General Computer Science
320101 — Fall 2016

Michael Kohlhase

Computer Science
Jacobs University, Bremen Germany
m.kohlhase@jacobs-university.de
office: Room 168@Research 1, phone: x3140

December 8, 2016

Preface

This Document

This document contains the course notes for the course General Computer Science I & II held at Jacobs University Bremen[1] in the academic years 2003-2016.

Contents: The document mixes the slides presented in class with comments of the instructor to give students a more complete background reference.

Caveat: This document is made available for the students of this course only. It is still a draft and will develop over the course of the current course and in coming academic years.

Licensing: This document is licensed under a Creative Commons license that requires attribution, allows commercial use, and allows derivative works as long as these are licensed under the same license.

Knowledge Representation Experiment: This document is also an experiment in knowledge representation. Under the hood, it uses the sTeX package [Koh08, Koh16], a TeX/LaTeX extension for semantic markup, which allows to export the contents into the eLearning platform PantaRhei.

Comments and extensions are always welcome, please send them to the author.

Other Resources: The course notes are complemented by a selection of problems (with and without solutions) that can be used for self-study. [Koh11a, Koh11b]

Course Concept

Aims: The course 320101/2 “General Computer Science I/II” (GenCS) is a two-semester course that is taught as a mandatory component of the “Computer Science” and “Electrical Engineering & Computer Science” majors (EECS) at Jacobs University. The course aims to give these students a solid (and somewhat theoretically oriented) foundation of the basic concepts and practices of computer science without becoming inaccessible to ambitious students of other majors.

Context: As part of the EECS curriculum GenCS is complemented with a programming lab that teaches the basics of `C` and `C`$^{++}$ from a practical perspective and a “Computer Architecture” course in the first semester. As the programming lab is taught in three five-week blocks over the first semester, we cannot make use of it in GenCS.

In the second year, GenCS, will be followed by a standard “Algorithms & Data structures” course and a “Formal Languages & Logics” course, which it must prepare.

Prerequisites: The student body of Jacobs University is extremely diverse — in 2011, we have students from 110 nations on campus. In particular, GenCS students come from both sides of the “digital divide”: Previous CS exposure ranges “almost computer-illiterate” to “professional `Java` programmer” on the practical level, and from “only calculus” to solid foundations in discrete Mathematics for the theoretical foundations. An important commonality of Jacobs students however is that they are bright, resourceful, and very motivated.

As a consequence, the GenCS course does not make any assumptions about prior knowledge, and introduces all the necessary material, developing it from first principles. To compensate for this, the course progresses very rapidly and leaves much of the actual learning experience to homework problems and student-run tutorials.

Course Contents

Goal: To give students a solid foundation of the basic concepts and practices of Computer Science we try to raise awareness for the three basic concepts of CS: “data/information”, “algorithms/programs” and “machines/computational devices” by studying various instances, exposing more and more characteristics as we go along.

[1] International University Bremen until Fall 2006

Computer Science: In accordance to the goal of teaching students to "think first" and to bring out the Science of CS, the general style of the exposition is rather theoretical; practical aspects are largely relegated to the homework exercises and tutorials. In particular, almost all relevant statements are proven mathematically to expose the underlying structures.

GenCS is not a programming course: even though it covers all three major programming paradigms (imperative, functional, and declarative programming). The course uses `SML` as its primary programming language as it offers a clean conceptualization of the fundamental concepts of recursion, and types. An added benefit is that `SML` is new to virtually all incoming Jacobs students and helps equalize opportunities.

GenCS I (the first semester): is somewhat oriented towards computation and representation. In the first half of the semester the course introduces the dual concepts of induction and recursion, first on unary natural numbers, and then on arbitrary abstract data types, and legitimizes them by the Peano Axioms. The introduction and of the functional core of `SML` contrasts and explains this rather abstract development. To highlight the role of representation, we turn to Boolean expressions, propositional logic, and logical calculi in the second half of the semester. This gives the students a first glimpse at the syntax/semantics distinction at the heart of CS.

GenCS II (the second semester): is more oriented towards exposing students to the realization of computational devices. The main part of the semester is taken up by a "building an abstract computer", starting from combinational circuits, via a register machine which can be programmed in a simple assembler language, to a stack-based machine with a compiler for a bare-bones functional programming language. In contrast to the "computer architecture" course in the first semester, the GenCS exposition abstracts away from all physical and timing issues and considers circuits as labeled graphs. This reinforces the students' grasp of the fundamental concepts and highlights complexity issues. The course then progresses to a brief introduction of Turing machines and discusses the fundamental limits of computation at a rather superficial level, which completes an introductory "tour de force" through the landscape of Computer Science. As a contrast to these foundational issues, we then turn practical introduce the architecture of the Internet and the World-Wide Web.

The remaining time, is spent on studying one class algorithms (search algorithms) in more detail and introducing the notition of declarative programming that uses search and logical representation as a model of computation.

Acknowledgments

Materials: Some of the material in this course is based on course notes prepared by Andreas Birk, who held the course 320101/2 "General Computer Science" at IUB in the years 2001-03. Parts of his course and the current course materials were based on the book "Hardware Design" (in German) [KP95]. The section on search algorithms is based on materials obtained from Bernhard Beckert (Uni Koblenz), which in turn are based on Stuart Russell and Peter Norvig's lecture slides that go with their book "Artificial Intelligence: A Modern Approach" [RN95].

The presentation of the programming language Standard ML, which serves as the primary programming tool of this course is in part based on the course notes of Gert Smolka's excellent course "Programming" at Saarland University [Smo08].

Contributors: The preparation of the course notes has been greatly helped by Ioan Sucan, who has done much of the initial editing needed for semantic preloading in sTEX. Herbert Jaeger, Christoph Lange, and Normen Müller have given advice on the contents.

GenCS Students: The following students have submitted corrections and suggestions to this and earlier versions of the notes: Saksham Raj Gautam, Anton Kirilov, Philipp Meerkamp, Paul Ngana, Darko Pesikan, Stojanco Stamkov, Nikolaus Rath, Evans Bekoe, Marek Laska, Moritz Beber, Andrei Aiordachioaie, Magdalena Golden, Andrei Eugeniu Ioniţă, Semir Elezović, Dimitar Asenov, Alen Stojanov, Felix Schlesinger, Ştefan Anca, Dante Stroe, Irina Calciu, Nemanja Ivanovski, Abdulaziz Kivaza, Anca Dragan, Razvan Turtoi, Catalin Duta, Andrei Dragan, Dimitar

Misev, Vladislav Perelman, Milen Paskov, Kestutis Cesnavicius, Mohammad Faisal, Janis Beckert, Karolis Uziela, Josip Djolonga, Flavia Grosan, Aleksandar Siljanovski, Iurie Tap, Barbara Khalibinzwa, Darko Velinov, Anton Lyubomirov Antonov, Christopher Purnell, Maxim Rauwald, Jan Brennstein, Irhad Elezovikj, Naomi Pentrel, Jana Kohlhase, Victoria Beleuta, Dominik Kundel, Daniel Hasegan, Mengyuan Zhang, Georgi Gyurchev, Timo Lücke, Sudhashree Sayenju, Lukas Kohlhase, Dmitrii Cucleschin, Aleksandar Gyorev, Tyler Buchmann, Bidesh Thapaliya, Dan Daniel Erdmann-Pham, Petre Munteanu, Utkrist Adhikari, Kim Philipp Jablonski, Aleksandar Gyorev, Tom Wiesing, Sourabh Lal, Nikhil Shakya, Otar Bichiashvili, Cornel Amariei, Enxhell Luzhnica, Rubin Deliallisi, Mariia Gladkova, Nikolay Raykov, Andreea Ciuprina, Dennis Ledwon.

Recorded Syllabus for 2016

In this document, we record the progress of the course in the academic year 2016 in the form of a "recorded syllabus", i.e. a syllabus that is created after the fact rather than before.

Recorded Syllabus Fall Semester 2016:

#	date	until	slide	page
1	Sep 1.	almost through with admin	8	7
2	Sep 6.	at end of motivation	28	19
3	Sep 8.	proofs with Peano axioms	34	27
4	Sep 13.	Defining Equations	40	31
5	Sep 15.	MathTalk, Greek letters & Sets	46	35
6	Sep 20.	Relations	50	38
7	Sep 22.	properties of functions	56	41
8	Sep 27.	SML basics	61	47
9	Sep 29	defining functions by cases	67	50
10	Oct 4.	reflections on recursively defined functions	78	57
11	Oct 6.	abstract data types	88	66
12	Oct 11.	constructor terms & abstract interpreter	93	70
13	Oct 13.	substitutions & subterms	103	76
14	Oct 18.	abstract vs. concrete procedures	111	82
15	Oct 20.	Programming with exceptions	122	89
16	Nov 1.	Prefix codes	133	99
17	Nov 1.	Unicode	138	103
18	Nov 3.	Semantics of Boolexp	147	110
19	Nov 8.	CNF & DNF	155	114
	Nov 10. Midterm			
20	Nov 15.	Cost Bounds for Boolean Expressions	165	120
21	Nov 15.	Quine McCluskey	181	127
22	Nov 22.	Intro to Propositional Logic	188	132
23	Nov 22.	The miracle of logics	192	136
24	Nov. 29.	Natural Deduction for Propositional Logic	206	144
25	Nov. 29.	Analytic Tableaux	223	156
26	Dec 3.	Intellectual Property, Copyright, & Licensing	252	179

Here the syllabus of of the last academic year for reference, the current year should be similar; see the course notes of last year available for reference at `http://kwarc.info/teaching/GenCS/notes2015.pdf`.

Recorded Syllabus Fall Semester 2015:

#	date	until
1	Sep 1.	through with admin
2	Sep 3.	at end of motivation
3	Sep 8.	proofs with Peano axioms
5	Sep 10.	Defining Equations
6	Sep 15.	MathTalk and Greek letters
7	Sep 17.	Properties of relations
8	Sep 22.	lambda-notation
9	Sep 24.	SML function by cases
8	Sep 29.	higher-order functions
9	Oct 1.	inductively defined sets
10	Oct 6.	SML data types
11	Oct 8.	constructor terms & abstract interpreter
12	Oct 13.	terms & subterms
13	Oct 15.	recursion & termination
14	Oct 22.	Finishing Up SML
	Oct 27	Midterm
15	Oct 29.	Prefix codes
16	Nov 3.	UTF encodings
17	Nov 5.	Semantics of Boolexp
18	Nov. 10.	CNF & DNF
19	Nov 12.	Cost Bounds for Boolean Expressions
20	Nov 17.	Quine McCluskey
21	Nov 19.	Intro to Propositional Logic
22	Nov 24.	The miracle of logics
23	Nov 26 .	Natural Deduction for Propositional Logic
24	Dec 1.	Analytic Tableaux
25	Dec 3.	Intellectual Property, Copyright, & Information Privacy

Contents

Chapter 1

Getting Started with “General Computer Science”

Jacobs University offers a unique CS curriculum to a special student body. Our CS curriculum is optimized to make the students successful computer scientists in only three years (as opposed to most US programs that have four years for this). In particular, we aim to enable students to pass the GRE subject test in their fifth semester, so that they can use it in their graduate school applications.

The Course 320101/2 “General Computer Science I/II” is a one-year introductory course that provides an overview over many of the areas in Computer Science with a focus on the foundational aspects and concepts. The intended audience for this course are students of Computer Science, and motivated students from the Engineering and Science disciplines that want to understand more about the “why” rather than only the “how” of Computer Science, i.e. the “science part”.

1.1 Overview over the Course

Overview: The purpose of this two-semester course is to give you an introduction to what the Science in “Computer Science” might be. We will touch on a lot of subjects, techniques and arguments that are of importance. Most of them, we will not be able to cover in the depth that you will (eventually) need. That will happen in your second year, where you will see most of them again, with much more thorough treatment.

Computer Science: We are using the term “Computer Science” in this course, because it is the traditional anglo-saxon term for our field. It is a bit of a misnomer, as it emphasizes the computer alone as a computational device, which is only one of the aspects of the field. Other names that are becoming increasingly popular are “Information Science”, “Informatics” or “Computing”, which are broader, since they concentrate on the notion of information (irrespective of the machine basis: hardware/software/wetware/alienware/vaporware) or on computation.

Definition 1.1.1 What we mean with Computer Science here is perhaps best represented by the following quote:

> The body of knowledge of computing is frequently described as the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, What can be (efficiently) automated? [Den00]

Plot of “General Computer Science”

- ▷ Today: Motivation, Admin, and find out what you already know
  - ▷ What is Computer Science?
  - ▷ Information, Data, Computation, Machines
  - ▷ a (very) quick walk through the topics
- ▷ Get a feeling for the math involved (⚠ not a programming course!!! ⚠)
  - ▷ learn mathematical language (so we can talk rigorously)
  - ▷ inductively defined sets, functions on them
  - ▷ elementary complexity analysis
- ▷ Various machine models (as models of computation)
  - ▷ (primitive) recursive functions on inductive sets
  - ▷ combinational circuits and computer architecture
  - ▷ Programming Language: Standard ML (great equalizer/thought provoker)
- ▷ Representing knowledge in formal languages and reasoning about them
  - ▷ formal languages and their operations
  - ▷ syntax vs. semantics (form vs. function)
  - ▷ inferenced systems for understanding (logical) argumenation

Not a Programming Course: Note “General CS” is not a programming course, but an attempt to give you an idea about the “Science” of computation. Learning how to write correct, efficient, and maintainable, programs is an important part of any education in Computer Science, but we will not focus on that in this course (we have the Labs for that). As a consequence, we will not concentrate on teaching how to program in “General CS” but introduce the `SML` language and assume that you pick it up as we go along (however, the tutorials will be a great help; so go there!).

Standard ML: We will be using Standard ML (`SML`), as the primary vehicle for programming in the course. The primary reason for this is that as a functional programming language, it focuses more on clean concepts like recursion or typing, than on coverage and libraries. This teaches students to “think first” rather than “hack first”, which meshes better with the goal of this course. There have been long discussions about the pros and cons of the choice in general, but it has worked well at Jacobs University (even if students tend to complain about `SML` in the beginning).

A secondary motivation for `SML` is that with a student body as diverse as the GenCS first-years at Jacobs[1] we need a language that equalizes them. `SML` is quite successful in that, so far none of the incoming students had even heard of the language (apart from tall stories by the older students).

Algorithms, Machines, and Data: The discussion in “General CS” will go in circles around the triangle between the three key ingredients of computation.

Algorithms are abstract representations of computation instructions

[1]traditionally ranging from students with no prior programming experience to ones with 10 years of semi-pro `Java`

Data are representations of the objects the computations act on

Machines are representations of the devices the computations run on

The figure below shows that they all depend on each other; in the course of this course we will look at various instantiations of this general picture.

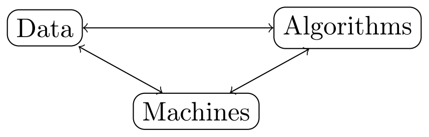

$$\begin{tikzcd}
\boxed{\text{Data}} \arrow[rr, leftrightarrow] \arrow[dr, leftrightarrow] & & \boxed{\text{Algorithms}} \\
& \boxed{\text{Machines}} \arrow[ur, leftrightarrow] &
\end{tikzcd}$$

Figure 1.1: The three key ingredients of Computer Science

Representation: One of the primary focal items in "General CS" will be the notion of *representation*. In a nutshell the situation is as follows: we cannot compute with objects of the "real world", but be have to make electronic counterparts that can be manipulated in a computer, which we will call representations. It is essential for a computer scientist to realize that objects and their representations are different, and to be aware of their relation to each other. Otherwise it will be difficult to predict the relevance of the results of computation (manipulating electronic objects in the computer) for the real-world objects. But if cannot do that, computing loses much of its utility.

Of course this may sound a bit esoteric in the beginning, but I will come back to this very often over the course, and in the end you may see the importance as well.

1.2 Administrativa

We will now go through the ground rules for the course. This is a kind of a social contract between the instructor and the students. Both have to keep their side of the deal to make learning and becoming Computer Scientists as efficient and painless as possible.

1.2.1 Grades, Credits, Retaking

Now we come to a topic that is always interesting to the students: the grading scheme. The grading scheme I am using has changed over time, but I am quite happy with it now.

Prerequisites, Requirements, Grades

- ▷ Prerequisites: Motivation, Interest, Curiosity, hard work
  - ▷ You can do this course if you want!
- ▷ Grades: (plan your work involvement carefully)

Tuesday Quizzes	30%
Graded Assignments	20%
Mid-term Exam	20%
Final Exam	30%

Note that for the grades, the percentages of achieved points are added with the weights above, and only then the resulting percentage is converted to a grade.

▷ Tuesday Quizzes: (Almost) every tuesday, we will use the first 10 minutes for a brief quiz about the material from the week before (you have to be there)

▷ Rationale: I want you to work continuously (maximizes learning)

▷ Requirements for Auditing: You can audit GenCS! (specify in Campus Net)

To earn an audit you have to take the quizzes and do reasonably well(I cannot check that you took part regularly otherwise.)

My main motivation in this grading scheme is to entice you to study continuously. You cannot hope to pass the course, if you only learn in the reading week. Let us look at the components of the grade. The first is the exams: We have a mid-term exam relatively early, so that you get feedback about your performance; the need for a final exam is obvious and tradition at Jacobs. Together, the exams make up 50% of your grade, which seems reasonable, so that you cannot completely mess up your grade if you fail one.

In particular, the 50% rule means that if you only come to the exams, you basically have to get perfect scores in order to get an overall passing grade. This is intentional, it is supposed to encourage you to spend time on the other half of the grade. The homework assignments are a central part of the course, you will need to spend considerable time on them. Do not let the 20% part of the grade fool you. If you do not at least attempt to solve all of the assignments, you have practically no chance to pass the course, since you will not get the practice you need to do well in the exams. The value of 20% is attempts to find a good trade-off between discouraging from cheating, and giving enough incentive to do the homework assignments. Finally, tuesday quizzes try to ensure that you will show up on time on tuesdays, and are prepared.

The (relatively severe) rule for auditing is intended to ensure that auditors keep up with the material covered in class. I do not have any other way of ensuring this (at a reasonable cost for me). Many students who think they can audit GenCS find out in the course of the semester that following the course is too much work for them. This is not a problem. An audit that was not awarded does not make any ill effect on your transcript, so feel invited to try.

Advanced Placement

▷ Generally: AP let's you drop a course, but retain credit for it(sorry no grade!)

- ▷ you register for the course, and take an AP exam
- ▷ you will need to have very good results to pass
- ▷ If you fail, you have to take the course or drop it!

▷ Specifically: AP exams (oral) some time next week (see me for a date)

- ▷ Be prepared to answer elementary questions about: discrete mathematics, terms, substitution, abstract interpretation, computation, recursion, termination, elementary complexity, Standard ML, types, formal languages, Boolean expressions (possible subjects of the exam)

▷ Warning: you should be very sure of yourself to try (genius in C^{++} insufficient)

Although advanced placement is possible, it will be very hard to pass the AP test. Passing an AP does not just mean that you have to have a passing grade, but very good grades in all the topics that we cover. This will be very hard to achieve, even if you have studied a year of Computer Science at another university (different places teach different things in the first year). You can still take the exam, but you should keep in mind that this means considerable work for the instrutor.

1.2.2 Homeworks, Submission, and Cheating

Homework assignments

- ▷ Goal: Reinforce and apply what is taught in class.
- ▷ Homeworks: will be small individual problem/programming/proof assignments (but take time to solve) group submission if and only if explicitly permitted
- ▷ Admin: To keep things running smoothly
  - ▷ Homeworks will be posted on PantaRhei
  - ▷ Homeworks are handed in electronically in `JGrader`(plain text, Postscript, PDF,. . .)
  - ▷ go to the tutorials, discuss with your TA (they are there for you!)
  - ▷ materials: sometimes posted ahead of time; then read before class, prepare questions, bring printout to class to take notes
- ▷ Homework Discipline:
  - ▷ start early! (many assignments need more than one evening's work)
  - ▷ Don't start by sitting at a blank screen
  - ▷ Humans will be trying to understand the text/code/math when grading it.

Homework assignments are a central part of the course, they allow you to review the concepts covered in class, and practice using them. They are usually directly based on concepts covered in the lecture, so reviewing the course notes often helps getting started.

Homework Submissions, Grading, Tutorials

- ▷ Submissions: We use Heinrich Stamerjohanns' `JGrader` system
  - ▷ submit all homework assignments electronically to `https://jgrader.de`.
  - ▷ you can login with your Jacobs account and password. (should have one!)
  - ▷ feedback/grades to your submissions
  - ▷ get an overview over how you are doing! (do not leave to midterm)
- ▷ Tutorials: select a tutorial group and actually go to it regularly
  - ▷ to discuss the course topics after class (lectures need pre/postparation)
  - ▷ to discuss your homework after submission (to see what was the problem)
  - ▷ to find a study group (probably the most determining factor of success)

The next topic is very important, you should take this very seriously, even if you think that this is just a self-serving regulation made by the faculty.

All societies have their rules, written and unwritten ones, which serve as a social contract among its members, protect their interestes, and optimize the functioning of the society as a whole. This is also true for the community of scientists worldwide. This society is special, since it balances intense cooperation on joint issues with fierce competition. Most of the rules are largely unwritten; you are expected to follow them anyway. The code of academic integrity at Jacobs is an attempt to put some of the aspects into writing.

It is an essential part of your academic education that you learn to behave like academics, i.e. to function as a member of the academic community. Even if you do not want to become a scientist in the end, you should be aware that many of the people you are dealing with have gone through an academic education and expect that you (as a graduate of Jacobs) will behave by these rules.

The Code of Academic Integrity

- ▷ Jacobs has a “Code of Academic Integrity”
  - ▷ this is a document passed by the Jacobs community (our law of the university)
  - ▷ you have signed it during enrollment (we take this seriously)
- ▷ It mandates good behaviors from both faculty and students and penalizes bad ones:
  - ▷ honest academic behavior (we don’t cheat/falsify)
  - ▷ respect and protect the intellectual property of others (no plagiarism)
  - ▷ treat all Jacobs members equally (no favoritism)
- ▷ this is to protect you and build an atmosphere of mutual respect
  - ▷ academic societies thrive on reputation and respect as primary currency
- ▷ The Reasonable Person Principle (one lubricant of academia)
  - ▷ we treat each other as reasonable persons
  - ▷ the other’s requests and needs are reasonable until proven otherwise
  - ▷ but if the other violates our trust, we are deeply disappointed (severe uncompromising consequences)

To understand the rules of academic societies it is central to realize that these communities are driven by economic considerations of their members. However, in academic societies, the primary good that is produced and consumed consists in ideas and knowledge, and the primary currency involved is academic reputation[2]. Even though academic societies may seem as altruistic — scientists share their knowledge freely, even investing time to help their peers understand the concepts more deeply — it is useful to realize that this behavior is just one half of an economic

[2]Of course, this is a very simplistic attempt to explain academic societies, and there are many other factors at work there. For instance, it is possible to convert reputation into money: if you are a famous scientist, you may get a well-paying job at a good university,. . .

transaction. By publishing their ideas and results, scientists sell their goods for reputation. Of course, this can only work if ideas and facts are attributed to their original creators (who gain reputation by being cited). You will see that scientists can become quite fierce and downright nasty when confronted with behavior that does not respect other's intellectual property.

The Academic Integrity Committee (AIC)

- ▷ Joint Committee by students and faculty (Not at "student honours court")
- ▷ Mandate: to hear and decide on any major or contested allegations, in particular,
  - ▷ the AIC decides based on evidence in a timely manner
  - ▷ the AIC makes recommendations that are executed by academic affairs
  - ▷ the AIC tries to keep allegations against faculty anonymous for the student
- ▷ we/you can appeal any academic integrity allegations to the AIC

One special case of academic rules that affects students is the question of cheating, which we will cover next.

Cheating [adapted from CMU:15-211 (P. Lee, 2003)]

- ▷ There is no need to cheat in this course!! (hard work will do)
- ▷ cheating prevents you from learning (you are cutting your own flesh)
- ▷ if you are in trouble, come and talk to me (I am here to help you)
- ▷ We expect you to know what is useful collaboration and what is cheating
  - ▷ you will be required to hand in your own original code/text/math for all assignments
  - ▷ you may discuss your homework assignments with others, but if doing so impairs your ability to write truly original code/text/math, you will be cheating
  - ▷ copying from peers, books or the Internet is plagiarism unless properly attributed (even if you change most of the actual words)
  - ▷ more on this as the semester goes on . . .
- ▷ ⚠ There are data mining tools that monitor the originality of text/code. ⚠
- ▷ Procedure: If we catch you at cheating (correction: if we suspect cheating)
  - ▷ we will confront you with the allegation (you can explain yourself)
  - ▷ if you admit or are silent, we impose a grade sanction and notify registrar
  - ▷ repeat infractions to go the AIC for deliberation (much more serious)
- ▷ Note: both active (copying from others) and passive cheating (allowing others to copy) are penalized equally

We are fully aware that the border between cheating and useful and legitimate collaboration is difficult to find and will depend on the special case. Therefore it is very difficult to put this into firm rules. We expect you to develop a firm intuition about behavior with integrity over the course of stay at Jacobs.

1.2.3 Resources

Even though the lecture itself will be the main source of information in the course, there are various resources from which to study the material.

Textbooks, Handouts and Information, Forum

- ▷ No required textbook, but course notes, posted slides
- ▷ Information resources (e.g. Course notes) will be posted at http://kwarc.info/teaching/GenCS
- ▷ Everything will be posted on PantaRhei (Notes+assignments+course forum)
  - ▷ announcements, contact information, course schedule and calendar
  - ▷ discussion among your fellow students(careful, I will occasionally check for academic integrity!)
  - ▷ http://panta.kwarc.info (use your Jacobs login)
  - ▷ Set Up PantaRhei Access: to get notifications
    1) Log into http://panta.kwarc.info, (use your Jacobs login)
    2) find the course "GenCS Fall 2016", (this course)
    3) request membership (I will approve you)
  - ▷ if there are problems send e-mail to course-gencs-tas@jacobs-university.de

No Textbook: Due to the special circumstances discussed above, there is no single textbook that covers the course. Instead we have a comprehensive set of course notes (this document). They are provided in two forms: as a large PDF that is posted at the course web page and on the PantaRhei system. The latter is actually the preferred method of interaction with the course materials, since it allows to discuss the material in place, to play with notations, to give feedback, etc. The PDF file is for printing and as a fallback, if the PantaRhei system, which is still under development, develops problems.

But of course, there is a wealth of literature on the subject, and the references at the end of the lecture notes can serve as a starting point for further reading. We will try to point out the relevant literature throughout the notes.

Software/Hardware tools

- ▷ You will need computer access for this course (come see me if you do not have a computer of your own)
- ▷ we recommend the use of standard software tools

- ▷ the `emacs` and `vi` text editor (powerful, flexible, available, free)
- ▷ UNIX (`linux`, `Mac OS X`, `cygwin`) (prevalent in CS)
- ▷ `FireFox` (just a better browser (for Math))

▷ learn how to touch-type NOW (reap the benefits earlier, not later)

Touch-typing: You should not underestimate the amount of time you will spend typing during your studies. Even if you consider yourself fluent in two-finger typing, touch-typing will give you a factor two in speed. This ability will save you at least half an hour per day, once you master it. Which can make a crucial difference in your success.

Touch-typing is very easy to learn, if you practice about an hour a day for a week, you will re-gain your two-finger speed and from then on start saving time. There are various free typing tutors on the network. At `http://typingsoft.com/all_typing_tutors.htm` you can find about programs, most for windows, some for linux. I would probably try `Ktouch` or `TuxType`

Darko Pesikan (one of the previous TAs) recommends the `TypingMaster` program. You can download a demo version from `http://www.typingmaster.com/index.asp?go=tutordemo`

You can find more information by googling something like "learn to touch-type". (goto `http://www.google.com` and type these search terms).

Chapter 2

Motivation and Introduction

Before we start with the course, we will have a look at what Computer Science is all about. This will guide our intuition in the rest of the course.

Consider the following situation, Jacobs University has decided to build a maze made of high hedges on the the campus green for the students to enjoy. Of course not any maze will do, we want a maze, where every room is reachable (unreachable rooms would waste space) and we want a unique solution to the maze to the maze (this makes it harder to crack).

Acknowledgement: The material in this Chapter is adapted from the introduction to a course held by Prof. Peter Lee at Carnegie Mellon university in 2002.

2.1 What is Computer Science?

What is Computer Science about?

- For instance: Software! (a hardware example would also work)
- **Example 2.1.1** writing a program to generate mazes.
- We want every maze to be solvable. (should have path from entrance to exit)
- Also: We want mazes to be fun, i.e.,
  - We want maze solutions to be unique
  - We want every "room" to be reachable
- How should we think about this?

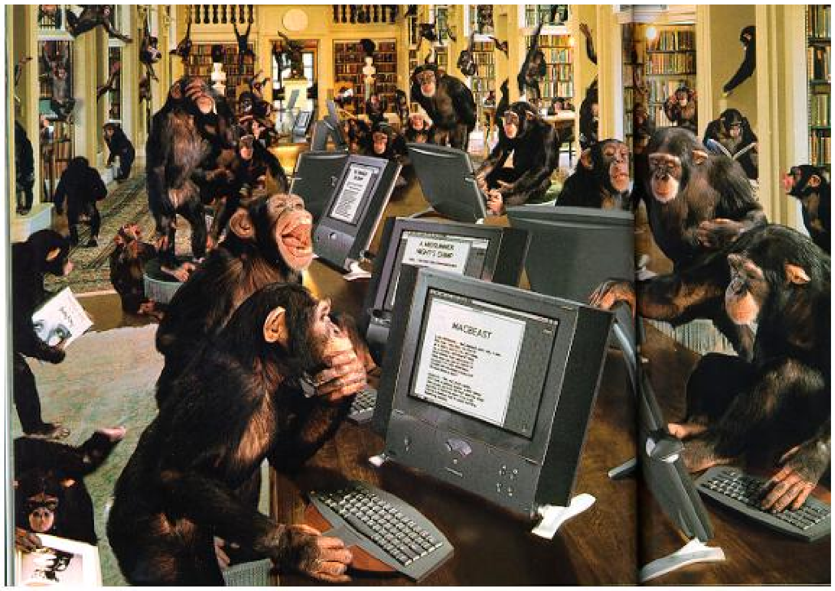
There are of course various ways to build such a a maze; one would be to ask the students from biology to come and plant some hedges, and have them re-plant them until the maze meets our criteria. A better way would be to make a plan first, i.e. to get a large piece of paper, and draw a maze before we plant. A third way is obvious to most students:

An Answer:

Let's hack

However, the result would probably be the following:

2am in the IRC Quiet Study Area



If we just start hacking before we fully understand the problem, chances are very good that we will waste time going down blind alleys, and garden paths, instead of attacking problems. So the main motto of this course is:

no, let's think

▷ "*The GIGO Principle: Garbage In, Garbage Out*" (– ca. 1967)

▷ "*Applets, Not Craplets*^{tm}" (– ca. 1997)

2.2 Computer Science by Example

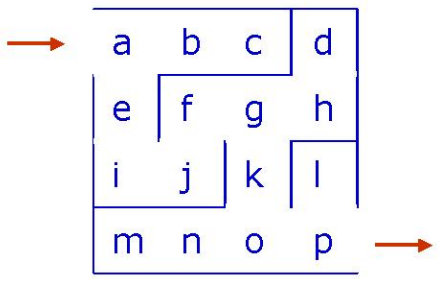
Thinking about a problem will involve thinking about the representations we want to use (after all, we want to work on the computer), which computations these representations support, and what constitutes a solutions to the problem.

This will also give us a foundation to talk about the problem with our peers and clients. Enabling students to talk about CS problems like a computer scientist is another important learning goal of this course.

We will now exemplify the process of "thinking about the problem" on our mazes example. It shows that there is quite a lot of work involved, before we write our first line of code. Of course, sometimes, explorative programming sometimes also helps understand the problem , but we would consider this as part of the thinking process.

Thinking about the problem

- ▷ Idea: Randomly knock out walls until we get a good maze
- ▷ Think about a grid of rooms separated by walls.
- ▷ Each room can be given a name.



- ▷ Mathematical Formulation:
  - ▹ a set of rooms: $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$
  - ▹ Pairs of adjacent rooms that have an open wall between them.
- ▷ **Example 2.2.1** For example, (a, b) and (g, k) are pairs.
- ▷ Abstractly speaking, this is a mathematical structure called a graph.

Of course, the "thinking" process always starts with an idea of how to attack the problem. In our case, this is the idea of starting with a grid-like structure and knocking out walls, until we have a maze which meets our requirements.

Note that we have already used our first representation of the problem in the drawing above: we have drawn a picture of a maze, which is of course not the maze itself.

Definition 2.2.2 A representation is the realization of real or abstract persons, objects, circumstances, Events, or emotions in concrete symbols or models. This can be by diverse methods, e.g. visual, aural, or written; as three-dimensional model, or even by dance.

Representations will play a large role in the course, we should always be aware, whether we are talking about "the real thing" or a representation of it (chances are that we are doing the latter in computer science). Even though it is important, to be able to always able to distinguish representations from the objects they represent, we will often be sloppy in our language, and rely on the ability of the reader to distinguish the levels.

From the pictorial representation of a maze, the next step is to come up with a mathematical representation; here as sets of rooms (actually room names as representations of rooms in the maze) and room pairs.

Why math?

- ▷ Q: Why is it useful to formulate the problem so that mazes are room sets/pairs?
- ▷ A: Data structures are typically defined as mathematical structures.
- ▷ A: Mathematics can be used to reason about the correctness and efficiency of data structures and algorithms.
- ▷ A: Mathematical structures make it easier to think — to abstract away from unnecessary details and avoid "hacking".

The advantage of a mathematical representation is that it models the aspects of reality we are interested in in isolation. Mathematical models/representations are very abstract, i.e. they have very few properties: in the first representational step we took we abstracted from the fact that we want to build a maze made of hedges on the campus green. We disregard properties like maze size, which kind of bushes to take, and the fact that we need to water the hedges after we planted them. In the abstraction step from the drawing to the set/pairs representation, we abstracted from further (accidental) properties, e.g. that we have represented a square maze, or that the walls are blue.

As mathematical models have very few properties (this is deliberate, so that we can understand all of them), we can use them as models for many concrete, real-world situations.

Intuitively, there are few objects that have few properties, so we can study them in detail. In our case, the structures we are talking about are well-known mathematical objects, called graphs.

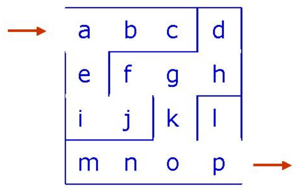
We will study graphs in more detail in this course, and cover them at an informal, intuitive level here to make our points.

Mazes as Graphs

▷ **Definition 2.2.3** Informally, a graph consists of a set of nodes and a set of edges. (a good part of CS is about graph algorithms)

▷ **Definition 2.2.4** A maze is a graph with two special nodes.

▷ Interpretation: Each graph node represents a room, and an edge from node x to node y indicates that rooms x and y are adjacent and there is no wall in between them. The first special node is the entry, and the second one the exit of the maze.



Can be represented as

$$\left\langle \left\{ \begin{array}{l} (a,e),(e,i),(i,j), \\ (f,j),(f,g),(g,h), \\ (d,h),(g,k),(a,b) \\ (m,n),(n,o),(b,c) \\ (k,o),(o,p),(l,p) \end{array} \right\}, a, p \right\rangle$$

So now, we have identified the mathematical object, we will use to think about our algorithm, and indeed it is very abstract, which makes it relatively difficult for humans to work with. To get around this difficulty, we will often draw pictures of graphs, and use them instead. But we will always keep in mind that these are not the graphs themselves but pictures of them — arbitrarily adding properties like color and layout that are irrelevant to the actual problem at hand but help the human cognitive apparatus in dealing with the problems. If we do keep this in mind, we can have both, the mathematical rigor and the intuitive ease of argumentation.

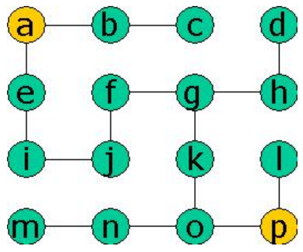
Mazes as Graphs (Visualizing Graphs via Diagrams)

▷ Graphs are very abstract objects, we need a good, intuitive way of thinking about them. We use diagrams, where the nodes are visualized as dots and the edges as lines between them.

Our maze

▷ $$\left\langle \left\{ \begin{array}{l} (a,e),(e,i),(i,j), \\ (f,j),(f,g),(g,h), \\ (d,h),(g,k),(a,b) \\ (m,n),(n,o),(b,c) \\ (k,o),(o,p),(l,p) \end{array} \right\}, a, p \right\rangle$$

can be visualized as



▷ Note that the diagram is a visualization (a representation intended for humans to process visually) of the graph, and not the graph itself.

Now that we have a mathematical model for mazes, we can look at the subclass of graphs that correspond to the mazes that we are after: unique solutions and all rooms are reachable! We will concentrate on the first requirement now and leave the second one for later.

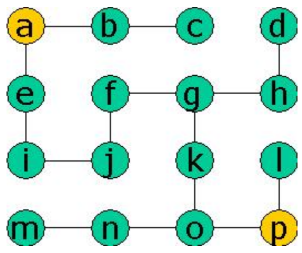
Unique solutions

▷ Q: What property must the graph have for the maze to have a solution?

▷ A: A path from a to p.

▷ Q: What property must it have for the maze to have a unique solution?

▷ A: The graph must be a tree.



Trees are special graphs, which we will now define.

Mazes as trees

▷ **Definition 2.2.5** Informally, a tree is a graph:

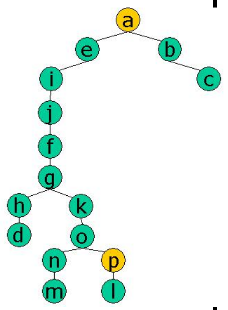
- ▷ with a unique root node, and
- ▷ each node having a unique parent.

▷ **Definition 2.2.6** A spanning tree is a tree that includes all of the nodes.

Q: Why is it good to have a spanning tree?

⊳▷ A: Trees have no cycles! (needed for uniqueness)

▷ A: Every room is reachable from the root!



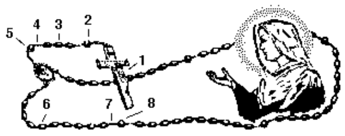
So, we know what we are looking for, we can think about a program that would find spanning trees given a set of nodes in a graph. But since we are still in the process of "thinking about the problems" we do not want to commit to a concrete program, but think about programs in the abstract (this gives us license to abstract away from many concrete details of the program and concentrate on the essentials).

The computer science notion for a program in the abstract is that of an algorithm, which we will now define.

Algorithm

▷ Now that we have a data structure in mind, we can think about the algorithm.

▷ **Definition 2.2.7** An algorithm is a series of instructions to control a (computation) process



▷ **Example 2.2.8 (Kruskal's algorithm, a graph algorithm for spanning trees)** ▷ Randomly add a pair to the tree if it won't create a cycle.(i.e. tear down a wall)

▷ Repeat until a spanning tree has been created.

Definition 2.2.9 An algorithm is a collection of formalized rules that can be understood and executed, and that lead to a particular endpoint or result.

Example 2.2.10 An example for an algorithm is a recipe for a cake, another one is a rosary — a kind of chain of beads used by many cultures to remember the sequence of prayers. Both the recipe and rosary represent instructions that specify what has to be done step by step. The instructions in a recipe are usually given in natural language text and are based on elementary forms of manipulations like "scramble an egg" or "heat the oven to 250 degrees Celsius". In a rosary, the instructions are represented by beads of different forms, which represent different prayers. The physical (circular) form of the chain allows to represent a possibly infinite sequence of prayers.

The name algorithm is derived from the word al-Khwarizmi, the last name of a famous Persian mathematician. Abu Ja'far Mohammed ibn Musa al-Khwarizmi was born around 780 and died around 845. One of his most influential books is "Kitab al-jabr w'al-muqabala" or "Rules of Restoration and Reduction". It introduced algebra, with the very word being derived from a part of the original title, namely "al-jabr". His works were translated into Latin in the 12th century, introducing this new science also in the West.

The algorithm in our example sounds rather simple and easy to understand, but the high-level formulation hides the problems, so let us look at the instructions in more detail. The crucial one is the task to check, whether we would be creating cycles.

Of course, we could just add the edge and then check whether the graph is still a tree, but this would be very expensive, since the tree could be very large. A better way is to maintain some information during the execution of the algorithm that we can exploit to predict cyclicity before altering the graph.

Creating a spanning tree

▷ When adding a wall to the tree, how do we detect that it won't create a cycle?

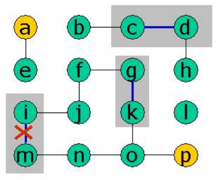
▷ When adding wall (x, y), we want to know if there is already a path from x to y in the tree.

▷ In fact, there is a fast algorithm for doing exactly this, called "Union-Find".

Definition 2.2.11 (Union Find Algorithm) The Union Find Algorithm successively puts nodes into an equivalence class if there is a path connecting them.

▷ Before adding an edge (x, y) to the tree, it makes sure that x and y are not in the same equivalence class.

Example 2.2.12 A partially constructed maze



Now that we have made some design decision for solving our maze problem. It is an important part of "thinking about the problem" to determine whether these are good choices. We have argued above, that we should use the Union-Find algorithm rather than a simple "generate-and-test" approach based on the "expense", by which we interpret temporally for the moment. So we ask ourselves

How fast is our Algorithm?

▷ Is this a fast way to generate mazes?
  ▷ How much time will it take to generate a maze?
  ▷ What do we mean by "fast" anyway?

▷ In addition to finding the right algorithms, Computer Science is about analyzing the performance of algorithms.

In order to get a feeling what we mean by "fast algorithm", we to some preliminary computations.

Performance and Scaling

▷ Suppose we have three algorithms to choose from. (which one to select)

▷ Systematic analysis reveals performance characteristics.

▷ **Example 2.2.13** For a problem of size n (i.e., detecting cycles out of n nodes) we have

	performance		
size	linear	quadratic	exponential
n	$100n\mu$s	$7n^2\mu$s	$2^n\mu$s
1	100μs	7μs	2μs
5	.5ms	175μs	32μs
10	1ms	.7ms	1ms
45	4.5ms	14ms	$1.1Y$
100	...	...	...
1 000	...	...	...
10 000	...	...	...
1 000 000	...	...	...

What?! One year?

▷ $2^{10} = 1\,024$ ($\approxeq 1024\mu$s, 1ms)

▷ $2^{45} = 35\,184\,372\,088\,832$ ($3.5 \times 10^{13}\mu$s$=3.5 \times 10^7$s$\sim 1.1Y$)

▷ **Example 2.2.14** we denote all times that are longer than the age of the universe with —

	performance		
size	linear	quadratic	exponential
n	$100n\mu$s	$7n^2\mu$s	$2^n\mu$s
1	100μs	7μs	2μs
5	.5ms	175μs	32μs
10	1ms	.7ms	1ms
45	4.5ms	14ms	$1.1Y$
< 100	100ms	7s	$10^{16}Y$
1 000	1s	12min	—
10 000	10s	20h	—
1 000 000	1.6min	2.5mon	—

So it does make a difference for larger problems what algorithm we choose. Considerations like the one we have shown above are very important when judging an algorithm. These evaluations go by the name of complexity theory.

2.3 Other Topics in Computer Science

We will now briefly preview other concerns that are important to computer science. These are essential when developing larger software packages. We will not be able to cover them in this course, but leave them to the second year courses, in particular "software engineering".

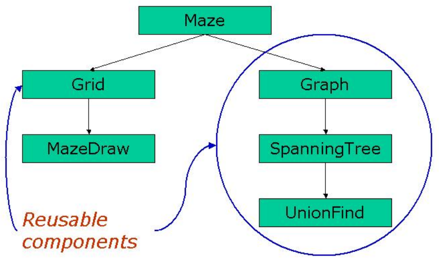
The first concern in software engineering is of course whether your program does what it is supposed to do.

Is it correct?

- ▷ How will we know if we implemented our solution correctly?
  - ▷ What do we mean by "correct"?
  - ▷ Will it generate the right answers?
  - ▷ Will it terminate?
- ▷ Computer Science is about techniques for proving the correctness of programs

Modular design

- ▷ By thinking about the problem, we have strong hints about the structure of our program
- ▷ Grids, Graphs (with edges and nodes), Spanning trees, Union-find.
- ▷ With disciplined programming, we can write our program to reflect this structure.
- ▷ Modular designs are usually easier to get right and easier to understand.



Indeed, modularity is a major concern in the design of software: if we can divide the functionality of the program in to small, self-contained "modules" that provide a well-specified functionality (possibly building on other modules), then we can divide work, and develop and test parts of the program separately, greatly reducing the overall complexity of the development effort.

In particular, if we can identify modules that can be used in multiple situations, these can be published as libraries, and re-used in multiple programs, again reducing complexity.

Modern programming languages support modular design by a variety of measures and structures. The functional programming language SML presented in this course, has a very elaborate module system, which we will not be able to cover in this course. Hover, SML data types allow to define what object-oriented languages use "classes" for: sets of similarly-structured objects that support the same sort of behavior (which can be the pivotal points of modules).

2.4 Summary

The science in CS: not "hacking", but

- ▷ Thinking about problems abstractly.
- ▷ Selecting good structures and obtaining correct and fast algorithms/machines.
- ▷ Implementing programs/machines that are understandable and correct.

In particular, the course “General Computer Science” is not a programming course, it is about being able to think about computational problems and to learn to talk to others about these problems.

Part I

Representation and Computation

Chapter 3

Elementary Discrete Math

We have seen in the last section that we will use mathematical models for objects and data structures throughout Computer Science. As a consequence, we will need to learn some math before we can proceed. But we will study mathematics for another reason: it gives us the opportunity to study rigorous reasoning about abstract objects, which is needed to understand the "science" part of Computer Science.

Note that the mathematics we will be studying in this course is probably different from the mathematics you already know; calculus and linear algebra are relatively useless for modeling computations. We will learn a branch of math. called "discrete mathematics", it forms the foundation of computer science, and we will introduce it with an eye towards computation.

Let's start with the math!

Discrete Math for the moment

▷ Kenneth H. Rosen *Discrete Mathematics and Its Applications*, McGraw-Hill, 1990 [Ros90].

▷ Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*, Prentice Hall, 1998 [LP98].

▷ Paul R. Halmos, *Naive Set Theory*, Springer Verlag, 1974 [Hal74].

The roots of computer science are old, much older than one might expect. The very concept of computation is deeply linked with what makes mankind special. We are the only animal that manipulates abstract concepts and has come up with universal ways to form complex theories and to apply them to our environments. As humans are social animals, we do not only form these theories in our own minds, but we also found ways to communicate them to our fellow humans.

3.1 Mathematical Foundations: Natural Numbers

The most fundamental abstract theory that mankind shares is the use of numbers. This theory of numbers is detached from the real world in the sense that we can apply the use of numbers to arbitrary objects, even unknown ones. Suppose you are stranded on an lonely island where you see a strange kind of fruit for the first time. Nevertheless, you can immediately count these fruits. Also, nothing prevents you from doing arithmetics with some fantasy objects in your mind. The question in the following sections will be: what are the principles that allow us to form and apply numbers in these general ways? To answer this question, we will try to find general ways to specify and manipulate arbitrary objects. Roughly speaking, this is what computation is all about.

Something very basic:

▷ Numbers are symbolic representations of numeric quantities.

▷ There are many ways to represent numbers (more on this later)

▷ let's take the simplest one (about 8,000 to 10,000 years old)

▷ we count by making marks on some surface.

▷ For instance / / / / stands for the number four (be it in 4 apples, or 4 worms)

▷ Let us look at the way we construct numbers a little more algorithmically,

▷ these representations are those that can be created by the following two rules.

o**-rule** consider ' ' as an empty space.

s**-rule** given a row of marks or an empty space, make another / mark at the right end of the row.

▷ **Example 3.1.1** For / / / /, Apply the o-rule once and then the s-rule four times.

▷ **Definition 3.1.2** we call these representations unary natural numbers.

In addition to manipulating normal objects directly linked to their daily survival, humans also invented the manipulation of place-holders or symbols. A *symbol* represents an object or a set of objects in an abstract way. The earliest examples for symbols are the cave paintings showing iconic silhouettes of animals like the famous ones of Cro-Magnon. The invention of symbols is not only an artistic, pleasurable "waste of time" for mankind, but it had tremendous consequences. There is archaeological evidence that in ancient times, namely at least some 8000 to 10000 years ago, men started to use tally bones for counting. This means that the symbol "bone" was used to represent numbers. The important aspect is that this bone is a symbol that is completely detached from its original down to earth meaning, most likely of being a tool or a waste product from a meal. Instead it stands for a universal concept that can be applied to arbitrary objects.

Instead of using bones, the slash / is a more convenient symbol, but it is manipulated in the same way as in the most ancient times of mankind. The *o*-rule allows us to start with a blank slate or an empty container like a bowl. The *s*- or successor-rule allows to put an additional bone into a bowl with bones, respectively, to append a slash to a sequence of slashes. For instance unary for the number four — be it 4 apples, or 4 worms. This representation is constructed by applying the *o*-rule once and then the *s*-rule four times.

So, we have a basic understanding of natural numbers now, but we will also need to be able to talk about them in a mathematically precise way. Generally, this additional precision will involve defining specialized vocabulary for the concepts and objects we want to talk about, making the assumptions we have about the objects exmplicit, and the use of special modes or argumentation.

We will introduce all of these for the special case of unary natural numbers here, but we will use the concepts and practices throughout the course and assume students will do so as well.

With the notion of a successor from Definition 3.1.3 we can formulate a set of assumptions (called axioms) about unary natural numbers. We will want to use these assumptions (statements we believe to be true) to derive other statements, which — as they have been obtained by generally accepted argumentation patterns — we will also believe true. This intuition is put more formally in ?peano-axiom.def? below, which also supplies us with names for different types of statements.

A little more sophistication (math) please

▷ **Definition 3.1.3** We call a unary natural number the successor (predecessor) of another, if it can be constructing by adding (removing) a slash. (successors are created by the *s*-rule)

▷ **Example 3.1.4** /// is the successor of // and // the predecessor of ///.

▷ **Definition 3.1.5** The following set of axioms are called the Peano axioms (Giuseppe Peano $*1858$, $\dagger 1932$)

▷ **Axiom 3.1.6 (P1)** " " (aka. "zero") is a unary natural number. " " (aka. "zero") is a unary natural number.

▷ **Axiom 3.1.7 (P2)** Every unary natural number has a successor that is a unary natural number and that is different from it.

▷ **Axiom 3.1.8 (P3)** Zero is not a successor of any unary natural number.

▷ **Axiom 3.1.9 (P4)** Different unary natural numbers have different successors.

▷ **Axiom 3.1.10 (P5: Induction Axiom)** Every unary natural number possesses a property P, if

- ▷ zero has property P and (base condition)
- ▷ the successor of every unary natural number that has property P also possesses property P (step condition)

Question: Why is this a better way of saying things (why so complicated?)

Note that the Peano axioms may not be the first things that come to mind when thinking about characteristic properties of natural numbers. Indeed they have been selected to to be minimal, so that we can get by with as few assumptions as possible; all other statements of properties can be derived from them, so minimality is not a bug, but a feature: the Peano axioms form the foundation, on which all knowledge about unary natural numbers rests.

We now come to the ways we can derive new knowledge from the Peano axioms.

3.2 Reasoning about Natural Numbers

▷ Reasoning about Natural Numbers

- ▷ The Peano axioms can be used to reason about natural numbers.
- ▷ **Definition 3.2.1** An axiom (or postulate) is a statement about mathematical objects that we assume to be true.
- ▷ **Definition 3.2.2** A theorem is a statement about mathematical objects that we know to be true.
- ▷ We reason about mathematical objects by inferring theorems from axioms or other theorems, e.g.

▷ “ ” is a unary natural number	(axiom P1)
▷ / is a unary natural number	(axiom P2 and 1.)
▷ // is a unary natural number	(axiom P2 and 2.)
▷ /// is a unary natural number	(axiom P2 and 3.)

- ▷ **Definition 3.2.3** We call a sequence of inferences a derivation or a proof (of the last statement).

If we want to be more precise about these (important) notions, we can define them as follows:

Definition 3.2.4 In general, a axiom is a starting point in logical reasoning with the aim to prove a mathematical statement (called a conjecture as long as it is unproven and unrefuted). A conjecture that is proven is called a theorem.

Conventionally, there are there are two subtypes of theorems. A lemma is an intermediate theorem that serves as part of a proof of a larger theorem. A corollary is a theorem that follows directly from another theorem.

A logical system consists of axioms and rules that allow inference, i.e. that allow to form new formal statements out of already proven ones. So, a proof of a conjecture starts from the axioms that are transformed via the rules of inference until the conjecture is derived.

We will now practice this reasoning on a couple of examples. Note that we also use them to introduce the inference system of mathematics via these example proofs.

Here are some theorems you may want to prove for practice. The proofs are relatively simple.

Let's practice derivations and proofs

- ▷ **Example 3.2.5** //////////// is a unary natural number
- ▷ **Theorem 3.2.6** *///* *is a different unary natural number than* *//.*
- ▷ **Theorem 3.2.7** */////* *is a different unary natural number than* *//.*

▷ **Theorem 3.2.8** *There is a unary natural number of which /// is the successor*

▷ **Theorem 3.2.9** *There are at least 7 unary natural numbers.*

▷ **Theorem 3.2.10** *Every unary natural number is either zero or the successor of a unary natural number.* (*we will come back to this later*)

Induction for unary natural numbers

▷ **Theorem 3.2.11** *Every unary natural number is either zero or the successor of a unary natural number.*

▷ Proof: We make use of the induction axiom P5:

P.1 We use the property P of "being zero or a successor" and prove the statement by convincing ourselves of the prerequisites of

P.2 ' ' is zero, so ' ' is "zero or a successor".

P.3 Let n be a arbitrary unary natural number that "is zero or a successor"

P.4 Then its successor "is a successor", so the successor of n is "zero or a successor"

P.5 Since we have taken n arbitrary (nothing in our argument depends on the choice)
we have shown that for any n, its successor has property P.

P.6 Property P holds for all unary natural numbers by P5, so we have proven the assertion □

We have already seen in the proof above, that it helps to give names to objects: for instance, by using the name n for the number about which we assumed the property P, we could just say that $P(n)$ holds. But there is more we can do.

We can give systematic names to the unary natural numbers. Note that it is often to reference objects in a way that makes their construction overt. the unary natural numbers we can represent as expressions that trace the applications of the o-rule and the s-rules.

This seems awfully clumsy, lets introduce some notation

▷ Idea: we allow ourselves to give names to unary natural numbers (we use n, m, l, k, n_1, n_2, ... as names for concrete unary natural numbers.)

