Logic and Verification

Prof. Dr. Jürgen Dix

What Song the Syrens sang,
or what name Achilles assumed
when he hid himself among women,
though puzzling Questions,
are not beyond all conjecture.

Sir Thomas Browne

Acknowledgment

This course grew out of my former AI course (held from 2004–2014 at TU Clausthal). The main new parts are the chapters about model checking, PROLOG, and answer set programming.
The author gratefully acknowledges material and slides (for the Prolog part) provided by Nils Bulling used in his BSc course TI1906 Logic-Based Artificial Intelligence at TU Delft in the summer term 2015.
Many thanks also to Tobias Ahlbrecht who helped with the exercises.

Time and place: Monday, Tuesday 10–12.
Exercises: See schedule (7 exercises in total).

Website

www.in.tu-clausthal.de/index.php?id=cig_logic-2018

Visit regularly!

There you will find important information about the lecture, documents, exercises et cetera.

Organization: Tobias Ahlbrecht;
Exercise class: A. Mantel, J. Schwede;
Exam: tbd

What is this course about?

This course is about logic based formal methods and how to use them for verification in computer science.

It rigorously defines the language and semantics of

- sentential logic (SL) and
- first-order logic (FOL).

We introduce the notion of a model, and show how hardware and software systems can be modelled within this framework. We also consider

- semantic entailment vs. syntactic derivability

of formulae and how these two notions are related through correctness and completeness. Completeness is formally proved for SL and extensively discussed for FOL (in the form of refutation completeness).

What is this course about? (cont.)

Both logics are applied to important verification problems:

- SL and its extension LTL for verifying linear temporal properties in concurrent systems, and
- FOL for verifying input/output properties of programs (the Hoare calculus).

We show how the **resolution calculus** for FOL leads to **PROLOG**, a programming language based on FOL.

We also show that SL leads to a declarative version of **PROLOG**: Answer Set Programming (**ASP**), that can be used to solve problems expressible on the second level of the polynomial hierarchy.

While **ASP** is not a full-fledged programming language, it can be used for many naturally ocurring problems and it allows to model them in a purely declarative way.

Lecture Overview

1. Introduction (1 lecture)
2. Sentential Logic (SL) (3 lectures)
3. Verification I: Reactive Systems (3 lectures)
4. First-Order Logic (FOL) (2,5 lectures)
5. Verification II: Hoare Calculus (2,5 lectures)
6. From FOL to PROLOG (2 lectures)
7. PROLOG (3 lectures)

1. Introduction

Content of this chapter (1):

Defining AI: There are several interpretations of AI. They lead to scientific areas ranging from Cognitive Science to Rational Agents.

History: We discuss some important philosophical ideas in the last three millennia and touch some events that play a role in later chapters (syllogisms of Aristotle, Ars Magna).

Content of this chapter (2):

Logic-Based AI since 1958: AI came into being in 1956-1958 with John McCarthy. We give a rough overview of its successes and failures.

Rational Agent: The viewpoint to consider an agent as a rationally acting entity.

Content of this chapter (3):

Model checking: Many problems can be modelled with finite state systems and solved by checking whether (1) a given model has a certain property, (2) a theory is satisfiable or (3) a formula is valid.

Verification: Programs are based on an infinite state space and thus require other methods, similar to the Hoare calculus.

1.1 What Is AI?

"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense" (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

Table 1: Several Definitions of AI

1. Cognitive science

2. "Socrates is a man. All men are mortal. Therefore Socrates is mortal."
   (Famous *syllogisms* by Aristotle.)

   (1) **Informal description** $\rightsquigarrow$
   (2) **Formal description** $\rightsquigarrow$
   (3) Problem solution

   (2) is often problematic due to under-specification

   (3) is deduction (correct inferences): only enumerable, but **not decidable**

3. Turing Test:

`http://cogsci.ucsd.edu/~asaygin/tt/ttest.html`

`http://www.loebner.net/Prizef/loebner-prize.html`

- Standard Turing Test
- Total Turing Test

Turing believed in 1950:

In 2000 a computer with 10^9 memory-units could be programmed such that it can chat with a human for 5 minutes and pass the Turing Test with a probability of 30 %.

4. In item **2.** *correct* inferences were mentioned.

   Often not enough information is available in order to act in a way that makes sense (to act in a provably correct way).
   $\rightsquigarrow$ Non-monotonic logics.

   The world is in general under-specified. It is also impossible to act rationally without *correct* inferences: **reflexes**.

The year 1943:

McCulloch and **W. Pitts** drew on three sources:

1. physiology and function of neurons in the brain,
2. propositional logic due to **Russell/Whitehead**,
3. Turing's theory of computation.

Model of artificial, connected neurons:

- Any computable function can be computed by some network of neurons.
- All the logical connectives can be implemented by simple net-structures.

The year 1956:

Two-month workshop at Dartmouth organized by **McCarthy**, **Minsky**, **Shannon** and **Rochester**.

Idea:

Combine knowledge about automata theory, neural nets and the studies of intelligence (10 participants)

Newell und **Simon** show a reasoning program, the Logic Theorist, able to prove most of the theorems in Chapter 2 of the *Principia Mathematica* (even one with a shorter proof).

But the *Journal of Symbolic Logic* rejected a paper **authored by *Newell, Simon and Logical Theorist***.

Newell and Simon claim to have solved the venerable mind-body problem.

The year 1958: Birthyear of AI

The term

Artificial Intelligence

is

proposed as the name of the new discipline.

McCarthy joins **MIT** and develops:

1. *Lisp*, the dominant AI programming language
2. *Time-Sharing* to optimize the use of computer-time
3. *Programs with Common-Sense*.

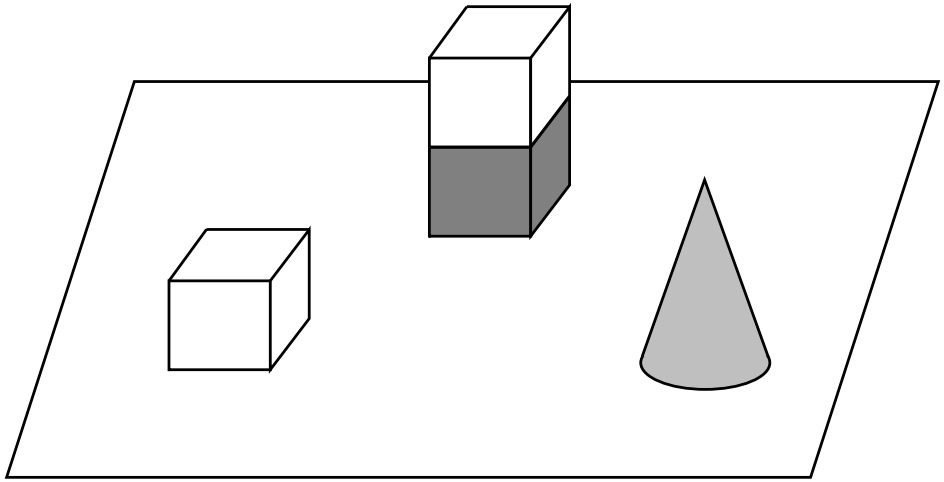
   Advice-Taker: A hypothetical program that can be seen as the first complete AI system. Unlike others it embodies general knowledge of the world.

The years 1960-1966:

McCarthy concentrates on knowledge-representation and reasoning in formal logic ($\rightsquigarrow$ Robinson's Resolution, $\rightsquigarrow$ Green's Planner, $\rightsquigarrow$ Shakey).

Minsky is more interested in getting programs to work and focusses on special worlds, the Microworlds.

Blocksworld is the most famous microworld.



From the 70'ies to the 90'ies

'73: **PROLOG (Colmerauer, Kowalski)**

'74: Relational databases, SQL (**Codd**)

81-91: Fifth generation project (Japan)

'91: **Dynamic Analysis and Replanning Tool (DART)** paid back DARPA's investment in AI during the last 30 years.

From the 90'ies until today

'93: Nine Men's Morris ("Mühle") solved (Gasser (ETH)). #: 10^{10}.

'97: IBM's Deep Blue wins against Kasparow. #: 10^{46}.

'98: NASA's remote agent program: **Deep Space 1**. Many modules have been model-checked.

'07: Checkers ("Dame") solved: **Chinook** by J. Schaeffers. #: 10^{20}.

'10: IBM's Watson winning Jeopardy

`http://www-05.ibm.com/de/pov/watson/`

'16: Google's AlphaGo winning **4: 1** against **Lee Sedol** in a professional Go match.
#: 10^{170}.

`https://de.wikipedia.org/wiki/AlphaG`

'17: Google's AlphaZero as a program learning arbitrary games by itself. Even Chess, as one

1.2 Logic: From Plato To Zuse

450 BC : **Plato, Socrates, Aristotle**

Sokr.: *"What is characteristic of piety which makes all actions pious?"*
Aris.: *"Which laws govern the rational part of the mind?"*

800 : **Al Chwarizmi** (Arabia): Algorithm

1300 : **Raymundus Lullus**: Ars Magna

1646–1716: **G. W. Leibniz:**
Materialism, uses ideas of Ars Magna to build a machine for simulating the human mind

1805 : **Jacquard:** Loom

1815–1864: **G. Boole:**
Formal language,
Logic as a mathematical discipline

1792–1871: **Ch. Babbage:**
Difference Engine: Logarithm-tables
Analytical Engine: with addressable memory,
stored programs and conditional jumps

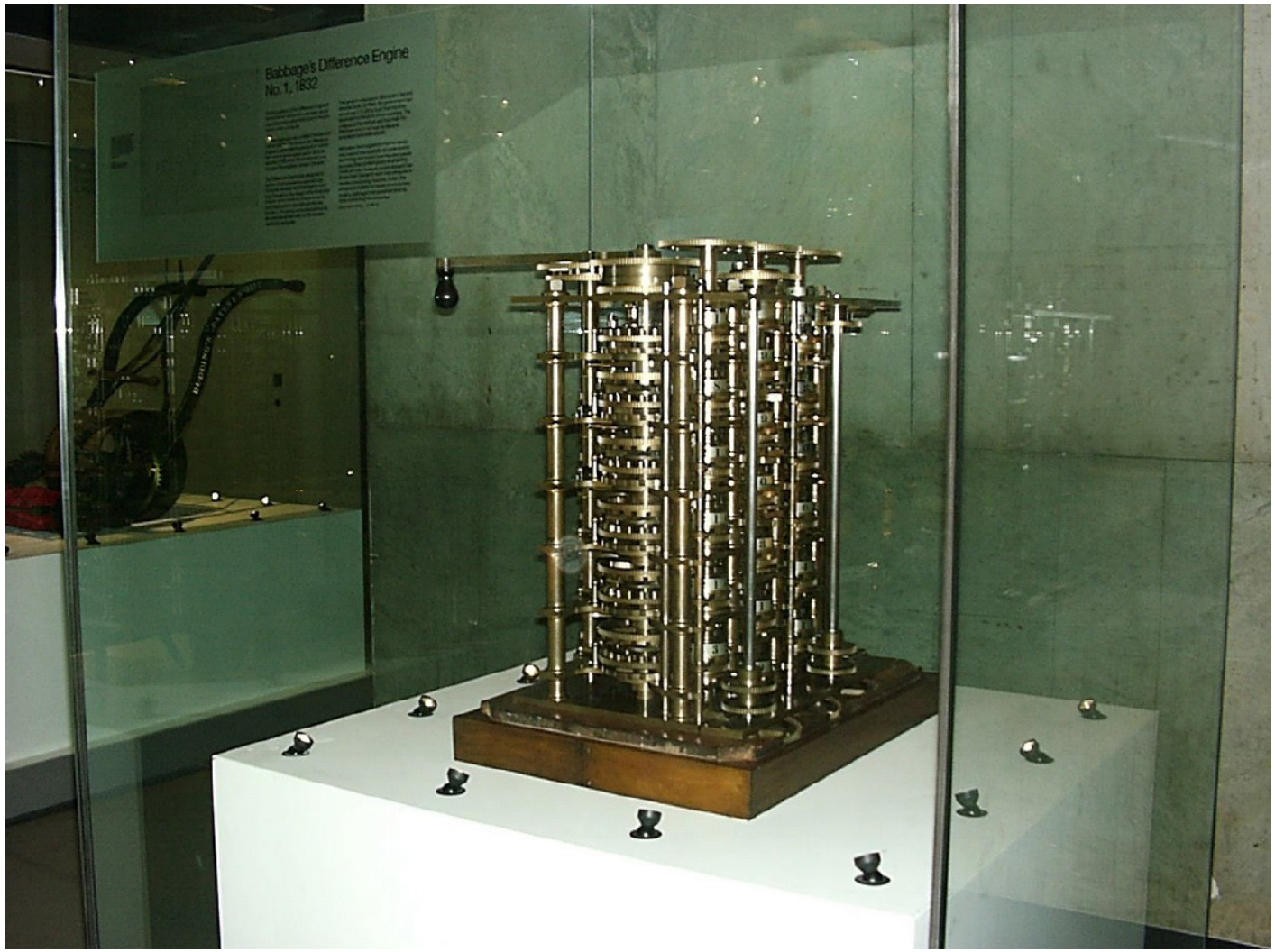


Figure 1.1: Reconstruction of Babbage's difference engine.

1848–1925 : G. Frege: Begriffsschrift

2-dimensional notation for PL1

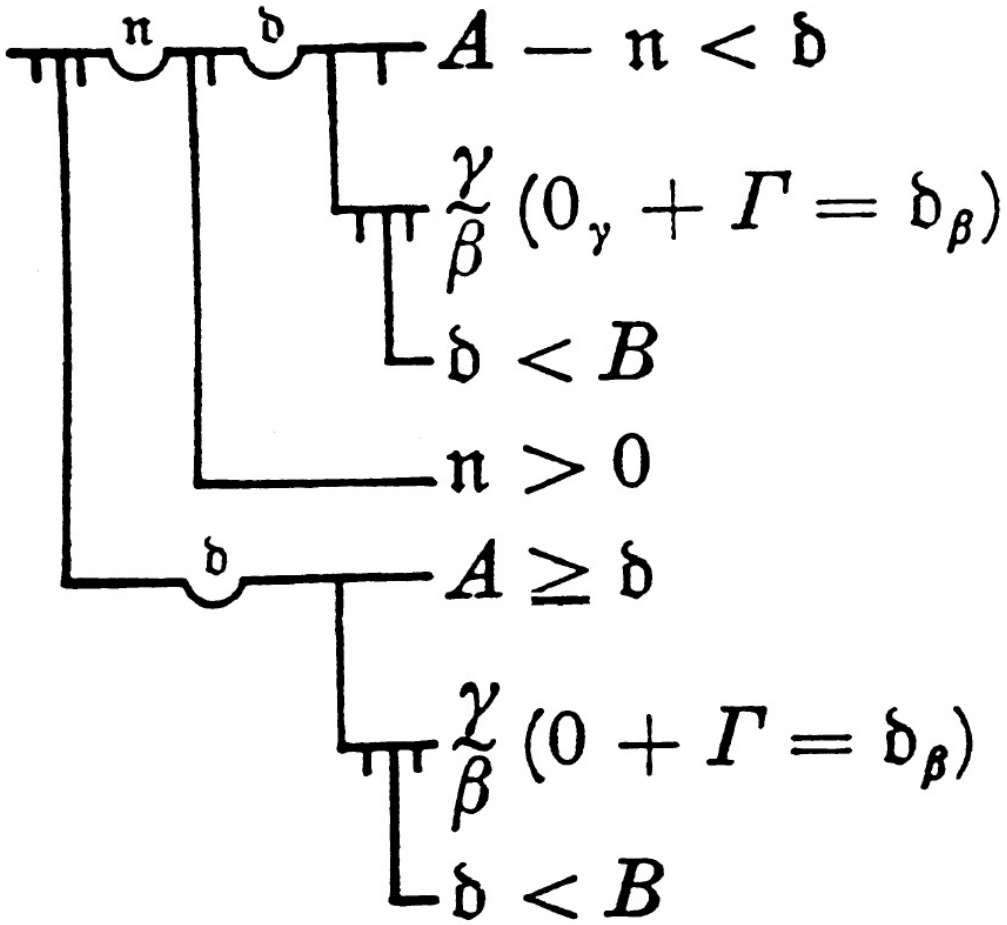


Figure 1.2: A formula from Frege's Begriffsschrift.

1862–1943: **D. Hilbert:**
Famous talk 1900 in Paris: 23 problems
23rd problem: The Entscheidungsproblem

1872–1970: **B. Russell:**
1910: Principia Mathematica
Logical positivism, *Vienna Circle* (1920–40)

1906–1978: **K. Gödel:**
Completeness theorem (1930)
Incompleteness theorem (1930/31)
Unprovability of theorems (1936)

1912–1954: **A. Turing:**
Turing-machine (1936)
Computability

1938/42: First operational programmable computer: Z 1
Z 3 by **K. Zuse** (Deutsches Museum)
with floating-point-arithmetic.
Plankalkül: First high-level programming language

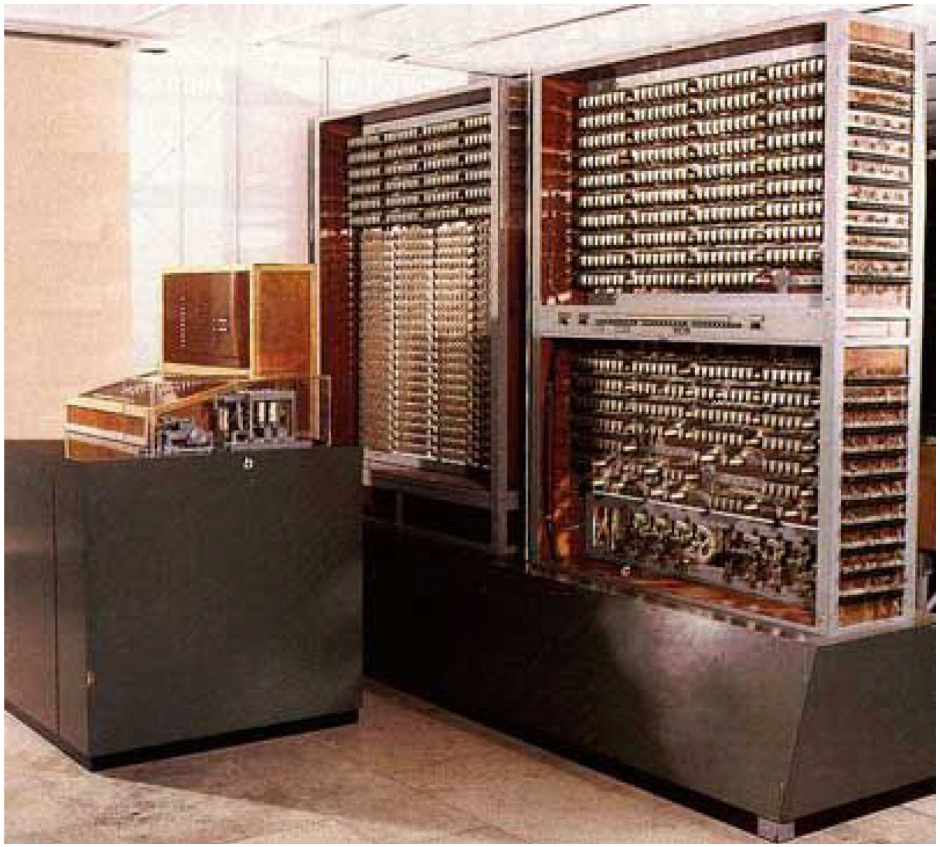


Figure 1.3: Reconstruction of Zuse's Z3.

1940 : First computer *"Heath Robinson"* built to decipher German messages (Turing)
1943 *"Collossus"* built from vacuum tubes

1940–45: **H. Aiken:** develops *MARK I, II, III*.
ENIAC
First general purpose electronic computer

1952 : IBM: *IBM 701*, first computer to yield profit (Rochester et alii)

1948: First stored program computer (*The Baby*)

Tom Kilburn (Manchester)

Manchester beats Cambridge by 3 months

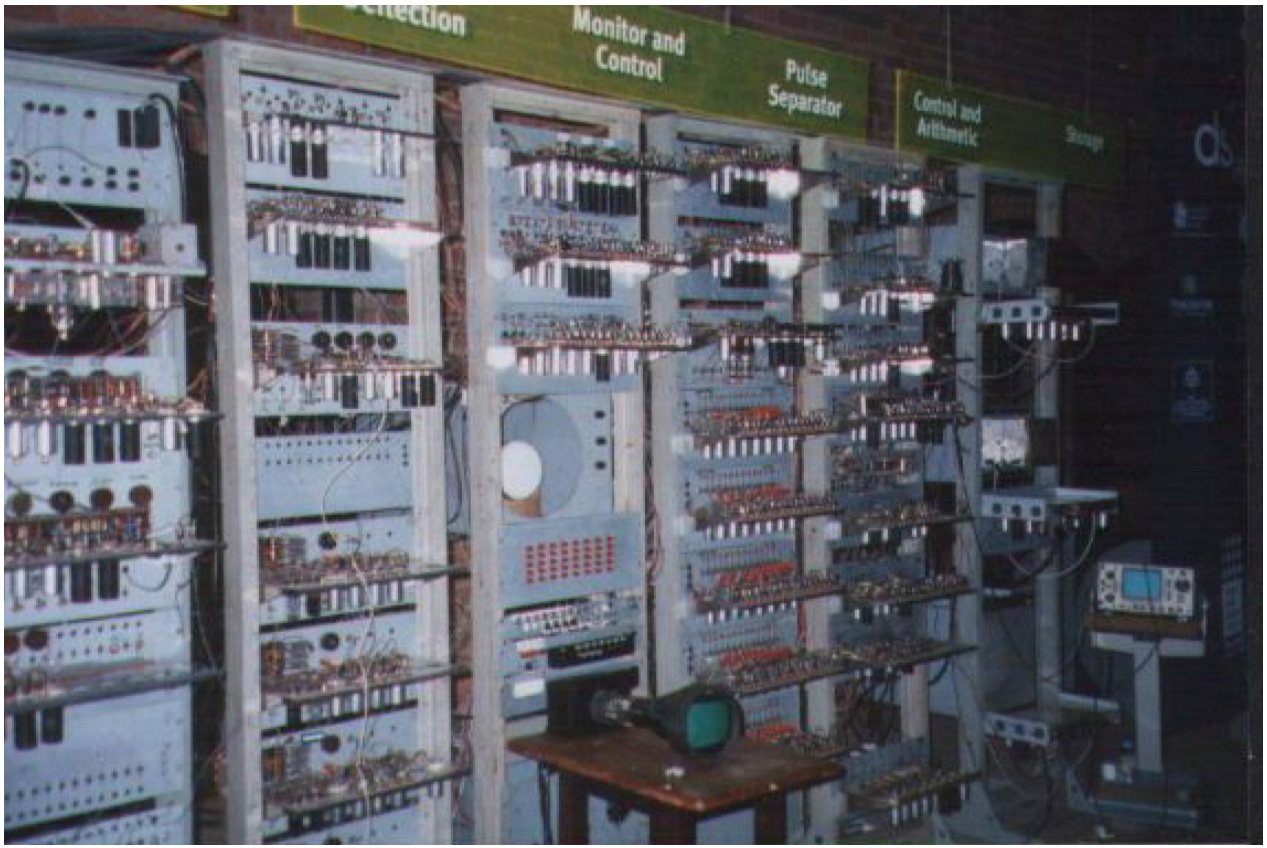


Figure 1.4: Reconstruction of Kilburn's baby.

First program run on The Baby in 1948:

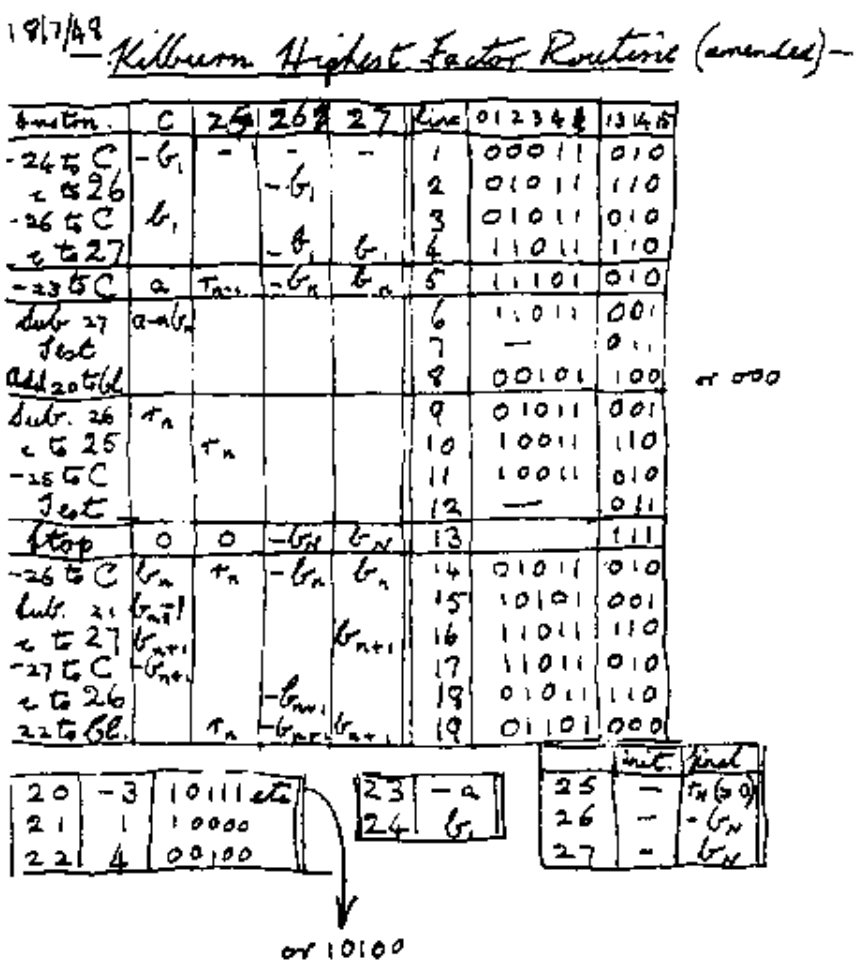


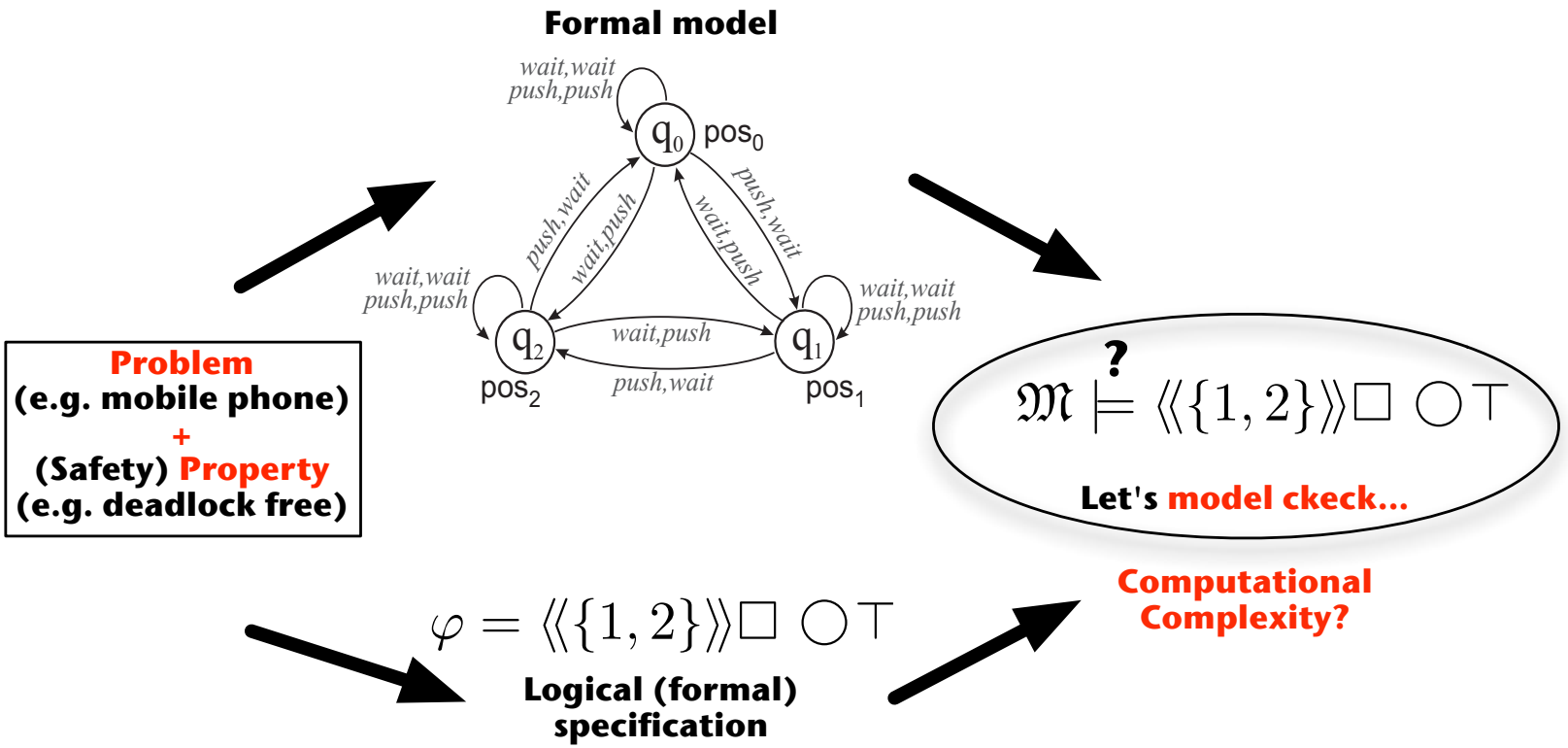
Figure 1.5: Reconstruction of first executed program on The Baby.

1.3 Verification

Model Checking Technique

Errors are expensive: Ariane 5 missile crash, . . .

Model checking provides means to **detect such erros**!



- **Model checking** refers to the problem to determine whether a given formula φ is satisfied in a state q of model $\mathcal{M}$.
- **Local model checking** is the decision problem that determines membership in the set

$$\mathsf{MC}(\mathcal{L}, \mathsf{MOD}, \models) =_{def}$$
$$\{(\mathcal{M}, q, \varphi) \in \mathsf{MOD} \times \mathcal{L} \mid \mathcal{M}, q \models \varphi\},$$

  where
  - $\mathcal{L}$ is a **logical language**,
  - MOD is a **class of (pointed) models** for $\mathcal{L}$ (i.e. a tuple consisting of a model and a state), and
  - $\models$ is a **semantic satisfaction relation** compatible with $\mathcal{L}$ and MOD.

Example 1.1 (Concurrency)

We assume the following three processes which run independently and parallel and share the integer variable x.

Inc	**while**	t	**do**	**if**	$x \lneq 200$	**then**	$x := x + 1$	**fi**	**do**
Dec	**while**	t	**do**	**if**	$x \gneq 0$	**then**	$x := x - 1$	**fi**	**do**
Reset	**while**	t	**do**	**if**	$x == 200$	**then**	$x := 0$	**fi**	**do**

Does this ensure that the value of x is always between (possibly including) 0 and 200?

Example 1.2 (Deadlock: Dining philosophers)

Five philosophers are sitting at a round table with a bowl of rice in the middle. They can just do three things: (1) thinking, (2) eating and (3) waiting to do (1) or (2). The problem is that they can eat only by using two chopsticks and eating rice out of the bowl.

But between two adjacent philosophers there is only one chopstick. Therefore, at any point in time, only two philosophers can eat concurrently.
If they all take the chopstick on their left, a deadlock occurs.

What is a fair synchronization so that there is no deadlock and no philosopher is starving?

How can this be modelled and the two properties formally verified?

Dining philosophers

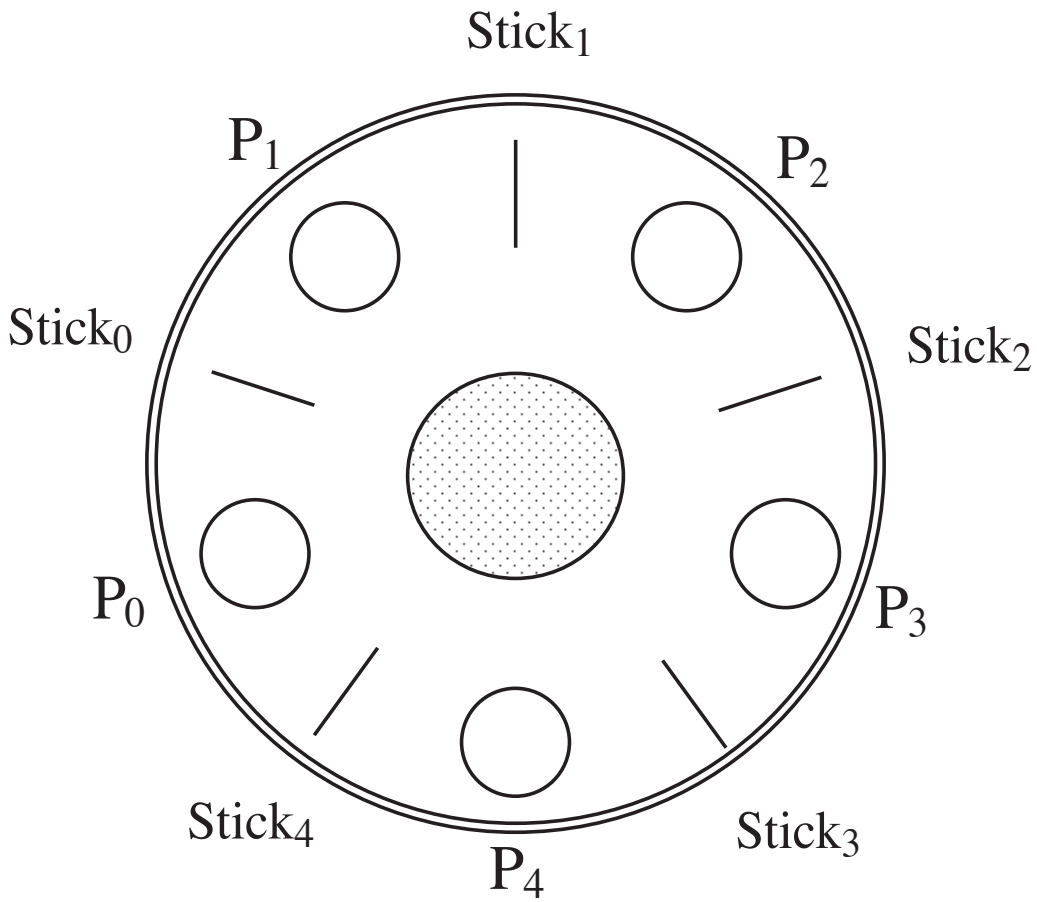


Figure 1.6: Dining philosophers.

Example 1.3 (What does program $\mathsf{P}(x)$ calculate?)

The input, x, ranges over integers.

```
y := 1;
z := 0;
while z ≠ x {
    z := z + 1;
    y := y · z
}
```

Example 1.4 (What do these programs calculate?)

```
int foo1(int n) {
local int k, int j;
k := 0;
j := 1;
while (k ≠ n){
    k := k + 1;
    j := 2 * j
} return(j) }
```

```
int foo2(int n) {
local int k, int j;
k := 0;
j := 1;
while (k ≠ n){
    k := k + 1;
    j := 2 + j
} return(j) }
```

1.4 References

- This lecture covers several areas, from classical logic, via classical verification techniques to the programming language **PROLOG**.
- There exist many good textbooks for all these areas.
- However, each has its own particular viewpoint and therefore introduces the notions in a different way. In particular concerning logic, many other, equivalent, approaches are possible.
- A theorem in our approach can be a definition in another and vice versa.
- **To solve the exercises, strictly stick to the exposition on the slides**, and what has been lectured until then.

Literature I

Christel Baier and Joost-Pieter Katoen.
Principles of model checking.
May 2008. Cambridge: MIT Press.

Rafael Bordini and Jürgen Dix.
Programming Multiagent Systems.
In G. Weiss, editor, *Multiagent systems*, pages 587–640, 2013. MIT-Press.

F. Dalpaiz and J. Dix and M. B. van Riemsdijk
Engineering Multi-Agent systems.
Second International Workshop (Paris 2014), revised and selected papers.
Springer LNAI series 8758, 2015.

Jürgen Dix and Michael Fisher.
Verifying Multi-Agent Systems.
In G. Weiss, editor, *Multiagent systems*, pages 641–693, 2013. MIT-Press.

Nils Bulling and Jürgen Dix and Wojciech Jamroga.
Model Checking Logics of Strategic Ability: Complexity.
In *Specification and Verification of Multi-Agent Systems*, pages 125–158, 2010. Springer.

Amir Pnueli.
The Temporal Logic of Programs.
In *Proceedings of the 18th IEEE Symp. on Foundations of Computer Science*, 1977.

A. Prasad Sistla and Edmund M. Clarke.
The complexity of Propositional Linear Temporal Logic.
In *Journal of the ACM*, volume32, no. 3, pages 733–749, 1985.

William Clocksin and Christopher S. Mellish.
Programming in PROLOG.
Springer Science & Business Media, 2003.

Michael Huth and Mark Ryan.
Logic in Computer Science: Modelling and reasoning about systems.
2000. Cambridge University Press.

D'Silva et al.
A Survey of Automated Techniques for Formal Software Verification.
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, voulme 27, no. 7, pages 1165–1178, 2008.

2. Sentential Logic (SL)

Content of this chapter (1):

Logic: Logics can be used to describe the world and how it evolves. We start with propositional logic as the fundamental basis. All important notions (model, theory, validity, deducibility, derivability, calculus, model checking, counterexamples) are rigorously introduced and are the basis for first-order logic.

Examples: We illustrate the use of SL with three examples: The game of Sudoku, the Wumpus world and one of the weekly puzzles in the newspaper *Die Zeit*. Finding a solution for these problems is reduced to computing models of a certain theory.

Content of this chapter (2):

Calculi for SL: While it is nice to describe the world, we also want to draw conclusions about it. Therefore we have to derive new information and deduce statements, that are not explicitly given, but somehow contained in the description. We introduce a Hilbert-type calculus and the notion of proof in a purely syntactical way.

While Hilbert-type calculi are easily motivated, they cannot be efficiently implemented. Robinson's resolution calculus is much more suited and will be later extended to FOL.

2.1 Motivation

Folklore (1)

We all know the laws of Boole in algebra (**Boolean Algebra**)

$$
\begin{array}{rcl}
A \cap (B \cup C) & = & (A \cap B) \cup (A \cap C) \\
A \cup (B \cap C) & = & (A \cup B) \cap (A \cup C) \\
\overline{(A \cap B)} & = & \overline{A} \cup \overline{B} \\
\overline{(A \cup B)} & = & \overline{A} \cap \overline{B} \\
\overline{\overline{A}} & = & A \\
A \cup (B \cap A) & = & A \\
A \cap (B \cup A) & = & A \\
A \cup \overline{A} & = & U \\
A \cap \overline{A} & = & \emptyset \\
\overline{\emptyset} & = & U \\
\overline{U} & = & \emptyset
\end{array}
$$

Expressions formed using $\cup, \cap, , (,), U, \emptyset$ are called Boolean expressions (U is the **universe**). $\rightsquigarrow$ **blackboard 2.1**

Folklore (2)

- Can we show, using these laws, that each Boolean expression (formed using $A, B, C, \ldots$ and $\cap, \cup, (,), \overline{\ }$) can be written as **an intersection of unions** of $A, \overline{A}, B, \overline{B}, C, \overline{C} \ldots$?
- Can we show, using these laws, that each Boolean expression (formed using $A, B, C, \ldots$ and $\cap, \cup, (,), \overline{\ }$) can be written as **a union of intersections** of $A, \overline{A}, B, \overline{B}, C, \overline{C} \ldots$?

$\rightsquigarrow$ **blackboard 2.2**

Folklore (3)

With the correspondence

$=$	corresponds to	$\leftrightarrow$
$\overline{}$	corresponds to	$\neg$
$\cap$	corresponds to	$\wedge$
$\cup$	corresponds to	$\vee$
$\emptyset$	corresponds to	false
U	corresponds to	true

we immediately get valid formulae in SL.

$\rightsquigarrow$ **blackboard 2.3**

Example 2.1 (Tweety and Friends)

We want to throw a party for Tweety, his friend Gentoo and Tux. But they have different circles of friends and dislike some. Tweety tells you that he would like to see either his friend the King or not to meet Gentoo's Adelie. But Gentoo proposes to invite Adelie or Humboldt or both. Tux, however, does not like Humboldt and the King too much, so he suggests to exclude at least one of them.

- Can we **represent** this using sentential logic?
- What do we *gain* by doing that?
- **Exactly how many** solutions are there?

Propositional Symbols

k means invite The King	$\neg k$ means exclude The King
a means invite Adelie	$\neg a$ means exclude Adelie
h means invite Humboldt	$\neg h$ means exclude Humboldt

Problem Formalized

Tweety: $k \vee \neg a$ means "invite The King or exclude Adelie", but not both: $\neg(k \wedge \neg a)$,

Gentoo: $a \vee h$ means "invite Adelie or Humboldt or both",

Tux: $\neg h \vee \neg k$ means "exclude Humboldt or The King or both".

Resulting Formula

Can we make the following formula true?

$$\varphi_{\text{party}}: \quad (k \vee \neg a) \wedge \neg(k \wedge \neg a) \wedge (a \vee h) \wedge (\neg h \vee \neg k)$$

There seems to be only one method of solution:
to check all possibilities.

$$\varphi_{\text{party}} : (k \vee \neg a) \wedge \neg(k \wedge \neg a) \wedge (a \vee h) \wedge (\neg h \vee \neg k)$$

k	a	h	$k \vee \neg a$	$\neg(k \wedge \neg a)$	$a \vee h$	$\neg h \vee \neg k$	φ_{party}
1	1	1	1	1	1	0	0
1	1	0	1	1	1	1	1
1	0	1	0	1	1	0	0
1	0	0	1	0	0	1	0
0	1	1	0	1	1	1	0
0	1	0	0	1	1	1	0
0	0	1	1	1	1	1	1
0	0	0	1	1	0	1	0

- Therefore there are exactly two solutions.
- We can also deduce, that **either Humboldt or both The King and Adelie can be invited**. Later we call the rows in the table models (or valuations).

Complexity

Truth tables have 2^n rows, where n is the number of propositional constants. In the worst case, all have to be checked (**or maybe not???????**).

Overview

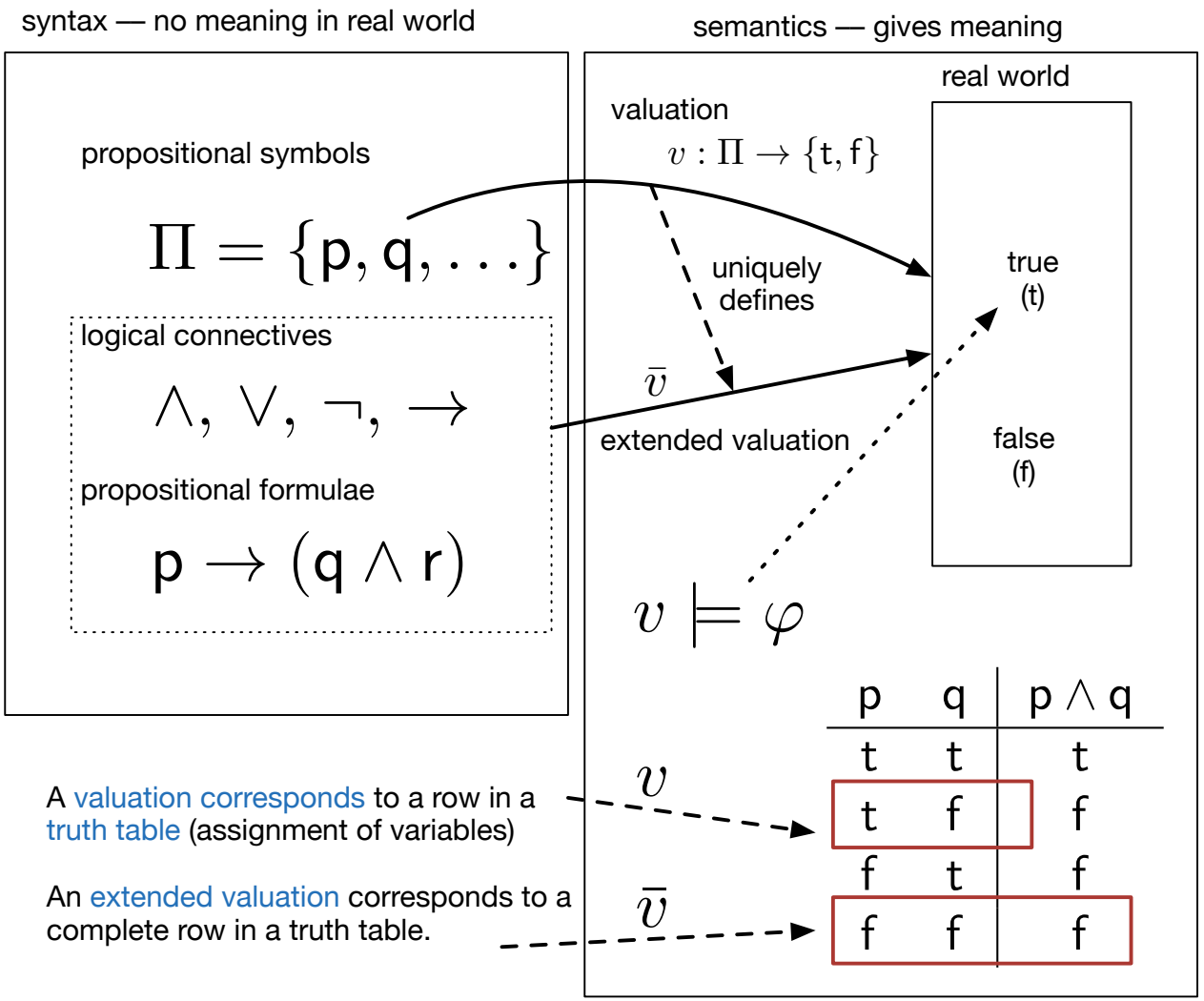


Figure 2.7: Syntax and Semantics for SL

2.2 Syntax and Semantics

Syntax: Language $\mathcal{L} = \mathcal{L}(\mathcal{P}rop)$

The sentential (propositional) language is built upon

propositional constants: $\square$, $p_1, p_2, p_3, \ldots$ (countably many). We also use $a, b, c, \ldots, p, q, r, \ldots$. The symbol $\square$ has a special meaning: it stands for **falsity**, the statement that is always false. This will become clear when we define the semantics of SL.

logical connectives: $\neg$ (unary), $\vee$ (binary), and

grouping symbols: $(,)$ for unique readability.

Often, for concrete applications, we consider only a finite, nonempty set of propositional constants and refer to it as $\mathcal{P}rop$ (e.g. $\mathcal{P}rop = \{a, p, q\}$). However, the symbol $\square$ is always present, we do not include it in the set $\mathcal{P}rop$.

Logical connectives are used to construct complex formulae from the propositional constants.

Definition 2.2 (Sentential Language $\mathcal{L}(\mathcal{P}rop)$)

Given a set $\mathcal{P}rop$, the signature, the sentential (or propositional) language $\mathcal{L}(\mathcal{P}rop)$ over $\mathcal{P}rop$ determines the set $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ of $\mathcal{L}(\mathcal{P}rop)$ formulae defined by

$$\varphi ::= \Box \mid \mathsf{p} \mid (\neg\varphi) \mid (\varphi \vee \varphi)$$

where $\mathsf{p} \in \mathcal{P}rop$.

Note that this only makes sense for finite $\mathcal{P}rop$. However, by adding symbols 0 and 1, one can easily produce infinitely many propositional symbols p_{100101} using finitely many grammar rules.

$\rightsquigarrow$ **blackboard 2.4**

We assume that $\mathcal{P}rop$ is fixed and omit it if clear from context: so we write $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}}$ instead of $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ to denote the set of all SL-formulae over $\mathcal{P}rop$. We freely omit parentheses in formulae for better readability.

What about **other connectives**? They are defined as macros.

Definition 2.3 (SL connectives as macros)

We define the following syntactic constructs as macros:

$$\begin{aligned} \top &=_{def} (\neg\Box) \\ \varphi \wedge \psi &=_{def} (\neg((\neg\varphi) \vee (\neg\psi))) \\ \varphi \to \psi &=_{def} ((\neg\varphi) \vee \psi) \\ \varphi \leftrightarrow \psi &=_{def} ((\varphi \to \psi) \wedge (\psi \to \varphi)) \end{aligned}$$

$\rightsquigarrow$ **blackboard 2.5**

Semantics

What should it mean that a $\mathcal{L}(\mathcal{P}rop)$-formula φ is true? Intuitively, we want the following:

$\top$		is always **t**
$\square$		is always **f**
$\neg\varphi$ is **t**	iff	φ is **f**
$\varphi \vee \psi$ is **t**	iff	φ or ψ (or both) are **t**
$\varphi \wedge \psi$ is **t**	iff	both φ and ψ are **t**
$\varphi \rightarrow \psi$ is **t**	iff	either φ is **f** or ψ is **t**
$\varphi \leftrightarrow \psi$ is **t**	iff	φ and ψ are both **t**, or both **f**

- Transitions systems (see next chapter) consist of a set of states and actions (transitions) between these states.
- States are determined by what holds true in them.
- A state will be uniquely determined by the facts that are true in it.
- These facts are exactly the elements in $\mathcal{P}rop$: the **propositional constants**.
- Therefore, we need to fix the truth values of the propositional constants.

Definition 2.4 (Valuation, truth assignment)

A valuation (or truth assignment) for a language $\mathcal{L}(\mathcal{P}rop)$ is a mapping $v : \mathcal{P}rop \to \{\mathsf{t}, \mathsf{f}\}$ from the set of propositional constants into the set $\{\mathbf{t}, \mathbf{f}\}$.

A valuation fixes the values of individual propositional constants.

We are particularly interested in the truth of **complex formulae** like $(\mathsf{p} \vee \mathsf{q}) \wedge \mathsf{r}$.

Semantics

The process of mapping a set of $\mathcal{L}$-formulae into $\{\mathbf{t}, \mathbf{f}\}$ is called semantics.

It suffices to state the semantics for the basic connectives.

Definition 2.5 (Semantics $v \models \varphi$, $\bar{v}$)

Let v be a valuation. We define inductively the notion of a formula $\varphi \in \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ being true or satisfied by v (notation: $v \models \varphi$):

$v \models \Box$ does not hold,

$v \models \mathsf{p}$ *if, by definition,* $v(\mathsf{p}) = \mathbf{t}$ and $\mathsf{p} \in \mathcal{P}rop$,

$v \models \neg\varphi$ *if, by definition,* not $v \models \varphi$,

$v \models \varphi \vee \psi$ *if, by definition,* $v \models \varphi$ or $v \models \psi$

We denote a set of $\mathcal{L}(\mathcal{P}rop)$-formulae by T. Given a set $T \subseteq \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ we write $v \models T$ *if, by definition,* $v \models \varphi$ for all $\varphi \in T$. We use $v \not\models \varphi$ instead of "not $v \models \varphi$".

Thus we have uniquely extended a valuation v to a mapping $\bar{v}$ from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ into the set $\{\mathbf{t}, \mathbf{f}\}$.

$\rightsquigarrow$ **blackboard 2.6**

Truth Tables

Truth tables are a conceptually simple way of working with SL (invented by Peirce in 1893 and used by Wittgenstein in 1910'ies (and in TLP)).

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p \rightarrow q$	$p \leftrightarrow q$
t	**t**	**f**	**t**	**t**	**t**	**t**
f	**t**	**t**	**t**	**f**	**t**	**f**
t	**f**	**f**	**t**	**f**	**f**	**f**
f	**f**	**t**	**f**	**f**	**t**	**t**

Definition 2.6 (Model, Theory, Tautology (Valid))

1. If $v \models \varphi$, we also call v (or $\bar{v}$) **a model of** φ. We write $MOD(T)$ for all models of a theory T:

$$MOD(T) =_{def} \{v : v \models T\}$$

2. A **theory** is any set $T \subseteq \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$.
   v satisfies T *if, by definition,* $v \models \varphi$ for all $\varphi \in T$.
3. A $\mathcal{L}$-formula φ is called **$\mathcal{L}$-tautology** (or simply called **valid**) if it is satisfied in all models: **for all models v it holds that $v \models \varphi$.**

We suppress the language $\mathcal{L}$ when obvious from context.

$\rightsquigarrow$ **blackboard 2.7**

Tweety revisited (1)

In Example 2.1 we can now state the following:

- The language we consider is $\mathcal{P}rop = \{\mathsf{a}, \mathsf{k}, \mathsf{h}\}$.
- The valuation v which makes $\mathsf{k}, \mathsf{a}, \mathsf{h}$ all true defines a structure which is not a model of the formula ϕ_{party}.
- The valuation which makes k, a true and h false, is a model of ϕ_{party}.
- ϕ_{party} can also be seen as the theory T_{party} consisting of the four formulae $\mathsf{k} \vee \neg\mathsf{a}$, $\neg(\mathsf{k} \wedge \neg\mathsf{a})$, $\mathsf{a} \vee \mathsf{h}$, and $\neg\mathsf{h} \vee \neg\mathsf{k}$.
- The formula ϕ_{party} is not a tautology, but $\mathsf{k} \to \mathsf{k}$ is one.

Definition 2.7 (Consequence Set $Cn(T)$, entailment)

A formula φ follows (semantically) from T (or is entailed from T) if for all models v of T (i.e. $v \models T$) also $v \models \varphi$ holds. We denote this by $T \models \varphi$.
We call

$$Cn_{\mathcal{L}}(T) =_{def} \{\varphi \in \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)} : T \models \varphi\},$$

or simply $Cn(T)$, the semantic consequence operator.

We also say φ can be deduced from T.
$\rightsquigarrow$ **blackboard back to 2.7**

Duality of *MOD* and *Cn*

Both *Cn* and *MOD* are defined on *theories* (sets of formulae). But the definition of *Cn* can easily be extended to also deal with sets of models. For a set M consisting solely of models, we define

$Cn(M) =_{def} \{\varphi \in \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)} : v \models \varphi \text{ for all } v \in M\}$.

MOD is obviously dual to *Cn*:

- $Cn(MOD(T)) = Cn(T)$,
- $MOD(Cn(T)) = MOD(T)$.

Tweety revisited (2)

Considering again Example 2.1 how does $Cn_{\mathcal{L}}(T_{\text{party}})$ look like?

- It is infinite.
- It contains all tautologies.
- It contains $(\mathsf{k} \wedge \mathsf{a} \wedge \neg \mathsf{h}) \vee \mathsf{h}$.
- Is $Cn_{\mathcal{L}}(T_{\text{party}}) = Cn_{\mathcal{L}}(\{(\mathsf{k} \wedge \mathsf{a} \wedge \neg \mathsf{h}) \vee \mathsf{h}\})$?
- Is $Cn_{\mathcal{L}}(T_{\text{party}}) = Cn_{\mathcal{L}}(\{(\mathsf{k} \wedge \mathsf{a} \wedge \neg \mathsf{h}) \vee (\neg \mathsf{k} \wedge \neg \mathsf{a} \wedge \mathsf{h})\})$?
- So there are different axiomatisations of $Cn_{\mathcal{L}}(T_{\text{party}})$.

  How to decide which are equivalent?

Duality

The duality principle is an important property in algebra and logic. It can be stated as follows.

Theorem 2.8 (Duality Principle)

We consider formulae ϕ, ψ over $\mathcal{P}rop$ built using only $\neg, \vee, \wedge, \Box, \top$. For such a formula φ, let $\mathsf{D}(\varphi)$ be the formula obtained by switching $\vee$ and $\wedge$, and $\Box$ and $\top$.

Then the following are equivalent:

- *ϕ is equivalent to ψ*
- *$D(\phi)$ is equivalent to $D(\psi)$.*

$\rightsquigarrow$ **blackboard 2.8**

Conjunctive/disjunctive Normal form

We consider formulae ϕ built over $\mathcal{P}rop$ from only $\neg, \vee, \wedge, \Box$.

Using the simple boolean rules, any formula ϕ can be written as a conjunction of disjunctions (conjunctive normal form)

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} \phi_{i,j}$$

The $\phi_{i,j}$ are just prop. constants or negated prop. constants from $\mathcal{P}rop$.

