic and Other Disciplines

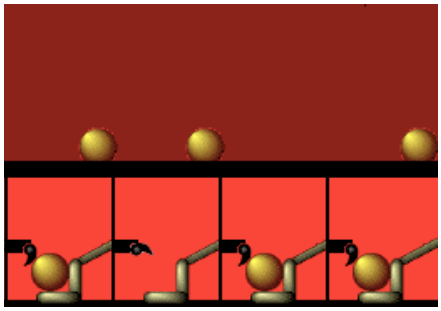
Looking at the list of topics discussed above, you have seen switches from language and conversation to mathematics and computation. Indeed, in a modern university, logic lies at a cross-roads of many academic disciplines. This course will make you acqainted with a number of important important *systems* for doing logic, but it will also draw many connections between logic and related disciplines. We have already given you a taste of what logic has to do with *mathematics*. Mathematics supplies logic with its techniques, but conversely, logic can also be used to analyze the foundations of mathematics. Now we look at a few more important alliances.

Logic, language and philosophy Perhaps the oldest connection of logic is with *philosophy*. Logic has to do with the nature of assertions, meaning, and knowledge, and philosophers have been interested in these topics from the birth of philosophy. Logic can serve as a tool for analyzing philosophical arguments, but it is also used to *create* philosophical systems. Logical forms and calculating with these is a role model for conceptual abstraction. It has even been claimed that logical patterns of the sort sketched here are close to being a 'universal language of thought'.

But it will also be clear that logic has much to do with *linguistics*, since logical patterns arise from abstraction out of the grammar of ordinary language, and indeed, logic and linguistics share a long history from Antiquity through the Middle Ages.

Logic and computation Another long-standing historical theme interleaves logic and *computation*. Since the Middle Ages, people have been fascinated by machines that would automate reasoning, and around 1700, Leibniz

Gottfried Wilhelm von Leibniz



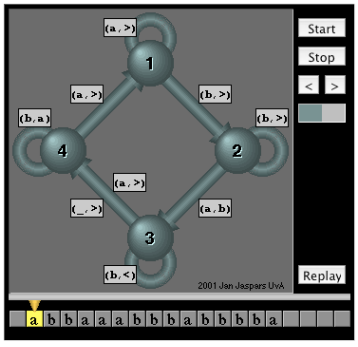
The first binary addition mechanism as described by Leibniz in a paper called 'Mechanica Dyadica' (around 1700)

(1.35)

realized that logical inference may be viewed as a sort of computation, though not with ordinary but with binary numbers. A straight line runs from here to modern computers and computer science, and the seminal work of Turing and others. This is explained really nicely in the book *Denkende Machines* (Amsterdam University Press):

Thinking Machines

Alan Turing



A so-called Turing machine which sorts any sequence of a's and b's in alphabetic order.

(1.36)

Logic and games While mathematics, philosophy, linguistics, and computer science are old neighbours of logic, new interfaces keep emerging. We end with one directed toward the social and behavioural sciences. As we have said before, logic had its origins in a tradition of conversation, debate, and perhaps legal procedure. This brings us back to our earlier theme that much logical behaviour is interactive, crucially involving other persons.

Argumentation itself is a key example. There are different parties playing different roles, and reacting to each other over time. This clearly has the structure of a game. In such a game logical operations like 'or', 'and' and 'not' function as a sort of 'switches', not just in a Boolean computer, but also in discussion. When I defend that 'A or B', then you can hold me to this, and I have to choose eventually which of the two I will defend. Thus, a disjunction offers a choice to its defender — and likewise, a conjunction 'A and B' offers a choice to the attacker: since the defender is committed to both parts. Interesting interactions also arise by means of the third item of Boolean algebra: logical negation. This triggers a role switch: defending 'not A' is attacking 'A', and vice versa. Indeed, being able to 'put yourself in another person's place' has been called the quintessential human cognitive achievement.

In this way, logic comes to describe the structure of rational interaction between conversation partners. Traditions of vigorous regimented logical debating games flourished in the Middle Ages, and they still do in some parts of the world:

(1.37)

Karma Guncho, ten monasteries battle each other
on Buddhist philosophy using logical analysis.

In this game setting, we may call an inference valid if the defender of the conclusion has a 'winning strategy': that is, a rule for playing which will always lead her to win the game against any defender of the premises, whatever that person brings up over time.

But if logic has much to do with games, then it also has links with economic game theory, and not surprisingly, this is another flourishing interface today. We will develop this topic in greater depth in a separate chapter, but now you know why.

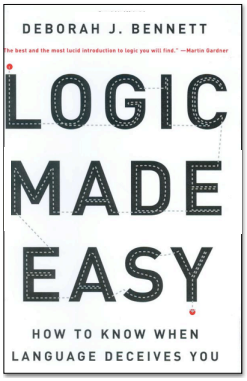
1.9 Logic and the Facts

These are all topics for academic theory. But what about the facts? Logic has traditionally been considered a *normative* subject that sets ideal standards. We describe when inferences are valid or not. And more generally, we describe what information flows in principle through inference, observation, questions, and other informational acts. But there is also the issue of what people actually do in practice. This is the domain of cognitive psychology, and cognitive science in general, now that we can also look inside people's brains as they engage in informational activities. There are many interesting prima facie discrepancies here that have been much studied. In later chapters, we will discuss a

few famous psychological experiments, so that you can make up your own mind on the distance between logical theory and human practice

We will address a number of these comparisons throughout this course. the moment, we just note that, after years of distance, logicians and (neuro-)psychologists are finding common ground these days, because systematic differences between logical theory and human practice are often interesting challenges that call for explanation.

Nowadays, a growing number of logicians has become interested in the boder-line between theory and practice: just like professional ethicists may be interested in the fascinating twilight area of temptation and sin. Some recent textbooks even integrate logical theory with an explanation why it is sometimes so easy to sin against logical theory — and how this can be remedied:



(1.38)

From the back cover: "Logic Made Easy
is indeed one of those rare books that will
actually make you a more logical human being."

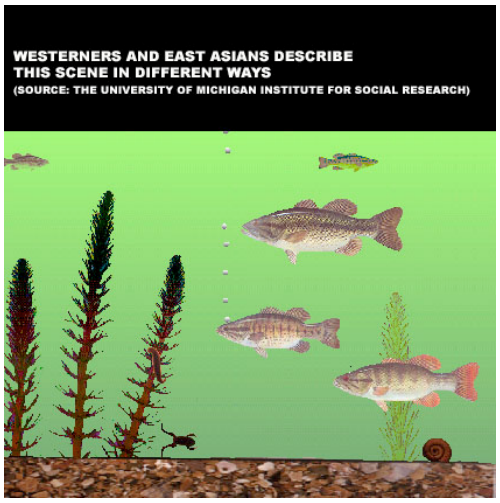
And what about practice? Forget basic theory, forget interdisciplinary connections inside Academia. Can logic also help us become better, more effective reasoners and problem solvers in the world outside? From Antiquity onward, there has been the accompanying topic of *rhetoric*, including rules for successful presentation and debate. In modern times, there has been the emergence of argumentation theory, with an array of methods for analyzing arguments, based on informal schemes of argumentation, and matching procedures. We will pay some attention to practical skills in the course, and at the end, we will also include some links with argumentation theory and 'informal logic'. But to our mind, the frequent opposition of formal and informal logic is fruitless: the two naturally fit together. And while we are at it, let us dispose quickly of a popular misconception about logic, namely, that its 'formality' is an obstacle to true live substance and creativity. To the contrary, formal methods are an invaluable way of enhancing creativity, in the tension between content and form, just as in the arts.

1.10 Our View of Logic

This course presents logic as a theory of key informational tasks performed by cognitive actors that reason, learn, act, and interact with others in an intelligent manner. This agenda may seem broad. But the unity of logic will show in the recurrence of the same methods across our chapters. By the end, you will hopefully have developed a sense of what it means to look at information flow 'from a logical point of view'.

All this puts logic at interfaces with many other disciplines, as we have seen, from the humanities to mathematics, informatics, and the social sciences. We will pursue some of these interfaces in this course, while also keeping this broader picture alive through a number of extra dimensions in our text, including history, empirical aspects, and uses in practice.

Finally, you will excuse us for occasional ad-hoc excursions in this course. We end with one here. Recall our intercultural theme: 'Is logic universal'? Deep differences in styles of thinking have been claimed in the recent literature between Western and Asian cultures.



(1.39)

But it is easy to overstate differences. For instance, when people read contradictions in Indian texts, they thought that these authors claimed that contradictions could be true, about the polar opposite of views in the West. But as Frits Staal convincingly showed in a famous study of Indian mysticism, reciting contradictions was a sort of procedural mantra to switch off rational thinking, precisely because they make no rational sense.

That contradictions may still hurt in unusual circumstances is shown by an experiment that the Dutch logician Henk Barendregt used to perform in lecture halls:

> Hypnotize a volunteer. Now tell him that there is a transparant screen between you and him, that can be manipulated as follows. If you raise your hand, the screen becomes opaque, while dropping the hand makes it transparant again. Now raise your hand. Then drop it a bit later. The volunteer then gets uneasy in his trance, since he 'feels' a real contradiction: he has seen you drop your hand, so the screen is transparant again, but equally well, he cannot have seen you drop your hand, as the screen was opaque.

1.11 Overview of the Course

You have now had a grand opening with many logical themes. The reality of this course is that you need to learn basic technique, just as one cannot become a great musician without basic practice.

This course starts with introductions to three important systems of reasoning: propositional logic (Chapter 2), syllogistics (Chapter 3), and predicate logic (Chapters 4 and 5). Together, these describe situations consisting of objects with a great variety of structure, and in doing so, they cover many basic patterns that are used from natural language to the depths of mathematics.

Next, we move on to the newer challenges in our general agenda of information flow. The first is agents' having information and interacting through questions, answers, and other forms of communication. This social aspect is crucial if you think about how we use language, or how we behave in scientific investigation. We will model observation and reasoning in a multi-agent setting, introducing the logic of knowledge in Chapter 6. To model the dynamic aspect of all this, we turn to the basic logic of action in Chapter 7.

The next group of chapters then develop three logical methods more in detail. Chapter 9 is about precise ways of testing logical validity, that give you a sense of how a significant logical calculus really works. Chapter 10 goes into mathematical proof and its structures. Chapter 11 gives more details on the many relations between logic and computation. And Chapter 8 takes up a more recent theme generalizing proof and computation, the use of games as a model of interaction.

Finally, Chapters 12 and **??** are gateways to further topics and literature on the two main faces of logic in this course: its connections to practice, and its role as a fundamental theory of information.

In all of these chapters you will find links to topics in philosophy, mathematics, linguistics, cognition and computation, and you will discover that logic is a natural 'matchmaker' between these disciplines.

Classical Systems

Chapter 2

Propositional Logic

Overview The most basic logical inferences are about combinations of sentences that describe situations, and what properties these situations have or lack. You could call this reasoning about 'classification', and it is the basis of all reasoning abilities. In this chapter you will be introduced to *propositional logic*, the logical system behind the reasoning with 'not', 'and', 'or', 'if, then' and other basic sentence-combining operators. You will get acquainted with the notions of formula, logical connective, truth, valid consequence, information update, formal proof, and expressive power, while we also present some backgrounds in computation and cognition.

2.1 Motivation — Classification, Consequence and Update

Classification The main ideas of propositional logic go back to Antiquity, but its modern version starts in the nineteenth century, with the work of the British mathematician George Boole (1815–1864). Many of our earlier examples were about combinations of propositions (assertions expressed by whole sentences), indicated by letters p, q, etcetera. A finite number of such propositions generates a finite set of possibilities, depending on which are true and which are false. For instance, with just p, q there are four true/false combinations, that we can write as

$$pq, p\overline{q}, \overline{p}q, \overline{pq} \tag{2.1}$$

where p represents that p is true and $\overline{p}$ that p is false. Thus, we are interested in a basic logic of this sort of classification. (Note that $\overline{p}$ is not meant as a logical proposition here, so that it is different from the negation not-p that occurs in inferences that we will use below. The distinction will only become clear later.)

Drawing consequences Now consider some earlier valid and invalid arguments that were introduced in Chapter 1. For instance,

(a) the argument "from if-p-then-q and not-p to not-q" was invalid,

whereas

(b) the argument "from if-p-then-q, not-q to not-p" was valid.

Our earlier explanation of validity for a logical consequence in Chapter 1 can now be sharpened up. In this setting, it essentially says the following:

> each of the above four combinations that makes the premises true must also make the conclusion true.

You can check whether this holds by considering all cases in the relevant list that satisfy the premises. For instance, in the first case mentioned above,

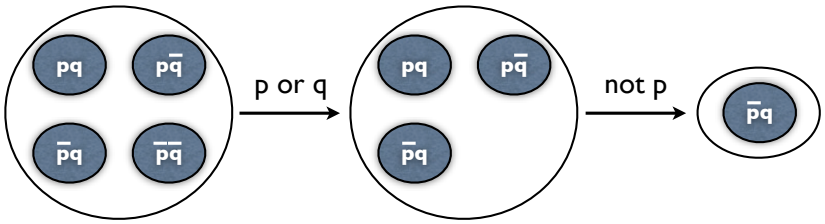
> (a) not-p is true at $\overline{p}q$ and $\overline{pq}$. if-p-then-q holds also in these two situations, since the condition p is not true. So, the first of the two situations, $\overline{p}q$, support the two premises but the conclusion not-q is false in this situation. The argument is therefore invalid!

For the second case we get

> (b) not-q is true at $p\overline{q}$ and $\overline{pq}$. if-p-then-q only holds in the second, so $\overline{pq}$ is the only situation in which all the premises hold. In this situation not-p also holds, and therefore, the argument is valid.

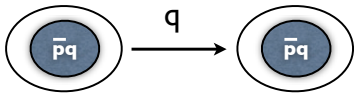
Updating information Propositional logic describes valid (and invalid) inference patterns, but it also has other important uses. In particular, it describes the information flow in earlier examples, that may arise from observation, or just being told facts.

With the set of all combinations present, we have no information about the actual situation. But we may get additional information, ruling out options. To see how, consider a simplified version of the earlier party scenario, with just two possible invitees Mary and John. We write p and q, respectively, for "Mary comes to the party" and "John comes to the party". Suppose that we were first told that at least one of the invitees comes to the party: p-or-q. Out of four possible situations this new information rules out just one, viz. $\overline{pq}$. Next, the we learn not-p. This rules out two more options, and we are left with only the actual situation $\overline{p}q$. Here is a 'video-clip' of the successive information states, that get 'updated' by new information:



(2.2)

Incidentally, you can now also see why the conclusion q is a valid inference. Adding the information that q does not change the final information state, nothing is ruled out:



(2.3)

Exercise 2.1 Consider the case where there are three facts that you are interested in. You wake up, you open your eyes, and you ask yourself three things: "Have I overslept?", "Is it raining?", "Are there traffic jams on the road to work?". To find out about the first question, you have to check your alarm clock, to find out about the second you have to look out of the window, and to find out about the third you have to listen to the traffic info on the radio. We can represent these possible facts with three basic propositions, p, q and r, with p expressing "I have overslept", q expressing "It is raining", and r expressing "There are traffic jams." Suppose you know nothing yet about the truth of your three facts. What is the space of possibilities?

Exercise 2.2 (Continued from previous exercise.) Now you check your alarm clock, and find out that you have not overslept. What happens to your space of possibilities?

Toward a system Once we have a system in place for these tasks, we can do many further things. For instance, instead of asking whether a given inference is valid, we can also just look at given premises, and ask what would be a most informative conclusion. Here is a case that you can think about (it is used as a basic inference step to program computers that perform reasoning automatically):

Exercise 2.3 You are given the information that p-or-q and (not-p)-or-r. What is the strongest valid conclusion you can draw?

A precise system for the above tasks can also be *automated*, and indeed, propositional logic is historically important also for its links with computation and computers. Computers become essential with complex reasoning tasks, that require many steps of inference or update of the above simple kinds, and logical systems are close to automated deduction. But as we shall see later in an outlook section, there is even a sense in which propositional logic is the language of computation, and it is tied up with deep open problems about the nature of computational complexity.

But the start of our story is not in computation, but in natural language. We will identify the basic expressions that we need, and then sharpen them up in a precise notation.

2.2 Notation and the Language of Propositional Logic

Reasoning about situations involves complex sentences with the 'logical connectives' of natural language, such as 'not', 'and', 'or' and 'if .. then'. These are not the only expressions that drive logical reasoning, but they do form the most basic level. We could

stay close to natural language itself to define our system (traditional logicians often did), but it has become clear over time that working with well-chosen notation makes things much clearer, and easier to manipulate. So, just like mathematicians, logicians use formal notations to improve understanding and facilitate computation, and this symbolic practice goes back to Greek Antiquity.

From natural language to logical notation As we have seen in Chapter 1, logical forms lie behind the valid inferences that we see around us in natural language. So we need a good notation to make them come out. For a start, we will use special symbols for the key logical operator words:

Symbol	In natural language	Technical name
$\neg$	not	negation
$\wedge$	and	conjunction
$\vee$	or	disjunction
$\rightarrow$	if ... then	implication
$\leftrightarrow$	if and only if	equivalence

(2.4)

Other notations occur in the literature, too: e.g., & for $\wedge$, and $\equiv$ for $\leftrightarrow$. We write small letters for basic propositions p, q, etcetera. For arbitrary propositions, which may contain connectives as given in the table (2.4), we write small Greek letters φ, ψ, χ etc.

Inclusive and exclusive disjunction The symbol $\vee$ is for inclusive disjunction, as in 'in order to pass the exam question 3 or question 4 must have been answered correctly'. Clearly, you don't want to be penalized if both are correct! This is different from the *exclusive disjunction* (most often written as $\oplus$), as in 'you can go to the movies or to the swimming pool'. It seems very hard to do both at the same time. When we use the word 'disjunction' without further addition we mean the inclusive disjunction.

Now we can write logical forms for given assertions, as 'formulas' with these symbols. Consider a card layer describing the hand of her opponent:

Sentence	"He has an Ace if he does not have a Knight or a Spade"
Logical formula	$\neg(k \vee s) \rightarrow a$

It is useful to see this process of formalization as something that is performed in separate steps, for example, as follows. In cases where you are in doubt about the formalization of a phrase in natural language, you can always decide to 'slow down' to such a stepwise analysis, to find out where the crucial formalization decision is made.

He has an Ace if he does not have a Knight or a Spade,
if (he does not have a Knight or a Spade), then (he has an Ace),
(he does not have a Knight or a Spade) $\rightarrow$ (he has an Ace),
not (he has a Knight or a Spade) $\rightarrow$ (he has an Ace),
$\neg$ (he has a Knight or a Spade) $\rightarrow$ (he has an Ace),
$\neg$ ((he has a Knight) or (he has a Spade)) $\rightarrow$ (he has an Ace),
$\neg$ ((he has a Knight) $\vee$ (he has a Spade)) $\rightarrow$ (he has an Ace),
$\neg(k \vee s) \rightarrow a$

In practice, one often also sees mixed notations where parts of sentences are kept intact, with just logical keywords in formal notation. This is like mathematical expressions that mix symbols with natural language. While this can be very useful (the notation enriches the natural language, and may then be easier to absorb in cognitive practice), you will learn more here by looking at the extreme case where the whole sentence is replaced by a logical form.

Ambiguity The above process taking natural language to logical forms is not a routine matter. There can be quite some slack, with genuine issues of interpretation. In particular, natural language sentences can be *ambiguous*, having different interpretations. For instance, another possible logical form for the card player's assertion is the formula

$$(\neg k \vee s) \rightarrow a \tag{2.5}$$

Check for yourself that this says something different from the above. One virtue of logical notation is that we see such differences at a glance: in this case, by the placement of the brackets, auxiliary devices that do not occur as such in natural language (though it has been claimed that some actual forms of expression do have 'bracketing functions').

Sometimes, the logical form of what is stated is even controversial. According to some people, 'You will get a slap (s), unless you stop whining ($\neg w$)' expresses the implication $w \rightarrow s$. According to others, it expresses the equivalence $w \leftrightarrow s$. Especially, negations in natural language may quickly get hard to grasp. Here is a famous test question in a psychological experiment that many people have difficulty with. What does the stacking of (how many are there?) negations mean in the sentence

"Nothing is too trivial to be ignored?"

Formal language and syntactic trees Logicians think of the preceding notations, not just as a device that can be inserted to make natural language more precise, but as something on its own, namely, an artificial or formal language.

You can think of formulas in such a language as being constructed, starting from basic propositions, often indicated by letters p, q, etcetera, and then applying logical operations, with brackets added to secure unambiguous readability.

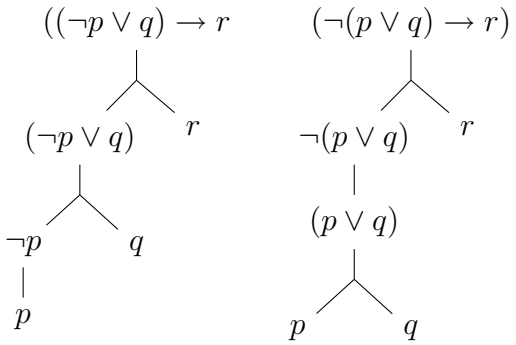
Example 2.4 The formula $((\neg p \vee q) \rightarrow r)$ is created stepwise from proposition letters p, q, r by applying the following construction rules successively:

(a) from p, create $\neg p$,

(b) from $\neg p$ and q, create $(\neg p \vee q)$

(c) from $(\neg p \vee q)$ and r, create $((\neg p \vee q) \rightarrow r)$

This construction may be visualized in *trees* that are completely unambiguous. Here are trees for the preceding example plus a variant that we already discussed above. Mathematically, bracket notation and tree notation are equivalent — but their cognitive appeal differs, and trees are widely popular in mathematics, linguistics, and elsewhere:



This example has prepared us for the formal presentation of the language of propositional logic. There are two ways to go about this, they amount to the same: an inductive definition. (See the appendix on sets.) This is one way:

> Every proposition letter $(p, q, r, ...)$ is a formula. If φ is a formula, then $\neg\varphi$ is also a formula. If φ_1 and φ_2 are formulas, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$ are also formulas. Nothing else is a formula.

We can now clearly recognize that the way we have constructed the formula in the example above is exactly according to this pattern. That is merely a particular instance of the above definition. The definition is formulated in more abstract terms, using the formula variables φ_1 and φ_2. An even more abstract specification, but one that amounts to exactly the same inductive definition, is the so-called BNF specification of the language of propositional logic. BNF stands for 'Backus Naur Form', after the two computer scientists John Backus and Peter Naur.

Definition 2.5 (Language of propositional logic) Let P be a set of proposition letters and let $p \in P$.

$$\varphi \quad ::= \quad p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi)$$

We should read such a definition as follows. In the definition we define objects of the type 'formula in propositional logic', in short: formulas. The definition starts by stating that every atomic proposition is of that type, i.e., is a formula. Then it says that if an object

is of type φ, then $\neg\varphi$ is also of type φ. Note that it does not say that $\neg\varphi$ is the same formula φ! It merely says that both can be called 'formula'. This definition then helps us to construct concrete formulas step by step, as in the previous example.

Backus Naur form is an example of linguistic specification. (In fact, BNF is a computer science re-invention of a way to specify languages that was proposed in 1956 by the linguist Noam Chomsky.)

In practice we often do not write the parentheses, and we only keep them if their removal would make the expression ambiguous, as in $p \vee q \wedge r$. This can mean $((p \vee q) \wedge r)$ but also $(p \vee (q \wedge r))$ and that makes quite a difference. The latter is already true if only p is true, but the former requires r to be true.

Exercise 2.6 Write in propositional logic:

- I will only go to school if I get a cookie now.
- John and Mary are running.
- A foreign national is entitled to social security if he has legal employment or if he has had such less than three years ago, unless he is currently also employed abroad.

Exercise 2.7 Which of the following are formulas in propositional logic:

- $p \rightarrow \neg q$
- $\neg\neg \wedge q \vee p$
- $p\neg q$

Exercise 2.8 Construct trees for the following formulas:

- $(p \wedge q) \rightarrow \neg q$
- $q \wedge r \wedge s \wedge t$ (draw all possible trees).

From the fact that several trees are possible for $q \wedge r \wedge s \wedge t$, we see that this expression can be read in more than one way. Is this ambiguity harmful or not? Why (not)?

A crucial notion: pure syntax Formulas and trees are pure symbolic forms, living at the level of syntax, as yet without concrete meaning. Historically, identifying this separate level of form has been a major abstraction step, that only became fully clear in 19th century mathematics. Most uses of natural language sentences and actual reasoning come with meanings attached, unless very late at parties. Pure syntax has become the basis for many connections between logic, mathematics, and computer science, where symbolic processes play an important role.

Logic, language, computation, and thought The above pictures may remind you of parse trees in grammars for natural languages. Indeed, translations between logical forms and linguistic forms are a key topic at the interface of logic and linguistics, which has also started working extensively with mathematical forms in the 20th century. Connections between logical languages and natural language have become important in Computational Linguistics and Artificial Intelligence, for instance when interfacing humans with computers and symbolic computer languages. In fact, you can view our syntax trees in two ways, corresponding to two major tasks in these areas. 'Top down' they analyze complex expressions into progressively simpler ones: a process of *parsing* given sentences. But 'bottom up' they construct new sentences, a task called language generation.

But also philosophically, the relation between natural and artificial languages has been long under debate. The more abstract level of logical form has been considered more 'universal' as a sort of 'language of thought'. You can even cast the relation as a case of replacement of messy ambiguous natural language forms by clean logical forms for reasoning and perhaps other purposes — which is what the founding fathers of modern logic had in mind, who claimed that natural languages are 'systematically misleading'. But less radically, and perhaps more realistic from an empirical cognitive viewpoint, you can also see the relation as a way of creating *hybrids* of existing and newly designed forms of expression. Compare the way the language of mathematicians consists of natural language plus a growing fund of notations, or the way in which computer science extends our natural repertoire of expression and communication.

2.3 Semantic Situations, Truth Tables, Binary Arithmetic

Differences in formal syntax often correspond to differences in meaning: the above two trees are an example. To explain this in more detail, we now need a semantics that, for a start, relates syntactic objects like formulas to truth and falsity in semantic situations. Thus, formulas acquire meaning in specific settings, and differences in meaning between formulas are often signalled by differences in truth in some situation.

Truth values and valuations for atoms As we said already, each set of proposition letters $p, q, r, \ldots$ generates a set of different *situations*, different ways the actual world might be, or different states that it could be in (all these interpretations make sense in applications). Three proposition letters generate $2^3 = 8$ situations:

$$\{pqr, pq\overline{r}, p\overline{q}r, p\overline{q}\overline{r}, \overline{p}qr, \overline{p}q\overline{r}, \overline{p}\overline{q}r, \overline{p}\overline{q}\overline{r}\} \tag{2.6}$$

Here proposition letters stand for 'atomic propositions', while logical operations form 'molecules'. Of course his is just a manner of speaking, since what counts as 'atomic' in a given application is usually just our decision 'not to look any further inside' the proposition. A convenient mathematical view of situations is as functions from atomic propositions to truth values 1 ('true') and 0 ('false'). For instance, the above situation $p\overline{q}r$ corresponds to the function sending p to 1, q to 0, and r to 1. An alternative notation

for truth values is t and f, but we use numbers for their suggestive analogy with binary arithmetic (the heart of computers). We call these functions V *valuations*; $V(\varphi) = 1$ says that the formula φ is true in the situation (represented by) V, and $V(\varphi) = 0$ says that the formula φ is false in the situation V. For $V(\varphi) = 1$ we also write $V \models \varphi$ and for $V(\varphi) = 0$ we also write $V \not\models \varphi$. One can read $V \models \varphi$ as "V makes true φ", or as "V *satisfies* φ" or "V is a *model* of φ". The notation using $\models$ will reappear in later chapters.

Boolean operations on truth values Any complex sentence constructed from the relevant atomic proposition letters is either true or false in each situation. To see how this works, we first need an account for the meaning of the logical operations. This is achieved by assigning them Boolean operations on the numbers 0, 1, in a way that respects (as far as reasonable) their intuitive usage. For instance, if $V(\varphi) = 0$, then $V(\neg\varphi) = 1$, and vice versa; and if $V(\varphi) = 1$, then $V(\neg\varphi) = 0$, and vice versa. Such relations are easier formatted in a table.

Definition 2.9 (Semantics of propositional logic) A valuation V is a function from proposition letters to truth values 0 and 1. The value or meaning of complex sentences is computed from the value of basic propositions according to the following truth tables.

φ	$\neg\varphi$
0	**1**
1	**0**

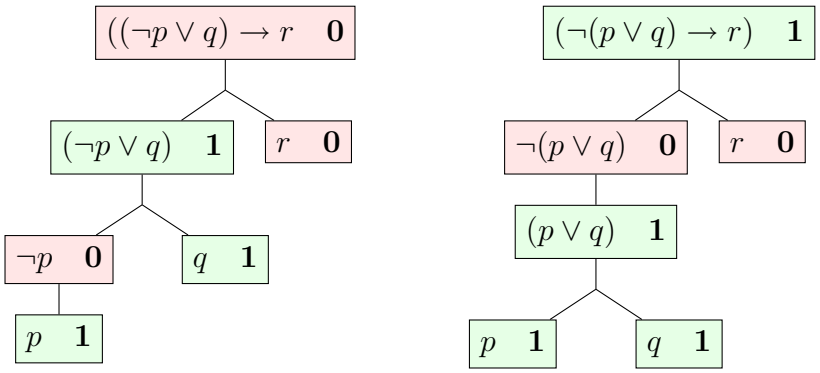
φ	ψ	$\varphi \wedge \psi$	$\varphi \vee \psi$	$\varphi \rightarrow \psi$	$\varphi \leftrightarrow \psi$
0	0	**0**	**0**	**1**	**1**
0	1	**0**	**1**	**1**	**0**
1	0	**0**	**1**	**0**	**0**
1	1	**1**	**1**	**1**	**1**

(2.7)

Bold-face numbers give the truth values for all relevant combinations of argument values: four in the case of connectives with two arguments, two in the case of the connective with one argument, the negation.

Explanation The tables for negation, conjunction, disjunction, and equivalence are usually seen as unproblematic. The table for implication has generated perennial debate, since it does not match the word 'implies' in natural language very well. E.g., does having a false *antecedent* (condition) φ and a true *consequent* ψ really make the implication if-φ-then-ψ true? But we are just doing the best we can in our simple two-valued setting. Thus, you will certainly accept the following assertion as true: 'All numbers greater than 13 are greater than 12'. Put differently, 'if a number n is greater than 13 (p), then n is greater than 12 (q)'. But now, just fill in different numbers n, and you get all combinations in the truth table. For instance, $n = 14$ motivates the truth-value 1 for $p \rightarrow q$ at pq, $n = 13$ motivates 1 for $p \rightarrow q$ at $\overline{p}q$, and $n = 12$ motivates 1 for $p \rightarrow q$ at $\overline{pq}$.

Computing truth tables for complex formulas How exactly can we compute truth values for complex formulas? This is done using our tables by following the construction stages of syntax trees. Here is how this works. Take the valuation V with $V(p) = V(q) = 1, V(r) = 0$ and consider two earlier formulas:



Incidentally, this difference in truth value explains our earlier point that these two variant formulas are different readings of the earlier natural language sentence.

Computing in this manner for all valuations, we can systematically tabulate the truth value behaviour of complex propositional formulas in all relevant situations:

p	q	r	$((\neg p \vee q) \rightarrow r)$	$(\neg(p \vee q) \rightarrow r)$
0	0	0	**0**	**0**
0	0	1	**1**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**1**
1	0	0	**1**	**1**
1	0	1	**1**	**1**
1	1	0	**0**	**1**
1	1	1	**1**	**1**

(2.8)

Paying attention to the proper placement of brackets in formulas, you can compute truth-tables step by step for all situations. As an example we take the second formula from (2.8). First, start with summing up the situations and copy the truth-values under the proposition

letters as has been done in the following table.

p	q	r	$(\neg$	$(p$	$\vee$	$q)$	$\rightarrow$	$r)$
0	0	0	·	0	·	0	·	0
0	0	1	·	0	·	0	·	1
0	1	0	·	0	·	1	·	0
0	1	1	·	0	·	1	·	1
1	0	0	·	1	·	0	·	0
1	0	1	·	1	·	1	·	1
1	1	0	·	1	·	0	·	0
1	1	1	·	1	·	1	·	1

(2.9)

Then start filling in the truth-values for the first possible operator. Here it is the disjunction: it can be computed because the values of its arguments are given (you can also see this from the construction tree). $(p \vee q)$ gets value 0 if and only if both p and q have the value 0. The intermediate result is given in the first table in (2.10). The next steps are the negation and then the conjunction. This gives the following results:

$(\neg$	$(p$	$\vee$	$q)$	$\rightarrow$	$r)$
·	0	0	0	·	0
·	0	0	0	·	1
·	0	1	1	·	0
·	0	1	1	·	1
·	1	1	0	·	0
·	1	1	1	·	1
·	1	1	0	·	0
·	1	1	1	·	1

$(\neg$	$(p$	$\vee$	$q)$	$\rightarrow$	$r)$
1	0	0	0	·	0
1	0	0	0	·	1
0	0	1	1	·	0
0	0	1	1	·	1
0	1	1	0	·	0
0	1	1	1	·	1
0	1	1	0	·	0
0	1	1	1	·	1

$(\neg$	$(p$	$\vee$	$q)$	$\rightarrow$	$r)$
1	0	0	0	**0**	0
1	0	0	0	**1**	1
0	0	1	1	**1**	0
0	0	1	1	**1**	1
0	1	1	0	**1**	0
0	1	1	1	**1**	1
0	1	1	0	**1**	0
0	1	1	1	**1**	1

(2.10)

One does not have to draw three separate tables. All the work can be done in a single table. We just meant to indicate the right order of filling in truth-values.

Exercise 2.10 Construct truth tables for the following formulas:

- $(p \rightarrow q) \vee (q \rightarrow p)$,
- $((p \vee \neg q) \wedge r) \leftrightarrow (\neg(p \wedge r) \vee q)$.

Exercise 2.11 Using truth tables, investigate all formulas that can be readings of

$$\neg p \rightarrow q \vee r$$

(by inserting brackets in appropriate places), and show that they are not equivalent.

If, Only If, If and Only If Here is a useful list of different ways to express implications:

If p then q	$p \to q$
p if q	$q \to p$
p only if q	$p \to q$

The third item on this list may come as a surprise. To see that the third item is correct, reflect on how one can check whether "We will help you only if you help us" is false. This can can happen only if "We help you" is true, but "You help us" is false.

These uses of 'if' and 'only if' explain the use of the common abbreviation 'if and only if' for an equivalence. "We will help you if and only if you help us" states that "you help you" implies "we help you", and vice versa. A common abbreviation for 'if and only of' that we will use occasionally is *iff*.

2.4 Valid Consequence and Consistency

We now define the general notion of valid consequence for propositional logic. It runs over all possible valuations, and as we will see in a moment, we can use truth tables to check given inferences for validity. (In what follows, k can be any number. If it is $k = 0$, then there are no premises.)

Definition 2.12 (Valid consequence) The inference from a finite set of premises

$$\varphi_1, \ldots, \varphi_k$$

to a conclusion ψ is a *valid consequence*, something for which we write

$$\varphi_1, \ldots, \varphi_k \models \psi,$$

if each valuation V with $V(\varphi_1) = \ldots = V(\varphi_k) = 1$ also has $V(\psi) = 1$.

Definition 2.13 (Logical equivalence) If $\varphi \models \psi$ and $\psi \models \varphi$ we say that φ and ψ are *logically equivalent*.

Here it is useful to recall a warning that was already stated in Chapter 1. Do not confuse valid consequence with truth of formulas in a given situation: validity quantifies over truth in many situations, but it has no specific claim about truth or falsity of the premises and conclusions. Indeed, validity rules out surprisingly little in this respect: of all the possible truth/false combinations that might occur for premises and conclusion, it only rules out one case: viz. that all φ_i get value 1, while χ gets value 0.

Another point from Chapter 1 that is worth repeating here is the use of valid inference, not to establish truth, but to achieve a refutation: when the conclusion of a valid consequence is false, at least one of the premises must be false. But logic does not tell us in general which one: some further investigation may be required to find the culprit(s). It

has been claimed by philosophers that this refutational use of logic may be the most important one, since it is the basis of *learning*, where we constantly have to give up current beliefs when they contradict new facts.

Here is a simple example of how truth tables can check for validity:

Example 2.14 (Modus Tollens) The simplest case of refutation depends on the rule of *nodus tollens*:

$$\varphi \to \psi, \neg\psi \models \neg\varphi.$$

Below is given the complete truth table demonstrating its validity:

φ	ψ	$\varphi \to \psi$	$\neg\psi$	$\neg\varphi$	
1	1	**1**	**0**	**0**	
1	0	**0**	**1**	**0**	
0	1	**1**	**0**	**1**	
0	0	**1**	**1**	**1**	!!

(2.11)

Of the four possible relevant situations here, only one satisfies both premises (the valuation on the fourth line), and we can check that there, indeed, the conclusion is true as well. Thus, the inference is valid.

By contrast, when an inference is invalid, there is at least one valuation (i.e., a line in the truth table) where its premises are all true, and the conclusion false. Such situations are called *counter-examples*. The preceding table also gives us a counter-example for the earlier invalid consequence

from $\varphi \to \psi, \neg\varphi$ to $\neg\psi$

namely, the valuation on the third line where $\varphi \to \psi$ and $\neg\varphi$ are true but $\neg\psi$ is false.

Please note that invalidity does not say that all valuations making the premises true make the conclusion false. The latter would express a valid consequence again, this time, the 'refutation' of ψ:

$$\varphi_1, \ldots, \varphi_k \models \neg\psi \tag{2.12}$$

Consistency Finally, here is another important logical notion that gives another perspective on the same issues:

Definition 2.15 (Consistent) A set of formulas X (say, $\varphi_1, \ldots, \varphi_k$) is *consistent* if there is a valuation that makes all formulas in X true.

Instead of 'not consistent' we also say *inconsistent*, which says that there is no valuation where all formulas in the set are true simultaneously.

Consistency is not the same as truth: it does not say that all formulas in X are actually true, but that they could be true in some situation. This suffices for many purposes. In conversation, we often cannot check directly if what people tell us is true, but we may believe them as long as what they say is consistent. Also, as we noted in Chapter 1, a lawyer does not have to prove that her client is innocent, she just has to show that it is consistent with the given evidence that he is innocent.

We can test for consistency in a truth table again, looking for a line making all relevant formulas true. This is like our earlier computations, and indeed, validity and consistency are related. For instance, it follows directly from our definitions that

$$\varphi \models \psi \text{ if and only if } \{\varphi, \neg\psi\} \text{ is not consistent.} \tag{2.13}$$

Tautologies Now we look briefly at the 'laws' of our system:

Definition 2.16 (Tautology) A formula ψ that gets the value 1 in every valuation is called a *tautology*. The notation for tautologies is $\models \psi$.

Many tautologies are well-known as general laws of propositional logic. They can be used to infer quick conclusions or simplify given assertions. Here are some useful tautologies:

$$\begin{array}{ll} \text{Double Negation} & \neg\neg\varphi \leftrightarrow \varphi \\ \text{De Morgan laws} & \neg(\varphi \vee \psi) \leftrightarrow (\neg\varphi \wedge \neg\psi) \\ & \neg(\varphi \wedge \psi) \leftrightarrow (\neg\varphi \vee \neg\psi) \\ \text{Distribution laws} & (\varphi \wedge (\psi \vee \chi)) \leftrightarrow ((\varphi \wedge \psi) \vee (\varphi \wedge \chi)) \\ & (\varphi \vee (\psi \wedge \chi)) \leftrightarrow ((\varphi \vee \psi) \wedge (\varphi \vee \chi)) \end{array} \tag{2.14}$$

Check for yourself that they all get values 1 on all lines of their truth tables.

Tautologies are a special zero-premise case of valid consequences, but via a little trick, they encode all of them. It is easy to see that:

$$\varphi_1, \ldots, \varphi_k \models \psi \text{ if and only if } (\varphi_1 \wedge \ldots \wedge \varphi_k) \rightarrow \psi \text{ is a tautology} \tag{2.15}$$

Exercise 2.17

Show using a truth table that: (a) the inference from $p \rightarrow (q \wedge r)$, $\neg q$ to $\neg p$ is valid and (b) the inference from $p \rightarrow (q \vee r)$, $\neg q$ to $\neg p$ is not valid.

Exercise 2.18 Using a truth table, determine if the two formulas

$$\neg p \rightarrow (q \vee r), \neg q$$

together logically imply

(1) $p \wedge r$.

(2) $p \vee r$.

Display the complete truth table, and use it to justify your answers to (a) and (b).

Exercise 2.19 Check if the following are valid consequences:

(1) $\neg(q \wedge r), q \models \neg r$

(2) $\neg p \vee \neg q \vee r, q \vee r, p \models r$.

Exercise 2.20 Give truth tables for the following formulas:

(1) $(p \vee q) \vee \neg(p \vee (q \wedge r))$

(2) $\neg((\neg p \vee \neg(q \wedge r)) \vee (p \wedge r))$

(3) $(p \to (q \to r)) \to ((p \to q) \to (p \to r))$

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

(5) $((p \leftrightarrow q) \wedge (\neg q \to r)) \leftrightarrow (\neg(p \leftrightarrow r) \to q)$

Exercise 2.21 Which of the following pairs are *logically equivalent*? Confirm your answer using truth tables:

(1) $\varphi \to \psi$ and $\psi \to \varphi$

(2) $\varphi \to \psi$ and $\neg\psi \to \neg\varphi$

(3) $\neg(\varphi \to \psi)$ and $\varphi \vee \neg\psi$

(4) $\neg(\varphi \to \psi)$ and $\varphi \wedge \neg\psi$

(5) $\neg(\varphi \leftrightarrow \psi)$ and $\neg\varphi \leftrightarrow \neg\psi$

(6) $\neg(\varphi \leftrightarrow \psi)$ and $\neg\varphi \leftrightarrow \psi$

(7) $(\varphi \wedge \psi) \leftrightarrow (\varphi \wedge \psi)$ and $\neg\varphi \wedge \neg\psi$

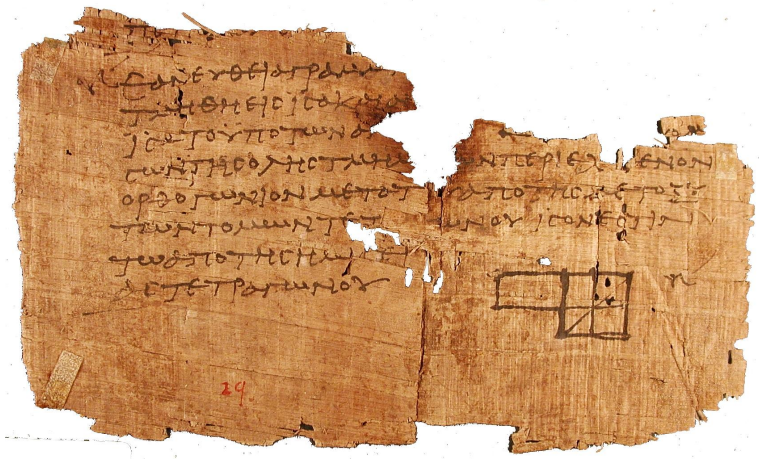
2.5 Proof

Proof: symbolic inference So far we tested inferences for validity with truth tables, staying close to the semantic meaning of the formulas. But a lot of inference happens automatically, by manipulating symbols. People usually do not reason via truth tables. They rather combine many simple proof steps that they already know, without going back to their motivation. Likewise, mathematicians often do formal calculation and proof via symbolic rules (think of your school algebra), and of course, computers have to do proof steps purely symbolically (as long as they have not yet learnt to think, like us, about what their actions might mean).

Logic has many formal calculi that can do proofs, and later on, we will devote a whole chapter to this topic. But in this chapter, we give you a first taste of what it means to do proof steps in a formal calculus. There is a certain pleasure and surprise to symbolic calculation that has to be experienced.

Below, we present an *axiomatic system* organized a bit like the famous geometry book of Euclid's *Elements*, where chains of many proof steps, each one simple by itself, can lead to very surprising theorems. This is only one method used in logic, others include *natural deduction* (used a lot in logical Proof Theory) and *resolution* (used in many automated theorem provers).

History: Axiomatic Proof in Euclid's Elements
A common source for axiomatic proof are the *Elements* of Euclid (see Chapter 1). Euclid was a 4th century B.C.E. Alexandrian mathematician. His textbook gave axiomatic proofs for many well-known theorems of the plane geometry of points and lines, later known as Euclidean geometry. Below is a page from one of the oldest papyrus manuscripts of the *Elements*, from 75-125 A.D. It reads "If a straight line be cut into equal and unequal segments, the rectangle contained by the unequal segments of the whole together with the square on the straight line between the points of section is equal to the square on the half." For more information, see `http://www.math.ubc.ca/~cass/Euclid/papyrus/papyrus.html`. An example of an axiom in Euclidean geometry is "Let the following be postulated: to draw a straight line from any point to any point," or in modern terms: "any two points are connected by a straight line."



Euclid gave axioms and rules for geometry, and only a few principles for the underlying logical reasoning. (The latter include principles like 'Adding equals to equals gives equals', that were called 'Common Notions'.) But now we define a modern axiomatic symbol game for logic:

Definition 2.22 (Axiomatization) A *proof* is a finite sequence of formulas, where each formula is either an *axiom*, or follows from previous formulas in the proof by a deduction *rule*. A formula is a *theorem* if it occurs in a proof, typically as the last formula in the sequence. A set of axioms and rules defines an *axiomatization* for a given logic.
The following is an axiomatization for propositional logic. The axioms are given in schematic form, with the formula variables that we have already seen. It means that we can put any specific formula in the place of these variables:

(1) $(\varphi \to (\psi \to \varphi))$

(2) $((\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi)))$

(3) $((\neg\varphi \to \neg\psi) \to (\psi \to \varphi))$

and there is only one deduction rule, the Modus Ponens that we have already encountered:

- from φ and $(\varphi \to \psi)$, infer ψ.

This axiomatization originates with the Polish logician Jan Łukasiewicz. In this system for propositional logic we may only use implication and negation symbols, and no other logical connectives, such as conjunctions. In the section on expressivity it will be become clear why this is sufficient.

Example 2.23 As an example of an axiomatic proof, we show that $p \to p$ is a theorem. This seems a self-evident tautology semantically, but now, the art is to derive it using only the rules of our game! In what follows we use well-chosen concrete instantiations of axiom schemas. For instance, the first line uses Axiom Schema 1 with the atomic proposition p for the variable φ and $q \to p$ for the variable ψ. And so on:

1. $p \to ((q \to p) \to p)$	Axiom (1)
2. $(p \to ((q \to p) \to p)) \to ((p \to (q \to p)) \to (p \to p))$	Axiom (2)
3. $(p \to (q \to p)) \to (p \to p)$	Modus Ponens, from steps $1, 2$
4. $p \to (q \to p)$	Axiom (1)
5. $p \to p$	Modus Ponens, from steps $3, 4$

History: Jan Łukasiewicz
Jan Łukasiewicz (1878 - 1956) was a Polish philosopher and mathematician from the town currently known as Lviv in the Ukraine. (The town was Austrian at the time of his birth, and Polish for long stretches of its history.) The above axiomatization is one of several due to him. He is also known for results in multi-valued logic, where there are more than two truth values (e.g., true, false and unknown – the sort of thing used in 'fuzzy logic', that drives elevators and washing machines). Łukasiewicz also invented 'Polish notation' for logic, where all operators come first and we do not need parentheses to disambiguate. E.g., Polish notation for $((p \vee q) \wedge r)$ is $\wedge \vee p\ q\ r$.

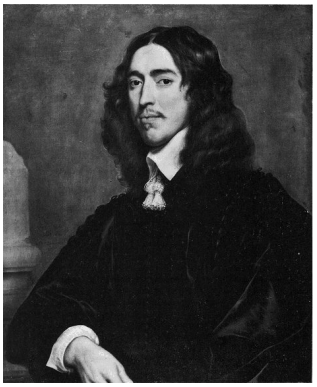
Jan Łukasiewicz

In this course, we do not ask you to find such proofs for yourself. But it is actually an exciting skill to many students, precisely because of its purely symbolic nature. You can taste its flavour by also allowing proofs that have certain assumptions, in addition to instances of axiom schemas. For instance, try to use Modus Ponens and suitable axioms to derive the solution to our earlier Party Puzzle from Chapter 1:

> (i) If Ann comes, John does not: $a \rightarrow \neg j$, (ii) Ann comes if Mary does not come: $\neg m \rightarrow a$, (c) John comes if Mary or Ann comes: here we note the equivalent conjunction John comes if Mary comes and John comes if Ann comes to produce two formulas that fall inside our language: $a \rightarrow j$, $m \rightarrow j$. Now try to give a proof just using the above principle for the solution, deriving successively that $\neg a$, m, j. Have fun!

Of course, there are many ways of solving this, but here we just want you to feel the spirit of a proof game with a fixed repertoire of rules.

Axiomatic proof may be good for mathematics, but it is not widespread in practice. Still, it is interesting to note that, when our great politician Johan de Witt wanted to explain the principles of life insurance to the Dutch nation in 1671, for the first time in mathematical history, his *Waerdye* was not a moralizing glossy brochure, but an axiomatic introduction to probability:



Johan de Witt

WAERDYE
Van
LYF-RENTEN
Naer proportie van
LOS-RENTEN.

IN 'sGRAVEN-HAGE,
By Jacobus Scheltus, Ordinaris Drucker van de Edele Groot Mog. Heeren Staten van Hollandt en Weſt-Vrieſlandt, woonende op het Binnen-Hof,
Anno 1671.

Title Page of *Waerdye*

Digression: other proof formats, and Löb's Paradox There are many other logical proof formats, that do more justice to reasoning practice. In particular, what is typical for proof in conversation or debate, is dynamic 'book-keeping' for assertions that have been made under different assumptions, or for case distinctions, that play a role as the reasoning progresses. We will not pursue this topic here, but we end with one example. It shows how careful book-keeping is crucial to the derivation of a famous paradox.

If you have had enough of formal proof, or for some strange reason, you dislike paradoxes, feel free to skip to the next section.

Short proofs may encode striking arguments. The following variant of the Liar Paradox, invented in 1955 by Martin Löb, proves the surprising assertion that

Every assertion ψ is true!

To show this, first consider the ('self-referent') auxiliary assertion

φ: if this assertion (i.e., φ) is true, then so is ψ.

By basic properties of truth, we then see at once that

$$\varphi \leftrightarrow (\varphi \rightarrow \psi) \tag{2.16}$$

But then consider this proof (with equivalence read as conjunction of two implications):

	Next formula	*Justification of step*	*Assumptions in force*
(i)	Suppose that φ		(2.16), (i)
(ii)	Then $\varphi \rightarrow \psi$	(i) + (2.16)	(2.16), (i)
(iii)	Then ψ	(i) + (ii)	(2.16), (i)
(iv)	$\varphi \rightarrow \psi$	summarizes sub-proof (i)-(iii)	(2.16)
(v)	So: φ	(iv) + (2.16)	(2.16)
(vi)	And also: ψ !!	(iv) + (v)	(2.16)

Thus, we have proved without any assumption, except the definition of φ.

System properties: soundness and completeness If all theorems of an axiomatic system are valid, the system is called *sound* , and conversely, if all valid formulas are provable theorems, the logic is called *complete*. Soundness seems an obvious requirement, as you want to rely totally on your proof procedure. The above system is sound, as you can see by noting that all axioms are tautologies, while Modus Ponens always takes tautologies to tautologies.

Completeness is a different matter, and can be harder to obtain for a given system. (Does Euclid's system of axioms suffice for proving all truths of geometry? The answer took centuries of investigation and reformulation of the system.) The above proof system is indeed complete, and so are the proof systems that we will present in later chapters.

But showing that completeness holds can be hard. The completeness of predicate logic, that we will discuss in later chapters, was one of the first deep results in modern logic, discovered only by Gödel in his 1929 dissertation.

2.6 Information Update

With all this in place, we can now also define our earlier notions of *information* structure and information growth:

> The information content of a formula φ is the set $\mathsf{MOD}(\varphi)$ of its *models*, that is, the valuations that assign the formula φ the truth-value 1.

You can think of this as the range of possible situations that φ still leaves open.

Information update by elimination of possibilities Here is the dynamics that changes such information states:

> An update with new information ψ reduces the current set of models X to the overlap or *intersection* of X and $\mathsf{MOD}(\psi)$. The valuations in X that assign the value 0 to ψ are *eliminated*.

Thus, propositional logic gives an account of basic cognitive dynamics, where information states shrink as new information comes in: growth of knowledge is loss of uncertainty.

We have seen earlier how this worked with simple inferences like

> 'from $p \vee q, \neg p$ to q',

if we assume that the premises update an initial information state of minimal information (maximal uncertainty: all valuations still present).

As a second example, we return to our earlier question

> What is best concluded from $p \vee q, \neg p \vee r$?

Here are the update stages:

initial state	$\{pqr, pq\overline{r}, p\overline{q}r, p\overline{qr}, \overline{p}qr, \overline{p}q\overline{r}, \overline{pq}r, \overline{pqr}\}$
update with $p \vee q$	$\{pqr, pq\overline{r}, p\overline{q}r, p\overline{qr}, \overline{p}qr, \overline{p}q\overline{r}\}$
update with $\neg p \vee r$	$\{pqr, p\overline{q}r, \overline{p}qr, \overline{p}q\overline{r}\}$

(2.17)

We can conclude whatever is true in all of the remaining four states. One valid conclusion is the inclusive disjunction $q \vee r$, and this is indeed the one used in the so-called *resolution rule* of many automated reasoning systems. But actually, the two premises are stronger

than the inference $q \vee r$. The situation $pq\overline{r}$ is not among the ones that are left after the updates in (2.17), but $q \vee r$ is obviously true in this situation. One way of really getting all content of the premises is the valid conjunction:

$$(p \vee q) \wedge (\neg p \vee r) \tag{2.18}$$

In practice, we usually want a more convenient description of such a complex formula, and later on we will look at some principles of Boolean Algebra that can help do this.

Planning Other information scenarios arise in planning problems. You want to throw a party, respecting people's incompatibilities. You know that:

(a) John comes if Mary or Ann comes.

(b) Ann comes if Mary does not come.

(c) If Ann comes, John does not.

Can you invite people under these constraints? You can use valid inference (see below), but a sure way is computing information updates from an initial state of no information:

$$\{ma\rfloor, ma\bar{\rfloor}, m\bar{a}\rfloor, m\bar{a}\bar{\rfloor}, \overline{m}a\rfloor, \overline{m}a\bar{\rfloor}, \overline{m}\bar{a}\rfloor, \overline{m}\bar{a}\bar{\rfloor}\} \tag{2.19}$$

Now the three given premises update this initial information state, by removing options incompatible with them. In successive steps, (a), (b), (c) give the following reductions:

$$\begin{array}{lll} \text{(a)} & (m \vee a) \rightarrow \rfloor & \{ma\rfloor, m\bar{a}\rfloor, \overline{m}a\rfloor, \overline{m}\bar{a}\rfloor, \overline{m}\bar{a}\bar{\rfloor}\} \\ \text{(b)} & \neg m \rightarrow a & \{ma\rfloor, m\bar{a}\rfloor, \overline{m}a\rfloor\} \\ \text{(c)} & a \rightarrow \neg\rfloor & \{m\bar{a}\rfloor\} \end{array} \tag{2.20}$$

Incidentally, this is a unique solution for the stated constraints – but this need not at all be the case in general: there could be none, or more than one option remaining, too.

Games as information processing Our update process describes the information flow in games like *Master Mind*, where players have to guess the correct position of some hidden coloured pegs. In each round, she can make a guess, that gets evaluated by black marks for colours in correct positions, and white marks for colours that do occur, but placed in wrong positions.

For instance, let there be four possible colours red, white, blue, orange, and three positions, with a hidden correct sequence red-white-blue. Here is a table for a possible run of the game, indicating the information game in successive answers:

guess	answer	possibilities remaining
START		24
red, orange, white	•○	6
white, orange, blue	•○	2
blue, orange, red	○○	1

You will find it useful to do the updates, and see why the given numbers are correct.

Master Mind is not really interactive (a machine could provide the answers to your guesses), though interactive variants are used right now in psychological experiments about reasoning with different agents. Information update with different agents, and more realistic games will be studied in Chapters 5, 10. Elimination of possibilities is still important there, so what you learnt here has a broad thrust.

2.7 Expressive Power

A logical language is not just an auxiliary tool for studying inferences and updates. It is also a language that can be used for the common things we have languages for: stating truths (and lies) about situations, communicating important facts to others, and so on. In this light, a very fundamental issue about a logical language is its *expressive power*. What can we say with it?

For a start, propositional logic may look very poor. We can combine sentences, but we cannot look 'inside' them: "Horatio Nelson died at Trafalgar" is just p. But the real issue how well it does within its own compass. And then we find a pleasant surprise:

Propositional logic is quite expressive! In total, there are sixteen possible Boolean operations with two arguments (count options in the truth table) many more than we had in our language. Some of these correspond to serious expressions in natural language. In particular, the *exclusive disjunction* $\varphi \oplus \psi$ corresponds to the natural language phrasing 'either-φ-or-ψ'. It has the following truth table, compare it to the one for disjunction $\vee$:

φ	ψ	$\varphi \oplus \psi$	$\varphi \vee \psi$
0	0	**0**	**0**
0	1	**1**	**1**
1	0	**1**	**1**
1	1	**0**	**1**

Now note that we could get the same truth table by *defining* exclusive disjunction $\varphi \oplus \psi$ in terms of notions that we already had:

$$(\varphi \vee \psi) \wedge \neg(\varphi \wedge \psi) \text{ or, alternatively } \neg(\varphi \leftrightarrow \psi) \tag{2.21}$$

More generally, it is not hard to prove that

> All sixteen possible binary propositional operations are *definable*
> in terms of just the three operations $\neg$, $\wedge$ and $\vee$.

As an illustration,

the implication $\varphi \rightarrow \psi$ has the same truth table as $\neg\varphi \vee \psi$ and as $\neg(\varphi \wedge \neg\psi)$.

In fact, even $\neg, \wedge$ alone suffice for defining all possible operations, and also $\neg, \vee$ alone, and $\neg, \rightarrow$ alone. As you will recall, the latter fact was used in the axiomatization of propositional logic in the section on proof.

Exercise 2.24 Define all connectives in terms of $\neg$ and $\wedge$.

Exercise 2.25 Define all connectives in terms of $\neg$ and $\rightarrow$.

Indeed, there is even one single operation that can define all propositional operations, the *Sheffer stroke*

$$\varphi \mid \psi,$$

defined as $\neg\varphi \vee \neg\psi$.

Now you know how expressive our language is on its own turf: it can express anything we want to say about combination of two-valued propositions.

This is just one of many interesting features of our language. Here is another, that we state without giving details. Every propositional logical formula, no matter how complex, is equivalent to a conjunction of disjunctions of proposition letters or their negations. This is called the 'conjunctive normal form' (there is also a disjunctive normal form). For instance, the conjunctive normal form for the earlier exclusive disjunction is

$$(\varphi \vee \psi) \wedge (\neg\varphi \wedge \neg\psi). \tag{2.22}$$

But of course, as we said, there are also many things that we cannot express in propositional logic. The following chapters are about more expressive languages, such as predicate logic or epistemic logic. Even so, an important thing to keep in mind is a *Balance*. In logic as in science, the art is to stay as simple as possible: 'Small is Beautiful'. A poor language may have special properties that make it elegant or useful. Propositional logic is very successful in bringing out basic reasoning patterns, and moreover, its very poverty leads to elegant and simple semantics and proof methods. In richer systems, the latter become more baroque, and sometimes essentially more complex.

This completes the standard part of this chapter. Next comes a sequence of special topics that will help you see where propositional logic lies in a larger scientific world.

2.8 Outlook — Logic, Mathematics, Computation

Studying logical phenomena via mathematical systems has proved a powerful method historically. Thinking about our language of logical forms yields general insights into expressive power, as we have just learnt. But also, thinking about a system of all validities per se yields new insights that can be used in many settings. Here are some examples:

Boolean algebra The system of laws shows many beautiful regularities. For instance, De Morgan and Distribution laws came in pairs with conjunction and disjunction interchanged. This 'duality' is general, and it reflects the close analogy between propositional logic and binary arithmetic mentioned in Chapter 1. In particular, the truth tables are just laws of binary arithmetic when we read:

> $\vee$ as the maximum of two numbers, $\wedge$ as the minimum,
> and $\neg$ as flipping 0 and 1.

Suppressing details, distribution for conjunction over disjunction then matches the arithmetical distribution law $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$. But binary arithmetic is more symmetric: it also validates another distribution law $x + (y \cdot z) = (x + y) \cdot (x + z)$ that does not hold for numbers in general. We will pursue such connections between logic and computation in more detail in later chapters.

Abstraction and application Logical systems originally arose out of concrete practice. But conversely, once we have such abstract systems, new concrete interpretations may be found. Boolean algebra is an example. It describes a whole range of phenomena: propositional reasoning, binary arithmetic, reasoning with sets (where $\neg$ is complement, $\wedge$ intersection, and $\vee$ union), and even electrical switching circuits where conjunction is serial composition of networks, and disjunction is *parallel composition*. Thus, one and the same formula says lots of things!

For instance, consider one single abstract principle, the Boolean law of 'Absorption':

$$\varphi \leftrightarrow (\varphi \wedge (\varphi \vee \psi)) \tag{2.23}$$

This is a tautology for propositional reasoning that helps remove redundancies from discourse. Next, in binary arithmetic, it expresses a valid equation

$$x = x \min (x \max y) \tag{2.24}$$

about computing with minima and maxima. In set theory, Absorption is the valid principle

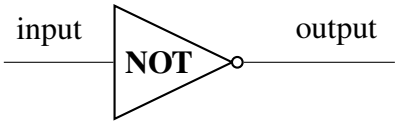
$$X = X \cap (X \cup Y) \tag{2.25}$$

which says that the intersection ('overlap') of the set X and the union of X and Y is the same as the set X itself.

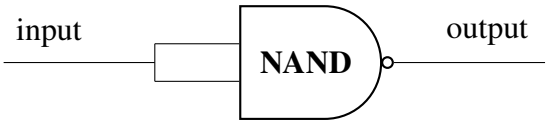
In a similar way, propositional logic plays a role in the design of logical (electronic) circuits. A NAND gate is an electronic circuit that behaves like the Sheffer stroke. The gate has two inputs and an output. If both inputs are 1 (carry a high voltage) then the output is low (carries a low voltage). For all other input combinations (high and low, low and high, low and low) the output is high. Since any propositional connective can be defined with just the Sheffer stroke, any desirable logical circuit can be built from a combination of NAND gates. Here is how negation is defined with the Sheffer stroke:

$$\varphi \mid \varphi.$$

The same principle can be used to build a NOT gate from a NAND gate:



NOT gate



NOT gate implemented with NAND gate

Thus we see that boolean algebra underlies real Boolean circuits in computers, with details that you can find in many sources, also on the internet. A nice historical story of how this came to be is in the book *Denkende Machines* that was mentioned earlier.

Soundness and completeness The same general properties that we stated for our proof system also make sense here. We have already printed a complete set of laws for Boolean algebra in Chapter 1. It is easy to see that these correspond to valid tautologies, when read as equivalences between propositional formulas. Thus we have soundness: the calculus proves only valid principles. Conversely, Boolean algebra is also complete, and any valid equation can be derived from it by ordinary algebraic manipulations.

Computational complexity Propositional logic is tied up with computation in many ways, as we have seen in this chapter. In particular, truth tables make testing for logical validity a simple procedure that can be done mechanically. And indeed, there exist computer programs for all of the tasks in this chapter. Within the compass of our simple language, this realizes the historical project of Leibniz's 'Calculus Ratiocinator' around 1700, based on the idea that deduction is also computation. Still, all this is computability in principle, and things are quite delicate.

A mechanical method with simple steps can still be highly complex when very many of these steps must be made. Consider truth tables. Computing truth values on a single line for a given formula goes fast: we write truth values in the construction tree, and the number of time steps required for this is 'linear': of the same order as the number of symbols in the formula. But now consider the whole table. With n atomic propositions we need 2^n lines, leading to *exponential growth* in the size of the input. This quickly outgrows the powers of even the fastest current computers. Therefore, smarter methods have been investigated, cutting down on the number of steps needed to test for validity — such as the semantic tableaus that you will see in Chapter 7. But it was always found that, in the worst case with difficult input formulas, these still require exponential time.

This is no coincidence. The exact computational complexity of validity in propositional logic is unknown: there may still be a faster method than existing ones that would work with a polynomial bound on processing time, though most experts doubt this. Determining this exact complexity is the essence of the famous

'$\mathbf{P} = \mathbf{NP}$ Problem' ,

that occurs on the famous 2000 Millennium List of open problems in mathematics posed by the Cray Institute.

This problem is urgent since it has been shown that many basic computational tasks reduce to solving problems of validity and consistency in propositional logic. Thus, on its two-thousandth anniversary, propositional logic still poses deep problems.

Higher expressive power and undecidability Whether highly complex or not, the problem of testing for validity in propositional logic is *decidable*: there exists a mechanical method that computes the answer, at least in principle. Thus it may seem that computers can always do the job of logicians. But things change when we move to logics with higher expressive power, such as the predicate logic of Chapter 4 with quantifiers 'all', and 'some'. It is known from the work of Gödel, Turing, and others in the 1930s that there is no mechanical method at all for testing validity of predicate-logical inferences: these major systems pay a price for their greater expressive power: they are *undecidable*.

2.9 Outlook — Logic and Practice

The art of modelling Propositional logic is a formal system that arose as an abstraction out of practice. To apply it in new concrete settings, you need additional 'modelling skills'. For instance, we have already observed that you need to translate from natural language sentences to logical forms to get at the essence of an inference. This often takes practice. The same is true for applying propositional reasoning to the puzzles that we mentioned above. There is also a whole literature on using propositional logic in legal reasoning, where again the structure of arguments has to be 'mined' from the linguistic formulations. Likewise, propositional logic has been applied to a wide variety of computational tasks, from Boolean circuits in your computer to complex train movements at the shunting yards of the Dutch Railways.

Such applications are not routine, and require creative skills.

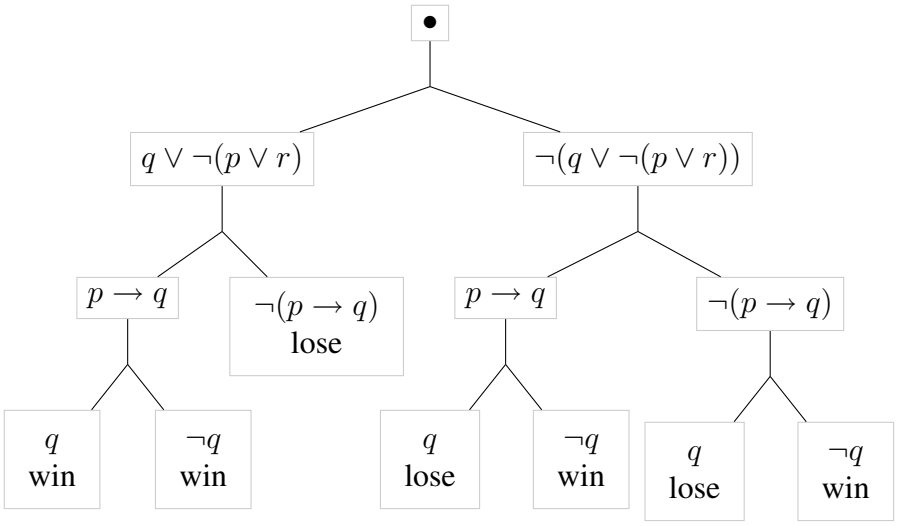
Improving practice Training in propositional logic is used to improve practical skills. This is an old tradition. Legend has it that medieval logic exams checked students' real-time skills as follows:

> *Obligatio Game* A finite number of rounds is chosen, the severity of the exam. The teacher gives the student successive assertions $\varphi_1, \ldots, \varphi_n$ that she has to 'accept' or 'reject' as they are put forward. In the former case, φ_i is added to the students stock of commitments — in the latter, the negation $\neg\varphi_i$ is added. The student passes if she maintains consistency throughout.

Suppose that a student is exposed to the following three statements:

$$(1)\ \ q \vee \neg(p \vee r), (2)\ \ p \rightarrow q, (3)\ \ q. \tag{2.26}$$

Here is one possible run. If you say YES to (1), you must say YES to (2), since it follows but then you can say either YES or NO to (3), since it is independent. Next, if you say NO to (1), you can say either YES or NO to (2), but then, in both cases, you must say NO to (3), as it follows from the negation of (1). The whole picture is:



This may be viewed as a *game tree* with all possible plays including the winning branches. (A complete tree would include Teacher's choices of the next assertion from some given set — possibly influenced by what Student has answered so far.) Either way, the tree will show that, as is only fair on exams, the student has a winning strategy for this game of consistency management. The logical reason is this:

> Any consistent set of assertions can always be consistently expanded with at least one of the propositions φ, $\neg\varphi$.

The winning strategy based on this seems to require consistency checking at each stage, a hard computational problem. A simpler strategy for the student is this:

> choose one model beforehand (say, a valuation making each atom true), and evaluate each incoming assertion there, giving the obvious answers.

Logic puzzles Propositional logic has generated many puzzles. The next exercise has one from Raymond Smullyan's *The Lady or the Tiger?*, Penguin Books, 1982.

Exercise 2.26 Consider these two room signs:

- A – In this room there is a lady, and in the other one there is a tiger.
- B – In one of these rooms, there is a lady, and in one of them there is a tiger"

One of these signs is true, the other false. Behind which door is the lady?

The recreational aspect of propositional logic is also found in Sudoku Puzzles, while more sophisticated examples will return with the systems of our next chapters.

2.10 Outlook — Logic and Cognition

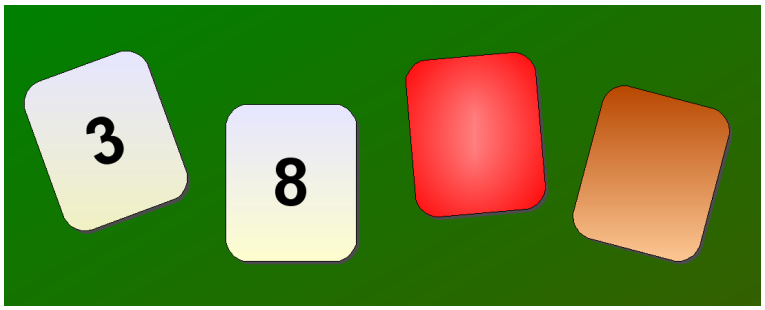
But how do logical systems relate to our daily practice where we reason and try to make sense without thinking about it? One interface has occurred a number of times now:

Logic and linguistics Natural languages have a much richer repertoire of meanings than the formal language of propositional logic. For instance, the expression *and* also has frequent non-Boolean readings. "John and Mary quarrelled" does not mean that "John quarrelled and Mary quarrelled", and a new semantics is needed for collective predicates where agents do things together. Likewise, conditional expressions like *if ... then* do not behave exactly like the truth-table conditional. In particular, a false antecedent does not necessarily make them true: "If I were rich, I would be generous" does not follow from my not being rich. But all this does not mean that logical methods do not apply. The richer structure of natural language has been an inexhaustible source of new logical theory. For instance, propositional logic has been generalized to work with more than two truth values to model vague or indeterminate uses of language, and the study of various sorts of conditional expression has in fact become a flourishing subdiscipline where logicians and linguists work together.

Logic and cognitive psychology The relation between logic and psychology has been somewhat stormier. It has been claimed that everyday reasoning is highly non-logical. Here is a famous example.

The Wason selection task is a logic puzzle that states the following question:

> You are shown a set of four cards placed on a table each of which has a number on one side and a colored patch on the other side. The visible faces of the cards show 3, 8, red and brown. Which card(s) should you turn over in order to test the truth of the proposition that if a card shows an even number on one face, then its opposite face is red?



The Wason selection task

Here is the correct response according to the logic of this chapter:

> turn the cards showing 8 and brown, but no other card.

The reason is this: to test the implication $even \rightarrow red$, we clearly need to check the card with the even number 8, but also should not forget the refutation case discussed several times before: if the card is not red, we need to make sure that it did not have an even number. Now the realities in the experiment, repeated over many decades:

most people either (a) turn the 8 only, or (b) they turn the 8 and the red card.

Psychologists have suggested many explanations, including a 'confirmation bias' (refutation comes less natural to us) and an 'association bias' (red is mentioned so we check it). This seems to suggest that real reasoning is very different from what logic says.

However, the selection task tends to produce the correct logical response when presented in more concrete contexts that the experimental subjects are familiar with. For example, if the rule is 'If you are drinking alcohol, then you must be over 18', and the cards have an age on one side and a beverage on the other, e.g., '17', 'beer', '22', 'coke', most people have no difficulty in selecting the correct cards ('17' and 'beer'). Psychologists have used this as another argument against logic: the two settings have the same logical form, but very different behaviour results from familiarity effects.

More information on this famous experiment can be found on the webpage `http://en.wikipedia.org/wiki/Wason_selection_task`.

Frankly, all this polemics is not interesting. Clearly, people are not irrational, and if they ignored logic all the time, extracting the wrong information from the data at their disposal, it is hard to see how our species could survive. What seems to be the case is rather an issue of *representation* of reasoning tasks, and additional principles that play a role there. Moreover, as we shall see in Chapter 12, the variation in outcomes fits with a conspicuous trend in modern logic, namely, the study of *task-dependent* forms of inference, whose rules may differ from the strict standards set in this chapter. These include more heuristic 'default rules' that are not valid in our strict sense, but that can be used until some problem arises that requires a revision of what we concluded so far.

But let us give the last word to George Boole, often considered the father of the purely mathematical approach to (propositional) logic. The title of his great work "The Laws of Thought" would seem to sit uneasily with a diehard normative mathematical perspective. But toward the end of the book, Boole remarks that he is serious about the title: the laws of propositional logic describe essential human thought. but he also acknowledges that human reasoning often deviates from this canon. What that means is, he says, that there are *further laws* of human thought that still need to be discovered. That is what the modern interface of logic and cognitive science is about.

Further Exercises

Exercise 2.27 Prove that all propositional connectives are definable with the 'Sheffer stroke'

$$\varphi \mid \psi,$$

defined by $\neg\varphi \vee \neg\psi$.

Exercise 2.28 In how many ways can you win the following *obligatio* game?

$$(1)\ \ (p \to q) \vee (r \to q), (2)\ \ \neg((p \wedge r) \to q), (3)\ \ q.$$

Exercise 2.29 Consider the following formula:

$$(p \wedge (q \to r)) \to \neg(\neg p \vee ((\neg q \to q) \wedge (r \to \neg r))).$$

The logical symbols in this formula are all the symbols except parentheses and propositional variables. As you can see, the formula as 11 logical symbols. Answer the following questions:

(1) How many truth value entries does the truth table for this formula have. How does that number depend on the number of logical symbols?

(2) The truth table for a formula with 3 propositional variables has $2^3 = 8$ rows. How many entries in the truth table for such a formula do you have to compute (in the worst case) in order to find out if the formula is valid or not, given that you know that the formula has n logical symbols?

Summary *You have now seen your first logical system, and know how to reason in an exact mathematical manner with propositions. In particular, you have learnt these skills:*

- *read and write propositional formulas,*
- *translate simple natural language sentences into formulas,*
- *compute truth tables for various purposes,*
- *test validity of inferences,*
- *compute updates of information states,*
- *do some very simple formal proofs.*

In addition, you now have a working knowledge of the notions of

> *syntax, semantics, valuation, truth, valid consequence, tautology, // consistency, axiomatic proof, expressive power, logical system.*

Finally, you have acquired some understanding of connections between propositional logic and mathematical proof, computation, complexity, and some cognitive topics, namely, natural language and psychological experiments.

Chapter 3

Syllogistic Reasoning

This chapter 'opens the box' of propositional logic, and looks inside the statements that we make when we communicate. Very often, these statements are about objects and their properties, and we will now show you a first logical system that deals with these. *Syllogistics* has been a standard of logical reasoning since Greek Antiquity. It deals with quantifiers like 'All P are Q' and 'Some P are Q', and it can express much of the common sense reasoning that we do about predicates and their corresponding sets of objects. You will learn a famous graphical method for dealing with this, the so-called 'Venn Diagrams', that can tell valid syllogisms from invalid ones. As usual, the chapter ends with some outlook issues, toward logical systems, computer science (this time, the area of databases and knowledge representation), and again cognitive phenomena in the real world of linguistics and psychology.

3.1 Motivation — Reasoning About Predicates and Classes

Aristotle

The Greek philosopher Aristotle (384 BC – 322 BC) proposed a system of reasoning in his *Prior Analytics* (350 BC) that was so successful that it has remained a paradigm of logical reasoning for more than two thousand years: the *Syllogistic*.

Syllogisms A *syllogism* is a logical argument where a quantified statement of a specific form (the conclusion) is inferred from two other quantified statements (the premises).

The quantified statements are all of the form "Some/all A are B," or "Some/all A are not B," and each syllogism combines three predicates or properties. Notice that "All A are not B" can be expressed equivalently in natural language as "No A are B," and "Some A are not B" as "Not all A are B."

A syllogism is called *valid* if the conclusion follows logically from the premises in the sense of Chapter 1, whatever we take the real predicates and objects to be: if the premises are true, the conclusion must be true. The syllogism is *invalid* otherwise.

Here is an example of a valid syllogism:

$$\begin{array}{c} \text{All Greeks are humans} \\ \text{All humans are mortal} \\ \hline \text{All Greeks are mortal.} \end{array} \tag{3.1}$$

We can express the validity of this pattern using the $\models$ sign introduced in Chapter 2:

$$\text{All Greeks are humans, All humans are mortal} \models \text{All Greeks are mortal.} \tag{3.2}$$

This inference is valid, and, indeed, this validity has nothing to do with the particular predicates that are used. If the predicates *human*, *Greek* and *mortal* are replaced by different predicates, the result will still be a valid syllogism. In other words, it is the *form* that makes a valid syllogism valid, not the *content* of the predicates that it uses. Replacing the predicates by symbols makes this clear:

$$\begin{array}{c} \text{All A are B} \\ \text{All B are C} \\ \hline \text{All A are C.} \end{array} \tag{3.3}$$

The classical quantifiers Syllogistic theory focusses on the quantifiers in the so called *Square of Opposition*, see Figure (3.1). The quantifiers in the square express relations between a first and a second predicate, forming the two arguments of the assertion. We think of these predicates very concretely, as sets of objects taken from some domain of discourse that satisfy the predicate. Say, 'boy' corresponds with the set of all boys in the relevant situation that we are talking about.

The quantified expressions in the square are related across the diagonals by external (sentential) negation, and across the horizontal edges by internal (or verb phrase) negation. It follows that the relation across the vertical edges of the square is that of internal plus external negation; this is the relation of so-called *quantifier duality*.

Because Aristotle assumes that the domain of discourse is non-empty, the two quantified expressions on the top edge of the square cannot both be true; these expressions are called *contraries*. Similarly, the two quantified expressions on the bottom edge cannot both be false: they are so-called *subcontraries*.

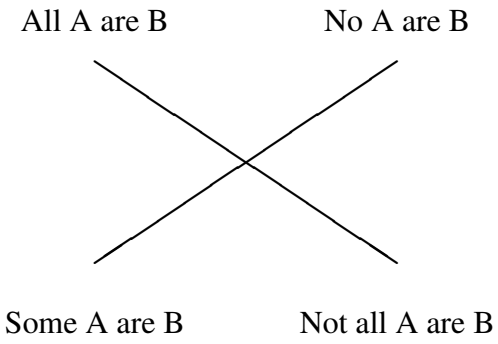


Figure 3.1: The Square of Opposition

Existential import Aristotle interprets his quantifiers with *existential import*: *All A are B* and *No A are B* are taken to imply that there are *A*. Under this assumption, the quantified expressions at the top edge of the square imply those immediately below them. The universal affirmative quantifier *all* implies the individual affirmative *some* and the universal negative *no* implies the individual negative *not all*. Existential import seems close to how we use natural language. We seldom discuss 'empty predicates' unless in the realm of phantasy. Still, modern logicians have dropped existential import for reasons of mathematical elegance, and so will we in this course.

The universal and individual affirmative quantifiers are said to be of types **A** and **I** respectively, from Latin **A***ff***I***rmo*, the universal and individual negative quantifiers of type **E** and **O**, from Latin *N***E***g***O**. Aristotle's theory was extended by logicians in the Middle Ages whose working language was Latin, whence this Latin mnemonics. Along these lines, *Barbara* is the name of the syllogism with two universal affirmative premises and a universal affirmative conclusion. This is the syllogism (3.1) above.

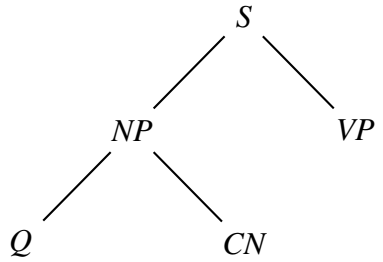
Here is an example of an invalid syllogism:

$$\begin{array}{l} \text{All warlords are rich} \\ \text{No students are warlords} \\ \hline \text{No students are rich} \end{array} \tag{3.4}$$

Why is this invalid? Because one can picture a situation where the premises are true but the conclusion is false. Such a *counterexample* consists of a situation with at least one rich student who is not a warlord. This situation is consistent with the two premises: simply make sure that any warlord in the situation is rich. This 'picturing' can be made precise, and we will do so in a moment.

3.2 The Language of Syllogistics

Syllogistic statements consist of a quantifier, followed by a common noun followed by a verb: $Q\ N\ V$. This is an extremely general pattern found across human languages. Sentences S consist of a Noun Phrase *NP* and a Verb Phrase *VP*, and the Noun Phrase can be decomposed into a Determiner Q plus a Common Noun *CN*:



Thus we are really at the heart of how we speak. In these terms, a bit more technically, Aristotle studied the following inferential pattern:

$$\frac{\begin{array}{l}\text{Quantifier}_1\ \text{CN}_1\ \text{VP}_1\\ \text{Quantifier}_2\ \text{CN}_2\ \text{VP}_2\end{array}}{\text{Quantifier}_3\ \text{CN}_3\ \text{VP}_3}$$

where the quantifiers are *All*, *Some*, *No* and *Not all*. The common nouns and the verb phrases both express properties, at least in our perspective here ('man' stands for all men, 'walk' for all people who walk, etcetera). To express a property means to refer to a class of things, at least in a first logic course. There is more to predicates than sets of objects when you look more deeply, but this 'intensional' aspect will not occupy us here.

In a syllogistic form, there are two premises and a conclusion. Each statement refers to two classes. Since the conclusion refers to two classes, there is always one class that figures in the premises but not in the conclusion. The *CN* or *VP* that refers to this class is called the *middle term* that links the information in the two premises.

Exercise 3.1 What is the middle term in the syllogistic pattern given in (3.3)?

To put the system of syllogistics in a formal setting we will first make a brief excursion to the topic of notation for operations on sets.

3.3 Sets and Operations on Sets

The binary relation $\in$ is called the element-of relation. If some object a is an element of a set A then we write $a \in A$ and if this is not the case we write $a \notin A$. Note that if $a \in A$, A is certainly a set, but a itself may also be a set. Example: $\{1\} \in \{\{1\}, \{2\}\}$.

If we want to collect all the objects together that have a certain property, then we write:

$$\{x \mid \varphi(x)\} \tag{3.5}$$

for the set of those x for which some property described by φ holds. Sometimes we restrict this property to a certain *domain or discourse* or *universe* U of individuals. To make this explicit, we write:

$$\{x \in U \mid \varphi(x)\} \tag{3.6}$$

to denote the set of all those x in U for which φ holds. Note that $\{x \in U \mid \varphi(x)\}$ defines a subset of U.

To describe a set of elements sharing multiple properties $\varphi_1, \ldots, \varphi_n$ we write:

$$\{x \mid \varphi_1(x), \ldots, \varphi_n(x)\} \tag{3.7}$$

Instead of a single variable, we also may have a sequence of variables. For example, we want to describe a set of pairs that have a certain relation, as in the following example.

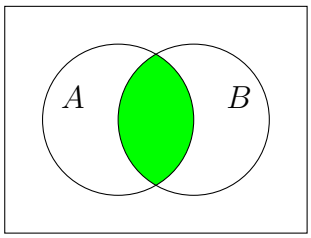
$$A = \{(x, y) \mid x \text{ is in the list of presidents of the US }, y \text{ is married to } x\} \tag{3.8}$$

For example, $(\text{Bill\_Clinton}, \text{Hillary\_Clinton}) \in A$ but, due to how the 2008 presidential election turned out, $(\text{Hillary\_Clinton}, \text{Bill\_Clinton}) \notin A$. In this example A is a set of pairs. Sets of pairs are in fact the standard mathematical representation of binary relations between objects (see below).

In talking about sets, one often also wants to discuss combinations of properties, and construct new sets from old sets. The most straightforward operation for this is the *intersection* of two sets:

$$A \cap B \;=\; \{x \mid x \in A \text{ and } x \in B\} \tag{3.9}$$

If A and B represent two properties then $A \cap B$ is the set of those objects that have both properties. In a picture:

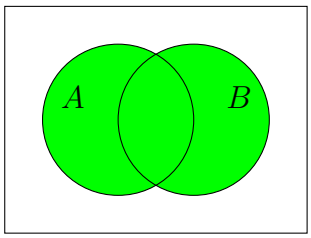


The intersection of the set of 'red things' and the set of 'cars' is the set of 'red cars'.

Another important operation is the *union* that represents the set of objects which have *at least one* of two given properties.

$$A \cup B \;=\; \{x \mid x \in A \text{ or } x \in B\} \tag{3.10}$$

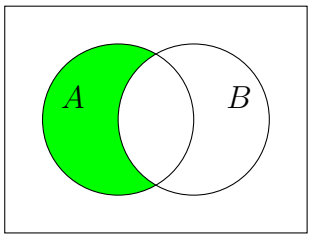
The 'or' in this definition should be read in the *inclusive* way. Objects which belong two both sets also belong to the union. Here is a picture:



A third operation which is often used is the *difference* of two sets:

$$A \setminus B \quad = \quad \{x \mid x \in A \text{ and } x \notin B\} \tag{3.11}$$

If we think of two properties represented by A and B then $A \setminus B$ represents those things that have the property A but not B. In a picture:

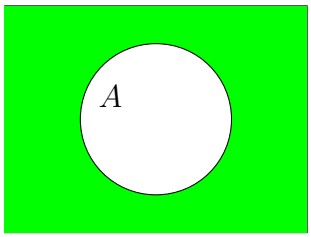


The picture representations of the set operations are called *Venn diagrams*, after the British mathematician John Venn (1834 - 1923). In a Venn diagram, sets are represented as circles placed in such a way that each combination of these sets is represented. In the case of two sets this is done by means of two partially overlapping circles.

Next, there is the *complement* of a set (relative to some given universe U (the domain of discourse):

$$\overline{A} \quad = \quad \{x \in U \mid x \notin A\} \tag{3.12}$$

In a picture:



Making use of complements we can describe things that do *not* have a certain property.

The complement operation makes it possible to define set theoretic operations in terms of each other. For example, the difference of two sets A and B is equal to the intersection of A and the complement of B:

$$A \setminus B \quad = \quad A \cap \overline{B} \tag{3.13}$$

Complements of complements give the original set back:

$$\overline{\overline{A}} \quad = \quad A \tag{3.14}$$

Complement also allows us to relate union to intersection, by means of the following so-called *de Morgan equations*:

$$\begin{aligned} \overline{A \cup B} \quad &= \quad \overline{A} \cap \overline{B} \\ \overline{A \cap B} \quad &= \quad \overline{A} \cup \overline{B} \end{aligned} \tag{3.15}$$

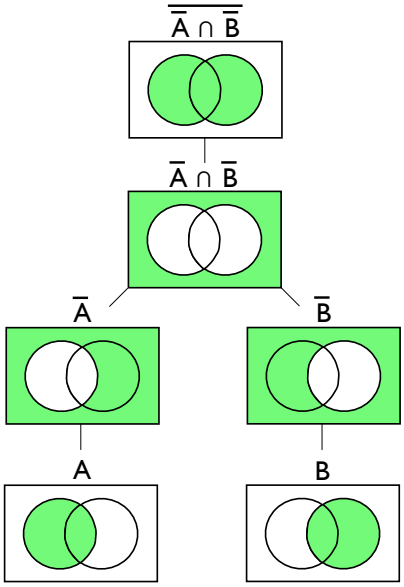


Figure 3.2: Construction of $A \cup B$ using intersection and complement.

From the second de Morgan equation we can derive a definition of the union of two sets in terms of intersection and complement:

$$A \cup B = \overline{\overline{A}} \cup \overline{\overline{B}} = \overline{\overline{A} \cap \overline{B}} \tag{3.16}$$

This construction is illustrated with Venn diagrams in Figure 3.2. Also important are the so-called distributive equations for set operations; they describe how intersection distributes over union and vice versa:

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C) \end{aligned} \tag{3.17}$$

Figure 3.3 demonstrates how the validity of the first of these equations can be computed by means of Venn-diagrams. Here we need three circles for the three sets A, B and C, positioned in such a graphical way that every possible combination of these three sets is represented in the diagrams.

The relation between sets and propositions The equalities between sets may look familiar to you. In fact, these principles have the same shape as propositional equivalences that describe the relations between $\neg$, $\wedge$ and $\vee$. In fact, the combinatorics of sets using complement, intersection and union *is* a Boolean algebra, where complement behaves like negation, intersection like conjunction and union like disjunction. The zero element of the algebra is the empty set $\emptyset$.

We can even say a bit more. The Venn-diagram constructions as in Figures 3.2 and 3.3 can be viewed as construction trees for set-theoretic expressions, and they can be

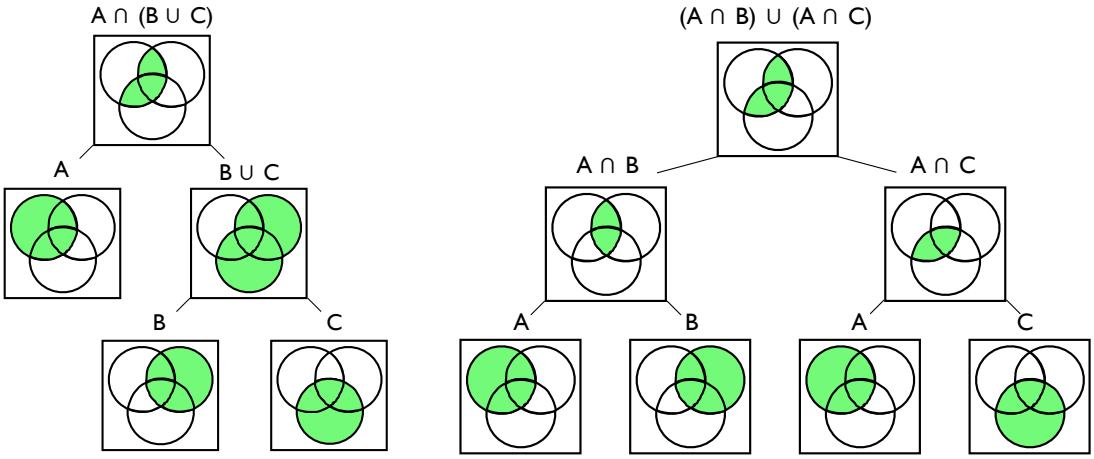


Figure 3.3: One of the distribution laws illustrated by means of Venn diagrams.

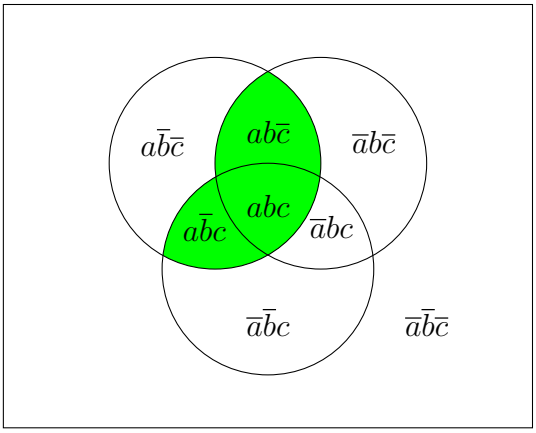
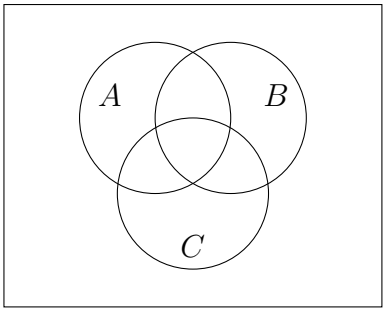


Figure 3.4: The support for $a \wedge (b \vee c)$ in a Venn-diagram.

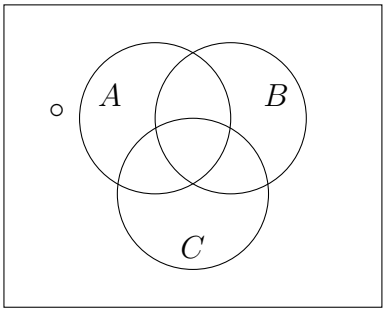
reinterpreted as construction trees for formulas of propositional logic. Substitution of proposition letters for the base sets and replacing the set operations by the corresponding connectives gives a parsing tree with the corresponding semantics for each subformula made visible in the tree. A green region corresponds to a valuation which assigns the truth-value 1 to the given formula, and a white region to valuation which assigns this formula the value 0. You can see in the left tree given in Figure 3.3 that the valuations which makes the formula $a \wedge (b \vee c)$ true are abc, $a\overline{b}c$ and $ab\overline{c}$ (see Figure 3.4).

3.4 Syllogistic Situations

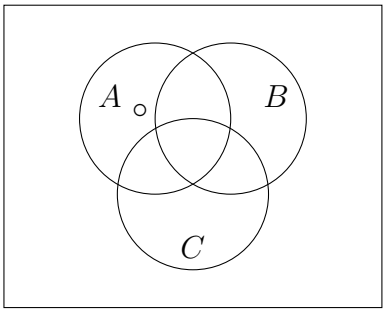
Since all syllogistic forms involve just three predicates A, B and C, we can draw a general picture of a syllogistic situation as the following *Venn Diagram*:



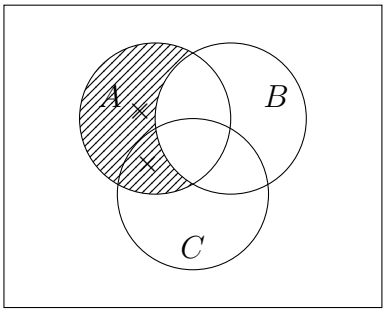
The rectangular box stands for a set of objects that form the domain of discourse, with three possible properties A, B and C. Note that there are 8 regions in all, quite properly, since that is the number of all possible combinations. An individual without any of these properties has to be outside of the three circles, like this:



An object with property A but lacking the properties B and C has to be inside the A circle, but outside the B and C circles, like this:

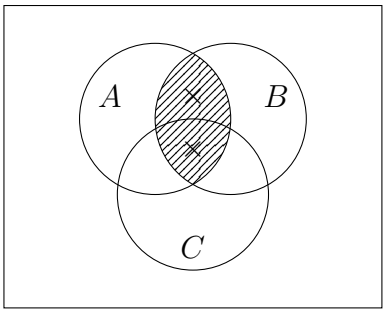


Now let us look in detail at what the Aristotelian quantifiers express. *All A are B* expresses that the part of the A circle outside the B circle has to be empty. We can indicate that in the picture by crossing out the forbidden regions, like this:



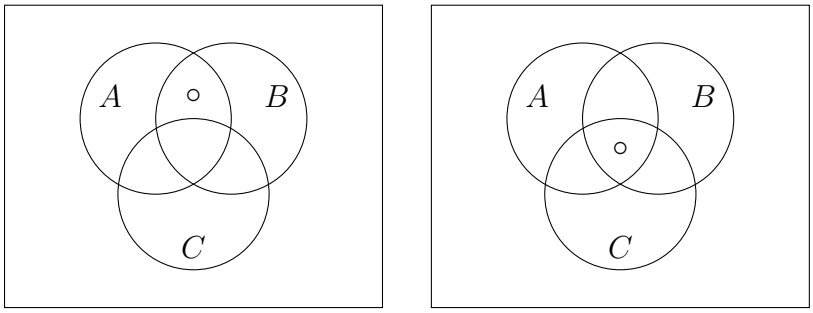
Note that the preceding picture does not take existential import into account. As we already said, we will leave it out in the interest of simplicity. And we lose nothing in this way. If you want to say that a predicate P is non-empty, you can always do so explicitly with a quantifier 'Some'.

No A are B expresses that the part of the A circle that overlaps with the B circle has to be empty. Again, we can indicate this in a picture by crossing out the forbidden areas:

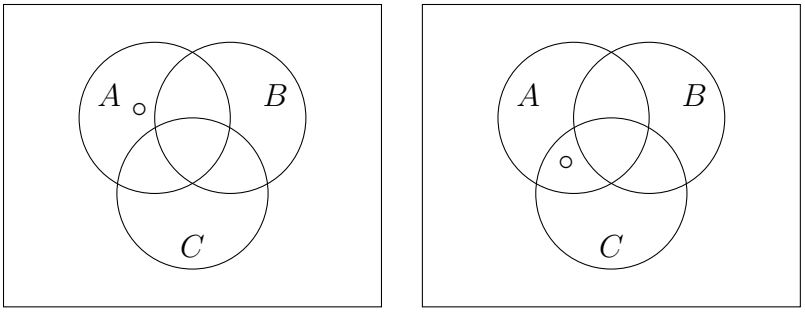


Again, existential import ("there must be A's") is not taken into account by this picture.

Now we move from universal quantifiers to existential ones. *Some A are B* expresses that the part of the picture where the A and the B circles overlap has to be non-empty. We can indicate that in the picture by putting an individual in an appropriate position. Since we do not know if that individual has property C or not, this can be done in two ways:



Not all A are B, or equivalently *Some are are not B*, expresses that the part of the A circle that falls outside the B circle has to be non-empty. There has to be at least one individual that is an A but not a B. Since we do not know whether this individual has property C or not, we can again picture this information in two possible ways:



Some authors do not like this duplication of pictures, and prefer putting the small round circle for the individual on the border line of several areas.

You no doubt realize that such a duplication of cases makes the picture method much harder in terms of complexity, and hence, as we shall see, the art in checking validity for syllogisms is avoiding it whenever possible.