▷ Remember the two rules we had for dealing with unary natural numbers

▷ Idea: represent a number by the trace of the rules we applied to construct it. (e.g. //// is represented as $s(s(s(s(o))))$)

▷ **Definition 3.2.12** We introduce some abbreviations

▷ we "abbreviate" o and ' ' by the symbol '0' (called "zero")

- ▷ we abbreviate $s(o)$ and / by the symbol '1' (called "one")
- ▷ we abbreviate $s(s(o))$ and // by the symbol '2' (called "two")
- ▷ ...
- ▷ we abbreviate $s(s(s(s(s(s(s(s(s(s(s(s(o))))))))))))$ and //////////// by the symbol '12' (called "twelve")
- ▷ ...

▷ **Definition 3.2.13** We denote the set of all unary natural numbers with $\mathbb{N}_1$. (either representation)

This systematic representation of natural numbers by expressions becomes very powerful, if we mix it with the practice of giving names to generic objects. These names are often called variables, when used in expressions.

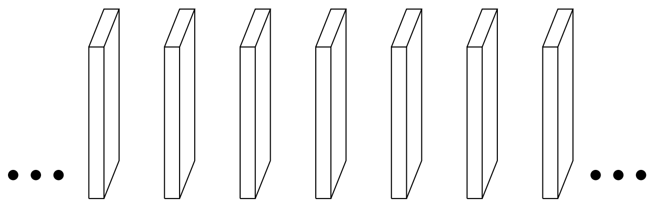
Theorem 3.2.11 is a very useful fact to know, it tells us something about the form of unary natural numbers, which lets us streamline induction proofs and bring them more into the form you may know from school: to show that some property P holds for every natural number, we analyze an arbitrary number n by its form in two cases, either it is zero (the base case), or it is a successor of another number (the step case). In the first case we prove the base condition and in the latter, we prove the step condition and use the induction axiom to conclude that all natural numbers have property P. We will show the form of this proof in the domino-induction below.

The Domino Theorem

▷ **Theorem 3.2.14** *Let $S_0, S_1, \ldots$ be a linear sequence of dominos, such that for any unary natural number i we know that*

- ▷ *the distance between S_i and $S_{s(i)}$ is smaller than the height of S_i,*
- ▷ *S_i is much higher than wide, so it is unstable, and*
- ▷ *S_i and $S_{s(i)}$ have the same weight.*

If S_0 is pushed towards S_1 so that it falls, then all dominos will fall.



The Domino Induction

▷ Proof: We prove the assertion by induction over i with the property P that "S_i falls in the direction of $S_{s(i)}$".

P.1 We have to consider two cases

P.1.1 base case: i is zero:

P.1.1.1 We have assumed that "S_0 is pushed towards S_1, so that it falls" $\square$

P.1.2 step case: $i = s(j)$ for some unary natural number j:

P.1.2.1 We assume that P holds for S_j, i.e. S_j falls in the direction of $S_{s(j)} = S_i$.

P.1.2.2 But we know that S_j has the same weight as S_i, which is unstable,

P.1.2.3 so S_i falls into the direction opposite to S_j, i.e. towards $S_{s(i)}$ (we have a linear sequence of dominos) $\square$

P.2 We have considered all the cases, so we have proven that P holds for all unary natural numbers i. (by induction)

P.3 Now, the assertion follows trivially, since if "S_i falls in the direction of $S_{s(i)}$", then in particular "S_i falls". $\square$

If we look closely at the proof above, we see another recurring pattern. To get the proof to go through, we had to use a property P that is a little stronger than what we need for the assertion alone. In effect, the additional clause "... in the direction ..." in property P is used to make the step condition go through: we we can use the stronger inductive hypothesis in the proof of step case, which is simpler.

Often the key idea in an induction proof is to find a suitable strengthening of the assertion to get the step case to go through.

3.3 Defining Operations on Natural Numbers

The next thing we want to do is to define operations on unary natural numbers, i.e. ways to do something with numbers. Without really committing what "operations" are, we build on the intuition that they take (unary natural) numbers as input and return numbers. The important thing in this is not what operations are but how we define them.

What can we do with unary natural numbers?

- ▷ So far not much (let's introduce some operations)
- ▷ **Definition 3.3.1 (the addition "function")** We "define" the addition operation $\oplus$ procedurally (by an algorithm)
  - ▷ adding zero to a number does not change it.
    written as an equation: $n \oplus o = n$
  - ▷ adding m to the successor of n yields the successor of $m \oplus n$.
    written as an equation: $m \oplus s(n) = s(m \oplus n)$

Questions: to understand this definition, we have to know

- ▷ ▷ Is this "definition" well-formed? (does it characterize a mathematical object?)
  - ▷ May we define "functions" by algorithms? (what is a function anyways?)

So we have defined the addition operation on unary natural numbers by way of two equations. Incidentally these equations can be used for computing sums of numbers by replacing equals by equals; another of the generally accepted manipulation

Definition 3.3.2 (Replacement) If we have a representation s of an object and we have an equation $l = r$, then we can obtain an object by replacing an occurrence of the sub-expression l in s by r and have $s = s'$.

In other words if we replace a sub-expression of s with an equal one, nothing changes. This is exactly what we will use the two defining equations from Definition 3.3.1 for in the following example

Example 3.3.3 (Computing the Sum Two and One) If we start with the expression $s(s(o)) \oplus s(o)$, then we can use the second equation to obtain $s(s(s(o)) \oplus o)$ (replacing equals by equals), and – this time with the first equation $s(s(s(o)))$.

Observe: in the computation in Example 3.3.3 at every step there was exactly one of the two equations we could apply. This is a consequence of the fact that in the second argument of the two equations are of the form o and $s(n)$: by Theorem 3.2.11 these two cases cover all possible natural numbers and by **P**3 (see Axiom 3.1.8), the equations are mutually exclusive. As a consequence we do not really have a choice in the computation, so the two equations do form an "algorithm" (to the extend we already understand them), and the operation is indeed well-defined. The form of the arguments in the two equations in Definition 3.3.1 is the same as in the induction axiom, therefore we will consider the first equation as the base equation and second one as the step equation.

We can understand the process of computation as a "getting-rid" of operations in the expression. Note that even though the step equation does not really reduce the number of occurrences of the operator (the base equation does), but it reduces the number of constructor in the second argument, essentially preparing the elimination of the operator via the base equation. Note that in any case when we have eliminated the operator, we are left with an expression that is completely made up of constructors; a representation of a unary natural number.

Now we want to see whether we can find out some properties of the addition operation. The method for this is of course stating a conjecture and then proving it.

Addition on unary natural numbers is associative

▷ **Theorem 3.3.4** *For all unary natural numbers n, m, and l, we have $n \oplus (m \oplus l) = (n \oplus m) \oplus l$.*

▷ Proof: we prove this by induction on l

P.1 The property of l is that $n \oplus (m \oplus l) = (n \oplus m) \oplus l$ holds.

P.2 We have to consider two cases

P.2.1 base case: $n \oplus (m \oplus o) = n \oplus m = (n \oplus m) \oplus o$

P.2.2 step case:

P.2.2.1 given arbitrary l, assume $n \oplus (m \oplus l) = (n \oplus m) \oplus l$, show $n \oplus (m \oplus s(l)) = (n \oplus m) \oplus s(l)$.

P.2.2.2 We have $n \oplus (m \oplus s(l)) = n \oplus s(m \oplus l) = s(n \oplus (m \oplus l))$

P.2.2.3 By inductive hypothesis $s((n \oplus m) \oplus l) = (n \oplus m) \oplus s(l)$ □

□

We observe that In the proof above, the induction corresponds to the defining equations of $\oplus$; in particular base equation of $\oplus$ was used in the base case of the induction whereas the step equation of $\oplus$ was used in the step case. Indeed computation (with operations over the unary natural numbers) and induction (over unary natural numbers) are just two sides of the same coin as we will see.

Let us consider a couple more operations on the unary natural numbers to fortify our intutions.

More Operations on Unary Natural Numbers

- ▷ **Definition 3.3.5** The unary multiplication operation can be defined by the equations $n \odot o = o$ and $n \odot s(m) = n \oplus n \odot m$.
- ▷ **Definition 3.3.6** The unary exponentiation operation can be defined by the equations $\exp(n, o) = s(o)$ and $\exp(n, s(m)) = n \odot \exp(n, m)$.
- ▷ **Definition 3.3.7** The unary summation operation can be defined by the equations $\bigoplus_{i=o}^{o} n_i = o$ and $\bigoplus_{i=o}^{s(m)} n_i = n_{s(m)} \oplus \bigoplus_{i=o}^{m} n_i$.
- ▷ **Definition 3.3.8** The unary product operation can be defined by the equations $\bigodot_{i=o}^{o} n_i = s(o)$ and $\bigodot_{i=o}^{s(m)} n_i = n_{s(m)} \odot \bigodot_{i=o}^{m} n_i$.

In Definition 3.3.5, we have used the operation $\oplus$ in the right-hand side of the step-equation. This is perfectly reasonable and only means that we have eliminate more than one operator.

Note that we did not use disambiguating parentheses on the right hand side of the step equation for $\odot$. Here $n \oplus n \odot m$ is a unary sum whose second summand is a product. Just as we did there, we will use the usual arithmetic precedences to reduce the notational overload.

The remaining examples are similar in spirit, but a bit more involved, since they nest more operators. Just like we showed associativity for $\oplus$ in slide 39, we could show properties for these operations, e.g.

$$\bigodot_{i=o}^{n \oplus m} k_i = \bigodot_{i=o}^{n} k_i \odot \bigodot_{i=o}^{m} k_{(i \oplus n)} \tag{3.1}$$

by induction, with exactly the same observations about the parallelism between computation and induction proofs as $\oplus$.

Definition 3.3.8 gives us the chance to elaborate on the process of definitions some more: When we define new operations such as the product over a sequence of unary natural numbers, we do have freedom of what to do with the corner cases, and for the "empty product" (the base case equation) we could have chosen

1) to leave the product undefined (not nice; we need a base case), or

2) to give it another value, e.g. $s(s(o))$ or o.

But any value but $s(o)$ would violate the generalized distributivity law in equation 3.1 which is exactly what we would expect to see (and which is very useful in calculations). So if we want to have this equation (and I claim that we do) then we have to choose the value $s(o)$.

In summary, even though we have the freedom to define what we want, if we want to define sensible and useful operators our freedom is limited.

3.4 Talking (and writing) about Mathematics

Before we go on, we need to learn how to talk and write about mathematics in a succinct way. This will ease our task of understanding a lot.

Talking about Mathematics (MathTalk)

▷ **Definition 3.4.1** Mathematicians use a stylized language that

- ▷ uses formulae to represent mathematical objects, e.g. $\int_0^1 x^{3/2}dx$
- ▷ uses math idioms for special situations (e.g. *iff*, *hence*, *let...be...*, *then...*)
- ▷ classifies statements by role (e.g. Definition, Lemma, Theorem, Proof, Example)

We call this language mathematical vernacular.

▷ **Definition 3.4.2** Abbreviations for Mathematical statements in MathTalk

- ▷ $\wedge$ and "$\vee$" are common notations for "and" and "or"
- ▷ "not" is in mathematical statements often denoted with $\neg$
- ▷ $\forall x.P$ ($\forall x \in S.P$) stands for "condition P holds for all x (in S)"
- ▷ $\exists x.P$ ($\exists x \in S.P$) stands for "there exists an x (in S) such that proposition P holds"
- ▷ $\nexists x.P$ ($\nexists x \in S.P$) stands for "there exists no x (in S) such that proposition P holds"
- ▷ $\exists^1 x.P$ ($\exists^1 x \in S.P$) stands for "there exists one and only one x (in S) such that proposition P holds"
- ▷ "iff" as abbreviation for "if and only if", symbolized by "$\Leftrightarrow$"
- ▷ the symbol "$\Rightarrow$" is used a as shortcut for "implies"

Observation: With these abbreviations we can use formulae for statements.

⧐ **Example 3.4.3** $\forall x.\exists y.x = y \Leftrightarrow \neg(x \neq y)$ reads

> "For all x, there is a y, such that $x = y$, iff (if and only if) it is not the case that $x \neq y$."

To fortify our intuitions, we look at a more substantial example, which also extends the usage of the expression language for unary natural numbers.

Peano Axioms in Mathtalk

▷ **Example 3.4.4** We can write the Peano Axioms in mathtalk: If we write $n \in \mathbb{N}_1$ for *n is a unary natural number*, and $P(n)$ for *n has property P*, then we can write

- ▷ $o \in \mathbb{N}_1$ (zero is a unary natural number)
- ▷ $\forall n \in \mathbb{N}_1.s(n) \in \mathbb{N}_1 \wedge n \neq s(n)$ ($\mathbb{N}_1$closed under successors, distinct)

- ▷ $\neg(\exists n \in \mathbb{N}_1 . o = s(n))$ (zero is not a successor)
- ▷ $\forall n \in \mathbb{N}_1 . \forall m \in \mathbb{N}_1 . n \neq m \Rightarrow s(n) \neq s(m)$ (different successors)
- ▷ $\forall P . (P(o) \wedge (\forall n \in \mathbb{N}_1 . P(n) \Rightarrow P(s(n)))) \Rightarrow (\forall m \in \mathbb{N}_1 . P(m))$ (induction)

We will use mathematical vernacular throughout the remainder of the notes. The abbreviations will mostly be used in informal communication situations. Many mathematicians consider it bad style to use abbreviations in printed text, but approve of them as parts of formulae (see e.g. Definition 45 for an example).

Mathematics uses a very effective technique for dealing with conceptual complexity. It usually starts out with discussing simple, *basic* objects and their properties. These simple objects can be combined to more complex, *compound* ones. Then it uses a definition to give a compound object a new name, so that it can be used like a basic one. In particular, the newly defined object can be used to form compound objects, leading to more and more complex objects that can be described succinctly. In this way mathematics incrementally extends its vocabulary by add layers and layers of definitions onto very simple and basic beginnings. We will now discuss four definition schemata that will occur over and over in this course.

Definition 3.4.5 The simplest form of definition schema is the simple definition. This just introduces a name (the definiendum) for a compound object (the definiens). Note that the name must be new, i.e. may not have been used for anything else, in particular, the definiendum may not occur in the definiens. We use the symbols $:=$ (and the inverse $=:$) to denote simple definitions in formulae.

Example 3.4.6 We can give the unary natural number $////$ the name φ. In a formula we write this as $\varphi := (\,////)$ or $//// =: \varphi$.

Definition 3.4.7 A somewhat more refined form of definition is used for operators on and relations between objects. In this form, then definiendum is the operator or relation is applied to n distinct variables $v_1, \ldots, v_n$ as arguments, and the definiens is an expression in these variables. When the new operator is applied to arguments $a_1, \ldots, a_n$, then its value is the definiens expression where the v_i are replaced by the a_i. We use the symbol $:=$ for operator definitions and $:\Leftrightarrow$ for pattern definitions.[1] EdN:1

Example 3.4.8 The following is a pattern definition for the set intersection operator $\cap$:

$$A \cap B := \{x \,|\, x \in A \wedge x \in B\}$$

The pattern variables are A and B, and with this definition we have e.g. $\emptyset \cap \emptyset = \{x \,|\, x \in \emptyset \wedge x \in \emptyset\}$.

Definition 3.4.9 We now come to a very powerful definition schema. An implicit definition (also called definition by description) is a formula $\mathbf{A}$, such that we can prove $\exists^1 n . \mathbf{A}$, where n is a new name.

Example 3.4.10 $\forall x . x \in \emptyset$ is an implicit definition for the empty set $\emptyset$. Indeed we can prove unique existence of $\emptyset$ by just exhibiting $\{\}$ and showing that any other set S with $\forall x . x \notin S$ we have $S \equiv \emptyset$. Indeed$S$ cannot have elements, so it has the same elements ad $\emptyset$, and thus $S \equiv \emptyset$.

To keep mathematical formulae readable (they are bad enough as it is), we like to express mathematical objects in single letters. Moreover, we want to choose these letters to be easy to remember; e.g. by choosing them to remind us of the name of the object or reflect the kind of object (is it a number or a set, …). Thus the 50 (upper/lowercase) letters supplied by most alphabets are not sufficient for expressing mathematics conveniently. Thus mathematicians and computer scientists use at least two more alphabets.

[1] EdNote: maybe better markup up pattern definitions as binding expressions, where the formal variables are bound.

The Greek, Curly, and Fraktur Alphabets $\rightsquigarrow$ Homework

▷ Homework: learn to read, recognize, and write the Greek letters

α	A	alpha	β	B	beta	γ	Γ	gamma
δ	Δ	delta	ϵ	E	epsilon	ζ	Z	zeta
η	H	eta	θ, ϑ	Θ	theta	ι	I	iota
κ	K	kappa	λ	Λ	lambda	μ	M	mu
ν	N	nu	ξ	Ξ	Xi	o	O	omicron
π, ϖ	Π	Pi	ρ	P	rho	σ	Σ	sigma
τ	T	tau	υ	Υ	upsilon	φ	Φ	phi
χ	X	chi	ψ	Ψ	psi	ω	Ω	omega

▷ we will need them, when the other alphabets give out.

▷ BTW, we will also use the curly Roman and "Fraktur" alphabets:
$\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$, $\mathcal{F}$, $\mathcal{G}$, $\mathcal{H}$, $\mathcal{I}$, $\mathcal{J}$, $\mathcal{K}$, $\mathcal{L}$, $\mathcal{M}$, $\mathcal{N}$, $\mathcal{O}$, $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$, $\mathcal{S}$, $\mathcal{T}$, $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$
$\mathfrak{A}$, $\mathfrak{B}$, $\mathfrak{C}$, $\mathfrak{D}$, $\mathfrak{E}$, $\mathfrak{F}$, $\mathfrak{G}$, $\mathfrak{H}$, $\mathfrak{I}$, $\mathfrak{J}$, $\mathfrak{K}$, $\mathfrak{L}$, $\mathfrak{M}$, $\mathfrak{N}$, $\mathfrak{O}$, $\mathfrak{P}$, $\mathfrak{Q}$, $\mathfrak{R}$, $\mathfrak{S}$, $\mathfrak{T}$, $\mathfrak{U}$, $\mathfrak{V}$, $\mathfrak{W}$, $\mathfrak{X}$, $\mathfrak{Y}$, $\mathfrak{Z}$

To be able to *read and understand* math and computer science texts profitably it is only only important to recognize the Greek alphabet, but also to know about the correspondences with the Roman one. For instance, ν corresponds to the n, so we often use ν as names for objects we would otherwise use n for (but cannot).

To be able to *talk about* math and computerscience, we also have to be able to pronounce the Greek letters, otherwise we embarrass ourselves by saying something like "the funny Greek letter that looks a bit like a w".

3.5 Naive Set Theory

We now come to a very important and foundational aspect in Mathematics: Sets. Their importance comes from the fact that all (known) mathematics can be reduced to understanding sets. So it is important to understand them thoroughly before we move on.

But understanding sets is not so trivial as it may seem at first glance. So we will just represent sets by various descriptions. This is called "naive set theory", and indeed we will see that it leads us in trouble, when we try to talk about very large sets.

Understanding Sets

▷ Sets are one of the foundations of mathematics,

▷ and one of the most difficult concepts to get right axiomatically

▷ Early Definition Attempt: A set is "everything that can form a unity in the face of God". (Georg Cantor ($*1845$, $\dagger1918$))

▷ For this course: no definition; just intuition (naive set theory)

▷ To understand a set S, we need to determine, what is an element of S and what isn't.

- We can represent sets by
  - listing the elements within curly brackets: e.g. $\{a, b, c\}$
  - describing the elements via a property: $\{x \mid x \text{ has property } P\}$
  - stating element-hood ($a \in S$) or not ($b \notin S$).
- **Axiom 3.5.1** Every set we can write down actually exists! (Hidden Assumption)

  Warning: Learn to distinguish between objects and their representations!($\{a, b, c\}$ and $\{b, a, a, c\}$ are different representations of the same set)

Indeed it is very difficult to define something as foundational as a set. We want sets to be collections of objects, and we want to be as unconstrained as possible as to what their elements can be. But what then to say about them? Cantor's intuition is one attempt to do this, but of course this is not how we want to define concepts in math.

$$AA \begin{matrix} a \\ b \\ b \end{matrix}$$

So instead of defining sets, we will directly work with representations of sets. For that we only have to agree on how we can write down sets. Note that with this practice, we introduce a hidden assumption: called set comprehension, i.e. that every set we can write down actually exists. We will see below that we cannot hold this assumption.

Now that we can represent sets, we want to compare them. We can simply define relations between sets using the three set description operations introduced above.

▷ Relations between Sets

- set equality: $(A \equiv B) :\Leftrightarrow (\forall x . x \in A \Leftrightarrow x \in B)$
- subset: $(A \subseteq B) :\Leftrightarrow (\forall x . x \in A \Rightarrow x \in B)$
- proper subset: $(A \subset B) :\Leftrightarrow (A \subseteq B) \wedge (A \not\equiv B)$
- superset: $(A \supseteq B) :\Leftrightarrow (\forall x . x \in B \Rightarrow x \in A)$
- proper superset: $(A \supset B) :\Leftrightarrow (A \supseteq B) \wedge (A \not\equiv B)$

We want to have some operations on sets that let us construct new sets from existing ones. Again, we can define them.

Operations on Sets

- union: $A \cup B := \{x \mid x \in A \vee x \in B\}$
- union over a collection: Let I be a set and S_i a family of sets indexed by I, then $\bigcup_{i \in I} S_i := \{x \mid \exists i \in I . x \in S_i\}$.
- intersection: $A \cap B := \{x \mid x \in A \wedge x \in B\}$

▷ intersection over a collection: Let I be a set and S_i a family of sets indexed by I, then $\bigcap_{i\in I} S_i := \{x \,|\, \forall i \in I.x \in S_i\}$.

▷ set difference: $A\backslash B := \{x \,|\, x \in A \wedge x \notin B\}$

▷ the power set: $\mathcal{P}(A) := \{S \,|\, S \subseteq A\}$

▷ the empty set: $\forall x.x \notin \emptyset$

▷ Cartesian product: $A \times B := \{(a, b) \,|\, a \in A \wedge b \in B\}$, call (a, b) pair.

▷ n-fold Cartesian product: $A_1 \times \ldots \times A_n := \{\langle a_1, \ldots, a_n\rangle \,|\, \forall i.1 \leq i \leq n \Rightarrow a_i \in A_i\}$, call $\langle a_1, \ldots, a_n\rangle$ an n-tuple

▷ n-dim Cartesian space: $A^n := \{\langle a_1, \ldots, a_n\rangle \,|\, 1 \leq i \leq n \Rightarrow a_i \in A\}$, call $\langle a_1, \ldots, a_n\rangle$ a vector

▷ **Definition 3.5.2** We write $\bigcup_{i=1}^n S_i$ for $\bigcup_{i\in\{i\in\mathbb{N} \,|\, 1\leq i\leq n\}} S_i$ and $\bigcap_{i=1}^n S_i$ for $\bigcap_{i\in\{i\in\mathbb{N} \,|\, 1\leq i\leq n\}} S_i$.

Finally, we would like to be able to talk about the number of elements in a set. Let us try to define that.

Sizes of Sets

▷ We would like to talk about the size of a set. Let us try a definition

▷ **Definition 3.5.3** The size $\#(A)$ of a set A is the number of elements in A.

▷ **Conjecture 3.5.4** *Intuitively we should have the following identities:*

- ▷ $\#(\{a, b, c\}) = 3$
- ▷ $\#(\mathbb{N}) = \infty$ *(infinity)*
- ▷ $\#(A \cup B) \leq \#(A) + \#(B)$ *(⚠ cases with ∞)*
- ▷ $\#(A \cap B) \leq \mathit{min}(\#(A), \#(B))$
- ▷ $\#(A \times B) = \#(A) \cdot \#(B)$

▷ But how do we prove any of them? (what does "number of elements" mean anyways?)

▷ Idea: We need a notion of "counting", associating every member of a set with a unary natural number.

▷ Problem: How do we "associate elements of sets with each other"? (wait for bijective functions)

Once we try to prove the identifies from Conjecture 3.5.4 we get into problems. Even though the notion of "counting the elements of a set" is intuitively clear (indeed we have been using that since we were kids), we do not have a mathematical way of talking about associating numbers with objects in a way that avoids double counting and skipping. We will have to postpone the discussion of sizes until we do.

But before we delve in to the notion of relations and functions that we need to associate set members and counting let us now look at large sets, and see where this gets us.

Sets can be Mind-boggling

- ▷ sets seem so simple, but are really quite powerful (no restriction on the elements)
- ▷ There are very large sets, e.g. "the set $\mathcal{S}$ of all sets"
  - ▷ contains the $\emptyset$,
  - ▷ for each object O we have $\{O\}, \{\{O\}\}, \{O, \{O\}\}, \ldots \in \mathcal{S}$,
  - ▷ contains all unions, intersections, power sets,
  - ▷ contains itself: $\mathcal{S} \in \mathcal{S}$ (scary!)
- ▷ Let's make $\mathcal{S}$ less scary

A less scary $\mathcal{S}$?

- ▷ Idea: how about the "set $\mathcal{S}'$ of all sets that do not contain themselves"
- ▷ Question: is $\mathcal{S}' \in \mathcal{S}'$? (were we successful?)
  - ▷ suppose it is, then then we must have $\mathcal{S}' \notin \mathcal{S}'$, since we have explicitly taken out the sets that contain themselves
  - ▷ suppose it is not, then have $\mathcal{S}' \in \mathcal{S}'$, since all other sets are elements.

  In either case, we have $\mathcal{S}' \in \mathcal{S}'$ iff $\mathcal{S}' \notin \mathcal{S}'$, which is a contradiction! (Russell's Antinomy [Bertrand Russell '03])
- ▷ Does MathTalk help?: no: $\mathcal{S}' := \{m \,|\, m \notin m\}$
  - ▷ MathTalk allows statements that lead to contradictions, but are legal wrt. "vocabulary" and "grammar".
- ▷ We have to be more careful when constructing sets! (axiomatic set theory)
- ▷ for now: stay away from large sets. (stay naive)

Even though we have seen that naive set theory is inconsistent, we will use it for this course. But we will take care to stay away from the kind of large sets that we needed to construct the paradox.

Now we will take a closer look at two very fundamental notions in mathematics that can be built on the notion of sets introduced above: relations and functions. We have already encountered functions and relations as set operations — e.g. the elementhood relation $\in$ which relates a set to its elements or the power set function that takes a set and produces another (its power set).

3.6 Relations

Intuitively, relations are mathematical objects that take arguments and state whether they are related in a partiular way.

Relations

- **Definition 3.6.1** $R \subseteq A \times B$ is a (binary) relation between A and B.
- **Definition 3.6.2** If $A = B$ then R is called a relation on A.
- **Definition 3.6.3** $R \subseteq A \times B$ is called total iff $\forall x \in A.\exists y \in B.(x,y) \in R$.
- **Definition 3.6.4** $R^{-1} := \{(y,x) \mid (x,y) \in R\}$ is the converse relation of R.
- Note: $R^{-1} \subseteq B \times A$.
- The composition of $R \subseteq A \times B$ and $S \subseteq B \times C$ is defined as $S \circ R := \{(a,c) \in A \times C \mid \exists b \in B.(a,b) \in R \wedge (b,c) \in$
- **Example 3.6.5** relation $\subseteq$, $=$, has_color
- Note: we do not really need ternary, quaternary, ... relations
  - Idea: Consider $A \times B \times C$ as $A \times (B \times C)$ and $\langle a,b,c \rangle$ as $(a,(b,c))$
  - we can (and often will) see $\langle a,b,c \rangle$ as $(a,(b,c))$ different representations of the same object.

We will need certain classes of relations in following, so we introduce the necessary abstract properties of relations. We will later combine these to obtain types of relations that behave like well-known ones like the *less than*, *less-or-equal*, or the equality relation.

Properties of binary Relations

- **Definition 3.6.6 (Relation Properties)** A relation $R \subseteq A \times A$ is called
  - reflexive on A, iff $\forall a \in A.(a,a) \in R$
  - irreflexive on A, iff $\forall a \in A.(a,a) \notin R$
  - symmetric on A, iff $\forall a,b \in A.(a,b) \in R \Rightarrow (b,a) \in R$
  - asymmetric on A, iff $\forall a,b \in A.(a,b) \in R \Rightarrow (b,a) \notin R$
  - antisymmetric on A, iff $\forall a,b \in A.((a,b) \in R \wedge (b,a) \in R) \Rightarrow a = b$
  - transitive on A, iff $\forall a,b,c \in A.((a,b) \in R \wedge (b,c) \in R) \Rightarrow (a,c) \in R$
  - equivalence relation on A, iff R is reflexive, symmetric, and transitive.
- **Example 3.6.7** The equality relation is an equivalence relation on any set.
- **Example 3.6.8** On sets of persons, the "mother-of" relation is an non-symmetric, non-reflexive relation.

Indeed the equivalencerelation defined last is a generalization of the equality relation – which is symmectric, reflexive, and transitive. We will see later that any equivalence relation behaves a bit like equality: we can do the same things with it. That makes the class of equivalence relations useful and interesting.

The abstract properties defined above allow us to easily define another very important class of relations, the ordering relations, which generalize the well-known *less-than* and *less-than-or-equal* relations: We just combine some other elementary properties.

Strict and Non-Strict Partial Orders

▷ **Definition 3.6.9** A relation $R \subseteq A \times A$ is called

- ▷ partial order on A, iff R is reflexive, antisymmetric, and transitive on A.
- ▷ strict partial order on A, iff it is irreflexive and transitive on A.

▷ In contexts, where we have to distinguish between strict and non-strict ordering relations, we often add an adjective like *non-strict* or *weak* or *reflexive* to the term *partial order*. We will usually write strict partial orderings with asymmetric symbols like $\prec$, and non-strict ones by adding a line that reminds of equality, e.g. $\preceq$.

▷ **Definition 3.6.10 (Linear order)** A partial order is called linear on A, iff all elements in A are comparable, i.e. if $(x,y) \in R$ or $(y,x) \in R$ for all $x,y \in A$.

▷ **Example 3.6.11** The $\leq$ relation is a linear order on $\mathbb{N}$ (all elements are comparable)

▷ **Example 3.6.12** The "ancestor-of" relation is a partial order that is not linear.

▷ **Lemma 3.6.13** *Strict partial orderings are asymmetric.*

▷ Proof Sketch: By contradiction: If $(a,b) \in R$ and $(b,a) \in R$, then $(a,a) \in R$ by transitivity □

▷ **Lemma 3.6.14** *If $\preceq$ is a (non-strict) partial order, then $\prec := \{(a,b) \mid (a \preceq b) \wedge a \neq b\}$ is a strict partial order. Conversely, if $\prec$ is a strict partial order, then $\preceq := \{(a,b) \mid (a \prec b) \vee a = b\}$ is a non-strict partial order.*

3.7 Functions

Intuitively, functions are mathematical objects that take arguments (as input) and return a result (as output). This already suggests defining them as special relations. But we have to be careful here; we want to specify the sets where the inputs can come from (the domain) and where the outputs go (the codomain), and whether a function needs to have outputs for all possible inputs (from the domain). This leads us to a distinction of "total" and "partial" functions.

Functions (as special relations)

▷ **Definition 3.7.1** $f \subseteq X \times Y$, is called a partial function, iff for all $x \in X$ there is at most one $y \in Y$ with $(x,y) \in f$.

- ▷ **Notation 3.7.2** $f\colon X \rightharpoonup Y; x \mapsto y$ if $(x,y) \in f$ (arrow notation)
- ▷ call X the domain (write $\mathbf{dom}(f)$), and Y the codomain $(\mathbf{codom}(f))$ (come with f)
- ▷ **Notation 3.7.3** $f(x) = y$ instead of $(x,y) \in f$ (function application)

▷ **Definition 3.7.4** We call a partial function $f\colon X \rightharpoonup Y$ undefined at $x \in X$, iff $(x,y) \notin f$ for all $y \in Y$. (write $f(x) = \bot$)

▷ **Definition 3.7.5** If $f\colon X \rightharpoonup Y$ is a total relation, we call f a total function and write $f\colon X \to Y$. $(\forall x \in X.\exists^1 y \in Y.(x,y) \in f)$

- ▷ **Notation 3.7.6** $f\colon x \mapsto y$ if $(x,y) \in f$ (arrow notation)

▷ **Definition 3.7.7** The identity function on a set A is defined as $\mathrm{Id}_A := \{(a,a) \mid a \in A\}$.

⚠: this probably does not conform to your intuition about functions. Do not worry, just think of them as two different things they will come together over time. (In this course we will use "function" as defined here!)

As functions are foundational in mathematics, we see a lot of suggestive notations, but they should not hide the fact that functions are just "right-unique" relations. Definition 3.7.1 gives us a solid foundation on which we can reason safely about functions.

Remark: It is crucial to understand that the domain and codomain is part of a functions (partial or total). In particular, a change in domain or codomain changes the function.

▷ **Example 3.7.8** The functions $f\colon \mathbb{R} \to \mathbb{R}; x \mapsto |x|$, $g\colon \mathbb{R}^+ \to \mathbb{R}; x \mapsto |x|$, and $h\colon \mathbb{R}^+ \to \mathbb{R}^+; x \mapsto |x|$ are different – they have different domains. In particular have different properties as we will see later.

Now that we have defined functions, it is natural to think about sets of functions from a given domain to a given codomain. If the latter are small enough, we can even write down the full set as a collection of sets of pairs.

Function Spaces

▷ **Definition 3.7.9** Given sets A and B We will call the set $A \to B$ $(A \rightharpoonup B)$ of all (partial) functions from A to B the (partial) function space from A to B.

▷ **Example 3.7.10** Let $\mathbb{B} := \{0,1\}$ be a two-element set, then

$$\mathbb{B} \to \mathbb{B} = \{\{(0,0),(1,0)\},\{(0,1),(1,1)\},\{(0,1),(1,0)\},\{(0,0),(1,1)\}\}$$

$$\mathbb{B} \rightharpoonup \mathbb{B} = \mathbb{B} \to \mathbb{B} \cup \{\emptyset,\{(0,0)\},\{(0,1)\},\{(1,0)\},\{(1,1)\}\}$$

▷ as we can see, all of these functions are finite (as relations)

We will now introduce still another notation for functions, which is commonly used in Computer Science, since it is more explicit in the arguments a function takes and allows to construct functions without directly having to give them a name.

Lambda-Notation for Functions

- Problem: In maths we write $f(x) := x^2 + 3x + 5$ to define a function f, then we can talk about $\mathbf{dom}(f)$. But if we do not want to use a name, we can only say $\mathbf{dom}(\{(x, y) \in \mathbb{R} \times \mathbb{R} \mid y = x^2 + 3x + 5\})$
- Problem: It is common mathematical practice to write things like $f_a(x) = ax^2 + 3x + 5$, meaning e.g. that we have a collection $\{f_a \mid a \in A\}$ of functions. (is a an argument or jut a "parameter"?)
- **Definition 3.7.11** To make the role of arguments extremely clear, we write functions in λ-notation. For $f = \{(x, E) \mid x \in X\}$, where E is an expression, we write $\lambda x \in X.E$.
- **Example 3.7.12** The simplest function we always try everything on is the identity function:

$$\begin{aligned}\lambda n \in \mathbb{N}.n &= \{(n, n) \mid n \in \mathbb{N}\} = \mathrm{Id}_{\mathbb{N}} \\ &= \{(0, 0), (1, 1), (2, 2), (3, 3), \ldots\}\end{aligned}$$

- **Example 3.7.13** We can also to more complex expressions, here we take the square function

$$\begin{aligned}\lambda x \in \mathbb{N}.x^2 &= \{(x, x^2) \mid x \in \mathbb{N}\} \\ &= \{(0, 0), (1, 1), (2, 4), (3, 9), \ldots\}\end{aligned}$$

- **Example 3.7.14** λ-notation also works for more complicated domains. In this case we have pairs as arguments.

$$\begin{aligned}\lambda(x, y) \in \mathbb{N} \times \mathbb{N}.x + y &= \{((x, y), x + y) \mid x \in \mathbb{N} \wedge y \in \mathbb{N}\} \\ &= \{((0, 0), 0), ((0, 1), 1), ((1, 0), 1), \\ &\qquad ((1, 1), 2), ((0, 2), 2), ((2, 0), 2), \ldots\}\end{aligned}$$

The three properties we define next give us information about whether we can invert functions, i.e. whether the converse relation of a given function is again a (partial) function.

Properties of functions, and their converses

- **Definition 3.7.15** A function $f : S \to T$ is called
  - injective iff $\forall x, y \in S.f(x) = f(y) \Rightarrow x = y$.
  - surjective iff $\forall y \in T.\exists x \in S.f(x) = y$.
  - bijective iff f is injective and surjective.

▷ **Observation 3.7.16** *If f is injective, then the converse relation f^{-1} is a partial function.*

▷ **Observation 3.7.17** *If f is surjective, then the converse f^{-1} is a total relation.*

▷ **Definition 3.7.18** If f is bijective, call the converse relation inverse function, we (also) write it as f^{-1}.

▷ **Observation 3.7.19** *If f is bijective, then f^{-1} is a total function.*

▷ **Observation 3.7.20** *If $f: A \to B$ is bijective, then $f \circ f^{-1} = Id_A$ and $f^{-1} \circ f = Id_B$.*

▷ **Example 3.7.21** The function $\nu: \mathbb{N}_1 \to \mathbb{N}$ with $\nu(o) = 0$ and $\nu(s(n)) = \nu(n) + 1$ is a bijection between the unary natural numbers and the natural numbers you know from elementary school.

Note: Sets that can be related by a bijection are often considered equivalent, and sometimes confused. We will do so with $\mathbb{N}_1$ and $\mathbb{N}$ in the future

With the notion of bijectivity defined above, we can make progress on the notion of the "size" of a set we failed on in Definition 3.5.3.

▷ Cardinality of Sets

▷ Now, we can make the notion of the size of a set formal, since we can associate members of sets by bijective functions.

▷ **Definition 3.7.22** We say that a set A is finite and has cardinality $\#(A) \in \mathbb{N}$, iff there is a bijective function $f: A \to \{n \in \mathbb{N} \mid n < \#(A)\}$.

▷ **Definition 3.7.23** We say that a set A is countably infinite, iff there is a bijective function $f: A \to \mathbb{N}$. A set is called countable, iff it is finite or countably infinite.

▷ **Theorem 3.7.24** *We have the following identities for finite sets A and B*

- ▹ $\#(\{a, b, c\}) = 3$ (*e.g. choose $f = \{(a, 0), (b, 1), (c, 2)\}$*)
- ▹ $\#(A \cup B) \leq \#(A) + \#(B)$
- ▹ $\#(A \cap B) \leq min(\#(A), \#(B))$
- ▹ $\#(A \times B) = \#(A) \cdot \#(B)$

▷ With the definition above, we can prove them (last three $\leadsto$ Homework)

Next we turn to operations on functions. These are actually functions themselves, they take functions as arguments, and may return functions as results. We call such functions higher-order. For instance the composition function takes two functions as arguments and yields a function as a result.

Operations on Functions

- ▷ **Definition 3.7.25** If $f \in A \to B$ and $g \in B \to C$ are functions, then we call

$$g \circ f \colon A \to C; x \mapsto g(f(x))$$

the composition of g and f (read g “after” f).

- ▷ **Definition 3.7.26** Let $f \in A \to B$ and $C \subseteq A$, then we call the function $f|_C := \{(c,b) \in f \,|\, c \in C\}$ the restriction of f to C.

- ▷ **Definition 3.7.27** Let $f \colon A \to B$ be a function, $A' \subseteq A$ and $B' \subseteq B$, then we call
  - ▷ $f(A') := \{b \in B \,|\, \exists a \in A'.(a,b) \in f\}$ the image of A' under f,
  - ▷ $\mathbf{Im}(f) := f(A)$ the image of f, and
  - ▷ $f^{-1}(B') := \{a \in A \,|\, \exists b \in B'.(a,b) \in f\}$ the pre-image of B' under f

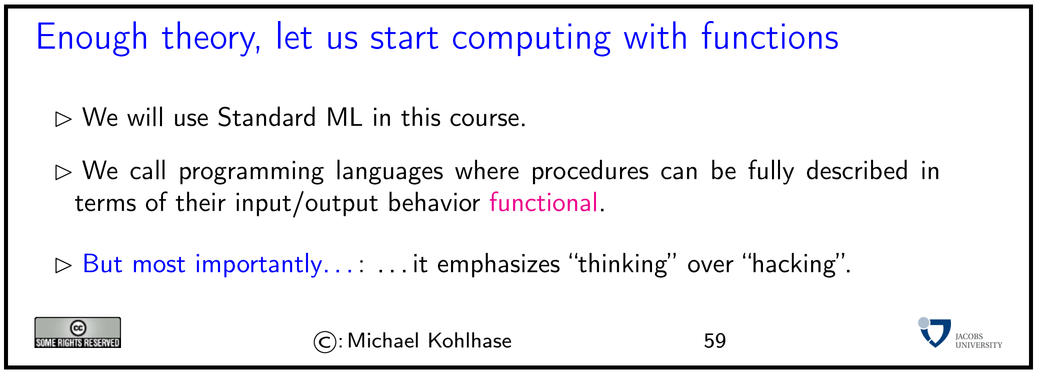
Chapter 4

Computing with Functions over Inductively Defined Sets

4.1 Standard ML: A Functional Programming Language

We will use the language SML for the course. This has three reasons

- The mathematical foundations of the computational model of SML is very simple: it consists of functions, which we have already studied. You will be exposed to imperative programming languages (`C` and `C++`) in the labs and later in your studies.
- As a functional programming language, SML introduces two very important concepts in a very clean way: typing and recursion.
- Finally, SML has a very useful secondary virtue for a course at Jacobs University, where students come from very different backgrounds: it provides a (relatively) level playing ground, since it is unfamiliar to all students.



Enough theory, let us start computing with functions

▷ We will use Standard ML in this course.

▷ We call programming languages where procedures can be fully described in terms of their input/output behavior functional.

▷ But most importantly. . . : . . . it emphasizes “thinking” over “hacking”.

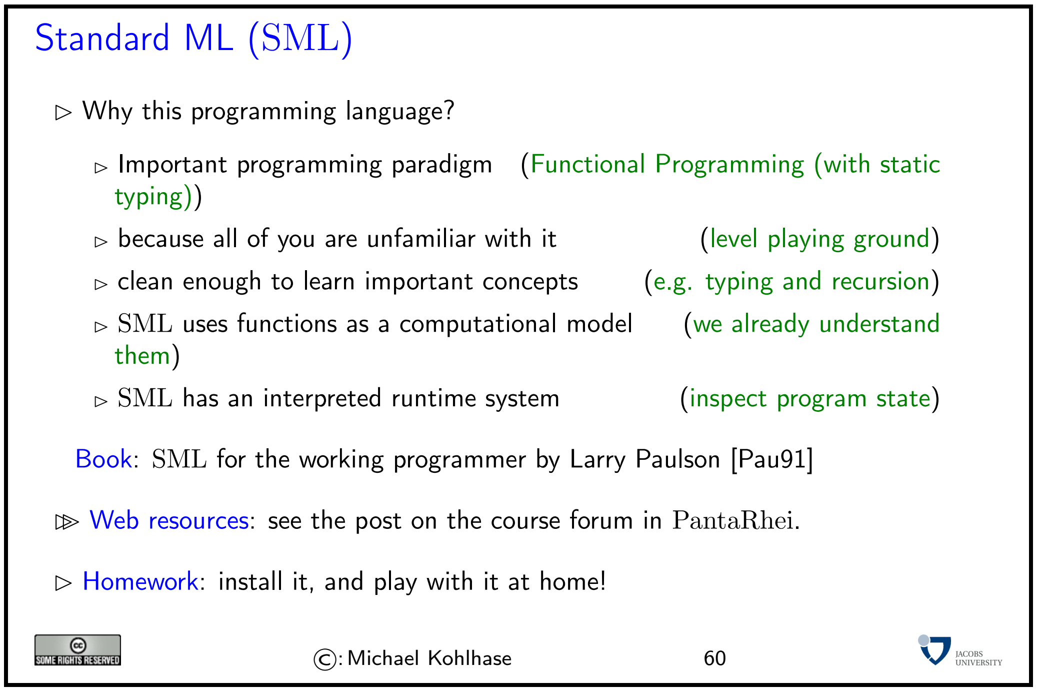
Generally, when choosing a programming language for a computer science course, there is the choice between languages that are used in industrial practice (`C`, `C++`, `Java`, `FORTRAN`, `COBOL`,. . .) and languages that introduce the underlying concepts in a clean way. While the first category have the advantage of conveying important practical skills to the students, we will follow the motto “No, let’s think” for this course and choose ML for its clarity and rigor. In our experience, if the concepts are clear, adapting the particular syntax of a industrial programming language is not that difficult.

Historical Remark: The name ML comes from the phrase “Meta Language”: ML was developed as the scripting language for a tactical theorem prover[1] — a program that can construct mathematical

[1] The “Edinburgh LCF” system

proofs automatically via “tactics” (little proof-constructing programs). The idea behind this is the following: ML has a very powerful type system, which is expressive enough to fully describe proof data structures. Furthermore, the ML compiler type-checks all ML programs and thus guarantees that if an ML expression has the type $A \to B$, then it implements a function from objects of type A to objects of type B. In particular, the theorem prover only admitted tactics, if they were type-checked with type $\mathcal{P} \to \mathcal{P}$, where $\mathcal{P}$ is the type of proof data structures. Thus, using ML as a meta-language guaranteed that theorem prover could only construct valid proofs.

The type system of ML turned out to be so convenient (it catches many programming errors before you even run the program) that ML has long transcended its beginnings as a scripting language for theorem provers, and has developed into a paradigmatic example for functional programming languages.



Standard ML (SML)

▷ Why this programming language?

- ▹ Important programming paradigm (Functional Programming (with static typing))
- ▹ because all of you are unfamiliar with it (level playing ground)
- ▹ clean enough to learn important concepts (e.g. typing and recursion)
- ▹ SML uses functions as a computational model (we already understand them)
- ▹ SML has an interpreted runtime system (inspect program state)

Book: SML for the working programmer by Larry Paulson [Pau91]

▷ Web resources: see the post on the course forum in PantaRhei.

▷ Homework: install it, and play with it at home!

Disclaimer: We will not give a full introduction to SML in this course, only enough to make the course self-contained. There are good books on ML and various web resources:

- A book by Bob Harper (CMU) http://www-2.cs.cmu.edu/~rwh/smlbook/
- The Moscow ML home page, one of the ML’s that you can try to install, it also has many interesting links http://www.dina.dk/~sestoft/mosml.html
- The home page of SML-NJ (SML of New Jersey), the standard ML http://www.smlnj.org/ also has a ML interpreter and links Online Books, Tutorials, Links, FAQ, etc. And of course you can download SML from there for Unix as well as for Windows.
- A tutorial from Cornell University. It starts with "Hello world" and covers most of the material we will need for the course. http://www.cs.cornell.edu/gries/CSCI4900/ML/gimlFolder/manual.html
- and finally a page on ML by the people who originally invented ML: http://www.lfcs.inf.ed.ac.uk/software/ML/

One thing that takes getting used to is that SML is an interpreted language. Instead of transforming the program text into executable code via a process called “compilation” in one go, the SML interpreter provides a run time environment that can execute well-formed program snippets in a dialogue with the user. After each command, the state of the run-time systems can be inspected to judge the effects and test the programs. In our examples we will usually exhibit the input to the interpreter and the system response in a program block of the form

```
– input to the interpreter
system response
```

Programming in SML (Basic Language)

- ▷ Generally: start the SML interpreter, play with the program state.
- ▷ **Definition 4.1.1 (Predefined objects in SML)** (SML comes with a basic inventory)
  - ▷ basic types `int`, `real`, `bool`, `string` , ...
  - ▷ basic type constructors $->$, $*$,
  - ▷ basic operators numbers, `true`, `false`, $+$, $*$, $-$, $>$, ^, ... (overloading)
  - ▷ control structures **if** Φ **then** E_1 **else** E_2;
  - ▷ comments (∗this is a comment ∗)

One of the most conspicuous features of SML is the presence of types everywhere.

Definition 4.1.2 types are program constructs that classify program objects into categories.

In SML, literally every object has a type, and the first thing the interpreter does is to determine the type of the input and inform the user about it. If we do something simple like typing a number (the input has to be terminated by a semicolon), then we obtain its type:

```
– 2;
val it = 2 : int
```

In other words the SML interpreter has determined that the input is a value, which has type “integer”. At the same time it has bound the identifier it to the number 2. Generally it will always be bound to the value of the last successful input. So we can continue the interpreter session with

```
– it;
val it = 2 : int
– 4.711;
val it = 4.711 : real
– it;
val it = 4.711 : real
```

Programming in SML (Declarations)

- ▷ **Definition 4.1.3** declarations bind variables (abbreviations for convenience)
  - ▷ value declarations e.g. **val** pi = 3.1415;
  - ▷ type declarations e.g. **type** twovec = int ∗ int;
  - ▷ function declarations e.g. **fun** square (x:real) = x∗x; (leave out type, if unambiguous)

A function declaration only declares the function name as a globally visible name. The formal parameters in brackets are only visible in the function body.

▷ SML functions that have been declared can be applied to arguments of the right type, e.g. square 4.0, which evaluates to 4.0 ∗ 4.0 and thus to 16.0.

▷ **Definition 4.1.4** A local declaration uses **let** to bind variables in its scope (delineated by **in** and **end**).

▷ **Example 4.1.5** Local definitions can shadow existing variables.

```
- val test = 4;
val it = 4 : int
- let val test = 7 in test * test end;
val it = 49 :int
- test;
val it = 4 : int
```

While the previous inputs to the interpreters do not change its state, declarations do: they bind identifiers to values. In the first example, the identifier twovec to the type int ∗ int, i.e. the type of pairs of integers. Functions are declared by the **fun** keyword, which binds the identifier behind it to a function object (which has a type; in our case the function type real −> real). Note that in this example we annotated the formal parameter of the function declaration with a type. This is always possible, and in this necessary, since the multiplication operator is overloaded (has multiple types), and we have to give the system a hint, which type of the operator is actually intended.

Programming in SML (Component Selection)

▷ **Definition 4.1.6** Using structured patterns, we can declare more than one variable. We call this pattern matching.

▷ **Example 4.1.7 (Component Selection)** (very convenient)

```
- val unitvector = (1,1);
val unitvector = (1,1) : int * int
- val (x,y) = unitvector
val x = 1 : int
val y = 1 : int
```

▷ **Definition 4.1.8** anonymous variables (if we are not interested in one value)

```
- val (x,_) = unitvector;
val x = 1 :int
```

▷ **Example 4.1.9** We can define the selector function for pairs in SML as

```
- fun first (p) = let val (x,_) = p in x end;
val first = fn : 'a * 'b -> 'a
```

Note the type: SML supports universal types with type variables 'a, 'b,. . . .
⊳▷ first is a function that takes a pair of type 'a∗'b as input and gives an object of type 'a as output.

Another unusual but convenient feature realized in SML is the use of pattern matching. In pattern matching we allow to use variables (previously unused identifiers) in declarations with the understanding that the interpreter will bind them to the (unique) values that make the declaration true. In our example the second input contains the variables x and y. Since we have bound the identifier unitvector to the value (1,1), the only way to stay consistent with the state of the interpreter is to bind both x and y to the value 1.

Note that with pattern matching we do not need explicit selector functions, i.e. functions that select components from complex structures that clutter the namespaces of other functional languages. In SML we do not need them, since we can always use pattern matching inside a **let** expression. In fact this is considered better programming style in SML.

What's next?

More SML constructs and general theory of functional programming.

One construct that plays a central role in functional programming is the data type of lists. SML has a built-in type constructor for lists. We will use list functions to acquaint ourselves with the essential notion of recursion.

Using SML lists

- ▷ SML has a built-in "list type" (actually a list type constructor)
- ▷ given a type ty, list ty is also a type.

```
- [1,2,3];
val it = [1,2,3] : int list
```

- ▷ constructors nil and :: (nil $\widehat{=}$ empty list, :: $\widehat{=}$ list constructor "cons")

```
- nil;
val it = [] : 'a list
- 9::nil;
val it = [9] : int list
```

- ▷ A simple recursive function: creating integer intervals

```
- fun upto (m,n) = if m>n then nil else m::upto(m+1,n);
val upto = fn : int * int -> int list
- upto(2,5);
val it = [2,3,4,5] : int list
```

Question: What is happening here, we define a function by itself? (circular?)

A constructor is an operator that "constructs" members of an SML data type.

The type of lists has two constructors: nil that "constructs" a representation of the empty list, and the "list constructor" :: (we pronounce this as "cons"), which constructs a new list h::l from a list l by pre-pending an element h (which becomes the new head of the list).

Note that the type of lists already displays the circular behavior we also observe in the function definition above: A list is either empty or the cons of a list. We say that the type of lists is inductive or inductively defined.

In fact, the phenomena of recursion and inductive types are inextricably linked, we will explore this in more detail below.

▷ Defining Functions by Recursion

▷ Observation: SML allows to call a function already in the function definition.

```
fun upto (m,n) = if m>n then nil else m::upto(m+1,n);
```

▷ Evaluation in SML is "call-by-value" i.e. to whenever we encounter a function applied to arguments, we compute the value of the arguments first.

▷ **Example 4.1.10** We have the following evaluation trace with result [2,3,4]

$$\text{upto}(2,4) \rightsquigarrow 2::\text{upto}(3,4) \rightsquigarrow 2::(3::\text{upto}(4,4)) \rightsquigarrow 2::(3::(4::\text{nil}))$$

▷ **Definition 4.1.11** We call an SML function recursive, iff the function is called in the function definition.

▷ Note that recursive functions need not terminate, consider the function

```
fun diverges (n) = n + diverges(n+1);
```

which has the evaluation sequence

$$\text{diverges}(1) \rightsquigarrow 1 + \text{diverges}(2) \rightsquigarrow 1 + (2 + \text{diverges}(3)) \rightsquigarrow \ldots$$

The key to understanding recursion is to understand the function as an equation – as the SML syntax already suggests – that replaces any expression that matches the left hand side with a suitably instantiated version of the right hand side. Using this, we can trace the computation (we write $\rightsquigarrow$ for any such replacement).

Note that not all computations stop with a base case: we may have forgotten to specify one, or or computation never reaches it. This is actually a feature of recursion, not a bug. The full power of programming languages necessarily comes with the "ability" to obtain infinite computations. In imperative languages, we call these "infinite loops", in functional programming languges like SML, we speak of "deep recursions".

Normally of course, we want our recursive computations to terminate after a finite number – which can be large – of steps. For that to happen, something needs to become smaller in the computation. In Example 4.1.10, this is the difference between the two arguments, it decreases by one in each step of the computation, and thus finally reaches zero, where the first alternative of the **if** expression applies. In the function of Example 66, the argument does not get smaller, indeed it becomes bigger with every recursive call, leading to the divergent behavior.

In our examples above we used recursion on an argument of type int using if_then_else expression to select between the base case and the step case. Recursion on list types is more elegant, since we can use pattern matching on the arguments.