Definition 2.9 (Clauses, literals)

Constants and negated constants ($\neg$p) are called literals.
Disjunctions $\bigvee_{j=1}^{m_i} \phi_{i,j}$ built from literals $\phi_{i,j}$ are called clauses.

$\rightsquigarrow$ **blackboard 2.9**

Theorem 2.10 (Conjunctive/disjunctive normal form)

Any formula over $\mathcal{P}rop$ built using only $\neg, \vee, \wedge, \Box$ can be equivalently transformed into one with the same constants of the form

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} \phi_{i,j},$$

(the $\phi_{i,j}$ are, possibly negated, constants from $\mathcal{P}rop$, the empty disjunction is identified with $\Box$): conjunctive normal form.

Dually, any formula can be transformed into one of the form

$$\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} \phi_{i,j},$$

(the $\phi_{i,j}$ are, possibly negated, constants from $\mathcal{P}rop$, the empty conjunction is identified with $\top$): disjunctive normal form.

Normal forms

What are the normal forms of the following formulae?

- p,
- $\neg p$,
- $p \rightarrow p$,
- $\Box$,
- $\neg\neg(p \rightarrow \neg\Box)$.

Clauses

Normal form

Instead of working on arbitrary formulae, it is sometimes easier to work on finite sets of clauses.

This is without loss of generality (wlog): all formulae can be equivalently represented as a set (conjunction) of clauses.

This approach is much more suited for implementing a calculus (theorem proving) and we discuss it in detail in Subsection 2.5 from Slide 143 on.

Lemma 2.11 (Properties of $Cn(T)$)

The semantic consequence operator has the following properties:

1. T-expansion: $T \subseteq Cn(T)$,
2. Monotony: $T \subseteq T' \Rightarrow Cn(T) \subseteq Cn(T')$,
3. Closure: $Cn(Cn(T)) = Cn(T)$.

To entail (or deduce) something from a theory is important. But it is equally important to know that something is not entailed from a theory (i.e it does not follow from it): we emphasize this importance in the next lemma.

Lemma 2.12 ($\varphi \notin Cn$ (T), countermodel)

$\varphi \notin Cn(T)$ if and only if there is a model v with $v \models T$ and $v \models \neg\varphi$.

This model is often referred to as a countermodel for φ.

$\rightsquigarrow$ **blackboard 2.10**

Definition 2.13 (Completeness of a Theory T)

T is called complete if for each formula $\varphi \in \mathrm{Fml}^{\mathrm{SL}}_{\mathcal{L}(\mathcal{P}rop)}$: $T \models \varphi$ or $T \models \neg\varphi$ holds.

- Do not mix up this last condition with the property of a valuation (model) v: **each model is complete in the above sense**.
- A complete theory gives us a perfect description of the world (or state) we are in: we know everything!
- In most cases, however, our knowledge is incomplete.
- An incomplete theory leaves open many possibilities: many complete extensions of it, namely exactly the models of it.

Lemma 2.14 (Ex Falso Quodlibet)

The following are equivalent:

- T *has a model,*
- $Cn(T) \neq \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$.

Russel story: Derive from "$7 = 8$*" that you are the pope.*

$\rightsquigarrow$ **blackboard 2.11**

Tweety revisited (3)

We discuss the following statements.

- Is the theory $Cn_{\mathcal{L}}(T_{\text{party}})$ complete, does it have a model?
- What about the theory $Cn_{\mathcal{L}}(\{A, A \to B, \neg B\})$?
- Does each theory with a model posess a maximal extension that still has a model?
- How many such extensions are there for T_{party}?
- Is there a theory T such that $Cn(T) = \emptyset$? What about $Cn(\emptyset)$?

Who killed Tuna, the cat?

Example 2.15 (Tuna the cat)

There are two people, Jack and Bill. There is also a cat, called Tuna, and a dog with no name, owned by Bill. Tuna has been killed by either Jack or Bill, but it is not known by whom precisely.
All we know is that animal lovers do not kill animals and that dog owners are animal lovers.
So who killed Tuna?

Who killed Tuna, the cat? (2)

How can we formalize this story in SL? We need to choose $\mathcal{P}rop$ appropriately.

- We need to express that Tuna is a cat: cat_{Tuna}.
- Either Jack or Bill killed Tuna: $\text{killer\_of\_Tuna}_{\text{Bill}}$, $\text{killer\_of\_Tuna}_{\text{Jack}}$.
- $\text{dog\_owner}_{\text{Bill}}$, $\text{dog\_owner}_{\text{Jack}}$.
- $\text{animal\_lover}_{\text{Bill}}$, $\text{animal\_lover}_{\text{Jack}}$.

What about $\text{dog\_owner}_{\text{Tuna}}$, cat_{Bill}?

Who killed Tuna, the cat? (3)

Our theory T consists of:

- cat_{Tuna}.
- $\text{killer\_of\_Tuna}_{\text{Bill}} \vee \text{killer\_of\_Tuna}_{\text{Jack}}$,
  $\neg(\text{killer\_of\_Tuna}_{\text{Bill}} \wedge \text{killer\_of\_Tuna}_{\text{Jack}})$
- $\text{dog\_owner}_{\text{Bill}}$.
- $\text{dog\_owner}_{\text{Bill}} \rightarrow \text{animal\_lover}_{\text{Bill}}$,
  $\text{dog\_owner}_{\text{Jack}} \rightarrow \text{animal\_lover}_{\text{Jack}}$. What about
  $\text{dog\_owner}_{\text{Tuna}} \rightarrow \text{animal\_lover}_{\text{Tuna}}$.
- $\text{animal\_lover}_{\text{Jack}} \rightarrow \neg\text{killer\_of\_Animals}_{\text{Jack}}$,
  $\text{animal\_lover}_{\text{Bill}} \rightarrow \neg\text{killer\_of\_Animals}_{\text{Bill}}$.
- $\text{animal}_{\text{Tuna}} \wedge (\neg\text{killer\_of\_Animals}_{\text{Bill}}) \rightarrow \neg\text{killer\_of\_Tuna}_{\text{Bill}}$,
  $\text{animal}_{\text{Tuna}} \wedge (\neg\text{killer\_of\_Animals}_{\text{Jack}}) \rightarrow \neg\text{killer\_of\_Tuna}_{\text{Jack}}$.

What is missing?

2.3 Some examples

Since some time, Sudoku puzzles are becoming quite famous.

			1	5		4	6	7
5	2	6		4				
4			9					
		4			5		2	6
6	9				7		1	
					1	8	3	9
	8							
	4	3	5	7	8	6		
			3				4	

Table 2: A simple Sudoku (S_1)

Can they be solved with sentential logic?

Idea: Given a Sudoku-Puzzle S, construct a language $\mathcal{P}rop_{\text{Sudoku}}$ and a theory $T_S \subseteq \text{Fml}^{\text{SL}}_{\mathcal{L}(}\mathcal{P}rop_{\text{Sudoku}})$ such that

$$MOD(T_S) = \text{ Solutions of the puzzle S}$$

Solution

In fact, we construct a theory T_{Sudoku} and for each (partial) instance of a 9×9 puzzle S a particular theory T_S such that

$$MOD(T_{\text{Sudoku}} \cup T_S) = \{\mathcal{S} : \mathcal{S} \text{ is a solution of S}\}$$

We introduce the following prop. constants:

1. $\text{eins}_{i,j}$, $1 \leq i, j \leq 9$,
2. $\text{zwei}_{i,j}$, $1 \leq i, j \leq 9$,
3. $\text{drei}_{i,j}$, $1 \leq i, j \leq 9$,
4. $\text{vier}_{i,j}$, $1 \leq i, j \leq 9$,
5. $\text{fuenf}_{i,j}$, $1 \leq i, j \leq 9$,
6. $\text{sechs}_{i,j}$, $1 \leq i, j \leq 9$,
7. $\text{sieben}_{i,j}$, $1 \leq i, j \leq 9$,
8. $\text{acht}_{i,j}$, $1 \leq i, j \leq 9$,
9. $\text{neun}_{i,j}$, $1 \leq i, j \leq 9$.

This completes the language $\mathcal{P}rop_{\text{Sudoku}}$.

How many symbols are these?

We distinguished between the puzzle S and a solution $\mathcal{S}$ of it.

What is a model (or valuation) in the sense of Definition 2.6 (Slide 69)?

			1	5		4	6	7
5	2	6		4				
4			9					
		4			5		2	6
6	9				7		1	
					1	8	3	9
	8							
	4	3	5	7	8	6		
			3				4	

Table 3: How to construct a model $\mathcal{S}$?

We have to give our symbols a meaning (**the semantics**), i.e. a valuation v.

$$\text{eins}_{i,j} \textbf{ means } \langle i,j \rangle \text{ contains a } 1$$
$$\text{zwei}_{i,j} \textbf{ means } \langle i,j \rangle \text{ contains a } 2$$
$$\vdots$$
$$\text{neun}_{i,j} \textbf{ means } \langle i,j \rangle \text{ contains a } 9$$

To be precise: given a 9×9 square that is completely filled out, we define our valuation v as follows (for all $1 \leq i,j \leq 9$).

$$v(\text{eins}_{i,j}) = \begin{cases} \text{true, if } 1 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

$$v(\text{zwei}_{i,j}) = \begin{cases} \text{true, if } 2 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

$$v(\text{drei}_{i,j}) = \begin{cases} \text{true, if } 3 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

$$v(\text{vier}_{i,j}) = \begin{cases} \text{true, if } 4 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

etc.

$$v(\text{neun}_{i,j}) = \begin{cases} \text{true, if } 9 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

Therefore any 9×9 square can be seen as a model or valuation with respect to the language $\mathcal{L}_{\text{Sudoku}}$.

How does T_S look like?

$$
\begin{aligned}
T_S = \{ \quad & \mathsf{eins}_{1,4}, \mathsf{eins}_{5,8}, \mathsf{eins}_{6,6}, \\
& \mathsf{zwei}_{2,2}, \mathsf{zwei}_{4,8}, \\
& \mathsf{drei}_{6,8}, \mathsf{drei}_{8,3}, \mathsf{drei}_{9,4}, \\
& \mathsf{vier}_{1,7}, \mathsf{vier}_{2,5}, \mathsf{vier}_{3,1}, \mathsf{vier}_{4,3}, \mathsf{vier}_{8,2}, \mathsf{vier}_{9,8}, \\
& \vdots \\
& \mathsf{neun}_{3,4}, \mathsf{neun}_{5,2}, \mathsf{neun}_{6,9}, \\
\} &
\end{aligned}
$$

How should the theory T_{Sudoku} look like (s.t. models of $T_{\text{Sudoku}} \cup T_S$ correspond to solutions of the puzzle)?

First square: T_1

1. $\text{eins}_{1,1} \vee \ldots \vee \text{eins}_{3,3}$
2. $\text{zwei}_{1,1} \vee \ldots \vee \text{zwei}_{3,3}$
3. $\text{drei}_{1,1} \vee \ldots \vee \text{drei}_{3,3}$
4. $\text{vier}_{1,1} \vee \ldots \vee \text{vier}_{3,3}$
5. $\text{fuenf}_{1,1} \vee \ldots \vee \text{fuenf}_{3,3}$
6. $\text{sechs}_{1,1} \vee \ldots \vee \text{sechs}_{3,3}$
7. $\text{sieben}_{1,1} \vee \ldots \vee \text{sieben}_{3,3}$
8. $\text{acht}_{1,1} \vee \ldots \vee \text{acht}_{3,3}$
9. $\text{neun}_{1,1} \vee \ldots \vee \text{neun}_{3,3}$

The formulae on the last slide are saying, that

1. The number 1 must appear somewhere in the first square.
2. The number 2 must appear somewhere in the first square.
3. The number 3 must appear somewhere in the first square.
4. etc

Does that mean, that each number $1, \ldots, 9$ occurs exactly once in the first square?

No! We have to say, that each number occurs only once:

T_1':

1. $\neg(\text{eins}_{i,j} \wedge \text{zwei}_{i,j})$, $1 \leq i, j \leq 3$,
2. $\neg(\text{eins}_{i,j} \wedge \text{drei}_{i,j})$, $1 \leq i, j \leq 3$,
3. $\neg(\text{eins}_{i,j} \wedge \text{vier}_{i,j})$, $1 \leq i, j \leq 3$,
4. etc
5. $\neg(\text{zwei}_{i,j} \wedge \text{drei}_{i,j})$, $1 \leq i, j \leq 3$,
6. $\neg(\text{zwei}_{i,j} \wedge \text{vier}_{i,j})$, $1 \leq i, j \leq 3$,
7. $\neg(\text{zwei}_{i,j} \wedge \text{fuenf}_{i,j})$, $1 \leq i, j \leq 3$,
8. etc

How many formulae are these?

Second square: T_2

1. $\text{eins}_{1,4} \vee \ldots \vee \text{eins}_{3,6}$
2. $\text{zwei}_{1,4} \vee \ldots \vee \text{zwei}_{3,6}$
3. $\text{drei}_{1,4} \vee \ldots \vee \text{drei}_{3,6}$
4. $\text{vier}_{1,4} \vee \ldots \vee \text{vier}_{3,6}$
5. $\text{fuenf}_{1,4} \vee \ldots \vee \text{fuenf}_{3,6}$
6. $\text{sechs}_{1,4} \vee \ldots \vee \text{sechs}_{3,6}$
7. $\text{sieben}_{1,4} \vee \ldots \vee \text{sieben}_{3,6}$
8. $\text{acht}_{1,4} \vee \ldots \vee \text{acht}_{3,6}$
9. $\text{neun}_{1,4} \vee \ldots \vee \text{neun}_{3,6}$

And all the other formulae from the previous slides (adapted to this case): T_2'

The same has to be done for all 9 squares.

What is still missing:

Rows: Each row should contain exactly the numbers from 1 to 9 (no number twice).

Columns: Each column should contain exactly the numbers from 1 to 9 (no number twice).

First Row: $T_{\text{Row 1}}$

1. $\text{eins}_{1,1} \vee \text{eins}_{1,2} \vee \ldots \vee \text{eins}_{1,9}$
2. $\text{zwei}_{1,1} \vee \text{zwei}_{1,2} \vee \ldots \vee \text{zwei}_{1,9}$
3. $\text{drei}_{1,1} \vee \text{drei}_{1,2} \vee \ldots \vee \text{drei}_{1,9}$
4. $\text{vier}_{1,1} \vee \text{vier}_{1,2} \vee \ldots \vee \text{vier}_{1,9}$
5. $\text{fuenf}_{1,1} \vee \text{fuenf}_{1,2} \vee \ldots \vee \text{fuenf}_{1,9}$
6. $\text{sechs}_{1,1} \vee \text{sechs}_{1,2} \vee \ldots \vee \text{sechs}_{1,9}$
7. $\text{sieben}_{1,1} \vee \text{sieben}_{1,2} \vee \ldots \vee \text{sieben}_{1,9}$
8. $\text{acht}_{1,1} \vee \text{acht}_{1,2} \vee \ldots \vee \text{acht}_{1,9}$
9. $\text{neun}_{1,1} \vee \text{neun}_{1,2} \vee \ldots \vee \text{neun}_{1,9}$

Analogously for all other rows, eg.

Ninth Row: $T_{\text{Row 9}}$

1. $\text{eins}_{9,1} \vee \text{eins}_{9,2} \vee \ldots \vee \text{eins}_{9,9}$
2. $\text{zwei}_{9,1} \vee \text{zwei}_{9,2} \vee \ldots \vee \text{zwei}_{9,9}$
3. $\text{drei}_{9,1} \vee \text{drei}_{9,2} \vee \ldots \vee \text{drei}_{9,9}$
4. $\text{vier}_{9,1} \vee \text{vier}_{9,2} \vee \ldots \vee \text{vier}_{9,9}$
5. $\text{fuenf}_{9,1} \vee \text{fuenf}_{9,2} \vee \ldots \vee \text{fuenf}_{9,9}$
6. $\text{sechs}_{9,1} \vee \text{sechs}_{9,2} \vee \ldots \vee \text{sechs}_{9,9}$
7. $\text{sieben}_{9,1} \vee \text{sieben}_{9,2} \vee \ldots \vee \text{sieben}_{9,9}$
8. $\text{acht}_{9,1} \vee \text{acht}_{9,2} \vee \ldots \vee \text{acht}_{9,9}$
9. $\text{neun}_{9,1} \vee \text{neun}_{9,2} \vee \ldots \vee \text{neun}_{9,9}$

Is that sufficient? What if a row contains several 1's?

First Column: $T_{\text{Column 1}}$

1. $\text{eins}_{1,1} \vee \text{eins}_{2,1} \vee \ldots \vee \text{eins}_{9,1}$
2. $\text{zwei}_{1,1} \vee \text{zwei}_{2,1} \vee \ldots \vee \text{zwei}_{9,1}$
3. $\text{drei}_{1,1} \vee \text{drei}_{2,1} \vee \ldots \vee \text{drei}_{9,1}$
4. $\text{vier}_{1,1} \vee \text{vier}_{2,1} \vee \ldots \vee \text{vier}_{9,1}$
5. $\text{fuenf}_{1,1} \vee \text{fuenf}_{2,1} \vee \ldots \vee \text{fuenf}_{9,1}$
6. $\text{sechs}_{1,1} \vee \text{sechs}_{2,1} \vee \ldots \vee \text{sechs}_{9,1}$
7. $\text{sieben}_{1,1} \vee \text{sieben}_{2,1} \vee \ldots \vee \text{sieben}_{9,1}$
8. $\text{acht}_{1,1} \vee \text{acht}_{2,1} \vee \ldots \vee \text{acht}_{9,1}$
9. $\text{neun}_{1,1} \vee \text{neun}_{2,1} \vee \ldots \vee \text{neun}_{9,1}$

Analogously for all other columns.

Is that sufficient? What if a column contains several 1's?

All put together:

$$\begin{aligned} T_{\text{Sudoku}} = \; & T_1 \cup T_1' \cup \ldots \cup T_9 \cup T_9' \\ & T_{\text{Row 1}} \cup \ldots \cup T_{\text{Row 9}} \\ & T_{\text{Column 1}} \cup \ldots \cup T_{\text{Column 9}} \end{aligned}$$

Here is a more difficult one.

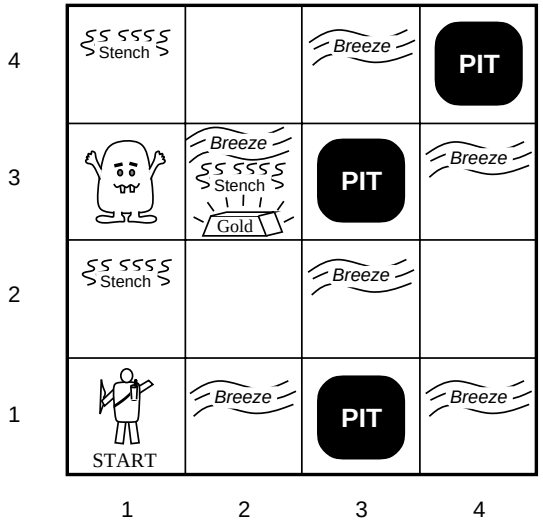
2		3		9				6
	1							4
	9				5			
		7						
6			9				7	
8	4							
9	3		1		7	5		
7		2			9	4		
1			2				6	

Table 4: A difficult Sudoku $S_{\text{difficult}}$

Example 2.16 (Wumpus)

A wumpus is moving around in a grid. A knight in shining armour moves around and is looking for gold. The wumpus is trying to kill him when they are on the same cell. The knight is also dying when he enters a pit. Fortunately, in the cells adjacent to pits there is a cold breeze. And in adjacent cells to the wumpus, there is an incredible stench.

How should the knight behave?



1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 **OK**	2,2	3,2	4,2
1,1 **A** **OK**	2,1 **OK**	3,1	4,1

(a)

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 **OK**	2,2 **P?**	3,2	4,2
1,1 **V** **OK**	2,1 **A** **B** **OK**	3,1 **P?**	4,1

(b)

1,4	2,4	3,4	4,4
1,3 **W!**	2,3	3,3	4,3
1,2 **A** **S** **OK**	2,2 **OK**	3,2	4,2
1,1 **V** **OK**	2,1 **B** **V** **OK**	3,1 **P!**	4,1

(a)

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

1,4	2,4 **P?**	3,4	4,4
1,3 **W!**	2,3 **A** **S** **G** **B**	3,3 **P?**	4,3
1,2 **S** **V** **OK**	2,2 **V** **OK**	3,2	4,2
1,1 **V** **OK**	2,1 **B** **V** **OK**	3,1 **P!**	4,1

(b)

Language definition:

$\mathsf{s}_{i,j}$	stench on field $\langle i, j\rangle$
$\mathsf{b}_{i,j}$	breeze on field $\langle i, j\rangle$
$\mathsf{pit}_{i,j}$	$\langle i, j\rangle$ is a pit
$\mathsf{gl}_{i,j}$	$\langle i, j\rangle$ glitters
$\mathsf{w}_{i,j}$	$\langle i, j\rangle$ contains Wumpus

General knowledge:

$$\neg\mathsf{s}_{1,1} \longrightarrow (\neg\mathsf{w}_{1,1} \wedge \neg\mathsf{w}_{1,2} \wedge \neg\mathsf{w}_{2,1})$$
$$\neg\mathsf{s}_{2,1} \longrightarrow (\neg\mathsf{w}_{1,1} \wedge \neg\mathsf{w}_{2,1} \wedge \neg\mathsf{w}_{2,2} \wedge \neg\mathsf{w}_{3,1})$$
$$\neg\mathsf{s}_{1,2} \longrightarrow (\neg\mathsf{w}_{1,1} \wedge \neg\mathsf{w}_{1,2} \wedge \neg\mathsf{w}_{2,2} \wedge \neg\mathsf{w}_{1,3})$$

$$\mathsf{s}_{1,2} \longrightarrow (\mathsf{w}_{1,3} \vee \mathsf{w}_{1,2} \vee \mathsf{w}_{2,2} \vee \mathsf{w}_{1,1})$$

Knowledge after the 3rd move:

$$\neg s_{1,1} \wedge \neg s_{2,1} \wedge s_{1,2} \wedge \neg b_{1,1} \wedge b_{2,1} \wedge \neg b_{1,2}$$

Question:

Can we derive that the wumpus is located at $\langle 1, 3 \rangle$?

Answer:

Yes. With any correct and complete calculus.
Because it is semantically entailed.

We now formalize a ”Logelei“ (in order to solve it with a **theorem prover**).

Example 2.17 (”Logelei“ from ”Die Zeit“ (1))

Alfred ist als neuer Korrespondent in Wongowongo. Er soll über die Präsidentschaftswahlen berichten, weiß aber noch nichts über die beiden Kandidaten, weswegen er sich unter die Leute begibt, um Infos zu sammeln. Er befragt eine Gruppe von Passanten, von denen drei Anhänger der Entweder-oder-Partei sind und drei Anhänger der Konsequenten.

”Logelei“ from ”Die Zeit“ (2)

Auf seinem Notizzettel notiert er stichwortartig die Antworten.
A: »Nachname Songo: Stadt Rongo«,
B: »Entweder-oder-Partei: älter«,
C: »Vorname Dongo: bei Umfrage hinten«,
A: »Konsequenten: Vorname Mongo«,
B: »Stamm Bongo: Nachname Gongo«,
C: »Vorname Dongo: jünger«,
D: »Stamm Bongo: bei Umfrage vorn«,
E: »Vorname Mongo: bei Umfrage hinten«,
F: »Konsequenten: Stamm Nongo«,
D: »Stadt Longo: jünger«,
E: »Stamm Nongo: jünger«.
F: »Konsequenten: Nachname Gongo«.

”Logelei“ from ”Die Zeit“ (3)

Jetzt grübelt Alfred. Er weiß, dass die Anhänger der Entweder-oder-Partei (A, B und C) immer eine richtige und eine falsche Aussage machen, während die Anhänger der Konsequenten (D, E und F) entweder nur wahre Aussagen oder nur falsche Aussagen machen.
Welche Informationen hat Alfred über die beiden Kandidaten?

(By Zweistein)

Towards a solution

- Selection of the language $\mathcal{L}$ ($\mathcal{P}rop$).
- Analysis and formalization of the problem.
- Transformation to the input format of a prover.
- Output of a solution, i.e. a **model**.

Definition of the constants

$\text{sur}_{x,\text{Songo}} \equiv$	x's surname is Songo
$\text{sur}_{x,\text{Gongo}} \equiv$	x's surname is Gongo
$\text{first}_{x,\text{Dongo}} \equiv$	x's first name is Dongo
$\text{first}_{x,\text{Mongo}} \equiv$	x's first name is Mongo
$\text{tribe}_{x,\text{Bongo}} \equiv$	x belongs to the Bongos
$\text{tribe}_{x,\text{Nongo}} \equiv$	x belongs to the Nongos
$\text{city}_{x,\text{Rongo}} \equiv$	x comes from Rongo
$\text{city}_{x,\text{Longo}} \equiv$	x comes from Longo
$\text{senior}_x \equiv$	x is the senior candidate
$\text{junior}_x \equiv$	x is the junior candidate
$\text{worse}_x \equiv$	x's poll is worse
$\text{better}_x \equiv$	x's poll is better

Here x is a candidate, i.e. $x \in \{a, b\}$. So we have 24 constants in total.

The correspondent Alfred noted 12 statements about the candidates (each interviewee gave 2 statements, ϕ, ϕ') which we enumerate as follows

$$\phi_A, \phi'_A, \phi_B, \phi'_B, \ldots, \phi_F, \phi'_F,$$

All necessary symbols are now defined, and we can formalize the given statements.

Formalization of the statements

$$
\begin{aligned}
\phi_A &\leftrightarrow (\mathsf{sur}_{a,\mathsf{Songo}} \wedge \mathsf{city}_{a,\mathsf{Rongo}}) \vee \\
&\quad\ (\mathsf{sur}_{b,\mathsf{Songo}} \wedge \mathsf{city}_{b,\mathsf{Rongo}}) \\
\phi'_A &\leftrightarrow \mathsf{first}_{b,\mathsf{Mongo}} \\
\phi_B &\leftrightarrow \mathsf{senior}_a \\
\phi'_B &\leftrightarrow (\mathsf{tribe}_{a,\mathsf{Bongo}} \wedge \mathsf{sur}_{a,\mathsf{Gongo}}) \vee \\
&\quad\ (\mathsf{tribe}_{b,\mathsf{Bongo}} \wedge \mathsf{sur}_{b,\mathsf{Gongo}}) \\
&\ \ \vdots
\end{aligned}
$$

Furthermore, **explicit** conditions between the statements are given, e.g.

$$(\phi_A \wedge \neg\phi'_A) \vee (\neg\phi_A \wedge \phi'_A)$$

and

$$(\phi_D \wedge \phi'_D) \vee (\neg\phi_D \wedge \neg\phi'_D).$$

Analogously, for the other statements.

Is this enough information to solve the puzzle?

E.g., can the following formula be satisfied?

$$\mathsf{sur}_{a,\mathsf{Songo}} \wedge \mathsf{sur}_{a,\mathsf{Gongo}}$$

We also need **implicit** conditions (**axioms**) which are required to solve this problem.
It is necessary to state that each candidate has only one name, comes from one city, etc.

We need the following background knowledge...

$$\text{sur}_{x,\text{Songo}} \leftrightarrow \neg\text{sur}_{x,\text{Gongo}}$$
$$\text{first}_{x,\text{Dongo}} \leftrightarrow \neg\text{first}_{x,\text{Mongo}}$$
$$\vdots$$
$$\text{worse}_x \leftrightarrow \neg\text{better}_x$$

Can we abstain from these axioms by changing our representation of the puzzle?

What is still missing?

Can we prove that when a's poll is worse, then s's poll is better?

We need to state the relationships between these attributes:

$$\text{worse}_x \leftrightarrow \text{better}_y$$

$$\text{senior}_x \leftrightarrow \text{junior}_y$$

Finally, we have modeled all "sensible" information. Does this yield a unique model?

No! There are **6 models** in total, but this is all right. It just means there is no unique solution.

What if a unique model is desirable?

Often, there are additional assumptions hidden "between the lines". Think, for example, of deductions by Sherlock Holmes (or Miss Marple, Spock, Monk etc).

For example, it might be sensible to assume that both candidates come from **different** cities:

$$\text{city}_{x,\text{Rongo}} \leftrightarrow \text{city}_{y,\text{Longo}}$$

Indeed, with this additional axiom there is an unique model.

But, be careful...

... this additional information may not be justified by the nature of the task!

2.4 Hilbert Calculus

Deriving formulae purely algorithmically

- We have a clear undestanding of what it means that "a formula follows or can be deduced from a theory T".
- Can we **automize** that?
- Can we find a procedure to systematically derive new formulae?
- In such a way that all formulae that do indeed follow from T will be eventually derived?
- Let us take inspiration of what mathematicians have done since centuries!

Definition 2.18 (Hilbert-Type Calculi)

A Hilbert-Type calculus over a language $\mathcal{L}$ is a pair $\langle \mathrm{Ax}, \mathrm{Inf} \rangle$ where

Ax: is a subset of the set of $\mathcal{L}$-formulae: they are called axioms,

Inf: is a set of pairs written in the form

$$\frac{\phi_1, \phi_2, \ldots, \phi_n}{\psi}$$

where $\phi_1, \phi_2, \ldots, \phi_n, \psi$ are from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$: they are called inference rules.

Intuitively, we assume the axioms to be true and use the inference rules to derive new formulae.

Definition 2.19 (Calculus for Propositional Logic SL)

We define $\mathrm{Hilbert}_{\mathcal{L}}^{SL} = \langle \{\neg\varphi \vee \varphi, \neg\Box\}, \mathtt{Inf} \rangle$, a Hilbert-Type calculus for SL. The only axiom schema is $\neg\varphi \vee \varphi$, the only axiom is $\neg\Box$.
The set $\mathtt{Inf}$ of inference rules consists of the following four schemata

Expansion: $\frac{\varphi}{\psi \vee \varphi}$,

Associativity: $\frac{\varphi \vee (\psi \vee \chi)}{(\varphi \vee \psi) \vee \chi}$,

Shortening: $\frac{\varphi \vee \varphi}{\varphi}$,

Cut: $\frac{\varphi \vee \psi, \neg\varphi \vee \chi}{\psi \vee \chi}$.

(φ, ψ, χ stand for arbitrarily complex formulae (not just constants). They represent schemata, rather than particular formulae in the language.)

Definition 2.20 (Proof)

A **proof** of a formula φ from a theory $T \subseteq \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ is a finite **sequence** $\varphi_1, \ldots, \varphi_n$ of formulae such that $\varphi_n = \varphi$ and for all i with $1 \leq i \leq n$ one of the following conditions holds:

- φ_i is **instance of the axiom schema**, or of the form $\neg\Box$,
- $\varphi_i \in T$,
- there is φ_l with $l < i$ and φ_i is obtained from φ_l by **expansion**, **associativity**, or **shortening**,
- there is φ_l, φ_k with $l, k < i$ and φ_i is obtained from φ_l, φ_k by **cut**.

We write: $T \vdash_{\mathsf{SL}} \varphi$ (φ can be **derived (or proved)** from T).

We have now introduced two important notions:

Syntactic derivability $\vdash_{SL}$: the notion that certain formulae can be derived (or proved) from other formulae using a certain calculus,

Semantic validity $\models$: the notion that certain formulae follow (semantically) (or can be deduced (are entailed)) from other formulae based on the semantic notion of a model.

Definition 2.21 (Correct-, Completeness for a calculus)

Given an arbitrary **calculus** (which defines a notion $\vdash$) and a **semantics** based on certain models (which defines a relation $\models$), we say that

Correctness: The calculus is **correct** with respect to the semantics, if the following holds:

$$T \vdash \phi \text{ implies } T \models \phi.$$

Completeness: The calculus is **complete** with respect to the semantics, if the following holds:

$$T \models \phi \text{ implies } T \vdash \phi.$$

Lemma 2.22 (Corrrectness of $\mathrm{Hilbert}_{\mathcal{L}}^{SL}$)

Let T be a (possibly infinite) theory and φ a formula over $\mathcal{P}rop$.

Then $T \vdash_{SL} \varphi$ implies that $T \models \varphi$.

I.e. each provable formula is also entailed!

Proof.

By induction on the structure of φ.
We have to show that

1. **all instances of the axiom schema are valid**, $\neg\Box$ is valid, and
2. **for each inference rule the conjunction of the premises entails its conclusion**.

□

We show a few simple facts that we need later to prove completeness.

Lemma 2.23 (Some derivations)

1. $\vdash_{\mathsf{SL}} \neg\Box$.
2. $\phi \vee \psi \vdash_{\mathsf{SL}} \psi \vee \phi$.
3. $\varphi \vdash_{\mathsf{SL}} \varphi \vee \psi$.
4. $\phi \vee \psi \vdash_{\mathsf{SL}} \neg\neg\phi \vee \psi$.
5. $\neg\neg\phi \vee \psi \vdash_{\mathsf{SL}} \phi \vee \psi$.
6. $\{\neg\phi \vee \chi, \neg\psi \vee \chi\} \vdash_{\mathsf{SL}} \neg(\phi \vee \psi) \vee \chi$.

The first three are shown on $\rightsquigarrow$ **blackboard 2.12**. (4) and (5) are $\rightsquigarrow$ **exercise**.

Lemma 2.24 (Derived inference rules)

The following rules can be added to the calculus without affecting the set of derivable formulae. They are also called derived rules.

1. **(Modus Ponens)** $\frac{\chi, \chi \to \varphi}{\varphi}$,
2. **(Commutativity)** $\frac{\psi \vee \varphi}{\varphi \vee \psi}$,
3. *Let* $i_1, i_2, \ldots, i_m \in \{1, 2, \ldots, n\}$. *Then*

   (Gen. Expansion) $\frac{\varphi_{i_1} \vee \ldots \vee \varphi_{i_m}}{\varphi_1 \vee \ldots \vee \varphi_n}$

Theorem 2.25 (Weak completeness)

Let $\varphi_1, \ldots, \varphi_n$ and φ formulae over $\mathcal{P}rop$.Then:

$\{\varphi_1, \ldots, \varphi_n\} \models \varphi$ *implies that* $\{\varphi_1, \ldots, \varphi_n\} \vdash_{SL} \varphi$.

Definition 2.26 ((In-) Consistency of a theory T)

A theory T is called **consistent**, if there is a formula that can not be proved from it.

A theory T is called **inconsistent**, if it is not consistent. I.e. from an inconsistent theory, **all** formulae can be proved.

Consistency is a property of a **calculus**. It is often mixed up with **existence of a model** (which it is often equivalent to).

Attention: Completeness

After we have proved the completeness theorem, we know that **inconsistent theories are exactly those that do not posess any models** (see Slide 82). But we do not know this yet! It has to be proved.

The key to prove completeness of our calculus is based on the following

Theorem 2.27 (Deduction Theorem)

Let T be a theory and ϕ, ψ be formulae from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$. Then the following holds

$$T \vdash_{\mathsf{SL}} (\phi \to \psi) \text{ if and only if } T \cup \{\phi\} \vdash_{\mathsf{SL}} \psi$$

Proof.

One direction (left to right) is trivial: just an application of Modus Ponens (which we have shown to be a derived rule on Slide 131). The other direction is by **induction on the length of a proof for ψ**.

A proof of ψ from $T \cup \{\phi\}$ might have used ϕ at several places. The idea is to replace in each step of the proof of ψ a formula η by $\phi \to \eta$. This then transforms the old proof into one of $(\phi \to \psi)$ in T. □

An important consequence of the deduction theorem is the following

Lemma 2.28 (Consistency and Derivability)

Let T be a theory and ϕ a formula from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ (both could be inconsistent). Then the following holds

1. $T \vdash_{\mathsf{SL}} \phi$ *if and only if* $T \cup \{\neg\phi\}$ *is inconsistent.*
2. $T \nvdash_{\mathsf{SL}} \phi$ *if and only if* $T \cup \{\neg\phi\}$ *is consistent.*

Proof of Lemma 2.28.

The second statement follows trivially from the first (contraposition).

Let $T \vdash_{\mathsf{SL}} \phi$. Adding $\neg\phi$ does not influence the proof of ϕ, so $T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \phi$. But, trivially, $T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \neg\phi$, so $T \cup \{\neg\phi\}$ is inconsistent. Conversely, if $T \cup \{\neg\phi\}$ is inconsistent, then $T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \phi$ (because any formula can be derived). By the deduction theorem, $T \vdash_{\mathsf{SL}} \neg\phi \to \phi$, thus $T \vdash_{\mathsf{SL}} \phi$. □

Theorem 2.29 (Correct-, Completeness for $\mathrm{Hilbert}_{\mathcal{L}}^{SL}$)

A formula follows semantically from a theory T if and only if it can be derived:

$$T \models \varphi \text{ if and only if } T \vdash_{\mathsf{SL}} \varphi$$

Proof of the Correctness part of Theorem 2.29.

Correctness follows by **induction on the length of proofs**: the axiom is valid and all four inference rules have the property, that from their premises, the conclusions follow. We have already discussed this in Lemma 2.22.

□

Proof of the Completeness part of Theorem 2.29.

To prove completeness, it is enough to show that **each consistent theory is satisfiable** (because if $T \models \varphi$ and not $T \vdash_{\mathsf{SL}} \varphi$, then $T \cup \{\neg\varphi\}$ is **consistent** (why?), thus satisfiable: a **contradiction**). Given a consistent theory T, how to construct a model for T? We claim that the **models** of T are exactly the **maximal consistent** extensions of T. Let $\varphi_0, \varphi_1, \ldots$ an enumeration of $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$. We construct the sequence T_i as follows: $T_0 := T$,

$$T_{i+1} := \begin{cases} T_i & \text{if } T_i \vdash_{\mathsf{SL}} \varphi_i, \\ T_i \cup \{\neg\varphi_i\} & \text{else.} \end{cases}$$

Then $\bigcup_{i=0}^{\infty} T_i$ is a maximal consistent extension of T:If it were inconsistent, then there were a proof of $\Box$; this proof uses only finitely many formulae; but then there is a n_0 such that T_{n_0} contains all of them, so T_{n_0} were inconsistent; but all T_i are consistent by Lemma 2.28! It is maximal because $\varphi_0, \varphi_1, \ldots$ is an enumeration of $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$. $\Box$

The following important result is equivalent to the completeness theorem:

Corollary 2.30 (Compactness for $\mathrm{Hilbert}_{\mathcal{L}}^{SL}$)

A formula **follows from a theory** T *if and only if it* **follows from a finite subset of** T*:*

$$Cn(T) = \bigcup \{Cn(T') : T' \subseteq T, T' \text{finite}\}.$$

Proof.

Derivability in a calculus means that **there is a proof**, a finite object. Thus, by completeness, if a formula follows from a (perhaps infinite) theory T, there is a finite proof of it which **involves only finitely many formulae** from T. Thus the formula follows from these finitely many formulae. □

Satisfiability of a theory

The following equally important result is again a corollary to the completeness theorem

Corollary 2.31 (Compactness for theories T)

Let T be a theory from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$. Then the following are equivalent:

- *T is satisfiable,*
- *each finite subset of T is satisfiable.*

2.5 Resolution Calculus

Conjunctive Normal Form

- We have already seen that each formula can be written in conjunctive normal form.
- Thus each formula can be identified with a set of clauses.
- We note that it is possible, that a clause is empty: it is represented by $\square$.
- We also note that it is possible, that a conjunction is empty. The empty conjunction is obviously dual to the empty disjunction: it is represented by $\neg\square$ (or by the macro $\top$ introduced in Definition 2.3 on Slide 63).

Language for the Resolution Calculus

How can we determine whether a disjunction $\bigvee_{j=1}^{m_i} \phi_{i,j}$ is satisfiable or not?

Definition 2.32 (Clauses, Language $\mathsf{Fml}_{\mathcal{L}}^{clausal}$)

A propositional formula of the form $\bigvee_{j=1}^{m_i} \phi_{i,j}$, where all $\phi_{i,j}$ are constants or negated constants, is called a **clause**.

We also allow the **empty clause**, represented by $\square$. Dually, $\neg\square$ can be interpreted as the **empty conjunction**.

Given a signature $\mathcal{P}rop$, determining a language $\mathcal{L}(\mathcal{P}rop)$ or just $\mathcal{L}$, we denote by $\mathsf{Fml}_{\mathcal{L}(\mathcal{P}rop)}^{clausal}$, **or just** $\mathsf{Fml}_{\mathcal{L}}^{clausal}$ the set of formulae consisting of just **clauses** over $\mathcal{P}rop$.

Definition 2.33 (Set-notation of clauses)

A clause $A \vee \neg B \vee C \vee \ldots \vee \neg E$ can also be represented as a set

$$\{A, \neg B, C, \ldots, \neg E\}.$$

Thus the **set-theoretic union** of such sets corresponds again to a clause: $\{A, \neg B\} \cup \{A, \neg C\}$ represents $A \vee \neg B \vee \neg C$. The empty set $\emptyset$ corresponds to the empty disjunction and is represented by $\square$.

But we know this already from Slides 51–53.
Note that we identify **double occurrences** implicitly by using the set-notation: $\{A, A, B, C\} = \{A, B, C\}$.

We define an inference rule on $\mathrm{Fml}_{\mathcal{L}}^{clausal}$:

Definition 2.34 (SL resolution)

Let C_1, C_2 be clauses and let X by any constant from $\mathcal{P}rop$. The following inference rule allows to derive the clause $C_1 \vee C_2$ from $C_1 \vee \mathsf{X}$ and $C_2 \vee \neg\mathsf{X}$:

$$\textbf{(Res)}\ \frac{C_1 \vee \mathsf{X},\ C_2 \vee \neg\mathsf{X}}{C_1 \vee C_2}$$

If $C_1 = C_2 = \emptyset$, then $C_1 \vee C_2 = \square$.

Compare with the **Cut** rule in Definition 2.19.

If we use the set-notation for clauses, we can formulate the inference rule as follows:

Definition 2.35 (SL resolution (Set notation))

Derive the clause $C_1 \cup C_2$ from $C_1 \cup \{\mathsf{X}\}$ and $C_2 \cup \{\neg\mathsf{X}\}$:

$$\textbf{(Res)}\ \frac{C_1 \cup \{\mathsf{X}\},\ C_2 \cup \{\neg\mathsf{X}\}}{C_1 \cup C_2}$$

In the rule C_1 or C_2 can be both empty, in which case $\square$ is derived from X and $\neg$X. (Note, that we identify the empty set $\emptyset$ with $\square$.)

Definition 2.36 (Resolution Calculus for SL)

We define the **resolution calculus** $\text{Robinson}^{SL}_{\mathcal{L}^{clausal}}$ **as the pair** $\langle \emptyset, \{\text{Res}\} \rangle$ operating on the set $\text{Fml}^{clausal}_{\mathcal{L}}$ of clauses.
We denote the **corresponding derivation relation** by $\vdash_{\text{Res}}$ (in contrast to $\vdash_{\text{SL}}$ induced by the Hilbert calculus from Definition 2.19).

- So there are no axioms at all and only one inference rule.
- The notion of a **proof** in this system is obvious: it is literally the same as given in Definition 2.19.

Question:

Is this calculus correct and complete?

Answer:

It is correct, but not complete!

But "$T \models \phi$" is equivalent to

$$\text{"}T \cup \{\neg\phi\} \text{ is unsatisfiable"}$$

or rather to

$$T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \square.$$

Now in such a case, the **resolution calculus is powerful enough** to derive $\square$.

Resolution calculus is not complete

Example 2.37 (Complete vs refutation complete)

We consider $\mathcal{P}rop = \{a, b\}$. Obviously $a \models_{\mathsf{SL}} a \vee b$ and $a \vdash_{\mathsf{SL}} a \vee b$. But a $\nvdash_{\mathsf{RES}}$ a $\vee$ b.

The reason is that with a alone, **there is no possibility to derive** a $\vee$ b with the resolution rule (the premises are not fulfilled). Whereas when $\neg$(a $\vee$ b) is added, i.e. both $\neg$a and $\neg$b, then the rules can be applied and lead to the derivation of $\square$.

- If $T \vdash_{\mathsf{SL}} \varphi$ does not imply $T \vdash_{\mathsf{Res}} \varphi$.
- If $T \vdash_{\mathsf{Res}} \varphi$ then also $T \vdash_{\mathsf{SL}} \varphi$ (for $\varphi \in \mathsf{Fml}_{\mathcal{L}}^{clausal}$).
- But the following holds:

$$T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \square \textit{ if and only if } T \cup \{\neg\phi\} \vdash_{\mathsf{Res}} \square$$

We say that resolution is refutation complete.

Theorem 2.38 (Completeness of resolution refutation)

If M is an unsatisfiable set of clauses then the empty clause $\square$ can be derived in $\mathrm{Robinson}_{\mathcal{L}^{clausal}}^{SL}$.

How to use the resolution calculus? (1)

Suppose we want to prove that $T \models \phi$. Using the Hilbert calculus, we could try to prove ϕ directly. This is not possible for the resolution calculus, for two different reasons:

- it **operates on clauses**, not on arbitrary formulae,
- there is **no completeness result** in a form similar to Theorem 2.29.

How to use the resolution calculus? (2)

But we have refutation completeness (see Theorem 2.38) of the resolution calculus, which allows us to do the following:

1. We transform $T \cup \{\neg\phi\}$ into a set of clauses.
2. Then we apply the resolution calculus and try to derive $\square$. I.e. instead of $T \vdash_{\text{res}} \phi$ we try to show $T \cup \{\neg\phi\} \vdash_{\text{res}} \square$.
3. If we succeed, we have shown $T \models \phi$.
4. If we can show that the empty clause is not derivable at all, then $T \not\models \phi$.

How to use the resolution calculus? (3)

How to show that the empty clause is **not** derivable from a theory T?

- One could try to formally prove that no such derivation is possible in the resolution calculus.
- This could be done by a clever induction on the structure of all possible derivations.
- But this is often very complicated and far from trivial.
- Just applying the calculus and not being able to derive the empty clause is not enough (there might be other ways).

The best way is to argue semantically: to show that there is a model of $T \cup \{\neg\phi\}$ by giving a concrete valuation.

3. Verification I: LT properties

Content of this chapter (1):

In this chapter we explore ways to use SL for the verification of reactive systems. First we need to model concurrent systems using SL. This allows us to analyse a broad class of hardware and software systems.

Transition Systems: They are the main underlying abstraction to describe concurrent systems. We start with **asynchronous systems** and then deal with **synchronization**: interleaving, and handshaking. An important classic example that we discuss is the dining philosophers problem.

LT properties: Which properties of the concurrent systems that we introduced do we want to verify? It turns out that many interesting conditions are linear temporal properties. Among them are fairness, safety as well as starvation and deadlock-freeness.

Content of this chapter (2):

LTL: This is an extension of SL to formulate linear-temporal properties within a logic. Given a model and a **LTL** formula, checking whether the formula is satisfied in the model is called LTL model checking. We describe and discuss this method in detail.

CTL, timed CTL: These are nontrivial extensions of **LTL** and allow us to express much more interesting properties. However, these are too advanced and will not be dealt with in this course.

3.1 Motivation

The need for verification

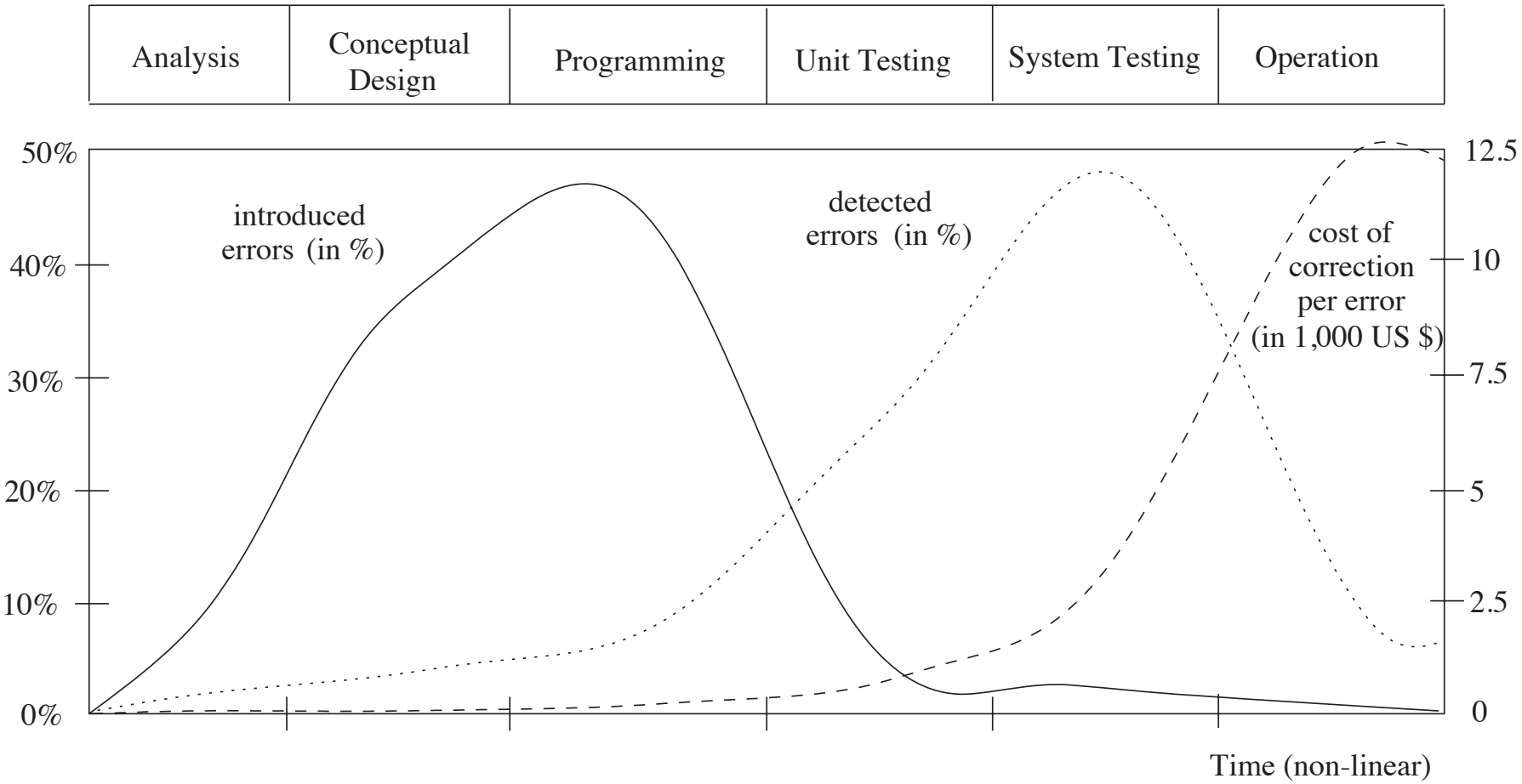


Figure 3.9: Errors in software development

The need for verification

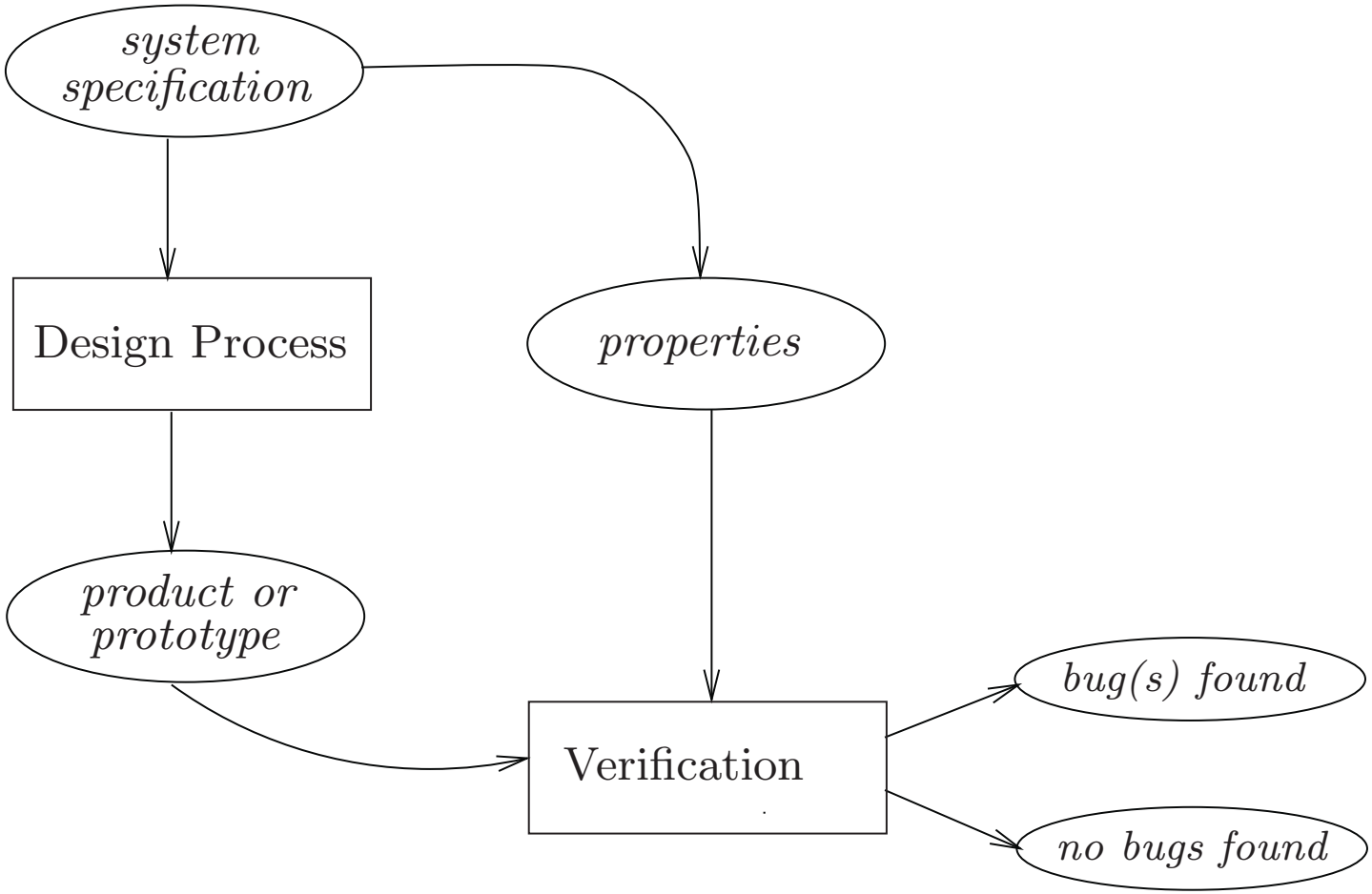


Figure 3.10: Overall approach

Inference Tasks: The Three Questions

The semantical perspective allows us to think about the following inference tasks:

Inference Tasks

Model Checking: Given φ and a model $\mathcal{M}$, does the formula correctly describe this model?

Satisfiability Checking: Given φ, does there exist a model in which the formula is true?

Validity Checking: Given φ, is it true in all models?

Model Checking

- This is the simplest of the three tasks.
- Nevertheless it is useful, e.g. for hardware verification.

Example 3.1

Think of a model $\mathcal{M}$ as a **mathematical picture** of a chip. A logical description φ might define some security issues. If $\mathcal{M}$ fulfills φ, then this means that the chip will be secure wrt. these issues.

Model checking for simple SL

Given a model v and a description φ.
Is $v \models \varphi$ true?

Example 3.2

Given a model v in which a is true and b is false.
Is $\varphi := (\mathsf{a} \vee \neg \mathsf{b}) \rightarrow \mathsf{b}$ true?