3.5 Validity Checking for Syllogistic Forms

The diagrams from the preceding section lead to a check for syllogistic validity:

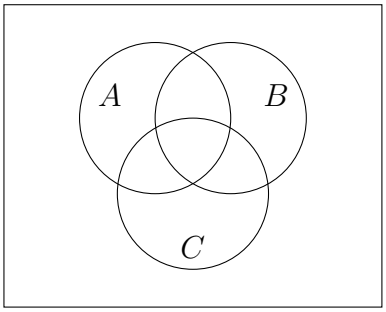
Working with diagrams We illustrate the method with the following valid syllogism:

$$\begin{array}{l}\text{All warlords are rich}\\ \text{No student is rich}\\ \hline \text{No warlord is a student}\end{array} \tag{3.18}$$

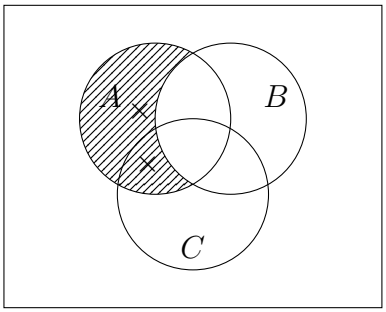
To carry out the validity check for this inference, we start out with the general picture of a domain of discourse with three properties. Next, we update the picture with the information provided by the premises. Here, the understanding is this:

> Crossing out a region with $\times$ means that this region is empty (there are no individuals in the domain of discourse with this combination of properties), while putting a $\circ$ in a region means that this region is non-empty (there is at least one individual with this combination of properties). Leaving a blank region means that there is no information about this region (there may be individuals with this combination of properties, or there may not).

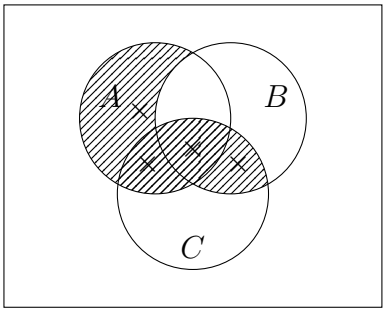
The method is this: we *update* with the information from the premises, and next *check* in the resulting picture whether the conclusion holds or not. Let A represent the property of being a warlord, B the property of being rich, and C the property of being a student. Then we start with the following general picture:



According to the first premise, *All A are B* has to be true, so we get:



By the second premise, *No C are B* has to be true, so we extend the picture as follows:



Finally, we have to check the conclusion. The conclusion says that the regions where A and C overlap have to be empty. Well, they are, for both of these regions have been crossed out. So the conclusion has to be true. Therefore, the inference is valid.

The general method The method we have used consists of the following four steps:

Draw the Skeleton Draw an empty picture of a domain of discourse with three properties A, B and C. Make sure that all eight combinations of the three sets are present.

Crossing out – Universal step Take the universal statements from the premises (the statements of the form "All ..." and "No ...", and cross out the forbidden regions in the diagram.

Filling up – Existential step Take the existential statements from the premises (the statements of the form "Some ..." and "Not all ..."), and try to make them true in the diagram by putting a $\circ$ in an appropriate region, while respecting the $\times$ signs. (This step might lead to several possibilities, all of which have to satisfy the check in the next item.)

Check Conclusion If the conclusion is universal it says that certain regions should have been crossed out. Are they? If the conclusion is existential it says that certain regions should have been marked with a $\circ$. Are they? If the answer to this question is affirmative the syllogism is valid; otherwise a counterexample can be constructed, indicating that the syllogism is invalid.

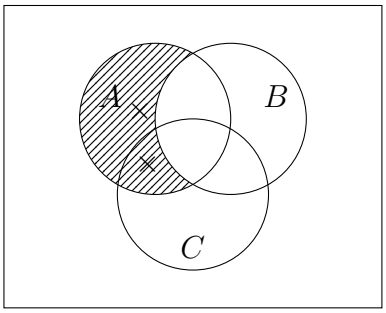
To illustrate the procedure once more, let us now take the invalid syllogism 3.4 that was mentioned before (repeated as 3.19).

$$\begin{array}{l} \text{All warlords are rich} \\ \text{No student is a warlord} \\ \hline \text{No student is rich} \end{array} \tag{3.19}$$

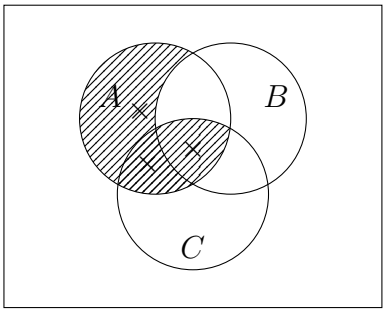
The symbolic form of this syllogism is:

$$\text{All A are B} \quad \text{No C are A} \quad \text{Therefore: No C are B.} \tag{3.20}$$

The premise statements are both universal. Crossing out the appropriate regions for the first premise gives us:



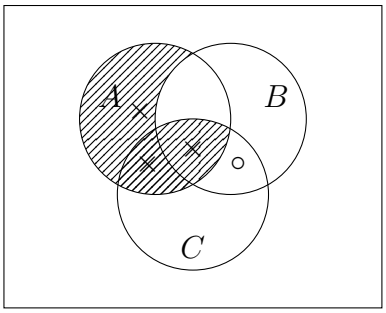
After also crossing out the regions forbidden by the second premise we get:



Note that the region for the AC's outside B gets ruled out twice. It looks like the second premise repeats some of the information that was already conveyed by the first premise (unlike the case with the previous example). But though this may say something about presentation of information, it does not affect valid or invalid consequences.

Finally, we check whether the conclusion holds. *No C are B* means that the regions where C and B overlap are forbidden. Checking this in the diagram we see that the region where A, B and C overlap is indeed crossed out, but the region outside A where B and C overlap is not. Indeed, the diagram does not contain information about this region. This means that we can use the diagram to construct a counterexample to the inference.

> The diagram allows us to posit the existence of an object that satisfies B and C but not A, in the concrete case of our example, a rich student who is not a warlord:



This final diagram gives the shape that all counterexamples to the valididity of 3.19 have in common. All these counterexamples will have no objects in the forbidden regions, and at least one object in the region marked with $\circ$.

Diagrams like this have a long history in logic, going back to Leibniz and the 18th century logicians.

Exercise 3.2 Check the following syllogistism for validity, using the method just explained.

> Some philosophers are Greek
>
> No Greeks are barbarians
>
> ---
>
> No philosophers are barbarians. (3.21)

Exercise 3.3 Check the following syllogistic pattern for validity.

$$\frac{\begin{array}{l}\text{No Greeks are barbarians}\\ \text{No barbarians are philosophers}\end{array}}{\text{No Greeks are philosophers.}} \tag{3.22}$$

Exercise 3.4 Check the following syllogistic pattern for validity.

$$\frac{\begin{array}{l}\text{No Greeks are barbarians}\\ \text{Some barbarians are philosophers}\end{array}}{\text{Not all philosophers are Greek.}} \tag{3.23}$$

Exercise 3.5 Can you modify the method so that it checks for syllogistic validity, but now with the quantifiers all read with existential import? How?

More than three predicates The validity check for syllogistics can also be extended to syllogistic inferences with more than two premises (and more than three predicates). This can still be done graphically (Venn had several beautiful visualizations), but you may also want to think a bit more prosaically in terms of tabulating possibilities. Here is one way (disregarding matters of computational efficiency, that will return below).

For purposes of exposition, assume that four predicates A, B, C, D occur in the inference. List all possible combinations in a table (compare the tables for the propositonal variables in Chapter 2 – we economized a bit here, writing the property only when it holds):

	A	B	AB
C	AC	BC	ABC
D	AD	BD	ABD
CD	ACD	BCD	ABCD

Take as example the following entailment

$$\text{All A are B, No C are B, Some C are D, Therefore: Not all D are A.} \tag{3.24}$$

Again we can use the update method to check whether this is valid. First update with the information that all A are B. This rules out certain possibilities:

	A $\times$	B	AB
C	AC $\times$	BC	ABC
D	AD $\times$	BD	ABD
CD	ACD $\times$	BCD	ABCD

All A are B

The information that no C are B also rules out possibilities, as follows:

	A	B	AB
C	AC	BC $\times$	ABC $\times$
D	AD	BD	ABD
CD	ACD	BCD $\times$	ABCD $\times$

No C are B

Combining these two updates, we get:

	A $\times$	B	AB
C	AC $\times$	BC $\times$	ABC $\times$
D	AD $\times$	BD	ABD
CD	ACD $\times$	BCD $\times$	ABCD $\times$

All A are B and No C are B

The third premise, "some C are D," is existential. It states that there has to at least one CD combination in the table. There is only one possibility for this:

	A $\times$	B	AB
C	AC $\times$	BC $\times$	ABC $\times$
D	AD $\times$	BD	ABD
CD $\circ$	ACD $\times$	BCD $\times$	ABCD $\times$

Finally, we must check whether "not all D are A" holds in the table that results from updating with the premises. And indeed it does: region CD is non-empty (indicated by the presence of the $\circ$), so it gives us a witness of a D which is not an A. Therefore, the given inference must be valid.

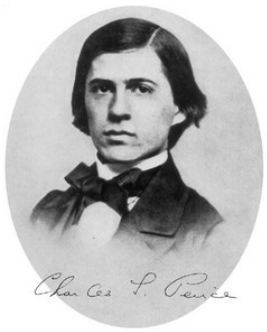
The system Working through the exercises of this section you may have realized that the diagrammatic validity testing method can be applied to any syllogism, and that, in the terms of Chapter 2, it is *sound* (only valid syllogism pass the test) and *complete* (all valid syllogisms pass the test). Moreover, the method decides the question of validity in a matter of a few steps. Thus, again in our earlier terms, it is a *decision method* for validity: the system of the Syllogistic is *decidable*. This is like what we saw for propositional logic, and indeed, the two systems are related, as we shall see in Section 3.7.

3.6 A Refutation Method for Syllogistic Validity

The validity testing method presented in the previous section proceeds by starting out from the set of all possible syllogistic situations. Since syllogistic arguments involve three properties, a situation for a syllogistic argument has $2^3 = 8$ regions, corresponding to all the possible combinations of the three properties. So there are $2^8 = 256$ different

situations to be considered altogether, for each region in the diagram can either be inhabited or empty. This is somewhat daunting, and we would like to do better. And we can.

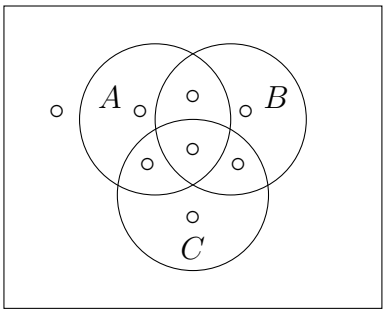
A variation on the validity testing method presented in the previous section was proposed by Christine Ladd, in a thesis written in 1882, under the direction of the famous American logician Charles Sanders Peirce (1839–1914). Ladd manages to test syllogistic validity while focussing on a *single* syllogistic situation.



Charles Sanders Peirce

Christine Ladd

Instead of starting out with a diagram representing all possible syllogistic situations, Ladd proposed to start out from a single situation, namely the situation where all regions are inhabited:



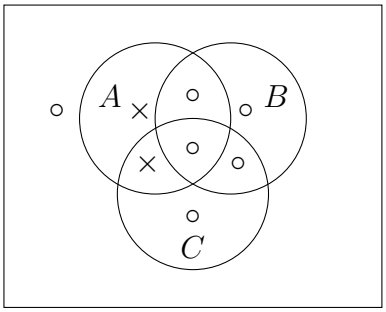
Now consider again the syllogism

$$\begin{array}{l} \text{All A are B} \\ \text{All B are C} \\ \hline \text{All A are C.} \end{array} \tag{3.25}$$

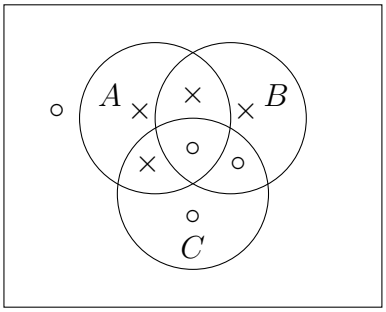
Now we make an important twist. To check whether this is valid, we attempt to *refute* the argument, by trying to find a situation where the premises and the *negation* of the conclusion are true. In other words, we are trying to find a model for:

$$\text{All A are B}, \text{All B are C}, \text{Not all A are C}.$$

First, observe that *Not all A are C* is equivalent to *Some A are not C*. Starting out from the picture above, we first process the universal statements, as instructions to *remove* individuals from the picture. The statement *All A are B* requires the region $A \setminus B$ to be empty, so we have to remove the inhabitants of this region, as follows:



The statement *All B are C* requires the region $B \setminus C$ to be empty, so we have to remove the inhabitants from this region too, as follows:



Does this leave us with a situation where *Some A are not C* is true? For that to be the case, we need to have at least one inhabitant in the region $A \setminus C$. But, as you can see in the picture, such an inhabitant cannot be found. Now what does this mean? We have taken care to start out with representatives for all possible combinations of the properties A, B and C. Next, we have take care not to eliminate more individuals than strictly necessary to make universal statements true. We end up with a situation where some existential statement fails. So there is *no way* to make this particular set of statements jointly true. This can only mean that the refutation attempt fails. In other words: the syllogism that we started out with must be valid. This method is much more efficient, especially when used for syllogisms with (many) more assumptions and classes.

The elimination process described above looks remarkably like update with propositional information. In fact, George Boole himself already suggested that propositional logic can be used for checking syllogistic validity. The ruling out of combinations that occurs in checking universal syllogistic statements works in the same way as the update method for propositional logic that was discussed in Chapter 2. In updating with propositional formulas, situations that make a formula false get eliminated from the set of possibilities. With universal syllogistic statements things work the same way. The universal statement "All A are B" eliminates situations where some members of A are

non-Bs. In the same way a universal statement of the form "No A are B" rules out the AB-representatives.

The treatment of existential statements is different in this setting. After the truth of universal statements is enforced by eliminative updates, existential statements are to be tested only. Verification of the existential statement "Some A are B" is the same as checking whether there is an A which is also B still present in the situation. An existential statement "Not all A are B" is true if an A which does not belong to B is still present.

As usual, the rest of this chapter explores a few connections with other areas, starting with mathematical systems, then moving to computation, and ending with cognition. These topics are not compulsory in terms of understanding all their ins and outs, but they should help broaden your horizon.

3.7 Outlook — The Complexity of Syllogistic Reasoning

The clear advantage of the refutation method as presented in the previous section is that only a single diagram is needed to test the validity of a syllogism. Still, this method suggests that syllogistic reasoning is just as complex as testing validity of propositional logical inferences since all the possible combinations of properties have to be represented in such a diagram or table. The number of combinations coincide with the number of rows in truth tables for propositional logic: 2^n where n is the number of variables. Fortunately, there exists an alternative translation to propositional logic such that it can be incorporated into a computationally well behaving fragment for which a simple algorithm can be given which decides validity in polynomial time.

Again, as in the previous section, this method is refutation based by means of checking the consistency of the assumptions extended with the negation of the conclusion.

Translation to propositional logic Let Σ be some finite set of syllogistic statements of which we want to test whether it is consistent or not. To begin with, we translate this set by means of a function T which assigns a set of propositional formulas to every statement in Σ. Let $\varphi_1, ..., \varphi_n$ be an enumeration of all existential statements in Σ, then T is defined in the following way for every φ_i:

$$T(\varphi_i) = \begin{cases} \{p_i, q_i\} & \text{if } \varphi_i = \text{ Some } P \text{ are } Q \\ \{p_i, \neg q_i\} & \text{if } \varphi_i = \text{ Not all } P \text{ are } Q \end{cases}$$

For the universal statements we define T in the following way:

$$\begin{aligned} T(\text{All } P \text{ are } Q) &= \{\neg p_1 \vee q_1, \ldots, \neg p_n \vee q_n\} \\ T(\text{No } P \text{ are } Q) &= \{\neg p_1 \vee \neg q_1, \ldots, \neg p_n \vee \neg q_n\} \end{aligned}$$

A propositional variable p_i can be read as the statement that the individual indexed by the number i belongs to the class P." The translation T introduces for each existential

statement an individual numbered i witnessing this statement: A PQ-er confirms *Some* P *are* Q, and a P-er who belongs to the non-Q's establishes *Not all* P *are* Q. For each universal statement *All* P *are (not)* Q the translation T settles for each of these individuals that if it is a P then it has to be a (non)-Q.

Let $T(\Sigma)$ be the set of propositional formulas obtained by joining all individual translations into one set. For this set we have the following property:

$$T(\Sigma) \text{ is consistent if and only if } \Sigma \text{ is consistent.} \tag{3.26}$$

In the last chapter of the third part on logical methods the technical details of this kind of correspondences will be explained.

The set $T(\Sigma)$ contains relatively simple formulas for which consistency can be tested by means of a simple procedure. All the formulas in $T(\Sigma)$ are so-called Horn-formulas and these formulas have the so-called *minimal valuation property*. This means that if such a set S is consistent then there exists a valuation V_S which supports all the formulas in the set S and that all other valuations V which supports S completely assigns 1 to all the propositional variables which are true under V_S.

Checking consistency of our syllogistic set Σ then comes down to computing first which variables have to be made true by a candidate minimal valuation for $T(\Sigma)$, and then we test whether this candidate also supports all negative information in $T(\Sigma)$. In order to get hold of the only candidate minimal valuation we take all propositional variables in $T(\Sigma)$ and combine these with the formulas of the form $\neg p \vee q$ as to specify what else has to be made true. So, we start with

$$\{p \mid p \in T(\Sigma)\}$$

and then add

$$\{q \mid \neg p \vee q \in T(\Sigma)\}$$

for all propositional variables p of which truth was already confirmed. The last extension step has then to be repeated until no further propositional variables are added. This procedure has to end at a certain stage since the number of propositional variables occuring in the translation set $T(\Sigma)$ is finite.

The second part of the consistency test requires the definition of our candidate valuation V as the valuation which assigns 1 to all the propositional variables in the set which has been constructed in the first part and the value 0 to those variables which are not in this set.

The last part of the test is whether this valuation V supports all negative formulas $\neg p$ and $\neg q \vee \neg r$ in $T(\Sigma)$. If this is the case then this valuation *is* the minimal valuation $V_{T(\Sigma)}$ and then $T(\Sigma)$, and also Σ, must be consistent. If it does not support all the negative information in $T(\Sigma)$ then Σ must be inconsistent.

The soundness of the last conclusion comes with the minimal valuation property. Surely, the valuation V supports all the formulas of the form p and $\neg q \vee r$ because of

the way it has been constructed. Moreover, it is a minimal valuation for this part of $T(\Sigma)$. If V falsifies one of the negative formulas then every valuation which extends V with more true propositional variables will also falsify such a formula. Formally, if V supports all the formulas in $T(\Sigma)$ then $V = V_{T(\Sigma)}$. If it does not, then $V_{T(\Sigma)}$ does not exist and therefore *no* valuation supports the full translation set. In the latter case $T(\Sigma)$, and therefore Σ itself, must be inconsistent.

Examples Let us illustrate the method by the following example with four properties:

$$\text{All } PQ,\ \text{All } QR,\ \text{Some } PS/\ \text{All } RS$$

Following the refutation procedure we instead check the consistency of the set

$$\{\text{All } PQ,\ \text{All } QR,\ \text{Some } PS,\ \text{Not all } RS\}$$

There are two existential statements in this set, and therefore, our procedure tries to construct a situation with two individuals, of which the first belongs to the PS's and the second belongs to R but not to S. Together with the translation of the two universal statements we obtain the following set of propositional formulas:

$$\{p_1, s_1, r_2, \neg s_2, \neg p_1 \vee q_1, \neg p_2 \vee q_2, \neg q_1 \vee r_1, \neg q_2 \vee r_2\}$$

The set of true propositional variables we start with is $\{p_1, s_1, r_2\}$. In the first round of the procedure to construct the minimal valuation this set is extended with q_1 and in the second round with r_1. Then in the third round no more information is added. The corresponding valuation verifies the negative information indeed, and therefore the given set is consistent. This means that the original syllogism must be invalid. A situation with two individuals of which the first has all four properties and the second belongs only to R is indeed a situation which supports the three assumptions but rejects the conclusion.

If we replace the conclusion of the first example by *Some* RS then we obtain a valid syllogism. We have to prove this by demonstrating the inconsistency of

$$\{\text{All } PQ,\ \text{All } QR,\ \text{Some } PS,\ \text{No } RS\}$$

This set contains only a single existential statement, and the translation procedure gives a smaller set than in the previous case:

$$\{p_1, s_1, \neg p_1 \vee q_1, \neg q_1 \vee r_1, \neg r_1 \vee \neg s_1\}$$

The set of true propositional variables we begin with is $\{p_1, s_1\}$. In the first extension round q_1 is added, and in the second r_1. The third round does not give a further extension. The resulting valuation falsifies $\neg r_1 \vee \neg s_1$ and therefore the set of syllogistic statements must be inconsistent, i.e., the original inference must be valid.

3.8 Outlook — Knowledge Representation

Our next theme concerns a connection with computer science. So far you we have mainly emphasized computation and algorithms. But of course, computer science is always about two things in tandem: *data representation* and *computation over these data*. This time we look at the former aspect.

Much of practical computing has to do with retrieving knowledge that has been stored in data-bases or knowledge-bases. These often have a logical flavour:

Syllogistic knowledge bases A *syllogistic knowledge base* (henceforth *KB*) is a set of triples of the form $(A, B, 1)$ or $(A, B, 0)$, where A and B are names of classes. If A is a name of a class, then $\overline{A}$ is the name of the complement of that class. The complement of a class A is the the class of all things that are not in A. If $(A, B, 1)$ is in the KB, then we take that to mean that $A \subseteq B$. If $(A, B, 0)$ is in the KB, then we take that to mean that $A \not\subseteq B$. Recall that $A \not\subseteq B$ is equivalent to $A \cap \overline{B} \neq \emptyset$. (Again, if you have difficulty with this set notation, you should consult Appendix A.)

Here is an example of a very simple *KB*:

$$\begin{array}{c}\{(\text{Academics}, \text{Europeans}, 0), (\text{Musicians}, \text{Europeans}, 0),\\ (\text{Dutch}, \text{Europeans}, 1), (\text{Academics}, \overline{\text{Musicians}}, 1)\}.\end{array} \tag{3.27}$$

The triple $(\text{Academics}, \text{Europeans}, 0)$ says that there are non-European academics. $(\text{Musicians}, \text{European}, 0)$ expresses that not all musicians are Europeans.

$(\text{Dutch}, \text{European}, 1)$ expresses that all Dutch are Europeans.

Finally, the triple $(\text{Academics}, \overline{\text{Musicians}}, 1)$ expresses that $\text{Academics} \subseteq \overline{\text{Musicians}}$, or equivalently, that no-one is both an academic and a musician (according to this *KB*, that is).

Such knowledge bases can store general conceptual knowledge about how predicates are related, but also particular connections between properties that hold for certain groups of objects. Both occur widely in practice.

Incidentally, when talking about complements, we want to avoid class names like $\overline{\overline{A}}$. We don't want to talk about the class of "un-unhappy people". We call such people "happy". Here is a trick for that. Let $\widetilde{A}$ be given by: if A is of the form $\overline{C}$ then $\widetilde{A} = C$, otherwise $\widetilde{A} = \overline{A}$. In other words, we let double overlines cancel out.

Extracting information through logic Now we can ask ourselves questions about what *follows* from the information in a *KB*. For example, does it follow from the information in (3.27) that some academics are Europeans? No, it does not, but it does follow that some academics are non-Europeans. Does it follow from the information in (3.27) that there are European musicians? No, it does not. Does it follow from the *KB* that all academics are non-European? No, it does not.

We will now give axioms and rules for deriving information from a KB. These axioms and rules give an *inference engine* for consulting a *KB*. The rules of the inference engine allow us to compute properties of the $\subseteq$ relation. Intuitively, $A \Longrightarrow B$ expresses dat $A \subseteq B$. That is, $\Longrightarrow$ is a syntactic symbol that is semantically interpreted as the subset relation. We will use $A \not\Longrightarrow B$ to express that $A \not\subseteq B$.

Here is the connection with the statements of Syllogistics:

- "All A are B" is expressed as $A \Longrightarrow B$,
- "No A are B" is expressed as $A \Longrightarrow \widetilde{B}$,
- "Not all A are B" is expressed as $A \not\Longrightarrow B$,
- "Some A are B" is expressed as $A \not\Longrightarrow \widetilde{B}$,

Now we formulate the axioms and rules of the system. You can see what follows as another exercise with a little symbolic proof calculus of the sort that we have also explored in Chapter 2 for propositional logic.

An axiom is a rule without a premise. Our first axiom says that all A are A:

$$\frac{}{A \Longrightarrow A}$$

Next, we will not follow Aristotle in reading the classes with existential import, but instead we will make the weaker assumption that our domain of discourse is not empty. One way to express this is by saying that it has to hold for any class A that if there are no non-A, then there must be A. (For if there are no A and there are no non-A, then this means that the universe is empty, and this we want to exclude.) Here is the rule that expresses our assumption of a non-empty domain of discourse:

$$\frac{\widetilde{A} \Longrightarrow A}{A \not\Longrightarrow \widetilde{A}}$$

Computing the subset relation from a knowledge base $\mathbf{K}$ is done with the following rules:

$$\frac{(A, B, 1) \in \mathbf{K}}{A \Longrightarrow B} \qquad \frac{A \Longrightarrow B}{\widetilde{B} \Longrightarrow \widetilde{A}} \qquad \frac{A \Longrightarrow B \quad B \Longrightarrow C}{A \Longrightarrow C}$$

Computing the non-subset relation from the knowledge base is done with the rules below:

$$\frac{(A, B, 0) \in \mathbf{K}}{A \not\Longrightarrow B} \qquad \frac{A \not\Longrightarrow B}{\widetilde{B} \not\Longrightarrow \widetilde{A}} \qquad \frac{A \not\Longrightarrow C \quad B \Longrightarrow C}{A \not\Longrightarrow B} \qquad \frac{A \Longrightarrow B \quad A \not\Longrightarrow C}{B \not\Longrightarrow C}$$

These are all the rules of the system.

Here is a concrete example of the derivation of a statement from the *KB* given in (3.27) with these rules:

$$\dfrac{\dfrac{(\text{Academics}, \text{Europeans}, 0) \in \mathbf{K}}{\text{Academics} \not\Longrightarrow \text{Europeans}} \quad \dfrac{(\text{Dutch}, \text{Europeans}, 1) \in \mathbf{K}}{\text{Dutch} \Longrightarrow \text{Europeans}}}{\text{Academics} \not\Longrightarrow \text{Dutch}}$$

Next we call a *KB inconsistent* if there exist A, B for which both $A \Longrightarrow B$ and $A \not\Longrightarrow B$ can be derived. If a *KB* is not inconsistent it is called *consistent*, and these are of course the ones that the information systems around us hopefully use:

Exercise 3.6 Show that the *KB* $\{(A, B, 1), (B, C, 1), (C, D, 1), (A, D, 0)\}$ is inconsistent.

If a knowledge base is consistent, then questions of the form *Do Q CN VP?'* will always have an unambiguous answer: "Yes", "No", or "I don't know". For example, the question about (3.27) "Are all musicians Dutch?" will get the answer "I don't know".

Exercise 3.7 What is the answer of (3.27) to the question "Are there any non-Dutch musicians?"

Updating knowledge bases Other topics from earlier chapters, too, make sense in this setting. In particular, if a knowledge base is consistent, then an *update operation* can be defined to create a larger knowledge base. To a new fact of the form "$QCNVP$" there can be three possible reactions:

- "I knew that already" if the information is already derivable from the *KB*.
- "Inconsistent with my information" if the information contradicts what is in the *KB*.
- "OK" if the information is consistent with the *KB* (this means: if adding the information to the knowledge base results in a new consistent knowledge base).

Example Update of (3.27) with "not all musicians are Dutch" results in "OK", and the fact $(\text{Musicians}, \text{Dutch}, 0)$ gets added to the *KB*.

Exercise 3.8 What is the reaction to update of (3.27) with "not all academics are Dutch"?

Exercise 3.9 What is the reaction to update of (3.27) with "no musicians are academics"?

Thus, the old Aristotelian language of predicates is at the same time a modern database language. It is also used in information retrieval, where computer systems (try to) derive information from text corpora. The area of *description logic* in computer science deals with such simple formalisms for representing knowledge, and efficient algorithms for extracting information from them.

3.9 Outlook — The Syllogistic and Actual Reasoning

Aristotle's system is closely linked to the grammatical structure of natural language, as we have said at the start. Indeed, many people have claimed that it stays so close to our ordinary language that it is part of the *natural logic* that we normally use. Medieval logicians tried to extend this, and found further patterns of reasoning with quantifiers that share these same features of staying close to linguistic syntax, and allowing for very simple inference rules. 'Natural Logic' is a growing topic these days, where one tries to find large simple inferential subsystems of natural language that can be described without too much mathematical system complexity. Even so, we have to say that the real logical hitting power will only come in our next chapter on predicate logic, which consciously deviates from natural language to describe more complex quantifier reasoning of types that Aristotle did not handle.

Syllogistic reasoning has also drawn the attention of cognitive scientists, who try to draw conclusions about what goes on in the human brain when we combine predicates and reason about objects. As with propositional reasoning, one then finds differences in performance that do not always match what our methods say, calling attention to the issue how the brain represents objects and their properties and relations. From another point of view, the *diagrammatic* aspect of our methods has attracted attention from cognitive scientists lately. It is known that the brain routinely combines symbolic language-oriented and visual and diagrammatic representations, and the Venn Diagram method is one of the simplest pilot settings for studying how this combination works.

Summary *In this chapter you have learnt how one simple but very widespread kind of reasoning with predicates and quantifiers works. This places you squarely in a long logical tradition, before we move to the radical revolutions of the 19th century in our next chapter. More concretely, you are now able to*

- *write basic syllogistic forms for quantifiers,*
- *understand set diagram notation for syllogistic forms,*
- *test syllogistic inferences using Venn diagrams,*
- *understand how diagrams allow for update,*
- *understand connections with propositional logic,*
- *understand connections with data representation.*

Chapter 4

Talking about the World with Predicate Logic

Overview At this stage of our course, you already know propositional logic, the system for reasoning with sentence combination. You have also seen how sentences make quantified statements about properties of objects, and you know the basics of syllogistic reasoning with quantifiers. In this Chapter you are going to learn the full system of 'predicate logic' of objects, predicates, and in principle, arbitrary forms of quantification. This is the most important system in logic today, because it is a *universal language* for talking about *structure*. A structure is a situation with objects, properties and relations, and it can be anything from daily life to science: your family tree, the information about you and your friends on facebook, the design of the town you live in, but also the structure of the number systems that are used in mathematics, geometrical spaces, or the universe of sets.

Predicate logic has been used to increase precision in describing and studying all these structures, from linguistics and philosophy to computer science and mathematics. In this chapter, you will learn how it works, first more informally with examples, later with more formal definitions. But this power comes at a price, this chapter is not easy, and mastering predicate logic until it comes naturally to you takes a while, as successive generations of students (including your teachers) have found.

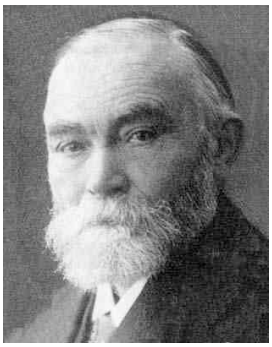
4.1 Learning the Language

Propositional logic is about classifying situations in terms of 'not', 'and', 'or' combinations of facts, whose structure we do not analyze any further. This truth-table perspective is powerful in its own way (it is the basis of all the digital circuits running your computer), but very poor in other respects. Basic propositions in propositional logic are not assumed to have internal structure. "John walks" is translated as p, "John talks" as q, and the information that both statements are about John gets lost. Predicate logic looks at the internal structure of such basic facts. It translates "John walks" as Wj and "John talks" as Tj, so that it becomes clear that the two facts express two properties of the same individual

(named by the constant j).

More generally, predicate logic is a language that can say much more than propositional logic about the internal structure of situations, especially, the objects that occur, the properties of these objects, and their relations to each other. In addition, predicate logic can talk about universal quantification (all, every, each, ...) and existential quantification (some, a, ...). This brings it much closer to two other languages that you already knew before this course: the natural languages that we use in the common sense world of our daily activities, and the symbolic languages that are used in mathematics and the sciences. You can view predicate logic as a bit of both, though in decisive points, it differs from natural language and follows a more mathematical system.

Predicate logic is a somewhat streamlined version of a "language of thought" that was proposed in 1878 by the German philosopher and mathematician Gottlob Frege (1848 – 1925). The experience of a century of work with this language is that, in principle, it can write all of mathematics as we know it today.



Gotlobb Frege

We will now introduce predicate logic via a long string of examples, the way you usually learn a language by doing. Grammar comes later: further on in this chapter we give precise grammatical definitions, plus other information. First comes our list of examples, to allow you to learn by cases. We do not start in a vacuum here: the natural language that you know already is our running source of examples and, in some cases, contrasts:

The basic alphabet We first need names for *objects*. We use constants ('proper names') a, b, c, ... for special objects, and variables x, y, z, ... when the object is indefinite. Later on, we will also talk about function symbols for complex objects.

Then, we need to talk about *properties and predicates* of objects. Capital letters are predicate letters, with different numbers of 'arguments' (i.e., the objects they relate) indicated. In natural language, 1-place predicates are intransitive verbs ("walk") and common nouns ("boy"), 2-place predicates are transitive verbs ("see"), and 3-place predicates are so-called ditransitive verbs ("give"). 1-place predicates are als called *unary predicates*, 2-place predicates are called *binary predicates*, and 3-place predicates are called *ternary predicates*. In natural language ternary predicates are enough to express the most complex verb pattern you can get, but logical languages can handle any number of arguments.

Next, there is still *sentence combination*. Predicate logic also has the usual operations from propositional logic: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.

Finally, and very importantly, we need to express *quantification*. Predicate logic has quantifiers $\forall x$ ("for all x") and $\exists x$ ("there exists an x") that can express an amazing number of things. The use of these key operators will become clear in what follows.

Learning by example: from natural language to predicate logic For now, here is a long list of examples exploring the style of thinking in predicate logic. You can think of this as learning to see the underlying 'logical form' of the statements that you would normally make when speaking or writing. Later on in this chapter we will say a bit more about the history of this important notion. Along the way we will point out various important features.

We start with the simplest statements about objects:

natural language	*logical formula*
John walks	Wj
John is a boy	Bj
He walks	Wx
John sees Mary	Sjm
John gives Mary the book	$Gjmb$

Predicate logic treats both verbs and nouns as standing for properties of objects, even though their syntax and communicative function is different in natural language. The predicate logical translation of "John walks" uses a predicate letter and a constant. The predicate logical translation of "John is a boy" also uses a predicate letter with a single constant: Bj.

Remark on natural language and logic This example is interesting, for it may seem that "John is a boy" contains an existential quantifier ("there is a boy"), but this appearance is misleading. This has been much discussed: already ancient Greek logicians felt that their own language has a redundancy here. Likewise, the Greek logicians already observed that the "is" of predication in "John is a boy" does not seem to do any real work, and again, might just be an accident of language. Indeed, many natural languages do not put an "a" in this position. An example from Latin: *"puer est"* ("he is a child"). But such languages also do not put an "a" in positions where there is a real existential quantifier. Another example from Latin: *"puer natus est"* ("a child is born"). As one can imagine, debates about how natural language structure relates to logical structure are far from over.

Translation key Note that in writing predicate logical translations, one has to choose a "key" that matches natural languages expressions with the corresponding logical letters. And then stick to it. For mnemonic purposes, we often take a capital letter for a predicate close to the natural language expression (B for "boy"). Technically, in the logical notation, we should indicate the exact number of object places that the predicate takes ("B has

one object place"), but we drop this information when it is clear from context. The object places of predicates are also called *argument places*. If a predicate takes more than one argument, the key should say in which order you read the arguments. E.g., our key here is that Sjm says that John sees Mary, not that Mary sees John. The latter would be Smj.

Predicates in language and mathematics In mathematics, 2-place predicates are most frequent. Common examples are $=$, $<$, $\in$. It is usual to write these predicates in between their arguments: $2 < 3$. (We will say more about the expressive possibilities of the predicate "$=$" in section 4.5.) Occasionally, we also have 3-place predicates. An example from geometry is "x lies between y and z". Note that this is not a conjunction of "x lies between y" and "x lies between z": that would be nonsense.

informal mathematics	*logical/mathematical formula*
Two is smaller than three	$2 < 3$
x is smaller than three	$x < 3$
x is even (i.e., 2 divides x)	$2\|x$
Point p lies between q and r	$Bpqr$

In the special case of talking about mathematics there are standard names for objects and relations. In $x < 3$, the term "3" is a constant that names a particular natural number, and "$<$" is a standard name for a specific relation. The notation $x|y$ expresses that x is a divisor of y, i.e., that division of y by x leaves no remainder.

Incidentally, an abbreviation that is often used in mathematics is $x < y < z$, to express that the number y is in between x and z. This is not an example of real 3-place predicate. Rather, it is an abbreviation of $x < y \wedge y < z$.

Exercise 4.1 Express $\neg(x < y < z)$ in terms of the binary predicate $<$ and propositional connectives, using the fact that $x < y < z$ is an abbreviation of $x < y \wedge y < z$.

The standard in predicate logic is to write the predicate first, then the objects. The exceptions to this rule are the names for binary relations in mathematics: $<$ for less than, $>$ for greater than, and so on. Still, the general rule is to write the predicate letter first. This is for uniformity, and it takes getting used to. Many natural languages put predicates in the middle (English, French, but also the informal language of mathematics), but there are also languages that put them first, or last. Dutch and German are interesting, since they put predicates in the middle in main clauses ("Jan zag Marie"), but change that order in subordinate clauses ("Ik hoorde dat Jan Marie zag").

Referring to objects: pronouns and variables We already saw how proper names like "John" or "Mary" refer to specific objects, for which we wrote constants like a, b. But both natural language and mathematics have found the need for 'variable names' as well, that stand for different objects in different contexts. Pronouns in language more or less correspond with variables: Sjx expresses "John sees her" for some contextually determined female, and $x > 2$ expresses that some contextually determined number x is

larger than 2. So pronouns are like variables, at least, to a first approximation. Now the famous author Italo Calvino once wittily called pronouns "the lice of thought" [Cal88]. But are they just a nuisance? To the contrary, what pronouns do is provide coherence in what you say, in referring back to the same individual in the right places. That is also exactly what mathematical variables do. So we get:

John sees her	Sjx
He sees her	Syx
This is less than that	$x < y$
He sees himself	Sxx

Note that whether $x < y$ is true totally depends on specifying x and y. 'This' and 'that' in natural language are demonstrative pronouns for pointing at things. Mathematicians use variables as pointers. Instead of "suppose we have some number greater than one" they say "suppose x is a number greater than one". Next, they use x to refer to this number.

Adding on propositional logic Propositional operators can be added in the obvious way to the preceding statements, and they function as before:

John does not see Mary	$\neg Sjm$
Three is not less than two	$\neg 3 < 2$, or abbreviated: $3 \not< 2$.
John sees Mary or Paula	$Sjm \vee Sjp$
Three is less than three or three is less than four	$3 < 3 \vee 3 < 4$.
x is odd (i.e., two does not divide x)	$\neg(2 \mid x)$
If John sees Mary, he is happy	$Sjm \rightarrow Hj$

In the last example, natural language uses a pronoun ("he is happy"), while the logical translation does not use a variable but a second occurrence of the constant j to refer to the same object as before.

Exercise 4.2 Translate the following sentences into predicate logical formulas:

(1) If John loves Mary then Mary loves John too.

(2) John and Mary love each other.

(3) John and Mary don't love each other.

(4) If John and Peter love Mary then Peter nor Mary loves John.

The binary predicate $\leq$ can be expressed in terms of $<$ and $=$:

$$x \text{ is less than or equal to } y \qquad x \leq y \qquad x < y \vee x = y$$

Exercise 4.3 Rephrase "$x \leq y \wedge y \leq z$" in predicate logic, using binary relations $<$ for "less than" and $=$ for "equal".

Exercise 4.4 Rephrase "$\neg(x \leq y \wedge y \leq z)$" in predicate logic, using binary relations $<$ for "less than" and $=$ for "equal".

A first glimpse of quantifiers Now for quantifiers, where we say that objects exist without naming them explicitly:

Someone walks	$\exists x Wx$
Some boy walks	$\exists x(Bx \wedge Wx)$
John sees a girl	$\exists x(Gx \wedge Sjx)$
A girl sees John	$\exists x(Gx \wedge Sxj)$
A girl sees herself	$\exists x(Gx \wedge Sxx)$

Here you can see what variables do: they keep track of which object does what when that object occurs several times in the sentence. With long sentences or long texts, a systematic mechanism like this becomes crucial in keeping reasoning straight.

Here are some examples with universal quantifiers:

Everyone walks	$\forall x Wx$
Every boy walks	$\forall x(Bx \rightarrow Wx)$
Every girl sees Mary	$\forall x(Gx \rightarrow Sxm)$

Note that "Some boy walks" is translated with a conjunction symbol, and "Every boy walks" is translated using an implication symbol. The following exercise explains why.

Exercise 4.5 Let B be the predicate for "boy" and W the predicate for "walk".

(1) What does $\forall x(Bx \wedge Wx)$ express?

(2) And what does $\exists x(Bx \rightarrow Wx)$ express?

There is much more to quantifiers, but we will take things slowly. For the moment, just let the above examples sink in. At first, you may find the predicate-logical way of thinking unfamiliar. But give it some more thought, and you may come to see that the predicate-logical formulas really get to the heart of meaning and structure of objects.

4.2 Monadic Predicate Logic

Before unleashing the full power of quantifiers in predicate logic, let us first use a more modest stepping stone.

The language of the Syllogism (Chapter 3) may be compared with a small fragment of predicate logic that imposes a restriction on the form of the predicates that are allowed. Predicate logic with only 1-place predicates (unary properties of objects) is called *monadic predicate logic*. This restriction on predicate form curbs the power of quantification considerably, but it also makes it easier to understand. In this section we will iscuss how syllogistic reasoning can be expressed in monadic predicate logic.

Here is how the syllogistic statements can be expressed in monadic predicate logic:

All A are B	$\forall x(Ax \rightarrow Bx)$
No A are B	$\neg\exists x(Ax \wedge Bx)$
Some A are B	$\exists x(Ax \wedge Bx)$
Not all A are B	$\neg\forall x(Ax \rightarrow Bx)$

Exercise 4.6 Give the predicate logical formulas for the following syllogistic statements:

(1) No B are C.

(2) Some A are C.

(3) Not all A are B.

Exercise 4.7 Take the following set of premises:

$$\forall x(Ax \to Bx), \forall x(Bx \to Cx), \forall x(Cx \to Ax).$$

Show that the conclusion $\forall x(Bx \to Ax)$ follows from these premises, by using a small variation of one of the methods of Chapter 3.

Syllogistic theory has the following equivalences:

- *Not all A are B* has the same meaning as *Some A are not B*.
- *All A are not B* has the same meaning is *there are no A that are also B*.

The predicate logical versions of these equivalences give important information about the interaction between quantification and negation:

- $\neg\forall x(Ax \to Bx)$ is equivalent to $\exists x\neg(Ax \to Bx)$, which is in turn equivalent to $\exists x(Ax \wedge \neg Bx)$,
- $\forall x(Ax \to \neg Bx)$ is equivalent to $\neg\exists x\neg(Ax \to \neg Bx)$, which is in turn equivalent to $\neg\exists x(Ax \wedge Bx)$.

From this we can distill some important general quantification principles:

- $\neg\forall x\varphi$ is equivalent to $\exists x\neg\varphi$,
- $\neg\exists x\varphi$ is equivalent to $\forall x\neg\varphi$.

Reflecting on these principles a bit further, we see that negation allows us to express one quantifier in terms of the other, as follows:

- $\forall x\varphi$ is equivalent to $\neg\exists x\neg\varphi$,
- $\exists x\varphi$ is equivalent to $\neg\forall x\neg\varphi$.

Exercise 4.8 Translate the following syllogistic statements into predicate logic, without using existential quantifiers:

(1) Some A are B.

(2) Some A are not B.

(3) No A are B.

Exercise 4.9 Translate the following syllogistic statements into predicate logic, without using universal quantifiers:

(1) All A are B.

(2) All A are non-B.

(3) No A are B.

In the chapter on computation (Chapter 11), we will show how all syllogistic reasoning can be translated in monadic predicate logic. Here we just summarize a few features of this important subsystem:

- Each formula of monadic predicate logic is equivalent to one in which no quantifier stands over another.

- In fact, we have a generalization of the earlier 'distributive normal forms' of propositional logic. Calling an atom of the form Px or a negated atom Px a *literal*, a 'type description' is a conjunction of literals: one for each monadic predicate letter. These describe the kinds of objects that can occur. A *state description* is then a conjunction, for each state description, of either its existential version that such objects occur, or the negation of that. This completely determines a 'kind of semantic model'. Now comes the key fact: *each formula in monadic predicate logic is equivalent to a disjunction of state descriptions*.

- Given this closeness to propositional logic, we have a fact that already showed when we saw the Venn Diagram method for testing syllogisms. *Valid consequence in monadic predicate logic is decidable*: there is a mechanical method for testing it.

- One way of testing validity in monadic predicate logic is by translation into propositional logic, but there are many other methods. All share one feature: the *computational complexity* goes up exponentially compared to propositional logic, since we get many more possible situations to check than there were lines in a simple truth table. Thus, as we have said several times already: one always pays a price, greater expressive power means higher computational complexity. To console you, this is not a special curse of logic, the maxim holds everywhere in science.

4.3 Binary Predicates and Quantifier Combinations

Now we make a step where predicate logic really comes into its own.

Predicates with more than one object The real power of quantifiers shows when we go beyond monadic predicates, i.e., properties of objects to *relations between* more objects at the same time. This move is crucial to mathematics and science, which all turn on comparative predicates such 'x is smaller than y', 'x is an element of y', or 'x lies in between y and z'. But the same move is central to natural language. Monadic predicates are like intransitive verbs 'Walk', 'Talk', etc. But equally ubiquitous are *transitive verbs* that take both a subject and an object, such as 'Sees', 'Loves'. In fact, some verbs even take 3 objects, as in 'x gives y to z'.

Actually, the hold of the syllogistic and unary predicates has been so strong that throughout history poeple have tried to reduce binary predicates to unary ones, reading, say, 'x is smaller than y' as 'x is small and y is not small'. But if you think about this a bit, you will see that it has little to recommend itself. Mount Everest is taller than the K2, but the K2 is still really tall!

Therefore, we now take binary and higher predicates seriously.

Repeated quantifiers And once we do that, we will also see that quantifiers latch on to them, creating patterns that are essentially more complex than those found in monadic predicate logic. These patterns occur in science and natural language, but predicate logic makes them especially perspicuous:

Everyone sees someone	$\forall x \exists y Sxy$
Someone sees everyone	$\exists x \forall y Sxy$
Everyone is seen by someone	$\forall x \exists y Syx$
Someone is seen by everyone	$\exists x \forall y Syx$

The domain of quantifiers Here an additional assumption is made. The quantifiers range over all objects in some given set of objects. This is called the *domain of discourse*. In this case the domain only contains human beings.

Most often quantification in natural language sentences range over a particular subset of the domain of discourse. If the domain of discourse is human beings, then 'every girl' talks about a subset of the domain, as does 'some boy'. In the case of the translations of syllogistic statements like "All A are B" the situation was similar. In a sense, this is not a statement about the whole domain of discourse, but only about the As in the domain of discourse. The patterns are:

- "All A are ..." translates into $\forall x(Ax \rightarrow \cdots)$.
- "Some A are ..." translates into $\exists x(Ax \wedge \cdots)$.

Translating sentences with multiple quantifiers We can also use these patterns for finding translations for sentences involving double quantifications, as in the following example:

Every boy loves a girl. (4.1)

To begin with we translate the pattern for "All A are B":

$$\forall x\,(Bx \to \varphi(x)) \tag{4.2}$$

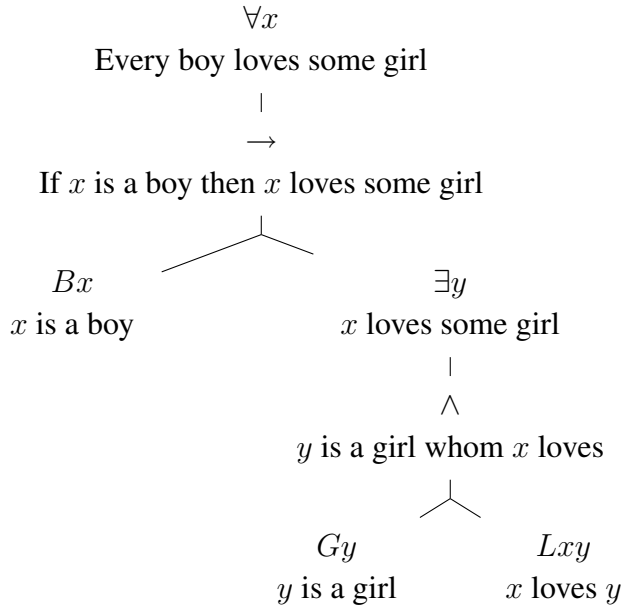
Here B is a unary predicate to represent 'boy', and $\varphi(x)$ stands for the property that we want to assign to all the boys x: "x loves a girl". This part of the sentence is in fact something of the form "Some C are D", where C represents the class of girls and D the class of those objects which are loved by x. The predicate logical translation of $\varphi(x)$ therefore looks as follows:

$$\exists y\,(Gy \wedge Lxy) \tag{4.3}$$

with Gy for "y is a girl" and Lxy for "x loves y". Substition of this translation for $\varphi(x)$ in (4.2) gives us the full predicate logical translation of (4.1):

$$\forall x\,(Bx \to \exists y\,(Gy \wedge Lxy)) \tag{4.4}$$

In the following figure, a parse tree of the translation is given which shows the compositional structure of predicate logic. Every subformula corresponds to a sentence in natural language:



(4.5)

This way of constructing translations can be of great help to find predicate logical formulas for natural language sentences. In ordinary language we are used to writing from left to right, but in predicate logic the order of putting together connectives and quantifiers is often non-linear. The following longer example illustrates this.

No girl who loves a boy is not loved by some boy. (4.6)

Although this sentence looks quite complicated, its surface structure coincides with the syllogistic form "No A are B", and so our first translation step is:

$$\neg\exists x\,(\varphi(x) \wedge \psi(x)) \tag{4.7}$$

where $\varphi(x)$ are those x who are girls who love some boy, and $\psi(x)$ represents the class of those x who are loved by no boy. The first part, $\varphi(x)$, is a conjunction of two properties: x is a girl *and* x loves a boy. This is just a small step towards a complete translation:

$$\neg\exists x\,((Gx \wedge \varphi_1(x)) \wedge \psi(x)) \tag{4.8}$$

with $\varphi_1(x)$ representing "x loves a boy". This part can be translated into:

$$\exists y\,(By \wedge Lxy) \tag{4.9}$$

Substitution of this result for $\varphi_1(x)$ in (4.8) gives us the following intermediate result:

$$\neg\exists x\,((Gx \wedge \exists y\,(By \wedge Lxy)) \wedge \psi(x)) \tag{4.10}$$

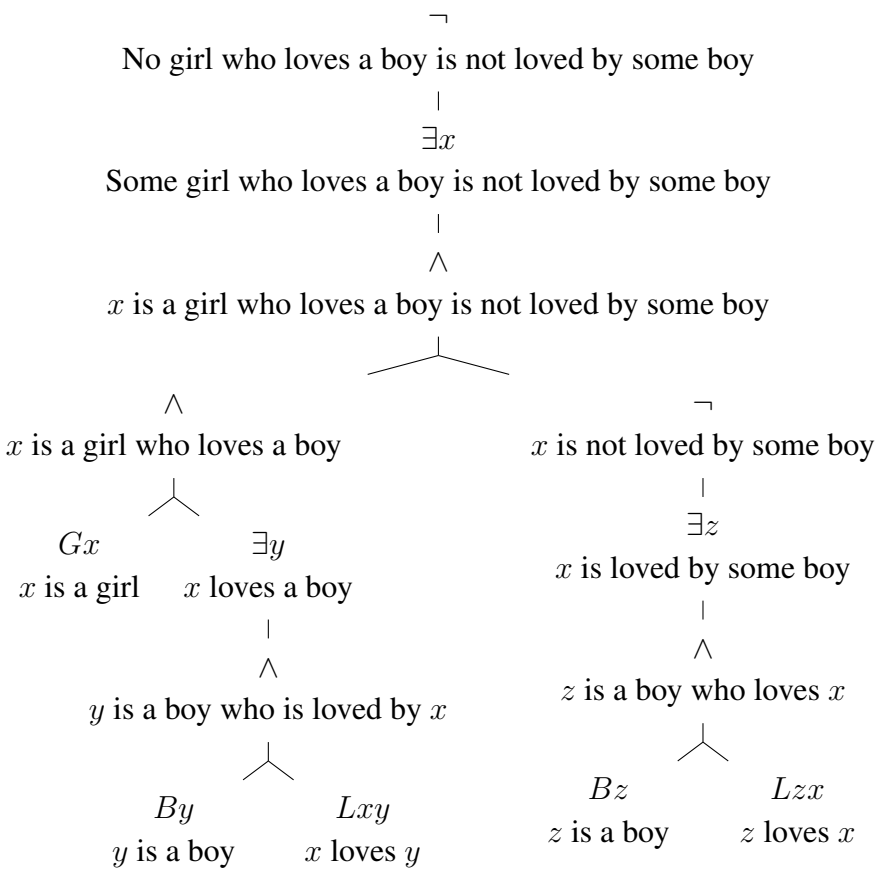
The subformula $\psi(x)$ represents "No boy loves x", which can be translated into:

$$\neg\exists z\,(Bz \wedge Lzx) \tag{4.11}$$

This yields the final result:

$$\neg\exists x\,((Gx \wedge \exists y\,(By \wedge Lxy)) \wedge \neg\exists z\,(Bz \wedge Lzx)) \tag{4.12}$$

Below a complete composition tree is given for this translation. Again, every subformula can be paraphrased in natural language:



(4.13)

Iterating quantifiers: twice, and more Two-quantifier combinations occur in natural language as we have just seen, and they are also very common in mathematics. The same logical form that expressed 'Everyone sees someone' is also that for a statement like 'Every number has a larger number'. And the above from for 'Some girl sees every boy' is also that for 'There is an odd number that divides every even number' (namely, the number 1).

Can there be still higher nestings of quantifiers? Yes, indeed. For instance, three quantifiers are involved in the famous saying that "You can fool some people some of the time, and you can fool some people all of the time, but you cannot fool all people all of the time". Likewise, three-quantifier combinations occur in mathematics. A typical example is the definition of 'continuity' of a function f in a point x:

> For every number r, there is a number s such that for all y with $|x - y| < s$:
> $|f(x) - f(y)| < r$.

Finally, nestings of four quantifiers seem to get very hard for humans to understand.

Exercise 4.10 Assume the domain of discourse to be all human beings. Translate the following sentences into predicate logic:

(1) Augustus is not loved by everyone. (Use a for Augustus, L for love.)

(2) Augustus and Livia respect each other. (Use a for Augustus, l for Livia and R for respect.)

(3) Livia respects everyone who loves Augustus.

Exercise 4.11 Assume the domain of discourse is all animals. Translate: *Some birds do not fly.* (Use B for being a bird and F for being able to fly.)

Exercise 4.12 For each of the following, specify an appropriate domain of discourse, specify a key, and translate into predicate logic. (Note: you have to understand what a sentence means before you can attempt to translate it.)

(1) Dogs that bark do not bite.

(2) All that glitters is not gold.

(3) Friends of Michelle's friends are her friends.

(4) There is a least natural number.

(5) There is no largest prime number.

Exercise 4.13 Translate the following sentences into predicate logical formulas. Assume the domain of discourse is human beings.

(1) Every boy loves Mary.

(2) Not all girls love themselves.

(3) No boy or girl loves Peter.

(4) Peter loves some girl that loves John.

Exercise 4.14 For each sentence from the previous exercise, draw a picture that makes it true.

Exercise 4.15 Translate the following sentences into predicate logical formulas. Assume the domain of discourse is human beings.

(1) Some boy doesn't love all girls.

(2) Every boy who loves a girl is also loved by some girl.

(3) Every girl who loves all boys does not love every girl.

(4) No girl who does not love a boy loves a girl who loves a boy.

Adding Function Symbols We actually left out some other devices that predicate logic has for speaking about objects.Mathematics is rand likewise, eplete with *functions* that create new objects out of old, and likewise, predicate logic becomes more expressive if we allow function symbols for functions like 'the square of x' or 'the sum of x and y'. We will see later how this can be used to write algebraic facts and do calculations, but for now, we just note that function symbols also make sense for natural language. On the domain of human beings, for instance, the *father* function gives the father of a person. Suppose we write $f(x)$ for the father of x and $m(x)$ for the mother of x, while helping ourselves to an *equality symbol* $=$ to state identity of objects. Then we can say things like:

Paula's father loves her	$Lf(p)p$
Bill and John are full brothers	$f(b) = f(j) \wedge m(b) = m(j)$
Not everyone has the same father	$\neg\forall x\forall y f(x) = f(y)$

4.4 Concrete Semantics: Formulas and Pictures

By now, it will have become clear to you that predicate logic is a very general language for talking about situations where objects have certain properties or are in certain relations.

Examples from daily life Here is a simple example from everyday life:

Suppose we use B for the property of being a bicycle, M for the property of being a man, and R for the relation of riding, then we can say that the following formula describes this salient fact about the situation in the picture:

$$\exists x \exists y (Mx \wedge By \wedge Rxy).$$

Of course, you can also add statements about the trees.

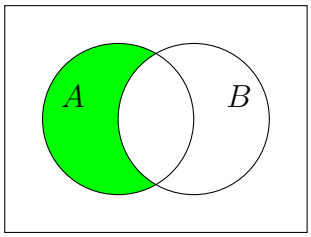
Exercise 4.16 Consider the following picture:

Use G for the property of being a girl, H for the property of being a hat, and W for the relation of wearing (so that Wxy expresses that x is wearing y). Now carry out the following tasks:

(1) Give a predicate logical formula that makes a *true* statement about the situation in the picture.

(2) Give a predicate logical formula that makes a *false* statement about the situation in the picture.

Venn diagrams The Venn diagrams that you encountered in section 3.3 are in fact also pictures of situations where objects have certain properties, although of a slightly more abstract nature. Take the Venn picture of the set of things that are in A but not in B as an example:



(4.14)

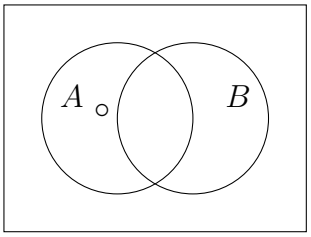
Here is the corresponding predicate logical formula:

$$Ax \wedge \neg Bx. \tag{4.15}$$

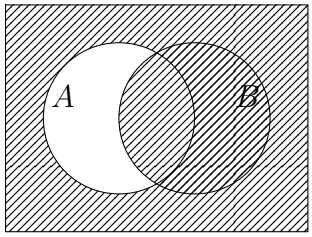
The green area in the picture corresponds to the set of individuals x that make formula 4.15 true. The formula (4.15) contains a *free variable* x, that is, x is *not bound* by a quantifier. In a given situation, a diagram or some other kind of structure, formulas with free variables, also called *open* formulas, may be without a determinate truth value.

This is not a peculiarity of predicate logic. In fact, this is similar to the way variables are used in mathematical equations: $x^2 \geq 0$ is true for the real numbers, whereas the truth-value of $x^2 < x$ depends on the value of x. It is true for values of x within the open interval $(0, 1)$ but false when x has a value outside this segment of the real line. As soon as we 'close' such open formulas or equations with a quantifier which binds the free occurrences of x then we have statements that obtain a determinate truth-value: $\forall x\,(x^2 \geq 0)$ and $\exists x\,(x^2 < x)$ are true for the real numbers and $\forall x\,(x^2 < x)$ is obviously false.

Putting a quantifier in front of (4.15) has the same determining effect. Venn diagrams can then be used in a similar way we have used them to represent syllogistic statements in the previous chapter. $\exists x\,(Ax \wedge \neg Bx)$ requires that the green area in (4.14) is inhabited (non-empty), whereas the formula $\forall x\,(Ax \wedge \neg Bx)$ states that the complement of the $Ax \wedge \neg Bx$-zone must be empty. Using the same indication of empty and non-empty areas as in the previous chapter, we get the following two diagrams for the two quantified versions of $Ax \wedge \neg Bx$:

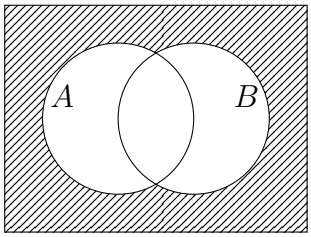


$\exists x\,(Ax \wedge \neg Bx)$



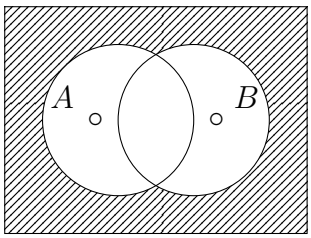
$\forall x\,(Ax \wedge \neg Bx)$

Testing validity by means of Venn diagrams Although this chapter is meant to learn to 'talk with predicate logic', Venn diagrams can be used, at this early stage, to 'reason with predicate logic' to understand simple inferential patterns. For the complete formal semantics of predicate logic, and a subsequent definition of valid inference, we will have to wait until the next chapter. Venn diagrams suffice for simple monadic inferences. Here is a first simple example:



(4.16)

This diagram corresponds to the formula $\forall x\,(Ax \vee Bx)$. The objects which are neither A nor B have been excluded. The diagram can be extended with an inhabitant in the $A \setminus B$ and the $B \setminus A$ region, respectively:



(4.17)

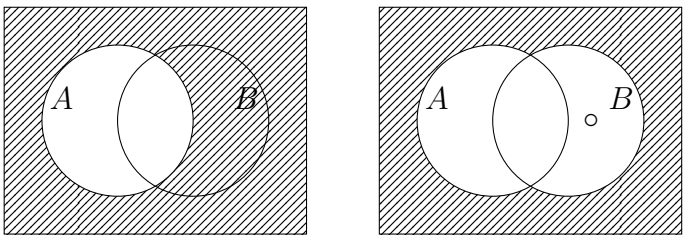
The latter makes the statement $\forall x\,Ax$ false and the former falsifies $\forall x\,Bx$. This means that $\forall x\,Ax \vee \forall x\,Bx$ is false in a diagram which supports $\forall x\,(Ax \vee Bx)$:

$$\forall x\,(Ax \vee Bx) \not\models \forall x\,Ax \vee \forall x\,Bx$$

On the other hand $\forall x\,Ax \vee \exists x\,Bx$ is a valid consequence of the same assumption:

$$\forall x\,(Ax \vee Bx) \models \forall x\,Ax \vee \exists x\,Bx \tag{4.18}$$

This can be proven by extending diagram (4.16) with the only two possible indications for the $B \setminus A$ area. It is either empty or not:



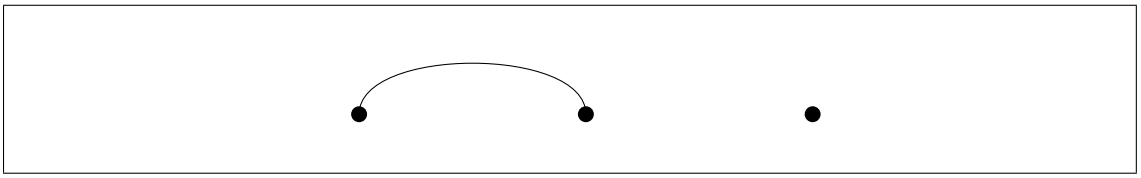
In the diagram on the left $\forall x\,Ax$ holds, whereas the one on the right hand supports $\exists x\,Bx$. This means that one of these two formulas must be true if $\forall x\,(Ax \vee Bx)$ is true, which proves that the inference (4.18) is indeed valid.

Exercise 4.17 Show that $\exists x\, Ax \wedge \exists x\, Bx \not\models \exists x\, (Ax \wedge Bx)$ by making use of Venn diagrams.

Exercise 4.18 Show that $\forall x\, Ax \wedge \exists x\, Bx \models \exists x\, (Ax \wedge Bx)$ by making use of Venn diagrams.

Graphs: representing relations Venn diagrams are for picturing sets of objects with certain *properties*. In case we want to talk about *relations* between objects we need pictures of sets with arrows between them. Such pictures are often called *graphs*. Graphs occur at many places in this course. A tree with a dominance relation is a graph, so you have already seen graphs for constructing formulas. Likewise, we will use graph models in the next two chapters to model information of agents, and the actions that they can perform. In fact, graphs are one of the most useful mathematical structures all around, and they are abstract yet simple to grasp. that is why one often uses them to demonstrate points about logic.

Undirected graphs In what follows we will give examples of how predicate logic can be used to talk about graph structure. We start with a simpler case, where points are just connected by lines ('edges'), without any direction to them. Consider the following picture of a situation with three objects, two of which are connected by an edge:



The domain of discourse now consists of three nameless points. What would be an appropriate key for talking about these points? Let us assume we have one binary predicate, call it R, for expressing that two objects in the picture are linked by an edge. Then it is easy for you to check that the following predicate logical statement is true for this situation:

$$\exists x \exists y Rxy. \tag{4.19}$$

This is true, for formula (4.19) says that there are two points that are linked.

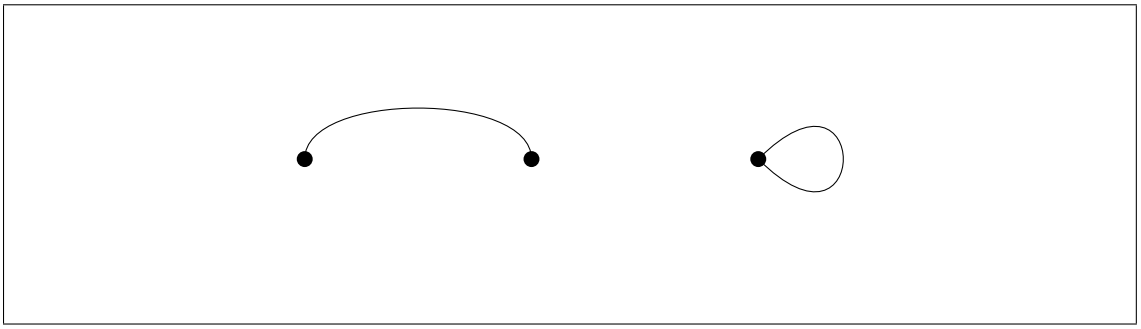
$$\forall x \forall y (Rxy \rightarrow Ryx). \tag{4.20}$$

Formula (4.20) is also true for the picture, for it expresses that if there is a link from a point to another point, then there is also a link *vice-versa*. This property is called *symmetry*. We say: the relation of the picture is *symmetric*: the direction of the links does not matter. Next take the following formula:

$$\exists x \forall y \neg Rxy. \tag{4.21}$$

This expresses that there is a point that is not linked to any point. Also true, for the point on the right has no links.

Exercise 4.19 Consider the following picture.



Which of the following formulas are true? (R expresses that there is a direct link.)

(1) $\exists x Rxx$,

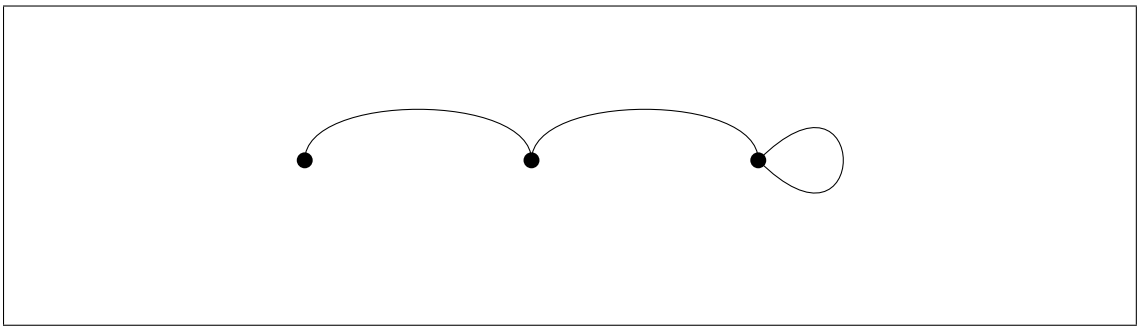
(2) $\exists x \neg Rxx$,

(3) $\exists x \forall y \neg Rxy$,

(4) $\forall x \exists y Rxy$,

(5) $\forall x \exists y \neg Rxy$.

Exercise 4.20 Consider the following picture.



Which of the following formulas are true? (R expresses that there is a direct link.)

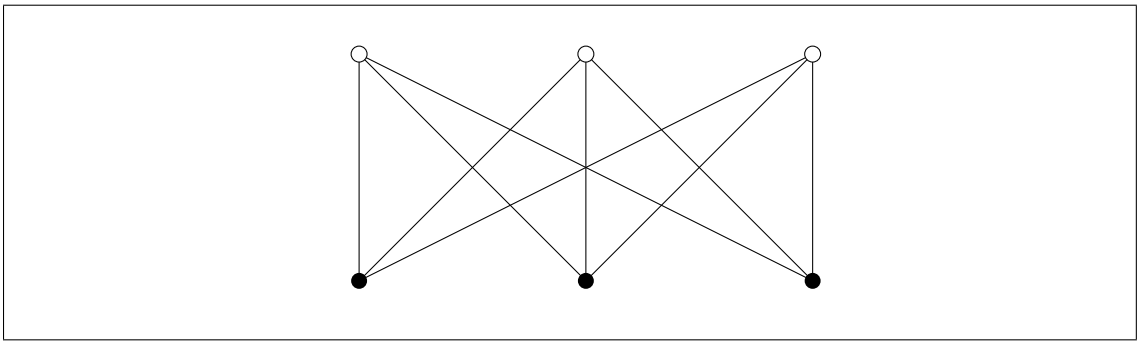
(1) $\exists x \neg Rxx$,

(2) $\exists x \forall y \neg Rxy$,

(3) $\exists x \exists y \neg Rxy$,

(4) $\forall x \exists y \neg Rxy$.

The next example has a distinction between two kinds of nodes. This is something we can talk about with a 1-place predicate:



Let us say that the solid dots $\bullet$ have the property P, and the open dots $\circ$ lack that property. We continue to use R for the link relation. "All solid dots are linked to at least one open dot" can now be expressed as follows:

$$\forall x(Px \rightarrow \exists y(\neg Py \wedge Rxy)).$$

Exercise 4.21 Express "No solid dots are linked to any open dot" in predicate logic.

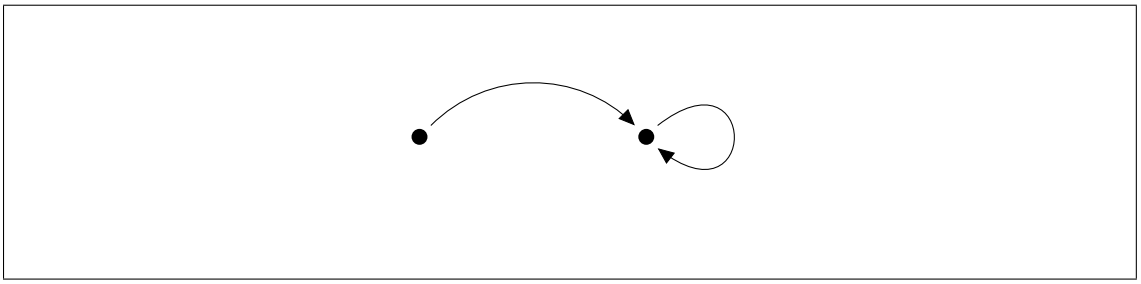
The links in the last picture also have the property that every link is between a solid dot and an open dot. How can we express this? A first attempt is the following:

$$\forall x \forall y(Rxy \rightarrow (Px \wedge \neg Py)).$$

To see why this is wrong, it is enough to consider any pair of an open dot x and a solid dot y. All these pairs are linked, but x does not have property P, whereas y does have P. To solve this, observe that we can express what we are after by saying that all links are between pairs which differ with respect to property P. This insight immediately leads to the correct translation:

$$\forall x \forall y(Rxy \rightarrow (Px \leftrightarrow \neg Py)).$$

Directed graphs The relations in the graphs so far were all symmetric: links were the same in both directions. Let us move on to cases of relations that have a direction, encoded by pointed arrows, the more common structure in the remainder of these notes. Here is an example of such a *directed graph*:



Again we use R to refer to the binary relation in the picture, and we use predicate logic to describe the properties of the relation.

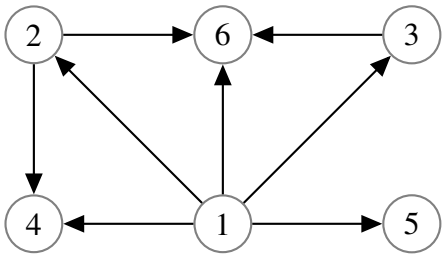
Exercise 4.22 Which of the following formulas are true of the last picture:

(1) $\forall x \exists y Rxy$,

(2) $\exists x \forall y Rxy$,

(3) $\forall x \exists y Ryx$.

Directed graphs are useful for visualisation of relations of many kinds. Consider the relation of divisibility on natural numbers. Here is a graph representation of that relation, within the set of numbers $\{1, 2, 3, 4, 5, 6\}$.



(4.22)

The circles around the numbers represent the reflexive arrows. Every number divides itself: $\forall x\,(x|x)$ where $|$ is used as an infix symbol for the relation of divisibility. Another property of the divisibility relation is the following:

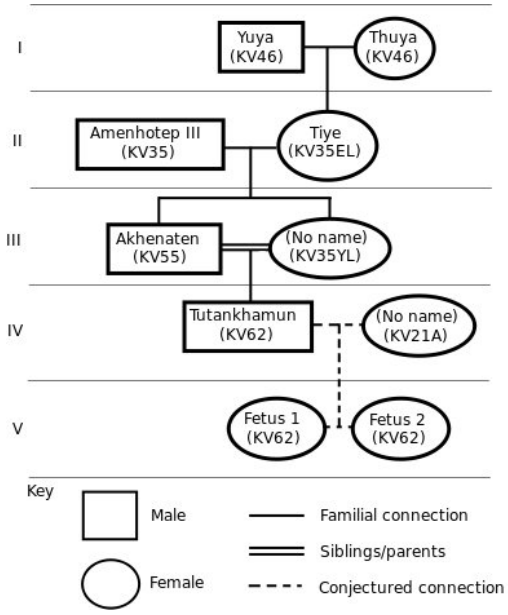
$$\forall x \forall y \forall z\,((x|y \wedge y|z) \rightarrow x|z).$$

This is called *transitivity*. In words, for every triple it is the case that if the first divides the second, and the second divides the third then the first must be a divisor of the third. Another typical distinctive feature of this arithmetical relation is *anti- symmetry*:

$$\forall x \forall y((x|y \wedge y|x) \rightarrow x = y)$$

It states that mutual divisibility only holds for identical objects.

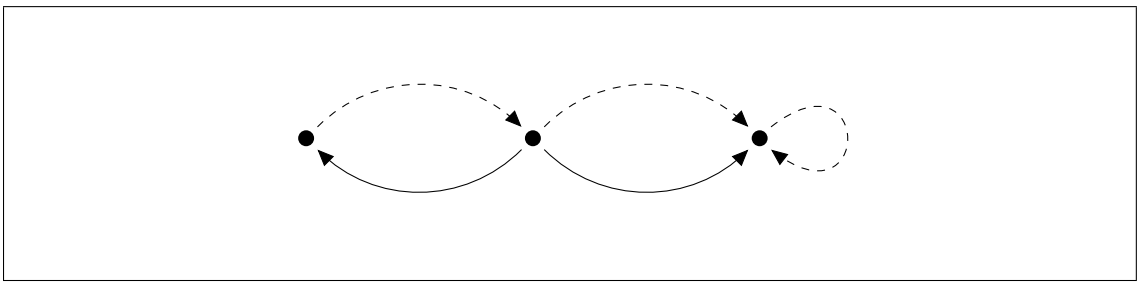
Trees Family trees are diagrams representing family relations in a tree structure, with the oldest generations at the top. Here is a famous example. The family tree of Tutankhamun was revealed by DNA research in 2010 to look as follows:



Exercise 4.23 Draw a version of this using just two kinds of arrows, $\xrightarrow{\male}$ for pointing to the father, and $\xrightarrow{\female}$ for pointing to the mother.

Exercise 4.24 Using function symbols m and f for 'mother of' and 'father of', give a predicate logical formula that expresses the surprising fact that was revealed in 2010 about the lineage of Tutankhamun: *The parents of Tutankhamun were brother and sister.*

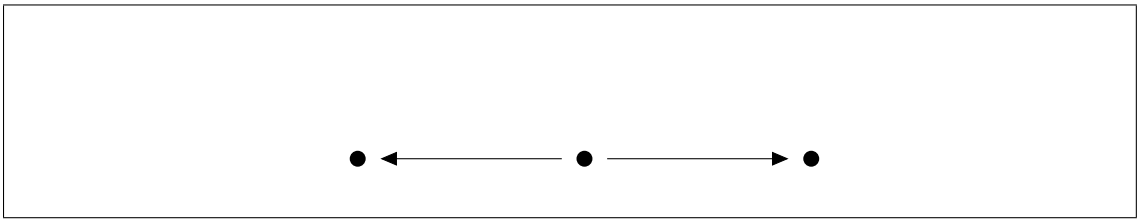
Truth, distinguishing power, and changing graphs Formulas of predicate logic need not just be interpreted in graphs: once we have done so, we can also use them to do other things. One is telling two pictures apart. The recipe for this is to find a formula that is true in one graph but not in the other. For example, $\forall x \exists y Rxy$ is *false* if we interpret R as the relation given by the solid arrows, but *true* if we interpret R as the relation given by the dashed arrows:



Exercise 4.25 Consider the following two pictures:

Give a predicate logical formula (using the predicate R for the arrow relation) that is *true* in the picture on the left and *false* in the picture on the right.

Exercise 4.26 Sometimes, when a formula is false in a graph, we may want to *change* that graph in some minimal manner to make the formula true after all. Compare the way engineers change blueprints to achieve some desired property of their constructions. Consider the following graph.



The formula $\exists x \forall y Rxy$ is *false* in the graph if R is interpreted as the $\rightarrow$ relation. Show how the formula can be made *true* by adding a single $\rightarrow$ link to the graph.

4.5 Predicate Logic and Equality

As we already hinted at above, the expressive power of predicate logic really grows when we add identity and function symbols. We will discuss a number of examples showing how this makes sense, and a the same time, increasing your familiarity with the formalism.

Equality We start with a simple example. Define a 'celebrity' as a person who knows no one, but is known by everyone. This means, of course, someone who does not know anyone else, but is known by everyone else. To express this we need a predicate for equality (or: identity). So let's use the predicate $x = y$ that was mentioned before. Now we can say:

x is a celebrity (i) $\forall y(\neg x = y \rightarrow (Kyx \wedge \neg Kxy))$.
(ii) $\forall y(\neg x = y \rightarrow Kyx) \wedge \forall y(\neg x = y \rightarrow \neg Kxy)$.

The two translations (i) and (ii) express the same thing: they are logically equivalent.

Using such translations, we can represent arguments in much finer detail than in propositional logic. For instance, the following formula expresses that C is a definition of the property of being a celebrity (this employs the useful abbreviation $x \neq y$ for $\neg x = y$).

$$\forall x(Cx \leftrightarrow \forall y(x \neq y \rightarrow (Kyx \wedge \neg Kxy))). \tag{4.23}$$

From definition (4.23) and the formula that expresses that Mary knows John, Kmj, it follows that $\neg Cm$ (Mary is not a celebrity). Clearly, this inference is far beyond the power of propositional logic.

Equality is also needed to translate the following example:

John loves Mary and Bill loves another girl. (4.24)

Clearly, with "another girl" is meant "a girl different from Mary." Using equality we can translate the example as follows:

$$Ljm \wedge \exists x(Gx \wedge x \neq m \wedge Lbx). \tag{4.25}$$

Equality is a binary predicate with a fixed logical meaning. It is always written in infix notation, as $x = y$, and with $\neg x = y$ abbreviated as $x \neq y$. Here is a similar case where equality is useful:

Mary is the only girl that John loves. (4.26)

Being the only girl with a certain property means that all other girls lack that property. Spelling this out we get:

$$Gm \wedge Ljm \wedge \forall x\,((x \neq m \wedge Gx) \rightarrow \neg Ljx) \tag{4.27}$$

By using an equivalence instead of an implication symbol we can express this a bit shorter:

$$\forall x\,(Gx \rightarrow (x = m \leftrightarrow Ljx)) \wedge Gm \tag{4.28}$$

Rephrasing this in English it says that all girls have the property that being loved by John boils down to the same thing as being equal to Mary.

Expressing Uniqueness Equality also provides a way of expressing uniqueness properties of objects. The following formula states that there is *exactly one* object which has the property P:

$$\exists x\,(Px \wedge \forall y\,(Py \rightarrow y = x)) \tag{4.29}$$

In words, there exists a 'P' such that each second object which has the property P as well must be equal to the first. Again, this can be formulated somewhat shorter by using an equivalence:

$$\exists x \forall y\,(Py \leftrightarrow y = x) \tag{4.30}$$

There is an object x such that for each object having the property P is the same thing as being equal to x. The British philosopher Bertrand Russell proposed to use this recipe to translate definite descriptions.

The father of Charles II was executed. (4.31)

This can be translated, using F for the property of being father of Charles II, as:

$$\exists x(Fx \wedge \forall y(Fy \rightarrow x = y) \wedge Ex), \tag{4.32}$$

or, as we have seen, more succinctly as:

$$\exists x(\forall y(Fy \leftrightarrow x = y) \wedge Ex). \tag{4.33}$$

Russell's analysis then gets extended to an account of the meaning of proper names as disguised or abbreviated descriptions.

Counting Objects in Predicate Logic In a similar way we can give a formula stating that there exists exactly two objects which have the property P:

$$\exists x \exists y \left(\neg x = y \wedge \forall z \left(Pz \leftrightarrow (z = y \vee z = x)\right)\right) \tag{4.34}$$

This says that there exists a pair (of two *different* things) such that having the property P is the same as being equal to one of the members of this pair. To avoid such long reformulation of relatively simple information (4.34) is sometimes abbreviated as $\exists!^2 x\, Px$, and in general $\exists!^n Px$ is used to express that exactly n things have the property P. In the case $n = 1$ we also write $\exists! x\, P$.

Exercise 4.27 Write out the predicate logical translation for "there are exactly three objects with property P".

Exercise 4.28 The $\exists!^2$-quantifier can be used to give a predicate logical description of prime numbers in the divisibility graph as depicted in (4.22) on page 4-20. How? And what kind of numbers do you get when you replace $\exists!^2$ in your definition by $\exists!^3$?

Exercise 4.29

(1) Explain why the two following sentences have different meanings (by a description of a situation in which the sentences have a different truth-value)

> There is exactly one boy who loves exactly one girl. ($\exists! x \exists! y \left(Bx \wedge Gy \wedge Lxy\right)$)
>
> There is exactly one girl who is loved by exactly one boy. ($\exists! y \exists! x \left(Bx \wedge Gy \wedge Lxy\right)$)

(2) Explain why $\exists! x \left(Ax \vee Bx\right)$ and $\exists! x\, Ax \vee \exists! x\, Bx$ have different meanings.

Applications to mathematics: expressing algebraic equations Now we move from natural language to mathematical language. As usual in this chapter, similar phenomena arise there. In the lingo of mathematics, function symbols are combined with equality to express algebraic equations like the following:

$$x \cdot (y + z) = x \cdot y + x \cdot z.$$

This is a statement of the sort that you learn to manipulate in high school. In fact, this can be viewed as a formula of predicate logic. There is the equality predicate here, but the main point is made with the function symbols for addition and multiplication.

In a domain of numbers (natural numbers, integers, fractions, reals) $+$ is a function symbol that takes two numbers and creates a number. The symbol $\cdot$ (also written as $\times$) is also a binary function symbol for the same domain of discourse.

Now that we can refer to objects in this way, we can also state basic facts about them. Algebraic equations typically say that two complex terms, viewed as two different

instructions for computation, denote the same object. If x, y, z refer to numbers, then $x \cdot (y + z)$ also refers to a number, and the equation states that $x \cdot y + x \cdot z$ refers to the same number.

The same syntax with function symbols works everywhere in mathematics. Think of mathematical notation for sets. The domain of discourse now is a universe of sets. Letters x, y, ..., stand for arbitrary sets, $\emptyset$ is the special constant name of the empty set, and function terms for intersection and union create complex terms like $x \cap (y \cup z)$ that denote sets in the way you have already seen when computing pictures for Venn Diagrams and related structures.

4.6 Outlook — Predicate Logic and Natural Language

In natural language, quantifier expressions may behave differently from what you would expect after this training in predicate logic.

A first peculiarity is that "a" may also be *generic*, with universal force. This would be written in logic with a universal quantifier:

A dog has fleas	$\forall x(Dx \rightarrow \exists y(Fy \wedge Hxy))$
A banker is trusted by no-one	$\forall x(Bx \rightarrow \forall y \neg Tyx)$.

*** To be extended ***

4.7 Outlook — Predicate Logic and Philosophy

Truth Correspondence theory of truth
Truth definition versus truth criterion.
Tarski on non-definability of truth predicate.

Meaning Categories of expressions. Meanings for nouns, verbs, determiners, sentences ...
Meaning and cognition.

Paradox Paradox of the liar, and its importance for Gödel's incompleteness proof.

4.8 Outlook — Predicate Logic and Cognition

Summary of Things You Have Learnt in This Chapter *If you have tackled the exercises in this Chapter you will have come to realize that predicate logic is a power tool of expression. You have started to learn a new language, and in using that language you have learnt to express yourself in new ways. You have learnt to recognize relevant aspects of structures. You have learnt to see the models of predicate logic as pictures with certain properties. When you read $\forall xRxx$ you think 'reflexivity', and you see a picture with loops.*

Chapter 5

Predicate Logic: Syntax and Semantics

Overview So far, we have looked at the language of predicate logic through a series of translations from natural language. This is the usual skill taught in logic courses: you learn to think in two related ways, once with our ordinary language whose meaning you understand, and once with a related formal language that is a bit more precise. But what does this practice really mean? In particular, how do languages acquire meaning, and how can we analyze this in logic? This chapter focusses on the connection between the language of predicate logic and models for that language. First we list the ingredients for talking about things with predicate logic: domain of discourse, structured situations, interpretation, and the linking of variables to objects. Next, we will define syntax and semantics of predicate logic in a more formal way. Finally, we turn to the activity of proving theorems in predicate logic.

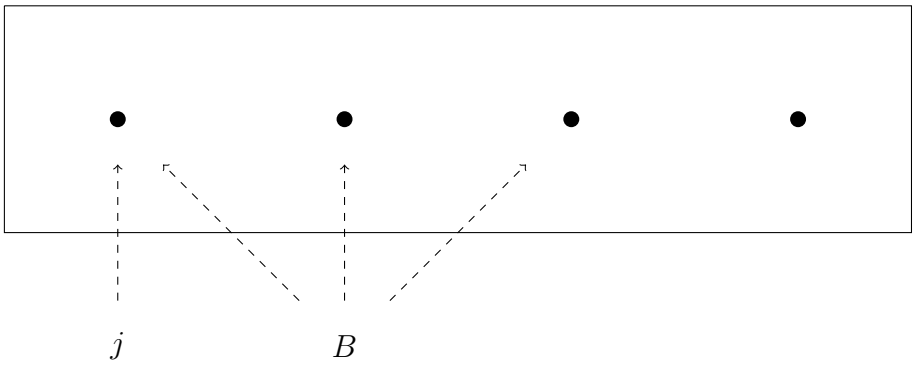
5.1 Domains of Discourse and Interpreting a Language

Semantics: structured situations, language, and interpretation A language describes situations, and the way this works really involves three ingredients. First, the situations described consist of objects with structure: properties, relations, and the like. Then there are the sentences of the language, that we could see in principle as syntactic code. This code acquires meaning by *connecting* it with structures, through a process of 'interpretation'. If you get a text in a language that you do not know, you just have the syntax. If you see a newspaper in a language you do not know with a front-page picture of Mount Etna erupting, you know ther situation and the language, but you still do not know what the message says, since you do not know its interpretation, its ties to that situation. This gives language an amazing versatility: one piece of code can talk about many situations under different interpretations, one situation can be described in many languages, if a situation does not fit what we say about it we can change it so that it does, and so on. In what follows, we develop these ideas for predicate logic, giving it the required flexibility.

Domains of discourse and interpretations We start with a simple observation. Interpreted predicate-logical formulas are about situations where a number of objects are present. One calls such a situation a 'domain of discourse' or 'universe of discourse'. In the translations that we discussed above the domain of discourse was implicitly assumed to be some set of human beings (when we were talking about John, Mary, Paul and their properties), or the set of natural numbers (when talking about being an even number, or being a prime number). We will keep giving examples of both kinds, to show you the range of predicate logic: from common sense situations to the world of science.

Next we need to say more precisely how an interpretation functions, linking the language to a given domain of discourse. In propositional logic, this was just done by a valuation, mapping proposition letters to truth values. But this will no longer do. For checking whether a statement saying that a certain object has a certain property, or that certain objects are in a certain relation is true we need something more refined. Instead of just saying that "John is boy" is assigned the value *true* by a valuation, we now need an interpretation for "John" and an interpretation for "being a boy".

Suppose the domain of discourse is a particular set of people. To interpret "John is a boy", or its predicate logical version Bj, in this given situation, we need a link between "John" or j and a particular individual in the domain that bears that name, and also a link between "being a boy" or B and a particular subset of the domain of discourse consisting of all its boys. Here is a picture that may help:



Constants are proper names Individual constants like j are like proper names of individuals in the domain of discourse. In other words: individual constants are interpreted as concrete individuals in the domain of discourse.

Unary predicates denote subsets Predicate letters that combine with just a single constant correspond to properties of individuals in the domain of discourse. In other words: a one-place predicate letter B is interpreted as a *subset* of the domain of discourse.

Binary predicates denote relations Predicate letters that combine with two constants correspond to relations between individuals in the domain of discourse. In other words, a two-place predicate letter R is interpreted as a *relation* on the domain of discourse.

Linking variables to objects Now how about variables? To interpret Rxy in a domain of discourse, it is not enough to know what the relation is that R refers to. We also need to have some way to link x and y to individuals in the domain of discourse. now variables do not refer to objects the same way that proper names do, since the object that they stand for can vary. Still, at any stage of interpreting a sentence, we may assume that variables link with objects in the domain, if only temporarily. Later on we will be more precise about how to do this, via so-called 'variable assignments'.

Interpreting complex sentences in stages So far, we have discussed how the basic alphabet of the language gets mapped to corresponding items in a structured situation. It is clear then how to interpret atomic statements saying that one or more objects stand in some relation. But of course, there are also much more complex sentences, formed using propositional operators and quantifiers. How do these get interpreted in a situation?

We will interpret the propositional operators just as before in our chapter on propositional logic. If you understood them there, you also do now. Next, we have already given informal readings for the quantifiers. In principe, this is enough: once you understand these meanings, every sentence, no matter how complex, can be understood *in stages* following its syntactic construction one step at a time. While we will sharpen this up later, for now, you will just learn to interpret complex sentences through a sequence of examples:

Change of discourse domain changes what a sentence says Summing up what we have said, sentences are true in situations, once a link is given betweenthe vocabulary and matching things in that situation. Here all aspects are equally important in determining whether given statements are true or false.

If the domain of discourse is human beings then *everyone likes Mary* gets the following interpretation: for each object d in the domain, the relation corresponding to the alphabet item 'likes' runs from d to the object whose name is 'Mary'. Whether this is true or not, depends on the situation: languages are not just used to state truths, they also serve for stating falsehoods. In fact, if we change the situation, the truth value may change as well: if the domain of discourse is human beings and pet animals then the given sentence says something quite different. And it will even be different if we just add one human object, say, a known misanthropist.

Exactly the same point holds for mathematical structures. The choice of domain of discourse can make the difference between truth and falsity of statements. Here is an example, for two domains of discourse. The set of natural numbers $\mathbb{N}$ is the infinite set

$$\{0, 1, 2, 3, \ldots\}.$$

The set of integer numbers $\mathbb{Z}$ is the infinite set

$$\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

The statement "there is no smallest number" (Exercise 5.1) is true when talking about $\mathbb{Z}$ but false when talking about $\mathbb{N}$.

Exercise 5.1 How would you express the statement "there is no smallest number" in predicate logic? (All you need is the binary predicate $<$ and logical connectives plus quantifiers.)

Exercise 5.2 Give a definition of being an odd number, using the predicate $x|y$ for "x divides y". The definition should run like this:

$$\forall x(Ox \leftrightarrow \cdots),$$

with the actual definition of "x is an odd number" at the place of the dots.

Exercise 5.3 Can you give a definition of being a prime number, using $x|y$ for "x divides y", the name 1 for the natural number one, and $x < y$ for "x is less than y"? (A prime number is a natural number greater than one that is not divided by any number greater than one and less than itself.)

Translations once more In this light, what is the status of the translations that we have been making? You can think of them as expressing more precisely what a natural language says about some relevant domain of discourse. In particular, if we can define precisely how our formulas are interpreted over given domains, we learn not only how this formal logical language works, but we also give a model of precision for the semantics of natural language or languages in science. This is in fact one of the greatest uses of predicate logic in a wide range of disciplines: a model for simple perspicuous syntax that can be interpreted semantically in a straightforward manner.

5.2 Syntax of Predicate Logic

You have now seen how the language of predicate logic works in a number of settings, and how it can describe various structures from mathematics to our daily world. Next, you may want to see how it works for the same purposes that you have seen in earlier chapters, such as information update and, in particular, inference. But before we do that, it is time to sharpen up things, and say exactly what we mean by the language and semantics of predicate logic.

The first step is an investment in 'formal grammar'. It looks a bit technical, but still a far cry from the complexities of a real natural language like English. The following grammar is a simple model of basics of the grammar of many languages, including even programming languages in computer science.

As we have seen, predicate logic is an extension of propositional logic with *structured basic propositions* and *quantification*.

- An atomic proposition consists of an n-ary predicate followed by n variables.
- A universally quantified formula consists of the symbol $\forall$ followed by a variable followed by a formula.
- An existentially quantified formula consists of the symbol $\exists$ followed by a variable followed by a formula.

- Other ingredients are as in propositional logic.

Other names for predicate logic are first-order logic or first-order predicate logic. 'First-order' indicates that the quantification is over entities (objects of the first order).

Formal grammar We will now give a formal definition of the language of predicate logic. The definition assumes that individual terms are either variables or constants. We use the 'BNF' (Backus-Naur Form) notation that was developed by computer scientists for defining programming languages, and that has also become popular among logicians for its compactness and perspicuity:

$$
\begin{array}{rcl}
\mathbf{v} & ::= & x \mid y \mid z \mid \cdots \\
\mathbf{c} & ::= & a \mid b \mid c \mid \cdots \\
\mathbf{t} & ::= & \mathbf{v} \mid \mathbf{c} \\
\mathbf{P} & ::= & P \mid Q \mid R \mid \cdots \\
\mathbf{Atom} & ::= & (\mathbf{t} = \mathbf{t}) \mid \mathbf{P}\,\mathbf{t}_1 \cdots \mathbf{t}_n \text{ where } n \text{ is the arity of } \mathbf{P} \\
\varphi & ::= & \mathbf{Atom} \mid \neg\,\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi) \mid \\
 & & \forall\,\mathbf{v}\,\varphi \mid \exists\,\mathbf{v}\,\varphi.
\end{array}
$$

Here is how you read such a schema. The lines separating items stand for a disjunction of cases. Then, e.g., the clause for formulas says that a formula is either an atomic formula, or a negation of something that is already a formula, and so on. The class of formulas is then understood to be the smallest set of symbol sequences that is generated in this way.

Here is how this works more concretely. The following strings are examples of formulas of this predicate logical language:

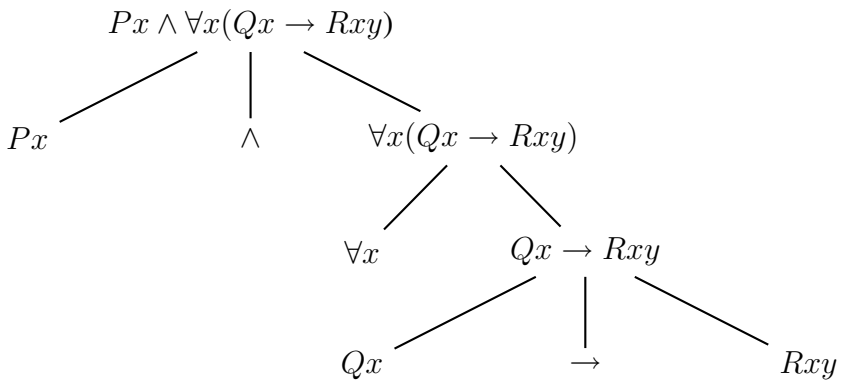
- $\neg Px$
- $(Px \wedge Qy)$
- $((Px \wedge Qy) \vee Rxz)$
- $\forall x Rxx$
- $\exists x(Rxy \wedge Sxyx)$
- $\forall x \forall y \neg((x = y) \wedge \neg(y = x))$

The semantics for this formal syntax will be given in the next section.

Note that the official definition of the language has more parentheses than we have used in the examples so far. It is a useful convention to omit parentheses when they are not essential for disambiguation.

Free and bound variables In talking about predicate logical formulas we want to be able to distinguish between the variable occurrences that are *bound* by a quantifier occurrence in that formula and the variable occurrences that are not. Binding is a syntactic notion, and it can simply be stated as follows: in a formula $\forall x\,\varphi$ (or $\exists x\,\varphi$), the quantifier occurrence binds all occurrences of x in φ that are not bound by any quantifier occurrence $\forall x$ or $\exists x$ inside φ.

As an example, consider the formula $Px \wedge \forall x(Qx \to Rxy)$. Here is its syntax tree:



The occurrence of x in Px is free, because it is not in the scope of a quantifier; the other occurrences of x (the one in Qx and the one in Rxy) are bound, because they are in the scope of $\forall x$. An occurrence of x is bound in φ if there is some quantifier occurrence that binds it, it is free otherwise.