Defining Functions by cases

▷ Idea: Use the fact that lists are either nil or of the form X::Xs, where X is an element and Xs is a list of elements.

▷ The body of an SML function can be made of several cases separated by the operator |.

▷ **Example 4.1.12** Flattening lists of lists (using the infix append operator @)

```
fun flat [] = [] (∗ base case ∗)
  | flat (h::t) = h @ flat t; (∗ step case ∗)
val flat = fn : 'a list list −> 'a list
```

Let's test it on an argument:

```
flat [["When","shall"],["we","three"],["meet","again"]];
["When","shall","we","three","meet","again"]
```

To understand pattern matching in this paragraph, consider the type string list list – for the argument here, we only need the fact that we are dealing with a list type here. And in those, elements are either nil or of the form cons(h,t) – i.e. a non-empty list made by pre-pending an element h to a list t. With the pattern matching mechanism we can select them directly, e.g.

```
− val h::t = [1,2];
val h = 1 : int
val t = [2] : int list
− val h::t = [1];
val h = 1 : int
val t = [] : int list
```

This is just what we do in the flat function. We have two cases, in the second – the step case – we match the argument (which is non-empty, since the first case already took care of the empty list) with the pattern h::t, which binds the parameters h and t, which we can then use to construct the value of flattening a non-empty list.

Defining functions by cases and recursion is a very important programming mechanism in SML. At the moment we have only seen it for the built-in type of lists. In the future we will see that it can also be used for user-defined data types.

We will now look at the the string type of SML and how to deal with it. But before we do, let us recap what strings are. Strings are just sequences of characters.

Therefore, SML just provides an interface to lists for manipulation.

Lists and Strings

▷ some programming languages provide a type for single characters (strings are lists of characters there)

▷ in SML, string is an atomic type

- ▷ function explode converts from string to char list
- ▷ function implode does the reverse

```
− explode "GenCS 1";
val it = [#"G",#"e",#"n",#"C",#"S",#" ",#"1"] : char list
− implode it;
val it = "GenCS 1" : string
```

Exercise: Try to come up with a function that detects palindromes like 'otto' or 'anna', try also (more at [Pal])

▷ ▹ 'Marge lets Norah see Sharon's telegram', or(up to case, punct and space)
  ▹ 'Ein Neger mit Gazelle zagt im Regen nie' (for German speakers)

The next feature of SML is slightly disconcerting at first, but is an essential trait of functional programming languages: functions are first-class objects. We have already seen that they have types, now, we will see that they can also be passed around as argument and returned as values. For this, we will need a special syntax for functions, not only the **fun** keyword that declares functions.

Higher-Order Functions

▷ Idea: pass functions as arguments (functions are normal values.)

▷ **Example 4.1.13** Mapping a function over a list

```
- fun f x = x + 1;
- map f [1,2,3,4];
[2,3,4,5] : int list
```

▷ **Example 4.1.14** We can program the map function ourselves!

```
  fun mymap (f, nil) = nil
    | mymap (f, h::t) = (f h) :: mymap (f,t);
```

▷ **Example 4.1.15** declaring functions (yes, functions are normal values.)

```
- val identity = fn x => x;
val identity = fn : 'a -> 'a
- identity(5);
val it = 5 : int
```

▷ **Example 4.1.16** returning functions: (again, functions are normal values.)

```
- val constantly = fn k => (fn a => k);
- (constantly 4) 5;
val it = 4 : int
- fun constantly k a = k;
```

One of the neat uses of higher-order function is that it is possible to re-interpret binary functions as unary ones using a technique called “Currying” after the Logician Haskell Brooks Curry (∗1900, †1982). Of course we can extend this to higher arities as well. So in theory we can consider n-ary functions as syntactic sugar for suitable higher-order functions.

Cartesian and Cascaded Functions

▷ We have not been able to treat binary, ternary,. . . functions directly

▷ Workaround 1: Make use of (Cartesian) products (unary functions on tuples)

▷ **Example 4.1.17** $+ \colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ with $+((3,2))$ instead of $+(3,2)$

```
fun cartesian_plus (x:int,y:int) = x + y;
cartesian_plus : int * int -> int
```

Workaround 2: Make use of functions as results

▷▷ **Example 4.1.18** $+ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ withn $+(3)(2)$ instead of $+((3,2))$.

```
fun cascaded_plus (x:int) = (fn y:int => x + y);
cascaded_plus : int -> (int -> int)
```

Note: cascaded_plus can be applied to only one argument: cascaded_plus 1 is the function (**fn** y:int => 1 + y), which increments its argument.

SML allows both Cartesian- and cascaded functions, since we sometimes want functions to be flexible in function arities to enable reuse, but sometimes we want rigid arities for functions as this helps find programming errors.

▷ Cartesian and Cascaded Functions (Brackets)

▷ **Definition 4.1.19** Call a function Cartesian, iff the argument type is a product type, call it cascaded, iff the result type is a function type.

▷ **Example 4.1.20** the following function is both Cartesian and cascading

```
- fun both_plus (x:int,y:int) = fn (z:int) => x + y + z;
val both_plus (int * int) -> (int -> int)
```

Convenient: Bracket elision conventions

▷ ▷ $e_1\ e_2\ e_3 \rightsquigarrow (e_1\ e_2)\ e_3$ (function application associates to the left)

▷ $\tau_1 {->} \tau_2 {->} \tau_3 \rightsquigarrow \tau_1 {->} (\tau_2 {->} \tau_3)$ (function types associate to the right)

▷ SML uses these elision rules

```
- fun both_plus (x:int,y:int) = fn (z:int) => x + y + z;
val both_plus int * int -> int -> int
cascaded_plus 4 5;
```

▷ Another simplification (related to those above)

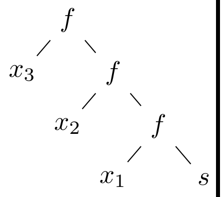
```
- fun cascaded_plus x y = x + y;
val cascaded_plus : int -> int -> int
```

We will now introduce two very useful higher-order functions. The folding operators iterate a binary function over a list given a start value. The folding operators come in two varieties: foldl ("fold left") nests the function in the right argument, and foldr ("fold right") in the left argument.

Folding Operators

▷ **Definition 4.1.21** SML provides the left folding operator to realize a recurrent computation schema

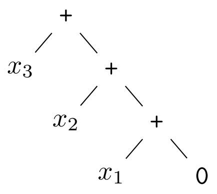
```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldl f s [x1,x2,x3] = f(x3,f(x2,f(x1,s)))
```



We call the function f the iterator and s the start value

▷ **Example 4.1.22** Folding the iterator **op+** with start value 0:

```
foldl op+ 0 [x1,x2,x3] = x3+(x2+(x1+0))
```



Thus the function given by the expression foldl **op+** 0 adds the elements of integer lists.

Summing over a list is the prototypical operation that is easy to achieve. Note that a sum is just a nested addition. So we can achieve it by simply folding addition (a binary operation) over a list (left or right does not matter, since addition is commutative). For computing a sum we have to choose the start value 0, since we only want to sum up the elements in the list (and 0 is the neural element for addition).

Note: We have used the binary function **op+** as the argument to the foldl operator instead of simply + in Example 4.1.22. The reason for this is that the infix notation for x +y is syntactic sugar for **op+**(x,y) and not for +(x,y) as one might think.

Note: We can use any function of suitable type as a first argument for foldl, including functions literally defined by **fn** x =>B or a n-ary function applied to $n-2$ arguments.

Finally note: That foldl is a cascading function. SML could have made it Cartesian, resulting in the type (âĂŹa âĹŮ âĂŹb âĹŠ> âĂŹb) ∗ âĂŹb ∗ âĂŹa list âĹŠ> âĂŹb. But the cascading foldl is more useful as you we do things like

```
- val sum = foldl op+ 0;
val sum = fn : int list -> int
```

Note that we are leaving over the list argument to get a function. If SML used Cartesian, then we would have to define the equivalent

```
fun sum (l) = foldl op+ 0 l
- val sum = fn : int list -> int;
```

which is longer. Also we can pass summation as an argument more elegantly as in

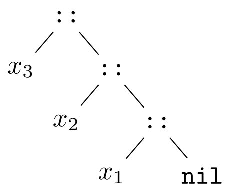
```
map (foldl op+ 0) [[1,2,3],[2,3,4],[67]]
```

with a cascading foldl.

Folding Procedures (continued)

▷ **Example 4.1.23 (Reversing Lists)**

foldl **op**:: nil $[x_1,x_2,x_3]$ = x_3 :: (x_2 :: (x_1:: nil))



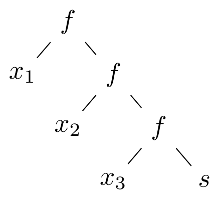
Thus the procedure **fun** rev xs = foldl **op**:: nil xs reverses a list

In Example 4.1.23, we reverse a list by folding the list constructor (which duly constructs the reversed list in the process) over the input list; here the empty list is the right start value.

Folding Procedures (foldr)

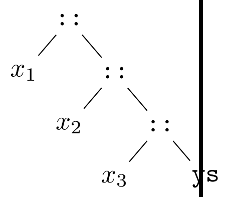
▷ **Definition 4.1.24** The right folding operator foldr is a variant of foldl that processes the list elements in reverse order.

foldr : ('a ∗ 'b −> 'b) −> 'b −> 'a list −> 'b
foldr f s $[x_1,x_2,x_3]$ = f(x_1,f(x_2,f(x_3,s)))



▷ **Example 4.1.25 (Appending Lists)**

foldr **op**:: ys $[x_1,x_2,x_3]$ = x_1 :: (x_2 :: (x_3 :: ys))



fun append(xs,ys) = foldr **op**:: ys xs

In Example 4.1.25 we fold with the list constructor again, but as we are using foldr the list is not reversed. To get the append operation, we use the list in the second argument as a base case of the iteration.

Now that we know some SML

SML is a "functional Programming Language"

What does this all have to do with functions?

Back to Induction, "Peano Axioms" and functions (to keep it simple)

4.2 Inductively Defined Sets and Computation

Let us now go back to looking at concrete functions on the unary natural numbers. We want to convince ourselves that addition is a (binary) function. Of course we will do this by constructing a proof that only uses the axioms pertinent to the unary natural numbers: the Peano Axioms.

What about Addition, is that a function?

▷ Problem: Addition takes two arguments (binary function)

▷ One solution: $+ : \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$ is unary

▷ $+((n, o)) = n$ (base) and $+((m, s(n))) = s(+((m, n)))$ (step)

▷ **Theorem 4.2.1** $+ \subseteq (\mathbb{N}_1 \times \mathbb{N}_1) \times \mathbb{N}_1$ *is a total function.*

▷ We have to show that for all $(n, m) \in \mathbb{N}_1 \times \mathbb{N}_1$ there is exactly one $l \in \mathbb{N}_1$ with $((n, m), l) \in +$.

▷ We will use functional notation for simplicity

But before we can prove function-hood of the addition function, we must solve a problem: addition is a binary function (intuitively), but we have only talked about unary functions. We could solve this problem by taking addition to be a cascaded function, but we will take the intuition seriously that it is a Cartesian function and make it a function from $\mathbb{N}_1 \times \mathbb{N}_1$ to $\mathbb{N}_1$. With this, the proof of functionhood is a straightforward induction over the second argument.

Addition is a total Function

▷ **Lemma 4.2.2** *For all* $(n, m) \in \mathbb{N}_1 \times \mathbb{N}_1$ *there is exactly one* $l \in \mathbb{N}_1$ *with* $+((n, m)) = l$.

▷ Proof: by induction on m. (what else)

P.1 we have two cases

P.1.1 base case $(m = o)$:

P.1.1.1 choose $l := n$, so we have $+((n, o)) = n = l$.

P.1.1.2 For any $l' = +((n, o))$, we have $l' = n = l$. □

P.1.2 step case $(m = s(k))$:

P.1.2.1 assume that there is a unique $r = +((n, k))$, choose $l := s(r)$, so we have $+((n, s(k))) = s(+((n, k))) = s(r)$.

P.1.2.2 Again, for any $l' = +((n, s(k)))$ we have $l' = l$. □

□

▷ **Corollary 4.2.3** $+: \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$ *is a total function.*

The main thing to note in the proof above is that we only needed the Peano Axioms to prove function-hood of addition. We used the induction axiom (P5) to be able to prove something about "all unary natural numbers". This axiom also gave us the two cases to look at. We have used the distinctness axioms (P3 and P4) to see that only one of the defining equations applies, which in the end guaranteed uniqueness of function values.

Reflection: How could we do this?

▷ we have two constructors for $\mathbb{N}_1$: the base element $o \in \mathbb{N}_1$ and the successor function $s: \mathbb{N}_1 \to \mathbb{N}_1$

▷ Observation: Defining Equations for $+$: $+((n, o)) = n$ (base) and $+((m, s(n))) = s(+((m, n)))$ (step)

- ▷ the equations cover all cases: n is arbitrary, $m = o$ and $m = s(k)$(otherwise we could have not proven existence)
- ▷ but not more (no contradictions)

▷ using the induction axiom in the proof of unique existence.

▷ **Example 4.2.4** Defining equations $\delta(o) = o$ and $\delta(s(n)) = s(s(\delta(n)))$

▷ **Example 4.2.5** Defining equations $\mu(l, o) = o$ and $\mu(l, s(r)) = +((\mu(l, r), l))$

▷ Idea: Are there other sets and operations that we can do this way?

- ▷ the set should be built up by "injective" constructors and have an induction axiom ("abstract data type")
- ▷ the operations should be built up by case-complete equations

The specific characteristic of the situation is that we have an inductively defined set: the unary natural numbers, and defining equations that cover all cases (this is determined by the constructors) and that are non-contradictory. This seems to be the pre-requisites for the proof of functionality we have looked up above.

As we have identified the necessary conditions for proving function-hood, we can now generalize the situation, where we can obtain functions via defining equations: we need inductively defined sets, i.e. sets with Peano-like axioms.

This observation directly leads us to a very important concept in computing.

Inductively Defined Sets

▷ **Definition 4.2.6** An inductively defined set $\langle S, C\rangle$ is a set S together with a finite set $C := \{c_i \mid 1 \leq i \leq n\}$ of k_i-ary constructors $c_i: S^{k_i} \to S$ with $k_i \geq 0$, such that

- ▹ if $s_i \in S$ for all $1 \leq i \leq k_i$, then $c_i(s_1, \ldots, s_{k_i}) \in S$ (generated by constructors)
- ▹ all constructors are injective, (no internal confusion)
- ▹ $\mathbf{Im}(c_i) \cap \mathbf{Im}(c_j) = \emptyset$ for $i \neq j$, and (no confusion between constructors)
- ▹ for every $s \in S$ there is a constructor $c \in C$ with $s \in \mathbf{Im}(c)$. (no junk)

▷ Note that we also allow nullary constructors here.

▷ **Example 4.2.7** $\langle \mathbb{N}_1, \{s, o\} \rangle$ is an inductively defined set.

▷ Proof: We check the three conditions in Definition 4.2.6 using the Peano Axioms

P.1 Generation is guaranteed by **P**1 and **P**2

P.2 Internal confusion is prevented **P**4

P.3 Inter-constructor confusion is averted by **P**3

P.4 Junk is prohibited by **P**5. □

This proof shows that the Peano Axioms are exactly what we need to establish that $\langle \mathbb{N}_1, \{s, o\} \rangle$ is an inductively defined set.

Now that we have invested so much elbow grease into specifying the concept of an inductively defined set, it is natural to ask whether there are more examples. We will look at a particularly important one next.

Peano Axioms for Lists $\mathcal{L}[\mathbb{N}]$

▷ Lists of (unary) natural numbers: $[1, 2, 3]$, $[7, 7]$, $[]$, ...

- ▹ nil-rule: start with the empty list $[]$
- ▹ cons-rule: extend the list by adding a number $n \in \mathbb{N}_1$ at the front

▷ two constructors: $\mathrm{nil} \in \mathcal{L}[\mathbb{N}]$ and $\mathrm{cons}\colon \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$

▷ **Example 4.2.8** e.g. $[3, 2, 1] \mathrel{\widehat{=}} \mathrm{cons}(3, \mathrm{cons}(2, \mathrm{cons}(1, \mathrm{nil})))$ and $[] \mathrel{\widehat{=}} \mathrm{nil}$

▷ **Definition 4.2.9** We will call the following set of axioms are called the Peano Axioms for $\mathcal{L}[\mathbb{N}]$ in analogy to the Peano Axioms in Definition 3.1.5.

▷ **Axiom 4.2.10 (LP1)** $\mathrm{nil} \in \mathcal{L}[\mathbb{N}]$ (generation axiom (nil))

▷ **Axiom 4.2.11 (LP2)** $\mathrm{cons}\colon \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$ (generation axiom (cons))

▷ **Axiom 4.2.12 (LP3)** nil is not a cons-value

▷ **Axiom 4.2.13 (LP4)** cons is injective

▷ **Axiom 4.2.14 (LP5)** If the nil possesses property P and(Induction Axiom)

- ▹ for any list l with property P, and for any $n \in \mathbb{N}_1$, the list $\mathrm{cons}(n, l)$ has property P

then every list $l \in \mathcal{L}[\mathbb{N}]$ has property P.

Note: There are actually 10 (Peano) axioms for lists of unary natural numbers: the original five for $\mathbb{N}_1$ — they govern the constructors o and s, and the ones we have given for the constructors nil and cons here.

Note furthermore: that the $\mathbf{P}i$ and the $\mathbf{LP}i$ are very similar in structure: they say the same things about the constructors.

The first two axioms say that the set in question is generated by applications of the constructors: Any expression made of the constructors represents a member of $\mathbb{N}_1$ and $\mathcal{L}[\mathbb{N}]$ respectively.

The next two axioms eliminate any way any such members can be equal. Intuitively they can only be equal, if they are represented by the same expression. Note that we do not need any axioms for the relation between $\mathbb{N}_1$ and $\mathcal{L}[\mathbb{N}]$ constructors, since they are different as members of different sets.

Finally, the induction axioms give an upper bound on the size of the generated set. Intuitively the axiom says that any object that is not represented by a constructor expression is not a member of $\mathbb{N}_1$ and $\mathcal{L}[\mathbb{N}]$.

A direct consequence of this observation is that

Corollary 4.2.15 *The set $\langle \mathbb{N}_1 \cup \mathcal{L}[\mathbb{N}], \{o, s, \mathrm{nil}, \mathrm{cons}\}\rangle$ is an inductively defined set in the sense of Definition 4.2.6.*

Operations on Lists: Append

▷ The append function $@ : \mathcal{L}[\mathbb{N}] \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$ concatenates lists
Defining equations: $\mathrm{nil} @ l = l$ and $\mathrm{cons}(n, l) @ r = \mathrm{cons}(n, l @ r)$

▷ **Example 4.2.16** $[3, 2, 1] @ [1, 2] = [3, 2, 1, 1, 2]$ and $[] @ [1, 2, 3] = [1, 2, 3] = [1, 2, 3] @ []$

▷ **Lemma 4.2.17** *For all $l, r \in \mathcal{L}[\mathbb{N}]$, there is exactly one $s \in \mathcal{L}[\mathbb{N}]$ with $s = l @ r$.*

▷ Proof: by induction on l. (what does this mean?)

P.1 we have two cases

P.1.1 base case: $l = \mathrm{nil}$: must have $s = r$.

P.1.2 step case: $l = \mathrm{cons}(n, k)$ for some list k:

P.1.2.1 Assume that here is a unique s' with $s' = k @ r$,

P.1.2.2 then $s = \mathrm{cons}(n, k) @ r = \mathrm{cons}(n, k @ r) = \mathrm{cons}(n, s')$. □

□

▷ **Corollary 4.2.18** *Append is a function* *(see, this just worked fine!)*

You should have noticed that this proof looks exactly like the one for addition. In fact, wherever we have used an axiom $\mathbf{P}i$ there, we have used an axiom $\mathbf{LP}i$ here. It seems that we can do anything we could for unary natural numbers for lists now, in particular, programming by recursive equations.

Operations on Lists: more examples

▷ **Definition 4.2.19** $\lambda(\mathrm{nil})=o$ and $\lambda(\mathrm{cons}(n,l))=s(\lambda(l))$

▷ **Definition 4.2.20** $\rho(\mathrm{nil})=\mathrm{nil}$ and $\rho(\mathrm{cons}(n,l))=\rho(l)\,@\,\mathrm{cons}(n,\mathrm{nil})$.

Now, we have seen that "inductively defined sets" are a basis for computation, we will turn to the programming language see them at work in concrete setting.

4.3 Inductively Defined Sets in SML

We are about to introduce one of the most powerful aspects of SML, its ability to let the user define types. After all, we have claimed that types in SML are first-class objects, so we have to have a means of constructing them.

We have seen above, that the main feature of an inductively defined set is that it has Peano Axioms that enable us to use it for computation. Note that specifying them, we only need to know the constructors (and their types). Therefore the **datatype** constructor in SML only needs to specify this information as well. Moreover, note that if we have a set of constructors of an inductively defined set — e.g. zero : mynat and suc : mynat −> mynat for the set mynat, then their codomain type is always the same: mynat. Therefore, we can condense the syntax even further by leaving that implicit.

Data Type Declarations

▷ **Definition 4.3.1** SML data type provide concrete syntax for inductively defined sets via the keyword **datatype** followed by a list of constructor declarations.

▷ **Example 4.3.2** We can declare a data type for unary natural numbers by

```
- datatype mynat = zero | suc of mynat;
datatype mynat = suc of mynat | zero
```

this gives us constructor functions zero : mynat and suc : mynat −> mynat.

▷ **Observation 4.3.3** *We can define functions by (complete) case analysis over the constructors*

▷ **Example 4.3.4 (Converting types)**

```
fun num (zero) = 0 | num (suc(n)) = num(n) + 1;
val num = fn : mynat -> int
```

▷ **Example 4.3.5 (Missing Constructor Cases)**

```
fun incomplete (zero) = 0;
stdIn:10.1-10.25 Warning: match non-exhaustive
          zero => ...
val incomplete = fn : mynat -> int
```

▷ **Example 4.3.6 (Inconsistency)**

```
fun ic (zero) = 1 | ic(suc(n))=2 | ic(zero)= 3;
stdIn:1.1-2.12 Error: match redundant
```

```
    zero => ...
    suc n => ...
    zero => ...
```

So, we can re-define a type of unary natural numbers in SML, which may seem like a somewhat pointless exercise, since we have integers already. Let us see what else we can do.

Data Types Example (Enumeration Type)

▷ a type for weekdays (nullary constructors)

```
    datatype day = mon | tue | wed | thu | fri | sat | sun;
```

▷ use as basis for rule-based procedure (first clause takes precedence)

```
- fun weekend sat = true
         | weekend sun = true
         | weekend _ = false
val weekend : day -> bool
```

▷ this give us

```
- weekend sun
true : bool
- map weekend [mon, wed, fri, sat, sun]
[false, false, false, true, true] : bool list
```

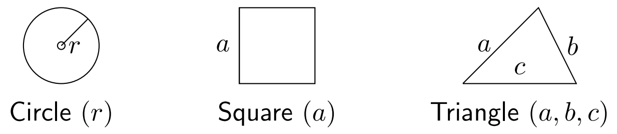
▷ nullary constructors describe values, enumeration types finite sets

Somewhat surprisingly, finite enumeration types that are separate constructs in most programming languages are a special case of **datatype** declarations in SML. They are modeled by sets of base constructors, without any functional ones, so the base cases form the finite possibilities in this type. Note that if we imagine the Peano Axioms for this set, then they become very simple; in particular, the induction axiom does not have step cases, and just specifies that the property P has to hold on all base cases to hold for all members of the type.

Let us now come to a real-world examples for data types in SML. Say we want to supply a library for talking about mathematical shapes (circles, squares, and triangles for starters), then we can represent them as a data type, where the constructors conform to the three basic shapes they are in. So a circle of radius r would be represented as the constructor term Circle r (what else).

Data Types Example (Geometric Shapes)

▷ describe three kinds of geometrical forms as mathematical objects



Mathematically: $\mathbb{R}^+ \uplus \mathbb{R}^+ \uplus (\mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+)$

⊳⊳ In SML: approximate $\mathbb{R}^+$ by the built-in type real.

```
datatype shape =
    Circle of real
  | Square of real
  | Triangle of real * real * real
```

▷ This gives us the constructor functions

```
Circle : real −> shape
Square : real −> shape
Triangle : real * real * real −> shape
```

Some experiments: We try out our new data type, and indeed we can construct objects with the new constructors.

```
− Circle 4.0
Circle 4.0 : shape
− Square 3.0
Square 3.0 : shape
− Triangle(4.0, 3.0, 5.0)
Triangle(4.0, 3.0, 5.0) : shape
```

The beauty of the representation in user-defined types is that this affords powerful abstractions that allow to structure data (and consequently program functionality).

Data Types Example (Areas of Shapes)

▷ a procedure that computes the area of a shape:

```
− fun area (Circle r) = Math.pi*r*r
    | area (Square a) = a*a
    | area (Triangle(a,b,c)) = let val s = (a+b+c)/2.0
                               in Math.sqrt(s*(s−a)*(s−b)*(s−c))
                               end
val area : shape −> real
```

New Construct: Standard structure Math (see [SML10])

▷ some experiments

```
− area (Square 3.0)
9.0 : real
− area (Triangle(6.0, 6.0, Math.sqrt 72.0))
18.0 : real
```

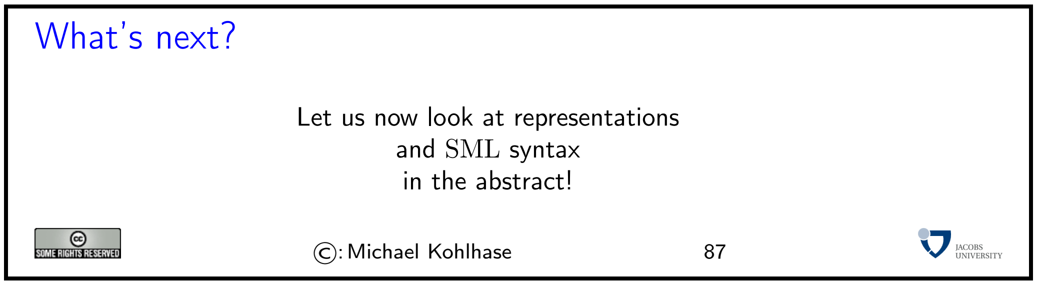
All three kinds of shapes are included in one abstract entity: the type shape, which makes programs like the area function conceptually simple — it is just a function from type shape to type real. The complexity — after all, we are employing three different formulae for computing the area of the respective shapes — is hidden in the function body, but is nicely compartmentalized, since the constructor cases in systematically correspond to the three kinds of shapes.

We see that the combination of user-definable types given by constructors, pattern matching, and function definition by (constructor) cases give a very powerful structuring mechanism for heterogeneous data objects. This makes is easy to structure programs by the inherent qualities of the data. A trait that other programming languages seek to achieve by object-oriented techniques.

Chapter 5

A Theory of SML: Abstract Data Types and Term Languages

We will now develop a theory of the expressions we write down in functional programming languages and the way they are used for computation.



What's next?

Let us now look at representations
and SML syntax
in the abstract!

In this Chapter, we will study computation in functional languages in the abstract by building mathematical models for them.

Warning: The Chapter on abstract data types we are about to start is probably the most daunting of the whole course (to first-year students), even though the concepts are very simple[1]. The reason for this seems to be that the models we create are so abstract, and that we are modeling language constructs (SML expressions and types) with mathematical objects (terms and sorts) that look very similar. The crucial step here for understanding is that sorts and terms are *similar to, but not equal* to SML types and expressions. The former are abstract mathematical objects, whereas the latter are concrete objects we write down for computing on a concrete machine. Indeed, the similarity in spirit is because we use sorts and terms to *model* SML types and expressions, i.e. to understand what the SML type checker and evaluators do and make predictions about their behavior.

The idea of building representations (abstract mathematical objects or expressions) that model other objects is a recurring theme in the GenCS course; here we get a first glimpse of it: we use sorts and terms to model SML types and expressions and programs.

Recall the diagram of things we want to model from Section 1.1 (see Figure 5.1). For representing data, we will use "ground constructor terms", for algorithms we will make the idea of "defining equations" precise by defining "abstract procedures", and for the machines, we will build an "abstract interpreter". In all of these notions, we will employ abstract/mathematical methods to model and understand the relevant concepts.

We will proceed as we often do in science and modeling: we build a very simple model, and "test-drive" it to see whether it covers the phenomena we want to understand. Following this lead

[1]. . . in retrospect: second-year students report that they cannot imagine how they found the material so difficult to understand once they look at it again from the perspective of having survived GenCS.

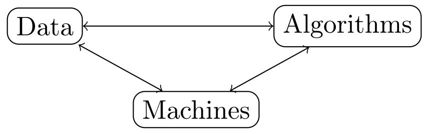


Figure 5.1: Data, Algorithms, and Machines

we will start out with a notion of "ground constructor terms" for the representation of data and with a simple notion of abstract procedures that allow computation by replacement of equals. We have chosen this first model intentionally naive, so that it fails to capture the essentials, so we get the chance to refine it to one based on "constructor terms with variables" and finally on "terms", refining the relevant concepts along the way.

This iterative approach intends to raise awareness that in CS theory it is not always the first model that eventually works, and at the same time intends to make the model easier to understand by repetition.

5.1 Abstract Data Types and Ground Constructor Terms

Abstract data types are abstract objects that specify inductively defined sets by declaring a set of constructors with their sorts (which we need to introduce first).

Abstract Data Types (ADT)

▷ **Definition 5.1.1** Let $\mathcal{S}^0 := \{\mathbb{A}_1, \ldots, \mathbb{A}_n\}$ be a finite set of symbols, then we call the set $\mathcal{S}$ the set of sorts over the set $\mathcal{S}^0$, if

- ▷ $\mathcal{S}^0 \subseteq \mathcal{S}$ (base sorts are sorts)
- ▷ If $\mathbb{A}, \mathbb{B} \in \mathcal{S}$, then $\mathbb{A} \times \mathbb{B} \in \mathcal{S}$ (product sorts are sorts)
- ▷ If $\mathbb{A} \in \mathcal{S}$ and $\mathbb{B} \in \mathcal{S}^0$, then $\mathbb{A} \to \mathbb{B} \in \mathcal{S}$ (function sorts are sorts)

▷ **Definition 5.1.2** If c is a symbol and $\mathbb{A} \in \mathcal{S}$, then we call a pair $[c \colon \mathbb{A}]$ a constructor declaration for c over $\mathcal{S}$.

▷ **Definition 5.1.3** Let $\mathcal{S}^0$ be a set of symbols and $\mathcal{D}$ a set of constructor declarations over $\mathcal{S}$, then we call the pair $\langle \mathcal{S}^0, \mathcal{D} \rangle$ an abstract data type

▷ **Example 5.1.4** $\mathcal{B} := \langle \{\mathbb{B}\}, \{[T \colon \mathbb{B}], [F \colon \mathbb{B}]\} \rangle$ is an abstract data for truth values.

▷ **Example 5.1.5** $\langle \{\mathbb{N}\}, \{[o \colon \mathbb{N}], [s \colon \mathbb{N} \to \mathbb{N}]\} \rangle$ represents unary natural numbers.

▷ **Example 5.1.6** $\mathfrak{L} := \langle \{\mathbb{N}, \mathcal{L}(\mathbb{N})\}, \{[o \colon \mathbb{N}], [s \colon \mathbb{N} \to \mathbb{N}], [\mathrm{nil} \colon \mathcal{L}(\mathbb{N})], [\mathrm{cons} \colon \mathbb{N} \times \mathcal{L}(\mathbb{N}) \to \mathcal{L}(\mathbb{N})]\} \rangle$
In particular, the term $\mathrm{cons}(s(o), \mathrm{cons}(o, \mathrm{nil}))$ represents the list $[1, 0]$

▷ **Example 5.1.7** $\langle \{\mathcal{S}^0, \mathcal{S}\}, \{[\iota \colon \mathcal{S}^0 \to \mathcal{S}], [\to \colon \mathcal{S} \times \mathcal{S}^0 \to \mathcal{S}], [\times \colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}]\} \rangle$
(what is this?)

The abstract data types in Example 5.1.4 and Example 5.1.5 are good old friends, the first models the build-in SML type bool and the second the unary natural numbers we started the course with.

In contrast to SML **datatype** declarations we allow more than one sort to be declared at one time (see Example 5.1.6). So abstract data types correspond to a group of **datatype** declarations.

The last example is more enigmatic. It shows how we can represent the set of sorts defined above as an abstract data type itself. Here, things become a bit mind-boggling, but it is useful to think this through and pay very close attention what the symbols mean.

First it is useful to note that the abstract data type declares the base sorts $\mathcal{S}^0$ and $\mathcal{S}$, which are just abstract mathematical objects that model the sets of base sorts and sorts from Definition 5.1.1 – without being the same, even though the symbols look the same. The declaration $[\iota\colon \mathcal{S}^0 \to \mathcal{S}]$ introduces an inclusion of the base sorts into the sorts: if a is a base sort, then $\iota(a)$ is a sort – this is a standard trick to represent the subset relation via a constructor.

Finally the two remaining declarations $[\to\colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}]$ and $[\times\colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}]$ introduce the sort constructors (see where the name comes from?) in Definition 5.1.1.

With Definition 5.1.3, we now have a mathematical object for (sequences of) data type declarations in SML. This is not very useful in itself, but serves as a basis for studying what expressions we can write down at any given moment in SML. We will cast this in the notion of constructor terms that we will develop in stages next.

Ground Constructor Terms

▷ **Definition 5.1.8** Let $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D} \rangle$ be an abstract data type, then we call a representation t a ground constructor term of sort $\mathbb{T}$, iff

- ▷ $\mathbb{T} \in \mathcal{S}^0$ and $[t\colon \mathbb{T}] \in \mathcal{D}$, or
- ▷ $\mathbb{T} = \mathbb{A} \times \mathbb{B}$ and t is of the form $\langle a, b \rangle$, where a and b are ground constructor terms of sorts $\mathbb{A}$ and $\mathbb{B}$, or
- ▷ t is of the form $c(a)$, where a is a ground constructor term of sort $\mathbb{A}$ and there is a constructor declaration $[c\colon \mathbb{A} \to \mathbb{T}] \in \mathcal{D}$.

We denote the set of all ground constructor terms of sort $\mathbb{A}$ with ${\mathcal{T}^g}_{\mathbb{A}}(\mathcal{A})$ and use $\mathcal{T}^g(\mathcal{A}) := \bigcup_{\mathbb{A} \in \mathcal{S}} {\mathcal{T}^g}_{\mathbb{A}}(\mathcal{A})$.

▷ **Example 5.1.9** $\mathrm{cons}(s(o), \mathrm{nil}) \in {\mathcal{T}^g}_{\mathcal{L}(\mathbb{N})}(\mathfrak{L})$ where $\mathfrak{L}$ is the ADT from Example 5.1.6.

▷ **Definition 5.1.10** If $t = c(t')$ then we say that the symbol c is the head of t (write $\mathrm{head}(t)$). If $t = a$, then $\mathrm{head}(t) = a$; $\mathrm{head}(\langle t_1, t_2 \rangle)$ is undefined.

▷ **Notation 5.1.11** We will write $c(a, b)$ instead of $c(\langle a, b \rangle)$ (cf. binary function)

The main purpose of ground constructor terms will be to represent data. In the data type from Example 5.1.5 the ground constructor term $s(s(o))$ can be used to represent the unary natural number 2. Similarly, in the abstract data type from Example 5.1.6, the term $\mathrm{cons}(uNatsucs(o), \mathrm{cons}(s(o), \mathrm{nil}))$ represents the list $[2, 1]$.

Note: that to be a good data representation format for a set S of objects, ground constructor terms need to

- cover S, i.e. that for every object $s \in S$ there should be a ground constructor term that represents s.
- be unambiguous, i.e. that we can decide equality by just looking at them, i.e. objects $s \in S$ and $t \in S$ are equal, iff their representations are.

But this is just what our Peano Axioms are for, so abstract data types come with specialized Peano axioms, which we can paraphrase as

Peano Axioms for Sorts in Abstract Data Types

▷ **Observation 5.1.12** *The set $\mathcal{T}^g(\mathcal{A})$ of ground constructor terms over an abstract data type $\mathcal{A}$ form an inductively defined set.*

▷ Proof Sketch: We just think of each of the clauses in the definition as constructors: "constants", "pairs", and "constructor applications". □

▷ **Example 5.1.13** We can even program it in SML with the datatype constructor:

```
datatype basesort = a | b | c | ...
datatype sort = base of basesort | pairsort of sort * sort| funsort of sort * sort
symbol = string
datatype gct = const of symbol | pair of gct * gct | app of symbol * gct
```

With this, the term $\text{cons}(s(o), \text{nil}) \in \mathcal{T}^g{}_{\mathcal{L}(\mathbb{N})}(\mathfrak{L})$ from Example 5.1.9 is represented as app("cons'',app("suc",const("zero")),const"nil").

▷ Idea: In abstract data types we have a kind of Peano Axioms as well.

▷ **Axiom 5.1.14** if t is a ground constructor term of sort $\mathbb{T}$, then $t \in \mathbb{T}$

▷ **Axiom 5.1.15** equality on ground constructor terms is trivial

▷ **Axiom 5.1.16** only ground constructor terms of sort $\mathbb{T}$ are in $\mathbb{T}$ (induction axioms)

Note that these Peano axioms are left implicit – we never write them down explicitly, but they are there if we want to reason about or compute with expressions in the abstract data types.

Now that we have established how to represent data, we will develop a theory of programs, which will consist of directed equations in this case. We will do this as theories often are developed; we start off with a very first theory will not meet the expectations, but the test will reveal how we have to extend the theory. We will iterate this procedure of theorizing, testing, and theory adapting as often as is needed to arrive at a successful theory.

But before we embark on this, we build up our intuition with an extended example

Towards Understanding Computation on ADTs

▷ Aim: We want to understand computation with data from ADTs

▷ Idea: Let's look at a concrete example: abstract data type $\mathcal{B} := \langle\{\mathbb{B}\}, \{[T : \mathbb{B}], [F : \mathbb{B}]\}\rangle$ and the operations we know from mathtalk: $\wedge$, $\vee$, $\neg$, for "and", "or", and "not".

▷ Idea: think of these operations as functions on $\mathbb{B}$ that can be defined by "defining equations" e.g. $\neg(T) = F$, which we represent as $\neg(T) \rightsquigarrow F$ to stress the direction of computation.

▷ **Example 5.1.17** We represent the operations by declaring sort and equations.

$\neg$: $\langle\neg{::}\mathbb{B}\to\mathbb{B}\,;\,\{\neg(T)\rightsquigarrow F,\neg(F)\rightsquigarrow T\}\rangle$,
$\wedge$: $\langle\wedge{::}\mathbb{B}\times\mathbb{B}\to\mathbb{B}\,;\,\{\wedge(T,T)\rightsquigarrow T,\wedge(T,F)\rightsquigarrow F,\wedge(F,T)\rightsquigarrow F,\wedge(F,F)\rightsquigarrow F\}\rangle$,
$\vee$: $\langle\vee{::}\mathbb{B}\times\mathbb{B}\to\mathbb{B}\,;\,\{\vee(T,T)\rightsquigarrow T,\vee(T,F)\rightsquigarrow T,\vee(F,T)\rightsquigarrow T,\vee(F,F)\rightsquigarrow F\}\rangle$.

Idea: Computation is just replacing equals by equals

$$\vee(T,\wedge(F,\neg(F)))\rightsquigarrow\vee(T,\wedge(F,T))\rightsquigarrow\vee(T,F)\rightsquigarrow T$$

⧐ Next Step: Define all the necessary notions, so that we can make this work mathematically.

5.2 A First Abstract Interpreter

Let us now come up with a first formulation of an abstract interpreter, which we will refine later when we understand the issues involved. Since we do not yet, the notions will be a bit vague for the moment, but we will see how they work on the examples.

But how do we compute?

- ▷ Problem: We can define functions, but how do we compute them?
- ▷ Intuition: We direct the equations (l2r) and use them as rules.
- ▷ **Definition 5.2.1** Let $\mathcal{A}$ be an abstract data type and $s,t\in\mathcal{T}^g{}_{\mathbb{T}}(\mathcal{A})$ ground constructor terms over $\mathcal{A}$, then we call a pair $s\rightsquigarrow t$ a rule for f, if $\mathrm{head}(s)=f$.
- ▷ **Example 5.2.2** turn $\lambda(\mathrm{nil})=o$ and $\lambda(\mathrm{cons}(n,l))=s(\lambda(l))$ to $\lambda(\mathrm{nil})\rightsquigarrow o$ and $\lambda(\mathrm{cons}(n,l))\rightsquigarrow s(\lambda(l))$
- ▷ **Definition 5.2.3** Let $\mathcal{A}:=\langle\mathcal{S}^0,\mathcal{D}\rangle$, then call a quadruple $\langle f{::}\mathbb{A}\to\mathbb{R}\,;\,\mathcal{R}\rangle$ an abstract procedure, iff $\mathcal{R}$ is a set of rules for f. $\mathbb{A}$ is called the argument sort and $\mathbb{R}$ is called the result sort of $\langle f{::}\mathbb{A}\to\mathbb{R}\,;\,\mathcal{R}\rangle$.
- ▷ **Definition 5.2.4** A computation of an abstract procedure p is a sequence of ground constructor terms $t_1\rightsquigarrow t_2\rightsquigarrow\ldots$ according to the rules of p.(whatever that means)
- ▷ **Definition 5.2.5** An abstract computation is a computation that we can perform in our heads. (no real world constraints like memory size, time limits)
- ▷ **Definition 5.2.6** An abstract interpreter is an imagined machine that performs (abstract) computations, given abstract procedures.

The central idea here is what we have seen above: we can define functions by equations. But of course when we want to use equations for programming, we will have to take some freedom of applying them, which was useful for proving properties of functions above. Therefore we restrict them to be applied in one direction only to make computation deterministic.

Let us now see how this works in an extended example; we use the abstract data type of lists from Example 5.1.6 (only that we abbreviate unary natural numbers).

Example: the functions ρ and @ on lists

▷ **Example 5.2.7** Consider the abstract procedures

$\langle\rho::\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N})\,;\,\{\rho(\mathrm{cons}(n,l))\rightsquigarrow @(\rho(l),\mathrm{cons}(n,\mathrm{nil})),\rho(\mathrm{nil})\rightsquigarrow\mathrm{nil}\}\rangle$
$\langle @::\mathcal{L}(\mathbb{N})\times\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N})\,;\,\{@(\mathrm{cons}(n,l),r)\rightsquigarrow\mathrm{cons}(n,@(l,r)),@(\mathrm{nil},l)\rightsquigarrow l\}\rangle$

▷ Then we have the following abstract computation

- ▷ $\rho(\mathrm{cons}(2,\mathrm{cons}(1,\mathrm{nil})))\rightsquigarrow @(\rho(\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))$ ($\rho(\mathrm{cons}(n,l))\rightsquigarrow @(\rho(l),\mathrm{cons}(n,\mathrm{nil}))$ with $n=2$ and $l=\mathrm{cons}(1,\mathrm{nil})$)
- ▷ $@(\rho(\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))\rightsquigarrow @(@(\rho(\mathrm{nil}),\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))$ ($\rho(\mathrm{cons}(n,l))\rightsquigarrow @(\rho(l),\mathrm{cons}(n,\mathrm{nil}))$ with $n=1$ and $l=\mathrm{nil}$)
- ▷ $@(@(\rho(\mathrm{nil}),\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))\rightsquigarrow @(@(\mathrm{nil},\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))$ ($\rho(\mathrm{nil})\rightsquigarrow\mathrm{nil}$)
- ▷ $@(@(\mathrm{nil},\mathrm{cons}(1,\mathrm{nil})),\mathrm{cons}(2,\mathrm{nil}))\rightsquigarrow @(\mathrm{cons}(1,\mathrm{nil}),\mathrm{cons}(2,\mathrm{nil}))$ ($@(\mathrm{nil},l)\rightsquigarrow l$ with $l=\mathrm{cons}(1,\mathrm{nil})$)
- ▷ $@(\mathrm{cons}(1,\mathrm{nil}),\mathrm{cons}(2,\mathrm{nil}))\rightsquigarrow\mathrm{cons}(1,@(\mathrm{nil},\mathrm{cons}(2,\mathrm{nil})))$ ($@(\mathrm{cons}(n,l),r)\rightsquigarrow\mathrm{cons}(n,@(l,r))$ with $n=1$, $l=\mathrm{nil}$, and $r=\mathrm{cons}(2,\mathrm{nil})$)
- ▷ $\mathrm{cons}(1,@(\mathrm{nil},\mathrm{cons}(2,\mathrm{nil})))\rightsquigarrow\mathrm{cons}(1,\mathrm{cons}(2,\mathrm{nil}))$ ($@(\mathrm{nil},l)\rightsquigarrow l$ with $l=\mathrm{cons}(2,\mathrm{nil})$)

Aha: ρ terminates on the argument $\mathrm{cons}(2,\mathrm{cons}(1,\mathrm{nil}))$

In the example above we try to understand the mechanics of "replacing equals by equals" with two concrete abstract procedures; one for ρ (think "reverse") and one for @ (think "append"). We start with an initial term $\rho(\mathrm{cons}(2,\mathrm{cons}(1,\mathrm{nil})))$ and realize that we can make this look equal to the left-hand side of the first rule of the abstract procedure ρ by making n be 2 and l be $\mathrm{cons}(1,\mathrm{nil})$. As the left hand side is equal to our expression, we can replace it with the right hand side of the same rule, again with $n=2$ and $l=\mathrm{cons}(1,\mathrm{nil})$. This gives us the first computation step.

The second computation step goes similarly: we work on the result of the first step and again find a left-hand-side of a rule that we can make equal. But this time we do not match the *whole* expression and replace it whit the right-hand-side, but only a part. We have indicated what we are working on by making the procedure symbol red.

The third computation step proceeds in style and eliminates the last occurrence of the procedure symbol ρ via the second rule. The last three steps proceed in kind and eliminate the occurrences of the procedure symbol @, so that we are left with a ground constructor term – that does not contain procedure symbols. This term is the last in the computation and thus constitutes its result.

Now let's get back to theory: let us see whether we can write down an abstract interpreter for this.

▷ An Abstract Interpreter (preliminary version)

▷ **Definition 5.2.8 (Idea)** Replace equals by equals! (this is licensed by the rules)

- Input: an abstract procedure $\langle f{::}\mathbb{A} \to \mathbb{R}\,;\,\mathcal{R}\rangle$ and an argument $a \in \mathcal{T}^g{}_{\mathbb{A}}(\mathcal{A})$.
- Output: a result $r \in \mathcal{T}^g{}_{\mathbb{R}}(\mathcal{A})$.
- Process:
  - find a part $t := f(t_1, \ldots t_n)$ in a,
  - find a rule $(l \rightsquigarrow r) \in \mathcal{R}$ and values for the variables in l that make t and l equal.
  - replace t with r' in a, where r' is obtained from r by replacing variables by values.
  - if that is possible call the result a' and repeat the process with a', otherwise stop.

▷ **Definition 5.2.9** We say that an abstract procedure $\langle f{::}\mathbb{A} \to \mathbb{R}\,;\,\mathcal{R}\rangle$ terminates (on $a \in \mathcal{T}^g{}_{\mathbb{A}}(\mathcal{A})$), iff the computation (starting with $f(a)$) reaches a state, where no rule applies.

▷ There are a lot of words here that we do not understand

▷ let us try to understand them better $\rightsquigarrow$ more theory!

Unfortunately we do not yet have the means to write down rules: they contain variables, which are not allowed in ground constructor rules. So we just extend the definition of the expressions we are allowed to write down.

Constructor Terms with Variables

▷ Wait a minute!: what are these rules in abstract procedures?

▷ Answer: pairs of constructor terms (really constructor terms?)

▷ Idea: variables stand for arbitrary constructor terms (let's make this formal)

▷ **Definition 5.2.10** Let $\langle \mathcal{S}^0, \mathcal{D}\rangle$ be an abstract data type. A (constructor term) variable is a pair of a symbol and a base sort.

▷ **Example 5.2.11** $x_{\mathbb{A}}$, $n_{\mathbb{N}}$, $x_{\mathbb{C}^3}$, . . . are variable .s

▷ **Definition 5.2.12** We denote the current set of variables of sort $\mathbb{A}$ with $\mathcal{V}_{\mathbb{A}}$, and use $\mathcal{V} := \bigcup_{\mathbb{A}\in\mathcal{S}^0} \mathcal{V}_{\mathbb{A}}$ for the set of all variables.

▷ Idea: add the following rule to the definition of constructor terms

- variables of sort $\mathbb{A} \in \mathcal{S}^0$ are constructor terms of sort $\mathbb{A}$.

▷ **Definition 5.2.13** If t is a constructor term, then we denote the set of variables occurring in t with $\mathbf{free}(t)$. If $\mathbf{free}(t) = \emptyset$, then we say t is ground or closed.

To have everything at hand, we put the whole definition onto one slide.

Constr. Terms with Variables: The Complete Definition

▷ **Definition 5.2.14** Let $\langle\mathcal{S}^0,\mathcal{D}\rangle$ be an abstract data type and $\mathcal{V}$ a set of variables, then we call a representation t a constructor term (with variables from $\mathcal{V}$) of sort $\mathbb{T}$, iff

- ▷ $\mathbb{T}\in\mathcal{S}^0$ and $[t\colon\mathbb{T}]\in\mathcal{D}$, or
- ▷ $t\in\mathcal{V}_{\mathbb{T}}$ is a variable of sort $\mathbb{T}\in\mathcal{S}^0$, or
- ▷ $\mathbb{T}=\mathbb{A}\times\mathbb{B}$ and t is of the form $\langle a,b\rangle$, where a and b are constructor terms with variables of sorts $\mathbb{A}$ and $\mathbb{B}$, or
- ▷ t is of the form $c(a)$, where a is a constructor term with variables of sort $\mathbb{A}$ and there is a constructor declaration $[c\colon\mathbb{A}\to\mathbb{T}]\in\mathcal{D}$.

We denote the set of all constructor terms of sort $\mathbb{A}$ with $\mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$ and use $\mathcal{T}(\mathcal{A};\mathcal{V}):=\bigcup_{\mathbb{A}\in\mathcal{S}}\mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$.

▷ **Definition 5.2.15** We define the depth of a constructor term $t\in\mathcal{T}(\mathcal{A};\mathcal{V})$ by the way it is constructed.

$$\mathrm{dp}(t):=\begin{cases}1 & \text{if } [t\colon\mathbb{T}]\in\Sigma \text{ or } t\in\mathcal{V}\\ \max(\mathrm{dp}(a),\mathrm{dp}(b))+1 & \text{if } t=\langle a,b\rangle\\ \mathrm{dp}(a)+1 & \text{if } t=f(a)\end{cases}$$

▷ **Observation 5.2.16** *This is a recursive function on the inductively defined set $\mathcal{T}(\mathcal{A};\mathcal{V})$.* *(made possible by Peano axioms)*

$\mathcal{T}(\mathcal{A};\mathcal{V})$ is an inductively defined set just as $\mathcal{T}^g(\mathcal{A})$ (see Observation 5.1.12 for details), but with four constructors now. So we can directly translate the function dp into SML using the types from Example 5.1.13:

```
fun dp const(s) = 1
  | dp var(s) = 1
  | dp pair (a,b) = max [dp a, dp b] + 1
  | dp app (s,a) = dp a + 1
```

The only difference to the mathematical recursion is that we have compacted the first two cases into one in Definition 5.2.15.

Now that we have extended our model of terms with variables, we will need to understand how to use them in computation. The main intuition is that variables stand for arbitrary terms (of the right sort). This intuition is modeled by the action of instantiating variables with terms, which in turn is the operation of applying a "substitution" to a term.

5.3 Substitutions

Substitutions are very important objects for modeling the operational meaning of variables: applying a substitution to a term instantiates all the variables with terms in it. Since a substitution only acts on the variables, we simplify its representation, we can view it as a mapping from variables to terms that can be extended to a mapping from terms to terms. The natural way to define substitutions would be to make them partial functions from variables to terms, but the definition below generalizes better to later uses of substitutions, so we present the real thing.

Substitutions

▷ **Definition 5.3.1** Let $\mathcal{A}$ be an abstract data type and $\sigma \in \mathcal{V} \to \mathcal{T}(\mathcal{A};\mathcal{V})$, then we call σ a substitution on $\mathcal{A}$, iff $\mathbf{supp}(\sigma) := \{x_{\mathbb{A}} \in \mathcal{V}_{\mathbb{A}} \,|\, \sigma(x_{\mathbb{A}}) \neq x_{\mathbb{A}}\}$ is finite and $\sigma(x_{\mathbb{A}}) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$. $\mathbf{supp}(\sigma)$ is called the support of σ.

▷ **Definition 5.3.2** If $\mathbf{supp}(\sigma) = \emptyset$, then we call σ the empty substitution and write σ as ϵ.

▷ **Notation 5.3.3** We denote the substitution σ with $\mathbf{supp}(\sigma) = \{x^i_{\mathbb{A}_i} \,|\, 1 \leq i \leq n\}$ and $\sigma(x^i_{\mathbb{A}_i}) = t_i$ by $[t_1/x^1_{\mathbb{A}_1}], \ldots, [t_n/x^n_{\mathbb{A}_n}]$.

▷ **Definition 5.3.4 (Substitution Extension)** Let σ be a substitution, then we denote with $\sigma, [t/x_{\mathbb{A}}]$ the function $\{\langle y_{\mathbb{B}}, t\rangle \in \sigma \,|\, y_{\mathbb{B}} \neq x_{\mathbb{A}}\} \cup \{\langle x_{\mathbb{A}}, t\rangle\}$.
($\sigma, [t/x_{\mathbb{A}}]$ coincides with σ off $x_{\mathbb{A}}$, and gives the result t there.)

▷ Note: If σ is a substitution, then $\sigma, [t/x_{\mathbb{A}}]$ is also a substitution.

Note that substitutions are "well-sorted" since we assume that $\sigma(x_{\mathbb{A}}) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A};\mathcal{V})$ in Definition 5.3.1. In other words, a substitution may only assign terms of the sort, the variable prescribes.

The extension of a substitution is an important operation, which you will run into from time to time. The intuition is that the values right of the comma overwrite the pairs in the substitution on the left, which already has a value for $x_{\mathbb{A}}$, even though the representation of σ may not show it.

Note that the use of the comma notation for substitutions defined in Notation 5.3.3 is consistent with substitution extension. We can view a substitution $[a/x], [f(b)/y]$ as the extension of the empty substitution (the identity function on variables) by $[f(b)/y]$ and then by $[a/x]$. Note furthermore, that substitution extension is not commutative in general.

Note that since we have defined constructor terms inductively, we can write down substitution application as a recursive function over the inductively defined set.