Model checking

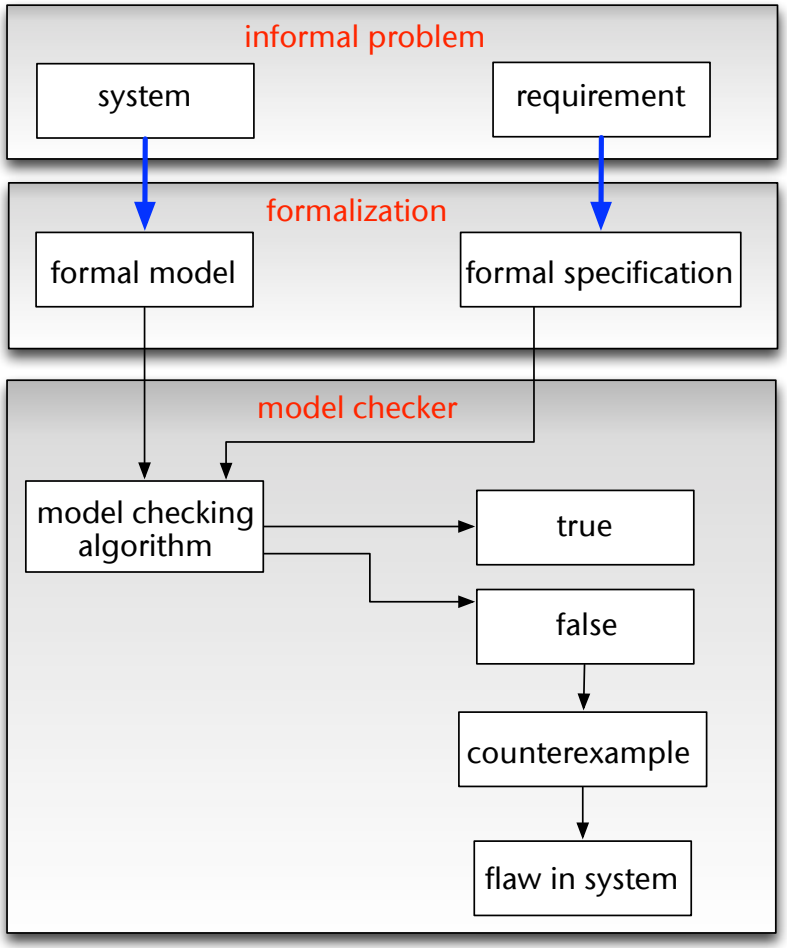


Figure 3.11: Model Checking

Satisfiability Checking

- One can interpret the description as a constraint.
- Is there anything that matches this description?
- We have to create model after model until we find one that satisfies φ.
- In the worst case we have to generate all models.

A good example is our sudoku problem.

Satisfiability Checking for SL

Given a description φ. Is there a model v s.t. $v \models \varphi$ is true?

Validity Checking

- The intuitive idea is that we write down a set of all our fundamental and indisputable axioms.
- Then, with this theory, we derive new formulae.

Example 3.3

Mathematics: We are usually given a set of axioms.
E.g. Euclid's axioms for geometry. We want to prove whether a certain statement follows from this set, or can be derived from it.

Validity Checking for SL

Given a description φ. Does $v \models \varphi$ hold for all models v?

3.2 Basic transition systems

- What is an appropriate abstraction to model hardware and software systems?

  $\rightsquigarrow$ transition systems.
- **Easy**: processes **run** completely **autonomously**.
- Difficult: processes communicate with each other.
- Main notion is a state: this is exactly what we called a model in Chapter 2.

Definition 3.4 (Transition system)

A **transition system** *TS* is a tuple $\langle \mathrm{S}, \mathrm{Act}, \longrightarrow, I, \mathcal{P}rop, \pi \rangle$ where

- S is a set of **states**, denoted by $s_1, s_2, \ldots$,
- Act is a set of **actions**, denoted by $\alpha, \beta, \gamma, \ldots$,
- $\longrightarrow \subseteq \mathrm{S} \times \mathrm{Act} \times \mathrm{S}$ is a **transition relation**,
- $I \subseteq \mathrm{S}$ is the set of **initial states**,
- $\mathcal{P}rop$ is a set of **atomic propositions**, and
- $\pi : \mathrm{S} \to 2^{\mathcal{P}rop}$ is a **labelling (or valuation)**.

A *TS* is **finite** *if, by definition,* S, Act and $\mathcal{P}rop$ are all finite.

This is in accordance with $\mathcal{L}_{SL}$. States are what we called **models**. A labelling corresponds to **a set of valuations**. The new feature is that we can **move between states by means of actions**: instead of $\langle s, \alpha, s' \rangle$ we write $s \xrightarrow{\alpha} s'$.

Example 3.5 (Beverage Vending Machine)

A machine delivers soda or beer after a coin has been inserted. A possible *TS* is:

- $\mathcal{P}rop = \{\text{pay}, \text{soda}, \text{beer}, \text{select}\}$,
- $\mathrm{S} = \{s_{\text{pay}}, s_{\text{beer}}, s_{\text{soda}}, s_{\text{select}}\}$, $I = \{s_{\text{pay}}\}$,
- $\text{Act} = \{\alpha_{\text{insert-coin}}, \alpha_{\text{get-soda}}, \alpha_{\text{get-beer}}, \tau\}$,
- the transitions: $s_{\text{pay}} \xrightarrow{\alpha_{\text{insert-coin}}} s_{\text{select}}$, $s_{\text{beer}} \xrightarrow{\alpha_{\text{get-beer}}} s_{\text{pay}}$, $s_{\text{soda}} \xrightarrow{\alpha_{\text{get-soda}}} s_{\text{pay}}$, $s_{\text{select}} \xrightarrow{\tau} s_{\text{beer}}$, $s_{\text{select}} \xrightarrow{\tau} s_{\text{soda}}$.

Beverage Vending Machine (cont.)

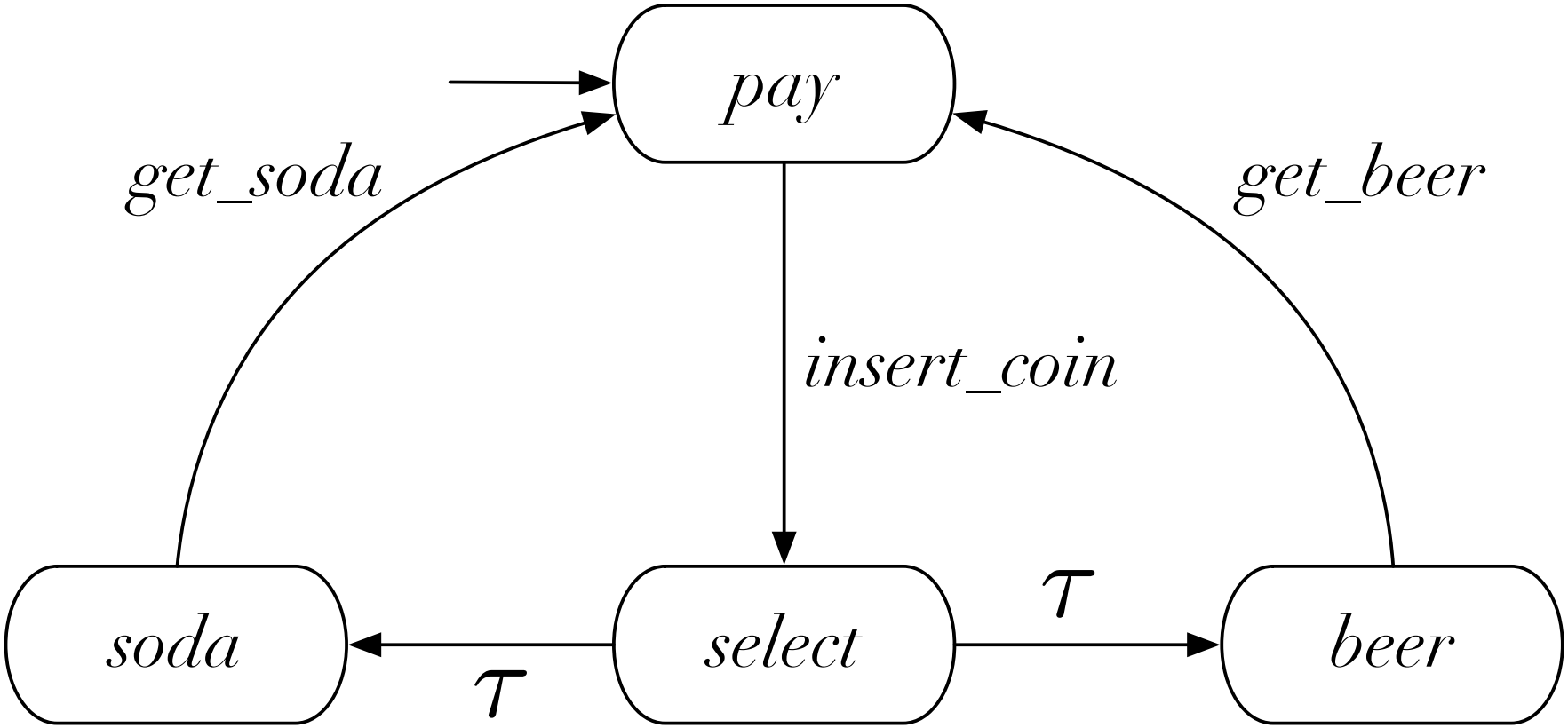


Figure 3.12: *TS* of a beverage vending machine

Simple Beverage Vending Machine (cont.)

- Note that the machine **nondeterministically** chooses beer or soda after a coin has been inserted. Nondeterminism can be seen as **implementation freedom**.
- Often we choose $\mathcal{P}rop = \mathrm{S}$, so that the labelling function is very simple: $\pi(s) = \{s\}$.
- When some properties do not refer to particular constants, we can also choose $\mathcal{P}rop \subsetneq \mathrm{S}$ and $\pi(s) = \{s\} \cap \mathcal{P}rop$.
- We can also choose $\mathcal{P}rop$ and S independently. **"The machine only delivers a drink after a coin has been inserted".** We choose $\mathcal{P}rop = \{\mathsf{paid}, \mathsf{drink}\}$ with labelling function: $\pi(s_{\mathrm{pay}}) =_{def} \emptyset, \pi(s_{\mathrm{soda}}) =_{def} \pi(s_{\mathrm{beer}}) =_{def} \{\mathsf{paid}, \mathsf{drink}\}, \pi(s_{\mathrm{select}}) =_{def} \{\mathsf{paid}\}$.
- Sometimes we want to abstract away from particular actions (internal actions of the machine not of any interest), like τ_1, τ_2. In that case, we use the symbol τ for all such actions.

Simple Hardware Circuit

We consider the hardware circuit diagram below. Next to it is the corresponding transition system *TS*.

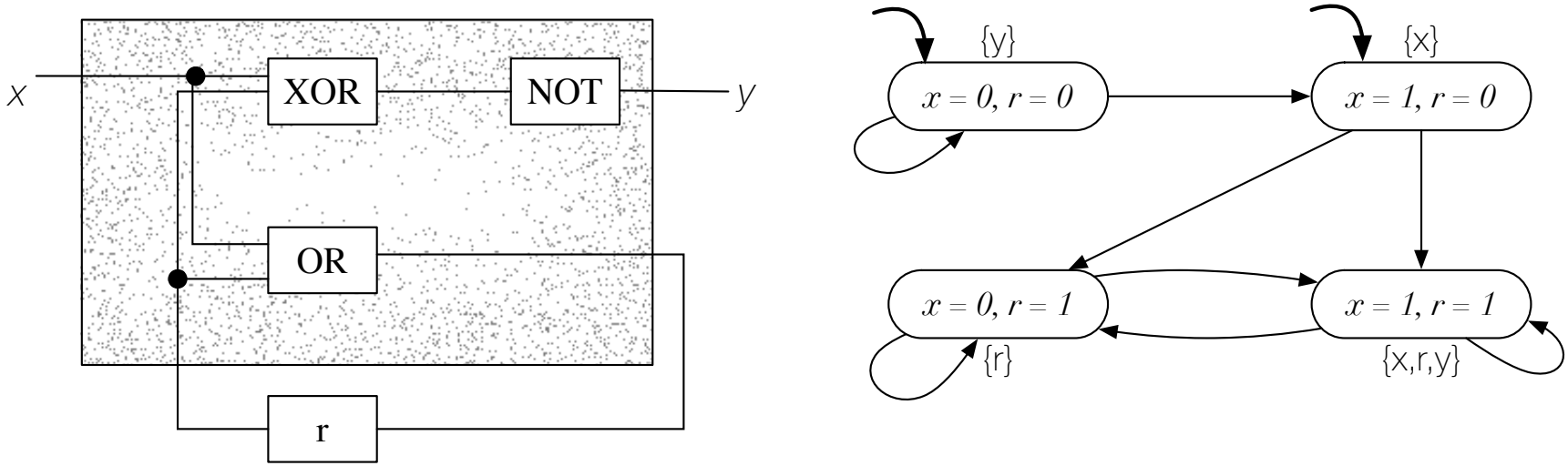


Figure 3.13: *TS* of a hardware circuit

Example 3.6 (Simple Hardware Circuit)

Here we are dealing with boolean variables: input x, output y and register r. They can be easily treated within $\mathcal{P}rop$.

- A state is determined by the contents of x and register r: $s_{x=0,r=0}$, $s_{x=0,r=1}$, $s_{x=1,r=0}$, $s_{x=1,r=1}$.
- $\mathcal{P}rop =_{def} \{\mathsf{x}, \mathsf{r}, \mathsf{y}\}$.
- The labelling is given by $\pi(s_{x=0,r=0}) = \{\mathsf{y}\}$, $\pi(s_{x=0,r=1}) = \{\mathsf{r}\}$, $\pi(s_{x=1,r=0}) = \{\mathsf{x}\}$, $\pi(s_{x=1,r=1}) = \{\mathsf{x}, \mathsf{r}, \mathsf{y}\}$.
- One could also use $\mathcal{P}rop =_{def} \{\mathsf{x}, \mathsf{y}\}$ and modify π accordingly.

Successor/predeccessor, terminal states

Successor (resp. predeccessor) states of a given state s in a transition system *TS* are important. They are determined by all **outgoing** (resp. **ingoing**) transitions.

- $\mathrm{Post}(s) =_{def} \bigcup_{\alpha \in \mathsf{Act}} \mathrm{Succ}(s,\alpha)$, where
  - $\mathrm{Succ}(s,\alpha) =_{def} \{s' : s \xrightarrow{\alpha} s'\}$.
- $\mathrm{Pre}(s) =_{def} \bigcup_{\alpha \in \mathsf{Act}} \mathrm{Anc}(s,\alpha)$, where
  - $\mathrm{Anc}(s,\alpha) =_{def} \{s' : s' \xrightarrow{\alpha} s\}$.

A terminal state s is a state without any outgoing transitions: $\mathrm{Post}(s) = \emptyset$.

Determinism/indeterminism

We already discussed the **indeterminism of transition systems** on Slide 173. It is similar to the **indeterministic choice in a proof-calculus**: there are many paths that are proofs.

- Sometimes the **observable behaviour** is deterministic. How can we formalize that?
- Level of actions versus level of states.

Definition 3.7 (Deterministic transition system)

A transition system *TS* is called

- action-deterministic, *if, by definition,* $|I| \leq 1$ and for all $s \in \mathrm{S}$ and $\alpha \in \mathrm{Act}$: $|\mathrm{Succ}(s, \alpha)| \leq 1$.
- $\mathcal{P}rop$-deterministic, *if, by definition,* $|I| \leq 1$ and for all $s \in \mathrm{S}$ and $T \subseteq \mathcal{P}rop$: $|\mathrm{Post}(s) \cap \{s' \in \mathrm{S} : \pi(s') = T\}| \leq 1$.

Executions

The informal working of a transition system is clear. How can we formally describe it and work with it?

- An execution of a *TS* is a (usually infinite) sequence of states and actions starting in an initial state and built using the transition relation of *TS*.
- We require the sequence to be maximal, i.e. it either ends in a terminal state or it is infinite.
- Sometimes we also consider initial fragments of an execution.
- Reachable states are those states of *TS* that can be reached with initial fragments of executions.

Executions/Reachable states

Definition 3.8 (Executions and reachable states)

Given a transition system *TS*, an execution (or run) is an alternating sequence of states and actions, written $s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n \ldots$ or

$$s_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} s_n \cdots$$

satisfying:

- $s_0 \in I$,
- $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all i,
- either (1) the sequence is finite and ends in a terminal state, or (2) the sequence is infinite.

A state $s \in \mathrm{S}$ is called reachable, if there is a finite initial fragment of an execution that ends in the state s.

Example 3.9 (Extended beverage vending machine)

The extended machine counts the number of bottles available and returns the coin if it is empty. It also refills automatically if there are no bottles left.

- We need more complex transitions: conditional transitions,
  e.g. select $\xrightarrow{\text{nsoda}=0\wedge\text{nbeer}=0:\ \text{ret-coin}}$ start,
  select $\xrightarrow{\text{nsoda}\geqslant 0:\ \text{sget}}$ start, select $\xrightarrow{\text{nbeer}\geqslant 0:\ \text{bget}}$ start,
  start $\xrightarrow{\mathbf{t}:\ \text{coin}}$ select, start $\xrightarrow{\mathbf{t}:\ \text{refill}}$ start.
- sget and bget decrement the numbers nsoda and nbeer by one, refill sets it to the maximum value max.
- This leads to the notion of a **program graph**.
- Such a graph can be unfolded to a transition system TS.
- So we end up with model checking transition systems.

The extended beverage vending machine has been introduced in Example 3.9 on Slide 180. It counts the number of bottles and returns the coin if it is empty. Setting max to 2 we get the unfolded transition system depicted on the right.

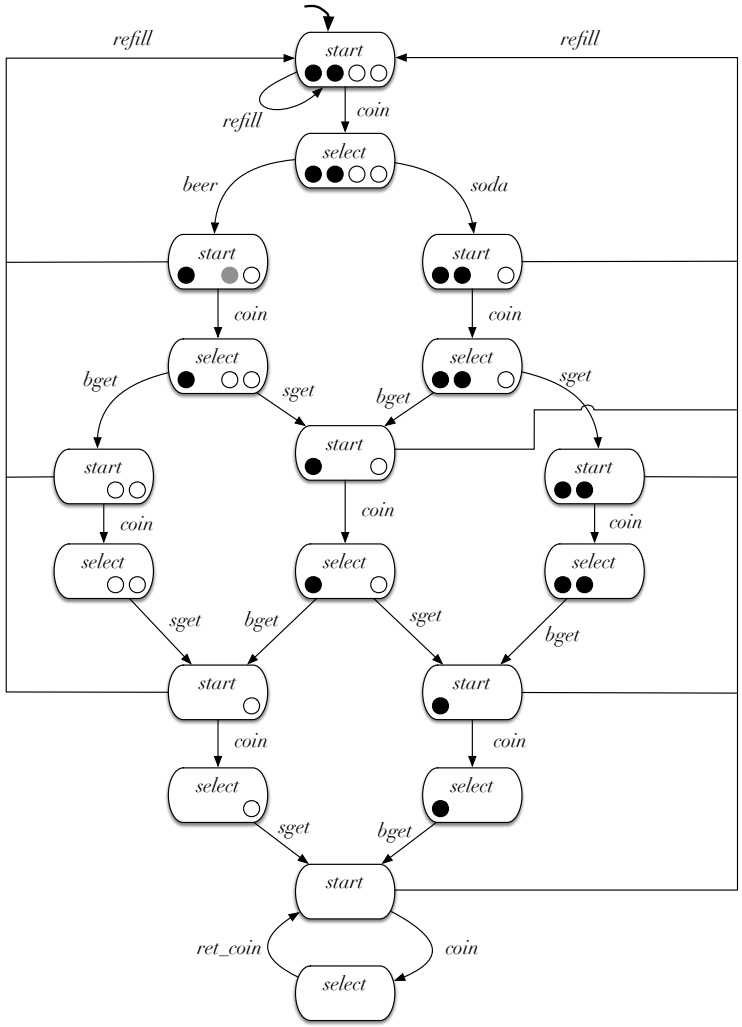


Figure 3.14: *TS* of the extended beverage vending machine

Program graph

- Instead of a *TS*, a **program graph** *PG* is the right notion, as it deals with **conditional transitions** and allows variables Var.
- The role of $\mathcal{P}rop$ and the labelling function π is taken over by an evaluation function Eval, that assigns values to the typed variables in Var (see Definition 3.10).
- The program graph is a directed graph, the nodes of which are called locations: they take the role of the **states** in a *TS*.

Program graph

Definition 3.10 (Program graph)

A program graph PG over a set of typed variables Var is a directed graph where the edges are labelled with conditions and actions: it is a tuple $\langle \text{Loc}, \text{Act}, \text{Effect}, \hookrightarrow, \text{Loc}_0, \mathcal{P}rop, g_0 \rangle$ where

- Loc is a set of **locations**, denoted by $l_1, l_2, \ldots$,
- Act is a set of **actions**, denoted by $\alpha, \beta, \gamma, \ldots$,
- Eval(Var) is the set of all assignments ϱ of values to variables in Var (compatible with their type).
- Effect : $\text{Act} \times \text{Eval}(\text{Var}) \to \text{Eval}(\text{Var})$ takes an assignment ϱ, applies action α and computes the resulting assignment ϱ'.
- $\hookrightarrow \subseteq \text{Loc} \times \text{Cond}(\text{Var}) \times \text{Act} \times \text{Loc}$ is a **conditional transition relation**,
- $\text{Loc}_0 \subseteq \text{Loc}$ is the set of **initial locations**,
- $g_0 \in \text{Cond}(\text{Var})$ is the initial condition.

Boolean conditions over Var

- Cond(Var) is the set of all boolean conditions over Var. E.g.:

$$(-3 \leq x \lneq 5) \wedge (y = \text{blue}) \vee (z = \square)$$

- How can we define these conditions formally?
- Assume we consider variables $x_1, \ldots, x_n$ of particular types ranging over domains $\mathrm{dom}_1, \ldots, \mathrm{dom}_n$. For $1 \leq i \leq n$ let $D_i \subseteq \mathrm{dom}_i$. We consider the set $\mathcal{P}rop_{\mathsf{Var}}$ of sentential constants of the form "$\mathsf{p}_{x \in D_i}$" (for arbitrary i, and variable x of type i and D_i). $\mathsf{p}_{x \in D_i}$ is true whenever the actual value of x is indeed in D_i, otherwise it is false.

  Then Cond(Var) is the set of all SL formulae over $\mathcal{P}rop_{\mathsf{Var}}$.

PG of extended beverage vending machine

Var: $\mathrm{nsoda}, \mathrm{nbeer}$, with domain $\{0, 1, \ldots, \mathrm{max}\}$;

Loc: start, select;

Act: bget, sget, coin, ret-coin, refill;

Effect: $\langle \mathsf{coin}, \eta \rangle \mapsto \eta$, $\langle \mathsf{ret\text{-}coin}, \eta \rangle \mapsto \eta$,
$\langle \mathsf{sget}, \eta \rangle \mapsto \eta[\mathrm{nsoda} - 1/\mathrm{nsoda}]$,
$\langle \mathsf{bget}, \eta \rangle \mapsto \eta[\mathrm{nbeer} - 1/\mathrm{nbeer}]$,
$\langle \mathsf{refill}, \eta \rangle \mapsto \eta'$, where $\eta'(\mathrm{nsoda}) := \eta'(\mathrm{nbeer}) := \mathrm{max}$.

$\hookrightarrow$: obvious from Figure 3.14,

Loc_0: start,

g_0: $\mathrm{nsoda} = \mathrm{max} \ \wedge \ \mathrm{nbeer} = \mathrm{max}$.

Synchronous product

- Often there exists a **central clock** that allows two transition systems to be synchronized. e.g. **synchronous hardware circuits**.
- This leads to the synchronous product $TS_1 \otimes TS_2$ of two transition systems: both systems have to perform **all steps** in a synchronous fashion.

Synchronous product (cont.)

Definition 3.11 (Synchronous product: $TS_1 \otimes TS_2$)

Given two transition systems TS_1, TS_2 ($\langle \mathrm{S}_i, \mathrm{Act}, \longrightarrow_i, I_i, \mathcal{P}rop_i, \pi_i \rangle$, $i = 1, 2$) with a common set of actions Act, we define the synchronous product $TS_1 \otimes TS_2$ by

$$\langle \mathrm{S}_1 \times \mathrm{S}_2, \mathrm{Act}, \longrightarrow, I_1 \times I_2, \mathcal{P}rop_1 \cup \mathcal{P}rop_2, \pi \rangle$$

where the labelling π and $\longrightarrow$ are defined by

- $\pi(\langle s_1, s_2 \rangle) =_{def} \pi_1(s_1) \cup \pi_2(s_2)$,
- $\langle s_1, s_2 \rangle \xrightarrow{\alpha \star \beta} \langle s_1', s_2' \rangle$ *if, by definition,* $s_1 \xrightarrow{\alpha}_1 s_1'$ and $s_2 \xrightarrow{\beta}_2 s_2'$.

$\star$ is a mapping from Act $\times$ Act into Act that assigns to each pair $\langle \alpha, \beta \rangle$ an action $\alpha \star \beta$.

$\star$ is usually assumed to be commutative and associative. Often action names are irrelevant (e.g. for hardware circuits), so they do not play any role in such cases.

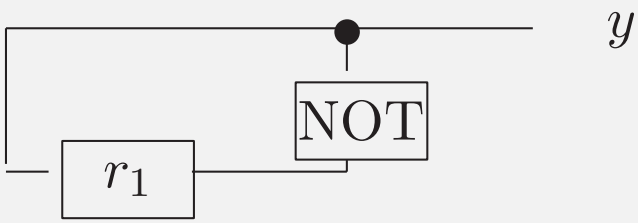
Synchronous product

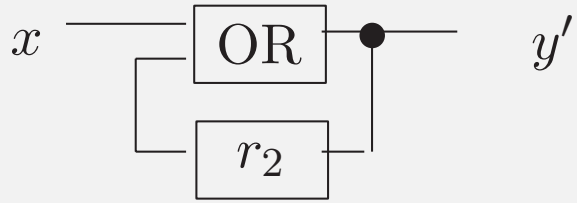
- There is no autonomy, as in the **interleaving operator** to be introduced below in Definition 3.14 on Slide 197.
- Normally, when asynchronous systems are represented by transition systems, one does not make any assumptions about **how long the processes take** for execution. Only that these are finite time intervals.

Synchronous product

Example 3.12 (Synchronous product of two circuits)

We consider the following circuits. The first one has no input variables (output is y defined by r_1 and register transition $\neg r_1$), the second one has input x, and output y' defined by $x \vee r_2$ and register transition $x \vee r_2$.

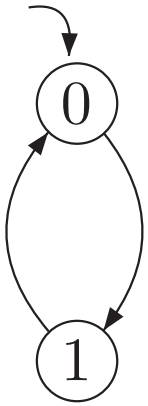




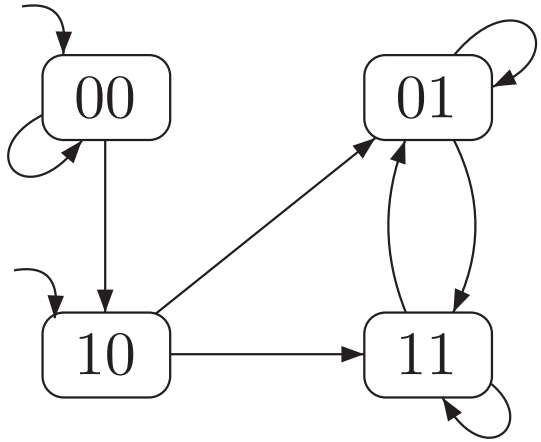
Synchronous product

The corresponding transition systems are:

TS_1 :



TS_2 :



Synchronous product

The synchronous product of Example 3.12 is:

$TS_1 \otimes TS_2$:

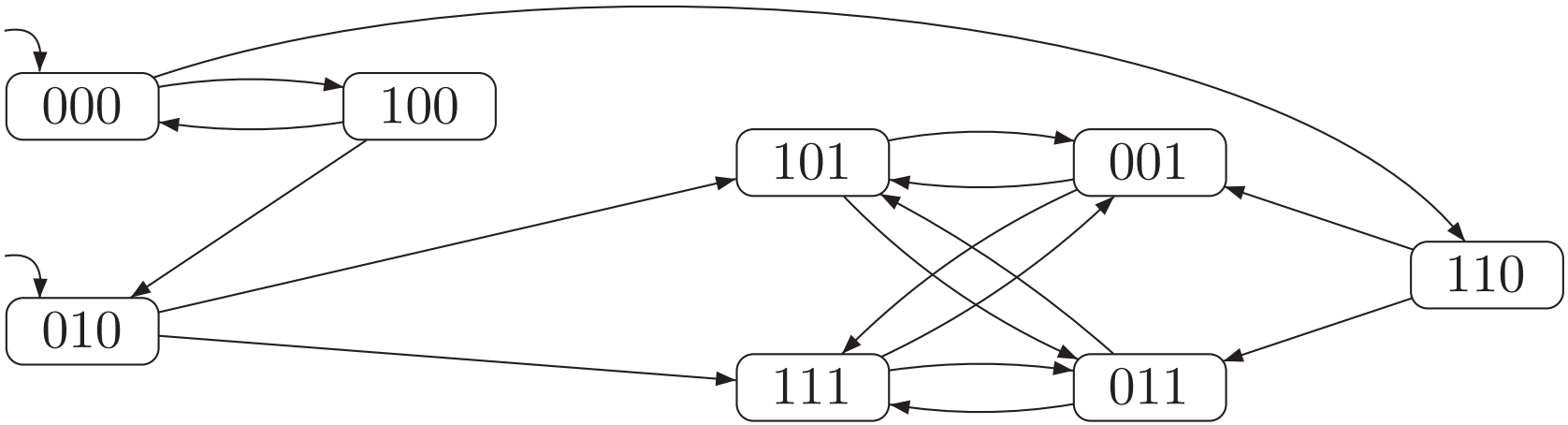


Figure 3.15: Synchronous product of two circuits

3.3 Interleaving and handshaking

Interleaving independent processses

How can we merge two completely independent transition systems? (Think of finite state automata!)

Consider traffic lights that can simply switch between $green$ and red.

Interleaving independent processses (cont.)

Example 3.13 (Two independent traffic lights)

Two independent traffic lights on two roads.

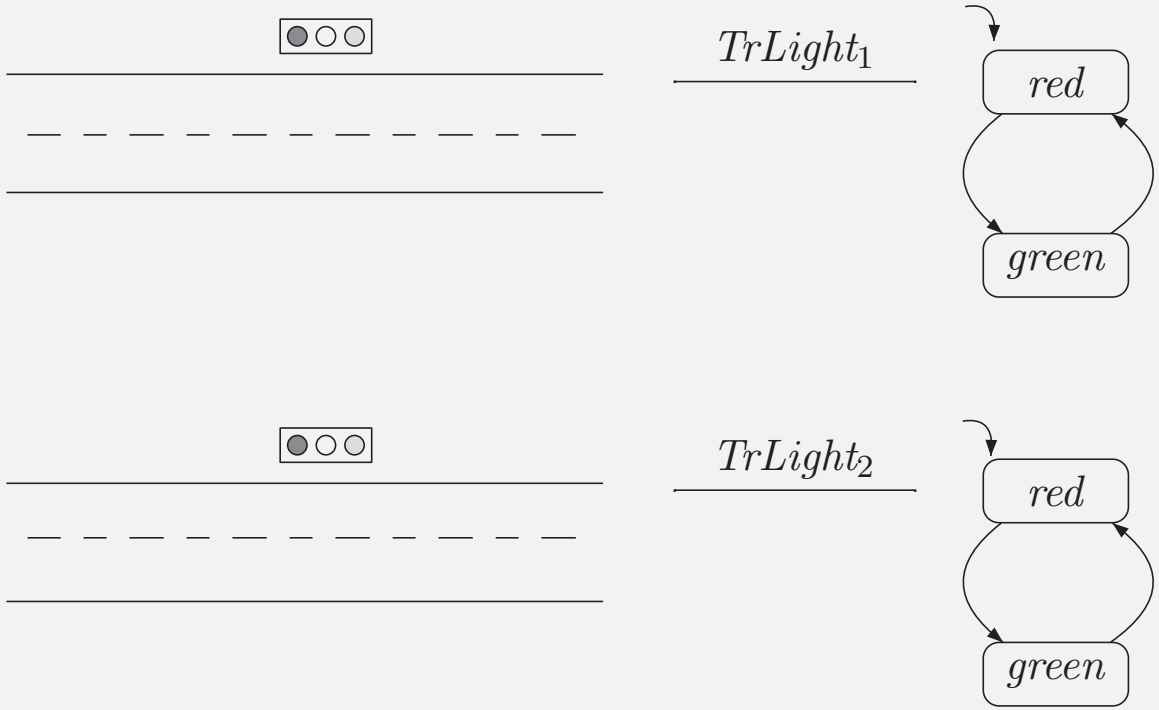


Figure 3.16: *TS* of two independent traffic lights

Interleaving independent processses (cont.)

The result should be:

$$TrLight_1 \;|||\; TrLight_2$$

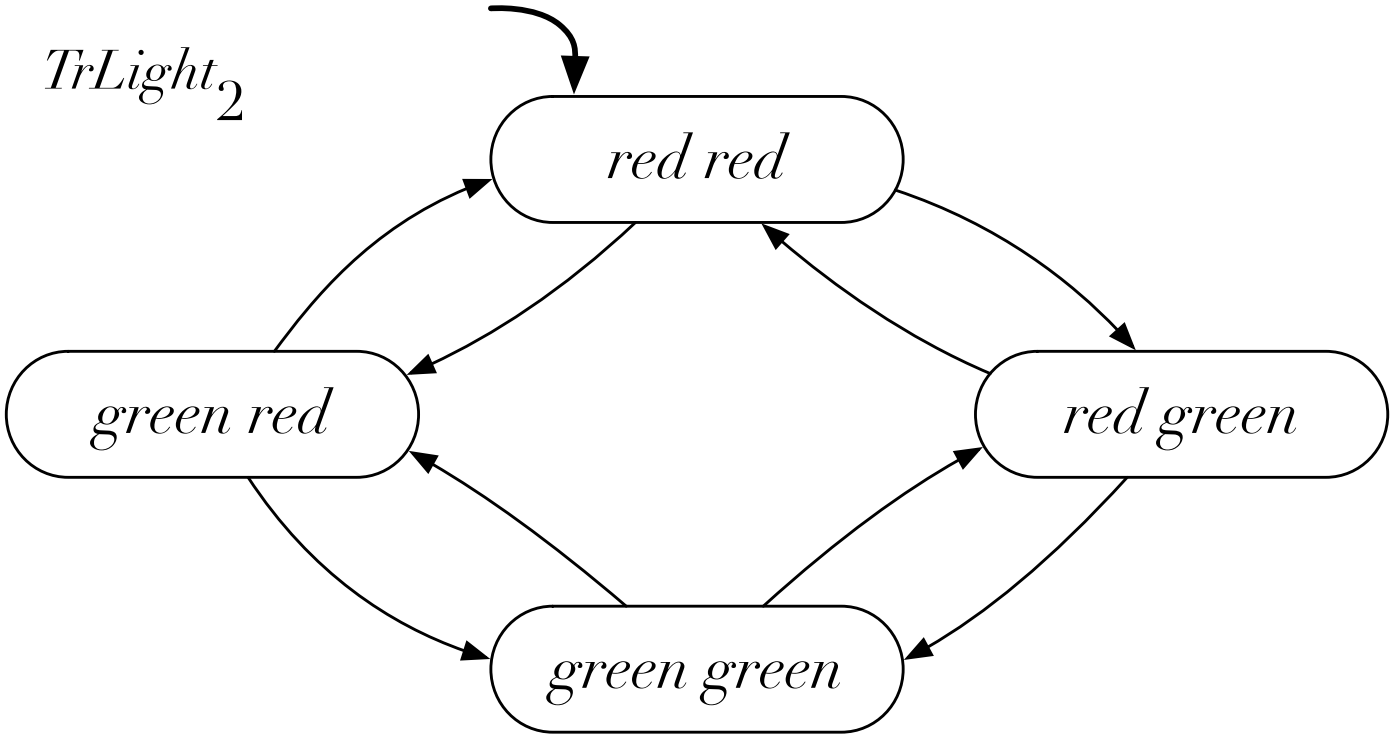


Figure 3.17: *TS* of two traffic lights

Interleaving independent processses

The result of interleaving the following processes is also obvious.

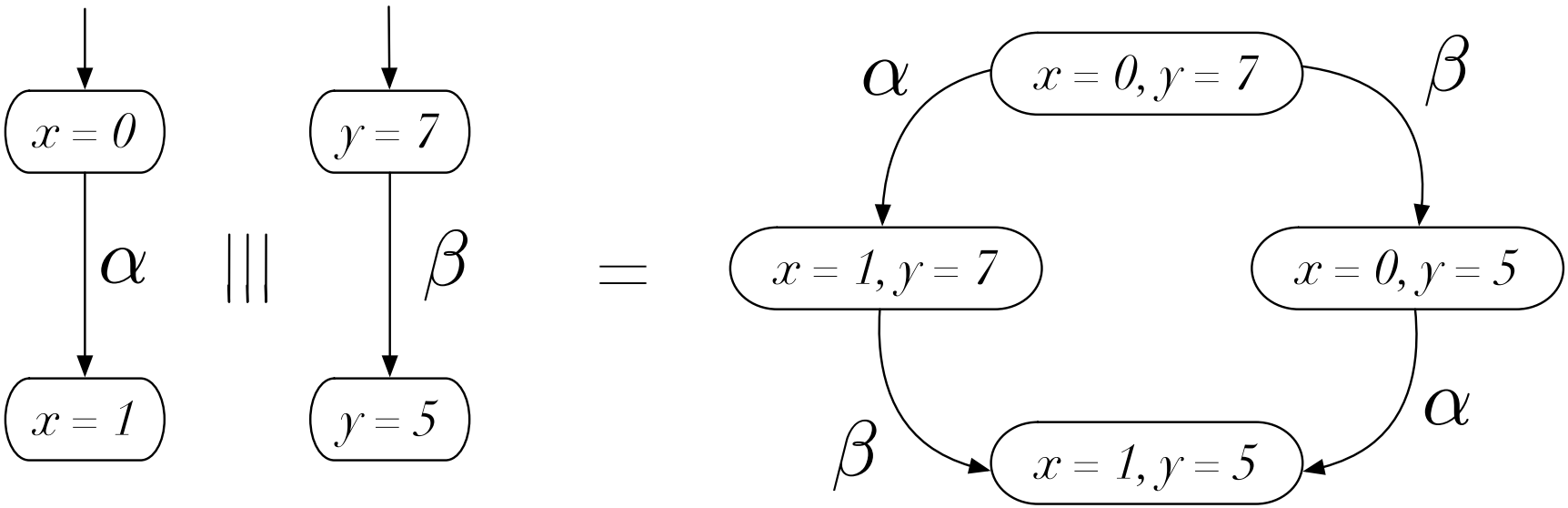


Figure 3.18: *TS* of two independent processes

Definition 3.14 (Interleaving: $TS_1 \,|||\, TS_2$)

Given two transition systems TS_1, TS_2 ($\langle S_i, Act_i, \longrightarrow_i, I_i, \mathcal{P}rop_i, \pi_i \rangle$, $i = 1, 2$) we define the interleaved transition system $TS_1 \,|||\, TS_2$ by

$$\langle S_1 \times S_2, Act_1 \cup Act_2, \longrightarrow, I_1 \times I_2, \mathcal{P}rop_1 \cup \mathcal{P}rop_2, \pi \rangle$$

where the labelling π and $\longrightarrow$ are defined by

- $\pi(\langle s_1, s_2 \rangle) =_{def} \pi_1(s_1) \cup \pi_2(s_2)$,
- $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2' \rangle$ *if, by definition,* $s_1 \xrightarrow{\alpha}_1 s_1'$ or $s_2 \xrightarrow{\alpha}_2 s_2'$.

In contrast to the synchronous product (Definition 3.11 on Slide 187) there is no clock: executions do not depend on time.

Shared variables

The examples on Slides 195 and 196 are simple, because there are no shared variables.
What happens, if we do the same construction in the following example:

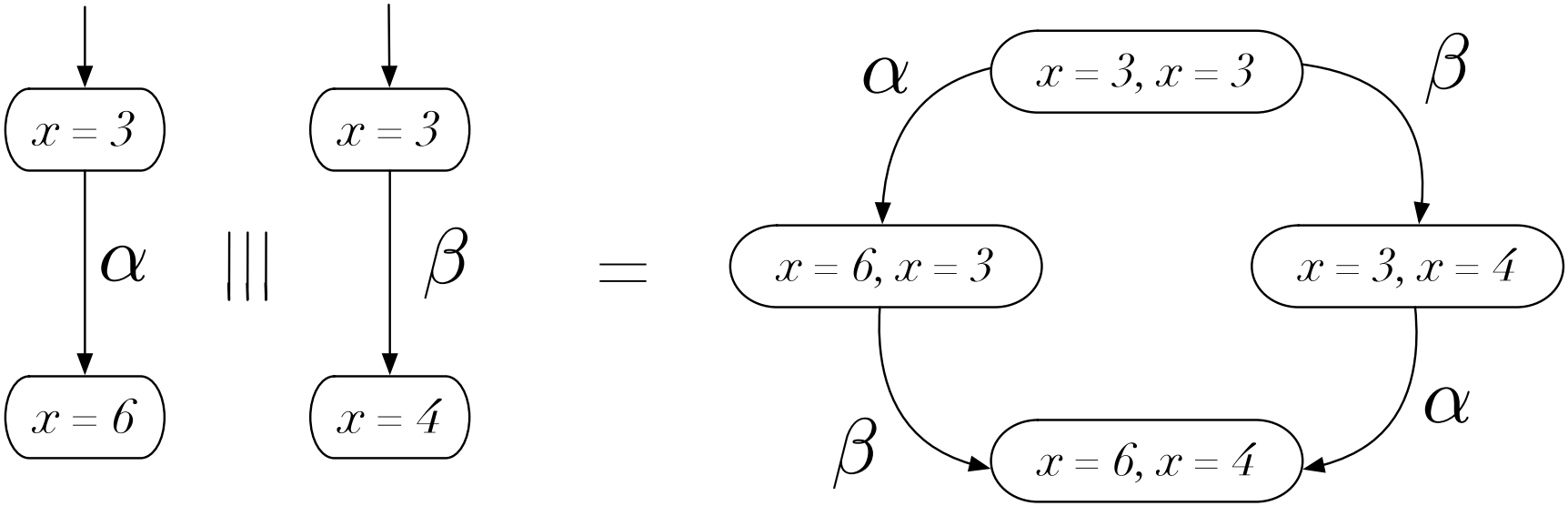


Figure 3.19: *TS* of processes with shared variable

Semaphores

Example 3.15 (Mutex via a semaphore)

Two processes P_1 and P_2 have access to a binary variable y, a **semaphore**. Value 0 of y means the semaphore **is possessed** by one process, value 1 means it is free.

- Each process P_i is in one of three states: noncrit_i, wait_i, crit_i.
- There is a transition from noncrit_i to wait_i and from wait_i to crit_i but the last one only when y is 1 (then the transition fires and the value is set to 0).
- There is a transition from crit_i to noncrit_i and y is set to 1.

How can we model this as a TS?

Semaphores (cont.)

We first model it as program graphs.

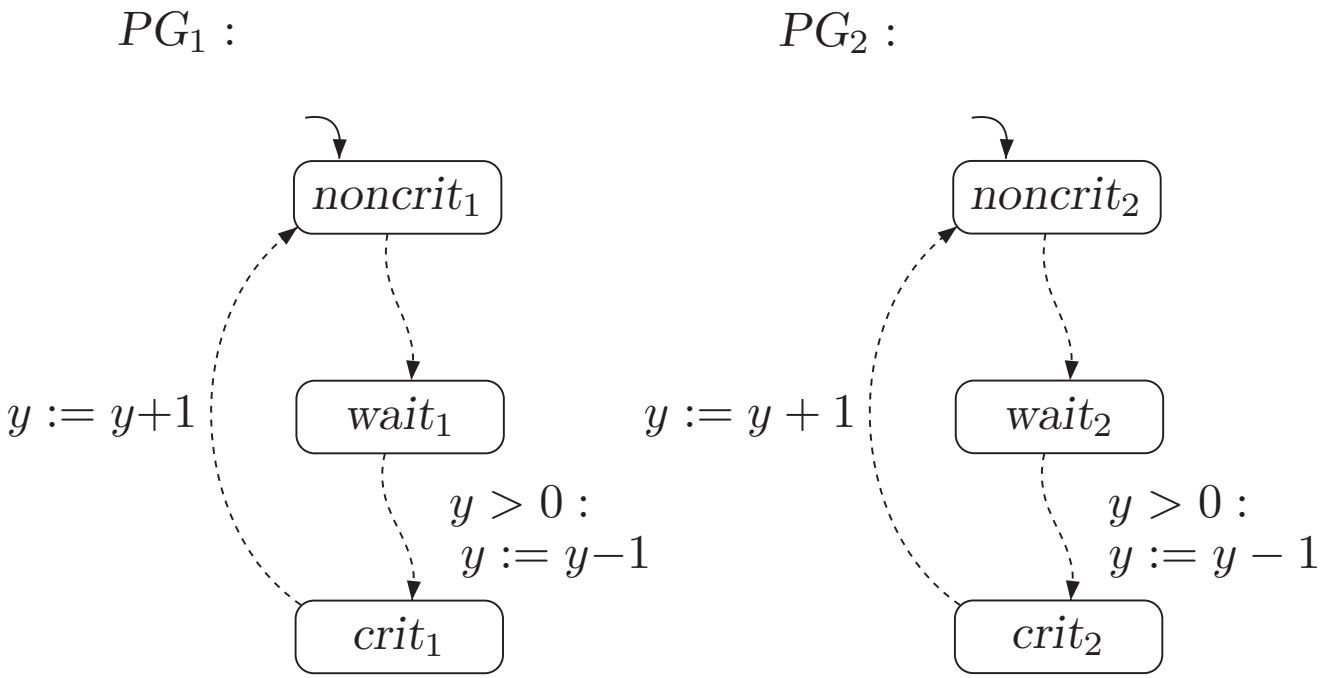


Figure 3.20: Semaphores and program graphs

Interleaved program graph

A construction similar to Definition 3.14 yields an **interleaved program graph**

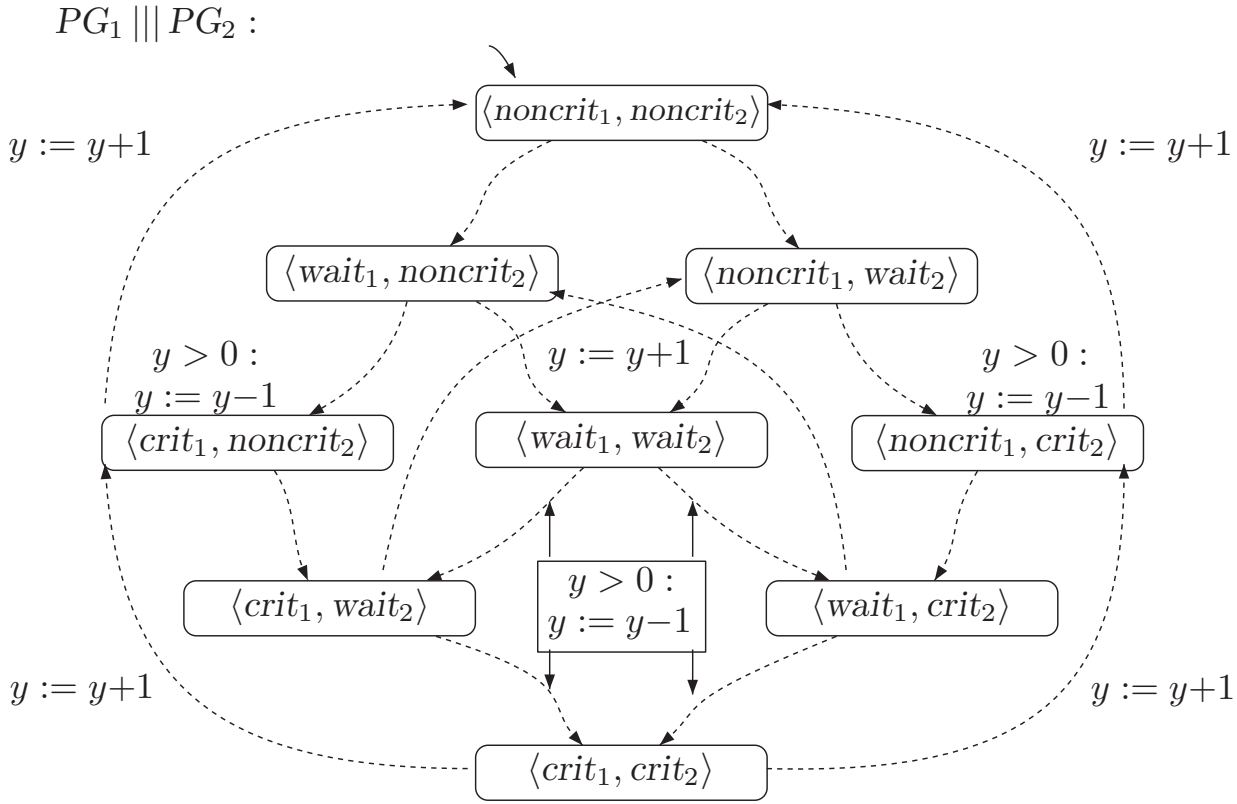


Figure 3.21: Interleaving program graphs

Unfolding program graph into *TS*

Here is the transition system that we obtain from Figure 3.21

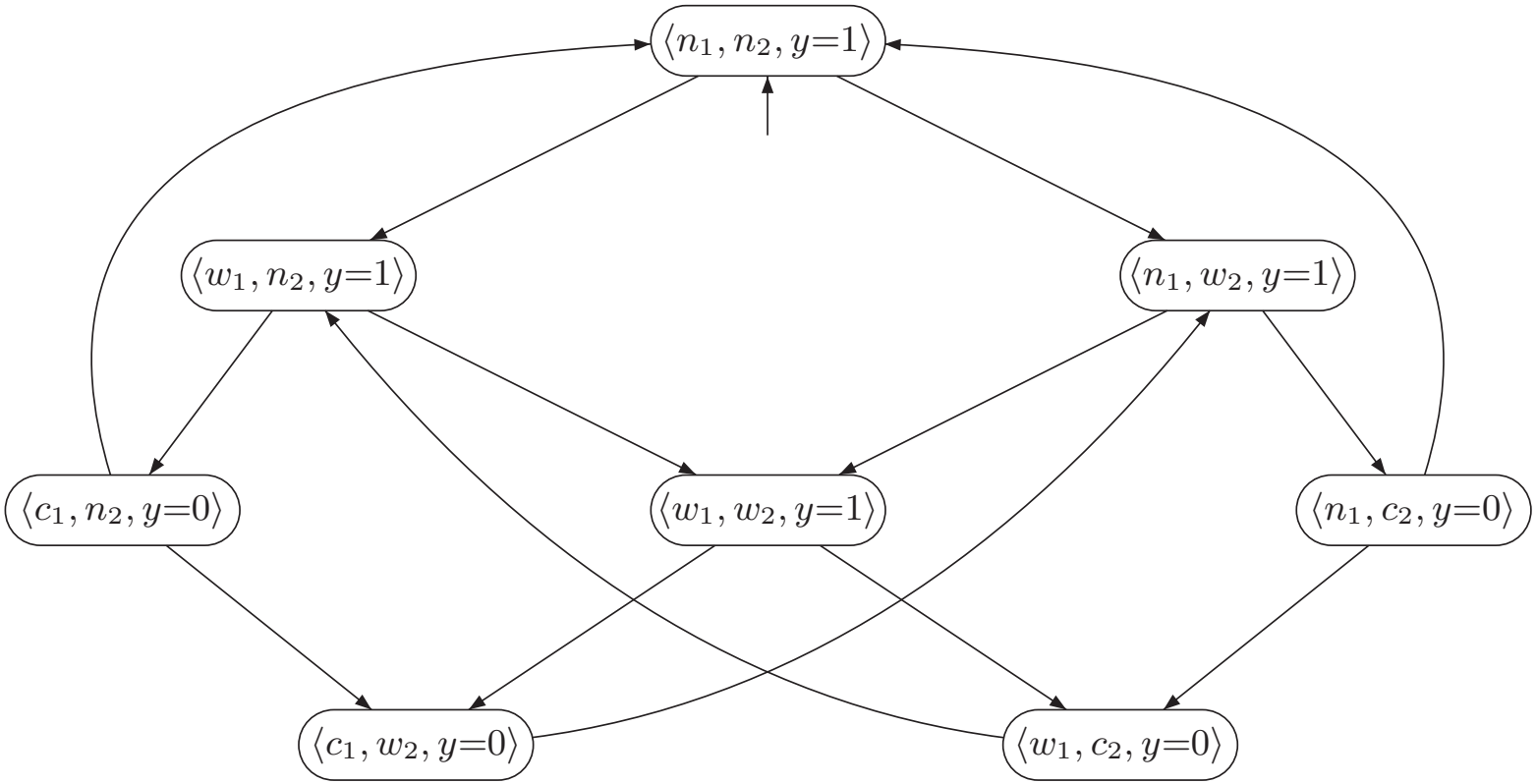


Figure 3.22: The final transition system

Semaphores (cont.)

- The transition system **does not contain anymore the critical state** (because it is not reachable). The system satisfies the mutual exclusion property.
- But it is possible that both processes are at the same time in the waiting state: this is a **deadlock** situation that should be avoided (by an appropriate scheduling algorithm to implement the nondeterministic choice).

What have we achieved so far?

- Interleaving of transition systems works well for **independent** systems.
- It is not appropriate when synchronization of certain parts is essential.
- Instead of a *TS*, the **program graph** *PG* (introduced in Definition 3.10 on Slide 183) is the right notion, as it deals with **conditional transitions** and allows **shared** variables Var.
- We have just seen that shared variable communication (e.g. semaphores) can be used to model synchronization.

What have we achieved so far? (cont.)

- An interleaving of program graphs can be defined similarly to Definition 3.14: see Slide 201.
- This **interleaved graph** can again be unfolded to a transition system *TS*: see Slide 202.
- We end up again with model checking transition systems.

We are now introducing another method for modelling **synchronization**: handshaking, a refined version of interleaving on the level of transition systems (not program graphs).

From interleaving to handshaking

- Concurrent processes that need to interact can do so in a synchronous fashion via handshaking.
- Sometimes one process has to wait for the other to finish: they have to synchronize.
- We introduce a set of distinguished handshake actions H.
- We define a transition system $TS_1 \|_H TS_2$ that takes into account the handshakes in H.
- For an empty set H we get back our notion of an interleaved transition system $TS_1 \,|||\, TS_2$.

Example 3.16 (Traffic junction)

We consider a traffic junction with two traffic lights. Both switch from green to red and green etc., but in a synchronized way: when one is green the other one should be red (to avoid accidents).