Exercise 5.4 Give the bound occurrences of x in the following formula.

$$\exists x(Rxy \vee Sxyz) \wedge Px$$

Exercise 5.5 Which quantifier occurrence binds which variable occurrences?

$$\forall x(Px \to \exists x Rxx).$$

A predicate logical formula is called *open* if it contains at least one variable occurrence which is free (not bound by any quantifier); it is called *closed* otherwise. A closed predicate logical formula is also called a predicate logical *sentence*. Thus, $Px \wedge \exists x Rxx$ is an open formula, but $\exists x(Px \wedge \exists x Rxx)$ is a sentence.

Exercise 5.6 Which of the following formulas are sentences?

(1) $\forall x Px \vee \forall x Qx$,

(2) $Px \vee \forall x Qx$,

(3) $Px \vee Qx$,

(4) $\forall x(Px \to Rxy)$,

(5) $\forall x(Px \rightarrow \exists y Rxy)$.

Suppose we have a formula φ that has no free occurrences of x. Then the quantifications $\forall x\varphi$ and $\exists x\varphi$ are called *vacuous*. Vacuous quantifiers bind all free occurrences of a variable in a formula that does not have such free occurrences.

Exercise 5.7 Which quantifications are vacuous? Replace each formula with vacuous quantification by an equivalent formula without vacious quantification.

(1) $\forall x \exists x Rxx$.

(2) $\forall x \exists y Rxx$.

(3) $\forall x \exists y Rxy$.

(4) $\forall x \exists y Ryy$.

Substitution Now let us talk about the ways we have for talking about objects. A *term* is either a constant or a variable. Thus, x is a term, and the constant c is also a term.

Here is a function that substitutes a term t for the occurrences of a variable v in a term s, with notation s_t^v:

$$
\begin{array}{rcl}
c_t^v & := & c \\
v_t^v & := & t \\
v'^v_t & := & v' \text{ for } v' \text{ different from } v
\end{array}
$$

Note that this follows the definition of terms in the syntax of predicate logic

Here is how this definition works:

x_z^y is equal to x, x_c^x is equal to c, x_y^x is equal to y.

Next, we use the definition of s_t^v to define the widely used notion of substitution of a term t for all free occurrences of a variable v in a formula φ, with notation φ_t^v. This tells us how the property expressed by φ holds of the object denoted by t. This time, the definition follows the above syntactic definition of the formulas of predicate logic:

$$
\begin{array}{rcl}
(t_1 = t_2)^v_t & := & (t_{1_t}{}^v = t_{2_t}{}^v) \\
(Pt_1 \cdots t_n)^v_t & := & Pt_{1_t}{}^v \cdots t_{n_t}{}^v \\
(\neg\varphi)^v_t & := & (\neg\varphi^v_t) \\
(\varphi_1 \wedge \varphi_2)^v_t & := & (\varphi_{1_t}{}^v \wedge \varphi_{2_t}{}^v) \\
(\varphi_1 \vee \varphi_2)^v_t & := & (\varphi_{1_t}{}^v \vee \varphi_{2_t}{}^v) \\
(\varphi_1 \rightarrow \varphi_2)^v_t & := & (\varphi_{1_t}{}^v \rightarrow \varphi_{2_t}{}^v) \\
(\varphi_1 \leftrightarrow \varphi_2)^v_t & := & (\varphi_{1_t}{}^v \leftrightarrow \varphi_{2_t}{}^v) \\
(\forall v_1\varphi)^v_t & := & \begin{cases} \forall v_1\varphi & \text{if } v_1 \text{ equals } v, \\ \forall v_1\varphi^v_t & \text{otherwise.} \end{cases} \\
(\exists v_1\varphi)^v_t & := & \begin{cases} \exists v_1\varphi & \text{if } v_1 \text{ equals } v, \\ \exists v_1\varphi^v_t & \text{otherwise.} \end{cases}
\end{array}
$$

Defining notions by recursion A remarkable feature of this definition is the use of *recursion*. We explain what $(\neg\varphi)^v_t$ means by assuming that we already know what φ^v_t means, and so on. This works because the definition follows the same pattern as the construction pattern that was used for defining the formulas in the first place.

You are invited to check whether you understand the definition of substitution of a term for a variable in a formula by carrying out the following exercise:

Exercise 5.8 Give the results of the following substitutions:

(1) $(Rxx)^x_c$.

(2) $(Rxx)^x_y$.

(3) $(\forall xRxx)^x_y$.

(4) $(\forall yRxx)^x_y$.

(5) $(\exists yRxy)^x_z$.

Alphabetic variants Using the notion of a substitution, we can say what it means that a formula is an *alphabetic variant* of another formula. This is useful since we often want to switch bound variables while retaining the essential structure of a formula.

Suppose φ does not have occurrences of z, and consider φ^x_z, the result of replacing all free occurrences of x in φ by z. Note that $\forall z\varphi^x_z$ quantifies over variable z in all places where $\forall x\varphi$ quantifies over x. We say that $\forall x\varphi$ and $\forall z\varphi^x_z$ are *alphabetic variants*.

Here are some examples: $\forall xRxx$ and $\forall yRyy$ are alphabetic variants, and so are $\forall x\exists yRxy$ and $\forall z\exists xRzx$. The quantification patterns are the same, although different

variable bindings are employed to express them. In the next section we will see that alphabetic variants always have the same meanings.

This section has given you some basic parts of the grammar of predicate logic. There are some other grammatical notions that are widely used, such as the 'quantifier depth' of a formula, being the longest 'nesting' of quantifiers that take scope over each other inside the formula. But for the moment, you have seen enough precision, and we can proceed.

5.3 Semantics of Predicate Logic

Now it is time to say in more detail how the preceding language gets interpreted on semantic structures.

For convenience, we will assume that all terms are either variables or individual constants, leaving the extension with functional terms for the end of the section.

Structures Defining the semantics of predicate logic involves linking formulas to situations that are somehow described for those formulas. So first we need to fix what the appropriate situations are.

What should a relevant situation or structure for the predicate letters P, R, S look like? (Let us suppose P has arity 1, R has arity 2, and S has arity 3.)
Such a structure should at least contain a domain of discourse D consisting of individual objects, with an interpretation for P, R and S. These interpretations are given by a function I that should be of the right kind for the predicates:

$$\begin{aligned} I(P) &\subseteq D, \\ I(R) &\subseteq D \times D, \\ I(S) &\subseteq D \times D \times D. \end{aligned}$$

Here $D \times D$ (sometimes also written as D^2) is the set of all pairs of elements from D, and $D \times D \times D$ (also written as D^3 sometimes) is the set of all ordered triples of elements from D.

Here is a useful notation:

We write $I(P)$ as P_I, $I(R)$ as R_I, and $I(S)$ as S_I. This gives a clear distinction between a predicate letter P and the predicate P_I that interprets this predicate letter.

Figure 5.1 gives a domain with a P_I and R_I. Shading marks objects with property P_I, $\rightarrow$ indicates that two objects are R_I-related, a $\longleftrightarrow$ link indicates that the R_I-relation runs in both directions, and loops indicate that an object is R_I-related to itself.

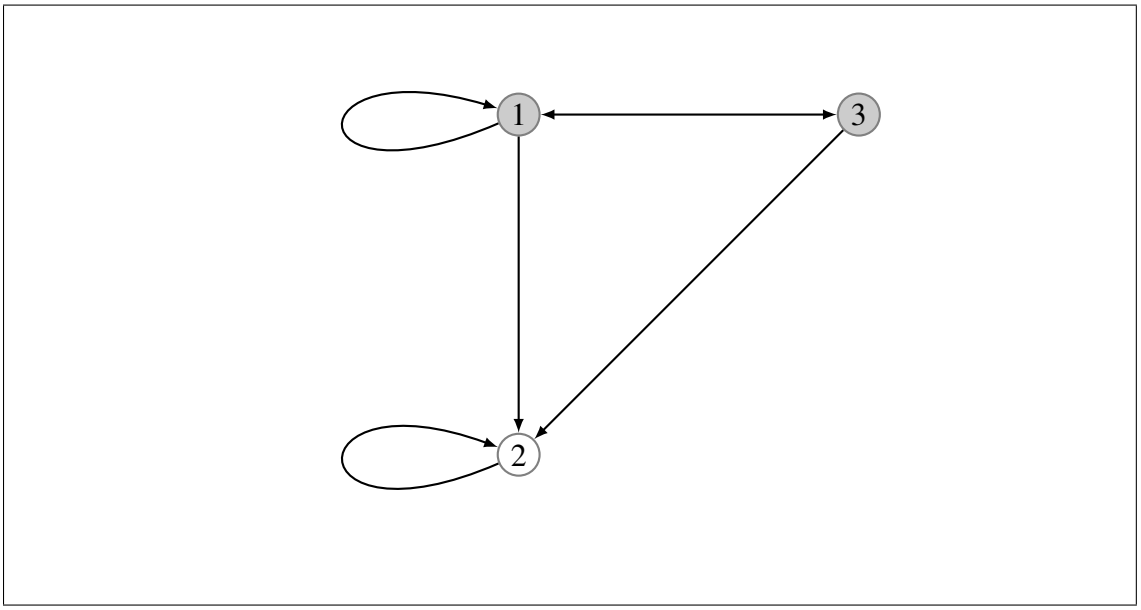


Figure 5.1: A model with interpretations for a unary and a binary relation symbol.

Languages and models Every time we fix a particular set of predicates and constants we fix a *predicate-logical language*. Examples that we have seen are the 'languages' for talking about example graph structures above.

In other words, a set of relation symbols, with their arities given, plus a set of constants, specifies a predicate logical language L. Here is the matching semantic notion:

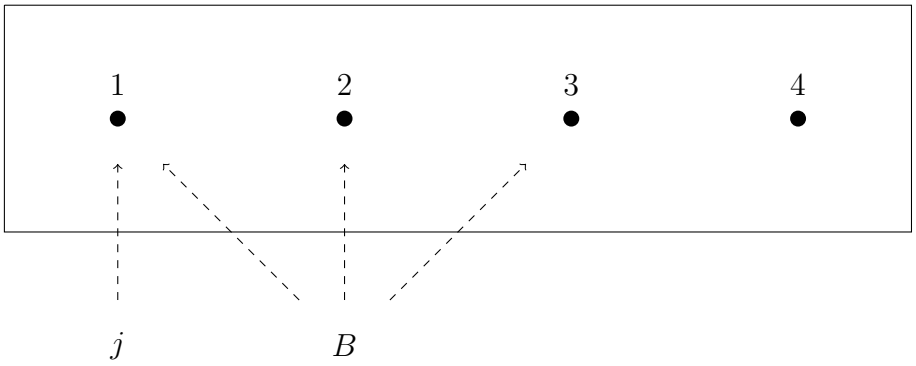
> A structure $M = (D, I)$ with a non-empty domain D of objects plus an interpretation function mapping the relation symbols of L to appropriate predicates on D, and the constants of L to objects in D is called a *model for* L.

Example: interpreting predicates and constants Looking at the example case above, the language is given by the relation symbols P and R. In Figure 5.1 we numbered the nodes, so we can say that the domain consists of the set $\{1, 2, 3\}$. The interpretation function I is given by:

$$P_I = \{1, 3\} \text{ and } R_I = \{(1, 1), (1, 2), (2, 2), (3, 1), (3, 2)\}.$$

If there are also individual constants in the language then the distinction is between the constant or name c and its interpretation or denotation c_I.

Recall our earlier example:



Here j is a constant of the language, and B is a 1 place predicate logical symbol. The interpretation of j is the individual j_I, which is object 1 in the model. The interpretation of B is the set B_I, which is $\{1, 2, 3\}$ in the model.

Now we have dealt with predicates and constants. It remains to make sense of our final important piece of syntax: *variables* for objects. Given a structure with interpretation function $M = (D, I)$, we can interpret all predicate logical formulas, provided we know how to deal with the values of individual variables. This calls for a new somewhat technical notion, that also has independent motivations:

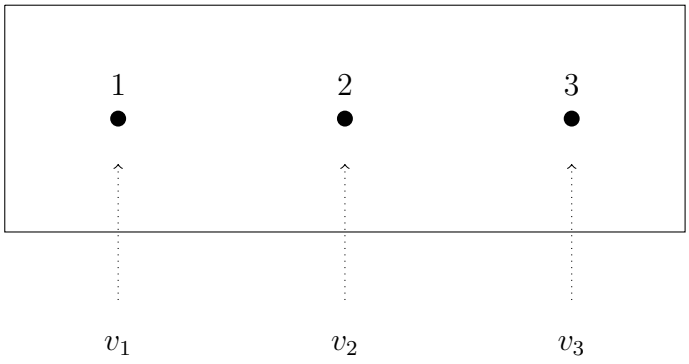
Variable assignments Let V be the set of variables of the language.:

A function $g\colon V \to D$ is called a *variable assignment*.

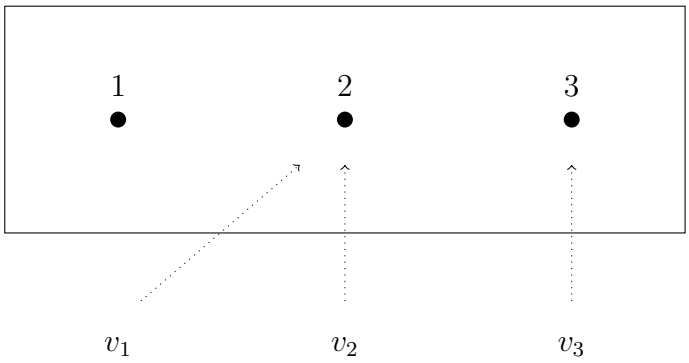
Making changes to variable assignments In the definition of the semantics of quantifiers it will be important to make certain mimimal changes to variable assignments:

We use $g[v := d]$ for the variable assignment that is like s except for the fact that v gets value d (where g might have assigned a different value).

For example, let $D = \{1, 2, 3\}$, and let $V = \{v_1, v_2, v_3\}$. Let g be given by $g(v_1) = 1, g(v_2) = 2, g(v_3) = 3$. See the following picture, where the g links are indicated by dotted arrows.



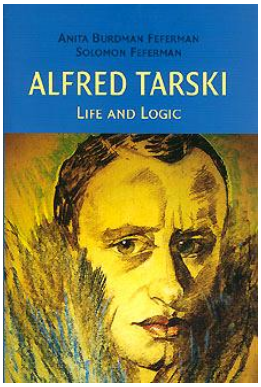
Then $g[v_1 := 2]$ is the variable assignment that is like g except for the fact that v_1 gets the value 2, i.e. the assignment that assigns 2 to v_1, 2 to v_2, and 3 to v_3. See the next picture, where the dotted arrow for v_1 has changed, indicating that the new assignment has a new value for v_1.



You will see in a moment how this works:

Truth Definition At last, we are ready for the core of treatment of the semantics of predicate logic: the definition of truth of a formula in a model. This truth definition for predicate logic is due to Alfred Tarski (1901–1983):

Alfred Tarski



Cover of Biography of Tarski by Anita and Sol Feferman, CUP 2004

Let M be a model for predicate logical language L (i.e., a pair consisting of a domain D and an interpretation function I), let g be a variable assignment for L in M, let φ be a formula of L. We will assume that L has unary predicate symbols P, binary predicate symbols R, ternary predicate symbols S, constant symbols c, but no function symbols.

Now we are ready to define the notion

$M \models_g \varphi$, for φ *is true in* M *under assignment* g,

We also sometimes say that g *satisfies* φ *in model* M.

Values of terms First, we take care of the interpretation of the terms of the language: variables are interpreted using the variable assignment, constants using the interpretation function. The following definition brings these together:

$$\begin{array}{lcl} [\![v]\!]^g_I & = & g(v) \\ [\![c]\!]^g_I & = & c_I \end{array}$$

Truth in a model Next, we use this to define truth of a formula in a model, given some variable assignment g. The variable assignment is needed to deal with free variables that may occur in the formula. Note that the definition follows the structure of the definition of predicate logical formulas in a stepwise manner:

$M \models_g t_1 = t_2$	iff	$[\![t_1]\!]^g_I = [\![t_2]\!]^g_I$
$M \models_g Pt_1 \cdots t_n$	iff	$([\![t_1]\!]^g_I, \ldots, [\![t_n]\!]^g_I) \in P_I$
$M \models_g \neg\varphi$	iff	it is not the case that $M \models_g \varphi$.
$M \models_g \varphi_1 \wedge \varphi_2$	iff	$M \models_g \varphi_1$ and $M \models_g \varphi_2$
$M \models_g \varphi_1 \vee \varphi_2$	iff	$M \models_g \varphi_1$ or $M \models_g \varphi_2$
$M \models_g \varphi_1 \rightarrow \varphi_2$	iff	$M \models_g \varphi_1$ implies $M \models_g \varphi_2$
$M \models_g \varphi_1 \leftrightarrow \varphi_2$	iff	$M \models_g \varphi_1$ if and only if $M \models_g \varphi_2$
$M \models_g \forall v\varphi$	iff	for all $d \in D$ it holds that $M \models_{g[v:=d]} \varphi$
$M \models_g \exists v\varphi$	iff	for at least one $d \in D$ it holds that $M \models_{g[v:=d]} \varphi$

What we have presented just now is a recursive definition of truth for predicate logical formulas in given models with a given assignment. Note in particular how the clauses for the quantifiers involve changing assignments.

Another important aspect is this. How can this small finite set of rules deal with all, infinitely many, formulas, that may contain highly complex combinations of quantifiers?

Do not these require lots of further clauses? The answer is that no such thing is needed: one explanation of single quantifiers suffices, provided we *repeat* it every time we encounter a quantifier as put there in the construction of the formula. Thus, if you think back of our earlier many-quantifier formulas on graphs, their meaning is completely described by the mechanism given here: *meaning of single words, repeated in tandem with formula structure*.

This completes our formal definition of the semantics. We now add a few further technical issues.

Closed formulas If we evaluate closed formulas (formulas without free variables), it is easy to see that the initial assignment g becomes irrelevant, so for a closed formula φ we can simply put $M \models \varphi$ iff there is *some* assignment g with $M \models_g \varphi$. Equivalently, we could say that $M \models \varphi$ iff for *all* assignments g it holds that $M \models_g \varphi$. Since the truth or falsity of a closed formula does not depend on the initial assignment, it makes no difference. And indeed, intuitively, closed formulas are just true or false.

Still, the truth definition makes essential use of assignments, and this is inevitable. For instance, when we apply the truth definition above to a sentence, say, $\forall x(Px \to \exists y Rxy)$, then the clause for dealing with the universal quantifier makes reference to the notion of truth for the formula $Px \to \exists y Rxy$, which is an open formula. In order to determine whether the latter is true we need to be told which object x denotes.

Exercise 5.9 Assume that we can extend the language with a set A of proper names for objects, in such a way that every object in D is named by an element from A, by means of an extension I' of the interpretation function I of the model. In other words, we have for all $d \in D$ that there is some $\hat{d} \in A$ with $\hat{d}_{I'} = d$. Give an alternative truth definition for predicate logic that does not use assignments, but these extra names, plus the notion of substituting names from A for variables. Use the function $\varphi^v_{\hat{d}}$ that was defined on page 5-8.

The importance of open formulas Sometimes, open formulas are just considered a nuisance, since they speak about 'transient objects' given by the current assignment. In mathematics, a useful convention is to interpret the free variables in open formulas *universally*. This convention is useful because it allows us to write, for instance, algebraic equations without universal quantifiers. Say, associativity of the $+$ operator can now be expressed as:

$$x + (y + z) = (x + y) + z.$$

The understanding is that this is supposed to hold for all choices of x, y, z.

Exercise 5.10 Let M be the model pictured in Figure 5.1 (page 5-10). Which of the following statements are true?

(1) $M \models \exists x(Px \wedge Rxx)$

(2) $M \models \forall x(Px \to Rxx)$.

(3) $M \models \forall x(Px \rightarrow \exists yRxy)$

(4) $M \models \exists x(Px \wedge \neg Rxx)$.

(5) $M \models \exists x(\exists yRyx \wedge Rxx)$.

(6) $M \models \forall x(\exists yRyx \rightarrow Rxx)$

(7) $M \models \forall x(Rxx \rightarrow \exists yRxy)$.

But formulas with free variables are not a nuisance. In fact, they are highly important in their own right. For instance, in computer science, they are often used to *query* a given model, say, a data base with information about individuals. In such a setting, the question $\varphi(x)$? ("Which objects are φ-ing?") asks for all objects in the moel that have the property φ.

Extending the semantics to function symbols In order to extend the truth definition to structured terms (terms containing function symbols), we have to assume that the interpretation function I knows how to deal with a function symbol. If f is a one-place function symbol, then I should map f to a unary operation on the domain D, i.e. to a function $f_I \colon D \rightarrow D$. In general it holds that if f is an n-place function symbol, then I should map f to an n-ary operation on the domain D, i.e. to a function $f_I \colon D^n \rightarrow D$. We use C for the set of zero-place function symbols; these are the constants of the language, and we have that if $c \in C$ then $c_I \in D$.

The interpretation of terms, given an assignment function g for variables and an interpretation function I for constants and function symbols, is now defined as follows. Note the by now familiar use of recursion in the definition.

$$[\![t]\!]^g_I = \begin{cases} t_I & \text{if} \quad t \in C, \\ g(t) & \text{if} \quad t \in V, \\ f_I([\![t_1]\!]^g_I, \ldots, [\![t_n]\!]^g_I) & \text{if} \quad t \text{ has the form } f(t_1, \ldots, t_n). \end{cases}$$

5.4 Valid Laws and Valid Consequence

Valid laws A predicate logical sentence φ is called *logically valid* if φ is true in every model. Notation for "φ is logically valid" is $\models \varphi$. From the convention that the domains of our models are always non-empty it follows that $\models \forall x\varphi \rightarrow \exists x\varphi$, for all φ with at most the variable x free.

Exercise 5.11 Which of the following statements are true?

(1) $\models \exists xPx \vee \neg\exists xPx$.

(2) $\models \exists xPx \vee \forall x\neg Px$.

(3) $\models \forall xPx \vee \forall x\neg Px$.

(4) $\models \forall x Px \vee \exists x \neg Px$

(5) $\models \exists x \exists y Rxy \rightarrow \exists x Rxx$.

(6) $\models \forall x \forall y Rxy \rightarrow \forall x Rxx$.

(7) $\models \exists x \exists y Rxy \rightarrow \exists x \exists y Ryx$

(8) $\models \forall x Rxx \vee \exists x \exists y \neg Rxy$.

(9) $\models \forall x Rxx \rightarrow \forall x \exists y Rxy$

(10) $\models \exists x Rxx \rightarrow \forall x \exists y Rxy$

(11) $\models \forall x \forall y \forall z((Rxy \wedge Ryx) \rightarrow Rxy)$.

Valid consequence More generally, an important aim of any logical language is to study the process of valid reasoning in that language. When can we draw a conclusion ψ from premises $\varphi_1, \ldots, \varphi_n$? As in earlier chapters, our answer is this. Valid consequence says that the premises can never be true while the conclusion is false, or in other words, when the truth of the premises always brings the truth of the conclusion in its wake.

Now that we have a fully precise notion of truth for predicate logic we can make the intuitive notion of valid consequence fully precise too:

> We say that a predicate logical sentence ψ *logically follows from* a sentence φ (alternatively, φ *logically implies* ψ) if every model which makes φ true also makes ψ true. Notation for 'φ logically implies ψ' is $\varphi \models \psi$.

The art of testing validity How do we judge statements of the form $\varphi \models \psi$. (where φ, ψ are closed formulas of predicate logic)? It is clear how we can refute the statement $\varphi \models \psi$, namely, by finding a *counterexample*. A counterexample to $\varphi \models \psi$ is a model M with $M \models \varphi$ but not $M \models \psi$, or in abbreviated notation: $M \not\models \psi$. Here are some example questions about valid consequence in predicate logic.

Exercise 5.12 Which of the following statements hold? If a statement holds, then you should explain why. If it does not, then you should give a counterexample.

(1) $\forall x Px \models \exists x Px$

(2) $\exists x Px \models \forall x Px$.

(3) $\forall x Rxx \models \forall x \exists y Rxy$.

(4) $\forall x \forall y Rxy \models \forall x Rxx$.

(5) $\exists x \exists y Rxy \models \exists x Rxx$

(6) $\forall x \exists y Rxy \models \forall x Rxx$.

(7) $\exists y \forall x Rxy \models \forall x \exists y Rxy$

(8) $\forall x \exists y Rxy \models \exists y \forall x Rxy$.

(9) $\exists x \exists y Rxy \models \exists x \exists y Ryx$.

(10) $\forall x Rxx \models \forall x \exists y Rxy$.

(11) $\exists x Rxx \models \exists x \exists y Rxy$.

We can make this slightly more general by allowing sets of more than one premise. Assume that $\varphi_1, \ldots, \varphi_n, \psi$ are closed formulas of predicate logic. We say that ψ logically follows from $\varphi_1, \ldots, \varphi_n$, or, formally, that $\varphi_1, \ldots, \varphi_n \models \psi$, if for every model M for the language with the property that $M \models \varphi_1$ and ... and $M \models \varphi_n$ it is the case that $M \models \psi$.

Exercise 5.13 Which of the following hold?

(1) $\forall x \forall y (Rxy \rightarrow Ryx), Rab \models Rba$

(2) $\forall x \forall y (Rxy \rightarrow Ryx), Rab \models Raa$

Maybe these exercises were not too difficult, but testing for predicate-logical validity is much harder than what we have seen with the truth tables for propositional logic in Chapter 2. We will devote a whole chapter later in this course to general methods for testing validity for our system.

A word on mathematical background Here are a few reasons why predicate-logical validity is much harder than that for propositional logic. One is that there are so many models for the language: namely, infinitely many (just think of all the different sizes an object domain can have). Another source of complexity is that models may be both finite (like most of our examples so far), or infinite (like the mathematical number systems we have also mentioned as examples occasionally). Together, this means that there is no finite enumeration of all relevant cases to be checked, the way we did in truth tables. In fact, it can even be *proved mathematically* that the notion of validity for predicate logic is harder than anything we have seen before: it is *undecidable*, that is, there is just no mechanical method that tests it automatically. This is the computational price that we pay for the much greater expressive power of predicate logic.

5.5 Proof

Here is an axiomatisation for predicate logic with equality.

(1) All propositional tautologies.

(2) $\forall x \varphi(x) \rightarrow \varphi(t)$. Condition: no variable in t occurs bound in φ.

(3) $\forall x(\varphi \to \psi) \to (\varphi \to \forall x\psi)$. Condition: x is not free in φ.

(4) $\forall x\ x = x$.

(5) $\forall x \forall y(x = y \to (\varphi \to \varphi^x_y)$. Condition: φ does not bind the variable y. φ^x_y is the result of replacing some or all free occurrences of x in φ by occurrences of y.

As deduction rule we again have Modus Ponens, plus a second rule called *universal generalization*:

- MP: from φ and $\varphi \to \psi$, infer ψ.
- UGEN: from φ infer $\forall x\varphi$, on condition that x does not occur free in any premise which has been used in the proof of φ.

We derive existential generalization from this, as follows.

1.	$\forall x\neg\varphi(x) \to \neg\varphi(t)$	axiom 2
2.	$(\forall x\neg\varphi(x) \to \neg\varphi(t)) \to (\varphi(t) \to \neg\forall x\neg\varphi(x))$	propositional tautology
3.	$\varphi(t) \to \neg\forall x\neg\varphi(x)$	from 1,2 by MP
4.	$\varphi(t) \to \exists x\varphi(x)$	from 3, by def of $\exists$

Next, assume that we have $\varphi(t)$. Then we can derive $\exists x\varphi(x)$, as follows:

1.	$\varphi(t)$	assumption
2.	$\varphi(t) \to \exists x\varphi(x)$	this we just proved
3.	$\exists x\varphi(x)$	MP from 1,2

What this means is that the following rule of existential generalization is derivable:

- EGEN: From $\varphi(t)$, infer $\exists x\varphi(x)$.

Using this proof system one can reason about all models of predicate logic, for a given choice of predicate letters. But it is also common to *add* axioms, in order to talk about specific topics. As you may already have guessed, predicate logic can be used to talk about *anything*.

5.6 Outlook — Logical Theories

Mathematical Theories: Arithmetic Suppose we want to talk about addition on the natural numbers. This is the aritmethic of first grade, the stuff that comes before the tables of multiplication. The domain of discourse is

$$\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}.$$

The predicate logical language that we can use for this has a constant for zero (we will use $\mathbf{0}$ for this; the intention is to use $\mathbf{0}$ as a name for the number 0), and two function symbols, one for taking the successor of a number (we will use s for this), and the familiar $+$ for addition. The successor of the number is the number immediately following it.

We have four axioms and an axiom scheme (a recipe for constructing further axioms).

The first axiom states that 0 is not the successor of any number:

$$\forall x \neg sx = \mathbf{0}. \tag{PA1}$$

The second axiom states that different numbers cannot have the same successor, or stated otherwise, that if $sx = sy$ then $x = y$.

$$\forall x \forall y (sx = sy \rightarrow x = y). \tag{PA2}$$

Note that $sx = sy \rightarrow x = y$ is equivalent to $x \neq y \rightarrow sx \neq sy$, so this does indeed express that different numbers have different successors.

The third axiom expresses that adding zero to a number doesn't change anything:

$$\forall x \; x + \mathbf{0} = x. \tag{PA3}$$

The fourth axiom expresses a sum of the form $x + sy$ as the successor of $x + y$:

$$\forall x \forall y \; x + sy = s(x + y). \tag{PA4}$$

The third and fourth axiom together define addition for any pair of numbers x and z, as follows. Either z is equal to 0, and then apply (PA3) to see that the outcome is x, or z is the successor of another number y and then apply (PA4) to see that the outcome is the successor of $x + y$. This is called a *recursive* definition, with recursion on the structure of y.

Below we will see how the recursion definition of addition will help us in constructing inductive proofs of facts about the addition operation.

The final axiom takes the form of a scheme. Assume that $\varphi(x)$ is a formula with x free. Then the following is an axiom:

$$(\varphi(\mathbf{0}) \wedge \forall x(\varphi(x) \rightarrow \varphi(sx))) \rightarrow \forall y \varphi(y). \tag{PA5–scheme}$$

This axiom scheme expresses mathematical induction on the natural numbers. If you are unfamiliar with mathematical induction you might wish to consult section A.6 in the Appendix.

The theory of arithmetic defined by (PA1) — (PA5-scheme) is known as *Presburger arithmetic*, after Mojzesz Presburger, student of Alfred Tarki.

Mojzesz Presburger
(picture from his death certificate in the database of YAD VASHEM, the Holocaust Martyrs' and Heroes' Remembrance Authority)

In the language of Presburger arithmetic one cannot express properties like being a prime number, or define the relation of divisibility. But one can express properties like being even, being odd, being a threefold, and so on. As an example, the following formula of Presburger arithmetic expresses the property of being even:

$$\exists y\ y + y = x. \qquad (x \text{ is even})$$

And the property of being odd:

$$\exists y\ s(y + y) = x. \qquad (x \text{ is odd})$$

Exercise 5.14 Express the property of x of being a threefold in the language of Presburger arithmetic.

Adding multiplication Presburger arithmetic is the restriction of the arithmetical theory of addition and multiplication to just addition. If we extend the language with an operator $\cdot$ for multiplication, the properties of multiplication can be captured by a recursive definition, as follows:

$$\forall x\ x \cdot \mathbf{0} = \mathbf{0}. \qquad \text{(MA1)}$$

$$\forall x \forall y\ x \cdot sy = (x \cdot y) + x. \qquad \text{(MA2)}$$

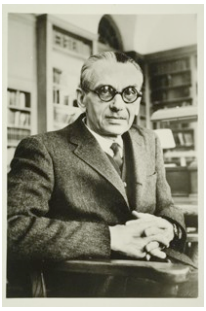
This defines $x \cdot y$ by recursion on the structure of y: MA1 gives the base case, MA2 the recursion case. In still other words: MA1 gives the first row in the multiplication table for x, and MA2 gives the recipe for computing the next row from wherever you are in the table, as follows:

$$\begin{array}{l} 0 \\ x \\ x + x \\ x + x + x \\ \vdots \end{array}$$

As you may have realized in second grade, the multiplication tables are constructed by means of addition.

Next, let IND be the induction axiom scheme for the language extended with multiplication.

Giuseppe Peano



Kurt Gödel

The theory consisting of PA1 – PA4 plus MA1, MA2 and IND is called *Peano arithmetic*, after Giuseppe Peano (1858–1932). Here the logical situation is dramatically different from the case of Presburger arithmetic. One of the most famous results in logic is Kurt Gödel's incompleteness proof (1931) for this predicate logical version of arithmetic:

> Any predicate logical theory T that is an extension of Peano arithmetic is incomplete: it is possible to find formulas in T that make true statements about addition and multiplication on the natural numbers, but that cannot be proved in T.

Five years later Alonzo Church and Alan Turing proved (independently) that the predicate logical language of arithmetic is undecidable, and more generally, that any predicate logical language with at least one 2-place relation symbol is undecidable (see section 13.2 below, and Chapter 11).

5.7 Outlook — Predicate Logic in Programming

The part of predicate logic without the quantifiers is called the Boolean part of predicate logic: we only allow the boolean connectives that you already know from propositional logic to form complex formulas. So the only difference with propositional logic is that we now talk about *objects* satisfying certain *properties* and taking part in certain *relations*. We have replaced proposition letters such as p, q, etcetera, by atomic formulas such as Bj or Rxy.

Booleans in programming The Boolean part of predicate logic is part and parcel of almost any programming language. Here is an example of a Ruby program (Ruby is a popular scripting language: see `http://www.ruby-lang.org/`):

```
if not (guess != secret1 && guess != secret2)
  print "You win."
else
  print "You lose."
end
```

The expression `not (guess != secret1 && guess != secret2)` is called a Boolean. This is in fact of the form $\neg(\neg p \wedge \neg q)$, and indeed, the program can be simplified by using a propositional equivalence:

$$\neg(\neg p \wedge \neg q) \longleftrightarrow p \vee q.$$

This yields a more straightforward version of the same program:

```
if guess == secret1 || guess == secret2
  print "You win."
else
  print "You lose."
end
```

In fact, the conditions used in a Boolean can also contain variables, like in predicate logic. See exercise **??**.

Exercise 5.15 (Only if you have some programming experience.) Simplify the following Ruby statement by writing the Boolean condition without negation:

```
if not (x < y && y < z)
   print "You win."
else
   print "You lose."
end
```

Domains of discourse in programming The domains of discourse of predicate logic also play an important role in programming. In many programming languages, when a programming variable is declared, the programmer has to specify what kind of thing the variable is going to be used for.

In programming terms: variable declarations have to include a *type declaration*. A Java declaration `int i j max` declares three variables of type *int*, and an *int* in Java is a signed integer that can be stored in 32 bits: Java *int* variables range from $-2^{31} = -2,147,483,648$ to $2^{31} - 1 = 2,147,483,647$. What this means is that the type declarations fix the domains of discourse for the programming variables.

In programming, *types* are important because once we know the type of a variable we know how much storage space we have to reserve for it. A Java *int* can be stored in four bytes (32 bits).

Statement of pre- and postconditions Predicate logic can be used to state pre- and postconditions of computational procedures. Consider the following function (in Ruby) for computing an integer output from an integer input, together with a precondition and a postcondition:

```
# precondition: n >= 0
def ld(n)
 d = 2
 while d**2 <= n
   return d if n.remainder(d) == 0
   d = d + 1
 end
 return n
end
# postcondition:
# ld(n) is the least positive integer that divides n
```

The precondition is a predicate logical formula without quantifiers:

$$n \geq 0.$$

The postcondition can be translated into a predicate logical formula, using $|$ for divides, as follows (we assume that the domain of discourse is $\mathbb{Z}$, the domain of integers):

$$\mathrm{ld}(n)|n \wedge \forall m(0 < m < \mathrm{ld}(n) \rightarrow \neg m|n).$$

The intended meaning is: if the input of the function satisfies the precondition then the output of the function will satisfy the postcondition.

Pre- and postconditions can be used for proving computational procedures correct. Examples of such correctness reasoning will be discussed in section 7.9.2.

Explicit statement of pre- and postconditions is also the key ingredient of a programming style called *design by contract*, the idea being that the precondition makes explicit what a computational procedure may assume about its input, while the postcondition is a statement about what the procedure is supposed to achieve, given the truth of the precondition. Preconditions state rights, postconditions list duties.

Exercise 5.16 (You should only attempt this if you have some programming experience.) Suppose the procedure for `ld` would have been stated like this:

```
# precondition: n >= 0
def ld(n)
 d = 2
 while d**2 < n
   return d if n.remainder(d) == 0
   d = d + 1
 end
```

```
 return n
end
# postcondition:
# ld(n) is the least positive integer that divides n
```

Why would this version of the program not satisfy the contract anymore? Try to find a simple counterexample.

5.8 Outlook — Decidable Parts of Predicate Logic

Predicate logic with at least one 2-place relation symbol is undecidable, as Church and Turing showed in the 1930s. But for monadic predicate logic, where all predicates are unary — things are better. In this section we will explain why.

Consider again syllogistic reasoning patterns. These patterns use three 1-place predicates, and we have seen that counterexamples to syllogistic validity have eight relevant regions, for the following sets:

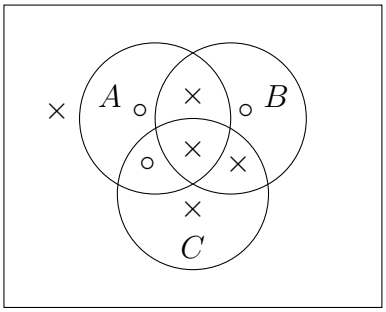
- the things that have none of the properties A, B, C, i.e., the things in the set $\overline{A} \cap \overline{B} \cap \overline{C}$.
- the things that have property A, but neither of B, C, i.e., the things in the set $A \cap \overline{B} \cap \overline{C}$.
- the things that have property B, but neither of A, C, i.e., the things in the set $B \cap \overline{A} \cap \overline{C}$.
- the things that have property C, but neither of A, B, i.e., the things in the set $C \cap \overline{A} \cap \overline{B}$.
- the things that have properties A and B but that lack C, i.e., the things in the set $A \cap B \cap \overline{C}$.
- the things that have properties B and C but that lack A, i.e., the things in the set $B \cap C \cap \overline{A}$.
- the things that have properties A and C but that lack B, i.e., the things in the set $A \cap C \cap \overline{B}$.
- the things that have all three of the properties A, B and C, i.e., the things in the set $A \cap B \cap C$.

Use of three predicates yields $2^3 = 8$ regions in the diagram, and for checking validity of syllogisms, what matters is whether these regions are empty or not. If a region is non-empty then putting more objects in it does not make any difference for the truth of a syllogistic statement. This is an important insight for it tells us that a domain of discourse of eight objects is always enough for a counterexample to syllogistic validity.

This insight can also help to see that monadic predicate logic can be translated into propositional logic. We will first carry this out for the syllogistic part. Possible syllogistic situations now correspond with valuations for proposition letters chosen in a special way. Note that the following correspondence of the syllogistic with propositional logic is different from the link we discussed in Chapter 3. Take the following propositional valuation:

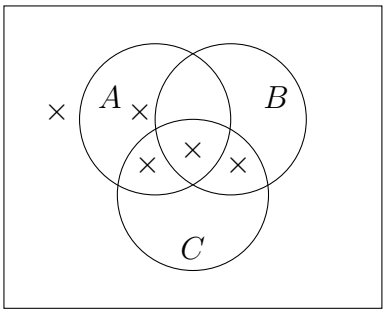
$p_\emptyset$	p_A	p_B	p_C	p_{AB}	p_{AC}	p_{BC}	p_{ABC}
0	1	1	0	1	0	0	0

The value 0 for $p_\emptyset$ expresses that $\overline{A} \cap \overline{B} \cap \overline{C}$ is empty, the value 1 for p_A expresses that $A \cap \overline{B} \cap \overline{C}$ is nonempty, the value 1 for p_B expresses that $\overline{A} \cap B \cap \overline{C}$ is nonempty, and so on. (If you find it hard to figure out which set is meant by the notation $\overline{A} \cap \overline{B} \cap \overline{C}$ you should consult Appendix A on elementary set notation.) In other words, the given valuation corresponds to the following syllogistic situation:



Note that this is a *total* syllogistic diagram: each region has either a $\circ$ or a $\times$ mark. A total diagram corresponds to a single valuation of the above sort, because the diagram states explicitly for each region whether the region is empty or not.

Now the pictures that we have seen in our examples in Chapter 3 were usually *partial* diagrams: these have some regions with neither a $\circ$ nor a $\times$ mark. Partial diagrams represent multiple possibilities for total diagrams. Each region without a mark can be either empty or non-empty, and the diagram rules out neither situation. Thus, a partial diagram corresponds to a set of propositional valuations. Consider the following diagram:



This corresponds to the following set of propositional valuations:

$p_\emptyset$	p_A	p_B	p_C	p_{AB}	p_{AC}	p_{BC}	p_{ABC}
0	0	1	1	1	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	1	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0

We can list these possibilities more succinctly by leaving blanks in all positions where neither of 0,1 is ruled out. This gives us a single partial valuation (this should be compared to the way the syllogistic diagrams abbreviate several possibilities):

$p_\emptyset$	p_A	p_B	p_C	p_{AB}	p_{AC}	p_{BC}	p_{ABC}
0	0				0	0	0

The propositional formula that corresponds to the syllogistic statement "All A are B" is $\neg p_A \wedge \neg p_{AC}$. This states that the region $A \cap \overline{B}$ is empty.

This may have shown you how syllogistic reasoning is close to a propositional logic as in Chapter 2, that talks about the kinds of objects that can occur.

Exercise 5.17 Give the *propositional formulas* for the following syllogistic statements:

(1) No B are C.

(2) Some A are C.

(3) Not all A are B.

In fact, this reasoning generalizes to the whole fragment of monadic predicate logic. Let $\varphi_1, \ldots, \varphi_n$ be a set of n formulas of monadic predicate logic, and assume these are the premises of an inference. Assume ψ is another predicate logical formula, the conclusion. Suppose we want to know whether ψ follows from the premises $\varphi_1, \ldots, \varphi_n$. Then we can proceed like in the procedure for testing syllogistic validity from Chapter 3. We consider the conjunction of the premises together with the negation of the conclusion:

$$\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \neg\psi. \tag{5.1}$$

If we can find a model for this, then we have a counterexample to the validity of the inference and we know that the inference is not valid. If we cannot find a model then this means that no counterexamples exists, and we know that the inference is valid.

The formulas that make up conjunction 5.1 may be quite complex, but however complex they are they will only contain a finite number of different predicate letters. Therefore the whole conjunction 5.1 can only contain a finite number of predicate letters. Suppose

there are m different predicate letters. Then the number of relevant regions in a model for 5.1 is still finite. In fact, there are 2^m relevant regions.

As in the case of syllogisms, all that matters for the truth or falsity of formulas of monadic predicate logic is whether these regions are empty or not. This means that we know beforehand that in our search for a situation where 5.1 is true, we do not need more than 2^m objects. But this in turn means that the search for a model for 5.1 (we call a situation where a certain formula is true a *model* for that formula) can stop after we have inspected all candidates with domains up to and including size 2^m.

The situation is somewhat like that of propositional logic, where we can check consistency of a formula by means of the truth table method. It may take a long time, but it always finishes. Same here: it may take a long time before we have checked all possible candidate situations, but after a finite number of checks we either have found an example or we can know for sure that no candidate situation will fit the bill.

Like propositional logic, monadic predicate logic has a *decidable* satisfiabily problem: there exists a method for settling this question, positively or negatively, in a finite number of steps. For short: monadic predicate logic is *decidable*.

Exercise 5.18 *Satisfiability* of a formula is the property that there are situations that make the formula true. *Validity* of a formula is the property that all relevant situations make that formula true. If satisfiability of formulas is decidable for a logic, does that mean that validity is also decidable? Why (not)?

Summary of Things You Have Learnt in This Chapter *You have learned the power of recursion in defining a formal language and in specifying its semantics.*

Knowledge, Action, Interaction

Chapter 6

Knowledge and Information Flow

Overview This chapter deals with the logic of knowledge, including change of knowledge as a result of communicative acts. This branch of logic is called *epistemic logic*. The main difference between epistemic logic and propositional logic is that in epistemic logic we also want to express facts about knowledge in the logical language itself. This will be possible in the 'modal' logic of knowledge. Because of its relation to reasoning about other people's knowledge, this is a very important generalization. One of this chapter's themes is how successive information processing steps can change the logical specification of a system.

6.1 Motivation — Logic and Information Processing

What makes humans rational? Is it their individual powers of reasoning and observation? Or is it their skill in interaction, communication and co-operation? No doubt, it is both. Traditionally, logic has been more concerned with an account of individual reasoning powers, but this is rapidly changing, and one of the ambitions of modern logic is to provide an account of rational interaction.

A sound argument such as 'from p and $p \to q$ follows q' is interpreted as 'if p is true and if $p \to q$ is true, then q is also true'. 'True' for whom? Isn't there some kind of conscious entity behind this argument, a knowing subject performing the inference

> If I know p and if I know $p \to q$, then I also know q'.

And if another person does not know p but, like me, knows $p \to q$, then she cannot conclude q the way 'I' did.

Being explicit about knowledge The propositional logic of the previous chapter is well-suited to model the knowledge of a single agent as in the update scenarios we have discussed earlier. But many logical processses concern more than one person. Derivations are often made in company, in which case they are truly arguments. One can then reason about other agents' knowledge, for example when I tell you "You *don't know* that I live in

Seville." Having now told you this, you *know* it! Isn't that a paradox? One of the things you will learn in this chapter is why this is not at all paradoxical.

This chapter deals with the logic of knowledge, including change of knowledge as a result of communicative acts such as the 'telling' (informing) above. This branch of logic is called *epistemic logic*. The main difference between epistemic logic and propositional logic is that in epistemic logic we also want to express facts about knowledge in the logical language itself. This will be possible in the 'modal' logic of knowledge. Because of its relation to reasoning about other people's knowledge, this is a very important generalization of what we have seen so far. One of this chapter's themes is how successive information processing steps can change the logical specification of a system.

Agents The subjects whose knowledge we describe are called *agents*. These agents are not necesarily human beings. We can call processes running on a computer agents as well. Epistemic logic is used to specify multi-agent systems. A multi-agent system accommodates multiple 'information systems' that have the capacity of goal-directed and autonomous interaction, and are set in a well-defined environment. The study of multi-agent systems is a research area of its own.

Example 6.1 Let's take a multi-agent system for playing soccer. It consists of 22 agents: the soccer players. A soccer player can only observe the players in his range of vision.

Player a assumes that opponent b is behind him, because in the previous stage of the game, before a received the ball, this was indeed the case. This is therefore a reasonable assumption, and a also knows that such an assumption can very well have become false. It turns out to be false: b is by now somewhere else on the playing ground, but a could not have seen that. The state of the game where b is behind a, is just as conceivable for a as the state of the game where b is not behind a.

Player a also sees player c right in front of himself. Player c is in his own team; a passes the ball to c to prevent b from intercepting it. Now, what? The targeted player indeed gets the ball, only it was player d instead of c, fortunately of the same team, who received the ball. Player a *believed* player d to be c, but must now revise the previous assumption he made.

In yet another scenario c fails to capture the ball, because a and c do not have eye-contact at the moment of a's pass: they do not have common knowledge of the situation leading to this pass.

6.2 Motivation — Modelling the Uncertainty of Agents

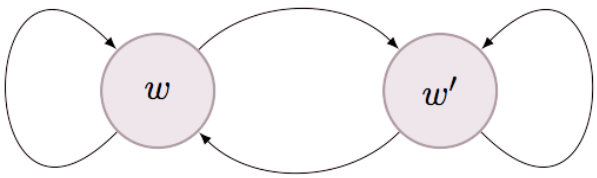
Example 6.2 You are holding three different cards in your hands. These cards are red, white and blue. Turn them around. The backsides are indistinguishable. Shuffle the cards. Now draw a card. Suppose this is the red card. From the two remaining cards you put one on the table in front of you on the left side, upside down, and the other card you put on the right side, also upside down. Which of these two is the white card? And which the

blue card? Let p stand for the proposition 'the left card is the white card'. Then there are now two possibilities: p is true, or p is false. If p is false the left card is the blue card and the right card the white one. If p is true, it is the other way round. In fact p is true: the left card is the white card. The two possibilities are your 'epistemic alternatives'.

The example does not say anything about which of the two situations is in fact the case. A possible state of affairs can be identified with a valuation of propositional variables. There are two different states of affairs. If p is true then we have the valuation V such that $V(p) = 1$ and if p is false then we have the valuation V' such that $V'(p) = 0$. Before you have picked up the cards on the table, you consider it *possible* that the left card is white and you also consider it possible that the left card is blue. The proposition 'I consider it possible that the left card on the table is blue' is rather peculiar: its truth-value depends on the truth-value of the proposition 'the left card on the table is blue' in two different situations. Therefore we cannot formalize the term 'possible' by simple truth-tables as we have done for the propositional connectives in Chapter 2. These are all so-called *truth functional* operations. For connectives such as 'and' and 'if ... then' the truth of the entire proposition can be computed from the truth of its constituents, e.g. for conjunction we need that both conjuncts are true.

The logic in which we can model 'it is possible that' explicitly is a *modal logic*, and the sentential construct 'it is possible that' is called a *modality*, or *modal operator*. There are various brands of modal logic. In this chapter we present the modal logic of knowledge, also known as *epistemic logic*. In all these modal logics a crucial part is that given some actual situation, there may also be other situations that play a part. In epistemic logic we speak of epistemic alternatives. In general in modal logic, we speak of *possible worlds*.

Accessible Consider example 6.2 again. The uncertainty about the real card deal can be represented as a relation between situations. The actual situation is w. Given that situation, it is conceivable that w' is the real one. 'w' is conceivable given w' can also be represented as the pair (w, w') is in the relation R, where R is the relation of accessibility. This relation may of course contain several other pairs. For example, you also consider it possible that w itself is the real world, so (w, w) is also in that accessibility relation. Also, in case w' had been the actual situation, both w and w' are accessible: (w', w) and (w', w') are also in R. We can visualize such models by creating pictures of possible worlds with an associated valuation, connected by links, the accessibility relation, which captures the uncertainty of the agent. This time, let us be completely explicit about the accessibility relation. The uncertainty about cards on the table in front of you can be visualized as:



(6.1)

Here we have drawn all the accessibilities as arrows: each world is accessible from itself, and the two worlds are accessible from each other.

Example 6.3 I am sitting in a bar, and I am in no doubt that the bespectacled bartender in front of me is wearing glasses, for I can see his specs clearly and distinctly on his nose. Now suppose a stranger walks into the bar, behind me and outside my range of vision, shouting 'Hi folks', I certainly consider it possible that she is not wearing glasses, even when in fact she *is* wearing them.

This sort of uncertainty about the actual states of affairs is often described in terms of propositions expressing knowledge and ignorance: 'I don't know if the white card is the one on the left or the one on the right', 'I know that one of the cards in front of me is white', 'I don't know if the newcomer is wearing glasses'. We will now formalize this epistemic modality in the logical language. Write $K\varphi$ for 'I know that φ'. The letter K is the k in *knowledge*.

We can freely combine this operator with the propositional connectives that have already been introduced. For example, we write $\neg K\varphi$ for 'it is not the case that I know φ' (I don't know that φ). The form $\neg K\neg\varphi$ is literally 'I do not know that not φ'. This is more commonly phrased as 'I consider it possible that φ.' There are other standard phrasings that in fact are abbreviations (from our current perspective) such as 'I do not know *whether* the person I hear entering the room behind me is wearing glasses' that is standardly taken to mean 'I do not know that the person I hear entering the room behind me is wearing glasses and I also do not know that the person I hear entering the room behind me is *not* wearing glasses'. In other words, if p stands for 'the lady behind me is wearing glasses' $\neg Kp \wedge \neg K\neg p$.

The language of epistemic logic is interpreted on structures consisting of a domain of worlds, with an accessibility relation between worlds, and with each world a valuation determining the truth-value of atomic propositions (the proposition letters). A proposition $K\varphi$ is true in a given world w if and only if φ is true in all worlds w' that are accessible from w.

Example 6.4 Let 'the left card is white' in example 6.2 be represented by p. Assume that this is indeed true. The actor whose knowledge we model is 'I', the first person singular. Some formalizable statements over the real situation w are:

- I know that the left card is white: Kp.
  This is false if the real state is w, because I consider a state w' possible wherein the left card is blue (so that $\neg p$ is true).

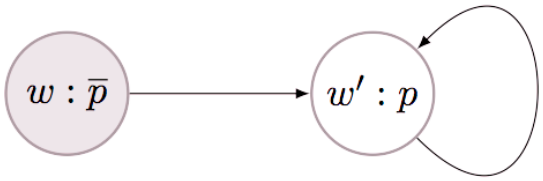
- I do not know whether the left card is white: $\neg Kp \wedge \neg K\neg p$.
  This is true in the real state w. I consider the state w' possible wherein the left card is blue, but I also consider the state w possible where the left card is white.

- I know that I don't know whether the left card is white:

$$K(\neg Kp \wedge \neg K\neg p)$$

This is also true in state w. The formula $\neg Kp \wedge \neg K\neg p$ that is bound by the first (and main) K operator is true in every state accessible from w, namely it is true in w and it is also true in w'.

Example 6.5 Here is a soccer example again, this time involving belief rather than knowledge. I am standing in the field. I don't have eyes in the back of my head, but I am pretty sure that Jan from the other team is standing behind me. In fact, it is Liza who is standing behind me. Let proposition p stand for 'Jan is standing behind me'. We can represent the situation as follows:



(6.2)

Proposition p is only true in w'. The real situation is w, and in w proposition p is false. The only other conceivable (accessible) situation is the one in w': therefore, there is an arrow from w to w', and no other outgoing arrow from w. I therefore 'know' that Jan is standing behind me. We quote the word know. The use of 'know' is restricted to bind propositions that are in fact true. Otherwise, we rather speak of 'think to know' or 'believe'. Jan in not standing behind me: Liza is behind me. But in the current modelling I do not consider this an option! (In English we might then say that we are *convinced* that Jan is standing behind me. This suggests a strong belief that we are unwilling to give up.)

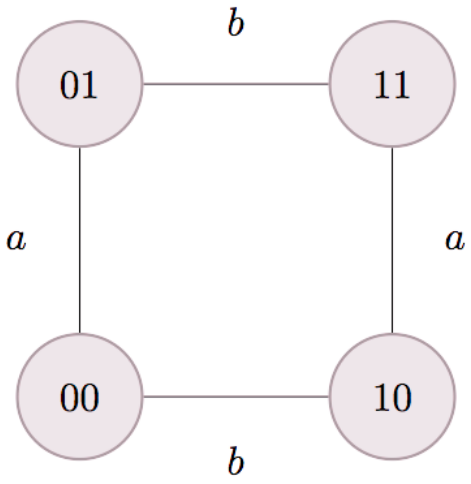
There is also an arrow from w' to itself. This is because, despite this concerns belief instead of knowledge, I am still aware of what I believe and what I don't believe. Because of that arrow we can also compute, using the structure, that KKp is true in state w: "I believe that I believe that Jan is behind me."

History
Epistemic logic has been introduced as a modal logic in 1962 by Jaakko Hintikka, a Finnish philosopher and logician. In his logic both knowledge and belief are introduced as two separate concepts. His logic had two modal operators K and B (for belief) to represent the two 'attitudes' separately.

Jaakko Hintikka

Now let us extend this to the case of multiple agents. What sort of models are common for multiple agents? Remember that agents need not be human. An agent may also be a processor in a computer. A fairly common situation is where every agent *only* knows its local state (the local state can e.g. be the value of a state variable stored locally), and where this is common or background knowledge between all processes. Such a set-up is known as an *interpreted system* or also as a *distributed system*.

Example 6.6 (Interpreted systems) Given are two processes a and b. Propositional variable p describes the state of process a. If it is true, its value is 1, and when false the state of a is 0. Propositional variable q describes the state of process b. Both processes only know their own state. This corresponds to the following model. We have named the states in the model in revealing ways: write 10 when p is true and q is false, etc. A connection labelled with a or b means that the connected states are indistinguishable for a, and b, respectively. Again, we picture arrows in two directions as single links, and we omit reflexive arrows.

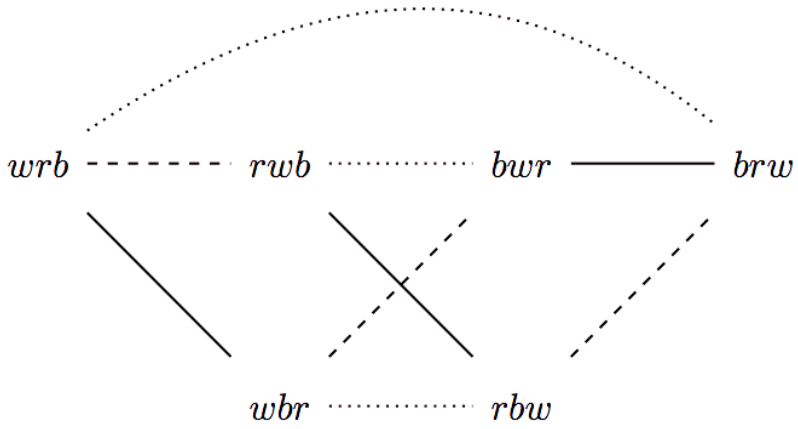


(6.3)

In state 11 of this model it holds that $K_a p$ (process a knows that its value is 1), it holds that $K_a \neg K_b p$ (a knows that b does not know that), and also holds that $\neg K_a \neg K_b q$ (a considers it possible that b knows that its local value is 1). We describe that processes a and b only know their own state by $p \rightarrow K_a p$ and $\neg p \rightarrow K_a \neg p$, and by, respectively, $q \rightarrow K_a q$ and $\neg q \rightarrow K_a \neg q$.

Example 6.7 (Card deals) Three players, Alice, Bob and Carol, each draw a card from a stack of three cards. The cards are red, white and blue and their backsides are indistinguishable (as before). Let rwb stand for the deal of cards where Alice holds the red card, Bob the white card, and Carol the blue card, etc. There are six different card deals. These are therefore the possible situations. Players can only see their own card, but not the cards of other players. They do see that other players only hold a single card, and that this cannot be their own card, and they do know that all the players know that, etcetera (it is an interpreted system). The players' knowledge can be represented as in the following

model. All states are accessible from themselves, we have therefore not drawn the reflexive arrows. We use solid lines to picture Alice's accessibilities, dotted lines for those of Bob and dashed lines for those of Carol.



(6.4)

Let propositional variable r_a stand for 'Alice holds the red card', etc. In situation rwb it is the case that: $K_a r_a$ (Alice knows that she holds the red card) and that $K_a(w_b \vee b_b)$ (Alice knows that Bob must hold the white card or the blue card).

Example 6.7 is related to the three-cards Example 6.2. In that case, the (anonymous) agent is Alice, the card on the left side of the table is Bob's card, and the card on the table's right side is Carol's card. Tables can't think, but agents can, therefore the situation is now much more complex. The proposition p previously standing for 'the left card is white' is now named w_c.

6.3 Language of Epistemic Logic

It is time to get a bit more precise. Before, our set of formulas was given by:

$$\varphi \quad ::= \quad p \mid \neg\varphi \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1 \rightarrow \varphi_2) \mid (\varphi_1 \leftrightarrow \varphi_2)$$

We now extend this with operations for 'i knows that φ', where i is an agent. We use K_i for "i knows that ...". This gives us the following language for the logic of knowledge, or epistemic logic.

Definition 6.8 (Language of epistemic logic)

$$\varphi \quad ::= \quad p \mid \neg\varphi \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1 \rightarrow \varphi_2) \mid (\varphi_1 \leftrightarrow \varphi_2) \mid K_i\varphi.$$

6.4 Semantics of Epistemic Logic – Possible Worlds Semantics

The language of epistemic logic is apparently a simple extension of the language of propositional logic. It took only a single additional operator. The interpretation of this language, as we have seen in the previous examples, requires quite a bit more then the relatively simple truth tables for propositional logic. The reason is that the evaluation of formulas given a situation depends on the state of affairs in other situations, the aforementioned epistemic alternatives or possible worlds. Moreover, the configuration of those possible worlds by means of individualized accessibility relations makes it even more complicated. Let us start with the definition of the intended class of models well-equipped for evaluation of the epistemic language.

Definition 6.9 (Possible world model or epistemic model) A *possible world model* M consists of three basic ingredients:

- a non-empty set of *worlds* W,
- for each agent an *accessibility relation* R_a, that is a set of pairs of worlds, and
- for each world a *valuation functions* V_w.

The definition of truth for formulas with respect to a world in such a model looks more complicated, but this is partly due to the extended format. In propositional logic we mostly wrote $V(\varphi) = 1$ for φ is true in situation/valuation V, but we also wrote $V \models \varphi$ for that. Here the situations are worlds in a model containing a set of possible worlds, and so evaluation of a formula also depends on the 'location' in the model. Similarly to the notation already introduced in propositional logic, but with an additional label for the location, we write $M \models_w \varphi$ to say that φ is *true at* w *in* M and $M \not\models_w \varphi$ to express that φ is *false at* w *in* M.

Definition 6.10 (Evaluation of formulas) To begin with, the evaluation of proposition letters is taken care of by the local valuation functions.

$$M \models_w p \quad \text{if and only if} \quad V_w(p) = 1$$

Evaluation of the propositional connectives runs — of course — in the same way as in the case of propositional logic as presented in the previous chapter.

$$\begin{array}{lll} M \models_w \neg\varphi & \text{if and only if} & M \not\models_w \varphi \\ M \models_w \varphi \wedge \psi & \text{if and only if} & M \models_w \varphi \text{ and } M \models_w \psi \\ M \models_w \varphi \vee \psi & \text{if and only if} & M \models_w \varphi \text{ or } M \models_w \psi \end{array}$$

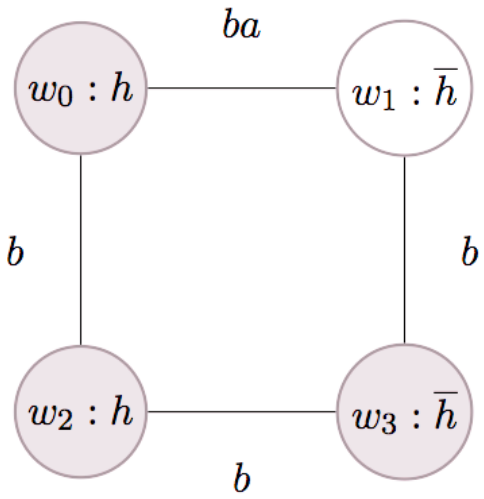
Evaluation of formulas of the form $K_a\varphi$, expressing that agent a knows that φ, requires the evaluation of φ in all epistemic alternatives for a as given by the accessibility relation

R_a. If φ holds in all those worlds the agent a is certain about φ and so $K_a\varphi$ would be true in such a case:

$$M \models_w K_a\varphi \quad \text{if and only if} \quad M \models_v \varphi \text{ for all } v \text{ with } \langle w, v\rangle \in R_a$$

$K_a\varphi$ is false — that is a does not know that φ — if there is at least one world accessible to a where φ does not hold.

Example As an example, consider this situation. A coin gets tossed under the cup. It lands heads up. Alice and Bob are present, but neither of them can see the outcome, and they both know this. Now Bob leaves the room for an instant. After he comes back, the cup is still over the coin. But Bob realizes that Alice might have taken a look. She might even have reversed the coin! Alice also realizes that Bob considers this possible. This can be pictured as below. In the visualization, states that are indistinguishable for an agent are linked with a label for that agent; therefore, all four worlds are indistinguishable for b, w_0 and w_1 are indistinguishable for a, and from a's point of view both w_2 and w_3 stand by themselves (they only have a reflexive accessibility arrow). Call this model M. In the section on information update we will explain how this situation can come about, in Example 6.17.



(6.5)

K_ah expresses "Alice knows that the coin is facing heads up". Is this true in world w_0? No, it is not. The truth definition of K_ah says that h has to be true in every world that is a-accessible to world w_0. We have (w_0, w_1) in R_a, so w_1 is a-accessible to w_0, moreover $M \not\models_{w_1} h$ and therefore

$$M \not\models_{w_0} K_ah$$

Is K_ah true in world w_2? Yes it holds in w_2, since the only world that is a-accessible to w_2 is w_2 itself, so from $M \models_{w_2} h$ it follows that

$$M \models_{w_2} K_ah$$

The formula K_bK_ah expresses the higher order information that Bob knows that Alice knows that the coin shows heads up. Is this true in world w_0? No it is not, for we have just seen that K_ah is not true in world w_0, and w_0 is b-accessible to itself, so we get

$$M \models_{w_0} \neg K_bK_ah$$

If we evaluate K_bK_ah in world w_2, we get the same result. In this case, we have $M \models_{w_2} K_ah$, but since w_0 is b-similar, we still get

$$M \models_{w_2} \neg K_bK_ah$$

Let us try something different. Does Bob know that Alice does not know that the coin is facing heads up, in world w_0? The formula for this is $K_b\neg K_ah$. This is false in w_0, since $\neg K_ah$ holds in w_0, there is a b-accessible world w_2 where, as we have seen, K_ah holds.

Finally, we check whether Alice knows that Bob does not know whether Alice knows which face the coin is showing. The formula for "Alice knows whether the coin shows heads up or tails up" is $K_ah \vee K_a\neg h$. The complete formula whose truth-value we are looking for is

$$K_a\neg K_b(K_ah \vee K_a\neg h).$$

Is this true in world w_0? This would be the case if

$$\neg K_b(K_ah \vee K_a\neg h)$$

holds in every world that is a-accessible to w_0. The two a-accessible worlds to w_0 are w_0 itself and w_1. For w_0 we have indeed:

$$M \models_{w_0} \neg K_b(K_ah \vee K_a\neg h)$$

Yes we do, for both K_ah and $K_a\neg h$ are false in w_0, so $K_ah \vee K_a\neg h$ is false in w_0, and therefore $K_b(K_ah \vee K_a\neg h)$ is also false in w_0. In fact, we also have

$$M \models_{w_1} \neg K_b(K_ah \vee K_a\neg h)$$

Both K_ah and $K_a\neg h$ are false in w_1, so $K_ah \vee K_a\neg h$ is false in w_1, and therefore $K_b(K_ah \vee K_a\neg h)$ is also false in w_1. Altogether, this settles

$$M \models_{w_0} K_a\neg K_b(K_ah \vee K_a\neg h)$$

In world w_0 it is indeed the case that Alice knows that Bob does not know whether Alice knows which side the coin is showing.

Exercise 6.11 In Example 6.6, check that the schema $\neg K_a\neg K_b\varphi \to K_b\neg K_a\neg\varphi$ is a validity on the entire model of example 6.6.

Exercise 6.12 In Example 6.7, check that it is that $r_a \to \neg K_cK_a(w_b \vee b_b)$ in all worlds of the model (if Alice has the red card, then Carol does not know that Alice knows that Bob has the white or the blue card).

6.5 Valid Consequence

Now that we defined under which circumstances a formula of our extended language is true, we can adopt the general notion of valid inference as has been defined in the previous chapter. A formula ψ is a *valid consequence* of a set of premises $\varphi_1, ..., \varphi_n$ if for all models M and worlds w in M

$$\text{if } M \models_w \varphi_i \text{ for all } i, \text{ then also } M \models_w \psi$$

In other words, there exists no model M containing a world w such that

$$M \models_w \varphi_i \text{ for all } i \text{ and } M \not\models_w \psi$$

If ψ is a valid consequence of $\varphi_1, ..., \varphi_n$ then we write, as in the previous chapter,

$$\varphi_1, ..., \varphi_n \models \psi$$

Logical closure of knowledge As we stated at the beginning of this chapter, the definition of valid inference may as well be defined in terms of knowledge. A valid consequence ψ of a set of premises $\varphi_1, ..., \varphi_n$ is indeed known by our agent once he knows that the premises are all true. Formally, this looks as follows

$$\text{if } \varphi_1, ..., \varphi_n \models \psi \text{ then } K_a\varphi_1, ..., K_a\varphi_n \models K_a\psi \tag{6.6}$$

This is called the logical closure of knowledge, and in general it holds in every normal modal logic as a result of how we have interpreted our only modal logical operaror. To see this, consider the case that the conclusion in (6.6) is invalid:

$$K_a\varphi_1, ..., K_a\varphi_n \not\models K_a\psi$$

This would mean that there exists a possible world model M such that

$$M \models_w K_a\varphi_i \text{ for all } i \text{ and } M \not\models_w K_a\psi$$

for a certain world w in M. Falsification of $K_a\psi$ at w in M means that there exists a world v in M which is a-accessible from w and that ψ is false at v in M:

$$M \not\models_v \psi$$

But since all formulas $K_a\varphi_i$ are true in w we also have

$$M \models_v \varphi_i \text{ for all } i$$

But then this world v in M is also a counter-example showing that ψ is not a valid consequence from $\varphi_1, .., \varphi_n$:

$$\varphi_1, ..., \varphi_n \not\models \psi$$

Deriving valid principles By means of the logical closure of knowledge we can derive other valid principles of epistemic logic.

To start with we show that knowing the conjunction of two proposition, $K_a(\varphi \wedge \psi)$, is logically equivalent with knowing both conjuncts, $K_a\varphi \wedge K_a\psi$.

1.	$\varphi, \psi \models \varphi \wedge \psi$	propositional logic
2.	$K_a\varphi, K_a\psi \models K_a(\varphi \wedge \psi)$	1 + logical closure of K_a
3.	$K_a\varphi \wedge K_a\psi \models K_a(\varphi \wedge \psi)$	2 + propositional logic

To show that the reverse also is valid we need another five steps:

4.	$\varphi \wedge \psi \models \varphi$	propositional logic
5.	$K_a(\varphi \wedge \psi) \models K_a\varphi$	4 + logical closure of K_a
6.	$\varphi \wedge \psi \models \psi$	propositional logic
7.	$K_a(\varphi \wedge \psi) \models K_a\psi$	6 + logical closure of K_a
8.	$K_a(\varphi \wedge \psi) \models K_a\varphi \wedge K_a\psi$	5,7 + propositional logic

For the disjunction this surely does not work. Take the model consisting of two worlds w and w' in Example 6.2, with the proposition p standing for 'the left card is white'. In the actual world we have $\neg(K_a p \vee K_a \neg p)$: the agent does not know whether the white card is on the left or on the right. But surely $K_a(p \vee \neg p)$ is true. The formula $p \vee \neg p$ is even a tautology, and therefore, $K_a(h \vee \neg h)$ is always true (valid without premises). On the other hand, for the disjunction we have validity in one direction:

$$K_a\varphi \vee K_a\psi \models K_a(\varphi \vee \psi)$$

This can be shown quite easily by using the logical closure property of knowledge:

1.	$\varphi \models \varphi \vee \psi$	propositional logic
2.	$K_a\varphi \models K_a(\varphi \vee \psi)$	1 + logical closure K_a
3.	$\psi \models \varphi \vee \psi$	propositional logic
4.	$K_a\psi \models K_a(\varphi \vee \psi)$	3 + logical closure K_a
5.	$K_a\varphi \vee K_a\psi \models K_a(\varphi \vee \psi)$	2,4 + propositional logic

To give you one more example we show that we may distribute the K_a-operator over implications:

1.	$\varphi \to \psi, \varphi \models \psi$	propositional logic
2.	$K_a(\varphi \to \psi), K_a\varphi \models K_a\psi$	1 + logical closure K_a
3.	$K_a(\varphi \to \psi) \models K_a\varphi \to K_a\psi$	2 + propositional logic

The reverse of this principle is not valid. We leave it to the reader to find a simple model M containing a world w such that $M \models_w K_a p \rightarrow K_a q$ and $M \not\models_w K_a(p \rightarrow q)$.

Restricting the class of models In epistemic logic we need firm restrictions of the class of models. For example, in the illustrations of the previous sections we assumed that models are *reflexive*, that is each world in a model should have access to itself. Formally speaking, for all worlds w in M and all agents a:

$$\langle w, w \rangle \in R_a$$

This means that our agents take the actual world always as one of their alternatives. If this were not the case then it may happen that an agent has false information. This seems very natural, but if this information is real knowledge then it is unacceptable. You cannot know something that is actually false, because then it is not knowledge but belief or expectation or some other 'weaker' attitude. In reflexive models the formula $K_a\varphi \rightarrow \varphi$ is true in all worlds for every proposition φ. Moreover, every non-reflexive accessibility relation can be dressed up with local valuations such that $K_a\varphi \rightarrow \varphi$ is not true for some formula φ.

Another restriction for knowledge, which is often accepted for weaker attitudes also, is that it is *fully introspective*. The agent may be very doubtful and uncertain, and therefore also ignorant. Nevertheless, he will always be certain about his uncertainty. The uncertainty is represented by the set of accessible worlds, this set should be the same as the set of accessible worlds in accessible worlds: the agent is not uncertain about this set representing his own uncertainty. Expressing this in a more formal way, we get that for all agents a, for all world pairs $\langle w, v \rangle$ in R_a and for all worlds u in M:

$$\langle w, u \rangle \in R_a \text{ if and only if } \langle v, u \rangle \in R_a$$

The corresponding principles for epistemic logic are $K_a\varphi \rightarrow K_aK_a\varphi$ and $\neg K_a\varphi \rightarrow K_a\neg K_a\varphi$. The former is called *positive introspection* since it says that if you know something, then you know that you know this. The latter is called *negative introspection* because it tells us that if there is something you don't know then you know that you don't know this. These principles always hold on the restricted class of models that we have defined here.

The principle of positive introspection is in fact a valid consequence of negative introspection if we combine this with the earlier principle saying that knowledge is always true. Just for sake of illustration we give a derivation here below:

1.	$\models K_a\varphi \rightarrow \varphi$	knowledge is true
2.	$\models \neg\varphi \rightarrow \neg K_a\varphi$	1 + propositional logic
3.	$\models \neg K_a\varphi \rightarrow K_a\neg K_a\varphi$	negative introspection
4.	$\models \neg\varphi \rightarrow K_a\neg K_a\varphi$	2,3 + propositional logic
5.	$\models \neg\neg\varphi \rightarrow K_a\neg K_a\neg\varphi$	4 + replacing φ by $\neg\varphi$
6.	$\models \varphi \rightarrow K_a\neg K_a\neg\varphi$	5 + propositional logic
7.	$\models K_a\varphi \rightarrow K_a\neg K_a\neg K_a\varphi$	6 + replacing φ by $K_a\varphi$
8.	$\models \neg K_a\neg K_a\varphi \rightarrow K_a\varphi$	3 + propositional logic
9.	$\neg K_a\neg K_a\varphi \models K_a\varphi$	8 + propositional logic
10.	$K_a\neg K_a\neg K_a\varphi \models K_aK_a\varphi$	9 + logical closure of K_a
11.	$\models K_a\neg K_a\neg K_a\varphi \rightarrow K_aK_a\varphi$	10 + propositional logic
12.	$\models K_a\varphi \rightarrow K_aK_a\varphi$	7,11 + propositional logic

It is a much longer derivation as the ones we have seen earlier. Just try to check the individual steps for yourself.

6.6 Proof

Just as for propositional logic, there is also an axiomatic proof system for epistemic logic. We recall that a *proof* is a finite sequence of one or more steps, where each step is either an *axiom*, or follows from previous steps in the proof by a (deduction) *rule*, and that a formula is a *theorem* if it occurs in a proof; where a set of axioms and rules defines an *axiomatization* for a given logic. In the case of epistemic logic, we have some extra axioms and also an extra rule. Further recalling propositional logic, the three axioms and one rule that we presented were:

(1) $(\varphi \rightarrow (\psi \rightarrow \varphi))$

(2) $((\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)))$

(3) $((\neg\varphi \rightarrow \neg\psi) \rightarrow (\psi \rightarrow \varphi))$

(4) from φ and $(\varphi \rightarrow \psi)$, infer ψ

As we have already seen there, proofs in this system can be quite cumbersome. One the main functions of a Hilbert-style proof system is to show that *in principle*, we have covered all valid principles of the logic and not forgotten any: this is the issue of *completeness*, that all true principles (validities) are also derivable (theorems). How these valid principles can then be derived, is certainly of interest but almost an aside. From that point of view, now that we know that all tautologies in propositional logic can be derived by the system above, we might as well forget about it and in further axiomatic extensions

take that for granted. Therefore, in the axiomatization for epistemic logic now to follow it simply says that 'all instances of propositional tautologies' are axioms.

Definition 6.13 (Axiomatization of epistemic logic)

(1) all instances of propositional tautologies

(2) $K_i(\varphi \to \psi) \to (K_i\varphi \to K_i\psi)$

(3) $K_i\varphi \to \varphi$

(4) $K_i\varphi \to K_iK_i\varphi$

(5) $\neg K_i\varphi \to K_i\neg K_i\varphi$

(6) from φ and $(\varphi \to \psi)$, infer ψ

(7) from φ, infer $K_i\varphi$

Several of these axioms we have already seen as valid principles, in the previous section, e.g., the axiom $K_i\varphi \to K_iK_i\varphi$ called positive introspection.

Example 6.14 This is derivation of the theorem $K_i\neg p \to \neg K_ip$ (if you know that p is false, you don't know that p):

1 $K_ip \to p$
2 $(K_ip \to p) \to (\neg p \to \neg K_ip)$
3 $\neg p \to \neg K_ip$
4 $K_i(\neg p \to \neg K_ip)$
5 $K_i(\neg p \to \neg K_ip) \to (K_i\neg p \to K_i\neg K_ip)$
6 $K_i\neg p \to K_i\neg K_ip$
7 $K_i\neg K_ip \to \neg K_ip$
8 $(K_i\neg p \to K_i\neg K_ip) \to ((K_i\neg K_ip \to \neg K_ip) \to (K_i\neg p \to \neg K_ip))$
9 $(K_i\neg K_ip \to \neg K_ip) \to (K_i\neg p \to \neg K_ip)$
10 $K_i\neg p \to \neg K_ip$

Can you find out which axioms and rules have been applied and where?

6.7 Information Update

The models with possible worlds and accessibility links for each agent provide complete description of the knowledge of agents, including all that agents know about the knowledge of the other agents. In this section we will show how relatively simple update scenarios can be modelled as modifications of these models.

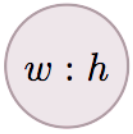
Example 6.15 You have to finish a paper, and you are faced with a choice: do it today, or put it off until tomorrow. You decide to play a little game with yourself. You will flip a coin, and you promise yourself: "If it lands heads I will have to do it now, if it lands tails I can postpone it until tomorrow. But wait, I don't have to know right away, do I? I will toss the coin in a cup." And this is what you do. Now the cup is upside down on the table, covering the coin. You know: if it has heads up, it is off to work. Better postpone looking for a while.

This state of affairs can be pictured as follows:

(6.7)

There are two situations, one in which h is the case (situation w, where the coin has landed heads up) and one where h is not the case (situation w', where the coin has landed tails up). The situation that is actually the case is shaded grey. But I do not know this.

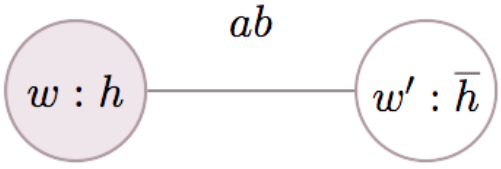
Now consider what happens when you summon up your courage and take a look. The picture simplifies, as follows:



(6.8)

To see how the epistemic analysis can be extended to the multi-agent case, let us complicate the coin toss situation from example 6.15 a bit.

Example 6.16 Suppose there are two people present, you and me. Or, let us say, Alice and Bob. We will use a for Alice and b for Bob. The result of a hidden coin toss with the coin heads up is now pictured like this. We leave out the reflexive arrows, and use link labels to indicate agents:

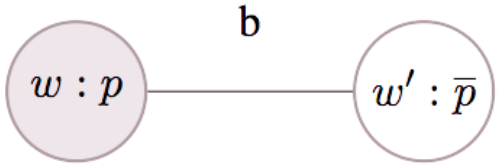


(6.9)

This is quite similar to the picture of example 6.15. The only difference is that now there is a similarity link for both a and b.

Assume that Alice is taking a look under the cup, while Bob is watching. Then we get a quite subtle situation. Alice knows whether the coin shows heads or tails. Bob knows

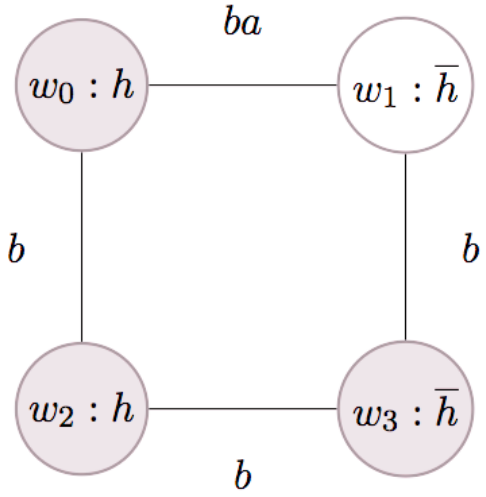
that Alice knows the outcome of the toss, but he does not know the outcome himself. Here is a picture:



(6.10)

The picture shows that Alice knows the outcome of the coin toss: if it is h, she does not confuse this with $\neg h$, if it is $\neg h$ she does not confuse this with h. But Bob does not know. He still cannot make up which of the two situations is the actual one. Bob also knows that Alice knows. Bob takes situations w and w' to be possible, but in situation w Alice is not uncertain about the toss outcome, and in situation w' neither. This means that Bob knows that Alice knows the toss outcome, without Bob himself knowing it.

Example 6.17 Now imagine a slightly different situation. The coin gets tossed under the cup. It lands heads up. Alice and Bob are present, but neither of them can see the outcome, and they both know this. Now Bob leaves the room for an instant. After he comes back, the cup is still over the coin. But Bob realizes that Alice might have taken a look. She might even have reversed the coin! Alice also realizes that Bob considers this possible. The picture gets a bit more complicated:



(6.11)

Let us analyze this state of affairs carefully. We see that three of the four situations in the picture are shaded. This means that each of these three can be the actual situation. To see that this is reasonable, observe that Alice might have done three things while Bob was away:

- Alice might have done nothing. In that case w_0 is the actual situation. But Bob does not *know* that she has done nothing. So Bob confuses w_0 with situation w_1 where

the outcome is tails, but Alice has not looked, with situation w_2 where the outcome is heads, and Alice has looked, and with situation w_3 where the outcome is tails and where Alice has reversed the coin.

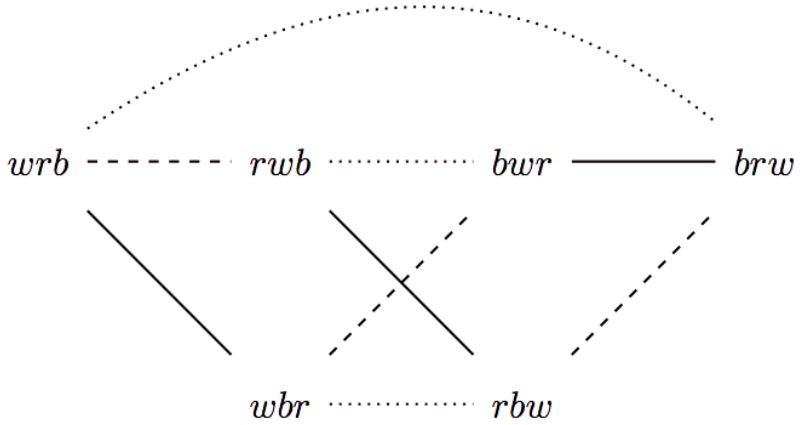
- Alice might have taken a look. In that case w_2 is the actual situation. But again, Bob does not *know* this. He confuses w_2 with situations w_0 and w_1 where Alice has not looked, and with situation w_3 where Alice has changed the outcome.

- Alice might have reversed the coin. In that case w_3 is the actual situation. But Bob does not *know* this, and he confuses this with the other possible situations.

Again, the situation is quite subtle. For Alice knows that Bob considers all these possibilities, so Alice knows that Bob does not know the outcome. But also, Alice knows that Bob does not know whether Alice knows the outcome. This is called higher order knowledge or knowledge about knowledge.

One particular form of such informative developments is what is known as a *public announcement*. A public announcement is spoken information, such that all agents hear what is being said, and also observe and assume that all others hear what is being said, and so on (I know that you hear what I hear, etcetera). Public announcement can be processed in the model in a fairly straightforward way: restrict the model to all states where the publicly announced formula is true, just like new incoming information was dealt with in the previous chapter. These changes can be considered more elementary than the model changes in the previous example, where (e.g., in the transition where Alice is taking a look under the cup, while Bob is watching) the states remain the same but the links change.

Example 6.18 Consider again the model of Example 6.7, and state rwb in that model.



(6.12)

Alice says: "I have the red card." This corresponds to an update with r_a. We restrict the model to situations in which Alice holds red. These are the states wherein r_a is true, i.e., rwb and rbw. The result is depicted below. It is clear that Alice is the only one who is left

ignorant about the actual state of affairs.

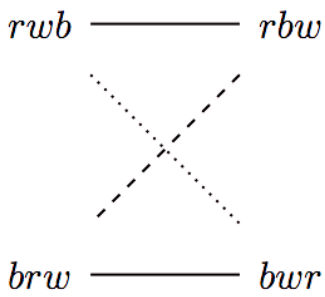
(6.13)

In this structure it holds that both Bob and Carol know what the card deal is. They have no alternatives for the actual state rwb. Bob learnt the card deal because deal bwr is no longer possible. Carol learnt the card deal because deal wrb is no longer possible.

At first sight it seems that 'publicly announcing that φ' is rather much like 'making φ true'. But this may be misleading. You can announce something that is true, but at the same time by conveying the meassage make it become false. This seems implausible. It is possible though by making statements about one's own ignorance or the ignorance of others. Consider the following example:

Example 6.19 In state rwb of the model of Example 6.7 Alice says: "Bob does not know that I have the red card." Literally, this statement is formalized as $\neg K_b r_a$ but in practice such an announcement is only informative if it means "I have the red card and Bob does not know that." Which is an update with $r_a \wedge \neg K_b r_a$. This statement is only true in states rwb and rbw. The resulting model is therefore again the one above. In that model Bob knows that Alice has the red card: $K_b r_a$. Therefore, after the update with $r_a \wedge \neg K_b r_a$ its negation is now true: $\neg(r_a \wedge \neg K_b r_a)$ is logically equivalent with $\neg r_a \vee K_b r_a$, which is a simple valid inference from the information $K_b r_a$.

Example 6.20 Suppose Alice said "I do not have the white card." instead. This statement is true in the four sitations rwb, rbw, brw, and bwr. The resulting model is:



(6.14)

In this model it holds that in state rwb Carol knows what the card deal is, but Bob still does not know. This makes sense, as Bob holds the white card himself, so he already knew that Alice does not hold white. Still, Bob has learnt something from Alice's announcement, namely that Carol now knows what the card deal is.

Formally speaking, public announcements change knowledge states, so their semantics can be given as a function from our possible world models to new models. An update

with φ changes M to $M \mid \varphi$, where $M \mid \varphi$ is the result of removing all worlds from M where φ is false. So the update map is:

$$M \mapsto M \mid \varphi$$

Exercise 6.21 Call a public announcement φ *unsuccessful* in a world in a model if $\varphi \wedge [!\varphi]\neg\varphi$ is true in that world in that model. In other words: φ is true, but a public announcement of φ makes φ false. Give a model, a world and a formula φ (other than in the example above!) such that public announcement of φ in that world in that model is unsuccessful.

Exercise 6.22 The formula $p \rightarrow K_i\neg K_i\neg p$ is a theorem of epistemic logic. Build a proof of it using the axiomatization given in Definition 6.13.

Exercise 6.23 Consider the epistemic situation described in the following scenario: "Three students Ann, Bob and Cath send their request for a certain logic textbook to the library server in the same day at 9am, 10am and 11am, respectively, without knowing this about each other. Because the library has only one copy of the book, the librarian replies by e-mail to everyone informing them that the loan will be given to whoever filed the request first, without any further detail."

(a) Represent the knowlege of the students in an epistemic model.

(b) Check in the model built for (a) if any of the students knows that (s)he will receive the book.

(c) Anxious to find out who will receive the book, Bob writes an e-mail to everione saying: "I filed my request at 10am." Compute the effect of this announcement on the epistemic model from (a).

(d) Check in this new model if any of the students knows that (s)he will get the book on loan.

(e) After this Cath sends a "reply-all" to Bob's previous message saying: "Bob, I think that you might get the book on loan." Compute the effect of this announcement on the epistemic model from (c).

(f) Check in this new model if any of the students knows that (s)he will get the book on loan.

6.8 Expressive Power

A whole lot of knowledge and relations between knowledge can be specified in the epistemic logic of this chapter. To express updates in the language, and such puzzling phenomena as formulas becoming false because they are announced (as when we announce $p \wedge \neg Kp$), we also need an update operator to formalize informational transitions. This will be presented in the outlook section, next. An interesting observation is that such an operator does not in fact increase the expressive power of the logical language! That means, technically, that if we have an formal description for "after announcing that p is true and you don't know, you know that p" then we have an equivalent formula in the epistemic language without the update operator. Really equivalent, in all models where one formula is true, the other must be true as well. If it does not increase expressive power, what's the use of doing that, then? Convenience and succinctness! In propositional logic you would not like to be forced to be precise about every formula employing the Sheffer stroke only (the single connective wherein all others can be defined). The formalizations

would become very long, and although it is fixed what they mean that would not at all be clear from their appearance. Similarly, in epistemic logic formalizations with update operators often appear more natural than those without.

Another epistemic concept that we introduce in the outlook section is that of common knowledge. Something is common knowledge if I know that you know that I know it, and so on. (Details are of course given fully in the next section only.) Now the interesting aspect is that if we extend epistemic logic with such a common knowledge operator, the expressive power of the language really increases, unlike in the case of adding the update operator. Combining such different extensions may also affect expressivity in yet other unexpected ways. We will not digress into such matters.

In the outlook sections that follow we focus on two common extensions of epistemic logic: with operators for common knowledge, and with operators for public announcements. There are more complex extensions than that, e.g., the action wherein Alice looks at the coin without Bob being able to see the result, requires yet another extension.

6.9 Outlook — Common Knowledge

Example 6.24 Imagine two generals who are planning a coordinated attack on a city. The generals are on two hills on opposite sides of the city, each with their armies, and they know they can only succeed in capturing the city if the two armies attack at the same time. But the valley that separates the two hills is in enemy hands, and any messengers that are sent from one army base to the other run a severe risk of getting captured. The generals have agreed on a joint attack, but they still have to settle the time.

So the generals start sending messengers. General A sends a soldier with the message "We will attack tomorrow at dawn". Call this message p. Suppose his messenger gets across to general B at the other side of the valley. Then $K_b p$ holds, but general A does not know this because he is uncertain about the transfer of his message. Now general B sends a messenger back to assure A that he has recieved his message. Suppose this messenger also gets across without being captured, then $K_a K_b p$ holds. But general B does not know this being uncertain about the success of transfer: $\neg K_b K_a K_b p$. General A now sends a second messenger. If this one also safely delivers his message we have $K_b K_a K_b p$. But general A does not know this ... etcetera, etcetera. In this way, they'll continue sending messages infinitely (and certainly not attack tomorrow at dawn).

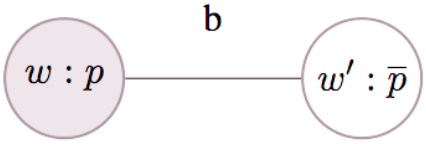
The point of the example is that this procedure will never establish genuine common knowledge between the two generals. They share the knowledge of p but that is surely not enough for them to be convinced that they will both attack at dawn. In case of real common knowledge every formula of the infinite set

$$\{K_a p, K_b p, K_a K_b p, K_b K_a p, K_a K_b K_a p, K_b K_a K_b p, \ldots\}$$

holds.

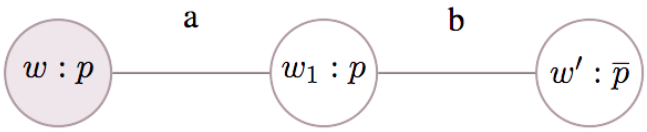
Common knowledge is a very important epistemic concept in communication systems especially when the information that is transfered demands for risk-bearing actions.

Let us draw some pictures of how the situation as given in the previous example develops after each messenger delivers his message. Initially, general A settles the time of the attack. He knows that p but he also knows that general B does not know:



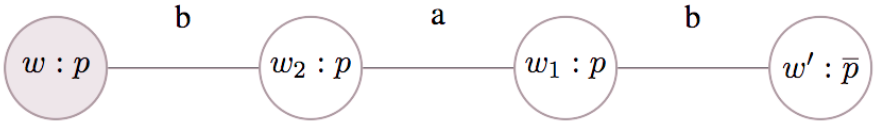
(6.15)

After the first messenger from A to B gets safely across we have:



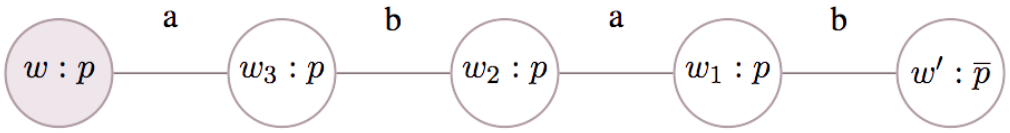
(6.16)

After the message of B to A is safely delivered we get:



(6.17)

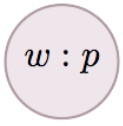
Successful transfer of the second message from A to B amounts to



(6.18)

Note that in world w_1 it does not hold that $K_b K_a K_b p$, and therefore $\neg K_a K_b K_a K_b p$ is true in the actual world w.

The example makes it seem that achieving common knowledge is an extremely complicated or even impossible task. This conclusion is too negative, for common knowledge can be established immediately by public announcement. Suppose the two generals take a risk and get together for a meeting. Then general A simply says to general B "We will attack tomorrow at dawn", and immediately we get:



(6.19)

Still, we cannot express common knowledge between a and b by means of a single formula of our language. What we want to say is that the stacking of knowledge operators goes on indefinitely, but we have no formula for this.

In epistemic logic common knowledge cannot be defined by means of knowledge operators, and therefore we need to extend the language with an additional modal operator C, where $C\varphi$ means that φ is commonly known by the whole group of agents. In order to set up a logical system for this extended language we also need a truth condition in terms of possible worlds models. To provide a satisfactory evaluation we need to define a new accessibility relation by using the accessibility relations of the individual agents. Say we have a model M with accessibility relations R_a for every agent a, then we define the desired new accessibility relation R^* as the set of pairs of worlds $\langle w, v\rangle$ such that there is a path of accessibility links from w to v. Each link in such a path may belong to any agent. The truth condition for $C\varphi$-formulas is then

$$M \models_w C\varphi \text{ if and only if } M \models_v \varphi \text{ for all } v \text{ such that } \langle w, v\rangle \in R^*$$

To see that this condition is the right one is given by the following result:

$$M \models_w C\varphi \text{ if and only if } M \models_w K_{a_0}...K_{a_n}\varphi \text{ for all sequences } a_0...a_n \text{ of agents}$$

Epistemic logic with this additional operator is a bit more difficult than the systems with individual knowledge operators that we have discussed so far. This is due to the infinite nature of common knowledge.

One way to describe that φ is common knowledge is by saying that φ is true and in addition to this everybody knows that φ is common knowledge. This is indeed one of the principles of the logic of common knowledge:

$$\models C\varphi \rightarrow \varphi \wedge EC\varphi. \tag{6.20}$$

where $EC\varphi$ is an abbreviation for 'everybody knows'. For example, if a, b, c are the agents, then $EC\varphi$ is shorthand for $K_aC\varphi \wedge K_bC\varphi \wedge K_cC\varphi$.

Just as in the case of individual knowledge, we have the following for common knowledge:

$$\text{if } \varphi_1, ..., \varphi_n \models \psi \text{ then } C\varphi_1, ..., C\varphi_n \models C\psi \tag{6.21}$$

Exercise 6.25 To convince yourself that (6.21) is true, carry out the reasoning for principle (6.6) for the case of common knowledge.

Finally, if it is common knowledge that φ implies $E\varphi$, then from φ it follows that φ is common knowledge:

$$\models C(\varphi \rightarrow E\varphi) \rightarrow (\varphi \rightarrow C\varphi). \tag{6.22}$$

Think of a situation where it is common knowledge that if φ gets announced then everyone will hear the announcement and thus get to know φ. Then it is clear that if indeed φ gets announced, φ becomes common knowledge.

Exercise 6.26 Describe an epistemic situation with two agents a and b where (i) p implies that both agents know that p and (ii) where p is actually true, but still (iii) p is not common knowledge. Hint: imagine that p is true, that a and b both know it, but that a does not know that b knows ...

Many social rituals are designed to create common knowledge. A prime example is cash withdrawal from a bank. You withdraw a large amount of money from your bank account and have it paid out to you in cash by the cashier. Typically, what happens is this. The cashier looks at you earnestly to make sure she has your full attention, and then she slowly counts out the banknotes for you: one thousand (counting ten notes while saying *one*, *two*, *three*, ..., *ten*), two thousand (counting another ten notes), three thousand (ten notes again), and four thousand (another ten notes). This ritual creates common knowledge that forty banknotes of fifty euros were paid out to you.

To see that this is different from mere knowledge, consider the alternative where the cashier counts out the money out of sight, puts it in an envelope, and hands it over to you. At home you open the envelope and count the money. Then the cashier and you have knowledge about the amount of money that is in the envelope. But the amount of money is not common knowledge among you. In order to create common knowledge you will have to insist on counting the money while the cashier is looking on, making sure that you have her full intention. For suppose you fail to do that. On recounting the money at home you discover there has been a mistake. One banknote is missing. Then the situation is as follows: the cashier believed that she knew there were forty banknotes. You now know there are only thirty-nine. How are you going to convince your bank that a mistake has been made, and that it is their mistake?

Exercise 6.27 Consider the situation where cash is paid out to you by an ATM. The machine counts the money, dispenses it, and you can count it afterwards. Does this create common knowledge between you and the machine that an X amount was paid out to you? Why (not)?

6.10 Outlook — Public Announcement Logic

There is a close connection between public announcements and common knowledge, for common announcement of propositional facts has the effect that those facts become common knowledge.