Substitution Application

▷ **Definition 5.3.5 (Substitution Application)** Let $\mathcal{A}$ be an abstract data type, σ a substitution on $\mathcal{A}$, and $t \in \mathcal{T}(\mathcal{A};\mathcal{V})$, then then we denote the result of systematically replacing all variables $x_{\mathbb{A}}$ in t by $\sigma(x_{\mathbb{A}})$ by $\sigma(t)$. We call $\sigma(t)$ the application of σ to t.

▷ With this definition we extend a substitution σ from a function $\sigma\colon \mathcal{V} \to \mathcal{T}(\mathcal{A};\mathcal{V})$ to a function $\sigma\colon \mathcal{T}(\mathcal{A};\mathcal{V}) \to \mathcal{T}(\mathcal{A};\mathcal{V})$.

▷ **Definition 5.3.6** Let s and t be constructor terms, then we say that s matches t, iff there is a substitution σ, such that $\sigma(s) = t$. σ is called a matcher that instantiates s to t. We also say that t is an instance of s.

▷ **Example 5.3.7** $[a/x], [f(b)/y], [a/z]$ instantiates $g(x, y, h(z))$ to $g(a, f(b), h(a))$.
(sorts elided here)

▷ **Definition 5.3.8** We give the defining equations for substitution application on an abstract data type $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D}\rangle$:

- ▷ $\sigma(c) = c$ if $[c \colon \mathbb{T}] \in \mathcal{D}$.
- ▷ $\sigma(x_{\mathbb{A}}) = t$ if $[t/x_{\mathbb{A}}] \in \sigma$.
- ▷ $\sigma(\langle a, b\rangle) = \langle \sigma(a), \sigma(b)\rangle$.
- ▷ $\sigma(f(a)) = f(\sigma(a))$.

▷ this definition uses the inductive structure of the terms.

Note that even though a substitution in and of itself is a total function on variables, it can be *extended* to a function on constructor terms by Definition 5.3.8. But we do not have a notation for this function. In particular, we may not write $[t/s]$, unless s is a variable. And that would not make sense, since substitution application is completely determined by the behavior on variables. For instance if we have $\sigma(c) = t$ for a constant c, then the value t must be c itself by the definition below.

We will now prove two important results that will be needed to establish that the operations in the upcoming abstract interpreter are well-defined.

Substitution Application conserves Sorts

▷ **Theorem 5.3.9** *Let $\mathcal{A}$ be an abstract data type, $t \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$, and σ a substitution on $\mathcal{A}$, then $\sigma(t) \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$.*

▷ Proof: by induction on $\mathrm{dp}(t)$ using Definition 5.2.14 and Definition 5.3.5

P.1 By Definition 5.2.14 we have to consider four cases

P.1.1 $[t \colon \mathbb{T}] \in \mathcal{D}$: $\sigma(t) = t$ by Definition 5.3.5, so $\sigma(t) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ by construction.

P.1.2 $t \in \mathcal{V}_{\mathbb{T}}$: We have $\sigma(t) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ by Definition 5.3.1,

P.1.3 $t = \langle a, b\rangle$ and $\mathbb{T} = \mathbb{A} \times \mathbb{B}$, where $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ and $b \in \mathcal{T}_{\mathbb{B}}(\mathcal{A}; \mathcal{V})$: We have $\sigma(t) = \sigma(\langle a, b\rangle) = \langle \sigma(a), \sigma(b)\rangle$. By inductive hypothesis we have $\sigma(a) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ and $\sigma(b) \in \mathcal{T}_{\mathbb{B}}(\mathcal{A}; \mathcal{V})$ and therefore $\langle \sigma(a), \sigma(b)\rangle \in \mathcal{T}_{\mathbb{A} \times \mathbb{B}}(\mathcal{A}; \mathcal{V})$ which gives the assertion.

P.1.4 $t = c(a)$, where $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ and $[c \colon \mathbb{A} \to \mathbb{T}] \in \mathcal{D}$: We have $\sigma(t) = \sigma(c(a)) = c(\sigma(a))$. By IH we have $\sigma(a) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ therefore $(c(\sigma(a))) \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$. □

The importance of this is that if we replace the part of the term we are working by the instantiated right-hand-side of the rule in the abstract interpreter, then we do not create an ill-sorted expression. Or in other words, the replacement operation in the abstract interpreter is “type-safe”.

Uniqueness of Matchers

▷ **Theorem 5.3.10** *For any $s, t \in \mathcal{T}(\mathcal{A}; \mathcal{V})$, there is at most one σ with $\sigma(s) = t$.*

▷ Proof: We prove this by induction on s (using the Peano Axioms for $\mathcal{T}(\mathcal{A}; \mathcal{V})$)

P.1 We have four cases to consider

P.1.1 s is a constant ($[c\colon \mathbb{T}] \in \mathcal{D}$): If $t = c$, then $\sigma = \epsilon$, else no matcher exists. □

P.1.2 s is a variable $x_{\mathbb{T}}$: Here σ must be $[t/x_{\mathbb{T}}]$. □

P.1.3 s is a pair $\langle a, b\rangle$:

P.1.3.1 Then t must be of the form $\langle c, d\rangle$ for some terms c and d by Definition 5.3.8.

P.1.3.2 By inductive hypothesis, we have at most one matcher σ_a and σ_b with $\sigma_a(a) = c$ and $\sigma_b(b) = d$ respectively.

P.1.3.3 Now let $C := \mathbf{supp}(\sigma_a, \sigma_b)$. If $\sigma_a|_C = \sigma_b|_C$, then $\sigma := \sigma_a \cup \sigma_b$ instantiates $s = \langle a, b\rangle$ to $t = \langle c, d\rangle$, otherwise no matcher exists. □

P.1.4 s is an application $f(a)$ with $[f\colon \mathbb{A} \to \mathbb{T}] \in \mathcal{D}$ and $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$:

P.1.4.1 Then t must be of the form $f(b)$ for some term b by Definition 5.3.8.

P.1.4.2 By inductive hypothesis, we have at most one matcher ρ with $\rho(a) = b$.

P.1.4.3 σ must be equal to ρ if that exists, since $\sigma(s) = \sigma(f(a)) = f(\sigma(a)) = f(\rho(a)) = f(b) = t$. □

□

The relevance of this result is that in the abstract interpreter we have to "choose" a substitution that instantiates the left-hand-side of the chosen rule to the term part we have chosen earlier. Theorem 5.3.10 tells us that there is no choice at all, which is a good thing – we want to control the abstract interpreter, not give it choices.

Note: that we used two different inductions in the proofs of these two theorems:

For the proof of Theorem 5.3.9, we used an "induction over the depth of [a constructor term]", which is really an induction on natural numbers and correspondingly uses the induction axiom for natural numbers. Here we use the trick of making the property of all constructor terms we want to prove a property of natural numbers by involving the depth function: The property P is that "substitution application conserves sorts on *all constructor terms of depth* n". The four cases in the recursive definition of dp in Definition 5.2.15 give rise to four cases in the proof: two for the base case $n = 0$ and two for the step case $n > 0$.

For the proof of Theorem 5.3.10 we directly used the induction axiom for the inductively defined set $\mathcal{T}(\mathcal{A}; \mathcal{V})$. Here the induction is over the "constructors" of $\mathcal{T}(\mathcal{A}; \mathcal{V})$, which correspond to the four cases in Definition 5.2.14.

So even though the two proofs start with very different induction axioms, they end up with a very similar case analysis. Some authors like the directness of "structural induction" (induction over the structure of terms), while some prefer the mor "elementary" nature of natural numbers induction. We have used both here to expose the two methods.

Now that we understand variable instantiation, we can see what it gives us for the meaning of rules: we get all the ground constructor terms a constructor term with variables stands for by applying all possible substitutions to it. Thus rules represent ground constructor subterm replacement actions in a computations, where we are allowed to replace all ground instances of the left hand side of the rule by the corresponding ground instance of the right hand side.

5.4 Terms in Abstract Data Types

Unfortunately, constructor terms are still not enough to write down rules, as rules also contain the symbols from the abstract procedures. So we have to extend our notion of expressions yet again.

Are Constructor Terms Really Enough for Rules?

- ▷ **Example 5.4.1** $\rho(\mathrm{cons}(n,l)) \rightsquigarrow @(\rho(l),\mathrm{cons}(n,\mathrm{nil}))$. ($\rho$ is not a constructor)
- ▷ Idea: need to include symbols for the defined procedures. (provide declarations)
- ▷ **Definition 5.4.2** Let $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D} \rangle$ be an abstract data type with $\mathbb{A} \in \mathcal{S}$ and let $f \notin \mathcal{D}$ be a symbol, then we call a pair $[f\colon \mathbb{A}]$ a procedure declaration for f over $\mathcal{S}$.
- ▷ **Definition 5.4.3** We call a finite set Σ of procedure declarations for distinct symbols a signature over $\mathcal{A}$.
- ▷ Idea: add the following rules to the definition of constructor terms
  - ▷ $\mathbb{T} \in \mathcal{S}^0$ and $[p\colon \mathbb{T}] \in \Sigma$, or
  - ▷ t is of the form $f(a)$, where a is a term of sort $\mathbb{A}$ and there is a procedure declaration $[f\colon \mathbb{A} \to \mathbb{T}] \in \Sigma$.

Again, we combine all of the rules for the inductive construction of the set of terms in one slide for convenience.

Terms: The Complete Definition

- ▷ Idea: treat procedures (from Σ) and constructors (from $\mathcal{D}$) at the same time.
- ▷ **Definition 5.4.4** Let $\langle \mathcal{S}^0, \mathcal{D} \rangle$ be an abstract data type, and Σ a signature over $\mathcal{A}$, then we call a representation t a term of sort $\mathbb{T}$ (over $\mathcal{A}$, Σ, and $\mathcal{V}$), iff
  - ▷ $\mathbb{T} \in \mathcal{S}^0$ and $[t\colon \mathbb{T}] \in \mathcal{D}$ or $[t\colon \mathbb{T}] \in \Sigma$, or
  - ▷ $t \in \mathcal{V}_{\mathbb{T}}$ and $\mathbb{T} \in \mathcal{S}^0$, or
  - ▷ $\mathbb{T} = \mathbb{A} \times \mathbb{B}$ and t is of the form $\langle a, b \rangle$, where a and b are terms of sorts $\mathbb{A}$ and $\mathbb{B}$, or
  - ▷ t is of the form $c(a)$, where a is a term of sort $\mathbb{A}$ and there is a constructor declaration $[c\colon \mathbb{A} \to \mathbb{T}] \in \mathcal{D}$ or a procedure declaration $[c\colon \mathbb{A} \to \mathbb{T}] \in \Sigma$.

  We denote the set of terms of sort $\mathbb{A}$ over $\mathcal{A}$, Σ, and $\mathcal{V}$ with $\mathcal{T}_{\mathbb{A}}(\mathcal{A},\Sigma;\mathcal{V})$ and the set of all terms with $\mathcal{T}(\mathcal{A},\Sigma;\mathcal{V})$.

Now that we have defined the concept of terms, we can ask ourselves which parts of terms are terms themselves. This results in the subterm relation, which is surprisingly intricate to define: we need an intermediate relation, the immediate subterm relation. The subterm relation is the transitive-reflexive closure of that.

Subterms

- ▷ Idea: Well-formed parts of constructor terms are constructor terms again (maybe of a different sort)
- ▷ **Definition 5.4.5** Let $\mathcal{A}$ be an abstract data type and s, t, and b be terms over $\mathcal{A}$, then we say that s is an immediate subterm of t, iff $t = f(s)$ or $t = \langle s, b\rangle$ or $t = \langle b, s\rangle$.
- ▷ **Definition 5.4.6** We say that a s is a subterm of t, iff $s = t$ or there is an immediate subterm t' of t, such that s is a subterm of t'.
- ▷ Note: that we see a recursive definition of a realtion in Maths here – recursion is not restricted to computation or computer science.
- ▷ **Example 5.4.7** $f(a)$ is a subterm of the terms $f(a)$ and $h(g(f(a), f(b)))$, and an immediate subterm of $h(f(a))$.

If we look at the definition of immediate subterms in Definition 5.4.5, then we can interpret the construction as reading the step cases in the rules for term construction (see Definition 5.4.4) backwards: What does is take to construct a term? The subterm relation corresponds to going back over the rules an arbitrary number of times (including zero times).

5.5 A Second Abstract Interpreter

Now that we have extended the notion of terms we can rethink the definition of abstract procedures and define an abstract notion of programs (coherent collections of abstract procedures).

For the final definition of abstract procedures we have to solve a tricky problem, which we have avoided in the treatment of terms above: To allow recursive procedures, we have to make sure that the (function) symbol that is introduced in the abstract procedure can already be used in the procedure body (the right hand side of the rules). So we have to be very careful with the signatures we use.

Abstract Procedures, Final Version

- ▷ **Definition 5.5.1 (Rules, final version)** Let $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D}\rangle$ be an abstract data type, Σ a signature over $\mathcal{A}$, and $f \notin (\mathbf{dom}(\mathcal{D}) \cup \mathbf{dom}(\Sigma))$ a symbol, then we call $f(s) \rightsquigarrow r$ a rule for $[f\colon \mathbb{A} \to \mathbb{B}]$ over Σ, if $s \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$ has no duplicate variables, constructors, or defined functions and $r \in \mathcal{T}_{\mathbb{B}}(\mathcal{A}, \Sigma, [f\colon \mathbb{A} \to \mathbb{B}]; \mathcal{V})$.
- ▷ Note: Rules are *well-sorted*, i.e. both sides have the same sort and *recursive*, i.e. rule heads may occur on the right hand side.
- ▷ **Definition 5.5.2 (Abstract Procedures, final version)**

  We call a quadruple $\mathcal{P} := \langle f{::}\mathbb{A} \to \mathbb{R}\,;\, \mathcal{R}\rangle$ an abstract procedure over Σ, iff $\mathcal{R}$ is a set of rules for $[f\colon \mathbb{A} \to \mathbb{R}] \in \Sigma$. We say that $\mathcal{P}$ induces the procedure declaration $[f\colon \mathbb{A} \to \mathbb{R}]$.
- ▷ **Example 5.5.3** Let $\mathcal{A}$ be the union of the abstract data types from Exam-

ple 5.1.6 and Example 5.1.17, then

$$\langle \mu{::}\mathbb{N}\times\mathcal{L}(\mathbb{N})\to\mathbb{B}\,;\,\{\mu(\langle x_{\mathbb{N}},\mathrm{nil}\rangle)\leadsto F,\mu(\langle x_{\mathbb{N}},\mathrm{cons}(h_{\mathbb{N}},t_{\mathcal{L}(\mathbb{N})})\rangle)\leadsto\vee(x=h,\mu(\langle y,t\rangle))\}\rangle$$

is an abstract procedure that induces the procedure declaration $[\mu\colon\mathbb{N}\times\mathcal{L}(\mathbb{N})\to\mathbb{B}]$

Note that we strengthened the restrictions on what we allow as rules in Definition 5.5.2, so that matching of rule heads becomes unique (remember that we want to take the choice out of computation in the interpreter).

But there is another namespacing problem we have to solve. The intuition here is that each abstract procedure introduces a new procedure declaration, which can be used in subsequent abstract procedures. We formalize this notion with the concept of an abstract program, i.e. a *sequence* of abstract procedures over the underlying abstract data type that behave well with respect to the induced signatures.

Abstract Programs

▷ **Definition 5.5.4 (Abstract Programs)** Let $\mathcal{A} := \langle\mathcal{S}^0,\mathcal{D}\rangle$ be an abstract data type, and $\mathcal{P} := \mathcal{P}_1,\ldots,\mathcal{P}_n$ a sequence of abstract procedures, then we call $\mathcal{P}$ an abstract program with signature Σ over $\mathcal{A}$, if the $\mathcal{P}_i$ induce (the procedure declarations) in Σ and

- ▷ $n=0$ and $\Sigma=\emptyset$ or
- ▷ $\mathcal{P}=\mathcal{P}',\mathcal{P}_n$ and $\Sigma=\Sigma',[f\colon\mathbb{A}]$, where
  - ▷ $\mathcal{P}'$ is an abstract program over Σ'
  - ▷ and $\mathcal{P}_n$ is an abstract procedure over Σ' that induces the procedure declaration $[f\colon\mathbb{A}]$.

▷ **Example 5.5.5** The two abstract procedures from Example 5.2.7

$$\langle @{::}\mathcal{L}(\mathbb{N})\times\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N})\,;\,\{@(\mathrm{cons}(n,l),r)\leadsto\mathrm{cons}(n,@(l,r)),@(\mathrm{nil},l)\leadsto l\}\rangle$$
$$\langle \rho{::}\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N})\,;\,\{\rho(\mathrm{cons}(n,l))\leadsto @(\rho(l),\mathrm{cons}(n,\mathrm{nil})),\rho(\mathrm{nil})\leadsto\mathrm{nil}\}\rangle$$

constitute an abstract program over the abstract data type from Example 5.1.6:

$$\langle\{\mathbb{N},\mathcal{L}(\mathbb{N})\},\{[o\colon\mathbb{N}],[s\colon\mathbb{N}\to\mathbb{N}],[\mathrm{nil}\colon\mathcal{L}(\mathbb{N})],[\mathrm{cons}\colon\mathbb{N}\times\mathcal{L}(\mathbb{N})\to\mathcal{L}(\mathbb{N})]\}\rangle$$

Now, we have all the prerequisites for the full definition of an abstract interpreter.

An Abstract Interpreter (second version)

▷ **Definition 5.5.6 (Abstract Interpreter (second try))** Let $a_0 := a$ repeat the following as long as possible:

- ▷ choose $(l\leadsto r)\in\mathcal{R}$, a subterm s of a_i and matcher σ, such that $\sigma(l)=s$.
- ▷ let a_{i+1} be the result of replacing s in a with $\sigma(r)$.

▷ **Definition 5.5.7** We say that an abstract procedure $\mathcal{P} := \langle f{::}\mathbb{A} \to \mathbb{R}\,;\,\mathcal{R}\rangle$ terminates (on $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$), iff the computation (starting with a) reaches a state, where no rule applies. Then a_n is the result of $\mathcal{P}$ on a

Question: Do abstract procedures always terminate?

⊳▷ Question: Is the result a_n always a constructor term?

Note: that we have taken care in the definition of the concept of abstract procedures in Definition 5.5.1 that computation (replacement of equals by equals by rule application) is well-sorted. Moreover, the fact that we always apply instances of rules yields the analogy that rules are "functions" whose input/output pairs are the instances of the rules.

5.6 Evaluation Order and Termination

To answer the questions remaining from the Definition 5.5.6 we will first have to think some more about the choice in this abstract interpreter: a fact we will use, but not prove here is we can make matchers unique once a subterm is chosen. Therefore the choice of subterm is all that we need to worry about. And indeed the choice of subterm does matter as we will see.

Evaluation Order in SML

▷ Remember in the definition of our abstract interpreter:

- ▷ choose a subterm s of a_i, a rule $(l \rightsquigarrow r) \in \mathcal{R}$, and a matcher σ, such that $\sigma(l) = s$.
- ▷ let a_{i+1} be the result of replacing s in a with $\sigma(r)$.

Once we have chosen s, the choice of rule and matcher become unique.

▷ ▷ the rule is unique, if the left-hand sides are non-overlapping (SML enforces this)

▷ the matcher is unique by Theorem 5.3.10

▷ **Observation 5.6.1** *sometimes there we can choose more than one subterm s and rule, and the computation and even the result differ.*

▷ **Example 5.6.2**

```
fun problem n = problem(n)+2;
datatype mybool = true | false;
fun myif(true,a,_) = a | myif(false,_,b) = b;
myif(true,3,problem(1));
```

Idea: Prescribe the "choice" of subterm

⊳▷ SML is a call-by-value language (values of arguments are computed first)

As we have seen in the example, we have to make up a policy for choosing subterms in evaluation to fully specify the behavior of our abstract interpreter. We will make the choice that corresponds to the one made in SML, since it was our initial goal to model this language.

An abstract call-by-value Interpreter (final)

▷ **Definition 5.6.3 (Call-by-Value Interpreter)** Given an abstract program $\mathcal{P}$ and a ground constructor term a, an abstract call-by-value interpreter creates a computation $a_1 \rightsquigarrow a_2 \rightsquigarrow \ldots$ with $a = a_1$ by the following process:

- ▷ Let s be the leftmost (of the) minimal subterms s of a_i, such that there is a rule $l \rightsquigarrow r \in \mathcal{R}$ and a substitution σ, such that $\sigma(l) = s$.
- ▷ let a_{i+1} be the result of replacing s in a with $\sigma(r)$.

Note: By this paragraph, this is a deterministic process, which can be implemented, once we understand matching fully (not covered in GenCS)

The name “call-by-value” comes from the fact that data representations as ground constructor terms are sometimes also called “values” and the act of computing a result for an (abstract) procedure applied to a bunch of argument is sometimes referred to as “calling an (abstract) procedure”. So we can understand the “call-by-value” policy as restricting computation to the case where all of the arguments are already values (i.e. fully computed to ground terms).

Other programming languages chose another evaluation policy called “call-by-reference”, which can be characterized by always choosing the outermost subterm that matches a rule. The most notable one is the Haskell language [Hut07, OSG08]. These programming languages are sometimes “lazy languages”, since they delay computing the values of their arguments. They are uniquely suited for dealing with objects that are potentially infinite in some form. “Infinite” arguments are not a problem, since “lazy” languages only compute enough of them so that the overall computation can proceed. In our example above, we can see the function problem as something that computes positive infinity. A lazy programming language would not be bothered by this and return the value 3.

▷ **Example 5.6.4** A lazy language language can even quite comfortably compute with possibly infinite objects, lazily driving the computation forward as far as needed. Consider for instance the following program:

```
myif(problem(1) > 999,"yes","no");
```

In a “call-by-reference” policy we would try to compute the outermost subterm (the whole expression in this case) by matching the myif rules. But they only match if there is a true or false as the first argument, which is not the case. The same is true with the rules for >, which we assume to deal lazily with arithmetical simplification, so that it can find out that $x + 1000 > 999$. So the outermost subterm that matches is problem(1), which we can evaluate 500 times to obtain true. Then and only then, the outermost subterm that matches a rule becomes the myif subterm and we can evaluate the whole expression to true.

Even though the problem of termination is difficult in full generality, we can make some progress on it. The main idea is to concentrate on the recursive calls in abstract procedures, i.e. the arguments of the defined function in the right hand side of rules. We will see that the recursion relation tells us a lot about the abstract procedure.

Analyzing Termination of Abstract Procedures

▷ **Example 5.6.5** $\tau\colon \mathbb{N}_1 \to \mathbb{N}_1$, where $\tau(n) \rightsquigarrow \tau(3n+1)$ for n odd and $\tau(n) \rightsquigarrow \tau(n/2)$ for n even. (does this procedure terminate?)

▷ **Definition 5.6.6** Let $\langle f{::}\mathbb{A} \to \mathbb{R}\,;\, \mathcal{R}\rangle$ be an abstract procedure, then we call a pair $\langle a, b\rangle$ a recursion step, iff there is a rule $f(x) \rightsquigarrow y$, and a substitution ρ, such that $\rho(x) = a$ and $\rho(y)$ contains a subterm $f(b)$.

▷ **Example 5.6.7** $\langle 4, 3\rangle$ is a recursion step for the abstract procedure

$$\langle \sigma{::}\mathbb{N}_1 \to \mathbb{N}_1\,;\, \{\sigma(o) \rightsquigarrow o, \sigma(s(n)) \rightsquigarrow n + \sigma(n)\}\rangle$$

▷ **Definition 5.6.8** We call an abstract procedure $\mathcal{P}$ recursive, iff it has a recursion step. We call the set of recursion steps of $\mathcal{P}$ the recursion relation of $\mathcal{P}$.

▷ Idea: analyze the recursion relation for termination.

Note that the procedure sketched in Example 5.6.5 is not really an abstract procedure since it has the even/odd condition, we cannot express in our setup. But even so, it shows us that termination is difficult.

Let us now turn to the question of termination of abstract procedures in general. Termination is a very difficult problem as Example 5.6.5 shows. In fact all cases that have been tried $\tau(n)$ diverges into the sequence $4, 2, 1, 4, 2, 1, \ldots$, and even though there is a huge literature in mathematics about this problem – the Collatz Conjecture, a proof that τ diverges on all arguments is still missing.

Another clue to the difficulty of the termination problem is (as we will see) that there cannot be a a program that reliably tells of any program whether it will terminate.

Now, we will define termination for arbitrary relations and present a theorem (which we do not really have the means to prove in GenCS) that tells us that we can reason about termination of abstract procedures — complex mathematical objects at best — by reasoning about the termination of their recursion relations — simple mathematical objects.

Termination

▷ **Definition 5.6.9** Let $R \subseteq \mathbb{A}^2$ be a binary relation, an infinite chain in R is a sequence $a_1, a_2, \ldots$ in $\mathbb{A}$, such that $\langle a_n, a_{n+1}\rangle \in R$ for all $n \in \mathbb{N}$.

▷ **Definition 5.6.10** We say that R terminates (on $a \in \mathbb{A}$), iff there is no infinite chain in R (that begins with a).

▷ **Definition 5.6.11** $\mathcal{P}$ diverges (on $a \in \mathbb{A}$), iff it does not terminate on a.

▷ **Theorem 5.6.12** *Let $\mathcal{P} = \langle f{::}\mathbb{A} \to \mathbb{R}\,;\, \mathcal{R}\rangle$ be an abstract procedure and $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$, then $\mathcal{P}$ terminates on a, iff the recursion relation of $\mathcal{P}$ does.*

▷ **Definition 5.6.13** Let $\mathcal{P} = \langle f{::}\mathbb{A} \to \mathbb{R}\,;\, \mathcal{R}\rangle$ be an abstract procedure, then we call the function $\{\langle a, b\rangle \mid a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V}) \text{ and } \mathcal{P} \text{ terminates for } a \text{ with } b\}$ in $\mathbb{A} \rightharpoonup \mathbb{B}$ the result function of $\mathcal{P}$.

▷ **Theorem 5.6.14** *Let $\mathcal{P} = \langle f{::}\mathbb{A} \to \mathbb{B}\,;\, \mathcal{D}\rangle$ be a terminating abstract procedure, then its result function satisfies the equations in $\mathcal{D}$.*

We should read Theorem 5.6.14 as the final clue that abstract procedures really do encode functions (under reasonable conditions like termination). This legitimizes the whole theory we have developed in this section.

We conclude the Section with a reflection the role of abstract data, procedures, and machines vs. concrete ones (see Figure 5.1); the result of this is that the abstract model is actually simpler than the concrete ones, since it abstracts away from many nasty real-world restrictions.

Abstract vs. Concrete Procedures vs. Functions

- ▷ An abstract procedure $\mathcal{P}$ can be realized as concrete procedure $\mathcal{P}'$ in a programming language
- ▷ Correctness assumptions (this is the best we can hope for)
  - ▹ If the $\mathcal{P}'$ terminates on a, then the $\mathcal{P}$ terminates and yields the same result on a.
  - ▹ If the $\mathcal{P}$ diverges, then the $\mathcal{P}'$ diverges or is aborted (e.g. memory exhaustion or buffer overflow)
- ▷ Procedures are not mathematical functions (differing identity conditions)
  - ▹ compare $\sigma\colon \mathbb{N}_1 \to \mathbb{N}_1$ with $\sigma(o) \rightsquigarrow o$, $\sigma(s(n)) \rightsquigarrow n + \sigma(n)$
    with $\sigma'\colon \mathbb{N}_1 \to \mathbb{N}_1$ with $\sigma'(o) \rightsquigarrow 0$, $\sigma'(s(n)) \rightsquigarrow ns(n) \,/\, 2$
  - ▹ these have the same result function, but σ is recursive while σ' is not!
  - ▹ Two functions are equal, iff they are equal as sets, iff they give the same results on all arguments

Chapter 6

More SML

6.1 Recursion in the Real World

We will now look at some concrete SML functions in more detail. The problem we will consider is that of computing the n^{th} Fibonacci number. In the famous Fibonacci sequence, the n^{th} element is obtained by adding the two immediately preceding ones.

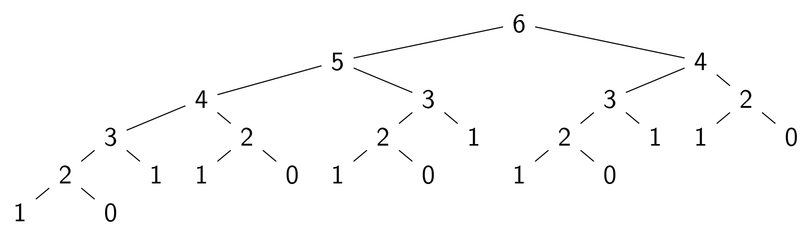
This makes the function extremely simple and straightforward to write down in SML. If we look at the recursion relation of this procedure, then we see that it can be visualized a tree, as each natural number has two successors (as the the function fib has two recursive calls in the step case).

Consider the Fibonacci numbers

- ▷ Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$
  - ▷ generally: $F_{n+1} := F_n + F_{n-1}$ plus start conditions
- ▷ easy to program in SML:

```
fun fib (0) = 0
  | fib (1) = 1
  | fib (n:int) = fib (n−1) + fib(n−2);
```

- ▷ Let us look at the recursion relation: $\{(n, n-1), (n, n-2) \mid n \in \mathbb{N}\}$ (it is a tree!)



Another thing we see by looking at the recursion relation is that the value fib(k) is computed $n - k + 1$ times while computing fib(k). All in all the number of recursive calls will be exponential in n, in other words, we can only compute a very limited initial portion of the Fibonacci sequence (the first 41 numbers) before we run out of time.

The main problem in this is that we need to know the last *two* Fibonacci numbers to compute the next one. Since we cannot "remember" any values in functional programming we take advantage of the fact that functions can ret'urn pairs of numbers as values: We define an auxiliary function fob (for lack of a better name) does all the work (recursively), and define the function fib(n) as the first element of the pair fob(n).

The function fob(n) itself is a simple recursive procedure with one! recursive call that returns the last two values. Therefore, we use a **let** expression, where we place the recursive call in the declaration part, so that we can bind the local variables a and b to the last two Fibonacci numbers. That makes the return value very simple, it is the pair (b,a+b).

A better Fibonacci Function

- ▷ Idea: Do not re-compute the values again and again!
  - ▷ keep them around so that we can re-use them. (e.g. let fib compute the two last two numbers)

```
fun fob 0 = (0,1)
  | fob 1 = (1,1)
  | fob (n:int) =
      let
         val (a:int, b:int) = fob(n−1)
      in
          (b,a+b)
      end;
fun fib (n) = let val (b:int,_) = fob(n) in b end;
```

- ▷ Works in linear time! (unfortunately, we cannot see it, because SML Int are too small)

If we run this function, we see that it is indeed much faster than the last implementation. Unfortunately, we can still only compute the first 44 Fibonacci numbers, as they grow too fast, and we reach the maximal integer in SML.

Fortunately, we are not stuck with the built-in integers in SML; we can make use of more sophisticated implementations of integers. In this particular example, we will use the module IntInf (infinite precision integers) from the SML standard library (a library of modules that comes with the SML distributions). The IntInf module provides a type IntINF.int and a set of infinite precision integer functions.

A better, larger Fibonacci Function

- ▷ Idea: Use a type with more Integers (Fortunately, there is IntInf)

```
val zero = IntInf.fromInt 0;
val one = IntInf.fromInt 1;

fun bigfob (0) = (zero,one)
  | bigfob (1) = (one,one)
  | bigfob (n:int) =
       let val (a, b) = bigfob(n−1)
```

```
          in (b,IntInf.+(a,b))
          end;

fun bigfib (n) = let val (a,_) = bigfob(n)
                     in IntInf.toString(a)
                     end;
```

We have seen that functions are just objects as any others in SML, only that they have functional type. If we add the ability to have more than one declaration at at time, we can combine function declarations for mutually recursive function definitions. In a mutually recursive definition we define n functions *at the same time*; as an effect we can use all of these functions in recursive calls. In our example below, we will define the predicates even and odd in a mutual recursion.

Mutual Recursion

▷ generally, we can make more than one declaration at one time, e.g.

```
– val pi = 3.14 and e = 2.71;
val pi = 3.14
val e = 2.71
```

▷ this is useful mainly for function declarations, consider for instance:

```
fun even (zero) = true
  | even (suc(n)) = odd (n)
and odd (zero) = false
  | odd(suc(n)) = even (n)
```

We trace the computation:

$$\text{even}(4) \rightsquigarrow \text{odd}(3) \rightsquigarrow \text{even}(2) \rightsquigarrow \text{odd}(1) \rightsquigarrow \text{even}(0) \rightsquigarrow \text{true}$$

This mutually recursive definition is somewhat like the children's riddle, where we define the "left hand" as that hand where the thumb is on the right side and the "right hand" as that where the thumb is on the right hand. This is also a perfectly good mutual recursion, only — in contrast to the even/odd example above — the base cases are missing.

6.2 Programming with Effects: Imperative Features in SML

So far, we have been programming in the "purely functional" fragment of SML. This will change now, indeed as we will see, purely functional programming languages are pointless, since they do not have any effects (such as altering the state of the screen or the disk). The advantages of functional languages like SML is that they limit effects (which pose difficulties for understanding the behavior of programs) to very special occasions

The hallmark of purely functional languages is that programs can be executed without changing the machine state, i.e. the values of variables. Indeed one of the first things we learnt was that variables are bound by declarations, and re-binding only shadows previous variable bindings. In the same way, the execution of functional programs is not affected by machine state, since the value of a function is fully determined by it arguments.

Note that while functions may be purely functional, the already SML interpreter cannot be, since it would be pointless otherwise: we *do want it to change the state of the machine*, if only to observe the computed values as changes of the (state of the) screen, similarly, we want it to be affected by the effects of typing with the keyboard.

But effects on the machine state can be useful in other situations, not just for input and output.

▷ Programming with Effects on the Machine State

- ▷ Until now, our procedures have been characterized entirely by their values on their arguments (as a mathematical function behaves)
- ▷ This is not enough, therefore SML also considers effects, e.g. for
  - ▷ *input/output*: the interesting bit about a print statement is the effect
  - ▷ mutation: allocation and modification of storage during evaluation
  - ▷ communication: data may be sent and received over channels
  - ▷ exceptions: abort evaluation by signaling an exceptional condition
- ▷ Idea: An effect is any action resulting from an evaluation that is not returning a value (formal definition difficult)
- ▷ Documentation: should always address arguments, values, and effects!

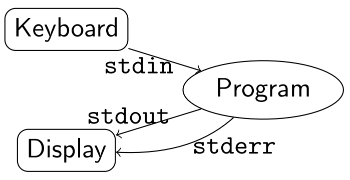
We will now look at the basics of input/output behavior and exception handling. They are state-changing and state-sensitive in completely different ways.

6.2.1 Input and Output

Like in many programming languages Input/Output are handled via "streams" in SML. Streams are special, system-level data structures that generalize files, data sources, and periphery systems like printers or network connections. You can think of them as possibly infinite strings. In SML, streams are supplied by module TextIO from the the SML basic library [SML10]. Note that as SML is typed, streams have types as well.

Input and Output in SML

- ▷ Input and Output is handled via "streams" (think of possibly infinite strings)
- ▷ there are two predefined streams TextIO.stdIn and TextIO.stdOut ($\widehat{=}$ keyboard input and screen)



▷ **Example 6.2.1 (Input)**
via TextIO.inputLine : TextIO.instream −> string

```
− TextIO.inputLine(TextIO.stdIn);
  sdflkjsdlfkj
val it = "sdflkjsdlfkj" : string
```

▷ **Example 6.2.2 (Printing to Standard Output)**
TextIO.print prints its argument to stdin (≙ screen)

```
print "sdfsfsdf''
```

The user can also create streams as files: TextIO.openIn and TextIO.openOut.

⊳ Streams should be closed when no longer needed: TextIO.closeIn and TextIO.closeOut.

The streams themselves are supplied by the underlying operating system, the TextIO library only wraps them for the program. Generally, any program (in this case our SML interpreter) has three streams: stdin (for input), stdout (for output), and stderr (for errors). The former is initially linked to the keyboard, the latter two initially to the display; but they can be remapped e.g. to files on the operating system level. This operating-level remapping should not be confused with the opening additional streams from the SML level.

But for input we have to deal with another aspect: Even if streams are potentially infinite, they can be finite, i.e. they may end. Therefore an input from a stream may very well fail. For that the SML library provides a typical and idiomatic construction: the option type constructor. It has two cases one is a nullary constructor NONE and a unary constructor SOME that wraps the result, it it exists. This is the target type of the input1 function which may or may not return a character, but always returns a char option.

Input and Output in SML

▷ Problem: How to handle the end of input files?

```
TextIO.input1 : instream −> char option
```

attempts to read one char from an input stream (may fail)

▷ The SML basis library supplies the datatype

```
datatype 'a option = NONE | SOME of 'a
```

which can be used in such cases together with lots of useful functions.

Now that we have all the parts in hand, we will combine them to a simple in/out function: copyTextFile, which copies a file character by character. Of course, we can also copy files at the operating system level, but with the function below, we can also treat some characters or character sequences specially, e.g. dropping all the xes.

IO Example: Copying a File – Char by Char

▷ **Example 6.2.3** The following function copies the contents of from one text file, infile, to another, outfile character by character:

```
fun copyTextFile(infile: string, outfile: string) =
  let
    val ins = TextIO.openIn infile
    val outs = TextIO.openOut outfile
    fun helper(copt: char option) =
      case copt of
           NONE => (TextIO.closeIn ins; TextIO.closeOut outs)
         | SOME(c) => (TextIO.output1(outs,c);
                       helper(TextIO.input1 ins))
  in
    helper(TextIO.input1 ins)
  end
```

Note the use of the char option to model the fact that reading may fail(EOF)

6.2.2 Programming with Exceptions

The first kind of stateful functionality is a generalization of error handling: when an error occurs, we do not want to to continue computation in the usual way, and we do not want to return regular values of a function. Therefore SML introduces the concept of "raising an exception". When an exception is raised, functional computation aborts and the exception object is passed up to the next level, until it is handled (usually) by the interpreter.

Raising Exceptions

▷ Idea: Exceptions are generalized error codes

▷ **Definition 6.2.4** An exception is a special SML object. Raising an exception e in a function aborts functional computation and returns e to the next level.

▷ **Example 6.2.5** predefined exceptions (exceptions have names)

```
- 3 div 0;
uncaught exception divide by zero
raised at: <file stdIn>
- fib(100);
uncaught exception overflow
raised at: <file stdIn>
```

Exceptions are first-class citizens in SML, in particular they

▷ ▹ have types, and
 ▹ can be defined by the user.

▷ **Example 6.2.6** user-defined exceptions (exceptions are first-class objects)

```
- exception Empty;
exception Empty
- Empty;
val it = Empty : exn
```

▷ **Example 6.2.7** exception constructors (exceptions are just like any other value)

```
- exception SysError of int;
exception SysError of int;
- SysError
val it = fn : int -> exn
```

Let us fortify our intuition with a simple example: a factorial function. As SML does not have a type of natural numbers, we have to give the factorial function the type int −> int; in particular, we cannot use the SML type checker to reject arguments where the factorial function is not defined. But we can guard the function by raising an custom exception when the argument is negative.

Programming with Exceptions

▷ **Example 6.2.8** A factorial function that checks for non-negative arguments (just to be safe)

```
exception Factorial;
- fun safe_factorial n =
      if n < 0 then raise Factorial
      else if n = 0 then 1
      else n * safe_factorial (n-1)
val safe_factorial = fn : int -> int
- safe_factorial(~1);
uncaught exception Factorial
raised at: stdIn:28.31-28.40
```

unfortunately, this program checks the argument in every recursive call

Note that this function is inefficient, as it checks the guard on every call, even though we can see that in the recursive calls the argument must be non-negative by construction. The solution is to use two functions, an interface function which guards the argument with an exception and a recursive function that does not check.

Programming with Exceptions (next attempt)

▷ Idea: make use of local function definitions that do the real work as in

```
local
  fun fact 0 = 1 | fact n = n * fact (n-1)
in
  fun safe_factorial n =
  if n >= 0 then fact n else raise Factorial
end
```

this function only checks once, and the local function makes good use of pattern matching (⇝ standard programming pattern)

```
- safe_factorial(~1);
uncaught exception Factorial
raised at: stdIn:28.31-28.40
```

In the improved implementation, we see another new SML construct. **local** acts just like a **let**, only that it also that the body (the part between **in** and **end**) contains sequence of declarations and not an expression whose value is returned as the value of the overall expression. Here the identifier fact is bound to the recursive function in the definition of safe_factorial but unbound outside. This way we avoid polluting the name space with functions that are only used locally.

There is more to exceptions than just raising them to all the way the SML interpreter, which then shows them to the user. We can extend functions by exception handler that deal with any exceptions that are raised during their execution.

Handling Exceptions

▷ **Definition 6.2.9 (Idea)** Exceptions can be raised (through the evaluation pattern) and handled somewhere above (throw and catch)

▷ Consequence: Exceptions are a general mechanism for non-local transfers of control.

▷ **Definition 6.2.10 (SML Construct)** exception handler: exp **handle** rules

▷ **Example 6.2.11** Handling the Factorial expression

```
fun factorial_driver () =
    let val input = read_integer ()
        val result = toString (safe_factorial input)
    in
        print result
    end
handle Factorial => print "Out of range."
         | NaN => print "Not a Number!"
```

▷ **Example 6.2.12** the read_integer function (just to be complete)

```
exception NaN; (* Not a Number *)
fun read_integer () =
    let
        val intstring = case TextIO.inputLine(TextIO.stdIn) of
                                                NONE => raise NaN
                                              | SOME(s) => s
    in
       case Int.fromString intstring of
               NONE => raise NaN
             | SOME i => i
    end
```

The function factorial_driver in Example 6.2.11 uses two functions that can raise exceptions: safe_factorial defined in Example 6.2.8 and the function read_integer in Example 6.2.12. Both are handled to give a nicer error message to the user.

Note that the exception handler returns situations of exceptional control to normal (functional) computations. In particular, the results of exception handling have to of the same type as the ones from the functional part of the function. In Example 6.2.11, both the body of the function as well as the handers print the result, which makes them type-compatible and the function factorial_driver well-typed.

The previous example showed how to make use of the fact that exceptions are objects that are different from values. The next example shows how to take advantage of the fact that raising exceptions alters the flow of computation.

Using Exceptions for Optimizing Computation

▷ **Example 6.2.13 (Nonlocal Exit)** If we multiply a list of integers, we can stop when we see the first zero. So

```
local
  exception Zero
  fun p [] = 1
      | p (0::_) = raise Zero
      | p (h::t) = h ∗ p t
in
  fun listProdZero ns = p ns
        handle Zero => 0
end
```

is more efficient than just

```
fun listProd ns = fold op∗ ns 1
```

and the more clever

```
fun listProd ns = if member 0 ns then 0 else fold op∗ ns 1
```

The optimization in Example 6.2.13 works as follows: If we call listProd on a list l of length n, then SML will carry out $n+1$ multiplications, even though we (as humans) know that the product will be zero as soon as we see that the k^{th} element of l is zero. So the last $n-k+1$ multiplications are useless.

In listProdZero we use the local function p which raises the exception Zero in the head 0 case. Raising the exception allows control to leapfrog directly over the entire rest of the computation and directly allow the handler of listProdZero to return the correct value (zero).

For more information on SML

RTFM ($\widehat{=}$ "read the fine manuals")

Part II

Syntax and Semantics

Chapter 7

Encoding Programs as Strings

With the abstract data types we looked at last, we studied term structures, i.e. complex mathematical objects that were built up from constructors, variables and parameters. The motivation for this is that we wanted to understand SML programs. And indeed we have seen that there is a close connection between SML programs on the one side and abstract data types and procedures on the other side. However, this analysis only holds on a very high level, SML programs are not terms per se, but sequences of characters we type to the keyboard or load from files. We only interpret them to be terms in the analysis of programs.

To drive our understanding of programs further, we will first have to understand more about sequences of characters (strings) and the interpretation process that derives structured mathematical objects (like terms) from them. Of course, not every sequence of characters will be interpretable, so we will need a notion of (legal) well-formed sequence.

7.1 Formal Languages

We will now formally define the concept of strings and (building on that) formal languages.

The Mathematics of Strings

▷ **Definition 7.1.1** An alphabet A is a finite set; we call each element $a \in A$ a character, and an n-tuple of $s \in A^n$ a string (of length n over A).

▷ **Definition 7.1.2** Note that $A^0 = \{\langle\rangle\}$, where $\langle\rangle$ is the (unique) 0-tuple. With the definition above we consider $\langle\rangle$ as the string of length 0 and call it the empty string and denote it with ϵ

▷ Note: Sets $\neq$ Strings, e.g. $\{1,2,3\} = \{3,2,1\}$, but $\langle 1,2,3\rangle \neq \langle 3,2,1\rangle$.

▷ **Notation 7.1.3** We will often write a string $\langle c_1,\ldots,c_n\rangle$ as "$c_1 \ldots c_n$", for instance "abc" for $\langle \mathsf{a},\mathsf{b},\mathsf{c}\rangle$

▷ **Example 7.1.4** Take $A = \{\mathtt{h}, 1, /\}$ as an alphabet. Each of the symbols $\mathtt{h}$, 1, and $/$ is a character. The vector $\langle /, /, 1, \mathtt{h}, 1\rangle$ is a string of length 5 over A.

▷ **Definition 7.1.5 (String Length)** Given a string s we denote its length with $|s|$.

▷ **Definition 7.1.6** The concatenation $\mathrm{conc}(s,t)$ of two strings $s = \langle s_1, ..., s_n\rangle \in A^n$ and $t = \langle t_1, ..., t_m\rangle \in A^m$ is defined as $\langle s_1, ..., s_n, t_1, ..., t_m\rangle \in A^{n+m}$.

We will often write conc(s,t) as $s+t$ or simply st (e.g. conc("text", "book") = "text" + "book" = "textbook")

We have multiple notations for concatenation, since it is such a basic operation, which is used so often that we will need very short notations for it, trusting that the reader can disambiguate based on the context.

Now that we have defined the concept of a string as a sequence of characters, we can go on to give ourselves a way to distinguish between good strings (e.g. programs in a given programming language) and bad strings (e.g. such with syntax errors). The way to do this by the concept of a formal language, which we are about to define.

Formal Languages

▷ **Definition 7.1.7** Let A be an alphabet, then we define the sets $A^+ := \bigcup_{i\in\mathbb{N}^+} A^i$ of nonempty strings and $A^* := A^+ \cup \{\epsilon\}$ of strings.

▷ **Example 7.1.8** If $A = \{a,b,c\}$, then $A^* = \{\epsilon, a, b, c, aa, ab, ac, ba, \ldots, aaa, \ldots\}$.

▷ **Definition 7.1.9** A set $L \subseteq A^*$ is called a formal language in A.

▷ **Definition 7.1.10** We use $c^{[n]}$ for the string that consists of n times c.

▷ **Example 7.1.11** $\#^{[5]} = \langle \#,\#,\#,\#,\# \rangle$

▷ **Example 7.1.12** The set $M = \{ba^{[n]} \,|\, n \in \mathbb{N}\}$ of strings that start with character b followed by an arbitrary numbers of a's is a formal language in $A = \{a,b\}$.

▷ **Definition 7.1.13** The concatenation conc(L_1, L_2) of two languages L_1 and L_2 over the same alphabet is defined as conc$(L_1, L_2) := \{s_1 s_2 \,|\, s_1 \in L_1 \wedge s_2 \in L_2\}$.

There is a common misconception that a formal language is something that is difficult to understand as a concept. This is not true, the only thing a formal language does is separate the "good" from the bad strings. Thus we simply model a formal language as a set of stings: the "good" strings are members, and the "bad" ones are not.

Of course this definition only shifts complexity to the way we construct specific formal languages (where it actually belongs), and we have learned two (simple) ways of constructing them by repetition of characters, and by concatenation of existing languages.

The next step will be to define some operations and relations on these new objects: strings. As always, we are interested in well-formed sub-objects and ordering relations.

Substrings and Prefixes of Strings

▷ **Definition 7.1.14** Let A be an alphabet, then we say that a string $s \in A^*$ is a substring of a string $t \in A^*$ (written $s \subseteq t$), iff there are strings $v, w \in A^*$, such that $t = vsw$.

▷ **Example 7.1.15** conc(/, 1, h) is a substring of conc(/, /, 1, h, 1), whereas conc(/, 1, 1) is not.

▷ **Definition 7.1.16** A string p is a called a prefix of s (write $p \unlhd s$), iff there is a string t, such that $s = \mathrm{conc}(p, t)$. p is a proper prefix of s (write $p \lhd s$), iff $t \neq \epsilon$.

▷ **Example 7.1.17** $text$ is a prefix of $textbook = \mathrm{conc}(text, book)$.

▷ Note: A string is never a proper prefix of itself.

We will now define an ordering relation for formal languages. The nice thing is that we can induce an ordering on strings from an ordering on characters, so we only have to specify that (which is simple for finite alphabets).

Lexical Order

▷ **Definition 7.1.18** Let A be an alphabet and $\prec$ a strict partial order on A, Then we define a relation $\prec_{\mathsf{lex}}$ on A^* by

$$(s \prec_{\mathsf{lex}} t) :\Leftrightarrow s \lhd t \vee (\exists u, v, w \in A^*. \exists a, b \in A. s = wau \wedge t = wbv \wedge (a \prec b))$$

for $s, t \in A^*$. We call $\prec_{\mathsf{lex}}$ the lexical order induced by $\prec$ on A^*.

▷ **Theorem 7.1.19** *$\prec_{lex}$ is a strict partial order on A^*. Moreover, if $\prec$ is linear on A, then $\prec_{lex}$ is linear on A^*.*

▷ **Example 7.1.20** Roman alphabet with $\mathtt{a} < \mathtt{b} < \mathtt{c} \cdots < \mathtt{z} \leadsto$ telephone book order ($computer <_{\mathsf{lex}} text, text <_{\mathsf{lex}} textbook$)

Even though the definition of the lexical ordering is relatively involved, we know it very well, it is the ordering we know from the telephone books.

7.2 Elementary Codes

The next task for understanding programs as mathematical objects is to understand the process of using strings to encode objects. The simplest encodings or "codes" are mappings from strings to strings. We will now study their properties.

The most characterizing property for a code is that if we encode something with this code, then we want to be able to decode it again: We model a code as a function (every character should have a unique encoding), which has a partial inverse (so we can decode). We have seen above, that this is is the case, iff the function is injective; so we take this as the defining characteristic of a code.

Character Codes

▷ **Definition 7.2.1** Let A and B be alphabets, then we call an injective function $c \colon A \to B^+$ a character code. A string $c(w) \in \{c(a) \mid a \in A\}$ is called a codeword.

▷ **Definition 7.2.2** A code is a called binary iff $B = \{0, 1\}$.

▷ **Example 7.2.3** Let $A = \{a, b, c\}$ and $B = \{0, 1\}$, then $c\colon A \to B^+$ with $c(a) = 0011$, $c(b) = 1101$, $c(c) = 0110$ c is a binary character code and the strings 0011, 1101, and 0110 are the codewords of c.

▷ **Definition 7.2.4** The extension of a code (on characters) $c\colon A \to B^+$ to a function $c'\colon A^* \to B^*$ is defined as $c'(\langle a_1, \ldots, a_n\rangle = \langle c(a_1), \ldots, c(a_n)\rangle)$.

▷ **Example 7.2.5** The extension c' of c from the above example on the string "bbabc"

$$c'(\texttt{"bbabc"}) = \underbrace{1101}_{c(b)}, \underbrace{1101}_{c(b)}, \underbrace{0011}_{c(a)}, \underbrace{1101}_{c(b)}, \underbrace{0110}_{c(c)}$$

▷ **Definition 7.2.6** A (character) code $c\colon A \to B^+$ is a prefix code iff none of the codewords is a proper prefix to an other codeword, i.e.,

$$\forall x, y \in A.x \neq y \Rightarrow (c(x) \ntriangleleft c(y) \wedge c(y) \ntriangleleft c(x))$$

Character codes in and of themselves are not that interesting: we want to encode strings in another alphabet. But they can be turned into mappings from strings to strings by extension: strings are sequences of characters, so we can encode them by concatenating the codewords.

We will now come to a paradigmatic example of an encoding function: the Morse code, which was used for transmitting text information as standardized sequences of short and long signals called "dots" and "dashes".

Morse Code

▷ In the early days of telecommunication the "Morse Code" was used to transmit texts, using long and short pulses of electricity.

▷ **Definition 7.2.7 (Morse Code)** The following table gives the Morse code for the text characters:

A	.—	B	—...	C	—.—.	D	—..	E	.
F	..—.	G	——.	H	....	I	..	J	.———
K	—.—	L	.—..	M	——	N	—.	O	———
P	.——.	Q	——.—	R	.—.	S	...	T	—
U	..—	V	...—	W	.——	X	—..—	Y	—.——
Z	——..								
1	.————	2	..———	3	...——	4	....—	5	.....
6	—....	7	——...	8	———..	9	————.	0	—————

Furthermore, the Morse code uses `.-.-.-` for full stop (sentence termination), `--..--` for comma, and `..--..` for question mark.

▷ **Example 7.2.8** The Morse Code in the table above induces a character code $\mu\colon \mathcal{R} \to \{\texttt{.}, \texttt{-}\}$.

Note: The Morse code is a character code, but its extension (which was actually used for transmission of texts) is not an injective mapping, as e.g. $\mu'(\texttt{AT}) = \mu'(\texttt{W})$. While this is mathematically a problem for decoding texts encoded by Morse code, for humans (who were actually decoding it) this ist not, since they understand the meaning of the texts and can thus eliminate nonsensical possibilities, preferring the string "ARRIVE AT 6" over "ARRIVE W 6".

The Morse code example already suggests the next topic of investigation: When are the extensions of character codes injective mappings (that we can decode automatically, without taking other information into account).

Codes on Strings

▷ **Definition 7.2.9** A function $c'\colon A^* \to B^*$ is called a code on strings or short string code if c' is an injective function.

▷ **Theorem 7.2.10** (⚠2) *There are character codes whose extensions are not string codes.*

▷ Proof: we give an example

P.1 Let $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $B = \{0, 1\}$, $c(\mathtt{a}) = 0$, $c(\mathtt{b}) = 1$, and $c(\mathtt{c}) = 01$.

P.2 The function c is injective, hence it is a character code.

P.3 But its extension c' is not injective as $c'(\mathtt{ab}) = 01 = c'(\mathtt{c})$. □

Question: When is the extension of a character code a string code?(so we can encode strings)

Note that in contrast to checking for injectivity on character codes – where we have to do $n^2/2$ comparisons for a source alphabet A of size n, we are faced with an infinite problem, since A^* is infinite. Therefore we look for sufficient conditions for injectivity that can be decided by a finite process.

We will answer the question above by proving one of the central results of elementary coding theory: *prefix codes induce string codes*. This plays back the infinite task of checking that a string code is injective to a finite task (checking whether a character code is a prefix code).

