Computability and Complexity

Lecture Notes

Herbert Jaeger, Jacobs University Bremen

Version history
Jan 30, 2018: created as copy of CC lecture notes of Spring 2017
Feb 16, 2018: cleaned up font conversion hickups up to Section 5
Feb 23, 2018: cleaned up font conversion hickups up to Section 6
Mar 15, 2018: cleaned up font conversion hickups up to Section 7
Apr 5, 2018: cleaned up font conversion hickups up to the end of these lecture notes

1 Introduction

1.1 Motivation

This lecture will introduce you to the theory of computation and the theory of computational complexity.

The theory of computation offers full answers to the questions,

- what problems can in principle be solved by computer programs?
- what functions can in principle be computed by computer programs?
- what formal languages can in principle be decided by computer programs?

Full answers to these questions have been found in the last 70 years or so, and we will learn about them. (And it turns out that these questions are all the same question). The theory of computation is well-established, transparent, and basically simple (you might disagree at first).

The theory of complexity offers many insights to questions like

- for a given problem / function / language that has to be solved / computed / decided by a computer program, how long does the fastest program actually run?
- how much memory space has to be used at least?
- can you speed up computations by using different computer architectures or different programming approaches?

The theory of complexity is historically younger than the theory of computation – the first surge of results came in the 60ties of last century. It grounds in the theory of computation. In comparison to the theory of computation, the theory of complexity is itself much more complex, is haunted by difficult unsolved fundamental questions, and is a highly active field of research.

Why should you want to know about either of the two? There are three sorts of reasons.

First, a **philosophical** reason. Being (or becoming) computer scientists, you should (want to) know what computation is. Just out of curiosity, just like a physicist should have sharp, rich and immediate intuitions about energy, mass or spacetime. We will use the largest part of today's lecture to outline the philosophical/mathematical history of how some of the most famous thinkers of mankind tried to come to terms with "computation".

Second, a **social** reason. If you are a computer science professional, you will talk to other computer science professionals over lunch or in cozy conference lounges. Your colleagues will have learnt all that is standardly learnt about computation and complexity. A lecture on computation and complexity is taught in every CS program on every habitable planet in the universe. Just in order to be accepted by your colleagues, you need to know how to keep a conversation running that starts like, "I have a problem that seems undecidable to me...", or "no chance to solve this task in a thousand years of runtime... ", or "smells like NP-complete, I don't envy you, ha-ha..." (very funny).

Third, some **practical** reasons:

1. The theory of computation has led to a mathematical formalism called *lambda calculus* that in turn has spawned a family of pragramming languages called *functional programming languages*. The best known among them are probably LISP, the workhorse language for many applications in artificial intelligence, and ML (you know it from your GenCS lecture).

2. In your future career as professional computer scientist you might be asked to write computer programs that solve tasks like the following:

   - Given an electronic ciruit, find a VLSI chip layout that minimizes connection wiring length.
   - Given a microprocessor program for an airplane autopilot, prove that the program will never stall (the certification-of-airworthiness authorities will want such a proof).
   - Given the daily assignment of customers to serve in different cities, schedule a fleet of delivery trucks such that effective mileage is minimized.
   - Given two DNA sequences, decide whether they share some subsequence, possibly with omissions and insertions within the shared subsequence.

   Three among these four problems are practically unsolvable, and one is (comparably) easy. Unfortunately, the autopilot certification task is not the simple one – and in fact, passenger airplanes have crashed because of onboard computer program failure. In this lecture you will

   - learn what “practically unsolvable” means (roughly, it means NP-complete),
   - get a hands-on feeling for detecting a practically unsolvable problem when you see it (by studying numerous examples),
   - learn about ways to attack "practically unsolvable" problems (by resorting to a slightly different but solvable problem; by finding approximate solutions; or by finding the correct solution with high probability).

   In sum, a training in computational complexity will prepare you for the diagnosis of, and the coping with, computationally complex tasks.

3. The theory of complexity (together with number theory) is at the core of modern cryptography techniques – and encrypting data is essential for safe communication protocols. In this sense, the future commercial impact of web-based technology hinges on (advanced) results from the theory of complexity.

1.2 Overview

This is the planned contents for this course. Parts marked by * are optional; we will enter them as time admits.

1. History of "computation"
2. Our workhorse: Turing machines
3. Computability and decidability, computing functions vs. deciding languages
4. Recursive functions
5. Lambda calculus
6. Functional programming
7. What "problems" are in the perspective of complexity theory
8. Complexity classes

9. NP-complete problems: your bread and butter as a computer scientist
10. *Logical characterization of complexity

In parts 1. through 6., I used many sources and recommend to rely mainly on these lecture notes. For parts 2. and 3., the textbook *Introduction to Automata Theory, Languages, and Computation* by Hopcroft, Motwani and Ullman will be helpful as an additional reference, but I will not follow it too closely. In 7. through 11., I will largely follow the book *Computational Complexity* by Christos H. Papadimitriou – a standard textbook. These lecture notes are intended to be a fully self-contained course companion, and the material covered herein is what you need to know for exams.