Example 6.28 If Hans send a group email with a party invitation to Alice, Bob and Carol, then the invitation becomes common knowledge.

Therefore, it is quite reasonable to study public announcements and common knowledge together, by studying a system that has expressions of the form $C\varphi$ ("it is common knowledge that φ") and expressions of the form $[!\varphi]\psi$ (public announcement of φ has the effect that ψ holds.

Still, it is interesting to study the effect of public announcements in a setting where individual knowledge is all one can reason about. Here is the language of this:

$$\varphi \quad ::= \quad p \mid \neg\varphi \mid (\varphi_1 \wedge \varphi_2) \mid K_i\varphi \mid [!\varphi_1]\varphi_2.$$

Note that we have left out $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \to \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$.

Exercise 6.29 Show that leaving out $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \to \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$ from the language definition has no effect on what we can say, by showing how such formulas can be defined from $\neg$ and $\wedge$.

Now it turns out that the following principles hold:

$$[!\varphi]p \leftrightarrow (\varphi \to p) \tag{6.23}$$
$$[!\varphi]\neg\psi \leftrightarrow (\varphi \to \neg[!\varphi]\psi) \tag{6.24}$$
$$[!\varphi](\psi_1 \wedge \psi_2) \leftrightarrow [!\varphi]\psi_1 \wedge [!\varphi]\psi_2) \tag{6.25}$$
$$[!\varphi]K_i\psi \leftrightarrow (\varphi \to K_i[!\varphi]\psi) \tag{6.26}$$
$$[!\varphi_1][!\varphi_2]\psi \leftrightarrow [!(\varphi_1 \wedge [!\varphi_2]\varphi_2)]\psi \tag{6.27}$$

What is remarkable about this is that these principles about public announcements are all equivalences. Also, on the left-hand sides the public announcement operator is the principal operator, but on the righthand sides it is not. What this means is that the principles can be used to 'translate' a formula of individual public announcement logic to a formula of the logic of individual knowledge. In other words: every statement about the effects of public announcement on individual knowledge is equivalent to a statement about just individual knowledge.

Exercise 6.30 Blissful ignorance about a set of facts is a situation where you don't know whether the facts are true or not, but you know there is no reason to worry for it is common knowledge that nobody knows about the truth of these facts. Give a model for a situation with agents a, b, c that are in blissful ignorance about the truth values of p and q.

The logic of common knowledge and public announcement has the following language:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi_1 \wedge \varphi_2) \mid K_i\varphi \mid C\varphi \mid [!\varphi_1]\varphi_2.$$

We now turn to a famous example where public announcements of ignorance can create (common) knowledge.

Example 6.31 A group of children has been playing outside and are called back into the house by their father. The children gather round him. As one may imagine, some of them have become dirty from the play and in particular: they may have mud on their forehead. Children can only see whether other children are muddy, and not if there is any mud on their own forehead. All this is commonly known, and the children are, obviously, perfect logicians. Father now says: "At least one of you has mud on his or her forehead." This is a public announcement. And then: "Will those who know whether they are muddy please step forward."

Suppose we have a situation where Alice, Bob and Carol have been playing outside. As it turns out, Bob and Carol are muddy, but Alice is not. Then, when Bob hears father

ask the question, he sees that Carol is muddy. So he knows that if he is muddy, he is not the only one. So he does not step forward. And he observes that Carol, who reasons like he does, does not step forward. But knowing that Carol is a perfect reasoner, he can draw his conclusion: Carol does not step forward because she sees the mud on *his* forehead. So when father asks again "Will those who know whether they are muddy please step forward," both Carol and Bob step forward.

Exercise 6.32 Suppose that there are four children, Alice, Bob, Carol and Dave. The situation is like in the example, but now Bob, Carol and Dave have mud on their foreheads. Father makes his announcement "At least one of you is muddy," and then starts repeating the question "Will those who know whether they are muddy please step forward." What will happen?

Summary In multi-agent epistemic logic we can describe what an agent knows and what agents know about each other. The logic has modal operators K_a, where $K_a\varphi$ means that 'agent a knows φ', and the logic is interpreted on epistemic models or possible world models. Such a model consists of a domain, a set of accessibility relations, one for each agent, and a valuation of atomic propositions. Due to informative developments, these models may change. You have to be able to model knowledge and ignorance of agents in simple scenarios, formalize their knowledge and ignorance in epistemic logic, and you have to be able to make simple informational transitions of these models based on descriptions of such information change in English.

Further Exercises

Exercise 6.33 Which of the following formulas are valid (you can assume that there are only two agents a and b):

(1) $((K_a\varphi \wedge K_b\varphi) \rightarrow (K_aK_b\varphi \wedge K_bK_a\varphi)) \rightarrow C\varphi$

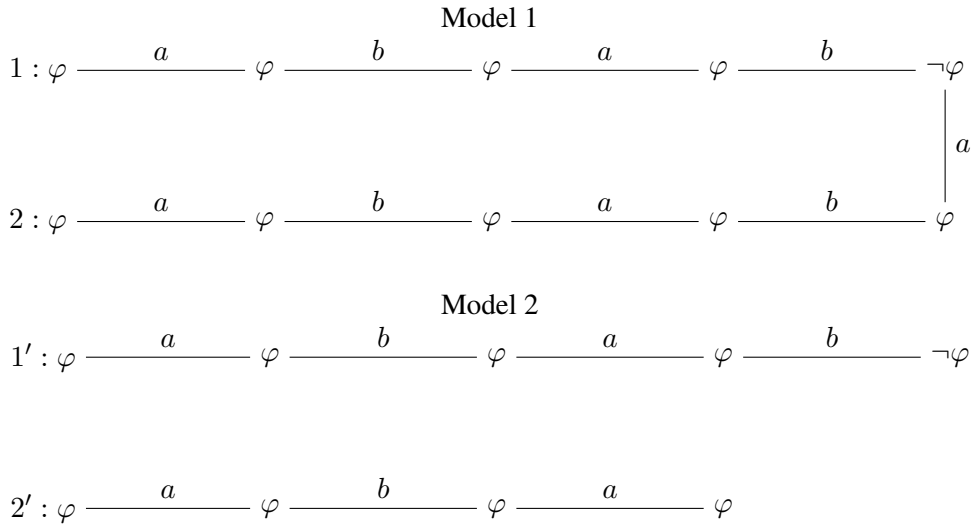
(2) $C\varphi \rightarrow K_aC\varphi$

(3) $C\varphi \rightarrow CC\varphi$

(4) $\neg C\varphi \rightarrow C\neg C\varphi$

Build a counterexample for the invalid formulas and a proof for the valid ones.

Exercise 6.34 Consider the following two epistemic models:



(1) In the first model, can you find am formula of epistemic logic that is true in world 1 and false in world 2?

(2) In the second model, can you find a formula of epistemic logic that is true in world $1'$ and false in world $2'$?

(3) Can you find a formula that, when announced, changes the first model into the second one?

(4) Can you find a formula of public announcement logic that is true in world 1 and false in world 2 of the first model?

Chapter 7

Logic and Action

Overview In this chapter you will learn how you can combine actions with static descriptions of the world. Actions can be communicative actions, facts changing in the world, and calculations. You will learn various operations that combine different actions, such as a sequence of actions and non-deterministic choice between actions. The logic of actions is also related to the logic of information change presented in the chapter 'Knowledge and information flow'.

7.1 Motivation — Actions in General

> Sitting quietly, doing nothing,
> Spring comes, and the grass grows by itself.
>
> *From: Zenrin kushu, compiled by Eicho (1429-1504)*

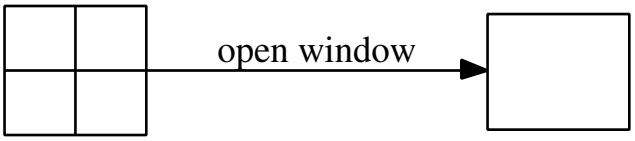
Action is change in the world. Change can take place by itself (see the poem above), or it can involve an agent who causes the change. You are an agent. Suppose you have a bad habit and you want to give it up. Then typically, you will go through various stages. At some point there is the *action* stage: you do what you have to do to effect a *change*.

Actions can be of various kinds. First and foremost there are *actions in the world*, such as preparing breakfast, cleaning dishes, or spilling coffee over your trousers. Such actions change the state of the world. Another type of action is that of *communicative actions*, such as reading an English sentence and updating one's state of knowledge accordingly, engaging in a conversation, sending an email with cc's, telling your partner a secret. These actions typically change the cognitive states of the agents involved. A third kind of actions are *computations*, i.e., actions performed on computers. Examples are computing the factorial function, computing square roots, etc. Such actions typically involve changing the memory state of a machine. Of course there are connections between these categories. A communicative action will usually involve some computation involving memory, and the utterance of an imperative ('Shut the door!') is a communicative action that is directed towards action in the world.

There is a very general way to model action and change, a way that we have in fact seen already. The key is to view a changing world as a set of worlds linked by labeled arcs. In the context of epistemic logic we have looked at a special case of this, the case where the arcs are epistemic accessibility relations: agent relations that are reflexive, symmetric, and transitive. Here we drop this restriction.

A labelled transition system (or LTS) over atoms P and agents A consists of a set S of states, a binary relation $\stackrel{a}{\rightarrow}$ (typically written in infix notation) between these states, and a valuation of atomic propositions, such that every atom is mapped to a subset of the domain of states, representing those where it is true. Let us illustrate the idea of an LTS by a few simple examples.

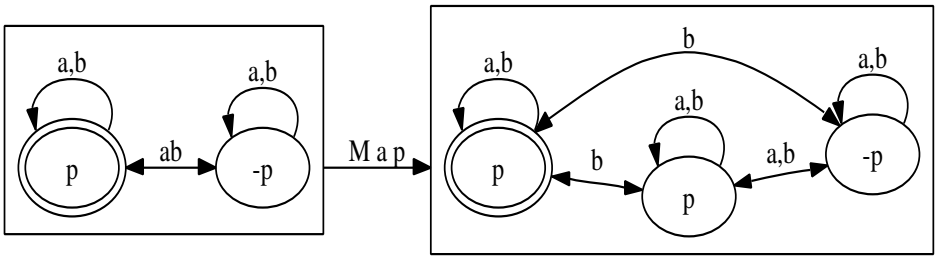
Example 7.1 A simple illustration of how LTSs can be used to model action is when one interprets labelled transitions as actions on the state of the world. In that case LTSs model changes in the world itself:



The action of window-opening changes a state in which the window is closed into one in which it is open.

Another kind of action is a change in the epistemic states of a number of agents.

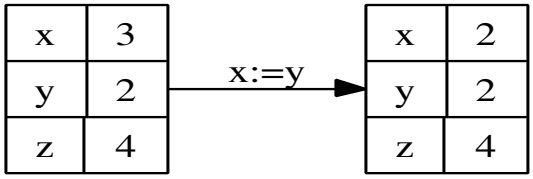
Example 7.2 On the left is an epistemic situation where p is in fact the case (indicated by a double circle), but a and b cannot distinguish between p and $\neg p$. If in such a situation a receives the message that p is the case, while b is not informed of this, the epistemic situation changes to what is pictured on the right.



In the new situation, a knows that p, and a also is aware of the fact that b does not know, while b still does not not know, and b still assumes that a does not know.

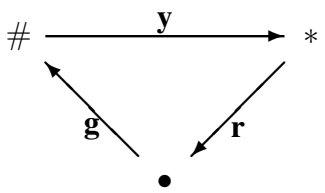
Finally, here is computational example.

Example 7.3 If one interprets the labelled transitions as the changes in the memory state of a computer, LTSs model computations, for example the simple assignment $x := y$:



The command to put the value of register y in register x makes the contents of registers x and y equal.

Example 7.4 Here is an example of an LTS with several transitions. The example models a traffic light that can turn from green to yellow to red and again to green. The transitions indicate which light is turned on (the light that is currently on is switched off). The state $\#$ is the state with the green light on, the state $\star$ the state with the yellow light on, and the state $\bullet$ the state with the red light on.



These examples illustrate that it is possible to approach a wide variety of kinds of actions from a unified perspective. In this chapter we will show that this is not only possible, but also fruitful.

In fact, much of the reasoning we do in everyday life is reasoning about change. If you reflect on an everyday life problem, one of the things you can do is run through various scenarios in your mind, and see how you would (re)act if things turn out as you imagine. Amusing samples are in the Dutch 'Handboek voor de Moderne Vrouw' (The Modern Woman's Handbook).

See `http://www.handboekvoordemodernevrouw.nl/`.

Example 7.5 Here is a sample question from 'Handboek voor de Moderne Vrouw': 'I am longing for a cosy Xmas party. What can I do to make our Xmas event happy and joyful?' Here is the recommendation for how to reflect on this:

Are you the type of a 'guest' or the type of a 'hostess'?
  If the answer is 'guest': Would you like to become a hostess?
    If the answer is 'not really' then
      your best option is to profile as an ideal guest
      and hope for a Xmas party invitation elsewhere.
    If the answer is 'yes' then here are some tips on how to become a great hostess: ...
  If the answer is 'hostess', then ask yourself:
    Are your efforts truly appreciated?
      If the answer is 'Yes, but only by my own husband' then

probably your kids are bored to death.
Invite friends with kids of the same age as yours.
If the answer is 'Yes, but nobody lifts a finger to help out' then
Ask everyone to prepare one of the courses.
If the answer is 'No, I only gets moans and sighs' then
put a pizza in the microwave for your spouse and kids
and get yourself invited by friends.

Exercise 7.6 Construct a so-called flow diagram for the recommendations in the preceding example. The questions should be put in $\diamond$ boxes, the answers should labels outgoing arrows of the $\diamond$ boxes, and the actions should be put in $\Box$ boxes.

7.2 Motivation — Composition, Choice, Repetition, Converse

In the logic of propositions, the natural operations are *not*, *and* and *or*. These operations are used to map truth values into other truth values. When we want to talk about action, the repertoire of operations gets extended. What are natural things to do with actions?

When we want to talk about action at a very general level, then we first have to look at how actions can be structured. Let's assume that we have a set of basic actions. We will assume that basic actions do not have internal structure. How can basic actions be combined to more complex actions? In the first place we can *perform one action after another*: first eat breakfast, then do the dishes. Also, a complex action can consist of a *choice* between simpler actions: either drink tea or drink coffee. Finally, actions can be *repeated*: lather, rinse, repeat.

Repeated actions usually have a *stop condition*: repeat the lather rinse sequence until your hair is clean. The phrase 'lather, rinse, repeat' is used as a joke at people who take instructions too literally (the stop condition is omitted). There is also a joke about an advertising executive who increases the sales of his client's shampoo by adding the word 'repeat' to its instructions.

Some actions can be undone by reversing them: the reverse of opening a window is closing it. Other actions are much harder to undo: if you smash a piece of china then it is sometimes hard to mend it again. So here we have a choice: do we assume that basic actions can be undone? If we do, we need an operation for this, for taking the *converse* of an action. If we assume that undoing an action is generally impossible we should omit the *converse* operation.

Exercise 7.7 Suppose $^\smile$ is used for reversing basic actions. So $a^\smile$ is the converse of action a, and $b^\smile$ is the converse of action b. Let $a; b$ be the sequential composition of a and b, i.e., the action that consists of first doing a and then doing b. What is the converse of $a; b$?

Since we are taking an abstract view, the basic actions can be anything. Still, there are a few special cases of basic action that are special. The action that always succeeds is

called *SKIP*. The action that always fails is called *ABORT*.

A test to see whether some condition holds can also be viewed as a basic action. Notation for the action that tests condition φ is $?\varphi$. The question mark turns a formula (something that can be true or false) into an action (something that can succeed or fail).

Exercise 7.8 Using the operation for turning a formula into a test, we can first test for p and next test for q by means of $?p; ?q$. Show how this can be turned into a single test.

Exercise 7.9 The choice between two tests $?p$ and $?q$ can be written as $?p \cup ?q$. Show how this can be turned into a single test.

Exercise 7.10 The *SKIP* action is the action that always succeeds and does not change anything. Show how this can be expressed as a test.

Exercise 7.11 The *ABORT* action is the action that always fails. Show how this can be expressed as a test.

7.3 Motivation — Operations on Relations

We will represent actions as binary relations. A binary relation on a state set S is a set of pairs (s, s') with s and s' taken from S.

The set of all pairs taken from S is called $S \times S$, or S^2.

Important operations on binary relations are the common Boolean set operations of taking *unions* $R_1 \cup R_2$, taking *intersections* $R_1 \cap R_2$, and taking complements $\overline{R}$.

Example 7.12 The union of the relations 'mother' and 'father' is the relation 'parent'.

Example 7.13 The intersection of the relations $\subseteq$ and $\supseteq$ is the equality relation $=$.

A notation that is often used for the equality relation (or: identity relation is I. The binary relation I on S is by definition the set of pairs given by:

$$I = \{(s, s) \mid s \in S\}.$$

The *Relational composition* of binary relations R_1 and R_2 on S, notation $R_1 \circ R_2$, is given by:

$$R_1 \circ R_2 = \{(t_1, t_2) \in S^2 \mid \exists t_3 \in S\ ((t_1, t_3) \in R_1 \wedge (t_3, t_2) \in R_2)\}.$$

Example 7.14 The relational composition of the relations 'mother' and 'parent' is the relation 'grandmother'.

Exercise 7.15 What is the relational composition of the relations 'father' and 'mother'?

Another important operation is *relational converse*. The relational converse of a binary relation R, notation $R^\smile$, is the relation given by:

$$R^\smile = \{(y, x) \in S^2 \mid (x, y) \in R\}.$$

Example 7.16 The relational converse of the 'parent' relation is the 'child' relation.

Exercise 7.17 What is the relational converse of the $\subseteq$ relation?

There exists a long list of logical principles that hold for binary relations. To start with, there are the usual Boolean principles that hold for all sets:

Commutativity $R_1 \cup R_2 = R_2 \cup R_1$, $R_1 \cap R_2 = R_2 \cap R_1$,

Idempotence $R \cup R = R$, $R \cap R = R$.

Laws of De Morgan $\overline{R_1 \cup R_2} = \overline{R_1} \cap \overline{R_2}$, $\overline{R_1 \cap R_2} = \overline{R_1} \cup \overline{R_2}$.

Next, there are obvious principles of relational composition:

Associativity $R_1 \circ (R_2 \circ R_3) = (R_1 \circ R_2) \circ R_3$.

Distributivity $R_1 \circ (R_2 \cup R3) = (R_1 \circ R_2) \cup (R_1 \circ R_3)$, $(R_1 \cup R_2) \circ R3) = (R_1 \circ R_3) \cup (R_2 \circ R_3)$.

There are also many principles that seem plausible but that are invalid. To see that a putative principle is invalid one should look for a counterexample.

Example 7.18 $R \circ R = R$ is invalid, for if R is the 'parent' relation, then the principle would state that 'grandparent' equals 'parent', which is false.

Exercise 7.19 Show by means of a counterexample that $R_1 \cup (R_2 \circ R_3) = (R_1 \cup R_2) \circ (R_1 \cup R_3)$ is invalid.

Exercise 7.20 Check from the definition that $R^{\smile} = R$ is valid.

Exercise 7.21 Check from the definitions that $(R_1 \cup R_2)^{\smile} = {R_1}^{\smile} \cup {R_2}^{\smile}$ is valid.

Recall that I is the identity relation on S. Then the n-fold composition of a binary relation R on S with itself is defined by recursion, as follows:

$$\begin{aligned} R^0 &= I \\ R^n &= R \circ R^{n-1} \end{aligned}$$

The reflexive transitive closure of R is by definition the smallest relation S that contains R and that is both reflexive and transitive. The reflexive transitive closure of R is given by:

$$R^* = \bigcup_{n \in \mathbb{N}} R^n.$$

7.4 Language — Combining Propositional Logic and Actions: PDL

If one combines propositional logic with actions one gets a basic logic of change called Propositional Dynamic Logic or PDL. Propositional dynamic logic abstracts over the set of basic actions, in the sense that basic actions can be anything. In the language of PDL they are atoms. This means that the range of applicability of PDL is vast. The only thing that matters about a basic action a is that it is interpreted by some binary relation on a state set.

Dynamic logics have two basic syntactic categories: *formulas* and *action statements*. Formulas are used for talking about states, action statements are used for classifying transitions between states. The same distinction between formulas and action statement can be found in all imperative programming languages. The statements of C or Java are the action statements. Basic actions in C are assigning a value to a variable. These are instructions to change the memory state of the machine. The so-called Boolean expressions in C behave like formulas of propositional logic. They appear as conditions in conditional expressions. Consider the following C statement:

```
if (y < z)
  x = y;
else
  x = z;
```

This is a description of an action. But the ingredient `(y<z)` is not a statement (description of an action) but a Boolean expression (description of a state).

Propositional dynamic logic is an extension of propositional logic with action statements, just like epistemic logic is an extension of propositional logic with epistemic modalities. Let a set of basic propositions P be given. Then appropriate states will contain valuations for these propositions. Let a set of basic actions A be given. Then every basic action corresponds to a binary relation on the state set. Together this gives a labeled transition system with valuations on states as subsets from P and labels on arcs between states taken from A.

Definition 7.22 (Language of PDL — propositional dynamic logic) Let p range over the set of basic propositions P, and let a range over a set of basic actions A. Then the formulas φ and action statements α of propositional dynamic logic are given by:

$$\begin{array}{lcl} \varphi & ::= & \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle\alpha\rangle\varphi \\ \alpha & ::= & a \mid ?\varphi \mid \alpha_1;\alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^* \end{array}$$

$\top$ is the formula that is always true. From this, we can define $\bot$, as shorthand for $\neg\top$.

Similarly, $\varphi_1 \wedge \varphi_2$ is shorthand for $\neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ is shorthand for $\neg\varphi_1 \vee \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$ is shorthand for $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$, and $[\alpha]\varphi$ is shorthand for $\neg\langle\alpha\rangle\neg\varphi$.

Exercise 7.23 Suppose we also want to introduce a shorthand α^n, for a sequence of n copies of action statement α. Show how this can be defined by induction. (Hint: use $\alpha^0 := ?\top$ as the base case.)

Let's get a feel for the kind of things we can express with PDL. For any action statement α,

$$\langle\alpha\rangle\top$$

expresses that the action α has at least one successful execution. Similarly,

$$[\alpha]\bot$$

expresses that the action fails (cannot be executed in the current state).

The basic actions can be anything, so assume for a moment that basic actions are public announcements. Then

$$\langle\alpha\rangle\varphi$$

expresses that a public announcement of α may have the effect that φ holds.

If the basic actions are changes in the world, such as spilling milk S or cleaning C, then $[C;S]d$ expresses that cleaning up followed by spilling milk always results in a dirty state, while $[S;C]\neg d$ expresses that the occurrence of these events in the reverse order always results in a clean state.

7.5 Semantics of propositional dynamic logic

The semantics of PDL over atoms P and agents A is given relative to a labelled transition system.

Definition 7.24 (Labelled transition system) Let P be a set of basic propositions and A a set of labels for basic actions. Then a labelled transition system (or LTS) over atoms P and agents A is a triple $M = \langle S, R, V\rangle$ where S is a set of states, $V : S \to \mathcal{P}(P)$ is a valuation function, and $R = \{\xrightarrow{a} \subseteq S \times S \mid a \in A\}$ is a set of labelled transitions, i.e., a set of binary relations on S, one for each label a.

The formulas of PDL are interpreted in states of the labeled transition system, and the actions a of PDL as binary relations on the domain S of the LTS, with the interpretation of basic actions a given as $\xrightarrow{a}$.

Definition 7.25 (Semantics of PDL) Given is a labelled transition system $M = \langle S, V, R\rangle$ for P and A.

$$\begin{array}{lcl}
M \models_s \top & \iff & \\
M \models_s p & \iff & p \in V(s) \\
M \models_s \neg\varphi & \iff & M \not\models_s \varphi \\
M \models_s \varphi \wedge \psi & \iff & M \models_s \varphi \text{ and } M \models_s \psi \\
M \models_s \langle\alpha\rangle\varphi & \iff & \exists t, (s,t) \in [\![\alpha]\!]^M \text{ and } M \models_t \varphi
\end{array}$$

where the binary relation $[\![\alpha]\!]^M$ interpreting the action α in the model M is defined as

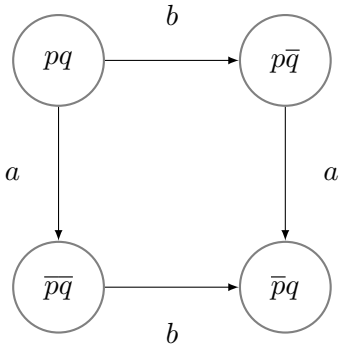
$$\begin{aligned}
[\![a]\!]^M &= \xrightarrow{a}_M \\
[\![?\varphi]\!]^M &= \{(s,s) \in S_M \times S_M \mid s \in [\![\varphi]\!]^M\} \\
[\![\alpha_1;\alpha_2]\!]^M &= [\![\alpha_1]\!]^M \circ [\![\alpha_2]\!]^M \\
[\![\alpha_1 \cup \alpha_2]\!]^M &= [\![\alpha_1]\!]^M \cup [\![\alpha_2]\!]^M \\
[\![\alpha^*]\!]^M &= ([\![\alpha]\!]^M)^*
\end{aligned}$$

These definitions specify how formulas of PDL can be used to make assertions about PDL models. The formula $\langle a\rangle\top$, when interpreted at some state in a PDL model, expresses that that state has a successor in the $\xrightarrow{a}$ relation in that model. A PDL formula φ is *true* in a model if it holds at every state in that model, i.e., if $[\![\varphi]\!]^M = S_M$. Truth of the formula $\langle a\rangle\top$ in a model expresses that $\xrightarrow{a}$ is serial in that model. A PDL formula φ is *valid* if it holds for all PDL models M that φ is true in that model, i.e., that $[\![\varphi]\!]^M = S_M$.

Exercise 7.26 Show that $\langle a;b\rangle\top \leftrightarrow \langle a\rangle\langle b\rangle\top$ is an example of a valid formula.

As was note before, $?$ is an operation for mapping formulas to action statements. Action statements of the form $?\varphi$ are called *tests*; they are interpreted as the identity relation, restricted to the states satisfying the formula.

Exercise 7.27 Let the following PDL model be given:



Give the interpretations of $?p$, of $?(p \vee q)$, of $a;b$ and of $b;a$.

Converse Let $\breve{}$ (converse) be an operator on PDL programs with the following interpretation:

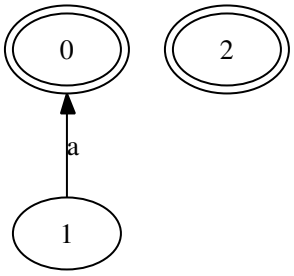
$$[\![\alpha\breve{}]\!]^M = \{(s,t) \mid (t,s) \in [\![\alpha]\!]^M\}.$$

Exercise 7.28 Show that the following equalities hold:

$$\begin{aligned}
(\alpha;\beta)\breve{} &= \beta\breve{};\alpha\breve{} \\
(\alpha \cup \beta)\breve{} &= \alpha\breve{} \cup \beta\breve{} \\
(\alpha^*)\breve{} &= (\alpha\breve{})^*
\end{aligned}$$

Exercise 7.29 Show how the equalities from the previous exercise, plus atomic converse $a^\smile$, can be used to define $\alpha^\smile$, for arbitrary α, by way of abbreviation.

It follows from Exercises 7.28 and 7.29 that it is enough to add converse to the PDL language for atomic actions only. To see that adding converse in this way increases expressive power, observe that in state 0 in the following picture $\langle a^\smile\rangle\top$ is true, while in state 2 in the picture $\langle a^\smile\rangle\top$ is false. On the assumption that 0 and 2 have the same valuation, no PDL formula without converse can distinguish the two states.



7.6 Axiomatisation

The logic of PDL is axiomatised as follows. Axioms are all propositional tautologies, plus the following axioms (we give box ($[\alpha]$)versions here, but every axiom has an equivalent diamond ($\langle\alpha\rangle$) version):

$$\begin{aligned}
\text{(K)}\ &\vdash\quad [\alpha](\varphi\to\psi)\to([\alpha]\varphi\to[\alpha]\psi)\\
\text{(test)}\ &\vdash\quad [?\varphi_1]\varphi_2\leftrightarrow(\varphi_1\to\varphi_2)\\
\text{(sequence)}\ &\vdash\quad [\alpha_1;\alpha_2]\varphi\leftrightarrow[\alpha_1][\alpha_2]\varphi\\
\text{(choice)}\ &\vdash\quad [\alpha_1\cup\alpha_2]\varphi\leftrightarrow[\alpha_1]\varphi\wedge[\alpha_2]\varphi\\
\text{(mix)}\ &\vdash\quad [\alpha^*]\varphi\leftrightarrow\varphi\wedge[\alpha][\alpha^*]\varphi\\
\text{(induction)}\ &\vdash\quad (\varphi\wedge[\alpha^*](\varphi\to[\alpha]\varphi))\to[\alpha^*]\varphi
\end{aligned}$$

and the following rules of inference:

(modus ponens) From $\vdash\varphi_1$ and $\vdash\varphi_1\to\varphi_2$, infer $\vdash\varphi_2$.

(modal generalisation) From $\vdash\varphi$, infer $\vdash[\alpha]\varphi$.

The first axiom is the familiar K axiom from modal logic. The second captures the effect of testing, the third captures concatenation, the fourth choice. These axioms together reduce PDL formulas without $*$ to formulas of multi-modal logic.

Exercise 7.30 Show how this reduction works for the formula $\langle(a;b)\cup(?\varphi;c)\rangle\psi$.

The fifth axiom, the so-called mix axiom, expresses the fact that α^* is a reflexive and transitive relation containing α, and the sixth axiom, the axiom of induction, captures the fact that α^* is the *least* reflexive and transitive relation containing α.

All axioms have dual forms in terms of $\langle\alpha\rangle$, derivable by propositional reasoning. For example, the dual form of the test axiom reads

$$\vdash \langle ?\varphi_1\rangle\varphi_2 \leftrightarrow (\varphi_1 \wedge \varphi_2).$$

The dual form of the induction axiom reads

$$\vdash \langle\alpha^*\rangle\varphi \rightarrow \varphi \vee \langle\alpha^*\rangle(\neg\varphi \vee \langle\alpha\rangle\varphi).$$

Exercise 7.31 Give the dual form of the mix axiom.

7.7 Expressive power — abbreviations defining programming constructs

With the PDL action operation we can define a number of well-known programming constructs. The following abbreviations illustrate how PDL expresses the key constructs of imperative programming:

$$\begin{aligned}
\text{SKIP} &:= ?\top \\
\text{ABORT} &:= ?\bot \\
\text{IF } \varphi \text{ THEN } \alpha_1 \text{ ELSE } \alpha_2 &:= (?\varphi;\alpha_1) \cup (?\neg\varphi;\alpha_2) \\
\text{WHILE } \varphi \text{ DO } \alpha &:= (?\varphi;\alpha)^*;?\neg\varphi \\
\text{REPEAT } \alpha \text{ UNTIL } \varphi &:= \alpha;(?\neg\varphi;\alpha)^*;?\varphi.
\end{aligned}$$

The definitions show how SKIP and ABORT can be viewed as particular kinds of tests.

The definitions make the difference clear between the WHILE and REPEAT statements.

A REPEAT statement always executes an action at least once, and next keeps on performing the action until the stop condition holds. A WHILE statement checks a continue condition and keeps on performing an action until that condition does not hold anymore.

If WHILE φ DO α gets executed, it may be that the α action does not even get executed once. This will happen if φ is false in the start state.

7.8 Expressive power — Epistemic PDL

Now we return to the idea of seeing actions as epistemic transitions. Given a set of action labels, we give some of those a very special meaning, namely these are the 'consider possible' actions of agents named by that label. If in state w agent a considers state v possible, we can see this as a possible a-transition in the model form w to v. It is an

action, and the dynamics of that action represents the mental energy involved to shift your position to that particular state of affairs, and subsequently reason about what is true and false there, instead of here (the actual state).

Of course, in the chapter on epistemic logic we have already seen that these mental shifts obey very specific rules, laid down in the relational properties of reflexivity, symmetry, and transitivity: together they guarantee that the accessibility relations associated with the knowledge of a given agent are equivalence relations, such that the usual properties of knowledge are satisfied: what you know is true, you know what you know (you are aware of what you know), and you know what you don't know (you are aware of what you don't know).

In PDL we can also derive these structural conditions by program operations. Let a again stand for the action representing the mental power of an agent, then: $a \cup ?\top$ is the reflexive closure of the action, $a \cup a^{\smile}$ is the symmetric closure, a^* is the reflexive transitive transitive closure, and $a; a^*$ is the transitive closure. Therefore, $(a \cup a^{\smile})*$ is an equivalence relation associated with this particular a, namely the reflexive symmetric transitive closure: the smallest relation that contains the a relation and that is also reflexive, symmetric and transitive. In other words, we have expressed in PDL (with converse) what it means to be in the smallest equivalence relation containing the a relation. In other words, we have *built* an epistemic operation from a primitive 'imaginability' action, instead of assuming such an action to have these particular structural properties already.

In the set of action labels we thus have some action labels that label epistemic accessibility relations, encoding what the agents with that name know, and the remaining labels can stand for what such agents do. We have created a version of PDL that combines knowledge with action.

Example 7.32 This is the famous chicken game, known from game theory and from movies like *Rebel Without a Cause* with James Dean.

Consider Jim and Buzz driving toward each other at high speed in fabulous racing cars, trying to impress the ladies. The one who bows out to the side of the road to let the other pass is a chicken (a wimp; Dutch: 'een watje'). Problem is that if both stubbornly continue in the straight line they crash and both die. Another problem is that if Buzz bows

out last split-second to the left but Jim last split-second to the right they still crash and both die. We now want to formalize that:

- If Buzz sees Jim swerve to the right he continues straight.
- If they simultaneously swerve to the left they will not die.
- Buzz considers it possible that Jim will swerve to the left, but just as well that Jim will swerve to the right.

Buzz is agent B and Jim is agent J, with corresponding operators $[B]$ and $[J]$. We will assume that B and J denote epistemic accessibility relations. Bowing out left is l_B for Buzz and l_J for Jim. Next, we have atomic action statements for bowing out right and going straight r_B, r_J, s_B, s_J. Note that we view going straight on as a conscious action, not as absence of acting. Crashing and dying are for simplicity identified with each other; this is the action d. To refer to the execution of an action x without reference to any of its postconditions is formalized as $\langle x\rangle\top$. The formalization of the above descriptions is therefore:

- $[B]\langle r_J\rangle \rightarrow \langle r_J\rangle\langle s_B\rangle\top$ (The action of Buzz going straight follows that of Jim going right, the sequential execution is expressed by the word 'continues'.)
- PDL has no way to express that l_J and l_B are concurrent actions, so we need a bit of a trick modelling. We will assume such actions when immediately following upon one another still have the desired effect:

  $$\langle (l_B; l_J) \cup (l_J; l_B)\rangle[d]\bot$$

  It is also reasonable to say that this is common knowledge among Buzz and Jim:

  $$[(B \cup J)^*]\langle (l_B; l_J) \cup (l_J; l_B)\rangle[d]\bot$$

  The $[d]\bot$ part says explicitly that every successful execution of dying results in a state where the false proposition holds. As this is never the case, there is no such execution, in other words, they will not die. Formula $[d]\bot$ is also equivalent to $\neg\langle d\rangle\top$: it is not the case that a successful execution of d can be done.
- $\langle B\rangle(\langle l_B\rangle\top \wedge \langle r_J\rangle\top)$; this is equivalent to $\langle B\rangle\langle l_B \cup r_J\rangle\top$. (But we do not have in general that $\langle x\rangle\varphi \wedge \langle y\rangle\varphi$ is equivalent to $\langle x \cup y\rangle\varphi$. The right is weaker than the left.)

Compared to the approach of the previous chapter that takes epistemic logic as a primitive to which informational and other dynamics can be added, the new PDL approach has a number of advantages:

- It provides a very clear and transparent way to define interaction of different agents in group operators such as general knowledge and common knowledge. Given a set of agents $\{a, b, c\}$, general knowledge is defined as $a \cup b \cup c$. This seems strange, as this represents that one of a, b, c 'takes a step', not necessarily all. But consider the box (necessity) version of the model operator. The formula $[a \cup b \cup c]p$ expresses that after every execution of the program $a \cup b \cup c$ the atomic proposition p should be true. In other words, it should hold after every execution of a and after every execution of b and after every execution of c. To get closer to our epistemic interpretation: p should hold in every state accessible for a, in every state accessible for b, and in every state accessible for c. But that means that p is known by a, and by b, and by c: general knowledge. Common knowledge for a, b, c is definable as the program $(a \cup b \cup c)^*$: iteration of the general knowledge program.

- It is very natural to combine 'true' action operators with the 'epistemic PDL' operators. Consider the following actions: $(a; ?p) \cup (b; ?p) \cup (c; ?p)$: this also describes that one of a, b, c considers it possible that p. The program is equivalent to $(a \cup b \cup c); ?p$ and we have that $\langle (a \cup b \cup c); ?p \rangle \top$ is equivalent to $\langle a \cup b \cup c \rangle p$. The box-version has the general knowledge meaning again: let d stand for the action of the stockmarket crashing, and p for the proposition that outstanding credit is a zillion dollars, then $[(a \cup b \cup c); ?p]\langle d \rangle \top$ says that it is general knowledge between a, b, c that in all states where outstanding credit is a zillion dollars, the stockmarket will crash. In this case, there is an equivalent where the test is outside the modal operator: $[a \cup b \cup c](p \rightarrow \langle d \rangle \top)$. But for more complex constructions this equivalent cannot always easily be found.

  The program α defined as $?p; (a \cup b \cup c)^*$ describes a guarded sort of common knowledge: $[\alpha]\varphi$ is true if φ is commonly known by a, b, c in every p world. Now try formalizing this in the epistemic logic with announcements... The equivalent is not that $p \rightarrow C_{abc}\varphi$: in this case, p only has to be true in the beginning, but not necessarily along the chain.

- In PDL it is much more natural to combine factual change and informational change than in a dynamic epistemic logic. In the dynamic epistemic logic of the previous chapter it may even seem impossible to do this, for all the changes are purely informational changes. While it is in fact possible to extend dynamic epistemic logic with factual change, this involves extending the language. It does not come for free, as in PDL.

7.9 Outlook

7.9.1 Programs and Computation

If one wishes to interpret PDL as a logic of computation, then a natural choice for interpreting the basic actions statements is as *register assignment statements*. If we do this,

then we effectively turn the action statement part of PDL into a very expressive programming language.

Let v range over a set of registers or memory locations V. A V-memory is a set of storage locations for integer numbers, each labelled by a member of V. A V-state s is a function $V \to \mathbb{Z}$. We can think of a V-state as a V-memory together with its contents. If s is a V-state, $s(v)$ gives the contents of register v in that state.

Let i range over integer names, such as 0, -234 or 53635 and let v range over V. Then the following defines arithmetical expressions:

$$a \quad ::= \quad i \mid v \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2.$$

It is clear that we can find out the value $[\![a]\!]_s$ of each arithmetical expression in a given V-state s.

Exercise 7.33 Provide the formal details, by giving a recursive definition of $[\![a]\!]_s$.

Next, assume that basic propositions have the form $a_1 \leq a_2$, and that basic action statements have the form $v := a$. This gives us a programming language for computing with integers as action statement language and a formula language that allows us to express properties of programs.

Determinism To say that program α is deterministic is to say that if α executes successfully, then the end state is uniquely determined by the initial state. In terms of PDL formulas, the following has to hold for every φ:

$$\langle\alpha\rangle\varphi \to [\alpha]\varphi.$$

Termination To say that program α terminates (or: halts) in a given initial state is to say that there is a successful execution of α from the current state. To say that α always terminates is to say that α has a successful execution from *any* initial state. Here is a PDL version:

$$\langle\alpha\rangle\top.$$

In fact, many more properties can be expressed, and in a very systematic way.

7.9.2 Hoare Correctness Reasoning

Consider the following problem concerning the outcome of a pebble drawing action.

> A vase contains 35 white pebbles and 35 black pebbles. Proceed as follows to draw pebbles from the vase, as long as this is possible. Every round, draw two pebbles from the vase. If they have the same colour, then put a black pebble back into the vase, if they have different colours, then put the white pebble back. You may assume that there are enough additional black pebbles. In every round one pebble is removed from the vase, so after 69 rounds there is a single pebble left. What is the colour of this pebble?

It may seem that the problem does not provide enough information for a definite answer, but in fact it does. The key to the solution is to discover an appropriate loop invariant: a property that is initially true, and that does not change during the procedure.

Exercise 7.34 Consider the property: 'the number of white pebbles is odd'. Obviously, this is initially true. Show that the property is a loop invariant of the pebble drawing procedure. What follows about the colour of the last pebble?

It is possible to formalize this kind of reasoning about programs. This formalization is called Hoare logic. One of the seminal papers in computer science is Hoare's [Hoa69]. where the following notation is introduced for specifying what a computer program written in an imperative language (like C or Java) does:

$$\{P\} \quad C \quad \{Q\}.$$

Here C is a program from a formally defined programming language for imperative programming, and P and Q are conditions on the programming variables used in C.

Statement $\{P\}\ C\ \{Q\}$ is true if whenever C is executed in a state satisfying P and if the execution of C terminates, then the state in which execution of C terminates satisfies Q. The 'Hoare-triple' $\{P\}\ C\ \{Q\}$ is called a *partial correctness specification*; P is called its *precondition* and Q its *postcondition*. Hoare logic, as the logic of reasoning with such correctness specifications is called, is the precursor of all the dynamic logics known today.

Hoare correctness assertions are expressible in PDL, as follows. If φ, ψ are PDL formulas and α is a PDL program, then

$$\{\varphi\}\ \alpha\ \{\psi\}$$

translates into

$$\varphi \rightarrow [\alpha]\psi.$$

Clearly, $\{\varphi\}\ \alpha\ \{\psi\}$ holds in a state in a model iff $\varphi \rightarrow [\alpha]\psi$ is true in that state in that model.

The Hoare inference rules can now be derived in PDL. As an example we derive the rule for guarded iteration:

$$\frac{\{\varphi \wedge \psi\}\ \alpha\ \{\psi\}}{\{\psi\}\ \text{WHILE}\ \varphi\ \text{DO}\ \alpha\ \{\neg\varphi \wedge \psi\}}$$

First an explanation of the rule. The correctness of WHILE statements is established by finding a so-called *loop invariant*. Consider the following C function:

```
int square (int n)
{
  int x = 0;
  int k = 0;
  while (k < n) {
```

```
    x = x + 2*k + 1;
    k = k + 1;
  }
  return x;
}
```

How can we see that this program correctly computes squares? By establishing a loop invariant:

$$\{x = k^2\}\ \texttt{x = x + 2*k + 1; k = k + 1;}\ \{x = k^2\}.$$

What this says is: if the state before execution of the program is such that $x = k^2$ holds, then in the new state, after execution of the program, with the new values of the registers x and k, the relation $x = k^2$ still holds. From this we get, with the Hoare rule for WHILE:

$\{x = k^2\}$
`while (k < n) { x = x + 2*k + 1; k = k + 1; }`
$\{x = k^2 \wedge k = n\}$

Combining this with the initialisation:

$\{\top\}$
`int x = 0 ; int k = 0;`
$\{x = k^2\}$
`while (k < n) { x = x + 2*k + 1; k = k + 1; }`
$\{x = k^2 \wedge k = n\}$

This establishes that the WHILE loop correctly computes the square of n in x.

So how do we derive the Hoare rule for WHILE in PDL? Let the premise $\{\varphi \wedge \psi\}\ \alpha\ \{\psi\}$ be given, i.e., assume (7.1).

$$\vdash (\varphi \wedge \psi) \rightarrow [\alpha]\psi. \tag{7.1}$$

We wish to derive the conclusion

$$\vdash \{\psi\}\ \text{WHILE}\ \varphi\ \text{DO}\ \alpha\ \{\neg\varphi \wedge \psi\},$$

i.e., we wish to derive (7.2).

$$\vdash \psi \rightarrow [(?\varphi;\alpha)^*;?\neg\varphi](\neg\varphi \wedge \psi). \tag{7.2}$$

From (7.1) by means of propositional reasoning:

$$\vdash \psi \rightarrow (\varphi \rightarrow [\alpha]\psi).$$

From this, by means of the test and sequence axioms:

$$\vdash \psi \rightarrow [\varphi;\alpha]\psi.$$

Applying the loop invariance rule gives:

$$\vdash \psi \rightarrow [(\varphi;\alpha)^*]\psi.$$

Since ψ is propositionally equivalent with $\neg\varphi \rightarrow (\neg\varphi \wedge \psi)$, we get from this by propositional reasoning:

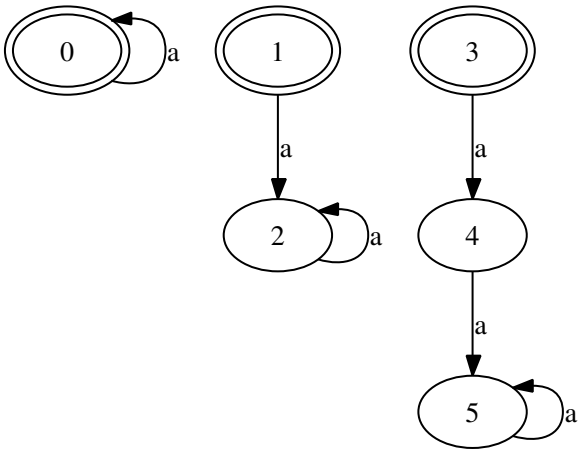
$$\vdash \psi \rightarrow [(\varphi;\alpha)^*](\neg\varphi \rightarrow (\neg\varphi \wedge \psi)).$$

The test axiom and the sequencing axiom yield the desired result (7.2).

7.9.3 Equivalence of programs and bisimulation

PDL is interpreted in labelled transition systems, and labelled transition systems represent processes. But the correspondence between labelled transition systems and processes is not one-to-one.

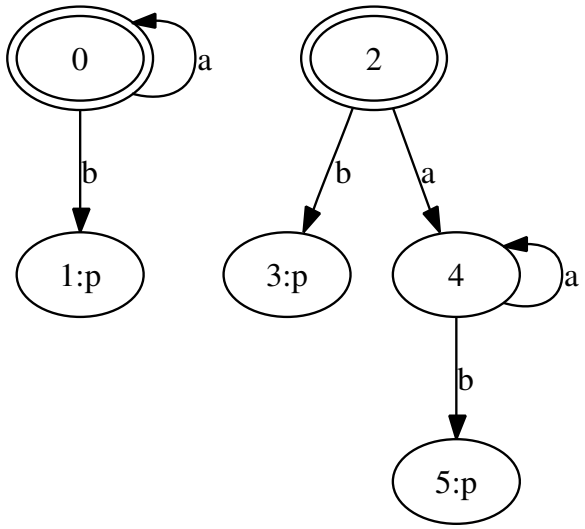
Example 7.35 The process that produces an infinite number of a transitions and nothing else can be represented as a labelled transition system in lots of different ways. The following representations are all equivalent, and all represent that process. We further assume that some atomic proposition p is false in all structures.



To realize such equivalences is important for working with PDL, for the previous example shows that the PDL action statement $a^*;?\bot$ is equivalent to the PDL action statement $a;a^*;?\bot$ which is in turn equivalent to the PDL action statement $a;a;a^*;?\bot$.

The structural equivalence that we observe here can be formalized as a binary relation between points in the first and points in the second structure. This relation is called a *bisimulation*, and it is commonly written as infix, with the symbol $\underline{\leftrightarrow}$. Two structures in such a relation are called bisimilar. A bisimulation connecting the left and the middle labelled transition structures above is: $0 \underline{\leftrightarrow} 1$, $0 \underline{\leftrightarrow} 2$. A bisimulation connecting the middle and the right labelled transition structures above is: $1 \underline{\leftrightarrow} 3$, $1 \underline{\leftrightarrow} 4$, $2 \underline{\leftrightarrow} 4$, $2 \underline{\leftrightarrow} 5$. (There may be more — another one is the product of $\{1,2\}$ and $\{3,4,5\}$.) The bisimulation relation is transitive: the left and the right structure are also bisimilar.

Example 7.36 For another example, consider the following picture. Atom p is false in states 0, 2, and 4, and true in states 1, 3 and 5.



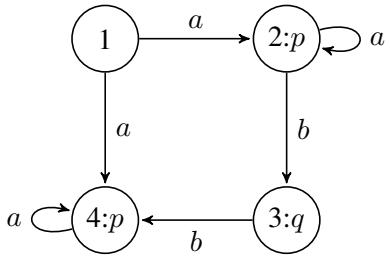
In the labelled transition structures of the picture, we have that $0 \leftrightarrows 2$ and that $2 \leftrightarrows 4$; and $1 \leftrightarrows 3$ and $1 \leftrightarrows 5$.

If two structures are equivalent in this sense (i.e., bisimilar), we cannot distinguish them by a process. Either it will run on all, or on none. These processes a form part of the logical language of PDL, in the dynamic modal operators $[a]$. So this suggests that bisimilar structures *cannot be distinguished from one another in the logical language*. And indeed, that is a basic result. Structures may be indistinguishable in one but distinguishable in another language. Again, expressivity lurks around the corner. A somewhat more detailed and technical introduction to bisimilarity is found in the chapter 'Logic, games and information'.

Summary You have to understand the meaning of the following terms: labelled transition system, PDL, composition, choice, repetition, converse. You have to know the definition of programming constructs 'skip','if-then-else', 'while-do', and 'repeat-until' in PDL. You have to be able to test if given programs can be executed on simple labelled transition systems. You have to know the language of PDL and be able to interpret the language on LTSs.

Exercises

Exercise 7.37 Consider the following process graph with 4 states. The usual conventions apply: transition relations are indicated by labeled arrows, all proposition letters true at worlds are written there:



(1) List in which states the following formulas are true:

a. $\neg p$

b. $\langle b \rangle q$

c. $[a](p \rightarrow \langle b \rangle q)$

(2) Give a formula (involving actions) that is true only at state 4.

(3) Give all the elements of the following relations

a. $b; b$

b. $a \cup b$

c. a^*

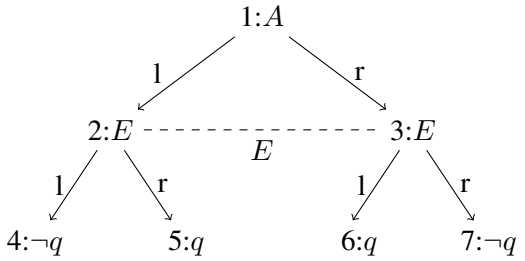
(4) By using any of the known operations (";", "$\cup$", "*", "?"), build an action defining the relation given by $\{(1, 3)\}$. (Hint: use one or more test actions.)

Exercise 7.38 (1) Explain why the following formula is valid (that is, true in all worlds of all models):

$$[R; (S \cup T)]\varphi \leftrightarrow ([R][S]\varphi \wedge [R][T]\varphi)$$

(2) The following identity between relations $(R \cup S)^* = R^*; S^*$ is not valid. Explain why not by giving a counter-example.

Exercise 7.39 Consider the following game, with turns for players A and E as indicated, which have two moves at each node, 'left' and 'right', as displayed. The proposition letter q is true at two of the four endpoints of the game (if you want to, think of q as the proposition "E wins the game"). The dashed lines represent epistemic uncertainty (indistinguishability relation) and the continuous lines represent the actions in the game. Remember that, for the undistinguishability relation, we assume reflexivity (that is, every world is connected to itself).



(1) Does player E know the move of player A before the game ends?

(2) But we can also interpret formulas involving both action and knowledge. In which node(s) are the following formulas true?

   a. $K_E q$

   b. $\langle l \rangle K_E q$

   c. $K_E \langle l \rangle q$

(3) Write a formula expressing that E knows that she has a move she can play that will lead to a state with q.

Chapter 8

Logic, Games and Interaction

Overview In this chapter you will learn that logic is very suitable for analyzing multi-party rational interactions, or rather, that much of what goes on in logic can be seen as a game between two players. Both players are making moves trying to get at a favourable outcome. Reasoning in public is like playing such a game. We will introduce the topic of logic games by means of examples.

8.1 Motivation — the game of logic, and the logic of games

The game of logic Much in logic can be seen as a game between two players. The outcomes are 'true' and 'false', one player is trying to prove his right and the other player is trying to prove him wrong. Moves in such a game correspond to presenting arguments for or against a certain conclusion (representing the 'right' of the first player). Just like in rhetorics and in the courtroom it not only matters if you *are* right but last and least it matters if you *get* it right. And also just like in court it matters how often you can move (how many witnesses you can call, strict rules on interrogation and court procedure) given the limited time of a court session to prove your point. More-than-two player games are so far not known in a logical setting.

Argumentation theory is also typical for logic as a game, and this tradition is going back to Greek logic from the classical period. Following Lorenzen's work in the 1950s it became again a flourishing area of research. From that time also date various games to prove a predicate logical formula 'right' when playing against a logical opponent trying to disprove that formula. An example is the computer program Tarski's World wherein the Hintikka game that you can play with a given blocks world is doing exactly that. One can also play such games with modal logical formulas and Kripke models describing the uncertainty of agents.

Games Games are already an object of study in themselves. If you are playful enough you can make any kind of interaction with other people into a game. Driving through heavy traffic, a conversation with the waitress at the coffee corner, an encounter with

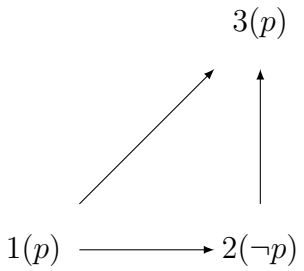
the press, a bet with a friend about who is right about something, a telephone conversation with your bank about the conditions of a loan, the list is endless. Most of these games have rather vague conditions for winning and losing. Card games and board games have very precise rules about how the players should behave, and about what counts as a win. The mathematical investigation of card games started in the 17th century when Antoine Gombaud, Chevalier de Méré, a flamboyant French gentleman, gambler and amateur mathematician, posed the following question to his mathematician friends: "Suppose two players have agreed to play a certain number of games, say a best of seven series. They start playing, but they are interrupted before they can finish. How should the stake be divided among them if, say, one has won three games and the other has won one?" The challenge of solving this so-called *problem of the points* was taken up by the famous mathematicians Blaise Pascal and Pierre de Fermat, and the result was the theory of probability. In the twentieth century the mathematical discipline of game theory has developed. The aim of game theory is to analyse the full range of situations where competing rational agents interact, with "being rational" roughly understood as "having a keen eye for one's own interest".

Antoine Gombaud

The logic of games Topics in game theory can also be addressed in logics, and with logical tools. Concepts such as 'I have a winning strategy' have logical formalizations. This matter we will not address in general in this chapter. Instead, as an interesting example we will play a logical game. This is a game of epistemic information, called public announcement game. The game moves are announcements. As so typical when game theory and logic interact, in this game we not only can compute the result of different informative changes, but we can also value one change over another, and based on such preferences choose a particular change. Logic is often merely descriptive: if you do A, then B will happen, and if you do C then D will happen. But a game is prescriptive: as you prefer B over D, you must do A and not C.

8.2 Game of logic — Evaluation games

Consider the following Kripke model, with designated state 1:



We can determine the truth of a formula in state 1 in a playful way, as the outcome of a game, called *evaluation game*, played between a player called Verifier and another player called Falsifier. Verifier is trying to establish that a given formula is true in a given model, and Falsifier is trying to establish the opposite: that it is false. Let us play the game for some formulas first, before we give the general rules.

Let $\Diamond p$ be the formula. Given that $\Diamond p$ is a diamond-formula, the first move is to Verifier. ($\Diamond p$ is a diamond-formula just like $\langle p\rangle q$ in the action chapter, where $\langle p\rangle q$ was defined as $\neg[p]\neg q$; $\Box p$, $K_i q$, and $[p]q$ are box-formulas.) She has to choose a state accessible from 1. She chooses state 3. The game continues with the formula p and state 3. We have that p is true in state 3. The game ends. Verifier wins. Suppose that instead of state 3 Verifier had chosen state 2. Then the game continues with the formula p and state 2. We have that p is false in state 2. The game ends. Falsifier wins. Verifier seems to have two different strategies, of which one results in a win and the one in a lose. Of course we know that $\Diamond$ p is true in state 1 of the model: there is an accessible state where p is true, namely state 3. This now means that Verifier has a winning strategy.

Now consider the formula $\Diamond\Box p \wedge \Diamond\neg p$. This is a conjunction. The game rule is that therefore Falsifier is the first to move, and has to select one of the conjuncts. (Namely attempting to falsify it! Indeed, a conjunction is false if one the conjuncts is false, this reflects the game rule.) Falsifier chooses the first conjunct. This subformula is a diamond formula, therefore Verifier is now to move. Verifier chooses state 2. The formula bound by the diamond, $\Box p$, is boxed formula. That means that Falsifier is again to move, now in state 2. But there is no move to be made in state 2, there are no accessible states! As Falsifier cannot move, Verifier wins.

Suppose that Falsifier had chosen $\Diamond\neg p$ instead in his first move. Again, Verifier proceeds, and, choosing well, selects state 2 in her move. Now we are stuck with formula $\neg p$ and state 2. In case of a negation, an interesting game rule applies: whoever is to move and play the game for given negation $\neg\varphi$, now switches roles with the other player and the game continues in the same state but for the other player to play with φ instead. So, instead of Verifier having to win for $\neg p$, we proceed with Falsifier having to 'win' for p (a win for Falsifier means succeeding in proving the formula *false*). Indeed, p is false in state 2. So, again, Verifier wins the game.

The general rules for these games are as follows:

Given a pointed Kripke model (M, s), and a formula φ, and two players Verifier and Falsifier. How to play φ in s:

- if $\varphi = p$, Verifier wins if p is true in s, else Falsifier wins.
- if $\varphi = \varphi' \vee \varphi''$, Verifier chooses a disjunct, say φ'. Play φ' in s.
- if $\varphi = \varphi' \wedge \varphi''$, Falsifier chooses a conjunct, say φ'. Play φ' in s.
- $\varphi = \neg\varphi'$, Verifier and Falsifier switch roles. Play φ' in s.
- if $\varphi = \Diamond\varphi'$, Verifier chooses a state accessible from s, say t. Play φ' in t. (If no such world exists, Falsifier wins.)
- if $\varphi = \varphi' \wedge \varphi''$, Falsifier chooses a state accessible from s, say t. Play φ' in t. (If no such world exists, Verifier wins.)

By 'switching roles' we mean that if Verifier wins the game for φ in s, then Falsifier wins the game for $\neg\varphi$ in s, and that if Falsifier wins the game for φ in s, then Verifier wins the game for $\neg\varphi$ in s.

The formula φ is true in (M, s) if and only if Verifier has a winning strategy for the game. What is a winning strategy? Let us play some other games first, before we answer that question.

This way to determine the truth of a formula in a game is called *game semantics*. Lorenzen was at the origin of this development. Later important contributions were made by Hintikka, Andreka & van Benthem, and Rahman. See also `http://en.wikipedia.org/wiki/Game_semantics`.

Paul Lorenzen

8.3 Game of logic — Bisimulation games

Other games can be played in logic, for example games to compare structures. A well-known one is the 'Ehrenfeucht-Fraïssé' game. Given are two structures consisting of domains of abstract entities (objects, elements, ...) with binary relations between them. There are two players: Spoiler and Duplicator. Spoiler tries to prove that the structures are different, whereas Duplicator tries to prove that they are the same. We will give one example only of a play of an 'Ehrenfeucht-Fraïssé' game.

Given are two models (structures) Three (left) and Four (right). The game is as follows. First, the players agree on the duration of the game. For example: two moves each. Then, Spoiler chooses a model and a point in that model. For example: 3. Then, Duplicator chooses a point in the *other* model. The idea is that this is the point that he finds most 'similar' to the point chosen by Spoiler, as he tries to 'duplicate' the structure that is built up by Spoiler. For example: d.

Now, Spoiler may choose one of the two models again, and a point in that model. For example, Spoiler choose the same model but now point 1. Then, Duplicator chooses in the other model yet again a point, for example a.

Both players now have made two moves, and the game is finished. To determine who has won, we restrict both models to the selected points, including all links between those points. In other words, on the left we keep $1 \rightarrow 3$ and on the right we keep $a \rightarrow d$.

If there is no structural difference between these substructures, Duplicator wins. Because this means that Duplicator has effectively duplicated the structure. Otherwise, Spoiler wins. Because that means that Spoiler has effectively spoiled Duplicator's efforts to duplicate the structure.

We play again. The players now may make four moves each. The play of the game is as follows:

$$S : 1 - D : a \, ; S : d - D : 3 \, ; S : 2 - D : b \, ; S : 3 - D : d$$

Duplicator wins again. Spoiler would have done better choosing the other model. Let's play once more. Now we get

$$S : 1 - D : a \, ; S : d - D : 3 \, ; S : 2 - D : b \, ; S : c - D : 3$$

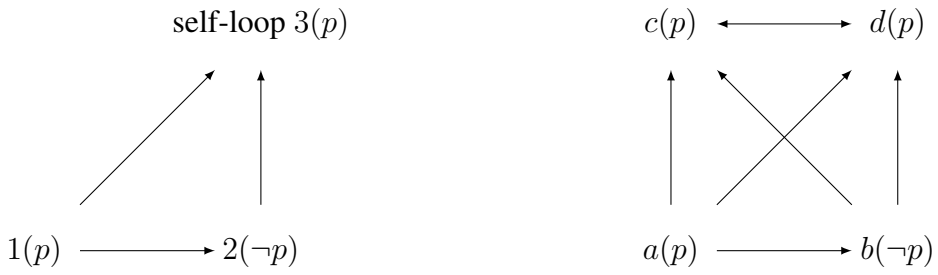
Now Spoiler wins, because there is an arrow from d to c but no arrow from 3 to 3. Spoiler should therefore not just do anything, but intelligently conceive a strategy. (We can show that Spoiler has an winning strategy.)

8.3.1 Logic — Bisimulation

The game can be played with first-order logical models, but also with Kripke models. In the latter case, the rules of the game change a bit, and matching states should have the same valuation of atomic propositions. For example, we get:

By this game it is determined whether two Kripke models are *bisimilar*. Bisimilarity is a technical notion from which it follows that the models are so alike that they cannot be distinguished from one another in the (modal) logical language. Indeed, the models above are not bisimilar, which we can e.g. determine by playing the same game $S : 1-D : a\,; S : d-D : 3\,; S : 2-D : b\,; S : c-D : 3$ as before. And again, we have to show that Spoiler has a winning strategy, not just this 'accidental' play of the game that he wins.

No matter how you play the game with the next couple of structures, Duplicator will always win. They are bisimilar.



Definition 8.1 (Bisimulation) Given possible world models $\mathcal{M}$ and $\mathcal{N}$, a *bisimulation* C between is a relation on $S_{\mathcal{M}} \times S_{\mathcal{N}}$ such that if sCt then the following hold:

atoms $V_{\mathcal{M}}(s) = V_{\mathcal{N}}(t)$ (the two states have the same valuation),

forth if for some $a \in S_1$ $s \xrightarrow{a} s' \in R_{\mathcal{M}}$ then there is a $t' \in S_2$ with $t \xrightarrow{a} t' \in R_{\mathcal{N}}$ and $s'Ct'$.

back same requirement in the other direction: if for some $a \in S_2$ $t \xrightarrow{a} t' \in R_{\mathcal{N}}$ then there is an $s' \in S_1$ with $s \xrightarrow{a} s' \in R_{\mathcal{M}}$ and $s'Ct'$.

The notation $\mathcal{M}, s \underline{\leftrightarrow} \mathcal{N}, t$ indicates that there is a bisimulation that connects s and t. In such a case one says that s and t are bisimilar.

Example 8.2 A bisimulation between possible world models is *total* if all worlds in the first are in some relation to a world in the second, and vice versa. A total bisimulation between the possible world models above (a description of the relation C in the definition) consists of the pairs: $(1, a), (2, b), (3, c), (3, d)$.

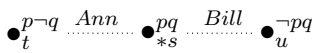
Exercise 8.3 Stefan

Exercise 8.4 Stefan

We have already discussed the notion of bisimilarity in Chapter **??** on actions. A play of the game between Spoiler and Duplicator also serves as well to construct a logical formula that distinguishes the models in case they are not bisimilar. We will not give the construction here. For example, in the model on the right we can make the formula $\Diamond\Diamond\Diamond p$ true, but not on the left. Such a formula is called a *difference formula*.

8.4 Logical game — Knowledge games

Consider two players Ann and Bill. They both have a secret! Ann knows that Helen is in love with Bill (p) but is afraid to tell him that, i.e., she probably has not told him but who knows. Whereas Bill knows that Ann got through her maths exam (q), although he reckons that she does not know that yet – he is unsure. If we assume that Ann and Bill are commonly aware of these propositions (not at all a reasonable assumption, but without that assumption the model would become more complex), this uncertainty about each other can be visualized as follows (see again the chapter on Knowledge and information flow). The actual state (with a *) is s.



In this picture we see that Anne (A) knows p and is uncertain about q, but in case that q is false she believes that Bill (B) knows that – to be more precise, she considers it possible that q is false, in which case Bill knows that q is false, but she also considers it possible that q is true, and in that case Bill does not know whether q is true. On the other hand, Bill knows that q is true but does not know whether p, and beyond that he considers it possible that either Anne knows that p is false or Ann does not know whether p.

The thing with secrets is that your prefer to keep them secret, but that in order to learn more secrets, you need to give some of your own away in exchange. In other words, you may be willing to share privileged information in return for learning more privileged information. Your ultimate goal is to learn as many secrets as possible. Ann wants to learn about her Maths exam, and Bill wants to learn about Helen's true feelings for him. Ann's goal is to prevent Bill from learning the truth about p unless she learns the truth about q first. We can capture this condition in the formula

$$(K_B p \vee K_B \neg p) \rightarrow (K_A q \vee K_A \neg q)$$

Bill's goal is, analogously, to prevent Ann from learning the truth about q unless he learns the truth about p first. This can be captured by

$$(K_A q \vee K_A \neg q) \rightarrow (K_B p \vee K_B \neg p)$$

We can see this as a game, where the moves in the game (the actions) are announcements.[1] In the setting above, Ann and Bill can both choose between two moves, being more or being less informative: Ann can either say p or nothing, and Bill can either say q or nothing. To say nothing means making the trivial announcement $\top$. We also assume that the players make their announcements simultaneously – to avoid scenarios of simultaneously shouting to make yourself understood, we can also imagine Ann and Bill to write the secret on a sheet of paper, fold it, and then exchange the sheets and read the one you received. As usual in the logic of public announcements, the announced formulas are supposed to be *true* and the only way for Ann or Bill to know that what they announce is true, is indeed to *know* the formula: when Ann announces a formula φ we need to interpret this as the public announcement of $K_A\varphi$, and similarly for Bill.

There are four possible outcomes of this process. That Ann plays p means that she checks whether p is true and if so announces that she knows that p, and otherwise she announces that she doesn't know p. Similarly for Bill and q. If Ann plays p and Bill plays q, this is the public announcement of $K_A p \wedge K_B q$ and the resulting model (i) is

$$\bullet^{pq}_{*s}$$

Note that the goals of both players are now realized. But of course, there are three other outcomes of this game. The models resulting from the announcement of $K_A p$, $K_B q$, and $\top$ (e.g., $K_A p \wedge K_B \top$ is equivalent to $K_A p \wedge \top$, and $K_A p \wedge \top$ is equivalent to $K_A p$), are respectively the following (ii), (iii), and (iv):

$$\bullet^{p\neg q}_{t} \overset{Ann}{\cdots\cdots} \bullet^{pq}_{*s}$$

$$\bullet^{pq}_{*s} \overset{Bill}{\cdots\cdots} \bullet^{\neg pq}_{u}$$

$$\bullet^{p\neg q}_{t} \overset{Ann}{\cdots\cdots} \bullet^{pq}_{*s} \overset{Bill}{\cdots\cdots} \bullet^{\neg pq}_{u}$$

[1] These knowledge games have been studied by Ågotnes and van Ditmarsch in recent work called 'What will they say? — Public announcement games', presented at the Logic, Game theory and Social Choice conference in Tsukuba, 2009.

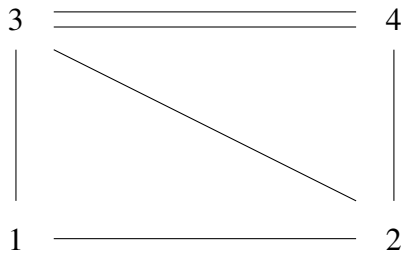
Ann's goal is realized in i, iii and iv but not in ii. Bill's goal is realized in i, ii, and iv but not in iii. These four are all possible outcomes. What should Ann and Bill do, given that they can choose what to do, in other words, what are good strategies to win this game? And, just to mention another complication: even if we were to know a strategy to win the game in state s, what use is that to Ann and Bill, given that they do not know that s is the actual state?

Before we can answer in full these questions, we need to be more precise about what strategies are, what winning is, and how one's choices in a game affect those of the other players. Therefore we first start with some real game theory.

8.5 Games — Winning strategies

Example 8.5 (The Obstruction Game) Given is a network consisting of nodes representing cities and links representing ways to travel between them. Also, there is a starting position. There are two players: 'Runner' and 'Chaser'. Runner moves first, and must try to visit all nodes in the network by making moves along existing connections. From his current node he chooses another one one step away. Chaser's move consists of removing links from the network, as he prefers, one link at the time. Runner and Chaser move in turn. The game ends if a player cannot make a move anymore. A finished game is a win for Runner if every node has been visited. Otherwise, Chaser has won.

It is easy to imagine Runner being the average low-cost air traveller, trying to criss-cross Europe by way of the cheapest flights available, whereas Chaser represents the joint airline companies, cancelling and adding connections between cities all the time. Let us have a look at an example play of this game.



One possible play of the game is

Runner moves from x to y	*Chaser removes connection between x and y*
1 – 3	3 – 2
3 – 4	4 – 2
4 – 3	1 – 2

etc. It will be clear that Runner has lost the game now, as node 2 has become unreachable from his current position.

Instead, now consider the following different play of this game:

Runner moves from x to y	*Chaser removes connection between x and y*
1 – 2	2 – 3
2 – 4	4 – 3
4 – 3	

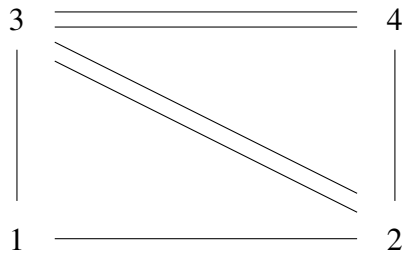
Now, Runner has won. In most games both players *can* win or lose, depending on how they play. But that's not all that there is to say. Because for this game we can observe that:

Chaser can always win, no matter how Runner plays!

We can see this as follows. If Runner goes to 3, then Chaser has enough time to 'isolate' node 2 in three moves. If, instead, Runner first goes to node 2, then the following is effective for Chaser. Let Chaser remove in his first two moves the links between 3 and 4. Then Runner is forced to return either from 3 to 4 or from 4 to 3. Chaser can then isolate one of these nodes from the other one in subsequent moves.

Strategy and winning strategy In the previous example, Chaser has a fitting response to each possible move of his opponent. This is called a *strategy*. A strategy is a sequences of moves (actions) where every next move is one of the possible choices that the player can make in that state of the game. A strategy guaranteeing that a player wins, no matter what his opponent(s) do, is a *winning strategy*.

Example 8.6 Let us now investigate which of the two players has a winning strategy in the Obstruction game with the following initial configuration:



Runner (instead of Chaser) has now a winning strategy! We sketch the argument. In order to win, Runner first has to go to node 2. It now does not serve Chaser to cut a 2–3 or a 2–4: Runner can still access 3 or 4! And removing the two links between 3 and 4 does not help either, as before. Runner has *just* enough time first to go to 4 and then by way of 2 also to get to 3. And wins the game ...

Exercise 8.7 Show that in Example 8.6 Runner has no winning strategy where his initial move is a step from node 1 to node 3.

8.6 Games — Zermelo's theorem and backward induction

A *game tree* is a tree such that: (a) the root of the tree is the initial game state, (b) every node wherein a given player is at move has children that stand for all the game states resulting from the different moves possible for that player, and (c) as leaves the end states of the game. A *zero-sum game* is a game where in every terminal state there is one winner and one loser. Another important concept is '*determined*'. A game is determined if we can determine who can win before we actually start to play. According to a theorem from 1913 by the mathematician Ernst Zermelo, in every finite game of a type called 'zero-sum' exactly one player *must* have a winning strategy, in other words, zero-sum games are determined.

Ernst Zermelo

We now proceed to prove Zermelo's Theorem. This is a quite strong result. It is easy to prove. In the proof, we will use formal terminology in order to illustrate the precise details of the proof. We introduce that first.

A *game* of *game tree* G is a finite process graph of the form $\langle P, S, s_0, M, T, W\rangle$, with

P: This is the set of *players*.

S: This is the set of *game states*.

s_0: This is the *initial state*, where $s_0 \in S$.

M: This is the set of *moves*. A move m is a partial function from S to S. I.e., for each $s \in S$ and each $m \in M$ either m cannot be applied (a *leaf* or *terminal state* of the tree), or there is a uniquely determined successor $m(s) \in S$ of s: This is the new game state that is created as a result of the move m in game state s.

T: This is a function $T : S \to P$ that determines whose *turn* it is in any given game state.

W: This is a subset of the set of the terminal states, indicating what are the winning states. (Typically, seen as a function from the set of players to the set of terminal states.)

A game is finite is there is no cycle of moves after which we get back to a previous state of the game:

> There is no $s \in S$ and sequence of moves $m_1, ..., m_n$ such that $m_n(m_{n-1}(..m_1(s)...)) = s$.

Theorem 8.8 (Zermelo's Theorem) Every finite zero-sum game is determined.

Proof: Let $G = \langle P, S, s_0, M, T, W \rangle$ be a game. The proof is by induction on $|S|$. If $|S| = 1$ then $|S| = s_0$. As the game is finite, s_0 must be an end state. The winner is $W(s_0)$ (and the winning strategy is 'do nothing').

Let now $|S| = n + 1 > 1$. Consider all successor states S_1 of s_0:

$$S_1 = \{s \mid \exists m \in M : m(s_0) = s\}$$

For each successor state s we can define a new game G_s:

$$G_s = \langle P, S - \{s_0\}, s, M_s, T_s, W_s \rangle$$

where M_s, T_s and W_s are the restrictions to $S - \{s_0\}$ of their corresponding versions in G. We now have, obviously, that $|S - \{s_0\}| = n$. Therefore, we can apply the induction hypotheses to each such game G_s. From that follows that for each such game there must be a player p_s with a winning strategy σ_s. We now face two possibilities, resulting in a different player having a winning strategy for G:

(1) $T(s_0) \neq p_s$ for all successor states $s \in S_1$. Then, the player starting the game has no winning strategy for any of the successor games G_s. Then the other player has a winning strategy for G, namely as follows. For each game G_s he has a winning strategy σ_s. Therefore, his winning strategy for G is then:

> Let initial player $T(s_0)$ make her move m and then execute strategy $\sigma_{m(s_0)}$.

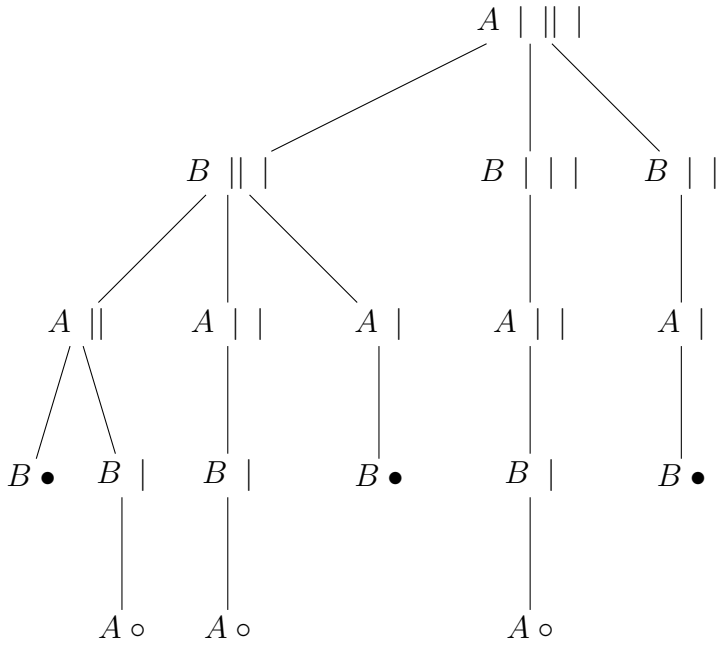
(2) $T(s_0) = p_s$ for some $s \in S_1$. Now the initial player has a winning strategy, which is:

> Choose a move $m \in M$ such that $s_{m(s_0)} = T(s_0)$ and then execute $\sigma_{m(s_0)}$.

Either way, there is a player with a winning strategy. Q.E.D.

We can illustrate Zermelo's result by building the game tree for the Nim-game.

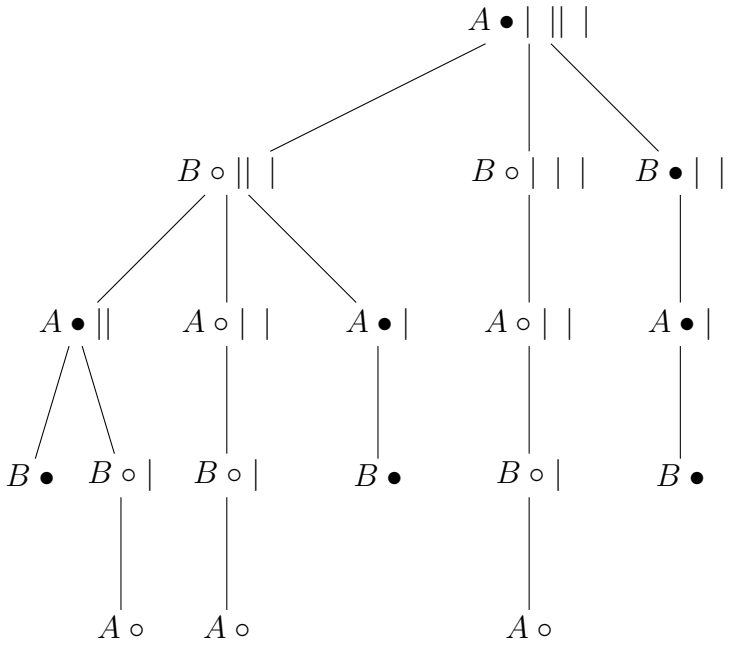
Example 8.9 (The Game of Nim) The Nim game is about matches. The initial game state is a number of stacks of matches. There are two players, A and B. Player A starts the game. The player at move has to select one of the stacks and remove at least one match from that stack. The last player removing a match wins the game. To determine a winning strategy we simply draw the entire game tree. This is the game tree for the initial situation | || | (four matches, in stacks of 1, 2, and 1 match(es) each).



In every state of the game is indicated whose turn it is. Therefore, in every terminal state of the game the player at move is the loser. The winning end states for A are in black en the winning end states for B are in white. We can now also colour the parents of those terminal states with black or white! If the player at move in a parent state can select a winning state (for *that* player) then the winning strategy is to select that move. But that means that we can colour that parent as a win for that player! Unless of course none of the possible moves results in a win, and we colour the parent in that case as a winning state for the opponent.

Having done that, we can assume this parent state to be some kind of terminal state with a winner and a loser. Of course this assumes that the players do not play moves contrary to their interests — but this is indeed assumed! Now we proceed this colouring procedure with the parents of those now coloured states, and so on, until we have coloured the initial state of the game. If that is now black, player A has a winning strategy, and if it is now white, player B has a winning strategy.

In the 1–2–1 game player A has a winning strategy: she can choose to remove the stack with two matches, after which we end up in one of two 'end games' (pictured as a single transition in the figure) both resulting in a win for A.



Having coloured the game tree, the strategy of the winning player is therefore always to select a descendent with his own colour. In other words: we have not merely determined who can win the game, but we even have a winning strategy, a procedure that when executed will result in a win.

Zermelo's Theorem has not much practical significance. The problem is that the game tree can be rather large, and colouring the entire tree may then be intractable. Zermelo's own interests were in games like chess wherein apart from win and lose one also can have a *draw*: neither win, nor lose. Zermelo's Theorem in that case becomes: one of the players has a strategy to *avoid losing* (although no player may have a winning strategy).

Each position of the chess game belongs either to the set of winning positions of White, G_W, or to the set of winning positions of Black, G_B, or it belongs to the set D of draw positions, i.e. positions where both White and Black can avoid a loss by using an appropriate non-losing strategy. For each position which belongs to G_W (G_B), there is a winning strategy g_W (b_B) by which player White (Black) can force a win. For each position which belongs to D, there is a non-losing strategy d_W (d_B) by which White (Black) can avoid a loss.

The difference between the theory and practice of chess playing is unbridgeably large, up to date. We are now one century after Zermelo's original result but it is still not known for chess if the starting player has a strategy to avoid losing the game. This is because the computational complications are simply far too large to tackle. There are roughly 20 possible moves per game position, an average game of chess take 30 to 50 moves, and the total number of legal positions of chess pieces on the chess board is about 10^{120}. This is the same order of magnitude as the number of particles in the universe!

Example 8.10 In the case of the game of Nim there is an elegant shortcut to determine who will win the game. Let $n_1, n_2, \ldots$ be the numbers of matches in each stack. We now write these numbers in binary notation, below one another in different rows. This is an example. Suppose we begin with three stacks containing $n_1 = 5$, $n_2 = 9$ and $n_3 = 4$ matches each. In binary notation that is $n_1 = 101$, $n_2 = 1001$ and $n_3 = 100$. We now count the number of 1s in nth position of these binaries. We get

n_1		1	0	1
n_2	1	0	0	1
n_3		1	0	0
	1	2	0	2

1, 2, 0 and 2 of such 1s. A game is called *balanced* if for every position in binary notation the number of 1s is even or 0. Otherwise, the game is unbalanced. The 1–2–1 game is unbalanced because there is just a single 1 on fourth position. The concept of balance is useful to analyze the game:

- A balanced game will always become unbalanced after a legal move. The number of matches will change in one stack. Therefore, in one row at least one 1 will change into a 0, and vice versa. If the number of 1s in each column was even, this therefore can no longer be the case after that move.

- An unbalanced game *can* always be made a balanced game with a move. Take the leftmost (highest) position p having an odd number of 1s. Let n_i be a number having a 1 on that position. We will now remove a number of matches from this stack of n_i matches. To determine this number we look at the binary representation of n_i from position p onwards, to the right (lower digits). Start by replacing the 1 in pth position by 0. Go one position to the right. If this is a column with an even number of 1s, do nothing. Otherwise, if it's a 1 make it a 0 and if it's a 0 make it a 1. Now repeat the procedure until the rightmost/smallest binary value. We have thus constructed an number n_i' in binary notation. In order to get that amount, the player has to remove $n_i - n_i'$ from the stack with n_i, thus making the game state balanced.

- A player at move in a balanced game cannot win in that move, because a balanced game state consists of at least two stacks.

The winning strategy for Nim is therefore to make an unbalanced game into a balanced game. After that, the opponent cannot finish the game (and win). Therefore, if the initial game state is unbalanced, the player who begins has a winning strategy.

Exercise 8.11 Consider the following initial state of the Nim game (see Example 8.9).

n_1		1	0	1
n_2	1	0	0	1
n_3		1	1	0

Is this a balanced game state? Who can win? Give a play of the game.

8.7 Games — Equilibrium strategies

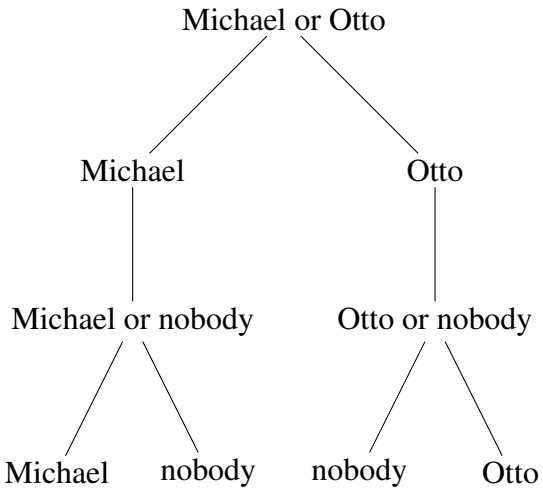
8.7.1 Preferences in games

In zero-sum games, the only possible outcomes are win and lose. This is like in the logical games, where the only possible outcomes are true and false. Human preferences tend to be more complex than 'winning' and 'losing' only: there can be more than two possible outcomes, and the players may have more complex preferences among them. The knowledge games that we already introduces could serve as an example: instead of a single goal, the players may have several goals, some of which are more important than others. An appealing example from standard game theory is about 'strategic voting'.

Example 8.12 (Elections) Mary, Nathasha, and Ophelia have a spare ticket for a performance of the Soundproof Ensemble. They are allowed to vote between Michael, nobody, and Otto. Their preferences are as follows:

	Mary	Natasha	Ophelia
1	Michael	nobody	Otto
2	nobody	Michael	Michael
3	Otto	Otto	nobody

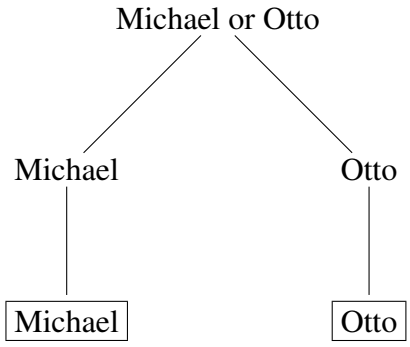
First they vote on Michael or Otto, then they vote on the remaining candidate or nobody. What should their votes be? Consider the tree visualizing the options:



If everyone votes according to their preferences, Michael wins the first round with two to one, and subsequently Michael also wins the second round, against nobody, with two to one again. Natasha can anticipate this! Therefore, she may change her voting behaviour in round 1 and can even consider voting *against* her preference. Suppose she is doing that: in round 1 she votes for Otto instead of for her prefered Michael. Now Otto is chosen in round 1. Then, given majority vote along preferences, nobody wins. Which was Natasha's preference anyway! Great.

Unfortunately, both other players can also make this argument, and adapt their voting behaviour consequently... Does this lead anywhere? Yes it does.

Just as in Example 8.9 we can start down below in the game tree and reason backwards. (This is called, indeed, 'backwards induction'.) In the second round of the voting procedure every player knows what will happen, so that deviating from preference makes no sense. For example, given the choice between nobody and Otto, nobody will be chosen, and given the choice between nobody and Michael, Michael will be chosen. That means we can give those parent nodes the value of the finally realized choice:



This also determines the rational choice in the first round of voting: A player should vote for Michael in round 1 (and not for Otto) if she prefers Michael over nobody (but regardless of whether she prefers Michael over Otto!). A player should vote for Otto in round 1 if she prefers nobody over Otto (but regardless of whether she prefers Otto over

Michael!). Rational players therefore initially vote as follows — note that both Natasha and Ophelia vote against their 'real' preference:

Mary	Natasha	Ophelia
Michael	Otto	Michael

And the conclusion of all that is that Michael will accompany them to the concert.

8.7.2 Strategic equilibrium

Consider a two-person game. Given a strategy for both players, there is a unique outcome of the game (a pair of values) that is brought about by both players simply executing these strategies when playing against each other. This outcome can be evaluated by the players, in relation to all other possible outcomes achieved by different strategies.

Definition 8.13 (Nash equilibrium) Two strategies are in Nash equilibrium if neither player can improve his outcome by changing his strategy, assuming that the other player does not change strategy.

The definition does not rule out that *both* players change their strategy at the same time. Which may result in an outcome better for both. We will give some examples of that. Such a Nash equilibrium in principle corresponds to the stable and prescribed behaviour for rational players. For more than two players, the concept is essentially the same. For example, in the election game above one can simply verify that the finally prescribed voting behaviour for the three players is a Nash equilibrium.

The election game is a very different sort of game from the Obstruction Game or the game of chess. Players have to make a move simultaneously, without knowing what the other players will do in that move. Many examples in game theory are of that kind, mainly so-called 'strategic games'. Strategic games can be represented by a matrix of outcomes. For two players, the rows stand for the strategies of player 1 and the column stand for the strategies of player 2. A pair (i, j) stands for the outcomes for the row player and the column player, respectively. We proceed with a number of famous 2×2 examples: two players and two strategies only, and both strategies consisting of a single move.

Example 8.14 (Prisoners' Dilemma) Consider two prisoners, suspect 1 and suspect 2. There is not enough evidence for a long-term prison sentence. If both deny the crime they will only get one year in prison. If one confesses to it but not the other, that prisoner will go free and the other one will get ten years in prison. If they both confess, they both get five years in prison. This is a matrix of the game.

	deny	admit
deny	$(-1,-1)$	$(-10,0)$
admit	$(0,-10)$	$(-5,-5)$

The Nash equilibrium is (admit,admit). This is Nash because: if suspect 2 admits, then for suspect 1 the output -5 (first argument) in $(-5,-5)$ is better than the output -10 in $(-10,0)$, therefore also for suspect 1 it is better to stick to the choice 'admit'. Similarly, if suspect 1 chooses 'admit', it is better for suspect 2 to stick to his choice, because similarly for 2 it holds that the output -10 in outcome $(0,-10)$ is worse then the output -5 in $(-5,-5)$. Of course, the situation is symmetrical for 1 and 2. This does not always have to be the case.

In this example the pair (admit, admit) is the only Nash equilibrium. The pair (deny, deny) is not a Nash equilibrium, because is 2 chooses deny then for 1 it is better to choose admit. Output 0 instead of -1. The pair (confess, deny) is also not a Nash equilibrium, because if 1 admits then for 2 it is better to change his choice and also to admit: output -5 instead of -10. Similarly, (deny, confess) is also not a Nash equilibrium.

So they end up being five years in jail both. The players have no rational incentive for cooperation. If they had cooperated they could have agreed both to deny, which would have meant only a single year in prison for both!

In the previous example it is always better for suspect 1 to deny, no matter what 2 chooses. If a player has a strategy that is always better no matter what the other players do, this is called a *dominant* strategy. Player 1 has a dominant strategy. Also player 2 has a dominant strategy, the same: 'deny'.

Example 8.15 We now get to some variations of Example 8.14. Consider non-cooperative behaviour between Coca Cola and Pepsi Cola. First, they agree to sell their sodas at a high price. The gain is then 6 for each company — assume some unity of gain, like a billion dollars. Of course, both *can* decide not to honour the agreement and sell at a lower price. In that case, assume that the one with the lower price earns 8 units (less profit, but far more sales), whereas the one that keeps the agreement makes 2 only. If both sell at that lower price they will both make 3 units of profit only... We get

	high	low
high	$(6,6)$	$(2,8)$
low	$(8,2)$	$(3,3)$

The non-cooperative pair (low,low) is the equilibrium. Rational behaviour therefore predicts that both parties will break the agreement. A kartel would have given a better result for both companies. Customers are better served by competition.

Example 8.16 Now consider this third version. Two students, Alice and Bob, collaborate when writing a report. There are, as always, two options: work very hard and see to it that everything is finished on time, *or* take it easy and see where the buck stops. If both work hard, the outcome is $(6, 6)$. If both are lazy, the outcome is $(3, 3)$. The best option for Alice (8) is to do nothing and let Bob do all of the work, with much extra effort on his part (so that he get the reward of 2 only), and vice versa. Again, we get the matrix above. Nash equilibrium therefore predicts that collaboration should fail.

Still, this is not true: collaboration often works. Why? This is because you tend to play such games more than once, and you remember the previous behaviour of your opponent / partner in crime and collaboration. Explaining that in detail would make an entirely different story, namely that of repeated games. It would define another game. In the infinitely repeated game, collaborate is after all a Nash equilibrium.

In all these cases communication (apart from repetition) is a way to come to collaboration. If the the two prisoners can exchange information and trust each other sufficiently, they can agree to keep silent and keep their promise. For companies such agreements (collusion!) are illegal. Although the same game, it all depends on your perspective. Agreements between companies are illegal because the equilibrium (low, low) is most in the interest of the consumers.

Example 8.17 A version on Prisoners' Dilemma is known as Arms Race, where the strategies are 'disarm' and 'arm'. An example is

	disarm	arm
disarm	$(3, 3)$	$(0, 4)$
arm	$(4, 0)$	$(1, 1)$

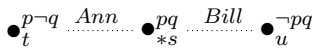
The Nash equilibrium is 'arm', even though both parties have an interest in disarmament. A variation of the game is as follows:

	disarm	arm
disarm	$(3, 3)$	$(0, 4)$
arm	$(4, 0)$	$(-10, -10)$

In this version, the outcome for (arm,arm) is strongly negative (MAD: 'mutually assured destruction'). Now (arm,arm) is not a Nash equilibrium. Note that (arm, disarm) and (disarm,arm) are now also Nash equilibria.

8.8 Logical game — Equilibria in knowledge games

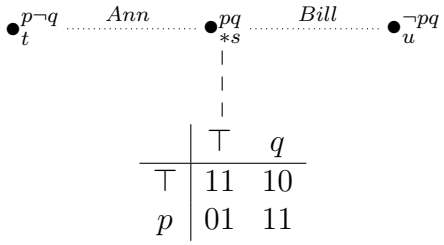
Reconsider the game where Ann knows p and Bill knows q and where they both can choose between two different announcements. The initial model was



and the goals were

$$(K_B p \vee K_B \neg p) \rightarrow (K_A q \vee K_A \neg q)$$
$$(K_A q \vee K_A \neg q) \rightarrow (K_B p \vee K_B \neg p)$$

Both players could choose between two possible moves, and there were four resulting models i, ii, iii, and iv. Let us say that a player who realizes his goal gets payoff 1, and that otherwise the payoff is 0. This results in the following payoff matrix.

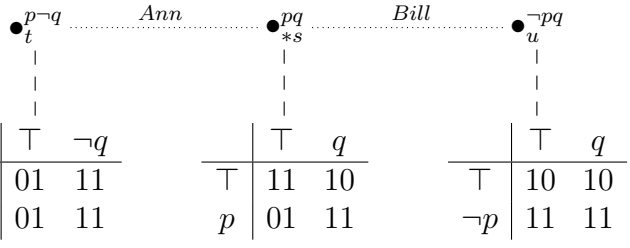


	$\top$	q
$\top$	11	10
p	01	11

This game has therefore two Nash equilibria, namely (p, q) and $(\top, \top)$. This does not help Ann to choose between playing p and playing $\top$, as she does not know what Bill is going to do.

But we even have another problem: Ann does not know that the actual state is s, and that therefore the game that is played is the one above. For all she knows, the actual state could be t. In that case a different game would be played, namely one wherein Ann again can choose between announcing p or $\top$, but Bill can choose between announcing $\neg q$ or $\top$ – instead of knowing q Bill now knows the opposite, but we see as the same q-move, that consists of observing whether q is known and if so, announce it, and if not, announce that you don't know q. The latter is in this particular model equivalent to knowing $\neg q$, so one might as well announce that. If a player does not know what game is being played, this is called an *imperfect information game*.

The situation for Bill is different from the situation for Ann, as he cannot distinguish state s from state u. So in fact we are facing three different games with different payoff matrices, as follows:



	$\top$	$\neg q$
$\top$	01	11
p	01	11

	$\top$	q
$\top$	11	10
p	01	11

	$\top$	q
$\top$	10	10
$\neg p$	11	11

The remainder of this subsection is merely to illustrate a further complication, and does not belong to the core of this chapter. If the players do not know which game is being played, how should they choose between these three games? Can we define a single imperfect information game that incorporates this uncertainty? This is indeed, and of course, possible. It is possible to associate a payoff matrix with the model, independent from which of the three states s, t, u is the case. This we do by redefining game moves conditional to the state that a player finds him- or herself in, and for each condition listing the different moves. There are now many more moves – let a^i_A be move i for Ann and a^j_B be move j for Bill:

- a^1_A: if Ann knows p then she announces $\top$, else she announces $\top$
- a^2_A: if Ann knows p then she announces $\top$, else she announces $\neg p$
- a^3_A: if Ann knows p then she announces p, else she announces $\top$
- a^4_A: if Ann knows p then she announces p, else she announces $\neg p$
- a^1_B: if Bill knows q then he announces $\top$, else he announces $\top$
- a^2_B: if Bill knows q then he announces $\top$, else he announces $\neg q$
- a^3_B: if Bill knows q then he announces q, else he announces $\top$
- a^4_B: if Bill knows q then he announces q, else he announces $\neg q$

In order to compute the payoffs, we need to check the payoffs for each state and combination of game moves / strategies. We have the following:

$\mathbf{a^x_A, a^y_B}$	**t**	**s**	**u**
$1,1$	01	11	10
$1,2$	11	11	10
$1,3$	01	10	10
$1,4$	11	10	10
$2,1$	01	11	11
$2,2$	11	11	10
$2,3$	01	10	11
$2,4$	11	10	11
$3,1$	01	01	10
$3,2$	11	01	10
$3,3$	01	11	10
$3,4$	11	11	10
$4,1$	01	01	11
$4,2$	11	01	11
$4,3$	01	11	11
$4,4$	11	11	11

On various additional assumptions, such as that a priori each initial state of the three states is equally likely to be the case, and that a posteriori – after noticing where in the model you live – each state in your equivalence class is equally likely, we get the following payoff matrix. The payoffs are now expected payoffs, and for convenience of presentation we do not divide the payoffs by the number of states (the equilibria do of course not depend on this):

	$\mathbf{a^1_B}$	$\mathbf{a^2_B}$	$\mathbf{a^3_B}$	$\mathbf{a^4_B}$
$\mathbf{a^1_A}$	$\underline{22}$	$\underline{32}$	21	31
$\mathbf{a^2_A}$	$\underline{23}$	32	22	32
$\mathbf{a^3_A}$	12	22	$\underline{22}$	$\underline{32}$
$\mathbf{a^4_A}$	13	23	$\underline{23}$	$\underline{33}$

The Nash equilibria are underlined. For example, When Ann plays strategy a^2_A and Bill plays strategy a^2_B the outcome in the matrix is (23), which means that Ann can expect a payoff of $\frac{2}{3}$ and that Bill can expect a payoff of $\frac{3}{3} = 1$, etc. Note that the strategies are indeed independent from states: Anne's strategy p consists of observing whether she knows p, if so, announce p ('I know p'), if not, announce $\neg p$ ('I don't know p, in the model equivalent to 'I know $\neg p$').

In game theory, the area of imperfect information games such as the above is full of pitholes and complexities, that we will not diverge into; but the applications of these games to 'knowledge games' as the above example seem numerous and promising.