▷ Prefix Codes induce Codes on Strings

▷ **Theorem 7.2.11** *The extension $c'\colon A^* \to B^*$ of a prefix code $c\colon A \to B^+$ is a string code.*

▷ Proof: We will prove this theorem via induction over the string length n

P.1 We show that c' is injective (decodable) on strings of length $n \in \mathbb{N}$.

P.1.1 $n = 0$ (base case): If $|s| = 0$ then $c'(\epsilon) = \epsilon$, hence c' is injective.

P.1.2 $n = 1$ (another): If $|s| = 1$ then $c' = c$ thus injective, as c is char. code.

P.1.3 Induction step (n to $n + 1$):

P.1.3.1 Let $a = a_0, \ldots, a_n$, And we only know $c'(a) = c(a_0), \ldots, c(a_n)$.

P.1.3.2 It is easy to find $c(a_0)$ in $c'(a)$: It is the prefix of $c'(a)$ that is in $c(A)$. This is uniquely determined, since c is a prefix code. If there were two distinct ones, one would have to be a prefix of the other, which contradicts our assumption that c is a prefix code.

P.1.3.3 If we remove $c(a_0)$ from $c(a)$, we only have to decode $c(a_1), \ldots, c(a_n)$, which we can do by inductive hypothesis. □

P.2 Thus we have considered all the cases, and proven the assertion. □

Even armed with Theorem 7.2.11, checking whether a code is a prefix code can be a tedious undertaking: the naive algorithm for this needs to check all pairs of codewords. Therefore we will look at a couple of properties of character codes that will ensure a prefix code and thus decodeability.

Sufficient Conditions for Prefix Codes

▷ **Theorem 7.2.12** *If c is a code with $|c(a)| = k$ for all $a \in A$ for some $k \in \mathbb{N}$, then c is prefix code.*

▷ Proof: by contradiction.

P.1 If c is not at prefix code, then there are $a, b \in A$ with $c(a) \lhd c(b)$.

P.2 clearly $|c(a)| < |c(b)|$, which contradicts our assumption. □

▷ **Theorem 7.2.13** *Let $c\colon A \to B^+$ be a code and $* \notin B$ be a character, then there is a prefix code $c^*\colon A \to (B \cup \{*\})^+$, such that $c(a) \lhd c^*(a)$, for all $a \in A$.*

▷ Proof: Let $c^*(a) := c(a) + "*"$ for all $a \in A$.

P.1 Obviously, $c(a) \lhd c^*(a)$.

P.2 If c^* is not a prefix code, then there are $a, b \in A$ with $c^*(a) \lhd c^*(b)$.

P.3 So, $c^*(b)$ contains the character $*$ not only at the end but also somewhere in the middle.

P.4 This contradicts our construction $c^*(b) = c(b) + "*"$, where $c(b) \in B^+$ □

▷ **Definition 7.2.14** The new character that makes an arbitrary code a prefix code in the construction of Theorem 7.2.13 is often called a stop character.

Theorem 7.2.13 allows another interpretation of the decodeability of the Morse code: it can be made into a prefix code by adding a stop character (in Morse code a little pause). In reality, pauses (as stop characters) were only used where needed (e.g. when transmitting otherwise meaningless character sequences).

7.3 Character Codes in the Real World

We will now turn to a class of codes that are extremely important in information technology: character encodings. The idea here is that for IT systems we need to encode characters from our alphabets as bit strings (sequences of binary digits 0 and 1) for representation in computers. Indeed the Morse code we have seen above can be seen as a very simple example of a character encoding that is geared towards the manual transmission of natural languages over telegraph lines. For the encoding of written texts we need more extensive codes that can e.g. distinguish upper and lowercase letters.

It is important to understand that encoding and decoding of characters is an activity that requires standardization in multi-device settings – be it sending a file to the printer or sending an e-mail to a friend on another continent. Concretely, the recipient wants to use the same character mapping for decoding the sequence of bits as the sender used for encoding them – otherwise the message is garbled.

We observe that we cannot just specify the encoding table in the transmitted document itself, (that information would have to be en/decoded with the other content), so we need to rely document-external external methods like standardization or encoding negotiation at the meta-level. In this Section we will focus on the former.

The `ASCII` code we will introduce here is one of the first standardized and widely used character encodings for a complete alphabet. It is still widely used today. The code tries to strike a balance between a being able to encode a large set of characters and the representational capabilities in the time of punch cards (see below).

The ASCII Character Code

▷ **Definition 7.3.1** The American Standard Code for Information Interchange (`ASCII`) is a character code that assigns characters to numbers 0-127

Code	···0	···1	···2	···3	···4	···5	···6	···7	···8	···9	···A	···B	···C	···D	···E	···F
0···	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1···	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2···		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3···	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4···	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5···	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6···	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7···	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

The first 32 characters are control characters for ASCII devices like printers

▷ Motivated by punchcards: The character 0 (binary 0000000) carries no information `NUL`, (used as dividers)
Character 127 (binary 1111111) can be used for deleting (overwriting) last value (cannot delete holes)

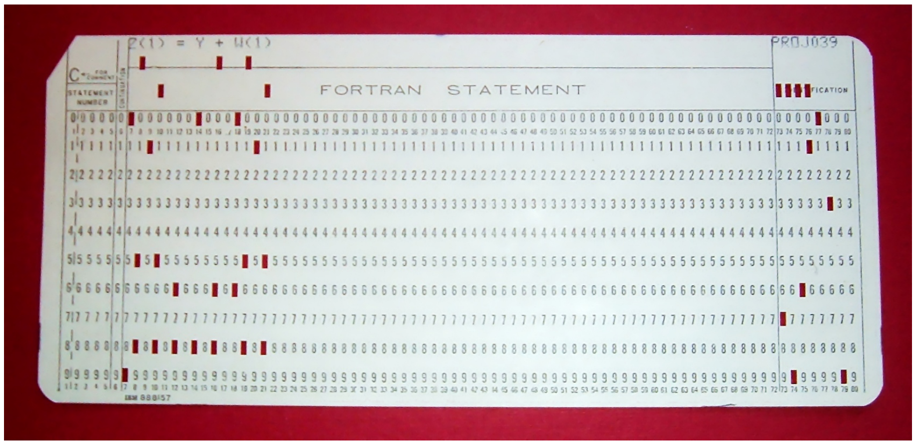
▷ The ASCII code was standardized in 1963 and is still prevalent in computers today (but seen as US-centric)

Punch cards were the the preferred medium for long-term storage of programs up to the late 1970s, since they could directly be produced by card punchers and automatically read by computers.

A Punchcard

▷ A punch card is a piece of stiff paper that contains digital information represented by the presence or absence of holes in predefined positions.

▷ **Example 7.3.2** This punch card encoded the `FORTRAN` statement Z(1) = Y + W(1)



Up to the 1970s, computers were batch machines, where the programmer delivered the program to the operator (a person behind a counter who fed the programs to the computer) and collected the printouts the next morning. Essentially, each punch card represented a single line (80 characters) of program code. Direct interaction with a computer is a relatively young mode of operation.

The `ASCII` code as above has a variety of problems, for instance that the control characters are mostly no longer in use, the code is lacking many characters of languages other than the English language it was developed for, and finally, it only uses seven bits, where a byte (eight bits) is the preferred unit in information technology. Therefore there have been a whole zoo of extensions, which — due to the fact that there were so many of them — never quite solved the encoding problem.

Problems with `ASCII` encoding

- ▷ Problem: Many of the control characters are obsolete by now (e.g. `NUL`,`BEL`, or `DEL`)
- ▷ Problem: Many European characters are not represented (e.g. è,ñ,ü,ß,. . .)
- ▷ European ASCII Variants: Exchange less-used characters for national ones
- ▷ **Example 7.3.3 (German `ASCII`)** remap e.g. `[` ↦ Ä, `]` ↦ Ü in German `ASCII` (“`Apple ][`” comes out as “`Apple ÜÄ`”)
- ▷ **Definition 7.3.4 (ISO-Latin (ISO/IEC 8859))** 16 Extensions of ASCII to 8-bit (256 characters) ISO-Latin 1 $\widehat{=}$ “Western European”, ISO-Latin 6 $\widehat{=}$ “Arabic”,ISO-Latin 7 $\widehat{=}$ “Greek”. . .
- ▷ Problem: No cursive Arabic, Asian, African, Old Icelandic Runes, Math,. . .
- ▷ Idea: Do something totally different to include all the world’s scripts: For a scalable architecture, separate
  - ▷ what characters are available from the (character set)
  - ▷ bit string-to-character mapping (character encoding)

The goal of the UniCode standard is to cover all the worlds scripts (past, present, and future) and provide efficient encodings for them. The only scripts in regular use that are currently excluded are fictional scripts like the elvish scripts from the Lord of the Rings or Klingon scripts from the Star Trek series.

An important idea behind UniCode is to separate concerns between standardizing the character set — i.e. the set of encodable characters and the encoding itself.

Unicode and the Universal Character Set

▷ **Definition 7.3.5 (Twin Standards)** A scalable Architecture for representing all the worlds scripts

- ▷ The universal character set defined by the ISO/IEC 10646 International Standard, is a standard set of characters upon which many character encodings are based.
- ▷ The unicode Standard defines a set of standard character encodings, rules for normalization, decomposition, collation, rendering and bidirectional display order

▷ **Definition 7.3.6** Each UCS character is identified by an unambiguous name and an integer number called its code point.

▷ The UCS has 1.1 million code points and nearly 100 000 characters.

▷ **Definition 7.3.7** Most (non-Chinese) characters have code points in $[1, 65536]$ (the basic multilingual plane).

▷ **Notation 7.3.8** For code points in the Basic Multilingual Plane (BMP), four digits are used, e.g. $\mathrm{U}+0058$ for the character LATIN CAPITAL LETTER X;

Note that there is indeed an issue with space-efficient encoding here. UniCode reserves space for 2^{32} (more than a million) characters to be able to handle future scripts. But just simply using 32 bits for every UniCode character would be extremely wasteful: UniCode-encoded versions of ASCII files would be four times as large.

Therefore UniCode allows multiple encodings. UTF-32 is a simple 32-bit code that directly uses the code points in binary form. UTF-8 is optimized for western languages and coincides with the ASCII where they overlap. As a consequence, ASCII encoded texts can be decoded in UTF-8 without changes — but in the UTF-8 encoding, we can also address all other UniCode characters (using multi-byte characters).

Character Encodings in Unicode

▷ **Definition 7.3.9** A character encoding is a mapping from bit strings to UCS code points.

▷ Idea: Unicode supports multiple encodings (but not character sets) for efficiency

▷ **Definition 7.3.10 (Unicode Transformation Format)**

- ▷ UTF-8, 8-bit, variable-width encoding, which maximizes compatibility with ASCII.
- ▷ UTF-16, 16-bit, variable-width encoding (popular in Asia)
- ▷ UTF-32, a 32-bit, fixed-width encoding (for safety)

▷ **Definition 7.3.11** The UTF-8 encoding follows the following encoding scheme

Unicode	Byte1	Byte2	Byte3	Byte4
$U+000000 - U+00007F$	0xxxxxxx			
$U+000080 - U+0007FF$	110xxxxx	10xxxxxx		
$U+000800 - U+00FFFF$	1110xxxx	10xxxxxx	10xxxxxx	
$U+010000 - U+10FFFF$	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

▷ **Example 7.3.12** \$ $= U+0024$ is encoded as 00100100 (1 byte)

¢ $= U+00A2$ is encoded as 11000010,10100010 (two bytes)

€ $= U+20AC$ is encoded as 11100010,10000010,10101100 (three bytes)

Note how the fixed bit prefixes in the encoding are engineered to determine which of the four cases apply, so that UTF-8 encoded documents can be safely decoded..

7.4 Formal Languages and Meaning

After we have studied the elementary theory of codes for strings, we will come to string representations of structured objects like terms. For these we will need more refined methods.

As we have started out the course with unary natural numbers and added the arithmetical operations to the mix later, we will use unary arithmetics as our running example and study object.

A formal Language for Unary Arithmetics

▷ Idea: Start with something very simple: Unary Arithmetics (i.e. $\mathbb{N}$ with addition, multiplication, subtraction, and integer division)

▷ E_{un} is based on the alphabet $\Sigma_{\text{un}} := C_{\text{un}} \cup V \cup F_{\text{un}}^2 \cup B$, where

- ▷ $C_{\text{un}} := \{/\}^*$ is a set of constant names,
- ▷ $V := \{\mathtt{x}\} \times \{\mathtt{1},\ldots,\mathtt{9}\} \times \{\mathtt{0},\ldots,\mathtt{9}\}^*$ is a set of variable names,
- ▷ $F_{\text{un}}^2 := \{\text{add}, \text{sub}, \text{mul}, \text{div}, \text{mod}\}$ is a set of (binary) function names, and
- ▷ $B := \{\mathtt{(}, \mathtt{)}\} \cup \{\mathtt{,}\}$ is a set of structural characters. (⚠ “,”,”(“,”)” characters!)

▷ define strings in stages: $E_{\text{un}} := \bigcup_{i\in\mathbb{N}} {E_{\text{un}}}^i$, where

- ▷ ${E_{\text{un}}}^1 := C_{\text{un}} \cup V$
- ▷ ${E_{\text{un}}}^{i+1} := \{a, \text{add}\mathtt{(}a\mathtt{,}b\mathtt{)}, \text{sub}\mathtt{(}a\mathtt{,}b\mathtt{)}, \text{mul}\mathtt{(}a\mathtt{,}b\mathtt{)}, \text{div}\mathtt{(}a\mathtt{,}b\mathtt{)}, \text{mod}\mathtt{(}a\mathtt{,}b\mathtt{)} \mid a, b \in {E_{\text{un}}}^i\}$

We call a string in E_{un} an expression of unary arithmetics.

The first thing we notice is that the alphabet is not just a flat any more, we have characters with different roles in the alphabet. These roles have to do with the symbols used in the complex objects (unary arithmetic expressions) that we want to encode.

The formal language E_{un} is constructed in stages, making explicit use of the respective roles of the characters in the alphabet. Constants and variables form the basic inventory in ${E_{\text{un}}}^1$, the respective next stage is built up using the function names and the structural characters to encode the applicative structure of the encoded terms.

Note that with this construction ${E_{\text{un}}}^i \subseteq {E_{\text{un}}}^{i+1}$.

A formal Language for Unary Arithmetics (Examples)

▷ **Example 7.4.1** add(//////,mul(x1902,///)) $\in E_{\text{un}}$

▷ Proof: we proceed according to the definition

P.1 We have ////// $\in C_{\text{un}}$, and x1902 $\in V$, and /// $\in C_{\text{un}}$ by definition

P.2 Thus ////// $\in {E_{\text{un}}}^1$, and x1902 $\in {E_{\text{un}}}^1$ and /// $\in {E_{\text{un}}}^1$,

P.3 Hence, ////// $\in {E_{\text{un}}}^2$ and mul(x1902,///) $\in {E_{\text{un}}}^2$

P.4 Thus add(//////,mul(x1902,///)) $\in {E_{\text{un}}}^3$

P.5 And finally add(//////,mul(x1902,///)) $\in E_{\text{un}}$ □

▷ other examples:

- ▷ div(x201,add(////,x12))
- ▷ sub(mul(///,div(x23,///)),///)

▷ what does it all mean? (nothing, E_{un} is just a set of strings!)

To show that a string is an expression s of unary arithmetics, we have to show that it is in the formal language E_{un}. As E_{un} is the union over all the ${E_{\text{un}}}^i$, the string s must already be a member of a set ${E_{\text{un}}}^j$ for some $j \in \mathbb{N}$. So we reason by the definintion establising set membership.

Of course, computer science has better methods for defining languages than the ones used here (context free grammars), but the simple methods used here will already suffice to make the relevant points for this course.

Syntax and Semantics (a first glimpse)

▷ **Definition 7.4.2** A formal language is also called a syntax, since it only concerns the "form" of strings.

▷ to give meaning to these strings, we need a semantics, i.e. a way to interpret these.

▷ Idea (Tarski Semantics): A semantics is a mapping from strings to objects we already know and understand (e.g. arithmetics).

- ▷ e.g. add(//////,mul(x1902,///)) $\mapsto 6 + (x_{1902} \cdot 3)$ (but what does this mean?)
- ▷ looks like we have to give a meaning to the variables as well, e.g. x1902 $\mapsto$ 3, then add(//////,mul(x1902,///)) $\mapsto 6 + (3 \cdot 3) = 15$

So formal languages do not mean anything by themselves, but a meaning has to be given to them via a mapping. We will explore that idea in more detail in the following.

Recommended Reading: Even though syntax and semantics are essential topics in Computer Science, we will not go any deeper here, and lay the topic to rest for the time beeing – though if you keep your eyes open, then you will encounter it on every step. For students who are interested in the topic we erecommend two delightful books that study the topic from a different – but no less serios angle: Douglas Hofstadter's "Gödel, Escher, Bach" [Hof79] and Doxiadēdes et al.'s "Logicomix" [DPPDD09]. And for students who like riddles and puzzles, we recommend any book by Raymond Smullyan (who says logic cannot be fun?).

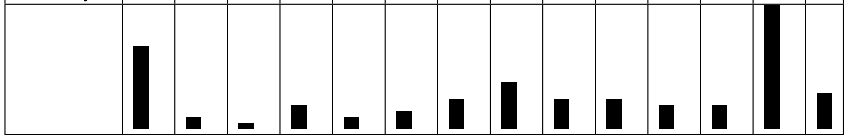
Chapter 8

Midterm Analysis

Mid-Term Results

- ▷ Midterm: 9 Problems, 55 Points,
- ▷ Grades: Average grade: 3.33 (barely "satisfactory")

Performance	exc.	very good		good		satisfactory			sufficient			failing		
Jacobs Grade	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33	3.67	4.00	4.33	4.67	5.00	∅
Cardinality	14	2	1	4	2	3	5	8	5	5	4	4	21	6



- ▷ simplest problems: Greek letters (avg.77%), function definition (avg. 64%),
- ▷ hardest problems: CNF/DNF (avg. 14%), SML (avg. 36%)

Procedure, Consequences

- ▷ Procedure (some order please to minimize this boring but necessary task)
  - ▷ Go to Exam Inspection (Today! 13:00-14:00 GenCS Lounge (103@R1))
  - ▷ You will check the grading, points summation, . . .
  - ▷ We will answer questions, and correct mistakes.
  - ▷ You will take home the test, when you leave the room the grade is final!
- ▷ Consequences (we all want a better result in the final and the final grade)
  - ▷ try to know actively (just passively understanding is not enough)
  - ▷ try to write anything at all (so we can give you partial points)
  - ▷ you will need to take more advantage of tutorials and TAs (we are here to help you!)

▷ There is really no need to fail this course (if you do, rethink your major)

Chapter 9

Boolean Algebra

We will now look a formal language from a different perspective. We will interpret the language of "Boolean expressions" as formulae of a very simple "logic": A logic is a mathematical construct to study the association of meaning to strings and reasoning processes, i.e. to study how humans[1] derive new information and knowledge from existing one.

9.1 Boolean Expressions and their Meaning

In the following we will consider the Boolean Expressions as the language of "Propositional Logic", in many ways the simplest of logics. This means we cannot really express very much of interest, but we can study many things that are common to all logics.

Let us try again (Boolean Expressions)

▷ **Definition 9.1.1 (Alphabet)** E_{bool} is based on the alphabet $\mathcal{A} := C_{\text{bool}} \cup V \cup F^1_{\text{bool}} \cup F^2_{\text{bool}} \cup B$, where $C_{\text{bool}} = \{0, 1\}$, $F^1_{\text{bool}} = \{\texttt{-}\}$ and $F^2_{\text{bool}} = \{\texttt{+}, \texttt{*}\}$. ($V$ and B as in E_{un})

▷ **Definition 9.1.2 (Formal Language)** $E_{\text{bool}} := \bigcup_{i \in \mathbb{N}} {E_{\text{bool}}}^i$, where ${E_{\text{bool}}}^1 := C_{\text{bool}} \cup V$ and
${E_{\text{bool}}}^{i+1} := \{a, (\texttt{-}a), (a\texttt{+}b), (a\texttt{*}b) \mid a, b \in {E_{\text{bool}}}^i\}$.

▷ **Definition 9.1.3** Let $a \in E_{\text{bool}}$. The minimal i, such that $a \in {E_{\text{bool}}}^i$ is called the depth of a.

▷ $e_1 :=$ `((-x1)+x3)` (depth 3)

▷ $e_2 :=$ `((-(x1*x2))+(x3*x4))` (depth 4)

▷ $e_3 :=$ `((x1+x2)+((-((-x1)*x2))+(x3*x4)))` (depth 6)

Boolean Expressions as Structured Objects.

[1]until very recently, humans were thought to be the only systems that could come up with complex argumentations. In the last 50 years this has changed: not only do we attribute more reasoning capabilities to animals, but also, we have developed computer systems that are increasingly capable of reasoning.

▷ Idea: As strings in in E_{bool} are built up via the "union-principle", we can think of them as constructor terms with variables

▷ **Definition 9.1.4** The abstract data type

$$\mathcal{B} := \langle\{\mathbb{B}\}, \{[1\colon \mathbb{B}], [0\colon \mathbb{B}], [\text{-}\colon \mathbb{B}\to\mathbb{B}], [+\colon \mathbb{B}\times\mathbb{B}\to\mathbb{B}], [*\colon \mathbb{B}\times\mathbb{B}\to\mathbb{B}]\}\rangle$$

▷ via the translation

▷ **Definition 9.1.5** $\sigma\colon E_{\text{bool}} \to \mathcal{T}_{\mathbb{B}}(\mathcal{B};\mathcal{V})$ defined by

$$\begin{array}{ll} \sigma(1) := 1 & \sigma(0) := 0 \\ \sigma((\text{-}A)) := (\text{-}\sigma(A)) & \\ \sigma((A*B)) := (\sigma(A)*\sigma(B)) & \sigma((A+B)) := (\sigma(A)+\sigma(B)) \end{array}$$

▷ We will use this intuition for our treatment of Boolean expressions and treat the strings and constructor terms synonymously. (σ is a (hidden) isomorphism)

▷ **Definition 9.1.6** We will write $(\text{-}A)$ as $\overline{A}$ and $(A*B)$ as $A*B$ (and similarly for +). Furthermore we will write variables such as $x71$ as x_{71} and elide brackets for sums and products according to their usual precedences.

▷ **Example 9.1.7** $\sigma(((\text{-}(x1*x2))+(x3*x4))) = \overline{x_1 * x_2} + x_3 * x_4$

▷ ⚠: Do not confuse + and ∗ (Boolean sum and product) with their arithmetic counterparts. (as members of a formal language they have no meaning!)

Now that we have defined the formal language, we turn the process of giving the strings a meaning. We make explicit the idea of providing meaning by specifying a function that assigns objects that we already understand to representations (strings) that do not have a priori meaning.

The first step in assigning meaning is to fix a set of objects what we will assign as meanings: the "universe (of discourse)". To specify the meaning mapping, we try to get away with specifying as little as possible. In our case here, we assign meaning only to the constants and functions and induce the meaning of complex expressions from these. As we have seen before, we also have to assign meaning to variables (which have a different ontological status from constants); we do this by a special meaning function: a variable assignment.

Boolean Expressions: Semantics via Models

▷ **Definition 9.1.8** A model $\langle\mathcal{U},\mathcal{I}\rangle$ for E_{bool} is a set $\mathcal{U}$ of objects (called the universe) together with an interpretation function $\mathcal{I}$ on $\mathcal{A}$ with $\mathcal{I}(C_{\text{bool}}) \subseteq \mathcal{U}$, $\mathcal{I}(F^1_{\text{bool}}) \subseteq \mathcal{F}(\mathcal{U};\mathcal{U})$, and $\mathcal{I}(F^2_{\text{bool}}) \subseteq \mathcal{F}(\mathcal{U}^2;\mathcal{U})$.

▷ **Definition 9.1.9** A function $\varphi\colon V \to \mathcal{U}$ is called a variable assignment.

▷ **Definition 9.1.10** Given a model $\langle\mathcal{U},\mathcal{I}\rangle$ and a variable assignment φ, the evaluation function $\mathcal{I}_\varphi\colon E_{\text{bool}} \to \mathcal{U}$ is defined recursively: Let $c \in C_{\text{bool}}$, $a, b \in E_{\text{bool}}$, and $x \in V$, then

- ▷ $\mathcal{I}_\varphi(c) = \mathcal{I}(c)$, for $c \in C_{\text{bool}}$
- ▷ $\mathcal{I}_\varphi(x) = \varphi(x)$, for $x \in V$

- ▷ $\mathcal{I}_\varphi(\overline{a}) = \mathcal{I}(\text{-})(\mathcal{I}_\varphi(a))$
- ▷ $\mathcal{I}_\varphi(a+b) = \mathcal{I}(+)(\mathcal{I}_\varphi(a), \mathcal{I}_\varphi(b))$ and $\mathcal{I}_\varphi(a*b) = \mathcal{I}(*)(\mathcal{I}_\varphi(a), \mathcal{I}_\varphi(b))$

▷ $\mathcal{U} = \{\mathsf{T}, \mathsf{F}\}$ with $0 \mapsto \mathsf{F}, 1 \mapsto \mathsf{T}, + \mapsto \vee, * \mapsto \wedge, \text{-} \mapsto \neg$.

▷ $\mathcal{U} = E_{\text{un}}$ with $0 \mapsto /, 1 \mapsto //, + \mapsto div, * \mapsto mod, \text{-} \mapsto \lambda x.(///)$.

▷ $\mathcal{U} = \{0,1\}$ with $0 \mapsto 0, 1 \mapsto 1, + \mapsto \min, * \mapsto \max, \text{-} \mapsto \lambda x.1-x$.

Note that all three models on the bottom of the last slide are essentially different, i.e. there is no way to build an isomorphism between them, i.e. a mapping between the universes, so that all Boolean expressions have corresponding values.

To get a better intuition on how the meaning function works, consider the following example. We see that the value for a large expression is calculated by calculating the values for its sub-expressions and then combining them via the function that is the interpretation of the constructor at the head of the expression.

Evaluating Boolean Expressions

▷ **Example 9.1.11** Let $\varphi := [\mathsf{T}/x_1], [\mathsf{F}/x_2], [\mathsf{T}/x_3], [\mathsf{F}/x_4]$, and $\mathcal{I} = \{0 \mapsto \mathsf{F}, 1 \mapsto \mathsf{T}, + \mapsto \vee, * \mapsto \wedge, \text{-} \mapsto \neg\}$, then

$$\begin{aligned}
& \mathcal{I}_\varphi((x_1+x_2)+(\overline{\overline{x_1}*x_2}+x_3*x_4)) \\
= \; & \mathcal{I}_\varphi(x_1+x_2) \vee \mathcal{I}_\varphi(\overline{\overline{x_1}*x_2}+x_3*x_4) \\
= \; & \mathcal{I}_\varphi(x_1) \vee \mathcal{I}_\varphi(x_2) \vee \mathcal{I}_\varphi(\overline{\overline{x_1}*x_2}) \vee \mathcal{I}_\varphi(x_3*x_4) \\
= \; & \varphi(x_1) \vee \varphi(x_2) \vee \neg(\mathcal{I}_\varphi(\overline{x_1}*x_2)) \vee \mathcal{I}_\varphi(x_3*x_4) \\
= \; & (\mathsf{T} \vee \mathsf{F}) \vee (\neg(\mathcal{I}_\varphi(\overline{x_1}) \wedge \mathcal{I}_\varphi(x_2)) \vee (\mathcal{I}_\varphi(x_3) \wedge \mathcal{I}_\varphi(x_4))) \\
= \; & \mathsf{T} \vee \neg(\neg(\mathcal{I}_\varphi(x_1)) \wedge \varphi(x_2)) \vee (\varphi(x_3) \wedge \varphi(x_4)) \\
= \; & \mathsf{T} \vee \neg(\neg(\varphi(x_1)) \wedge \mathsf{F}) \vee (\mathsf{T} \wedge \mathsf{F}) \\
= \; & \mathsf{T} \vee \neg(\neg(\mathsf{T}) \wedge \mathsf{F}) \vee \mathsf{F} \\
= \; & \mathsf{T} \vee \neg(\mathsf{F} \wedge \mathsf{F}) \vee \mathsf{F} \\
= \; & \mathsf{T} \vee \neg(\mathsf{F}) \vee \mathsf{F} = \mathsf{T} \vee \mathsf{T} \vee \mathsf{F} = \mathsf{T}
\end{aligned}$$

▷ What a mess!

Boolean Algebra

▷ **Definition 9.1.12** A Boolean algebra is E_{bool} together with the models

- ▷ $\langle\{\mathsf{T}, \mathsf{F}\}, \{0 \mapsto \mathsf{F}, 1 \mapsto \mathsf{T}, + \mapsto \vee, * \mapsto \wedge, \text{-} \mapsto \neg\}\rangle$.
- ▷ $\langle\{0,1\}, \{0 \mapsto 0, 1 \mapsto 1, + \mapsto \max, * \mapsto \min, \text{-} \mapsto \lambda x.1-x\}\rangle$.

▷ BTW, the models are equivalent ($0 \hat{=} \mathsf{F}$, $1 \hat{=} \mathsf{T}$)

▷ **Definition 9.1.13** We will use $\mathbb{B}$ for the universe, which can be either $\{0,1\}$ or $\{\mathsf{T}, \mathsf{F}\}$

▷ **Definition 9.1.14** We call two expressions $e_1, e_2 \in E_{\text{bool}}$ equivalent (write $e_1 \equiv e_2$), iff $\mathcal{I}_\varphi(e_1) = \mathcal{I}_\varphi(e_2)$ for all φ.

▷ **Theorem 9.1.15** $e_1 \equiv e_2$, *iff* $\mathcal{I}_\varphi((\overline{e1} + e_2) * (e_1 + \overline{e_2})) = \mathsf{T}$ *for all variable assignments* φ.

As we are mainly interested in the interplay between form and meaning in Boolean Algebra, we will often identify Boolean expressions, if they have the same values in all situations (as specified by the variable assignments). The notion of equivalent formulae formalizes this intuition.

A better mouse-trap: Truth Tables

▷ Truth tables to visualize truth functions:

$\overline{\cdot}$	
T	F
F	T

$*$	T	F
T	T	F
F	F	F

$+$	T	F
T	T	T
F	T	F

▷ If we are interested in values for all assignments (e.g. of $x_{123} * x_4 + \overline{x_{123} * x_{72}}$)

assignments			intermediate results			full
x_4	x_{72}	x_{123}	$e_1 := x_{123} * x_{72}$	$e_2 := \overline{e_1}$	$e_3 := x_{123} * x_4$	$e_3 + e_2$
F	F	F	F	T	F	T
F	F	T	F	T	F	T
F	T	F	F	T	F	T
F	T	T	T	F	F	F
T	F	F	F	T	F	T
T	F	T	F	T	T	T
T	T	F	F	T	F	T
T	T	T	T	F	T	T

Boolean Equivalences

▷ Given $a, b, c \in E_{\text{bool}}$, $\circ \in \{+, *\}$, let $\hat{\circ} := \begin{cases} + & \text{if } \circ = * \\ * & \text{else} \end{cases}$

▷ We have the following equivalences in Boolean Algebra:

- ▷ $a \circ b \equiv b \circ a$ (commutativity)
- ▷ $(a \circ b) \circ c \equiv a \circ (b \circ c)$ (associativity)
- ▷ $a \circ (b \hat{\circ} c) \equiv (a \circ b) \hat{\circ} (a \circ c)$ (distributivity)
- ▷ $a \circ (a \hat{\circ} b) \equiv a$ (covering)
- ▷ $(a \circ b) \hat{\circ} (a \circ \overline{b}) \equiv a$ (combining)
- ▷ $(a \circ b) \hat{\circ} ((\overline{a} \circ c) \hat{\circ} (b \circ c)) \equiv (a \circ b) \hat{\circ} (\overline{a} \circ c)$ (consensus)
- ▷ $\overline{a \circ b} \equiv \overline{a} \hat{\circ} \overline{b}$ (De Morgan)

9.2 Boolean Functions

We will now turn to "semantical" counterparts of Boolean expressions: Boolean functions. These are just n-ary functions on the Boolean values.

Boolean functions are interesting, since can be used as computational devices; we will study this extensively in the rest of the course. In particular, we can consider a computer CPU as collection of Boolean functions (e.g. a modern CPU with 64 inputs and outputs can be viewed as a sequence of 64 Boolean functions of arity 64: one function per output pin).

The theory we will develop now will help us understand how to "implement" Boolean functions (as specifications of computer chips), viewing Boolean expressions very abstract representations of configurations of logic gates and wiring. We will study the issues of representing such configurations in more detail in a later part of the course.

Boolean Functions

▷ **Definition 9.2.1** A Boolean function is a function from $\mathbb{B}^n$ to $\mathbb{B}$.

▷ **Definition 9.2.2** Boolean functions $f, g\colon \mathbb{B}^n \to \mathbb{B}$ are called equivalent, (write $f \equiv g$), iff $f(c) = g(c)$ for all $c \in \mathbb{B}^n$. (equal as functions)

▷ Idea: We can turn any Boolean expression into a Boolean function by ordering the variables (use the lexical ordering on $\{X\} \times \{1, \ldots, 9\}^+ \times \{0, \ldots, 9\}^*$)

▷ **Definition 9.2.3** Let $e \in E_{\text{bool}}$ and $\{x_1, \ldots, x_n\}$ the set of variables in e, then call $VL(e) := \langle x_1, \ldots, x_n \rangle$ the variable list of e, iff $x_i \prec_{\text{lex}} x_j$ where $i \leq j$ and $\prec$ is the "numerical order" on variables.

▷ **Definition 9.2.4** Let $e \in E_{\text{bool}}$ with $VL(e) = \langle x_1, \ldots, x_n \rangle$, then we call the function

$$f_e\colon \mathbb{B}^n \to \mathbb{B} \text{ with } f_e\colon c \mapsto \mathcal{I}_{\varphi_c}(e)$$

the Boolean function induced by e, where $\varphi_{\langle c_1, \ldots, c_n \rangle}\colon x_i \mapsto c_i$. Dually, we say that e realizes f_e.

▷ **Theorem 9.2.5** *$e_1 \equiv e_2$, iff $f_{e_1} = f_{e_2}$.*

The definition above shows us that in theory every Boolean Expression induces a Boolean function. The simplest way to compute this is to compute the truth table for the expression and then read off the function from the table.

Boolean Functions and Truth Tables

▷ The truth table of a Boolean function is defined in the obvious way:

x_1	x_2	x_3	$f_{x_1 * (\overline{x_2} + x_3)}$
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

▷ compute this by assigning values and evaluating

- Question: can we also go the other way? (from function to expression?)
- Idea: read expression of a special form from truth tables (Boolean Polynomials)

Computing a Boolean expression from a given Boolean function is more interesting — there are many possible candidates to choose from; after all any two equivalent expressions induce the same function. To simplify the problem, we will restrict the space of Boolean expressions that realize a given Boolean function by looking only for expressions of a given form.

Boolean Polynomials

- special form Boolean Expressions
  - a literal is a variable or the negation of a variable
  - a monomial or product term is a literal or the product of literals
  - a clause or sum term is a literal or the sum of literals
  - a Boolean polynomial or sum of products is a product term or the sum of product terms
  - a clause set or product of sums is a sum term or the product of sum terms

  For literals x_i, write x_i^1, for $\overline{x_i}$ write x_i^0. (⚠ not exponentials, but intended truth values)
- **Notation 9.2.6** Write x_ix_j instead of $x_i * x_j$. (like in math)

Armed with this normal form, we can now define an way of realizing Boolean functions.

Normal Forms of Boolean Functions

- **Definition 9.2.7** Let $f\colon \mathbb{B}^n \to \mathbb{B}$ be a Boolean function and $c \in \mathbb{B}^n$, then $M_c := \prod_{j=1}^n x_j^{c_j}$ and $S_c := \sum_{j=1}^n x_j^{1-c_j}$
- **Definition 9.2.8** The disjunctive normal form (DNF) of f is $\sum_{c \in f^{-1}(1)} M_c$ (also called the canonical sum (written as $\mathrm{DNF}(f)$))
- **Definition 9.2.9** The conjunctive normal form (CNF) of f is $\prod_{c \in f^{-1}(0)} S_c$ (also called the canonical product (written as $\mathrm{CNF}(f)$))

x_1	x_2	x_3	f	monomials	clauses
0	0	0	1	$x_1^0\,x_2^0\,x_3^0$	
0	0	1	1	$x_1^0\,x_2^0\,x_3^1$	
0	1	0	0		$x_1^1 + x_2^0 + x_3^1$
0	1	1	0		$x_1^1 + x_2^0 + x_3^0$
1	0	0	1	$x_1^1\,x_2^0\,x_3^0$	
1	0	1	1	$x_1^1\,x_2^0\,x_3^1$	
1	1	0	0		$x_1^0 + x_2^0 + x_3^1$
1	1	1	1	$x_1^1\,x_2^1\,x_3^1$	

▷ DNF of f: $\overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}\,\overline{x_2}\,x_3 + x_1\,\overline{x_2}\,\overline{x_3} + x_1\,\overline{x_2}\,x_3 + x_1\,x_2\,x_3$

▷ CNF of f: $(x_1 + \overline{x_2} + x_3)\,(x_1 + \overline{x_2} + \overline{x_3})\,(\overline{x_1} + \overline{x_2} + x_3)$

In the light of the argument of understanding Boolean expressions as implementations of Boolean functions, the process becomes interesting while realizing specifications of chips. In particular it also becomes interesting, which of the possible Boolean expressions we choose for realizing a given Boolean function. We will analyze the choice in terms of the "cost" of a Boolean expression.

Costs of Boolean Expressions

▷ Idea: Complexity Analysis is about the estimation of resource needs

- ▷ if we have two expressions for a Boolean function, which one to choose?

▷ Idea: Let us just measure the size of the expression (after all it needs to be written down)

▷ Better Idea: count the number of operators (computation elements)

▷ **Definition 9.2.10** The cost $C(e)$ of $e \in E_{\text{bool}}$ is the number of operators in e.

▷ **Example 9.2.11** $C(\overline{x_1} + x_3) = 2$, $C(\overline{\overline{x_1} * \overline{x_2}} + x_3 * x_4) = 4$, $C((x_1 + x_2) + (\overline{\overline{x_1} * x_2} + x_3 * x_4)) = 7$

▷ **Definition 9.2.12** Let $f\colon \mathbb{B}^n \to \mathbb{B}$ be a Boolean function, then $C(f) := \min\{C(e) \mid f = f_e\}$ is the cost of f.

▷ Note: We can find expressions of arbitrarily high cost for a given Boolean function. ($e \equiv e * 1$)

▷ but how to find such an e with minimal cost for f?

9.3 Complexity Analysis for Boolean Expressions

We have learned that we can always realize a Boolean function as a Boolean Expression – which can ultimately serve as a blueprint for a realization in a circuit. But to get a full understanding of the problem, we should be able to predict the cost/size of the realizing Boolean expressions.

In principle, knowing how to realize a Boolean function would be sufficient to start a business as a chip manufacturer: clients come with a Boolean function, and the manufacturer computes the Boolean expression and produces the chip. But to be able to quote a price for the chip, the manufacturer needs to be able to predict the size of the Boolean expression *before* it is actually computed. Thus the need for prediction.

Before we turn to the prediction of cost/size for Boolean expressions, we will introduce the basic tools for this kind of analysis: A way of even expressing the results.

9.3.1 The Mathematics of Complexity

We will now introduce some very basic concepts of computational complexity theory – the discipline of classifying algorithms and the computational problems they solve into classes of inherent difficulty. In this classification, we are interested only in properties that are inherent to the problem or algorithm, not in the specific hardware or realization. This requires a special vocabulary which we will now introduce.

We want to classify the resource consumption of a computational process or problem in terms of the size of the "size" of the problem description. Thus we use functions from $\mathbb{N}$ (the problem size) to $\mathbb{N}$ (the resources consumed).

Example 9.3.1 (Mazes) We have already looked at different algorithms for constructing mazes in Example 2.2.14. There the problem size was the number of cells in the maze and the resource was the runtime in micro-seconds. The simple computation there shows that different growth patterns have great effects on computation time.

Note that the actual resource consumption function of actual computations depends very much on external factors that have nothing to do with the algorithm itself, e.g. the startup time or memory space of the runtime environment, garbage collection cycles, memory size that forces swapping, etc, and of course the processor speed itself. Therefore we want to abstract away from such factors in disregarding constant factors (processor speed) and startup effects.

Finally, we are always interested in the "worst-case behavior" – i.e. the maximal resource consumption over all possible inputs for the algorithm. But – as this may be difficult to predict, we are interested in upper and lower bounds of that.

All of these considerations lead to the notion of Landau sets we introduce now

The Landau Notations (aka. "big-O" Notation)

- ▷ **Definition 9.3.2** Let $f, g\colon \mathbb{N} \to \mathbb{N}$, we say that f is asymptotically bounded by g, written as $(f \leq_a g)$, iff there is an $n_0 \in \mathbb{N}$, such that $f(n) \leq g(n)$ for all $n > n_0$.
- ▷ **Definition 9.3.3** The three Landau sets $O(g), \Omega(g), \Theta(g)$ are defined as
  - ▹ $O(g) = \{f \mid \exists k>0. f \leq_a k \cdot g\}$
  - ▹ $\Omega(g) = \{f \mid \exists k>0. f \geq_a k \cdot g\}$
  - ▹ $\Theta(g) = O(g) \cap \Omega(g)$

  Intuition: The Landau sets express the "shape of growth" of the graph of a function.
- ▷ ▹ If $f \in O(g)$, then f grows at most as fast as g. ("f is in the order of g")
  - ▹ If $f \in \Omega(g)$, then f grows at least as fast as g. ("f is at least in the order of g")
  - ▹ If $f \in \Theta(g)$, then f grows as fast as g. ("f is strictly in the order of g")
- ▷ **Notation 9.3.4** (⚠) We often see $f = O(g)$ as a statement of complexity; this is a funny notation for $n \in fO(g)$!

The practical power of Landau sets as a tool for classifying resource consumption of algorithms and computational problems comes from the way we can calculate with them.

Computing with Landau Sets

▷ **Lemma 9.3.5** *We have the following computation rules for Landau sets:*

- ▹ *If $k \neq 0$ and $f \in O(g)$, then $(k\ f) \in O(g)$.*
- ▹ *If $f_i \in O(g_i)$, then $(f_1 + f_2) \in O(g_1 + g_2)$*
- ▹ *If $f_i \in O(g_i)$, then $(f_1\ f_2) \in O(g_1\ g_2)$*

▷ **Notation 9.3.6** If e is an expression in n, we write $O(e)$ for $O(\lambda n.e)$ (for Ω/Θ too)

Idea: the fastest growth function in sum determines the O-class

▷ **Example 9.3.7** $(\lambda n.263748) \in O(1)$

▷ **Example 9.3.8** $(\lambda n.26n + 372) \in O(n)$

▷ **Example 9.3.9** $(\lambda n.857n^{10} + 7342n^7 + 26n^2 + 902) \in O(n^{10})$

▷ **Example 9.3.10** $(\lambda n.3 \cdot 2^n + 72) \in O(2^n)$

▷ **Example 9.3.11** $(\lambda n.3 \cdot 2^n + 7342n^7 + 26n^2 + 722) \in O(2^n)$

Note that the "arithmetic operations" in Lemma 9.3.5 are applied to functions as "pointwise" operations.
A useful consequence of the addition rule is the following:

Lemma 9.3.12 *If $f_i \in O(g)$, then $(f_1 + f_2) \in O(g)$*

The examples show that the computation rules from Lemma 9.3.5 allow us to classify complex functions by dropping lower-growth summands and scalar factors. They also show us that we only need to consider very simple representatives for the Landau sets.

Commonly used Landau Sets

▷

Landau set	class name	rank	Landau set	class name	rank
$O(1)$	constant	1	$O(n^2)$	quadratic	4
$O(\log_2(n))$	logarithmic	2	$O(n^k)$	polynomial	5
$O(n)$	linear	3	$O(k^n)$	exponential	6

▷ **Theorem 9.3.13** *These Ω-classes establish a ranking (increasing rank $\leadsto$ increasing growth)*

$$O(1) \subset O(\log_2(n)) \subset O(n) \subset O(n^2) \subset O(n^{k'}) \subset O(k^n)$$

where $k'>2$ and $k>1$. The reverse holds for the Ω-classes

$$\Omega(n)1 \supset \Omega(n)\log_2(n) \supset \Omega(n)n \supset \Omega(n)n^2 \supset \Omega(n)n^{k'} \supset \Omega(n)k^n$$

▷ Idea: Use O-classes for worst-case complexity analysis and Ω-classes for best-case.

With the basics of complexity theory well-understood, we can now analyze the cost-complexity of Boolean expressions that realize Boolean functions.

9.3.2 Asymptotic Bounds for Costs of Boolean Expressions

In this Subsection we will derive some results on the (cost)-complexity of realizing a Boolean function as a Boolean expression. The first result is a very naive counting argument based on the fact that we can always realize a Boolean function via its DNF or CNF. The second result gives us a better complexity with a more involved argument. Another difference between the proofs is that the first one is constructive, i.e. we can read an algorithm that provides Boolean expressions of the complexity claimed by the algorithm for a given Boolean function. The second proof gives us no such algorithm, since it is non-constructive. Finally, we derive a lower bound of the worst-case complexity that shows that the upper bound from the second proof is almost tight.

An Upper Bound for the Cost of BF with n variables

▷ Idea: Every Boolean function has a DNF and CNF, so we compute its cost.

▷ **Example 9.3.14** Let us look at the size of the DNF or CNF for $f : \mathbb{B}^3 \to \mathbb{B}$.

x_1	x_2	x_3	f	monomials	clauses
0	0	0	1	$x_1^0\,x_2^0\,x_3^0$	
0	0	1	1	$x_1^0\,x_2^0\,x_3^1$	
0	1	0	0		$x_1^1 + x_2^0 + x_3^1$
0	1	1	0		$x_1^1 + x_2^0 + x_3^0$
1	0	0	1	$x_1^1\,x_2^0\,x_3^0$	
1	0	1	1	$x_1^1\,x_2^0\,x_3^1$	
1	1	0	0		$x_1^0 + x_2^0 + x_3^1$
1	1	1	1	$x_1^1\,x_2^1\,x_3^1$	

▷ **Theorem 9.3.15** *Any $f : \mathbb{B}^n \to \mathbb{B}$ is realized by an $e \in E_{\mathrm{bool}}$ with $C(e) \in O(n \cdot 2^n)$.*

Proof: by counting (constructive proof (we exhibit a witness))

▷ **P.1** either $e_n := \mathrm{CNF}(f)$ has $\frac{2^n}{2}$ clauses or less or $\mathrm{DNF}(f)$ does monomials

take smaller one, multiply/sum the monomials/clauses at cost $2^{n-1} - 1$

there are n literals per clause/monomial e_i, so $C(e_i) \leq 2n - 1$

so $C(e_n) \leq 2^{n-1} - 1 + 2^{n-1} \cdot (2n - 1)$ and thus $C(e_n) \in O(n \cdot 2^n)$ □

For this proof we will introduce the concept of a "realization cost function" $\kappa : \mathbb{N} \to \mathbb{N}$ to save space in the argumentation. The trick in this proof is to make the induction on the arity work by splitting an n-ary Boolean function into two $n-1$-ary functions and estimate their complexity separately. This argument does not give a direct witness in the proof, since to do this we have to decide which of these two split-parts we need to pursue at each level. This yields an algorithm for determining a witness, but not a direct witness itself.

P.2 P.3 P.4

We can do better (if we accept complicated witness)

▷ **Theorem 9.3.16** *Let $\kappa(n) :=$ max$\{C(f) \mid f : \mathbb{B}^n \to \mathbb{B}\}$, then $\kappa \in O(2^n)$.*

▷ Proof: we show that $\kappa(n) \leq 2^{n+1}$ by induction on n

P.1.1 base case ($n = 1$): We count the operators in all members: $\mathbb{B} \to \mathbb{B} = \{f_1, f_0, f_{x_1}, f_{\overline{x_1}}\}$, so $\kappa(1) = 1$ and thus $\kappa(1) \leq 2^2$.

P.1.2 step case ($n>1$):

P.1.2.1 given $f : \mathbb{B}^n \to \mathbb{B}$, then $f(a_1, \ldots, a_n) = 1$, iff either

- ▷ $a_n = 0$ and $f(a_1, \ldots, a_{n-1}, 0) = 1$ or
- ▷ $a_n = 1$ and $f(a_1, \ldots, a_{n-1}, 1) = 1$

P.1.2.2 Let $f_i(a_1, \ldots, a_{n-1}) := f(a_1, \ldots, a_{n-1}, i)$ for $i \in \{0, 1\}$,

P.1.2.3 then there are $e_i \in E_{\text{bool}}$, such that $f_i = f_{e_i}$ and $C(e_i) = 2^n$. (IH)

P.1.2.4 thus $f = f_e$, where $e := \overline{x_n} * e_0 + x_n * e_1$ and $\kappa(n) = 2 \cdot 2^n + 4 \leq 2^{n+1}$ as $2 \leq n$. □

□

The next proof is quite a lot of work, so we will first sketch the overall structure of the proof, before we look into the details. The main idea is to estimate a cleverly chosen quantity from above and below, to get an inequality between the lower and upper bounds (the quantity itself is irrelevant except to make the proof work).

A Lower Bound for the Cost of BF with n Variables

▷ **Theorem 9.3.17** $\kappa \in \Omega(\frac{2^n}{\log_2(n)})$

▷ Proof: Sketch (counting again!)

P.1 the cost of a function is based on the cost of expressions.

P.2 consider the set $\mathcal{E}_n$ of expressions with n variables of cost no more than $\kappa(n)$.

P.3 find an upper and lower bound for $\#(\mathcal{E}_n)$: $\Phi(n) \leq \#(\mathcal{E}_n) \leq \Psi(\kappa(n))$

P.4 in particular: $\Phi(n) \leq \Psi(\kappa(n))$

P.5 solving for $\kappa(n)$ yields $\kappa(n) \geq \Xi(n)$ so $\kappa \in \Omega(\frac{2^n}{\log_2(n)})$ □

▷ We will expand P.3 and P.5 in the next slides

A Lower Bound For $\kappa(n)$-Cost Expressions

▷ **Definition 9.3.18** $\mathcal{E}_n := \{e \in E_{\text{bool}} \mid e$ has n variables and $C(e) \leq \kappa(n)\}$

▷ **Lemma 9.3.19** $\#(\mathcal{E}_n) \geq \#(\mathbb{B}^n \to \mathbb{B})$

▷ Proof:

P.1 For all $f_n \in \mathbb{B}^n \to \mathbb{B}$ we have $C(f_n) \leq \kappa(n)$

P.2 $C(f_n) = \min\{C(e) \mid f_e = f_n\}$ choose e_{f_n} with $C(e_{f_n}) = C(f_n)$

P.3 all distinct: if $e_g \equiv e_h$, then $f_{e_g} = f_{e_h}$ and thus $g = h$. □

▷ **Corollary 9.3.20** $\#(\mathcal{E}_n) \geq 2^{2^n}$

Proof: consider the n dimensional truth tables

▷ **P.1** 2^n entries that can be either 0 or 1, so 2^{2^n} possibilities

so $\#(\mathbb{B}^n \to \mathbb{B}) = 2^{2^n}$ □

P.2

An Upper Bound For $\kappa(n)$-cost Expressions

▷ Idea: Estimate the number of E_{bool} strings that can be formed at a given cost by looking at the length and alphabet size.

▷ **Definition 9.3.21** Given a cost c let $\Lambda(e)$ be the length of e considering variables as single characters. We define

$$\sigma(c) := \max\{\Lambda(e) \mid e \in E_{\text{bool}} \wedge (C(e) \leq c)\}$$

▷ **Lemma 9.3.22** $\sigma(n) \leq 5n$ *for* $n > 0$.

▷ Proof: by induction on n

P.1.1 base case: The cost 1 expressions are of the form $(v \circ w)$ and $(-v)$, where v and w are variables. So the length is at most 5.

P.1.2 step case: $\sigma(n) = \Lambda((e_1 \circ e_2)) = \Lambda(e_1) + \Lambda(e_2) + 3$, where $C(e_1) + C(e_2) \leq n - 1$. so $\sigma(n) \leq \sigma(i) + \sigma(j) + 3 \leq 5 \cdot C(e_1) + 5 \cdot C(e_2) + 3 \leq 5 \cdot n - 1 + 5 = 5n$ □

▷ **Corollary 9.3.23** *max*$\{\Lambda(e) \mid e \in \mathcal{E}_n\} \leq 5 \cdot \kappa(n)$

An Upper Bound For $\kappa(n)$-cost Expressions

▷ Idea: $e \in \mathcal{E}_n$ has at most n variables by definition.

▷ Let $\mathcal{A}_n := \{x_1, \ldots, x_n, 0, 1, *, +, -, (,)\}$, then $\#(\mathcal{A}_n) = n + 7$

▷ **Corollary 9.3.24** $\mathcal{E}_n \subseteq \bigcup_{i=0}^{5\kappa(n)} {\mathcal{A}_n}^i$ *and* $\#(\mathcal{E}_n) \leq \frac{(n+7)^{5\kappa(n)+1} - 1}{n+7}$

▷ Proof Sketch: Note that the $\mathcal{A}_j$ are disjoint for distinct n, so

$$\#\left(\bigcup_{i=0}^{5\kappa(n)} {\mathcal{A}_n}^i\right) = \sum_{i=0}^{5\kappa(n)} \#({\mathcal{A}_n}^i) = \sum_{i=0}^{5\kappa(n)} \#({\mathcal{A}_n}^i) = \sum_{i=0}^{5\kappa(n)} (n+7)^i = \frac{(n+7)^{5\kappa(n)+1} - 1}{n+6}$$

$\square$

Solving for $\kappa(n)$

▷ $\frac{(n+7)^{5\kappa(n)+1}-1}{n+6} \geq 2^{2^n}$

▷ $(n+7)^{5\kappa(n)+1} \geq 2^{2^n}$ (as $(n+7)^{5\kappa(n)+1} \geq \frac{(n+7)^{5\kappa(n)+1}-1}{n+6}$)

▷ $5\kappa(n)+1 \cdot \log_2(n+7) \geq 2^n$ (as $\log_a(x) = \log_b(x) \cdot \log_a(b)$)

▷ $5\kappa(n)+1 \geq \frac{2^n}{\log_2(n+7)}$

▷ $\kappa(n) \geq 1/5 \cdot \frac{2^n}{\log_2(n+7)} - 1$

▷ $\kappa(n) \in \Omega(\frac{2^n}{\log_2(n)})$

9.4 The Quine-McCluskey Algorithm

After we have studied the worst-case complexity of Boolean expressions that realize given Boolean functions, let us return to the question of computing realizing Boolean expressions in practice. We will again restrict ourselves to the subclass of Boolean polynomials, but this time, we make sure that we find the optimal representatives in this class.