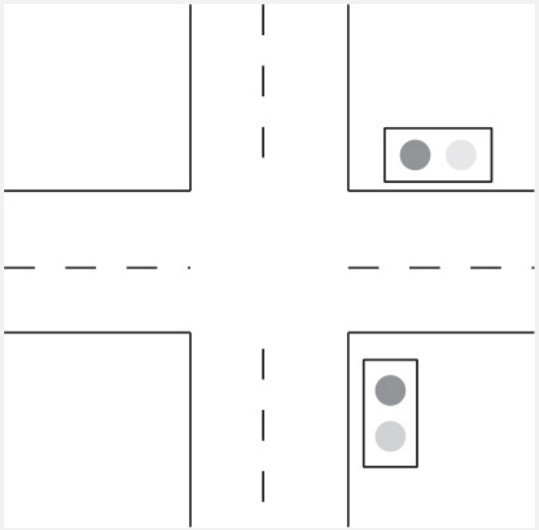


Figure 3.23: Traffic junction

Traffic junction

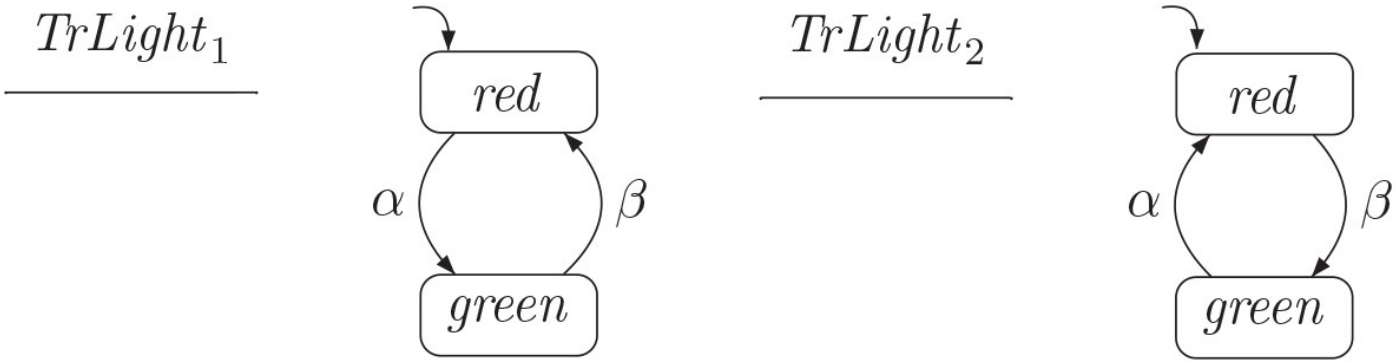
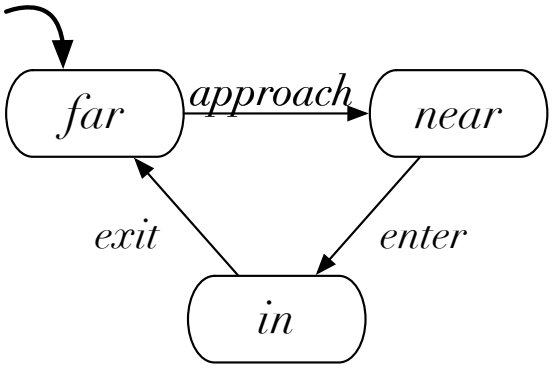


Figure 3.24: Traffic junction

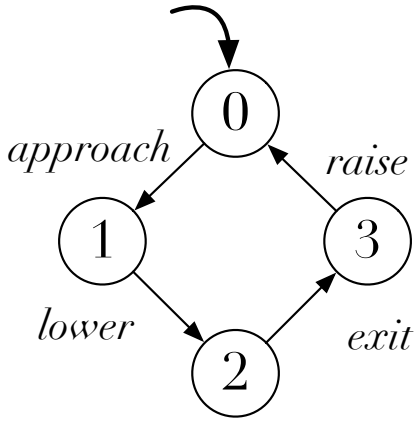
The interleaved system $TrLight_1 \;|||\; TrLight_1$ does not work anymore!

Example 3.17 (Railroad crossing)

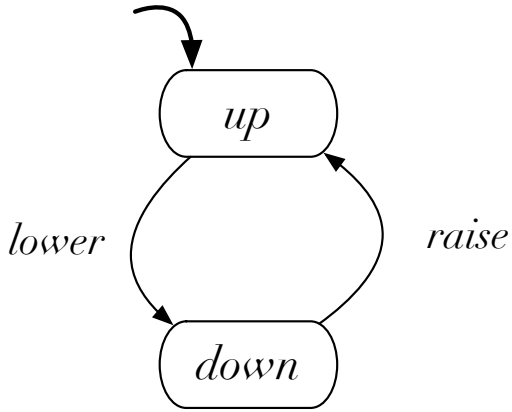
A railroad crossing consists of a train, a gate and a controller. An approaching train sends a signal so that the gates have to be closed. The gates open again only when another signal, indicating that the train has passed, has been sent.



Train



Controller



Gate

Figure 3.25: Components of Railroad crossing

Mutual exclusion in general

Suppose we have two processes P_1, P_2 with **shared variables**.

- When they are operating on these shared variables, they are in a **critical** phase. Otherwise they are in an **uncritical** phase.
- We assume the processes **alternate between critical and noncritical** phases (infinitely often).
- The problem is to avoid concurrency when **both are in critical phases**.

Mutual exclusion in general (cont.)

- Mutual exclusion algorithms are scheduling algorithms to ensure that no critical actions are executed in parallel.
- One of the most prominent solutions is Peterson's mutex algorithm.
- As Peterson's algorithm is based on the **interleaving of program graphs** (which we have not formally introduced but illustrated on Slides 200—202), we introduce a variant based on $\|_H$ between transition systems.

Definition 3.18 (Handshaking: $TS_1 \|_H TS_2$)

For transition systems TS_1, TS_2 ($\langle S_i, Act_i, \longrightarrow_i, I_i, \mathcal{P}rop_i, \pi_i \rangle$), $H \subseteq Act_1 \cap Act_2$ and $\tau \notin H$, we define the transition system with handshaking $TS_1 \|_H TS_2$ by

$$\langle S_1 \times S_2, Act_1 \cup Act_2, \longrightarrow, I_1 \times I_2, \mathcal{P}rop_1 \cup \mathcal{P}rop_2, \pi \rangle$$

where the labelling π and $\xrightarrow{\alpha}$ are defined by

- $\pi(\langle s_1, s_2 \rangle) =_{def} \pi_1(s_1) \cup \pi_2(s_2)$,
- for $\alpha \notin H$, $\xrightarrow{\alpha}$ is defined as in Definition 3.14,
- for $\alpha \in H$, $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2' \rangle$ *if, by definition,* $s_1 \xrightarrow{\alpha}_1 s_1'$, and $s_2 \xrightarrow{\alpha}_2 s_2'$.

When $H = Act_1 \cap Act_2$, we simply write $TS_1 \| TS_2$ instead of $TS_1 \|_H TS_2$. When $H = \emptyset$ then $TS_1 \|_\emptyset TS_2 = TS_1 \,|||\, TS_2$.

Example 3.19 (Mutual exclusion via an arbiter)

Suppose we have two processes P_1 and P_2. Both alternate infinitely often between critical and noncritical phases. The interleaved system $TS_1 \,|||\, TS_2$ has a critical state that has to be avoided (shared resources). **How can a third process, the arbiter, be used to resolve the conflicts?**

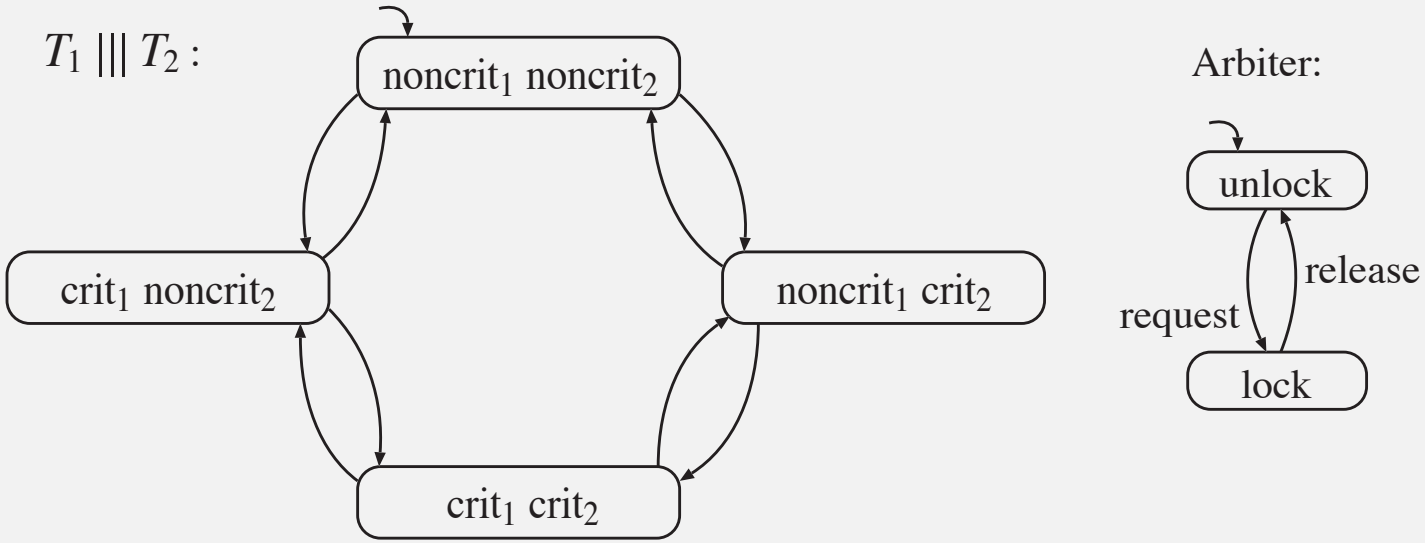


Figure 3.26: Transition systems and arbiter

Mutual exclusion via an arbiter (cont.)

Asssume P_1 and P_2 have each actions request and release. We set $H = \{\text{request}, \text{release}\} = \text{Act}_1 \cap \text{Act}_2$ and get the system:

$$(T_1 \;|||\; T_2) \;||\; \text{Arbiter} :$$

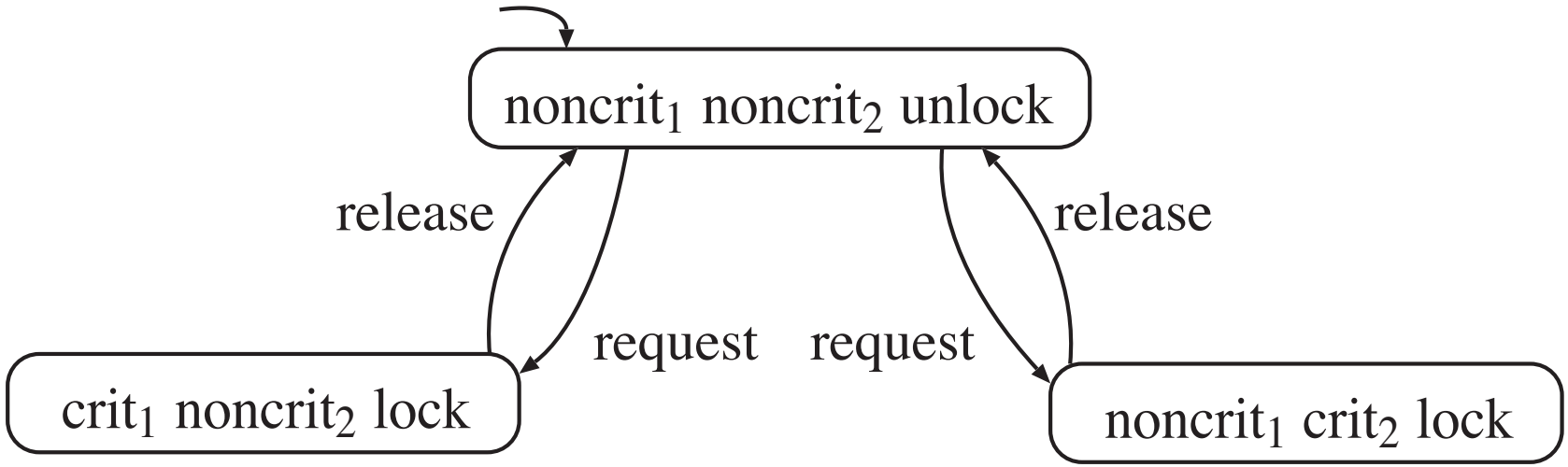


Figure 3.27: Mutual exclusion via arbiter

Traffic junction revisited

- We reconsider Example 3.16 from Slide 207.
- This time we build $TS_1 \|_H TS_2$ with $H = \{\alpha, \beta\}$ and it all works!
- However, there is a small problem. The state $\langle red, red \rangle$ is a **deadlock**.
- So even if two transition systems do not have deadlocks (an assumption to be made on Slide 228) their parallel composition might have.

Railroad crossing revisited

- We reconsider Example 3.17 from Slide 209.
- This time we build

$$Train \|_{\{approach,exit\}} Controller \|_{\{lower,raise\}} Gate,$$

or, according to our convention, simply

$$Train \| Controller \| Gate.$$

Railroad crossing: almost working

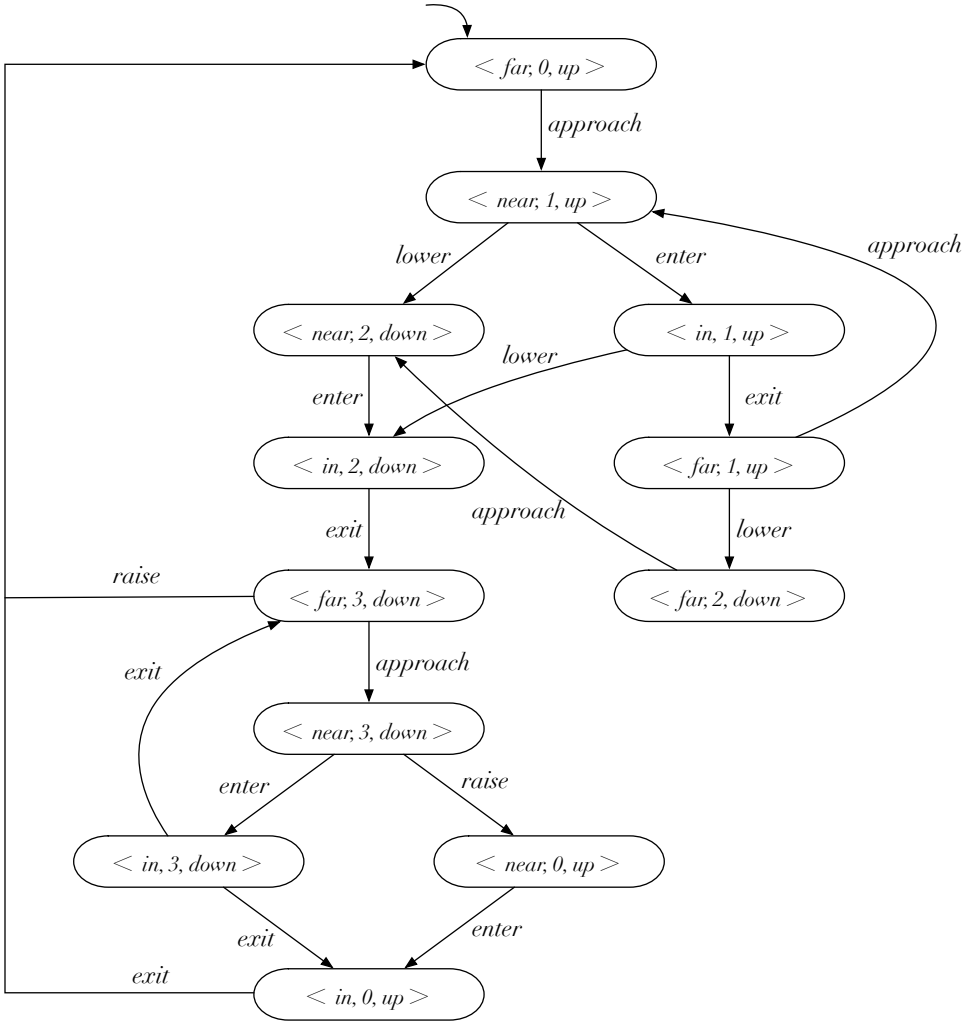


Figure 3.28: *TS* of Railroad crossing

Dining Philosophers revisited

We reconsider Example 1.2 from Slide 40. An obvious representation is for each philosopher to have **four actions available** (request left stick, request right stick, release left stick, release right stick) — we model "thinking" and "waiting" in the states. And for each stick also two requests and two releases (from the adjacent philosophers). This results in the **overall system**

$$Phil_0 || Stick_4 || Phil_4 || Stick_3 || Phil_3 || Stick_2 || \ldots || Phil_4 || Stick_0$$

Dining Philosophers revisited (cont)

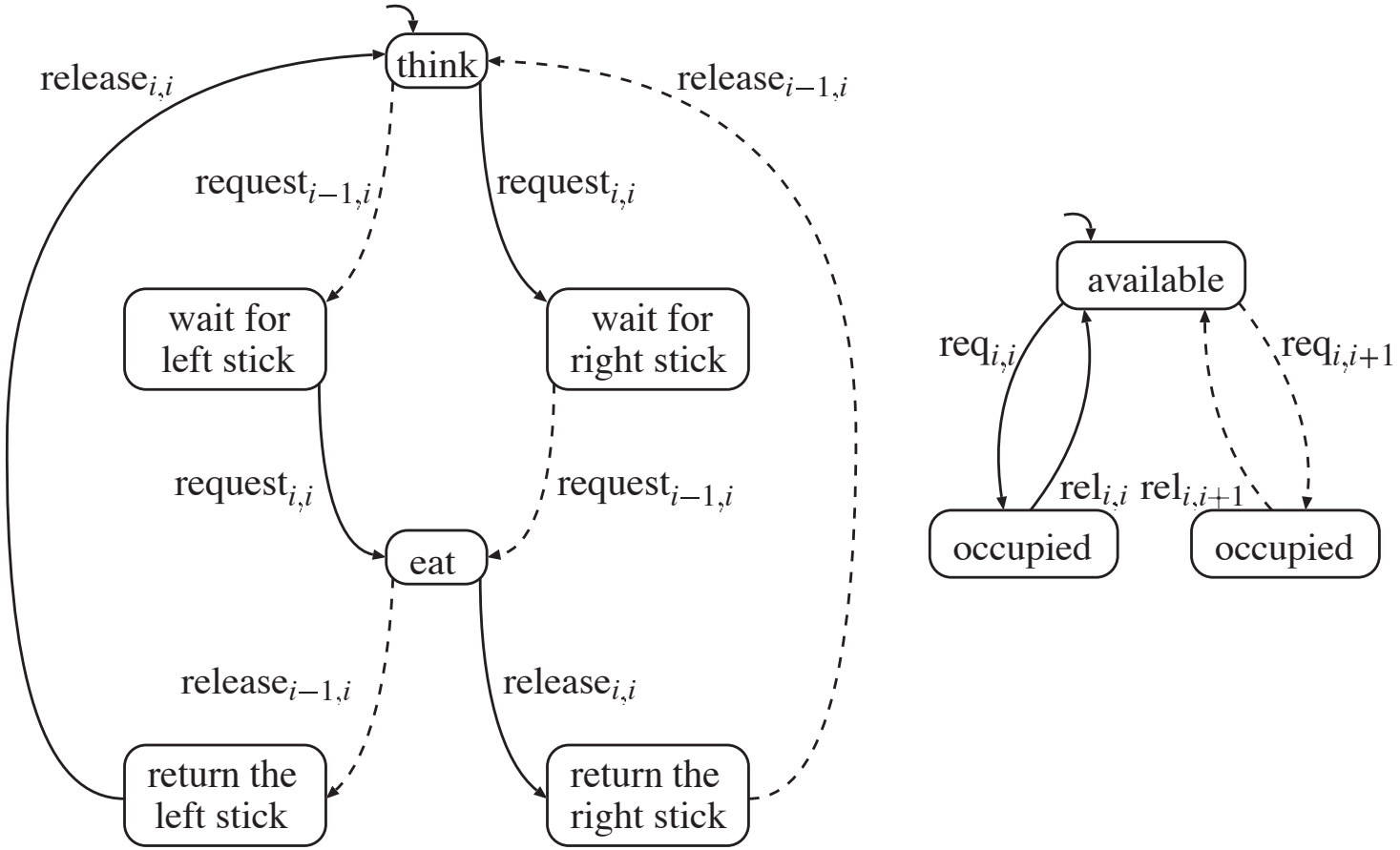


Figure 3.29: Dining Philosophers

Dining Philosophers revisited (cont.)

- What is wrong with this model?
- deadlock, starvation.
- How can it be improved?
- Idea: make sticks available only for one philosopher at a time.
- And make sure, that one half of the sticks start in a different state than the other half.

Dining Philosophers revisited (cont)

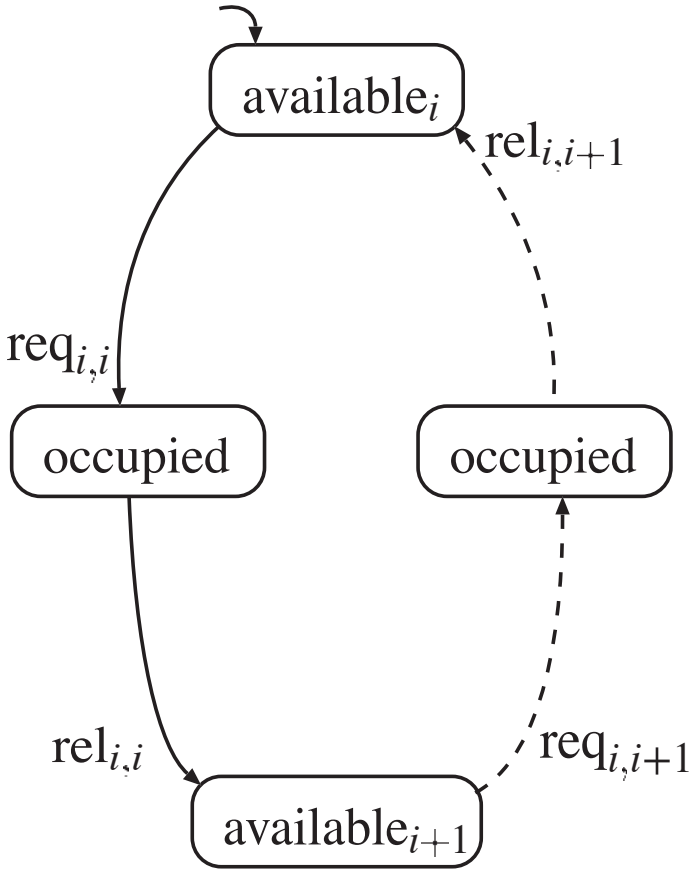


Figure 3.30: Dining Philosophers refined

3.4 State-space explosion

Input size

- Remember: input size of an integer n is $\log n$, not n.
- n is **exponential** in $\log n$.
- Similarly, the program graph is a compact representation of the underlying transition system, which is obtained by **unfolding**.
- When asking **"how complex is it to check a property of a *TS*"**, the representation of the *TS* matters.
- Size of program graph versus size of TS.

Size of unfolded transition system

What is the size of a transition system obtained from unfolding a program graph with variables $x \in \mathrm{Var}$?

- Size of **underlying domains** is important.
- $\mathrm{dom}(x)$ infinite: unfolded transition system is **infinite** and undecidability issues arise.
- $\mathrm{dom}(x)$ finite: number of states is $|\mathrm{Loc}| \ \prod_{x \in \mathrm{Var}} \ |\mathrm{dom}(x)|$.
- $\#\,\mathrm{states}$ is exponential in $\#\,\mathrm{variables}$.

Influence of $|\mathcal{P}rop|$ and $|S|$

Clearly also the **# constants** in a state plays a role. The same is true for the size of the **state space of the interleaved system**.

- $\mathcal{P}rop$ can be large, due to the allowed **conditions of the variables** in the program graph. In practice only a small number of such conditions are interesting.
- What about the labelling function π?
- Representing π explicitly is expensive. Often, this info can be derived from the state.
- State space of $TS_1 \,|||\, \ldots \,|||\, TS_n$ is cartesian product: $\Pi_{i=1}^{n} S_i$. Size is exponential in # components.
- In general, the size of the TS obtained from a program graph is exponential in the size of the program graph.

3.5 LT properties in general

LT properties of transition systems

Transition systems are abstractions from systems in the real world. We would like to

- reason about particular computations of a *TS*;
- reason about all possible executions in a *TS*,
- formulate interesting properties such a *TS* should satisfy.

Main notion: a path in a *TS*. It is a sequence of states starting in an initial state. It corresponds to an execution of the *TS* and is closely related to the notion of run introduced in Definition 3.8 on Slide 179.

Transitions systems without terminal nodes

From now on, we assume wlog that all transition systems do not have any terminal nodes. In case a *TS* has terminal nodes, we could simply add for each such node a new action and a new state (with an arrow pointing to itself) and extend the *TS* appropriately.

Definition 3.20 (Path λ (of a *TS*))

A path $\lambda : \mathbb{N}_0 \to \mathrm{S}$ in a *TS* is a sequence of states starting with an initial state such that this sequence corresponds to a **run of the *TS***.

Paths versus traces

Often, we are not so much interested in the states as such, only **what is true** in them, i.e. which propositions from $\mathcal{P}rop$ are true: $\pi(s_i)$.

Definition 3.21 (Trace of λ of a *TS*)

The **trace** of a path $\lambda : \mathbb{N}_0 \to \mathrm{S}$ in a *TS* with $\mathcal{P}rop$ is the following infinite sequence

$$\pi(\lambda(0))\pi(\lambda(1))\pi(\lambda(2))\ldots\pi(\lambda(i))\ldots$$

($\pi(\lambda(i))$ is the set of all propositions that are true in state $\lambda(i)$). We call this also an **ω-word** over $2^{\mathcal{P}rop}$.

The set of all traces of a *TS* is the set of the traces of all paths from *TS*: it is denoted by *Traces*(*TS*).

Runs or paths?

We want to define what it means that
two transition systems behave the same.

1. We take the set of **runs** of a *TS* as the defining behaviour.
2. Often the actions do not play any role, only the states do. Then we could take the set of **paths** of a *TS* as the defining behaviour.

Both possibilities rely on the **internal** behaviour, that we might not be able to determine.

Runs or paths? (cont.)

Often one cannot distinguish between certain states, we only know what is true in them (and that **depends on the language** $\mathcal{P}rop$).

3 Therefore we choose from now on the observable behaviour to describe a *TS*: the set of traces as the defining behaviour of a TS.

Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.

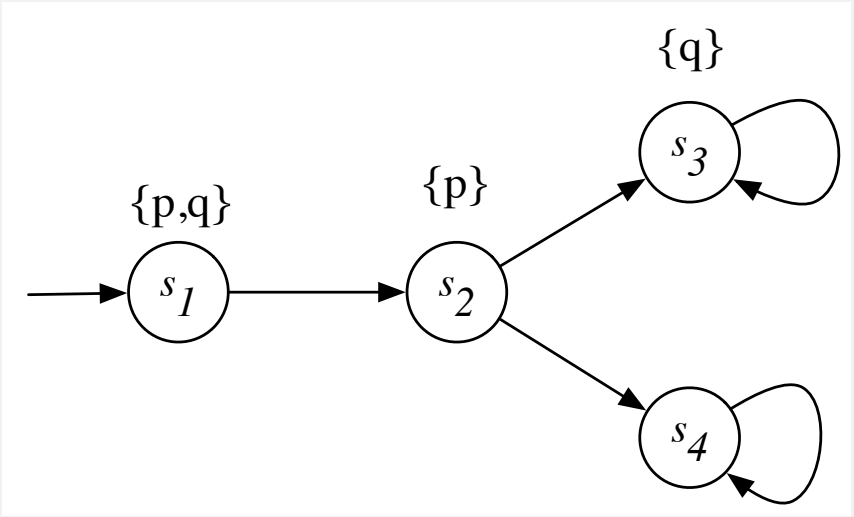
L. Wittgenstein, TLP, Satz 5.6

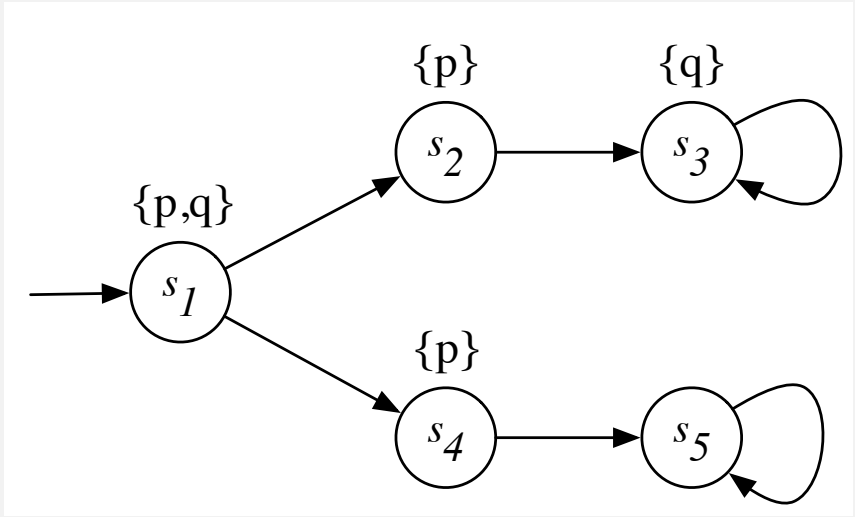
Definition 3.22 (Equivalence of transition systems)

Let TS_1 and TS_2 be two transition systems over $\mathcal{P}rop$ and let $A \subseteq \mathcal{P}rop$. We say that

1. TS_1 and TS_2 are A-equivalent, *if, by definition,* $\text{Traces}(TS_1)\,|_A = \text{Traces}(TS_2)\,|_A$,
2. TS_1 is a correct implementation of TS_2, *if, by definition,* $\text{Traces}(TS_1) \subseteq \text{Traces}(TS_2)$.

Example 3.23 (Two transition systems TS_1, TS_2)





What are the paths, traces of them?

1. $s_1 s_2 s_3^\omega$, $s_1 s_2 s_4^\omega$. $\{p,q\}\{p\}\{q\}^\omega$, $\{p,q\}\{p\}\{\}^\omega$
2. $s_1 s_2 s_3^\omega$, $s_1 s_4 s_5^\omega$. $\{p,q\}\{p\}\{q\}^\omega$, $\{p,q\}\{p\}\{\}^\omega$.
3. Both transition systems have the same set of traces. **Is there any difference between them?**

Properties of transition systems

- "Whenever p holds, a state with q is reachable."
- Obviously this is true in TS_1 but not in TS_2.
- Later: this cannot be expressed in LTL.
- Any property that is solely based on the set of traces of a TS can not distinguish between TS_1 and TS_2.
- But still many useful properties can be defined: linear time (LT) properties.
- The former two transition systems **cannot be distinguished** by any LT-property.

Properties of transition systems

Definition 3.24 (LT properties)

Given a set $\mathcal{P}rop$, a LT property over $\mathcal{P}rop$ is any subset of $(2^{\mathcal{P}rop})^\omega$.
A transition system *TS* satisfies such a property, if all its traces are contained in it.

- A LT property is a, possibly infinite, set of infinite words.

Attention

$\{p,q\}\{p\}\{q\}^\omega$ really means the infinite word $\{p,q\}\{p\}\{q\}\ldots\{q\}\ldots$, not $\{p\}\{p\}\{q\}\ldots\{q\}\ldots$, or $\{q\}\{p\}\{q\}\ldots\{q\}\ldots$ as in regular expressions.

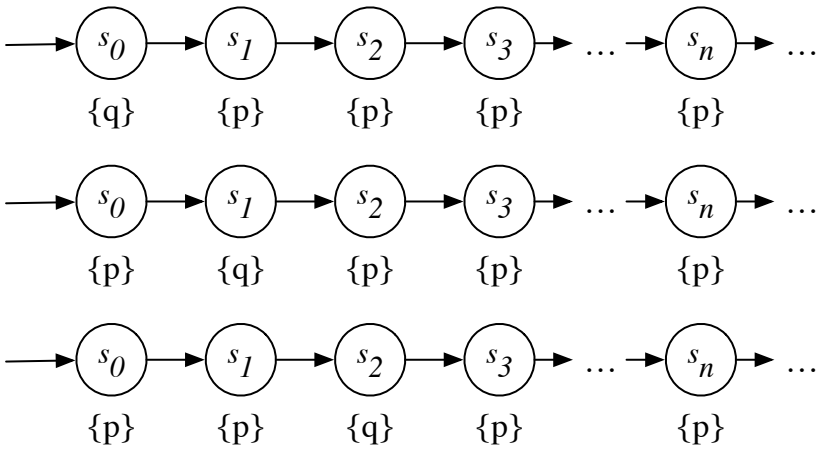
Properties of transition systems

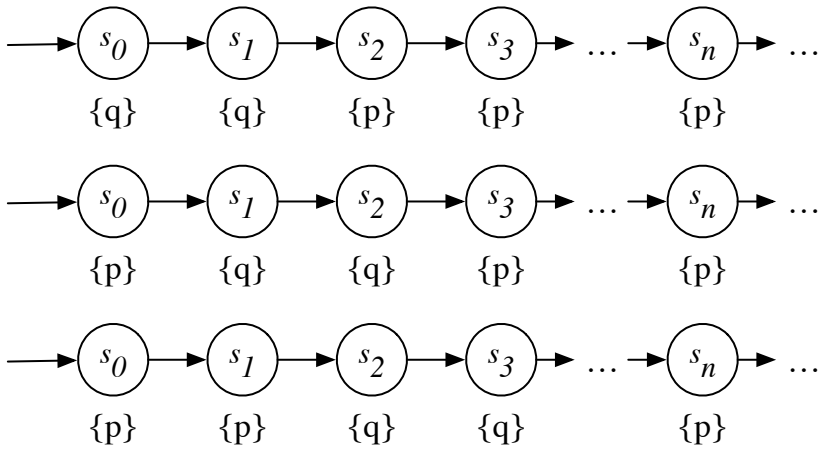
We also use $\{\{p,q\},\{p\}\}\{p\}\{q\}^\omega$ to denote the set of words that either start with $\{p,q\}$ or with $\{p\}$.

Example 3.25

1. When p then q two steps farther ahead.
2. It is never the case that p and q. This is important for mutual exclusion algorithms: one wants to ensure that never two processes are at the same time in their critical section.
3. When p then eventually q. When a message has been sent, it will eventually be received.

Example 3.26 (Sets of paths)





The first infinite set of paths M_1 can be denoted more succinctly by $\{p\}^*\{q\}\{p\}^\omega$ and the second, M_2, by $\{p\}^*\{q\}\{q\}\{p\}^\omega$ (we are using the Kleene * as in regular expressions). These expressions are called ω regular expressions.

Which properties does M_1 satisfy?

- At some time q is true and then always p holds.
- q is true exactly once.
- p is always true with one single exception, when q is true.
- **q is true exactly once (and in that state p is not true), and before and after that, only p holds true.** This will be expressed as a LTL formula shortly.

Can M_1 be represented as a (finite) transition system at all?

Theorem 3.27 (Traces and LT-properties)

Let TS_1 and TS_2 be two transition systems over $\mathcal{P}rop$. Then the following are equivalent:

- $Traces(TS_1) \subseteq Traces(TS_2)$
- *for all LT properties M: if TS_2 satisfies M, so does TS_1.*

Corollary 3.28

TS_1 and TS_2 satisfy the same LT properties if and only if $Traces(TS_1) = Traces(TS_2)$.

More interesting properties

For the formal specification and verification of concurrent and distributed systems, the following **useful concepts can be formally, and concisely, specified** as LT properties (and later also using temporal logics):

- safety properties,
- liveness properties,
- fairness properties.

Safety Properties

Many safety properties are quite simple, they are just conditions on the states and called invariants. They have the form ($A_i \subset 2^{Prop}$)

$$M = \{A_0 A_1 \ldots A_i \ldots : \text{ for all } i: \ A_i \models \varphi\}$$

for a **propositional formula** φ.

Examples are

- **mutual exclusion properties**: $\neg \text{crit}_1 \vee \neg \text{crit}_2$,
- **deadlock freedom**: $\neg \text{wait}_0 \vee \ldots \vee \neg \text{wait}_5$. Deadlock freedom does not imply a fair distribution, i.e. $\neg \text{wait}_0$ can always hold.

Others require **conditions on finite fragments**, for example a traffic light with three phases requiring an **orange phase immediately before a red phase**. This is not an invariant.

Safety Properties (cont.)

Definition 3.29 (Safety property)

A LT property M_{safe} is called a safety property, if for all words $\lambda \in (2^{\mathcal{P}rop})^\omega \setminus M_{\text{safe}}$ there exists a finite prefix (a bad prefix) $\hat{\lambda}$ of λ such that

$$M_{\text{safe}} \cap \{\lambda' : \lambda' \in (2^{\mathcal{P}rop})^\omega, \hat{\lambda} \text{ is a finite prefix of } \lambda'\} = \emptyset$$

So it is a **condition on a finite initial fragment**: no extended word resulting from such a bad prefix is allowed.

Liveness Properties

Definition 3.30 (Liveness property)

A LT property M is a liveness property, if the set of finite prefixes of the elements of M is identical to $(2^{\mathcal{P}rop})^*$. I.e. each finite prefix can be extended to an infinite word that satisfies the property.

- Each process will **eventually** enter its critical section.
- Each process will enter its critical section **infinitely** often.
- Each waiting process will **eventually** enter its critical section.

Liveness Properties (cont.)

Starvation freedom in the dining philosophers is a typical example: each philosopher is getting her sticks infinitely often.

Starvation freedom: For all timepoints i, if there is a waiting process at time i, then the process gets into its critical section eventually.

Safety versus Liveness

- Are safety properties also liveness properties? Vice versa?
- There is only one property that is both: $(2^{\mathcal{P}rop})^{\omega}$, i.e. the trivial property that contains all paths.

Theorem 3.31 (LT properties as intersections)

Each LT-property can be represented as the intersection of a safety with a liveness property.

But there are LT properties that are neither safe nor live.

Fairness Properties

Definition 3.32 (Fairness property)

A LT property is a fairness property, if one of the following applies:

Each process gets its turn infinitely often

provided that

unconditional: (no restrictions)

strong: it is enabled infinitely often,

weak: it is continuously enabled from a certain time on.

LT properties used in practice

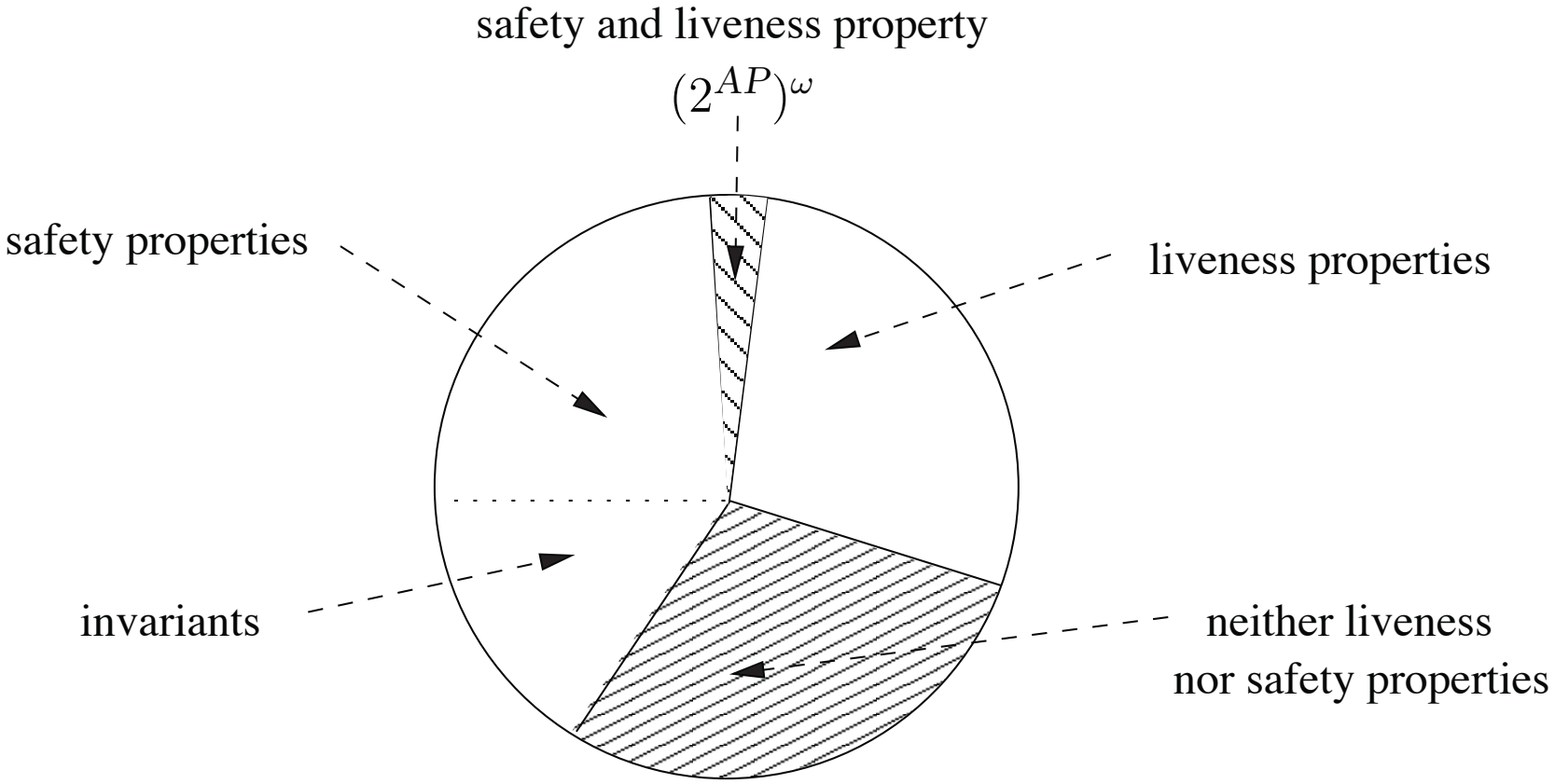


Figure 3.31: Overview LT-properties

3.6 LTL

Typical temporal operators

$\mathbf{X}\varphi$	φ is true in the neXt moment in time
$\mathbf{G}\varphi$	φ is true Globally: in all future moments
$\mathbf{F}\varphi$	φ is true Finally: eventually (in the future)
$\varphi \mathcal{U} \psi$	φ is true Until at least the moment when ψ becomes true (and this eventually happens)

$$\mathbf{G}((\neg \text{passport} \vee \neg \text{ticket}) \rightarrow \mathbf{X}\neg \text{board\_flight})$$

$$\text{send}(\text{msg}, \text{rcvr}) \rightarrow \mathbf{F}\text{receive}(\text{msg}, \text{rcvr})$$

Safety: **Something bad will not happen, something good will always hold.**

$\mathbf{G}\neg$bankrupt,

$\mathbf{G}$fuelOK,

Usually: $\mathbf{G}\neg\ldots$.

Liveness: **Something good will happen.**

$\mathbf{F}$rich,

power_on $\rightarrow \mathbf{F}$online,

Usually: $\mathbf{F}\ldots$.

Fairness: Combinations of safety and liveness:

$\mathbf{FG}\neg$dead or

$\mathbf{G}(\text{request\_taxi} \rightarrow \mathbf{F}\text{arrive\_taxi})$.

Strong fairness: "If something is requested then it will be allocated":

$$\mathbf{G}(\text{attempt} \rightarrow \mathbf{F}\text{success}),$$

$$\mathbf{GF}\text{attempt} \rightarrow \mathbf{GF}\text{success}.$$

Scheduling processes, responding to messages, no process is blocked forever, etc.

Definition 3.33 (Language $\mathcal{L}_{LTL}$ [Pnueli 1977])

The language $\mathcal{L}_{LTL}(\mathcal{P}rop)$ is given by all formulae generated by the following grammar, where $p \in \mathcal{P}rop$ is a proposition:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U} \varphi \mid \mathbf{X}\varphi.$$

The additional operators

- **F** (eventually in the future) and
- **G** (always from now on)

can be defined as macros :

$$\mathbf{F}\varphi \equiv (\neg\Box)\mathcal{U}\,\varphi \qquad \text{and} \qquad \mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$$

The standard Boolean connectives $\top, \wedge, \rightarrow$, and $\leftrightarrow$ are defined in their usual way as **macros** (see Definition 2.3 on Slide 63).

Models of $\mathcal{L}_{LTL}$

The semantics is given over paths, which are **infinite sequences of states** from S, and a standard labelling function $\pi : \mathrm{S} \to \mathcal{P}(\mathcal{P}rop)$ that determines which propositions are true at which states.

Definition 3.34 (Path $\lambda = q_0 q_1 q_2 q_3 \ldots$)

- A path λ over a set of states S is an infinite sequence of states. We can view it as a mapping $\mathbb{N}_0 \to \mathrm{S}$. The set of all sequences is denoted by S^ω.
- $\lambda[i]$ denotes the ith position on path λ (starting from $i = 0$) and
- $\lambda[i, \infty]$ denotes the subpath of λ starting from i ($\lambda[i, \infty] = \lambda[i]\lambda[i+1]\ldots$).

$$\lambda = q_0 q_1 q_2 q_3 \ldots \in \mathrm{S}^\omega$$

Definition 3.35 (Semantics of $\mathcal{L}_{LTL}$)

Let λ be a path and π be a labelling function over S. The semantics of **LTL**, $\models^{\mathbf{LTL}}$, is defined as follows:

- $\lambda, \pi \models^{\mathbf{LTL}} \mathrm{p}$ *if, by definition,* $\mathrm{p} \in \pi(\lambda[0])$ and $\mathrm{p} \in \mathcal{P}rop$;
- $\lambda, \pi \models^{\mathbf{LTL}} \neg\varphi$ *if, by definition, not* $\lambda, \pi \models^{\mathbf{LTL}} \varphi$ (we write $\lambda, \pi \not\models^{\mathbf{LTL}} \varphi$);
- $\lambda, \pi \models^{\mathbf{LTL}} \varphi \vee \psi$ *if, by definition,* $\lambda, \pi \models^{\mathbf{LTL}} \varphi$ **or** $\lambda, \pi \models^{\mathbf{LTL}} \psi$;
- $\lambda, \pi \models^{\mathbf{LTL}} \mathbf{X}\varphi$ *if, by definition,* $\lambda[1, \infty], \pi \models^{\mathbf{LTL}} \varphi$; and
- $\lambda, \pi \models^{\mathbf{LTL}} \varphi \mathcal{U} \psi$ *if, by definition, there is an* $i \in \mathbb{N}_0$ such that $\lambda[i, \infty], \pi \models \psi$ and $\lambda[j, \infty], \pi \models^{\mathbf{LTL}} \varphi$ for all $0 \leq j < i$.

Other temporal operators

$\lambda, \pi \models \mathbf{F}\varphi$ *if, by definition,* $\lambda[i, \infty], \pi \models \varphi$ for some $i \in \mathbb{N}_0$;
$\lambda, \pi \models \mathbf{G}\varphi$ *if, by definition,* $\lambda[i, \infty], \pi \models \varphi$ for all $i \in \mathbb{N}_0$;

Exercise

Prove that the semantics does indeed match the definitions;

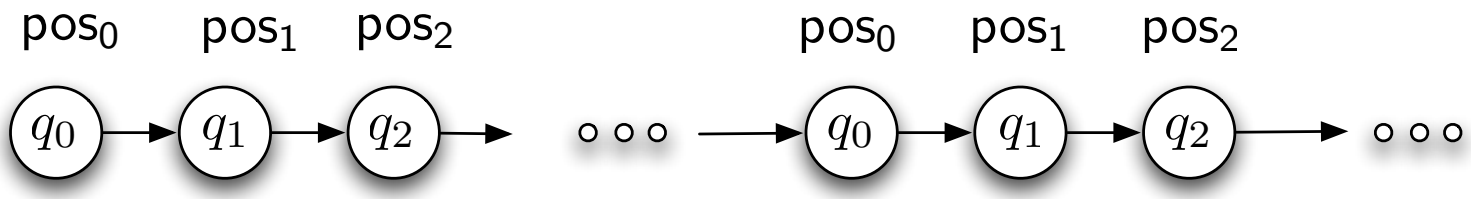
- $\mathbf{F}\varphi$ is equivalent to $(\neg\Box)\,\mathcal{U}\,\varphi$, and
- $\mathbf{G}\varphi$ is equivalent to $\neg\mathbf{F}\neg\varphi$.

Validity, satisfiability

satisfiable: a LTL formula is satisfiable, *if, by definition,* there is a model for it,

valid: a LTL formula is valid, *if, by definition,* it is true in all models,

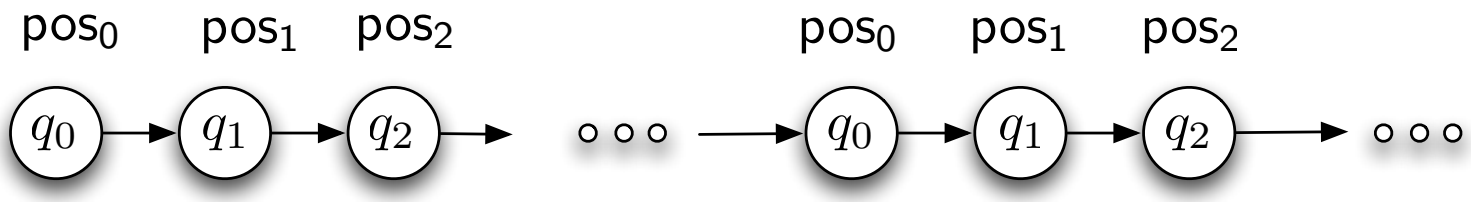
contradictory: a LTL formula is contradictory, *if, by definition,* there is no model for it.



$$\lambda, \pi \models \mathbf{F}\mathsf{pos}_1$$

$$\lambda' = \lambda[1, \infty], \pi \models \mathsf{pos}_1$$

$$\mathsf{pos}_1 \in \pi(\lambda'[0])$$



$\lambda, \pi \models \mathbf{GF}\mathsf{pos}_1$ *if and only if*

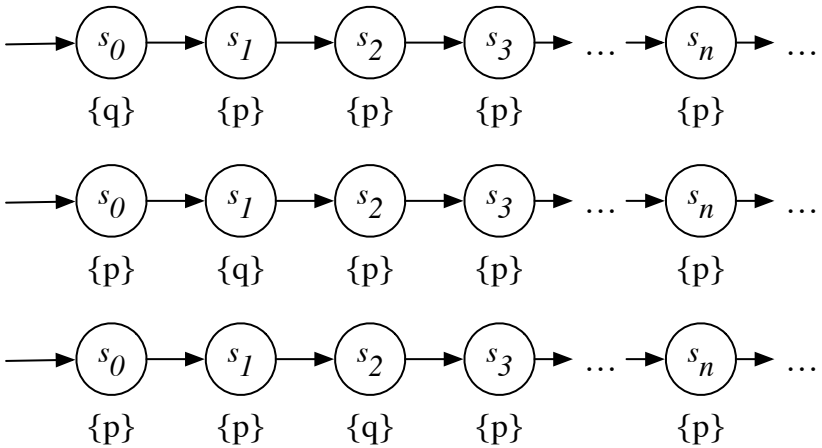
$\lambda[0, \infty], \pi \models \mathbf{F}\mathsf{pos}_1$ and
$\lambda[1, \infty], \pi \models \mathbf{F}\mathsf{pos}_1$ and
$\lambda[2, \infty], \pi \models \mathbf{F}\mathsf{pos}_1$ and
$\ldots$

Paths and infinite sets of paths

- Paths are infinite entities, and so are infinite sets of them.
- They are both theoretical constructs.
- In order to work with them we need a finite representation:
- namely transition systems (also called pointed Kripke structures).

A set of paths (1)

We reconsider the paths $\lambda_0, \lambda_1, \ldots, \lambda_i, \ldots$ from Example 3.26 on Slide 237:



- $q \wedge \neg p \wedge \mathbf{XG}(p \wedge \neg q)$
- $p \wedge \mathbf{X}q \wedge \mathbf{XXG}p$
- $p \wedge \mathbf{X}p \wedge \mathbf{XX}q$

etc.

Can we distinguish between them (using LTL)?

Indistinguishable paths

Observation

While any two paths can be distinguished by appropriate LTL formulae, these formulae get more and **more complicated**: operators need to be nested.

- The first two paths can be distinguished by just propositional logic, no LTL connectives are needed.
- But the second and third cannot: we need **X**.
- For the third and fourth we need a nesting **XX**.

By induction over the structure of φ

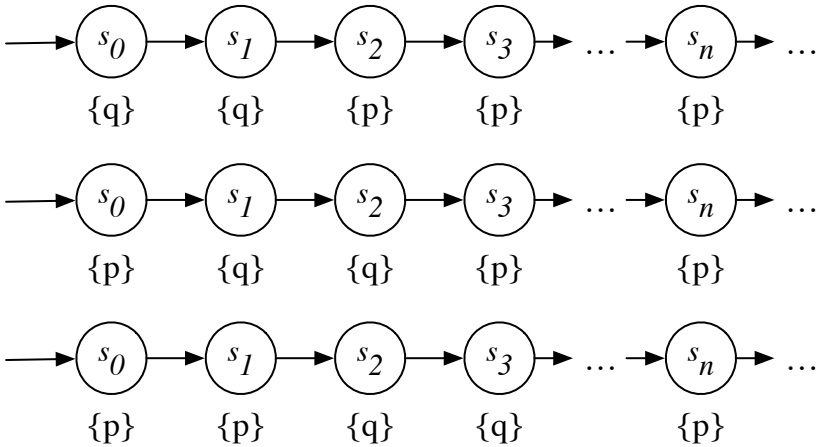
Given any LTL formula φ, there is a $i_0 \in \mathbb{N}$ such that for all $i, j \geq i_0$: $\lambda_i \models \varphi$ if and only if $\lambda_j \models \varphi$.

A set of paths (2)

- Can we find a LTL formula, or a set of LTL formulae, that **characterize exactly the whole set of paths**?
- So what holds true in all of these paths?
- $p \mathcal{U} q$. But this is also true in other paths not listed above.
- $(p \wedge \neg q) \mathcal{U} (q \wedge \neg p)$. Again, this is also true in other paths.
- $(p \wedge \neg q) \mathcal{U} (q \wedge \neg p \wedge \mathbf{XG}(p \wedge \neg q))$. That is it. This describes exactly the set of paths above.

Another set of paths (1)

We reconsider the second set of paths from Example 3.26 on Slide 237:



- $q \wedge \mathbf{X}q \wedge \mathbf{XXG}(p \wedge \neg q)$
- $p \wedge \mathbf{X}q \wedge \mathbf{XX}q \wedge \mathbf{XXXG}p$
- etc.

etc.

What holds true in exactly these paths?

$$(p \wedge \neg q)\,\mathcal{U}\,(q \wedge \neg p \wedge \mathbf{X}(q \wedge \neg p) \wedge \mathbf{XXG}(p \wedge \neg q))$$

LTL formulae and transition systems: $TS \models \phi$

- Up to now we have defined LTL formulae **only for paths**.
- For a transition system *TS*, we say that an LTL formula is true in a state s, if it is true in all runs resulting from that state.
- A LTL formula is true in the whole transition system, if it is true in all runs resulting from the initial states.

LTL formulae and transition systems: $TS \models \varphi$

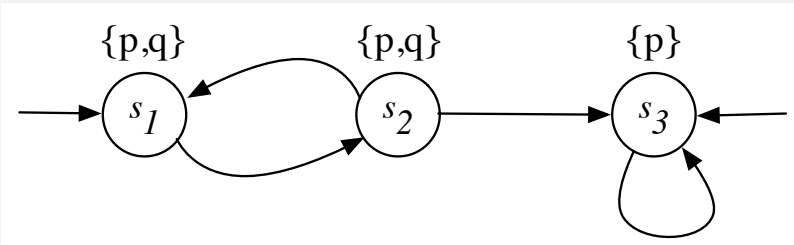
Definition 3.36 (*TS* and LTL formulae: $TS \models \phi$)

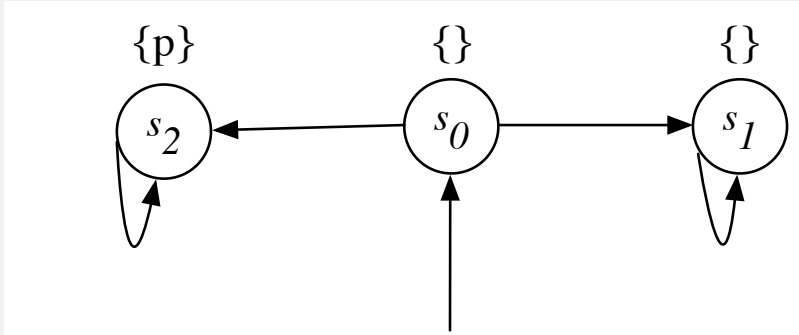
Let a *TS* and a LTL formula φ be given.

- **A LTL formula φ is true in a state s of *TS*, *if, by definition,* it is true in all runs resulting from s.**
- *A LTL formula φ is true in TS, if, by definition, it is true in all runs resulting from all initial states.*

LTL formulae and transition systems

Example 3.37 (LTL formulae)





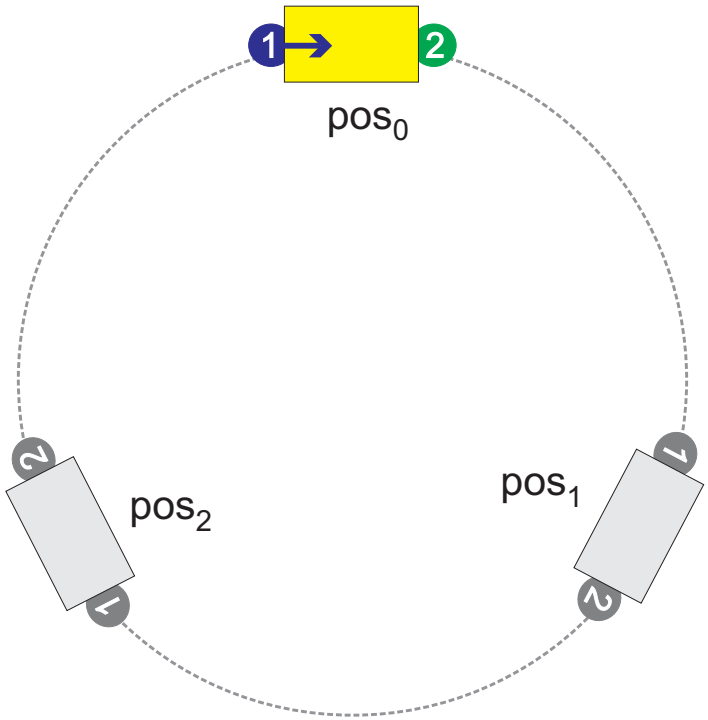
Which formulae hold true?