8.9 Outlook — The limits of rationality

We now reach the limits of game theory and its applications. Humans do not always take rational decisions—and the challenge for logic and logicians, in collaboration with cognitive science, is to find precise and formal explanations for this. Consider the following *ultimatum game* that has received wide-spread attention.

The ultimatum game is a game between two players. Player 1 is given the amount of 100.000 euro, and may distribute this amount over the two players. Then, player 2 may decide whether the players can keep their allocated sums, or not. To be precise: if player 2 agrees to the distribution, he gets the amount player 1 has attributed to him, and 1 gets the amount allocated to himself. Otherwise, both players get nothing. The game is played once only, the players are total strangers to each other, and will never meet each other again.

Rational choice predicts that player 2 must accept any allocated amount larger than zero. Even if player 1 allocates a single euro to player 2 and keeps 99.999 himself, it is rational to accept the allocation because 1 euro is more than 0 euros, the alternative. But experiment shows that player 2 will turn down any distributed allocation that he considers 'too low' and 'unfair'. A 'reasonable' distribution allocates at least 30% to player 1...

How come? A possible explanation is that humans are hardwired to take long-term consequences of their decisions into account. (And of course, although we present the ultimatum game here as a one-shot game, the reality is that we face the other players that

we treated unfairly again and again.) How about our 'reputation' if we accept a very low amount? We could be called cheap! And that would be, obviously, disastrous. We are now getting into the areas where game theory and logic overlap with cognitive science and psychology, an ever shifting boundary for further logical enterprises into areas possibly benefiting from formalization.

Further Exercises

Exercise 8.18 Stefan

Exercise 8.19 Stefan

Exercise 8.20 Stefan

Summary *After having finished this chapter you should have mastered the following concepts and techniques:*

- *playing verifier and falsifier in Kripke models, given a formula (know the rules of the game);*
- *playing bisimulation games with spoiler and duplicator, difference formula;*
- *the concept of bisimulation, and constructing simple bisimulation relations;*
- *compute equilibrium from pointed knowledge game (non-pointed version, with expected payoff matrix, not required);*
- *game theory terminology: game tree, move, zero-sum game, Zermelo's Theorem, winning strategy, Nash equilibrium, determined game;*
- *proof of Zermelo's Theorem.*

Methods

Chapter 9

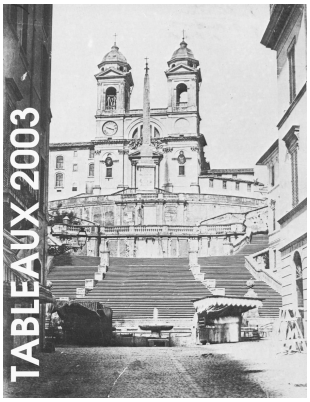
Validity Testing

In the first three chapters various methods have been introduced to decide the validity of different sorts of inferences. We have discussed truth-tables and the update method for propositional logic, and also a method using Venn-diagrams for syllogistic reasoning. In this chapter we will introduce a uniform method to decide validity for the logics of the first part of this book. This method has been introduced for propositional logic and predicate logic by the Dutch philosopher and logician Evert Willem Beth (1908-1964) in the fifties of the previous century.

The basic idea behind this method comes down to the following principle which we have stressed at earlier occasions. ref

> An inference is valid if and only if there exists *no* counter-examples, i.e., there is no situation in which the premises hold and the conclusion is false.

The method consists of a rule-based construction of a counter-example for a given inference. Each step of the construction is given account of in a tree-like structure which is called a *tableau*. During this construction it may be that, due to conflicting information, the system detects that no counter-examples can be constructed. We speak of a *closed tableau* in such a case, and it implies that no counter-examples exist. We may then safely conclude that the inference which we are analyzing must be valid.



Evert Willem Beth (left). TABLEAUX is a large biannual conference where computer scientists and logicians meet to present and discuss the latest developments on tableau methods and their application in automated reasoning systems.

The tableau method is a very powerful method. It is complete for propositional and predicate logical reasoning. This means that in case of a valid inference the validity can always be proved by means of a *closed* tableau, that is, the exclusion of counter-examples. Moreover, the tableau method can be implemented quite easily within computer programs, and is therefore used extensively in the development of automated reasoning systems.

In the case of propositional logic the tableau method we will discuss here can generate *all* counter-models for invalid inferences. In this respect, the situation in predicate logic is quite different. If an inference is invalid a counter-model must exist, but it may be that it can not be constructed by means of the rules of the tableau system. In this chapter we will introduce two tableau systems for predicate logic of which one is better (but a bit more difficult) than the other in finding counter-models for invalid inferences, but still this more advanced system is not able to specify infinite counter-models, which means that invalid inferences with *only* infinite counter-models — we will see one example in the section on predicate logic — their invalidity can not be demonstrated by this system. In fact, a perfect tableau system does not exist for predicate logic. Since the thirties of the previous century, due to the work of Alonzo Church and Alan Turing, we know that there exists no decision method in general which detects invalidity for all invalid predicate logical inferences.

9.1 Tableaus for propositional logic

But let us first start with the propositional logical case. For checking the validity of a propositional logical inference we can use the method of truth-tables (Chapter 2). If we have an inference $\varphi_1, ..., \varphi_n/\psi$ then we need to set up truth-tables for all the formulas $\varphi_1, ..., \varphi_n, \psi$ and then see whether there is one row, at which the formulas $\varphi_1, ..., \varphi_n$ are all true (1) and ψ is false (0). If this is the case we have detected a counter-model, and

then the inference must be invalid. If such a row can not be found then the inference must be valid since it does not have counter-models in this case.

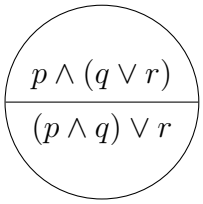
The tables are built up step by step, assigning truth-values to the proposition letters, who represent some atomic bit of propositional information, and then assigning truth-values to all the formulas following the grammatical structures of the formulas. It is therefore called a *bottom up* method.

The tableau method works exactly in the opposite direction: *top-down*. It starts with the original inference and then tries to break it down into smaller pieces. If it arrives at the smallest parts, the proposition letters, and has not run into contradictions then this atomic information can be used to specify a counter-model and invalidity for the given inference has then been proved. If it does not succeed to do so then the tableau is a proof that no counter-model exists, and in this case, the inference must be valid.

Let us get more specific and take a simple valid inference:

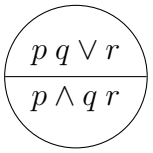
$$p \wedge (q \vee r) \models (p \wedge q) \vee r \tag{9.1}$$

We start with a simplistic representation of a candidate counter-example. It depicts a world with two hemispheres of which the true information is contained in the upper half, and the false information in the lower part.



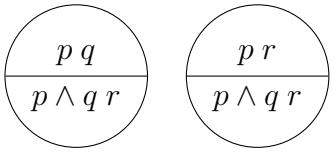
(9.2)

The truth-conditions of the propositions, as defined by the connectives they contain, determine whether this potential counter-example can be realized. As the only true formula is a conjunction, we know that the two conjuncts must be true. The only false proposition is a disjunction, and therefore both these disjuncts must also be false. This leads to the following further specification of our potential counter-example:

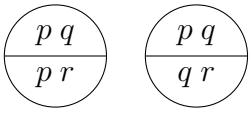


(9.3)

We know now that our candidate counter-example must at least support p and falsify r. The exclusion of six other valuations has already taken place by this simple derivation. Still it is not sure whether the picture in (9.3) captures a real counter-example since $q \vee r$ must be true and $p \wedge q$ must be false. The first formula is a disjunction and because it is true, the formula itself does not give us accurate information about the truth-values of the the arguments q and r. The only thing we know is that at least one of them must be true. This makes our search more complicated. The following two candidates are then both potential counter-examples.



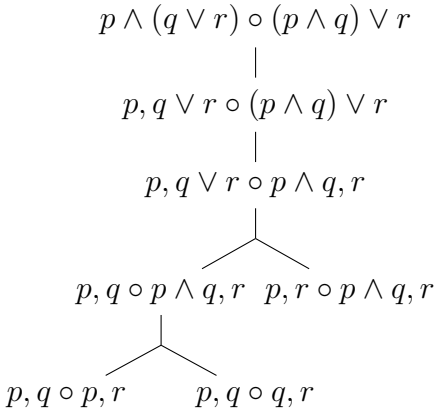
The world on the right hand can not be a counter-example because it requires r to be both true and false. This can never be the case in one single world, and therefore this possibility has to be canceled as a counter-model. The candidate on the left contains a false conjunction, $p \wedge q$. Again, this gives us no precise information, since the falsity of a conjunction only claims the falsity of at least one of the conjuncts. As a consequence, this world must be separated into the following two possibilities.



(9.4)

The first of these two possibilities can not represent a real world because p is both true and false there. The second can not be realized either since q is both true and false in this case. Real counter-examples for this inference do not exist! The inevitable conclusion is that $(p \wedge q) \vee r$ must be true whenever $p \wedge (q \vee r)$ is true.

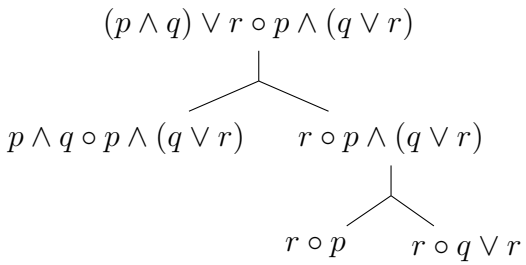
For sake of notation, we will not continue to use the encircled representations as in this first example. We will use a little circle $\circ$ instead and write the *true* formulas on its *left* side, and the *false* formulas on the *right* side of this circle. Doing so, we can summarize our search for a counter-example for the inference $p \wedge (q \vee r)/(p \wedge q) \vee r$ as a tree in the following way.



(9.5)

Each node in the tree is called a *sequent*. A tree of sequents is called a *tableau*. A branch of such a tableau is *closed* if its end node contains a sequent with a formula which appears both on the left (true) *and* on the right (false) part of the sequent. It means that this branch does not give a counter-example for the sequent as given at the top of the tableau. If all branches are closed then the tableau is also *closed*, and it says, just as in the earlier example, that the top-sequent represents in fact a valid inference. A branch of a tableau is

called *open* if its final node is not closed and contains no logical symbols. In this case we have found a counter-example since there are only propositional letters left. A valuation which assigns the value 1 to all the proposition letters on the left part of such a sequent in this end node and 0 to those on the right side will be a counter-model for the inference with which you started the tableau. To illustrate this we can take the earlier example and interchange premise and conclusion. The inference $(p \wedge q) \vee r / p \wedge (q \vee r)$ is an invalid inference, and by using the tableau method we should be able to find a counter-model.



(9.6)

In the first step we have removed the disjunction on the left which led to two possibilities. Then in the second resulting sequent we have removed the conjunction on the right part of the sequent, which led to two new possibilities. The final node with the sequent $r \circ p$ represents a real counter-example. This branch is open. A valuation V with $V(p) = 0$ and $V(r) = 1$ is a counter-model indeed: $V((p \wedge q) \vee r) = 1$ and $V(p \wedge (q \vee r)) = 0$. In fact, this open branch represents two counter-examples since the truth-value of q does not matter in this case. The situations $\overline{p}\overline{q}r$ and $\overline{p}qr$ are both counter-examples.

The reader may check for himself that the other branches do not give other counter-models. They all close eventually. This means that there are only two counter-models. The tableau as given in (9.6) suffices as a proof of invalidity here. As soon as an open branch has been constructed it is not needed to inspect the other branches.

9.1.1 Reduction rules

A proper tableau needs to be set up according precise *reduction rules*. A reduction rule is specified by the logical symbol that is to be removed, and the truth-value of the formula as indicated by the sequent (left or right of the truth-falisity separation symbol $\circ$). Such a rule is defined by the truth-conditions for the logical symbols. The following schema depicts the rules for conjunction and disjunction, which we already have used in the previous

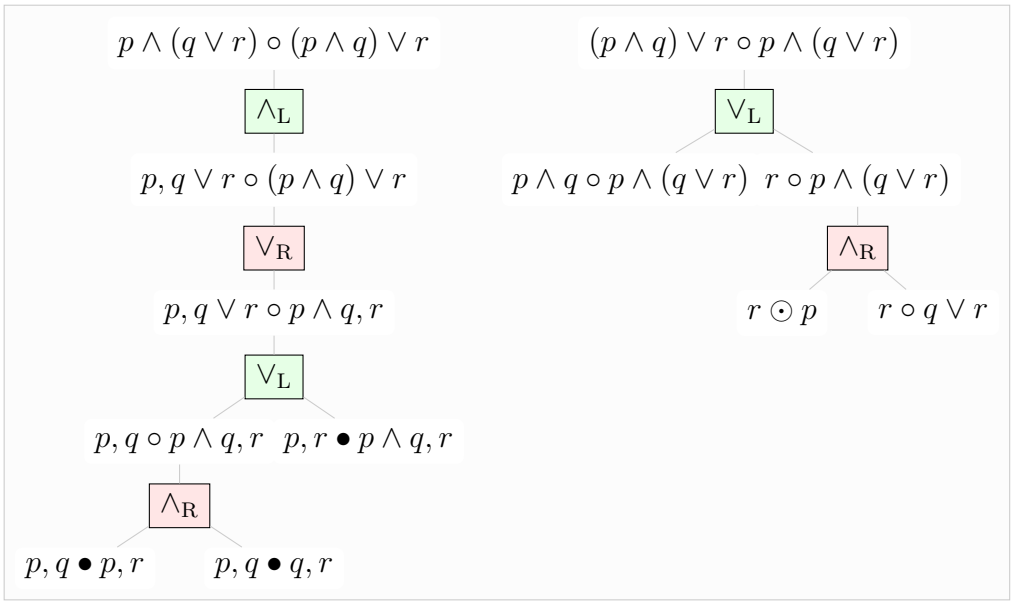
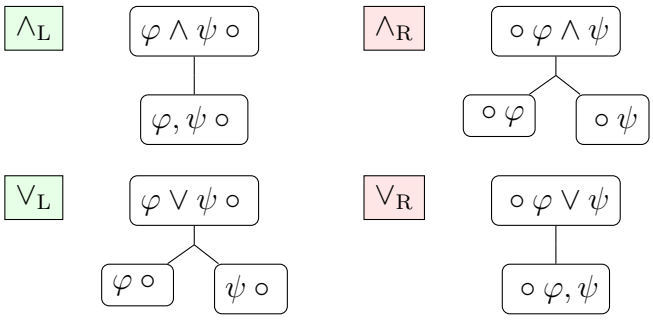


Figure 9.1: Complete tableaus for the earlier examples.

examples.



(9.7)

The rules $\wedge_{\mathrm{L}}$ and $\vee_{\mathrm{L}}$ tell us what to do with a conjunction and disjunction, respectively, when it appears on the left side of a sequent. We use a green background here to make it explicit that when we apply such a rule, we are working on a formula which is claimed to be true. The R-rules are rules which deal with false conjunctions and disjunctions. The background color red is used to stress that we are reducing a formula which is claimed to be false.

In figure 9.1 the earlier examples are given once more, but extended with specifications of the rules we use in each step. As to distinguish open and closed branches we replace the truth-falsity separation symbol $\circ$ by $\odot$ and $\bullet$ respectively. We will continue to use this way of indication in the sequel of this chapter.

For the other connectives rules can be given quite straightforwardly by using the truth-conditions which have been defined in the introductory chapter on propositional logic. The negation rules are the most simple ones. A negation switches truth-values, so the

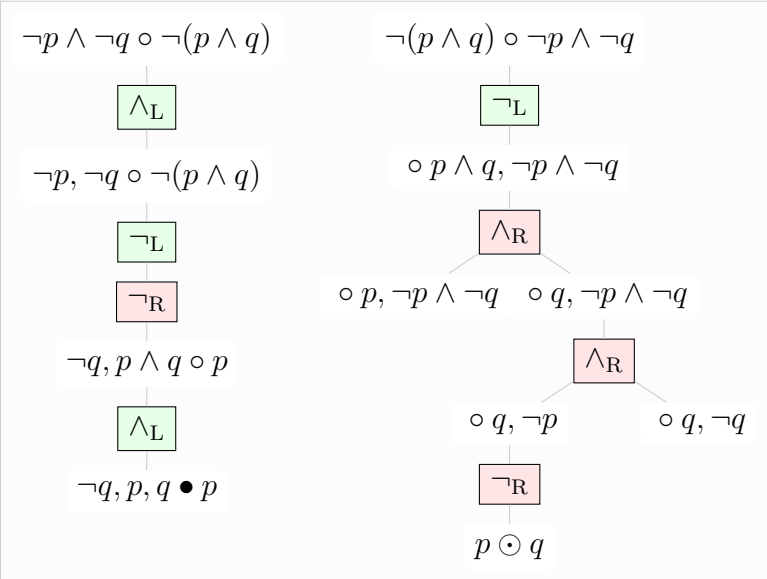
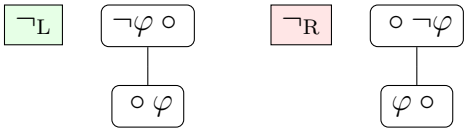


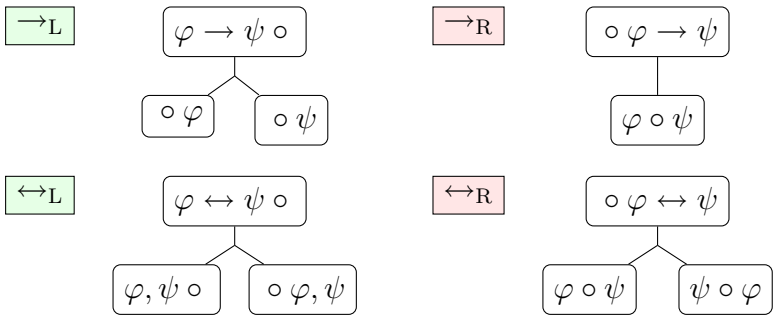
Figure 9.2: Two tableaus with negations. The left tableau shows that $\neg p \wedge \neg q \models \neg(p \wedge q)$. The right tableau shows that the converse of this inference is not valid $\neg(p \wedge q) \not\models \neg p \wedge \neg q$. The counter-model which has been found in the open branch is the valuation which assigns 1 to p and 0 to q. This suffices to show the invalidity of the inference. If we would have worked out the left branch as well we would have found the other counter-example $\overline{p}q$.

proper way to remove a negation is to transfer its argument from one side of the sequent to the other.



(9.8)

In Figure 9.2 two simple tableaus are given with occurences of negations. The rules for implication and equivalence are the following:



(9.9)

The rules for equivalence are quite easy ato understand. An equivalence is true if the truth-values of the two arguments are the same. In terms of reductions this means that

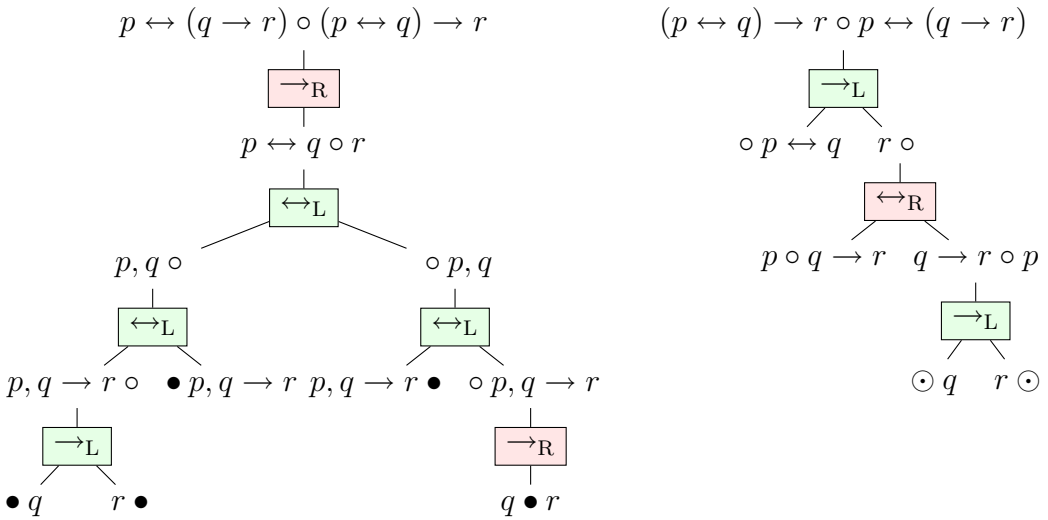
if the equivalence appear on the left hand side of the sequent the two arguments remain on the left hand side (both true) or they switch both to the right hand side (both false). If an equivalence is false the two truth-values of the arguments differ, which gives the two possibilities as captured by $\leftrightarrow_{\mathrm{R}}$-rule as shown in the schema (9.9).

The R-rule for implication captures the only possibility for an implication to be false. The antecedent should be true (moves to the left) and the consequent should be false (stays on the right). The L-rule captures the other possibilities: φ is false (moves to the right) or ψ is true (stays on the left).

Exercise 9.1 Define appropriate reduction rules for the exclusive disjunction $\sqcup$. Remember that $\varphi \sqcup \psi$ is true if and only if exactly one of the arguments φ or ψ is true.

Exercise 9.2 Show that $\neg(\varphi \sqcup \psi)$ is logically equivalent with $\neg\varphi \sqcup \psi$ using the rules that you have defined for $\sqcup$ in the previous exercise. You will need two tableaus here, one for proving that $\neg(\varphi \sqcup \psi) \models \neg\varphi \sqcup \psi$ and one for proving that $\neg\varphi \sqcup \psi \models \neg(\varphi \sqcup \psi)$.

Below in (9.10) two tableaus are given of which the first shows that $p \leftrightarrow (q \to r) \models (p \leftrightarrow q) \to r$. The second demonstrates that the converse is invalid.



(9.10)

For sake of shorter notation we have left out the repetition of formulas, and only kept track of new formulas. This makes it bit harder to read the tableau, but it may be worth the effort to get used to this shorter notation since tableaus, especially in the case of predicate logic as we will see in the next section, tend to get very large.

In order to conclude closure of a branch we need to scan it for contradiction in backward direction. This also is the case for defining a counter-model for an open branch. For example, the counter-examples as given in the right-most branch in the second tableau of (9.10) are those who falsify p and verify r. About the proposition letter q no information is given in this open branch.

Exercise 9.3 Use tableaus to test the validity of the following inferences.

(1) $p \vee (q \wedge r)/(p \vee q) \wedge (p \vee r)$

(2) $p \to q, q \to r/\neg r \to \neg p$

Exercise 9.4 Use the tableau method to find out whether the following sets of formulas are consistent (satisfiable), i.e., check whether there is a valuation which makes all the formulas in the given set true.

(1) $\{p \leftrightarrow (q \vee r), \neg q \to \neg r, \neg(q \wedge p), \neg p\}$

(2) $\{p \vee q, \neg(p \to q), (p \wedge q) \leftrightarrow p\}$

Exercise 9.5 Check, using the tableau method, whether the following formulas are tautologies or not.

(1) $(p \to q) \vee (q \to p)$

(2) $\neg(p \leftrightarrow q) \leftrightarrow (\neg p \leftrightarrow \neg q)$

9.2 Tableaus for predicate logic

A tableau system consists of the rules for the connectives as given in the previous section and four rules for the quantifiers, two rules for each of the two quantifiers $\forall$ and $\exists$.[1] These rules are a bit more complicated because the quantifiers range over the individual objects in the domain of the models. Beforehand, however, we do not know how many of those individuals are needed to provide real counter-models. The domain has to be constructed step by step. This makes it harder to process universal information adequately because it need to be applied to *all* the objects, and it may be that it will not be clear at that stage what the required set of objects is to provide a counter-example. In simple cases this can be avoided by dealing with the existential information first. Let us have a look at such an easy going example:

$$\forall x\,(Px \vee Qx)/\forall x\,Px \vee \forall x\,Qx \tag{9.11}$$

It may be clear that this an invalid inference. Every integer is even or odd ($P \vee Q$) but it is surely not the case that all integers are even (P) or that all integers are odd (Q). Let us see what happens if we want to demonstrate this with a tableau. At first we can apply the $\vee_{\mathrm{R}}$-rule as we have defined in the previous section:

$$\begin{array}{c} \forall x\,(Px \vee Qx) \circ \forall x\,Px \vee \forall x\,Qx \\ \Big| \\ \boxed{\vee_{\mathrm{R}}} \\ \Big| \\ \forall x\,(Px \vee Qx) \circ \forall x\,Px, \forall x\,Qx \end{array} \tag{9.12}$$

[1] For sake of keeping things simple, we will not deal with the equality sign $=$ and function symbols here. Moreover, we assume that all formulas contain no free variables.

For the potential counter-model this means the following. All individuals are $P \vee Q$-s but not all of them are P-s and not all of them are Q-s, since $\forall x\, Px$ and $\forall x\, Qx$ must be falsified. They occur on the right side of the last sequent. A universally quantified formula $\forall x\, \varphi$ on the right hand side of a sequent conveys an *existential* claim, we need at least one non-φ-er within the candidate counter-model. As we said earlier, it is better to deal with this existential information first. Removal of the formula $\forall x\, Px$ can be done by replacing it by Pd_1 where d_1 is some additional name for the object which does not have the property P. We do not know who or what this non-P-object is, and therefore we need a neutral name to denote it. So our next step is:

$$\begin{array}{c} \forall x\,(Px \vee Qx) \circ \forall x\, Px, \forall x\, Qx \\ \hline \forall_{\mathrm{R}} \\ \forall x\,(Px \vee Qx) \circ Pd_1, \forall x\, Qx \end{array} \tag{9.13}$$

Elimination of the last universal quantifier on the right hand side requires a non-Q-object. This object may be different from d_1 and therefore we choose a new neutral name d_2.[2]

$$\begin{array}{c} \forall x\,(Px \vee Qx) \circ Pd_1, \forall x\, Qx \\ \hline \forall_{\mathrm{R}} \\ \forall x\,(Px \vee Qx) \circ Pd_1, Qd_2 \end{array} \tag{9.14}$$

At this stage we have to eliminate the universal quantifier on the left hand side of the sequent. We need to apply the property $Px \vee Qx$ to all the objects in the domain. This far we only have objects called d_1 and d_2 and therefore we *only* apply it to those objects, which brings two new formulas on the stage $Pd_1 \vee Qd_1$ and $Pd_2 \vee Qd_2$. In this case we are sure that no other objects may be needed because all the existential information has been dealt with in the two steps before.

$$\begin{array}{c} \forall x\,(Px \vee Qx) \circ Pd_1, Qd_2 \\ \hline \forall_{\mathrm{L}} \\ Pd_1 \vee Qd_1, Pd_2 \vee Qd_2 \circ Pd_1, Qd_2 \end{array} \tag{9.15}$$

[2]Note that we do not exclude the possibility that d_1 and d_2 are equal here. In predicate logic it is possible that one object carries two names.

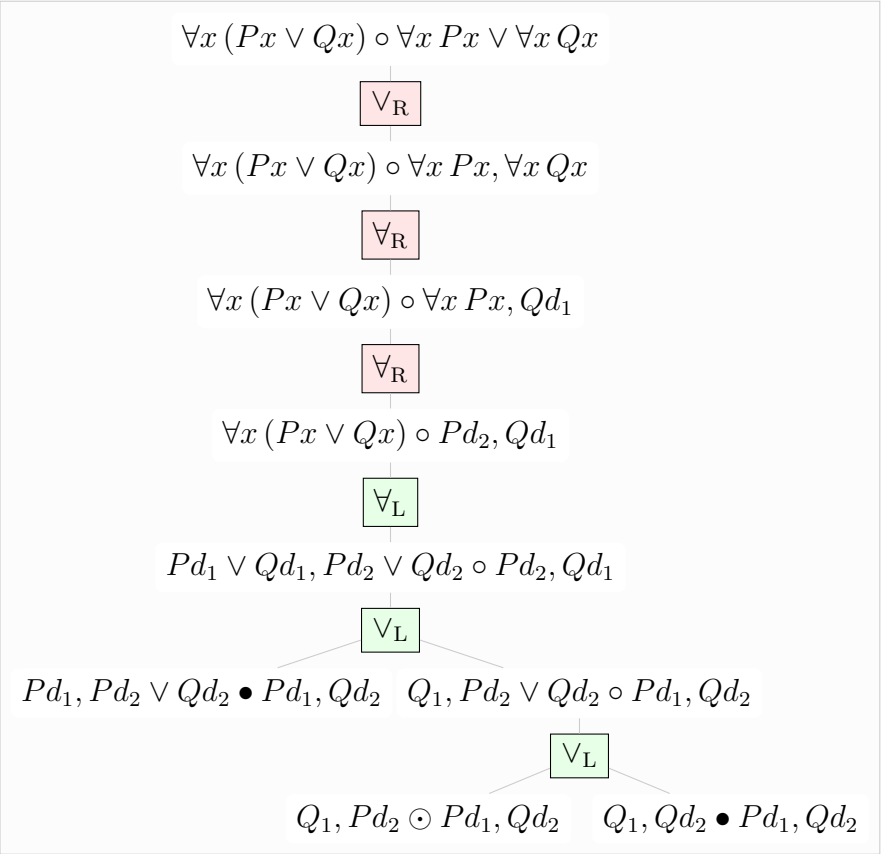
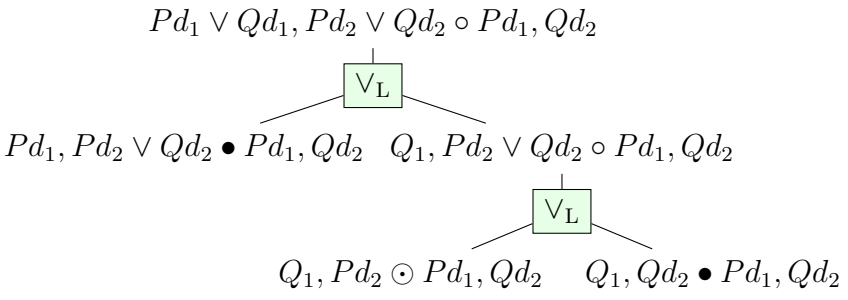


Figure 9.3: The full tableau demonstrating that $\forall x\,(Px \vee Qx) \not\models \forall x\,Px \vee \forall x\,Qx$. The counter-example contains a P who is not Q and a Q who is not P.

The last two steps deal with the two disjunctions.



(9.16)

Finally, we have found a counter-model. The open branch tells us that we need a model with two objects. The first one needs to be a Q-object which does not have the property P, and the second has to be a P-object which does not have the property Q. This is indeed a counter-model for the original inference as given in (9.11). In Figure 9.3 the full tableau is given.

Exercise 9.6 Show with a tableau that $\exists x\,(Px \wedge Qx) \models \exists x\,Px \wedge \exists x\,Qx$.

Exercise 9.7 Show with a tableau that $\exists x\,Px \wedge \exists x\,Qx \not\models \exists x\,(Px \wedge Qx)$.

Exercise 9.8 Show with a tableau that $\forall x\,(Px \vee Qx) \models \forall x\,Px \vee \exists x\,Qx$.

In the last example it we dealt with existential information before we used the universal information. This is not always possible. Here is a short but more complicated case.

$$\exists x\,(Px \to \forall y\,Py) \tag{9.17}$$

The formula says that there exists an object such that if this object has the property P then every object has the property P. We do not know which object is meant here so let us give it a neutral name. The formula then reduces to $Pd_1 \to \forall y\,Py$. Such an object can then always be chosen. If all objects have the property P then it does not matter which object you choose, since the consequent is true in this case. If, on the other hand, not all objects have the property P then you can pick one of the non-P-objects for d_1. The antecedent is then false and therefore the implication $Pd_1 \to \forall y\,Py$ holds. In other words, $\exists x\,(Px \to \forall y\,Py)$ is *valid*.

In order to prove that (9.17) is valid by means of a tableau we have to show that it never can be false. Putting it on the right side of the top-sequent, we then should be able to construct a closed tableau. Here is a first try in three steps.

$$\begin{array}{c} \circ\ \exists x\,(Px \to \forall y\,Py) \\ \downarrow\ \exists_{\mathrm{R}} \\ \circ\ Pd_1 \to \forall y\,Py \\ \downarrow\ \to_{\mathrm{R}} \\ Pd_1 \circ \forall y\,Py \\ \downarrow\ \forall_{\mathrm{R}} \\ Pd_1 \circ Pd_2 \end{array} \tag{9.18}$$

Foremost, we need to explain the first step. An existential quantified formula on the right yields a universal claim. If $\exists x\,\varphi$ is false it means that there exists *no* φ-er: φ is false for all individuals in the domain. Since there are no objects introduced so far the reader may think that this leads to an empty sequent. But in predicate logic we have a minimal convention that every model has at least one object.This means that if we want to fulfill a universal claim, that is, a true formula of the form $\forall x\,\varphi$ or a false formula of the form $\exists x\,\varphi$, and there are no objects introduced so far then we introduce one. This is what has been done in the first step in (9.18).

The second and the third step are as before. Now, it may seem as if we have an open branch here since there is no contradictory information and there are no logical symbols

left. But we made a logistic mistake here. We removed the false formula $\forall y\, Py$ here by introducing a new non-P-object called d_2. The universal claim by the false formula $\exists x\,(Px \to \forall y\, Py)$ however has been applied to d_1 only, whereas $Px \to \forall y\, Py$ has to be false for *all* objects, and therefore, also for d_2! In tableau-systems for predicate logic this means that whenever a new name is to be introduced the formulas which have universal strength which have been removed at an earlier stage in the tableau will become active again, and then need to be dealt with at a later stage. So the last step of (9.18) need to be extended in the following way:

$$\begin{array}{c} Pd_1 \circ \forall y Py \\ \hline \forall_{\mathrm{R}} \\ \hline Pd_1 \circ Pd_2, \exists x\,(Px \to \forall y\, Py) \end{array} \tag{9.19}$$

The formula $\exists x\,(Px \to \forall y\, Py)$ is supposed to be falsified in the end, and becomes active again when the new object called d_2 is introduced. The next step then is to deny the property $Px \to \forall y\, Py$ for all objects. Since it has been already denied for d_1 in the first step in (9.18), the only new information is that $Pd_2 \to \forall y\, Py$ must be false.

$$\begin{array}{c} Pd_1 \circ Pd_2, \exists x\,(Px \to \forall y\, Py) \\ \hline \exists_{\mathrm{R}} \\ \hline Pd_1 \circ Pd_2, Pd_2 \to \forall y\, Py \end{array} \tag{9.20}$$

One may think, at first sight, that this leads to an infinite procedure. In this case, things work out nicely, since the tableau closes in the next step. The implication will be removed, and then we run into a conflict: Pd_2 must be true and false at the same time.

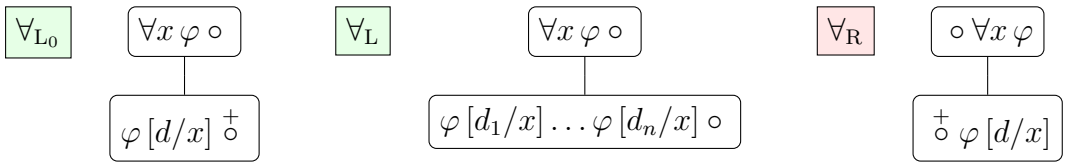
$$\begin{array}{c} Pd_1 \circ Pd_2, Pd_2 \to \forall y\, Py \\ \hline \to_{\mathrm{R}} \\ \hline Pd_1, Pd_2 \bullet Pd_2, \forall y\, Py \end{array} \tag{9.21}$$

This means that there are no models which falsify $\exists x\,(Px \to \forall y\, Py)$. This formula must be valid.

9.2.1 Rules for quantifiers

In the two examples above we have indicated how we should deal with quantified predicate logical formulas in a tableau. Here we want to give a formal status to the reduction

rules for the quantifiers. Let us start with the universal quantifier.



(9.22)

There are two left rules for the universal quantifier when it appears on the left part of a sequent.

$\forall_{L_0}$: The first rule (0) is meant to deal with the exceptional case when no names are present in the sequent, that is, there are no names to apply the property φ to. In this case we introduce a new name d and replace all free occurences of x in φ by d. We write this as $\varphi\,[d/x]$. In addition, the truth-falsity separation symbol $\circ$ is designated with a $+$ on top to indicate that a *new* name has been added within the branch of the tableau.

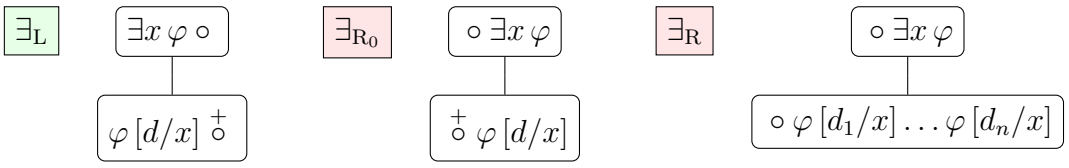
$\forall_{L}$: If names are present in the input sequent then $\forall x\,\varphi$ can be removed from the left part of the sequent by applying φ to the names $d_1, ..., d_n$, all occurring in the sequent and which φ has *not* been applied to *yet*.

$\forall_{R}$: A false formula $\forall x\,\varphi$ is removed by applying φ to a new name d. This denotes the object we need as an example of a non-φ-er in the counter-model which we are constructing. In order to show that this name is new we use the additional $+$-indication.

In the end we need to distinguish $\circ$ from $\overset{+}{\circ}$-sequents in which new name are introduced.

$\overset{+}{\circ}$: If a new name is introduced then all formulas of the form $\forall x\,\varphi$ appearing on the left part and those of the form $\exists x\,\varphi$ on the right part of preceding sequents in this branch re-appear in the output sequent.

The rules for the existential quantifiers are defined analogously to the rules for the universal quantifier:



(9.23)

The following example, which shows that $\exists y \forall x\, Rxy \models \forall x \exists y\, Rxy$, make use of all the general rules.

$$\begin{array}{c}
\exists y \forall x\, Rxy \circ \forall x\, \exists y\, Rxy \\
\Big| \\
\boxed{\exists_{\mathrm{L}}} \\
\Big| \\
\forall x\, Rxd_1 \overset{+}{\circ} \forall x\, \exists y\, Rxy \\
\Big| \\
\boxed{\forall_{\mathrm{R}}} \\
\Big| \\
\forall x\, Rxd_1 \overset{+}{\circ} \exists y\, Rd_2y \\
\Big| \\
\boxed{\forall_{\mathrm{L}}} \\
\Big| \\
Rd_1d_1, Rd_2d_1 \circ \exists y\, Rd_2y \\
\Big| \\
\boxed{\exists_{\mathrm{R}}} \\
\Big| \\
Rd_1d_1, Rd_2d_1 \bullet Rd_2d_1, Rd_2d_2
\end{array} \tag{9.24}$$

In this example the quantifiers were in optimal position. We could fulfill the existential claims ($\exists_{\mathrm{L}}$ and $\forall_{\mathrm{R}}$) before we dealt with the universal requirements ($\forall_{\mathrm{L}}$ and $\exists_{\mathrm{R}}$) for the potential counter-model. As a result of this no reintroduction of universal information was needed.

In (9.18) we already have seen that this reintroduction can not always be avoided. Fortunately, this did not lead to an infinite procedure, because the tableau could be closed. But in other cases we may run into real trouble due to continuing introduction of new names, and consequently, unstoppable re-appearance of universal information. Below such an example is given. Let us first look at the first two steps.

$$\begin{array}{c}
\forall x\, \exists y\, Rxy \circ \exists y \forall x\, Rxy \\
\Big| \\
\boxed{\forall_{\mathrm{L}_0}} \\
\Big| \\
\exists y\, Rd_1y \overset{+}{\circ} \exists y \forall x\, Rxy \\
\Big| \\
\boxed{\exists_{\mathrm{L}}} \\
\Big| \\
\forall x \exists y\, Rxy, Rd_1d_2 \overset{+}{\circ} \exists y \forall x\, Rxy
\end{array} \tag{9.25}$$

The two formulas in the top-sequent have universal status. The left formula is true and says that every object is R-related to some object. In a domain of persons, taking the relation Rxy to represent the relation 'x loves y', $\forall x \exists y\, Rxy$ means "Everybody loves somebody". The formula $\exists y \forall x\, Rxy$ on the right hand should be falsified, and therefore

the claim is that is not the case that there exists an object such that all objects are R-related to it. In the context mentioned here above, this means that there is no person who is loved by everybody. So, there is no other option than to apply one of the exceptional universal rules $\forall_{\mathrm{L}_0}$ or $\exists_{\mathrm{R}_0}$. We have chosen to take the former.

In the second step we took the new existential formula on the left since we prefer to deal with existential information first. Here we introduced a new name, and therefore, the universal formula which has been removed in the first step pops up again. Repetition of the same procedure would introduce a third object and a second re-appearance of $\forall x \exists y\, Rxy$. If, instead, we would choose to remove the formula $\exists y \forall x\, Rxy$ on the right we would then get the following two successive steps:

$$\begin{array}{c} \forall x\, \exists y\, Rxy, Rd_1d_2 \circ \exists y \forall x\, Rxy \\ \downarrow \\ \boxed{\exists_{\mathrm{R}}} \\ \downarrow \\ \forall x\, \exists y\, Rxy, Rd_1d_2 \circ \forall x\, Rxd_1, \forall x\, Rxd_2 \\ \downarrow \\ \boxed{\forall_{\mathrm{R}}} \\ \downarrow \\ \forall x\, \exists y\, Rxy, Rd_1d_2 \overset{+}{\circ} Rd_3d_1, \exists y \forall x\, Rxy \end{array} \tag{9.26}$$

In the last step a third object is introduced, and then $\exists x \forall y\, Rxy$ re-appears on the right part of the sequent. The sequent in the last node contains the same formulas as in the top node with two additional atomic formulas who do not contradict each other. Moreover, we know that this tableau will never close since the top sequent represents an invalid inference. This branch will *never* end with the desired final sequent free of logical symbols.

Without applying the rules it is not hard to find a simple counter-example. Take the situation of two persons who love themselves but not each other. In such a case, $\forall x \exists y\, Rxy$ is true and $\exists y \forall x\, Rxy$ is false, since there is no person who is loved by everybody. Apparently, our tableau system is not able to find such a simple counter-model. In fact the rules guide us towards an *infinite* counter-example which can never be constructed since in each step at most one additional object is introduced.

Despite this inability of the system, the rules make up a complete validity testing method. If an inference $\varphi_1, ...\varphi_n/\psi$ is valid, $\varphi_1, ..., \varphi_n \models \psi$, then there exists a closed tableau with $\varphi_1, ..., \varphi_n \circ \psi$ as the top sequent. We will not prove this completeness result here, but instead, get into more detail at a later stage.

Exercise 9.9 Test the validity of the following syllogisms with tableaus:

(1) $\forall x\,(Ax \rightarrow Bx), \exists x\,(Ax \wedge Cx)/\exists x\,(Cx \wedge Bx)$

(2) $\forall x\,(Ax \rightarrow Bx), \exists x\,(Ax \wedge \neg Cx)/\exists x\,(Cx \wedge \neg Bx)$

(3) $\neg\exists x\,(Ax \wedge Bx), \forall x\,(Bx \rightarrow Cx)/\neg\exists x\,(Cx \wedge Ax)$

Exercise 9.10 Prove the validity of the following inference with tableaus:

(1) $\forall x\,(Ax \to Bx) \vee \forall y\,(By \to Ay) \models \forall x\,\forall y\,((Ax \wedge By) \to (Bx \vee Ay))$

(2) $\forall x\,\forall y\,((Ax \wedge By) \to (Bx \vee Ay)) \models \forall x\,(Ax \to Bx) \vee \forall y\,(By \to Ay)$

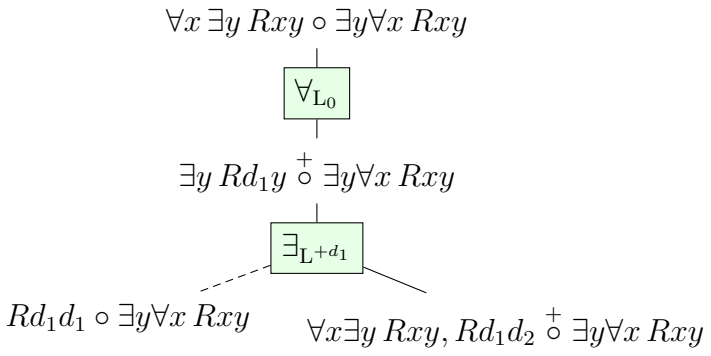
9.2.2 Alternative rules for finding finite counter-models

In (9.25) and (9.26) we have seen an example of an invalid inference with quite simple finite counter-models which can not be found by means of the rules for the quantifiers. In order to find such finite counter-models with a tableau system we need to extend the rules for the quantifiers a bit. The problem with the earlier rules was the introduction of new names which caused repetitive reintroduction of formulas. This can be avoided by facilitating the 'old' objects to support existential information. These extended versions of the 'existential' rules $\exists_{\mathrm{L}}$ and $\forall_{\mathrm{R}}$ have the following general format, where the name d is some name which occurs in the input node and d' does not.

(9.27)

The truth-falsity separation sign $\circ$ only has a $+$-sign in the right branch. In the left branch we have used an old object called d which does not provoke reintroduction of universal information. We indicate these special *try-out branches* with a dashed line.

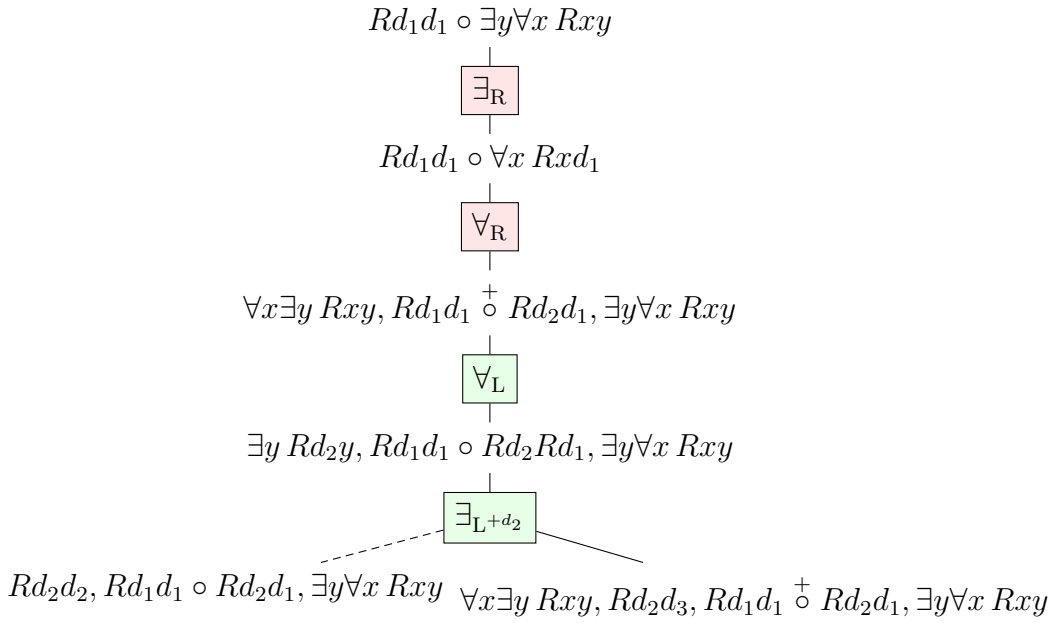
Let us try these extended rules to find a simple finite counter-model for the example we started with in (9.25). Here are the first two steps.



(9.28)

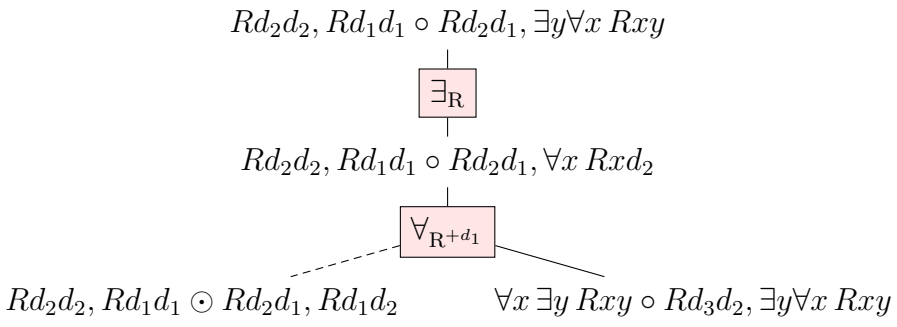
The first step is the same as in (9.25). The second step is the application of the extended version of $\exists_{\mathrm{L}}$. We apply in the left branch the property $Rd_1 y$ to the only known name d_1. In this branch the true formula $\forall x\,\exists y\,Rxy$ is *not* reintroduced. This try-out branch can

then be extended with the following four steps.



(9.29)

In the second step we did not apply $\forall_{\mathrm{R}^{+d_1}}$ but the old version $\forall_{\mathrm{R}}$ instead. A try-out branch would close immediately because of the true formula Rd_1d_1. In the last step we have chosen for $\exists_{\mathrm{L}^{d_2}}$. The d_1-version would have given a closed branch because of the false formula Rd_2d_1. Extension of this new try-out branch results into our desired counter-model in two more steps.



(9.30)

In the first step the false formula $\exists y \forall x\, Rxy$ results into $\forall x\, Rxd_2$ only because $\forall x\, Rxy$ has been applied to d_1 in the third step of this branch (9.29). In the second step we used a d_1-try-out branch. The d_2-variant would have given closure because of the true formula Rd_2d_2. This third try-out branch has finally determined a counter-example. The objects called d_1 and d_2 are R-related to themselves but are mutually not R-related.

It is not hard to see that the d_1-try-out branch in the second step of this tableau (9.28) can not give any other counter-examples with only two objects. If we would have chosen for the regular branch after this second step we could have constructed the other two

object counter-model, that consists of two objects who are mutually related but are not related to themselves. We leave this as an exercise to the reader.

Exercise 9.11 Try to find the other counter-model as mentioned here above using the try-out branches on other places.

Exercise 9.12 Show the invalidity of the following inference with a tableau dressed up with try-out branches. Try to keep the final counter-model as small and simple as possible.

(1) $\forall x \exists y\, Rxy / \forall x \exists y\, Ryx$

(2) $\exists x \forall y\, Rxy / \exists x \forall y\, Ryx$

9.2.3 Invalid inferences without finite counter-examples

With these new extended 'existential' rules we can always find finite counter-examples, but this does not mean that every invalid inference can be recognized as such by the extended tableau system. In predicate logic we can make up invalid inferences with only infinite counter-models. Here is an example with two premises:

$$\forall x \exists y\, Rxy, \forall x \forall y \forall z\, ((Rxy \wedge Ryz) \rightarrow Rxz) \not\models \exists x \exists y\, (Rxy \wedge Ryx) \tag{9.31}$$

Take again the 'love'-interpretation for the relation R then the inference can be rephrased as follows:

Everybody loves somebody

Everybody loves all persons who are loved by his loved ones. (9.32)

There is at least a pair of persons who love each other.

We would expect that the seemingly cautious conclusion would follow from the happy hippie optimism conveyed by the two premises. And in fact it holds as long as we would stick to situations with only a finite number of persons.

Exercise 9.13 Show that for finite models which satisfy the two premises as given (9.31) will always contain a symmetric pair: $\exists x \exists y\, (Rxy \wedge Ryx)$. A finite happy hippie community always contains a happily loving couple!

In situations with an infinite number of objects we can interpret R in such a way that the two premises are true and the conclusion is false. For example, take the integers instead of people with R interpreted as the relation $<$. The inference then can be recaptured as follows:

Every integer is smaller than some integer

If one integer is smaller than a second one, then all the integers which are larger than the second are also larger than the first.

There is at least one pair of integers who are smaller than each other.

Here the premises are clearly true, and the conclusion is false, which proves that the inference as given in (9.31) is indeed invalid. Such infinite counter-models can never be constructed by our tableaus, and since the inference of (9.31) has only infinite counter-examples its invalidity can never be demonstrated by the system, not even with the help of the extended 'existential' rules.

9.2.4 Tableaus versus natural reasoning

Although the tableau systems which we have discussed so far are pretty good computational methods for testing the validity of inferences, there is also a clear disadvantage. Each of the individual steps in a tableau can be understood quite easily, but a tableau as a full line of argumentation is not very transparent, and it seems that it does not reflect the way ordinary humans reason.

One part of this type of objection against the tableau method is superficial because there are many small steps in a tableau that are never made explicitly in a 'normal' argumentation because they are too trivial to be mentioned. Due to the fact that the tableau method is a pure symbolic method all the small steps have to be taken into account, and therefore these steps may look quite artificial.

But there is more to it. In a tableau we reason in a negative way, which tends to be unnatural. Validity is demonstrated by the exclusion of counter-examples, whereas humans tend to prove the validity of an inference directly, from the given facts to what has to be proven. If this does not work, or if uncertainty about the validity of an inference arises, one tries to use his imagination to think of a counter-example to refute the validity. Proof and refutation are most often considered to be two different sorts of mental activity. The tableau method incorporates the two in one computational system by argumentation in an indirect way.

In one of the next chapters we will discuss another proof system which can only be used to demonstrate validity and which makes use of rules which are close to the way humans reason. Therefore, these system are referred to as 'natural deduction'. The tableau systems are considered as 'unnatural deduction'. They are very useful for 'black box' automated reasoning systems, where the users are only interested in a final answer about the validity of an inference (and maybe also a specification of a counter-example) but not *how* it has been computed.

This is quite an exaggerated qualification. It is true that tableau systems may behave in an unnatural way. We have seen an example of the first tableau system for predicate logic where the system tried to construct a complicated infinite counter-example for an invalid inference for which everybody can make a very simple counter-example. We have also shown how the rules can be 'tamed' to perform in a more 'reasonable' way. The development of more natural and more 'intelligent' tableau systems is an important issue of contemporary research of applied computational logic.

Moreover, systems which may perform strangely in certain circumstances may behave remarkably intelligent in others. Here is an example:

$$\exists x \forall y \, (Rxy \leftrightarrow \neg \exists z \, (Ryz \wedge Rzy)) \tag{9.33}$$

This is a predicate logical formulation of what is known as the Quine paradox (after the American philosopher and logician Willard Van Orman Quine). This formula turns out to be inconsistent. This is not directly obvious. It really takes some argumentation.

Exercise 9.14 The formula $\exists x \forall y \, (Ryx \leftrightarrow \neg Ryy)$ is a predicate logical version of the famous Russell paradox (take R to be the relation 'element of' to get the exact version). Show with a tableau that this formula can never be true (is inconsistent).

The subformula $\neg \exists z \, (Ryz \wedge Rzy)$ of (9.33) means that there is no object which is mutually related to y. In a domain of persons and by interpreting Rxy as that 'x knows y' this means that y has no *acquaintances*, that is, persons known by y who also know y. Let us say that such a person without acquaintances is called a 'loner.' The full formula as given in (9.33) says that there exists some person who knows all the loners and nobody else. Let us call this person the 'loner-knower'. According to the following infallible argumentation this loner-knower cannot exist.

- If the loner-knower knows himself, then he has an acquaintance, and therefore he is not a loner himself. But then he knows a person who is not a loner, which contradicts the fact that he only knows loners.
- If he does not know himself, then he is not a loner, otherwise he would know himself. But if he is not a loner then he must have an acquaintance. This acquaintance is a loner neither, since he knows the loner-knower and the loner-knower knows him. So, the loner-knower knows a non-loner, which also contradicts the fact that the loner-knower only knows loners.
- The inevitable conclusion is that the loner-knower does not exist, because for every person it must be the case that he knows himself or not. For the loner-knower both options lead to a contradiction.

In Figure (9.4) on page 9-22 the inconsistency of (9.33) has been demonstrated by means of a closed tableau. This closed tableau, starting with a sequent with the formula on the left hand side, shows that the Quine paradox can never be true. If we rephrase the information in the closing tableau of Figure 9.4 we get in fact quite the same line of argumentation as have been outlined here above. In Figure 9.5 on page 9-23 the tableau has been translated back to the interpretation in natural language as has been described here.

9.3 Tableaus for epistemic logic

The tableau method is used extensively for classical logics such as propositional and predicate logic. It does not have been applied very much in the field of modal logic, such as the epistemic and dynamic logics that have been introduced in the first part. In modal logic ref the tradition tends much more towards axiomatic systems and model-checking. Part of

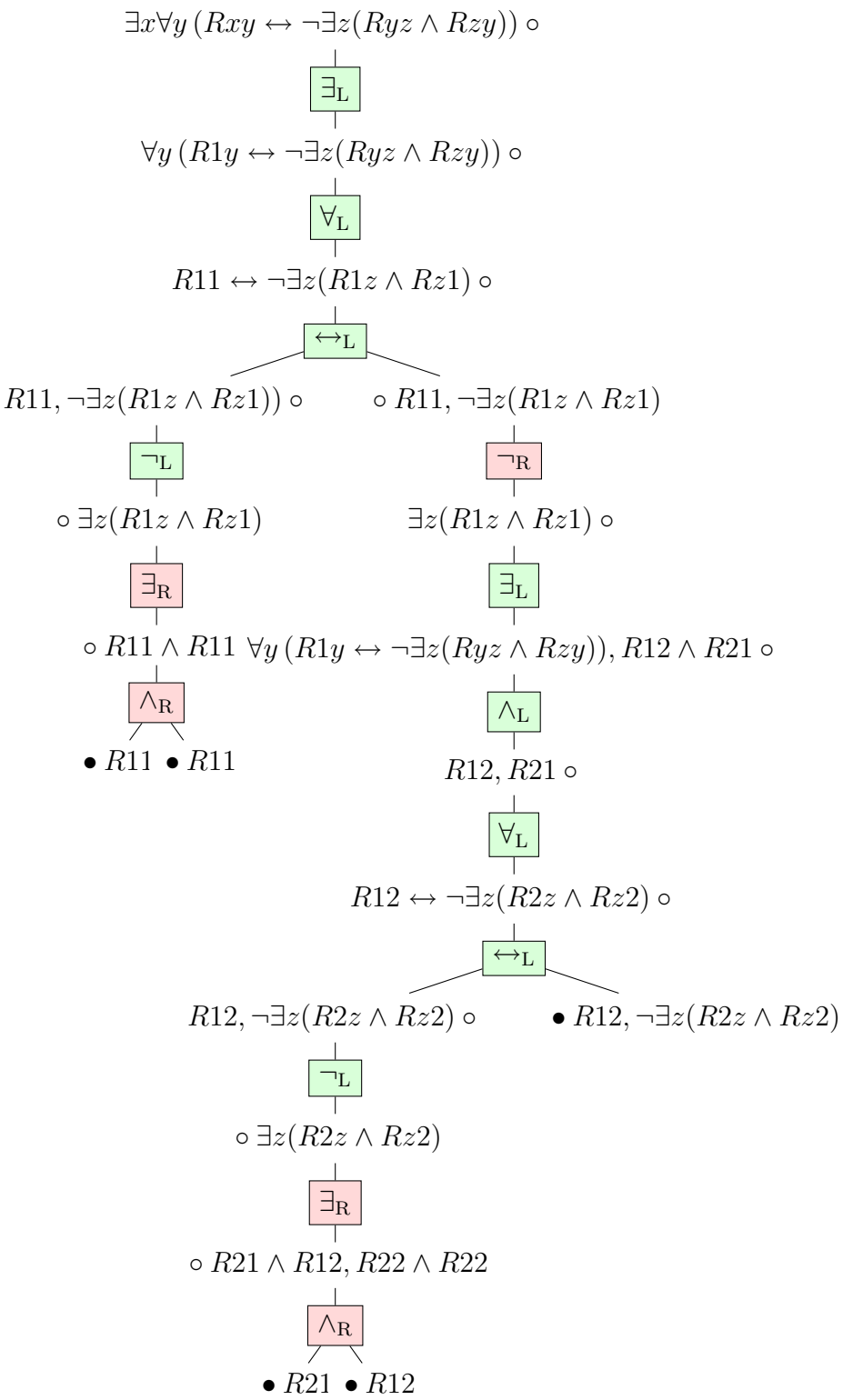


Figure 9.4: A tableau which proves that the Quine-paradox is not satisfiable.

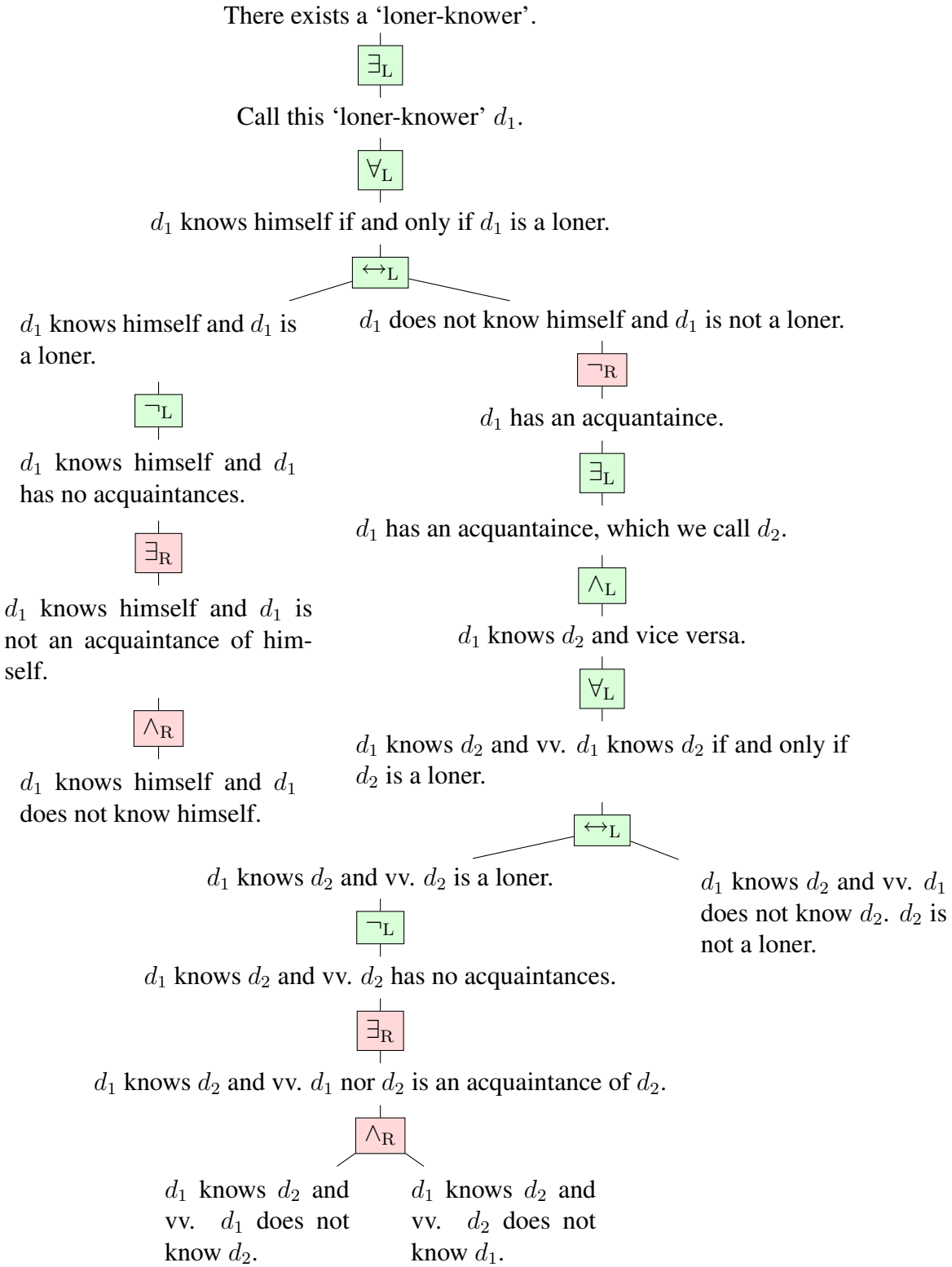


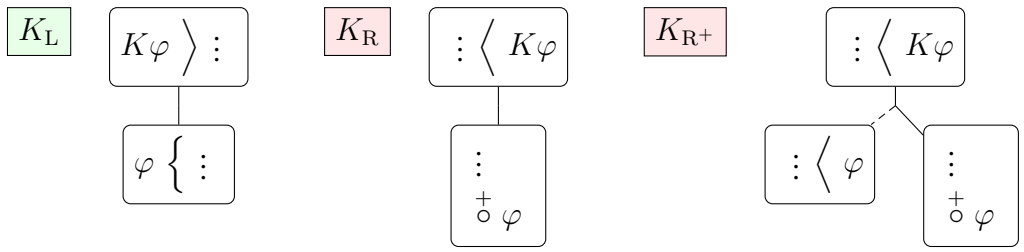
Figure 9.5: A tableau which proves that the Quine-paradox is not satisfiable.

the underlying reasons are purely cultural, but there are also important technical reasons. There are many different modal logics using a wide variety of different kind of modal operators. On top of that, these logics also make use of different classes of possible world models. Technically, it is just easier to capture these differences by means of axioms. It is just a matter of replacing some axioms by others in order to skip from one modal logic to the other. Directed search methods, such as the tableau method, are much harder to modify appropriately.

We will avoid technical details here. In order to illustrate a tableau method for a modal system we take the simplest one of which rules look very much like those of the last system we have presented for predicate logical reasoning. The system we will discuss here is a validity test for inferences in epistemic propositional logic with only one agent, that is, propositional logic with a single K operator.

Remember that $K\varphi$ stands for the proposition which says the agent knows that φ is the case, and in terms of possible world semantics, it meant that φ is true in all the agent's epistemic alternatives. This means that we have to keep track of more worlds in one node in a tableau, since a counter-model may exist of multiple worlds. In tableau terminology these are called *multi-sequents* which are represented by boxes which may contain more than one sequent.

Below the rules have been given in a brief schematic way. The vertical lines of dots represent one or more sequents, and $\varphi >$ means that the formula φ appears on the left hand side of at least one of the sequents in the box, and $\varphi \{$ means it appears in all of them. The symbols $< \varphi$ and $\{ \varphi$ are used to describe analogous situations for formulas on the right side.



(9.34)

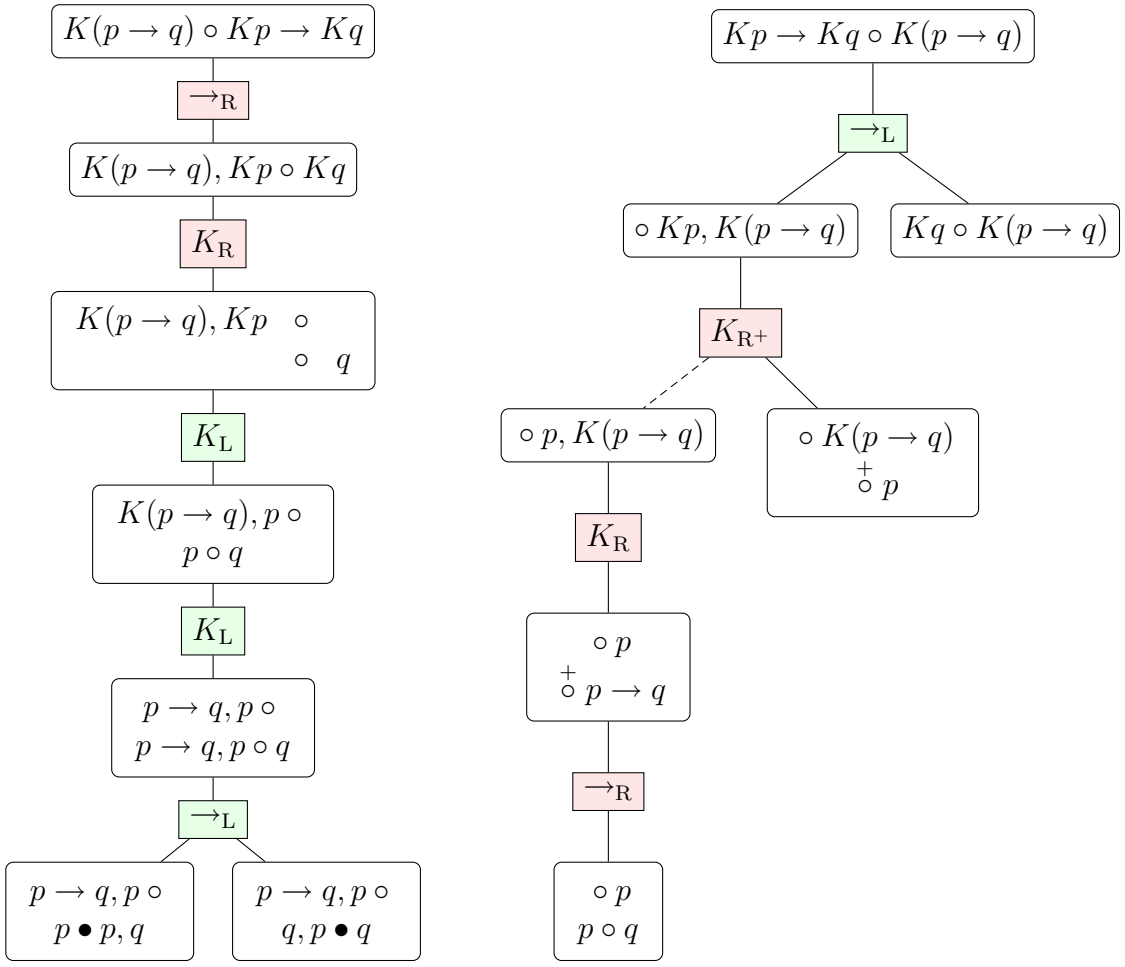
K_{L}: If a formula $K\varphi$ appears on the left part of at least one of the sequents in the box then remove this formula from those sequents and add φ to all the sequents in the box.

K_{R}: If a formula $K\varphi$ appears in the right part of at least one of the sequents then remove them and add the sequent $\circ \varphi$ to the box.

K_{R^+}: If a formula $K\varphi$ appears on the right part of at least one of the sequents then add the sequent $\circ \varphi$ to the box, *and* add a try-out branch with the original sequents of which one is extended with φ on the right part of it.

The symbol $\overset{+}{\circ}$ means that a new sequent (world) has been added to the box. This also implies that all formulas $K\varphi$ which were removed in preceding steps becomes active again. They are to be placed in left part of the first sequent of the sequent box.

Below two relatively simple examples are given. The first demonstrates that $K(p \to q) \models Kp \to Kq$ by means of a closed tableau. As you can see, the two end nodes consist of sequent boxes of which each contains a contradictory sequent. The second tableau shows that the converse of this inference is invalid: $Kp \to Kq \not\models K(p \to q)$.



(9.35)

Before we may jump to conclusions we need to be precise about closed and open multi-sequents. A multi-sequent is *closed* if it contains an impossible sequent containing contradictory information, i.e., a formula which appears on the left and on the right part of the sequent. All the worlds as described in a multi-sequent need to be possible to provide a counter-example. A tableau is closed if it contains only branches with closed multi-sequents in the terminal nodes. A multi-sequent is *open* if it is not closed and all of its sequents are free of logical symbols. A tableau is open if it contains at least one open multi-sequent. As for propositional and predicate logic, an open tableau detects invalidity and the open multi-sequent is nothing less than the description of a counter-model. The

first sequent of the open node is then the world at which rejection of the inference takes place: all the premises are true there, and the conclusion will be false. A closed tableau tells us that the top sequent represents a valid inference.

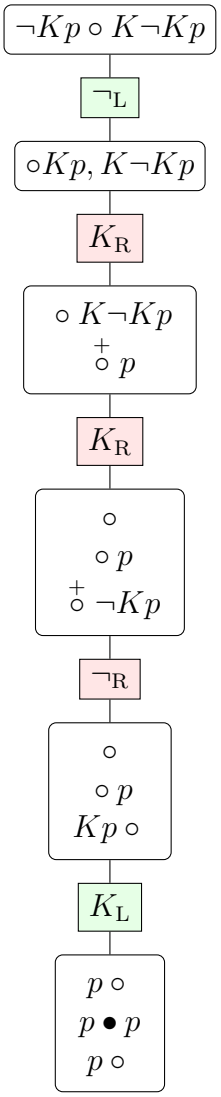
The first tableau in (9.35) showed a direct consequence of the logical closure property of the epistemic operator K, which holds for all 'necessity' operators in modal logics such as the dynamic operator $[\pi]$. The second tableau in (9.35) shows the invalidity of the converse by means of the construction of a two worlds counter-model of which one falsifies p and the other verifies p and falsifies q. We have used the try-out version of K_{R} in the second step in order to find the smallest counter-model. The third step is a regular K_{R}-step because an additional try-out branch would close immediately ($p \bullet p, q$).

If we would have used K_{R} twice we would have end up with a three worlds counter-model. In other more complicated cases of invalid inference the try-out version of the K_{R} is really needed to find finite counter-models, just as we have seen for certain predicate logical inferences.

Exercise 9.15 Show with two tableaus that $Kp \vee Kq \models K(p \vee q)$ and $K(p \vee q) \not\models Kp \vee Kq$.

The following tableau shows a principle which holds specifically for epistemic logic:

negative introspection.



(9.36)

The tableau ends with a description of a single 'impossible' possible worlds model. In fact it tells us that a counter-model requires at least one impossible world at which p is both true and false, and therefore, a counter-model for negative introspection does not exist.

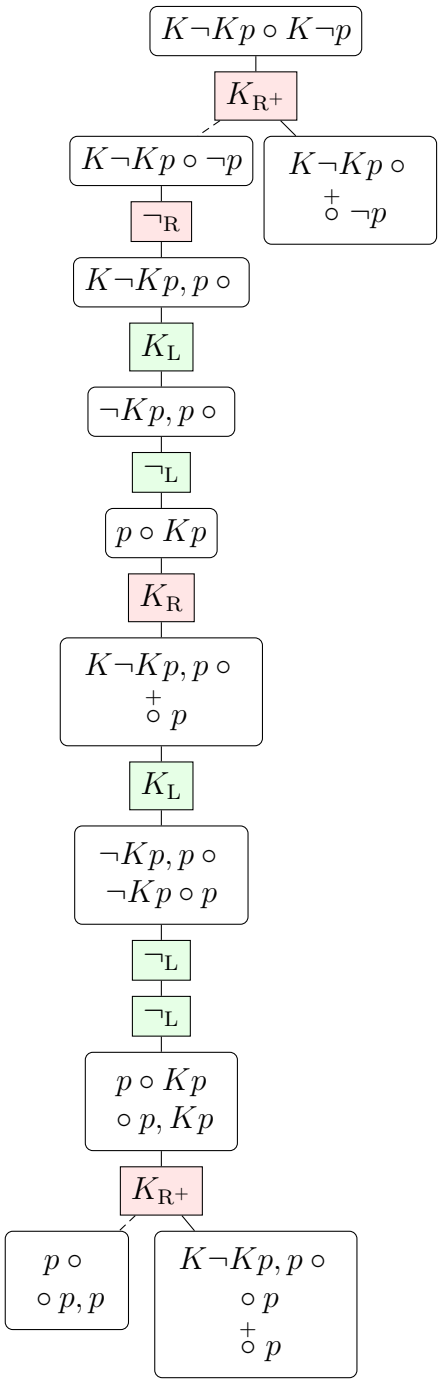
Exercise 9.16 Show with two tableaus that $Kp \models p$ and $p \not\models Kp$.

Exercise 9.17 Show with a closed tableau that $Kp \models KKp$ (positive introspection).

Exercise 9.18 Show with a closed tableau that $K(Kp \vee q) \models Kp \vee Kq$.

As a last example we demonstrate one other tableau where the try-out version of K_{R} are required to get a counter-model to compute an invalidity: $K\neg Kp \not\models K\neg p$. It says that if the agent knows that he does not know that p does not imply that he must know that $\neg p$

is the case. The tableau to find the smallest counter-model requires two applications of K_{R^+}.



(9.37)

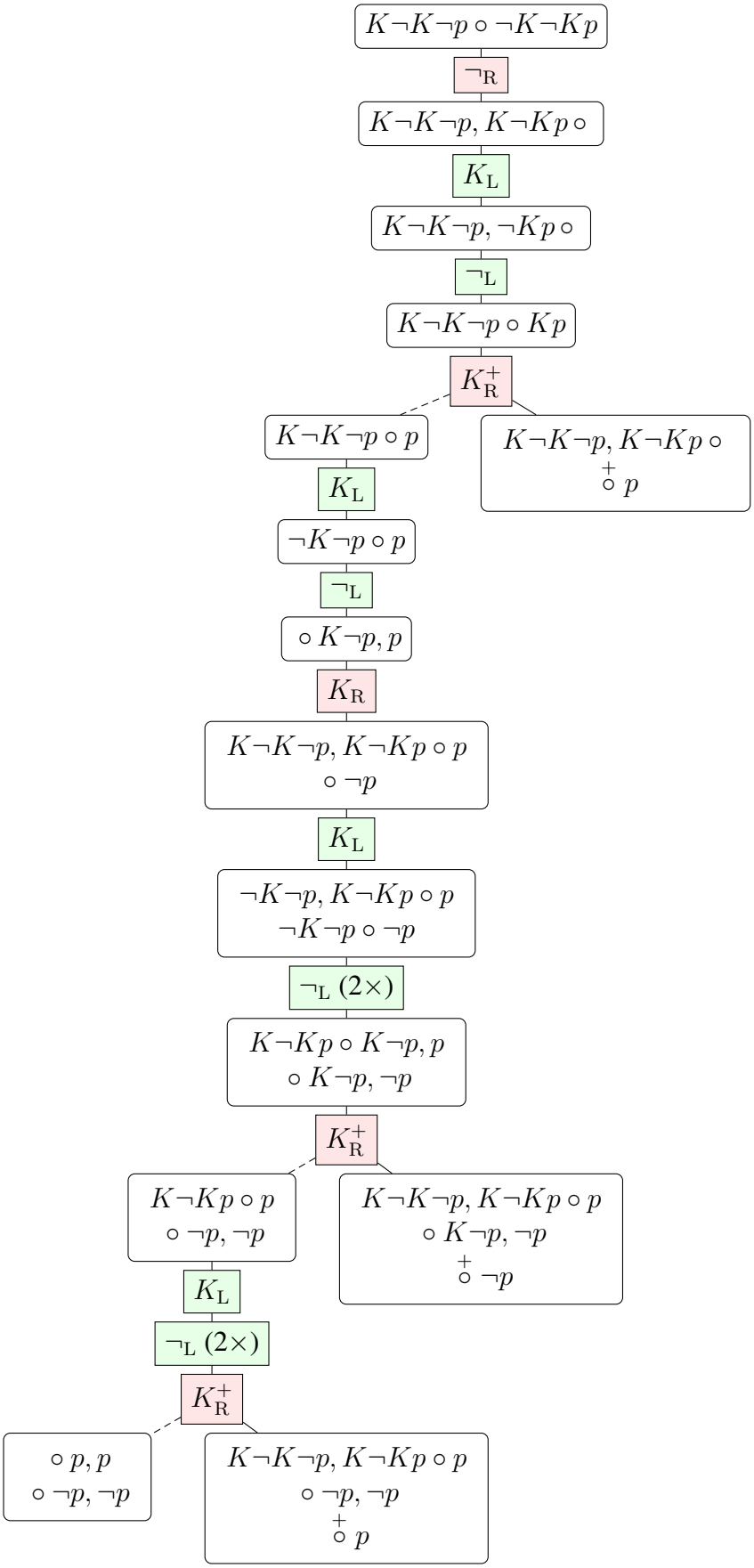
The counter-model which has been found in the left-most branch contains two worlds, one which verifies p and one which falsifies p. In both worlds the agent does not know that p and so $K\neg Kp$ is true in the first world (and also in the second), but $K\neg p$ is false in this world because p is false in the second.

Exercise 9.19 Show with a tableau that $K\neg K\neg p \not\models \neg K\neg Kp$.

Exercise 9.20 Show with a tableau that $\neg K\neg Kp \models K\neg K\neg p$.

In many modal logics such as the epistemic and dynamic logic of the first part of this book the so-called *finite model property* holds. This means that there exist no inferences (with a finite set of premises) with only infinite counter-models such as we have seen for predicate logic in the example (9.31) on page 9-19. This also means that we can always detect invalidity for invalid inferences in single agent epistemic logic by using the tableau method with the given rules for the knowledge operator.

Single agent epistemic logic is by far the easiest modal logic when it comes down to defining a complete tableau system. For other modal logics this is much harder, but not impossible. Instead of multi-sequents so-called *hyper-sequents* are needed to search and specify counter-models by means of reduction rules. A hyper-sequent may not only contain multiple sequents but also other hyper-sequents. Using the format we have been using for single agent epistemic logic here this would look like nested boxes which can be used to capture the accessibility relation of potential counter-models. For multi-modal logics such as multi-agent epistemic logic and dynamic logic we need in addition labeling mechanisms for the nested boxes as to keep track of multiple accessibility relations. On top of that we also need quite complicated rules for 'path'-operators such as the common knowledge operator in multi-agent epistemic logic or the iteration operator in dynamic logic. All these technical complications are the main reason that tableau methods for advanced modal logics have not been standardized yet. Construction of models, whether they are realized by means of extended tableau techniques or alternative methods, are in the field of applied modal logic a very important theme of ongoing research. For sake of presentation and clarity, we do not want to drag along our readers into the highly technical mathematics of it.


$K\neg K\neg p \circ \neg K\neg Kp$
$\neg_{\mathrm{R}}$
$K\neg K\neg p, K\neg Kp \circ$
K_{L}
$K\neg K\neg p, \neg Kp \circ$
$\neg_{\mathrm{L}}$
$K\neg K\neg p \circ Kp$
K^{+}_{R}
$K\neg K\neg p \circ p$
$K\neg K\neg p, K\neg Kp \circ \overset{+}{\circ} p$
K_{L}
$\neg K\neg p \circ p$
$\neg_{\mathrm{L}}$
$\circ K\neg p, p$
K_{R}
$K\neg K\neg p, K\neg Kp \circ p \circ \neg p$
K_{L}
$\neg K\neg p, K\neg Kp \circ p \neg K\neg p \circ \neg p$
$\neg_{\mathrm{L}}\ (2\times)$
$K\neg Kp \circ K\neg p, p \circ K\neg p, \neg p$
K^{+}_{R}
$K\neg Kp \circ p \circ \neg p, \neg p$
$K\neg K\neg p, K\neg Kp \circ p \circ K\neg p, \neg p \overset{+}{\circ} \neg p$
K_{L}
$\neg_{\mathrm{L}}\ (2\times)$
K^{+}_{R}
$\circ p, p \circ \neg p, \neg p$
$K\neg K\neg p, K\neg Kp \circ p \circ \neg p, \neg p \overset{+}{\circ} p$

(9.38)

Chapter 10

Proofs

In the first part of this book we have discussed complete axiomatic systems for propositional and predicate logic. In the previous chapter we have introduced the tableau systems of Beth, which was a method to test validity. This method is much more convenient to work with since it tells you exactly what to do when a given formula has to be dealt with during such a validity test. Despite the convenience of Beth's system it does not represent the way humans argue.

In the late 1930s the German mathematician Gerhard Gentzen developed a system which het called *natural deduction*, in which the deduction steps as made in mathematical proofs are formalized. This system is based not so much on axioms but on rules instead. For each logical symbol, connectives and quantifiers, rules are given just in the way they are dealt with in mathematical proofs.

Gerhard Gentzen

Dag Prawitz

In this chapter we will demonstrate how this system works. The precise definition of these rules goes back to the Swedish logician Dag Prawitz, who gave a very elegant reformulation of Gentzen's work in the 1960s.[1]

[1]The format of the proofs in this chapter has been introduced by the American logician John Fitch.

10.1 Natural deduction for propositional logic

In chapter 2 we have introduced an axiomatic system for propositional logic. By means of the rule of modus ponens one may jump from theorems to new theorems. In addition we had three axioms, theorems that do not have to be proven, and which may be used as starting points. Since these axioms are tautologies, and the rule of modus ponens is a sound rule of inference, a proof is then just a list of tautologies, propositions that are true under all circumstances.

Although this system is fun to work with for enthusiast readers who like combinatoric puzzles, it is surely not the way people argue. You may remember that it took us even five steps to prove that an extremely simple tautology as $\varphi \rightarrow \varphi$ is valid. This may even be worse for other trivial cases. It takes almost a full page to prove that $\varphi \rightarrow \neg\neg\varphi$ is valid (a real challenge for the fanatic puzzler)!

The pragmatic problem of a purely axiomatic system is that it does not facilitate a transparent manner of *conditional reasoning*, which makes it deviate very much from human argumentation. In an ordinary setting people derive conclusions which hold under *certain* circumstances, rather than summing up information which always hold. Especially when conditional propositions, such as the implicative formulas as mentioned here above, have to be proven the conditionions are used as presuppositions. Let us illustrate this with a simple mathematical example.

> If a square of a positive integer doubles the square of another positive integer then these two integers must both be even.
>
> Suppose m, n are two positive integers such that $m^2 = 2n^2$. This means m must be even, because if m^2 is even then m must be even as well. So, $m = 2k$ for some positive integer k. Since $m^2 = 2n^2$ we get $2n^2 = (2k)^2 = 4k^2$, and therefore, $n^2 = 2k^2$ which means that n must be even as well.

In the proof we presuppose that the antecedent ($m^2 = 2n^2$) of the conditional proposition ($m^2 = 2n^2 \rightarrow m, n$ even) that is to be proven holds. This is what is called an *hypothesis*. In the proof we derived that the consequent (m, n even) of the proposition holds under the circumstances that the hypothesis holds. The validity of this type of conditional reasoning reflects an important formal property of propositional logic (and also of the other logics which has been introduced in the first part of this book), which is called the *deduction property*. Formally it looks as follows: For every set of formulas Σ and for every pair of formulas φ and ψ:

$$\Sigma, \varphi \models \psi \text{ if and only if } \Sigma \models \varphi \rightarrow \psi \tag{10.1}$$

It says that by means of the implication we can reason about valid inference within the propositional language explicitly. A conditional proposition $\varphi \rightarrow \psi$ is a valid inference within a context Σ if and only if ψ is a valid conclusion from Σ extended with φ as an

Prawitz used tree like structures, whereas here, in analogy of Fitch's presentation proofs are divided into so-called subproofs.

additional assumption (hypothesis). The deduction property reveals the operational nature of implication: φ leads to the conclusion ψ.

Exercise 10.1 Show that this deduction property holds for propositional logic by making use of truth tables.

Exercise 10.2 The modus ponens rule and the deduction property are characteristic for the implication in propositional logic. Let © be some propositional connective which has the modus ponens and deduction property:

$$\varphi, \varphi \copyright \psi \models \psi \qquad \varphi \models \psi \text{ if and only if } \models \varphi \copyright \psi$$

Show that © must be the implication $\rightarrow$.

Integration of the deduction property in a deduction system requires accomodation of hypotheses, i.e., additional assumptions that a reasoner uses in certain parts of his line of argumentation or proof. A proof of $\varphi \rightarrow \varphi$ then becomes trivial. Since assuming φ leads to φ we may conclude that $\varphi \rightarrow \varphi$ is always true. We may write this as follows:

$$\begin{array}{l} \left[\dfrac{\varphi}{\varphi \quad \text{repeat}}\right] \\ \varphi \rightarrow \varphi \quad \text{Ded} \end{array} \tag{10.2}$$

The first part between square brackets we call a *subproof* of the full proof. A subproof starts with an hypothesis (underlined) which is assumed to hold within this subproof. Proving a conditional proposition $\varphi \rightarrow \psi$ requires a subproof with hypothesis φ and conclusion ψ within this subproof. For our simple example (10.2) this immediately leads to success, but it may involve much more work for longer formulas. Consider the following example where we want to prove the second axiom of the axiomatic system as given in chapter 2: $(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$.

$$\begin{array}{l} \left[\begin{array}{l} \underline{\varphi \rightarrow (\psi \rightarrow \chi)} \\ \vdots \\ (\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi) \end{array}\right] \\ (\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)) \quad \text{Ded} \end{array} \tag{10.3}$$

We have first set up a preliminary format of our proof. The conditional proposition that we want to prove has been rewritten as a subproof, which we have to establish later on. We need to show that the antecedent of the proposition indeed leads to the consequent. Since the desired conclusion of the subproof is an implication again we may follow the

same procedure and extend our first format in the following way:

$$\left[\begin{array}{l} \varphi \to (\psi \to \chi) \\ \hline \left[\begin{array}{l} \varphi \to \psi \\ \hline \vdots \\ \varphi \to \chi \end{array}\right] \\ (\varphi \to \psi) \to (\varphi \to \chi) \quad \text{Ded} \end{array}\right] \tag{10.4}$$
$$(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi)) \quad \text{Ded}$$

Here we have a subproof within a subproof, in which we need to show that the additional assumption $\varphi \to \psi$ leads to a conclusion $\varphi \to \chi$. This second hypothesis has been added to the hypothesis of the first subproof. In order to obtain the desired conclusion we may therefore use both hypotheses.

Again, the conclusion is a conditional proposition, and so, for the third time, we squeeze in a new subproof.

$$\left[\begin{array}{l} \varphi \to (\psi \to \chi) \\ \hline \left[\begin{array}{l} \varphi \to \psi \\ \hline \left[\begin{array}{l} \varphi \\ \hline \vdots \\ \chi \end{array}\right] \\ \varphi \to \chi \quad \text{Ded} \end{array}\right] \\ (\varphi \to \psi) \to (\varphi \to \chi) \quad \text{Ded} \end{array}\right] \tag{10.5}$$
$$(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi)) \quad \text{Ded}$$

Given this reformulation, we need to prove that χ holds given three hypotheseses: $\varphi \to (\psi \to \chi)$, $\varphi \to \psi$ and φ. This is not very hard to prove by making use of our earlier rule of modus ponens. From the second and the third ψ follows and from the first and the third $\psi \to \chi$. These new propositional formulas can then be combined to establish χ. Here is

our final result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\varphi}{\dfrac{\dfrac{\varphi}{\dfrac{\psi \quad \text{MP}}{\dfrac{\psi \to \chi \quad \text{MP}}{\chi \quad \text{MP}}}}}{\varphi \to \chi \quad \text{Ded}}}}{\varphi \to \psi}}{(\varphi \to \psi) \to (\varphi \to \chi) \quad \text{Ded}}}{(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi)) \quad \text{Ded}} \tag{10.6}$$

This result means that we no longer have to use the second axiom of the axiomatic system as described in chapter 2. It can derived by means of our new deduction rule.

The first axiom of the system, $\varphi \to (\psi \to \varphi)$, can be established also quite straightforwardly by means of the deduction rule. In order to prove $\varphi \to (\psi \to \varphi)$ we need to show that $\psi \to \varphi$ can be proved from φ. This can be shown then by simply concluding that φ follows from φ and ψ:

$$\dfrac{\dfrac{\dfrac{\varphi}{\dfrac{\dfrac{\psi}{\varphi \quad \text{rep.}}}{\psi \to \varphi \quad \text{Ded}}}}{\varphi \to (\psi \to \varphi) \quad \text{Ded}}} \tag{10.7}$$

Exercise 10.3 Prove $(\varphi \to (\psi \to \chi)) \to (\psi \to (\varphi \to \chi))$.

10.1.1 Proof by refutation

It seems that we can replace the axiomatic system by a natural deduction system by simply replacing the axioms by a single rule, the deduction rule. This is not the case, however. The third axiom of the axiomatic system $(\neg\varphi \to \neg\psi) \to (\psi \to \varphi)$, also called *contraposition*, can not be derived by deduction and modus ponens only. We need something to deal with the negations in this formula.

There seems to be a way out by taking $\neg\varphi$ to be an abbreviation of the conditional formula $\varphi \to \bot$. This establishes a procedure to prove negative information by means of the deduction rule. Proving $\neg\varphi$ requires a proof that the assumption that φ holds leads to a contradiction ($\bot$). This is indeed a natural way to establish negative information, as shown in the following example

$\sqrt{2}$ is not a rational number.

Suppose $\sqrt{2}$ were a rational number. This means there are two positive integers m and n such that $(m/n)^2 = 2$ and, in addition, that m or n is odd, since we can simply take the smallest pair such that $(m/n)^2 = 2$ (they cannot both be even since then it would not be the smallest pair for which this equation holds). But then $m^2 = 2n^2$ and therefore m and n must be even, as we have shown in an earlier example (page 10-2). Clearly, we have derived a contradiction, and therefore $\sqrt{2}$ must be an irrational number.

A reformulation in natural deduction style looks as follows:

$$\dfrac{\left[\begin{array}{l}\sqrt{2} \in Q \\ \hline (m/n)^2 = 2 \text{ for certain pair of positive integers} \\ m, n \text{ with } m \text{ or } n \text{ being odd.} \\ m = 2n^2 \\ m \text{ and } n \text{ are both even positive integers} \\ \bot\end{array}\right]}{\neg(\sqrt{2} \in Q)} \tag{10.8}$$

This way of proving negative statements suffices to derive certain propositional logical theorems containing negative information. For example, the converse of the contraposition axiom can be established in this way;

$$\dfrac{\left[\begin{array}{l}\varphi \to \psi \\ \hline \left[\begin{array}{l}\psi \to \bot \\ \hline \left[\begin{array}{l}\varphi \\ \hline \psi \quad \text{MP} \\ \bot \quad \text{MP}\end{array}\right] \\ \varphi \to \bot \quad \text{Ded}\end{array}\right] \\ (\psi \to \bot) \to (\varphi \to \bot) \quad \text{Ded}\end{array}\right]}{(\varphi \to \psi) \to ((\psi \to \bot) \to (\varphi \to \bot)) \quad \text{Ded}} \tag{10.9}$$

Replacing $\to \bot$ by negations then settles $(\varphi \to \psi) \to (\neg\psi \to \neg\varphi)$. Unfortunately, this simple solution does not work for the axiom of contraposition. To get a complete system we need an additional rule.

Exercise 10.4 Show, by trying out the procedure which we have used for the previous examples, that you can not derive the axiom of contraposition by modus ponens and the deduction rule only.

This supplementary rule that we will need is in fact quite close to the deduction rule for negations. To derive a formula $\neg\varphi$ we prove that φ leads to a contradiction, which in fact says that φ can not be true. Our new rule says that φ can be proven by showing that φ can not be *false*. In terms of subproofs, if the hypothesis $\neg\varphi$ leads to a contradiction we may conclude that φ is the case. In this way we can prove the contraposition indeed.

$$
\begin{array}{|l}
\hline
1.\quad \neg\varphi \to \neg\psi \\
\hline
\quad \begin{array}{|ll}
\quad 2.\quad \psi & \\
\quad \begin{array}{|ll}
\hline
\quad 3.\quad \neg\varphi & \\
\hline
\quad 4.\quad \neg\psi & \text{MP 1,3} \\
\quad 5.\quad \bot & \text{MP 2,5}
\end{array} & \\
\quad 6.\quad \varphi & \text{new rule 3-5}
\end{array} \\
7.\quad \psi \to \varphi \qquad\qquad \text{Ded 2-6} \\
\hline
\end{array}
\qquad (10.10)
$$

$$
8.\quad (\neg\varphi \to \neg\psi) \to (\psi \to \varphi) \qquad \text{Ded 1-7}
$$

In step 6 we derived φ from the subproof $[\neg\varphi \mid \ldots \bot]$. The hypothesis that φ is false has led to a contradiction. The contradiction in 5 is obtained by a modus ponens, since $\neg\psi$ is an abbreviation of $\psi \to \bot$ here.

Exercise 10.5 Prove $((\varphi \to \psi) \to \varphi) \to \varphi$. Despite the absence of negations in this formula, you will need the new rule.

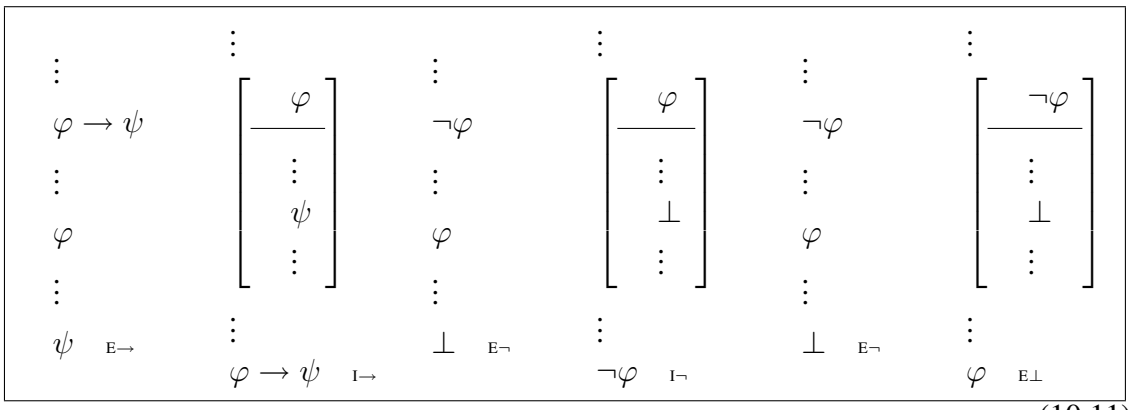
Exercise 10.6 Prove $\varphi \to \neg\neg\varphi$ and $\neg\neg\varphi \to \varphi$. Which of those proofs makes use of the new rule?

In logic this new rule is also called *proof by refutation*, or more academically, *reductio ad absurdum*. In fact, it captures the same way of reasoning as we have used in the tableau systems of the previous chapter. Proving the validity of an inference by presenting a closed tableau we show that the given formula can never be false, and therefore must be true, under the circumstances that the premises hold.

10.1.2 Introduction and elimination rules

The three rules suffice to obtain a complete system for propositional logic. The tradition in natural deduction is to separate the treatment of negations and implications which leads

to the following five rules.