The first step in the endeavor of finding minimal polynomials for a given Boolean function is to optimize monomials for this task. We have two concerns here. We are interested in monomials that contribute to realizing a given Boolean function f (we say they imply f or are implicants), and we are interested in the cheapest among those that do. For the latter we have to look at a way to make monomials cheaper, and come up with the notion of a sub-monomial, i.e. a monomial that only contains a subset of literals (and is thus cheaper.)

Constructing Minimal Polynomials: Prime Implicants

▷ **Definition 9.4.1** We will use the following ordering on $\mathbb{B}$: $\mathsf{F} \leq \mathsf{T}$(remember $0 \leq 1$)

and say that that a monomial M' dominates a monomial M, iff $f_M(c) \leq f_{M'}(c)$ for all $c \in \mathbb{B}^n$. (write $M \leq M'$)

▷ **Definition 9.4.2** A monomial M implies a Boolean function $f\colon \mathbb{B}^n \to \mathbb{B}$ (M is an implicant of f; write $M \succ f$), iff $f_M(c) \leq f(c)$ for all $c \in \mathbb{B}^n$.

▷ **Definition 9.4.3** Let $M = L_1 \cdots L_n$ and $M' = L'_1 \cdots L'_{n'}$ be monomials, then M' is called a sub-monomial of M (write $M' \subset M$), iff $M' = 1$ or

- ▷ for all $j \leq n'$, there is an $i \leq n$, such that $L'_j = L_i$ and
- ▷ there is an $i \leq n$, such that $L_i \neq L'_j$ for all $j \leq n$

In other words: M is a sub-monomial of M', iff the literals of M are a proper subset of the literals of M'.

With these definitions, we can convince ourselves that sub-monomials dominate their super-monomials. Intuitively, a monomial is a conjunction of conditions that are needed to make the Boolean function f true; if we have fewer of them, then f becomes "less true". In particular, if we have too few of them, then we cannot approximate the truth-conditions of f sufficiently. So we will look for monomials that approximate f well enough and are shortest with this property: the prime implicants of f.

▷ Constructing Minimal Polynomials: Prime Implicants

▷ **Lemma 9.4.4** *If $M' \subset M$, then M' dominates M.*

▷ Proof:

P.1 Given $c \in \mathbb{B}^n$ with $f_M(c) = \mathsf{T}$, we have, $f_{L_i}(c) = \mathsf{T}$ for all literals in M.

P.2 As M' is a sub-monomial of M, then $f_{L'_j}(c) = \mathsf{T}$ for each literal L'_j of M'.

P.3 Therefore, $f_{M'}(c) = \mathsf{T}$. □

▷ **Definition 9.4.5** An implicant M of f is a prime implicant of f iff no sub-monomial of M is an implicant of f.

The following Theorem verifies our intuition that prime implicants are good candidates for constructing minimal polynomials for a given Boolean function. The proof is rather simple (if notationally loaded). We just assume the contrary, i.e. that there is a minimal polynomial p that contains a non-prime-implicant monomial M_k, then we can decrease the cost of the of p while still inducing the given function f. So p was not minimal which shows the assertion.

Prime Implicants and Costs

▷ **Theorem 9.4.6** *Given a Boolean function $f \neq \lambda x.\mathsf{F}$ and a Boolean polynomial $f_p \equiv f$ with minimal cost, i.e., there is no other polynomial $p' \equiv p$ such that $C(p') < C(p)$. Then, p solely consists of prime implicants of f.*

▷ Proof: The theorem obviously holds for $f = \lambda x.\mathsf{T}$.

P.1 For other f, we have $f \equiv f_p$ where $p := \sum_{i=1}^{n} M_i$ for some $n \geq 1$ monomials M_i.

P.2 Now, suppose that M_i is not a prime implicant of f, i.e., $M' \succ f$ for some $M' \subset M_k$ with $k < i$.

P.3 Let us substitute M_k by M': $p' := \sum_{i=1}^{k-1} M_i + M' + \sum_{i=k+1}^{n} M_i$

P.4 We have $C(M') < C(M_k)$ and thus $C(p') < C(p)$ (def of sub-monomial)

P.5 Furthermore $M_k \leq M'$ and hence that $p \leq p'$ by Lemma 9.4.4.

P.6 In addition, $M' \leq p$ as $M' \succ f$ and $f = p$.

P.7 similarly: $M_i \leq p$ for all M_i. Hence, $p' \leq p$.

P.8 So $p' \equiv p$ and $f_p \equiv f$. Therefore, p is not a minimal polynomial. □

This theorem directly suggests a simple generate-and-test algorithm to construct minimal polynomials. We will however improve on this using an idea by Quine and McCluskey. There are of course better algorithms nowadays, but this one serves as a nice example of how to get from a theoretical insight to a practical algorithm.

The Quine/McCluskey Algorithm (Idea)

- ▷ Idea: use Theorem 9.4.6 to search for minimal-cost polynomials
  - ▷ Determine all prime implicants (sub-algorithm QMC_1)
  - ▷ choose the minimal subset that covers f (sub-algorithm QMC_2)
- ▷ Idea: To obtain prime implicants,
  - ▷ start with the DNF monomials (they are implicants by construction)
  - ▷ find submonomials that are still implicants of f.
- ▷ Idea: Look at polynomials of the form $p := mx_i + m\overline{x_i}$ (note: $p \equiv m$)

Armed with the knowledge that minimal polynomials must consist entirely of prime implicants, we can build a practical algorithm for computing minimal polynomials: In a first step we compute the set of prime implicants of a given function, and later we see whether we actually need all of them.

For the first step we use an important observation: for a given monomial m, the polynomials $m\,x + m\,\overline{x}$ are equivalent, and in particular, we can obtain an equivalent polynomial by replace the latter (the partners) by the former (the resolvent). That gives the main idea behind the first part of the Quine-McCluskey algorithm. Given a Boolean function f, we start with a polynomial for f: the disjunctive normal form, and then replace partners by resolvents, until that is impossible.

The algorithm QMC_1, for determining Prime Implicants

- ▷ **Definition 9.4.7** Let M be a set of monomials, then
  - ▷ $\mathcal{R}(M) := \{m \,|\, (m\,x) \in M \wedge (m\,\overline{x}) \in M\}$ is called the set of resolvents of M
  - ▷ $\widehat{\mathcal{R}}(M) := \{m \in M \,|\, m \text{ has a partner in } M\}$ ($n\,\overline{x_i}$ and $n\,x_i$ are partners)
- ▷ **Definition 9.4.8 (Algorithm)** Given $f \colon \mathbb{B}^n \to \mathbb{B}$
  - ▷ let $M_0 := \mathrm{DNF}(f)$ and for all $j > 0$ compute (DNF as set of monomials)
    - ▷ $M_j := \mathcal{R}(M_{j-1})$ (resolve to get sub-monomials)
    - ▷ $P_j := M_{j-1} \backslash \widehat{\mathcal{R}}(M_{j-1})$ (get rid of redundant resolution partners)
  - ▷ terminate when $M_j = \emptyset$, return $P_{\mathrm{prime}} := \bigcup_{j=1}^{n} P_j$

We will look at a simple example to fortify our intuition.

Example for QMC_1

x1	x2	x3	f	monomials
F	F	F	T	$x1^0\,x2^0\,x3^0$
F	F	T	T	$x1^0\,x2^0\,x3^1$
F	T	F	F	
F	T	T	F	
T	F	F	T	$x1^1\,x2^0\,x3^0$
T	F	T	T	$x1^1\,x2^0\,x3^1$
T	T	F	F	
T	T	T	T	$x1^1\,x2^1\,x3^1$

$$P_{prime} = \bigcup_{j=1}^{3} P_j = \{x1\,x3, \overline{x2}\}$$

$$\begin{aligned}
M_0 &= \{\underbrace{\overline{x1}\,\overline{x2}\,\overline{x3}}_{=:\,e_1^0}, \underbrace{\overline{x1}\,\overline{x2}\,x3}_{=:\,e_2^0}, \underbrace{x1\,\overline{x2}\,\overline{x3}}_{=:\,e_3^0}, \underbrace{x1\,\overline{x2}\,x3}_{=:\,e_4^0}, \underbrace{x1\,x2\,x3}_{=:\,e_5^0}\} \\
M_1 &= \{ \underbrace{\overline{x1}\,\overline{x2}}_{\substack{\mathcal{R}(e_1^0,e_2^0)\\ =:\,e_1^1}} , \underbrace{\overline{x2}\,\overline{x3}}_{\substack{\mathcal{R}(e_1^0,e_3^0)\\ =:\,e_2^1}} , \underbrace{\overline{x2}\,x3}_{\substack{\mathcal{R}(e_2^0,e_4^0)\\ =:\,e_3^1}} , \underbrace{x1\,\overline{x2}}_{\substack{\mathcal{R}(e_3^0,e_4^0)\\ =:\,e_4^1}} , \underbrace{x1\,x3}_{\substack{\mathcal{R}(e_4^0,e_5^0)\\ =:\,e_5^1}} \} \\
P_1 &= \emptyset \\
M_2 &= \{ \underbrace{\overline{x2}}_{\mathcal{R}(e_1^1,e_4^1)} , \underbrace{\overline{x2}}_{\mathcal{R}(e_2^1,e_3^1)} \} \\
P_2 &= \{x1\,x3\} \\
M_3 &= \emptyset \\
P_3 &= \{\overline{x2}\}
\end{aligned}$$

▷ But: even though the minimal polynomial only consists of prime implicants, it need not contain all of them

We now verify that the algorithm really computes what we want: all prime implicants of the Boolean function we have given it. This involves a somewhat technical proof of the assertion below. But we are mainly interested in the direct consequences here.

Properties of QMC_1

▷ **Lemma 9.4.9** *(proof by simple (mutual) induction)*

1) all monomials in M_j have exactly $n - j$ literals.

2) M_j contains the implicants of f with $n - j$ literals.

3) P_j contains the prime implicants of f with $n - j + 1$ for $j > 0$. literals

▷ **Corollary 9.4.10** QMC_1 *terminates after at most n rounds.*

▷ **Corollary 9.4.11** P_{prime} *is the set of all prime implicants of f.*

Note that we are not finished with our task yet. We have computed all prime implicants of a given Boolean function, but some of them might be un-necessary in the minimal polynomial. So we have to determine which ones are. We will first look at the simple brute force method of finding the minimal polynomial: we just build all combinations and test whether they induce the right Boolean function. Such algorithms are usually called generate-and-test algorithms.

They are usually simplest, but not the best algorithms for a given computational problem. This is also the case here, so we will present a better algorithm below.

Algorithm QMC_2: Minimize Prime Implicants Polynomial

- **Definition 9.4.12 (Algorithm)** Generate and test!
  - enumerate $S_p \subseteq P_{prime}$, i.e., all possible combinations of prime implicants of f,
  - form a polynomial e_p as the sum over S_p and test whether $f_{e_p} = f$ and the cost of e_p is minimal
- **Example 9.4.13** $P_{prime} = \{x1\,x3, \overline{x2}\}$, so $e_p \in \{1, x1\,x3, \overline{x2}, x1\,x3 + \overline{x2}\}$.
- Only $f_{x1\,x3+\overline{x2}} \equiv f$, so $x1\,x3 + \overline{x2}$ is the minimal polynomial
- Complaint: The set of combinations (power set) grows exponentially

A better Mouse-trap for QMC_2: The Prime Implicant Table

- **Definition 9.4.14** Let $f\colon \mathbb{B}^n \to \mathbb{B}$ be a Boolean function, then the PIT consists of
  - a left hand column with all prime implicants p_i of f
  - a top row with all vectors $x \in \mathbb{B}^n$ with $f(x) = \mathsf{T}$
  - a central matrix of all $f_{p_i}(x)$
- **Example 9.4.15**

	FFF	FFT	TFF	TFT	TTT
$x1\,x3$	F	F	F	T	T
$\overline{x2}$	T	T	T	T	F

- **Definition 9.4.16** A prime implicant p is essential for f iff
  - there is a $c \in \mathbb{B}^n$ such that $f_p(c) = \mathsf{T}$ and
  - $f_q(c) = \mathsf{F}$ for all other prime implicants q.

Note: A prime implicant is essential, iff there is a column in the PIT, where it has a T and all others have F.

Essential Prime Implicants and Minimal Polynomials

- **Theorem 9.4.17** *Let $f\colon \mathbb{B}^n \to \mathbb{B}$ be a Boolean function, p an essential prime implicant for f, and p_{min} a minimal polynomial for f, then $p \in p_{min}$.*
- Proof: by contradiction: let $p \notin p_{min}$

P.1 We know that $f = f_{p_{min}}$ and $p_{min} = \sum_{j=1}^{n} p_j$ for some $n \in \mathbb{N}$ and prime implicants p_j.

P.2 so for all $c \in \mathbb{B}^n$ with $f(c) = \mathsf{T}$ there is a $j \leq n$ with $f_{p_j}(c) = \mathsf{T}$.

P.3 so p cannot be essential □

Let us now apply the optimized algorithm to a slightly bigger example.

A complex Example for QMC (Function and DNF)

x1	x2	x3	x4	f	monomials
F	F	F	F	T	$x1^0\,x2^0\,x3^0\,x4^0$
F	F	F	T	T	$x1^0\,x2^0\,x3^0\,x4^1$
F	F	T	F	T	$x1^0\,x2^0\,x3^1\,x4^0$
F	F	T	T	F	
F	T	F	F	F	
F	T	F	T	T	$x1^0\,x2^1\,x3^0\,x4^1$
F	T	T	F	F	
F	T	T	T	F	
T	F	F	F	F	
T	F	F	T	F	
T	F	T	F	T	$x1^1\,x2^0\,x3^1\,x4^0$
T	F	T	T	T	$x1^1\,x2^0\,x3^1\,x4^1$
T	T	F	F	F	
T	T	F	T	F	
T	T	T	F	T	$x1^1\,x2^1\,x3^1\,x4^0$
T	T	T	T	T	$x1^1\,x2^1\,x3^1\,x4^1$

A complex Example for QMC (QMC_1)

$$
\begin{aligned}
M_0 &= \{x1^0\,x2^0\,x3^0\,x4^0, x1^0\,x2^0\,x3^0\,x4^1, x1^0\,x2^0\,x3^1\,x4^0,\\
&\quad x1^0\,x2^1\,x3^0\,x4^1, x1^1\,x2^0\,x3^1\,x4^0, x1^1\,x2^0\,x3^1\,x4^1,\\
&\quad x1^1\,x2^1\,x3^1\,x4^0, x1^1\,x2^1\,x3^1\,x4^1\}\\[1ex]
M_1 &= \{x1^0\,x2^0\,x3^0, x1^0\,x2^0\,x4^0, x1^0\,x3^0\,x4^1, x1^1\,x2^0\,x3^1,\\
&\quad x1^1\,x2^1\,x3^1, x1^1\,x3^1\,x4^1, x2^0\,x3^1\,x4^0, x1^1\,x3^1\,x4^0\}\\
P_1 &= \emptyset\\[1ex]
M_2 &= \{x1^1\,x3^1\}\\
P_2 &= \{x1^0\,x2^0\,x3^0, x1^0\,x2^0\,x4^0, x1^0\,x3^0\,x4^1, x2^0\,x3^1\,x4^0\}\\[1ex]
M_3 &= \emptyset\\
P_3 &= \{x1^1\,x3^1\}
\end{aligned}
$$

$$P_{\mathrm{prime}} = \{\overline{x1}\,\overline{x2}\,\overline{x3}, \overline{x1}\,\overline{x2}\,\overline{x4}, \overline{x1}\,\overline{x3}\,x4, \overline{x2}\,x3\,\overline{x4}, x1\,x3\}$$

A better Mouse-trap for QMC_1: optimizing the data structure

▷ Idea: Do the calculations directly on the DNF table

x1	x2	x3	x4	monomials
F	F	F	F	$x1^0\, x2^0\, x3^0\, x4^0$
F	F	F	T	$x1^0\, x2^0\, x3^0\, x4^1$
F	F	T	F	$x1^0\, x2^0\, x3^1\, x4^0$
F	T	F	T	$x1^0\, x2^1\, x3^0\, x4^1$
T	F	T	F	$x1^1\, x2^0\, x3^1\, x4^0$
T	F	T	T	$x1^1\, x2^0\, x3^1\, x4^1$
T	T	T	F	$x1^1\, x2^1\, x3^1\, x4^0$
T	T	T	T	$x1^1\, x2^1\, x3^1\, x4^1$

- ▷ Note: the monomials on the right hand side are only for illustration
- ▷ Idea: do the resolution directly on the left hand side
- ▷ Find rows that differ only by a single entry. (first two rows)
- ▷ resolve: replace them by one, where that entry has an X (canceled literal)
- ▷ **Example 9.4.18** ⟨F, F, F, F⟩ and ⟨F, F, F, T⟩ resolve to ⟨F, F, F, X⟩.

A better Mouse-trap for QMC_1: optimizing the data structure

- ▷ One step resolution on the table

x1	x2	x3	x4	monomials
F	F	F	F	$x1^0\, x2^0\, x3^0\, x4^0$
F	F	F	T	$x1^0\, x2^0\, x3^0\, x4^1$
F	F	T	F	$x1^0\, x2^0\, x3^1\, x4^0$
F	T	F	T	$x1^0\, x2^1\, x3^0\, x4^1$
T	F	T	F	$x1^1\, x2^0\, x3^1\, x4^0$
T	F	T	T	$x1^1\, x2^0\, x3^1\, x4^1$
T	T	T	F	$x1^1\, x2^1\, x3^1\, x4^0$
T	T	T	T	$x1^1\, x2^1\, x3^1\, x4^1$

$\rightsquigarrow$

x1	x2	x3	x4	monomials
F	F	F	X	$x1^0\, x2^0\, x3^0$
F	F	X	F	$x1^0\, x2^0\, x4^0$
F	X	F	T	$x1^0\, x3^0\, x4^1$
T	F	T	X	$x1^1\, x2^0\, x3^1$
T	T	T	X	$x1^1\, x2^1\, x3^1$
T	X	T	T	$x1^1\, x3^1\, x4^1$
X	F	T	F	$x2^0\, x3^1\, x4^0$
T	X	T	F	$x1^1\, x3^1\, x4^0$

- ▷ Repeat the process until no more progress can be made

x1	x2	x3	x4	monomials
F	F	F	X	$x1^0\, x2^0\, x3^0$
F	F	X	F	$x1^0\, x2^0\, x4^0$
F	X	F	T	$x1^0\, x3^0\, x4^1$
T	X	T	X	$x1^1\, x3^1$
X	F	T	F	$x2^0\, x3^1\, x4^0$

- ▷ This table represents the prime implicants of f

A complex Example for QMC_1

- ▷ The PIT:

	F F F F	F F F T	F F T F	F T F T	T F T F	T F T T	T T T F	T T T T
$\overline{x1}\,\overline{x2}\,\overline{x3}$	T	T	F	F	F	F	F	F
$\overline{x1}\,\overline{x2}\,\overline{x4}$	T	F	T	F	F	F	F	F
$\overline{x1}\,\overline{x3}\,x4$	F	T	F	T	F	F	F	F
$\overline{x2}\,x3\,\overline{x4}$	F	F	T	F	T	F	F	F
$x1\,x3$	F	F	F	F	T	T	T	T

- ▷ $\overline{x1}\,\overline{x2}\,\overline{x3}$ is not essential, so we are left with

	F F F F	F F F T	F F T F	F T F T	T F T F	T F T T	T T T F	T T T T
$\overline{x1}\,\overline{x2}\,\overline{x4}$	T	F	T	F	F	F	F	F
$\overline{x1}\,\overline{x3}\,x4$	F	T	F	T	F	F	F	F
$\overline{x2}\,x3\,\overline{x4}$	F	F	T	F	T	F	F	F
$x1\,x3$	F	F	F	F	T	T	T	T

▷ here $\overline{x2}, x3, \overline{x4}$ is not essential, so we are left with

	FFFF	FFFT	FFTF	FTFT	TFTF	TFTT	TTTF	TTTT
$\overline{x1}\,\overline{x2}\,\overline{x4}$	T	F	T	F	F	F	F	F
$\overline{x1}\,\overline{x3}\,x4$	F	T	F	T	F	F	F	F
$x1\,x3$	F	F	F	F	T	T	T	T

▷ all the remaining ones ($\overline{x1}\,\overline{x2}\,\overline{x4}$, $\overline{x1}\,\overline{x3}\,x4$, and $x1\,x3$) are essential

▷ So, the minimal polynomial of f is $\overline{x1}\,\overline{x2}\,\overline{x4} + \overline{x1}\,\overline{x3}\,x4 + x1\,x3$.

The Quine-McCluskey Algorithm (final version)

▷ We started from a simple idea suggested by Theorem 9.4.6

- ▷ Determine all prime implicants (sub-algorithm QMC_1)
- ▷ choose the minimal subset that covers f (sub-algorithm QMC_2)

and optimized the parts (data structures and partial algorithms) considerably.

▷ **Definition 9.4.19** The Quine-McCluskey algorihtm ()computes minimal polynomials for a given Boolean function f by computing the set of prime implicants and choosing a covering subset.

▷ Observation: Good algorithms are often based on mathematical insights (theorems), but math is not enough, considerable work goes into finding good representations (data structures) and clever sub-algorithms.

Chapter 10

Propositional Logic

10.1 Boolean Expressions and Propositional Logic

We will now look at Boolean expressions from a different angle. We use them to give us a very simple model of a representation language for

- knowledge — in our context mathematics, since it is so simple, and
- argumentation — i.e. the process of deriving new knowledge from older knowledge

Still another Notation for Boolean Expressions

▷ Idea: get closer to MathTalk

  ▷ Use $\vee$, $\wedge$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ directly (after all, we do in MathTalk)

  ▷ construct more complex names (propositions) for variables (Use ground terms of sort $\mathbb{B}$ in an ADT)

▷ **Definition 10.1.1** Let $\Sigma = \langle \mathcal{S}, \mathcal{D} \rangle$ be an abstract data type, such that $\mathbb{B} \in \mathcal{S}$ and

$$[\neg \colon \mathbb{B} \to \mathbb{B}], [\vee \colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}] \in \mathcal{D}$$

then we call the set $\mathcal{T}^g_{\mathbb{B}}(\Sigma)$ of ground Σ-terms of sort $\mathbb{B}$ a formulation of Propositional Logic.

▷ We will also call this formulation Predicate Logic without Quantifiers and denote it with PLNQ.

▷ **Definition 10.1.2** Call terms in $\mathcal{T}^g_{\mathbb{B}}(\Sigma)$ without $\vee$, $\wedge$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ atoms. (write $\mathcal{A}(\Sigma)$)

▷ Note: Formulae of propositional logic "are" Boolean Expressions

  ▷ replace $\mathbf{A} \Leftrightarrow \mathbf{B}$ by $(\mathbf{A} \Rightarrow \mathbf{B}) \wedge (\mathbf{B} \Rightarrow \mathbf{A})$ and $\mathbf{A} \Rightarrow \mathbf{B}$ by $\neg \mathbf{A} \vee \mathbf{B}$...

  ▷ Build print routine $\hat{\cdot}$ with $\widehat{\mathbf{A} \wedge \mathbf{B}} = \widehat{\mathbf{A}} * \widehat{\mathbf{B}}$, and $\widehat{\neg \mathbf{A}} = \overline{\widehat{\mathbf{A}}}$ and that turns atoms into variable names. (variables and atoms are countable)

Conventions for Brackets in Propositional Logic

- ▷ we leave out outer brackets: $\mathbf{A} \Rightarrow \mathbf{B}$ abbreviates $(\mathbf{A} \Rightarrow \mathbf{B})$.
- ▷ implications are right associative: $\mathbf{A}^1 \Rightarrow \cdots \Rightarrow \mathbf{A}^n \Rightarrow \mathbf{C}$ abbreviates $\mathbf{A}^1 \Rightarrow \cdots \Rightarrow \cdots \Rightarrow \mathbf{A}^n \Rightarrow \mathbf{C}$
- ▷ a **.** stands for a left bracket whose partner is as far right as is consistent with existing brackets $(\mathbf{A} \Rightarrow .\mathbf{C} \wedge \mathbf{D} = \mathbf{A} \Rightarrow (\mathbf{C} \wedge \mathbf{D}))$

We will now use the distribution of values of a Boolean expression under all (variable) assignments to characterize them semantically. The intuition here is that we want to understand theorems, examples, counterexamples, and inconsistencies in mathematics and everyday reasoning[1].

The idea is to use the formal language of Boolean expressions as a model for mathematical language. Of course, we cannot express all of mathematics as Boolean expressions, but we can at least study the interplay of mathematical statements (which can be true or false) with the copula "and", "or" and "not".

Semantic Properties of Boolean Expressions

- ▷ **Definition 10.1.3** Let $\mathcal{M} := \langle \mathcal{U}, \mathcal{I} \rangle$ be our model, then we call e
  - ▷ true under φ in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{T}$ (write $\mathcal{M} \models^\varphi e$)
  - ▷ false under φ in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{F}$ (write $\mathcal{M} \not\models^\varphi e$)
  - ▷ satisfiable in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{T}$ for some assignment φ
  - ▷ valid in $\mathcal{M}$, iff $\mathcal{M} \models^\varphi e$ for all assignments φ (write $\mathcal{M} \models e$)
  - ▷ falsifiable in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{F}$ for some assignments φ
  - ▷ unsatisfiable in $\mathcal{M}$, iff $\mathcal{I}_\varphi(e) = \mathsf{F}$ for all assignments φ
- ▷ **Example 10.1.4** $x \vee x$ is satisfiable and falsifiable.
- ▷ **Example 10.1.5** $x \vee \neg x$ is valid and $x \wedge \neg x$ is unsatisfiable.
- ▷ **Notation 10.1.6** (alternative) Write $[\![e]\!]^{\mathcal{M}}_\varphi$ for $\mathcal{I}_\varphi(e)$, if $\mathcal{M} = \langle \mathcal{U}, \mathcal{I} \rangle$. (and $[\![e]\!]^{\mathcal{M}}$, if e is ground, and $[\![e]\!]$, if $\mathcal{M}$ is clear)
- ▷ **Definition 10.1.7 (Entailment)** (aka. logical consequence)
  We say that e entails f ($e \models f$), iff $\mathcal{I}_\varphi(f) = \mathsf{T}$ for all φ with $\mathcal{I}_\varphi(e) = \mathsf{T}$ (i.e. all assignments that make e true also make f true)

Let us now see how these semantic properties model mathematical practice.

In mathematics we are interested in assertions that are true in all circumstances. In our model of mathematics, we use variable assignments to stand for circumstances. So we are interested in Boolean expressions which are true under all variable assignments; we call them valid. We often give examples (or show situations) which make a conjectured assertion false; we call such examples counterexamples, and such assertions "falsifiable". We also often give examples for certain

[1]Here (and elsewhere) we will use mathematics (and the language of mathematics) as a test tube for understanding reasoning, since mathematics has a long history of studying its own reasoning processes and assumptions.

assertions to show that they can indeed be made true (which is not the same as being valid yet); such assertions we call "satisfiable". Finally, if an assertion cannot be made true in any circumstances we call it "unsatisfiable"; such assertions naturally arise in mathematical practice in the form of refutation proofs, where we show that an assertion (usually the negation of the theorem we want to prove) leads to an obviously unsatisfiable conclusion, showing that the negation of the theorem is unsatisfiable, and thus the theorem valid.

Example: Propositional Logic with ADT variables

▷ Idea: We use propositional logic to express things about the world (PLNQ $\hat{=}$ Predicate Logic without Quantifiers)

▷ **Example 10.1.8** $\mathcal{A} := \langle\{\mathbb{B},\mathbb{I}\},\{\ldots,[\mathrm{love}\colon\mathbb{I}\times\mathbb{I}\to\mathbb{B}],[\mathrm{bill}\colon\mathbb{I}],[\mathrm{mary}\colon\mathbb{I}],\ldots\}\rangle$
The abstract data type $\mathcal{A}$ has the ground terms:

- ▷ $g_1 := \mathrm{love}(\mathrm{bill},\mathrm{mary})$ (how nice)
- ▷ $g_2 := \mathrm{love}(\mathrm{mary},\mathrm{bill})\wedge\neg\mathrm{love}(\mathrm{bill},\mathrm{mary})$ (how sad)
- ▷ $g_3 := \mathrm{love}(\mathrm{bill},\mathrm{mary})\wedge\mathrm{love}(\mathrm{mary},\mathrm{john})\Rightarrow\mathrm{hate}(\mathrm{bill},\mathrm{john})$ (how natural)

▷ Semantics: by mapping into known stuff, (e.g. $\mathbb{I}$ to persons $\mathbb{B}$ to $\{\mathsf{T},\mathsf{F}\}$)

▷ Idea: Import semantics from Boolean Algebra (atoms "are" variables)

- ▷ only need variable assignment $\varphi\colon\mathcal{A}(\Sigma)\to\{\mathsf{T},\mathsf{F}\}$

▷ **Example 10.1.9** $\mathcal{I}_\varphi(\mathrm{love}(\mathrm{bill},\mathrm{mary})\wedge(\mathrm{love}(\mathrm{mary},\mathrm{john})\Rightarrow\mathrm{hate}(\mathrm{bill},\mathrm{john}))) = \mathsf{T}$ if $\varphi(\mathrm{love}(\mathrm{bill},\mathrm{mary})) = \mathsf{T}$, $\varphi(\mathrm{love}(\mathrm{mary},\mathrm{john})) = \mathsf{F}$, and $\varphi(\mathrm{hate}(\mathrm{bill},\mathrm{john})) = \mathsf{T}$

▷ **Example 10.1.10** $g_1\wedge g_3\wedge\mathrm{love}(\mathrm{mary},\mathrm{john})\models\mathrm{hate}(\mathrm{bill},\mathrm{john})$

What is Logic?

▷ formal languages, inference and their relation with the world

- ▷ Formal language $\mathcal{FL}$: set of formulae ($2+3/7$, $\forall x.x+y=y+x$)
- ▷ Formula: sequence/tree of symbols ($x,y,f,g,p,1,\pi,\in,\neg,\wedge\forall,\exists$)
- ▷ Models: things we understand (e.g. number theory)
- ▷ Interpretation: maps formulae into models ($[\![\text{three plus five}]\!]=8$)
- ▷ Validity: $\mathcal{M}\models\mathbf{A}$, iff $[\![\mathbf{A}]\!]^{\mathcal{M}}=\mathsf{T}$ (five greater three is valid)
- ▷ Entailment: $\mathbf{A}\models\mathbf{B}$, iff $\mathcal{M}\models\mathbf{B}$ for all $\mathcal{M}\models\mathbf{A}$. (generalize to $\mathcal{H}\models\mathbf{A}$)
- ▷ Inference: rules to transform (sets of) formulae ($\mathbf{A},\mathbf{A}\Rightarrow\mathbf{B}\vdash\mathbf{B}$)

▷ Syntax: formulae, inference (just a bunch of symbols)

▷ Semantics: models, interpr., validity, entailment (math. structures)

▷ Important Question: relation between syntax and semantics?

So logic is the study of formal representations of objects in the real world, and the formal statements that are true about them. The insistence on a *formal language* for representation is actually something that simplifies life for us. Formal languages are something that is actually easier to understand than e.g. natural languages. For instance it is usually decidable, whether a string is a member of a formal language. For natural language this is much more difficult: there is still no program that can reliably say whether a sentence is a grammatical sentence of the English language.

We have already discussed the meaning mappings (under the monicker "semantics"). Meaning mappings can be used in two ways, they can be used to understand a formal language, when we use a mapping into "something we already understand", or they are the mapping that legitimize a representation in a formal language. We understand a formula (a member of a formal language) $\mathbf{A}$ to be a representation of an object $\mathcal{O}$, iff $[\![\mathbf{A}]\!] = \mathcal{O}$.

However, the game of representation only becomes really interesting, if we can do something with the representations. For this, we give ourselves a set of syntactic rules of how to manipulate the formulae to reach new representations or facts about the world.

Consider, for instance, the case of calculating with numbers, a task that has changed from a difficult job for highly paid specialists in Roman times to a task that is now feasible for young children. What is the cause of this dramatic change? Of course the formalized reasoning procedures for arithmetic that we use nowadays. These *calculi* consist of a set of rules that can be followed purely syntactically, but nevertheless manipulate arithmetic expressions in a correct and fruitful way. An essential prerequisite for syntactic manipulation is that the objects are given in a formal language suitable for the problem. For example, the introduction of the decimal system has been instrumental to the simplification of arithmetic mentioned above. When the arithmetical calculi were sufficiently well-understood and in principle a mechanical procedure, and when the art of clock-making was mature enough to design and build mechanical devices of an appropriate kind, the invention of calculating machines for arithmetic by Wilhelm Schickard (1623), Blaise Pascal (1642), and Gottfried Wilhelm Leibniz (1671) was only a natural consequence.

We will see that it is not only possible to calculate with numbers, but also with representations of statements about the world (propositions). For this, we will use an extremely simple example; a fragment of propositional logic (we restrict ourselves to only one logical connective) and a small calculus that gives us a set of rules how to manipulate formulae.

A simple System: Prop. Logic with Hilbert-Calculus

▷ Formulae: built from prop. variables: $P, Q, R, \ldots$ and implication: $\Rightarrow$

▷ Semantics: $\mathcal{I}_\varphi(P) = \varphi(P)$ and $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) =$ T, iff $\mathcal{I}_\varphi(\mathbf{A}) =$ F or $\mathcal{I}_\varphi(\mathbf{B}) =$ T.

▷ $\mathbf{K} := P \Rightarrow Q \Rightarrow P$, $\mathbf{S} := (P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R$

▷ $\dfrac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}}\,\mathrm{MP}$ $\qquad \dfrac{\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})}\,\mathrm{Subst}$

▷ Let us look at a $\mathcal{H}^0$ theorem (with a proof)

▷ $\mathbf{C} \Rightarrow \mathbf{C}$ (*Tertium non datur*)

▷ Proof:

P.1 $(\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}$ (**S** with $[\mathbf{C}/P], [\mathbf{C} \Rightarrow \mathbf{C}/Q], [\mathbf{C}/R]$)

P.2 $\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C}$ (**K** with $[\mathbf{C}/P], [\mathbf{C} \Rightarrow \mathbf{C}/Q]$)

P.3 $(\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}$ (MP on P.1 and P.2)

P.4 $\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}$ (**K** with $[\mathbf{C}/P], [\mathbf{C}/Q]$)

P.5 $\mathbf{C} \Rightarrow \mathbf{C}$ (MP on P.3 and P.4)

P.6 We have shown that $\emptyset \vdash_{\mathcal{H}^0} \mathbf{C} \Rightarrow \mathbf{C}$ (i.e. $\mathbf{C} \Rightarrow \mathbf{C}$ is a theorem) (is is also valid?)

$\square$

This is indeed a very simple logic, that with all of the parts that are necessary:

- A formal language: expressions built up from variables and implications.
- A semantics: given by the obvious interpretation function
- A calculus: given by the two axioms and the two inference rules.

The calculus gives us a set of rules with which we can derive new formulae from old ones. The axioms are very simple rules, they allow us to derive these two formulae in any situation. The inference rules are slightly more complicated: we read the formulae above the horizontal line as assumptions and the (single) formula below as the conclusion. An inference rule allows us to derive the conclusion, if we have already derived the assumptions.

Now, we can use these inference rules to perform a proof. A proof is a sequence of formulae that can be derived from each other. The representation of the proof in the slide is slightly compactified to fit onto the slide: We will make it more explicit here. We first start out by deriving the formula

$$(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R \tag{10.1}$$

which we can always do, since we have an axiom for this formula, then we apply the rule *subst*, where $\mathbf{A}$ is this result, $\mathbf{B}$ is $\mathbf{C}$, and X is the variable P to obtain

$$(\mathbf{C} \Rightarrow Q \Rightarrow R) \Rightarrow (\mathbf{C} \Rightarrow Q) \Rightarrow \mathbf{C} \Rightarrow R \tag{10.2}$$

Next we apply the rule *subst* to this where $\mathbf{B}$ is $\mathbf{C} \Rightarrow \mathbf{C}$ and X is the variable Q this time to obtain

$$(\mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow R) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{C} \Rightarrow R \tag{10.3}$$

And again, we apply the rule *subst* this time, $\mathbf{B}$ is $\mathbf{C}$ and X is the variable R yielding the first formula in our proof on the slide. To conserve space, we have combined these three steps into one in the slide. The next steps are done in exactly the same way.

10.2 A digression on Names and Logics

The name MP comes from the Latin name "modus ponens" (the "mode of putting" [new facts]), this is one of the classical syllogisms discovered by the ancient Greeks. The name Subst is just short for substitution, since the rule allows to instantiate variables in formulae with arbitrary other formulae.

Digression: To understand the reason for the names of **K** and **S** we have to understand much more logic. Here is what happens in a nutshell: There is a very tight connection between types of functional languages and propositional logic (google Curry/Howard Isomorphism). The **K** and **S** axioms are the types of the K and S combinators, which are functions that can make all other functions. In SML, we have already seen the K in Example 4.1.16:

```
val K = fn x => (fn y => x) : `a -> `b -> `a
```

Note that the type `a -> `b -> `a looks like (is isomorphic under the Curry/Howard isomorphism) to our axiom $P \Rightarrow Q \Rightarrow P$. Note furthermore that K a function that takes an argument n and returns a constant function (the function that returns n on all arguments). Now the German name for "constant function" is "Konstante Function", so you have letter K in the name. For the **S** aiom (which I do not know the naming of) you have

```
val S = fn x => (fn y => (fn z => x z (y z))) : (`a -> `b -> `c) - (`a -> `c) -> `a -> `c
```

Now, you can convince yourself that $SKKx = x = Ix$ (i.e. the function S applied to two copies of K is the identity combinator I). Note that

```
val I = x => x : `a -> `a
```

where the type of the identity looks like the theorem $\mathbf{C} \Rightarrow \mathbf{C}$ we proved. Moreover, under the Curry/Howard Isomorphism, proofs correspond to functions (axioms to combinators), and SKK is the function that corresponds to the proof we looked at in class.

We will now generalize what we have seen in the example so that we can talk about calculi and proofs in other situations and see what was specific to the example.

10.3 Calculi for Propositional Logic

Let us now turn to the syntactical counterpart of the entailment relation: derivability in a calculus. Again, we take care to define the concepts at the general level of logical systems.

The intuition of a calculus is that it provides a set of syntactic rules that allow to reason by considering the form of propositions alone. Such rules are called inference rules, and they can be strung together to derivations — which can alternatively be viewed either as sequences of formulae where all formulae are justified by prior formulae or as trees of inference rule applications. But we can also define a calculus in the more general setting of logical systems as an arbitrary relation on formulae with some general properties. That allows us to abstract away from the homomorphic setup of logics and calculi and concentrate on the basics.

Derivation Systems and Inference Rules

▷ **Definition 10.3.1** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system, then we call a relation $\vdash \subseteq \mathcal{P}(\mathcal{L}) \times \mathcal{L}$ a derivation relation for $\mathcal{S}$, if it

- ▷ is proof-reflexive, i.e. $\mathcal{H} \vdash \mathbf{A}$, if $\mathbf{A} \in \mathcal{H}$;
- ▷ is proof-transitive, i.e. if $\mathcal{H} \vdash \mathbf{A}$ and $\mathcal{H}' \cup \{\mathbf{A}\} \vdash \mathbf{B}$, then $\mathcal{H} \cup \mathcal{H}' \vdash \mathbf{B}$;
- ▷ admits weakening, i.e. $\mathcal{H} \vdash \mathbf{A}$ and $\mathcal{H} \subseteq \mathcal{H}'$ imply $\mathcal{H}' \vdash \mathbf{A}$.

▷ **Definition 10.3.2** We call $\langle \mathcal{L}, \mathcal{K}, \models, \vdash \rangle$ a formal system, iff $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ is a logical system, and $\vdash$ a derivation relation for $\mathcal{S}$.

▷ **Definition 10.3.3** Let $\mathcal{L}$ be a formal language, then an inference rule over $\mathcal{L}$

$$\frac{\mathbf{A}_1 \quad \cdots \quad \mathbf{A}_n}{\mathbf{C}}\mathcal{N}$$

where $\mathbf{A}_1, \ldots, \mathbf{A}_n$ and $\mathbf{C}$ are formula schemata for $\mathcal{L}$ and $\mathcal{N}$ is a name.
The $\mathbf{A}_i$ are called assumptions, and $\mathbf{C}$ is called conclusion.

▷ **Definition 10.3.4** An inference rule without assumptions is called an axiom (schema).

▷ **Definition 10.3.5** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system, then we call a set $\mathcal{C}$ of inference rules over $\mathcal{L}$ a calculus for $\mathcal{S}$.

With formula schemata we mean representations of sets of formulae, we use boldface uppercase letters as (meta)-variables for formulae, for instance the formula schema $\mathbf{A} \Rightarrow \mathbf{B}$ represents the set of formulae whose head is $\Rightarrow$.

Derivations and Proofs

▷ **Definition 10.3.6** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system and $\mathcal{C}$ a calculus for $\mathcal{S}$, then a $\mathcal{C}$-derivation of a formula $\mathbf{C} \in \mathcal{L}$ from a set $\mathcal{H} \subseteq \mathcal{L}$ of hypotheses (write $\mathcal{H} \vdash_{\mathcal{C}} \mathbf{C}$) is a sequence $\mathbf{A}_1, \ldots, \mathbf{A}_m$ of $\mathcal{L}$-formulae, such that

- ▷ $\mathbf{A}_m = \mathbf{C}$, (derivation culminates in $\mathbf{C}$)
- ▷ for all $1 \leq i \leq m$, either $\mathbf{A}_i \in \mathcal{H}$, or (hypothesis)
- ▷ there is an inference rule $\frac{\mathbf{A}_{l_1} \quad \cdots \quad \mathbf{A}_{l_k}}{\mathbf{A}_i}$ in $\mathcal{C}$ with $l_j < i$ for all $j \leq k$. (rule application)

Observation: We can also see a derivation as a tree, where the $\mathbf{A}_{l_j}$ are the children of the node $\mathbf{A}_k$.

▷ **Example 10.3.7** In the propositional Hilbert calculus $\mathcal{H}^0$ we have the derivation $P \vdash_{\mathcal{H}^0} Q \Rightarrow P$: the sequence is $P \Rightarrow Q \Rightarrow P, P, Q \Rightarrow P$ and the corresponding tree on the right.

$$\dfrac{\dfrac{}{P \Rightarrow Q \Rightarrow P}K \qquad P}{Q \Rightarrow P}MP$$

▷ **Observation 10.3.8** *Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system and $\mathcal{C}$ a calculus for $\mathcal{S}$, then the $\mathcal{C}$-derivation relation $\vdash_{\mathcal{D}}$ defined in Definition 10.3.6 is a derivation relation in the sense of Definition 10.3.1.*[2]

▷ **Definition 10.3.9** We call $\langle \mathcal{L}, \mathcal{K}, \models, \mathcal{C} \rangle$ a formal system, iff $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ is a logical system, and $\mathcal{C}$ a calculus for $\mathcal{S}$.

▷ **Definition 10.3.10** A derivation $\emptyset \vdash_{\mathcal{C}} \mathbf{A}$ is called a proof of $\mathbf{A}$ and if one exists (write $\vdash_{\mathcal{C}} \mathbf{A}$) then $\mathbf{A}$ is called a $\mathcal{C}$-theorem.

▷ **Definition 10.3.11** an inference rule $\mathcal{I}$ is called admissible in $\mathcal{C}$, if the extension of $\mathcal{C}$ by $\mathcal{I}$ does not yield new theorems.

[b] EdNote: MK: this should become a view!

Inference rules are relations on formulae represented by formula schemata (where boldface, uppercase letters are used as meta-variables for formulae). For instance, in Example 10.3.7 the inference rule $\frac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}}$ was applied in a situation, where the meta-variables $\mathbf{A}$ and $\mathbf{B}$ were instantiated by the formulae P and $Q \Rightarrow P$.

As axioms do not have assumptions, they can be added to a derivation at any time. This is just what we did with the axioms in Example 10.3.7.

In general formulae can be used to represent facts about the world as propositions; they have a semantics that is a mapping of formulae into the real world (propositions are mapped to truth values.) We have seen two relations on formulae: the entailment relation and the deduction relation. The first one is defined purely in terms of the semantics, the second one is given by a calculus, i.e. purely syntactically. Is there any relation between these relations?

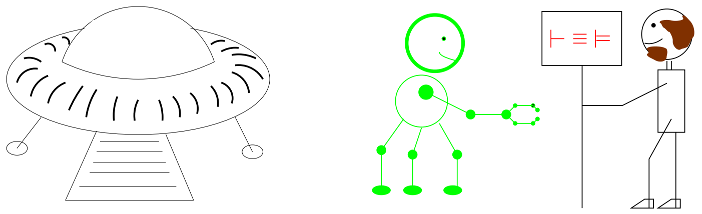
Soundness and Completeness

▷ **Definition 10.3.12** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system, then we call a calculus $\mathcal{C}$ for $\mathcal{S}$

- ▹ sound (or correct), iff $\mathcal{H} \models \mathbf{A}$, whenever $\mathcal{H} \vdash_{\mathcal{C}} \mathbf{A}$, and
- ▹ complete, iff $\mathcal{H} \vdash_{\mathcal{C}} \mathbf{A}$, whenever $\mathcal{H} \models \mathbf{A}$.

▷ Goal: $\vdash \mathbf{A}$ iff $\models \mathbf{A}$ (provability and validity coincide)

- ▹ To TRUTH through PROOF (CALCULEMUS [Leibniz ∼1680])



Ideally, both relations would be the same, then the calculus would allow us to infer all facts that can be represented in the given formal language and that are true in the real world, and only those. In other words, our representation and inference is faithful to the world.

A consequence of this is that we can rely on purely syntactical means to make predictions about the world. Computers rely on formal representations of the world; if we want to solve a problem on our computer, we first represent it in the computer (as data structures, which can be seen as a formal language) and do syntactic manipulations on these structures (a form of calculus). Now, if the provability relation induced by the calculus and the validity relation coincide (this will be quite difficult to establish in general), then the solutions of the program will be correct, and we will find all possible ones.

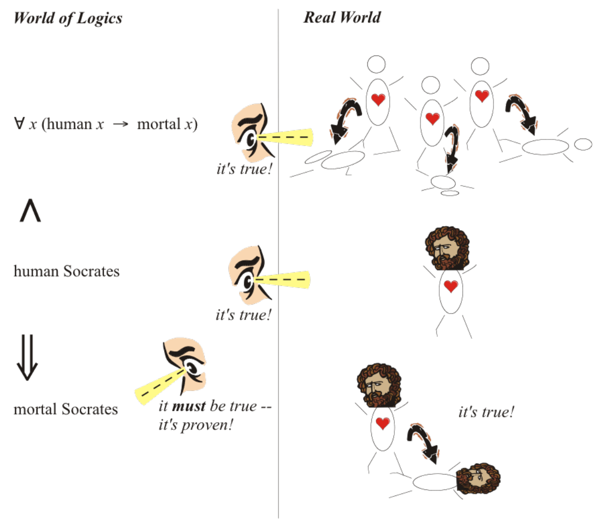
Of course, the logics we have studied so far are very simple, and not able to express interesting facts about the world, but we will study them as a simple example of the fundamental problem of Computer Science: How do the formal representations correlate with the real world.

Within the world of logics, one can derive new propositions (the *conclusions*, here: *Socrates is mortal*) from given ones (the *premises*, here: *Every human is mortal* and *Sokrates is human*). Such derivations are *proofs*.

In particular, logics can describe the internal structure of real-life facts; e.g. individual things, actions, properties. A famous example, which is in fact as old as it appears, is illustrated in the slide below.

The miracle of logics

▷ Purely formal derivations are true in the real world!



If a logic is correct, the conclusions one can prove are true (= hold in the real world) whenever the premises are true. This is a miraculous fact (think about it!)

10.4 Proof Theory for the Hilbert Calculus

We now show one of the meta-properties (soundness) for the Hilbert calculus $\mathcal{H}^0$. The statement of the result is rather simple: it just says that the set of provable formulae is a subset of the set of valid formulae. In other words: If a formula is provable, then it must be valid (a rather comforting property for a calculus).

$\mathcal{H}^0$ is sound (first version)

▷ **Theorem 10.4.1** $\vdash \mathbf{A}$ *implies* $\models \mathbf{A}$ *for all propositions* $\mathbf{A}$.

▷ Proof: show by induction over proof length

P.1 Axioms are valid (we already know how to do this!)

P.2 inference rules preserve validity (let's think)

P.2.1 Subst: complicated, see next slide

P.2.2 MP:

P.2.2.1 Let $\mathbf{A} \Rightarrow \mathbf{B}$ be valid, and $\varphi\colon \mathcal{V}_o \to \{\mathsf{T}, \mathsf{F}\}$ arbitrary

P.2.2.2 then $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{F}$ or $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$ (by definition of $\Rightarrow$).

P.2.2.3 Since $\mathbf{A}$ is valid, $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{T} \neq \mathsf{F}$, so $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$.

P.2.2.4 As φ was arbitrary, $\mathbf{B}$ is valid. □

□

To complete the proof, we have to prove two more things. The first one is that the axioms are valid. Fortunately, we know how to do this: we just have to show that under all assignments, the axioms are satisfied. The simplest way to do this is just to use truth tables.

$\mathcal{H}^0$ axioms are valid

▷ **Lemma 10.4.2** *The $\mathcal{H}^0$ axioms are valid.*

▷ Proof: We simply check the truth tables

P.1

P	Q	$Q \Rightarrow P$	$P \Rightarrow Q \Rightarrow P$
F	F	T	T
F	T	F	T
T	F	T	T
T	T	T	T

P.2

P	Q	R	$\mathbf{A} := P \Rightarrow Q \Rightarrow R$	$\mathbf{B} := P \Rightarrow Q$	$\mathbf{C} := P \Rightarrow R$	$\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{C}$
F	F	F	T	T	T	T
F	F	T	T	T	T	T
F	T	F	T	T	T	T
F	T	T	T	T	T	T
T	F	F	T	F	F	T
T	F	T	T	F	T	T
T	T	F	F	T	F	T
T	T	T	T	T	T	T

□

The next result encapsulates the soundness result for the substitution rule, which we still owe. We will prove the result by induction on the structure of the formula that is instantiated. To get the induction to go through, we not only show that validity is preserved under instantiation, but we make a concrete statement about the value itself.

A proof by induction on the structure of the formula is something we have not seen before. It can be justified by a normal induction over natural numbers; we just take property of a natural number n to be that all formulae with n symbols have the property asserted by the theorem. The only thing we need to realize is that proper subterms have strictly less symbols than the terms themselves.

Substitution Value Lemma and Soundness

▷ **Lemma 10.4.3** *Let $\mathbf{A}$ and $\mathbf{B}$ be formulae, then $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_\psi(\mathbf{A})$, where $\psi = \varphi, [\mathcal{I}_\varphi(\mathbf{B})/X]$*

▷ Proof: by induction on the depth of $\mathbf{A}$ (number of nested $\Rightarrow$ symbols)

P.1 We have to consider two cases

P.1.1 depth=0, then $\mathbf{A}$ is a variable, say Y.:

P.1.1.1 We have two cases

P.1.1.1.1 $X = Y$: then $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_\varphi([\mathbf{B}/X](X)) = \mathcal{I}_\varphi(\mathbf{B}) = \psi(X) = \mathcal{I}_\psi(X) = \mathcal{I}_\psi(\mathbf{A})$.

P.1.1.1.2 $X \neq Y$: then $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_\varphi([\mathbf{B}/X](Y)) = \mathcal{I}_\varphi(Y) = \varphi(Y) = \psi(Y) = \mathcal{I}_\psi(Y) = \mathcal{I}_\psi(\mathbf{A})$.

P.1.2 depth>0, then $\mathbf{A} = \mathbf{C} \Rightarrow \mathbf{D}$:

P.1.2.1 We have $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathsf{T}$, iff $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{C})) = \mathsf{F}$ or $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{D})) = \mathsf{T}$.

P.1.2.2 This is the case, iff $\mathcal{I}_\psi(\mathbf{C}) = \mathsf{F}$ or $\mathcal{I}_\psi(\mathbf{D}) = \mathsf{T}$ by IH ($\mathbf{C}$ and $\mathbf{D}$ have smaller depth than $\mathbf{A}$).

P.1.2.3 In other words, $\mathcal{I}_\psi(\mathbf{A}) = \mathcal{I}_\psi(\mathbf{C} \Rightarrow \mathbf{D}) = \mathsf{T}$, iff $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathsf{T}$ by definition. □

P.2 We have considered all the cases and proven the assertion. □

Armed with the substitution value lemma, it is quite simple to establish the soundness of the substitution rule. We state the assertion rather succinctly: "Subst preservers validity", which means that if the assumption of the Subst rule was valid, then the conclusion is valid as well, i.e. the validity property is preserved.

Soundness of Substitution

▷ **Lemma 10.4.4** Subst *preserves validity.*

▷ Proof: We have to show that $[\mathbf{B}/X](\mathbf{A})$ is valid, if $\mathbf{A}$ is.

P.1 Let $\mathbf{A}$ be valid, $\mathbf{B}$ a formula, $\varphi\colon \mathcal{V}_o \to \{\mathsf{T},\mathsf{F}\}$ a variable assignment, and $\psi := \varphi, [\mathcal{I}_\varphi(\mathbf{B})/X]$.

P.2 then $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_{\varphi,[\mathcal{I}_\varphi(\mathbf{B})/X]}(\mathbf{A}) = \mathsf{T}$, since $\mathbf{A}$ is valid.

P.3 As the argumentation did not depend on the choice of φ, $[\mathbf{B}/X](\mathbf{A})$ valid and we have proven the assertion. □

The next theorem shows that the implication connective and the entailment relation are closely related: we can move a hypothesis of the entailment relation into an implication assumption in the conclusion of the entailment relation. Note that however close the relationship between implication and entailment, the two should not be confused. The implication connective is a syntactic formula constructor, whereas the entailment relation lives in the semantic realm. It is a relation between formulae that is induced by the evaluation mapping.

The Entailment Theorem

▷ **Theorem 10.4.5** *If* $\mathcal{H}, \mathbf{A} \models \mathbf{B}$, *then* $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$.

▷ Proof: We show that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{T}$ for all assignments φ with $\mathcal{I}_\varphi(\mathcal{H}) = \mathsf{T}$ whenever $\mathcal{H}, \mathbf{A} \models \mathbf{B}$

P.1 Let us assume there is an assignment φ, such that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{F}$.

P.2 Then $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{T}$ and $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{F}$ by definition.

P.3 But we also know that $\mathcal{I}_\varphi(\mathcal{H}) = \mathsf{T}$ and thus $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$, since $\mathcal{H}, \mathbf{A} \models \mathbf{B}$.

P.4 This contradicts our assumption $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$ from above.

P.5 So there cannot be an assignment φ that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{F}$; in other words, $\mathbf{A} \Rightarrow \mathbf{B}$ is valid. □

Now, we complete the theorem by proving the converse direction, which is rather simple.