TS_1: $TS_1 \models \mathbf{G}p$, $TS_1 \not\models \mathbf{X}(p \wedge q)$,
$TS_1 \not\models \mathbf{G}(q \rightarrow \mathbf{G}(p \wedge q))$,
but $TS_1 \models \mathbf{G}(q \rightarrow (p \wedge q)))$

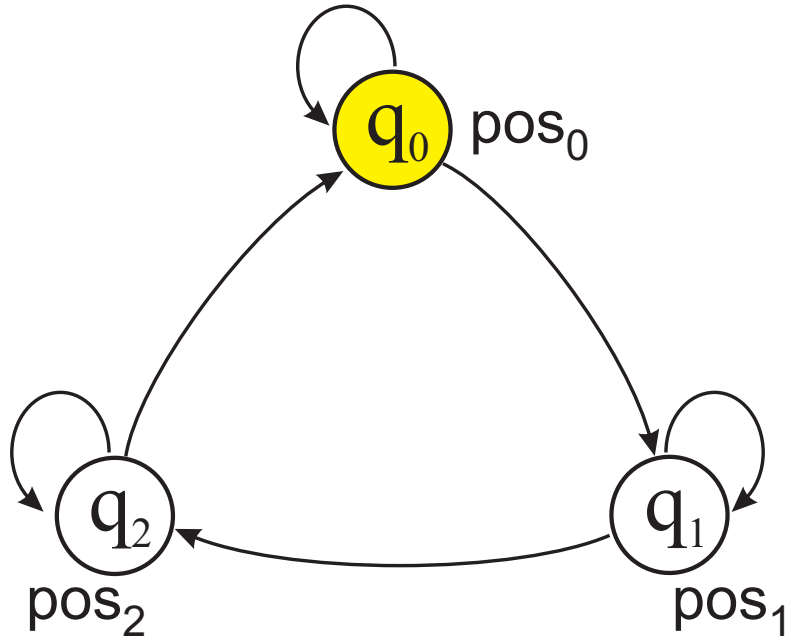
TS_2: $TS_2 \not\models \mathbf{F}p$ and $TS_2 \not\models \neg\mathbf{F}p$

From system to behavioral structure

System

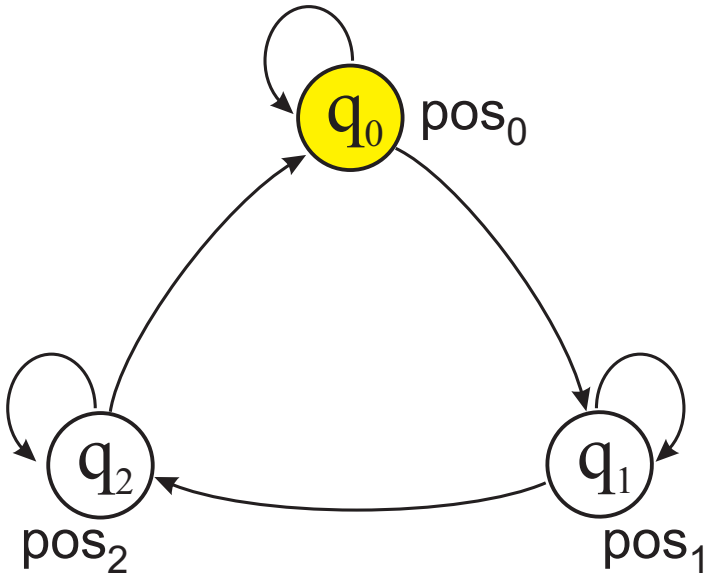


Computational str.

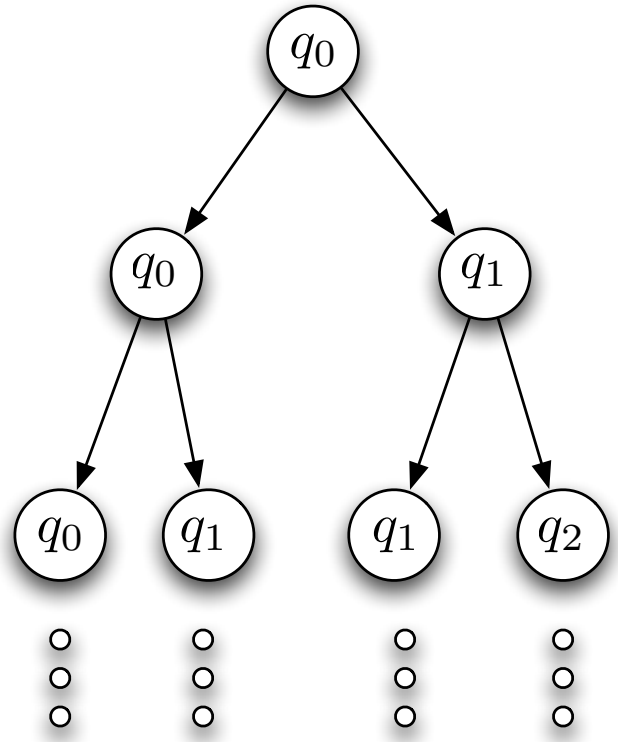


From computational to behavioral structure

Computational str.



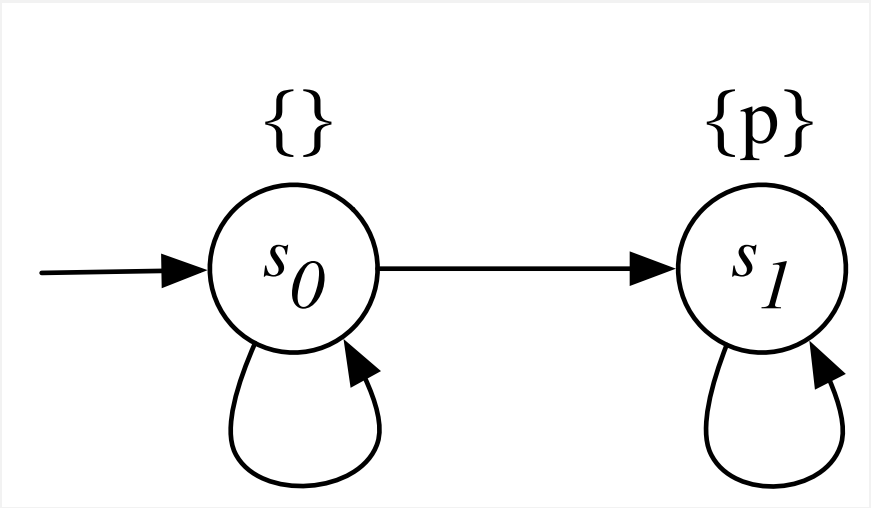
Behavioral str.



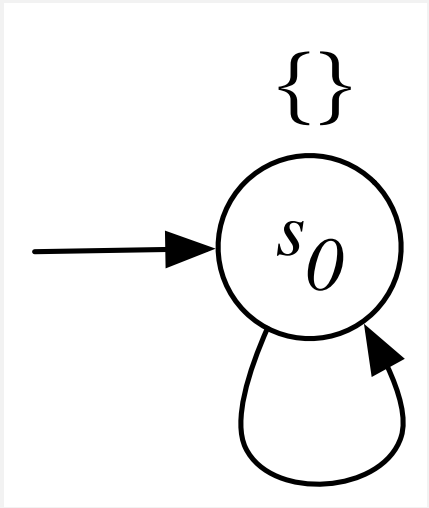
The behavioral structure is usually infinite! Here, it is an infinite tree. We say it is the q_0-unfolding of the model.

Example 3.38 (LTL indistinguishable transition systems)

TS_1:



TS_2:



Both systems can be distinguished by the property **"a state where p holds can be reached"**.

- Each trace of TS_2 is also one of TS_1: **each LTL formula true in TS_1 is also true in TS_2.**
- So the above property cannot be expressed in LTL.

LT-Properties and their expressibility

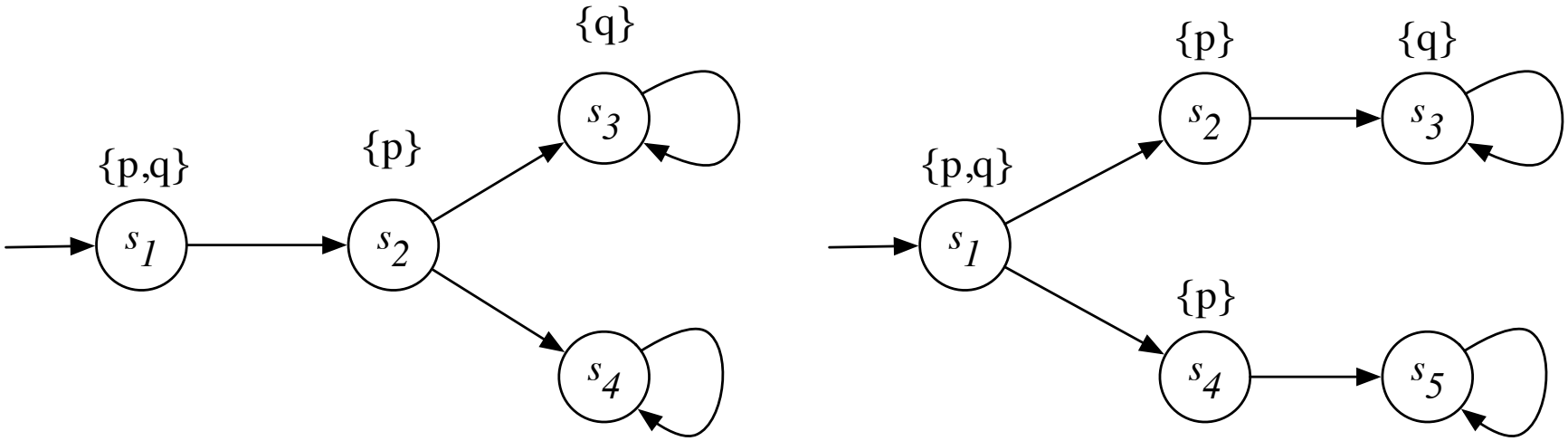
- The example on the preceding slide shows that a certain property is not a LT property.
- **This does not mean that the two transition systems cannot be distinguished.**
- In fact the formula $\mathbf{X}\neg p$ is true in TS_2 but not in TS_1.
- The traces of the two systems are also different, so both define different LT properties.

Are there TS_1 and TS_2 that **define different LT-properties but cannot be distinguished by any set of $\mathcal{L}_{LTL}$ formulae**?

Yes. $\mathcal{L}_{LTL}$ formulae express exactly "$*$-free ω-regular properties", a strict subset of "ω-regular properties", which is itself a strict subset of LT-properties.

LT-Properties and their expressibility (cont.)

We consider again Example 3.23 with the two transition systems TS_1 and TS_2.



1. The property **whenever p a state with q can be reached** distinguishes them.
2. Can this property be expressed in LTL?
3. No, because both have the same set of traces.

Some Exercises

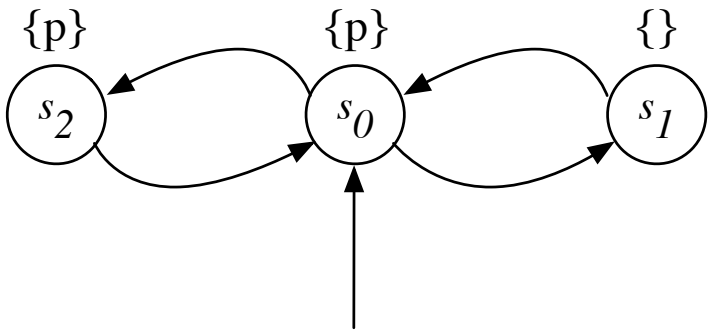
Example 3.39 (Formalizing properties in LTL)

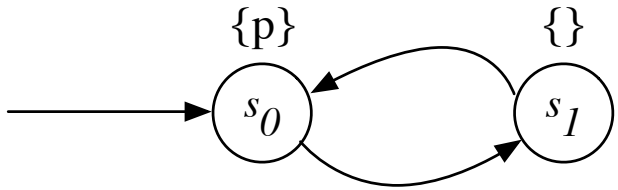
Formalise the following properties as **LTL** formulae over $\mathcal{P}rop = \{\mathsf{p}\}$

1. p should never occur.
2. p should occur exactly once.
3. At least once p is directly be followed by $\neg$p.
4. p is true at exactly all even states.

Some Exercises (cont.)

Compare the following two transition systems:





Example 3.40 (Evenness)

Formalise the following as a **LTL** formula: p is true at **all even** states (the odd states do not matter).

Does $p \wedge \mathbf{G}(p \rightarrow \mathbf{XX}p)$ work?

Satisfiability of LTL formulae

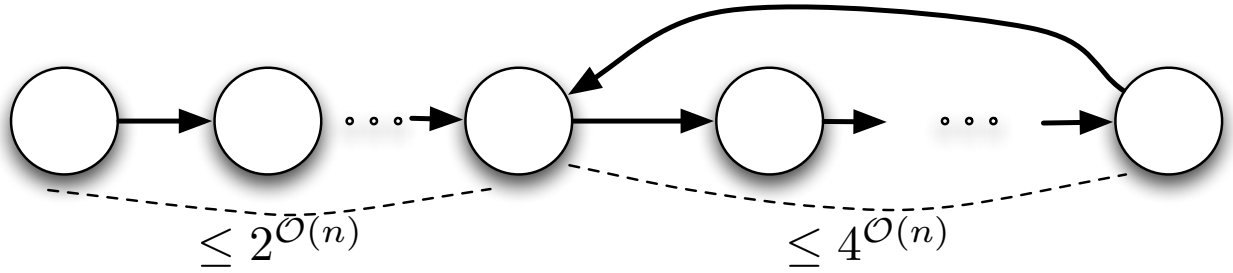
A formula is satisfiable, if there is a model (i.e. path) where it holds true. Can we **restrict the structure** of such paths? I.e. can we restrict to simple paths, for example paths that are periodic?

- If this is the case, then we might be able to **construct counterexamples** more easily, as we need only check very specific paths.
- It would be also useful to know **how long the period is** and **within which initial segment** of the path it starts, depending on the length of the formula φ.

Satisfiability of LTL formulae (cont.)

Theorem 3.41 (Periodic model theorem [SistlaClarke 1985])

*A formula $\varphi \in$ **LTL** is satisfiable if and only if there is a path λ which is ultimately periodic, and the period starts within $2^{1+|\varphi|}$ steps and has a length which is $\leq 4^{1+|\varphi|}$.*



4. First-Order Logic: Semantics

Content of this chapter:

FOL: While sentential logic SL has some nice features, it is quite restricted in expressivity. First order logic (FOL) is an extension of SL which allows us to express much more in a succinct way.

LTL: We show that LTL, introduced in the previous chapter, is but a subsystem of FOL: the monadic theory of order (Kamp's theorem).

SIT-calculus: The Situation Calculus, introduced by John McCarthy, is a special method for using FOL to express the dynamics of an agent. Applying actions to the world leads to a series of successor worlds that can be represented with special FOLterms.

Content of this chapter (2):

Blocksworld: We consider the blocksworld scenario and discuss how to formalize that with FOL.

HOL: Finally, we give an outlook to higher order logics (HOL), in particular to second order logic. While we can express much more than in 1st order logic, the price we have to pay is that there is no correct and complete calculus.

Declarative: In this chapter we only consider the question How to use FOL to model the world? We are not concerned with **deriving new information** or with implementing FOL. This will be done in the next chapter.

4.1 Motivation

Example 4.1 (Continuity of a function)

For the classical definition of the **continuity** of a function, we need:

- Variables: $\varepsilon, \delta, x, x_0$,
- functions: $f(\cdot)$, $|\ \cdot\ |$,
- relations (or predicates): $<, >, =$ and
- quantifiers: $\forall, \exists$.

$$\forall \varepsilon > 0\ \exists \delta > 0\ \big(\forall x\ (\,|\,x - x_0\,| < \delta) \rightarrow |\,f(x) - f(x_0)\,| < \epsilon\big)$$

So we need terms that we can compare with each other and functions that represent values (that can be compared with the values of terms).

Terms and predicates

- In SL we only have **sentential constants** which can be true or false in a model.
- In FOL we introduce terms, which can be built up inductively. They represent objects, a category which is not available in SL.
- E.g. given an object tom the term $father\_of(tom)$ represents another object (intuitively Tom's father).
- These objects can be used in FOL predicates: Is_Father_of$(erich, tom)$. The binary predicate Is_Father_of$(\cdot, \cdot)$ represents a relation between two objects.

Terms and predicates (cont.)

- Terms are built recursively out of constants (not to be confused with propositional constants) using function symbols.
- We also allow a new category which is not available in SL: variables. They are denoted by $x, y, z, \ldots$.
- Variables allow us to formulate conditions that represent interesting entities. They can be instantiated with terms.
- Constants can be $tom, erich, tobias$, function symbols $sister\_of(\cdot), father\_of(\cdot)$. They lead to complex terms like

$$father\_of(sister\_of(father\_of(tom)))$$

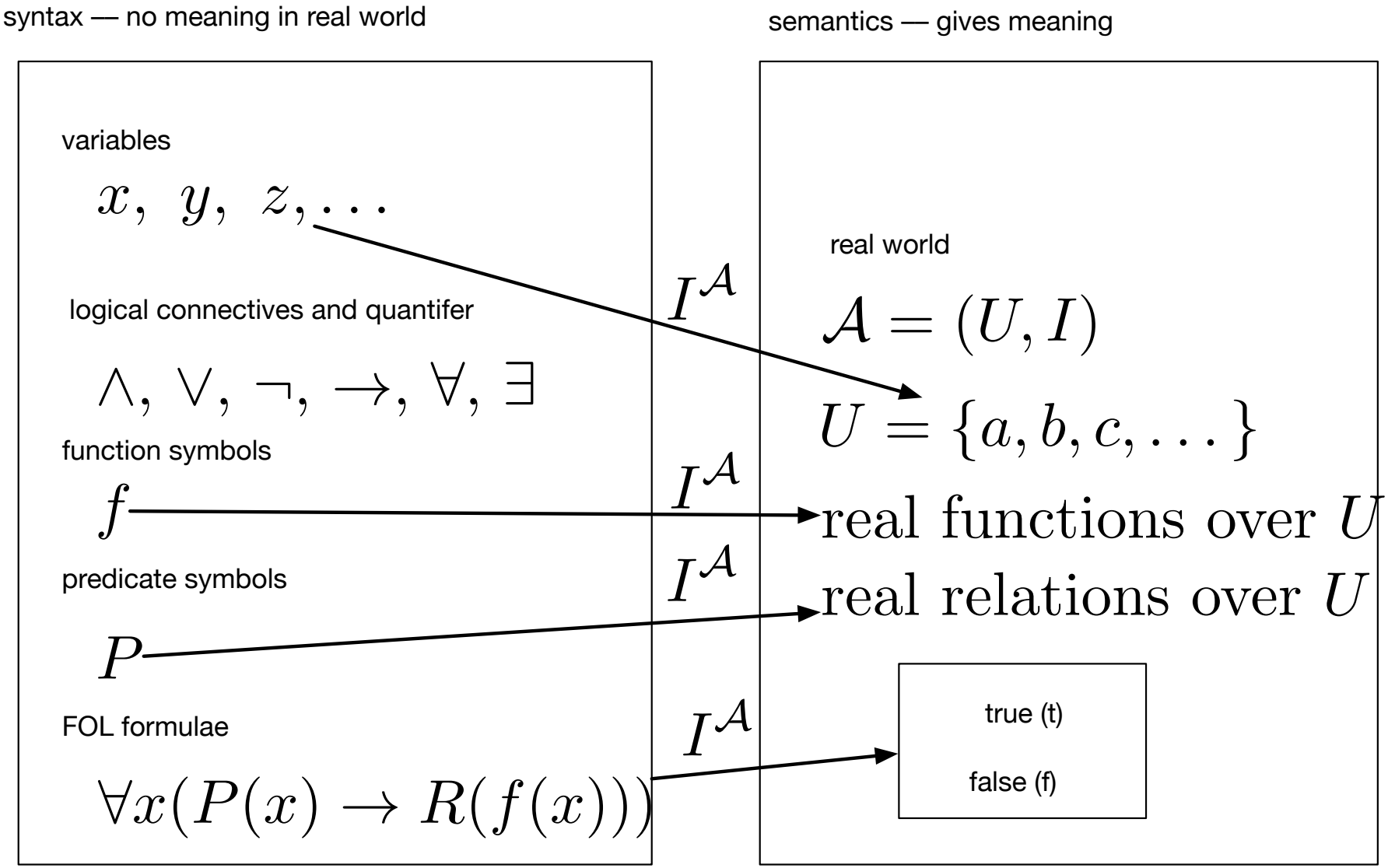
- These objects can be used in FOL predicates: is_Father_of$(erich, tom)$.

Tuna revisited

Our Example 2.15 on Slide 84 can be defined in FOL in a much more intuitive way. There are objects like cats, dogs, animals, humans etc.

- $tuna$ is such an object, as well as $jack$ or $bill$.
- While $tuna$ belongs to the class of cats (defined by a predicate cat), $jack$ is a human, or an animal_lover.
- That animal lovers do not kill animals, can now by expressed more intuitively as just one sentence:
  $\forall x \left(\text{animal\_lover}(x) \rightarrow (\forall y \left(\text{animal}(y)\right) \rightarrow \neg\text{killer\_of}(x, y)\right).$
- **In SL we needed many such implications (for all potential objects).**

Overview


syntax — no meaning in real world
semantics — gives meaning
variables
$x,\ y,\ z, \ldots$
logical connectives and quantifer
$\wedge, \vee, \neg, \rightarrow, \forall, \exists$
function symbols
f
predicate symbols
P
FOL formulae
$\forall x(P(x) \rightarrow R(f(x)))$
$I^{\mathcal{A}}$
$I^{\mathcal{A}}$
$I^{\mathcal{A}}$
$I^{\mathcal{A}}$
real world
$\mathcal{A} = (U, I)$
$U = \{a, b, c, \ldots\}$
real functions over U
real relations over U
true (t)
false (f)

4.2 FOL: Syntax

Definition 4.2 (First order logic $\mathcal{L}(\Sigma)$)

The language $\mathcal{L}(\Sigma)$ of first order logic (*Prädikatenlogik erster Stufe*) consists of:

- $x, y, z, x_1, x_2, \ldots, x_n, \ldots$: a countable set $\mathcal{V}ar$ of variables,
- for each $k \in \mathbb{N}_0$: $P^k_1, P^k_2, \ldots, P^k_n, \ldots$ a countable set $\mathcal{P}red^k$ of k-dimensional predicate symbols (the 0-dimensional predicate symbols consist of the propositional logic constants from $\mathcal{L}_{SL}$, i.e. $\Box$, $\mathcal{P}rop$).
- for each $k \in \mathbb{N}_0$: $f^k_1, f^k_2, \ldots, f^k_n, \ldots$ a countable set $\mathcal{F}unc^k$ of k-dimensional function symbols. The 0-dimensional function symbols are also called individuum constants.
- $\neg$, $\vee$: the sentential connectives,
- $($, $)$: the parentheses, and
- $\exists$: the existential quantifier.

Definition (continued)

The used set of predicate and function symbols and $X \subseteq \mathcal{V}ar$ is also called **signature** Σ.
The signature defines a language $\mathcal{L}(\Sigma)$ (analogously to Definition 2.2 on Slide 62). When clear from context, we omit Σ and use $\mathcal{L}$ instead of $\mathcal{L}(\Sigma)$.

The signature Σ plays in FOL the same role as $\mathcal{P}rop$ in SL: it determines the **set of formulae** $\mathsf{Fml}^{\mathsf{FOL}}_{\mathcal{L}(\Sigma)}$.

Definition (continued)

The concepts of an $\mathcal{L}(\Sigma)$-**term** t and an $\mathcal{L}(\Sigma)$-**formula** φ are defined inductively:

Term: $\mathcal{L}(\Sigma)$-terms t are defined as follows:

1. each variable is a $\mathcal{L}(\Sigma)$-term,
2. if f^k is a k-dimensional function symbol from Σ and $t_1, \ldots, t_k$ are $\mathcal{L}(\Sigma)$-**terms**, then $f^k(t_1, \ldots, t_k)$ is a $\mathcal{L}(\Sigma)$-**term** (for $k = 0$ we write f^0, not $f^0()$; in fact, we use symbols $c, d, \ldots$).

The set of all $\mathcal{L}(\Sigma)$-terms over a set $X \subseteq Var$ is called $Term_{\mathcal{L}(\Sigma)}(X)$ or $Term_{\mathcal{L}}(X)$. Using $X = \emptyset$ we get the set of basic terms $Term_{\mathcal{L}(\Sigma)}(\emptyset)$, or just $\mathbf{Term}_{\mathcal{L}}$.

Definition (continued)

Formulae: φ is defined inductively:

1. if P^k is a k-dimensional predicate symbol from Σ and $t_1, \ldots, t_k$ are $\mathcal{L}(\Sigma)$-terms then $\mathsf{P}^k(t_1, \ldots, t_k)$ is a $\mathcal{L}(\Sigma)$-formula (for $k = 0$ we get back the propositional constants, which we also write without parentheses).
2. For all $\mathcal{L}(\Sigma)$-formulae φ: $(\neg\varphi)$ is a $\mathcal{L}$-formula.
3. For all $\mathcal{L}(\Sigma)$-formulae φ and ψ: $(\varphi \vee \psi)$ is a $\mathcal{L}(\Sigma)$-formula.
4. If x is a variable and φ a $\mathcal{L}(\Sigma)$-formula then $(\exists\, x\, \varphi)$ is a $\mathcal{L}(\Sigma)$-formula.

Definition (continued)

Atomic $\mathcal{L}(\Sigma)$-formulae are those which are composed according to (1), we call them $\text{Atomic}_{\mathcal{L}(\Sigma)}$. The set of all $\mathcal{L}(\Sigma)$-formulae over X is called $\text{Fml}^{\text{FOL}}_{\mathcal{L}(\Sigma)}$.

Positive formulae ($\text{Fml}^{\text{FOL}+}_{\mathcal{L}(\Sigma)}$) are those which are composed using only (1), (3) and (4).

If φ is a $\mathcal{L}$-formula and is part of another $\mathcal{L}$-formula ψ then φ is called **sub-formula** of ψ.

As in SL (where we assumed $\mathcal{P}rop$ to be fixed), we assume that the signature Σ (i.e. the set of function and predicate symbols) is fixed: thus the language $\mathcal{L}(\Sigma)$ is fixed and we omit it if clear from context (like parentheses).

Other connectives are defined as macros, as in SL.

Definition 4.3 (FOL connectives as macros)

We define the following syntactic constructs as macros:

$$\begin{aligned}
\top &=_{def} (\neg\Box) \\
\varphi \wedge \psi &=_{def} (\neg((\neg\varphi) \vee (\neg\psi))) \\
\varphi \to \psi &=_{def} (((\neg\varphi) \vee \psi) \\
\varphi \leftrightarrow \psi &=_{def} ((\varphi \to \psi) \wedge (\psi \to \varphi)) \\
\forall x\varphi &=_{def} (\neg(\exists x(\neg\varphi)))
\end{aligned}$$

$\rightsquigarrow$ **blackboard**

Example 4.4

Let d be a constant, $g \in \mathcal{F}unc^2$, and $f \in \mathcal{F}unc^3$, and $\mathsf{p} \in \mathcal{P}red^2$ and $\mathsf{q} \in \mathcal{P}red^0$.
Which of the following strings are **terms**, which are **formulae**?

1. $g(d, d)$, $g(x, \mathsf{p}(x, y))$,
2. $\mathsf{p}(d)$, $f(x, g(y, z), d)$,
3. $g(x, f(y, z), d)$, $f(x, g(y, z), d)$, $\forall z \mathsf{q}$,
4. $g(f(g(d, x), g(f(x, a, z), y), f(x, y, g(x, y))), a)$,
5. $d \wedge f(x)$, $\forall z \neg f(x, y)$,
6. $\mathsf{p}(\mathsf{p}(d, d), x)$, $\neg \mathsf{p}(d, g(d, x))$.

Example 4.5 (From semigroups to rings)

We consider $\mathcal{L} = \{0, 1, +, \cdot, \leq, \doteq\}$, where $0, 1$ are **constants** ($\in \mathcal{F}unc^0$), $+, \cdot$ binary **operations** ($\in \mathcal{F}unc^2$) and $\leq, \doteq$ binary **relations** ($\in \mathcal{P}red^2$) . What can be expressed in this language?

Ax 1:	$\forall x \forall y \forall z$	$x + (y + z) \doteq (x + y) + z$
Ax 2:	$\forall x$	$(x + 0 \doteq 0 + x) \ \wedge \ (0 + x \doteq x)$
Ax 3:	$\forall x \exists y$	$(x + y \doteq 0) \ \wedge \ (y + x \doteq 0)$
Ax 4:	$\forall x \forall y$	$x + y \doteq y + x$
Ax 5:	$\forall x \forall y \forall z$	$x \cdot (y \cdot z) \doteq (x \cdot y) \cdot z$
Ax 6:	$\forall x \forall y \forall z$	$x \cdot (y + z) \doteq x \cdot y + x \cdot z$
Ax 7:	$\forall x \forall y \forall z$	$(y + z) \cdot x \doteq y \cdot x + z \cdot x$

Axiom 1 describes a *semigroup*, the axioms 1-2 describe a *monoid*, the axioms 1-3 a *group*, and the axioms 1-7 a *ring*.

Example 4.6

Formalize the following sentence:

> *Every student x is younger than some teacher y.*

- $\mathsf{s}(x)$: x is a student
- $\mathsf{t}(x)$: x is a teacher
- $\mathsf{y}(x, y)$: x is younger than y

$$\forall x(\mathsf{s}(x) \rightarrow (\exists y(\mathsf{t}(y) \wedge \mathsf{y}(x, y))))$$

Example 4.7

Formalize the following sentence:

> *Andy and Paul have the same maternal grandmother.*

- a: Andy
- p: Paul
- $\mathsf{m}(x, y)$: x is y's mother

$$\forall x \forall y \forall u \forall v (\mathsf{m}(x, y) \land \mathsf{m}(y, a) \land \mathsf{m}(u, v) \land \mathsf{m}(v, p) \rightarrow x \doteq u)$$

Unary function or binary predicate?

Given a unary function symbol $f(x)$ and a constant c. Then the only terms that can be built are of the form $f^n(c)$. Assume we have $\doteq$ as the **only predicate**. Then the atomic formulae that we can built have the form $f^n(c) \doteq f^m(c)$. We call this language $\mathcal{L}_1$.

Unary function or binary predicate? (2)

Assume now that instead of f and $\doteq$, we use a binary predicate $p_f(y, x)$ which formalizes $y \doteq f(x)$. We call this language $\mathcal{L}_2$.

- Can we **express** all formulae of $\mathcal{L}_1$ in $\mathcal{L}_2$?
- And vice versa?
- What is the difference between both approaches?

FOL with equality $\doteq$?

Often we want to use real equality (identity): a binary relation that is true only if the objects are identical (not just in a equivalence relation).

This means that we assume to have a distinguished symbol $\doteq$ among the binary predicates in $\mathcal{P}red^2$ which is interpreted as identity in all models, i.e. its interpretation is fixed.

Tuna the cat in FOL

We formalize again the example of the killed cat (Example 2.15), this time in FOL.

1. Bill owns a dog.
   $\rightsquigarrow$ $\mathsf{owns}(x, y), \mathsf{dog}(x), bill.$
2. Dog owners are animal lovers.
   $\rightsquigarrow$ $\mathsf{animal\_lover}(x).$
3. Animal lovers do not kill animals.
   $\rightsquigarrow$ $\mathsf{killer\_of}(x, y), \mathsf{animal}(x).$
4. Either Jack or Bill killed Tuna, the cat.
   $\rightsquigarrow$ $jack, tuna, \ \mathsf{cat}(x), \mathsf{killer\_of}(x, y).$

$\rightsquigarrow$ **blackboard**

Tuna the cat in FOL (2)

1. Bill owns a dog.
   $\exists y\,(\text{owns}(bill, y) \wedge \text{dog}(y))$.
2. Dog owners are animal lovers.
   $\forall x\,((\text{owns}(x, y) \wedge \text{dog}(y)) \rightarrow \text{animal\_lover}(x))$.
3. Animal lovers do not kill animals.
   $\forall x \forall y\,(\,(\text{animal\_lover}(x) \wedge \text{animal}(y)) \rightarrow \neg\text{killer\_of}(x, y)\,)$.
4. Either Jack or Bill killed Tuna, the cat.
   $\text{cat}(tuna)$,
   $(\text{killer\_of}(jack, tuna) \vee \text{killer\_of}(bill, tuna)$,
   $\neg(\text{killer\_of}(jack, tuna) \wedge \text{killer\_of}(bill, tuna)))$.

4.3 FOL: Models

Definition 4.8 ($\mathcal{L}$-structure $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$)

A $\mathcal{L}(\Sigma)$-structure or a $\mathcal{L}(\Sigma)$-interpretation over the signature Σ is a pair $\mathcal{A} =_{def} (U_\mathcal{A}, I_\mathcal{A})$ with $U_\mathcal{A}$ being an arbitrary non-empty set, which is called the **basic set** (the **universe** or the **individuum range**) of $\mathcal{A}$. Further $I_\mathcal{A}$ is a mapping which

- assigns to each k-dimensional predicate symbol p^k in $\mathcal{L}(\Sigma)$ a k-dimensional predicate over $U_\mathcal{A}$ ($\square$ is assigned the empty set),
- assigns to each k-dimensional function symbol f^k in $\mathcal{L}(\Sigma)$ a k-dimensional function on $U_\mathcal{A}$.

In other words: the domain of $I_\mathcal{A}$ is exactly the set of predicate and function symbols of $\mathcal{L}(\Sigma)$.

Where is the valuation v from SL hidden?

Definition (continued)

The range of $I_{\mathcal{A}}$ consists of the predicates and functions on $U_{\mathcal{A}}$. We write:

$$I_{\mathcal{A}}(\mathsf{p}) = \mathsf{p}^{\mathcal{A}},\ I_{\mathcal{A}}(f) = f^{\mathcal{A}}.$$

Let φ be a $\mathcal{L}_1$-formula and $\mathcal{A} =_{def} (U_{\mathcal{A}}, I_{\mathcal{A}})$ a $\mathcal{L}$-structure. $\mathcal{A}$ is called **matching with φ** if $I_{\mathcal{A}}$ is defined for all predicate and function symbols which appear in φ, i.e. if $\mathcal{L}(\Sigma_1) \subseteq \mathcal{L}(\Sigma)$.

FOL with Equality

Often, one assumes that the predicate symbol $\doteq$ is **built-in** and **interpreted by identity** in all structures $\mathcal{A}$: $I_{\mathcal{A}}(\doteq) =_{def} \{(x, x) : x \in U_{\mathcal{A}}\}$. In that case, $\doteq$ is not listed among the predicates in a model $\mathcal{A}$.

Definition 4.9 (Variable assignment ϱ)

A variable assignment ϱ over a $\mathcal{L}$-structure $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ is a function

$$\varrho : \mathrm{Var} \to U_\mathcal{A};\ x \mapsto \varrho(x).$$

In the next chapter, the Hoare calculus, this is exactly what we will call state of a program.

Example 4.10

Let φ be the formula

$$\mathsf{q}(x) \vee \forall z(\mathsf{p}(x, g(z))) \vee \exists x(\forall y(\mathsf{p}(f(x), y) \wedge \mathsf{q}(a)))$$

- $U = \mathbb{R}$
- $I(a)$: the constant π,
- $I(f) : I(f) = \sin : \mathbb{R} \to \mathbb{R}$ and
  $I(g) = \cos : \mathbb{R} \to \mathbb{R}$,
- $I(\mathsf{p}) = \{(r, s) \in \mathbb{R}^2 : r \leq s\}$ and
  $I(\mathsf{q}) = [3, \infty) \subseteq \mathbb{R}$,
- $I(x)$: the constant $\frac{\pi}{2}$.

Is φ true in $\mathcal{M} = (U, I)$?

- $\mathcal{M} \not\models \mathsf{q}(x)$ because $\mathcal{M}(\mathsf{q}) = [3, \infty)$ and $\mathcal{M}(x) = \frac{\pi}{2} \notin [3, \infty)$,
- $\mathcal{M} \not\models \forall z(\mathsf{p}(x, g(z)))$ because $(\mathcal{M}(x), \mathcal{M}(g)(\mathcal{M}(z))) = (\frac{\pi}{2}, \cos(u)) \notin \mathcal{M}(\mathsf{p}) = \{(r, s) \in \mathbb{R}^2 : r \leq s\}$ and $\cos(u) < \frac{\pi}{2}$ for all u.
- $\mathcal{M} \models \mathsf{q}(a)$ because $\mathcal{M}(a) = \pi \in \mathcal{M}(\mathsf{q}) = [3, \infty)$ is satisfied.
- $\mathcal{M} \models \exists x(\forall y(\mathsf{p}(f(x), y)))$ because not for all $u' \in \mathbb{R}$ exists a $u \in \mathbb{R}$ such that $(\mathcal{M}(f)(\mathcal{M}(x)), \mathcal{M}(y)) = (\sin(u), u') \in \mathcal{M}(\mathsf{p}) = \{(r, s) \in \mathbb{R}^2 \mid r \leq s\}$ is fulfilled (e.g. $u' = -2$).

$\Rightarrow \mathcal{M} \not\models \varphi$

Some structures

Details $\rightsquigarrow$ **blackboard**.

We consider $\mathcal{L}(\Sigma) = \{\leq, \doteq\}$ and the structures

1. $(\mathbb{Z}, \leq^{\mathbb{Z}})$
2. $(\mathbb{R}, \leq^{\mathbb{R}})$
3. $(\mathbb{Q}, \leq^{\mathbb{Q}})$
4. $(\mathbb{Q}_0^+, \leq^{\mathbb{Q}_0^+})$

Give formulae in $\mathcal{L}(\Sigma)$ that are true in all structures, or just in only one of them.

Is there a formula ϕ which can distinguish between $(\mathbb{R}, \leq^{\mathbb{R}})$ and $(\mathbb{Q}, \leq^{\mathbb{Q}})$?

Ludwig Wittgenstein

Ein Bild hielt uns gefangen. Und heraus konnten wir nicht, denn es lag in unserer Sprache, und sie schien es uns nur unerbittlich zu wiederholen.

Philosophische Untersuchungen, §115

Some structures (2)

We have seen some formulae that are true in these structures. Can we come up with a finite axiomatization?

Dense linear order without endpoints

It turns out, that the set of all formulae true in $(\mathbb{R}, \leq^{\mathbb{R}})$ coincides with the set all formulae true in $(\mathbb{Q}, \leq^{\mathbb{Q}})$. An axiomatization is given by the finite set of formulae stating that $<$ is a dense, linear order without endpoints. This theory is also **complete**.

Definition 4.11 (Semantics of first order logic, Model $\mathcal{A}$)

Let φ be a formula, $\mathcal{A}$ a structure matching with φ and ϱ a variable assignment over $\mathcal{A}$. For each term t, which can be built from components of φ, we define the value of t in the structure $\mathcal{A}$, called $\mathcal{A}(t)$.

1. for a variable x: $\mathcal{A}(x) =_{def} \varrho(x)$.
2. if t has the form $t = f^k(t_1, \ldots, t_k)$, with $t_1, \ldots, t_k$ being terms and f^k a k-dimensional function symbol, then $\mathcal{A}(t) =_{def} f^{\mathcal{A}}(\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k))$.

Compare with Definition 5.10.

Definition (continued)

We define inductively the logical value of a formula φ in $\mathcal{A}$:

1. if $\varphi =_{def} \mathsf{p}^k(t_1, \ldots, t_k)$ with the terms $t_1, \ldots, t_k$ and the k-dimensional predicate symbol p^k, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \mathbf{t}, & \text{if } (\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)) \in \mathsf{p}^{\mathcal{A}}, \\ \mathbf{f}, & \text{else.} \end{cases}$$

2. if $\varphi =_{def} \neg\psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \mathbf{t}, & \text{if } \mathcal{A}(\psi) = \mathbf{f}, \\ \mathbf{f}, & \text{else.} \end{cases}$$

Definition (continued)

3. if $\varphi =_{def} (\psi \vee \eta)$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \mathbf{t}, & \text{if } \mathcal{A}(\psi) = \mathbf{t} \text{ or } \mathcal{A}(\eta) = \mathbf{t}, \\ \mathbf{f}, & \text{else.} \end{cases}$$

4. if $\varphi =_{def} \exists x\, \psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \mathbf{t}, & \text{if } \exists d \in U_{\mathcal{A}} : \mathcal{A}_{[x/d]}(\psi) = \mathbf{t}, \\ \mathbf{f}, & \text{else.} \end{cases}$$

For $d \in U_{\mathcal{A}}$ let $\mathcal{A}_{[d/x]}$ be the structure $\mathcal{A}'$, **identical to $\mathcal{A}$** except for the definition of $x^{\mathcal{A}'}$: $x^{\mathcal{A}'} =_{def} d$ (whether $I_{\mathcal{A}}$ is defined for x or not).

Definition (continued)

We write:

- $\mathcal{A} \models \varphi[\varrho]$ for $\mathcal{A}(\varphi) = \mathbf{t}$: $\mathcal{A}$ is a **model** for φ with respect to ϱ.
- If φ does not contain free variables, then $\mathcal{A} \models \varphi[\varrho]$ is independent from ϱ (and we leave it out).
- If **there is at least one model for** φ, then φ is called satisfiable or consistent.

A **free variable** is a variable which is not in the scope of a quantifier. For instance, z is a free variable of $\forall x \mathsf{p}(x, z)$ but not free (or bounded) in $\forall z \exists x \mathsf{p}(x, z)$.

A variable can occur free and bound in the same formula. So, to be precise, we have to talk about a particular **occurrence** of a variable: the very position of it in the formula.

Definition 4.12 (Valid)

1. A **theory** is a set of formulae without free variables: $T \subseteq \mathrm{Fml}_{\mathcal{L}}$. The structure $\mathcal{A}$ **satisfies** T if $\mathcal{A} \models \varphi$ holds for all $\varphi \in T$. We write $\mathcal{A} \models T$ and call $\mathcal{A}$ **a model of** T.
2. A $\mathcal{L}$-formula φ is called **$\mathcal{L}$-valid**, if **for all matching $\mathcal{L}$-structures $\mathcal{A}$ the following holds:** $\mathcal{A} \models \varphi$.

From now on we suppress the language $\mathcal{L}$, because it is obvious from context.

Definition 4.13 (Consequence set $Cn(T)$)

A formula φ follows semantically from T, if for all structures $\mathcal{A}$ with $\mathcal{A} \models T$ also $\mathcal{A} \models \varphi$ holds.
We write: $T \models \varphi$.

In other words: all models of T do also satisfy φ.

We denote by $Cn_{\mathcal{L}}(T) =_{def} \{\varphi \in \mathsf{Fml}_{\mathcal{L}} : T \models \varphi\}$, or simply $Cn(T)$, the semantic consequence operator.

Lemma 4.14 (Properties of $Cn(T)$)

The semantic consequence operator Cn has the following properties

1. T-*extension:* $T \subseteq Cn(T)$,
2. *Monotony:* $T \subseteq T' \Rightarrow Cn(T) \subseteq Cn(T')$,
3. *Closure:* $Cn(Cn(T)) = Cn(T)$.

Lemma 4.15 ($\varphi \notin Cn$ (T))

$\varphi \notin Cn(T)$ *if and only if there is a structure* $\mathcal{A}$ *with* $\mathcal{A} \models T$ *and* $\mathcal{A} \models \neg\varphi$.

Or: $\varphi \notin Cn(T)$ *if and only if* there is a counterexample (a model of T in which φ is not true).

Definition 4.16 (*MOD*(T), *Cn*$(\mathcal{U})$)

For $T \subseteq \mathsf{Fml}_{\mathcal{L}}$, then we denote by **MOD**(T) the set of all $\mathcal{L}$-structures $\mathcal{A}$ which are models of T:

$$MOD(T) =_{def} \{\mathcal{A} : \mathcal{A} \models T\}.$$

If $\mathcal{U}$ is a set of structures then we can consider all sentences, which are true in all structures. We call this set also *Cn*$(\mathcal{U})$:

$$Cn(\mathcal{U}) =_{def} \{\varphi \in \mathsf{Fml}_{\mathcal{L}} : \text{ for all } \mathcal{A} \in \mathcal{U} : \mathcal{A} \models \varphi\}.$$

MOD is obviously dual to *Cn*:

$$Cn(MOD(T)) = Cn(T), \quad MOD(Cn(T)) = MOD(T).$$

Most theorems and notions from SL carry over **literally** to FOL.

Definition 4.17 (Completeness of a theory T)

T is called **complete**, if for each formula $\varphi \in \mathsf{Fml}_{\mathcal{L}}$: $T \models \varphi$ or $T \models \neg\varphi$ holds.

Attention:

Do not mix up this last condition with the property of a structure v (or a model): **each structure is complete in the above sense**.

Lemma 4.18 (Ex Falso Quodlibet)

T is consistent if and only if $Cn(T) \neq \mathsf{Fml}_{\mathcal{L}}$.

Predicate- or function- symbols?

How to formalize **each animal has a brain**?

1. Two unary predicate symbols: $\mathsf{animal}(x)$, $\mathsf{has\_brain}(x)$. The statement becomes

$$\forall x\ (\ \mathsf{animal}(x) \rightarrow \mathsf{has\_brain}(x)\ )$$

2. Finally, what about a binary predicate $\mathsf{is\_brain\_of}(y, x)$ and the statement

$$\forall x\ \ (\ \mathsf{animal}(x)\ \rightarrow\ \exists y\, \mathsf{is\_brain\_of}(y, x)\ )$$

3. But why not introducing a unary function symbol $brain\_of(x)$ denoting x's brain? Then

$$\forall x\, \exists y\ \ (\ \mathsf{animal}(x)\ \rightarrow\ y \doteq brain\_of(x)\ )$$

4.4 Examples: LTL and arithmetic

Example: FO ($\leq$)

Monadic first-order logic of order, denoted by **FO ($\leq$)**, is first-order logic with the only binary symbol $\leq$ (except equality, which is also allowed) and, additionally, any number of **unary predicates**. The theory assumes that $\leq$ is a linear order with least element, but nothing else:
$\forall x\, x \leq x$, $\forall x \forall y\, (x \leq y \vee y \leq x)$,
$\forall x \forall y\, ((x \leq y \wedge y \leq x) \rightarrow x \doteq y)$, $\exists x \forall y\, x \leq y$.

A typical model is given by

$$\mathcal{N} = \langle \mathbb{N}_0, \leq^{\mathbb{N}_0}, \mathsf{p}_1^{\mathcal{N}}, \mathsf{p}_2^{\mathcal{N}}, \ldots \mathsf{p}_n^{\mathcal{N}} \rangle$$

where $\leq^{\mathbb{N}_0}$ is the usual ordering on the natural numbers and $\mathsf{p}_i^{\mathcal{N}} \subseteq \mathbb{N}_0$.

The sets $\mathsf{p}_i^{\mathcal{N}}$ determine the **timepoints** where the property p_i holds.

What can we express in FO ($\leq$)?

Can we find formulae that express that

- a property r is true **infinitely often**?
- whenever r is true, then s is true in the next timepoint?
- r is true at all even timepoints and $\neg$r at all **odd timepoints**?

LTL revisited (1)

1. The monadic first-order theory of (linear) order, **FO**$(\leq)$ (see Slide 322) is equivalent to **LTL**:
   There is a **translation** *from sentences of* ***LTL*** *to sentences of* ***FO***$(\leq)$ *and vice versa, such that the* **LTL sentence is true in** λ, π *if and only if* **its translation is true in the associated first-order structure**.

   This was proved by Hans Kamp in his PhD thesis 1968.

Relation to first-order logic (2)

1. More precisely: an infinite **path** λ is described as a first-order structure with **domain** $\mathbb{N}$ and predicates P_p for $\mathsf{p} \in \mathcal{P}rop$. The predicates stand for the set of timepoints where p is true. So each path λ can be represented as a structure $\mathcal{N}_\lambda = \langle \mathbb{N}, \leq^{\mathbb{N}}, \mathsf{P}_1^{\mathcal{N}}, \mathsf{P}_2^{\mathcal{N}}, \ldots \mathsf{P}_n^{\mathcal{N}} \rangle$.

   Then each **LTL** formula ϕ translates to a first-order formula $\alpha_\phi(x)$ with one free variable s.t.

   $$\phi \text{ is true in } \lambda[n, \infty] \quad \textit{if and only if} \quad \alpha_\phi(n) \text{ is true in } \mathcal{N}_\lambda.$$

   And conversely: for each first-order formula with a free variable there is a corresponding **LTL** formula s.t. the same condition holds.

Does that mean that LTL is useless?

$\mathbb{N}$ in different languages

Example 4.19 (Natural numbers in different languages)

- $\mathcal{N}_{Pres} = (\mathbb{N}_0, 0^{\mathcal{N}}, +^{\mathcal{N}}, \dot{=}^{\mathcal{N}})$ („*Presburger Arithmetik*"),
- $\mathcal{N}_{Peano} = (\mathbb{N}_0, 0^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, \dot{=}^{\mathcal{N}})$ („*Peano Arithmetik*"),
- $\mathcal{N}_{PA'} = (\mathbb{N}_0, 0^{\mathcal{N}}, 1^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, \dot{=}^{\mathcal{N}})$ (variant of $\mathcal{N}_{Peano}$).

These sets each define the **same natural numbers**, but in different languages.

Question:

If the language is bigger then we might express more.
Is $\mathcal{L}_{PA'}$ strictly more expressive than $\mathcal{L}_{Peano}$?

Answer:

No, because one can replace the $1^{\mathcal{N}}$ by a $\mathcal{L}_{Peano}$-formula: there is a $\mathcal{L}_{Peano}$-formula $\phi(x)$ so that for each variable assignment ρ the following holds:

$$\mathcal{N}_{PA'} \models_\rho \phi(x) \textit{ if and only if } \rho(x) = 1^{\mathcal{N}}$$

- Thus we can define a macro for 1.
- Each formula of $\mathcal{L}_{PA'}$ can be transformed into an equivalent formula of $\mathcal{L}_{Peano}$.

Question:

Is $\mathcal{L}_{Peano}$ perhaps more expressive than $\mathcal{L}_{Pres}$, or can the multiplication be defined somehow?

Indeed, $\mathcal{L}_{Peano}$ is more expressive:

- the set of sentences valid in $\mathcal{N}_{Pres}$ is decidable, whereas
- the set of sentences valid in $\mathcal{N}_{Peano}$ is not even recursively enumerable.

Different languages for $\mathcal{N}_{Pres}$

- $\mathcal{N}_{Pres} = (\mathbb{N}_0, 0^{\mathcal{N}}, 1^{\mathcal{N}}, +^{\mathcal{N}}, \dot{=}^{\mathcal{N}})$
- $\mathcal{N}_{Pres} = (\mathbb{N}_0, 0^{\mathcal{N}}, S^{\mathcal{N}}, +^{\mathcal{N}}, \dot{=}^{\mathcal{N}})$
- $\mathcal{N}_{Pres} = (\mathbb{N}_0, 0^{\mathcal{N}}, 1^{\mathcal{N}}, S^{\mathcal{N}}, +^{\mathcal{N}}, \dot{=}^{\mathcal{N}})$
- $\mathcal{N}_{Pres} = (\mathbb{N}_0, 1^{\mathcal{N}}, +^{\mathcal{N}}, \dot{=}^{\mathcal{N}})$

All these structures, resp. their formulae are interdefinable.

Question:

We shall introduce $\mathcal{Z} = (\mathbb{Z}, 0^{\mathbb{Z}}, 1^{\mathbb{Z}}, +^{\mathbb{Z}}, -^{\mathbb{Z}}, \cdot^{\mathbb{Z}}, <^{\mathbb{Z}})$ in the chapter about the Hoare calculus. **How does it compare with $\mathcal{N}_{Peano}$?**

- "$\doteq$" can be **defined** with $<$ and vice versa in $\mathcal{Z}$ and $\mathcal{N}_{Peano}$ (resp. in $\mathcal{N}_{Pres}$).
- "$-$" can also be **defined** with the other constructs.
- $\mathcal{N}_{Peano}$ can be **defined in** $\mathcal{Z}$ with an appropriate formula $\phi(x)$:

$$\mathcal{Z} \models_\rho \phi(x) \textit{ if and only if } \rho(x) \in \mathbb{N}_0$$

- Can $\mathcal{N}_{Peano}$ be defined in $(\mathbb{Z}, +^{\mathbb{Z}}, \cdot^{\mathbb{Z}})$?

To be more precise, for each $\mathcal{L}_{Peano}$ formula ϕ, there is a $\mathcal{L}_{\mathcal{Z}}$ formula ϕ' such that: if $\mathcal{N}_{PA'} \models \phi$ then $\mathcal{Z} \models \phi'$.

So $\mathcal{Z}$ is at least as difficult as $\mathcal{N}_{Peano}$.

The converse is true as well. Therefore although the theories of $\mathcal{Z}$ and $\mathcal{N}_{Peano}$ are **not identical**, the truth of a formula in one of them can be reduced to the truth of a translated formula in the other one.

Question:

What about $\mathcal{R} = (\mathbb{R}, 0^{\mathbb{R}}, 1^{\mathbb{R}}, +^{\mathbb{R}}, -^{\mathbb{R}}, .^{\mathbb{R}}, <^{\mathbb{R}})$ and $\mathcal{Q} = (\mathbb{Q}, 0^{\mathbb{Q}}, 1^{\mathbb{Q}}, +^{\mathbb{Q}}, -^{\mathbb{Q}}, .^{\mathbb{Q}}, <^{\mathbb{Q}})$? **How do they compare with $\mathcal{N}_{Peano}$?**

- State a formula that **distinguishes** both structures.
- Can one define $\mathcal{Q}$ within $\mathcal{R}$ (as we did define $\mathcal{N}_{Peano}$ in $\mathcal{Z}$)?
- Is there an axiomatization of $\mathcal{R}$?

In general, theories of **specific** structures are undecidable. But it depends on the underlying language.

4.5 Prenex normal form and clauses

Prenex normal form (1)

Definition 4.20 (Prenex normal form)

We consider FOL-formulae built not just over $\exists, \neg, \vee$, but also using the macros $\forall, \wedge, \rightarrow, \leftrightarrow$.
A FOL formula of the form

$$Q_1 x_1 \ldots Q_n x_n \psi$$

where ψ does not contain any quantifiers, is called in prenex normal form.

Proposition 4.21

Each FOL*-formula is equivalent to one in* prenex normal form.

Prenex normal form (2)

Instead of proving the last proposition by induction on the structure of formulae, we illustrate how this can be done.

1. Move the $\neg$ sign to the innermost parts (the atomic formulae). Here we use just the boolean laws.
2. Move the leftmost quantifier to the front in such a way, that its scope is the whole formula.
3. Do the last step recursively until all quantifiers are in front.

Prenex normal form (3)

How can we do the second step?

- $\forall x\varphi(x) \wedge \psi$ is replaced by $\forall x(\varphi(x) \wedge \psi)$ if x does not occur in ψ, and by $\forall z(\varphi(z) \wedge \psi)$ otherwise (for suitable z).
- $\forall x\varphi(x) \vee \psi$ is replaced by $\forall x(\varphi(x) \vee \psi)$ if x does not occur in ψ, and by $\forall z(\varphi(z) \vee \psi)$ otherwise (for suitable z).
- $\exists x\varphi(x) \wedge \psi$ is replaced by $\exists x(\varphi(x) \wedge \psi)$ if x does not occur in ψ, and by $\forall z(\varphi(z) \wedge \psi)$ otherwise (for suitable z).
- $\exists x\varphi(x) \vee \psi$ is replaced by $\exists x(\varphi(x) \vee \psi)$ if x does not occur in ψ, and by $\forall z(\varphi(z) \vee \psi)$ otherwise (for suitable z).