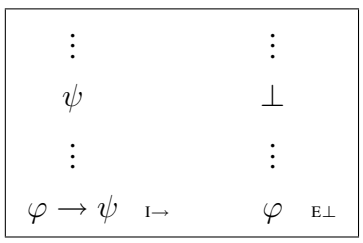
(10.11)

These rules are called *elimination* (E) and *introduction* (I) rules. The modus ponens is called an elimination rule since it says how to remove an implication $\varphi \to \psi$ and replace it by its consequent ψ. The rule then obtains the structural name E$\to$. Elimination of negation, E$\neg$, is then, as a consequence, the derivation of $\bot$ from $\neg\varphi$ and φ. The introduction rule for implication, I$\to$, is the deduction rule because it puts an implication on stage. I$\neg$ is defined analogously. The fifth additional rule represents the rule of proof by refutation and is most often seen as elimination of $\bot$ (E$\bot$).[2]

Two simpeler versions of the deduction rule and the rule of proof by refutation are sometimes added to the system such that repetitions, as for example in the proof of $\varphi \to (\psi \to \varphi)$ as given in (10.7), can be avoided. If a statement φ is true then it also holds under arbitrary conditions: $\psi \to \varphi$. This is in fact a variation of the deduction rule (without hypothesis).

$$\dfrac{\dfrac{\varphi}{\psi \to \varphi}\ \text{I}\to\text{ 'simple'}}{\varphi \to (\psi \to \varphi)}\ \text{I}\to \tag{10.12}$$

For proofs of refutation the anologous simplication is called *ex falso*. Everything may be derived from a contradiction. We will use these simplified versions also in the sequel of this chapter. In general deductive form they look as follows:



(10.13)

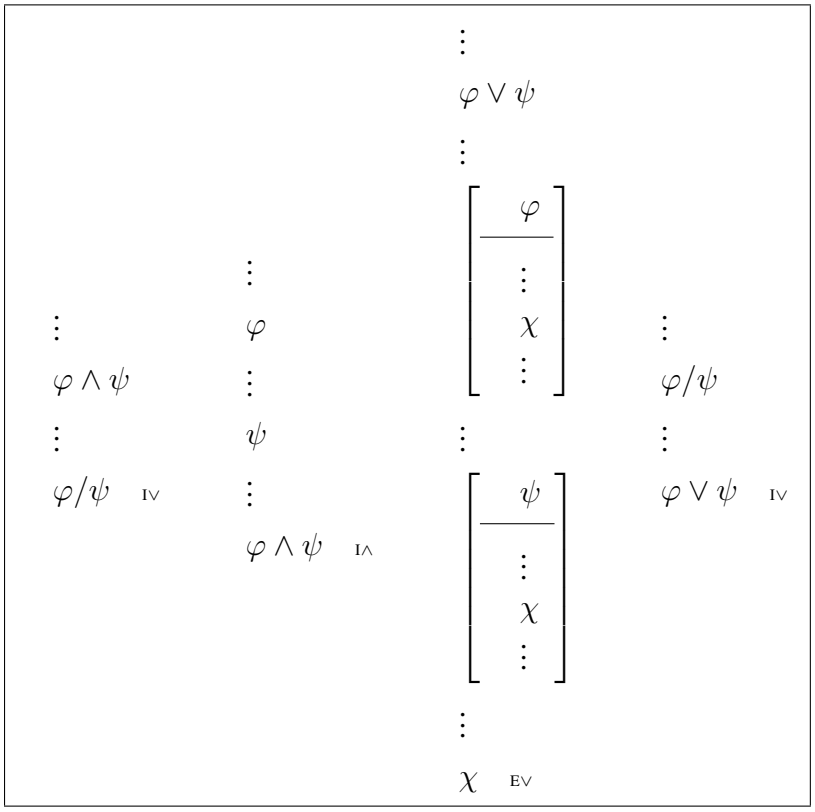
[2]Sometimes I$\neg$ is used for this rule, and then the introduction rule for negation is called falsum introduction (I$\bot$).

10.1.3 Rules for conjunction and disjunction

In propositional logic we also want to have rules for the other connectives. We could try the same procedure as we have done for negation. Find an equivalent formulation in terms of $\rightarrow$ and $\bot$ and then derive rules for these connectives.

$$\varphi \vee \psi \equiv (\varphi \rightarrow \bot) \rightarrow \psi \qquad \varphi \wedge \psi \equiv (\varphi \rightarrow (\psi \rightarrow \bot)) \rightarrow \bot \tag{10.14}$$

This option does not lead to what may be called a system of natural deduction. The equivalent conditional formulas are much too complicated. Instead, we use direct rules for manipulating conjunctive and disjunctive propositions. Below the introduction and elimination rules are given for the two connectives.



(10.15)

The rules for conjunction are quite straightforward. The elimination of a conjunction is carried out by selecting one of its arguments. Since we know that they are both true this is perfectly sound and a natural way of eliminating conjunctions. Introduction of a conjunction is just as easy. Derive a conjunction if both arguments have already been derived.

The introduction of a disjunction is also very simple. If you have derived one of the arguments then you may also derive the disjunction. The rule is perfectly correct but it is not very valuable, since in general, the disjunction contains less information then the information conveyed by one of the arguments.

The elimination of the disjunction is the most complicated rule. It uses two subproofs, one for each of the arguments of the disjunction. If in both subproofs, starting with one

of the disjuncts (φ, ψ) as a hypothesis, the same information can be derived (χ) then we know that this must also hold in a context in which we are uncertain which of the arguments in fact holds $(\varphi \vee \psi)$. Despite the complexity of the rule, its soundness can be seen quite easily. We leave this to the reader in the next exercise.

Exercise 10.7 Show that $\Sigma, \varphi \vee \psi \models \chi$ if and only if $\Sigma, \varphi \models \chi$ and $\Sigma, \psi \models \chi$.

The elimination rule of disjunction reflects a natural way of dealing with uncertainty in argumentation. Here is an example of a mathematical proof.

> There exists two irrational numbers x and y such that x^y is rational.
>
> Let $z = \sqrt{2}^{\sqrt{2}}$. This number must be either irrational *or* rational. Although, we are uncertain about the status of z we can find in both cases two irrational x and y such that x^y must be rational.
>
> Suppose that z is rational, then we may take $x = y = \sqrt{2}$. We have just seen earlier that $\sqrt{2}$ is irrational, so this choice would be satisfactory.
>
> Suppose that z is irrational. Then we may take $x = z$ and $y = \sqrt{2}$, because then $x^y = z^{\sqrt{2}} = \sqrt{2}^{\sqrt{2} \cdot \sqrt{2}} = \sqrt{2}^2 = 2$, and that is a perfect rational number.

In the deduction style we could reformulate our argumentation as follows

$$
\begin{array}{l}
\sqrt{2}^{\sqrt{2}} \in Q \vee \sqrt{2}^{\sqrt{2}} \notin Q \\
\left[\begin{array}{l}
\sqrt{2}^{\sqrt{2}} \in Q \\
\hline
x = y = \sqrt{2} \\
x^y = \sqrt{2}^{\sqrt{2}} \\
x^y \in Q \text{ for certain } x, y \notin Q
\end{array}\right] \\
\left[\begin{array}{l}
\sqrt{2}^{\sqrt{2}} \notin Q \\
\hline
x = \sqrt{2}^{\sqrt{2}}, y = \sqrt{2} \\
x^y = 2 \\
x^y \in Q \text{ for certain } x, y \notin Q
\end{array}\right] \\
x^y \in Q \text{ for certain } x, y \notin Q
\end{array}
\tag{10.16}
$$

In practical reasoning disjunction elimination is also manifest as a way to jump to conclusions when only uncertain information is available. The following realistic scenario gives an illustration of this.

> I am traveling from A to B by train. If I run to the railway station of my home town A then I'll be in time to catch the train to B at 7.45AM, and then in B I

> will take the bus to the office and I will be there in time. If I won't run then I won't catch the 7.45AM train, but in this case I could take the train to B at 8.00AM instead. I would then need to take a cab from the railway station in B to arrive in time at the office. I start running to the railway station, not being sure whether my physical condition this morning will be enough to make me catch the first train (last night I have been to the cinema, and later on we went to the pub, etcetera). But no worries, I'll be in time at the office anyway (okay, it will cost me a bit more money if I won't catch the first train, since taking a cab is more expensive then taking the bus).

Here is the deductive representation of my reasoning:

$$\begin{array}{l}\text{Catch 7.45AM-train} \lor \text{Catch 8.00AM-train}\\ \left[\begin{array}{l}\text{Catch 7.45AM-train.}\\ \hline \text{Take the bus in B to the office.}\\ \text{I'll be in time at the office.}\end{array}\right]\\ \left[\begin{array}{l}\text{Catch 8.00AM-train.}\\ \hline \text{Take a cab in B to the office.}\\ \text{I'll be in time at the office.}\end{array}\right]\\ \text{I'll be in time at the office.}\end{array} \tag{10.17}$$

Exercise 10.8 Can you make up a similar scenario, jumping to safe conclusion while being uncertain about the conditions, from your personal daily experience? Now, reformulate this as a deduction such as given for the previous example.

Here is a very simple example of disjunction elimination in propositional logic. We derive $\psi \lor \varphi$ from the assumption $\varphi \lor \psi$:

$$\begin{array}{ll}1. & \varphi \lor \psi\\ \left[\begin{array}{ll}2. & \varphi\\ \hline 3. & \psi \lor \varphi \quad \text{I}\lor\ 2\end{array}\right]\\ \left[\begin{array}{ll}4. & \psi\\ \hline 5. & \psi \lor \varphi \quad \text{I}\lor\ 4\end{array}\right]\\ 6. & \psi \lor \varphi \quad \text{E}\lor\ 1,2\text{-}3,4\text{-}5\end{array} \tag{10.18}$$

The formula $\psi \lor \varphi$ can be derived from φ and ψ by applying I$\lor$, so we can safely conclude $\psi \lor \varphi$ from $\varphi \lor \psi$ by E$\lor$.

Exercise 10.9 Prove $\varphi \to \psi$ from the assumption $\neg\varphi \lor \psi$.

Exercise 10.10 Prove $\neg(\varphi \land \psi)$ from $\neg\varphi \lor \neg\psi$.

Exercise 10.11 Prove $(\varphi \vee \psi) \wedge (\varphi \vee \chi)$ from $\varphi \vee (\psi \wedge \chi)$.

In general, disjunction elimination applies whenever we need to prove a certain formula χ from a disjunctive assumption $\varphi \vee \psi$. The strength of the elimination rule for disjunction is reflected by the equivalence of the inference $\varphi \vee \psi \models \chi$ on the one hand and the two inferences $\varphi \models \chi$ and $\psi \models \chi$ on the other (as you may have computed for yourself when you have made exercise 10.7 on page 10-10).

Disjunctive conclusions are much harder to establish in a deduction because of the earlier mentioned weakness of the introduction rule for disjunctions. Direct justification of a conclusion $\varphi \vee \psi$ by means of I$\vee$ requires a proof of one of the arguments, φ or ψ, which in many cases is simply impossible. Often a refutational proof is needed to obtain the desired disjunctive conclusion, that is, we show that $\neg(\varphi \vee \psi)$ in addition to the assumptions leads to a contradiction ($\bot$).

A clear illustation can be given by one of the most simple tautologies: $\varphi \vee \neg\varphi$. When it comes to reasoning with truth-values the principle simply says that there are only two opposite truth-values, and therefore it is also called 'principle of the excluded third' or 'tertium non datur'. From an operational or deductive point of view the truth of $\varphi \vee \neg\varphi$ is much harder to see. Since, in general, φ and $\neg\varphi$ are not tautologies, we have to prove that $\neg(\varphi \vee \neg\varphi)$ leads to a contradiction. Below a deduction, following the refutational strategy, has been given:

$$
\left[\begin{array}{lll}
1 & \neg(\varphi \vee \neg\varphi) & \\
\hline
 & \left[\begin{array}{lll} 2 & \varphi & \\ \hline 3 & \varphi \vee \neg\varphi & \text{I}\vee\ 2 \\ 4 & \bot & \text{E}\neg\ 1,3 \end{array}\right] & \\
5 & \neg\varphi & \text{I}\neg\ 2\text{-}4 \\
6 & \varphi \vee \neg\varphi & \text{I}\vee\ 5 \\
7 & \bot & \text{E}\rightarrow\ 1,6
\end{array}\right]
$$
$$
8 \quad \varphi \vee \neg\varphi \qquad \bot\text{E}\ 1\text{-}7 \tag{10.19}
$$

As you can see $\neg\varphi$ is derived from $\neg(\varphi \vee \neg\varphi)$ and this gives us finally the contradiction that we aimed at. In general this is the way to derive a disjunctive conclusion $\varphi \vee \psi$ for which a direct proof does not work. We assume the contrary $\neg(\varphi \vee \psi)$ then derive $\neg\varphi$ or $\neg\psi$ (or both) and show that this leads to a contradiction.

Exercise 10.12 Prove by a deduction that $(\varphi \rightarrow \psi) \vee (\psi \rightarrow \varphi)$ is a tautogy.

Exercise 10.13 Deduce $\neg\varphi \vee \neg\psi$ from $\neg(\varphi \wedge \psi)$

Exercise 10.14 Prove by a deduction that $\neg\varphi \vee \psi$ follows from $\varphi \rightarrow \psi$.

10.2 Natural deduction for predicate logic

The natural deduction system for predicate logic consists of two simple rules and two more complicated, but at the same time more compelling, rules for the quantifiers $\forall$ and $\exists$. The easy weaker rules are $\forall$-elimination and $\exists$-introduction. They are just generalizations of the earlier elimination rule for $\wedge$ and the introduction rule for $\vee$.

From $\forall x\, \varphi$ we may derive that φ holds for 'everything'. This means that we substitute a term for x in φ. Substitution only has a small syntactic limitation. A term may contain variables, and we have to take care that no variable which occurs in such a 'substitute' gets bound by a quantifier in φ after replacing the occurence of x by this term. If this is the case we say that this term is *substitutable* for x in φ. As an illustration that things go wrong when we neglect this limitation take the formula $\forall x \exists y\, \neg(x = y)$. Obviously, this formula is true in every model with more than one object. If we substitutes y for x in $\exists y\, \neg(x = y)$ we get $\exists y\, \neg(y = y)$ which is an inconsistent formula. The term y is not substitutable since y gets bound by the existential quantifier in $\exists y\, \neg(x = y)$.

Introduction of the existential quantifier works in the same way. If you have derived a property φ for certain term t you may replace this term by x and derive $\exists x\, \varphi$ successively. If we write $\varphi[t/x]$ for the result of substitution of t for x in φ and in addition prescribing that t must be substitutable for x in φ, we can formulate the rules mentioned here as follows:

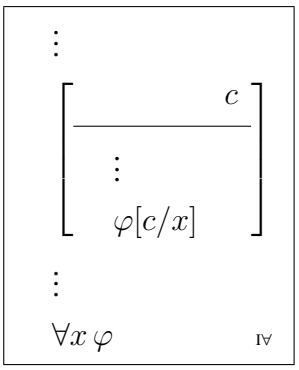
$$\begin{array}{ll} \vdots & \vdots \\ \forall x\, \varphi & \varphi[t/x] \\ \vdots & \vdots \\ \varphi[t/x] \quad \text{E}\forall & \exists x\, \varphi \quad \text{I}\exists \end{array} \tag{10.20}$$

In practice these weak rules are only used to make small completing steps in a proof.

Also in the condition of I$\exists$ it is required that t is substitutable for x in φ. To see that neglecting this additional constraint leads to incorrect result take the formula $\forall y\, y = y$. This is a universally valid formula. It is also the result of replacing x by y in the formula $\forall y\, x = y$, but $\exists x \forall y\, x = y$ is certainly not a valid consequence of $\forall y\, y = y$: $\exists x \forall y\, x = y$ only holds in models containing only a single object.

The introduction rule of the universal quantifier is a bit more complicated rule, but, at the same time, it is a very strong rule. The rule is also referred at as *generalization*. By proving that a property φ holds for an *arbitrary* object we derive that φ holds for *all* objects: $\forall x\, \varphi$. To make sure that the object of which we prove φ-ness is indeed completely arbitrary we use a new name which is not a constant in the language. Starting the subproof we extend the language with this new name only within the range of this subproof. Such an additional constant is also called a *parameter*. It may not be used in the main line of

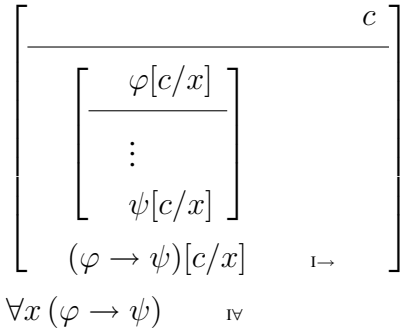
the proof which contains this subproof. Here is the formal version of the rule I$\forall$.



(10.21)

As you can see the subproof does not contain an hypothesis. The only information which is relevant here is that the parameter c does not appear in the line of the argument outside the subproof (represented by the vertical dots outside the subproof box), and that it is not a constant in the base language. To stress this minor syntactical limitation we indicate this c on top of the line where the subproof starts. This makes it clear that this is the reference to the arbitrary object for which we have to prove the desired property φ.

In natural settings of argumentation the generalization rule is most often combined with the deduction rule (I$\rightarrow$). If the assumption that an arbitrary object has the property φ leads to the conclusion that it also must have another property ψ we have proven that 'All φ are ψ' or in predicate logical notation $\forall x\,(\varphi \rightarrow \psi)$. In a formal deduction this looks as follows:



(10.22)

If we are able to prove for an arbitrary man that he must be mortal, we have proven that all men are mortal.

Take, as a mathematical example of this combination, the proof that if the square of a positive integer m doubles the square of another positive integer n, $m^2 = 2n^2$, they must both be even (page 10-2). The generalization rule, applied twice, would then rephrase this as universal result (given that the domain of discourse here contains only positive integers)

$$\forall x \forall y\,(x^2 = 2y^2 \rightarrow x, y \text{ both even})$$

Here is a first example deduction in predicate logic, showing that $\forall x\,(Px \wedge Qx)$ follows

from two assumptions, $\forall x\, Px$ and $\forall x\, Qx$:

$$
\begin{array}{lll}
1. & \forall x\, Px & \text{Ass} \\
2. & \forall x\, Qx & \text{Ass} \\
\left[\begin{array}{lll} 3. & & c \\ \hline 4. & Pc & \text{E}\forall\,1 \\ 5. & Qc & \text{E}\forall\,2 \\ 6. & Pc \wedge Qc & \text{I}\wedge\,4,5 \end{array}\right] & & \\
7. & \forall x\, (Px \wedge Qx) & \text{I}\forall\,3\text{-}6
\end{array}
\tag{10.23}
$$

As you can see the generalization rule dominates the proof. It determines the external structure of the proof, whereas the weaker rule E$\forall$ shows up only within the very inner part of the proof. The generalization rule works pretty much the same way as the deduction rule in propositional logic. In pure predicate logic a proof of a universal statement requires most often the generalization procedure. As we will see in the next section, there are other rules to prove statements with universal strength when we apply predicate logic for reasoning about a specific mathematical structure: the natural numbers.

Just as I$\exists$ is a generalization of I$\vee$, the elimination of existential quantifiers is taken care of by a generalization of disjunction elimination. A formula $\exists x\, \varphi$ represents that there is an object which has the property φ but, in general, we do not know who or what this φ-er is. To jump to a fixed conclusion we introduce an arbitrary φ-er, without caring about who or what this φ-er is, and show that this is enough to derive the conclusion that we are aiming at. The rule looks as follows:

$$
\begin{array}{l}
\vdots \\
\exists x\, \varphi \\
\vdots \\
\left[\begin{array}{ll} \varphi[c/x] & c \\ \hline \vdots & \\ \psi & \end{array}\right] \\
\vdots \\
\psi \qquad\qquad \text{E}\exists
\end{array}
\tag{10.24}
$$

The conclusion ψ can be derived on the basis of the introduction of an arbitrary φ-er (and nothing else). This means that if such a φ-er exists ($\exists x\, \varphi$) then we may safely conclude that ψ must hold. Again, the indication of the parameter c reminds us that it restricted by the same limitations as in the generalization rule I$\forall$.

There is also a close relation with the generalization and the deduction rule. Combination of the latter two facilitated a way to prove statements of the form 'All φ are ψ'.

In fact a slight variation of this is presupposed by means of the subproof in the E∃- rule. Here it in fact says that it has been proven for an arbitrary object that if this object has the property φ then ψ must hold. And then we conclude that, given the assumption that there exists such a φ-er ($\exists x\, \varphi$), we know that ψ must hold.

The following variant of the train scenario as discussed on page 10-10 illustrates elimination of uncertainty conveyed by existential information in practice.

> Again, I am traveling from A to B. I don't know when trains leave, but I know at least there is a train departing from A going to B every half hour. Right now it is 7.35AM, and it will take me only ten minutes to get to the station. This means that I'll catch some train before 8.15AM: or *some point in time* t between 7.45AM and 8.15AM. The train from A to B takes 35 minutes, and my arrival at B will therefore be before 8.50AM (t + 35' $<$ 8.50AM). A cab ride will bring me in less than 15 minutes to the office and so I will be at the office before 9.05AM (t + 35' + 15' $<$ 9.05AM). This means I will be there before 9.15AM, when the first meeting of this morning starts.

Although I am quite uncertain about the time of departure I can safely conclude that I will be in time at the office.

Below a deduction is given which proves that $\forall x \exists y\, Rxy$ follows from $\exists y \forall x\, Rxy$. Each of the quantifier rules is used once:

$$
\begin{array}{lll}
1. & \exists y \forall x\, Rxy & \text{Ass} \\
\left[\begin{array}{lll}
2. & \forall x\, Rxc & c \\
\hline
\left[\begin{array}{lll} 3. & & d \end{array}\right] & & \\
\hline
4. & Rdc & \text{E}\forall\, 2 \\
5. & \exists y\, Rdy & \text{I}\exists\, 4 \\
6. & \forall x \exists y\, Rxy & \text{I}\forall\, 3\text{-}5
\end{array}\right] & & \\
7. & \forall x \exists y\, Rxy & \text{E}\exists\, 2\text{-}6
\end{array}
\tag{10.25}
$$

As an explanation what the individual steps in this proof mean, let us say that Rxy stands for 'x knows y' in some social setting. The assumption says there is some 'famous' person known by everybody. The conclusion that we want to derive means that 'everybody knows someone'. We started with E∃, introducing an arbitrary person known by everybody, and we called him or her c ($\forall x\, Rxc$), and from this we want to derive the same conclusion ($\forall x \exists y\, Rxy$). To get this settled, we introduced an arbitrary object d and proved that d must know somebody ($\exists y\, Rdy$). This is proved by using Rdc (I∃) which follows from $\forall x\, Rxc$ (E∀).

Let us try a more difficult case: $\forall x\, Px \vee \exists x\, Qx$ follows from $\forall x\, (Px \vee Qx)$. The conclusion is a disjunction, and one can easily see that both disjuncts are not valid consequences of the given assumption. This means we have to prove this by using refutation

(E$\bot$). We need to show that the assumption and the negation of the conclusion lead to a contradiction ($\bot$). Here is the complete deduction.

$$
\begin{array}{l}
1.\ \ \forall x\,(Px \vee Qx) \qquad \text{Ass} \\
\hline
\left[\begin{array}{l}
2.\ \ \neg(\forall x\,Px \vee \exists x\,Qx) \\
\hline
\left[\begin{array}{ll}
3. & c \\
\hline
4.\ \ Pc \vee Qc & \text{E}\forall\ 1 \\
\left[\begin{array}{l} 5.\ \ Pc \\ \hline \end{array}\right] & \\
\left[\begin{array}{ll}
6.\ \ Qc & \\
\hline
7.\ \ \exists x\,Qx & \text{I}\exists\ 6 \\
8.\ \ \forall x\,Px \vee \exists x\,Qx & \text{I}\exists\ 7 \\
9.\ \ \bot & \text{E}\neg\ 2,8 \\
10.\ \ Pc & \text{E}\bot\ 9\ \text{(simple version)}
\end{array}\right] & \\
11.\ \ Pc & \text{E}\vee\ 4,5\text{-}5,6\text{-}10
\end{array}\right] \\
12.\ \ \forall x\,Px \qquad \text{I}\forall\ 3\text{-}11 \\
13.\ \ \forall x\,Px \vee \exists x\,Qx \qquad \text{I}\vee\ 12 \\
14.\ \ \bot \qquad \text{E}\neg\ 2,13
\end{array}\right] \\
15.\ \ \forall x\,Px \vee \exists x\,Qx \qquad \text{E}\bot\ 2\text{-}14
\end{array}
\tag{10.26}
$$

In the outermost subproof we have shown that $\forall x\,(Px \vee Qx)$ in combination with $\neg(\forall x\,Px \vee \exists x\,Qx)$ leads to the conclusion $\forall x\,Px$ (12) which gives indeed an immediate contradiction (13,14). $\forall x\,Px$ can be obtained by proving the property P for an arbitrary object (c), which is carried in the second subproof. This can be proved then by using $Pc \vee Qc$ and disjunction elimination. Pc immediately follows from the first disjunct, Pc itself, and it also follows from Qc, since this leads to a contradiction and by applying ex falso, the simple form of proof by refutation, also to Pc.

Exercise 10.15 Prove that $\forall x\,\neg Px$ follows from $\neg\exists x\,Px$.

Exercise 10.16 Prove that $\exists x\,Px \wedge \exists x\,Qx$ follows from $\exists x\,(Px \wedge Qx)$.

Exercise 10.17 Prove that $\exists x\,\neg Px$ follows from $\neg\forall x\,Px$. You need to prove this by refutation, since a direct proof of the existential conclusion is not possible.

Exercise 10.18 Prove that $\exists x\,(Px \vee Qx)$ follows from $\exists x\,Px \vee \exists x\,Qx$, and also the other way around.

Exercise 10.19 (*) Prove that $\exists x\,(Px \rightarrow \forall x\,Px)$ is valid. This one requires a proof by refutation as well: show that $\bot$ follows from $\neg\exists x\,(Px \rightarrow \forall x\,Px)$.

10.2.1 Rules for identity

In addition to the rules for the quantifiers we also have to formulate rules for identity which are particularly important for mathematical proofs. The introduction rule is the simplest of all rules. It just states that an object is always equal to itself. It is in fact an axiom, there are no conditions which restrict application of this rule.

$$\begin{array}{l} \vdots \\ t = t \quad \text{I=} \end{array} \tag{10.27}$$

The elimination rule says that we always may replace terms by other terms which refer to the same object. We only have to take care that the variables which occur within these terms do not mess up the binding of variables by quantifiers. The term that we replace may only contain variables that occur freely (within the formula which is subject to the replacement), and the substitute may not contain variables which get bound after replacement. If these condition hold then we may apply the following rule:

$$\begin{array}{l} \vdots \\ t_1 = t_2 / t_2 = t_1 \\ \vdots \\ \varphi \\ \vdots \\ \varphi' \qquad \text{E=} \end{array} \tag{10.28}$$

where φ' is the result of replacing occurrences of t_1 by t_2 in φ (not necessarily all). Here are two simple examples showing the symmetry and transitivity of equality:

$$\begin{array}{llll} 1. & a = b & \text{Ass} & \\ 2. & a = a & \text{I=} & \\ 3. & b = a & \text{E= 1,2} & \end{array} \qquad \begin{array}{llll} 1. & a = b & \text{Ass} & \\ 2. & b = c & \text{Ass} & \\ 3. & a = c & \text{E= 1,2} & \end{array} \tag{10.29}$$

In the first derivation the first occurrence of a in 2 is replaced by b. In the second derivation the occurrence of b in 2 is replaced by a.

10.3 Natural deduction for natural numbers

In chapter 4 an axiomatic system of arithmetic, as introduced by the Italian logician and mathematician Giuseppe Peano, in predicate logical notation has been discussed.

Giuseppe Peano

In this section we want to give a natural deduction format for Peano's arithmetic, as an example of 'real' mathematical proof by means of natural deduction. These kind of systems are used for precise formalization of mathematical proofs, such that they can be checked, or sometimes be found (that is much harder of course), by computers.

Let us first repeat the axioms as discussed in chapter 4.

$$\begin{array}{ll} P1. & \forall x\,(x+0=x) \\ P2. & \forall x\,\forall y\,(x+sy=s(x+y)) \\ P3. & \forall x\,(x\cdot 0=0) \\ P4. & \forall x\,\forall y\,(x\cdot sy=x\cdot y+x) \\ P5. & \neg\exists x\; sx=0 \\ P6. & (\varphi[0/x]\wedge\forall x\,(\varphi\rightarrow\varphi[sx/x]))\rightarrow\forall x\,\varphi \end{array} \tag{10.30}$$

A straightforward manner to build a predicate logical system for arithmetic is to add these axioms to the system as has been introduced in the previous section. For the first five axioms we do not have an alternative. These axioms are then treated as rules without conditions, and can therefore be applied at any time at any place in a mathematical proof.

The last axiom, the principle of induction, can be reformulated as a conditional rule of deduction, in line with the way it is used in mathematical proofs. For the reader who is not familiar with the induction principle, the following simple example clarifies how it works.

> For every natural number n the sum of the first n odd numbers equals n^2.
>
> For 0 this property holds in a trivial way. The sum of the first zero odd numbers is an empty sum and therefore equals 0, which is also 0^2.
>
> Suppose that the property holds for a certain number k (induction hypothesis).
>
> $$1+3+\ldots+(2k-1)=k^2$$

We need to prove that under this condition the property must also hold for $k+1$ (sk). The sum of the first $k+1$ odd numbers is the same as

$$1 + 3 + ... + (2k - 1) + (2k + 1)$$

According the induction hypothesis, this must be equal to

$$k^2 + 2k + 1$$

and this equals $(k+1)^2$.

We have proven the property for 0 and also shown that if it holds for a certain natural number then it must also hold for its successors. From this we derive by induction that the property holds for all natural numbers.

10.3.1 The rule of induction

The inductive axiom in Peano's system can be rephrased as a conditional rule in the following way.

$$\begin{array}{l} \vdots \\ \varphi[0/x] \\ \vdots \\ \left[\begin{array}{l} \dfrac{\varphi[c/x] \quad {\scriptstyle c}}{} \\ \vdots \\ \varphi[sc/x] \\ \vdots \end{array}\right] \\ \vdots \\ \forall x\, \varphi \qquad\qquad {\scriptstyle \text{Ind}} \end{array} \tag{10.31}$$

It mimics the format as has been described by the proof example here above. The formula $\varphi[0/x]$ says that the property φ holds for 0. The subproof represents the inductive step, and starts with the induction hypothesis. We assume $\varphi[c/x]$, i.e., an arbitrary φ-er represented by the parameter c (the induction hypothesis). If this assumption suffices to derive that the successor of c, sc, also must have the property φ, $\varphi[sc/x]$, then φ must hold for all objects, i.e., all the natural numbers.

In terms of the natural deduction system for predicate logic, the induction rule is an additional introduction rule for the universal quantifier. For some cases we can do without this rule and use the generalization rule instead. Here is a simple example which proves

that $x + 1$ coincides with the successor sx of x.

$$
\begin{array}{l}
\left[\begin{array}{ll}
1. & \hfill c \\ \hline
2. & \forall x \forall y\,(x + sy = s(x+y)) \quad \text{P2} \\
3. & c + s0 = s(c+0) \quad \text{E}\forall\ 2\ \text{(twice)} \\
4. & \forall x\,(x + 0 = x) \quad \text{P1} \\
5. & c + 0 = c \quad \text{E}\forall\ 4 \\
6. & c + s0 = sc \quad \text{E=}\ 5{,}3
\end{array}\right] \\
7.\ \ \forall x\,(x + s0 = sx) \quad \text{I}\forall\ 1\text{-}6
\end{array}
\tag{10.32}
$$

The proof demonstrates that this arithmetical theorem is a pure predicate logical consequence of the two first axioms of the Peano system.

In other cases we have to rely on the induction rule to derive a universal statement about the natural numbers. Here is a very simple example:

$$
\begin{array}{l}
1.\ \ 0 + 0 = 0 \quad \text{E}\forall\ \text{P1} \\
\left[\begin{array}{ll}
2. & 0 + c = c \hfill c \\ \hline
3. & 0 + sc = s(0 + c) \quad \text{E}\forall\ \text{P2 (twice)} \\
4. & 0 + sc = sc \quad \text{E=}\ 2{,}3
\end{array}\right] \\
5.\ \ \forall x\,(0 + x = x) \quad \text{Ind}\ 1{,}2\text{-}4
\end{array}
\tag{10.33}
$$

$0 + x = x$ is the property we have proved for all natural numbers x. First we have shown this is true for 0 and then in the induction step, the subproof 2-4, we have shown that the property $0 + c = c$ for an arbitrary c leads to $0 + sc = sc$.

Exercise 10.20 Prove that $\forall x\,(x \cdot s0 = x)$.

Exercise 10.21 Prove that $\forall x\,(0 \cdot x = 0)$.

The following proof uses both rules, generalization *and* induction.

$$
\left[\begin{array}{lll}
\hline
1. & & c \\
\hline
2. & c+s0=sc & \text{E}\forall\ (10.32) \\
3. & sc+0=sc & \text{E}\forall\ \text{P1} \\
4. & c+s0=sc+0 & \text{E}=\ 3,2 \\
& \left[\begin{array}{lll}
\hline
5. & c+sd=sc+d & d \\
\hline
6. & c+ssd=s(c+sd) & \text{E}\forall\ \text{P2} \\
7. & c+ssd=s(sc+d) & \text{E}=\ 5,6 \\
8. & sc+sd=s(sc+d) & \text{E}\forall\ \text{P2} \\
9. & c+ssd=sc+sd & \text{E}=\ 8,7
\end{array}\right] & \\
10. & \forall y\,(c+sy=sc+y) & \text{Ind } 4,5\text{-}9
\end{array}\right] \qquad (10.34)
$$

$$
11. \quad \forall x \forall y\,(x+sy=sx+y) \qquad \text{I}\forall\ 1\text{-}10
$$

The outermost subproof justifies the use of a generalization in the last step, whereas the inner subproof contains the induction step of the inductive proof of 10.

Exercise 10.22 Prove that $\forall x\,\forall y\,(x+y=y+x)$ by filling in the gap represented by the vertical dots. You may use the theorem which has already been proved in (10.34).

$$
\left[\begin{array}{lll}
\hline
1. & & c \\
\hline
& \vdots & \\
n. & c+0=0+c & \ldots \\
& \left[\begin{array}{lll}
\hline
n+1. & c+d=d+c & d \\
\hline
& \vdots & \\
n+k. & c+sd=sd+c & \ldots
\end{array}\right] & \\
n+k+1. & \forall y\,(c+y=y+c) & \text{Ind } n,n+1\text{-}n+k
\end{array}\right]
$$

$$
n+k+2. \quad \forall x \forall y\,(x+y=y+x) \qquad \text{I}\forall\ 1\text{-}n+k+1
$$

Exercise 10.23 Prove that $\forall x\,(x \cdot ss0 = x+x)$.

Exercise 10.24 (*) Prove that $\forall x \forall y\,(x \cdot y = y \cdot x)$.

Exercise 10.25 Prove that $\forall x \forall y \forall z\,(x+(y+z)=(x+y)+z)$.

10.4 Outlook

10.4.1 Completeness and incompleteness

10.4.2 Natural deduction, tableaus and sequents

10.4.3 Intuitionistic logic

10.4.4 Automated deduction

Chapter 11

Computation

Things You Will Learn in This Chapter This chapter gives a lightning introduction to computation with logic. First we will look at computing with propositional logic. You will learn how to put propositional formulas in a format suitable for computation, and how to use the so-called resolution rule. Next, we turn to computation with predicate logic. The procedure for putting predicate logical formulas into computational format is a bit more complicated. You will learn how to transform a predicate logical formula into a set of clauses. Next, in order to derive conclusions from predicate logical clauses, we need to apply a procedure called *unification*. Terms containing variables can sometimes made equal by means of substitution. We will present the so-called *unification algorithm*, and we will prove that if two terms can be made equal, then the unification algoritm computes the most general way of doing so. Finally, unification will be combined with resolution to give an inference mechanism that is very well suited for predicate logical computation, and we will see how this method is put to practical use in the Prolog programming language.

11.1 A Bit of History

Leibniz in his youth

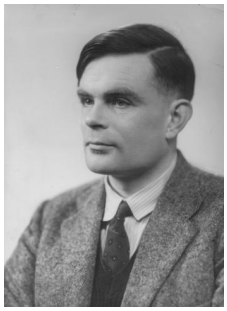
In 1673 the polymath Godfried Wilhelm Leibniz (1645–1716) demonstated to the Royal Society in London a design for a calculation device that was intended to solve mathematical problems by means of execution of logical inference steps. Leibniz was not

only a mathematician, a philosopher and a historian, but also a diplomat, and he dreamed of rational approaches to conflict resolution. Instead of quarrelling without end or even resorting to violence, people in disagreement would simply sit down with their reasoning devices, following the adage *Calculemus* ("Let's compute the solution"). Mechanical computation devices were being constructed from that time on, and in 1928 the famous mathematician David Hilbert posed the challenge of finding a systematic method for mechanically settling mathematical questions formulated in a precise logical language.

David Hilbert

This challenge was called the *Entscheidungsproblem* ("the decision problem"). In 1936 and 1937 Alonzo Church and Alan Turing independently proved that it is impossible to decide algorithmically whether statements of simple school arithmetic are true or false. This result, now known as the Church-Turing theorem, made clear that a general solution to the *Entscheidungsproblem* is impossible. It follows from the Church-Turing theorem that a decision method for predicate logic does not exist. Still, it is possible to define procedures for computing inconsistency in predicate logic, provided that one accepts that these procedures may run forever for certain (consistent) input formulas.

Alonzo Church



Alan Turing

11.2 Processing Propositional Formulas

For computational processing of propositional logic formulas, it is convenient to first put them in a particular syntactic shape.

The simplest propositional formulas are called **literals**. A literal is a proposition letter or the negation of a proposition letter. Here is a BNF definition of literals. We assume that p ranges over a set of proposition letters P.

$$L \ ::= \ p \mid \neg p.$$

Next, a disjunction of literals is called a **clause**. Clauses are defined by the following BNF rule:

$$C \ ::= \ L \mid L \vee C.$$

Finally a CNF formula (formula in conjunctive normal form) is a conjunction of clauses. In a BNF rule:

$$\varphi \ ::= \ C \mid C \wedge \varphi.$$

Formulas in CNF are useful, because it is easy to test them for validity. For suppose φ is in CNF. Then φ consists of a conjunction $C_1 \wedge \cdots \wedge C_n$ of clauses. For φ to be valid, each conjunct clause C has to be valid, and for a clause C to be valid, it has to contain a proposition letter p and its negation $\neg p$. So to check φ for validity, find for each of its clauses C a proposition letter p such that p and $\neg p$ are both in C. In the next section, we will see that there is a simple powerful rule to check CNF formulas for satisfiability.

We will now start out from arbitrary propositional formulas, and show how to convert them into equivalent CNF formulas, in a number of steps. Here is the BNF definition of the language of propositional logic once more.

$$\varphi \ ::= \ p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi)$$

Translating into CNF, first step The first step translates propositional formulas into equivalent formulas that are arrow-free: formulas without $\leftrightarrow$ and $\rightarrow$ operators. Here is how this works:

- Use the equivalence between $p \rightarrow q$ and $\neg p \vee q$ to get rid of $\rightarrow$ symbols.
- Use the equivalence of $p \leftrightarrow q$ and $(\neg p \vee q) \wedge (p \vee \neg q)$, to get rid of $\leftrightarrow$ symbols.

Here is the definition of arrow-free formulas of propositional logic:

$$\varphi \ ::= \ p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi).$$

Translating into CNF, first step in pseudocode We will now write the above recipe in so-called *pseudocode*, i.e., as a kind of fake computer program. Pseudocode is meant to be readable by humans (like you), while on the other hand it is so close to computer digestible form that an experienced programmer can turn it into a real program as a matter of routine.

The pseudocode for turning a propositional formula into an equivalent arrow free formula takes the shape of a function. The function has a name, *ArrowFree*. A key

feature of the definition of *ArrowFree* is that inside the definition, the function that is being defined is mentioned again. This is an example of a phenomenon that you will encouter often in recipes for computation. It is referred to as a *recursive function call*.

What do you have to do to make a formula of the form $\neg\psi$ arrow free? First you ask your dad to make ψ arrow free, and then you put $\neg$ in front of the result. The part where you ask your dad is the recursive function call.

function ArrowFree (φ):
/$\star$ precondition: φ is a formula. $\star$/
/$\star$ postcondition: ArrowFree (φ) returns arrow free version of φ $\star$/
begin function
case
φ is a literal: **return** φ
φ is $\neg\psi$: **return** $\neg$ ArrowFree (ψ)
φ is $\psi_1 \wedge \psi_2$: **return** (ArrowFree (ψ_1) $\wedge$ ArrowFree (ψ_2))
φ is $\psi_1 \vee \psi_2$: **return** (ArrowFree (ψ_1) $\vee$ ArrowFree (ψ_2))
φ is $\psi_1 \rightarrow \psi_2$: **return** ArrowFree ($\neg\psi_1 \vee \psi_2$)
φ is $\psi_1 \leftrightarrow \psi_2$: **return** ArrowFree $((\neg\psi_1 \vee \psi_2) \wedge (\psi_1 \vee \neg\psi_2))$
end case
end function

Note that the pseudocode uses comment lines: everything that is between /$\star$ and $\star$/ is a comment. The first comment of the function states the *precondition*. This is the assumption that the argument of the function is a propositional formula. This assumption is used in the function definition, for notice that the function definition follows the BNF definition of the formulas of propositional logic. The second comment of the function states the *postcondition*. This is the statement that all propositional formulas will be turned into equivalent arrow free formulas.

You can think of the precondition of a function recipe as a statement of rights, and of the postcondition as a statement of duties. The pre- and postcondition together form a *contract*: if the precondition is fulfilled (i.e., if the function is called in accordance with its rights) the function definition ensures that the postcondition will be fulfilled (the function will perform its duties). This way of thinking about programming is called *design by contract*.

Exercise 11.1 Work out the result of the function call ArrowFree $(p \leftrightarrow (q \leftrightarrow r))$.

Translating into CNF, second step Our next step is to turn an arrow free formula into a formula that only has negation signs in front of proposition letters. A formula in this shape is called a formula in *negation normal form*. Here is the BNF definition of formulas in negation normal form:

$$\begin{array}{rcl} L & ::= & p \mid \neg p \\ \varphi & ::= & L \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi). \end{array}$$

What this says is that formulas in negation normal form are formulas that are constructed out of literals by means of taking conjunctions and disjunctions.

The principles we use for translating formulas into negation normal form are the equivalence between$\neg(p \wedge q)$ and $\neg p \vee \neg q$, and that between $\neg(p \vee q)$ and $\neg p \wedge \neg q$. If we encounter a formula of the form $\neg(\psi_1 \wedge \psi_1)$, we can "push the negation sign inward" by replacing it with $\neg\psi_1 \vee \neg\psi_2$, and similarly for formulas of the form $\neg(\psi_1 \vee \psi_1)$. Again, we have to take care of the fact that the procedure will have to be carried out recursively. Also, if we encounter double negations, we can let them cancel out: formula $\neg\neg\psi$ is equivalent to ψ. Here is the pseudocode for turning arrow free formulas into equivalent formulas in negation normal form.

function NNF (φ):
/$\star$ precondition: φ is arrow-free. $\star$/
/$\star$ postcondition: NNF (φ) returns NNF of φ $\star$/
begin function
case
  φ is a literal: **return** φ
  φ is $\neg\neg\psi$: **return** NNF (ψ)
  φ is $\psi_1 \wedge \psi_2$: **return** (NNF (ψ_1) $\wedge$ NNF (ψ_2))
  φ is $\psi_1 \vee \psi_2$: **return** (NNF (ψ_1) $\vee$ NNF (ψ_2))
  φ is $\neg(\psi_1 \wedge \psi_2)$: **return** (NNF ($\neg\psi_1$) $\vee$ NNF ($\neg\psi_2$))
  φ is $\neg(\psi_1 \vee \psi_2)$: **return** (NNF ($\neg\psi_1$) $\wedge$ NNF ($\neg\psi_2$))
end case
end function

Again, notice the recursive function calls. Also notice that there is a contract consisting of a precondition stating that the input to the NNF function has to be arrow free, and guaranteeing that the output of the function is an equivalent formula in negation normal form.

Exercise 11.2 Work out the result of the function call NNF ($\neg(p \vee \neg(q \wedge r))$).

Translating into CNF, third step The third and final step takes a formula in negation normal form and produces an equivalent formula in conjunctive normal form. This function uses an auxiliary function DIST, to be defined below. Intuitively, DIST(ψ_1, ψ_2) gives the CNF of the *disjunction* of ψ_1 and ψ_2, on condition that ψ_1, ψ_2 are themselves in CNF.

function CNF (φ):
/$\star$ precondition: φ is arrow-free and in NNF. $\star$/
/$\star$ postcondition: CNF (φ) returns CNF of φ $\star$/
begin function
case
  φ is a literal: **return** φ
  φ is $\psi_1 \wedge \psi_2$: **return** CNF (ψ_1) $\wedge$ CNF (ψ_2)

φ is $\psi_1 \vee \psi_2$: **return** DIST (CNF (ψ_1), CNF (ψ_2))
end case
end function

Translating into CNF, auxiliary step The final thing that remains is define the CNF of the disjunction of two formulas φ_1, φ_2 that are both in CNF. For that, we use:

- $(p \wedge q) \vee r$ is equivalent to $(p \vee r) \wedge (q \vee r)$,
- $p \vee (q \wedge r)$ is equivaent to $(p \vee q) \wedge (p \vee r)$.

The assumption that φ_1 and φ_2 are themselves in CNF helps us to use these principles. The fact that φ_1 is in CNF means that either φ_1 is a conjunction $\psi_{11} \wedge \psi_{12}$ of clauses, or it is a single clause. Similarly for φ_2. This means that either at least one of the two principles above can be employed, or both of φ_1, φ_2 are single clauses. In this final case, $\varphi_1 \vee \varphi_2$ is in CNF.

function DIST (φ_1, φ_2):
/$\star$ precondition: φ_1, φ_2 are in CNF. $\star$/
/$\star$ postcondition: DIST (φ_1, φ_2) returns CNF of $\varphi_1 \vee \varphi_2$ $\star$/
begin function
case
φ_1 is $\psi_{11} \wedge \psi_{12}$: **return** DIST ($\psi_{11}$,$\varphi_2$) $\wedge$ DIST (ψ_{12},φ_2)
φ_2 is $\psi_{21} \wedge \psi_{22}$: **return** DIST ($\varphi_1$,$\psi_{21}$) $\wedge$ DIST (φ_1,ψ_{22})
otherwise: **return** $\varphi_1 \vee \varphi_2$
end case
end function

In order to put a propositional formula φ in conjunctive normal form we can proceed as follows:

(1) First remove the arrows $\rightarrow$ and $\leftrightarrow$ by means of a call to ArrowFree.

(2) Next put the result of the first step in negation normal form by means of a call to NNF.

(3) Finally, put the result of the second step in conjunctive normal form by means of a call to CNF.

In other words, if φ is an arbitrary propositional formula, then

$$\text{CNF}(\text{NNF}(\text{ArrowFree}(\varphi)))$$

gives an equivalent formula in conjunctive normal form.

Exercise 11.3 Work out the result of the function call CNF $((p \vee \neg q) \wedge (q \vee r))$.

Exercise 11.4 Work out the result of the function call CNF $((p \wedge q) \vee (p \wedge r) \vee (q \wedge r))$.

11.3 Resolution

It is not hard to see that if $\neg\varphi \vee \psi$ is true, and $\varphi \vee \chi$ is also true, then $\psi \vee \chi$ has to be true as well. For assume $\neg\varphi \vee \psi$ and $\varphi \vee \chi$ are true. If φ is true, then it follows from $\neg\varphi \vee \psi$ that ψ. If on the other hand $\neg\varphi$ is true, then it follows from $\varphi \vee \chi$ that χ. So in any case we have $\psi \vee \chi$. This inference principle is called *resolution*. We can write the resolution rule as:

$$\frac{\neg\varphi \vee \psi \quad \varphi \vee \chi}{\psi \vee \chi}$$

Note that Modus Ponens can be viewed as a special case of this. Modus Ponens is the rule:

$$\frac{\varphi \rightarrow \psi \quad \varphi}{\psi}$$

But this can be written with negation and disjunction:

$$\frac{\neg\varphi \vee \psi \quad \varphi \vee \top}{\psi}$$

The idea of resolution leads to a powerful inference rule if we apply it to two clauses. Clauses are disjunctions of literals, so suppose have two clauses $A_1 \vee \cdots \vee A_n$ and $B_1 \vee \cdots \vee B_m$, where all of the A and all of the B are literals. Assume that A_i and B_j are complements (one is the negation of the other, i.e., one has the form p and the other the form $\neg p$). Then the following inference step is valid:

$$\frac{A_1 \vee \cdots \vee A_n \qquad B_1 \vee \cdots \vee B_m}{A_1 \vee \cdots \vee A_{i-1} \vee A_{i+1} \vee \cdots \vee A_n \vee B_1 \vee \cdots \vee B_{j-1} \vee B_{j+1} \vee \cdots \vee B_m}$$

This rule is called the *resolution rule*. It was proposed by J. Alan Robinson (one of the inventors of the Prolog programming language) in 1965, in a landmark paper called "A Machine-Oriented Logic Based on the Resolution Principle." The rule allows to fuse two clauses together in a single clause.

Before we go on, it is convenient to switch to set notation. Let us day that a clause is a *set of literals*, and a *clause form* a *set of clauses*. Then here is an example of a clause form:

$$\{\{p, \neg q, r, \neg r\}, \{p, \neg p\}\}.$$

Resolution can now be described as an operation on pairs of clauses, as follows:

$$\frac{C_1 \cup \{p\} \qquad \{\neg p\} \cup C_2}{C_1 \cup C_2}$$

Alternatively, we may view resolution as an operation on clause forms, as follows:

$$\frac{C_1, \ldots, C_i \cup \{p\}, \{\neg p\} \cup C_{i+1}, C_{i+2}, \ldots, C_n}{C_1, \ldots, C_i \cup C_{i+1}, C_{i+2}, \ldots, C_n}$$

The empty clause, notation [], corresponds to an empty disjunction. To make a disjunction true, at least one of the disjuncts has to be true. It follows that the empty clause is always *false*.

The empty clause form, notation $\emptyset$, corresponds to an empty conjunction, for clause form is conjunctive normal form. A conjunction is true if all of its conjuncts are true. It follows that the empty clause form is always *true*.

Exercise 11.5 Suppose a clause C_i contains both p and $\neg p$, for some proposition letter p. Show that the following rule can be used to simplify clause forms:

$$\frac{C_1, \ldots, C_i, \ldots, C_n}{C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n}\ p \in C_i, \neg p \in C_i$$

You have to show that this rule is *sound*. Assuming that the premisse is true, show that the conclusion is also true.

If a clause form has [] (the empty clause) as a member, then, since [] is always false, and since clause forms express conjunctions, the clause form is always *false*. In other words, a clause form that has [] as a member expresses a contradiction. So if we can derive the empty clause [] from a clause form, we know that the clause form is *not satisfiable*.

Thus, resolution can be used as a *refutation technique*. To check whether ψ follows logically from $\varphi_1, \ldots, \varphi_n$, check whether the clause form corresponding to

$$\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \neg\psi$$

is satisfiable, by attempting to derive the empty clause [] from the clause form, by means of the resolution rule. If the clause form is not satisfiable, the original inference is valid.

Example: we want to check whether from $\neg p \vee \neg q \vee r$, and $\neg p \vee q$ it follows that $\neg p \vee r$. Construct the formula

$$(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge \neg(\neg p \vee r).$$

This is the conjunction of the premisses together with a negation of the conclusion. Bring this in conjunctive normal form:

$$(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge p \wedge \neg r.$$

Write this formula in clause form:

$$\{\{\neg p, \neg q, r\}, \{\neg p, q\}, \{p\}, \{\neg r\}\}.$$

Applying resolution for $\neg q, q$ to the first two clauses gives:

$$\{\{\neg p, r\}, \{p\}, \{\neg r\}\}.$$

Applying resolution for $\neg p, p$ to the first two clauses gives:

$$\{\{r\}, \{\neg r\}\}.$$

Applying resolution for $r, \neg r$ gives:

$$\{[]\}$$

We have derived a clause form containing the empty clause. This is a proof by resolution that the inference is valid. We have tried to construct a situation where the premisses are true and the conclusion is false, but this attempt has led us to a contradiction. No doubt you will have noticed that this *refutation strategy* is quite similar to the strategy behind tableau style theorem proving.

Exercise 11.6 Stefan

Exercise 11.7 Stefan

Exercise 11.8 You are a professor and you are trying to organize a congress. In youer attempt to draw up a list of invited speakers, you are considering professors a, b, c, d, e, f. Unfortunately, your colleagues have big egos, and informal consultation concerning their attitudes towards accepting your invitation reveals the following constraints:

- At least one of a, b is willing to accept.
- Exactly two of a, e, f will accept.
- b will accept if and only if c also accepts an invitation.
- a will accept if and only if d will not get invited.
- Similarly for c and d.
- If d will not get an invitation, e will refuse to come.

Use propositional logic to set up a clause set representing these constraints. (Hint: first express the constraints as propositional formulas, using proposition letters a, b, c, d, e, f. Next, convert this into a clause form.)

Exercise 11.9 As it turns out, there is only one way to satisfy all constraints of Exercise 11.8. Give the corresponding propositional valuation. (Hint: you can use resolution to simplify the clause form of the previous exercise.)

We know that checking (un)satisfiability for propositional logic can always be done. It cannot always be done efficiently. The challenge of building so called *sat solvers* for propositional logic is to speed up satisfiability checking for larger and larger classes of propositional formulas. Modern sat solvers can check satisfiability of clause forms containing hunderds of proposition letters. The usual way to represent a clause form is as a list of lines of integers. Here is an example of this so-called DIMACS format:

```
c Here is a comment.
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

The first line gives a comment (that's what it says, and what is says is correct). The second line states that this is a problem in conjunctive normal form with five proposition letters and three clauses. Each of the next three lines is a clause. 0 indicates the end of a clause.

The home page of a popular sat solver called *MiniSat* can be found at `http://minisat.se/`. MiniSat calls itself a minimalistic, open-source SAT solver. It was developed to help researchers and developers to get started on SAT. So this is where *you* should start also if you want to learn more. Running the example (stored in file `sat.txt`) in *MiniSat* gives:

```
jve@vuur:~/tmp$ minisat2 sat.txt
This is MiniSat 2.0 beta
WARNING: for repeatability, setting FPU to use double precision
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:  5                                                    |
|  Number of clauses:    3                                                    |
|  Parsing time:         0.00         s                                       |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
|         0 |       0        0        0 |        0        0    nan |  0.000 % |
===============================================================================
restarts              : 1
conflicts             : 0              (nan /sec)
decisions             : 1              (0.00 % random) (inf /sec)
propagations          : 0              (nan /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 14.58 MB
CPU time              : 0 s

SATISFIABLE
```

Now let's have another look at the earlier clause form we computed:

$$\{\{\neg p, \neg q, r\}, \{\neg p, q\}, \{p\}, \{\neg r\}\}.$$

Written with indices, it looks like this:

$$\{\{\neg p_1, \neg p_2, p_3\}, \{\neg p_1, p_2\}, \{p_1\}, \{\neg p_3\}\}.$$

And here is the clause form in DIMACS format:

```
p cnf 4 3
-1 -2 3 0
-1 2 0
1 0
-3 0
```

If this text is stored in file `sat2.txt` then here is the result of feeding it to `minisat`:

```
jve@vuur:~/tmp$ minisat2 sat2.txt
This is MiniSat 2.0 beta
WARNING: for repeatability, setting FPU to use double precision
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:  4                                                    |
|  Number of clauses:    3                                                    |
|  Parsing time:         0.00         s                                       |
Solved by unit propagation
UNSATISFIABLE
```

General background on propositional satisfiability checking can be found at `http://www.satisfiability.org/`.

11.4 Automating Predicate Logic

Alloy (`http://alloy.mit.edu`) is a software specification tool based on first order logic plus some relational operators. Alloy automates predicate logic by using bounded exhaustive search for counterexamples in small domains [Jac00]. Alloy does allow for automated checking of specifications, but only for small domains. The assumption that most software design errors show up in small domains is known as the *small domain hypothesis* [Jac06]. The *Alloy* website links to a useful tutorial, where the three key aspects of Alloy are discussed: logic, language and analysis.

The *logic* behind Alloy is predicate logic plus an operation to compute the transitive closures of relations. The transitive closure of a relation R is by definition the smallest transitive relation that contains R.

Exercise 11.10 Give the transitive closures of the following relations. (Note: if a relation is already transitive, the transitive closure of a relation is that relation itself.)

(1) $\{(1,2),(2,3),(3,4)\}$,

(2) $\{(1,2),(2,3),(3,4),(1,3),(2,4)\}$,

(3) $\{(1,2),(2,3),(3,4),(1,3),(2,4),(1,4)\}$,

(4) $\{(1,2),(2,1)\}$,

(5) $\{(1,1),(2,2)\}$.

The language is the set of syntactic conventions for writing specifications with logic. The analysis of the specifications takes place by means of bounded exhaustive search for counterexamples. The technique used for this is translation to a propositional satisfiability problem, for a given domain size.

Here is an example of a check of a fact about relations. We just defined the transitive closure of a relation. In a similar way, the symmetric closure of a relation can be defined. The symmetric closure of a relation R is the smallest symmetric relation that contains R.

We call the *converse* of a binary R the relation that results from changing the direction of the relation. A common notation for this is $R^\smile$. The following holds by definition:

$$R^\smile \;=\; \{(y,x) \mid (x,y) \in R\}.$$

We claim that $R \cup R^\smile$ is the symmetric closure of R. To establish this claim, we have to show two things: (i) $R \cup R^\smile$ is symmetric, and (ii) $R \cup R^\smile$ is the least symmetric relation that contains R. (i) is obvious. To establish (ii), we assume that there is some symmetric relation S with $R \subseteq S$ (S contains R). If we can show that $R \cup R^\smile$ is contained in S we

know that $R \cup R^\smile$ is the least relation that is symmetric and contains R, so that it has to be the symmetric closure of R, by definition.

So assume $R \subseteq S$ and assume S is symmetric. Let $(x, y) \in R \cup R^\smile$. We have to show that $(x, y) \in S$. From $(x, y) \in R \cup R^\smile$ it follows either that $(x, y) \in R$ or that $(x, y) \in R^\smile$. In the first case, $(x, y) \in S$ by $R \subseteq S$, and we are done. In the second case, $(y, x) \in R$, and therefore $(y, x) \in S$ by $R \subseteq S$. Using the fact that S is symmetric we see that also in this case $(x, y) \in S$. This settles $R \cup R^\smile \subseteq S$.

Now that we know what the symmetric closure of R looks like, we can define it in predicate logic, as follows:

$$Rxy \vee Ryx.$$

Now here is a question about operations on relations. Given a relation R, do the following two procedures boil down to the same thing?

First take the symmetric closure, next the transitive closure

First take the transitive closure, next the symmetric closure

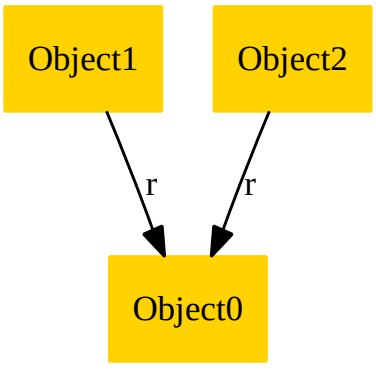
If we use R^+ for the transitive closure of R and $R \cup R^\smile$ for the symmetric closure, then the question becomes:

$$(R \cup R^\smile)^+ \stackrel{?}{=} R^+ \cup R^{+\smile}$$

Here is an Alloy version of this question:

```
sig Object { r : set Object }
assert claim { *(r + ~r) = *r + ~*r }
check claim
```

If you run this in Alloy, the system will try to find counterexamples. Here is a counterexample that it finds:



To see that this is indeed a counterexample, note that for this R we have:

$$\begin{aligned} R &= \{(1,0),(2,0)\} \\ R^{\smile} &= \{(0,1),(0,2)\} \\ R \cup R^{\smile} &= \{(1,0),(2,0),(0,1),(0,2)\} \\ R^{+} &= \{(1,0),(2,0)\} \\ R^{\smile +} &= \{(0,1),(0,2)\} \\ R^{+} \cup R^{+\smile} &= \{(1,0),(2,0),(0,1),(0,2)\} \\ (R \cup R^{\smile})^{+} &= \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\} \end{aligned}$$

Here is another question about relations. Suppose you know that R and S are transitive. Does it follow that their composition, the relation you get by first taking an R step and next an S step, is also transitive? The composition of R and S is indicated by $R \circ S$.

Here is a definition of the composition of R and S in predicate logic:

$$\exists z(Rxz \wedge Szy).$$

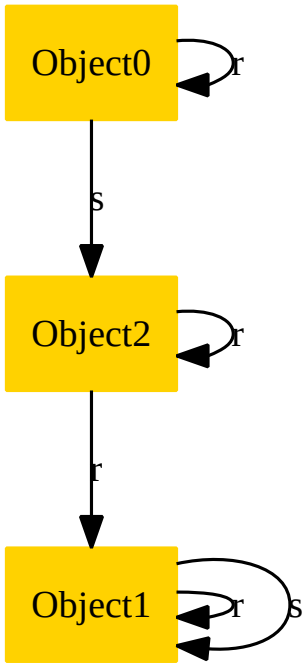
Exercise 11.11 Find a formula of predicate logic stating that if R and S are transitive then their composition is transitive as well.

The answer to exercise 11.11 gives us a rephrasing of our original question: does the formula φ that you constructed have counterexamples (model where it is not true), or not?

The Alloy version of the question is again very succinct. This is because we can state the claim that R is transitive simply as: $R = R^{+}$.

```
sig Object { r,s: set Object }
fact { r = ^r and s = ^s }
assert claim { r.s = ^(r.s) }
check claim
```

Again the system finds counterexamples:



In this example, $R = \{(0,0),(2,2),(2,1),(1,1)\}$ and $S = \{(0,2),(1,1)\}$.

Exercise 11.12 This exercise is about the example relations R and S that were found by Alloy. For these R and S, give $R \circ S$ and give $(R \circ S)^+$. Check that these relations are not the same, so $R \circ S$ is not transitive.

11.5 Conjunctive Normal Form for Predicate Logic

Now suppose we have a predicate logical formula. We will assume that there are no free variables: each variable occurrence is bound by a quantifier. In other words: we assume that the formula is *closed*.

To convert closed formulas of predicate logic to conjunctive normal form, the following steps have to be performed:

(1) Convert to arrow-free form.

(2) Convert to negation normal form by moving $\neg$ signs inwards. This involves the laws of De Morgan, plus the following quantifier principles:

- $\neg\forall x\varphi \leftrightarrow \exists x\neg\varphi$.
- $\neg\exists x\varphi \leftrightarrow \forall x\neg\varphi$.

(3) Standardize variables, in orde to make sure that each variable binder $\forall x$ or $\exists x$ occurs only once in the formula. For example, $\forall xPx \vee \exists xQx$ should be changed to $\forall xPx \vee \exists yQy$. Or a more complicated example: $\forall x(\exists y(Py \wedge Rxy) \vee \exists ySxy)$ gets

changed to $\forall x(\exists y(Py \wedge Rxy) \vee \exists z Sxz)$. In the standardized version, each variable name x will have exactly one binding quantifier in the formula. This will avoid confusion later, when we are going to drop the quantifiers.

(4) Move all quantifiers to the outside, by using the following equivalences:

- $(\forall x\varphi \wedge \psi) \leftrightarrow \forall x(\varphi \wedge \psi)$,
- $(\forall x\varphi \vee \psi) \leftrightarrow \forall x(\varphi \vee \psi)$.
- $(\exists x\varphi \wedge \psi) \leftrightarrow \exists x(\varphi \wedge \psi)$,
- $(\exists x\varphi \vee \psi) \leftrightarrow \exists x(\varphi \vee \psi)$.

Note that these principles hold because accidental capture of variables is impossible. We standardized the variables, so we may assume that every variable name x has exactly one binding quantifier in the formula. Recall that there are no free variables.

(5) Get rid of existential quantifiers, as follows.

- If the outermost existential quantifier $\exists x$ of the formula is not in the scope of any universal quantifiers, remove it, and replace every occurrence of x in the formula by a fresh constant c.
- If the outermost existential quantifier $\exists x$ of the formula is in the scope of universal quantifiers $\forall y_1$ through $\forall y_n$, remove it, and replace every occurrence of x in the formula by a fresh function $f(y_1, \ldots y_n)$. (Such a function is called a *Skolem function*.)
- Continue like this until there are no existential quantiers left.

This process is called *skolemization*.

(6) Remove the universal quantifiers.

(7) Distribute disjunction over conjunction, using the equivalences:

- $((\varphi \wedge \psi) \vee \chi) \leftrightarrow ((\varphi \vee \chi) \wedge (\psi \vee \chi))$,
- $(\varphi \vee (\psi \wedge \chi)) \leftrightarrow ((\varphi \vee \chi) \wedge (\varphi \vee \chi))$.

To illustrate the stages of this process, we run through an example. We start with the formula:

$$\forall x(\exists y(Py \vee Rxy) \rightarrow \exists y Sxy).$$

First step: make this arrow-free:

$$\forall x(\neg\exists y(Py \vee Rxy) \vee \exists y Sxy).$$

Second step: move negations inwards:

$$\forall x(\forall y(\neg Py \wedge \neg Rxy) \vee \exists y Sxy).$$

Third step: standardize variables:

$$\forall x(\forall y(\neg Py \wedge \neg Rxy) \vee \exists z Sxz).$$

Fourth step: move quantifiers out:

$$\forall x \forall y \exists z((\neg Py \wedge \neg Rxy) \vee Sxz).$$

Fifth step: skolemization:

$$\forall x \forall y((\neg Py \wedge \neg Rxy) \vee Sxf(x,y)).$$

Sixth step: remove universal quantifiers:

$$((\neg Py \wedge \neg Rxy) \vee Sxf(x,y)).$$

Seventh step, distribute disjunction over conjunction:

$$(\neg Py \vee Sxf(x,y)) \wedge (\neg Rxy \vee Sxf(x,y)).$$

The clause form of the predicate logical formula contains two clauses, and it looks like this:

$$\{\{\neg Py, Sxf(x,y)\}, \{\neg Rxy, Sxf(x,y)\}\}.$$

Exercise 11.13 Stefan

Exercise 11.14 Stefan

11.6 Substitutions

If we want to compute with first order formulas in clause form, it is necessary to be able to handle substitution of terms in such forms. In fact, we will look at the effets of substitutions on terms, on clauses, and on clause forms.

A *variable binding* is a pair consisting of a variable and a term. A binding *binds* the variable to the term. A binding (v,t) is often represented as $v \mapsto t$. A binding is *proper* if it does not bind variable v to term v (the same variable, viewed as a term). A variable substitution is a finite list of proper bindings, satisfying the requirement that no variable v occurs as a lefthand member in more than one binding $v \mapsto t$.

The substitution that changes nothing is called the *identity substitution*. It is represented by the empty list of variable bindings. We will denote it as ϵ.

The domain of a substitution is the list of all lefthand sides of its bindings. The range of a substitution is the list of all righthand sides of its bindings. For example, the domain of the substitution $\{x \mapsto f(x), y \mapsto x\}$ is $\{x,y\}$, and its range is $\{x, f(x)\}$.

Substitutions give rise to mappings from terms to terms via the following recursion. Let σ be a substitution. Then a term t either has the form v (the term is a variable) or the form c (the term is a constant) or the form $f(t_1, \ldots, t_n)$ (the term is a function with n argument terms). The result σt of applying the substitution to the term t is given by:

- $\sigma v := \sigma(v)$,
- $\sigma c := c$,
- $\sigma f(t_1, \ldots, t_n) := f(\sigma t_1, \ldots, \sigma t_m)$.

Next, we define the result of applying a substitution σ to a formula φ, again by recursion on the structure of the formula.

- $\sigma P(t_1, \ldots, t_n) := P(\sigma t_1, \ldots, \sigma t_n)$,
- $\sigma(\neg\varphi) := \neg(\sigma\varphi)$,
- $\sigma(\varphi \wedge \psi) := (\sigma\varphi \wedge \sigma\psi)$,
- $\sigma(\varphi \vee \psi) := (\sigma\varphi \vee \sigma\psi)$,
- $\sigma(\varphi \to \psi) := (\sigma\varphi \to \sigma\psi)$,
- $\sigma(\varphi \leftrightarrow \psi) := (\sigma\varphi \leftrightarrow \sigma\psi)$,
- $\sigma(\forall v\varphi) := \forall v\sigma'\varphi$, where σ' is the result of removing the binding for v from σ,
- $\sigma(\exists v\varphi) := \exists v\sigma'\varphi$, where σ' is the result of removing the binding for v from σ.

Exercise 11.15 Stefan

Exercise 11.16 Stefan

The composition of substitution σ with substitution τ should result in the substitution that one gets by applying σ after τ. The following definition has the desired effect.

Definition 11.17 (Composition of substitution representations) Let

$$\theta = [v_1 \mapsto t_1, \ldots, v_n \mapsto t_n] \text{ and } \sigma = [w_1 \mapsto r_1, \ldots, w_m \mapsto r_m]$$

be substitution representations. Then $\theta \cdot \sigma$ is the result of removing from the sequence

$$[w_1 \mapsto \theta(r_1), \ldots, w_m \mapsto \theta(r_m), v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$$

the bindings $w_1 \mapsto \theta(r_i)$ for which $\theta(r_i) = w_i$, and the bindings $v_j \mapsto t_j$ for which $v_j \in \{w_1, \ldots, w_m\}$.

Exercise 11.18 Prove that this definition gives the correct result.

Applying the recipe for composition to $\{x \mapsto y\} \cdot \{y \mapsto z\}$ gives $\{y \mapsto z, x \mapsto y\}$, applying it to $\{y \mapsto z\} \cdot \{x \mapsto y\}$ gives $\{x \mapsto z, y \mapsto z\}$. This example illustrates the fact that order of application of substitution matters. Substitutions do *not* commute.

Exercise 11.19 Stefan

Exercise 11.20 Stefan

We use the notion of composition to define a relation $\sqsubseteq$ on the set S of all substitutions (for given sets of variables V and terms T), as follows. $\theta \sqsubseteq \sigma$ iff there is a substitution ρ with $\theta = \rho \cdot \sigma$. ($\theta \sqsubseteq \sigma$ is sometimes pronounced as: 'θ is less general than σ.')

The relation $\sqsubseteq$ is **reflexive**. For all θ we have that $\theta = \epsilon \cdot \theta$, and therefore $\theta \sqsubseteq \theta$. The relation is also **transitive**. $\sqsubseteq$ is transitive because if $\theta = \rho \cdot \sigma$ and $\sigma = \tau \cdot \gamma$ then $\theta = \rho \cdot (\tau \cdot \gamma) = (\rho \cdot \tau) \cdot \gamma$, i.e., $\theta \sqsubseteq \gamma$. A relation that is reflexive and transitive is called a **pre-order**, so what we have just shown is that $\sqsubseteq$ is a pre-order.

11.7 Unification

If we have two expressions A and B (where A, B can be terms, or formulas, or clauses, or clause forms), each containing variables, then we are interested in the following questions:

- Is there a substitution θ that makes A and B equal?
- How do we find such a substitution in an efficient way?

We introduce some terminology for this. The substitution θ *unifies* expressions A and B if $\theta A = \theta B$. The substitution θ *unifies* two sequences of expressions $(A_1, \ldots, A_n)$ and $(B_1, \ldots, B_n)$ if, for $1 \leq i \leq n$, θ unifies A_i and B_i. Note that unification of pairs of atomic formulas reduces to unification of sequences of terms, for two atoms that start with a different predicate symbol do not unify, and two atoms $P(t_1, \ldots, t_n)$ and $P(s_1, \ldots, s_n)$ unify iff the sequences $(t_1, \ldots, t_n)$ and $(s_1, \ldots, s_n)$ unify.

What we are going to need to apply resolution reasoning (Section 11.3) to predicate logic is unification of pairs of atomic formulas.

For example, we want to find a substitution that unifies the pair

$$P(x, g(a, z)), P(g(y, z), x).$$

In this example case, such unifying substitutions exist. A possible solution is

$$\{x \mapsto g(a, z), y \mapsto a\}.$$

for applying this substitution gives $P(g(a, z), g(a, z))$. Another solution is

$$\{x \mapsto g(a, b), y \mapsto a, z \mapsto b\}.$$

In this case, the second solution is an instance of the first, for

$$\{x \mapsto g(a,b), y \mapsto a, z \mapsto b\} \sqsubseteq \{x \mapsto g(a,z), y \mapsto a\},$$

because

$$\{x \mapsto g(a,b), y \mapsto a, z \mapsto b\} = \{z \mapsto b\} \cdot \{x \mapsto g(a,z), y \mapsto a\}.$$

So we see that solution $\{x \mapsto g(a,z), y \mapsto a\}$ is more general than solution $\{x \mapsto g(a,b), y \mapsto a, z \mapsto b\}$.

If a pair of atoms is unifiable, it is useful to try and identify a solution that is as general as possible, for the more general a solution is, the less unnecessary bindings it contains. These considerations motivate the following definition.

Definition 11.21 If θ is a unifier for a pair of expressions (a pair of sequences of expressions), then θ is called an mgu (a most general unifier) if $\sigma \sqsubseteq \theta$ for every unifier σ for the pair of expressions (the pair of sequences of expressions).

In the above example, $\{x \to g(a,z), y \mapsto a\}$ is an mgu for the pair

$$P(x, g(a,z)), P(g(y,z), x).$$

The **Unification Theorem** says that if a unifier for a pair of sequences of terms exists, then an mgu for that pair exists as well. Moreover, there is an algorithm that produces an mgu for any pair of sequences of terms in case these sequences are unifiable, and otherwise ends with failure.

We will describe the *unification algorithm* and prove that it does what it is supposed to do. This constitutes the proof of the theorem.

We give the algorithm in stages.

First we define unification of terms *UnifyTs*, in three cases.

- Unification of two variables x and y gives the empty substitution if the variables are identical, and otherwise a substitution that binds one variable to the other.
- Unification of x to a non-variable term t fails if x occurs in t, otherwise it yields the binding $\{x \mapsto t\}$.
- Unification of $f\bar{t}$ and $g\bar{r}$ fails if the two variable names are different, otherwise it yields the return of the attempt to do term list unification on $\bar{t}$ and $\bar{r}$.

If unification succeeds, a unit list containing a representation of a most general unifying substitution is returned. Return of the empty list indicates unification failure.

Unification of term lists (*UnifyTlists*):

- Unification of two empty term lists gives the identity substitution.
- Unification of two term lists of different length fails.
- Unification of two term lists $t_1, \ldots, t_n$ and $r_1, \ldots, r_n$ is the result of trying to compute a substitution $\sigma = \sigma_n \circ \cdots \circ \sigma_1$, where
  - σ_1 is a most general unifier of t_1 and r_1,
  - σ_2 is a most general unifier of $\sigma_1(t_2)$ and $\sigma_1(r_2)$,
  - σ_3 is a most general unifier of $\sigma_2\sigma_1(t_3)$ and $\sigma_2\sigma_1(r_3)$,
  - and so on.

Our task is to show that these two unification functions do what they are supposed to do: produce a unit list containing an mgu if such an mgu exists, produce the empty list in case unification fails.

The proof consists of a Lemma and two Theorems. The Lemma is needed in Theorem 11.23. The Lemma establishes a simple property of mgu's. Theorem 11.24 establishes the result.

Lemma 11.22 If σ_1 is an mgu of t_1 and s_1, and σ_2 is an mgu of

$$(\sigma_1 t_2, \ldots, \sigma_1 t_n) \text{ and } (\sigma_1 s_2, \ldots, \sigma_1 s_n),$$

then $\sigma_2 \cdot \sigma_1$ is an mgu of $(t_1, \ldots, t_n)$ and $(s_1, \ldots, s_n)$.

Proof. Let θ be a unifier of $(t_1, \ldots, t_n)$ and $(s_1, \ldots, s_n)$. Given this assumption, we have to show that $\sigma_2 \cdot \sigma_1$ is more general than θ.

By assumption about θ we have that $\theta t_1 = \theta s_1$. Since σ_1 is an mgu of t_1 and s_1, there is a substitution ρ with $\theta = \rho \cdot \sigma_1$.

Again by assumption about θ, it holds for all i with $1 < i \leq n$ that $\theta t_i = \theta s_i$. Since $\theta = \rho \cdot \sigma_1$, it follows that

$$(\rho \cdot \sigma_1)t_i = (\rho \cdot \sigma_1)s_i,$$

and therefore,

$$\rho(\sigma_1 t_i) = \rho(\sigma_1 s_i).$$

Since σ_2 is an mgu of $(\sigma_1 t_2, \ldots, \sigma_1 t_n)$ and $(\sigma_1 s_2, \ldots, \sigma_1 s_n)$, there is a substitution ν with $\rho = \nu \cdot \sigma_2$. Therefore,

$$\theta = \rho \cdot \sigma_1 = (\nu \cdot \sigma_2) \cdot \sigma_1 = \nu \cdot (\sigma_2 \cdot \sigma_1).$$

This shows that $\sigma_2 \cdot \sigma_1$ is more general than θ, which establishes the Lemma. □

Theorem 11.23 shows, by induction on the length of term lists, that if *unifyTs* t s does what it is supposed to do, then *unifyTlists* also does what it is supposed to do.

Theorem 11.23 Suppose *unifyTs t s* yields a unit list containing an mgu of t and s if the terms are unifiable, and otherwise yields the empty list. Then *unifyTlists* $\bar{t}$ $\bar{s}$ yields a unit list containing an mgu of $\bar{t}$ and $\bar{s}$ if the lists of terms $\bar{t}$ and $\bar{s}$ are unifiable, and otherwise produces the empty list.

Proof. If the two lists have different lengths then unification fails.

Assume, therefore, that $\bar{t}$ and $\bar{s}$ have the same length n. We proceed by induction on n.

Basis $n = 0$, i.e., both $\bar{t}$ and $\bar{s}$ are equal to the empty list. In this case the ϵ substitution unifies $\bar{t}$ and $\bar{s}$, and this is certainly an mgu.

Induction step $n > 0$. Assume $\bar{t} = (t_1, \ldots, t_n)$ and $\bar{s} = (s_1, \ldots, s_n)$, with $n > 0$. Then $\bar{t} = t_1 : (t_2, \ldots, t_n)$ and $\bar{s} = s_1 : (s_2, \ldots, s_n)$, where : expresses the operation of putting an element in front of a list.

What the algorithm does is:

(1) It checks if t_1 and s_1 are unifiable by calling *unifyTs* t_1 s_1. By the assumption of the theorem, *unifyTs* t_1 s_1. yields a unit list (σ_1), with σ_1 an mgu of t_1 and s_1 if t_1 and s_1 are unifiable, and yields the empty list otherwise. In the second case, we know that the lists $\bar{t}$ and $\bar{s}$ are not unifiable, and indeed, in this case *unifyTlists* will produce the empty list.

(2) If t_1 and s_1 have an mgu σ_1, then the algorithm tries to unify the lists

$$(\sigma_1 t_2, \ldots, \sigma_1 t_n) \text{ and } (\sigma_1 s_2, \ldots, \sigma_1 s_n),$$

i.e., the lists of terms resulting from applying σ_1 to each of $(t_2, \ldots, t_n)$ and each of $(s_2, \ldots, s_n)$. By induction hypothesis we may assume that applying `unifyTlists` to these two lists produces a unit list (σ_2), with σ_2 an mgu of the lists, if the two lists are unifiable, and the empty list otherwise.

(3) If σ_2 is an mgu of the two lists, then the algorithm returns a unit list containing $\sigma 2 \cdot \sigma_1$. By Lemma 11.22, $\sigma 2 \cdot \sigma_1$ is an mgu of $\bar{t}$ and $\bar{s}$.

□

Theorem 11.24 clinches the argument. It proceeds by structural induction on terms. The induction hypothesis will allow us to use Theorem 11.23.

Theorem 11.24 The function *unifyTs t s* either yields a unit list (γ) or the empty list. In the former case, γ is an mgu of t and s. In the latter case, t and s are not unifiable.

Proof. Structural induction on the complexity of (t, s). There are 4 cases.

1. Both terms are variables, i.e., t equals x, s equals y. In this case, if x and y are identical, the ϵ substitution is surely an mgu of t and s. This is what the algorithm yields. If x and y are different variables, then the substitution $\{x \mapsto y\}$ is an mgu of x and y. For suppose $\sigma x = \sigma y$. Then $\sigma x = (\sigma \cdot \{x \mapsto y\})x$, and for all z different from x we have $\sigma z = (\sigma \cdot \{x \mapsto y\})z$. So $\sigma = \sigma \cdot \{x \mapsto y\}$.

2. $t = x$ and s is not a variable. If x is not an element of the variables of s, then $\{x \mapsto s\}$ is an mgu of t and s. For if $\sigma x = \sigma s$, then $\sigma x = (\sigma \cdot \{x \mapsto s\})x$, and for all variables z different from x we have that $\sigma z = (\sigma \cdot \{x \mapsto s\})z$. $\sigma = \sigma \cdot \{x \mapsto s\}$. If x is an element of the varialbes of s, then unification fails (and this is what the algorithm yields).

3. $s = x$ and t not a variable. Similar to case 2.

4. $t = f(\bar{t})$ and $s = g(\bar{s})$. Then t and s are unifiable iff (i) f equals g and (ii) the term lists $\bar{t}$ and $\bar{s}$ are unifiable. Moreover, ν is an mgu of t and s iff f equals g and ν is an mgu of $\bar{t}$ and $\bar{s}$.

By the induction hypothesis, we may assume for all subterms t' of t and all subterms s' of s that *unifyTs t' s'* yields the empty list if t' and s' do not unify, and a unit list (ν), with ν an mgu of t' and s' otherwise. This means the condition of Theorem 11.23 is fulfilled, and it follows that *unifyTlists* $\bar{t}$ $\bar{s}$ yields (ν), with ν an mgu of $\bar{t}$ and $\bar{s}$, if the term lists $\bar{t}$ and $\bar{s}$ unify, and *unifyTlists* $\bar{t}$ $\bar{s}$ yields the empty list if the term lists do not unify.

This establishes the Theorem. □

Some examples of unification attempts:

- unifyTs x $(f(x)$ yields $()$.
- unifyTs x $(f(y)$ yields $(\{x \mapsto y\})$.
- unifyTs $g(x, a)$ $g(y, x)$ yields $(\{x \mapsto a, y \mapsto a\})$.

Further examples are in the exercises.

Exercise 11.25 Stefan

Exercise 11.26 Stefan

Exercise 11.27 Stefan

11.8 Resolution with Unification

Suppose we have clausal forms for predicate logic. Then we can adapt the resolution rule to predicate logic by combining resolution with unification, as follows. Assume that $C_1 \cup \{P\bar{t}\}$ and $C_2 \cup \{\neg P\bar{s}\}$ are predicate logical clauses. The two literals $P\bar{t}$ and $P\bar{s}$ need not be the same in order to apply resolution to the clauses. It is enough that $P\bar{t}$ and $P\bar{s}$ are *unifiable*.

For what follows, let us assume that the clauses in a predicate logical clause form do not have variables in common. This assumption is harmless: see Exercise 11.28.

Exercise 11.28 Suppose C and C' are predicate logical clauses, and they have a variable x in common. Show that it does not affect the meaning of the clause form $\{C, C'\}$ if we replace the occurrence(s) of x in C' by occurrences of a fresh variable z ("freshness" of z means that z occurs in neither C nor C'.)

Assume that $C_1 \cup \{P\bar{t}\}$ and $C_2 \cup \{\neg P\bar{s}\}$ do not have variables in common. Then the following inference rule is sound:

Resolution Rule with Unification

$$\frac{C_1 \cup \{P\bar{t}\} \qquad \{\neg P\bar{s}\} \cup C_2}{\theta C_1 \cup \theta C_2} \ \theta \text{ is mgu of } \bar{t} \text{ and } \bar{s}$$

Here is an example application:

$$\frac{\{Pf(y), Qg(y)\} \quad \{\neg Pf(g(a)), Rby\}}{\{Qg(g(a)), Rbg(a)\}} \ \text{mgu } \{y \mapsto g(a)\} \text{ applied to } Pf(y) \text{ and } Pf(g(a))$$

It is also possible to use unification to 'simplify' individual clauses. If $P\bar{t}$ and $P\bar{s}$ (or $\neg P\bar{t}$ and $\neg P\bar{s}$) occur in the same clause C, and θ is an mgu of $\bar{t}$ and $\bar{s}$, then θC is called a **factor** of C. The following inference rules identify literals by means of factorisation:

Factorisation Rule (pos)

$$\frac{C_1 \cup \{P\bar{t}, P\bar{s}\}}{\theta(C_1 \cup \{P\bar{t}\})} \ \theta \text{ is mgu of } \bar{t} \text{ and } \bar{s}$$

Factorisation Rule (neg)

$$\frac{C_1 \cup \{\neg P\bar{t}, \neg P\bar{s}\}}{\theta(C_1 \cup \{\neg P\bar{t}\})} \ \theta \text{ is mgu of } \bar{t} \text{ and } \bar{s}$$

An example application:

$$\frac{\{Px, Pf(y), Qg(y)\}}{\{Pf(y), Qg(y)\}} \ \text{mgu } \{x \mapsto f(y)\} \text{ applied to } Px \text{ and } Pf(y))a$$

Resolution and factorisation can also be combined, as in the following example:

$$\dfrac{\dfrac{\{Px, Pf(y), Qg(y)\}}{\{Pf(y), Qg(y)\}} \text{ factorisation} \qquad \{\neg Pf(g(a)), Rby\}}{\{Qg(g(a)), Rbg(a)\}} \text{ resolution}$$

Computation with first order logic uses these rules, together with a **search strategy** for selecting the clauses and literals to which resolution and unification are going to be applied. A particularly simple strategy is possible if we restrict the format of the clauses in the clause forms.

It can be proved (although we will not do so here) that resolution and factorisation for predicate logic form a *complete* calculus for predicate logic. What this means is that a

clause form F is unsatisfiable if and only if there exists a deduction of the empty clause $[]$ from F by means of resolution and factorisation.

On the other hand, there is an important difference with the case of propositional logic. Resolution refutation is a *decision method* for (un)satisfiability in propositional logic. In the case of predicate logic, this cannot be the case, for predicate logic has no decision mechanism. Resolution/factorisation refutation does not decide predicate logic. More precisely, if a predicate logical clause F is unsatisfiable, then there existss a resolution/factorisation derivation of $[]$ from F, but if F is satisfiable, then the derivation process may never stop, as the possibilities of finding ever new instantiations by means of unification are inexhaustible.

11.9 Prolog

Prolog, which derives its name from *programming with logic*, is a general purpose programming language that is popular in artificial intelligence and computational linguistics, and that derives its force from a clever search strategy for a particular kind of restricted clause form for predicate logic.

Alain Colmerauer

The language was conceived in the 1970s by a group around Alain Colmerauer in Marseille. The first Prolog system was developed in 1972 by Alain Colmerauer and Phillipe Roussel. A well known public domain version of Prolog us SWI-Prolog, developed in Amsterdam by Jan Wielemaker. See `http://www.swi-prolog.org/`.

Jan Wielemaker

Definition 11.29 A clause with just one positive literal is called a *program clause*. A clause with only negative literals is called a *goal clause*.

A program clause $\{\neg A_1, \ldots, \neg A_n, B\}$ can be viewed as an implication $(A_1 \wedge \cdots, A_n) \rightarrow B$. A goal clause $\{\neg A_1, \ldots, \neg A_n\}$ can be viewed as a degenerate implication $(A_1 \wedge \cdots, A_n) \rightarrow []$, where [] is the empty clause (expressing a contradiction). Goal and program clauses together constitute what is is called *pure Prolog*. The computation strategy of Prolog consists of combining a goal clause with a number of program clauses in an attempt to derive the empty clause. Look at the goal clause like this:

$$(A_1 \wedge \cdots, A_n) \rightarrow [].$$

From this, [] can be derived if we manage to derive each of $A_1, \ldots, A_n$ from the Prolog program clauses. An example will clarify this. In the following example of a pure Prolog program, we use the actual Prolog notation, where predicates are lower case, variables are upper case, and implications $(A_1 \wedge \cdots, A_n) \rightarrow B$ are written backwards, as $B :- A_1, \ldots, A_n$.

```
plays(heleen,X) :- haskeys(X).
plays(heleen,violin).
plays(hans,cello).
plays(jan,clarinet).

haskeys(piano).
haskeys(accordeon).
haskeys(keyboard).
haskeys(organ).

woodwind(clarinet).
woodwind(recorder).
woodwind(oboe).
woodwind(bassoon).
```

Each line is a program clause. All clauses except one consist of a single positive literal. The exception is the clause `plays(heleen,X) :- haskeys(X).` This is the Prolog version of $\forall x(H(x) \rightarrow P(h, x))$. Here is an example of interaction with this database (read from a file `music.pl`) in SWI-Prolog:

```
[jve@pidgeot lia]$ pl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.6.64)
Copyright (c) 1990-2008 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.
```

```
For help, use ?- help(Topic). or ?- apropos(Word).

?- [music].
% music compiled 0.00 sec, 3,328 bytes
true.

?- plays(heleen,X).
```

The last line constitutes the Prolog query. The system now computes a number of answers, and we can use ; after each answer to prompt for more, until the list of answers is exhausted. This is what we get:

```
X = piano ;
X = accordeon ;
X = keyboard ;
X = organ ;
X = violin.
```

Prolog queries can also be composite:

```
?- woodwind(X),plays(Y,X).
X = clarinet,
Y = jan ;
false.