The Entailment Theorem (continued)

▷ **Corollary 10.4.6** $\mathcal{H}, \mathbf{A} \models \mathbf{B}$, *iff* $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$

▷ Proof: In the light of the previous result, we only need to prove that $\mathcal{H}, \mathbf{A} \models \mathbf{B}$, whenever $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$

P.1 To prove that $\mathcal{H}, \mathbf{A} \models \mathbf{B}$ we assume that $\mathcal{I}_\varphi(\mathcal{H}, \mathbf{A}) = \mathsf{T}$.

P.2 In particular, $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \mathsf{T}$ since $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$.

P.3 Thus we have $\mathcal{I}_\varphi(\mathbf{A}) = \mathsf{F}$ or $\mathcal{I}_\varphi(\mathbf{B}) = \mathsf{T}$.

P.4 The first cannot hold, so the second does, thus $\mathcal{H}, \mathbf{A} \models \mathbf{B}$. □

The entailment theorem has a syntactic counterpart for some calculi. This result shows a close connection between the derivability relation and the implication connective. Again, the two should not be confused, even though this time, both are syntactic.

The main idea in the following proof is to generalize the inductive hypothesis from proving $\mathbf{A} \Rightarrow \mathbf{B}$ to proving $\mathbf{A} \Rightarrow \mathbf{C}$, where $\mathbf{C}$ is a step in the proof of $\mathbf{B}$. The assertion is a special case then, since $\mathcal{B}$ is the last step in the proof of $\mathbf{B}$.

The Deduction Theorem

▷ **Theorem 10.4.7** *If $\mathcal{H}, \mathbf{A} \vdash \mathbf{B}$, then $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{B}$*

▷ Proof: By induction on the proof length

P.1 Let $\mathbf{C}_1, \ldots, \mathbf{C}_m$ be a proof of $\mathbf{B}$ from the hypotheses $\mathcal{H}$.

P.2 We generalize the induction hypothesis: For all l $1 \leq i \leq m$ we construct proofs $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$. (get $\mathbf{A} \Rightarrow \mathbf{B}$ for $i = m$)

P.3 We have to consider three cases

P.3.12 Case 1: $\mathbf{C}_i$ axiom or $\mathbf{C}_i \in \mathcal{H}$:

P.3.12.1 Then $\mathcal{H} \vdash \mathbf{C}_i$ by construction and $\mathcal{H} \vdash \mathbf{C}_i \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Subst from Axiom 1.

P.3.12.2 So $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$ by MP. □

P.3.13 Case 2: $\mathbf{C}_i = \mathbf{A}$:

P.3.13.1 We have already proven $\emptyset \vdash \mathbf{A} \Rightarrow \mathbf{A}$, so in particular $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$. (more hypotheses do not hurt) □

P.3.14 Case 3: everything else:

P.3.14.1 $\mathbf{C}_i$ is inferred by MP from $\mathbf{C}_j$ and $\mathbf{C}_k = \mathbf{C}_j \Rightarrow \mathbf{C}_i$ for $j, k < i$

P.3.14.2 We have $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_j$ and $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i$ by IH

P.3.14.3 Furthermore, $(\mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{C}_j) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Axiom 2 and Subst

P.3.14.4 and thus $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$ by MP (twice). □

P.4 We have treated all cases, and thus proven $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$ for $1 \leq i \leq m$.

P.5 Note that $\mathbf{C}_m = \mathbf{B}$, so we have in particular proven $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{B}$. □

In fact (you have probably already spotted this), this proof is not correct. We did not cover all cases: there are proofs that end in an application of the Subst rule. This is a common situation, we think we have a very elegant and convincing proof, but upon a closer look it turns out that there is a gap, which we still have to bridge.

This is what we attempt to do now. The first attempt to prove the subst case below seems to work at first, until we notice that the substitution $[\mathbf{B}/X]$ would have to be applied to $\mathbf{A}$ as well, which ruins our assertion.

The missing Subst case

- ▷ Oooops: The proof of the deduction theorem was incomplete (we did not treat the Subst case)
- ▷ Let's try:
- ▷ Proof: $\mathbf{C}_i$ is inferred by Subst from $\mathbf{C}_j$ for $j<i$ with $[\mathbf{B}/X]$.

P.1 So $\mathbf{C}_i = [\mathbf{B}/X](\mathbf{C}_j)$; we have $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_j$ by IH

P.2 so by Subst we have $\mathcal{H} \vdash [\mathbf{B}/X](\mathbf{A} \Rightarrow \mathbf{C}_j)$. (Oooops! $\neq \mathbf{A} \Rightarrow \mathbf{C}_i$)

□

In this situation, we have to do something drastic, like come up with a totally different proof. Instead we just prove the theorem we have been after for a variant calculus.

Repairing the Subst case by repairing the calculus

- ▷ Idea: Apply Subst only to axioms (this was sufficient in our example)
- ▷ $\mathcal{H}^1$ Axiom Schemata: (infinitely many axioms)

  $\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}$, $(\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}$
  Only one inference rule: MP.
- ▷ **Definition 10.4.8** $\mathcal{H}^1$ introduces a (potentially) different derivability relation than $\mathcal{H}^0$ we call them $\vdash_{\mathcal{H}^0}$ and $\vdash_{\mathcal{H}^1}$

Now that we have made all the mistakes, let us write the proof in its final form.

Deduction Theorem Redone

- ▷ **Theorem 10.4.9** *If* $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{H}^1} \mathbf{B}$, *then* $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{B}$
- ▷ Proof: Let $\mathbf{C}_1, \ldots, \mathbf{C}_m$ be a proof of $\mathbf{B}$ from the hypotheses $\mathcal{H}$.

P.1 We construct proofs $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ for all $1 \leq i \leq n$ by induction on i.

P.2 We have to consider three cases

P.2.1 $\mathbf{C}_i$ is an axiom or hypothesis:

P.2.1.1 Then $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{C}_i$ by construction and $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{C}_i \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Ax1.

P.2.1.2 So $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{C}_i$ by MP □

P.2.2 $\mathbf{C}_i = \mathbf{A}$:

P.2.2.1 We have proven $\emptyset \vdash_{\mathcal{H}^0} \mathbf{A} \Rightarrow \mathbf{A}$, (check proof in $\mathcal{H}^1$)
We have $\emptyset \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$, so in particular $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ □

P.2.3 else:

P.2.3.1 $\mathbf{C}_i$ is inferred by MP from $\mathbf{C}_j$ and $\mathbf{C}_k = \mathbf{C}_j \Rightarrow \mathbf{C}_i$ for $j, k<i$

P.2.3.2 We have $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_j$ and $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i$ by IH

P.2.3.3 Furthermore, $(\mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{C}_j) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Axiom 2

P.2.3.4 and thus $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ by MP (twice). (no Subst)

□

□

The deduction theorem and the entailment theorem together allow us to understand the claim that the two formulations of soundness ($\mathbf{A} \vdash \mathbf{B}$ implies $\mathbf{A} \models \mathbf{B}$ and $\vdash \mathbf{A}$ implies $\models \mathbf{B}$) are equivalent. Indeed, if we have $\mathbf{A} \vdash \mathbf{B}$, then by the deduction theorem $\vdash \mathbf{A} \Rightarrow \mathbf{B}$, and thus $\models \mathbf{A} \Rightarrow \mathbf{B}$ by soundness, which gives us $\mathbf{A} \models \mathbf{B}$ by the entailment theorem. The other direction and the argument for the corresponding statement about completeness are similar.

Of course this is still not the version of the proof we originally wanted, since it talks about the Hilbert Calculus $\mathcal{H}^1$, but we can show that $\mathcal{H}^1$ and $\mathcal{H}^0$ are equivalent.

But as we will see, the derivability relations induced by the two caluli are the same. So we can prove the original theorem after all.

The Deduction Theorem for $\mathcal{H}^0$

▷ **Lemma 10.4.10** $\vdash_{\mathcal{H}^1} = \vdash_{\mathcal{H}^0}$

▷ Proof:

P.1 All $\mathcal{H}^1$ axioms are $\mathcal{H}^0$ theorems. (by Subst)

P.2 For the other direction, we need a proof transformation argument:

P.3 We can replace an application of MP followed by Subst by two Subst applications followed by one MP.

P.4 $\ldots \mathbf{A} \Rightarrow \mathbf{B} \ldots \mathbf{A} \ldots \mathbf{B} \ldots [\mathbf{C}/X](\mathbf{B}) \ldots$ is replaced by

$$\ldots \mathbf{A} \Rightarrow \mathbf{B} \ldots [\mathbf{C}/X](\mathbf{A}) \Rightarrow [\mathbf{C}/X](\mathbf{B}) \ldots \mathbf{A} \ldots [\mathbf{C}/X](\mathbf{A}) \ldots [\mathbf{C}/X](\mathbf{B}) \ldots$$

P.5 Thus we can push later Subst applications to the axioms, transforming a $\mathcal{H}^0$ proof into a $\mathcal{H}^1$ proof. □

▷ **Corollary 10.4.11** $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$, *iff* $\mathcal{H} \vdash_{\mathcal{H}^0} \mathbf{A} \Rightarrow \mathbf{B}$.

▷ Proof Sketch: by MP and $\vdash_{\mathcal{H}^1} = \vdash_{\mathcal{H}^0}$ □

We can now collect all the pieces and give the full statement of the soundness theorem for $\mathcal{H}^0$

$\mathcal{H}^0$ is sound (full version)

▷ **Theorem 10.4.12** *For all propositions* $\mathbf{A}$, $\mathbf{B}$, *we have* $\mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$ *implies* $\mathbf{A} \models \mathbf{B}$.

▷ Proof:

P.1 By deduction theorem $\mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$, iff $\vdash \mathbf{A} \Rightarrow \mathbf{C}$,

P.2 by the first soundness theorem this is the case, iff $\models \mathbf{A} \Rightarrow \mathbf{B}$,

P.3 by the entailment theorem this holds, iff $\mathbf{A} \models \mathbf{C}$. □

Now, we can look at all the results so far in a single overview slide:

Properties of Calculi (Theoretical Logic)

▷ Correctness: (provable implies valid)

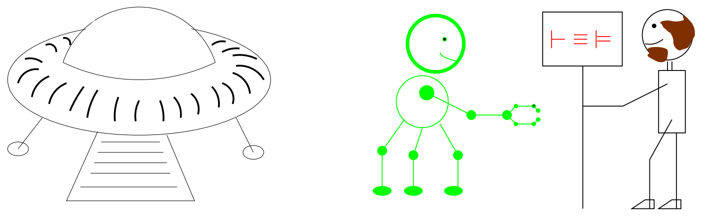
- ▹ $\mathcal{H} \vdash \mathbf{B}$ implies $\mathcal{H} \models \mathbf{B}$ (equivalent: $\vdash \mathbf{A}$ implies $\models \mathbf{A}$)

▷ Completeness: (valid implies provable)

- ▹ $\mathcal{H} \models \mathbf{B}$ implies $\mathcal{H} \vdash \mathbf{B}$ (equivalent: $\models \mathbf{A}$ implies $\vdash \mathbf{A}$)

▷ Goal: $\vdash \mathbf{A}$ iff $\models \mathbf{A}$ (provability and validity coincide)

- ▹ To TRUTH through PROOF (CALCULEMUS [Leibniz ∼1680])



10.5 A Calculus for Mathtalk

In our introduction to Section 10.1 we have positioned Boolean expressions (and proposition logic) as a system for understanding the mathematical language "mathtalk" introduced in Section 3.4. We have been using this language to state properties of objects and prove them all through this course without making the rules the govern this activity fully explicit. We will rectify this now: First we give a calculus that tries to mimic the the informal rules mathematicians use int their proofs, and second we show how to extend this "calculus of natural deduction" to the full language of "mathtalk".

10.5.1 Propositional Natural Deduction Calculus

We will now introduce the "natural deduction" calculus for propositional logic. The calculus was created in order to model the natural mode of reasoning e.g. in everyday mathematical practice. This calculus was intended as a counter-approach to the well-known Hilbert style calculi, which were mainly used as theoretical devices for studying reasoning in principle, not for modeling particular reasoning styles.

Rather than using a minimal set of inference rules, the natural deduction calculus provides two/three inference rules for every connective and quantifier, one "introduction rule" (an inference rule that derives a formula with that symbol at the head) and one "elimination rule" (an inference rule that acts on a formula with this head and derives a set of subformulae).

Calculi: Natural Deduction ($\mathcal{ND}^0$; Gentzen [Gen35])

- Idea: $\mathcal{ND}^0$ tries to mimic human theorem proving behavior (non-minimal)
- **Definition 10.5.1** The propositional natural deduction calculus $\mathcal{ND}^0$ has rules for the introduction and elimination of connectives

Introduction	Elimination	Axiom
$\dfrac{\mathbf{A}\quad\mathbf{B}}{\mathbf{A}\wedge\mathbf{B}}\wedge I$	$\dfrac{\mathbf{A}\wedge\mathbf{B}}{\mathbf{A}}\wedge E_l \qquad \dfrac{\mathbf{A}\wedge\mathbf{B}}{\mathbf{B}}\wedge E_r$	$\dfrac{}{\mathbf{A}\vee\neg\mathbf{A}}\mathrm{TND}$
$\dfrac{\begin{array}{c}[\mathbf{A}]^1\\ \hline\hline \mathbf{B}\end{array}}{\mathbf{A}\Rightarrow\mathbf{B}}\Rightarrow I^1$	$\dfrac{\mathbf{A}\Rightarrow\mathbf{B}\quad\mathbf{A}}{\mathbf{B}}\Rightarrow E$	

- TND is used only in classical logic (otherwise constructive/intuitionistic)

The most characteristic rule in the natural deduction calculus is the $\Rightarrow I$ rule. It corresponds to the mathematical way of proving an implication $\mathbf{A}\Rightarrow\mathbf{B}$: We assume that $\mathbf{A}$ is true and show $\mathbf{B}$ from this assumption. When we can do this we discharge (get rid of) the assumption and conclude $\mathbf{A}\Rightarrow\mathbf{B}$. This mode of reasoning is called hypothetical reasoning. Note that the local hypothesis is discharged by the rule $\Rightarrow I$, i.e. it cannot be used in any other part of the proof. As the $\Rightarrow I$ rules may be nested, we decorate both the rule and the corresponding assumption with a marker (here the number 1).

Let us now consider an example of hypothetical reasoning in action.

Natural Deduction: Examples

- Inference with local hypotheses

$$\dfrac{\dfrac{\dfrac{[\mathbf{A}\wedge\mathbf{B}]^1}{\mathbf{B}}\wedge E_r \qquad \dfrac{[\mathbf{A}\wedge\mathbf{B}]^1}{\mathbf{A}}\wedge E_l}{\mathbf{B}\wedge\mathbf{A}}\wedge I}{\mathbf{A}\wedge\mathbf{B}\Rightarrow\mathbf{B}\wedge\mathbf{A}}\Rightarrow I^1$$

$$\begin{array}{c}[A]^1\\ [B]^2\\ \dfrac{\dfrac{A}{B\Rightarrow A}\Rightarrow I^2}{A\Rightarrow B\Rightarrow A}\Rightarrow I^1\end{array}$$

One of the nice things about the natural deduction calculus is that the deduction theorem is almost trivial to prove. In a sense, the triviality of the deduction theorem is the central idea of the calculus and the feature that makes it so natural.

A Deduction Theorem for $\mathcal{ND}^0$

▷ **Theorem 10.5.2** $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{ND}^0} \mathbf{B}$, *iff* $\mathcal{H} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$.

▷ Proof: We show the two directions separately

P.1 If $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{ND}^0} \mathbf{B}$, then $\mathcal{H} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$ by $\Rightarrow I$, and

P.2 If $\mathcal{H} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$, then $\mathcal{H}, \mathcal{A} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$ by weakening and $\mathcal{H}, \mathcal{A} \vdash_{\mathcal{ND}^0} \mathbf{B}$ by $\Rightarrow E$. ☐

Another characteristic of the natural deduction calculus is that it has inference rules (introduction and elimination rules) for all connectives. So we extend the set of rules from Definition 10.5.1 for disjunction, negation and falsity.

More Rules for Natural Deduction

▷ **Definition 10.5.3** $\mathcal{ND}^0$ has the following additional rules for the remaining connectives.

$$\frac{\mathbf{A}}{\mathbf{A} \vee \mathbf{B}} \vee I_l \qquad \frac{\mathbf{B}}{\mathbf{A} \vee \mathbf{B}} \vee I_r \qquad \frac{\mathbf{A} \vee \mathbf{B} \quad \begin{array}{c}[\mathbf{A}]^1\\ \vdots\\ \mathbf{C}\end{array} \quad \begin{array}{c}[\mathbf{B}]^1\\ \vdots\\ \mathbf{C}\end{array}}{\mathbf{C}} \vee E^1$$

$$\frac{\begin{array}{c}[\mathbf{A}]^1\\ \vdots\\ F\end{array}}{\neg \mathbf{A}} \neg I^1 \qquad \frac{\neg \neg \mathbf{A}}{\mathbf{A}} \neg E$$

$$\frac{\neg \mathbf{A} \quad \mathbf{A}}{F} FI \qquad \frac{F}{\mathbf{A}} FE$$

The next step now is to extend the language of propositional logic to include the quantifiers $\forall$ and $\exists$. To do this, we will extend the language PLNQ with formulae of the form $\forall x.\mathbf{A}$ and $\exists x.\mathbf{A}$, where x is a variable and $\mathbf{A}$ is a formula. This system (which ist a little more involved than we make believe now) is called "first-order logic".[3] EdN:3

Building on the calculus $\mathcal{ND}^0$, we define a first-order calculus for "mathtalk" by providing introduction and elimination rules for the quantifiers.

[3] EdNote: give a forward reference

To obtain a first-order calculus, we have to extend $\mathcal{ND}^0$ with (introduction and elimination) rules for the quantifiers.

First-Order Natural Deduction ($\mathcal{ND}^1$; Gentzen [Gen35])

- ▷ Rules for propositional connectives just as always
- ▷ **Definition 10.5.4 (New Quantifier Rules)** The first-order natural deduction calculus $\mathcal{ND}^1$ extends $\mathcal{ND}^0$ by the following four rules

$$\frac{\mathbf{A}}{\forall X.\mathbf{A}}\forall I^* \qquad \frac{\forall X.\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})}\forall E$$

$$\frac{[\mathbf{B}/X](\mathbf{A})}{\exists X.\mathbf{A}}\exists I \qquad \frac{\exists X.\mathbf{A} \quad \begin{array}{c}[[c/X](\mathbf{A})]^1\\ \vdots\\ \mathbf{C}\end{array}}{\mathbf{C}}\exists E^1$$

* means that $\mathbf{A}$ does not depend on any hypothesis in which X is free.

The intuition behind the rule $\forall I$ is that a formula $\mathbf{A}$ with a (free) variable X can be generalized to $\forall X.\mathbf{A}$, if X stands for an arbitrary object, i.e. there are no restricting assumptions about X. The $\forall E$ rule is just a substitution rule that allows to instantiate arbitrary terms $\mathbf{B}$ for X in $\mathbf{A}$. The $\exists I$ rule says if we have a witness $\mathbf{B}$ for X in $\mathbf{A}$ (i.e. a concrete term $\mathbf{B}$ that makes $\mathbf{A}$ true), then we can existentially close $\mathbf{A}$. The $\exists E$ rule corresponds to the common mathematical practice, where we give objects we know exist a new name c and continue the proof by reasoning about this concrete object c. Anything we can prove from the assumption $[c/X](\mathbf{A})$ we can prove outright if $\exists X.\mathbf{A}$ is known.

The only part of MathTalk we have not treated yet is equality. This comes now.

Natural Deduction with Equality

- ▷ **Definition 10.5.5 (First-Order Logic with Equality)** We extend PL^1 with a new logical symbol for equality $= \in \Sigma_2^p$ and fix its semantics to $\mathcal{I}(=) := \{(x,x) \mid x \in \mathcal{D}_\iota\}$. We call the extended logic first-order logic with equality ($\mathrm{PL}^1_=$)
- ▷ We now extend natural deduction as well.
- ▷ **Definition 10.5.6** For the calculus of natural deduction with equality $\mathcal{ND}^1_=$ we add the following two equality rules to $\mathcal{ND}^1$ to deal with equality:

$$\frac{}{\mathbf{A}=\mathbf{A}}=I \qquad \frac{\mathbf{A}=\mathbf{B} \quad \mathbf{C}\,[\mathbf{A}]_p}{[\mathbf{B}/p]\mathbf{C}}=E$$

where $\mathbf{C}\,[\mathbf{A}]_p$ if the formula $\mathbf{C}$ has a subterm $\mathbf{A}$ at position p and $[\mathbf{B}/p]\mathbf{C}$ is the result of replacing that subterm with $\mathbf{B}$.

Again, we have two rules that follow the introduction/elimination pattern of natural deduction calculi.

With the $\mathcal{ND}^1$ calculus we have given a set of inference rules that are (empirically) complete for all the proof we need for the General Computer Science courses. Indeed Mathematicians are convinced that (if pressed hard enough) they could transform all (informal but rigorous) proofs into (formal) $\mathcal{ND}^1$ proofs. This is however seldom done in practice because it is extremely tedious, and mathematicians are sure that peer review of mathematical proofs will catch all relevant errors.

We will now show this on an example: the proof of the irrationality of the square root of two.

Theorem 10.5.7 $\sqrt{2}$ *is irrational*

Proof: We prove the assertion by contradiction

P.1 Assume that $\sqrt{2}$ is rational.

P.2 Then there are numbers p and q such that $\sqrt{2} = p\,/\,q$.

P.3 So we know $2\ s^2 = r^2$.

P.4 But $2\ s^2$ has an odd number of prime factors while r^2 an even number.

P.5 This is a contradiction (since they are equal), so we have proven the assertion □

If we want to formalize this into $\mathcal{ND}^1$, we have to write down all the assertions in the proof steps in MathTalk and come up with justifications for them in terms of $\mathcal{ND}^1$ inference rules. Figure 10.1 shows such a proof, where we write $\prime n$ is prime, use $\#(n)$ for the number of prime factors of a number n, and write $\text{irr}(r)$ if r is irrational. Each line in Figure 10.1 represents one "step" in the proof. It consists of line number (for referencing), a formula for the asserted property, a justification via a $\mathcal{ND}^1$ rules (and the lines this one is derived from), and finally a list of line numbers of proof steps that are local hypotheses in effect for the current line. Lines 6 and 9 have the pseudo-justification "local hyp" that indicates that they are local hypotheses for the proof (they only have an implicit counterpart in the inference rules as defined above). Finally we have abbreviated the arithmetic simplification of line 9 with the justification "arith" to avoid having to formalize elementary arithmetic.

We observe that the $\mathcal{ND}^1$ proof is much more detailed, and needs quite a few Lemmata about # to go through. Furthermore, we have added a MathTalk version of the definition of irrationality (and treat definitional equality via the equality rules). Apart from these artefacts of formalization, the two representations of proofs correspond to each other very directly.

In some areas however, this quality standard is not safe enough, e.g. for programs that control nuclear power plants. The field of "Formal Methods" which is at the intersection of mathematics and Computer Science studies how the behavior of programs can be specified formally in special logics and how fully formal proofs of safety properties of programs can be developed semi-automatically. Note that given the discussion in Section 10.3 fully formal proofs (in sound calculi) can be that can be checked by machines since their soundness only depends on the form of the formulae in them.

#	hyp	formula	NDjust
1		$\forall n, m . \neg (2\ n+1) = (2\ m)$	lemma
2		$\forall n, m . \#(n^m) = m\ \#(n)$	lemma
3		$\forall n, p . \prime p \Rightarrow \#(p\ n) = \#(n)+1$	lemma
4		$\forall x . \mathrm{irr}(x) := \neg (\exists p, q . x = p\,/\,q)$	definition
5		$\mathrm{irr}(\sqrt{2}) = \neg (\exists p, q . \sqrt{2} = p\,/\,q)$	$\forall E(4)$
6	6	$\neg\, \mathrm{irr}(\sqrt{2})$	local hyp
7	6	$\neg \neg (\exists p, q . \sqrt{2} = p\,/\,q)$	$=E(5,4)$
8	6	$\exists p, q . \sqrt{2} = p\,/\,q$	$\neg E(7)$
9	6,9	$\sqrt{2} = r\,/\,s$	local hyp
10	6,9	$2\ s^2 = r^2$	arith(9)
11	6,9	$\#(r^2) = 2\ \#(r)$	$\forall E^2(2)$
12	6,9	$\prime 2 \Rightarrow \#(2\ s^2) = \#(s^2)+1$	$\forall E^2(1)$
13		$\prime 2$	lemma
14	6,9	$\#(2\ s^2) = \#(s^2)+1$	$\Rightarrow E(13,12)$
15	6,9	$\#(s^2) = 2\ \#(s)$	$\forall E^2(2)$
16	6,9	$\#(2\ s^2) = 2\ \#(s)+1$	$=E(14,15)$
17		$\#(r^2) = \#(r^2)$	$=I$
18	6,9	$\#(2\ s^2) = \#(r^2)$	$=E(17,10)$
19	6.9	$2\ \#(s)+1 = \#(r^2)$	$=E(18,16)$
20	6.9	$2\ \#(s)+1 = 2\ \#(r)$	$=E(19,11)$
21	6.9	$\neg (2\ \#(s)+1) = (2\ \#(r))$	$\forall E^2(1)$
22	6,9	F	$FI(20,21)$
23	6	F	$\exists E^6(22)$
24		$\neg \neg\, \mathrm{irr}(\sqrt{2})$	$\neg I^6(23)$
25		$\mathrm{irr}(\sqrt{2})$	$\neg E^2(23)$

Figure 10.1: A $\mathcal{ND}^1$ proof of the irrationality of $\sqrt{2}$

Chapter 11

Machine-Oriented Calculi

Now we have studied the Hilbert-style calculus in some detail, let us look at two calculi that work via a totally different principle. Instead of deducing new formulae from axioms (and hypotheses) and hoping to arrive at the desired theorem, we try to deduce a contradiction from the negation of the theorem. Indeed, a formula $\mathbf{A}$ is valid, iff $\neg\mathbf{A}$ is unsatisfiable, so if we derive a contradiction from $\neg\mathbf{A}$, then we have proven $\mathbf{A}$. The advantage of such "test-calculi" (also called negative calculi) is easy to see. Instead of finding a proof that ends in $\mathbf{A}$, we have to find any of a broad class of contradictions. This makes the calculi that we will discuss now easier to control and therefore more suited for mechanization.

11.1 Calculi for Automated Theorem Proving: Analytical Tableaux

11.1.1 Analytical Tableaux

Before we can start, we will need to recap some nomenclature on formulae.

Recap: Atoms and Literals

▷ **Definition 11.1.1** We call a formula atomic, or an atom, iff it does not contain connectives. We call a formula complex, iff it is not atomic.

▷ **Definition 11.1.2** We call a pair $\mathbf{A}^\alpha$ a labeled formula, if $\alpha \in \{\mathsf{T}, \mathsf{F}\}$. A labeled atom is called literal.

▷ **Definition 11.1.3** Let Φ be a set of formulae, then we use $\Phi^\alpha := \{\mathbf{A}^\alpha \mid \mathbf{A} \in \Phi\}$.

The idea about literals is that they are atoms (the simplest formulae) that carry around their intended truth value.

Now we will also review some propositional identities that will be useful later on. Some of them we have already seen, and some are new. All of them can be proven by simple truth table arguments.

Test Calculi: Tableaux and Model Generation

▷ Idea: instead of showing $\emptyset \vdash Th$, show $\neg Th \vdash trouble$ (use $\bot$ for trouble)

▷ **Example 11.1.4** Tableau Calculi try to construct models.

Tableau Refutation (Validity)	Model generation (Satisfiability)
$\models P \wedge Q \Rightarrow Q \wedge P$	$\models P \wedge (Q \vee \neg R) \wedge \neg Q$
$\begin{array}{c} P \wedge Q \Rightarrow Q \wedge P^{\mathsf{F}} \\ P \wedge Q^{\mathsf{T}} \\ Q \wedge P^{\mathsf{F}} \\ P^{\mathsf{T}} \\ Q^{\mathsf{T}} \\ P^{\mathsf{F}} \mid Q^{\mathsf{F}} \\ \bot \mid \bot \end{array}$	$\begin{array}{c} P \wedge (Q \vee \neg R) \wedge \neg Q^{\mathsf{T}} \\ P \wedge (Q \vee \neg R)^{\mathsf{T}} \\ \neg Q^{\mathsf{T}} \\ Q^{\mathsf{F}} \\ P^{\mathsf{T}} \\ Q \vee \neg R^{\mathsf{T}} \\ Q^{\mathsf{T}} \mid \neg R^{\mathsf{T}} \\ \bot \mid R^{\mathsf{F}} \end{array}$
No Model	Herbrand Model $\{P^{\mathsf{T}}, Q^{\mathsf{F}}, R^{\mathsf{F}}\}$ $\varphi := \{P \mapsto \mathsf{T}, Q \mapsto \mathsf{F}, R \mapsto \mathsf{F}\}$

Algorithm: Fully expand all possible tableaux, (no rule can be applied)

▷ ▹ Satisfiable, iff there are open branches (correspond to models)

Tableau calculi develop a formula in a tree-shaped arrangement that represents a case analysis on when a formula can be made true (or false). Therefore the formulae are decorated with exponents that hold the intended truth value.

On the left we have a refutation tableau that analyzes a negated formula (it is decorated with the intended truth value F). Both branches contain an elementary contradiction $\bot$.

On the right we have a model generation tableau, which analyzes a positive formula (it is decorated with the intended truth value T. This tableau uses the same rules as the refutation tableau, but makes a case analysis of when this formula can be satisfied. In this case we have a closed branch and an open one, which corresponds a model).

Now that we have seen the examples, we can write down the tableau rules formally.

Analytical Tableaux (Formal Treatment of $\mathcal{T}_0$)

▷ formula is analyzed in a tree to determine satisfiability

▷ branches correspond to valuations (models)

▷ one per connective

$$\frac{\mathbf{A} \wedge \mathbf{B}^{\mathsf{T}}}{\begin{array}{c}\mathbf{A}^{\mathsf{T}} \\ \mathbf{B}^{\mathsf{T}}\end{array}} \mathcal{T}_0\wedge \qquad \frac{\mathbf{A} \wedge \mathbf{B}^{\mathsf{F}}}{\mathbf{A}^{\mathsf{F}} \mid \mathbf{B}^{\mathsf{F}}} \mathcal{T}_0\vee \qquad \frac{\neg \mathbf{A}^{\mathsf{T}}}{\mathbf{A}^{\mathsf{F}}} \mathcal{T}_0{\neg}^{\mathsf{T}} \qquad \frac{\neg \mathbf{A}^{\mathsf{F}}}{\mathbf{A}^{\mathsf{T}}} \mathcal{T}_0{\neg}^{\mathsf{F}} \qquad \frac{\begin{array}{c}\mathbf{A}^{\alpha} \\ \mathbf{A}^{\beta}\end{array} \quad \alpha \neq \beta}{\bot} \mathcal{T}_0\mathrm{cut}$$

▷ Use rules exhaustively as long as they contribute new material

▷ **Definition 11.1.5** Call a tableau saturated, iff no rule applies, and a branch closed, iff it ends in $\bot$, else open. (open branches in saturated tableaux yield models)

▷ **Definition 11.1.6 ($\mathcal{T}_0$-Theorem/Derivability)** $\mathbf{A}$ is a $\mathcal{T}_0$-theorem ($\vdash_{\mathcal{T}_0} \mathbf{A}$), iff there is a closed tableau with $\mathbf{A}^{\mathsf{F}}$ at the root.

$\Phi \subseteq wff_o(\mathcal{V}_o)$ derives $\mathbf{A}$ in $\mathcal{T}_0$ ($\Phi \vdash_{\mathcal{T}_0} \mathbf{A}$), iff there is a closed tableau starting with $\mathbf{A}^\mathsf{F}$ and Φ^T.

These inference rules act on tableaux have to be read as follows: if the formulae over the line appear in a tableau branch, then the branch can be extended by the formulae or branches below the line. There are two rules for each primary connective, and a branch closing rule that adds the special symbol $\perp$ (for unsatisfiability) to a branch.

We use the tableau rules with the convention that they are only applied, if they contribute new material to the branch. This ensures termination of the tableau procedure for propositional logic (every rule eliminates one primary connective).

Definition 11.1.7 We will call a closed tableau with the signed formula $\mathbf{A}^\alpha$ at the root a tableau refutation for $\mathcal{A}^\alpha$.

The saturated tableau represents a full case analysis of what is necessary to give $\mathbf{A}$ the truth value α; since all branches are closed (contain contradictions) this is impossible.

Definition 11.1.8 We will call a tableau refutation for $\mathbf{A}^\mathsf{F}$ a tableau proof for $\mathbf{A}$, since it refutes the possibility of finding a model where $\mathbf{A}$ evaluates to F. Thus $\mathbf{A}$ must evaluate to T in all models, which is just our definition of validity.

Thus the tableau procedure can be used as a calculus for propositional logic. In contrast to the calculus in section ?sec.hilbert? it does not prove a theorem $\mathbf{A}$ by deriving it from a set of axioms, but it proves it by refuting its negation. Such calculi are called negative or test calculi. Generally negative calculi have computational advantages over positive ones, since they have a built-in sense of direction.

We have rules for all the necessary connectives (we restrict ourselves to $\wedge$ and $\neg$, since the others can be expressed in terms of these two via the propositional identities above. For instance, we can write $\mathbf{A} \vee \mathbf{B}$ as $\neg(\neg\mathbf{A} \wedge \neg\mathbf{B})$, and $\mathbf{A} \Rightarrow \mathbf{B}$ as $\neg\mathbf{A} \vee \mathbf{B}$,. . . .)

We will now look at an example. Following our introduction of propositional logic in in Example 10.1.9 we look at a formulation of propositional logic with fancy variable names. Note that love(mary, bill) is just a variable name like P or X, which we have used earlier.

A Valid Real-World Example

▷ **Example 11.1.9** *If Mary loves Bill and John loves Mary, then John loves Mary*

$$\begin{array}{c}
\mathrm{love}(\mathrm{mary},\mathrm{bill}) \wedge \mathrm{love}(\mathrm{john},\mathrm{mary}) \Rightarrow \mathrm{love}(\mathrm{john},\mathrm{mary})^\mathsf{F} \\
\neg(\neg\neg(\mathrm{love}(\mathrm{mary},\mathrm{bill}) \wedge \mathrm{love}(\mathrm{john},\mathrm{mary})) \wedge \neg\mathrm{love}(\mathrm{john},\mathrm{mary}))^\mathsf{F} \\
\neg\neg(\mathrm{love}(\mathrm{mary},\mathrm{bill}) \wedge \mathrm{love}(\mathrm{john},\mathrm{mary})) \wedge \neg\mathrm{love}(\mathrm{john},\mathrm{mary})^\mathsf{T} \\
\neg\neg(\mathrm{love}(\mathrm{mary},\mathrm{bill}) \wedge \mathrm{love}(\mathrm{john},\mathrm{mary}))^\mathsf{T} \\
\neg(\mathrm{love}(\mathrm{mary},\mathrm{bill}) \wedge \mathrm{love}(\mathrm{john},\mathrm{mary}))^\mathsf{F} \\
\mathrm{love}(\mathrm{mary},\mathrm{bill}) \wedge \mathrm{love}(\mathrm{john},\mathrm{mary})^\mathsf{T} \\
\neg\mathrm{love}(\mathrm{john},\mathrm{mary})^\mathsf{T} \\
\mathrm{love}(\mathrm{mary},\mathrm{bill})^\mathsf{T} \\
\mathrm{love}(\mathrm{john},\mathrm{mary})^\mathsf{T} \\
\mathrm{love}(\mathrm{john},\mathrm{mary})^\mathsf{F} \\
\perp
\end{array}$$

This is a closed tableau, so the $\mathrm{love}(\mathrm{mary},\mathrm{bill}) \wedge \mathrm{love}(\mathrm{john},\mathrm{mary}) \Rightarrow \mathrm{love}(\mathrm{john},\mathrm{mary})$ is a $\mathcal{T}_0$-theorem.

As we will see, $\mathcal{T}_0$ is sound and complete, so $\mathrm{love}(\mathrm{mary},\mathrm{bill}) \wedge \mathrm{love}(\mathrm{john},\mathrm{mary}) \Rightarrow \mathrm{love}(\mathrm{john},\mathrm{mary})$ is valid.

We could have used the entailment theorem (Corollary 10.4.6) here to show that *If Mary loves Bill and John loves Mary* entails *John loves Mary*. But there is a better way to show entailment: we directly use derivability in $\mathcal{T}_0$

Deriving Entailment in $\mathcal{T}_0$

▷ **Example 11.1.10** *Mary loves Bill* and *John loves Mary* together entail that *John loves Mary*

$$\begin{array}{c} \mathrm{love}(\mathrm{mary},\mathrm{bill})^{\mathsf{T}} \\ \mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{T}} \\ \mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{F}} \\ \bot \end{array}$$

This is a closed tableau, so the $\{\mathrm{love}(\mathrm{mary},\mathrm{bill}),\mathrm{love}(\mathrm{john},\mathrm{mary})\} \vdash_{\mathcal{T}_0} \mathrm{love}(\mathrm{john},\mathrm{mary})$, again, as $\mathcal{T}_0$ is sound and complete we have $\{\mathrm{love}(\mathrm{mary},\mathrm{bill}),\mathrm{love}(\mathrm{john},\mathrm{mary})\} \models \mathrm{love}(\mathrm{john},\mathrm{mary})$

Note: that we can also use the tableau calculus to try and show entailment (and fail). The nice thing is that the failed proof, we can see what went wrong.

A Falsifiable Real-World Example

▷ **Example 11.1.11** **If Mary loves Bill or John loves Mary, then John loves Mary*
Try proving the implication (this fails)

$$\begin{array}{c} (\mathrm{love}(\mathrm{mary},\mathrm{bill}) \vee \mathrm{love}(\mathrm{john},\mathrm{mary})) \Rightarrow \mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{F}} \\ \neg(\neg\neg(\mathrm{love}(\mathrm{mary},\mathrm{bill}) \vee \mathrm{love}(\mathrm{john},\mathrm{mary})) \wedge \neg\mathrm{love}(\mathrm{john},\mathrm{mary}))^{\mathsf{F}} \\ \neg\neg(\mathrm{love}(\mathrm{mary},\mathrm{bill}) \vee \mathrm{love}(\mathrm{john},\mathrm{mary})) \wedge \neg\mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{T}} \\ \neg\mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{T}} \\ \mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{F}} \\ \neg\neg(\mathrm{love}(\mathrm{mary},\mathrm{bill}) \vee \mathrm{love}(\mathrm{john},\mathrm{mary}))^{\mathsf{T}} \\ \neg(\mathrm{love}(\mathrm{mary},\mathrm{bill}) \vee \mathrm{love}(\mathrm{john},\mathrm{mary}))^{\mathsf{F}} \\ \mathrm{love}(\mathrm{mary},\mathrm{bill}) \vee \mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{T}} \\ \begin{array}{c|c} \mathrm{love}(\mathrm{mary},\mathrm{bill})^{\mathsf{T}} & \mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{T}} \\ & \bot \end{array} \end{array}$$

Indeed we can make $\mathcal{I}_\varphi(\mathrm{love}(\mathrm{mary},\mathrm{bill})) = \mathsf{T}$ but $\mathcal{I}_\varphi(\mathrm{love}(\mathrm{john},\mathrm{mary})) = \mathsf{F}$.

Obviously, the tableau above is saturated, but not closed, so it is not a tableau proof for our initial entailment conjecture. We have marked the literals on the open branch green, since they allow us to read of the conditions of the situation, in which the entailment fails to hold. As we intuitively argued above, this is the situation, where Mary loves Bill. In particular, the open branch gives us a variable assignment (marked in green) that satisfies the initial formula. In this case, *Mary loves Bill*, which is a situation, where the entailment fails.

Again, the derivability version is much simpler

Testing for Entailment in $\mathcal{T}_0$

▷ **Example 11.1.12** Does *Mary loves Bill or John loves Mary* entail that *John loves Mary*?

$$\begin{array}{c|c} \multicolumn{2}{c}{\mathrm{love}(\mathrm{mary}, \mathrm{bill}) \vee \mathrm{love}(\mathrm{john}, \mathrm{mary})^{\mathsf{T}}} \\ \multicolumn{2}{c}{\mathrm{love}(\mathrm{john}, \mathrm{mary})^{\mathsf{F}}} \\ \mathrm{love}(\mathrm{mary}, \mathrm{bill})^{\mathsf{T}} & \mathrm{love}(\mathrm{john}, \mathrm{mary})^{\mathsf{T}} \\ & \bot \end{array}$$

This saturated tableau has an open branch that shows that the interpretation with $\mathcal{I}_\varphi(\mathrm{love}(\mathrm{mary}, \mathrm{bill})) = \mathsf{T}$ but $\mathcal{I}_\varphi(\mathrm{love}(\mathrm{john}, \mathrm{mary})) = \mathsf{F}$ falsifies the derivability/entailment conjecture.

11.1.2 Practical Enhancements for Tableaux

Propositional Identities

▷ **Definition 11.1.13** Let T and F be new logical constants with $\mathcal{I}(T) = \mathsf{T}$ and $\mathcal{I}(F) = \mathsf{F}$ for all assignments φ.

▷ We have to following identities:

Name	for $\wedge$	for $\vee$
Idenpotence	$\varphi \wedge \varphi = \varphi$	$\varphi \vee \varphi = \varphi$
Identity	$\varphi \wedge T = \varphi$	$\varphi \vee F = \varphi$
Absorption I	$\varphi \wedge F = F$	$\varphi \vee T = T$
Commutativity	$\varphi \wedge \psi = \psi \wedge \varphi$	$\varphi \vee \psi = \psi \vee \varphi$
Associativity	$\varphi \wedge (\psi \wedge \theta) = (\varphi \wedge \psi) \wedge \theta$	$\varphi \vee (\psi \vee \theta) = (\varphi \vee \psi) \vee \theta$
Distributivity	$\varphi \wedge (\psi \vee \theta) = \varphi \wedge \psi \vee \varphi \wedge \theta$	$\varphi \vee \psi \wedge \theta = (\varphi \vee \psi) \wedge (\varphi \vee \theta)$
Absorption II	$\varphi \wedge (\varphi \vee \theta) = \varphi$	$\varphi \vee \varphi \wedge \theta = \varphi$
De Morgan's Laws	$\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$	$\neg(\varphi \vee \psi) = \neg\varphi \wedge \neg\psi$
Double negation	$\neg\neg\varphi = \varphi$	
Definitions	$\varphi \Rightarrow \psi = \neg\varphi \vee \psi$	$\varphi \Leftrightarrow \psi = (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

We have seen in the examples above that while it is possible to get by with only the connectives $\vee$ and $\neg$, it is a bit unnatural and tedious, since we need to eliminate the other connectives first. In this section, we will make the calculus less frugal by adding rules for the other connectives, without losing the advantage of dealing with a small calculus, which is good making statements about the calculus.

The main idea is to add the new rules as derived rules, i.e. inference rules that only abbreviate deductions in the original calculus. Generally, adding derived inference rules does not change the derivability relation of the calculus, and is therefore a safe thing to do. In particular, we will add the following rules to our tableau system.

We will convince ourselves that the first rule is a derived rule, and leave the other ones as an exercise.

Derived Rules of Inference

▷ **Definition 11.1.14** Let $\mathcal{C}$ be a calculus, a rule of inference $\frac{\mathbf{A}_1 \quad \cdots \quad \mathbf{A}_n}{\mathbf{C}}$ is called a derived inference rule in $\mathcal{C}$, iff there is a $\mathcal{C}$-proof of $\mathbf{A}_1, \ldots, \mathbf{A}_n \vdash \mathbf{C}$.

▷ **Definition 11.1.15** We have the following derived rules of inference

$$\frac{\mathbf{A} \Rightarrow \mathbf{B}^{\mathsf{T}}}{\mathbf{A}^{\mathsf{F}} \mid \mathbf{B}^{\mathsf{T}}} \qquad \frac{\mathbf{A} \Rightarrow \mathbf{B}^{\mathsf{F}}}{\begin{array}{c}\mathbf{A}^{\mathsf{T}} \\ \mathbf{B}^{\mathsf{F}}\end{array}} \qquad \frac{\begin{array}{c}\mathbf{A}^{\mathsf{T}} \\ \mathbf{A} \Rightarrow \mathbf{B}^{\mathsf{T}}\end{array}}{\mathbf{B}^{\mathsf{T}}} \qquad \begin{array}{c|c} \multicolumn{2}{c}{\mathbf{A}^{\mathsf{T}}} \\ \multicolumn{2}{c}{\mathbf{A} \Rightarrow \mathbf{B}^{\mathsf{T}}} \\ \multicolumn{2}{c}{\neg \mathbf{A} \vee \mathbf{B}^{\mathsf{T}}} \\ \multicolumn{2}{c}{\neg (\neg\neg \mathbf{A} \wedge \neg \mathbf{B})^{\mathsf{T}}} \\ \multicolumn{2}{c}{\neg\neg \mathbf{A} \wedge \neg \mathbf{B}^{\mathsf{F}}} \\ \neg\neg \mathbf{A}^{\mathsf{F}} & \neg \mathbf{B}^{\mathsf{F}} \\ \neg \mathbf{A}^{\mathsf{T}} & \mathbf{B}^{\mathsf{T}} \\ \mathbf{A}^{\mathsf{F}} & \\ \bot & \end{array}$$

$$\frac{\mathbf{A} \vee \mathbf{B}^{\mathsf{T}}}{\mathbf{A}^{\mathsf{T}} \mid \mathbf{B}^{\mathsf{T}}} \qquad \frac{\mathbf{A} \vee \mathbf{B}^{\mathsf{F}}}{\begin{array}{c}\mathbf{A}^{\mathsf{F}} \\ \mathbf{B}^{\mathsf{F}}\end{array}} \qquad \frac{\mathbf{A} \Leftrightarrow \mathbf{B}^{\mathsf{T}}}{\begin{array}{c|c}\mathbf{A}^{\mathsf{T}} & \mathbf{A}^{\mathsf{F}} \\ \mathbf{B}^{\mathsf{T}} & \mathbf{B}^{\mathsf{F}}\end{array}} \qquad \frac{\mathbf{A} \Leftrightarrow \mathbf{B}^{\mathsf{F}}}{\begin{array}{c|c}\mathbf{A}^{\mathsf{T}} & \mathbf{A}^{\mathsf{F}} \\ \mathbf{B}^{\mathsf{F}} & \mathbf{B}^{\mathsf{T}}\end{array}}$$

With these derived rules, theorem proving becomes quite efficient. With these rules, the tableau (?tab:firsttab?) would have the following simpler form:

Tableaux with derived Rules (example)

Example 11.1.16

$$\begin{array}{c} \mathrm{love}(\mathrm{mary}, \mathrm{bill}) \wedge \mathrm{love}(\mathrm{john}, \mathrm{mary}) \Rightarrow \mathrm{love}(\mathrm{john}, \mathrm{mary})^{\mathsf{F}} \\ \mathrm{love}(\mathrm{mary}, \mathrm{bill}) \wedge \mathrm{love}(\mathrm{john}, \mathrm{mary})^{\mathsf{T}} \\ \mathrm{love}(\mathrm{john}, \mathrm{mary})^{\mathsf{F}} \\ \mathrm{love}(\mathrm{mary}, \mathrm{bill})^{\mathsf{T}} \\ \mathrm{love}(\mathrm{john}, \mathrm{mary})^{\mathsf{T}} \\ \bot \end{array}$$

Another thing that was awkward in (?tab:firsttab?) was that we used a proof for an implication to prove logical consequence. Such tests are necessary for instance, if we want to check consistency
EdN:4 or informativity of new sentences[4]. Consider for instance a discourse $\Delta = \mathbf{D}^1, \ldots, \mathbf{D}^n$, where n is large. To test whether a hypothesis $\mathcal{H}$ is a consequence of Δ ($\Delta \models \mathbf{H}$) we need to show that $\mathbf{C} := (\mathbf{D}^1 \wedge \ldots) \wedge \mathbf{D}^n \Rightarrow \mathbf{H}$ is valid, which is quite tedious, since $\mathcal{C}$ is a rather large formula, e.g. if Δ is a 300 page novel. Moreover, if we want to test entailment of the form ($\Delta \models \mathbf{H}$) often, – for instance to test the informativity and consistency of every new sentence $\mathbf{H}$, then successive Δs will overlap quite significantly, and we will be doing the same inferences all over again; the entailment check is not incremental.

Fortunately, it is very simple to get an incremental procedure for entailment checking in the model-generation-based setting: To test whether $\Delta \models \mathbf{H}$, where we have interpreted Δ in a model generation tableau $\mathcal{T}$, just check whether the tableau closes, if we add $\neg \mathbf{H}$ to the open branches.
EdN:5 Indeed, if the tableau closes, then $\Delta \wedge \neg \mathbf{H}$ is unsatisfiable, so $\neg ((\Delta \wedge \neg \mathbf{H}))$ is valid[5], but this is equivalent to $\Delta \Rightarrow \mathbf{H}$, which is what we wanted to show.

Example 11.1.17 Consider for instance the following entailment in natural language.

Mary loves Bill. John loves Mary $\models$ *John loves Mary*

[4]EdNote: add reference to presupposition stuff
[5]EdNote: Fix precedence of negation

[6] We obtain the tableau EdN:6

$$\begin{array}{c} \mathrm{love}(\mathrm{mary},\mathrm{bill})^{\mathsf{T}} \\ \mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{T}} \\ \neg(\mathrm{love}(\mathrm{john},\mathrm{mary}))^{\mathsf{T}} \\ \mathrm{love}(\mathrm{john},\mathrm{mary})^{\mathsf{F}} \\ \bot \end{array}$$

which shows us that the conjectured entailment relation really holds.

11.1.3 Soundness and Termination of Tableaux

As always we need to convince ourselves that the calculus is sound, otherwise, tableau proofs do not guarantee validity, which we are after. Since we are now in a refutation setting we cannot just show that the inference rules preserve validity: we care about unsatisfiability (which is the dual notion to validity), as we want to show the initial labeled formula to be unsatisfiable. Before we can do this, we have to ask ourselves, what it means to be (un)-satisfiable for a labeled formula or a tableau.

Soundness (Tableau)

- ▷ Idea: A test calculus is sound, iff it preserves satisfiability and the goal formulae are unsatisfiable.
- ▷ **Definition 11.1.18** A labeled formula $\mathbf{A}^{\alpha}$ is valid under φ, iff $\mathcal{I}_{\varphi}(\mathbf{A}) = \alpha$.
- ▷ **Definition 11.1.19** A tableau $\mathcal{T}$ is satisfiable, iff there is a satisfiable branch $\mathcal{P}$ in $\mathcal{T}$, i.e. if the set of formulae in $\mathcal{P}$ is satisfiable.
- ▷ **Lemma 11.1.20** *Tableau rules transform satisfiable tableaux into satisfiable ones.*
- ▷ **Theorem 11.1.21 (Soundness)** *A set Φ of propositional formulae is valid, if there is a closed tableau $\mathcal{T}$ for Φ^{F}.*
- ▷ Proof: by contradiction: Suppose Φ is not valid.

  P.1 then the initial tableau is satisfiable (Φ^{F} satisfiable)

  P.2 so $\mathcal{T}$ is satisfiable, by Lemma 11.1.20.

  P.3 there is a satisfiable branch (by definition)

  P.4 but all branches are closed ($\mathcal{T}$ closed)

  □

Thus we only have to prove Lemma 11.1.20, this is relatively easy to do. For instance for the first rule: if we have a tableau that contains $\mathbf{A} \wedge \mathbf{B}^{\mathsf{T}}$ and is satisfiable, then it must have a satisfiable branch. If $\mathbf{A} \wedge \mathbf{B}^{\mathsf{T}}$ is not on this branch, the tableau extension will not change satisfiability, so we can assue that it is on the satisfiable branch and thus $\mathcal{I}_{\varphi}(\mathbf{A} \wedge \mathbf{B}) = \mathsf{T}$ for some variable assignment φ. Thus $\mathcal{I}_{\varphi}(\mathbf{A}) = \mathsf{T}$ and $\mathcal{I}_{\varphi}(\mathbf{B}) = \mathsf{T}$, so after the extension (which adds the formulae $\mathbf{A}^{\mathsf{T}}$ and $\mathbf{B}^{\mathsf{T}}$ to the branch), the branch is still satisfiable. The cases for the other rules are similar.

The next result is a very important one, it shows that there is a procedure (the tableau procedure) that will always terminate and answer the question whether a given propositional formula is valid

[6] EdNote: need to mark up the embedding of NL strings into Math

or not. This is very important, since other logics (like the often-studied first-order logic) does not enjoy this property.

Termination for Tableaux

▷ **Lemma 11.1.22** *The tableau procedure terminates, i.e. after a finite set of rule applications, it reaches a tableau, so that applying the tableau rules will only add labeled formulae that are already present on the branch.*

▷ Let us call a labeled formulae $\mathbf{A}^{\alpha}$ worked off in a tableau $\mathcal{T}$, if a tableau rule has already been applied to it.

▷ Proof:

P.1 It is easy to see that applying rules to worked off formulae will only add formulae that are already present in its branch.

P.2 Let $\mu(\mathcal{T})$ be the number of connectives in labeled formulae in $\mathcal{T}$ that are not worked off.

P.3 Then each rule application to a labeled formula in $\mathcal{T}$ that is not worked off reduces $\mu(\mathcal{T})$ by at least one. (inspect the rules)

P.4 At some point the tableau only contains worked off formulae and literals.

P.5 Since there are only finitely many literals in $\mathcal{T}$, so we can only apply the tableau cut rule a finite number of times. □

The Tableau calculus basically computes the disjunctive normal form: every branch is a disjunct that is a conjunct of literals. The method relies on the fact that a DNF is unsatisfiable, iff each monomial is, i.e. iff each branch contains a contradiction in form of a pair of complementary literals.

11.2 Resolution for Propositional Logic

The next calculus is a test calculus based on the conjunctive normal form. In contrast to the tableau method, it does not compute the normal form as it goes along, but has a pre-processing step that does this and a single inference rule that maintains the normal form. The goal of this calculus is to derive the empty clause (the empty disjunction), which is unsatisfiable.

Another Test Calculus: Resolution

▷ **Definition 11.2.1** A clause is a disjunction of literals. We will use □ for the empty disjunction (no disjuncts) and call it the empty clause.

▷ **Definition 11.2.2 (Resolution Calculus)** The resolution calculus operates a clause sets via a single inference rule:

$$\frac{P^{\mathsf{T}} \vee \mathbf{A} \quad P^{\mathsf{F}} \vee \mathbf{B}}{\mathbf{A} \vee \mathbf{B}}$$

This rule allows to add the clause below the line to a clause set which contains the two clauses above.