Prenex normal form (4)

A formula in prenex form $Q_1 x_1 \ldots Q_n x_n \psi$ can be even more normalised: the ψ part can be put in conjunctive or disjunctive normal form (they can be viewed as SL formulae and the boolean laws can be applied).

Later, we need to get rid of existential quantifiers: working with universally quantified formulae is more appropriate for the resolution calculus (as will become clear in a while).

Satisfiability equivalence (1)

Proposition 4.22 (Skolemization)

Each FOL *formula φ in a language $\mathcal{L}$ can be transformed to a formula φ' in an* extended language *$\mathcal{L}' \supseteq \mathcal{L}$ (which contains* **additional function symbols***) such that the following holds: (1) φ' is in prenex normal form and contains only $\forall$ quantifiers, and (2) if φ is satisfiable in a $\mathcal{L}$-model, then this $\mathcal{L}$-model can be extended to a model in the language $\mathcal{L}'$ and φ' is satisfiable in this extended model.*

Thus any formula is satisfiability equivalent to a **universal formula in an extended language**.

Note: **it is not possible to transform each formula into an equivalent formula that contains only universal quantifiers (in its prenex normal form).**

Satisfiability equivalence (2)

Example 4.23 (Eliminating $\exists$ quantifiers)

We consider the two formulae

1. $\exists x \exists y \, (\mathsf{q}(x) \wedge \neg \mathsf{q}(y))$,
2. $\forall x \exists y \, (\mathsf{p}(x, y) \rightarrow \neg \mathsf{q}(y))$.

How can we get rid of the existential quantifiers?

Satisfiability equivalence (3)

The first formula is easy to transform: we just add two new constants c, c' and instantiate them for the variables. This leads to $\mathsf{q}(c) \wedge \neg \mathsf{q}(c')$, or the two clauses $\mathsf{q}(c)$ and $\mathsf{q}(c')$.

The second formula is more complicated, because there is no single y. We have to take into account, that the y **depends on the chosen** x. Therefore we introduce a **new function symbol** $f(x)$: $\forall x\, \mathsf{p}(x, f(x)) \rightarrow \neg \mathsf{q}(f(x))$. Then we have to transform the formula into disjunctive form: $\forall x\, \neg \mathsf{p}(x, f(x)) \vee \neg \mathsf{q}(f(x))$.

The newly introduced function symbols are also called **Skolem functions**, the technique is called **skolemization**.

Satisfiability equivalence (4)

Applying the technique of the last slide recursively, we can transform each set M of $\mathcal{L}$-formulae into a set M' of universally quantified formulae in prenex form **in an extended language** $\mathcal{L}'$ (where $\mathcal{L}'$ is obtaind from $\mathcal{L}$ by adding certain function symbols).

Theorem 4.24 (Satisfiability equivalence)

Let M be a set of FOL *sentences in a language $\mathcal{L}$ and M' be its transform in $\mathcal{L}' \supseteq \mathcal{L}$, which is universal and in prenex form.*

M *is satisfiable if and only if* M' *is satisfiable.*

So we do not get equivalence in the full sense, but satisfiability equivalence. This suffices to prove refutation completeness of the resolution calculus for FOL.

Universal clauses

Using the techniques just introduced, it is possible to transform any set M of FOL formulae into a set M' of formulae with the following properties:

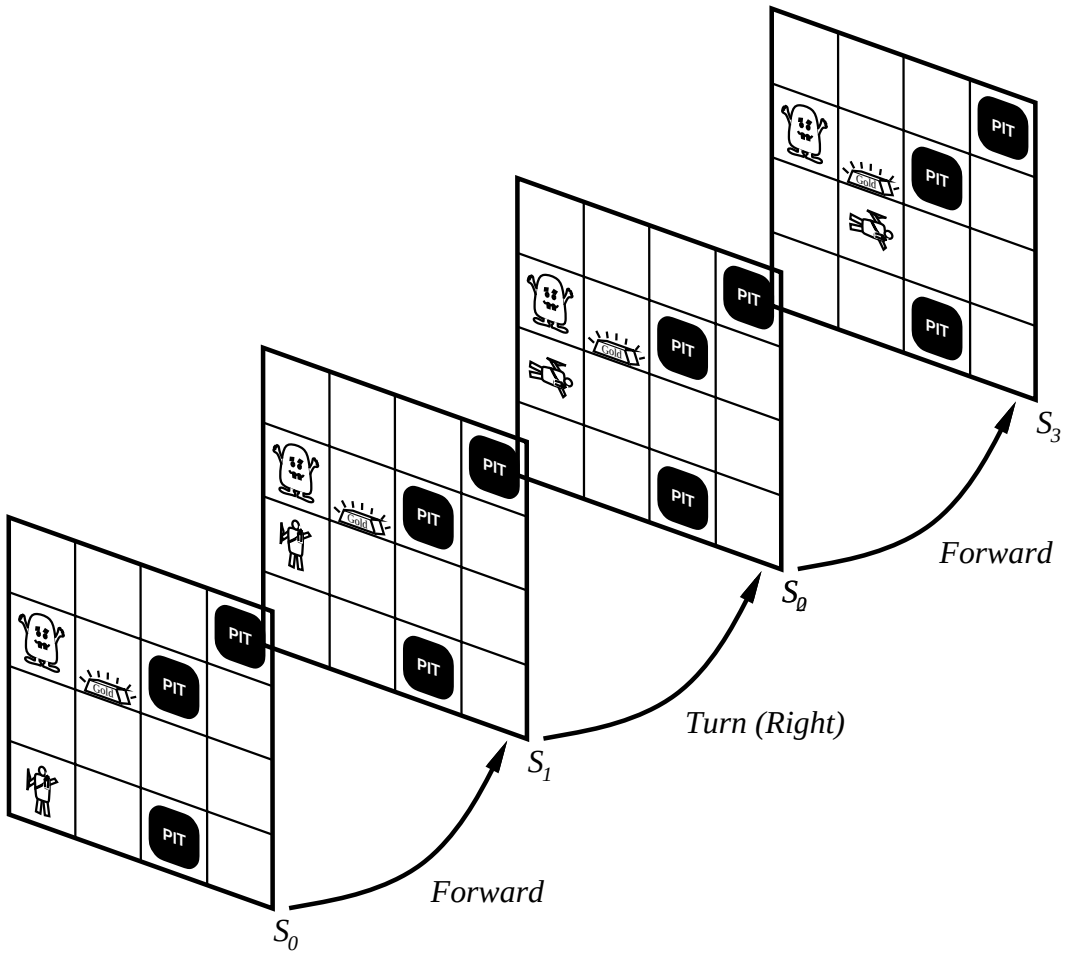
- all formulae are universal,
- all formulae are in prenex form $\mathsf{Q}_1 x_1 \ldots \mathsf{Q}_n x_n \psi$
- the formulae ψ after the quantifiers are in conjunctive normal form,
- M and M' are satisfiability equivalent.

We say that the set M' is in clausal form.

4.6 Sit-Calculus

Question:

How do we axiomatize the Wumpus-world in FOL?



Idea:

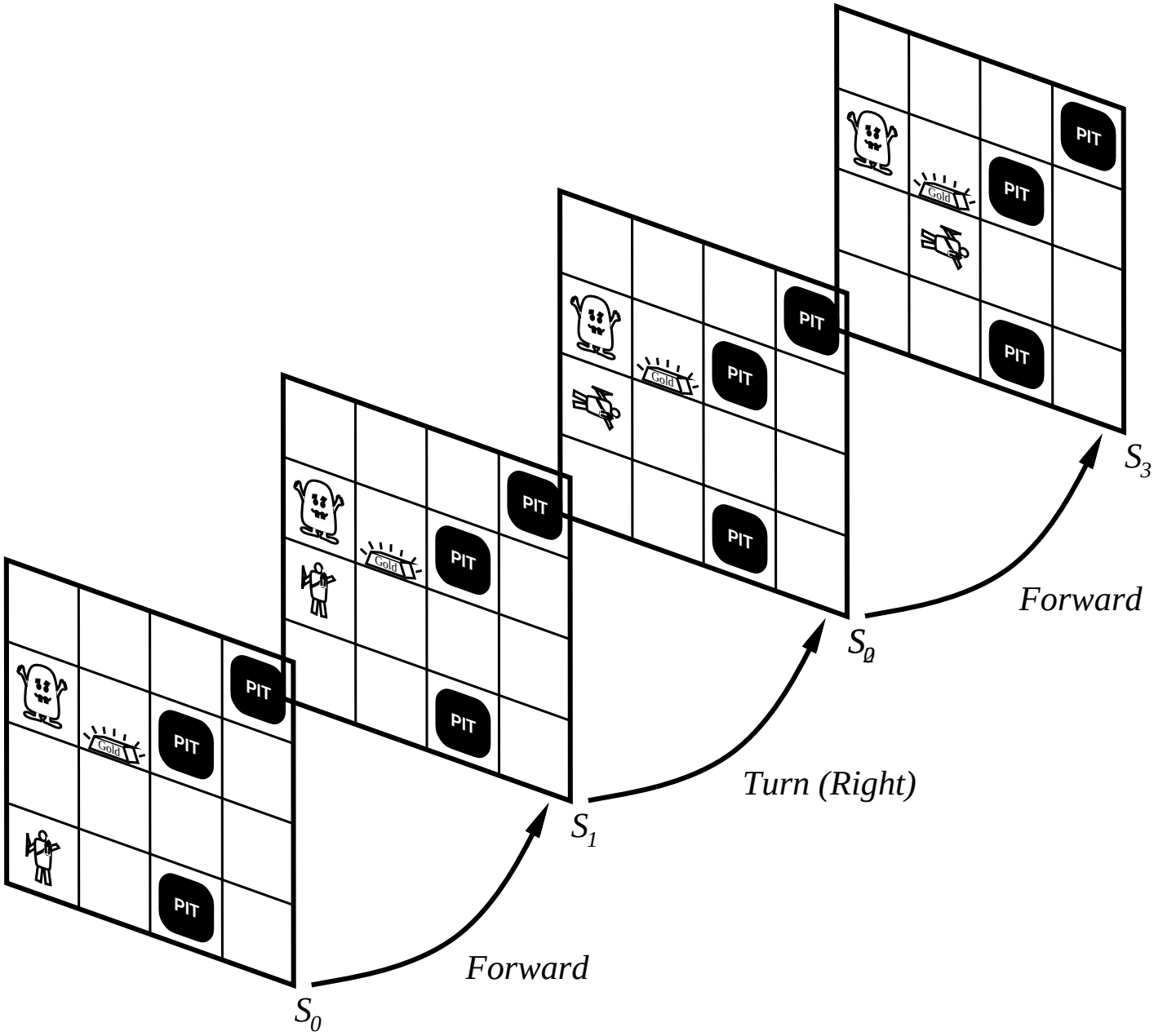
In order to describe actions or their effects consistently we consider the world as a sequence of situations (snapshots of the world). Therefore we have to extend each predicate by an additional argument.

We use the function symbol

$$result(action, situation)$$

as the term for the situation which emerges when the action $action$ is executed in the situation $situation$.

Actions: $Turn\_right$, $Turn\_left$, $Forward$, $Shoot$, $Grab$, $Release$, $Climb$.


PIT
Gold
S_0
Forward
S_1
Turn (Right)
S_2
Forward
S_3

We also need a **memory**, a predicate

$$\mathrm{At}(person, location, situation)$$

with $person$ being either $Wumpus$ or $Agent$ and $location$ being the actual position (stored as pair [i,j]).

Important axioms are the so called **successor-state axioms**, they describe how actions effect situations. The most general form of these axioms is

true afterwards $\iff$ an action made it true
or it is already true and
no action made it false

Axioms about $\mathsf{At}(p, l, s)$:

$$\mathsf{At}(p, l, \textbf{result}(Forward, s)) \leftrightarrow ((l \doteq location\_ahead(p, s) \wedge \neg \mathsf{Wall}(l))$$

$$\mathsf{At}(p, l, s) \rightarrow Location\_ahead(p, s) \doteq Location\_toward(l, Orient.(p, s))$$

$$\mathsf{Wall}([x, y]) \leftrightarrow (x \doteq 0 \vee x \doteq 5 \vee y \doteq 0 \vee y \doteq 5)$$

$$Location\_toward([x,y],0) \doteq [x+1,y]$$
$$Location\_toward([x,y],90) \doteq [x,y+1]$$
$$Location\_toward([x,y],180) \doteq [x-1,y]$$
$$Location\_toward([x,y],270) \doteq [x,y-1]$$

$$Orient.(Agent, s_0) \doteq 90$$
$$Orient.(p, \textbf{result}(a,s)) \doteq d \leftrightarrow$$
$$(a \doteq turn\_right \wedge d \doteq mod(Orient.(p,s) - 90, 360))$$
$$\vee (a \doteq turn\_left \wedge d \doteq mod(Orient.(p,s) + 90, 360))$$
$$\vee (Orient.(p,s) \doteq d \wedge \neg(a \doteq turn\_right \vee a \doteq turn\_left))$$

$mod(x,y)$ is the implemented "modulo"-function, assigning a value between 0 and y to each variable x.

Axioms about percepts, useful new notions:

$$\begin{aligned}
\text{Percept}([Stench, b, g, u, c], s) &\rightarrow \text{Stench}(s)\\
\text{Percept}([a, Breeze, g, u, c], s) &\rightarrow \text{Breeze}(s)\\
\text{Percept}([a, b, Glitter, u, c], s) &\rightarrow \text{At_Gold}(s)\\
\text{Percept}([a, b, g, Bump, c], s) &\rightarrow \text{At_Wall}(s)\\
\text{Percept}([a, b, g, u, Scream], s) &\rightarrow \text{Wumpus_dead}(s)
\end{aligned}$$

$$\begin{aligned}
\text{At}(Agent, l, s) \wedge \text{Breeze}(s) &\rightarrow \text{Breezy}(l)\\
\text{At}(Agent, l, s) \wedge \text{Stench}(s) &\rightarrow \text{Smelly}(l)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Adjacent}(l_1, l_2) &\leftrightarrow \exists\, d\, l_1 \doteq Location_toward(l_2, d) \\
\mathsf{Smelly}(l_1) &\rightarrow \exists l_2\, \mathsf{At}(Wumpus, l_2, s) \wedge \\
&\quad (l_2 \doteq l_1 \vee \mathsf{Adjacent}(l_1, l_2))
\end{aligned}$$

$$\mathsf{Percept}([none, none, g, u, c], s) \wedge$$
$$\mathsf{At}(Agent, x, s) \wedge \mathsf{Adjacent}(x, y) \rightarrow OK(y)$$
$$(\neg\mathsf{At}(Wumpus, x, t) \wedge \neg\mathsf{Pit}(x)) \rightarrow OK(y)$$
$$\mathsf{At}(Wumpus, l_1, s) \wedge \mathsf{Adjacent}(l_1, l_2) \rightarrow \mathsf{Smelly}(l_2)$$
$$\mathsf{At}(Pit, l_1, s) \wedge \mathsf{Adjacent}(l_1, l_2) \rightarrow \mathsf{Breezy}(l_2)$$

Axioms to describe actions:

$$
\begin{array}{lcl}
\text{Holding}(Gold, \textbf{result}(Grab, s)) & \leftrightarrow & \text{At\_}Gold(s) \vee \\
 & & \text{Holding}(Gold, s) \\
\text{Holding}(Gold, \textbf{result}(Release, s)) & \leftrightarrow & \square \\
\text{Holding}(Gold, \textbf{result}(Turn\_right, s)) & \leftrightarrow & \text{Holding}(Gold, s) \\
\text{Holding}(Gold, \textbf{result}(Turn\_left, s)) & \leftrightarrow & \text{Holding}(Gold, s) \\
\text{Holding}(Gold, \textbf{result}(Forward, s)) & \leftrightarrow & \text{Holding}(Gold, s) \\
\text{Holding}(Gold, \textbf{result}(Climb, s)) & \leftrightarrow & \text{Holding}(Gold, s)
\end{array}
$$

Each effect must be described carefully.

Axioms describing preferences among actions:

$$\mathsf{Great}(a,s) \rightarrow \mathsf{Action}(a,s)$$
$$(\mathsf{Good}(a,s) \wedge \neg\exists b\, \mathsf{Great}(b,s)) \rightarrow \mathsf{Action}(a,s)$$
$$(\mathsf{Medium}(a,s) \wedge \neg\exists b\, (\mathsf{Great}(b,s) \vee \mathsf{Good}(b,s))) \rightarrow \mathsf{Action}(a,s)$$
$$(\mathsf{Risky}(a,s) \wedge \neg\exists b\, (\mathsf{Great}(b,s) \vee \mathsf{Good}(b,s) \vee \mathsf{Medium}(a,s))) \rightarrow$$
$$\rightarrow \mathsf{Action}(a,s)$$
$$\mathsf{At}(Agent,[1,1],s) \wedge \mathsf{Holding}(Gold,s) \rightarrow \mathsf{Great}(Climb,s)$$
$$\mathsf{At\_}Gold(s) \wedge \neg\mathsf{Holding}(Gold,s) \rightarrow \mathsf{Great}(Grab,s)$$
$$\mathsf{At}(Agent,l,s) \wedge \neg\mathsf{Visited}(Location\_ahead(Agent,s)) \wedge$$
$$\wedge\mathsf{OK}(Location\_ahead(Agent,s)) \rightarrow \mathsf{Good}(Forward,s)$$
$$\mathsf{Visited}(l) \leftrightarrow \exists s\, \mathsf{At}(Agent,l,s)$$

The goal is not only to find the gold but also to return safely.

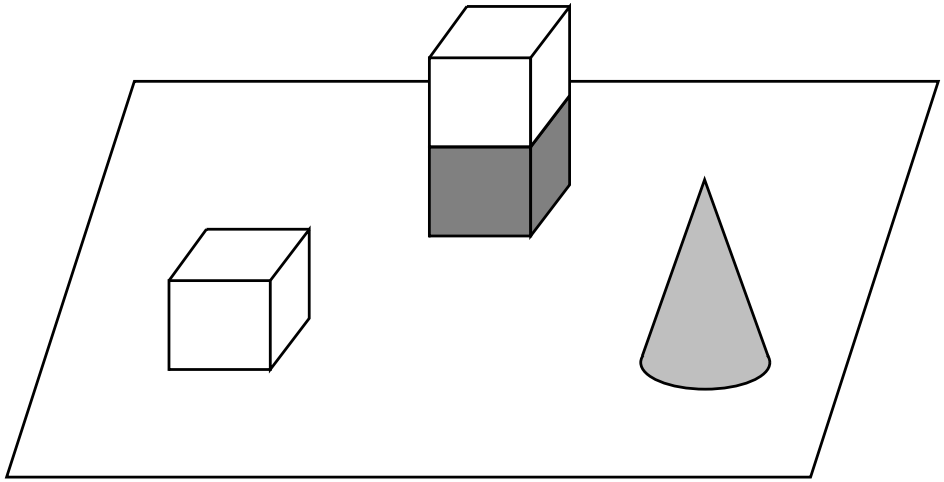
We need additional axioms like

$$\mathsf{Holding}(Gold,s) \rightarrow \mathsf{Go\_back}(s).$$

4.7 The Blocksworld

Blocksworld

The Blocks World (BW) is one of the most popular domains in AI (first used in the 1970s). However, the setting is easy:



Domain

Blocks of various shapes, sizes and colors sitting on a table and on each other.

(Here: Quadratic blocks of equal size.)

Actions

Pick up a block and put it to another position (tower of blocks or table). Only the topmost blocks can be used.

How to formalize this? What language to use?

Choosing the predicates: First try.

1. One action: move. We add a ternary predicate move(b, x, y): The block b is moved from x to y if both b and y are clear (nothing on top of them).
2. So we need a predicate clear(x).
3. One predicate on(b, x) meaning *block b is on x*.
4. But then clear(x) can be defined by $\forall y \neg \text{on}(y, x)$.

Problem: What about the table?

If we view the table as a simple block: clear$(table)$ means that **there is nothing on the table**.

Choosing the predicates: Second try.

Keep the current framework, view the table as a block but add an additional binary predicate

$$\text{move\_to\_table}(b, x),$$

meaning that the block b is moved from x to the table.

clear(x) is interpreted as there is free place on x to put a block on x. This interpretation does also work for $x \doteq table$.

Choosing the predicates: Third try.

Definition 4.25 ($\mathcal{L}_{BW}$)

The BW language $\mathcal{L}_{BW} \subseteq \mathcal{L}_{FOL}$ is the subset of FOL having a signature consisting of the two **binary** predicate symbols above and $\doteq$.

We define the following macros:

$$\begin{aligned} \mathsf{on}(x,y) &: \mathsf{above}(x,y) \wedge \neg(\exists z(\mathsf{above}(x,z) \wedge \mathsf{above}(z,y))) \\ \mathsf{onTable}(x) &: \neg(\exists y(\mathsf{above}(x,y))) \\ \mathsf{clear}(x) &: \neg(\exists y(\mathsf{above}(y,x))) \end{aligned}$$

How do the $\mathcal{L}_{BW}$-structures look like?

Do they all make sense?

Our blocksworld has a very specific structure, which should be reflected in the models!

Definition 4.26 (Chain)

Let A be a nonempty set. We say that $(A, <)$ is a chain if $<$ is a binary relation on A which is

1. **irreflexive**,
2. **transitive**, and
3. **connected**, (i.e., for all $a, b \in A$ it holds that either $a < b$, $a = b$, or $a > b$).

(A chain is interpreted as a tower of blocks.)

Definition 4.27 ($\mathcal{L}_{\mathsf{BW}}$-BW-structure)

An **BW-structure** is a $\mathcal{L}_{\mathsf{BW}}$-structure $\mathcal{A} = (U, I)$ in which $I(\mathsf{above})$ is a finite and disjoint union of chains over U, i.e.

$$I(\mathsf{above}) = \bigcup_{(A,<)\in A'} \{(a,b) \in A^2 \mid a > b\}\}$$

where A' is a set of chains over U such that for all $(A_1, <_1), (A_2, <_2) \in A'$ with $(A_1, <_1) \neq (A_2, <_2)$ it holds that $A_1 \cap A_2 = \emptyset$.

(Note: $I(\mathsf{above})$ is transitive!)

Definition 4.28 (BW-theory)

The theory $Th(BW)$ consists of all $\mathcal{L}_{\mathsf{BW}}$-sentences which are true in all BW-structures.

Is the theory complete?

No, consider for example

$$\forall x, y(\mathsf{onTable}(x) \wedge \mathsf{onTable}(y) \rightarrow x \doteq y)$$

- What about planning in the blocksworld?
- This should be done automatically as in the case of the Sudoku game or the puzzle.
- Thus we need a FOL theorem prover.

$\rightsquigarrow$ An axiomatization is needed!

A complete axiomatization

The set $\mathcal{AX}_{\mathsf{BW}}$ was proposed by Cook and Liu (2003):

1. $\forall x\, \neg \mathsf{above}(x, x)$
2. $\forall x\, \forall y\, \forall z\, (\mathsf{above}(x, y) \wedge \mathsf{above}(y, z)) \rightarrow \mathsf{above}(x, z)$
3. $\forall x\, \forall y\, \forall z\, (\mathsf{above}(x, y) \wedge \mathsf{above}(x, z)) \rightarrow (y = z \vee \mathsf{above}(y, z) \vee \mathsf{above}(z, y))$
4. $\forall x\, \forall y\, \forall z\, (\mathsf{above}(y, x) \wedge \mathsf{above}(z, x)) \rightarrow (y = z \vee \mathsf{above}(y, z) \vee \mathsf{above}(z, y))$
5. $\forall x\, (\mathsf{onTable}(x) \vee \exists y(\mathsf{above}(x, y) \wedge \mathsf{onTable}(y)))$
6. $\forall x\, (\mathsf{clear}(x) \vee \exists y(\mathsf{above}(y, x) \wedge \mathsf{clear}(y)))$
7. $\forall x\, \forall y\, (\mathsf{above}(x, y) \rightarrow (\exists z \mathsf{on}(x, z) \wedge \exists z \mathsf{on}(z, y))$

The last statement says that if an element is not on top (y) then there is a block above it, and if an element is not at the bottom (x) then there is an element below it.

Is every $\mathcal{L}_{\mathsf{BW}}$-BW-structure also a model for $\mathcal{AX}_{\mathsf{BW}}$?

$\rightsquigarrow$ Exercise

Lemma 4.29

$Cn(\mathcal{AX}_{BW}) \subseteq Th(BW)$.

Proof: $\rightsquigarrow$ Exercise

Indeed, both sets are identical:

Theorem 4.30 (Cook and Liu)

$Cn(\mathcal{AX}_{BW}) = Th(BW)$.

Thus the axiomatization is sound and complete.

Additionally, the theory is decidable!

$\rightsquigarrow$ **We are ready to use a theorem prover!**

4.8 Higher order logic

Definition 4.31 (Second order logic $\mathcal{L}_{PL\,2}$)

The language $\mathcal{L}_{2nd\,OL}$ of second order logic consists of the language $\mathcal{L}_{\mathsf{FOL}}$ and additionally

- for each $k \in \mathbb{N}_0$: $X_1^k, X_2^k, \ldots, X_n^k, \ldots$ a countable set $\mathcal{RelVar}^k$ of k-ary predicate variables.

Thereby the set of terms gets larger:

- if X^k is a k-ary predicate variable and $t_1, \ldots, t_k$ are terms, then $X^k(t_1, \ldots, t_k)$ is also a term,

and also the set of formulae:

- if X is a predicate variable, φ a formula, then $(\exists X \varphi)$ and $(\forall X \varphi)$ are also formulae.

Not only elements of the universe can be quantified but also arbitrary subsets resp. k-ary relations.
The semantics do not change much – except for the new interpretation of formulae like $(\exists X \varphi)$, $(\forall X \varphi)$).

We also require from $I_{\mathcal{A}}$ that the new k-ary predicate variables are mapped onto k-ary relations on $U_{\mathcal{A}}$.

- if $\varphi =_{def} \forall X^k \psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \mathbf{t}, & \text{if for all } R^k \subseteq U_{\mathcal{A}} \times \cdots \times U_{\mathcal{A}} : \ \mathcal{A}_{[X^k/R^k]}(\psi) = \mathbf{t}, \\ \mathbf{f}, & else. \end{cases}$$

- if $\varphi =_{def} \exists X^k \psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \mathbf{t}, & \text{if there is a } R^k \subseteq U_{\mathcal{A}} \times \cdots \times U_{\mathcal{A}} \text{ with } \mathcal{A}_{[X^k/R^k]}(\psi) = \mathbf{t}, \\ \mathbf{f}, & else. \end{cases}$$

We can quantify over arbitrary n-ary relations, not just over elements (like in first order logic).

Dedekind/Peano Characterization of $\mathbb{N}$

The natural numbers satisfy the following axioms:

- Ax_1: 0 is a natural number.
- Ax_2: For each natural number n, there is exactly one **successor** $S(n)$ of it.
- Ax_3: There is no natural number which has 0 as its successor.
- Ax_4: Each natural number is successor of **at most** one natural number.
- Ax_5: The set of natural numbers is the smallest set N satisfying the following properties:
  1. $0 \in N$,
  2. $n \in N \Rightarrow S(n) \in N$.

The natural numbers are characterized up to isomorphy by these axioms.

How to formalize the last properties?

Language: We choose "0" as a constant and "$S(\cdot)$" as a unary function symbol.

Axiom for 0: $\forall x \neg S(x) \doteq 0$.

Axiom for S: $\forall x \forall y \, (S(x) \doteq S(y) \rightarrow x \doteq y)$.

Axiom 5: $\forall X \, (\, (X(0) \wedge \forall x \, (X(x) \rightarrow X(S(x)))) \rightarrow \forall y X(y))$.

While the first two axioms are **first-order**, the last one is essentially second-order.

Question

What do the following two 2nd order sentences mean?

- $$\forall x \forall y \, (x \doteq y \iff \forall X \, (X(x) \iff X(y))),$$

- $$\begin{aligned} \forall X \, \big(\; & \forall x \exists ! y X(x,y) \wedge \\ & \forall x \forall y \forall z \, (\, (X(x,z) \wedge X(y,z)) \rightarrow x \doteq y)) \\ & \rightarrow \forall x \exists y X(y,x) \, \big) \end{aligned}$$

Answer:

The first sentence shows that equality can be defined in 2nd OL (in contrast to FOL).
The second sentence holds in a structure *if and only if* **it is finite**. Note that this cannot be expressed in FOL, for a good reason: compactness would not hold anymore. Tghis would destroy all good properties of FOL.

While the semantics of $\mathcal{L}_{2nd\ OL}$ is a canonical extension of $\mathcal{L}_{\mathsf{FOL}}$, this does not hold for the calculus level. It can be shown that the set of **valid sentences** in $\mathcal{L}_{2nd\ OL}$ is not even recursively enumerable.

Attention:

There is no correct and complete calculus for 2nd Order Logic!

5. Verification II: Hoare calculus

Content of this chapter (1):

In this chapter we explore ways to use FOL for the verification of programs: does a program really do what we want? The method we discuss has been developed by C.A.R. Hoare and M. Floyd in the 60'ies.

We introduce a calculus to prove correctness of computer programs.

Verification: We argue why formal methods are useful/necessary for program verification.

Core Programming Language: An abstract but powerful programming language is introduced.

Hoare Logic: We introduce the Hoare Calculus which operates on triples $\{\phi_{pre}\}\ \mathsf{P}\ \{\psi_{post}\}$.

Content of this chapter (2):

Proof Calculi (Partial Correctness): Here we present three calculi (proof rules, annotation calculus, and the weakest precondition calculus) for partial correctness.

Proof Calculi (Total Correctness): Partial correctness is trivially satisfied when a program does not terminate. Therefore, we consider total correctness as well and an extension of the calculi to deal with it. We consider the partial and total correctness of programs.

Sound & Completeness: We briefly discuss the sound and completeness of the Hoare calculus.

5.1 Motivation

Why to formally specify/verify code?

- Formal specifications are often important for **unambiguous** documentations.
- Formal methods cut down software development and maintenance costs.
- Formal specification makes software easier to reuse due to a **clear specification**.
- Formal verification can ensure **error-free** software required by safety-critical computer systems.

Verification of Programs

How can we define the state of a program?

The state of a program is given by the contents of all variables.

What about the **size** of the state-space of a program?

The state space is usually infinite! Hence, the technique of model checking is inappropriate.

Software Verification Framework

Main reasons for formal specifications:

- **Informal** descriptions often lead to **ambiguities** which can result in serious (and potentially expensive) design flaws.
- Without formal specifications a rigorous verification is not possible!

Methodology

We assume the following methodology:

1. Build an **informal description** D of the program and the domain.
2. Convert D into an equivalent formula ϕ in a suitable logic.
3. (Try to) build a program P **realizing** ϕ.
4. Prove that P satisfies ϕ.

Methodology (2)

Some points to think about:

(i) Point 2 is a **non-trivial** and **non-formal** problem and thus "cannot be proven": D is an **informal** specification!

(ii) Often there are alternations between 3 and 4.

(iii) Sometimes one might realize that ϕ is not equivalent to D and thus one has to revise ϕ.

(iv) Often, P must have a **specific structure** to prove it against φ.

In this lecture, we will focus on points 3,4, and (iv).

Some Properties of the Procedure

Proof-based: Not every **state** of the system is considered (there are infinitely many anyway), rather a proof for correctness is constructed: this works then for **all** states.

Semi-automatic: Fully automatic systems are desireable but they are not always possible: undecidability, time constraints, efficiency, and “lack of intelligence”.

Some Properties of the Procedure (2)

Property-oriented: Only certain properties of a program are proven and not the "complete" behavior.

Application domain: Only sequential programs are considered.

Pre/post-development: The proof techniques are designed to be used **during the programming process** (development phase).

5.2 Core Programming Language

Core Programming Language

To deal with an up-to-date programming language like Java or C++ is out of scope of this introductory lecture. Instead we identify some core programming constructs and abstract away from other syntactic and language-specific variations.

Our programming language is built over

1. integer expressions,
2. boolean expressions, and
3. commands.

Definition 5.1 (Program Variables)

We use $\mathcal{V}ars$ to denote the set of (program) variables. Typically, variables will be denoted by $u, \ldots, x, y, z$ or $x_1, x_2, \ldots$.

Definition 5.2 (Integer Expression)

Let $x \in \mathcal{V}ars$. The set of integer expressions $\mathcal{I}$ is given by all terms generated according to the following grammar:

$$I ::= 0 \mid 1 \mid x \mid (I + I) \mid (I - I) \mid (I \cdot I)$$

We also use $-I$ to stand for $0 - I$.

Although the term "$1-1$" is different from "0", its **meaning (semantics)** is the same. We identify "$1+1+\ldots+1$" with "n" (if the term consist of n 1's). The precise notion of meaning will be given in Definition 5.10.

Integer expressions are **terms** over a set of function symbols $\mathcal{F}unc$, here

$$\mathcal{F}unc = \mathcal{V}ars \cup \{0, 1, +, -, \cdot\}$$

Note also, that we consider elements from $\mathbb{Z}$ as **constants** with their **canonical denotation**! Thus we write "3" instead of "$1+1+1$".

Example 5.3 (Some integer expressions)

$5,\ x,\ 6+(3\cdot x),\ x\cdot y+z-3, -x\cdot x,\ \ldots$

Note that x^x or $y!$ are not integer expressions. **Do we need them?**

Definition 5.4 (Boolean Expression)

The set of **Boolean expressions** $\mathcal{B}$ is given by all formulae generated according to the following grammar:

$$B ::= \Box \mid (\neg B) \mid (B \wedge B) \mid (I < I)$$

We also use $(B \vee B)$ as an abbreviation of $\neg(\neg B \wedge \neg B)$ and $\top$ as an abbreviation for $\neg\Box$. Boolean expressions are **formulae** over the set of relation symbols $\mathcal{P}red = \{\Box, <\}$. So, boolean expressions are similar to **sentential logic formulae**.

Note that we use $I = I'$ to stand for $\neg(I < I') \wedge \neg(I' < I)$. We also write $I \neq I'$ for $\neg(I = I')$.

Formulae over $\mathbb{Z}$

When describing the Hoare calculus, we also need formulae to express (1) what should hold **before** a program is executed, and (2) what should be the case **after** the program has been executed. These formulae are built using the boolean expressions just introduced.

Definition 5.5 (Formulae over $\mathbb{Z}$)

The set of **formulae** $\mathcal{F}ml_{\mathcal{V}ars}$ is given by all formulae generated according to the following grammar:

$$\phi ::= \exists x \phi \mid B \mid (\neg \phi) \mid (\phi \wedge \phi)$$

where B is a boolean expression according to Definition 5.4 and $x \in \mathcal{V}ars$.

We note that "$\forall x \phi$" is just an abbreviation for "$\neg \exists x \neg \Phi$"

Formulae over $\mathbb{Z}$ (2)

We have to give a meaning (semantics) to the syntactic constructs! Our model is given by $\mathbb{Z}$ where $\mathbb{Z}$ is the set of integer numbers. Note that in $\mathbb{Z}$ we have a natural meaning of the constructs $0, 1, -, +, \cdot, <$:

$$\mathcal{Z} = \langle \mathbb{Z}, 0^{\mathbb{Z}}, 1^{\mathbb{Z}}, -^{\mathbb{Z}}, +^{\mathbb{Z}}, \cdot^{\mathbb{Z}}, \leq^{\mathbb{Z}} \rangle.$$

In Chapter 4 (Section 3) we have defined what it means that a formula is true in such a structure:

1. $\forall x \forall y \forall z \, (x \cdot x + y \cdot y + z \cdot z \; > 0)$,
2. $\forall x \exists y \, (y < x)$,
3. $x \cdot x \; < \; x \cdot x \cdot x$,
4. $x + 4 < x + 5$.

Formulae over $\mathbb{Z}$ (3)

Note that when evaluating a formula with free variables, there are two possibilities:

1. Either we need to assign values to these variables, or
2. we add "$\forall$" quantifiers to bind all variables.

The truth of a formula can only be established if the formula is a sentence, i.e. does not contain free variables. Let $\phi(x, y)$ be the formula $\exists z\ x + z < y$ and let $n, n' \in \mathbb{Z}$. Then we denote

- by $\overline{\forall}\phi(x, y)$ the formula $\forall x \forall y(\exists z\ x + z < y)$,
- by $\phi(x, y)[n/x, n'/y]$ the formula where x is replaced by n and y is replaced by n': $\exists z\ n + z < n'$.

Formulae over $\mathbb{Z}$ (4)

How large is this class of formulae over $\mathbb{Z}$? Is it expressive enough?

1. How can we express the function $x!$ or x^x?
2. Which functions are not expressible? Are there any?
3. Is there an algorithm to decide whether a given sentence ϕ (formulae without free variables) holds in $\mathbb{Z}$, i.e. whether $\mathbb{Z} \models \phi$?
4. Attention: everything radically changes, when we do not allow multiplication! Then the resulting theory is **decidable**.

Formulae over $\mathbb{Z}$ (5)

Let us express some functions $f : \mathbb{Z} \to \mathbb{Z}$ that are not available as integer expressions.

Definition 5.6 (Functions Expressible over $\mathbb{Z}$)

A function $f : \mathbb{Z} \to \mathbb{Z}$ is expressible over $\mathbb{Z}$ if there is a formula $\Phi(x, y)$ with x, y as the only free variables such that the following holds for all $z, z' \in \mathbb{Z}$:

$$\mathbb{Z} \models \Phi(x, y)[z/x, z'/y] \text{ iff } f(z) = z'$$

Formulae over $\mathbb{Z}$ (6)

$x!$: Not obvious.

x^x: Not obvious.

Using Gödelization, it can be shown that all naturally occurring functions can be expressed. In fact, all **recursive functions** are expressible. We shall therefore use functions like $x!$ as macros (knowing that they can be expressed as formulae in the language).

Definition 5.7 (Program)

The set $\mathcal{P}rog$ of (while) programs over $\mathcal{V}ars$, $\mathcal{I}$ and $\mathcal{B}$ is given by all well-formed sequences which can be formed according to the following grammar:

$$\text{C} ::= \text{skip} \mid \text{x} := \text{I} \mid \text{C; C} \mid \text{if B } \{\text{C}\} \text{ else } \{\text{C}\} \mid \text{while B } \{\text{C}\}$$

where $B \in \mathcal{B}$, $I \in \mathcal{I}$, and $x \in \mathcal{V}ars$.

We usually write programs in lines (for better readability).

skip: Do nothing.

x := I: Assign I to x.

$C_1; C_2$: Sequential execution: C_2 is executed after C_1 provided that C_1 terminates.

if B $\{C_1\}$ else $\{C_2\}$: If B is true then C_1 is executed otherwise C_2.

while B $\{C\}$: C is executed as long as B is true.

The statement "B is true" is defined in Definition 5.11.

Example 5.8

What does the following program Fac(x) calculate?

```
y := 1;
z := 0;
while z ≠ x {
    z := z + 1;
    y := y · z
}
```

Is this class of programs large or small?

Can we express everything we want?

1. No, because we need better software engineering constructs.
2. Yes, because we have "while" and therefore we can do anything.
3. No, because we can not simulate (emulate) Java or C++.
4. It is already too large, because we assume that we can store arbitrary numbers. This is clearly not true on real computers.
5. Yes, because this class of programs corresponds to the class of deterministic Turing machines. And we cannot aim for more.

While Programs = Turing Machines

Turing Machines

The class of programs we have introduced corresponds exactly to programs of Turing machines. Thus it is an **idealization** (arbitrary numbers can be stored in one cell) and therefore it is much more expressive than any real programming language.
But playing quake requires a lot of coding . . .

- In particular, there is no algorithm to decide whether a given program terminates or not.
- The set of all terminating programs is recursively enumerable, but not recursive.
- Therefore the set of non-terminating programs is not even recursively enumerable.

Meaning of a Program

Definition 5.9 (State)

A **state** s is a mapping

$$s : \mathcal{V}ars \to \mathbb{Z}$$

A state assigns to each variable an integer. The set of all states is denoted by S.

The **semantics of a program** P is a **partial function**

$$[\![\mathrm{P}]\!] : S \to S$$

that describes how a state s changes after executing the program.

Definition 5.10 (Semantics: Integer Expression)

The semantics of integer expressions is defined:

$$\begin{aligned} [\![0]\!]_s &= 0 \\ [\![1]\!]_s &= 1 \\ [\![x]\!]_s &= s(x) \text{ for } \quad x \in \mathcal{V}ars \\ [\![E * E']\!]_s &= [\![E]\!]_s * [\![E']\!]_s \quad \text{for } * \in \{+, -, \cdot\} \end{aligned}$$

Definition 5.11 (Meaning: Boolean Expression)

The semantics of a Boolean expression is given as for sentential logic; that is, we write $\mathbb{Z}, s \models B$ if B is true in $\mathbb{Z}$ wrt to state (valuation) s.

Definition 5.12 (Meaning: Satisfaction $\mathbb{Z}, s \models \phi$)

The semantics of a formula ϕ is defined inductively. We have already defined it in Definition 5.11 for atomic expressions. Arbitrary formulae can also contain the quantifier $\exists$.

$$\mathbb{Z}, s \models \exists x \phi(x) \quad \text{iff} \quad \text{there is a } n \in \mathbb{Z} \text{ such that } \mathbb{Z}, s \models \phi(x)[n/x]$$

1. $\exists x \phi : \ \exists x \ 3x < 4$,
2. $\exists x \phi : \ \exists x \ 3x < 4y$.

Definition 5.13 (Meaning: Program)

1. $[\![\mathsf{skip}]\!](s) = s$
2. $[\![\mathsf{x} := \mathsf{I}]\!](s) = t$ where $t(y) = \begin{cases} s(y) & \text{if } y \neq x \\ [\![I]\!]_s & \text{else.} \end{cases}$
3. $[\![\mathsf{C}_1; \mathsf{C}_2]\!](s) = [\![\mathsf{C}_2]\!]([\![\mathsf{C}_1]\!](s))$
4. $[\![\ \mathsf{if\ B\ \{C_1\}\ else\ \{C_2\}}]\!](s) = \begin{cases} [\![\mathsf{C}_1]\!](s) & \text{if } \mathbb{Z}, s \models B, \\ [\![\mathsf{C}_2]\!](s) & \text{else.} \end{cases}$
5. $[\![\ \mathsf{while\ B\ \{C\}}]\!](s) =$
$\begin{cases} [\![\ \mathsf{while\ B\ \{C\}}]\!]([\![\mathsf{C}]\!](s)) & \text{if } \mathbb{Z}, s \models B, \\ s & \text{else.} \end{cases}$

Note that the recursive definition of the while cases is the reason that $[\![\cdot]\!]$ might be a **partial function**: it is (perhaps) not everywhere defined.

5.3 Hoare Logic

Hoare Triples

How can we prove that a program/method P really does what it is intended to do?

We describe the desired state of the (overall) program before and after the execution of P.

For example, let $\mathsf{P}(x)$ be a program that should return a number whose square is strictly less than x.

Is the correctness of the program ensured when we require that $\mathsf{P}(x) \cdot \mathsf{P}(x) < x$?

Not completely! What if $x < 0$? So, the precondition is also very important!

Definition 5.14 (Hoare Triple)

Let ϕ and ψ be **formulae of** $\mathcal{F}ml_{\mathcal{V}ars}$ and P be a **program**. A Hoare triple is given by

$$\{\phi\}\mathsf{P}\{\psi\}$$

where ϕ is said to be the precondition and ψ the postcondition.

A Hoare triple $\{\phi\}\mathsf{P}\{\psi\}$ is read as follows:

- If the overall program is in a state that satisfies ϕ, then,
- after the execution (and termination) of P the resulting state of the program satisfies ψ.

Let P be the program given above and y be the variable returned. Then the following Hoare triple would be a valid specification:

$$\{x > 0\}\mathsf{P}\{y \cdot y < x\}.$$

Partial and Total Correctness

We already introduced the informal reading of Hoare triples. Now we will be more formal. There are two cases one has to distinguish: Programs that terminate and ones which do not. Accordingly, we have two definitions of correctness:

partially correct: ψ holds after execution of P,
provided that P **terminates**,

totally correct: we require in addition that P terminates.

Definition 5.15 (Partial Correctness)

A triple $\{\phi\}\mathsf{P}\{\psi\}$ is satisfied under partial correctness, written as

$$\models^{\mathsf{p}} \{\phi\}\mathsf{P}\{\psi\},$$

if for each state s

$$\mathbb{Z}, [\![\mathsf{P}]\!](s) \models \psi$$

provided that $\mathbb{Z}, s \models \phi$ and $[\![\mathsf{P}]\!](s)$ are defined.

The following program is always partially correct: while $\top$ $\{\mathsf{x} := 0\}$, **for arbitrary pre and postconditions**.

Definition 5.16 (Total Correctness)

A triple $\{\phi\}\mathsf{P}\{\psi\}$ is satisfied under total correctness, written as

$$\models^{\mathsf{t}} \{\phi\}\mathsf{P}\{\psi\},$$

if for each state s, $[\![\mathsf{P}]\!](s)$ is defined and

$$\mathbb{Z}, [\![\mathsf{P}]\!](s) \models \psi$$

provided that $\mathbb{Z}, s \models \phi$.

The following program is usually **not** totally correct: while $\top$ $\{\mathsf{x} := 0\}$. Why "usually"?

Example 5.17

Consider the following program Succ(x):

```
a := x + 1;
if (a − 1 = 0){
    y := 1
} else{
    y := a
}
```

Under which semantics does the program satisfy $\{\top\}\text{Succ}\{y = x + 1\}$?

Note, that the last program is only one program (and a very silly one) that ensures the condition $\{\top\}\mathsf{Succ}\{y = x + 1\}$. There are many more.

Example 5.18

Recall the program $\mathsf{Fac}(x)$ stated in Example 5.8. Which of the following statements are correct?

- $\models^t \{x \geq 0\}\mathsf{Fac}\{y = x!\}$,
- $\models^t \{\top\}\mathsf{Fac}\{y = x!\}$,
- $\models^p \{x \geq 0\}\mathsf{Fac}\{y = x!\}$ and
- $\models^p \{\top\}\mathsf{Fac}\{y = x!\}$.

Program and Logical Variables

The pre- and postconditions in a Hoare triple may contain two kinds of variables:

- program variables and
- logical variables.

Given a program P the former kind occurs in the program whereas the latter refers to fresh variables.

The following example makes clear why we need logical variables.

Example 5.19

The following program Fac2 works as well.

```
y := 1;
while (x ≠ 0){
    y := y · x;
    x := x − 1
}
```

Why is it not a good idea to use the Hoare triple $\{x \geq 0\}\text{Fac2}\{y = x!\}$ in this case?

What about $\{x = x_0 \wedge x \geq 0\}\text{Fac2}\{y = x_0!\}$? The variable x_0 is a **logical variable**!

5.4 Proof Calculi: Partial Correctness

Proof Rules

We have introduced a semantic notion of (partial and total) correctness. As for resolution we are after syntactic versions ($\vdash^t$ and $\vdash^p$) which can be used on computers.

Sound Calculus: Can we define a calculus, that allows us to derive **only valid Hoare triples**?

Complete Calculus: Can we define a calculus, that generates all valid Hoare triples?

Answer to question 1: Yes, for both versions.

Answer to question 2: No, not even for $\vdash^p$:

$\vdash^p \{\top\}\, P\, \{\square\}$ *if and only if* P **is not terminating.**

And this set is **not recursively enumerable**.

The following rules were proposed by R. Floyd and C.A.R. Hoare. A rule for each basic program construct is presented.

If a program is correct we may be able to show it by only applying the following rules (compare with Modus Ponens).

Composition

Definition 5.20 (Composition Rule (comp))

$$\frac{\{\phi\}\mathsf{C}_1\{\eta\} \qquad \{\eta\}\mathsf{C}_2\{\psi\}}{\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}}$$

In order to prove that $\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}$ we have to prove the Hoare triples $\{\phi\}\mathsf{C}_1\{\eta\}$ and $\{\eta\}\mathsf{C}_2\{\psi\}$ for some appropriate η.

An axiom

Definition 5.21 (Assignment Rule (assign))

$$\frac{}{\{\psi[I/x]\}\mathsf{x} := \mathsf{I}\{\psi\}}$$

The rule is self-explanatory. Recall that $\psi[I/x]$ denotes the formula that is equal to ψ but each free occurrence of x is replaced by I.

Another axiom

Definition 5.22 (Skip Rule (skip))

$$\frac{}{\{\phi\}\text{skip}\{\phi\}}$$

If Rule

Definition 5.23 (If Rule (if))

$$\frac{\{\phi \wedge B\}\mathsf{C}_1\{\psi\} \qquad \{\phi \wedge \neg B\}\mathsf{C}_2\{\psi\}}{\{\phi\}\ \mathsf{if\ B\ \{C_1\}\ else\ \{C_2\}}\{\psi\}}$$

In order to prove the conclusion we prove that ψ holds for **both** possible program executions of the if-rule: when B holds and when it does not.

Partial-While Rule

Definition 5.24 (Partial-While Rule (while))

$$\frac{\{\psi \wedge B\}\mathsf{C}\{\psi\}}{\{\psi\}\ \mathsf{while\ B\ \{C\}}\{\psi \wedge \neg B\}}$$

The while-rule is the most sophisticated piece of code; it may allow infinite looping. The formula ψ in the premise plays a decisive rule: ψ is true **before** and **after** the execution of C, i.e. C does not change the truth value of ψ.

Invariant

Definition 5.25 (Invariant)

An invariant of the while-statement while B {C} is any formula ψ such that $\models^p \{\psi \wedge B\}\mathsf{C}\{\psi\}$.

The conclusion says that ψ does not change, even when C is **executed several times** and if C terminates then B is false.

Example 5.26

What is an invariant of the following program?

```
y := 1;
z := 0;
while z ≠ x {
    z := z + 1;
    y := y · z
}
```

An invariant is "$y = z!$".

Implication Rule

Definition 5.27 (Implication Rule (impl))

$$\frac{\{\phi\}\mathsf{C}\{\psi\}}{\{\phi'\}\mathsf{C}\{\psi'\}}$$

whenever $\mathbb{Z} \models \overline{\forall}(\phi' \to \phi)$ and $\mathbb{Z} \models \overline{\forall}(\psi \to \psi')$.

Implication Rule (2)

The implication rule allows us to **strengthen** the precondition ϕ to ϕ' and to **weaken** the postcondition ψ to ψ', provided that the two implications hold.

Note, that this rule links program logic with the truths of formulae in $\mathbb{Z}$, which have to be established. We call them proof obligations.

An Example

We will try to prove the correctness of the program Fac given in Example 5.8. That is, we would like to derive

$$\{\top\}\text{Fac}\{y = x!\}$$

For any input state after the execution of Fac the return value y should have the value $x!$.

We have to start with axioms (by the assignment rule):

$$\frac{\dfrac{\{1=1\}\mathsf{y:=1}\{y=1\}}{\{\top\}\mathsf{y:=1}\{y=1\}}(impl) \qquad \dfrac{\{y=1\wedge 0=0\}\mathsf{z:=0}\{y=1\wedge z=0\}}{\{y=1\}\mathsf{z:=0}\{y=1\wedge z=0\}}(impl)}{\underbrace{\{\top\}\mathsf{y := 1; z := 0}\{y = 1 \wedge z = 0\}}_{\psi_1}}(comp)$$

Again, on top are axioms:

$$\frac{\dfrac{\dfrac{\{y\cdot(z+1)=(z+1)!\}\mathsf{z:=z+1}\{y\cdot z=z!\}}{\{y=z!\wedge z\neq x\}\mathsf{z:=z+1}\{y\cdot z=z!\}}(impl) \qquad \{y\cdot z=z!\}\mathsf{y:=y\cdot z}\{y=z!\}}{\{y=z!\wedge z\neq x\}\mathsf{z:=z+1;y:=y\cdot z}\{y=z!\}}(comp)}{\underbrace{\{y = z!\}\ \mathsf{while\ z \neq x\ \{z := z + 1; y := y \cdot z\}}\{y = z! \wedge z = x\}}_{\psi_2}}(while)$$

Putting both parts together:

$$\frac{\psi_1 \qquad \dfrac{\psi_2}{\{y=1\wedge z=0\}\ \mathsf{while\ z\neq x\ \{z:=z+1;y:=y\cdot z\}}\{y=x!\}}(impl)}{\{\top\}\mathsf{Fac}\{y = x!\}}(comp)$$

Proof obligations

We have to show that the following formulae hold in $\mathbb{Z}$:

1. $\top \to 1 = 1$,
2. $\forall y(y = 1 \to y = 1 \wedge 0 = 0)$,
3. $\forall x \forall y \forall z(y = z! \wedge z \neq x \to (y(z + 1) = (z + 1)!))$,
4. $\forall y \forall z(y = 1 \wedge z = 0 \to y = z!)$,
5. $\forall x \forall y \forall z(y = z! \wedge z = x \to y = x!)$.

Annotation Calculus

- The proof rules just presented are very similar to Hilbert style calculus rules.
- They inherit some undesirable properties: The proof calculus is not “easy to use”.
- The proof given for the small Fac-program looks already quite complicated.

In this section we present an annotation calculus which is much more convenient for practical purposes.

We consider a program P as a sequence of **basic commands**:

$$C_1;\ C_2; \ldots\ ; C_n$$

That is, none of the commands C_i is directly composed of smaller programs by means of composition. Assume we intend to show that

$$\vdash^p \{\phi\}P\{\psi\}$$

In the annotation calculus we try to find **appropriate** ϕ_i, $i = 0, \ldots, n$ such that

$$\textbf{if } \vdash^p \{\phi_i\}C_i\{\phi_{i+1}\} \text{ for all } i = 0, \ldots, n-1$$
$$\textbf{then also } \vdash^p \{\phi\}P\{\psi\}.$$

In other words, we **interleave** the program P with **intermediate conditions** ϕ_i

$$\{\phi_0\}C_1\{\phi_1\}\ C_2\{\phi_2\}\ldots\{\phi_{n-1}\}C_n\{\phi_n\}$$

where each step $\{\phi_i\}C_i\{\phi_{i+1}\}$ is justified by one of the proof rules given below.

That is, an annotation calculus is a way of summarizing the application of the proof rules to a program.

How to find the appropriate ϕ_i?

We determine "something like" the **weakest preconditions** successively such that

$$\vdash^p \{\phi'_0\}\mathsf{C}_1\{\phi'_1\}\ \mathsf{C}_2\{\phi'_2\}\ldots\{\phi'_{n-1}\}\mathsf{C}_n\{\phi'_n\}$$

Under which condition would this guarantee $\vdash^p \{\phi_0\}\mathsf{P}\{\phi_n\}$?

It must be the case that $\mathbb{Z} \models \overline{\forall}\,(\phi_0 \to \phi'_0)$ and $\mathbb{Z} \models \overline{\forall}\,(\phi'_n \to \phi_n)$

Why do we say "something like" the weakest precondition? We come back to this point later.

In the following we label the program P with preconditions by means of rewrite rules

$$\frac{X}{Y}$$

Such a rule denotes that X can be rewritten (or replaced) by Y.

Each rewrite rule results from a proof rule.

Definition 5.28 (Assignment Rule)

$$\frac{\mathsf{x} := \mathsf{E}\{\psi\}}{\{\psi[E/x]\}\mathsf{x} := \mathsf{E}\{\psi\}}$$

Definition 5.29 (If Rule (1))

$$\frac{\mathsf{if\ B}\ \{\mathsf{C}_1\}\ \mathsf{else}\ \{\mathsf{C}_2\}\{\psi\}}{\mathsf{if\ B}\ \{\mathsf{C}_1\{\psi\}\}\ \mathsf{else}\ \{\mathsf{C}_2\{\psi\}\}\{\psi\}}$$

Definition 5.30 (If Rule (2))

$$\frac{\mathsf{if\ B}\ \{\{\phi_1\}\cdots\}\ \mathsf{else}\ \{\{\phi_2\}\cdots\}}{\{(B \to \phi_1) \wedge (\neg B \to \phi_2)\}\ \mathsf{if\ B}\ \{\{\phi_1\}\cdots\}\ \mathsf{else}\ \{\{\phi_2\}\cdots\}}$$

Definition 5.31 (Partial-While Rule)

$$\frac{\text{while B } \{P\}}{\{\phi\}\ \text{while B } \{\{\phi \wedge \text{B}\}P\{\phi\}\}\{\phi \wedge \neg B\}}$$

where ϕ is an invariant of the while-loop.