?-
```

The strategy that Prolog uses to compute answers is resolution refutation. Take the first query as an example. The Prolog system combines the database clauses (the program clauses in the file `music.pl`) with the goal clause `plays(heleen,X)` $\rightarrow$ $[]$, and sure enough, the system can derive the empty clause $[]$ from this, in quite a number of ways. Each derivation involves a unifying substitution, and these substitutions are what the system computes for us. The exercises to follow invite you to play a bit more with Prolog programming.

Exercise 11.30 Stefan

Exercise 11.31 Stefan

Exercise 11.32 Stefan

Exercise 11.33 Stefan

Exercise 11.34 Stefan

Summary *After having finished this chapter you can check whether you have mastered the material by answering the following questions:*

- *What is the definition of clausal form for propositional logic?*
- *How can formulas of propositional logic be translated into clausal form?*
- *How does the resolution rule work for propositional logic, and why is it sound?*
- *What are SAT solvers? How do they work?*
- *What is the definition of clausal form for predicate logic?*
- *How can formulas of predicate logic be translated into clausal form?*
- *How can variable substitutions be represented as finite sets of bindings?*
- *How are substitutions composed?*
- *What does it mean that one substitution is more general than another one?*
- *What is an mgu?*
- *What is unification? What does the unification algorithm do?*
- *What is the rule of resolution with unification? Why is it sound?*
- *What is the rule of factorisation? Why is it sound?*
- *What are program clauses and goal clauses?*
- *What is the computation mechanism behind Prolog?*

Outlook

Chapter 12

Logic and Reality

Chapter 13

Logic and Science

Just material for the chapter:

13.1 Formal Derivations in a Logical Theory

In this section we give some demonstrations of formal derivations in Presburger arithmetic. If you dislike the challenge of formal puzzles or if, for some reason or other, you do not want to learn about formal reasoning inside predicate logic then you can skip to the next section.

We start with a formal proof of the fact that 0 is even (recall the definition of being even from the previous section):

1.	$\forall x\ x + \mathbf{0} = x$	PA1
2.	$(\forall x\ x + \mathbf{0} = x) \rightarrow \mathbf{0} + \mathbf{0} = \mathbf{0}$	PL axiom 2
3.	$\mathbf{0} + \mathbf{0} = \mathbf{0}$	MP from 1,2
4.	$\exists y\ \ y + y = \mathbf{0}.$	EGEN from 3.

Next, here are some facts about $+$ that we would like to prove:

- $x + 0 = x$,
- $x + (y + z) = (x + y) + z$,
- $x + y = y + x$.

Clearly, these facts should hold, for they are properties of the operation of adding natural numbers that we are all familiar with. But the challenge now is to *prove* these properties using the axioms and rules of predicate logic, together with the axioms of the theory of Presburger arithmetic.

The proof of $x + 0 = x$ is easy:

1.	$\forall x\ x + 0 = x$	PA1
2.	$\forall x\ x + 0 = x \rightarrow x + 0 = x$	PL Axiom 2
3.	$x + 0 = x$	MP from 1,2

The proof of $(x + y) + z = x + (y + z)$ is more involved. For this we have to use the induction scheme. In particular, we will prove this fact by induction on the structure of z. First we prove the case where z equals $\mathbf{0}$.

1.	$(x + y) + \mathbf{0} = x + y$	PA3
2.	$y + \mathbf{0} = y$	PA3
3.	$y + \mathbf{0} = y \rightarrow (x + y) + \mathbf{0} = x + (y + \mathbf{0})$	From 1,2 by PL axioms 2,5
4.	$(x + y) + \mathbf{0} = x + (y + \mathbf{0})$	MP from 2,3

Next, we assume $(x + y) + z = x + (y + z)$ and we prove $(x + y) + sz = x + (y + sz)$:

1.	$(x + y) + z = x + (y + z)$	assumption
2.	$(x + y) + sz = s((x + y) + z)$	PA4
3.	$s((x + y) + z) = s(x + (y + z))$	from 1 with PL
4.	$x + s(y + z) = s(x + (y + z))$	PA4
5.	$y + sz = s(y + z)$	PA4
6.	$s(x + (y + z)) = x + (y + sz)$	from 4,5 with PL
7.	$(x + y) + sz = x + (y + sz)$	from 3,6 with PL
8.	$(x + y) + z = x + (y + z) \rightarrow (x + y) + sz = x + (y + sz)$	1, conditionalisation
9.	$\forall z\ ((x + y) + z = x + (y + z) \rightarrow (x + y) + sz = x + (y + sz))$	8, UGEN

We can consider $(x + y) + z = x + (y + z)$ as a property of z. Let us call it $P(z)$. Then we see that we have proved two facts: $P(\mathbf{0})$ and $\forall z(P(z) \rightarrow P(sz))$. This means we can use the induction scheme PA5-scheme to prove the fact for *all numbers*.

1.	$P(\mathbf{0})$	just proved
2.	$\forall z(P(z) \rightarrow P(sz))$	just proved
3.	$(P(\mathbf{0}) \wedge \forall z(P(z) \rightarrow P(sz))) \rightarrow \forall w\ P(w)$	PA5-scheme
4.	$\forall w\ P(w)$	from 1,2,3 by MP

If this meticulous way of proving things seems convoluted to you then you are missing a point. We are not proving $(x + y) + z = x + (y + z)$ because we are worried about

is truth, but because we are worried about whether this fact can be established with the rather minimal means at our disposal. Another important point is that every fact we have proved can be used as a building block for the next proof. As an example of this we invite you to prove that $x + y = y + x$ by induction on the structure of y, using the fact that $(x + y) + z = x + (y + z)$ holds, in the next couple of exercises.

Exercise 13.1 Let $P(y)$ express the property of y that $x + y = y + x$. In order to prove this property of all numbers we have to prove it for zero, and next prove that if it holds for a number then it also holds for its successor. In this particular case we need an induction inside an induction, for as it turns out, we need induction to establish $P(\mathbf{0})$. Your first exercise is to prove $P(\mathbf{0})$, i.e.,$x + \mathbf{0} = \mathbf{0} + x$, by induction on the structure of x.

Exercise 13.2 Again, let $P(y)$ express the property of y that $x+y = y+x$. What you have proved in the previous exercise is the fact $P(\mathbf{0})$. In this exercise we invite you to establish $\forall y(P(y) \rightarrow P(sy))$.

Combining the results of exercises 13.1 and 13.2, we can conclude from $P(\mathbf{0})$ and $\forall y(P(y) \rightarrow P(sy))$ that $\forall y\ P(y)$, with an application of the PA5-scheme.

For a final example, let $EO(x)$ express the property that x is even or odd:

$$\exists y\ y + y = x \vee \exists y\ s(y + y) = x. \qquad (x \text{ is even-or-odd})$$

We have already derived that $\mathbf{0}$ has the property of being even, and it follows from this by propositional reasoning that $\mathbf{0}$ also has the property of being even-or-odd. So we know that $EO(\mathbf{0})$ is derivable in Presburger arithmetic.

Exercise 13.3 [Challenging] Assume of a certain x that x has property EO. Next, derive in Presburger arithmetic that it follows that sx also has the property. Conclude with the PA5-scheme that every x has the property of being even-or-odd.

Presburger arithmetic is a *decidable theory*: there exists a method for checking whether a formula is derivable from the axioms or not. It also has the (rather exceptional) property that if a statement is *not* derivable then the negation of that statement is derivable.

13.2 Undecidability

The deep reason behind the undecidability of predicate logic is the fact that its expressive power is so great that it is possible to state undecidable queries.

One of the famous undecidable queries is the *halting problem*. The halting problem is a problem of computational procedures that compute a result for a given input. Here is what a *halting algorithm* would look like:

- Input: a specification of a computational procedure P, and an input I for that procedure
- Output: an answer 'halt' if P halts when applied to I, and an answer 'loop' otherwise.

The following piece of reasoning, in 5 steps, shows that such an algorithm cannot exist:

(1) Suppose there is an algorithm to solve the halting problem. Call this H.

(2) Then H takes a computational procedure P as input, together with an input I to that procedure, and decides. Note that H is itself a procedure; H takes two inputs, P and I.

(3) Let S be the procedure that is like H, except that it takes one input P, and then calls $H(P, P)$.

(4) Consider the following new procedure N for processing inputs P:

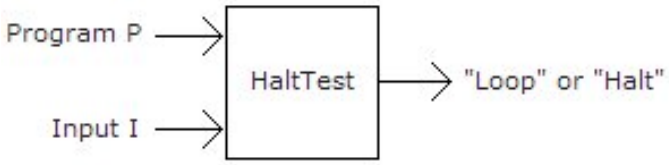
> If $S(P)$ says "halt", then loop,
> and if $S(P)$ says "loop", then print "halt" and terminate.

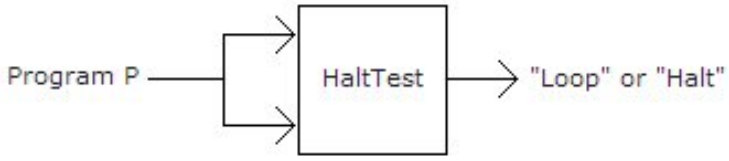
(5) What does N do when applied to N itself? In other words, what is the result of executing $N(N)$?

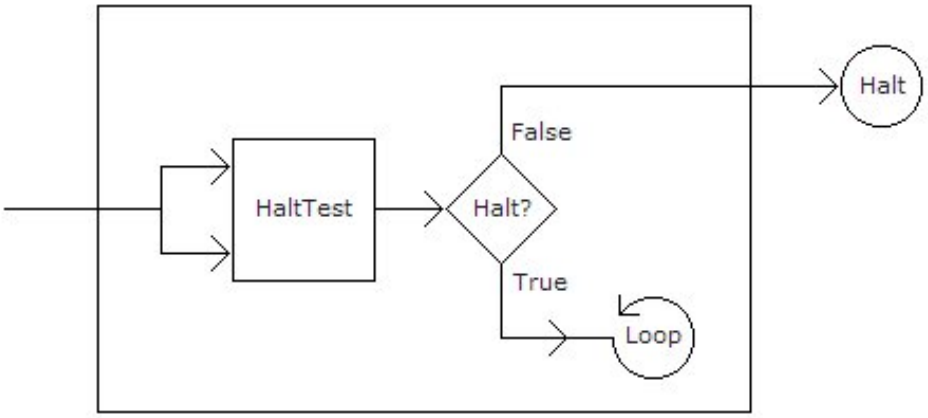
- Suppose N halts on input N. Then H should answer 'halt' when H is applied to N with input N, for H is supposed to be a correct halting algorithm. But then, by construction of the procedure, N loops. Contradiction.
- Suppose N loops on input N. Then H should answer 'loop' when H is applied to N with input N, for H is supposed to be a correct halting algorithm. But then, by construction of the procedure, N prints 'halt' and stops. Contradiction.

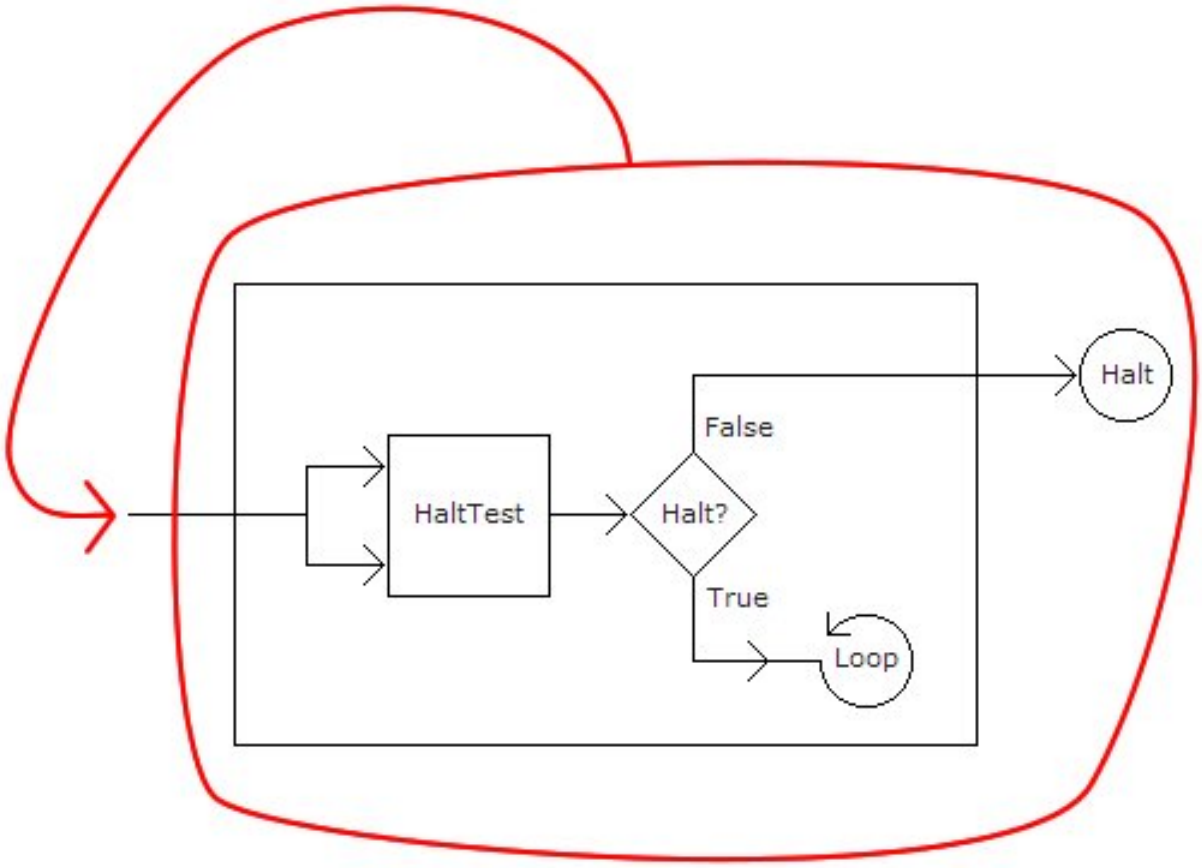
We have a contradiction in both cases. Therefore a halting algorithm H cannot exist.

In Pictures . . .





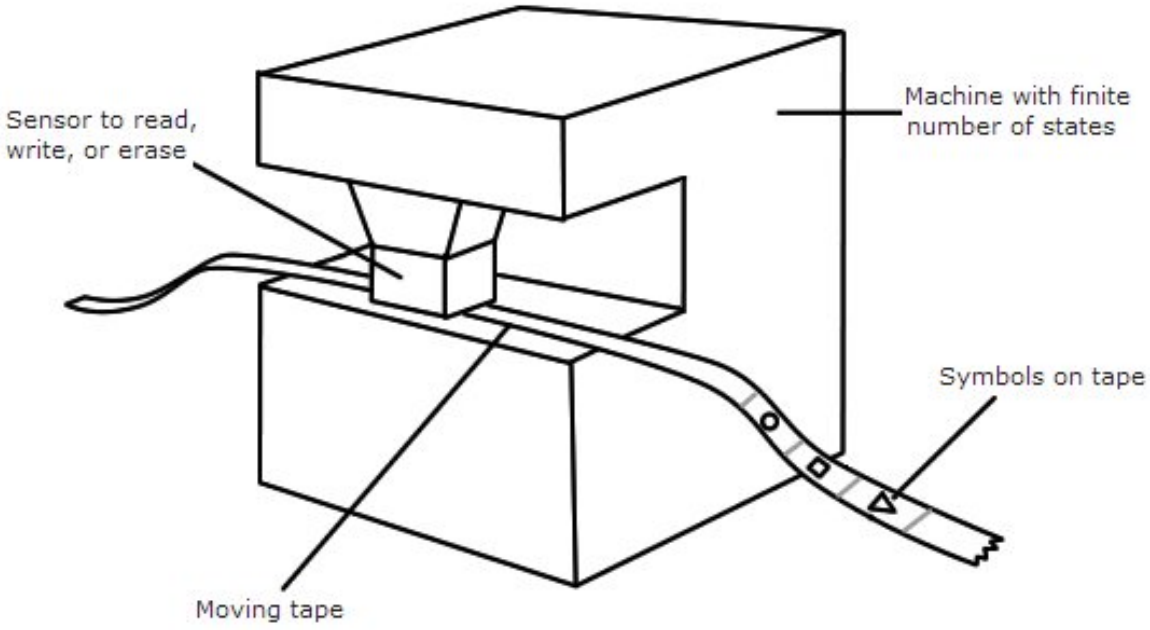




Alan Turing's insight: a language that allows the specification of 'universal procedures' such as H, S and N cannot be decidable. And predicate logic *is* such a language. The formal proof of the undecidability of predicate logic consists of:

- A **very general** definition of *computational procedures*.
- A demonstration of the fact that such computational procedures can be expressed in predicate logic.
- A demonstration of the fact that the halting problem for computational procedures is undecidable (see the above sketch).
- A formulation of the halting problem in predicate logic.

This formal proof was provided by Alan Turing in [Tur36]. The computational procedures he defined for this purpose were later called *Turing machines*.



A Turing Machine

Chapter 14

Logic and Philosophy

(Very preliminary list of topics)

14.1 The Meaning of Meaning

What are meanings?

Extension and Intension

Sense and Reference

Possible worlds

Rigid and Non-rigid Designation

Truth and Falsehood

14.2 Philosophical Distinctions

Analytic versus Synthetic

A priori versus a posteriori

14.3 Knowledge, Justification, Belief

Knowledge and Truth

The Nature of Justification

Knowledge as Justified True Belief

Belief and Belief Revision

Scientific Knowledge

The Problem of Induction

Mathematical Knowledge

The Problem of Logical Omniscience

14.4 Ontology

Is existence a property?

Essential versus accidental properties

Existence and Identity

Leibniz' Law

Existence as "being the value of a variable"

14.5 Paradoxes

Implementations

Chapter 15

Introduction to Functional Programming

15.1 Functions and Programs

Programs and subroutines within programs are actually very similar to functions. They take parameters as arguments and return a result. For example, a program for calculating the prime factors of some number takes an integer as input and returns a list of integers. The difference to functions, however, is that programs can have side effects. Side effects are changes in the state of the system besides the output that the function returns and that remain after the procedure has returned. They can consist in a change of variable names, in writing some intermediate result to a database, and so on. For example, the Ruby program

```
x = 4 ; x += 2
```

first assigns the value 4 to the variable x, then changes this assignment to $x + 2$. So after execution of the program the variable x evaluates to 6.

There are basically two things about computation: the evaluation of an expression and the change of state that this evaluation brings about. Imperative programming focuses on the state and how to modify it with a sequence of commands. In contrast to this, declarative programming focuses on the evaluation itself. In the case of functional programming, computation corresponds to the evaluation of functions.

Functional programming is not a spectator sport, so let us dive into it, and learn about it by doing. The language we are going to use is called Haskell. Haskell is a member of the Lisp family, as are Scheme, ML, Occam, Clean and Erlang. It was designed to form a standard for functional programming languages and was named after the mathematician and logician Haskell B. Curry.

Three very important characteristic features of Haskell are the following. First, functions are first-class citizens. This means that functions may be passed as arguments to other functions and also can be returned as the result of some function. Second, functions are permitted to be recursive. The significance of this we will see in Section 15.4. And

third, arguments of functions are only evaluated when needed, if at all. This is called *lazy evaluation* and it allows the use of infinite and partially-defined data structures.

15.2 Using the Book Code

In order to run the programs in this book, you need a so-called *interpreter* or *compiler*. An interpreter is a system that allows you to execute function definitions interactively. The two most widely used for Haskell are *Hugs* and *GHCi*. The former can be downloaded from `http://haskell.org/hugs/`, and the latter comes with the *Glasgow Haskell Compiler* (GHC), which can be found at `http://haskell.org/ghc/`. Both of these interpreters will cater for your needs. It may be a good idea to install both, for they give slightly different error messages, and if the feedback of one interpreter baffles you, the other one may give you a better hint. If the error messages of both Hugs or GHCi confuse you, you might want to try *Helium*, a Haskell interpreter designed for learning Haskell, with better error messages than either Hugs or GHCi. See `http://www.cs.uu.nl/helium/`. (But be aware that it does not yet support all Haskell features that we will use in this book.)

Haskell code will be typeset in frames, in `typewriter` font. It constitutes the chapter modules, with exception of code that defines functions that are already predefined by the Haskell system or somewhere else in the chapter. This code defined elsewhere is shaded `grey`. Typewriter font is also used for pieces of interaction with the Haskell interpreter, but these illustrations are not boxed.

Subparts of bigger programs are usually collected in modules, each of which sits in its own file. This way you can easily use old code when writing new programs. The following lines declare the Haskell module for the code of this chapter. This module is called `FP`.

```
module FP

where

import Data.List
import Data.Char
```

We import the predefined modules `List` and `Char`, which provide useful functions for handling lists and characters as well as strings. The `Data` in front of them specifies their place in Haskell's module hierarchy.

All functional programs discussed in this book can be found at the book website: `http://www.cwi.nl/~jve/lai/`.

15.3 First Experiments

We assume that you succeeded in retrieving a Haskell interpreter and that you managed to install it on your computer. When you start it, you will get a prompt that looks as follows in the case of Hugs:

```
Prelude>
```

or

```
Hugs.Base>
```

And like this in the case of GHCi and Helium:

```
Prelude>
```

This string is the prompt when no user-defined files are loaded. If this prompt is shown, only the predefined functions from the Haskell Prelude are available; these definitions are given in the online Haskell report[1]. If you want to quickly learn a lot about how to program in Haskell, you should get into the habit of consulting this file regularly. The definitions of all the standard operations are open source code, and are there for you to learn from. The Haskell Prelude may be a bit difficult to read at first, but you will soon get used to the syntax.

Online user manuals for Hugs and GHCi can be found on the internet.[2] The most important commands to know in the beginning are:

- `:l` ⟨ *file name* ⟩ for loading a file or module
- `:r` for reloading the currently loaded file
- `:t` ⟨ *expression* ⟩ for displaying the type of an expression
- `:q` for quitting the interpreter.

You can use the interpreter as a calculator. Let's calculate the number of seconds in a leap year. A leap year has 366 days, each day has 24 hours, each hour has 60 minutes and each minute has 60 seconds, so this is what we get:

```
Prelude> 366 * 24 * 60 * 60
31622400
```

Exercise 15.1 Try out a few calculations using $\star$ for multiplication, + for addition, − for subtraction, ^ for exponentiation, / for division. By playing with the system, find out what the precedence order is among these operators.

[1] `http://www.haskell.org/onlinereport/standard-Prelude.html`

[2] At `http://cvs.haskell.org/Hugs/pages/hugsman/index.html` for Hugs, and at `http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html` for GHCi.

Parentheses can be used to override the built-in operator precedences:

```
Prelude> (2 + 3)^4
625
```

Exercise 15.2 How much is `2^3^4`? Does the interpreter read this as `(2^3)^4` or as `2^(3^4)`?

We can also define our own functions. Here is an example:

```
Prelude> square 2 where square x = x * x
4
```

We use a function `square`, and on the same line we tell the interpreter what we mean by it, using the reserved keyword `where`. Here is another way of achieving the same, this time using the reserved keywords `let` and `in`:

```
Prelude> let square x = x * x in square 3
9
```

We could also write the square function by means of lambda abstraction and simply apply this lambda expression to the desired argument.

```
Prelude> (\ x -> x * x) 4
16
```

Usually, however, we define functions not in the prompt but in a module, give it a name and a proper type declaration. For the square function, this can look like follows.

```
square :: Int -> Int
square x = x * x
```

The intention is that variable `x` stands proxy for a number of type `Int`. The result, the squared number, also has type `Int`. The function `square` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`. This is precisely what the type indication `Int -> Int` expresses.

In Haskell it is not strictly necessary to always give explicit type declarations. For instance, the definition of `square` would also work without the type declaration, since the system can infer the type from the definition. However, it is good programming practice to give explicit type declarations even when this is not strictly necessary. These type declarations are an aid to understanding, and they greatly improve the digestibility of functional programs for human readers. Moreover, they ensure that to some extent the code must do what you want it to do. By writing down the intended type of a function you exclude the option of implementing a function that does not have this type, because the compiler would promptly reject it.

To load and use the chapter code, simply start up the interpreter with `hugs FP` or `ghci FP`, or explicitly load the module with `:l FP`.

```
Prelude> :l FP
FP> square 7
49
FP> square (-3)
9
FP> square (square 7)
2401
FP> square (square (square 7))
5764801
```

Let us briefly look at three other types that we are going to use quite often: types for characters and strings, and the so-called Boolean type. The Haskell type of characters is `Char`. Strings of characters have type `String`, which is an abbreviation for `[Char]`, a list of characters. Similarly, lists of integers have type `[Int]`. The empty string (or the empty list) is `[]`. Examples of characters are `'a'`, `'b'` (note the single quotes) and an example of a string is `"Hello World!"` (note the double quotes). In fact, `"Hello World!"` can be seen as an abbreviation of the following character list:

```
['H','e','l','l','o',' ','W','o','r','l','d','!']
```

This in turn can be seen as shorthand for the result of putting 'H' in front of the list that results from putting 'e' in front of the list ... in front of the list that results from putting '!' in front of the empty list, as follows:

```
'H':'e':'l':'l':'o':' ':'w':'o':'r':'l':'d':'!':[]
```

Exercise 15.3 The colon `:` in the last example is an operator. Can you see what is its type?

Since strings have type `String`, properties of strings have type `String -> Bool`. (Recall that properties are unary relations, which can be seen as characteristic functions of sets, i.e. functions from elements of the set to truth values.) Here is a simple property, the property of being a word that contains an 'h':

```
hword :: String -> Bool
hword []     = False
hword (x:xs) = (x == 'h') || (hword xs)
```

This definition uses *pattern matching*: `(x:xs)` is the prototypical non-empty list. More on this in Section **??** below.

What the definition of `hword` says is that the empty string is not an `hword`, and a non-empty string is an `hword` if either the head of the string is the character `h`, or the tail of the string is an `hword`. Note that the function `hword` is called again from the body of its own definition. You will encounter such *recursive* function definitions again and again below. We say more about them in section 15.4.

As you can see, characters are indicated in Haskell with single quotes. The following calls to the definition show that strings are indicated with double quotes:

```
FP> hword "haskell"
True
FP> hword "curry"
False
```

The type `Bool` of Booleans (so-called after George Boole) consists of the two truth-values `True` and `False`. If we say that an expression is of type Boolean, we mean that the expression is either true or false, i.e. that it denotes a truth value. Boolean expressions can be combined with the logical operators for negation, conjunction and disjunction. Here are the Haskell versions:

- Conjunction is `&&`.
- Disjunction is `||`.
- Negation is `not`.

The type of negation is `Bool -> Bool`. The types of the prefix versions of conjunction and disjunction are given by:

- `(&&) :: Bool -> Bool -> Bool`.
- `(||) :: Bool -> Bool -> Bool`.

Note that the parentheses `( )` change an infix operator into a prefix operator. To express 'bright and beautiful' in Haskell, we can either say `bright && beautiful` or `(&&) bright beautiful`. In general, if `op` is an infix operator, `(op)` is the prefix version of the operator. Thus, `2^10` can also be written as `(^) 2 10`. In general, if `op` is an infix operator, `(op x)` is the operation resulting from applying `op` to its right hand side argument, `(x op)` is the operation resulting from applying `op` to its left hand side argument, and `(op)` is the prefix version of the operator (this is like the abstraction of the operator from both arguments). Thus `(^2)` is the squaring operation, `(2^)` is the operation that computes powers of 2, and `(^)` is exponentiation. Functions like these are called *sections*.

Exercise 15.4 Which property does `(>3)` denote? And which property does `(3>)` denote?

We can also change a prefix operator into an infix operator by using backticks. As an example, assume we define a function `plus x y = x + y`. Instead of calling it with `plus 2 3`, we can also call it with `2 `plus` 3`. We will use backticks very often with the function `elem`, introduced in Section 16.4 below.

15.4 Recursion

Point your webcam at your computer screen. What do you see? A computer screen, with on it a picture of a computer screen, with on it .... This is an example of *recursion*.

A recursive definition is a definition that refers to itself, while avoiding an infinite regress. We have seen many examples of this in previous chapters, and in the previous section, in the definition of `hword`.

Let us illustrate the principle once more for the case of the recursive screen picture. A screen picture is either (i) a tiny blob showing nothing, or (ii) a screen showing a screen picture. The case of the tiny blob showing nothing is the base case. The case of the screen showing a screen picture is the recursive case.

Definition by recursion is important in mathematics, programming, and linguistics. A recursive function is a function that calls itself, but without infinite regress. To ensure that the recursion will eventually stop, we have to give a base case, a case that can be computed without calling the function. Keep in mind that recursive definitions always have a base case. Recall the definitions of addition and multiplication in section 5.6. Here are Haskell versions, with the natural numbers represented as strings ending in `z`:

```
add :: String -> String -> String
add x "z" = x
add x ('s':y) = 's':(add x y)

mult :: String -> String -> String
mult x "z" = "z"
mult x ('s':y) = add (mult x y) x
```

This represents the multiplication of 8 and 3 as follows:

```
FP> mult "sssssssssz" "sssz"
"sssssssssssssssssssssssz"
```

The following multiplication table function gives another example of recursion.

```
table :: Int -> Int -> String
table x 0 =
 show 0 ++ " times " ++ show x ++ " is " ++ show 0
table x n =
 table x (n-1) ++ "\n" ++
 show n ++ " times " ++ show x ++ " is " ++ show (x*n)
```

Line ends are encoded as `\n`. This is the linefeed control character. To display the story on the screen we will use an output function that executes the linefeeds: `putStrLn`. To generate five rows of the multiplication table of 7, we say

```
FP> putStrLn (table 7 5)
```

Exercise 15.5 What happens if you ask for `putStrLn (table 7 (-1))`? Why?

Exercise 15.6 Why is the definition of 'GNU' as 'GNU's Not Unix' not a recursive definition?

15.5 List Types and List Comprehension

An important operation is :, which appends an element to a list.

```
FP> :t (:)
(:) :: a -> [a] -> [a]
```

The expression `(x:xs)` is used to denote the prototypical non-empty list. The *head* of `(x:xs)` is the element `x` of type `a`, the *tail* is the list `xs` of type `[a]`. According functions are defined in the Haskell Prelude as follows:

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

The underscores _ indicate anonymous variables that can be matched by anything of the right type. Thus, in `(x:_)`, the underscore _ matches any list, and in `(_:xs)`, the underscore _ matches any object. The important list patterns for pattern matching are:

- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,
- the list pattern `(x:xs)` matches any non-empty list.

These patterns are used again and again in recursive definitions over list types. It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.

Lists in computer science are an approximation of the mathematical notion of a set. The important differences are that multiplicity (whether a certain element occurs once or more than once) matters for lists, but not for sets, and that order of occurrence matters for lists but not for sets. Two sets that contain the same elements are the same, but not so for lists. The sets $\{1,2\}$, $\{2,1\}$ and $\{1,1,2\}$ are all identical, but the lists `[1,2]`, `[2,1]` and `[1,1,2]` are all different.

Lists can also be given by not enumerating all their elements but by indicating the range of elements: `[n..m]` is the list bounded below by `n` and above by `m`. For example, the list `[1..423]` are all numbers from 1 to 423, and the list `['g'..'s']` are all characters from g to s. Of course, it only works for objects that are ordered in a way that Haskell knows. In general, the ordered types are the types in class `Ord`. The data types

in this class are the types on which functions <= (less than or equal), >= (greater than or equal), < (less than) and > (greater than) are defined. Predefined instances of class `Ord` are numerical types and characters. (For more on type classes see Section 16.4.)

By use of `..`, we can also work with infinite lists, e.g. `[0..]` denotes the list of all positive natural numbers. When typing in `[0..]` at the interpreter system prompt, the interpreter will start printing the list of all natural numbers starting from 0. This display will go on until you abort it with *Ctrl-c* or your computer runs out of memory, for Haskell knows arbitrary size integers.

Infinite lists are where lazy evaluation matters. Since Haskell does not evaluate an argument unless it needs it, it can handle infinite lists as long as it has to compute only a finite amount of its elements. If you, for example, ask for `take 5 [0..]`, the interpreter will only consider the first five elements of the infinite list of natural numbers and give you `[0,1,2,3,4]`.

And there is yet another way to form lists, which is analogous to set comprehension. The latter says that if A is a set and P is a property, then $\{x \mid x \in A,\ P(x)\}$ is the set of all objects from A that satisfy P. *List comprehension* is the list counterpart of that; it is written as `[x | x <- A, P x]`. Here are some examples:

```
FP> [ n | n <- [0..10], odd n ]
[1,3,5,7,9]
FP> [ even n | n <- [0..10] ]
[True,False,True,False,True,False,True,False,True,False,True]
FP> [ square n | n <- [0..10] ]
[0,1,4,9,16,25,36,49,64,81,100]
FP> [ x ++ y | x <- ["talk","walk"], y <- ["s", "", "ed"] ]
["talks","talk","talked","walks","walk","walked"]
```

15.6 `map` and `filter`

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members. Thus, if `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : map f xs
```

For example, `map (+ 1) [0..9]` will add 1 to each number in `[0..9]`, so it will give the list of numbers from 1 to 10. Another example is the following one:

```
FP> map (++ "un") ["friendly","believable"]
["unfriendly","unbelievable"]
```

Accordingly, `map hword` will produce a list of truth values.

```
FP> map hword ["fish","and","chips"]
[True,False,True]
```

The `filter` function takes a property and a list, and returns the sublist of all list elements satisfying the property. It has the type `(a -> Bool) -> [a] -> [a]`. Here `a` again denotes an arbitrary type. Indeed, one can filtrate strings, or lists of integers, or lists of whatever, given a property of the right type: `String -> Bool` for strings, `Int -> Bool` for integers, and so on. Here is the definition of the `filter` function:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

A new programming element is the use of *guarded equations* by means of the Haskell condition operator `|`. A definition with a list of guarded equations such as

```
foo t | condition_1 = body_1
      | condition_2 = body_2
      | otherwise   = body_3
```

is read as follows:

- in case `condition_1` holds, `foo t` is by definition equal to `body_1`,
- in case `condition_1` does not hold but `condition_2` holds, `foo t` is by definition equal to `body_2`,
- and in case none of `condition_1` and `condition_2` hold, `foo t` is by definition equal to `body_3`.

Of course you can specify as many conditions as you want to.

The first line of the definition of `filter p (x:xs)` handles the case where the head of the list `(x:xs)` satisfies property `p`. The second line applies otherwise, i.e. it covers the case where the head of the list `(x:xs)` does not satisfy property `p`.

The `filter` function can be applied as follows:

```
FP> filter even [1..10]
[2,4,6,8,10]
FP> filter hword ["fish","and","chips"]
["fish","chips"]
```

15.7 Strings and Texts

Consider the following differences:

```
Prelude> "Hello World!"
"Hello World!"
Prelude> putStr "Hello World!"
Hello World!
Prelude> show "Hello World!"
"\"Hello World!\""
```

The double quotes `"` and `"` serve to indicate that what is between them is a string. The command `putStr` displays its string argument, but without the enclosing double quotes. The command `show` quotes its argument, by putting double quotes around it, and treating the inner quotes as part of the string. These inner quotes have changed function from string markers to characters. The double quote character is a so-called escaped `"`; it is rendered in Haskell as `\"`.

In Section 15.4 we saw Haskell's linefeed character `\n`. The following example shows how `putStr` executes linefeeds:

```
FP> putStr "Hello\nWorld!"
Hello
World!
```

Exercise 15.7 Try to figure out what the following Haskell program does, without actually running it. Next, save the program in a file `xxx.hs`, and run it as a script, by means of the command `runhugs xxx.hs` or `runghc xxx.hs`, from the command prompt, to check your guess. If you run a Haskell program as a script, its `main` function will get executed.

```
main = putStrLn (s ++ show s)
  where s = "main = putStrLn (s ++ show s) \n  where s = "
```

Our next example for manipulating strings is a program `reversal` for the reversal of a string, which is of type `String -> String`. The instruction for `reversal` explains how lists of characters are reversed. A useful description is the following:

- For reversing the empty string, you have to do nothing (the reversal of the empty string is the empty string).
- For reversing a string consisting of a first element x followed by a tail, first reverse the tail, and next put x at the end of the result of that.

We can render this string reversal instruction in Haskell as the following function.

```
reversal :: String -> String
reversal []    = []
reversal (x:t) = reversal t ++ [x]
```

You can try this out, as follows:

```
FP> reversal "Tarski"
"iksraT"
FP> reversal (reversal "Tarski")
"Tarski"
FP> (reversal . reversal) "Tarski"
"Tarski"
```

Next, we want to consider some examples of shallow text processing, using a piece of prose from Lewis Carroll's *The Game of Logic* as source material.

```
gameoflogic =
   "The world contains many THINGS (such as 'Buns', 'Babies', 'Beetles'.\n"
 ++ "'Battledores'. &c.);  and these Things possess many ATTRIBUTES\n"
 ++ "(such as 'baked', 'beautiful', 'black', 'broken', &c.:  in fact,\n"
 ++ "whatever can be 'attributed to', that is 'said to belong to', any\n"
 ++ "Thing, is an Attribute).  Whenever we wish to mention a Thing, we\n"
 ++ "use a SUBSTANTIVE:  when we wish to mention an Attribute, we use\n"
 ++ "an ADJECTIVE.  People have asked the question 'Can a Thing exist\n"
 ++ "without any Attributes belonging to it?'  It is a very puzzling\n"
 ++ "question, and I'm not going to try to answer it:  let us turn up\n"
 ++ "our noses, and treat it with contemptuous silence, as if it really\n"
 ++ "wasn't worth noticing.  But, if they put it the other way, and ask\n"
 ++ "'Can an Attribute exist without any Thing for it to belong to?', we\n"
 ++ "may say at once 'No:  no more than a Baby could go a railway-journey\n"
 ++ "with no one to take care of it!'  You never saw 'beautiful' floating\n"
 ++ "about in the air, or littered about on the floor, without any Thing\n"
 ++ "to BE beautiful, now did you?\n"
```

To split a string consisting of several lines separated by linefeed characters, use

```
lines gameoflogic
```

This function creates a list of strings.

We will now develop a simple counting tool, for counting characters and words in a piece of text. Instead of defining two different counting functions, one for words and one for text, it is more convenient to define a single polymorphic function that can count any data type for which equality tests are possible, i.e. which are in class `Eq`. The counting function we are after has type `Eq a => a -> [a] -> Int`. What this means is that for any type `a` that is an instance of the `Eq` class, the function has type `a -> [a] -> Int`.

We should be able to use `count` for counting the number of `'e'` characters in the string `"littered"`, for counting the number of `"Thing"` occurrences in the piece of Game of Logic text, and so on.

The following function defines `count`:

```
count :: Eq a => a -> [a] -> Int
count x []                  = 0
count x (y:ys) | x == y     = succ (count x ys)
               | otherwise = count x ys
```

Here are examples of using the `count` function, employing the predefined functions `toLower` for converting characters to lowercase, and `words` for splitting lines into words. You should check out their definitions in the Haskell Prelude.

```
FP> count 'e' gameoflogic
72
FP> [ count x (map toLower gameoflogic) | x <- ['a'..'w'] ]
[68,28,17,15,77,10,17,31,61,2,6,21,8,65,58,7,2,30,44,102,38,7,23]
FP> count "The" (words gameoflogic)
1
FP> count "the" (words gameoflogic)
4
FP> count "the" (words (map toLower gameoflogic))
5
```

Here is a function that calculates the average of a list of integers: the average of m and n is given by $\frac{m+n}{2}$, the average of a list of k integers $n_1, \ldots, n_k$ is given by $\frac{n_1+\cdots+n_k}{k}$. In general, averages are fractions, so the result type of `average` should not be `Int` but the Haskell data type for fractional numbers, which is `Rational`. There are predefined functions `sum` for the sum of a list of integers, and `length` for the length of a list. The Haskell operation for division `/` expects arguments of type `Rational` (or more precisely, of a type in the class `Fractional`, and `Rational` is in that class), so we need a conversion function for converting `Int`s into `Rational`s. This is done by `toRational`. The function `average` can now be written as:

```
average :: [Int] -> Rational
average [] = error "empty list"
average xs = toRational (sum xs) / toRational (length xs)
```

Haskell allows a call to the `error` operation in any definition. This is used to break off operation and issue an appropriate message when `average` is applied to an empty list. Note that `error` has a parameter of type `String`.

Exercise 15.8 Write a function to compute the average word length in Lewis Carroll's piece of prose. Be careful: Haskell takes `"attribute)."` and `"attribute"` to be different words. You can use `filter (`notElem` "?';:,.")` to get rid of the interpunction signs. The predefined function `length` (from the `List` module) gives the length of a list.

Suppose we want to check whether a list `xs1` is a prefix of a list `xs2`. Then the answer to the question `prefix xs1 xs2` should be either yes (true) or no (false), i.e. the type declaration for `prefix` should be:

```
prefix :: Eq a => [a] ->[a] -> Bool.
```

Prefixes of a list `ys` are defined as follows:

(1) `[]` is a prefix of `ys`.

(2) If `xs` is a prefix of `ys`, then `x:xs` is a prefix of `x:ys`.

(3) Nothing else is a prefix of `ys`.

Here is the code for `prefix` that implements this definition:

```
prefix :: Eq a => [a] -> [a] -> Bool
prefix []     ys     = True
prefix (x:xs) []     = False
prefix (x:xs) (y:ys) = (x==y) && prefix xs ys
```

Exercise 15.9 Write a function `sublist` that checks whether `xs1` is a sublist of `xs2`. The type declaration should be:

```
sublist ::  Ord a => [a] -> [a] -> Bool.
```

The sublists of an arbitrary list `ys` are given by:

(1) If `xs` is a prefix of `ys`, `xs` is a sublist of `ys`.

(2) If `ys` equals `y:ys'` and `xs` is a sublist of `ys'`, `xs` is a sublist of `ys`.

(3) Nothing else is a sublist of `ys`.

In order to remove duplicates from a list, the items in the list have to belong to a type in the `Eq` class, the class of data types for which `(==)` is defined. It is standard to call the function for duplicate removal `nub`:

```
nub :: Eq a => [a] -> [a]
nub []     = []
nub (x:xs) = x : nub (filter (\ y -> (y /= x)) xs)
```

The code below uses the built-in Haskell function `sort` (from the `List` module); `sort xs` sorts the list `xs` in increasing order. The code provides utilities for analysing Shakespeare's sonnets. `cnt sonnet18` will give a word count for the words occurring more than once in the sonnet.

```
preprocess :: String -> String
preprocess = (map toLower) . filter (`notElem` "?;:,.")

process :: String -> [String]
process = sort . nub . words

cnt :: String -> [(String,Int)]
cnt string = [ (x,n)| x <- (process.preprocess) string,
                 n <- [count x (words (preprocess string))],
                 n > 1                                    ]
```

Haskell has two ways to locally define auxiliary functions, the `where` and `let` constructions. The `where` construct we have seen in the `main` function given in Exercise 15.7. Here is an example of the `let` construction:

```
solveQdr :: (Float,Float,Float) -> (Float,Float)
solveQdr (a,b,c) = let d = b^2 - 4*a*c in
                   if d < 0 then error "no real solutions"
                   else
                     ((- b + sqrt d) / 2*a,
                      (- b - sqrt d) / 2*a)
```

The `let` construction uses the reserved keywords `let` and `in`.

Exercise 15.10 Give a reformulation of `solveQdr`, using `where` instead of `let`.

15.8 Further Reading

The Haskell homepage `http://www.haskell.org` provides links to everything about Haskell that you might want to know, including information on how to download the latest version and the online tutorial. Among the tutorials on Haskell that can be found on the internet, [DI07] and [HFP96] are recommended. The definitive reference for the language is [PJ03]. Recommended textbooks on functional programming in Haskell are [Bir98], [Tho99] and [Hut07], and the more recent [OSG08]. A textbook that bridges the gap between reasoning and functional programming is [DvE04]. Another textbook that combines a course in computational semantics with a introduction to Haskell is [EU10].

Chapter 16

More About Functional Programming

16.1 Type Polymorphism

First, we wrap the code of this chapter in a module:

```
module MFP where
```

One of Haskell's characteristics is a polymorphic type system. For example, the Haskell Prelude contains the following definition of the identity function:

```
id :: a -> a
id x = x
```

This function can be applied to objects of any type. Applied to an argument, it just returns that argument. Since the type of the argument does not matter, we say that the `id` function is *polymorphic*. In the type declaration of `id`, `a` is a type variable. The use of `a` indicates that any type will fit. So, instances of `id`'s type are `Bool -> Bool`, `Int -> Int`, `String -> String`, but also

```
(String -> Bool) -> (String -> Bool)
```

and so on.

```
MFP> id True
True
MFP> id 42
42
MFP> id "Jabberwocky"
"Jabberwocky"
MFP> (id hword) "Jabberwocky"
False
```

Type polymorphism, and the use of `a`, `b` as type variables, allow us to define polymorphic types for properties and relations. The polymorphic type of a property is `a -> Bool`. The polymorphic type of a relation is `a -> b -> Bool`. The polymorphic type of a relation over a single type is `a -> a -> Bool`.

If `a` is a type, `[a]` is the type of lists over `a`. Since it does not matter which type `a` is, `[a]` is a polymorphic type. The type of the concatenation function `(++)` illustrates this:

```
MFP> :t (++)
(++) :: [a] -> [a] -> [a]
```

The type indicates that `(++)` not only concatenates strings. It works for lists in general.

```
MFP> [2,3] ++ [4,7]
[2,3,4,7]
MFP> "Hello " ++ "world!"
"Hello world!"
```

16.2 Function Composition

Function composition is the application of one function after another (see Section A.5 of the appendix). Notation: $f \cdot g$ for the composition of the two functions f and g, i.e. for applying f after g.

Assume that we have a Dutch to English dictionary and an English to French dictionary. These can be viewed as a function g for translating Dutch words into English words, and a function f for translating English words into French words. E.g. $g(\textit{haas})$ will give *hare*, and $f(\textit{ant})$ will give *fourmi*. Now we can simply combine the two dictionaries into a Dutch to French dictionary by first looking up the meaning of a Dutch word in English, and then look up the French meaning of the result. Thus, if we want to know the French word for Dutch *haas*, we first find $g(\textit{haas}) = \textit{hare}$, and next, $f(\textit{hare}) = \textit{lièvre}$.

Composing two functions f and g means that f is applied to the result of g: $(f \cdot g)(x) = f(g(x))$. In Haskell, we use the operator `(.)` instead of $\cdot$ and write `f . g`. For that to make sense, the result type of `g` should equal the argument type of `f`, i.e. if `f :: b -> c` then `g :: a -> b`. Under this typing, the type of `f . g` is `a -> c`.

Thus, the operator `(.)` that composes two functions has the following type and definition:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

This is used, for example, in the Haskell Prelude to define `odd` as `not . even`.

16.3 Generalized 'and' and 'or', and Quantification

In Section 15.3 we saw the Haskell versions of `&&` and `||` of the logical operators conjunction and disjunction. The conjunction operator `&&` is generalized to lists of Booleans by the predefined function `and`.

```
and ::  [Bool] -> Bool
and []     = True
and (x:xs) = x && (and xs)
```

And the disjunction operator `||` is generalized to lists of Booleans by the predefined function `or`.

```
or ::  [Bool] -> Bool
or []     = False
or (x:xs) = x || (or xs)
```

These generalized Boolean operations are used for stating the definitions of the list quantification operators `any` and `all` (these are also predefined in the Haskell Prelude):

```
any, all  :: (a -> Bool) -> [a] -> Bool
any p = or  . map p
all p = and . map p
```

They can be used for checking whether all elements of a list satisfy a property, or any elements of the list, respectively.

```
MFP> all hword ["fish","and","chips"]
False
MFP> any hword ["fish","and","chips"]
True
```

16.4 Type Classes

Consider the function `(\ x y -> x /= y)`. At first sight, this is just a check for inequality. Let us check its type.

Exercise 16.1 Check the type of the function `(\ x y -> x /= y)` in Haskell. What do you expect? What do you get? Can you explain what you get?

Exercise 16.2 Is there a difference between `(\ x y -> x /= y)` and `(/=)`?

Exercise 16.3 Check the type of the function composition `all . (/=)`. If you were to give this function a name, what name would be appropriate?

Exercise 16.4 Check the type of the function composition `any . (==)`. If you were to give this function a name, what name would be appropriate?

The exercises show that the Haskell interpreter puts a constraint on the types of all functions that somehow involve equality or inequality. Why is this? Because not all types of things are testable for equality. Numbers and strings are examples of types that can be tested for equality, but functions generally are not.

Exercise 16.5 How would you go about to test two infinite strings for equality?

The predefined Haskell functions `elem` and `notElem` that check whether an object is element of a list are only defined for types that allow for equality tests. They have type `Eq a => a -> [a] -> Bool`. In this type specification, `Eq a =>` specifies `a` as a type in the `Eq` class, i.e. as a type for which equality (==) and inequality (/=) are defined. Here is what happens if you try to use `elem` with a function:

```
MFP> elem not [not]
ERROR - Cannot infer instance
*** Instance   : Eq (Bool -> Bool)
*** Expression : not `elem` [not]
```

Let us look at another type constraint. The predefined Haskell function `min` computes the minimum of two objects. If we check its type, we find that it is

```
min :: Ord a => a -> a -> a
```

The part `Ord a` indicates that `a` is constrained to be a type for which 'less than' and 'less than or equal' are defined (Haskell `(<)` and `(<=)`).

We can also have types that have more than one constraint. Constraints are combined by means of parentheses: `(Eq a,Ord a) => a` is the class of types for which equality and an ordering are defined, and so on.

Exercise 16.6 Use `min` to define a function `minList :: Ord a => [a] -> a` for computing the minimum of a non-empty list.

Exercise 16.7 Define a function `delete` that removes an occurrence of an object x from a list of objects in class `Ord`. If x does not occur in the list, the list remains unchanged. If x occurs more than once, only the first occurrence is deleted.

How would you need to change `delete` in order to delete every occurence of x?

Now we can define a function `srt` that sorts a list of objects (in class `Ord`) in order of increasing size, by means of the following algorithm:

- An empty list is already sorted.

- If a list is non-empty, we put its minimum in front of the result of sorting the list that results from removing its minimum.

Note that this sorting procedure can be viewed as a kind of dual to insertion sort (where you sort by going through a list and construct a new, sorted list by inserting each item from the old list at the correct position in the new list). The implementation is an exercise for you.

Exercise 16.8 Define a function `srt :: Ord a => [a] -> [a]` that implements the above. Use the function `minList` from Exercise 16.6.

Another type class, that we will use very often throughout the book, is `Show`. It contains all types that can be printed on the screen. We will see some usage of it in Section 16.6.

16.5 Identifiers in Haskell

As the reader may have noticed already from the code examples in this chapter, Haskell uses the distinction between upper and lower case to tell two kinds of identifiers apart:

- Variable identifiers are used to name functions. They have to start with a lower-case letter. Examples are `map`, `swedishVowels`, `list2OnePlacePred`, `foo_bar`.

- Constructor identifiers are used to name types and their inhabitants. Constructor identifiers have to start with an upper-case letter. An example of a type constructor is `Bool`, and examples of data constructors are `True`, `False`, the two constructors that make up the type `Bool`.

Functions are operations on data structures, constructors are the building blocks of the data structures themselves (trees, lists, Booleans, and so on).

Names of functions always start with lower-case letters, and may contain both upper- and lower-case letters, but also digits, underscores and the prime symbol `'`. The following *reserved keywords* have special meanings and cannot be used to name functions.

```
case  class   data     default  deriving  do      else
if    import  in       infix    infixl    infixr  instance
let   module  newtype  of       then      type    where
_
```

The character `_` at the beginning of a word is treated as a lower-case character; `_` all by itself is a reserved word for the wild card pattern that matches anything. And there is one more reserved keyword that is particular to Hugs: `forall`, for the definition of functions that take polymorphic arguments. See the Hugs documentation for further particulars.

16.6 User-defined Data Types

Besides the predefined data types like `Int`, `Char`, and `Bool`, you can define data types yourself. Imagine, for example, we want to implement our own data structure for subjects and predicates as sentence ingredients.

```
data Subject   = Frege | Tarski | Turing deriving Show
data Predicate = Wrote String            deriving Show
```

The data type `Predicate` is built from another data type; in fact, the data type declaration specifies `Wrote` as a function from strings to predicates:

```
MFP> :t Wrote
Wrote :: String -> Predicate
```

Note that we added `deriving Show` to the specification of our data types. This puts these data types in the `Show` class and tells Haskell to print its instances exactly like they are given. Another way would be to tell Haskell that these data types are instances of the `Show` class and then define a show function for them ourselves. This is useful if we want a data type to be displayed in a certain way. We will do that below for the data type `Sentence`.

We would like to use these data types to construct a more complex data type, say a `Sentence`. For this we use a data constructor `S`, as follows:

```
data Sentence  = S Subject Predicate

type Sentences = [Sentence]
```

The last line defines a type synonym. We added it simply to illustrate this abbreviated way of talking about types. This is exactly like `String` was defined as `[Char]`.

The specification of `Sentence` says that the data type has the form of a tree. But maybe we do not want it to get displayed like a tree. In order to display instances of the `Sentence` type in a more natural way, we can tell Haskell explicitly how to display a value of the type.

```
instance Show Sentence where
  show (S subj pred) = show subj ++ " " ++ show pred
```

We can specify functions for constructing predicates and sentences:

```
makeP :: String -> Predicate
makeP title = Wrote title

makeS :: Subject -> Predicate -> Sentence
makeS subj pred = S subj pred
```

Here they are in action:

```
MFP> makeS Frege (makeP "Begriffsschrift")
Frege Wrote "Begriffsschrift"
```

In the following chapters we will use Haskell to give implementations of many key definitions from the first and second part of this book. In the course of these implemenation efforts you will have occasion to use the knowledge gained from this and the previous chapter, and to exercise and extend your Haskell programming skills. These implementation exercises will drive home the message that formal details matter in the definition of logical concepts.

Chapter 17

Propositional Logic Implemented

17.1 A Data Type for Propositional Formulas

Representation for formulas of propositional logic. Note that the conjunction and disjunction operators `Cnj` and `Dsj` take lists of formulas as arguments.

```
module PropLogicI where
import Data.List
```

```
data Frm =  P String | Ng Frm | Cnj [Frm] | Dsj [Frm]
            deriving Eq

instance Show Frm where
 show (P name) = name
 show (Ng f)   = '-': show f
 show (Cnj fs) = '&': show fs
 show (Dsj fs) = 'v': show fs
```

Example formulas.

```
form1, form2 :: Frm
form1 = Cnj [P "p", Ng (P "p")]
form2 = Dsj [P "p1", P "p2", P "p3", P "p4"]
```

This is how they get displayed.

```
PropLogicI> form1
&[p,-p]
```

```
PropLogicI> form2
v[p1,p2,p3,p4]
```

Exercise 17.1 Implement a function `depth` for computing the depth of the parse tree of a formula. The type is `depth :: Frm -> Int`. The call `depth form1` should yield 2.

Getting the names of basic propositions from a formula:

```
propNames :: Frm -> [String]
propNames (P name) = [name]
propNames (Ng f)   = propNames f
propNames (Cnj fs) = (sort.nub.concat) (map propNames fs)
propNames (Dsj fs) = (sort.nub.concat) (map propNames fs)
```

This gives:

```
PropLogicI> propNames form1
["p"]
PropLogicI> propNames form2
["p1","p2","p3","p4"]
```

17.2 Evaluation and Propositional Consequence

A valuation for a propositional formula is a map from its variable names to the Booleans. Alternatively, it is a list of those variable names that map to `True`. So we can define a valuation as a list of variable names, i.e., as a list of strings:

```
type Val = [String]
```

The following function generates the list of all valuations over a set of names:

```
genVals :: [String] -> [Val]
genVals [] = [[]]
genVals (name:names) =
  genVals names ++ map (name:) (genVals names)
```

The list of all valuations for a propositional formula:

```
allVals :: Frm -> [Val]
allVals = genVals . propNames
```

This gives for `form1` a list of two valuations.

```
PropLogicI> allVals form1
[[],["p"]]
```

`form2` has four proposition letters, so there are $2^4 = 16$ valuations:

```
PropLogicI> allVals form2
[[],["p4"],["p3"],["p3","p4"],["p2"],["p2","p4"],["p2","p3"],
 ["p2","p3","p4"],["p1"],["p1","p4"],["p1","p3"],["p1","p3","p4"],
 ["p1","p2"],["p1","p2","p4"],["p1","p2","p3"],["p1","p2","p3","p4"]]
```

Evaluation of a formula with respect to a valuation.

```
eval :: Val -> Frm -> Bool
eval [] (P c)    = False
eval (i:xs) (P c)
     | c == i    = True
     | otherwise = eval xs (P c)
eval xs (Ng f)   = not (eval xs f)
eval xs (Cnj fs) = all (eval xs) fs
eval xs (Dsj fs) = any (eval xs) fs
```

Tautologies are formulas that evaluate to `True` for every valuation:

```
tautology :: Frm -> Bool
tautology f = all (\ v -> eval v f) (allVals f)
```

Contradictions are formulas that evaluate to `True` for no valuation:

```
contradiction :: Frm -> Bool
contradiction f = not (any  (\ v -> eval v f) (allVals f))
```

A propositional formula is called satisfiable if there is a propositional valuation that makes the formula true. Clearly, a formula is satisfiable if it is not a contradiction. Or we can give a direct implementation:

```
satisfiable :: Frm -> Bool
satisfiable f = any (\ v -> eval v f) (allVals f)
```

A propositional formula F_1 implies another formula F_1 if every valuation that satisfies F_1 also satisfies F_2. From this definition:

F_1 implies F_2 iff $F_1 \wedge \neg F_2$ is a contradiction.

The implementation is straightforward:

```
implies :: Frm -> Frm -> Bool
implies f1 f2 = contradiction (Cnj [f1, Ng f2])
```

Exercise 17.2 Extend the check for propositional implication to the case with a list of premisses. The type is `impliesL :: [Frm] -> Frm -> Bool`.

Exercise 17.3 Implement a check for logical equivalence between propositional formulas. The type is `propEquiv :: Frm -> Frm -> Bool`.

17.3 Propositional Logic in Update Form

The current state is a set of relevant valuations, and these sets are represented as lists, so we can define:

```
type State = [Val]
```

Updating the state with a formula gives the list of those valuations that satisfy the formula.

```
update :: State -> Frm -> State
update vals f = [ v | v <- vals, eval v f ]
```

Exercise 17.4 Complete the following definition:

```
tautologyUPD :: Frm -> Bool
tautologyUPD f = update (allVals f) f ==
```

Exercise 17.5 Complete the following definition:

```
contradictionUPD :: Frm -> Bool
contradictionUPD f = update (allVals f) f ==
```

Chapter 18

Model Checking for Predicate Logic

18.1 Variables, Predicates, Formulas

```
module MCPL where

import Data.List
```

Variables with names and indices:

```
type Name     = String
type Index    = [Int]
data Variable = Variable Name Index deriving (Eq,Ord)
```

Variables are the type of object that can be shown on the screen. We therefore put them in the `Show` class, and define a function for showing them:

```
instance Show Variable where
  show (Variable name [])  = name
  show (Variable name [i]) = name ++ show i
  show (Variable name is ) = name ++ showInts is
     where showInts []     = ""
           showInts [i]    = show i
           showInts (i:is) = show i ++ "_" ++ showInts is
```

Some example variables, that we will use later on:

```
x, y, z :: Variable
x = Variable "x" []
y = Variable "y" []
z = Variable "z" []
```

In the definition of formulas we let conjunction and disjunction operate on lists of formulas, and we use prefix notation for implications and equivalences. To start with, we will assume that all terms are variables, but we will modify this in the next section. It therefore make sense to define a so-called parametrized data type or abstract data type, where we use a type variable for the terms occurring in formulas. This gives:

```
data Formula a = Atom String [a]
               | Eq a a
               | Neg  (Formula a)
               | Impl (Formula a) (Formula a)
               | Equi (Formula a) (Formula a)
               | Conj [Formula a]
               | Disj [Formula a]
               | Forall Variable (Formula a)
               | Exists Variable (Formula a)
               deriving Eq
```

The construct `Atom String [a]` corresponds to predicate constants named by `String`, with a list of terms (or more general values of type `a`) as arguments. For example, $R(x, y)$ would be represented as `Atom "R" [x,y]`.

The function for showing formulas assumes that we use terms that are themselves showable. This is indicated by means of a type class constraint `Show a`.

```
instance Show a => Show (Formula a) where
  show (Atom s [])   = s
  show (Atom s xs)   = s ++ concat [ show xs ]
  show (Eq t1 t2)    = show t1 ++ "==" ++ show t2
  show (Neg form)    = '~' : (show form)
  show (Impl f1 f2)  = "(" ++ show f1 ++ "==>"
                           ++ show f2 ++ ")"
  show (Equi f1 f2)  = "(" ++ show f1 ++ "<=>"
                           ++ show f2 ++ ")"
  show (Conj [])     = "true"
  show (Conj fs)     = "conj" ++ concat [ show fs ]
  show (Disj [])     = "false"
  show (Disj fs)     = "disj" ++ concat [ show fs ]
  show (Forall v f)  = "A " ++  show v ++ (' ' : show f)
  show (Exists v f)  = "E " ++  show v ++ (' ' : show f)
```

Why we chose to display `Conj []` as `"true"` and `Disj []` as `"false"` will become clear when we look at their semantics in the next chapter on page **??**.

Here are some example formulas of type `Formula Variable`:

```
formula0 = Atom "R" [x,y]
formula1 = Forall x (Atom "R" [x,x])
formula2 = Forall x
             (Forall y
               (Impl (Atom "R" [x,y]) (Atom "R" [y,x])))
```

These get displayed as follows:

```
MCPL> formula0
R[x,y]
MCPL> formula1
Ax R[x,x]
MCPL> formula2
Ax Ay (R[x,y]==>R[y,x])
```

Formula `formula1` expresses that the R relation is reflexive, while `formula2` expresses that R is symmetric.

Exercise 18.1 Write a function `closedForm :: Formula Variable -> Bool` that checks whether a formula is closed. (Hint: first write a function that collects the list of free variables of a formula. The closed formulas are the ones with an empty free variable list.)

The formulas we have used so far are of type `Formula [Variable]`. It is simple to extend this to formulas with arbitrary terms (including function symbols),

Chapter 19

Public Announcement Logic Implemented

19.1 Agents, Propositions, Epistemic Models

We will now implement a so-called epistemic model checker, a program that can keep track of the epistemic effects caused by public announcements and public changes.

```
module PAL where

import Data.List
```

Let's keep our set of agents limited to five, and give them some easy-to-use names:

```
data Agent = A | B | C | D | E deriving (Eq,Ord,Enum)

a,alice, b,bob, c,carol, d,dave, e,ernie  :: Agent
a = A; alice = A
b = B; bob   = B
c = C; carol = C
d = D; dave  = D
e = E; ernie = E

instance Show Agent where
  show A = "a"; show B = "b";
  show C = "c"; show D = "d";
  show E = "e"
```

Basic propositions are p_i, q_i, r_i, where i is an appropriate index:

```
data Prop = P Int | Q Int | R Int deriving (Eq,Ord)

instance Show Prop where
  show (P 0) = "p"; show (P i) = "p" ++ show i
  show (Q 0) = "q"; show (Q i) = "q" ++ show i
  show (R 0) = "r"; show (R i) = "r" ++ show i
```

The implementation of epistemic models closely follows their definition. We add an extra parameter: a list of agents that occur in the model.

```
data EpistM state = Mo {
       dom       :: [state],
       agents    :: [Agent],
       valuation :: [(state,[Prop])],
       rels      :: [(Agent,state,state)],
       actual    :: [state]
                        } deriving (Eq,Show)
```

This uses Haskell *record syntax*. The data type for an epistemic model has five fields, and each of these fields can be accessed with an appropriate function. The record syntax makes for a more compact definition, because the access functions for the fields are part of the definition of the data type. To see how this works, consider the following example model.

```
exampleModel :: EpistM Integer
exampleModel =
 Mo [0..3]
    [a..c]
    [(0,[]),(1,[P 0]),(2,[Q 0]),(3,[P 0, Q 0])]
    ([ (a,x,x) | x <- [0..3] ] ++
     [ (b,x,x) | x <- [0..3] ] ++
     [ (c,x,y) | x <- [0..3], y <- [0..3] ])
    [1]
```

We can access the valuation of this model by means of

```
PAL> valuation exampleModel
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
```

and similarly for the domain, the list of agents, the relations and the actual worlds.

19.2 S5 Models

As was explained above, public announcements can be viewed as actions that change a model by removing the worlds that do not satisfy the contents of the announcement. Each announcement φ comes with a function $M \mapsto M \mid \varphi$. In case we add a set of actual worlds U to the model, the effect of $M \mid \varphi$ on U is given by:

$$U \mapsto \{w \in U \mid M \models_w \varphi\}.$$

The advantage of adding a set of actual worlds is that the effect of a public announcement with falsehood can now simply be encoded, as follows: φ is a falsehood in pointed model $M = (W, V, R, U)$ if

$$(W, V, R) \not\models_u \varphi$$

for all $u \in U$. The result of updating with a falsehood is an *inconsistent* pointed model, i.e. a pointed model of the form $(W', V', R', \emptyset)$.

We define relations as lists of pairs:

```
type Rel a = [(a,a)]
```

We also define list inclusion and the converse of a relation:

```
containedIn :: Eq a => [a] -> [a] -> Bool
containedIn xs ys = all (\ x -> elem x ys) xs

cnv :: Rel a -> Rel a
cnv r = [ (y,x) | (x,y) <- r ]
```

Here is the definition of an infix operator for relational composition (cf. page A-3):

```
infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s = nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Here are the three checks on accessibility relations that we need:

```
reflR :: Eq a => [a] -> Rel a -> Bool
reflR xs r = [(x,x) | x <- xs] `containedIn` r

symmR :: Eq a => Rel a -> Bool
symmR r = cnv r `containedIn` r

transR :: Eq a => Rel a -> Bool
transR r = (r @@ r) `containedIn` r
```

Together these allow us to express that a certain relation is an equivalence relation. In modal logic, such a relation is called an S5 relation:

```
isS5 :: Eq a => [a] -> Rel a -> Bool
isS5 xs r = reflR xs r && transR r && symmR r
```

The epistemic model instance `exampleModel` that was given above is in fact an example of a multi S5 model: each of its three accessibility relations is S5, i.e. an equivalence relation. Here is an auxiliary function for extracting each of the relations from an epistemic model:

```
rel :: Agent -> EpistM a -> Rel a
rel a model = [ (x,y) | (b,x,y) <- rels model, a == b ]
```

This gives:

```
PAL> rel a exampleModel
[(0,0),(1,1),(2,2),(3,3)]
PAL> rel b exampleModel
[(0,0),(1,1),(2,2),(3,3)]
PAL> rel c exampleModel
[(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3),
 (2,0),(2,1),(2,2),(2,3),(3,0),(3,1),(3,2),(3,3)]
PAL>  isS5 (dom exampleModel) (rel a exampleModel)
True
```

Every equivalence relation R on A corresponds to a partition on A, namely the set $\{[a]_R \mid a \in A\}$, where $[a]_R = \{b \in A \mid (a, b) \in R\}$. Here is an implementation:

```
rel2partition :: Ord a => [a] -> Rel a -> [[a]]
rel2partition []     r = []
rel2partition (x:xs) r = xclass : rel2partition (xs\\xclass) r
             where xclass = x : [ y | y <- xs, (x,y) `elem` r ]
```

The function `rel2partition` can be used to write a display function for S5 models that shows each accessibility relation as a partition:

```
showS5 :: (Ord a,Show a) => EpistM a -> [String]
showS5 m = show (dom m)
           : show (valuation m)
           : map show [ (a,(rel2partition (dom m)) (rel a m))
                                              | a <- (agents m)]
         ++ [show (actual m)]
```

Here `@` is used to introduce a shorthand or name for a data structure. The following is a function for example display:

```
displayS5 ::  (Ord a,Show a) => EpistM a -> IO()
displayS5 = putStrLn . unlines . showS5
```

It works as follows:

```
PAL> displayS5 exampleModel
[0,1,2,3]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0],[1],[2],[3]])
(b,[[0],[1],[2],[3]])
(c,[[0,1,2,3]])
[1]
```

19.3 Blissful Ignorance

Blissful ignorance is the state where you don't know anything, but you know also that there is no reason to worry, for you know that nobody knows anything.

A Kripke model where every agent from agent set A is in blissful ignorance about a (finite) set of propositions P, with $|P| = k$, looks as follows:

$$M = (W, V, R) \text{ where}$$

$$\begin{aligned} W &= \{0, \ldots, 2^k - 1\} \\ V &= \text{ any surjection in } W \to \mathcal{P}(P) \\ R &= \{x \xrightarrow{a} y \mid (x, y) \in W, a \in A\}. \end{aligned}$$

A surjection is a function which in onto (cf. page **??**). Note that V is in fact a bijection, for $|\mathcal{P}(P)| = 2^k = |W|$. Generating models for blissful ignorance is done with:

```
initM :: [Agent] -> [Prop] -> EpistM Integer
initM ags props = (Mo worlds ags val accs points)
   where worlds = [0..(2^k-1)]
         k      = length props
         val    = zip worlds (sortL (powerList props))
         accs   = [ (ag,st1,st2) | ag  <- ags,
                                   st1 <- worlds,
                                   st2 <- worlds  ]
         points = worlds
```

Here `powerList` is the list counterpart of power set:

```
powerList  :: [a] -> [[a]]
powerList  []     = [[]]
powerList  (x:xs) = (powerList xs) ++ (map (x:) (powerList xs))
```

The following function sorts lists by their length:

```
sortL :: Ord a => [[a]] -> [[a]]
sortL  = sortBy (\ xs ys -> if   length xs < length ys
                            then LT
                            else if   length xs > length ys
                                 then GT
                                 else compare xs ys)
```

This gives:

```
PAL> zip [0..2^3-1] (sortL (powerList [P 1,P 2,P 3]))
[(0,[]),(1,[p1]),(2,[p2]),(3,[p3]),(4,[p1,p2]),
 (5,[p1,p3]),(6,[p2,p3]),(7,[p1,p2,p3])]
```

19.4 General Knowledge and Common Knowledge

The general knowledge accessibility relation of a set of agents C is given by

$$\bigcup_{c \in C} R_c.$$

Exercise 19.1 Assume that R_a, the knowledge relation for agent a, and R_b, the knowledge relation for agent b are equivalence relations. Does it follow that the general knowledge relation for a and b is also an equivalence? Give a proof or a counterexample.

Computing the reflexive transitive closure works as follows. If A is finite, any binary relation R on A is finite as well. In particular, there will be k with $R^{k+1} \subseteq R^0 \cup \cdots \cup R^k$. Thus, in the finite case reflexive transitive closure can be computed by successively computing $\bigcup_{n \in \{0,..,k\}} R^n$ until $R^{k+1} \subseteq \bigcup_{n \in \{0,..,k\}} R^n$. In other words: the reflexive transitive closure of a relation R can be computed from I by repeated application of the operation

$$\lambda S \mapsto (S \cup (R \circ S)),$$

until the operation reaches a fixpoint.

A fixpoint of an operation f is an x for which $f(x) = x$. Least fixpoint calculation is done by means of the following function:

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

We used this to compute the reflexive transitive closure as follows:

```
rtc :: Ord a => [a] -> Rel a -> Rel a
rtc xs r = lfp (\ s -> (sort.nub) (s ++ (r@@s))) i
 where i = [ (x,x) | x <- xs ]
```

Here is an example:

```
PAL> rtc [1,2,3] [(1,2),(2,3)]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Now it is easy to also compute common knowledge, for the common knowledge relation for a group of agents C is the relation

$$(\bigcup_{c \in C} R_c)^*.$$

19.5 Formulas and Evaluation

We represent formulas in the following way:

```
data Form = Top
          | Prop Prop
          | Neg  Form
          | Conj [Form]
          | Disj [Form]
          | K    EE Form
          | PA   Form Form
          | PC   [(Prop,Form)] Form
          deriving (Eq,Ord)
```

Formulas of the form `K e f` express that knowledge `e` ensures truth of `f`, formulas of the form `PA f1 f2` express that public announcement `f1` makes `f2` true, and formulas of the form `PC s f` express that public change `s` makes `f` true.

Here is the data type of epistemic expressions:

```
data EE = Agent Agent
        | Test Form
        | Cmp  EE EE
        | Sum  [EE]
        | Star EE
        deriving (Eq,Ord)
```

Example formulas are `p` and `q`:

```
p = Prop (P 0)
q = Prop (Q 0)
```

Note the following type difference:

```
PAL> :t (P 0)
P 0 :: Prop
PAL> :t p
p :: Form
```

For better readability we define show functions for `Form` and `EE`:

```
instance Show Form where
  show Top           = "T"
  show (Prop p)      = show p
  show (Neg f)       = '-':(show f)
  show (Conj fs)     = '&': show fs
  show (Disj fs)     = 'v': show fs
  show (K e f)       = '[':show e ++"]"++show f
  show (PA f1 f2)    = '[':'!': show f1  ++"]"++show f2
  show (PC s f)      = show s ++ show f

instance Show EE where
  show (Agent a)   = show a
  show (Test f)    = '?': show f
  show (Cmp e1 e2) = show e1  ++ ";" ++ show e2
  show (Sum es)    = 'U': show es
  show (Star e)    = '(': show e  ++ ")*"
```

It is also useful to have an abbreviation for common knowledge formulas:

```
ck :: [Agent] -> Form -> Form
ck ags f = K (Star (Sum [Agent a | a <- ags])) f
```

Valuation lookup can be used to look up the valuation for a world in a model.

```
apply :: Eq a => [(a,b)] -> a -> b
apply [] _ = error "argument not in list"
apply ((x,z):xs) y | x == y    = z
                   | otherwise = apply xs y
```

If R is a binary relation on A and $a \in A$, then aR denotes the set

$$\{b \in A \mid Rab\}.$$

This is called the right section of a relation. The implementation:

```
rightS :: Ord a => a -> Rel a -> [a]
rightS x r =  (sort.nub) [ z | (y,z) <- r, x == y ]
```

Now we turn to the evaluation of formulas. We start with the Boolean cases:

```
isTrueAt :: Ord state => EpistM state -> state -> Form -> Bool
isTrueAt m w Top       = True
isTrueAt m w (Prop p)  = p `elem` (concat
                         [ ps | (w',ps) <- valuation m, w'==w])
isTrueAt m w (Neg  f)  = not (isTrueAt m w f)
isTrueAt m w (Conj fs) = and (map (isTrueAt m w) fs)
isTrueAt m w (Disj fs) = or  (map (isTrueAt m w) fs)
```

This is the epistemic case:

```
isTrueAt m w (K e f)   = and (map (flip (isTrueAt m) f)
                                  (rightS w (evalEE m e)))
```

Next we consider the public announcement case:

```
isTrueAt m w (PA f1 f2) = not (isTrueAt m  w f1)
                           ||   isTrueAt m' w f2
               where m' = upd_pa m f1
```

Finally, this is the public change case:

```
isTrueAt m w (PC s f) = isTrueAt (upd_pc m s) w f
```

The evaluation of epistemic expressions is defined as follows:

```
evalEE :: Ord state => EpistM state -> EE -> Rel state
evalEE m (Agent a)   = rel a m
evalEE m (Test f)    = [(w,w) | w <- dom m, isTrueAt m w f]
evalEE m (Cmp e1 e2) = (evalEE m e1) @@ (evalEE m e2)
evalEE m (Sum (es))  = (sort.nub) (concat (map (evalEE m) es))
evalEE m (Star e)    = rtc (dom m) (evalEE m e)
```

The state of bliss is evaluated like this:

```
test1 = isTrueAt (initM [a..c] [P 0]) 0
                 (ck [a..c] (Neg (K (Agent a) p)))
```

Now we can use the function `isTrueAt` to implement a function that checks for truth at all the actual states of an epistemic model:

```
isTrue :: Ord state => EpistM state -> Form -> Bool
isTrue m f = and [ isTrueAt m s f | s <- actual m ]
```

Here is another test of `initM`:

```
test2 = isTrue (initM [a..c] [P 0])
               (ck [a..c] (Neg (K (Agent a) p)))
```

Next we implement public announcement updates:

```
upd_pa :: Ord state => EpistM state -> Form -> EpistM state
upd_pa m@(Mo states  agents val  rels  actual)  f =
         (Mo states' agents val' rels' actual')
   where
     states' = [  s      |  s      <- states,
                            isTrueAt m s f    ]
     val'    = [ (s,p)   | (s,p)   <- val,
                            s `elem` states'  ]
     rels'   = [ (a,x,y) | (a,x,y) <- rels,
                            x `elem` states',
                            y `elem` states'  ]
     actual' = [  s      |  s      <- actual,
                            s `elem` states'  ]
```

For substitutions we define a type synonym:

```
type Subst = [(Prop,Form)]
```

Public change updates can then be implemented as follows:

```
upd_pc :: Ord state => EpistM state -> Subst -> EpistM state
upd_pc m@(Mo worlds agents val  acc points) subst =
         (Mo worlds agents val' acc points)
   where
     val' = [ (w,[p | p <- ps, isTrueAt m w (liftS subst p)])
            | w <- worlds ]
     ps   = (sort.nub) (concat (map snd val))
     liftS :: Subst -> Prop -> Form
     liftS []         p             = Prop p
     liftS ((x,z):xs) y | x == y    = z
                        | otherwise = liftS xs y
```

Here is an example model:

```
m0 = initM [a..c] [P 0,Q 0]
```

This gives:

```
PAL> displayS5 m0
[0,1,2,3]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
(c,[[0,1,2,3]])
[0,1,2,3]

PAL> displayS5 (upd_pa m0 (Disj [p,q]))
[1,2,3]
[(1,[p]),(2,[q]),(3,[p,q])]
(a,[[1,2,3]])
(b,[[1,2,3]])
(c,[[1,2,3]])
[1,2,3]
```

It is useful to be able to convert any type of state list to `[0..]`:

```
convert :: Eq state => EpistM state -> EpistM Integer
convert (Mo states  agents val  rels  actual) =
         Mo states' agents val' rels' actual'
   where
     states' = map f states
     val'    = map (\ (x,y)   -> (f x,y))     val
     rels'   = map (\ (x,y,z) -> (x,f y,f z)) rels
     actual' = map f actual
     f       = apply (zip states [0..])
```

This gives:

```
PAL> (displayS5.convert) (upd_pa m0 (Disj [p,q]))
[0,1,2]
[(0,[p]),(1,[q]),(2,[p,q])]
(a,[[0,1,2]])
(b,[[0,1,2]])
(c,[[0,1,2]])
[0,1,2]
```

19.6 Generated Submodels

In an epistemic model with a set of designated points, we are only interested in the set of those worlds that are accessible via some path from any of the designated worlds. This is called the *generated submodel*. Here is an implementation:

```
gsm :: Ord state => EpistM state ->  EpistM state
gsm (Mo states  ags val  rel  points) =
     Mo states' ags val' rel' points
   where
     states' = closure rel ags points
     val'    = [(s,ps)    | (s,ps)    <- val,
                             s  `elem` states'  ]
     rel'    = [(ag,s,s') | (ag,s,s') <- rel,
                             s  `elem` states',
                             s' `elem` states'  ]
```

This uses the closure of a state list, given a relation and a list of agents:

```
closure ::  Ord state => [(Agent,state,state)] ->
                         [Agent] -> [state] -> [state]
closure rel agents xs = lfp f xs where
 f = \ ys -> (nub.sort) (ys ++ (expand rel agents ys))
```

The expansion of a relation R given a state set S and a set of agents B is given by $\{t \mid s \xrightarrow{b} t \in R, s \in S, b \in B\}$. Here is an implementation:

```
expand :: Ord state => [(Agent,state,state)] ->
                       [Agent] -> [state] -> [state]
expand rel agents ys = (nub.sort.concat)
  [ alternatives rel ag state | ag    <- agents,
                                state <- ys      ]
```

The epistemic alternatives for agent a in state s are the states in sR_a (i.e. the states reachable through R_a from s):

```
alternatives :: Eq state => [(Agent,state,state)] ->
                            Agent -> state -> [state]
alternatives rel ag current = [ s' | (a,s,s') <- rel,
                                     a == ag, s == current ]
```

This ends the implementation of the epistemic model checker.

19.7 The Muddy Children Example

To show how this can be used, we demonstrate a check of a muddy children scenario. We model the case where there are four children, Alice, Bob, Carol and Dave, where Bob, Carol and Dave are the muddy ones.

We use propositions p_1, p_2, p_3, p_4 to express that the first, second, third, or fourth child is muddy:

```
p1, p2, p3, p4 :: Form
p1 = Prop (P 1); p2 = Prop (P 2)
p3 = Prop (P 3); p4 = Prop (P 4)
```

The initial muddy model has 16 worlds; one for each muddiness possibility. Agent a cannot distinguish worlds that differ only in the value of p_1, for Alice cannot see whether she herself is muddy. Agent b cannot distinguish worlds that differ only in the value of p_2,

for Bob cannot see whether he himself is muddy. Agent c cannot distinguish worlds that differ only in the value of p_3, for Carol cannot see whether she herself is muddy. Agent d cannot distinguish worlds that differ only in the value of p_4, for Dave cannot see whether he himself is muddy. This information completely determines the accessibility relation.

```
computeAcc :: Agent -> [Integer] -> [Prop] -> [(Integer,[Prop])]
                    -> [(Agent,Integer,Integer)]
computeAcc ag states props val =
 [ (ag,x,y) | x <- states, y <- states,
              apply val x \\ props == apply val y \\ props ]
```

The actual world is the world where p_1 is false and p_2, p_3, p_4 are true. This yields the following definition for the initial epistemic model.

```
initMuddy :: EpistM Integer
initMuddy = Mo states
               [a..d]
               valuation
               (computeAcc a states [P 1] valuation ++
                computeAcc b states [P 2] valuation ++
                computeAcc c states [P 3] valuation ++
                computeAcc d states [P 4] valuation)
               [7]
  where states    = [0..15]
        valuation = zip states (powerList [P 1,P 2,P 3,P 4])
```

Here is a check that this is correct:

```
PAL> displayS5 initMuddy
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[(0,[]),(1,[p4]),(2,[p3]),(3,[p3,p4]),(4,[p2]),(5,[p2,p4]),
 (6,[p2,p3]),(7,[p2,p3,p4]),(8,[p1]),(9,[p1,p4]),(10,[p1,p3]),
 (11,[p1,p3,p4]),(12,[p1,p2]),(13,[p1,p2,p4]),(14,[p1,p2,p3]),
 (15,[p1,p2,p3,p4])]
(a,[[0,8],[1,9],[2,10],[3,11],[4,12],[5,13],[6,14],[7,15]])
(b,[[0,4],[1,5],[2,6],[3,7],[8,12],[9,13],[10,14],[11,15]])
(c,[[0,2],[1,3],[4,6],[5,7],[8,10],[9,11],[12,14],[13,15]])
(d,[[0,1],[2,3],[4,5],[6,7],[8,9],[10,11],[12,13],[14,15]])
[7]
```

The statement of the father that at least one child is muddy should rule out one of these worlds:

```
m1 = convert (upd_pa initMuddy (Disj [p1,p2,p3,p4]))
```

This yields:

```
PAL> displayS5 m1
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]
[(0,[p4]),(1,[p3]),(2,[p3,p4]),(3,[p2]),(4,[p2,p4]),
 (5,[p2,p3]),(6,[p2,p3,p4]),(7,[p1]),(8,[p1,p4]),
 (9,[p1,p3]),(10,[p1,p3,p4]),(11,[p1,p2]),(12,[p1,p2,p4]),
 (13,[p1,p2,p3]),(14,[p1,p2,p3,p4])]
(a,[[0,8],[1,9],[2,10],[3,11],[4,12],[5,13],[6,14],[7]])
(b,[[0,4],[1,5],[2,6],[3],[7,11],[8,12],[9,13],[10,14]])
(c,[[0,2],[1],[3,5],[4,6],[7,9],[8,10],[11,13],[12,14]])
(d,[[0],[1,2],[3,4],[5,6],[7,8],[9,10],[11,12],[13,14]])
[6]
```

The following formulas express that the children know their states:

```
aK =  Disj [K (Agent a) p1, K (Agent a) (Neg p1)]
bK =  Disj [K (Agent b) p2, K (Agent b) (Neg p2)]
cK =  Disj [K (Agent c) p3, K (Agent c) (Neg p3)]
dK =  Disj [K (Agent d) p4, K (Agent d) (Neg p4)]
```

In the first round, they all say that they do not know their state:

```
m2 = convert (upd_pa m1 (Conj [Neg aK,Neg bK,Neg cK,Neg dK]))
```

Here is the result:

```
PAL> displayS5 m2
[0,1,2,3,4,5,6,7,8,9,10]
[(0,[p3,p4]),(1,[p2,p4]),(2,[p2,p3]),(3,[p2,p3,p4]),
 (4,[p1,p4]),(5,[p1,p3]),(6,[p1,p3,p4]),(7,[p1,p2]),
 (8,[p1,p2,p4]),(9,[p1,p2,p3]),(10,[p1,p2,p3,p4])]
(a,[[0,6],[1,8],[2,9],[3,10],[4],[5],[7]])
(b,[[0,3],[1],[2],[4,8],[5,9],[6,10],[7]])
(c,[[0],[1,3],[2],[4,6],[5],[7,9],[8,10]])
(d,[[0],[1],[2,3],[4],[5,6],[7,8],[9,10]])
[3]
```

In the second round, again they all say that they do not know their state:

```
m3 = convert (upd_pa m2 (Conj [Neg aK,Neg bK,Neg cK,Neg dK]))
```

The result is:

```
PAL> displayS5 m3
[0,1,2,3,4]
[(0,[p2,p3,p4]),(1,[p1,p3,p4]),(2,[p1,p2,p4]),
 (3,[p1,p2,p3]),(4,[p1,p2,p3,p4])]
(a,[[0,4],[1],[2],[3]])
(b,[[0],[1,4],[2],[3]])
(c,[[0],[1],[2,4],[3]])
(d,[[0],[1],[2],[3,4]])
[0]
```

In the third round, a still does not know, but b, c, d know their state.

```
m4 = convert (upd_pa m3 (Conj [Neg aK, bK, cK, dK]))
```

After this round, everything is known:

```
PAL> displayS5 m4
[0]
[(0,[p2,p3,p4])]
(a,[[0]])
(b,[[0]])
(c,[[0]])
(d,[[0]])
[0]
```

Exercise 19.2 Carry out the model checking process for the wise men puzzle.

Exercise 19.3 Implement translations that remove public announcement operators from the language. The types are `tr1 :: Form -> Form` and `tr2 :: EE -> EE`. Next, perform a number of checks to test whether the translations respect the semantics.

Exercise 19.4 Implement translations that remove public change operators from the language. The types are `tr3 :: Form -> Form` and `tr4 :: EE -> EE`. Next, perform a number of checks to test whether the translations respect the semantics.

Appendices

Appendix A

Sets, Relations and Functions

Summary

This chapter explains the basics of formal set notation, and gives an introduction to relations and functions. The chapter ends with a short account of the principle of proof by mathematical induction.

A.1 Sets and Set Notation

Many mathematical notions — some say all mathematical notions — can be defined in terms of the fundamental concept of a set. This is good reason for starting with some basic set theory.

A set is a collection of definite, distinct objects. Examples are the set of colours of the Dutch flag, or the set of letters of the Greek alphabet. Yet another example is the set of even natural numbers greater than seven. And so on.

The elements of a set are also called its *members*. To indicate that a is an element of a set A we write $a \in A$. To deny that a is an element of a set A we write $a \notin A$. The symbol $\in$ is the symbol for membership.

The elements of a set can be anything: words, colours, people, numbers. The elements of a set can also themselves be sets. The set consisting of the set of even natural numbers and the set of odd natural numbers is an example. This set has two elements; each of these elements has itself an infinite number of elements.

To check whether two sets are the same one has to check that they have the same elements. The fact that membership is all there is to set identity, or that sets are fully determined by their members, is called the *principle of extensionality*. It follows that to check that two sets A and B are identical, one has to check two things:

- does it hold that every element a of A is also an element of B, and
- does it hold that every element b of B is also an element of A?

To specify a set, there are several methods: give a list of its members, as in 'the set having the numbers 1, 2 and 3 as its only members', give some kind of semantic description, as in 'the set of colours of the Dutch flag', or separate out a set from a larger set by means of a suitable restriction. This last method is called the method of *set comprehension*. Here is an example: the odd natural numbers are the natural numbers with the property that division by 2 leaves a remainder of 1. We can express this by means of the pattern $2n + 1$, as follows:

$$O = \{2n + 1 \mid n \in \mathbb{N}\}.$$

The braces are also used to list the members of a finite set:

$$D = \{\text{red}, \text{white}, \text{blue}\}.$$

Mentioning a set element more than once does not make a difference. The set

$$\{\text{white}, \text{blue}, \text{white}, \text{red}\}$$

is identical to the set D, for it has the same members.

Another way of specifying sets is by means of operations on other sets. An example is the following definition of the odd natural numbers:

$$E = \mathbb{N} - O.$$

Here $\mathbb{N}-O$ is the set of all elements of $\mathbb{N}$ that are not members of O. Equivalent definitions are the following:

$$E = \{n \in \mathbb{N} \mid n \notin O\}$$

or

$$E = \{2n \mid n \in \mathbb{N}\}.$$

Some important sets have special names. $\mathbb{N}$ is an example. Another example is $\mathbb{Z}$, for the set of integer numbers. Yet another example is the set without any members. Because of the principle of extensionality there can be only one such set. It is called $\emptyset$ or the empty set.

If every member of a set A is also a member of set B we say that A is a subset of B, written as $A \subseteq B$. If $A \subseteq B$ and $B \subseteq A$ then it follows by the principle of extensionality that A and B are the same set. Conversely, if $A = B$ then it follows by definition that $A \subseteq B$ and $B \subseteq A$.

Exercise A.1 Explain why $\emptyset \subseteq A$ holds for every set A.

Exercise A.2 Explain the difference between $\emptyset$ and $\{\emptyset\}$.

The *complement* of a set A, with respect to some fixed universe, or: domain, U with $A \subseteq U$, is the set consisting of all objects in U that are not elements of A. The complement set is written as $\overline{A}$. It is defined as the set $\{x \mid x \in U, x \notin A\}$. For example, if we take U to be the set $\mathbb{N}$ of natural numbers, then the set of even numbers is the complement of the set of odd numbers and vice versa.

A.2 Relations

By a relation we mean a meaningful link between people, things, objects, whatever. Usually, it is quite important what kind of relationship we have in mind.

Formally, we can describe a *relation* between two sets A and B as a collection of ordered pairs (a, b) such that $a \in A$ and $b \in B$. An ordered pair is, as the name already gives away, a collection of two distinguishable objects, in which the order plays a role. E.g., we use $(\textit{Bill}, \textit{Hillary})$ to indicate the ordered pair that has *Bill* as its first element and *Hillary* as its second element. This is different from the pair $(\textit{Hillary}, \textit{Bill})$ where Bill plays second fiddle.

The notation for the set of all ordered pairs with their first element taken from A and their second element taken from B is $A \times B$. This is called the *Cartesian product* of A and B. A relation between A and B is a subset of $A \times B$.

The Cartesian product of the sets $A = \{a, b, \ldots, h\}$ and $B = \{1, 2, \ldots, 8\}$, for example, is the set

$$A \times B = \{(a,1), (a,2), \ldots, (b,1), (b,2), \ldots, (h,1), (h,2), \ldots, (h,8)\}.$$

This is the set of positions on a chess board. And if we multiply the set of chess colours $C = \{\text{White}, \text{Black}\}$ with the set of chess figures,

$$F = \{\text{King}, \text{Queen}, \text{Knight}, \text{Rook}, \text{Bishop}, \text{Pawn}\},$$

we get the set of chess pieces $C \times F$. If we multiply this set with the set of chess positions, we get the set of piece positions on the board, with $(\text{White}, \text{King}, (e,1))$ indicating that the white king occupies square $e1$. To get the set of moves on a chess board, take $((C \times F) \times ((A \times B) \times (A \times B)))$, and read $(\text{White}, \text{King}, ((e,1), (f,2))$ as 'white king moves from $e1$ to $f2$', but bear in mind that not all moves in $((C \times F) \times ((A \times B) \times (A \times B)))$ are legal in the game.

$A \times A$ is sometimes also denoted by A^2. Similarly for $A \times A \times A$ and A^3, and so on.

As an example of a relation as a set of ordered pairs consider the relation of authorship between a set A of authors and a set B of books. This relation associates with every author the book(s) he or she wrote.

Sets of ordered pairs are called binary relations. We can easily generalize this to sets of triples, to get so-called ternary relations, to sets of quadruples, and so on. An example of a ternary relation is that of borrowing something from someone. This relation consists of triples, or: 3-tuples, (a, b, c), where a is the borrower, b is the owner, and c is the thing borrowed. In general, an n-ary relation is a set of n-tuples (ordered sequences of n objects). We use A^n for the set of all n-tuples with all elements taken from A.

Unary relations are called *properties*. A property can be represented as a set, namely the set that contains all entities having the property. For example, the property of being divisible by 3, considered as a property of integer numbers, corresponds to the set $\{\ldots, -9, -6, -3, 0, 3, 6, 9, \ldots\}$.

An important operation on binary relations is composition. If R and S are binary relations on a set U, i.e. $R \subseteq U^2$ and $S \subseteq U^2$, then the composition of R and S, notation

$R \circ S$, is the set of pairs (x, y) such that there is some z with $(x, z) \in R$ and $(z, y) \in S$. E.g., the composition of $\{(1, 2), (2, 3)\}$ and $\{(2, 4), (2, 5)\}$ is $\{(1, 4), (1, 5)\}$.

Exercise A.3 What is the composition of $\{(n, n + 2) \mid n \in \mathbb{N}\}$ with itself?

Another operation on binary relations is converse. If R is a binary relation, then its converse (or: inverse) is the relation given by $R^{\smile} = \{(y, x) \mid (x, y) \in R\}$. The converse of the relation 'greater than' on the natural numbers is the relation 'smaller than' on the natural numbers. If a binary relation has the property that $R^{\smile} \subseteq R$ then R is called *symmetric*.

Exercise A.4 Show that it follows from $R^{\smile} \subseteq R$ that $R = R^{\smile}$.

If U is a set, then the relation $I = \{(x, x) \mid x \in U\}$ is called the identity relation on U. If a relation R on U has the property that $I \subseteq R$, i.e. if every element of U stands in relation R to itself, then R is called *reflexive*. The relation $\leq$ ('less than or equal') on the natural numbers is reflexive, the relation $<$ ('less than') is not. The relation $A^2 - I$ is the set of all pairs $(x, y) \in A^2$ with $x \neq y$. If A is the set $\{a, bc\}$, then $A^2 - I$ gives the following relation:

$$\{(a, b), (a, c), (b, a), (b, c), (c, a)(c, b)\}.$$

A relation R is called *transitive* if it holds for all x, y, z that if $(x, y) \in R$ and $(y, z) \in R$, then also $(x, z) \in R$. To say that the relation of friendship is transitive boils down to saying that it holds for anyone that the friends of their friends are their friends.

Exercise A.5 Which of the following relations are transitive?

(1) $\{(1, 2), (2, 3), (3, 4)\}$

(2) $\{(1, 2), (2, 3), (3, 4), (1, 3), (2, 4)\}$

(3) $\{(1, 2), (2, 3), (3, 4), (1, 3), (2, 4), (1, 4)\}$

(4) $\{(1, 2), (2, 1)\}$

(5) $\{(1, 1), (2, 2)\}$

The next exercise shows that transitivity can be expressed in terms of relational composition.

Exercise A.6 Check that a relation R is transitive if and only if it holds that $R \circ R \subseteq R$.

Exercise A.7 Can you give an example of a transitive relation R for which $R \circ R = R$ does not hold?

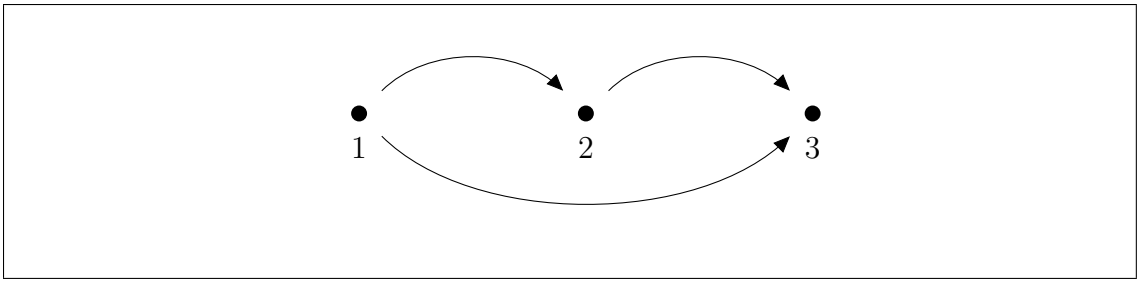
A.3 Back and Forth Between Sets and Pictures

A domain of discourse with a number of 1-place and 2-place predicates on is in fact a set of entities with certain designated subsets (the 1-place predicates) and designated sets of pairs of entities (the 2-place predicates).

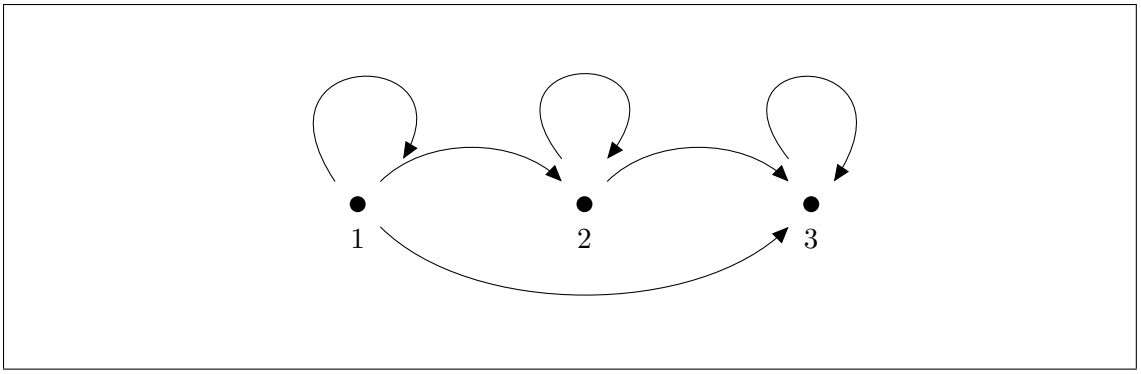
Relations are sets of pairs, and it is useful to acquire the skill to mentally go back and forth between sets-of-pairs representation and picture representation. Take the following simple example of a relation on the set $\{1, 2, 3\}$.

$$\{(1, 2), (1, 3), (2, 3)\}. \tag{A.1}$$

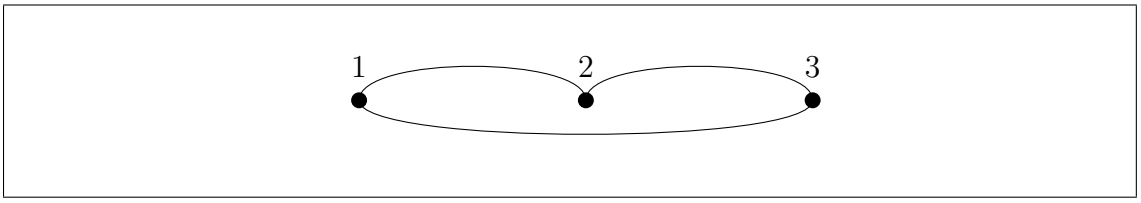
Here is the corresponding picture:



Exercise A.8 Give the set of pairs that constitutes the relation of the following picture:



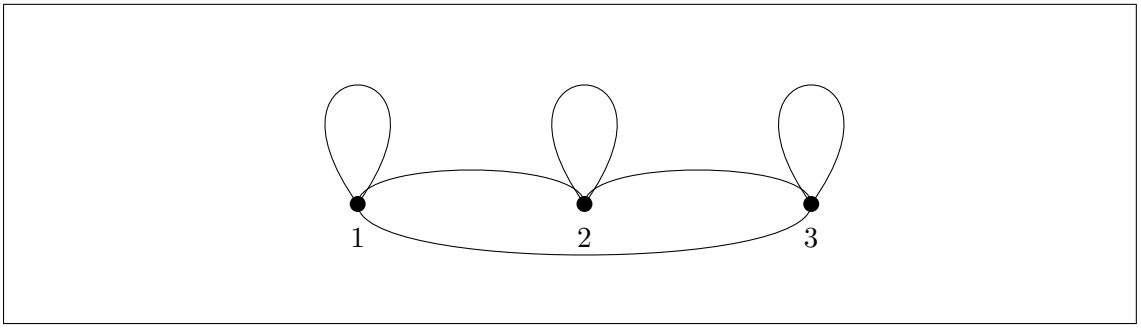
For another example, consider the picture:



No arrowheads are drawn, which indicates that the pictured relation is symmetric. Here is the representation of the same relation as a set of pairs:

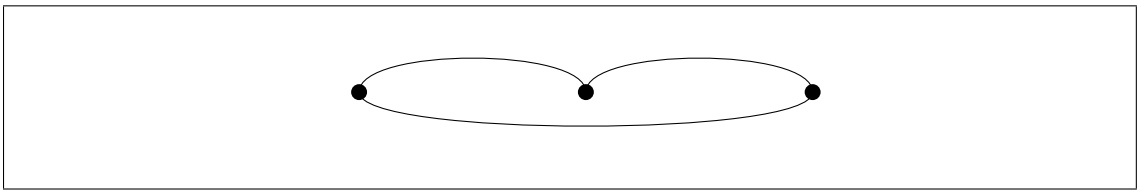
$$\{(1, 2), (2, 1), (1, 3), (3, 1), (2, 3), (3, 2)\}.$$

Exercise A.9 Give the representation of the pictured relation as a set of pairs:



A.4 Relational Properties

Talking about pictures with predicate logic is very useful to develop a clear view of what relational properties the predicate logical formulas express. Predicate logic is very precise, and it takes practice to get used to this precision. Consider the following picture.

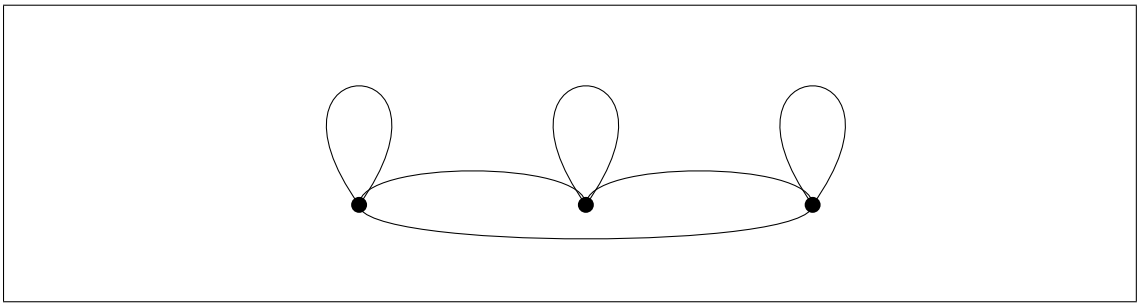


A relation is *transitive* if from the facts that there are links from x to y and a link from y to z it follows that there is a link from x to y. Here is a formula for this:

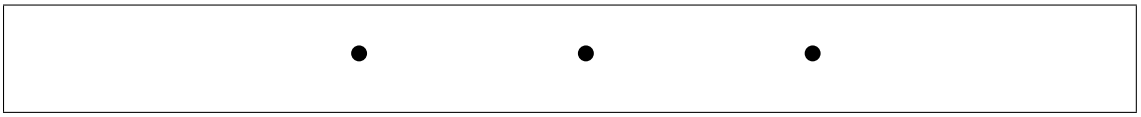
$$\forall x \forall y \forall z((Rxy \wedge Ryz) \rightarrow Rxz). \tag{A.2}$$

Let us check whether the link relation in the last picture is transitive. It may seem at first sight that it is, for what transitivity expresses is that if you can go from x to z by first taking an R step from x to y and next another R step from y to z, between, then there is also a direct R step from x to y. This seems indeed to be the case in the picture. But there is a snag. In reasoning like this, we assume that the three points x, y and z are all *different*. But this is not what the formula says. Take any two different points in the picture. Surely there is a link from the first point to the second. But the linking relation is symmetric: it goes in both directions. Therefore there also is a link from the second point back to the first. But this means that the first point has to be R related to itself, and it isn't. So the relation in the picture is not transitive after all.

Can we also come up with a picture of three points with a symmetric linking relation, where the relation is transitive? Yes, there are several possibilities. Here is the first one:



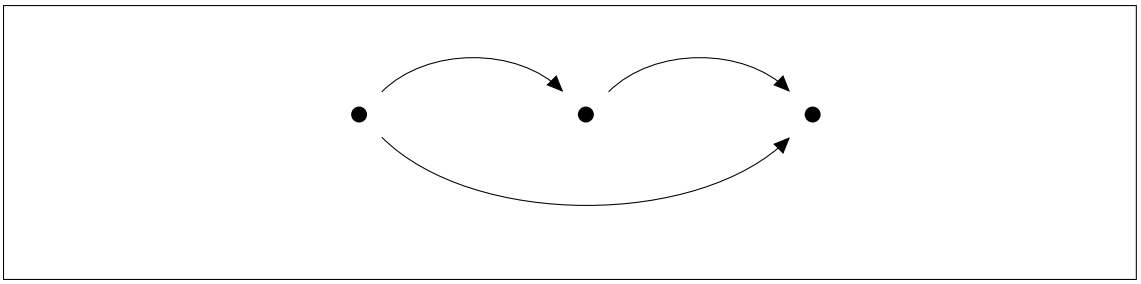
But there is another possibility. Take the following picture:



This is a picture where the link relation is empty. There are no links, so it trivially holds that if one can get from a point to a point via two links, then one can also get there with a single link. So the empty relation is transitive.

Exercise A.10 Give all the transitive link relations on a domain consisting of three individuals, on the assumption that the link relation is symmetric. We have already seen two examples: the empty relation (no points are linked) and the total relation (all points are linked). What are the other possibilities? Draw pictures!

The relations in the pictures above were all symmetric: links were the same in both directions. The following picture with arrows gives a relation that is not symmetric. We need the arrows, for now the directions of the links matter:



Again we use R to refer to the binary relation in the picture. Again we can ask if the relation of the picture is transitive. This time the answer is 'yes'. If we can get from x to y with two $\longrightarrow$ steps, then we can also get from x to y with a single step.

Not only is the relation in the picture not symmetric, but something stronger holds:

$$\forall x \forall y (Rxy \rightarrow \neg Ryx). \tag{A.3}$$

Formula (A.3) expresses that the relation R is *asymmetric*.

Exercise A.11 Give an example of a binary relation on a domain of three objects that it is neither symmetric nor asymmetric.