▷ **Definition 11.2.3 (Resolution Refutation)** Let S be a clause set, and $\mathcal{D}\colon S \vdash_{\mathcal{R}} T$ a $\mathcal{R}$ derivation then we call $\mathcal{D}$ resolution refutation, iff $\Box \in T$.

A calculus for CNF Transformation

▷ **Definition 11.2.4 (Transformation into Conjunctive Normal Form)** The CNF transformation calculus $\mathcal{CNF}$ consists of the following four inference rules on clause sets.

$$\frac{\mathbf{C} \vee (\mathbf{A} \vee \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}} \quad \frac{\mathbf{C} \vee (\mathbf{A} \vee \mathbf{B})^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{F}}} \qquad \frac{\mathbf{C} \vee \neg \mathbf{A}^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}}} \quad \frac{\mathbf{C} \vee \neg \mathbf{A}^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}}$$

▷ **Definition 11.2.5** We write $CNF(\mathbf{A})$ for the set of all clauses derivable from $\mathbf{A}^{\mathsf{F}}$ via the rules above.

▷ **Definition 11.2.6 (Resolution Proof)** We call a resolution refutation $\mathcal{P}\colon CNF(\mathbf{A}) \vdash_{\mathcal{R}} T$ a resolution sproof for $\mathbf{A} \in wff_o(\mathcal{V}_o)$.

Note: Note that the **C**-terms in the definition of the resolution calculus are necessary, since we assumed that the assumptions of the inference rule must match full formulae. The **C**-terms are used with the convention that they are optional. So that we can also simplify $(\mathbf{A} \vee \mathbf{B})^{\mathsf{T}}$ to $\mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}$.

The background behind this notation is that $\mathbf{A}$ and $T \vee \mathbf{A}$ are equivalent for any $\mathbf{A}$. That allows us to interpret the **C**-terms in the assumptions as T and thus leave them out.

The resolution calculus as we have formulated it here is quite frugal; we have left out rules for the connectives $\vee$, $\Rightarrow$, and $\Leftrightarrow$, relying on the fact that formulae containing these connectives can be translated into ones without before CNF transformation. The advantage of having a calculus with few inference rules is that we can prove meta-properties like soundness and completeness with less effort (these proofs usually require one case per inference rule). On the other hand, adding specialized inference rules makes proofs shorter and more readable.

Fortunately, there is a way to have your cake and eat it. Derived inference rules have the property that they are formally redundant, since they do not change the expressive power of the calculus. Therefore we can leave them out when proving meta-properties, but include them when actually using the calculus.

Derived Rules of Inference

▷ **Definition 11.2.7** Let $\mathcal{C}$ be a calculus, a rule of inference $\dfrac{\mathbf{A}_1 \quad \ldots \quad \mathbf{A}_n}{\mathbf{C}}$ is called a derived inference rule in $\mathcal{C}$, iff there is a $\mathcal{C}$-proof of $\mathbf{A}_1, \ldots, \mathbf{A}_n \vdash \mathbf{C}$.

▷ **Example 11.2.8**

$$\dfrac{\dfrac{\dfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee (\neg \mathbf{A} \vee \mathbf{B})^{\mathsf{T}}}}{\mathbf{C} \vee \neg \mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}} \quad \mapsto \quad \dfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}}$$

▷ Others:

$$\dfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}} \quad \dfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{F}}} \qquad \dfrac{\mathbf{C} \vee \mathbf{A} \wedge \mathbf{B}^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{T}}} \quad \dfrac{\mathbf{C} \vee \mathbf{A} \wedge \mathbf{B}^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{F}}}$$

With these derived rules, theorem proving becomes quite efficient. To get a better understanding of the calculus, we look at an example: we prove an axiom of the Hilbert Calculus we have studied above.

Example: Proving Axiom S

▷ **Example 11.2.9** Clause Normal Form transformation

$$\dfrac{\dfrac{\dfrac{(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R^{\mathsf{F}}}{P \Rightarrow Q \Rightarrow R^{\mathsf{T}}; (P \Rightarrow Q) \Rightarrow P \Rightarrow R^{\mathsf{F}}}}{P^{\mathsf{F}} \vee (Q \Rightarrow R)^{\mathsf{T}}; P \Rightarrow Q^{\mathsf{T}}; P \Rightarrow R^{\mathsf{F}}}}{P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}; P^{\mathsf{F}} \vee Q^{\mathsf{T}}; P^{\mathsf{T}}; R^{\mathsf{F}}}$$

$CNF = \{P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}\ ,\ P^{\mathsf{F}} \vee Q^{\mathsf{T}}\ ,\ P^{\mathsf{T}}\ ,\ R^{\mathsf{F}}\}$

▷ **Example 11.2.10** Resolution Proof

1	$P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}$	initial
2	$P^{\mathsf{F}} \vee Q^{\mathsf{T}}$	initial
3	P^{T}	initial
4	R^{F}	initial
5	$P^{\mathsf{F}} \vee Q^{\mathsf{F}}$	resolve 1.3 with 4.1
6	Q^{F}	resolve 5.1 with 3.1
7	P^{F}	resolve 2.2 with 6.1
8	$\square$	resolve 7.1 with 3.1

11.3 Completenes Proofs (Optional Material, not Exam-Relevant)

The next step is to analyze the two calculi for completeness. For that we will first give ourselves a very powerful tool: the model "existence theorem", which encapsulates the model-theoretic part of completeness theorems. With that, completeness proofs – which are quite tedious otherwise – become a breeze.

11.3.1 Abstract Consistency and Model Existence

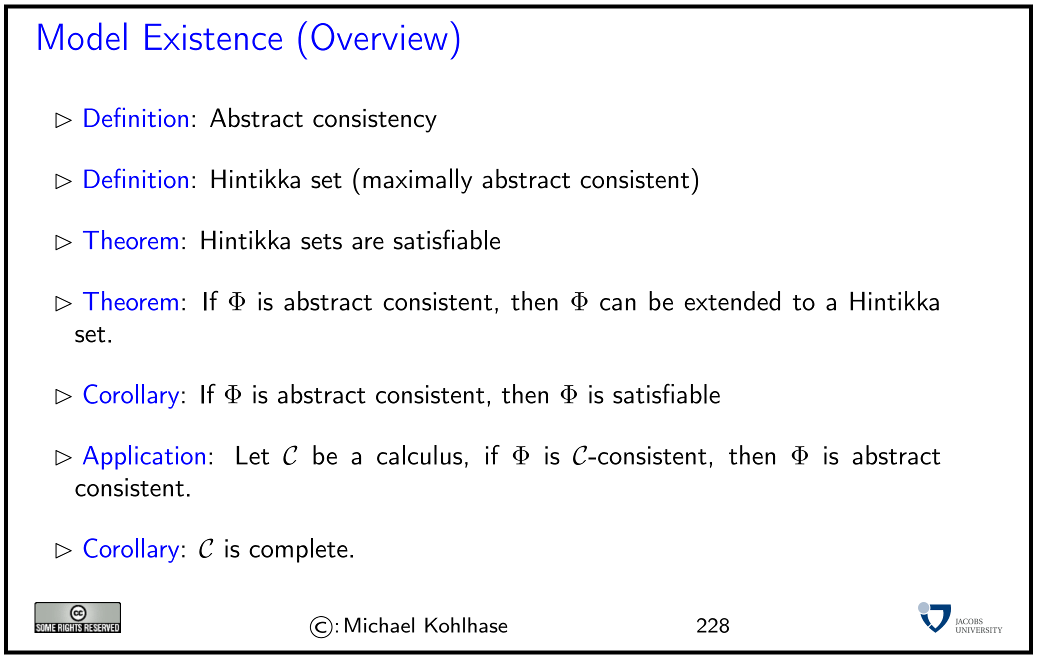
We will now come to an important tool in the theoretical study of reasoning calculi: the "abstract consistency"/"model existence" method. This method for analyzing calculi was developed by Jaako Hintikka, Raymond Smullyan, and Peter Andrews in 1950-1970 as an encapsulation of similar constructions that were used in completeness arguments in the decades before. The basis for this method is Smullyan's Observation [Smu63] that completeness proofs based on Hintikka sets only certain properties of consistency and that with little effort one can obtain a generalization "Smullyan's Unifying Principle".

The basic intuition for this method is the following: typically, a logical system $\mathcal{S} = \langle \mathcal{L}, \mathcal{K}, \models \rangle$ has multiple calculi, human-oriented ones like the natural deduction calculi and machine-oriented ones like the automated theorem proving calculi. All of these need to be analyzed for completeness (as a basic quality assurance measure).

A completeness proof for a calculus $\mathcal{C}$ for $\mathcal{S}$ typically comes in two parts: one analyzes $\mathcal{C}$-consistency (sets that cannot be refuted in $\mathcal{C}$), and the other construct $\mathcal{K}$-models for $\mathcal{C}$-consistent sets.

In this situtation the "abstract consistency"/"model existence" method encapsulates the model construction process into a meta-theorem: the "model existence" theorem. This provides a set of syntactic ("abstract consistency") conditions for calculi that are sufficient to construct models.

With the model existence theorem it suffices to show that $\mathcal{C}$-consistency is an abstract consistency property (a purely syntactic task that can be done by a $\mathcal{C}$-proof transformation argument) to obtain a completeness result for $\mathcal{C}$.



Model Existence (Overview)

- ▷ Definition: Abstract consistency
- ▷ Definition: Hintikka set (maximally abstract consistent)
- ▷ Theorem: Hintikka sets are satisfiable
- ▷ Theorem: If Φ is abstract consistent, then Φ can be extended to a Hintikka set.
- ▷ Corollary: If Φ is abstract consistent, then Φ is satisfiable
- ▷ Application: Let $\mathcal{C}$ be a calculus, if Φ is $\mathcal{C}$-consistent, then Φ is abstract consistent.
- ▷ Corollary: $\mathcal{C}$ is complete.

The proof of the model existence theorem goes via the notion of a Hintikka set, a set of formulae with very strong syntactic closure properties, which allow to read off models. Jaako Hintikka's original idea for completeness proofs was that for every complete calculus $\mathcal{C}$ and every $\mathcal{C}$-consistent set one can induce a Hintikka set, from which a model can be constructed. This can be considered as a first model existence theorem. However, the process of obtaining a Hintikka set for a set $\mathcal{C}$-consistent set Φ of sentences usually involves complicated calculus-dependent constructions.

In this situation, Raymond Smullyan was able to formulate the sufficient conditions for the existence of Hintikka sets in the form of "abstract consistency properties" by isolating the calculus-independent parts of the Hintikka set construction. His technique allows to reformulate Hintikka sets as maximal elements of abstract consistency classes and interpret the Hintikka set construction as a maximizing limit process.

To carry out the "model-existence"/"abstract consistency" method, we will first have to look at the notion of consistency.

Consistency and refutability are very important notions when studying the completeness for calculi; they form syntactic counterparts of satisfiability.

Consistency

- ▷ Let $\mathcal{C}$ be a calculus
- ▷ **Definition 11.3.1** Φ is called $\mathcal{C}$-refutable, if there is a formula $\mathbf{B}$, such that $\Phi \vdash_{\mathcal{C}} \mathbf{B}$ and $\Phi \vdash_{\mathcal{C}} \neg \mathbf{B}$.
- ▷ **Definition 11.3.2** We call a pair $\mathbf{A}$ and $\neg \mathbf{A}$ a contradiction.
- ▷ So a set Φ is $\mathcal{C}$-refutable, if $\mathcal{C}$ can derive a contradiction from it.
- ▷ **Definition 11.3.3** Φ is called $\mathcal{C}$-consistent, iff there is a formula $\mathbf{B}$, that is not derivable from Φ in $\mathcal{C}$.
- ▷ **Definition 11.3.4** We call a calculus $\mathcal{C}$ reasonable, iff implication elimination and conjunction introduction are admissible in $\mathcal{C}$ and $\mathbf{A} \wedge \neg \mathbf{A} \Rightarrow \mathbf{B}$ is a $\mathcal{C}$-theorem.
- ▷ **Theorem 11.3.5** *$\mathcal{C}$-inconsistency and $\mathcal{C}$-refutability coincide for reasonable calculi.*

It is very important to distinguish the syntactic $\mathcal{C}$-refutability and $\mathcal{C}$-consistency from satisfiability, which is a property of formulae that is at the heart of semantics. Note that the former specify the calculus (a syntactic device) while the latter does not. In fact we should actually say $\mathcal{S}$-satisfiability, where $\mathcal{S} = \langle \mathcal{L}, \mathcal{K}, \models \rangle$ is the current logical system.

Even the word "contradiction" has a syntactical flavor to it, it translates to "saying against each other" from its latin root.

Abstract Consistency

- ▷ **Definition 11.3.6** Let ∇ be a family of sets. We call ∇ closed under subset s, iff for each $\Phi \in \nabla$, all subsets $\Psi \subseteq \Phi$ are elements of ∇.
- ▷ **Notation 11.3.7** We will use $\Phi * \mathbf{A}$ for $\Phi \cup \{\mathbf{A}\}$.
- ▷ **Definition 11.3.8** A family ∇ of sets of propositional formulae is called an abstract consistency class, iff it is closed under subsets, and for each $\Phi \in \nabla$

  ∇_c) $P \notin \Phi$ or $\neg P \notin \Phi$ for $P \in \mathcal{V}_o$

  $\nabla_\neg$) $\neg \neg \mathbf{A} \in \Phi$ implies $\Phi * \mathbf{A} \in \nabla$

  $\nabla_\vee$) $(\mathbf{A} \vee \mathbf{B}) \in \Phi$ implies $\Phi * \mathbf{A} \in \nabla$ or $\Phi * \mathbf{B} \in \nabla$

  $\nabla_\wedge$) $\neg (\mathbf{A} \vee \mathbf{B}) \in \Phi$ implies $(\Phi \cup \{\neg \mathbf{A}, \neg \mathbf{B}\}) \in \nabla$
- ▷ **Example 11.3.9** The empty set is an abstract consistency class
- ▷ **Example 11.3.10** The set $\{\emptyset, \{Q\}, \{P \vee Q\}, \{P \vee Q, Q\}\}$ is an abstract consistency class
- ▷ **Example 11.3.11** The family of satisfiable sets is an abstract consistency class.

So a family of sets (we call it a family, so that we do not have to say "set of sets" and we can distinguish the levels) is an abstract consistency class, iff if fulfills five simple conditions, of which the last three are closure conditions.

Think of an abstract consistency class as a family of "consistent" sets (e.g. $\mathcal{C}$-consistent for some calculus $\mathcal{C}$), then the properties make perfect sense: They are naturally closed under subsets — if we cannot derive a contradiction from a large set, we certainly cannot from a subset, furthermore,

∇_c) If both $P \in \Phi$ and $\neg P \in \Phi$, then Φ cannot be "consistent".

$\nabla_\neg$) If we cannot derive a contradiction from Φ with $\neg\neg\mathbf{A} \in \Phi$ then we cannot from $\Phi * \mathbf{A}$, since they are logically equivalent.

The other two conditions are motivated similarly.

Compact Collections

▷ **Definition 11.3.12** We call a collection ∇ of sets compact, iff for any set Φ we have
$\Phi \in \nabla$, iff $\Psi \in \nabla$ for every finite subset Ψ of Φ.

▷ **Lemma 11.3.13** *If ∇ is compact, then ∇ is closed under subsets.*

▷ Proof:

P.1 Suppose $S \subseteq T$ and $T \in \nabla$.

P.2 Every finite subset A of S is a finite subset of T.

P.3 As ∇ is compact, we know that $A \in \nabla$.

P.4 Thus $S \in \nabla$. □

The property of being closed under subsets is a "downwards-oriented" property: We go from large sets to small sets, compactness (the interesting direction anyways) is also an "upwards-oriented" property. We can go from small (finite) sets to large (infinite) sets. The main application for the compactness condition will be to show that infinite sets of formulae are in a family ∇ by testing all their finite subsets (which is much simpler).

We will carry out the proof here, since it gives us practice in dealing with the abstract consistency properties.

We now come to a very technical condition that will allow us to carry out a limit construction in the Hintikka set extension argument later.

Compact Collections

▷ **Definition 11.3.14** We call a collection ∇ of sets compact, iff for any set Φ we have
$\Phi \in \nabla$, iff $\Psi \in \nabla$ for every finite subset Ψ of Φ.

▷ **Lemma 11.3.15** *If ∇ is compact, then ∇ is closed under subsets.*

▷ Proof:

P.1 Suppose $S \subseteq T$ and $T \in \nabla$.

P.2 Every finite subset A of S is a finite subset of T.

P.3 As ∇ is compact, we know that $A \in \nabla$.

P.4 Thus $S \in \nabla$. $\Box$

The property of being closed under subsets is a "downwards-oriented" property: We go from large sets to small sets, compactness (the interesting direction anyways) is also an "upwards-oriented" property. We can go from small (finite) sets to large (infinite) sets. The main application for the compactness condition will be to show that infinite sets of formulae are in a family ∇ by testing all their finite subsets (which is much simpler).

The main result here is that abstract consistency classes can be extended to compact ones. The proof is quite tedious, but relatively straightforward. It allows us to assume that all abstract consistency classes are compact in the first place (otherwise we pass to the compact extension).

Compact Abstract Consistency Classes

▷ **Lemma 11.3.16** *Any abstract consistency class can be extended to a compact one.*

▷ Proof:

P.1 We choose $\nabla' := \{\Phi \subseteq wff_o(\mathcal{V}_o) \mid \text{every finite subset of } \Phi \text{ is in } \nabla\}$.

P.2 Now suppose that $\Phi \in \nabla$. ∇ is closed under subsets, so every finite subset of Φ is in ∇ and thus $\Phi \in \nabla'$. Hence $\nabla \subseteq \nabla'$.

P.3 Next let us show that each ∇' is compact.

P.3.1 Suppose $\Phi \in \nabla'$ and Ψ is an arbitrary finite subset of Φ.

P.3.2 By definition of ∇' all finite subsets of Φ are in ∇ and therefore $\Psi \in \nabla'$.

P.3.3 Thus all finite subsets of Φ are in ∇' whenever Φ is in ∇'.

P.3.4 On the other hand, suppose all finite subsets of Φ are in ∇'.

P.3.5 Then by the definition of ∇' the finite subsets of Φ are also in ∇, so $\Phi \in \nabla'$. Thus ∇' is compact.

P.4 Note that ∇' is closed under subsets by the Lemma above.

P.5 Now we show that if ∇ satisfies ∇_*, then ∇' satisfies ∇_*.

P.5.1 To show ∇_c, let $\Phi \in \nabla'$ and suppose there is an atom $\mathbf{A}$, such that $\{\mathbf{A}, \neg\mathbf{A}\} \subseteq \Phi$. Then $\{\mathbf{A}, \neg\mathbf{A}\} \in \nabla$ contradicting ∇_c.

P.5.2 To show $\nabla_\neg$, let $\Phi \in \nabla'$ and $\neg\neg\mathbf{A} \in \Phi$, then $\Phi * \mathbf{A} \in \nabla'$.

P.5.2.1 Let Ψ be any finite subset of $\Phi * \mathbf{A}$, and $\Theta := (\Psi \backslash \{\mathbf{A}\}) * \neg\neg\mathbf{A}$.

P.5.2.2 Θ is a finite subset of Φ, so $\Theta \in \nabla$.

P.5.2.3 Since ∇ is an abstract consistency class and $\neg\neg\mathbf{A} \in \Theta$, we get $\Theta * \mathbf{A} \in \nabla$ by $\nabla_\neg$.

P.5.2.4 We know that $\Psi \subseteq \Theta * \mathbf{A}$ and ∇ is closed under subsets, so $\Psi \in \nabla$.

P.5.2.5 Thus every finite subset Ψ of $\Phi * \mathbf{A}$ is in ∇ and therefore by definition $\Phi * \mathbf{A} \in \nabla'$.

P.5.3 the other cases are analogous to $\nabla_{\neg}$. □

Hintikka sets are sets of sentences with very strong analytic closure conditions. These are motivated as maximally consistent sets i.e. sets that already contain everything that can be consistently added to them.

∇-Hintikka Set

▷ **Definition 11.3.17** Let ∇ be an abstract consistency class, then we call a set $\mathcal{H} \in \nabla$ a ∇-Hintikka Set, iff $\mathcal{H}$ is maximal in ∇, i.e. for all $\mathbf{A}$ with $\mathcal{H} * \mathbf{A} \in \nabla$ we already have $\mathbf{A} \in \mathcal{H}$.

▷ **Theorem 11.3.18 (Hintikka Properties)** *Let ∇ be an abstract consistency class and $\mathcal{H}$ be a ∇-Hintikka set, then*

$\mathcal{H}_c$) For all $\mathbf{A} \in wff_o(\mathcal{V}_o)$ we have $\mathbf{A} \notin \mathcal{H}$ or $\neg \mathbf{A} \notin \mathcal{H}$

$\mathcal{H}_{\neg}$) If $\neg\neg \mathbf{A} \in \mathcal{H}$ then $\mathbf{A} \in \mathcal{H}$

$\mathcal{H}_{\vee}$) If $(\mathbf{A} \vee \mathbf{B}) \in \mathcal{H}$ then $\mathbf{A} \in \mathcal{H}$ or $\mathbf{B} \in \mathcal{H}$

$\mathcal{H}_{\wedge}$) If $\neg(\mathbf{A} \vee \mathbf{B}) \in \mathcal{H}$ then $\neg \mathbf{A}, \neg \mathbf{B} \in \mathcal{H}$

Proof:

▷ **P.1** We prove the properties in turn

P.1.1 $\mathcal{H}_c$: by induction on the structure of $\mathbf{A}$

P.1.1.1.1 $\mathbf{A} \in \mathcal{V}_o$: Then $\mathbf{A} \notin \mathcal{H}$ or $\neg \mathbf{A} \notin \mathcal{H}$ by ∇_c.

P.1.1.1.2 $\mathbf{A} = \neg \mathbf{B}$:

P.1.1.1.2.1 Let us assume that $\neg \mathbf{B} \in \mathcal{H}$ and $\neg\neg \mathbf{B} \in \mathcal{H}$,

P.1.1.1.2.2 then $\mathcal{H} * \mathbf{B} \in \nabla$ by $\nabla_{\neg}$, and therefore $\mathbf{B} \in \mathcal{H}$ by maximality.

P.1.1.1.2.3 So both $\mathbf{B}$ and $\neg \mathbf{B}$ are in $\mathcal{H}$, which contradicts the inductive hypothesis. □

P.1.1.1.3 $\mathbf{A} = \mathbf{B} \vee \mathbf{C}$: similar to the previous case: □

□

P.1.2 We prove $\mathcal{H}_{\neg}$ by maximality of $\mathcal{H}$ in ∇.:

P.1.2.1 If $\neg\neg \mathbf{A} \in \mathcal{H}$, then $\mathcal{H} * \mathbf{A} \in \nabla$ by $\nabla_{\neg}$.

P.1.2.2 The maximality of $\mathcal{H}$ now gives us that $\mathbf{A} \in \mathcal{H}$. □

P.1.3 other $\mathcal{H}_*$ are similar:

The following theorem is one of the main results in the "abstract consistency"/"model existence" method. For any abstract consistent set Φ it allows us to construct a Hintikka set $\mathcal{H}$ with $\Phi \in \mathcal{H}$.

Extension Theorem

▷ **Theorem 11.3.19** *If ∇ is an abstract consistency class and $\Phi \in \nabla$, then there is a ∇-Hintikka set $\mathcal{H}$ with $\Phi \subseteq \mathcal{H}$.*

▷ Proof:

P.1 Wlog. we assume that ∇ is compact (otherwise pass to compact extension)

P.2 We choose an enumeration $\mathbf{A}^1, \mathbf{A}^2, \ldots$ of the set $\mathit{wff}_o(\mathcal{V}_o)$

P.3 and construct a sequence of sets H^i with $H^0 := \Phi$ and

$$H^{n+1} := \begin{cases} H^n & \text{if } H^n * \mathbf{A}^n \notin \nabla \\ H^n * \mathbf{A}^n & \text{if } H^n * \mathbf{A}^n \in \nabla \end{cases}$$

P.4 Note that all $H^i \in \nabla$, choose $\mathcal{H} := \bigcup_{i \in \mathbb{N}} H^i$

P.5 $\Psi \subseteq \mathcal{H}$ finite implies there is a $j \in \mathbb{N}$ such that $\Psi \subseteq H^j$,

P.6 so $\Psi \in \nabla$ as ∇ closed under subsets and $\mathcal{H} \in \nabla$ as ∇ is compact.

P.7 Let $\mathcal{H} * \mathbf{B} \in \nabla$, then there is a $j \in \mathbb{N}$ with $\mathbf{B} = \mathbf{A}^j$, so that $\mathbf{B} \in H^{j+1}$ and $H^{j+1} \subseteq \mathcal{H}$

P.8 Thus $\mathcal{H}$ is ∇-maximal □

Note that the construction in the proof above is non-trivial in two respects. First, the limit construction for $\mathcal{H}$ is not executed in our original abstract consistency class ∇, but in a suitably extended one to make it compact — the original would not have contained $\mathcal{H}$ in general. Second, the set $\mathcal{H}$ is not unique for Φ, but depends on the choice of the enumeration of $\mathit{wff}_o(\mathcal{V}_o)$. If we pick a different enumeration, we will end up with a different $\mathcal{H}$. Say if $\mathbf{A}$ and $\neg\mathbf{A}$ are both ∇-consistent[7] with Φ, then depending on which one is first in the enumeration $\mathcal{H}$, will contain that one; with all the consequences for subsequent choices in the construction process. EdN:7

Valuation

▷ **Definition 11.3.20** A function $\nu \colon \mathit{wff}_o(\mathcal{V}_o) \to \mathcal{D}_o$ is called a valuation, iff

- ▷ $\nu(\neg\mathbf{A}) = \mathsf{T}$, iff $\nu(\mathbf{A}) = \mathsf{F}$
- ▷ $\nu(\mathbf{A} \vee \mathbf{B}) = \mathsf{T}$, iff $\nu(\mathbf{A}) = \mathsf{T}$ or $\nu(\mathbf{B}) = \mathsf{T}$

▷ **Lemma 11.3.21** *If $\nu \colon \mathit{wff}_o(\mathcal{V}_o) \to \mathcal{D}_o$ is a valuation and $\Phi \subseteq \mathit{wff}_o(\mathcal{V}_o)$ with $\nu(\Phi) = \{\mathsf{T}\}$, then Φ is satisfiable.*

▷ Proof Sketch: $\nu|_{\mathcal{V}_o} \colon \mathcal{V}_o \to \mathcal{D}_o$ is a satisfying variable assignment. □

▷ **Lemma 11.3.22** *If $\varphi \colon \mathcal{V}_o \to \mathcal{D}_o$ is a variable assignment, then $\mathcal{I}_\varphi \colon \mathit{wff}_o(\mathcal{V}_o) \to \mathcal{D}_o$ is a valuation.*

Now, we only have to put the pieces together to obtain the model existence theorem we are after.

Model Existence

▷ **Lemma 11.3.23 (Hintikka-Lemma)** *If ∇ is an abstract consistency class*

[7] EdNote: introduce this above

and $\mathcal{H}$ a ∇-Hintikka set, then $\mathcal{H}$ is satisfiable.

▷ Proof:

P.1 We define $\nu(\mathbf{A}) := \mathsf{T}$, iff $\mathbf{A} \in \mathcal{H}$

P.2 then ν is a valuation by the Hintikka properties

P.3 and thus $\nu|_{\mathcal{V}_o}$ is a satisfying assignment. □

▷ **Theorem 11.3.24 (Model Existence)** *If ∇ is an abstract consistency class and $\Phi \in \nabla$, then Φ is satisfiable.*

Proof:

▷ **P.1** There is a ∇-Hintikka set $\mathcal{H}$ with $\Phi \subseteq \mathcal{H}$ (Extension Theorem)

We know that $\mathcal{H}$ is satisfiable. (Hintikka-Lemma)

In particular, $\Phi \subseteq \mathcal{H}$ is satisfiable. □

11.3.2 A Completeness Proof for Propositional Tableaux

With the model existence proof we have introduced in the last section, the completeness proof for first-order natural deduction is rather simple, we only have to check that Tableaux-consistency is an abstract consistency property.

We encapsulate all of the technical difficulties of the problem in a technical Lemma. From that, the completeness proof is just an application of the high-level theorems we have just proven.

P.2 P.3

Abstract Completeness for $\mathcal{T}_0$

▷ **Lemma 11.3.25** $\{\Phi \,|\, \Phi^{\mathsf{T}}$ has no closed Tableau$\}$ *is an abstract consistency class.*

▷ Proof: Let's call the set above ∇

P.1 We have to convince ourselves of the abstract consistency properties

P.1.1 ∇_c: $P, \neg P \in \Phi$ implies $P^{\mathsf{F}}, P^{\mathsf{T}} \in \Phi^{\mathsf{T}}$. □

P.1.2 $\nabla_\neg$: Let $\neg\neg\mathbf{A} \in \Phi$.

P.1.2.1 For the proof of the contrapositive we assume that $\Phi * \mathbf{A}$ has a closed tableau $\mathcal{T}$ and show that already Φ has one:

P.1.2.2 applying $\mathcal{T}_0\neg$ twice allows to extend any tableau with $\neg\neg\mathbf{B}^\alpha$ by $\mathbf{B}^\alpha$.

P.1.2.3 any path in $\mathcal{T}$ that is closed with $\neg\neg\mathbf{A}^\alpha$, can be closed by $\mathbf{A}^\alpha$. □

P.1.3 $\nabla_\vee$: Suppose $(\mathbf{A} \vee \mathbf{B}) \in \Phi$ and both $\Phi * \mathbf{A}$ and $\Phi * \mathbf{B}$ have closed tableaux

P.1.3.1 consider the tableaux:

$$\begin{array}{c}\Phi^{\mathsf{T}}\\ \mathbf{A}^{\mathsf{T}}\\ Rest^1\end{array} \qquad \begin{array}{c}\Phi^{\mathsf{T}}\\ \mathbf{B}^{\mathsf{T}}\\ Rest^2\end{array} \qquad \begin{array}{c}\Psi^{\mathsf{T}}\\ \mathbf{A} \vee \mathbf{B}^{\mathsf{T}}\\ \begin{array}{c|c}\mathbf{A}^{\mathsf{T}} & \mathbf{B}^{\mathsf{T}}\\ Rest^1 & Rest^2\end{array}\end{array}$$

□

P.1.4 $\nabla_\wedge$: suppose, $\neg(\mathbf{A}\vee\mathbf{B})\in\Phi$ and $\Phi\{\neg\mathbf{A},\neg\mathbf{B}\}$ have closed tableau $\mathcal{T}$.

P.1.4.1 We consider

$$\begin{array}{c} \Phi^{\mathsf{T}} \\ \mathbf{A}^{\mathsf{F}} \\ \mathbf{B}^{\mathsf{F}} \\ Rest \end{array} \qquad \begin{array}{c} \Psi^{\mathsf{T}} \\ \mathbf{A}\vee\mathbf{B}^{\mathsf{F}} \\ \mathbf{A}^{\mathsf{F}} \\ \mathbf{B}^{\mathsf{F}} \\ Rest \end{array}$$

where $\Phi=\Psi*\neg(\mathbf{A}\vee\mathbf{B})$. □

□

Observation: If we look at the completeness proof below, we see that the Lemma above is the only place where we had to deal with specific properties of the tableau calculus.

So if we want to prove completeness of any other calculus with respect to propositional logic, then we only need to prove an analogon to this lemma and can use the rest of the machinery we have already established "off the shelf".

This is one great advantage of the "abstract consistency method"; the other is that the method can be extended transparently to other logics.

Completeness of $\mathcal{T}_0$

▷ **Corollary 11.3.26** *$\mathcal{T}_0$ is complete.*

▷ Proof: by contradiction

P.1 We assume that $\mathbf{A}\in wff_o(\mathcal{V}_o)$ is valid, but there is no closed tableau for $\mathbf{A}^{\mathsf{F}}$.

P.2 We have $\{\neg\mathbf{A}\}\in\nabla$ as $\neg\mathbf{A}^{\mathsf{T}}=\mathbf{A}^{\mathsf{F}}$.

P.3 so $\neg\mathbf{A}$ is satisfiable by the model existence theorem (which is applicable as ∇ is an abstract consistency class by our Lemma above)

P.4 this contradicts our assumption that $\mathbf{A}$ is valid. □

Part III

Legal Foundations of Information Technology

In this Part, we cover a topic that is a very important secondary aspect of our work as Computer Scientists: the legal foundations that regulate how the fruits of our labor are appreciated (and recompensated), and what we have to do to respect people's personal data.

Chapter 12

Intellectual Property, Copyright, and Licensing

The first complex of questions centers around the assessment of the products of work of knowledge/information workers, which are largely intangible, and about questions of recompensation for such work.

Intellectual Property: Concept

- ▷ Question: Intellectual labour creates (intangible) objects, can they be owned?
- ▷ Answer: Yes: in certain circumstances they are property like tangible objects.
- ▷ **Definition 12.0.1** The concept of intellectual property motivates a set of laws that regulate property rights on intangible objects, in particular
  - ▷ Patents grant exploitation rights on original ideas.
  - ▷ Copyrights grant personal and exploitation rights on expressions of ideas.
  - ▷ Industrial Design Rights protect the visual design of objects beyond their function.
  - ▷ Trademarks protect the signs that identify a legal entity or its products to establish brand recognition.
- ▷ Intent: Property-like treatment of intangibles will foster innovation by giving individuals and organizations material incentives.

Naturally, many of the concepts are hotly debated. Especially due to the fact that intuitions and legal systems about property have evolved around the more tangible forms of properties that cannot be simply duplicated and indeed multiplied by copying them. In particular, other intangibles like physical laws or mathematical theorems cannot be property.

Intellectual Property: Problems

▷ Delineation Problems: How can we distinguish the product of human work, from "discoveries", of e.g. algorithms, facts, genome, algorithms. (not property)

▷ Philosophical Problems: The implied analogy with physical property (like land or an automobile) fails because physical property is generally rivalrous while intellectual works are non-rivalrous (the enjoyment of the copy does not prevent enjoyment of the original).

▷ Practical Problems: There is widespread criticism of the concept of intellectual property in general and the respective laws in particular.

  ▷ (software) patents are often used to stifle innovation in practice. (patent trolls)

  ▷ copyright is seen to help big corporations and to hurt the innovating individuals

We will not go into the philosophical debates around intellectual property here, but concentrate on the legal foundations that are in force now and regulate IP issues. We will see that groups holding alternative views of intellectual properties have learned to use current IP laws to their advantage and have built systems and even whole sections of the software economy on this basis.

Many of the concepts we will discuss here are regulated by laws, which are (ultimately) subject to national legislative and juridicative systems. Therefore, none of them can be discussed without an understanding of the different jurisdictions. Of course, we cannot go into particulars here, therefore we will make use of the classification of jurisdictions into two large legal traditions to get an overview. For any concrete decisions, the details of the particular jurisdiction have to be checked.

Legal Traditions

▷ The various legal systems of the world can be grouped into "traditions".

▷ **Definition 12.0.2** Legal systems in the common law tradition are usually based on case law, they are often derived from the British system.

▷ **Definition 12.0.3** Legal systems in the civil law tradition are usually based on explicitly codified laws (civil codes).

▷ As a rule of thumb all English-speaking countries have systems in the common law tradition, whereas the rest of the world follows a civil law tradition.

Another prerequisite for understanding intellectual property concepts is the historical development of the legal frameworks and the practice how intellectual property law is synchronized internationally.

Historic/International Aspects of Intellectual Property Law

▷ Early History: In late antiquity and the middle ages IP matters were regulated by royal privileges

- ▷ History of Patent Laws: First in Venice 1474, Statutes of Monopolies in England 1624, US/France 1790/1...
- ▷ History of Copyright Laws: Statue of Anne 1762, France: 1793, ...
- ▷ Problem: In an increasingly globalized world, national IP laws are not enough.
- ▷ **Definition 12.0.4** The Berne convention process is a series of international treaties that try to harmonize international IP laws. It started with the original Berne convention 1886 and went through revision in 1896, 1908, 1914, 1928, 1948, 1967, 1971, and 1979.
- ▷ The World Intellectual Property Organization Copyright Treaty was adopted in 1996 to address the issues raised by information technology and the Internet, which were not addressed by the Berne Convention.
- ▷ **Definition 12.0.5** The Anti-Counterfeiting Trade Agreement (ACTA) is a multinational treaty on international standards for intellectual property rights enforcement.
- ▷ With its focus on enforcement ACTA is seen my many to break fundamental human information rights, criminalize FLOSS

12.1 Copyright

In this Section, we go into more detail about a central concept of intellectual property law: copyright is the component most of IP law applicable to the individual computer scientist. Therefore a basic understanding should be part of any CS education. We start with a definition of what works can be copyrighted, and then progress to the rights this affords to the copyright holder.

Copyrightable Works

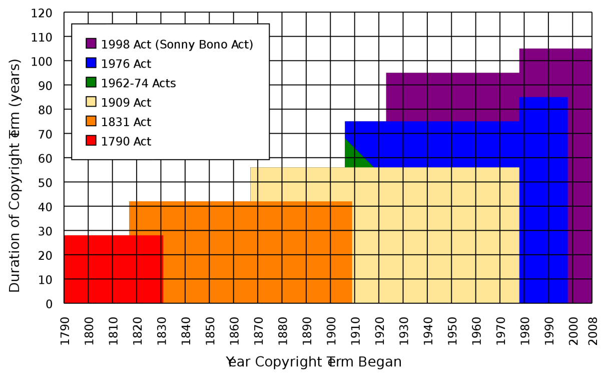
- ▷ **Definition 12.1.1** A copyrightable work is any artefact of human labor that fits into one of the following eight categories:
  - ▷ Literary works: Any work expressed in letters, numbers, or symbols, regardless of medium. (Computer source code is also considered to be a literary work.)
  - ▷ Musical works: Original musical compositions.
  - ▷ Sound recording s of musical works. (different licensing)
  - ▷ Dramatic works: literary works that direct a performance through written instructions.
  - ▷ Choreographic works must be âĂIJfixed,âĂİ either through notation or video recording.
  - ▷ Pictorial, Graphic and Sculptural works (PGS works): Any two-dimensional or three-dimensional art work
  - ▷ Audiovisual works: work that combines audio and visual components. (e.g. films, television programs)

  - ▷ Architectural works (copyright only extends to aesthetics)
- ▷ The categories are interpreted quite liberally (e.g. for computer code).
- ▷ There are various requirements to make a work copyrightable: it has to
  - ▷ exhibit a certain originality (Schöpfungshöhe)
  - ▷ require a certain amount of labor and diligence ("sweat of the brow" doctrine)

In short almost all products of intellectual work are copyrightable, but this does not mean copyright applies to all those works. Indeed there is a large body of works that are "out of copyright", and can be used by everyone. Indeed it is one of the intentions of intellectual property laws to increase the body of intellectual resources a society a draw upon to create wealth. Therefore copyright is limited by regulations that limit the duration of copyright and exempts some classes of works from copyright (e.g. because they have already been payed for by society).

Limitations of Copyrightabilitiy: The Public Domain

- ▷ **Definition 12.1.2** A work is said to be in the public domain, if no copyright applies, otherwise it is called copyrighted.
- ▷ **Example 12.1.3** Works made by US government employees (in their work time) are in the public domain directly (Rationale: taxpayer already payed for them)
- ▷ Copyright expires: usually 70 years after the death of the creator
- ▷ **Example 12.1.4 (US Copyright Terms)** Some people claim that US copyright terms are extended, whenever Disney's Mickey Mouse would become public domain.



Now that we have established, which works are copyrighted — i.e. to which works are intellectual property, let us see who owns them, and how that ownership is established.

Copyright Holder

▷ **Definition 12.1.5** The copyright holder is the legal entity that holds the copyright to a copyrighted work.

▷ By default, the original creator of a copyrightable work holds the copyright.

▷ In most jurisdictions, no registration or declaration is necessary (but copyright ownership may be difficult to prove)

▷ copyright is considered intellectual property, and can be transferred to others (e.g. sold to a publisher or bequeathed)

▷ **Definition 12.1.6 (Work for Hire)** A work made for hire is a work created by an employee as part of his or her job, or under the explicit guidance or under the terms of a contract.

▷ *In jurisdictions from the common law tradition, the copyright holder of a work for hires the employer, in jurisdictions from the civil law tradition, the author, unless the respective contract regulates it otherwise.*

We now turn to the rights owning a copyright entails for the copyright holder.

Rights under Copyright Law

▷ **Definition 12.1.7** The copyright is a collection of rights on a copyrighted work;

- ▷ personal rights: the copyright holder may
  - ▷ determine whether and how the work is published (right to publish)
  - ▷ determine whether and how her authorship is acknowledged. (right of attribution)
  - ▷ to object to any distortion, mutilation or other modification of the work, which would be prejudicial to his honor or reputation (droit de respect)
- ▷ exploitation rights: the owner of a copyright has the exclusive right to do, or authorize to do any of the following:
  - ▷ to reproduce the copyrighted work in copies (or phonorecords);
  - ▷ to prepare derivative works based upon the copyrighted work;
  - ▷ to distribute copies of the work to the public by sale, rental, lease, or lending;
  - ▷ to perform the copyrighted work publicly;
  - ▷ to display the copyrighted work publicly; and
  - ▷ to perform the copyrighted work publicly by means of a digital-audio transmission.

▷ **Definition 12.1.8** The use of a copyrighted material, by anyone other than the owner of the copyright, amounts to copyright infringement only when the use is such that it conflicts with any one or more of the exclusive rights conferred to the owner of the copyright.

Again, the rights of the copyright holder are mediated by usage rights of society; recall that intellectual property laws are originally designed to increase the intellectual resources available to society.

Limitations of Copyright (Citation/Fair Use)

- ▷ There are limitations to the exclusivity of rights of the copyright holder(some things cannot be forbidden)
- ▷ Citation Rights: Civil law jurisdictions allow citations of (extracts of) copyrighted works for scientific or artistic discussions. (note that the right of attribution still applies)
- ▷ In the civil law tradition, there are similar rights:
- ▷ **Definition 12.1.9 (Fair Use/Fair Dealing Doctrines)** Case law in common law jurisdictions has established a fair use doctrine, which allows e.g.
  - ▷ making safety copies of software and audiovisual data
  - ▷ lending of books in public libraries
  - ▷ citing for scientific and educational purposes
  - ▷ excerpts in search engine

  Fair use is established in court on a case-by-case taking into account the purpose (commercial/educational), the nature of the work the amount of the excerpt, the effect on the marketability of the work.

12.2 Licensing

Given that intellectual property law grants a set of exclusive rights to the owner, we will now look at ways and mechanisms how usage rights can be bestowed on others. This process is called licensing, and it has enormous effects on the way software is produced, marketed, and consumed. Again, we will focus on copyright issues and how innovative license agreements have created the open source movement and economy.

Licensing: the Transfer of Rights

- ▷ Remember: the copyright holder has exclusive rights to a copyrighted work.
- ▷ In particular: all others have only fair-use rights (but we can transfer rights)
- ▷ **Definition 12.2.1** A license is an authorization (by the licensor) to use the licensed material (by the licensee).
- ▷ Note: a license is a regular contract (about intellectual property) that is handled just like any other contract. (it can stipulate anything the licensor and licensees agree on) in particular a license may

- ▷ involve term, territory, or renewal provisions
- ▷ require paying a fee and/or proving a capability.
- ▷ require to keep the licensor informed on a type of activity, and to give them the opportunity to set conditions and limitations.

▷ Mass Licensing of Computer Software: Software vendors usually license software under extensive end-user license agreement (EULA) entered into upon the installation of that software on a computer. The license authorizes the user to install the software on a limited number of computers.

Copyright law was originally designed to give authors of literary works — e.g. novelists and playwrights — revenue streams and regulate how publishers and theatre companies can distribute and display them so that society can enjoy more of their work.

With the inclusion of software as "literary works" under copyright law the basic parameters of the system changed considerably:

- modern software development is much more a collaborative and diversified effort than literary writing,
- re-use of software components is a decisive factor in software,
- software can be distributed in compiled form to be executable which limits inspection and re-use, and
- distribution costs for digital media are negligible compared to printing.

As a consequence, much software development has been industrialized by large enterprises, who become copyrights the software was created as work for hire This has led to software quasi-monopolies, which are prone to stifling innovation and thus counteract the intentions of intellectual property laws.

The Free/Open Source Software movement attempts to use the intellectual property laws themselves to counteract their negative side effects on innovation and collaboration and the (perceived) freedom of the programmer.

Free/Open Source Licenses

▷ Recall: Software is treated as literary works wrt. copyright law.

▷ But: Software is different from literary works wrt. distribution channels (and that is what copyright law regulates)

▷ In particular: When literary works are distributed, you get all there is, software is usually distributed in binary format, you cannot understand/cite/modify/fix it.

▷ So: Compilation can be seen as a technical means to enforce copyright. (seen as an impediment to freedom of fair use)

▷ Recall: IP laws (in particular patent law) was introduced explicitly for two things

- ▷ incentivize innovation (by granting exclusive exploitation rights)
- ▷ spread innovation (by publishing ideas and processes)

Compilation breaks the second tenet (and may thus stifle innovation)

▷ Idea: We should create a public domain of source code

▷ **Definition 12.2.2** Free/Libre/Open-Source Software (FLOSS) is software that is and licensed via licenses that ensure that its source is available.

▷ Almost all of the Internet infrastructure is (now) FLOSS; so are the Linux and Android operating systems and applications like OpenOffice and The GIMP.

The relatively complex name Free/Libre/Open Source comes from the fact that the English[1] word “free” has two meanings: free as in “freedom” and free as in “free beer”. The initial name “free software” confused issues and thus led to problems in public perception of the movement. Indeed Richard Stallman’s initial motivation was to ensure the freedom of the programmer to create software, and only used cost-free software to expand the software public domain. To disambiguate some people started using the French “libre” which only had the “freedom” reading of “free”. The term “open source” was eventually adopted in 1998 to have a politically less loaded label.

The main tool in brining about a public domain of open-source software was the use of licenses that are cleverly crafted to guarantee usage rights to the public and inspire programmers to license their works as open-source systems. The most influential license here is the Gnu public license which we cover as a paradigmatic example.

GPL/Copyleft: Creating a FLOSS Public Domain?

▷ Problem: How do we get people to contribute source code to the FLOSS public domain?

▷ Idea: Use special licenses to:

- ▷ allow others to use/fix/modify our source code (derivative works)
- ▷ require them to release their modifications to the FLOSS public domain if they do.

▷ **Definition 12.2.3** A copyleft license is a license which requires that allows derivative works, but requires that they be licensed with the same license.

▷ **Definition 12.2.4** The General Public License (GPL) is a copyleft license for FLOSS software originally written by Richard Stallman in 1989. It requires that the source code of GPL-licensed software be made available.

▷ The GPL was the first copyleft license to see extensive use, and continues to dominate the licensing of FLOSS software.

▷ FLOSS based development can reduce development and testing costs (but community involvement must be managed)

▷ Various software companies have developed successful business models based on FLOSS licensing models. (e.g. Red Hat, Mozilla, IBM, ...)

Note: that the GPL does not make any restrictions on possible uses of the software. In particular, it does not restrict commercial use of the copyrighted software. Indeed it tries to allow commercial use

[1]the movement originated in the USA

without restricting the freedom of programmers. If the unencumbered distribution of source code makes some business models (which are considered as "extortion" by the open-source proponents) intractable, this needs to be compensated by new, innovative business models. Indeed, such business models have been developed, and have led to an "open-source economy" which now constitutes a non-trivial part of the software industry.

With the great success of open-source sofware, the central ideas have been adapted to other classes of copyrightable works; again to create and enlarge a public domain of resources that allow re-use, derived works, and distribution.

Open Content via Open Content Licenses

▷ Recall: FLOSS licenses have created a vibrant public domain for software.

▷ How about: other copyrightable works: music, video, literature, technical documents

▷ **Definition 12.2.5** The Creative Commons licenses are

- ▹ a common legal vocabulary for sharing content
- ▹ to create a kind of "public domain" using licensing
- ▹ presented in three layers (human/lawyer/machine)-readable

▷ Creative Commons license provisions (http://www.creativecommons.org)

- ▹ author retains copyright on each module/course
- ▹ author licenses material to the world with requirements

+/- attribuition	(must reference the author)
+/- commercial use	(can be restricted)
+/- derivative works	(can allow modification)
+/- share alike (copyleft)	(modifications must be donated back)

Chapter 13

Information Privacy

Information/Data Privacy

▷ **Definition 13.0.1** The principle of information privacy comprises the idea that humans have the right to control who can access their personal data when.

▷ Information privacy concerns exist wherever personally identifiable information is collected and stored – in digital form or otherwise. In particular in the following contexts

- ▷ Healthcare records
- ▷ Criminal justice investigations and proceedings
- ▷ Financial institutions and transactions
- ▷ Biological traits, such as ethnicity or genetic material
- ▷ Residence and geographic records

▷ Information privacy is becoming a growing concern with the advent of the Internet and search engines that make access to information easy and efficient.

▷ The "reasonable expectation of privacy" is regulated by special laws.

▷ These laws differ considerably by jurisdiction; Germany has particularly stringent regulations (and you are subject to these.)

Acquisition and storage of personal data is only legal for the purposes of the respective transaction, must be minimized, and distribution of personal data is generally forbidden with few exceptions. Users have to be informed about collection of personal data.

Organizational Measures or Information Privacy (under German Law)

▷ Physical Access Control: Unauthorized persons may not be granted physical access to data processing equipment that process personal data. ($\rightsquigarrow$ locks, access control systems)

▷ System Access Control: Unauthorized users may not use systems that process personal data ($\rightsquigarrow$ passwords, firewalls, ...)

▷ Information Access Control: Users may only access those data they are authorized to access. ($\rightsquigarrow$ access control lists, safe boxes for storage media, encryption)

▷ Data Transfer Control: Personal data may not be copied during transmission between systems ($\rightsquigarrow$ encryption)

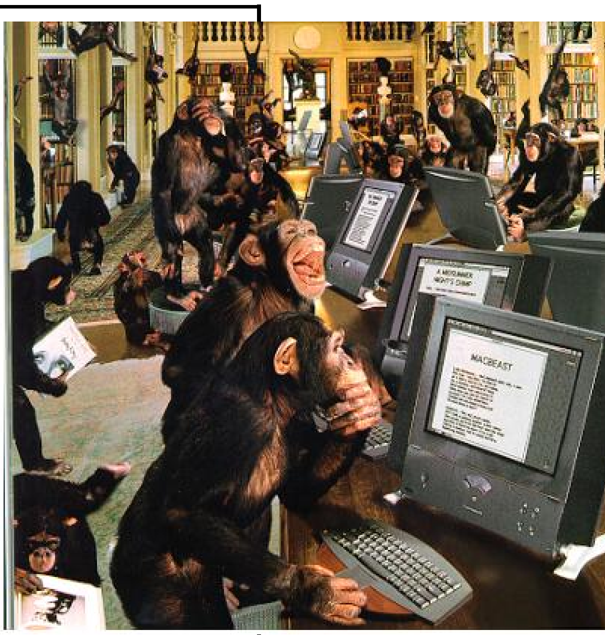
▷ Input Control: It must be possible to review retroactively who entered, changed, or deleted personal data. ($\rightsquigarrow$ authentification, journaling)

▷ Availability Control: Personal data have to be protected against loss and accidental destruction ($\rightsquigarrow$ physical/building safety, backups)

▷ Obligation of Separation: Personal data that was acquired for separate purposes has to be processed separately.

Part IV

A look back; What have we learned?



Let's hack! ⚠ ⇝ 2am in the CLAMV cluster

⚠ no, let's think ⚠

▷ GIGO: Garbage In, Garbage Out (– ca. 1967)

▷ Applets, Not Craplets^{tm} (– ca. 1997)

What have we thought about in this year?

We talked about various forms of

Machines Models
Algorithms, Languages and Programs
Information/Data/Representations

and their relation to each other

(and of course Math!)

Machine Models

▷ Abstract Interpreters (mind games)

▷ The SML interpreter/compiler (didn't we love recursive programming?)

Algorithms Languages and Programs

▷ Abstract data types (defining equations as recursive programs)

▷ standard ML (SML) (concrete ADTs with strong types, HO functions)

▷ elementary complexity analysis ($\mathcal{O}$ooooh, how fast this class grows)

Information, Data, and Representations

▷ term representations and substitutions (constants,variables, function application)

▷ Codes as transformations on formal languages. (programs as a special case)

▷ Boolean Expressions and Boolean functions (syntax and semantics)

▷ Hilbert, Resolution, Tableau, calculi (correct, complete)

The (Discrete) Math involved

▷ sets, relations, functions,

▷ natural numbers and proof by induction (such a lot of talk about something so simple)

▷ the axiomatic/deductive method in Math (play the math game by the rules)

▷ formal languages and codes (are just sets of strings and injective mappings)

▷ Boolean algebra, axioms, deduction, prime implicants

Bibliography

[Den00] Peter Denning. Computer science: The discipline. In A. Ralston and D. Hemmendinger, editors, *Encyclopedia of Computer Science*, pages 405–419. Nature Publishing Group, 2000.

[DPPDD09] A.K. Doxiadēs, C.H. Papadimitriou, A. Papadatos, and A. Di Donna. *Logicomix: An Epic Search for Truth.* Bloomsbury, 2009.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39:176–210, 1935.

[Hal74] Paul R. Halmos. *Naive Set Theory.* Springer Verlag, 1974.

[Hof79] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1979.

[Hut07] Graham Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[Koh08] Michael Kohlhase. Using LaTeX as a semantic markup format. *Mathematics in Computer Science*, 2(2):279–304, 2008.

[Koh11a] Michael Kohlhase. General Computer Science; Problems for 320101 GenCS I. Online practice problems at http://kwarc.info/teaching/GenCS1/problems.pdf, 2011.

[Koh11b] Michael Kohlhase. General Computer Science: Problems for 320201 GenCS II. Online practice problems at http://kwarc.info/teaching/GenCS2/problems.pdf, 2011.

[Koh16] Michael Kohlhase. sTeX: Semantic markup in TeX/LaTeX. Technical report, Comprehensive TeX Archive Network (CTAN), 2016.

[KP95] Paul Keller and Wolfgang Paul. *Hardware Design.* Teubner Leibzig, 1995.

[LP98] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1998.

[OSG08] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell.* O'Reilly, 2008.

[Pal] Neil/Fred's gigantic list of palindromes. web page at http://www.derf.net/palindromes/.

[Pau91] Lawrence C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence — A Modern Approach.* Prentice Hall, Upper Saddle River, NJ, 1995.

[Ros90] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill, 1990.

[SML10] The Standard ML basis library, 2010.

[Smo08] Gert Smolka. *Programmierung - eine Einführung in die Informatik mit Standard ML.* Oldenbourg, 2008.

[Smu63] Raymond M. Smullyan. A unifying principle for quantification theory. *Proc. Nat. Acad Sciences*, 49:828–832, 1963.

Index