Definition 5.32 (Skip Rule)

$$\frac{\text{skip}\{\phi\}}{\{\phi\}\text{skip}\{\phi\}}$$

Applying the rules just introduced, we end up with a finite sequence

$$s_1 s_2 \dots s_m$$

where each s_i is of the form $\{\phi\}$ or it is a command of a program (which in the case of if or while commands can also contain such a sequence).

It can also happen that two subsequent elements have the same form: $\dots\{\phi\}\{\psi\}\dots$. Whenever this occurs, we have to show that ϕ implies ψ: a proof obligation (see Slide 425).

Definition 5.33 (Implied Rule)

Whenever applying the rules lead to a situation where two formulae stand next to each other $\dots\{\phi\}\{\psi\}\dots$, then we add a proof obligation of the form

$$\mathbb{Z} \models \overline{\forall}\,(\phi \to \psi).$$

Consider $\{\phi\}$ while B $\{$P$\}\{\psi\}$. Then the while-rule yields the following construct:

$$\{\phi\}\{\eta\} \text{ while B } \{\{\eta \wedge \text{B}\}\text{P}\{\eta\}\}\{\eta \wedge \neg B\}\{\psi\}$$

That is, we have to show that the invariant η satisfies the following properties:

1. $\mathbb{Z} \models \overline{\forall}\,(\phi \to \eta)$,
2. $\mathbb{Z} \models \overline{\forall}\,((\eta \wedge \neg B) \to \psi)$,
3. $\vdash^p \{\eta\}$ while B $\{$P$\}\{\eta \wedge \neg B\}$, and
4. $\vdash^p \{\eta \wedge B\}P\{\eta\}$ (this is the fact that it is an invariant).

Example 5.34

Prove: $\models^p \{y = 5\}\mathsf{x := y + 1}\{x = 6\}$

1. $\mathsf{x := y + 1}\{x = 6\}$
2. $\{y + 1 = 6\}\mathsf{x := y + 1}\{x = 6\}$ (Assignment)
3. $\{y = 5\}\{y + 1 = 6\}\mathsf{x := y + 1}\{x = 6\}$ (Implied)

Example 5.35

Prove: $\models^p \{y < 3\}\mathsf{y := y + 1}\{y < 4\}$

1. $\mathsf{y := y + 1}\{y < 4\}$
2. $\{y + 1 < 4\}\mathsf{y := y + 1}\{y < 4\}$ (Assignment)
3. $\{y < 3\}\{y + 1 < 4\}\mathsf{y := y + 1}\{y < 4\}$ (Implied)

Example 5.36

$\{y = y_0\}$

z := 0;

while y $\neq$ 0 {

    z := z + x;

    y := y − 1

}

$\{z = xy_0\}$

Firstly, we have to use the rules of the annotation calculus.

$\{y = y_0\}$
$\{I[0/z]\}$
z := 0;
$\{I\}$
while y $\neq$ 0 {
    $\{I \wedge y \neq 0\}$
    $\{I[y-1/y][z+x/z]\}$
    z := z + x;
    $\{I[y-1/y]\}$
    y := y − 1
    $\{I\}$
}
$\{I \wedge y = 0\}$
$\{z = xy_0\}$

What is a suitable invariant?

We have to choose an invariant such that the following hold in $\mathbb{Z}$:

1. $y = y_0 \to I[0/z]$,
2. $I \wedge y \neq 0 \to I[y - 1/y][z + x/z]$,
3. $I \wedge y = 0 \to z = xy_0$.

What about $I : \top$? What about $I : (z + xy = xy_0)$?

It is easy to see that the latter invariant satisfies all three conditions. This proves the partial correctness of $\{y = y_0\}\mathsf{P}\{z = xy_0\}$.

How to ensure that $\models^p \{\phi\}P\{\psi\}$?

Definition 5.37 (Valid Annotation)

A valid annotation of $\{\phi\}P\{\psi\}$ is given if

1. only the rules of the **annotation calculus** are used;
2. each command in P is **embraced by a post and a precondition**;
3. the assignment rule is applied **to each** assignment;
4. each proof obligation introduced by the implied rule has to be verified.

Proposition 5.38 ($\vdash^p$ *if and only if* Valid Annotation)

Constructing valid annotations *is equivalent to* deriving the Hoare triple $\{\phi\}\mathsf{P}\{\psi\}$ *in the Hoare calculus:*

$$\vdash^p \{\phi\}\mathsf{P}\{\psi\}$$
if and only if
there is a valid annotation *of* $\{\phi\}\mathsf{P}\{\psi\}$.

Note, this does **not** mean that the calculus is complete!

Exercise

Let the program P be:

z := x;

z := z + y;

u := z

Use the annotation calculus to prove that
$\vdash^p \{\top\}\mathsf{P}\{u = x + y\}$.

Exercise

Given the program P:

$$\mathsf{a} := \mathsf{x} + 1;$$
$$\text{if } \mathsf{a} - 1 = 0 \ \{\mathsf{y} := 1\} \text{ else } \{\mathsf{y} := \mathsf{a}\}$$

Use the annotation calculus to prove that $\vdash^p \{\top\}\mathsf{P}\{y = x + 1\}$.

Exercise

Given the program Sum:

```
z := 0;
while x > 0 {
    z := z + x;
    x := x − 1
}
```

Use the annotation calculus to prove that
$\vdash^p \{x = x_0 \wedge x \geq 0\}\text{Sum}\{z = \frac{x_0(x_0+1)}{2}\}$.

Weakest Liberal Precondition

In principle, the **annotation calculus** determines the weakest precondition of a program and checks whether the given precondition implies the calculated precondition. I.e. when we start with something of the form

$$\mathsf{P}\{\psi\}$$

then the annotation calculus leads to a Hoare triple

$$\{\phi\}\mathsf{P}'\{\psi\}$$

where ϕ is "something like the weakest precondition":

$$\vdash^p \{\phi\}\mathsf{P}\{\psi\}.$$

Without the while-command the annotation calculus does calculate the weakest precondition! But in the rule for the while-command **some** invariant is selected. This does not ensure that the **weakest** precondition is determined!

A formula ψ is said to be weaker than χ if the following holds: $\mathbb{Z} \models \overline{\forall}(\chi \to \psi)$.

Theorem 5.39 (Weakest Liberal Precondition)

The weakest liberal precondition of a program P and postcondition ϕ, denoted by $\mathbf{wp}(\mathsf{P}, \phi)$, is the weakest formula ψ such that $\models^p \{\psi\}\mathsf{P}\{\phi\}$. Such a formula exists and can be constructed as a formula in our language.

The reason for the last theorem is that the model $\mathbb{Z}$ with $+, \cdot, <$ is powerful enough to express all notions we need.

The weakest liberal precondition for each program is constructed inductively:

Proposition 5.40 (wp- Partial Correctness)

$$
\begin{array}{rll}
\mathbf{wp}(\mathsf{x} := \mathsf{I}, \phi) & \textit{if and only if} & \phi[x/I] \\
\mathbf{wp}(\mathsf{P}_1; \mathsf{P}_2, \phi) & \textit{if and only if} & \mathbf{wp}(\mathsf{P}_1, \mathbf{wp}(\mathsf{P}_2, \phi)) \\
\mathbf{wp}(\textit{ if } \mathsf{B} \ \{\mathsf{P}_1\} \textit{ else } \{\mathsf{P}_2\}) & \textit{if and only if} & (B \to \mathbf{wp}(\mathsf{P}_1, \phi)) \wedge \\
 & & (\neg B \to \mathbf{wp}(\mathsf{P}_2, \phi)) \\
\mathbf{wp}(\textit{ while } \mathsf{B} \ \{\mathsf{P}\}, \phi) & \textit{if and only if} & \forall i \in \mathbb{N}_0 \ (L_i(B, \mathsf{P}, \phi))
\end{array}
$$

where

$$
\begin{array}{rll}
L_0 & (B, P, \phi) & := \top \\
L_{i+1} & (B, P, \phi) & := (\neg B \to \phi) \wedge (B \to \mathbf{wp}(P, L_i(B, P, \phi))
\end{array}
$$

Proposition 5.41

The $L_i(B, \mathsf{P}, \phi)$ *are* **monotonically increasing***:*
$\forall i \in \mathbb{N}_0 \, (L_{i+1}(B, \mathsf{P}, \phi) \rightarrow L_i(B, \mathsf{P}, \phi))$.

5.5 Proof Calculi: Total Correctness

Proof Rules

We extend the proof calculus for partial correctness presented to one that covers **total correctness**.

Partial correctness does not say anything about the termination of programs. It is easy to see that only the **while** construct can cause the nontermination of a program. Hence, in addition to the partial correctness calculus we have to prove that a while loop terminates.

The proof of termination of while statements follows the following schema:

- Given a program P, identify an integer expression I whose value decreases after performing P but which is always non-negative.

Such an expression E is called a variant.
Obviously: a while loop terminates if such a variant exists. The corresponding rule is given below.

Definition 5.42 (Total-While Rule)

$$\frac{\{\phi \wedge B \wedge 0 \leq E = E_0\}\mathsf{C}\{\phi \wedge 0 \leq E < E_0\}}{\{\phi \wedge 0 \leq E\}\ \mathsf{while\ B\ \{C\}}\{\phi \wedge \neg B\}}$$

where

- ϕ is an **invariant** of the while-loop,
- E is a **variant** of the while-loop,
- E_0 represents the **initial value** of E before the loop.

Example 5.43

Let us consider the program Fac once more:

$$\begin{array}{l} \mathsf{y} := 1; \\ \mathsf{z} := 0; \\ \quad \text{while } (\mathsf{z} \neq \mathsf{x}) \ \{\mathsf{z} := \mathsf{z} + 1;\ \mathsf{y} := \mathsf{y} \cdot \mathsf{z}\} \end{array}$$

How does a variant for the proof of $\{x \geq 0\}\mathsf{Fac}\{y = x!\}$ look like?

A possible variant is "$x - z$". The first time the loop is entered we have that $x - z = x$ and then the expression decreases step by step until $x - z = 0$.

Annotation Calculus

Definition 5.44 (Total-While Rule)

$$\frac{\text{while B } \{P\}}{\{\phi \wedge 0 \leq E\}\ \text{while B}\ \{\{\phi \wedge \mathsf{B} \wedge 0 \leq \mathsf{E} = \mathsf{E}_0\}\mathsf{P}\{\phi \wedge 0 \leq \mathsf{E} < \mathsf{E}_0\}\}\{\phi \wedge \neg B\}}$$

where

- ϕ is an **invariant** of the while-loop,
- E is a **variant** of the while-looping,
- E_0 represents the **initial value** of E before the loop.

Example 5.45

Prove that $\models^t \{x \geq 0\}\mathsf{Fac}\{y = x!\}$. What do we have to do at the beginning?

We have to determine a suitable **variant** and an **invariant**. As variant we may choose $x - z$ and as invariant "$y = z!$". Now we can apply the annotation calculus.

$\{x \geq 0\}$
$\{1 = 0! \wedge 0 \leq x - 0\}$
y := 1;
$\{y = 0! \wedge 0 \leq x - 0\}$
z := 0;
$\{y = z! \wedge 0 \leq x - z\}$
while x $\neq$ z {
$\{y = z! \wedge x \neq z \wedge 0 \leq x - z = E_0\}$
$\{y(z+1) = (z+1)! \wedge 0 \leq x - (z+1) < E_0\}$
z := z + 1;
$\{yz = z! \wedge 0 \leq x - z < E_0\}$
y := y · z
$\{y = z! \wedge 0 \leq x - z < E_0\}$ }
$\{y = z! \wedge x = z\}$
$\{y = x!\}$

Weakest Precondition

Proposition 5.46 (wp- Total Correctness)

The weakest precondition for total correctness exists and can be expressed as a formula in our language.

Example 5.47

What is the weakest precondition for

$$\text{while i} < \text{n } \{\text{i} := \text{i} + 1\}\{i = n + 5\}?$$

$i = n + 5.$

5.6 Sound and Completeness

Theoretical Aspects

Finally, we consider some theoretical aspects with respect to the sound and completeness of the introduced calculi.

Recall, that a calculus is **sound** if everything that can be derived is also semantically true:

$$\text{If } \vdash^x \{\phi\}\mathsf{P}\{\psi\} \text{ then } \models^x \{\phi\}\mathsf{P}\{\psi\}$$

where $x \in \{p, t\}$ stands for partial/total correctness.

The reverse direction is referred to as completeness.

A calculus is complete if everything that can semantically be derived is also derivable by the calculus:

$$\text{If } \models^x \{\phi\}\mathsf{P}\{\psi\} \text{ then } \vdash^x \{\phi\}\mathsf{P}\{\psi\}$$

where $x \in \{p, t\}$ stands for partial/total correctness.

Which direction is more difficult to prove?

For the soundness we just have to make sure that all proof rules introduced make sense. That is, given a rule $\frac{X}{Y}$ it has to be shown that Y holds whenever X holds.

Theorem 5.48

The Hoare calculus is **sound**.

Proof.

Exercise! □

Theorem 5.49

The Hoare calculus is complete.

- The Hoare calculus contains axioms and rules that require to determine whether certain formulae (**proof obligations**) are true in $\mathbb{Z}$.
- The theory of $\mathbb{Z}$ is undecidable.
- Thus any re axiomatization of $\mathbb{Z}$ is incomplete.
- However, most proof obligations occurring in practice can be checked by theorem provers.

6. From FOL to **PROLOG**

Content of this chapter:

Resolution: We extend our resolution calculus for SL to one for FOL. We need (2) one more rule, factorization to deal with terms, and (2) the concept of unification. Our calculus is refutation complete for arbitrary clauses.

Completeness: We extend our SL Hilbert calculus to one for FOL. Such a calculus is still **recursively enumerable**, but not recursive: FOL is undecidable.

Content of this chapter (2):

PROLOG: While we introduce **PROLOG** in the next chapter, the last two sections of this chapter prepare the ground for introducing **PROLOG**. We discuss Herbrand models and input and linear resolution.

SLD resolution: If we restrict the class of formulae, we get a more efficient calculus: SLD resolution. This directly leads to **PROLOG** as a programming language.

6.1 Motivation

Computational complexity of the problem

- Propositional logic: co-NP-complete.
- FOL: **recursively enumerable (re)**.
- Higher-order logic: Not even re.

Way out of the complexity

- **Interactive** theorem provers require a **human** to give hints to the system.
- The interaction may be at a very detailed level, where the user guides the inferences made by the system, or at a much higher level where the user determines intermediate lemmas to be proved on the way to the proof of a conjecture.
- Often: the user defines a number of proof strategies $\rightsquigarrow$ proving toolbox (e.g., Isabelle).

Popular Theorem proving techniques

- First-order resolution with unification,
- Higher-order unification,
- Model elimination,
- Superposition and term rewriting,
- Lean theorem proving,
- Method of analytic tableaux,
- Mathematical induction,
- DPLL (Davis-Putnam-Logemann-Loveland algorithm).

All the previously mentioned complexity bounds still apply!

6.2 A calculus for FOL

We extend our calculus for SL from Definition 2.19

Definition 6.1 (Calculus for FOL: Axioms)

We define $\mathrm{Hilbert}^{\mathrm{FOL}}_{\mathcal{L}} = \langle \{\neg\varphi \vee \varphi, \neg\Box\}, \mathbf{Inf}\rangle$, a Hilbert-Type calculus for FOL. We keep the axiom schema $\neg\varphi \vee \varphi$ and $\neg\Box$ and add the following ones:

Substitution: $\varphi[t/x] \to \exists x\varphi$, if t can be substituted in φ,

Equality:

- $x \doteq x$,
- $x_1 \doteq y_1 \wedge \ldots \wedge x_m \doteq y_m \to f(x_1, \ldots, x_m) \doteq f(y_1, \ldots, y_m)$, for all variables and function symbols of matching arities,
- $x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n \to (R(x_1, \ldots, x_n) \to R(y_1, \ldots, y_n))$, for all variables and relation symbols of matching arities (in particular for $\doteq$).

(ϕ, ψ, χ stand for arbitrarily complex formulae (not just constants)—they represent schemata, rather than particular formulae in the language)

Definition 6.2 (Calculus for FOL: Rules)

As for SL, we keep the four schemata expansion, associativity, shortening and cut. In addition, we add the following rule schema

∃-Introduction: $\frac{\varphi \to \psi}{\exists x \varphi \to \psi}$, if x does not occur free in ψ.

(ϕ, ψ, χ stand for arbitrarily complex formulae (not just constants)— they represent schemata, rather than particular formulae in the language)

The notion of a proof is literally the same as in SL (Definition 2.20 on Slide 126). We denote the resulting relation with $\vdash_{\mathsf{FOL}}$.

Completeness for FOL

In much the same way as for SL, but more cumbersome and with much more formal effort, the following can be proved

Theorem 6.3 (Correct-, Completeness for $\mathrm{Hilbert}^{\mathrm{FOL}}_{\mathcal{L}(\mathcal{P}rop)}$)

A formula follows semantically from a theory T if and only if it can be derived:

$$T \models \varphi \text{ if and only if } T \vdash_{FOL} \varphi$$

We are not giving the proof in this lecture, as we concentrate on the resolution calculus for FOL.

6.3 Resolution

Unifiers

- Applying resolution in SL requires to find two clauses so that one contains p and the other $\neg$p.
- In FOL we have a more difficult situation. Consider pred1$(f(x), y) \vee$ pred2$(g(x))$ and $\neg$pred1$(f(c), g(c))$.
- The predicate $\neg$pred1$(f(c), g(c))$ **is not exactly the negation** of pred1$(f(x), y)$.

Unifiers (cont.)

- But in FOL we have more freedom: perhaps we can find substitutions for the free variables, such that we can make one predicate exactly the negation of the other?
- $[c/x, g(c)/y]$ is a substitution that we can use. Resolving with this substitution we get: pred2$(g(c))$.
- The technique is called unification. The substitutions are called unifiers.

Unifiers

Example 6.4

We consider a more difficult situation:
$\mathsf{p}(g(x,c),x) \vee \mathsf{q}(y)$ and $\neg\mathsf{p}(g(f(h(y)),y),d)$.

- Is it possible to resolve these two clauses?
- What if we replace the last one by $\neg\mathsf{p}(g(f(h(y)),y),z)$?
- **How many** unifiers could there be?

Is there one unifier that represents all unifiers?

Most general unifier (1)

- Can we resolve $\mathsf{p}(x, a)$ with $\neg\mathsf{q}(y, b)$?
- Or $\mathsf{p}(g(a), f(x))$ with $\neg\mathsf{p}(g(y), z)$?
  Basic substitutions are:
  $[a/y, a/x, f(a)/z]$, $[a/y, f(a)/x, f(f(a))/z]$, and many more. The following substitution is the most interesting

$$[a/y, f(x)/z]$$

  It contains all other substitutions as special cases.
- This is no coincidence: it is a most general unifier.

Most general unifier (2)

Definition 6.5 (Most general unifier: mgU)

We consider a finite set of equations between terms or **equations between literals**.
Then there is an algorithm which calculates a most general solution substitution (i.e. a substitution of the involved variables so that the left sides of all equations are syntactically identical to the right sides) if one exists, or which returns *fail*, if such a substitution does not exist.
In the first case the **most general solution substitution** is defined (up to renaming of variables): it is called

mgU, most general unifier

Most general unifier (3)

Given: $f(x, g(h(y), y)) = f(x, g(z, a))$

The algorithm successively calculates the following sets of equations:

$$\{\ x = x,\ g(h(y), y) = g(z, a)\ \}$$
$$\{\ g(h(y), y) = g(z, a)\ \}$$
$$\{\ h(y) = z,\ y = a\ \}$$
$$\{\ z = h(y),\ y = a\ \}$$
$$\{\ z = h(a),\ y = a\ \}$$

Thus the mgU is: $[x/x, a/y, h(a)/z]$.

The occurs-check

Given: $f(x, g(x)) = f(c, c)$

Is there an mgU?

The algorithm gives the following:

$$\{\, x = c,\ g(x) = c \,\}$$

- But setting $x = c$ is not a unifying substitution, because $c \neq g(c)$.
- Therefore there is no mgU. And the algorithm has to do this check, called occurs check, to test whether the substitution is really correct.
- However, this check is computationally expensive and many algorithms **do not do it**.

Clauses in FOL

- We have already introduced the notion of **clauses** in Definition 2.33 on Slide 146.
- But now we are faced with variables and quantifiers. However, we first use Proposition 4.22 to transform any FOL-formula into prenex form without existential quantifiers: $\forall_1 x_1 \ldots \forall_n x_n \psi$. Then we apply Theorem 2.10 to ψ and transform it to a clause.
- The resulting formula, where we usually remove all quantifiers for better reading (all variables are universally quantified), is called a clause in FOL.
- We note that, during skolemization, we have extended the language. But for our calculus that does not matter. We have just additional skolem functions.

A resolution calculus for FOL

The resolution calculus is defined over the language $\mathcal{L}^{clausal} \subseteq \mathcal{L}_{FOL}$ where the set of well-formed formulae $\text{Fml}_{\mathcal{L}^{clausal}}$ consists of all disjunctions of the following form

$$A \vee \neg B \vee C \vee \ldots \vee \neg E,$$

i.e. the disjuncts are only atoms or their negations (containing variables). **No implications, conjunctions, double negations, or quantifiers are allowed.** These formulae are also called clauses.
Such a clause is also written as the set

$$\{A, \neg B, C, \ldots, \neg E\}.$$

The union of such sets corresponds again to a clause. The empty clause is represented by $\square$.

A clause consists of FOL-**atoms** (or their negations) rather than propositional constants (as in SL).

Definition 6.6 (Resolution and factorization for $\mathcal{L}^{clausal}$)

The resolution calculus operates on **clauses** and consists of the following two rules: resolution and factorization.

$$\textbf{(Res)}\frac{C_1 \cup \{A_1\} \;\; C_2 \cup \{\neg A_2\}}{(C_1 \cup C_2)mgU(A_1, A_2)}$$

where $C_1 \cup \{A_1\}$ and $C_2 \cup \{A_2\}$ are assumed to be disjunct wrt the variables,

$$\textbf{(Fac)}\;\frac{C_1 \cup \{L_1, L_2\}}{(C_1 \cup \{L_1\})mgU(L_1, L_2)}$$

Illustration of the resolution rule

Example 6.7

Consider the set

$$M = \{\mathsf{r}(x) \vee \neg\mathsf{p}(x), \mathsf{p}(a), \mathsf{s}(a)\}$$

and the question

$$M \models \exists x(\mathsf{s}(x) \wedge \mathsf{r}(x))?$$

Definition 6.8 (Resolution Calculus for FOL)

We define the resolution calculus $\mathrm{Robinson}^{\mathrm{FOL}}_{\mathcal{L}^{clausal}} = \langle \emptyset, \{\mathbf{Res}, \mathbf{Fac}\}\rangle$ as follows. The underlying language is $\mathcal{L}^{clausal} \subseteq \mathcal{L}_{\mathrm{FOL}}$ defined on Slide 486 together with the set of well-formed formulae $\mathrm{Fml}_{\mathcal{L}^{clausal}}$.

Thus there are **no axioms** and only **two inference rules**. The well-formed formulae are just clauses.

Question:

Is this calculus correct and complete for $\mathrm{Fml}_{\mathcal{L}^{clausal}}$?

Question:

Why do we need factorization?

Answer:

Consider

$$M = \{\{\mathsf{s}(x_1), \mathsf{s}(x_2)\}, \{\neg\mathsf{s}(y_1), \neg\mathsf{s}(y_2)\}\}$$

Resolving both clauses gives

$$\{\mathsf{s}(x_1)\} \cup \{\neg\mathsf{s}(y_1)\}$$

or variants of it.
Resolving this new clause with one in M only leads to variants of the respective clause in M.

Answer (continued):

$\square$ can not be derived (using resolution only).

Factorization solves the problem, we can deduce both $\mathsf{s}(x)$ and $\neg\mathsf{s}(y)$, and from there the empty clause $\square$.

Theorem 6.9 (Resolution is refutation complete)

Robinsons resolution calculus $\mathrm{Robinson}^{\mathrm{FOL}}_{\mathcal{L}^{clausal}}$ *is* refutation complete*: Given an unsatisfiable set, the empty clause can eventually be derived using resolution and factorization.*

Proof.

We prove the contraposition: Assume a set of clauses is consistent (i.e. $\square$ cannot be derived). Then there is a model of it.

$\square$

6.4 Herbrand

The relation $T \models \phi$ states that **each model** of T is also a model of ϕ. But because there are many models with very large universes the following question comes up: **can we restrict to particular models ?**

Theorem 6.10 (Löwenheim-Skolem)

$T \models \phi$ holds if and only if ϕ holds in all countable models of T.

By countable we mean the size of the universe of the model.

Quite often the universes of models (which we are interested in) consist exactly of the basic terms $Term_{\mathcal{L}}(\emptyset)$. This leads to the following notion:

Definition 6.11 (Herbrand model)

A model $\mathcal{A}$ is called Herbrand model with respect to a language if the universe of $\mathcal{A}$ consists exactly of $Term_{\mathcal{L}}(\emptyset)$ and the function symbols f_i^k are interpreted as follows:

$$f_i^{k\mathcal{A}}: \quad Term_{\mathcal{L}}(\emptyset) \times \ldots \times Term_{\mathcal{L}}(\emptyset) \to Term_{\mathcal{L}}(\emptyset);$$
$$(t_1, \ldots, t_k) \mapsto f_i^k(t_1, \ldots, t_k)$$

We write $T \models_{\text{Herb}} \phi$ if each Herbrand model of T is also a model of ϕ.

So the freedom in defining a Herbrand model is only in the interpretation of the predicates: the function symbols are fixed.

Theorem 6.12 (Reduction to Herbrand models)

If T is universal and ϕ existential, then the following holds:

$$T \models \phi \text{ if and only if } T \models_{Herb} \phi$$

Question:

Is $T \models_{\text{Herb}} \phi$ not much easier, because we have to consider only Herbrand models? Is it perhaps decidable?

No, truth in Herbrand models is highly undecidable.

The following theorem is the basic result for applying resolution. In a way it states that FOL **can be somehow reduced to SL**.

Theorem 6.13 (Herbrand)

Let T be universal and ϕ without quantifiers. Then:

$T \models \exists x \phi(x)$ *if and only if* there are $t_1, \ldots, t_n \in Term_{\mathcal{L}}(\emptyset)$ with: $T \models \phi(t_1) \vee \ldots \vee \phi(t_n)$

Or: Let M be a set of clauses of FOL (formulae in the form $P_1(t_1) \vee \neg P_2(t_2) \vee \ldots \vee P_n(t_n)$ with $t_i \in Term_{\mathcal{L}}(X)$). Then:

M is unsatisfiable
if and only if
there is a finite and unsatisfiable set M_{inst} of basic instances of M.

In automatic theorem proving, or in logic programming (**PROLOG**), we are always interested in the question:

Given M it is the case that

$$M \models \exists x_1, \ldots x_n \bigwedge_i \phi_i \; ?$$

But this is equivalent to

$$M \cup \{\neg \exists x_1, \ldots x_n \; \textstyle\bigwedge_i \phi_i\}$$

is an unsatisfiable set of clauses.

6.5 Variants of resolution

Our general goal is to derive an existentially quantified formula from a set of formulae:

$$M \vdash \exists\varphi.$$

To use resolution we consider

$$M \cup \{\neg\exists\varphi\}$$

and **put it into clausal form**.
This set is called input.

Instead of allowing arbitrary resolvents, we try to restrict the search space.

Example 6.14 (Unlimited Resolution)

Let $M := \{r(x) \vee \neg p(x),\ p(a),\ s(a)\}$ and
$\square :- \mathtt{s(x)} \wedge \mathtt{r(x)}$ the query.
An **unlimited resolution** might look like this:

$$\dfrac{\dfrac{r(x) \vee \neg p(x) \quad p(a)}{r(a)} \qquad \dfrac{s(a) \quad \neg s(x) \vee \neg r(x)}{\neg r(a)}}{\square}$$

Input resolution: in each resolution step **one of the two parent clauses must be from the input**. In our example:

$$\dfrac{\dfrac{\dfrac{\neg s(x) \vee \neg r(x) \quad s(a)}{\neg r(a)} \qquad r(x) \vee \neg p(x)}{\neg p(a)} \qquad p(a)}{\square}$$

Unfortunately, this is too restrictive.

Lemma 6.15

Input resolution is correct but not refutation complete.

Here is a generalization of input resolution:

Linear resolution: in each resolution step **one of the two parent clauses must be the previously computed clause**. So it is possible to **resolve with a previously derived** clause.

Theorem 6.16 (Completeness of resolution variants)

Linear *resolution is* refutation complete.

Comparing input and linear resolution: While the first is more efficient (search space is smaller) but **not refutation complete**, the second is refutation complete but **more complex**.

Idea:

Maybe input resolution is complete for a restricted class of formulae?

6.6 SLD resolution

Definition 6.17 (Horn clause)

A clause is called Horn clause if it contains at most one positive atom.

A Horn clause is called definite if it contains exactly one positive atom. It has the form

$$\mathtt{A}(\mathtt{t}) \text{ :- } \mathtt{A}_1(\mathtt{t}_1), \ldots, \mathtt{A}_n(\mathtt{t}_n).$$

A Horn clause without positive atom is called query:

$$\square \text{ :- } \mathtt{A}_1(\mathtt{t}_1), \ldots, \mathtt{A}_n(\mathtt{t}_n).$$

Theorem 6.18 (Input resolution for Horn clauses)

Input *resolution for* Horn clauses *is refutation complete.*

Definition 6.19 (SLD resolution wrt P and query Q)

SLD resolution with respect to a program P and the query Q is input resolution beginning with the query $\square :- \mathtt{A_1}, \ldots, \mathtt{A_n}$. Then one A_i is chosen and resolved with a clause of the program. A new query emerges, which will be treated as before. If the empty clause $\square$ can be derived, then SLD resolution was successful and the instantiation of the variables is called computed answer.

Theorem 6.20 (Correctness of SLD resolution)

Let P be a definite program and Q a query. Then each **calculated** *answer for P wrt Q is correct.*

Question:

Is SLD completely instantiated?

Definition 6.21 (Computation rule)

A computation rule R is a function which assigns an atom $A_i \in \{A_1, \ldots, A_n\}$ to each query $\square :- \mathtt{A_1}, \ldots, \mathtt{A_n}$. This A_i is the chosen atom against which we will resolve in the next step.

Note:

PROLOG always uses the leftmost atom.

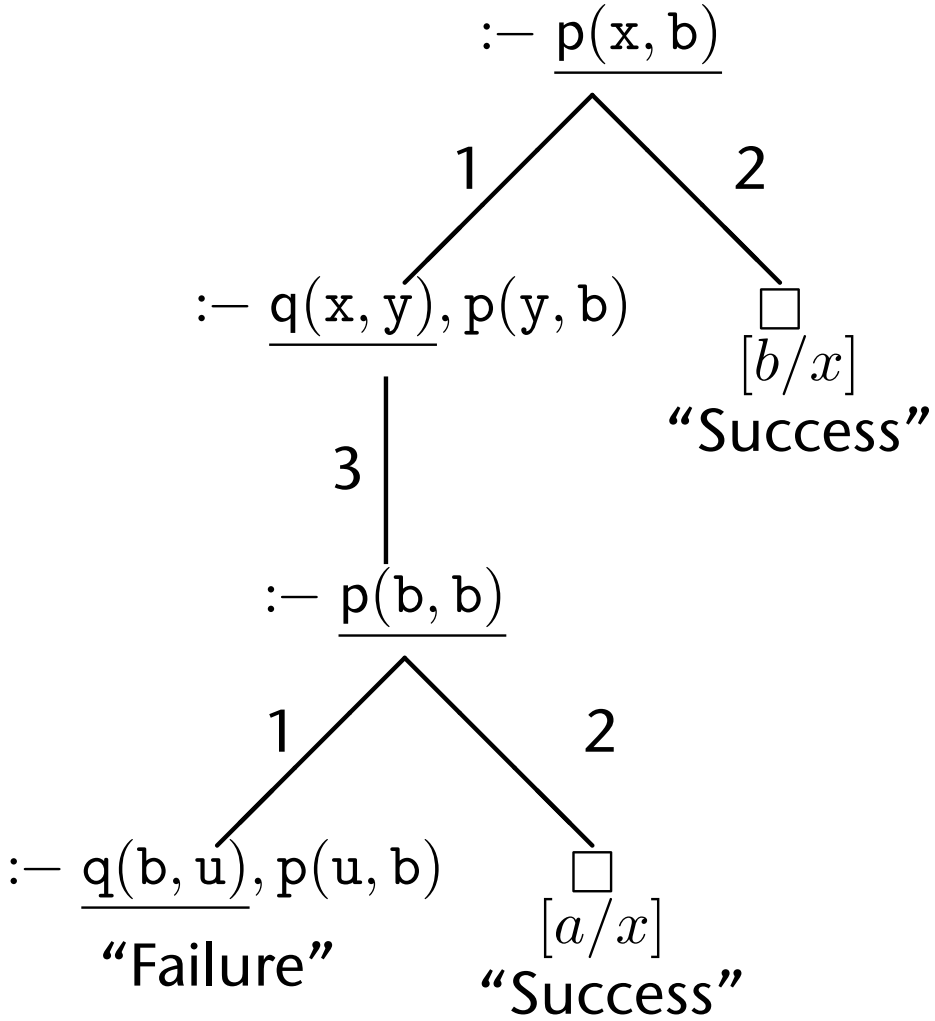
In the following, we are illustrating SLD resolution on the following program:

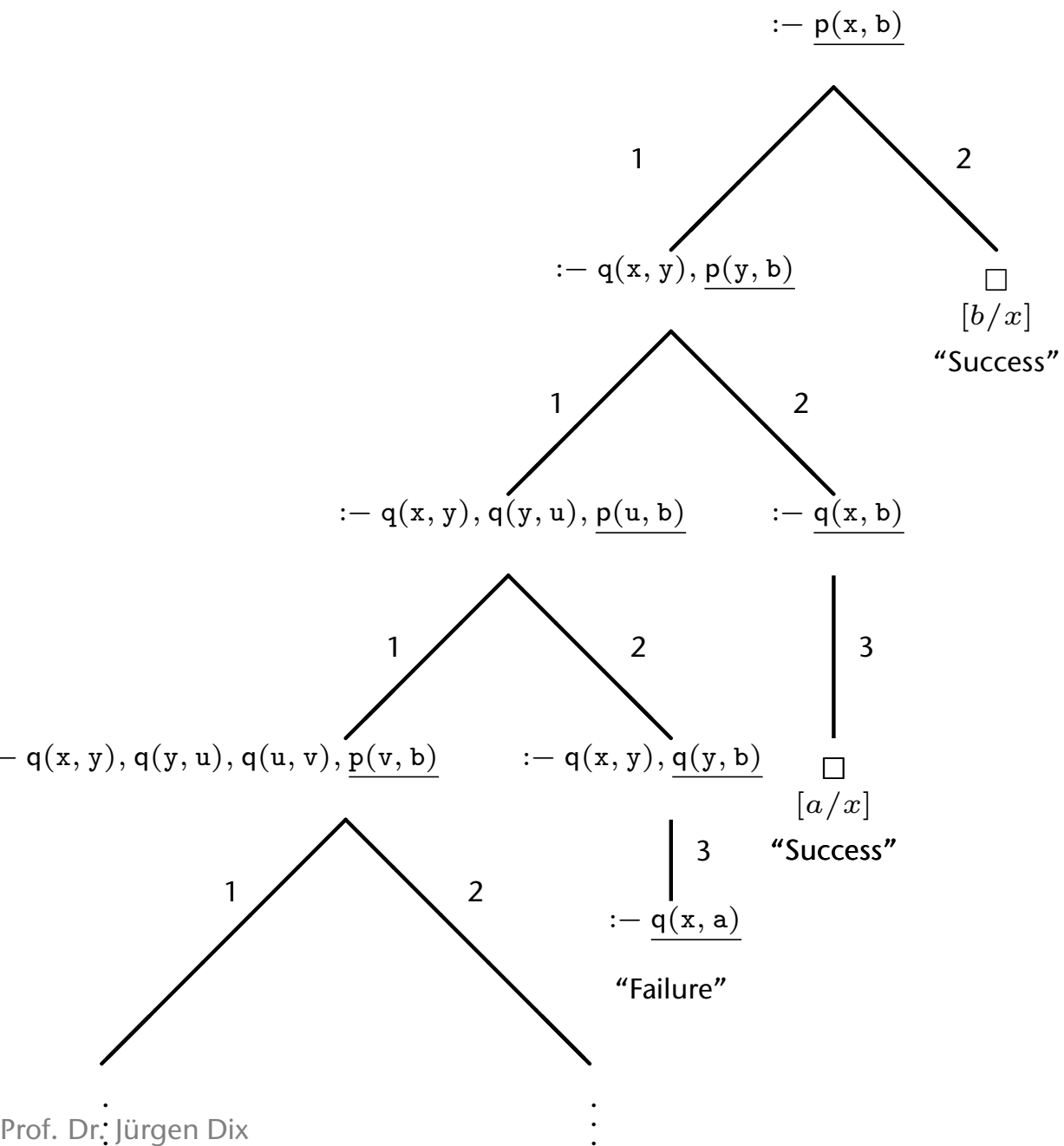
$$\begin{aligned} \mathsf{p}(x,z) &\leftarrow \mathsf{q}(x,y), \mathsf{p}(y,z) \\ \mathsf{p}(x,x) & \\ \mathsf{q}(a,b) & \end{aligned}$$

We would like to know for which instances for x, the fact $\mathsf{p}(x,b)$ follows from the above theory.

Obviously, there are two solutions: $x = a$ and $x = b$ and these are the only ones.

We are now showing how to derive these solutions more formally using SLD resolution.


:– p(x, b)
1
2
:– q(x, y), p(y, b)
□
[b/x]
“Success”
3
:– p(b, b)
1
2
:– q(b, u), p(u, b)
□
[a/x]
“Failure”
“Success”


:— p(x, b)
1
2
:— q(x, y), p(y, b)
□
[b/x]
"Success"
1
2
:— q(x, y), q(y, u), p(u, b)
:— q(x, b)
1
2
3
— q(x, y), q(y, u), q(u, v), p(v, b)
:— q(x, y), q(y, b)
□
[a/x]
"Success"
3
1
2
:— q(x, a)
"Failure"

A SLD tree may have three different kinds of branches:

1. infinite ones,
2. **branches ending with the empty clause** (and leading to an answer) and
3. **failing branches** (dead ends).

Theorem 6.22 (Independence of computation rule)

Let R be a computation rule and σ an answer **calculated** *wrt R (i.e. there is a successful SLD resolution). Then there is also a successful SLD resolution for each other computation rule R' and the answer σ' belonging to R' is a* **variant** *of σ.*

Theorem 6.23 (Completeness of SLD resolution)

Each **correct** *answer substitution is* subsumed through a calculated answer substitution. *I.e.:*

$$P \models \forall Q\Theta$$

implies

SLD computes an answer τ *with:* $\exists\sigma : Q\tau\sigma = Q\Theta$

Question:

How to find successful branches in a SLD tree?

Definition 6.24 (Search rule)

A search rule is a strategy to search for successful branches in SLD trees.

Note:

PROLOG uses depth-first-search.

A SLD resolution is determined by a **computation rule** and a **search rule**.

SLD trees for $P \cup \{Q\}$ are determined by the computation rule.

PROLOG is incomplete because of two reasons:

- depth-first-search, and
- incorrect unification (no occurs check).

A third reason comes up if we also ask for finite and failed SLD resolutions:

- the computation rule must be **fair**, i.e. there must be a guarantee that each atom on the list of goals is **eventually** chosen.

Programming versus knowledge engineering

programming	knowledge engineering
choose language	choose logic
write program	define knowledge base
write compiler	implement calculus
run program	derive new facts

7. PROLOG

Content of this chapter:

We introduce **PROLOG** on a beginners level. With some self study, students should be able to write their own **PROLOG** programs.

Syntax: We describe in detail **PROLOG** as a programming language and introduce the declarative programming paradigm: describe what you want, but not how to achieve it.

Queries and matching: The binding mechanism of queries is presented and unification versus matching explained.

Content of this chapter (2):

Search trees: The most important concept is the search tree of **PROLOG**, based on SLD resolution. It constitutes the procedural kernel of **PROLOG**.

Negation: Negation is differently handled in **PROLOG**: it is closely related to detecting cycles and it is not classical negation.

Recursion: is the most important mechanism of **PROLOG**. We illustrate it with lists, the datastructure in **PROLOG**.

Content of this chapter (3):

Arithmetic: **PROLOG** uses several built-in arithmetic predicates. Because it is built on terms, arithmetic expressions are differently handled than in other programming languages.

Programming: We present a few examples of programming in **PROLOG** to illustrate the use of recursive predicates. To distinguish between input and output variables is paramount.

Acknowledgment

This chapter is heavily built on material provided by Nils Bulling in his BSc course TI1906 Logic-Based Artificial Intelligence at TU Delft in the summer term 2015.

7.1 Motivation

The Art of PROLOG

A logic program is a **set of axioms**, *or* rules, *defining* **relations between objects**. *A* computation *of a logic program is a* **deduction of consequences** *of the program.*

A program defines a set of consequences, which is its meaning. The art of logic programming *is constructing concise and elegant programs that have the desired meaning.*

[Leon Sterling. The Art of ***PROLOG****.]*

Logic Programming: Basics

What is **PROLOG**?

- logic-based programming language,
- focusses on the description of the problem and not how to solve it,
- problem solving closer to human reasoning/thinking,
- allows to **elegantly solve specific** (knowledge-based) problems.

PROLOG is a different programming paradigm. **PROLOG** requires a different way of thinking in order to solve problems. One needs to get used to it. Practice!

- Free online book: Learn **PROLOG** Now!
  (`http://www.learnprolognow.org/`)
  Attention: There are different version of this book. We always refer to the online html version!
- SWI **PROLOG**
  - Free **PROLOG** interpreter
  - Linux, Windows, Mac OS
  - There are more interpreters, but be careful not all are ISO compliant.
  - We use SWI **PROLOG** 6.0.2 (`http://www.swi-prolog.org/`)

Important

The lecture does not cover the whole book **Learn PROLOG Now**.
Read the relevant parts of the book for some exercises!

What is PROLOG?

LP is **closer to natural language** than machine or other higher-level programming languages

Procedural vs. Declarative Programming

This is what Wikipedia says:

- Imperative programming *"is a programming paradigm that* **describes computation in terms of statements** *that change a program state. In much the same way that imperative mood in natural languages expresses commands to take action,* **imperative programs define sequences of commands for the computer to perform**.*"*
- Procedural programming *"is a list or set of instructions* **telling a computer what to do step by step** *and how to perform from the first code to the second code. Procedural programming languages include C, Go, Fortran, Pascal, and BASIC."*

- **Declarative programming** "is a programming paradigm, a style of building the structure and elements of computer programs, that **expresses the logic of a computation without describing its control flow**. Many languages applying this style attempt to minimize or eliminate side effects by **describing what the program should accomplish** in terms of the problem domain, **rather than describing how** to go about accomplishing it as a sequence of the programming language primitives."

Important to know!

PROLOG is not a fully declarative programming language: it has procedural elements! In contrast, **ASP** is the declarative counterpart of **PROLOG**.

Benefits of declarative programming

Suppose we are given a list L of integers. Write a program which removes all 3's in the program.

Java:

```
static void remove
     (ArrayList<Integer> L)
{
   Iterator<Integer> it= L.it();
   while (it.hasNext())
   {
      if (it.next()==3)
         it.remove();
   }}
```

How to solve the problem?

PROLOG:

```
remove([], []).
remove([3|T], L) :-
    remove(T, L).
remove([H|T1], [H, T2]) :-
    not(H = 3),
    remove(T1, T2).
```

What is a solution?.

PROLOG in the Real World

PROLOG Development Center: **advanced scheduling systems** and **speech based applications** for manufacturing, retail and service businesses, larger public institutions, airlines and airports

`http://www.pdc.dk`

Visual PROLOG : support **industrial strength programming of complex knowledge emphasized problems**. Combining the very best features of logical, functional and object-oriented programming paradigms in a consistent and elegant way.

`http://www.visual-prolog.com`

Clarissa at ISS: fully **voice-operated procedure browser**, enabling **astronauts** to be **more efficient** with their hands and eyes and to give full attention to the task while they navigate through the procedure using spoken commands

`http://www.nasaspaceflight.com/2005/06/iss-set-to-get-chatty-with-clarissa/`

OntoDLP: OntoDLV is an intelligent ally for quickly and **easily developing powerful knowledge-based applications** and **decision support systems** (e.g. Planning, Customer Profiling or Workforce Scheduling) that otherwise require complex and labourious programming.

`http://www.exeura.eu`

7.2 Syntax

Example 7.1 (FOL and PROLOG)

$T =_{def} \{\text{agent}(bond), \forall x(\text{agent}(x) \rightarrow \text{indanger}(x))\}$
What is the difference between **facts** and **rules**?
What can you infer? $T \models \text{indanger}(bond)$

- all models satisfying T must satisfy $\text{indanger}(bond)$
- apply resolution to the **clauses**:

$$\text{agent}(bond),\ \neg\text{agent}(x) \vee \text{indanger}(x),\ \neg\text{indanger}(bond)$$

Corresponding **PROLOG program**:

```
agent(bond).                (this is a fact)
indanger(X) :- agent(X).    (this is a rule)
```

We can **ask a query**: `?- indanger(bond).`

PROLOG answers "yes".

Example 7.2 (A first example)

The knowledge base or database consists of the following facts.

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

Symbol ?− shows that **PROLOG** waits for a command: a **query**.
Load the program and query the following:

- ?− woman(mia).
- ?− woman(alice).
- ?− woman(alice), woman(mia).
- ?− woman(X).

- `consult/1` loads a **PROLOG** program.
- `assert/1` adds new facts and rules to the database.
- Sometimes `assert/1` requires predicates to be declared `dynamic`. e.g. `:-dynamic woman/1.`

Example 7.3

Do `assert(woman(alice)).` in the previous example. You get the following error:

```
ERROR: assert/1: No permission to modify static
procedure ‘woman/1’
```

We add the clause `:- dynamic woman/1` and try again.

Remark 7.4

- `:-` *Implication* ($\varphi \to \psi$ *becomes* ψ`:-`φ)
- `;` *disjunction and* `,` *conjunction*

Definition 7.5 (Constant)

A constant (sometimes also called atom) is one of the following:

1. a non-empty sequence of letters, digits, or the underscore _ that **starts with a lower-case letter**;
2. any sequence of letters, digits, or the underscore **enclosed in single quote** ‘ . . . ‘;
3. a string of special characters, like `@=`, `===>`, `;`, and `:-`.

See **PROLOG** manual for more details.

Examples of atoms are: `car`, `hOUSE_2`, `father`, `‘House‘`.

Definition 7.6 (Number)

A number is is any sequence of numbers, possibly containing a dot.

For example `1`, `1991`, `1.92` are numbers in **PROLOG**.

Definition 7.7 (Variable)

A variable is a sequence of letters, digits, or the underscore _ that starts with an **upper-case letter or the underscore**.
The variable _ is called anonymous variable.

Examples of variables are: `X`, `Car`, `HOUSE_2`, `_father`, `House`.

If the anonymous variable is used in a predicate `p(X,_)` then it is handled as an arbitrary variable without a specific name.

Definition 7.8 (Terms, predicate, arity)

A term is defined recursively:

1. Each constant, number, and variable is a **term**.
2. $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_k)$ is a term, if all $\mathtt{t}_i$, $i = 1, \ldots, k$, are terms and $\mathtt{f}$ is a constant.

A term $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_k)$ is called compound term. A term without variables is a ground term. $\mathtt{f}$ is called functor. Each function $\mathtt{f}$ is assigned an arity, which is the number of terms it accepts. We write $\mathtt{f}/\mathtt{k}$ for a functor $\mathtt{f}$ of **arity** k. A term is also called predicate.

Remark 7.9 (Attention)

In contrast to FOL, *the* **distinction between terms and predicates is blurred in PROLOG**.

Examples: $\mathtt{X}$, $\mathtt{bond}$, $\mathtt{mother}(\mathtt{X}, \mathtt{Y})$, $\mathtt{father}(\mathtt{son}(\mathtt{X}), \mathtt{Y})$

Note: The **arity of a predicate is important**! The same functor with a different arity gives rise to a different predicate.

Definition 7.10 (PROLOG Fact)

A **PROLOG** fact is a predicate followed by a full stop.

The following are facts:
`this_is_the_Logic_and_Verification_lecture.` and `father(bob, X).`

Facts express specific **pieces of knowledge**.

Definition 7.11 (PROLOG rule)

A **PROLOG rule** is given by

$$\texttt{A}_0 \texttt{ :- } \texttt{A}_1\texttt{,} \ldots\texttt{,} \texttt{ A}_n\texttt{.}$$

where $n \geq 0$ and each $\texttt{A}_i$ for $i = 0, \ldots, n$ is a predicate. The predicate $\texttt{A}_0$ is the **head** of the rule, and $\texttt{A}_1, \ldots, \texttt{A}_n$ is the **body** of the rule.

Note that a rule ends with a full stop.

The following two are rules

```
father(X,Y):- son(Y,X).
daughter(X,Y):- parent(Y,X), female(X).
```

In logical terms, a rule $\texttt{A}_0\texttt{:- A}_1\texttt{,} \ldots\texttt{,} \texttt{ A}_n\texttt{.}$ corresponds to the formula $A_1 \wedge \ldots \wedge A_n \rightarrow A_0$.

Definition 7.12 (PROLOG clause)

A clause is a **PROLOG fact** or a **PROLOG rule**.

Remember: `p(X) :- q(X).` means $\forall x\,(\mathsf{q}(x) \to \mathsf{p}(x))$ which is equivalent to the clause

$$\neg \mathsf{p}(x) \vee \mathsf{q}(x).$$

Remark 7.13 (Clauses are universal)

It is important to note that a clause is universal. That is, the rule `p(X) :- q(X).` *is true for all* `X`*. Remember our discussion on Slide 485.*

Example 7.14

```
happy(yolanda).
listens2music(mia).
listens2music(yolanda):- happy(yolanda).
playsAirGuitar(mia):- listens2music(mia).
playsAirGuitar(yolanda):- listens2music(yolanda).
```

There are 5 clauses, 2 facts and 3 rules. **What are the answers to the following queries?**

1. `?-listens2music(yolanda).`
2. `?-playsAirGuitar(yolanda).`
3. `?-playsAirGuitar(Mia).` (Attention, read carefully!)

Definition 7.15 (PROLOG program)

A **PROLOG** program is a finite sequence of clauses.

Remark 7.16 (Conjunction and disjunction)

*The comma in a **PROLOG** rule represents a conjunction. It is also possible to express a disjunction by a semicolon. So the rule*

```
r(X) :- s(X), t(X); u(X).
```

is a shortcut for the two rules

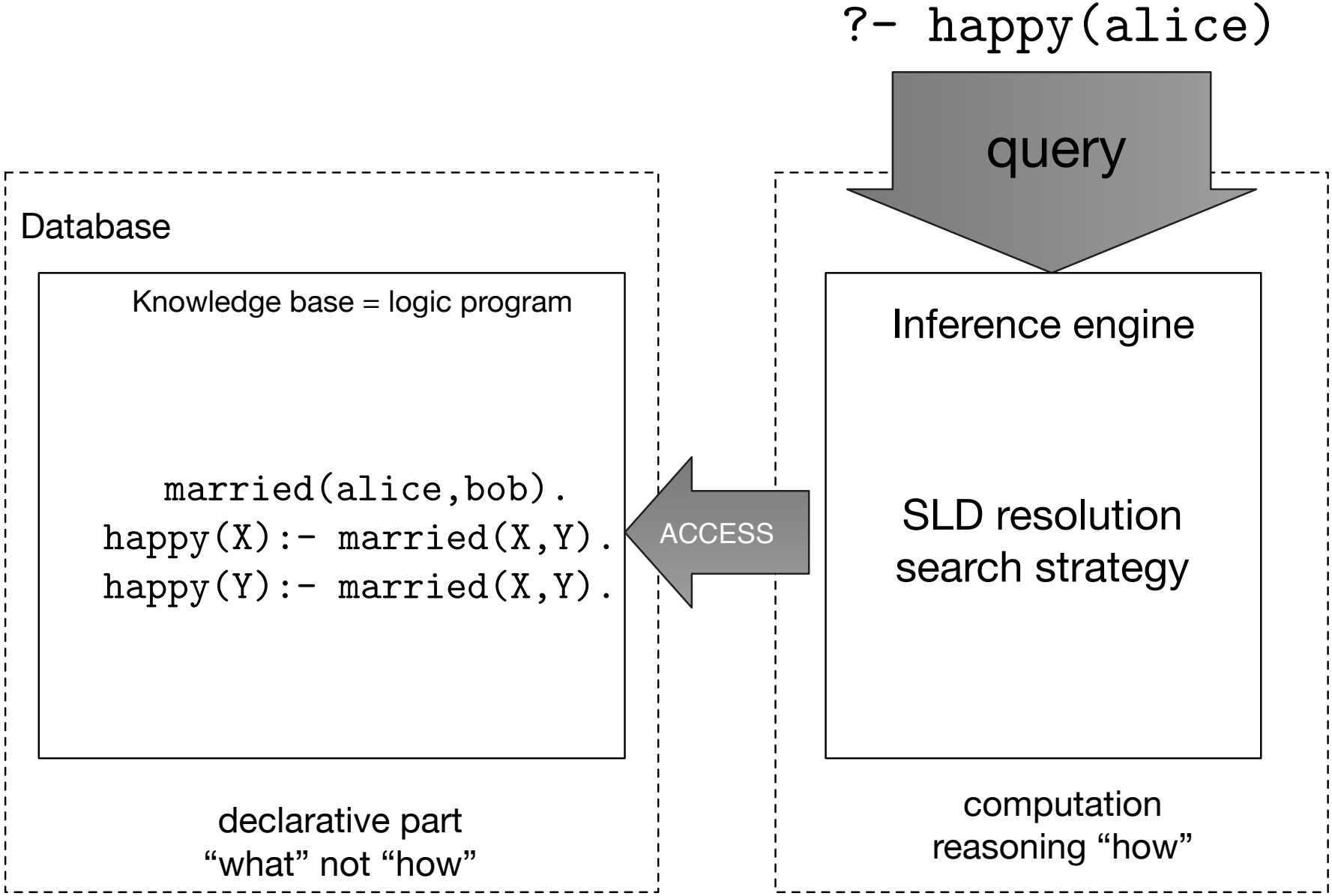
```
r(X) :- s(X), t(X).
r(X) :- u(X).
```

Remark 7.17 (Some built-in predicates)

- *The matching predicate* `=(X, Y)` *is true if and only if the terms* `X` *and* `Y` *can be made equal. We also write* `X = Y`*. It is important to note that this is* **different from FOL unification** *as we will discuss in more detail later. (**PROLOG** lacks the occurs check.)*
- *The predicates* `fail` *and* `true` *are always false and true, respectively.*
- *To load a **PROLOG** program the predicate* `consult/1` *is used.* `consult('filename.pl')` *compiles the file "filename.pl".*
- *The write predicate* `write/1` *allows to write a term to the output.*

7.3 Queries and Matching

How PROLOG Works



Queries, Goals and Matching

Definition 7.18 (Query and goal)

A (simple) query is of type `?- p.` where `p` is a term. It is also possible to combine terms within queries to **complex queries** by **conjunctions** (,) and **disjunctions** (;). For example, `?-p1,...,pk` with $k \geq 1$ is a complex query.
The element on the right-hand-side of a query is called goal. In case of complex queries, each term on the right-hand side is called a subgoal.

Remark 7.19 (Queries are existential)

In contrast to Remark 7.13, queries are existential. *That is, the query* `?- p(X)` *asks whether* **there is some** `X`. *That is, it corresponds to a* FOL *formula* $\exists x \mathsf{p}(x)$.