The relation in the current picture also has another property, called *irreflexivity*:

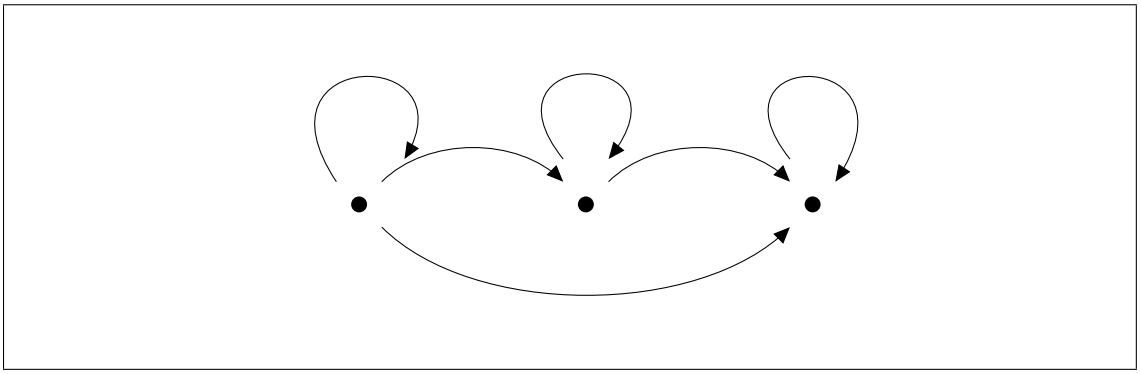
$$\forall x \neg Rxx. \tag{A.4}$$

This expresses that the R relation does not have self loops. We say: the relation is *irreflexive*

The dual to *irreflexivity* is the property of having *all* self loops. This is called *reflexivity*:

$$\forall x Rxx. \tag{A.5}$$

Here is an example of a reflexive relation:



Exercise A.12 Show that any asymmetric relation has to be irreflexive. (Hint: assume that a relation is asymmetric, and suppose it contains a loop (x, x). Why is this impossible?)

A binary relation R is called an *equivalence relation* if R has the following three properties: (i) R is reflexive, (ii) R is symmetric, (iii) R is transitive.

Exercise A.13 Give all equivalence relations on a domain consisting of three objects. Draw pictures!

Exercise A.14 Consider the three predicate logical sentences (4.20), (A.2) and (A.5), These sentences together express that a certain binary relation R is an equivalence relation: symmetric, transitive and reflexive. Show that none of these sentences is semantically entailed by the other ones by choosing for each pair of sentences a model (situation) that makes these two sentences true but makes the third sentence false. In other words: find three examples of binary relations, each satisfying just two of the properties in the list (4.20), (A.2) and (A.5). This shows, essentially, that the definition of being an equivalence cannot be simplified (why?).

Exercise A.15 Consider the following predicate logical formulas:

- $\forall x \forall y (Rxy \rightarrow \neg Ryx)$ (R is asymmetric)
- $\forall x \exists y Rxy$ (R is serial)
- $\forall x \forall y \forall z ((Rxy \wedge Ryz) \rightarrow Rxz)$ (R is transitive).

Take any situation with a non-empty domain of discourse, with a binary relation on it. Show: if the three formulas are true of this situation, then the domain of discourse must be *infinite*. (Hint: start with a domain consisting of a single individual d_1. Then by seriality there has to be an R-successor to d_1. Suppose we take d_1 as its own R-successor. Then this would get us in conflict with we are in conflict with asymmetry, by Exercise **??**. So there has to be a d_2 with (d_1, d_2) in R. And so on ...)

Exercise A.16 Consider again the three properties of asymmetry, seriality and transitivity of the previous exercise.

(1) Give a picture of a finite situation with a relation R that is asymmetric and serial but not transitive.

(2) Give a picture of a finite situation with a relation R that is serial and transitive but not asymmetric.

(3) Give a picture of a finite situation with a relation R that is transitive and asymmetric but not serial.

A.5 Functions

Functions are relations with the following special property: for any (a, b) and (a, c) in the relation it has to hold that b and c are equal. Thus a *function* from a set A (called *domain*) to a set B (called *range*) is a relation between A and B such that for each $a \in A$ there is one and only one associated $b \in B$. In other words, a function is a mechanism that maps an input value to a uniquely determined output value. Looking at the relation *author of* from above, it is immediately clear that it is not a function, because the input *Michael Ende* is not mapped to a unique output but is related to more than one element from the set B of books.

Functions are an important kind of relations, because they allow us to express the concept of *dependence*. For example, we know that the gravitational potential energy of a wrecking ball depends on its mass and the height we elevated it to, and this dependence is most easily expressed in a functional relation.

Functions can be viewed from different angles. On the one hand, they can be seen as sets of data, represented as a collection of pairs of input and output values. This tells us something about the behaviour of a function, i.e. what input is mapped to which output.

The function converting temperatures from Kelvin to Celsius can be seen as a set of pairs $\{(0, -273.15), \ldots\}$, and the function converting temperatures from Celsius to Fahrenheit as a set $\{(-273.15, -459.67), \ldots\}$. Determining the output of the function, given some input, simply corresponds to a table lookup. Any function can be viewed as a – possibly infinite – database table. This is called the extensional view of functions. Another way to look at functions is as *instructions for computation*. This is called the intensional view of functions. In the case of temperature conversion the intensional view is more convenient than the extensional view, for the function mapping Kelvin to Celsius

can easily be specified as a simple subtraction

$$x \mapsto x - 273.15$$

This is read as 'an input x is mapped to x minus 273.15'. Similarly, the function from Celsius to Fahrenheit can be given by

$$x \mapsto x \times \frac{9}{5} + 32$$

For example, if we have a temperature of 37 degrees Celsius and want to convert it to Fahrenheit, we replace x by 37 and compute the outcome by multiplying it with $\frac{9}{5}$ and then adding 32.

$$37 \times \frac{9}{5} + 32 \rightarrow 66.6 + 32 \rightarrow 98.6$$

The example shows that the intensional view of functions can be made precise by representing the function as an expression, and specifying the principles for simplifying (or: rewriting) such functional expressions. Rewriting functional expressions is a form of simplification where part of an expression is replaced by something simpler, until we arrive at an expression that cannot be simplified (or: reduced) any further. This rewriting corresponds to the computation of a function. For example, the function converting Celsius to Fahrenheit applied to the input 37 is the expression $37 \times \frac{9}{5} + 32$. This expression denotes the output, and at the same time it shows how to arrive at this output: First, $37 \times \frac{9}{5}$ is rewritten to 66.6, according to the rewriting rules for multiplication. The result of this simplification is $66.6 + 32$, which is then rewritten to 98.6, in accordance with the rewriting rules for addition.

Functions can be composed, as follows. Let g be the function that converts from Kelvin to Celsius, and let f be the function that converts from Celsius to Fahrenheit. Then $f \cdot g$ is the function that converts from Kelvin to Fahrenheit, and that works as follows. First convert from Kelvin to Celsius, then take the result and convert this to Fahrenheit. It should be clear from this explanation that $f \cdot g$ is defined by

$$x \mapsto f(g(x)),$$

which corresponds to

$$x \mapsto (x - 273.15) \times \frac{9}{5} + 32$$

Exercise A.17 The successor function $s : \mathbb{N} \rightarrow \mathbb{N}$ on the natural numbers is given by $n \mapsto n + 1$. What is the composition of s with itself?

A special function which is simple yet very useful is the *characteristic function* of a set. The characteristic function of subset A of some universe (or: domain) U is a function that maps all members of A to the truth-value **True** and all elements of U that are not members of A to **False**. E.g. the function representing the property of being divisible by 3, on the domain of integers, would map the numbers

$$\ldots, -9, -6, -3, 0, 3, 6, 9, \ldots$$

to **True**, and all other integers to **False**. Characteristic functions characterize membership of a set. Since we specified relations as sets, this means we can represent every relation as a characteristic function.

Exercise A.18 $\leq$ is a binary relation on the natural numbers. What is the corresponding characteristic function?

Exercise A.19 Let $f\colon A \to B$ be a function. Show that the relation $R \subseteq A^2$ given by $(x, y) \in R$ if and only if $f(x) = f(y)$ is an equivalence relation (reflexive, transitive and symmetric) on A.

A.6 Recursion and Induction

A recursive definition is a recipe for constructing objects from a finite number of ingredients in a finite number of ways. An example is the following recursive definition of natural numbers:

- 0 is a natural number.
- adding 1 to a natural number n gives a new natural number $n + 1$.
- nothing else is a natural number.

This recipe gives rise to an important method for proving things: proof by *mathematical induction*.

As an example, we prove the fact about natural numbers that the sum of the first n odd natural numbers equals n^2. For example $1 + 3 = 2^2$, $1 + 3 + 5 + 7 = 4^2$, and so on. More formally and generally, we have for all natural numbers n:

$$\sum_{k=0}^{n-1}(2k + 1) = n^2.$$

Here is a proof of this fact by mathematical induction.

Basis. For $n = 0$, we have $\Sigma_{k=0}^{0}(2k + 1) = 1 = 1^2$, so for this case the statement holds.

Induction step. We assume the statement holds for some particular natural number n and we show that it also holds for $n + 1$. So assume $\sum_{k=0}^{n-1}(2k + 1) = n^2$. This is the **induction hypothesis**. We have to show: $\sum_{k=0}^{n}(2k + 1) = (n + 1)^2$. Indeed,

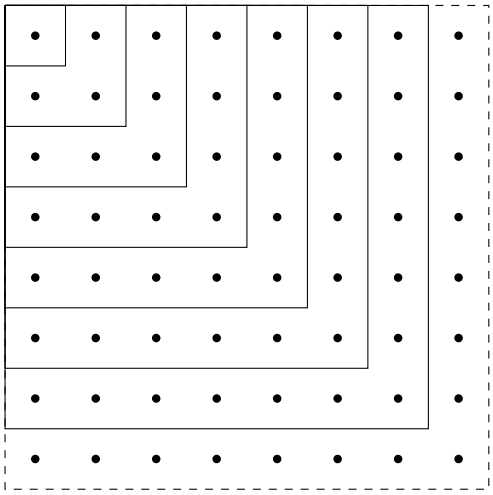
$$\sum_{k=0}^{n}(2k + 1) = \sum_{k=0}^{n-1}(2k + 1) + 2n + 1.$$

Now use the induction hypothesis to see that this is equal to $n^2 + 2n + 1$, which in turn equals $(n + 1)^2$. Therefore we have:

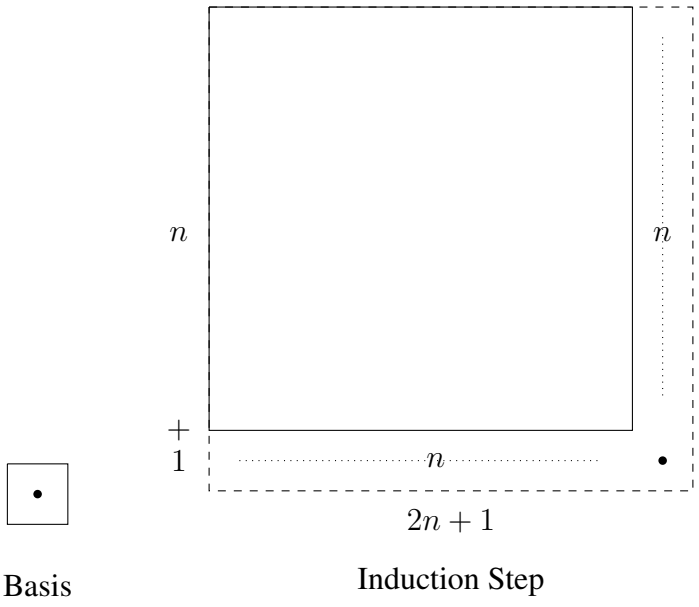
$$\sum_{k=0}^{n}(2k + 1) = \sum_{k=0}^{n-1}(2k + 1) + 2n + 1 \stackrel{ih}{=} n^2 + 2n + 1 = (n + 1)^2.$$

The equality $\stackrel{ih}{=}$ is the step where the induction hypothesis was used. We have checked two cases: the case 0 and the case $n+1$. By the recursive definition of natural numbers, we have covered *all* cases, for these are the two possible shapes of natural numbers. So we have proved the statement for all natural numbers n.

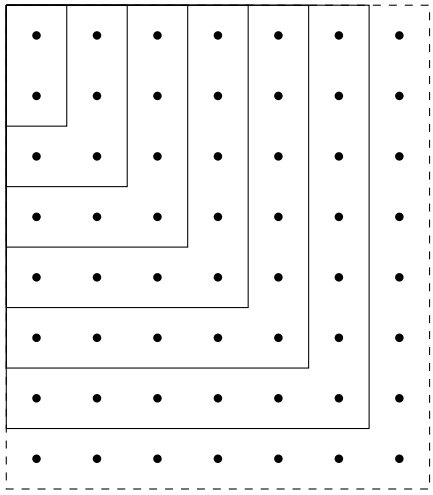
The procedure of proof by mathematical induction does not help to *find* interesting patterns, but once such a pattern is found it is very helpful to *check* whether the pattern really holds. So how can one find a pattern like $\sum_{k=0}^{n-1}(2k+1) = n^2$ in the first place? By imagining the following way to build up a square with side n:



Such a picture is what the ancient Greeks called a *gnomon* ("thing by which one knows"). The structure of the inductive proof can now be pictured as follows:



Exercise A.20 Consider the following *gnomon*:



What does this suggest for the sum of the first n even numbers? Give a form for $\sum_{k=0}^{n} 2k$, and prove with induction that your form is correct.

Appendix B

Solutions to the Exercises

Solutions to Exercises from Chapter 2

Exercise 2.3 on page 2-3: You are given the information that p-or-q and (not-p)-or-r. What is the strongest valid conclusion you can draw?

$$\left\{\begin{array}{c} p\,q\,r \\ p\,q\,\overline{r} \\ p\,\overline{q}\,r \\ p\,\overline{q}\,\overline{r} \\ \overline{p}\,q\,r \\ \overline{p}\,q\,\overline{r} \\ \overline{p}\,\overline{q}\,r \\ \overline{p}\,\overline{q}\,\overline{r} \end{array}\right\} \quad \begin{array}{c} p \vee q \\ \Longrightarrow \end{array} \quad \left\{\begin{array}{c} p\,q\,r \\ p\,q\,\overline{r} \\ p\,\overline{q}\,r \\ p\,\overline{q}\,\overline{r} \\ \overline{p}\,q\,r \\ \overline{p}\,q\,\overline{r} \end{array}\right\} \quad \begin{array}{c} \neg p \vee r \\ \Longrightarrow \end{array} \quad \left\{\begin{array}{c} p\,q\,r \\ p\,\overline{q}\,r \\ \overline{p}\,q\,r \\ \overline{p}\,q\,\overline{r} \end{array}\right\}$$

Any valid conclusion has to be true in the set of remaining alternatives $\left\{\begin{array}{c} p\,q\,r \\ p\,\overline{q}\,r \\ \overline{p}\,q\,r \\ \overline{p}\,q\,\overline{r} \end{array}\right\}$. If it is also false in the set of eliminated alternatives $\left\{\begin{array}{c} \overline{p}\,\overline{q}\,\overline{r} \\ \overline{p}\,\overline{q}\,r \\ p\,\overline{q}\,\overline{r} \\ p\,q\,\overline{r} \end{array}\right\}$ then it is among the strongest ones. For instance $p \vee q$ is a valid conclusion but it is not strong enough because it is also true, for instance, in $p\,q\,\overline{r}$. The formula $(p \vee q) \wedge (\neg p \vee r)$ is among the strongest conclusions that you can draw from the given information (and so is any formula equivalent to it).

Exercise 2.1 on page 2-3: Consider the case where there are three facts that you are interested in. You wake up, you open your eyes, and you ask yourself three things: "Have I overslept?", "Is it raining?", "Are there traffic jams on the road to work?". To find out about the first question, you have to check your alarm clock, to find about the second

you have to look out of the window, and to find out about the third you have to listen to the traffic info on the radio. We can represent these possible facts with three basic propositions, p, q and r, with p expressing "I have overslept", q expressing "It is raining", and r expressing "There are traffic jams." Suppose you know nothing yet about the truth of your three facts. What is the space of possibilities?

$$\left\{ \begin{array}{ll} p\,q\,r & p\,q\,\overline{r} \\ p\,\overline{q}\,r & p\,\overline{q}\,\overline{r} \\ \overline{p}\,q\,r & \overline{p}\,q\,\overline{r} \\ \overline{p}\,\overline{q}\,r & \overline{p}\,\overline{q}\,\overline{r} \end{array} \right\}$$

Exercise 2.2 on page 2-3: (Continued from previous exercise.) Now you check your alarm clock, and find out that you have not overslept. What happens to your space of possibilities?

$$\left\{ \begin{array}{ll} p\,q\,r & p\,q\,\overline{r} \\ p\,\overline{q}\,r & p\,\overline{q}\,\overline{r} \\ \overline{p}\,q\,r & \overline{p}\,q\,\overline{r} \\ \overline{p}\,\overline{q}\,r & \overline{p}\,\overline{q}\,\overline{r} \end{array} \right\} \quad \stackrel{\neg p}{\Longrightarrow} \quad \left\{ \begin{array}{ll} p\,q\,r & p\,q\,\overline{r} \\ p\,\overline{q}\,r & p\,\overline{q}\,\overline{r} \end{array} \right\}$$

Exercise 2.6 on page 2-7:

- I will only go to school if I get a cookie now:

  $$(p \rightarrow q) \wedge (q \rightarrow p)$$

  where p ="I get a cooky now" and q ="I will go to school".

- John and Mary are running:

  $$p \wedge q$$

  where p ="John is running" and q ="Mary is running".

- A foreign national is entitled to social security if he has legal employment or if he has had such less than three years ago, unless he is currently also employed abroad:

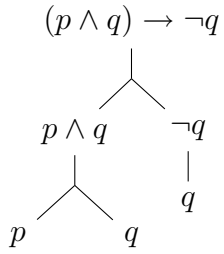
  $$((p \vee q) \wedge \neg r) \rightarrow s$$

  where p ="A foreign national has legal employment", q ="A foreign national has had legal employment less then three years ago", r ="A foreign national is currently also employed abroad" and s ="A foreign national is entitled to social security".

Exercise 2.7 on page 2-7: Only the first one.

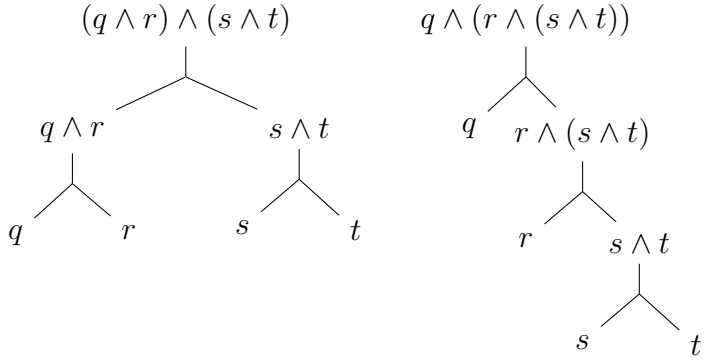
Exercise 2.8 on page 2-7: Construct a tree for the following formulae:

$(p \wedge q) \rightarrow \neg q$:



$q \wedge r \wedge s \wedge t$ (draw all possible trees; and does it matter?)

Two possible trees are depicted below, you should build the remaining ones and check that the order doesn't matter in either construction (in the sense that the logical meaning of this particular formula is invarinat under different construction orders). This is not, however, a general result: sometimes the order in the construction tree changes the logical meaning (truth value) of composed formulae.



Exercise 2.19 on page 2-15:

(a)

$$\left\{ \begin{array}{l} q\,r \\ q\,\overline{r} \\ \overline{q}\,r \\ \overline{q}\,\overline{r} \end{array} \right\} \quad \begin{array}{c} \neg(q \wedge r) \\ \Longrightarrow \end{array} \quad \left\{ \begin{array}{l} q\,\overline{r} \\ \overline{q}\,r \\ \overline{q}\,\overline{r} \end{array} \right\} \quad \begin{array}{c} q \\ \Longrightarrow \end{array} \quad \left\{ \; q\,\overline{r} \; \right\}$$

All the valuations making the premisses true also make the conclusion true. We can also see that updating with the conclusion is redundant.

$$\left\{\begin{array}{l} p\,q\,r \\ p\,q\,\overline{r} \\ p\,\overline{q}\,r \\ p\,\overline{q}\,\overline{r} \\ \overline{p}\,q\,r \\ \overline{p}\,q\,\overline{r} \\ \overline{p}\,\overline{q}\,r \\ \overline{p}\,\overline{q}\,\overline{r} \end{array}\right\} \quad \begin{array}{c} \neg p \vee \neg q \vee r \\ \Longrightarrow \end{array} \quad \left\{\begin{array}{l} p\,q\,r \\ p\,\overline{q}\,r \\ p\,\overline{q}\,\overline{r} \\ \overline{p}\,q\,r \\ \overline{p}\,q\,\overline{r} \\ \overline{p}\,\overline{q}\,r \\ \overline{p}\,\overline{q}\,\overline{r} \end{array}\right\} \quad \begin{array}{c} q \vee r \\ \Longrightarrow \end{array} \quad \left\{\begin{array}{l} p\,q\,r \\ p\,\overline{q}\,r \\ \overline{p}\,q\,r \\ \overline{p}\,q\,\overline{r} \\ \overline{p}\,\overline{q}\,r \end{array}\right\} \quad \begin{array}{c} p \\ \Longrightarrow \end{array} \quad \left\{\begin{array}{l} p\,q\,r \\ p\,\overline{q}\,r \end{array}\right\}$$

We can see that updating with the conclusion has no further effect, hence the consequence relation is valid.

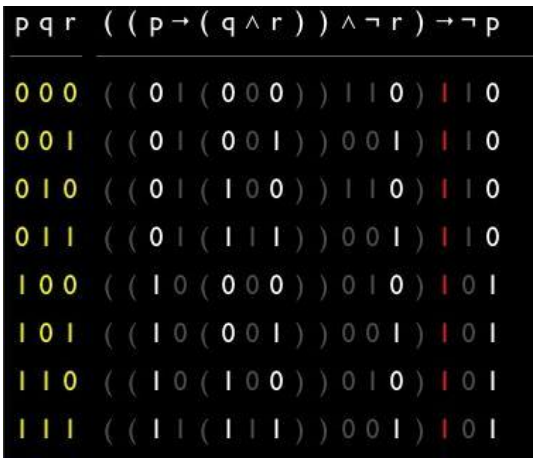
Exercise 2.11 on page 2-11:

p	q	r	$\neg p$	$\neg q$	$q \vee r$	$\neg p \rightarrow (q \vee r)$	$p \wedge r$	$p \vee r$
0	0	0	1	1	0	0	0	0
0	0	1	1	1	1	1	0	1
0	1	0	1	0	1	1	0	0
0	1	1	1	0	1	1	0	1
1	0	0	0	1	0	1	0	1
1	0	1	0	1	1	1	1	1
1	1	0	0	0	1	1	0	1
1	1	1	0	0	1	1	1	1

You can check by inspecting the rows of the table that $p \wedge r$ is not a valid consequence and $p \vee r$ is.

Exercise 2.17 on page 2-14:

You can check item (a) with the following table:



You might want to use the *Truth Tabulator* applet to build the truth table for item (b). The address is:

`http://staff.science.uva.nl/~jaspars/AUC/apps/javascript/proptab`

Exercise 2.27 on page 2-29:

The following table shows how starting from p and q we can obtain new truth functions applying $\mid$ to previous combinations.

		s_1:	s_2:	s_3:	s_4:	s_5:	s_6:	s_7:	s_8:	s_9:
p	q	$p \mid p$	$p \mid q$	$q \mid q$	$p \mid s_1$	$p \mid s_2$	$q \mid s_1$	$s_1 \mid s_3$	$s_2 \mid s_4$	$s_2 \mid s_7$
1	1	0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	0	0
0	1	1	1	0	1	1	0	1	0	0
0	0	1	1	1	1	1	1	0	0	1

Note that there is no unique way these combinations can be obtained, for instance, s_8 could have been obtained also as $s_2 \mid s_2$. Note that $\varphi \mid \varphi$ defines $\neg\varphi$. Using this observation, the remaining five truth functions can be obtained as negations: for instance, $s_{10} = s_9 \mid s_9$.

Exercise 2.28 on page 2-30: In three ways: (1) yes,no,yes; (2) yes,no,no; (3) no,yes,no.

Exercise 2.29 on page 2-30:

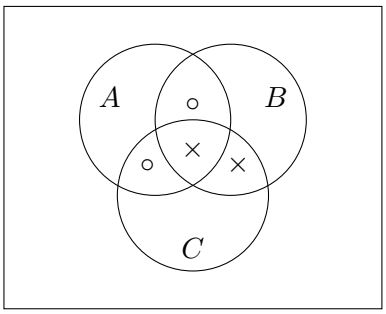
(1) You have to fill in 11 entries.

(2) In the worst case, you have to fill 88 cells in the truth table, which is eight times the number of logical symbols.

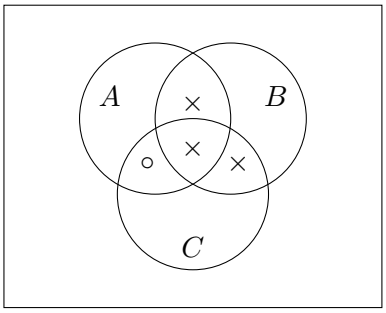
Solutions to Exercises from Chapter 3

Exercise 3.1 on page 3-4: The middle term is A.

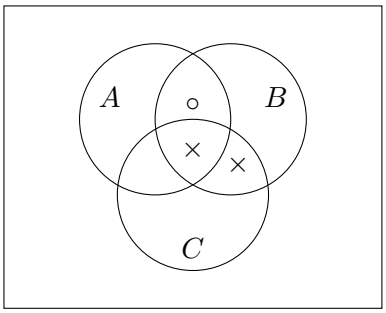
Exercise 3.2 on page 3-14: The syllogistic pattern is not valid because it is possible to build a counter-example in which the premises are both true but the conclusion is false. This can be seen in the following diagram, where A = 'philosopher', B = 'barbarian' and C = 'greek':



Exercise 3.3 on page 3-15: The syllogistic pattern is not valid because it is possible to build a counter-example in which the premises are both true but the conclusion is false. This can be seen in the following diagram, where A = 'philosopher', B = 'barbarian' and C = 'greek':



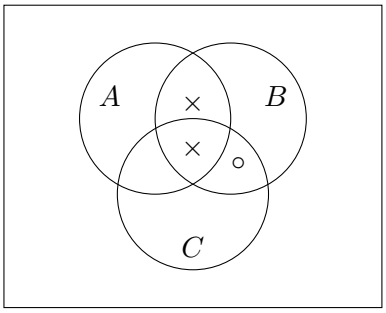
Exercise 3.4 on page 3-15: The syllogistic pattern is valid because after we update with the information given in the premises it is impossible for the conclusion to be false. This can be illustated by the following diagram, where A = 'philosopher', B = 'barbarian' and C = 'greek':



Exercise 3.5 on page 3-15: To obtain the required modification of the method for checking syllogistic validity with the all quantifiers read with existential import we have to represent explicitly the implicit assumption that we are not thinking with empty terms. In this way even universally quantified sentences, which give empty regions in the diagram, might also implicitly give the information that another region is not empty (otherwise some of the terms used in the reasoning will become empty, agains the assumption). To do this in a systematic manner we have to add three existential sentences to the given premises: one for each of the terms in the syllogism. This will correspond to putting a $\circ$ symbol in each circle representing a term while respecting the other premisses. Only after this step will our method further proceed towards checking the effects of updating with the information given in the conclusion. In this way, the following example turns out to be a valid patern of syllogistic reasoning: 'All men are mortal, all greeks are men, therefore, some greeks are mortal'. (this syllogistic mode, mnemotechnically called Barbari, was considered valid by Aristotle as it is the subordinate mode of Barbara). In general, the inference from 'All A are B' to 'Some A are B' is valid under existential import.

Exercise **??** on page **??**: (1) $\neg p_{ABC} \wedge \neg p_{BC}$ (2) $p_{AC} \vee p_{ABC}$ (3) $p_{AC} \vee p_A$.

Exercise **??** on page **??**: (1) The first syllogism is invalid and the second one is valid. (2) The following diagram illustrates the validity of the right syllogism:



An alternative representation that makes the stages in the update explicit is the following:

	A	B	C
AB	AC	BC	ABC

$\Downarrow \neg p_{AB} \wedge \neg p_{ABC}$

	A	B	C
× AB	AC	BC	× ABC

$\Downarrow p_{BC} \vee p_{ABC}$

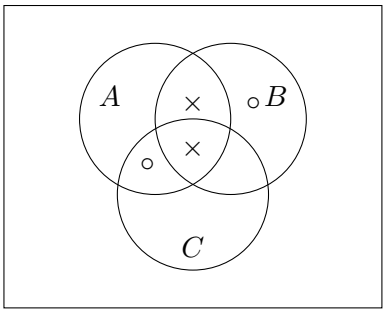
	A	○ B	○ C
× AB	AC	○ BC	× ABC

$\Downarrow p_{BC} \vee p_{C}$

	A	○ B	○ C
× AB	AC	○ BC	× ABC

We can see that updating with the conclusion does not add any new information that was not already present in the premises, hence the syllogism is valid.

(3) The following diagram ilustrates how a counterexample to the left syllogism can be constructed:



An alternative way to represent the process of finding a counterexample lists the update at each stage of the reasoning process:

	A	B	C
AB	AC	BC	ABC

$$\Downarrow \neg p_{AB} \wedge \neg p_{ABC}$$

	A	B	C
× AB	AC	BC	× ABC

$$\Downarrow p_B \vee p_{AB}$$

	A	∘ B	C
× AB	AC	BC	× ABC

$$\Downarrow p_{ABC} \vee p_{AC}$$

	∘ A	∘ B	∘ C
× AB	∘ AC	BC	× ABC

We can see that updating with the conclusion adds some new information which was not already contained in the information from the premisses, hence the inference amplifies the infomation and makes the syllogisatic reasoning invalid.

(4) For the left syllogism we have:

The initial space of valuations is a state of ignorance about the 8 propositional valuations describing the regions of a Venn diagram, this is a big number: $2^8 = 256$. Using blanks to condense the set gives us a very simple representation:

$p_\emptyset$	p_A	p_B	p_C	p_{AB}	p_{AC}	p_{BC}	p_{ABC}

After the update with the information in the first premise this space of posible valuation is $2^6 = 64$. This is 4 times smaller than before, but still to large to fit on a A4 page, unless we use abbreviated notation:

$p_\emptyset$	p_A	p_B	p_C	p_{AB}	p_{AC}	p_{BC}	p_{ABC}
				0			0

After the update with the second premise the space of valuations is halved, so now we have $2^5 = 32$ possible valuations. Again, it makes sense to abbreviate. Here is the condensed version of the list of possibilities at this stage:

$p_\emptyset$	p_A	p_B	p_C	p_{AB}	p_{AC}	p_{BC}	p_{ABC}
		1		0			0

To check whether the conclusion holds we have to check whether $p_{ABC} \vee p_{AC}$ holds in all these valuations. This is not the case, hence the inference is invalid.

For the right syllogism the initial space of possible valuations is also the state of ignorance about the 8 propositions describing the regions of a Venn diagram. The condensed version of this list of valuations is the same as before. After the first update, with $\neg p_{AB} \wedge \neg p_{ABC}$, we get:

$p_\emptyset$	p_A	p_B	p_C	p_{AB}	p_{AC}	p_{BC}	p_{ABC}
				0			0

After the second update, with $p_{BC} \vee p_{ABC}$, we get:

$p_\emptyset$	p_A	p_B	p_C	p_{AB}	p_{AC}	p_{BC}	p_{ABC}
				0		1	0

All of the valuations represented here make $p_{BC} \vee p_C$ true, hence the argument is valid.

Exercise 3.6 on page 3-24: We will show that there exist A and D for which both $A \not\Longrightarrow D$ and $A \Longrightarrow D$ can be derived.

We obtain $A \not\Longrightarrow D$ by a simple rule application:

$$\frac{(\mathrm{A}, \mathrm{D}, 0) \in \mathbf{K}}{\mathrm{A} \not\Longrightarrow \mathrm{D}}$$

For $A \Longrightarrow D$ we use the transitivity of $\Longrightarrow$:

$$\frac{\dfrac{\dfrac{(\mathrm{A}, \mathrm{B}, 1) \in \mathbf{K}}{\mathrm{A} \Longrightarrow \mathrm{B}} \quad \dfrac{(\mathrm{B}, \mathrm{C}, 1) \in \mathbf{K}}{\mathrm{B} \Longrightarrow \mathrm{C}}}{\mathrm{A} \Longrightarrow \mathrm{C}} \quad \dfrac{(\mathrm{C}, \mathrm{D}, 1) \in \mathbf{K}}{\mathrm{C} \Longrightarrow \mathrm{D}}}{\mathrm{A} \Longrightarrow \mathrm{D}}$$

Therefore, the knowledge base is inconsistent.

Exercise 3.7 on page 3-24: “I don’t know”

Exercise 3.8 on page 3-24: “I knew that already”

Exercise 3.9 on page 3-24: “I knew that already”

Solutions to Exercises from Chapter 4

Exercise 4.1 on page 4-4: $y < x \vee (\neg x < y \wedge \neg y < x) \vee z < y \vee (\neg z < y \wedge \neg y < z)$ or, by also using the symbol for the equality predicate, $y < x \vee x = y \vee z < y \vee y = z$.

Exercise 4.2 . . .

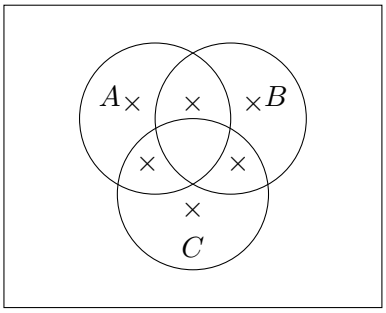
Exercise 4.3 on page 4-5: $(x < y \vee x = y) \wedge (y < z \vee y = z)$

Exercise 4.4 on page 4-5: $\neg((x < y \vee x = y) \wedge (y < z \vee y = z))$ or $\neg(x < y \vee x = y) \vee \neg(y < z \vee y = z)$

Exercise 4.5 on page 4-6: $\forall x(Bx \wedge Wx)$ expresses “All x have the B and W properties” or “Everithing is a boy and walks”. $\exists x(Bx \rightarrow Wx)$ expresses “There is an x such that if he is B then he has the W property” or “There is something (someone) that walks if it’s a boy”.

Exercise **??** on page **??**: . . .

Exercise 4.7 on page 4-7: We can read the desired entailment from the following diagram:



Exercise 4.8 on page 4-7 ...

Exercise 4.9 on page 4-8 ...

Exercise 4.10 on page 4-12: (1) $\exists x \neg Lxb$, (2) $Rbh \wedge Rhb$, (3) $\forall x(Rxb \implies Rmx)$.

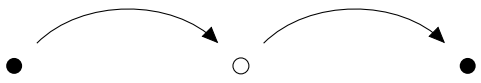
Exercise 4.11 on page 4-12: $\exists x(Bx \wedge \neg Fx)$

Exercise 4.12 on page 4-12: (1) Domain of discourse: all dogs, key: B for the "barking" property, W for the "biting" property, translation: $\forall x \neg(Bx \wedge Wx)$ (2) Domain of discourse: all inaminate objects, key: G for the "glittering" property, A for the "being gold" property, translation: $\exists x(Gx \wedge \neg Ax)$ (3) Domain of discourse: all human beings, key: F for the "frienship" relation, m for the Michelle's name, translation: $\forall xy((Fmx \wedge Fxy) \rightarrow Fmy)$ (4) Domain of discourse: the set of natural numbers $\mathbb{N}$, key: S for the "smaller then" relation, translation: $\exists x \forall y Sxy$ (5) Domain of discourse: the set of natural numbers $\mathbb{N}$, key: P for the propery of "being prime", S for the "smaller then" relation, translation:$\forall x(Px \rightarrow \exists y(Py \wedge Sxy))$.

Exercise 4.13 on page 4-12:
(1) $\forall x(Bx \rightarrow Lxm)$ (2) $\exists x(Gx \wedge \neg Lxx)$ (3) $\neg\exists x((Bx \vee Gx) \wedge Lxp)$ (4) $\exists x(Gx \wedge Lpx \wedge Lxj)$.

Exercise 4.14 on page 4-13: (1):

where solid dots are boys, open dots are girls, the arrow represents the love relation, and Mary is the right dot. (2) the same picture as for point (1). (3) the same picture as for point (1), with Peter the left dot. (4) the same picture as for point (1) with John the dot on the Right:



Exercise 4.15 on page 4-13:

(1) $\exists x \exists y(Bx \wedge Gy \wedge \neg Lxy)$,

(2) $\forall x((Bx \wedge \exists y(Gy \wedge Lxy)) \rightarrow \exists z(Gz \wedge Lzx))$,

(3) $\forall x((Gx \wedge \forall y(By \rightarrow Lxy)) \rightarrow \exists z(Gz \wedge \neg Lxz))$,

(4) $\forall x \forall y(((Gx \wedge \forall v(Bv \rightarrow \neg Lxv)) \wedge (Gy \wedge \exists z(Bz \wedge Lyz))) \rightarrow \neg Lxy)$.

Exercise 4.16 . . .

Exercise 4.19 on page 4-18: (1) T, (2) T, (3) F, (4) T, (5) T.

Exercise 4.20 on page 4-18: (1) T, (2) F, (3) T, (4) T.

Exercise 4.21 on page 4-19:
$\forall x(Px \rightarrow \neg\exists y(\neg Py \wedge Rxy))$

Exercise 4.22 on page 4-20: . . .

Exercise **??** on page **??**: . . .

Exercise 4.23 on page 4-21: . . .

Exercise 4.24 on page 4-21: . . .

Exercise 4.25 on page 4-21: . . .

Exercise 4.26 on page 4-22: . . .

Exercise 4.27 on page 4-24:
$\exists x \exists y \exists z\,(\neg x = y \wedge \neg x = z \wedge \neg x = y \wedge \forall v\,(Pv \leftrightarrow (v = y \vee v = x \vee v = z)))$

Exercise 4.29 on page 4-24:
(1):

(2) in a model with two elements one of which is A but not B and the other is B but not A the formulas have a different meaning (i.e. truth value).

Solutions to Exercises from Chapter 5

Exercise 5.1 on page 5-4: $\forall x \exists y\ y < x$.

Exercise 5.2 on page 5-4: $\forall x(Ox \leftrightarrow \neg 2|x)$

Exercise 5.3 on page 5-4: $\forall x(Px \leftrightarrow (1 < x \wedge \forall y((1 < y \wedge y < x) \rightarrow \neg y|x))))$, where $x \in \mathbb{N}$.

Exercise 5.4 on page 5-6: The occurrences of x in Rxy and $Sxyz$.

Exercise 5.5 on page 5-6: $\exists x$ binds the occurrences of x in Rxx and $\forall x$ binds the occurrence of x in Px.

Exercise 5.6 on page 5-6: (1) and (5).

Exercise 5.7 on page 5-7:

Exercise 5.8 on page 5-8:

Exercise 5.9 on page 5-14:

$$
\begin{array}{rcl}
(Pv_1)^v_a & := & P(v_1)^v_a \\
(Rv_1v_2)^v_a & := & R(v_1)^v_a(v_2)^v_a \\
(Sv_1v_2v_3)^v_a & := & S(v_1)^v_a(v_2)^v_a(v_3)^v_a \\
(v_1 = v_2)^v_a & := & (v_1)^v_a = (v_2)^v_a \\
(\neg\varphi)^v_a & := & \neg(\varphi)^v_a \\
(\varphi_1 \otimes \varphi_2)^v_a & := & (\varphi_1)^v_a \otimes (\varphi_2)^v_a, \quad \text{for} \otimes \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\
(Qu\varphi)^v_a & := & \begin{cases} Qu\varphi & \text{if } u = v \\ Qu(\varphi)^v_a & \text{otherwise} \end{cases} \text{, for } Q \in \{\forall, \exists\}
\end{array}
$$

Exercise 5.10 on page 5-14:

Exercise 5.11 on page 5-15: (1), (2), (4), (6), (9), (11).

Exercise 5.12 on page 5-16: (1) Holds because we assume nonempty domains, (2) Doesn't hold: $D = \{1, 2\}, I(P) = \{1\}$, (3) Holds because we can chose the same object twice, (4) Doesn't hold: $D = \{1, 2\}, I(R) = \{(1, 2), (2, 1)\}$, (5) Doesn't hold: $D = \{1, 2\}, I(R) = \{(1, 2)\}$, (6) Doesn't hold: $D = \{1, 2\}, I(R) = \{(1, 2), (2, 2)\}$, (7) Holds because we can reuse the choice for y in the premise when we chose again in the conclusion, (8) Doesn't hold: $D = \{1, 2\}, I(R) = \{(1, 2), (2, 1)\}$, (9) Doesn't hold: $D = \{1, 2\}, I(R) = \{(1, 2)\}$, (10) see point (3), (11) Holds because the same object can be chosen for both x and y in the conclusion.

Exercise 5.13 on page 5-17: (1) Holds, (2) Holds.

Exercise 5.14 on page 5-20:

$$\exists y\ y + y + y = x. \qquad (x \text{ is a threefold})$$

Exercise 5.15 on page 5-22:

```
x >= y || y >= z
```

Another solution is to change the order of the conditional statements:

```
if x < y && y < z
   print "You lose."
else
   print "You win."
end
```

Exercise 5.16 on page 5-23: Consider what the function would do for input 4. This input satisfies the precondition, so according to the contract the result should be $\texttt{ld}(4) = 2$. But this is not what

we get. The procedure starts by assigning 2 to `d`. Next, the check `d**2 < n` is performed. This check fails, for $d^2 = 2^2 = 4$. Therefore, the while loop will not be executed, and the function returns the value of `n`, i.e., the function returns 4.

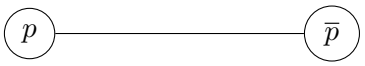
Exercise 5.18 on page 5-27: Yes it does, because in order to decide validity of a formula φ it is enough to know if $\neg\varphi$ is or is not satisfiable.

Solutions to Exercises from Chapter 6

Exercise 6.11 on page 6-10: Suppose $M \not\models \neg K_a \neg K_b \varphi \rightarrow K_b \neg K_a \neg \varphi$ then there must be some $w \in M$ such that $M \models_w \neg K_a \neg K_b \varphi \wedge \neg K_b \neg K_a \neg \varphi$. This means that for some $w \in \{00, 01, 10, 11\}$ we have $M \models_w \widehat{K}_a K_b \varphi \wedge \widehat{K}_b K_a \neg \varphi$. If $M \models_{00} \widehat{K}_a K_b \varphi \wedge \widehat{K}_b K_a \neg \varphi$ then $M \models_{11} \varphi$ and $M \models_{11} \neg\varphi$. If $M \models_{01} \widehat{K}_a K_b \varphi \wedge \widehat{K}_b K_a \neg \varphi$ then $M \models_{10} \varphi$ and $M \models_{10} \neg\varphi$. If $M \models_{10} \widehat{K}_a K_b \varphi \wedge \widehat{K}_b K_a \neg \varphi$ then $M \models_{01} \varphi$ and $M \models_{01} \neg\varphi$. If $M \models_{11} \widehat{K}_a K_b \varphi \wedge \widehat{K}_b K_a \neg \varphi$ then $M \models_{00} \varphi$ and $M \models_{00} \neg\varphi$.

Exercise 6.12 on page 6-10: For $w \in \{wrb, wbr, brw, bwr\}$ the dormula is true because the antecedent is false. We have $M \not\models_{brw} w_b \vee b_b$ hence $M \models_{brw} \neg Ka(w_b \vee b_b)$, therefore $M \models_{rbw} \neg KcKa(w_b \vee b_b)$. We have $M \not\models_{wrb} w_b \vee b_b$ hence $M \models_{wrb} \neg Ka(w_b \vee b_b)$, therefore $M \models_{rwb} \neg KcKa(w_b \vee b_b)$.

Exercise 6.21 on page 6-20: In the model below the formula $p \wedge \neg Kp$ is unsuccesful in the left world:



Exercise 6.22 on page 6-20: The formula $p \rightarrow K_i \neg K_i \neg p$ is a theorem of epistemic logic. Build a proof of it using the axiomatization given in Definition 6.13.

(1)	$K\neg p \rightarrow \neg p$	Axiom (Truth)
(2)	$(K\neg p \rightarrow \neg p) \rightarrow (\neg\neg p \rightarrow \neg K\neg p)$	Theorem (Prop. Logic)
(3)	$\neg\neg p \rightarrow \neg K\neg p$	Modus Ponens (1,2)
(4)	$\neg K\neg p \rightarrow K\neg K\neg p$	Axiom (Introspection$^-$)
(5)	$(\neg\neg p \rightarrow \neg K\neg p) \rightarrow$ $((\neg K\neg p \rightarrow K\neg K\neg p) \rightarrow (\neg\neg p \rightarrow K\neg K\neg p))$	Theorem (Prop. Logic)
(6)	$(\neg K\neg p \rightarrow K\neg K\neg p) \rightarrow (\neg\neg p \rightarrow K\neg K\neg p)$	Modus Ponens (3,5)
(7)	$\neg\neg p \rightarrow K\neg K\neg p$	Modus Ponens (4,6)
(8)	$(\neg\neg p \rightarrow K\neg K\neg p) \rightarrow (p \rightarrow K\neg K\neg p)$	Theorem (Prop. Logic)
(9)	$p \rightarrow K\neg K\neg p$	Modus Ponens (7,8)

Exercise 6.23 on page 6-20: Consider the epistemic situation described in the following scenario: "Three students Ann, Bob and Cath send their request for a certain logic textbook to the library server in the same day at 9am, 10am and 11am, respectivelly, without knowing this about each other. Because the library has only one copy of the book, the librarian replies by e-mail to everyone

informing them that the loan will be given to whoever filed the request first, without any further detail".

(a) Represent the student's knowlege in an epistemic model.

(b) Check in the model build in point (a) if any of the students knows that (s)he will receive the book.

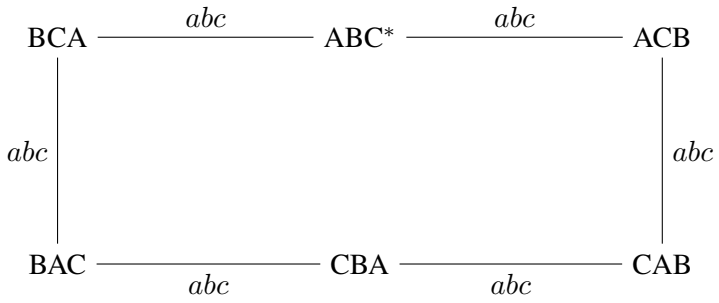
(c) Anxious to find out who will receive the book, Bob writes an e-mail to everione saying: 'I filed my request at 10am'. Compute the effect of this announcement on the epistemic model from point (a).

(d) Check in this new model if if any of the students knows that (s)he will receive the book.

(e) After this Cath sends a "reply-all" to Bob's previous message saying: 'Bob, I think that you might receive the book'. Compute the effect of this announcement on the epistemic model from point (c).

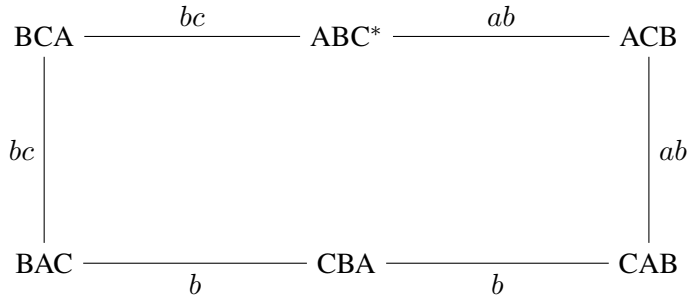
(f) Check in this new model if if any of the students knows that (s)he will receive the book.

(a) The initial epistemic situation can be represented by the following model (reflexive and transitive relations are skipped):



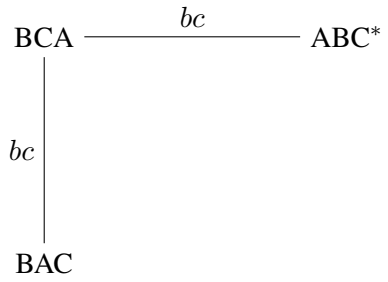
(b): Nobody knows.

(c) The new epistemic situation can be represented as follows (again reflexive and transitive links are skipped):



(d): Nobody knows.

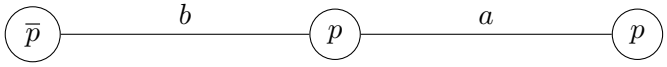
(e) The new epistemic situation can be represented as follows (again reflexive and transitive links are skipped):



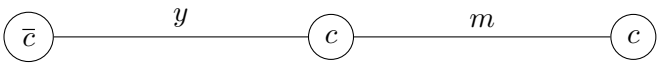
(f): Ann knows that she will receive the book, Bob and Cath do not know.

Exercise 6.25 on page 6-23: Suppose that $C\varphi_1, \ldots, C\varphi_n \not\models C\psi$. Hence there must be an epistemic model M ande a world w such that $M \models_w C\varphi_i$ for all i and $M \not\models_w C\psi$. This means that there exists a world v in M which is $(\overset{G}{\sim})^*$-accessible from w and $M \not\models_v \psi$. But since all formulas $C\varphi_i$ are true in w we also have $M \models_v \varphi_i$ for all i. But then this world v in M is also a counter-example showing that ψ is not a valid consequence from $\varphi_1, \ldots, \varphi_n$.

Exercise 6.26 on page 6-24: This happens in the model below where the actual world is the rightmost one:

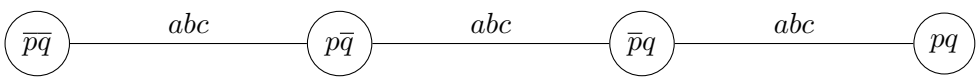


Exercise 6.27 on page 6-24: After the ATM machine counts the amount X and you also do it yourself the epistemic situation can be modeled by the model below where the actual world is the rightmost one. Common knowledge is not achieved because the ATM machine has no way of knowing whether you counted the money, so mutual knowledge stops after one itaration.

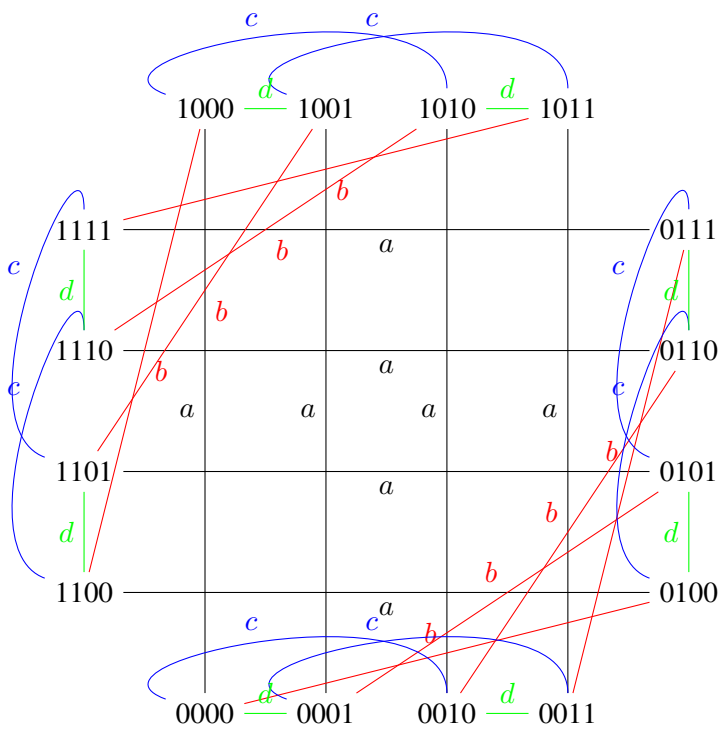


Exercise 6.29 on page 6-25: $p \vee q \equiv \neg(\neg p \wedge \neg q); p \rightarrow q \equiv \neg p \vee q; p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$

Exercise 6.30 on page 6-25:



Exercise 6.32 on page 6-26: The innitial epistemic situation can be modelled as depicted in the figure below:



After this the information evolves by eliminating one world (0000) after the father's announcement, four worlds (1000, 0100, 0010 and 0001) after the first round of ignorance answers (you can think about this in a positive twist as four instances of announcements giving what the children **do** know, or can observe: "I do not see three clean faces"), and six worlds (1001, 10101, 0110, 0101, 0011 and 1100) after the second round of ignarance answers (a positive stance would equate this again with announcing what each of the children **does** know by observation: "I do not see two clean faces" and can infer from this). In the end five worlds (1111, 1110, 1101, and 0111) survive this sequence of announcements and in the actual one b, c, and d already know that they are muddy.

Exercise 6.33 on page 6-26: Which of the following formulas are valid (you can assume that there are only two agents a and b):

(1) $((K_a\varphi \wedge K_b\varphi) \rightarrow (K_aK_b\varphi \wedge K_bK_a\varphi)) \rightarrow C\varphi$

(2) $C\varphi \rightarrow K_aC\varphi$

(3) $C\varphi \rightarrow CC\varphi$

(4) $\neg C\varphi \rightarrow C\neg C\varphi$

Build a counterexample for the invalid formulas and a proof for the valid ones.

(1) is not valid, check it in the leftmost world of the following model:

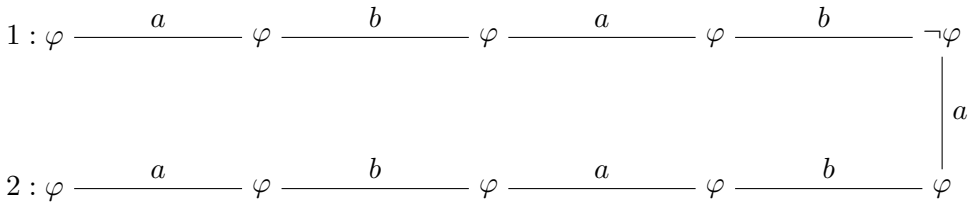
$$\varphi^* \xrightarrow{\ a\ } \varphi \xrightarrow{\ b\ } \varphi \xrightarrow{\ a\ } \varphi \xrightarrow{\ b\ } \neg\varphi$$

(2) Assume $M \models_w C\varphi$ for arbitrary M and w. Suppose $M \not\models_w K_a C\varphi$. Then there must be an v such that $w \overset{a}{\sim} v$ and $M \not\models_v C\varphi$. Therefore, there must be some u sucht that $v(\overset{ab}{\sim})^* u$ and $M \not\models_u \varphi$. Hence we also have $w(\overset{ab}{\sim})^* u$, hence $M \not\models_w C\varphi$ against the assumption.

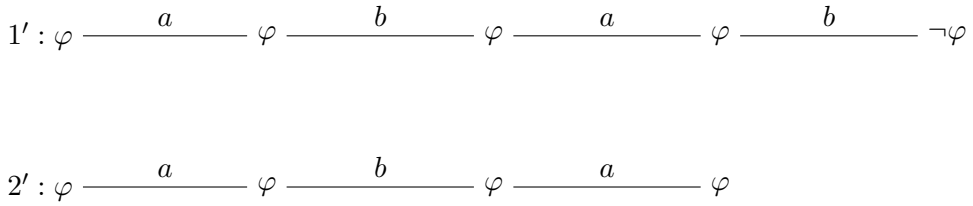
(3) Assume $M \models_w C\varphi$ for arbitrary M and w. Suppose $M \not\models_w CC\varphi$. Then there must be an v such that $w(\overset{ab}{\sim})^* v$ and $M \not\models_v C\varphi$. Therefore, there must be some u sucht that $v(\overset{ab}{\sim})^* u$ and $M \not\models_u \varphi$. Hence we also have $w(\overset{ab}{\sim})^* u$, hence $M \not\models_w C\varphi$ against the assumption.

(4) is not valid, the antecedent is true and the consequent false in the rightmost world of the model from point (1).

Exercise 6.34 on page 6-27: Consider the following two wpistemic models:



(a) Can you find am formula of epistemic logic that is true in world 1 and false in world 2?

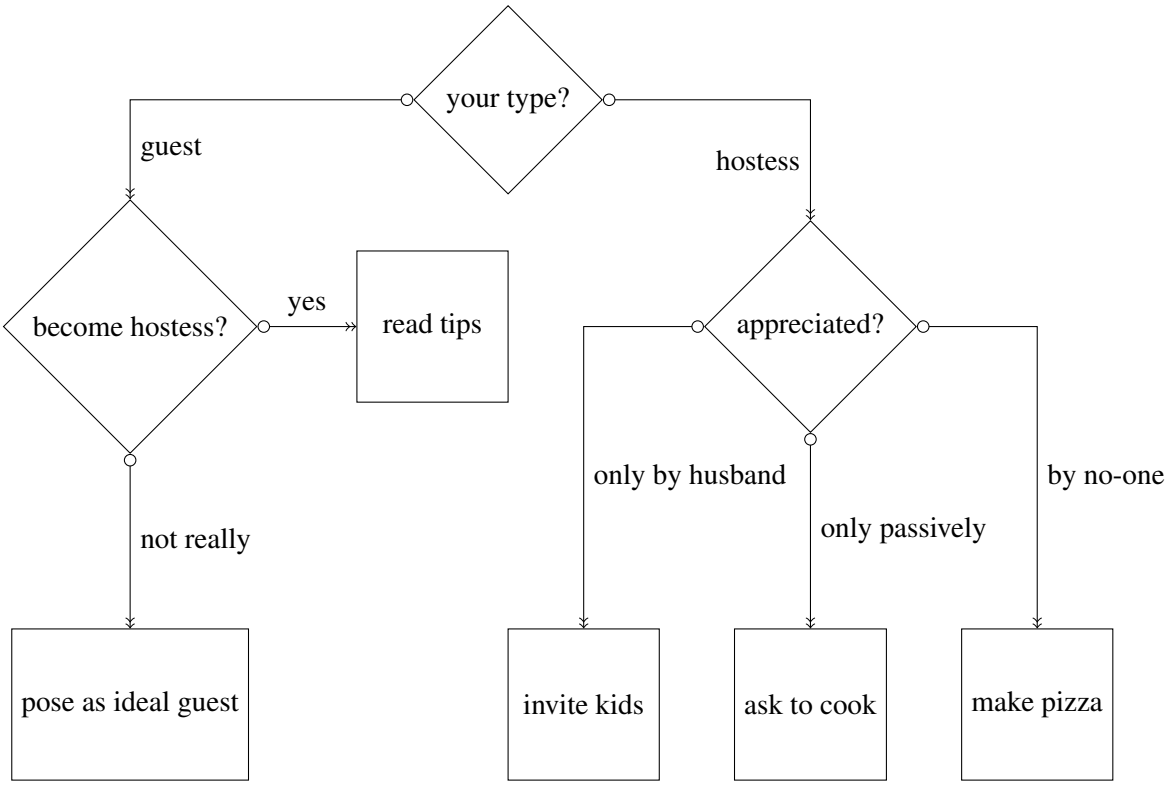


(b) Can you find a formula of epistemic logic that is true in world $1'$ and false in world $2'$? (c) Can you find a formula that, when announced, changes the first model into the second one? (d) can you find a formula of public announcement logic that is true in world 1 and false in world 2?

(a) No, (b) $C\varphi$, (c) $\varphi \to K_a\varphi$ (d) $[\varphi \to K_a\varphi]C\varphi$

Solutions to Exercises from Chapter 7

Exercise 7.6 on page 7-4:



Exercise 7.7 on page 7-4: $b^{\smile}; a^{\smile}$.

Exercise 7.8 on page 7-5: $?(p \wedge q)$.

Exercise 7.9 on page 7-5: $?(p \vee q)$.

Exercise 7.10 on page 7-5: $?\top$.

Exercise 7.11 on page 7-5: $?\bot$.

Exercise 7.15 on page 7-5: maternal grandfather.

Exercise 7.17 on page 7-6: $\supseteq$.

Exercise 7.19 on page 7-6: $R_1 = \{(1,2),(3,4)\}; R_2 = \{(1,3)\}; R_3 = \{(3,1)\}$

Exercise 7.20 on page 7-6: $R^{\smile\smile} = \{(x,y) \in S^2 \mid (y,x) \in R^{\smile}\} = \{(x,y) \in S^2 \mid (x,y) \in R\} = R$

Exercise 7.21 on page 7-6: $(R_1 \cup R_2)^{\smile} = \{(x,y) \in S^2 \mid (y,x) \in (R_1 \cup R_2)\} = \{(x,y) \in S^2 \mid (y,x) \in R_1$ or $(y,x) \in \cup R_2)\} = {R_1}^{\smile} \cup {R_2}^{\smile}$.

Exercise 7.23 on page 7-8: $\alpha^0 :=?\top$ and $\alpha^n := \alpha^{n-1}; \alpha$, for any $n > 0$.

Exercise 7.26 on page 7-9: $M \models_w \langle a;b\rangle\top$ is equivalent to $w \in [\![\langle a;b\rangle\top]\!]^{\mathcal{M}}$. From the semantic definition we have: $[\![\langle a;b\rangle\top]\!]^{\mathcal{M}} = \{w \in W_{\mathcal{M}} \mid \exists v \in W_{\mathcal{M}} : (w,v) \in [\![(a;b)]\!]^{\mathcal{M}}$ and $v \in$

$[\![\top]\!]^{\mathcal{M}}\} = \{w \in W_{\mathcal{M}} \mid \exists v \in W_{\mathcal{M}} : (w,v) \in [\![a]\!]^{\mathcal{M}} \circ [\![b]\!]^{\mathcal{M}}\} = \{w \in W_{\mathcal{M}} \mid \exists v \in W_{\mathcal{M}} : (w,v) \in \{(x,y) \in W^2_{\mathcal{M}} \mid \exists z \in W_{\mathcal{M}}((x,z) \in \xrightarrow{a}_{\mathcal{M}} \wedge (z,y) \in \xrightarrow{b}_{\mathcal{M}})\}\} = \{w \in W_{\mathcal{M}} \mid \exists v \in W_{\mathcal{M}} : (w,v) \in \{(x,y) \in W^2_{\mathcal{M}} \mid \exists z \in W_{\mathcal{M}}((x,z) \in \xrightarrow{a}_{\mathcal{M}} \wedge (z,y) \in \xrightarrow{b}_{\mathcal{M}})\}\} = \{w \in W_{\mathcal{M}} \mid \exists v \in W_{\mathcal{M}} : (w,v) \in \xrightarrow{a}_{\mathcal{M}}$ and $v \in \{u \in W_{\mathcal{M}} \mid (v,u) \xrightarrow{b}_{\mathcal{M}}$ and $u \in W_{\mathcal{M}}\}\} = \{w \in W_{\mathcal{M}} \mid \exists v \in W_{\mathcal{M}} : (w,v) \in \xrightarrow{a}_{\mathcal{M}}$ and $v \in [\![\langle b\rangle\top]\!]^{\mathcal{M}}\} = [\![\langle a\rangle\langle b\rangle\top]\!]^{\mathcal{M}}$. $[\![\langle a\rangle\langle b\rangle\top]\!]^{\mathcal{M}}$ is equivalent to $M \models_w \langle a\rangle\langle b\rangle\top$.

Exercise 7.27 on page 7-9: $[\![?p]\!]^{\mathcal{M}} = \{(pq,pq),(p\overline{q},p\overline{q})\}$, $[\![?(p \vee q)]\!]^{\mathcal{M}} = \{(pq,pq),(p\overline{q},p\overline{q}),(\overline{p}q,\overline{p}q)\}$, $[\![a;b]\!]^{\mathcal{M}} = [\![b;a]\!]^{\mathcal{M}} = \{(pq,\overline{p}q)\}$.

Exercise 7.28 on page 7-9: $[\![\beta\check{}\,;\alpha\check{}\,]\!]^{\mathcal{M}} = [\![\beta\check{}\,]\!]^{\mathcal{M}} \circ [\![\alpha\check{}\,]\!]^{\mathcal{M}} = \{(s,t) \mid \exists u \in S_{\mathcal{M}}((s,u) \in [\![\beta\check{}\,]\!]^{\mathcal{M}} \wedge (u,t) \in [\![\alpha\check{}\,]\!]^{\mathcal{M}}\} = \{(s,t) \mid \exists u \in S_{\mathcal{M}}((u,s) \in [\![\beta]\!]^{\mathcal{M}} \wedge (t,u) \in [\![\alpha]\!]^{\mathcal{M}}\} = \{(s,t) \mid (t,s) \in [\![\alpha]\!]^{\mathcal{M}} \circ [\![\beta]\!]^{\mathcal{M}}\} = \{(s,t) \mid (t,s) \in [\![\alpha;\beta]\!]^{\mathcal{M}}\} = [\![(\alpha;\beta)\check{}\,]\!]^{\mathcal{M}}$

$[\![\beta\check{} \cup \alpha\check{}\,]\!]^{\mathcal{M}} = [\![\beta\check{}\,]\!]^{\mathcal{M}} \cup [\![\alpha\check{}\,]\!]^{\mathcal{M}} = [\![\beta\check{}\,]\!]^{\mathcal{M}} \cup [\![\alpha\check{}\,]\!]^{\mathcal{M}} = \{(s,t) \mid (t,s) \in [\![\alpha]\!]^{\mathcal{M}}\} \cup \{(s,t) \mid (t,s) \in [\![\beta]\!]^{\mathcal{M}}\} = \{(s,t) \mid (t,s) \in [\![\beta]\!]^{\mathcal{M}} \cup [\![\alpha]\!]^{\mathcal{M}}\} = [\![(\beta \cup \alpha)\check{}\,]\!]^{\mathcal{M}}$

$[\![(\alpha^*)\check{}\,]\!]^{\mathcal{M}} = \{(s,t) \mid (t,s) \in [\![\alpha^*]\!]^{\mathcal{M}}\} = \{(s,t) \mid (t,s) \in ([\![\alpha]\!]^{\mathcal{M}})^*\} = \{(s,t) \mid (t,s) \in \bigcup_{n\in\mathbb{N}}([\![\alpha]\!]^{\mathcal{M}})^n\} = \bigcup_{n\in\mathbb{N}}([\![\alpha\check{}\,]\!]^{\mathcal{M}})^n = ([\![\alpha\check{}\,]\!]^{\mathcal{M}})^*$

Exercise 7.29 on page 7-10: $\alpha\check{} = \begin{cases} \beta\check{} \cup \gamma\check{} & \text{if } \alpha = (\beta \cup \gamma)\check{} \\ \beta\check{};\gamma\check{} & \text{if } \alpha = (\beta;\gamma)\check{} \\ ?\varphi & \text{if } \alpha = ?\varphi\check{} \\ (\beta\check{})^* & \text{if } \alpha = (\beta^*)\check{} \\ \beta & \text{if } \alpha = \beta\check{}\check{} \\ \{(t,s) \mid (s,t) \in a\} & \text{if } \alpha = a \end{cases}$

Exercise 7.30 on page 7-10: $\langle(a;b) \cup (?\varphi;c)\rangle\psi \leftrightarrow \langle a;b\rangle\psi \vee \langle ?\varphi;c\rangle\psi \leftrightarrow \langle a\rangle\langle b\rangle\psi \vee \langle ?\varphi\rangle\langle c\rangle\psi \leftrightarrow \langle a\rangle\langle b\rangle\psi \vee (\varphi \wedge \langle c\rangle\psi)$.

Exercise 7.31 on page 7-11: $\vdash \langle\alpha^*\rangle\varphi \leftrightarrow \varphi \vee \langle\alpha\rangle\langle\alpha^*\rangle\varphi$.

Exercise 7.33 on page 7-15: $[\![i]\!]_s = i, [\![v]\!]_s = s(v), [\![a_1 + a_2]\!]_s = [\![a_1]\!]_s + [\![a_2]\!]_s, [\![a_1 * a_2]\!]_s = [\![a_1]\!]_s * [\![a_2]\!]_s, [\![a_1 - a_2]\!]_s = [\![a_1]\!]_s - [\![a_2]\!]_s$.

Exercise 7.34 on page 7-16: There are three possible procedure stages: (1) both drawn pebbles are black (2) both drawn pebbles are white (3) there have been drawn a white and a black pebble. For (1) the number of white pebbles remains unchanged, for (2) the number of white pebbles remains odd ($n' = n - 2$) for (3) the number of white pebbels remains unchanged ($n' = n - 1 + 1$). Hence the "oddness" property of the number of white pebbels is invariant during any number of executions of the drawing procedure. Therefore, if there is only one pebble left it must be white.

Exercise 7.37 on page 7-19:

(1)

a. $\{1,3\}$

b. $\{2\}$

c. $\{2,3\}$

(2) The formula $\langle ?p\rangle \wedge [b](p \wedge \neg p)$ is true only at state 4 (but there are others).

(3)

a. $b; b = \{ (2,4) \}$

b. $a \cup b = \{ (1,2), (2,2), (1,4), (4,4), (2,3), (3,4) \}$

c. $a^* = \{ (1,1), (2,2), (3,3), (4,4), (1,2), (1,4) \}$

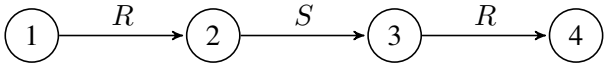
(4) The action $\big(?(\neg p \wedge \neg q); a; b\big)$ is given by $\{ (1,3) \}$.

Exercise 7.38 on page 7-20:

(1)

$$\begin{array}{lcll} [R;(S \cup T)]\varphi & \leftrightarrow & [R][S \cup T]\varphi & \text{by axiom } [\pi_1;\pi_2]\varphi \leftrightarrow [\pi_1][\pi_2]\varphi \\ & \leftrightarrow & [R]([S]\varphi \wedge [T]\varphi) & \text{by axiom } [\pi_1 \cup \pi_2]\varphi \leftrightarrow ([\pi_1]\varphi \wedge [\pi_2]\varphi) \\ & \leftrightarrow & [R][S]\varphi \wedge [R][T]\varphi & \text{by axiom } [\pi](\varphi \wedge \psi) \leftrightarrow ([\pi]\varphi \wedge [\pi]\psi) \end{array}$$

(2) The relation $(R \cup S)^*$ allow you to choose between R and S any number of times so, for example, $R; S; R$ is allowed. The relation $R^*; S^*$ tells you to apply R any number of times, *and then* apply S any number of times, so $R; S; R$ is not allowed. The following model is a counter-example:

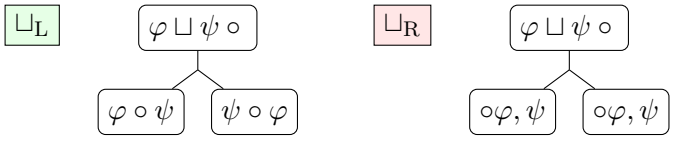


We have $(1,4) \in (R \cup S)^*$, but $(1,4) \notin R^*; S^*$.

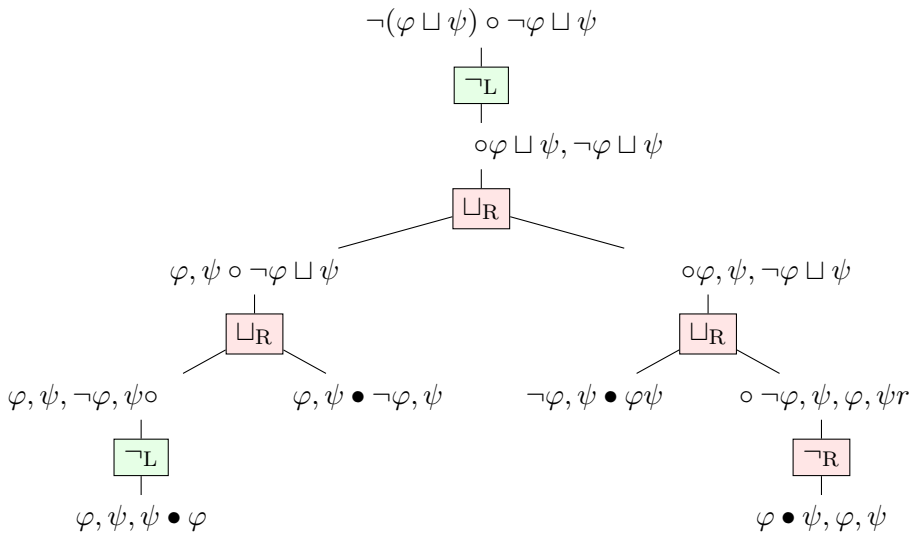
Exercise 7.39 on page 7-20:
(a) No.
(b) (1): $\{5,6\}$, (2): $\{3\}$, (3): $\emptyset$;
(c) $K_E(\langle l\rangle q \vee \langle r\rangle q)$.

Solutions to Exercises from Chapter 9

Exercise 9.1 on page 9-8:

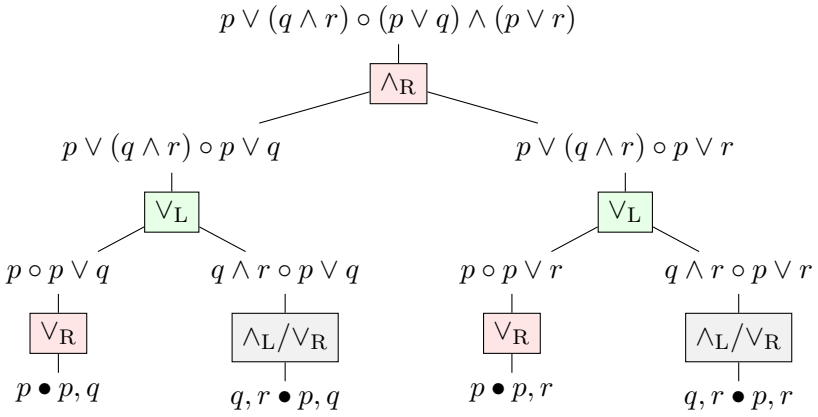


Exercise 9.2 on page 9-8: Here you have the left-to-right direction:



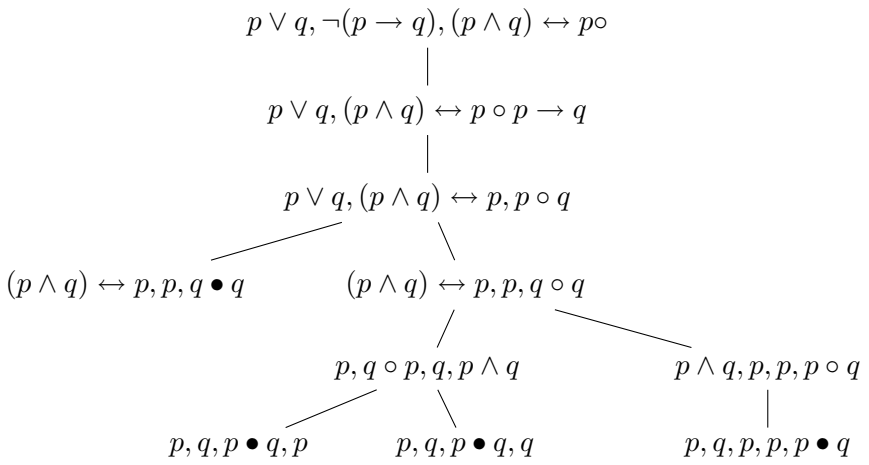
You can check that the right-to-left direction also holds.

Exercise 9.3 on page 9-8: Here you have the table for point (1):



You can chack that point (2) also tests positively for validity.

Exercise 9.4 on page 9-9: Here you have the table for point (2), now with implicite rules:



You can check that point (1) is not satisfiable either.

Exercise 9.5 on page 9-9:

Exercise 9.6 on page 9-11:

Exercise 9.7 on page 9-12:

Exercise 9.8 on page 9-12:

Exercise 9.9 on page 9-16:

Exercise 9.10 on page 9-16:

Exercise 9.11 on page 9-19:

Exercise 9.12 on page 9-19:

Exercise 9.13 on page 9-19:

Exercise 9.14 on page 9-21:

Exercise 9.15 on page 9-26:

Exercise 9.16 on page 9-27:

Exercise 9.17 on page 9-27:

Exercise 9.18 on page 9-27:

Exercise 9.19 on page 9-29:

Exercise 9.20 on page 9-29:

Solutions to Exercises from Chapter 10

Exercise 10.1 on page **??**

Exercise 10.2 on page **??**

Exercise 10.3 on page **??**

Exercise **??** on page **??**

Exercise 10.5 on page **??**

Exercise 10.6 on page **??**

Exercise 10.7 on page 10-10

Exercise 10.8 on page **??**

Exercise 10.9 on page **??**

Exercise 10.10 on page **??**

Exercise 10.11 on page **??**

Exercise 10.12 on page **??**

Exercise 10.13 on page **??**

Exercise 10.14 on page **??**

Exercise **??** on page **??**

Exercise **??** on page **??**

Solutions to Exercises from Chapter 11

Exercise 11.1 on page 11-4:

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

Exercise 11.2 on page 11-5:

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

Exercise 11.3 on page 11-6:

$$\begin{aligned} CNF((p \vee \neg q) \wedge (q \vee r)) &= CNF(p \vee \neg q) \wedge CNF(q \vee r) \\ &= DIST(CNF(p), CNF(\neg q)) \wedge DIST(CNF(q), CNF(r)) \\ &= DIST(p, \neg q) \wedge DIST(q, r) \\ &= (p \vee \neg q) \wedge (q \vee r) \end{aligned}$$

Exercise 11.4 on page 11-6:

$$\begin{aligned} & CNF((p \wedge q) \vee (p \wedge r) \vee (q \wedge r)) = \\ = \; & DIST(CNF(p \wedge q), CNF((p \wedge r) \vee (q \wedge r))) \\ = \; & DIST((CNF(p) \wedge CNF(q)), DIST(CNF(p \wedge r), CNF(q \wedge r))) \\ = \; & DIST((p \wedge q), DIST((CNF(p) \wedge CNF(r)), (CNF(q) \wedge CNF(r)))) \\ = \; & DIST((p \wedge q), DIST((p \wedge r), (q \wedge r))) \\ = \; & DIST((p \wedge q), (DIST(p, (q \wedge r)) \wedge DIST(r, (q \wedge r)))) \\ = \; & DIST((p \wedge q), (DIST(p, q) \wedge DIST(p, r)) \wedge (DIST(r, q) \wedge DIST(r, r))) \\ = \; & DIST((p \wedge q), ((p \vee q) \wedge (p \vee r)) \wedge ((r \vee q) \wedge (r \vee r))) \\ = \; & DIST(p, ((p \vee q) \wedge (p \vee r)) \wedge ((r \vee q) \wedge (r \vee r))) \wedge \\ & \qquad \wedge DIST(q, ((p \vee q) \wedge (p \vee r)) \wedge ((r \vee q) \wedge (r \vee r))) \\ = \; & (DIST(p, ((p \vee q) \wedge (p \vee r))) \wedge DIST(p, ((r \vee q) \wedge (r \vee r)))) \wedge \\ & \qquad \wedge (DIST(q, ((p \vee q) \wedge (p \vee r))) \wedge DIST(q, ((r \vee q) \wedge (r \vee r)))) \\ = \; & (DIST(p, (p \vee q)) \wedge DIST(p, (p \vee r))) \wedge (DIST(p, (r \vee q)) \wedge DIST(p, (r \vee r)))) \wedge \\ & \qquad \wedge (DIST(q, (p \vee q)) \wedge DIST(q, (p \vee r))) \wedge (DIST(q, (r \vee q)) \wedge DIST(q, (r \vee r)))) \\ = \; & (p \vee p \vee q) \wedge (p \vee p \vee r) \wedge (p \vee r \vee q) \wedge (p \vee r \vee r) \wedge (q \vee p \vee q) \wedge (q \vee p \vee r) \wedge (q \vee r \vee q) \wedge (q \vee r \vee r) \end{aligned}$$

Exercise 11.5 on page 11-8:

Assume the premisse is true. Then, because the premise is a clause form $C_1, \ldots, C_i, \ldots, C_n$, every conjunct C_k for $k \in \{1, ..., n\}$ is true. Therefore every C_k for $k \in \{1, \ldots, i-1, i+1, \ldots, n\}$ is true. Hence the conclusion is true and the inference rule is sound.

Exercise 11.6 on page 11-9:

Test the validity of the following inferences using resolution:

(1) $((p \vee q) \wedge \neg q) \rightarrow r, q \leftrightarrow \neg p \models r$

(2) $(p \vee q) \rightarrow r, \neg q, \neg q \leftrightarrow p \models r$

(1) First we translate the inferences in a corresponding clause form as follows:

$$\{\{\neg p, q, r\}, \{\neg p, \neg q\}, \{q, p\}, \{\neg r\}\}$$

next, we apply resolution, to the first and second clauses:

$$\{\{\neg p, r\}, \{q, p\}, \{\neg r\}\}$$

we apply resolution again, to the first and second clauses:

$$\{\{r, q\}, \{\neg r\}\}$$

we apply resolution one more time, to the first and second clauses:

$$\{\{q\}\}$$

The clause form containing the premises and the conclusion negated is satisfiable, therefore the inference is not valid.

(2) First we translate the inferences in a corresponding clause form as follows:

$$\{\{\neg p, r\}, \{\neg q, r\}, \{\neg q\}, \{\neg p, \neg q\}, \{q, p\}, \{\neg r\}\}$$

next, we apply resolution, to the first and fifth clauses:

$$\{\{r, q\}, \{\neg q, r\}, \{\neg q\}, \{\neg p, \neg q\}, \{\neg r\}\}$$

we apply resolution again, to the first and second clauses:

$$\{\{r\}, \{\neg q\}, \{\neg p, \neg q\}, \{\neg r\}\}$$

we apply resolution one more time, to the first and last clauses:

$$\{[], \{\neg q\}, \{\neg p, \neg q\}\}$$

The clause form containing the premises and the conclusion negated is not satisfiable, therefore the inference is valid.

Exercise 11.7 on page 11-9:

Determine which of the following clause forms are satisfiable:

(1) $\{\{\neg p, q\}, \{\neg q\}, \{p, \neg r\}, \{\neg s\}, \{\neg t, s\}, \{t, r\}\}$

(2) $\{\{p, \neg q, r\}, \{q, r\}, \{q\}, \{\neg r, q\}, \{\neg p, r\}\}$

Give a satisfying valuation for the satisfyable case(s).

(1) We start with the clause form:

$$\{\{\neg p, q\}, \{\neg q\}, \{p, \neg r\}, \{\neg s\}, \{\neg t, s\}, \{t, r\}\}.$$

Applying resolution for $\neg q, q$ to the first two clauses gives:

$$\{\{\neg p\}, \{p, \neg r\}, \{\neg s\}, \{\neg t, s\}, \{t, r\}\}.$$

Applying resolution for $\neg p, p$ to the first two clauses gives:

$$\{\{\neg r\}, \{\neg s\}, \{\neg t, s\}, \{t, r\}\}.$$

Applying resolution for $\neg r, r$ to the first and last clauses gives:

$$\{\{\neg s\}, \{\neg t, s\}, \{t\}\}.$$

Applying resolution for $\neg s, s$ to the first two clauses gives:

$$\{\{\neg t\}, \{t\}\}.$$

Applying resolution for $\neg t, t$ to the first two clauses gives:

$$\{[]\}.$$

We have derived a clause form containing the empty clause. We have tried to construct a situation where all clauses are true but this attempt has led us to a contradiction. Hence the clause form is not satisfiable.

(2) We start with the clause form:

$$\{\{p, \neg q, r\}, \{q, r\}, \{q\}, \{\neg r, q\}, \{\neg p, r\}\}$$

Applying resolution for $\neg q, q$ to the first two clauses gives:

$$\{\{p, r\}, \{r\}, \{q\}, \{\neg r, q\}, \{\neg p, r\}\}$$

Applying resolution for $r, \neg r$ to the second and fourth clauses gives:

$$\{\{p, r\}, \{q\}, \{q\}, \{\neg p, r\}\}$$

Applying resolution for $p, \neg p$ to the first and last clauses gives:

$$\{\{r\}, \{q\}, \{q\}, \{r\}\}$$

The clause form is satisfiable, and a valuation that satisfy it is the one in which all propositional atoms p, q and r are true.

Exercise 11.8 on page 11-9:

First we express the constraints as logical formulas, as follows:

- $a \vee b$
- $(a \wedge e \wedge \neg f) \vee (a \wedge \neg e \wedge f) \vee (\neg a \wedge e \wedge f)$
- $b \leftrightarrow c$
- $a \leftrightarrow \neg d$
- $c \leftrightarrow \neg d$
- $\neg d \rightarrow \neg r$

Next we translate each formula in conjunctive normal form, as follows:

- $a \vee b$
- $(a \vee e) \wedge (a \vee f) \wedge (e \vee f) \wedge (\neg a \vee \neg e \vee \neg f)$
- $(\neg b \vee c) \wedge (b \vee \neg c)$
- $(\neg a \vee \neg d) \wedge (a \vee d)$
- $(\neg c \vee \neg d) \wedge (c \vee d)$
- $d \vee \neg e$

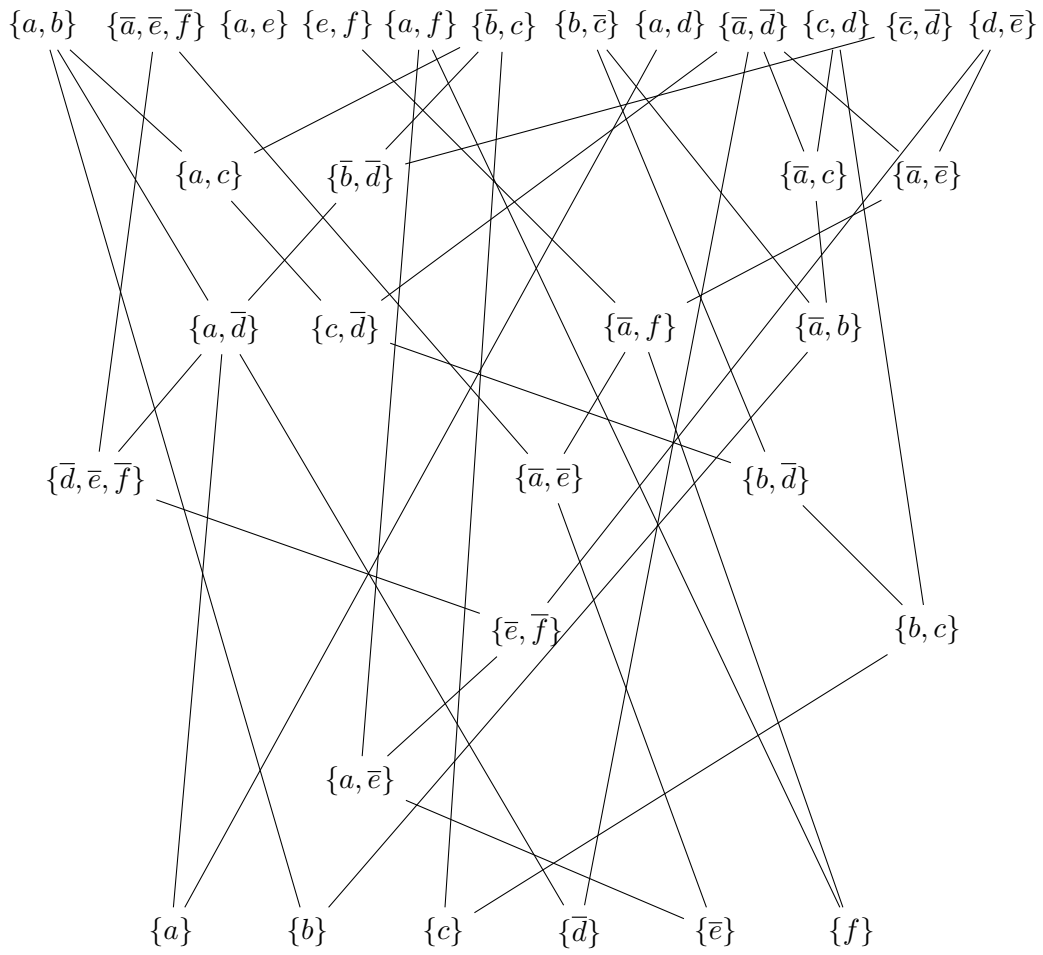
From this we can construct the following clause form:

$$\{\{a, b\}, \{a, e\}, \{a, f\}, \{e, f\}, \{\neg a, \neg e, \neg f\}, \{\neg b, c\}, \{b, \neg c\}, \{\neg a, \neg d\}, \{a, d\}, \{\neg c, \neg d\}, \{c, d\}, \{d, \neg e\}\}$$

Exercise 11.9 on page 11-9:

$$\{\{a, b\}, \{a, e\}, \{a, f\}, \{e, f\}, \{\neg a, \neg e, \neg f\}, \{\neg b, c\}, \{b, \neg c\}, \{\neg a, \neg d\}, \{a, d\}, \{\neg c, \neg d\}, \{c, d\}, \{d, \neg e\}\}$$

We can apply resolution and find out the satisfying valuation in the following way:



Exercise 11.10 on page 11-11:

(1) $\{(1,2),(2,3),(3,4),(1,3),(1,4),(2,4)\}$,

(2) $\{(1,2),(2,3),(3,4),(1,3),(1,4),(2,4)\}$,

(3) $\{(1,2),(2,3),(3,4),(1,3),(1,4),(2,4)\}$,

(4) $\{(1,2),(2,1),(1,1),(2,2)\}$,

(5) $\{(1,1),(2,2)\}$.

Exercise 11.11 on page 11-13: $(\forall x\forall y\forall z((Rxy \wedge Ryz) \rightarrow Rxz) \wedge \forall x\forall y\forall z((Sxy \wedge Syz) \rightarrow Sxz)) \rightarrow (\forall x\forall y\forall z((R \circ Sxy \wedge R \circ Syz) \rightarrow R \circ Sxz))$

Exercise 11.12 on page 11-14:

$$\begin{aligned} R \circ S &= \{(0,2),(2,1),(1,1)\}, \\ (R \circ S)^+ &= \{(0,2),(2,1),(1,1),(0,1)\}. \end{aligned}$$

Solutions to Exercises from Chapter 13

Exercise 13.1 on page 13-3:

1.	$\mathbf{0}+\mathbf{0}=\mathbf{0}+\mathbf{0}$	PL axioms 2,4
2.	$y+\mathbf{0}=\mathbf{0}+y$	assumption
3.	$sy+\mathbf{0}=sy$	PA3
4.	$sy=s(y+\mathbf{0})$	PA3, PL
5.	$s(y+\mathbf{0})=s(\mathbf{0}+y)$	from 2 by PL
6.	$s(\mathbf{0}+y)=\mathbf{0}+sy$	PA3
7.	$sy+\mathbf{0}=\mathbf{0}+sy$	from 3,4,5,6 by PL
8.	$y+\mathbf{0}=\mathbf{0}+y\rightarrow sy+\mathbf{0}=\mathbf{0}+sy$	2, conditionalisation
9.	$\forall y(y+\mathbf{0}=\mathbf{0}+y\rightarrow sy+\mathbf{0}=\mathbf{0}+sy)$	8, UGEN
10.	$\forall y\ y+\mathbf{0}=\mathbf{0}+y)$	from 1, 6, PA5-scheme by MP

Exercise 13.2 on page 13-3:

1.	$x+y=y+x$	assumption
2.	$x+sy=s(x+y)$	PA4
3.	$s(x+y)=s(y+x)$	from 1 by PL
4.	$s(y+x)=y+sx$	PA4
5.	$y+sx=y+s(x+\mathbf{0})$	PA3
6.	$y+s(x+\mathbf{0})=y+(x+s\mathbf{0})$	PA4
7.	$y+(x+s\mathbf{0})=y+(s\mathbf{0}+x)$	from 1
8.	$y+(s\mathbf{0}+x)=(y+s\mathbf{0})+x$	proved in the text
9.	$(y+s\mathbf{0})+x=s(y+\mathbf{0})+x$	PA4
10.	$s(y+\mathbf{0})+x=sy+x$	PA3
11.	$x+sy=sy+x$	from 2, . . . , 10 by PL
12.	$x+y=y+x\rightarrow x+sy=sy+x$	1, conditionalisation
13.	$\forall y(x+y=y+x\rightarrow x+sy=sy+x)$	12, UGEN

Exercise 13.3 on page 13-3:
. . . Maybe this is too difficult . . .

Solutions to Exercises from Chapter A

Exercise A.1 on page A-2

Exercise A.2 on page A-2

Exercise A.3 on page A-4

Exercise A.4 on page A-4

Exercise A.5 on page A-4

Exercise A.6 on page A-4

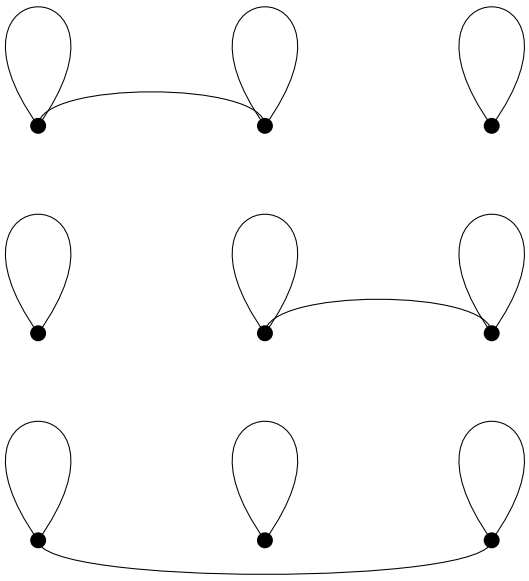
Exercise A.7 on page A-4

Exercise A.8 on page A-5:
$\{(1,1),(2,2),(3,3),(1,2),(1,3),(2,3)\}$

Exercise A.9 on page A-6:
$\{(1,1),(2,2),(3,3),(1,2),(1,3),(2,3),(2,1),(3,1),(3,2)\}$

Exercise A.10 on page A-7: Besides the empty and the universal relations we also have the identity relation:

and also the following, each one with a version in which the isolated point can be not reflexive:



Exercise A.11

Exercise A.12

Exercise A.13 on page A-8: These are the same as the ones given in the solution to exercise A.10, except the empty relation.

Exercise A.14 on page A-8:

R$\not$TS = $\{(2,1),(1,2)\}$;

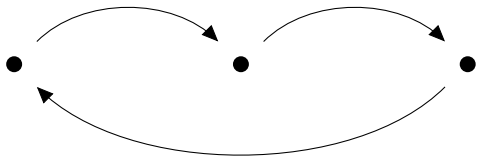
RT$\not$S = $\{(1,1),(2,2),(3,3),(1,2),(2,1),(2,3),(3,2)\}$;

RTS$\not$ = $\{(1,1),(2,2),(1,2)\}$.

What these examples show is that reflexivity does not follow from transitivity and symmetry, that transitivity does not follow from reflexivity and symmetry, and that symmetry does not follow from reflexivity and transitity.

Exercise A.15 on page A-8: Base case: Proved in the text. Induction Hypothesis: If $|M| < n$ then $\exists x \exists y(Rxy \wedge Ryx)$ or $\exists x \forall y \neg Rxy$ or $\exists x \exists y \exists z(Rxy \wedge Ryx \wedge \neg Rxz)$. Induction step: take an arbitrary model such that $|M| = n - 1$ and build M' with $dom(M') = dom(M) \cup \{a\}$, such that $R' = R \cup R^a$ where $R^a \subseteq R^A$ with $R^A = \{(x,y) \mid x \in dom(M), y = a\} \cup \{(x,y) \mid y \in dom(M), x = a\} \cup \{(x,y) \mid x = a, y = a\}$. If $\exists x \exists y \exists z(Rxy \wedge Ryx \wedge \neg Rxz)$ then $\exists x \exists y \exists z(R'xy \wedge R'yx \wedge \neg R'xz)$ and we are done. If $\exists x \exists y(Rxy \wedge Ryx)$ then $\exists x \exists y(R'xy \wedge R'yx)$ and we are done. If $\neg \exists x \exists y(Rxy \wedge Ryx)$ and $\neg \exists x \exists y \exists z(Rxy \wedge Ryx \wedge \neg Rxz)$ and $\exists x \forall y \neg Rxy$ then if for some $x \in M$ that has no successor we have $\neg R'xa$ we are done. Suppose that for all $x \in M$ that have no successors we have $R'xa$, then in order for M' to be serial we need to have $R'ay$ for some $y \in M'$. If Raa then asymmetry is violated and we are done. Take an arbitrary $z \in M'$ such that $R'za$ and $z \neq a$ then if we have $R'az$, asymmetry is again violated and we are done. If, for arbitrary $z, y \in M'$ we have $R'za$, $R'ay$ and $y \neq z$ we have to consider two cases. If $\neg Ryz$ then transitivity is violated and we are done. In case Ryz then we have $R'ay$, $R'yz$, and $R'za$. If we satisfy transitivity of R' then we must also have $R'aa$ which in turn violates the asymmetry condition.

Exercise A.16 on page A-9: (1)



(2)

(3)

Exercise A.17 on page A-10:

Exercise A.18 on page A-11:

Exercise A.19 on page A-11:

Exercise A.20 on page A-12: We prove that for all natural numbers n:

$$\sum_{k=0}^{n} 2k = n(n+1).$$

Basis. For $n = 0$, we have $\sum_{k=0}^{0} 2k = 0 = 0 \cdot 1$, so for this case the statement holds.
Induction step. We assume the statement holds for some particular natural number n and we show that it also holds for $n + 1$. So assume $\sum_{k=0}^{n} 2k = n(n+1)$. This is the **induction hypothesis**. We have to show: $\sum_{k=0}^{n+1} 2k = (n+1)(n+2)$.

We have:

$$\sum_{k=0}^{n+1} 2k = \sum_{k=0}^{n} 2k + 2(n+1).$$

Now use the induction hypothesis:

$$\sum_{k=0}^{n+1} 2k = \sum_{k=0}^{n} 2k + 2(n+1) \stackrel{ih}{=} n(n+1) + 2(n+1) = n^2 + 3n + 2 = (n+1)(n+2).$$

This settles the induction step. It follows that for all natural numbers n, $\sum_{k=0}^{n} 2k = n(n+1)$.

Bibliography

[Bir98] R. Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, 1998.

[Cal88] Italo Calvino. *Six Memos for the Next Millennium*. Harvard University Press, 1988.

[DI07] Hal Daume III. *Yet another Haskell tutorial*, wikibooks.org edition, 2007.

[DvE04] K. Doets and J. van Eijck. *The Haskell Road to Logic, Maths and Programming*, volume 4 of *Texts in Computing*. College Publications, London, 2004.

[EU10] Jan van Eijck and Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010.

[HFP96] P. Hudak, J. Fasel, and J. Peterson. A gentle introduction to Haskell. Technical report, Yale University, 1996. Online version: `http://www.haskell.org/tutorial/`.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):567–580, 583, 1969.

[Hut07] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[Jac00] Daniel Jackson. Automating first-order relational logic. *ACM SIGSOFT Software Engineering Notes*, 25(6):130–139, 2000.

[Jac06] Daniel Jackson. *Software Abstractions; Logic, Language and Analysis*. MIT Press, 2006.

[OSG08] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly, 2008.

[PJ03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries; The Revised Report*. Cambridge University Press, 2003.

[Rus05] B. Russell. On denoting. *Mind*, 14:479–493, 1905.

[Tho99] S. Thompson. *Haskell: the craft of functional programming (second edition)*. Addison Wesley, 1999.

[Tur36] A.M. Turing. On computable real numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.