Asking a query ?– p(a) lets **PROLOG** try to derive p(a) from the program using input (or SLD) resolution (we refer to Definition 6.19 on Slide 506).

1. It tries to find a rule p(X) :– q(X), r(Y). with head p(X) which can be **unified** with the goal p(a) (topmost-rules are tried first!).
2. p(X) and p(a) are unified with $X = a$ (**substitution**!)
3. In order to derive a the **goals** q(a), r(Y) have to be derived. (Note that the X is instantiated). The new query is: ?– q(a), r(Y)..
4. The process repeats for q(a) and r(Y) **in this very order**!
5. Proceed until a goal is already contained in the knowledge base as a fact.

We first give an example and then consider it in more detail.

Example 7.20

```
loves(vince,mia).
loves(marcel,mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
jealous(X,Y):- loves(X,Z), loves(Y,Z), X \= Y.
```

What does **PROLOG** return for `?- jealous(marcel, W).`?

Therefore: We have to add that X and Y should be distinct. This is expressed by `\=`.

Matching

Before discussing proof search in more detail, we consider the matching of predicates. This is an important technique which **gives PROLOG its programming power**.

We already discussed unification in the context of FOL. **PROLOG** also unifies terms but **does not perform the occurs check**! We call it matching rather than unification.

- unification: as introduced for FOL
- matching: **PROLOG**'s way to unify, no occurs check.

Remark 7.21 (Attention)

Often, matching is just called unification!

Definition 7.22 (Matching in PROLOG)

Two terms `t` and `r` match if one of the following conditions hold:

1. `t` and `r` are **identical constants** or **numbers**.
2. If `t` is a **variable** and `r` **is not**, then `t` and `r` match and `t` **is instantiated with** `r`. Analogously, if `r` is a variable. If both terms are variables they are instantiated with each other.
3. If `t` and `r` are **compound terms** then they match if each of the following conditions hold:
   - `t` and `r` have the **same outermost functor**, i.e. `t=f(t1,...,tk)` and `r=f(r1,...,rk)`;
   - each `ti` **matches** with `ri` for $i = 1, \ldots, k$ and
   - the variables **instantiation are compatible** (i.e. a variable cannot be assigned different terms which are not variables).

Definition 7.23 (Matching)

In **PROLOG**, the matching predicate is given as `=/2`. If two terms match **PROLOG** returns the most general **instantiation** σ of the variables. We write `t`σ to denote the application of σ to `t`.The predicate `\=/2` is true if two terms do not match.

PROLOG also offers a predicate to perform (FOL-)unification: `unify_with_occurs_check/2`.

Example 7.24

What is the result of the following queries?

1. `X=a`
2. `f(X,a)=f(b,X)`
3. `f(X,a)=f(b,Y)`
4. `X=f(X)`
5. `X+3=5+Y`
6. `X+2=5`
7. `1+2=2+1`
8. `X+3\=5+Y`

Example 7.25 (Variable instantiation)

```
woman(mia).
woman(jody).
woman(alice).
```

We **query** `?- woman(X).` What is the answer?

The answer of the first matching rule is returned. With `;` further matching instantiations are returned.

```
?- woman(X).
X = mia;
X = jody;
X = alice.
```

Example 7.26 (Variable instantiation)

```
woman(mia).
woman(jody).
woman(alice).
happy(mia).
```

We **query** `?- woman(X), happy(X).` What is the answer?

`X = mia.`

There are no more answers! Why not?

Matching is very important in **PROLOG** and also a powerful programming technique as shown by the following examples.

Example 7.27 (Programming with matching)

Consider the program:

```
vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).
```

What are the answers to the following queries?

1. `?- horizontal(line(point(1,1),point(1,3)))`
2. `?- horizontal(line(point(1,1),point(3,2)))`
3. `?- horizontal(line(point(1,1),point(2,Y)))`
4. `?- horizontal(line(point(1,1),Point))`

PROLOG computes a most general substitution.

Remark 7.28 (Attention)

Again, be careful: matching can give different results than (FOL-)unification. We will come back to this issue: ***PROLOG*** *does not perform the occurs check! The occurs check is computationally expensive.*

Proposition 7.29

*If two terms FOL-**unify**, then **they match**. The other direction is not true.*

7.4 Search Trees and Negation

Example 7.30 (Deduction — Informal)

Suppose we are given the program

```
q(a).
r(a).
p(X) :- q(X), r(X).
```

and ask the query `?- p(a)`. What happens? The only way to derive the goal `p(a)` is by proving `p(X)` for `X = a`.

This means to show that the goal `q(a), r(a)` holds.

PROLOG selects the new (sub)goal `q(a)` which can be derived because of the fact `q(a)`.

The goal `r(a)` remains. It is true because of the fact `r(a)`..

Searching for Proofs

Searching for a proof is a three-steps process:

1. ask a query,
2. find a rule (which one?) the **head of which matches with the query**,
3. try to derive the body of the rule (**new query**).

To prove a goal **PROLOG** selects a subgoal to prove, considers this subgoal and so on. There are **many ways**, however, to **select a subgoal**: heads of **different rules may match**, and within a rule the **predicates in the body** can be treated in different **orders**. In this section we discuss **PROLOG**'s search strategy.

This is crucial, as the correctness of programs depends on the right understanding of **PROLOG**'s functioning.

In the following we assume that a logic program is ordered, from top to bottom.

Three principles underly the search for proofs.

1. backward chaining: start from the current goal and consider bodies which allow to derive the given goal:
   (i) goal `p(X)`; (ii) consider a rule `p(Y) :- q(Z).`; and (iii) try to derive `q(Z)`.
2. linear: traverse the logic program **from top to bottom** (selecting appropriate rules)
3. backtracking: if the goal can not be derived (or has been successfully derived) try **other alternative rules** in step 2 (i.e. the 2nd topmost, 3rd topmost etc.). **Backtrack to the closest possible choice point** and try a different alternative.

Backward chaining (simple queries)

- Given a query $?- \texttt{q(t)}.$
- find a rule $\texttt{q(r)} :- \texttt{p(s)}.$ the head of which matches with $\texttt{q(t)}$. Choose the **topmost** rule if there are several such rules. Let σ denote the most general instantiation.
- If **PROLOG** is able to derive the goal $\texttt{p(s)}\sigma$ then $\texttt{p(t)}$ would be derivable.
- Thus, ask the **new query** $?- \texttt{p(s)}\sigma$

This is called **backward chaining** because we chain the rules one after another, starting from the goal/query we want to derive.

What if queries are compound?

Backward chaining (compound queries)

- Given a query ?– q1(t1), q2(t2).
- find a rule q1(t3) :– p1(t4), p2(t5). the head of which matches with q(t1). (**topmost** again)
- If **PROLOG** is able to derive the goals q2(t2), p1(t4)σ and p2(t5)σ then p(t1) is derivable.
- We get the **new query** ?– p1(t4)σ, p2(t5)σ, q2(t2)

What do we observe?

Important points

1. Topmost rules are selected first.
2. Queries are treated from left to right in a depth-first search manner (new goals p1 and p2 are placed before the "old goal" q2 in the new query)!

When can we terminate? That is, when have we shown whether the initial query is true or false.

- If we obtain the empty query $?-$, we are done. There are no more goals to derive. The original query holds. $\rightsquigarrow$ compare with **resolution**
- We may also end up with a query $?- \mathtt{p(t)}$ and no rule's head matches with $\mathtt{p(t)}$. What now?

Backtracking

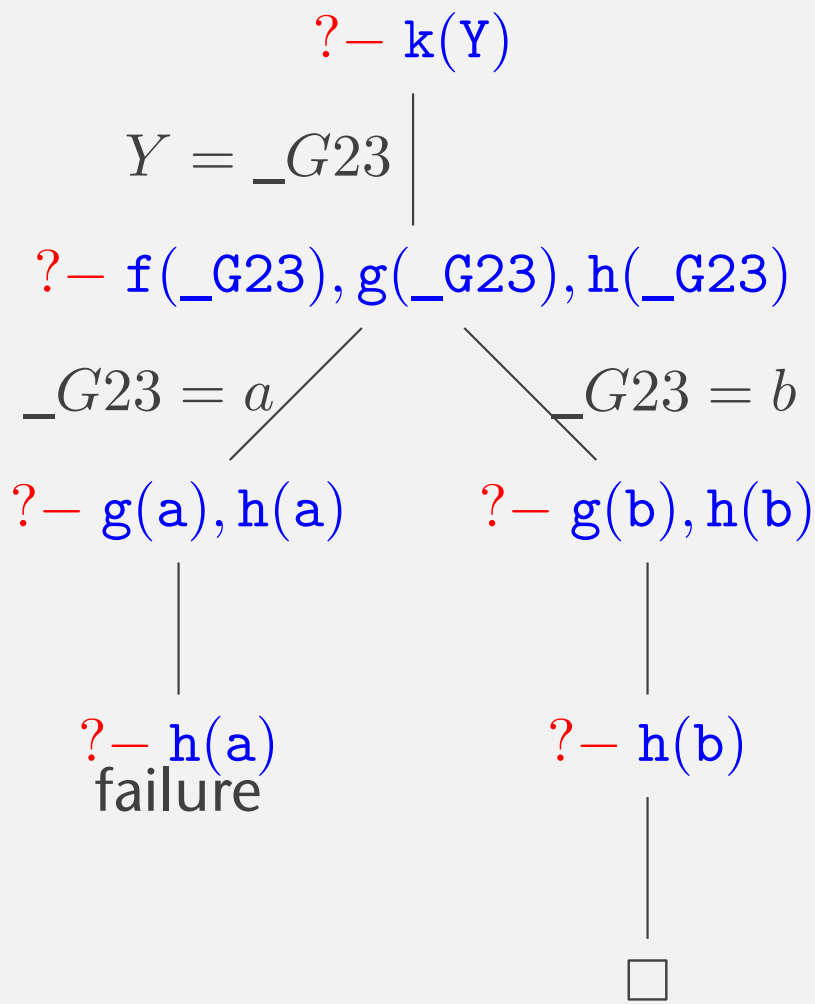
If we end up with a query $?- \mathtt{p_1(t_1), \ldots, p_k(t_k)}$ where the head of no rule matches with $\mathtt{p_1(t_1)}$ then:

1. In the previous selection of rules, go back to the most recent query and **choose another alternative rule** (again as high up in the logic program as possible)
2. Proceed as before.

Example 7.31

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X), g(X), h(X).
```

Query: ?− k(Y)



Example 7.32

1. First branch leads to a **failure**: there is no rule with a head that matches with `h(a)`.
2. **PROLOG backtracks** to the most recent choice point . . .
3. . . . and tries a different rule/fact.
4. This branch leads to the **empty clause**.
5. The query is shown to be derived.
6. The calculated variable instantiation gives the solution.

Example 7.33

```
f(1).
f(2).
f(3).
p(3,4).
q(4).
r(3).
r(4).
r(5).
g(X) :- p(X,Y), q(Y), r(X).
h(X) :- f(X).
h(X) :- g(X).
```

Which of the following is the correct output on `?- h(X)`?

1. $X = 1; X = 2; X = 3; X = 4; X = 5.$
2. $X = 1; X = 2; X = 3; X = 3.$
3. $X = 1; X = 2; X = 3; X = 4.$
4. $X = 1; X = 2; X = 3.$
5. $X = 1; X = 2; X = 3; X = 3; X = 4.$

Example 7.34

```
loves(vince,mia).
loves(marcel,mia).
jealous(A,B) :-
        loves(A,C),
        loves(B,C).
```

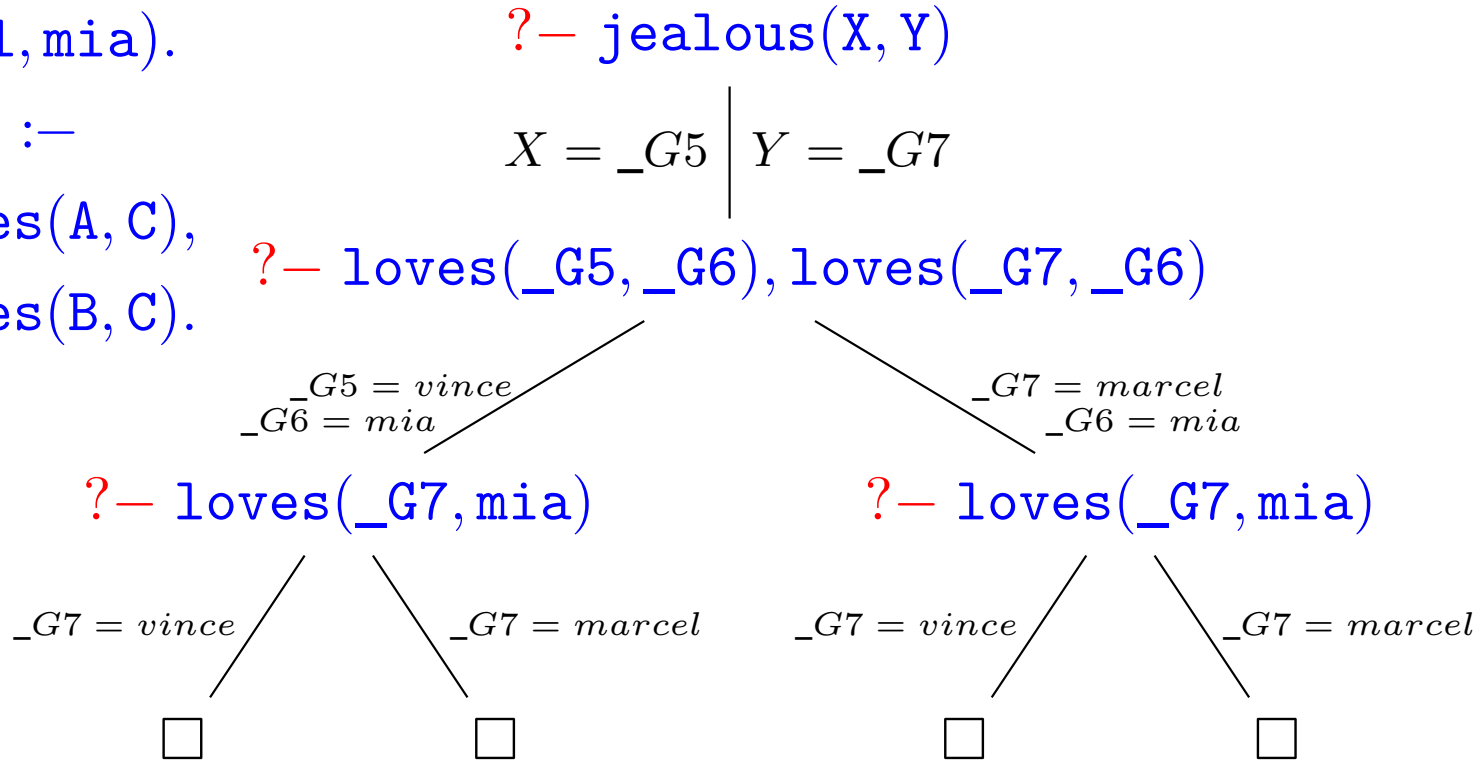
Query: `jealous(X,Y).`

What is the output?

1.
```
X = vince, Y = vince;
X = marcel, Y = marcel;
X = vince, Y = marcel.
```
2.
```
X = vince, Y = vince;
X = vince, Y = marcel;
X = marcel, Y = vince;
X = marcel, Y = marcel.
```
3.
```
X = vince, Y = vince;
X = vince, Y = marcel;
X = marcel, Y = marcel;
X = marcel, Y = vince.
```

Corresponding search tree

```
loves(vince,mia).
loves(marcel,mia).
jealous(A,B) :-
        loves(A,C),
        loves(B,C).
```



Definition 7.35 (PROLOG search tree)

Given a **PROLOG** program P and a query Q the above described procedure results in a tree with the following properties:

- the **root** is labelled with query Q,
- nodes are labelled with **queries**,
- child **nodes are ordered**,
- edges are labelled with **variable instantiations**.

We call this tree the search tree of P and Q (sometimes also called proof tree).

A search tree may have three different kinds of branches:

1. **infinite ones**,
2. **succesful branches ending with the empty clause** (and leading to an answer), and
3. **failing branches** (dead ends).

Example 7.36

Consider the program:

```
s.
p:-r.
p:-s.
p:-p.
```

For the query `?-p` can you identify a **failing**, **successful**, and **infinite branch** in the search tree?

1. The branch which always uses the rule `p :- r.`
2. The branch which always uses the rule `p :- s.`
3. The branch which always uses the rule `p :- p.`

How does PROLOG traverse the search tree?

1. **PROLOG** searches the tree via **depth first search** with backtracking where child nodes corresponding to input rules higher up in the program are traversed first.
2. Be careful: **PROLOG** is both incomplete and incorrect because:
   - **PROLOG** uses depth-first-search
   - **PROLOG** uses incorrect unification (no occurs check).

   We will discuss this in more detail later in the lecture.

Definition 7.37 (Derived)

A goal `G` can be derived from a program P if **PROLOG** finds a **succesful trace** in the search tree of P for `G`. We write $P \vdash$ `G` and write $D(P) = \{G \mid P \vdash \texttt{G}\}$ to denote the set of all derivable terms.

Example 7.38 (Matching and search)

```
p(a) :- p(f(X)).
p(X) :- q(X,X).
q(X,f(X)).
```

What is the result of the queries below?

1. `?- p(a)`
2. `?- p(X)`
3. `?- p(f(X))`
4. `?- p(f(a))`

Note, that some branches of the search tree may never be traversed by **PROLOG**. This depends on the order of the rules. (E.g. if the initial branch in the tree is infinite.)

Declarative and Procedural Meaning

In the previous section we investigated how **PROLOG** answers queries; in particular, how it searches for a refutation of a query:

- **PROLOG** uses depth first search
- top-most clauses are considered **first**
- left-most subgoals are evaluated **first**

Understanding these details is of utmost importance when programming in **PROLOG** because the behavior of the programs depends on it.

Declarative meaning: The **logical meaning** of a program, what it should mean in logical terms only considering the knowledge base. Apply logical reasoning and **forget about PROLOG and its search strategy**.

Procedural meaning: The actual behavior of the program given **PROLOG**'s search strategy. The solutions **that are calculated**. The way **how PROLOG** answers a query. ($\rightsquigarrow$ backtracking, depth-first search, linear, and goal ordering)

Attention

The declarative and procedural meaning of a program can be different.

Example 7.39 (Rule ordering)

Consider the following two programs and the query `?- p(X).`
Program 1:

```
p(a).
p(b).
```

Program 2:

```
p(b).
p(a).
```

What can one observe? Solutions are found in a **different order**.

Example 7.40 (Rule and goal ordering)

Now, we query ?– q(a). Program 1:

```
p(a).
q(X) :– p(a).
q(X) :– q(f(X)).
```

Program 2:

```
p(a).
q(X) :– q(f(X)).
q(X) :– p(a).
```

What can one observe?
Program 1 terminates, program 2 does not terminate. Rule ordering affects the procedural meaning of a program.

Example 7.41 (How to design a program?)

Often, a good way to design a **PROLOG** program is to work along the following steps:

1. Understand the problem (of course!).
2. Think about a declarative solution, and come up with a logic program.
3. Refine the program and take into consideration **PROLOG**'s **procedural meaning**.

Negation as Finite Failure

So far, we have only asked **positive queries**: Does `Q` follow from the program P? Can we ask whether some predicate does not follow?

PROLOG allows a predicate `\+ /1` which expresses a specific type of negation. It is important to understand, however, that this is **not logical negation**.

In short, `\+ (p)` is true in program P *if and only if* `p` **cannot be derived** from P because any attempt fails finitely.

Note the difference:

- ¬p in logic: proposition `p` **is false**
- `\+ p` in **PROLOG**: any attempt to prove `p` fails finitely.

What does it mean that something fails finitely?

Example 7.42 (Non-derivability due to infinite branches)

Consider the program P given by `p :- p.`. Obviously `p` can not be derived (we enter an infinite loop). In general, determining such infinite cycles is undecidable.

Therefore $P \nvdash$ `p`: but not all attempts fail finitely (in fact, no attempt fails finitely).

Definition 7.43 (Finitely failed)

A **search tree for a program P and a goal `G`** is **finitely failed** if there is **no successful branch** in the tree **and** also **no infinite branch**. The set of goals which have a finitely failed search tree with respect to a program P is denoted $FF(P)$.

Thus, while trying to prove `G`, a finite tree is constructed with no successful branch. The fact that the tree is finite means that the procedure terminates and we know that there is no successful branch.

Definition 7.44 (Negation-as-finite-failure)

The predicate `\+` is called negation-as-finite-failure (naff). We say that `\+ (G)` can be derived from P *if and only if* $G \in FF(P)$.

As before, we write $P \vdash$ `\+ (G)` *if, by definition,* $G \in FF(P)$. Thus, we have

$$\texttt{\textbackslash+ (G)} \in D(P) \quad \textit{if and only if} \quad G \in FF(P).$$

Example 7.45 (Naff is not logical negation)

In classical (propositional) logic, we have:

1. $\{\neg p, \neg p \to q\} \models q$
2. $\{\neg p \to q\} \not\models q$

and in **PROLOG** we have:

1. $\{$`f(b). q :- \+ (f(a)).`$\} \vdash q$

While `\+` is on the one hand weaker than classical negation, it is at the same time stronger, because we use the orientation of the rules.

The following lemma shows that **negation-as-finite-failure** is different from "**negation-as-failure**" (i.e. something is false if it cannot be shown to be true).

Proposition 7.46

Negation-as-failure does not imply negation-as-finite-failure*:* `G` $\notin D(P)$ *does not imply that* `G` $\in FF(P)$.

Example 7.47 (Finite vs. infinite failure)

Consider the program:

```
above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).
on(c, b).
on(b, a).
```

We have that $P \nvdash$ `above(b, c)` and `above(b, c)` $\in FF(P)$. Thus, we also have $P \vdash$ `\+ (above(b, c))`.

Now, let us add the rule `above(X, Y) :- above(X, Y).` to P. Can we still derive `\+ (above(b, c))`? No, because an infinite branch is contained in the proof tree.

Negation-as-finite-failure has to be used with caution.

Example 7.48 (Negation-as-failure)

Consider the program:

```
daughterOf(alice,bob).
sonOf(paul,mary).
onlyDaugther1(X):- \+ sonOf(Y,X), daughterOf(Z,X).
onlyDaugther2(X):- daughterOf(Z,X), \+ sonOf(Y,X).
```

We would like to derive all parents who have a daughter but no son. Which of the following queries gives the correct answer?

1. `?- onlyDaughter1(X).`
2. `?- onlyDaughter2(X).`

Cuts

- It is possible to control how Prolog traverses the search tree.
- More precisely, it is possible to prune the search tree.
- That is, to remove subtrees from the search tree.

Definition 7.49 (Cut)

The cut is a predicate `!/0` which can be used in the body of a rule. The predicate `!/0` is always true.

Example 7.50 (Removing subsequent f)

We write a program that takes as input terms consisting of nestings of f and g applied on constant a and returns the same term with **subsequent occurences of f removed**.

```
(1) rem(f(a),f(a)).
(2) rem(g(a),g(a)).
(3) rem(f(X),Res) :- begin_f(X),rem(X,Res).
(4) rem(f(X),f(Res)) :- rem(X,Res).
(5) rem(g(X),g(Res)) :- rem(X,Res).
(6) begin_f(f(X)).
```

The solution found first is **correct**. But then wrong solutions are returned because of backtracking. Why?

On query $?-\ \texttt{rem(f(f(g(f(f(a))))),X).}$ Prolog gives the following output:

$$\texttt{X = f(g(f(a)));}$$
$$\texttt{X = f(g(f(f(a))));}$$
$$\texttt{X = f(f(g(f(a))));}$$
$$\texttt{X = f(f(g(f(f(a)))));}$$
$$\texttt{false.}$$

Example 7.51 (Previous example with cut)

Now we change clause number (3) as follows:

$$(3) \quad \texttt{rem(f(X),Res)} :- \texttt{begin\_f(X),!,rem(X,Res).}$$

On query $?-$ `f(f(g(f(f(a)))))`. Prolog gives the following output:

```
f(g(f(a)));
false.
```

I.e. only the first correct solution is returned. Once the cut ! is encountered, backtracking below the node in the search tree containing the goal !, `rem(X,Res)`, ... is no longer possible.

The cut predicate prunes the search tree above the node that caused the cut of the rule to be added to a node as a subgoal. We make this more formal in Definition 7.52

Cut in Example 7.51

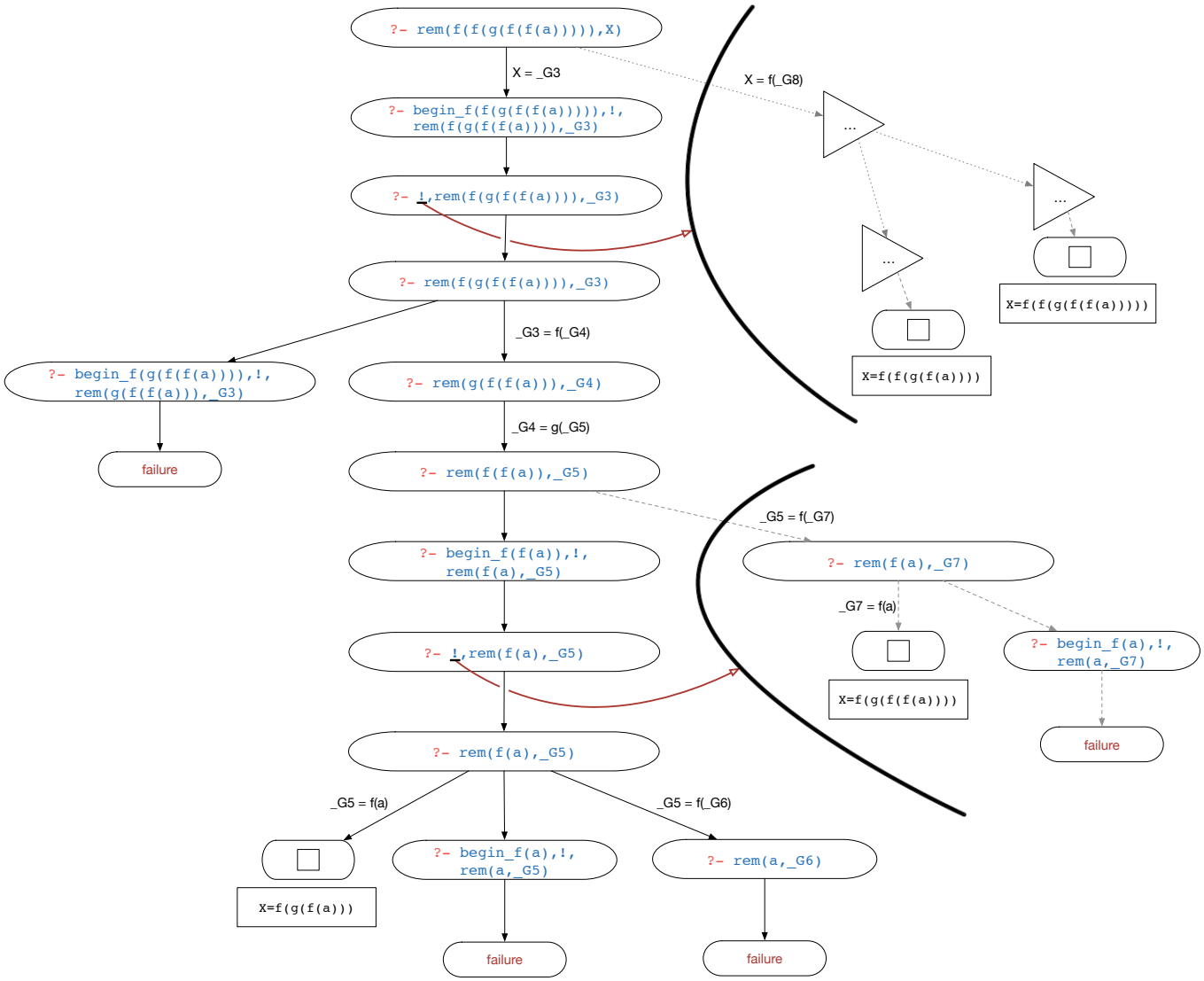


Figure 7.36: How the cut works.

Origin, Cut-node

- A node in the search tree labelled $!, A_1, \ldots, A_k$ is called cut-node.
- The origin of a cut node is the closest ancestor of the node which contains fewer cut predicates.

Intuitively, the origin of a cut node is the **node which caused the introduction of the cut**.

If the cut is evaluated, the overall search tree is pruned below its origin.

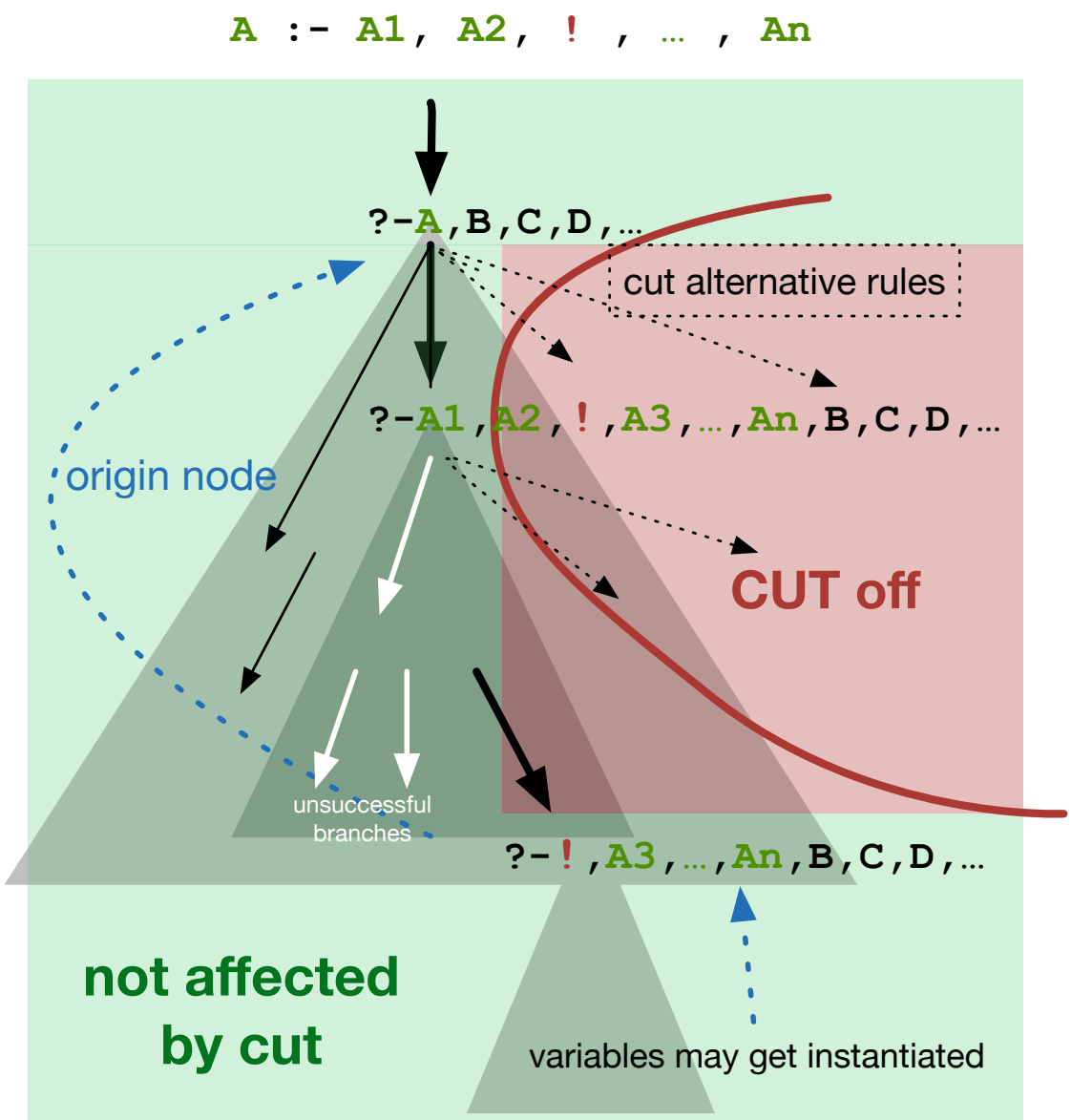
Definition 7.52 (Search tree with cut)

Let T be the search tree of a query Q and a program P.

If the cut-node $!, \mathtt{A}_1, \ldots, \mathtt{A}_k$ is reached, then

- $\mathtt{A}_1, \ldots, \mathtt{A}_k$ is the only descendant, and
- no alternative outgoing branches of the origin of the cut node are considered anymore.

How the cut works in general



Example 7.53

Query `?- father(X, tom).`

```
(1)  father(X, Y) :- parent(X, Y), male(X).
(2)  parent(ben, tom).
(3)  parent(ben, tom).
(4)  parent(sam, ben).
(5)  parent(alice, ben).
(6)  male(ben).
(7)  male(sam).
```

What happens when we change rule (1) as follows:

```
(1)  father(X, Y) :- parent(X, Y), male(X), !.
```

Remark 7.54 (Cuts)

The previous examples have shown two different types of cuts: removal of subtrees with and without solutions.

- *A cut is called* green *if the removed tree does not contain any successful branches.*
- *A cut is called* red *if the removed tree does contain successful branches.*

Green cuts are unproblematic whereas red cuts have to be used with caution.

Cuts can improve the efficiency of a PROLOG program dramatically.

Example 7.55

Discuss the following two programs, intended to compute the minimum of two numbers:

Program 1:

```
min(X,Y,X) :- X < Y,!.
min(X,Y,Y).
```

Program 2:

```
min(X,Y,X) :- X < Y,!.
min(X,Y,Y) :- X >= Y.
```

7.5 Recursion and Lists

Left and right recursion

Definition 7.56 (Recursion)

A predicate is defined recursively if it occurs in the head **and** the body of a (the same) rule.

Example 7.57

We would like to define a descent relation, using predicates `child/2` and `descendant/2`, between Paul and Anne:

```
child(paul, anna).
child(anna, sara).
descendant(X, Y) :- child(X, Y).
descendant(X, Y) :- child(X, Z), child(Z, Y).
```

Recursive rules

What if we add the fact `child(sara, frank).`? In this case we have to add the rule:

```
descendant(X, Y) :- child(X, W), child(W, Z), child(Z, Y).
```

What if we add another fact etc.? This doesn't seem to scale and only works if we know the facts. Any better solution?

```
descendant(X, Y) :- child(X, Y).
```

base clause

```
descendant(X, Y) :- child(X, Z), descendant(Z, Y).
```

recursive clause

Attention: Recursion is dangerous! Think about your program carefully.

Example 7.58 (Left and right recursion)

We consider the following variants of the rules of the descendant example:

(1) base clause on top, right recursion

```
descendant(X, Y) :- child(X, Y).
descendant(X, Y) :- child(X, Z), descendant(Z, Y).
```

(2) base clause not on top, right recursion

```
descendant(X, Y) :- child(X, Z), descendant(Z, Y).
descendant(X, Y) :- child(X, Y).
```

The following uses left recursion, in contrast to right recursion

(3) base clause on top, left recursion

```
descendant(X, Y) :- child(X, Y).
descendant(X, Y) :- descendant(Z, Y), child(X, Z).
```

(4) base clause not on top, left recursion

```
descendant(X, Y) :- descendant(Z, Y), child(X, Z).
descendant(X, Y) :- child(X, Y).
```

The previous examples showed that the declarative meaning of a program can differ from its **procedural meaning**. The **order of the goals** as well as the **order of the rules** does affect its procedural meaning.

Remark 7.59 (Recursion)

- *Recursion is powerful and dangerous.*
- *Try to avoid left-recursion.*
- *Rule and predicate ordering is important.*
- *Try to put base clause before recursive clause.*
- *Put a recursive call as far as possible to the right of a rule.*

Remark 7.60 (Inductive definitions)

Consider the following inductive definition *of the natural numbers:*

1. 0 *is a natural number.*
2. *If* n *is a natural number, then* $n + 1$ *is a natural number.*
3. *There are no more natural numbers then those obtained by applying the two rules above.*

This definition consists of three parts: the **base case**, *the* **induction step**, *and a* **maximality condition**.

Remark 7.61

Our previous examples followed such an inductive definition, for example:

Basic clause:	`member(X, [X|_]).`
Inductive clause:	`member(X, [_|Tail]) :- member(X, Tail).`
Extremal clause:	*?*

Why don't we have an extremal clause? Think about negation-as-finite-failure and how ***PROLOG*** *derives new facts.*

Lists

PROLOG supports the data structure of lists. Lists can store (compound) terms. How does a list look like? Suppose we have a functor ./2 which allows to "concatenate" terms:

[]	: the empty list
.(a, [])	: corresponds to the list a.
.(a, .(b, []))	: corresponds to the list a, b.
.(a, .(b, .(c, [])))	: corresponds to the list a, b, c.

We use the function ./2 to construct recursive data structures: lists. [] is a constant denoting the empty list.

Definition 7.62 (List)

A list is a term $.(t_1, .(t_2, .(\ldots, .(t_k, [])))))$ where each t_i, $i = 1, k$ is a term, or $[]$, the empty list. The term t_1 is the head of the list, and the list $.(t_2, .(\ldots, .(t_k, [])))))$ is the tail of the list. The empty list has neither a tail nor a head.

This notation is cumbersome. **PROLOG** offers alternative ways to define lists. For example:

- $[t_1, \ldots, t_k]$: List of k elements
- $[\texttt{Head}|\texttt{Tail}]$ list with head `Head` and tail `Tail`.

Definition 7.63 (List notation)

In **PROLOG** a list can also be denoted by the following two predicates:

- $[t_1, \ldots, t_k]$ is a shortcut for $.(t_1, .(t_2, .(\ldots, .(t_k, [])\ldots)))$, and
- $[\mathtt{Head}|\mathtt{Tail}]$ is a shortcut for $.(\mathtt{Head}, \mathtt{Tail})$.

Further shortcuts are:

- $[s_1, \ldots, s_m|[t_1, \ldots, t_n]]$ is written as $[s_1, \ldots, s_m, t_1, \ldots, t_n]$, $m > 0, n \geq 0$ and.
- $[s_1, \ldots, s_m|[t_1, \ldots, t_n|X]]$ is written as $[s_1, \ldots, s_m, t_1, \ldots, t_n|X]$, $m, n > 0$.

In the following we consider some examples for list operations

Example 7.64

If possible, rewrite the following into lists of type $[t_1, \ldots, t_n]$:

1. $[1 \mid [2,3,4]]$
   $[1,2,3,4]$
2. $[1,2,3 \mid []]$
   $[1,2,3]$
3. $[1 \mid 2,3,4]$
   not a list
4. $[[] \mid []]$
   $[[]]$
5. $[[1,2],[3,4] \mid [5,6,7]]$
   $[[1,2],[3,4],5,6,7]$

Example 7.65 (Checking membership)

A program which checks whether a term is a member of a list.

```
member(X, [X|Tail]).
member(X, [Y|Tail]) :- member(X, Tail).
```

This program is equivalent to.

```
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
```

The predicate `member/2` is pre-built in **PROLOG**. Note that `member(a, [a|b])` can be derived even when `[a|b]` is not a list. Again, this is due to efficiency.

What does `?- member(a, X)` return?

Example 7.66 (Appending lists)

We define a predicate `append/3` where `append(X, Y, Z)` is true if list `X` combined with list `Y` is list `Z`.

```
append([], X, X).
append([X|Y], Z, [X|W]) :- append(Y, Z, W).
```

The predicate can be used in different ways. Explain the use of the following queries:

1. `?- append([a, 2], [b, 3], [a, 2, b, b]).`
2. `?- append([a, b], [c, d], X).`
3. `?- append(X, Y, [a, b, c, d]).`

Write down the search trees.

Example 7.67 (Last element of a list)

We define a predicate `last/2` which is true if the first argument is the last element of the list in the second argument:

```
last(X, [X]).
last(X, [H|T]) :- last(X, T).
```

Write a predicate `allSame(L)`, where `L` is a list, which is true if all elements in `L` are identical.

The following program defines the predicate:

```
allSame([]).
allSame([X]).
allSame([H1|[H2|T]]) :- H1 = H2, allSame([H2|T]).
```

This can further be simplified:

```
allSame([]).
allSame([_]).
allSame([X,X|T]]) :- allSame([X|T]).
```

Example 7.68 (Removing duplicate elements and cut)

```
(1) remove_duplicates([], []).
(2) remove_duplicates([H|T], Res) :- member(H, T),
                remove_duplicates(T, Res).
(3) remove_duplicates([H|T], [H|Res]) :-
                remove_duplicates(T, Res).
```

We replace the second clause by

```
(4) remove_duplicates([H|T], Res) :- member(H, T), !,
                remove_duplicates(T, Res).
```

7.6 Arithmetic

Arithmetic

We first make the following remark:

Remark 7.69 (Arithmetic)

PROLOG *is already Turing complete without arithmetic. We can simulate arithmetic with programs and define arithmetic operators by predicates (viewing numbers as compound terms). For example, we can write a recursive predicate* `plus(A,B,C)` *to represent* $A + B = C$. **So we don't need arithmetic.**

However, using arithmetic improves readability, allows for easier (to read) programs, and can be implemented (in hardware) much more efficiently.

PROLOG allows a whole bunch of arithmetic functors (on purpose we don't say arithmetic operators). We list a few with their reading:

$+/2$	addition
$-/2$	subtraction
$-/1$	negative number
$*/2$	multiplication
$//2$	(floating point) division
$///2$	(integer) division
$\texttt{mod}/2$	remainder after division
$\texttt{max}/2$	maximum of two numbers

It is important to note that $4 + 3$ is not 7 but it is a **PROLOG term**!

Definition 7.70 (Arithmetic expression)

An arithmetic expression is a term constructed from numbers, variables and arithmetic functors. The arithmetic functors are usually written in **infix** notation, i.e. $5 + 2$ instead of $+(5, 2)$.

We are able to write down arithmetic expressions, but how to evaluate them?

PROLOG offers built-in predicates to evaluate and compare arithmetic terms. The most important are:

`is/2`	right evaluation
`=:= /2`	left-right evalution
`=\= /2`	inequality
`< /2`	(strictly) larger
`=< /2`	larger or equal
`> /2`	(strictly) smaller
`>= /2`	smaller or equal

The reading of t_1 `is` t_2 is that t_1 equals the evalution of t_2 where t_2 must be a ground artihmetic constraint at the time of evaluation.
The reading of t_1 `=:=` t_2 is that t_1 evaluates to t_2 and vice versa. Both terms must be ground at the time of evaluation.
The reading of t_1`=\=`t_2 is that t_1 does not evaluate to t_2 or vice versa.
The other predicates have their usual interpretation.

Example 7.71

Which of the following expressions are true/false?

- `2 + 2 = 2 + 2`
- `4 = 2 + 2`
- `X = 2 + 2`
  What is the instantiation of `X`?
- `4 is 2 + 2`
- `2 + 2 is 4`
- `X =:= 2 + 2`
- `3 + 2 =:= 2 + 3`

Example 7.72

Write the following in **PROLOG**

- The remainder of 7 divided by 2 is 1:

```
1 is 7 mod 2.
```

- The value of 2^{10} is 1024:

```
1014 is 2 * *10.
```

- Compute maximum of 4 and 89:

```
X is max(4, 89).
```

Remark 7.73

Example 7.74 (Factorial)

We define a predicate `fac/2` which computes the factorial of a number n:

```
fac(0,1).
fac(N,F) :- N > 0, M is N - 1, fac(M,Fm), F is N * Fm.
```

Then, we query `?- fac(n,X).` Variable `X` contains the factorial of `n`.

Example 7.75 (Length of a list)

We define a predicate `length/2` which computes the length of a list:

```
length([], 0).
length([H|Tail], N) :- length(Tail, P), N is P + 1.
```

Example 7.76 (For-loop)

We simulate a for-loop with recursion

```
printnr(1) :- writeln(1).
printnr(N) :- N > 1, M is N - 1, printnr(M), writeln(N).
```

Example 7.77 (Sum numbers in a list)

We would like to define a predicate `sum/2` which sums up all numbers in a list. What about the two programs

```
sum([], 0).
sum([H|Tail], Sum) :- sum(Tail, Tailsum), Sum is H + Tailsum.
```

and

```
sum([], 0).
sum([H|Tail], Sum) :- Sum is H + Tailsum, sum(Tail, Tailsum).
```

Why does the former program work but not the latter?

7.7 Programming

Comments and Modes of Variables

Example 7.78 (Computing the power)

Write a **PROLOG** program which computes:

$$X \text{ to the power of } N \text{ and return/store in } P$$

- We define a predicate `power(X, N, P)`.
- How to find out about the purpose of the predicate and how to use it? *Add comments:*

```
% Computes X to the power of N
% and return/store in P
power(_, 0, 1).
power(X, N, P) :- N > 0, M is N - 1, power(X, M, MP),
                  P is X * MP.
```

```
% Computes X to the power of N
% and return/store in P
power(_, 0, 1).
power(X, N, P) :- N > 0, M is N - 1, power(X, M, MP),
                  P is X * MP.
```

- Is this everything we need? What if we query `power(2, Y, 8)`? That is, we want to know which power x is needed such that $2^x = 8$.
- We get an error message:
  `>/2: Arguments are not sufficiently instantiated`.
  Why?
- It is possible to query `power(2, 10, P)` where P is instantiated to the solution.

```
% Computes X to the power of N
% and return/store in P
power(_, 0, 1).
power(X, N, P) :- N > 0, M is N - 1, power(X, M, MP),
                  P is X * MP.
```

- X and N are input variables, they need to be ground when evaluated.
- P can be an input as well as an output variable.
- We use $+$, $-$ and $?$ to indicate **input**, **output**, and **input/output variables**, respectively.
- We add the comment explaining the definition of the predicate:
  `% power(+Integer1,+Integer2,?Integer3)`

Definition 7.79 (Predicates, mode)

When defining a predicate `p(t1,...,tN)` we group the definition of the predicate and put above a comment clarifying the type of a variable together with its **mode**, i.e. whether it is an input (+), output (-) or input/output (?) variable. For example, it could look as follows:

```
% p(+Int1, -Int2, ...,?IntN)
```

Note: Formally this is just a comment!

Example 7.80

- We have already seen that the misuse of input variables can yield errors.
- What if we use a ground term as output variable? The following examples show that one has to be careful:

```
% sort(+List,-List)
?- sort([3,1,2],[1,2,3])
true


% sort(+List,-List)
?- sort([A,B,3],[2,1,3])
A = 2, B = 1.
```

- In the **PROLOG** manual the types indicate the following:
  - \+ argument must be fully instantiated
  - \- argument should be unbound
  - ? argument bound to a term of the specified type
- Declaration of variables is important when programming **for readability**.

Collecting Solutions

`findall/3`: collects all objects in a list that satisfy a given goal,

`bagof/3`: (clustered solutions),

`setof/3`: (no duplicates).

In the following two definitions, we consider the program:

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), descend(Z,Y).
```

Definition 7.81

`findall(X, G, L)` collects in L

- all instantiations of X that
- correspond to solutions of G

The query `?- findall(X,descend(martha,X),Z).` returns
`L=[charlotte, caroline, laura, rose]`

If there are no instantiations that match the goal the empty list is returned. Note that X can be an arbitrary term. The list can contain **duplicate entries**.

Definition 7.82

`bagof(X, G, L)` collects in L

- all instantiations of X that
- correspond to solutions of G
- and clusters these solutions in a separate list for all instantiations for variables in G not contained in X.

If there are no instantiations matching the goal, `fail` is returned.

The query `?- bagof(X,descend(Y,X),Z).` returns

```
Y = caroline,
L = [laura, rose]
Y = charlotte,
L = [caroline, laura, rose]
Y = laura,
L = [rose]
Y = martha,
L = [charlotte, caroline, laura, rose].
```

Definition 7.83

`setof(X,G,L)` behaves like `bagof(X,G,L)` but

- each list is sorted alphabetically and
- duplicate entries are removed.

Costly operations

It is important to note that these three operations are costly in terms of computation time. **Use them carefully.**

Anonymous variables

On query `?- sister(X,Y)` Prolog returns all pairs `X` and `Y` such that `X` is a sister of `Y`.

We may only be interested in knowing who has a sister, i.e. in the instantiations of `Y`. In this case we can use the **anonymous variable**:

```
?- sister(_,Y)
```

Prolog only returns instantiations for `Y`. Thus, the anonymous variable is used to hide specific instantiation.

The anonymous variable can be used at several places in a query: **each occurrence of _ is independent** of the others.

Example 7.84 (Anonymous variable)

Consider the program:

```
?- [X1,X2,X3,X4| Tail]=[[], d(z),[2,[b,c]],[],Z].
X1 = X4, X4 = [],
X2 = d(z),
X3 = [2, [b, c]],
Tail = [Z].
```

We may only want to know the value of the second and fourth position on the list: Only interested in 2nd and 4th component:

```
?- [_,X2,_,X4|_]=[[], d(z),[2,[b,c]],[],Z].
X2 = d(z),
X4 = [].
```

Compare the difference to (occurrences of _ are independent):

```
?- [Y,X2,Y,X4|Y]=[[], d(z),[2,[b,c]],[],Z].
false.
```

Tail recursion and accumulators

```
append([], X, X).
append([X|Y], Z, [X|W]) :- append(Y, Z, W).
```

Example 7.85 (Reversing a list)

We would like to reverse a list. Consider the following program, where append is defined as before.

```
naiveReverse([],[]).
naiveReverse([H|T],R):- naiveReverse(T,RTemp),
                        append(RTemp,[H],R).
```

Query `?- naiveReverse([1,2,3,4,5,6,7,8],L).` Have a look at the trace, what can you observe?

The program is **very inefficient**. Can we do better?

Consider the following approach to reverse a list:

`List: [a,b,c,d]`	`Accumulator: []`
`List: [b,c,d]`	`Accumulator: [a]`
`List: [c,d]`	`Accumulator: [b,a]`
`List: [d]`	`Accumulator: [c,b,a]`
`List: []`	`Accumulator: [d,c,b,a]`

Example 7.86 (More efficient program for Reverse)

We first define an **accumulator**:

```
accRev([H|T],A,R) :- accRev(T,[H|A],R).
accRev([],A,A).
```

The predicate to reverse the list is: `rev(L,R) :- accRev(L,[],R).`
Compare the trace of `?- rev([1,2,3,4,5,6,7,8],L).` to the corresponding trace of Example 7.85.

Compare the traces of `naiveReverse([1,2,3,4,5],L).` (cf. Fig. 7.38) and `rev([1,2,3,4,5],L).` (cf. Figure 7.39)

```
[trace] 74 ?- naiveReverse([1,2,3,4,5],L).
   Call: (6) naiveReverse([1, 2, 3, 4, 5], _G1207) ? creep
   Call: (7) naiveReverse([2, 3, 4, 5], _G1298) ? creep
   Call: (8) naiveReverse([3, 4, 5], _G1298) ? creep
   Call: (9) naiveReverse([4, 5], _G1298) ? creep
   Call: (10) naiveReverse([5], _G1298) ? creep
   Call: (11) naiveReverse([], _G1298) ? creep
   Exit: (11) naiveReverse([], []) ? creep
   Call: (11) append([], [5], _G1302) ? creep
   Exit: (11) append([], [5], [5]) ? creep
   Exit: (10) naiveReverse([5], [5]) ? creep
   Call: (10) append([5], [4], _G1305) ? creep
   Call: (11) append([], [4], _G1297) ? creep
   Exit: (11) append([], [4], [4]) ? creep
   Exit: (10) append([5], [4], [5, 4]) ? creep
   Exit: (9) naiveReverse([4, 5], [5, 4]) ? creep
   Call: (9) append([5, 4], [3], _G1311) ? creep
   Call: (10) append([4], [3], _G1303) ? creep
   Call: (11) append([], [3], _G1306) ? creep
   Exit: (11) append([], [3], [3]) ? creep
   Exit: (10) append([4], [3], [4, 3]) ? creep
   Exit: (9) append([5, 4], [3], [5, 4, 3]) ? creep
   Exit: (8) naiveReverse([3, 4, 5], [5, 4, 3]) ? creep
   Call: (8) append([5, 4, 3], [2], _G1320) ? creep
   Call: (9) append([4, 3], [2], _G1312) ? creep
   Call: (10) append([3], [2], _G1315) ? creep
   Call: (11) append([], [2], _G1318) ? creep
   Exit: (11) append([], [2], [2]) ? creep
   Exit: (10) append([3], [2], [3, 2]) ? creep
   Exit: (9) append([4, 3], [2], [4, 3, 2]) ? creep
   Exit: (8) append([5, 4, 3], [2], [5, 4, 3, 2]) ? creep
   Exit: (7) naiveReverse([2, 3, 4, 5], [5, 4, 3, 2]) ? creep
   Call: (7) append([5, 4, 3, 2], [1], _G1207) ? creep
   Call: (8) append([4, 3, 2], [1], _G1324) ? creep
   Call: (9) append([3, 2], [1], _G1327) ? creep
   Call: (10) append([2], [1], _G1330) ? creep
   Call: (11) append([], [1], _G1333) ? creep
   Exit: (11) append([], [1], [1]) ? creep
   Exit: (10) append([2], [1], [2, 1]) ? creep
   Exit: (9) append([3, 2], [1], [3, 2, 1]) ? creep
   Exit: (8) append([4, 3, 2], [1], [4, 3, 2, 1]) ? creep
   Exit: (7) append([5, 4, 3, 2], [1], [5, 4, 3, 2, 1]) ? creep
   Exit: (6) naiveReverse([1, 2, 3, 4, 5], [5, 4, 3, 2, 1]) ? creep
L = [5, 4, 3, 2, 1].
```

Figure 7.38: Trace of naive reverse.

```
[trace] 73 ?- rev([1,2,3,4,5],L).
   Call: (6) rev([1, 2, 3, 4, 5], _G749) ? creep
   Call: (7) accRev([1, 2, 3, 4, 5], [], _G749) ? creep
   Call: (8) accRev([2, 3, 4, 5], [1], _G749) ? creep
   Call: (9) accRev([3, 4, 5], [2, 1], _G749) ? creep
   Call: (10) accRev([4, 5], [3, 2, 1], _G749) ? creep
   Call: (11) accRev([5], [4, 3, 2, 1], _G749) ? creep
   Call: (12) accRev([], [5, 4, 3, 2, 1], _G749) ? creep
   Exit: (12) accRev([], [5, 4, 3, 2, 1], [5, 4, 3, 2, 1]) ? creep
   Exit: (11) accRev([5], [4, 3, 2, 1], [5, 4, 3, 2, 1]) ? creep
   Exit: (10) accRev([4, 5], [3, 2, 1], [5, 4, 3, 2, 1]) ? creep
   Exit: (9) accRev([3, 4, 5], [2, 1], [5, 4, 3, 2, 1]) ? creep
   Exit: (8) accRev([2, 3, 4, 5], [1], [5, 4, 3, 2, 1]) ? creep
   Exit: (7) accRev([1, 2, 3, 4, 5], [], [5, 4, 3, 2, 1]) ? creep
   Exit: (6) rev([1, 2, 3, 4, 5], [5, 4, 3, 2, 1]) ? creep
L = [5, 4, 3, 2, 1].
```

Figure 7.39: Trace of reverse with accumulator.

Tidy your room!

A very simple example showing in blocksworld-like style, how to deal with actions as elements of a list.

Example 7.87

Your room is very untidy. You should make it nice as you are expecting friends to come over. You have three simple actions: to pick, to move and to drop an object.

Write a binary predicate that takes a list of objects and generates a list of actions (pick, move to box, drop) for each object.

Refine your program by putting the objects into different boxes according to their colors.

Tidy your room! (cont.)

```
tidy([],[]).
tidy([Obj|Oth],[pick(Obj),move(Obj,box),drop(Obj)|OthActs])
:- tidy(Oth,OthActs).

tidy([],[]).
tidy([Obj|Oth],[pick(Obj),move(Obj,box1),drop(Obj)|OtActs])
:- tidy(Oth,OthActs), red(Obj).
tidy([Obj|Oth],[pick(Obj),move(Obj,box2),drop(Obj)|OthActs])
:- tidy(Oth,OthActs), green(Obj).
tidy([Obj|Oth],[pick(Obj),move(Obj,box3),drop(Obj)|OthActs])
:- tidy(Oth,OthActs), blue(Obj).
```