Part 1: Computation

2. A short history of "computation"

Philosophers and mathematicians struggled with the question what "computation" is for over two thousand years. You are in the happy historical condition that today this question is deemed understood and even answered. In the beginning, computation was not called computation. Instead, the question first appeared in the form of what is *logical reasoning*. Roughly speaking, logical reasoning is the art of arriving at a conclusion from some assumptions in a step-by-step fashion, executing justifiable rules of argumentation for every step of the argumentation chain. Only today we understand that this is largely the same thing as performing a computation – which also, on a digital von-Neumann computer such as your PC, proceeds by executing "correct" instructions in step-by-step fashion. The theory of computation departed from logics only quite recently, roughly since 130 years. In this little overview we will cover both the "ancient" and the more modern developments.

A handy overview[1] of this philosophical history of logic and computation can be found at the Website of theoretical computer scientist Grant Malcolm[2] – I have taken some inspirations for the following overview from it, as well as from the magnificent collection of mathematician's biographies[3] maintained by John J O'Connor[4] and Edmund F Robertson[5] at the University of St. Andrews, Scotland.

Aristotle[6] (384-322 B.C.) is, in many respects, the original inventor of the natural sciences. He emphasized rational reasoning over myth and mystics and painstakingly described and catalogued natural phenomena, especially animals and plants. He also investigated what correct argumentation (as opposed to mere rhetoric and "sophisms") is and found formal reasoning rules called syllogisms. The most famous syllogism is

1. All humans are mortal.
2. All Greeks are humans.
3. Therefore: All Greeks are mortal.

Syllogisms describe "logical" derivations of one conclusion (3.) from two premises (1., 2.). When the premises are true, the conclusion is true, too – because of the logical form of the argument itself. Aristotle presented and systematicised 19 syllogisms and claimed *"that every deductive argument can be expressed as a series of syllogistic inferences. That the argument is unconvincing masks the fact that simply by raising the problem, Aristotle earns the right to be considered not only the father of logic, but also the (grand)father of metalogic."* (Jonathan Lear, *Aristotle and Logical Theory*. Cambridge: Cambridge University Press, 1980.).

Euclid[7] (330? – 275? B.C.) invented what is today understood by mathematicians as mathematics. He used sequences of strict arguments to derive true (geometrical) theorems from axioms which were evidently true. You can still buy Euclid's book "The Elements" at Amazon, which is remarkable in many ways (consider, for instance, that the webserver

[1] http://www.csc.liv.ac.uk/~grant/Teaching/COMP317/logic.html
[2] http://www.csc.liv.ac.uk/~grant/index.html
[3] http://www-history.mcs.st-and.ac.uk/
[4] http://www-history.mcs.st-andrews.ac.uk/%7Ejohn
[5] http://www-history.mcs.st-andrews.ac.uk/%7Eedmund
[6] http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Aristotle.html
[7] http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Euclid.html

software of Amazon runs by the same principles as the arguments in Euclid's book – not from axioms to theorems but from your typed-in commands to the billing message).

Middle ages in Europe: academic thinking – both subjects and methods – was rigidly ruled by the ideology of scholasticism[8], very much a re-brew of Aristotelian philosophy. Logic itself was not further developed, but applied to develop intricate conceptual systems to explain God, the universe, and everything – in conformity with the dogmas of the Christian Church. The formal rigour of syllogisms crept into, and stifled, academic life itself: *"In the scholastic period, undergraduates at Oxford University could be fined for arguments that contravened the valid syllogisms.*" (from Grant Malcolm's page). Suggested reading: Umberto Eco, The Name of the Rose[9].

Little is known about the life of Al-Khwarizmi[10] (~780 – ~850), Arabian mathematician who wrote application-oriented treatises on the calculation of roots of quadratic equations. One manuscript introduces the Hindu decimal system of the natural numbers into the Arabian (and from there, later, Christian / Western) culture. The original text is lost but a Latin translation, *Algoritmi de numero Indorum* ("*Al-Khwarizmi on the Hindu Art of Reckoning"*) gave rise to the word algorithm deriving from the author's name in the title.

Gottfried Leibniz[11] (1646-1716), philosopher, broadly innovative mathematician (co-invented calculus with Newton, invented binary numbers, much modern mathematical notation originates from him), large-scale academic politician, literary author, inventor of mining technology, correspondent to 600 scholars throughout Europe – *"as capable of thinking for several days sitting in the same chair as of travelling the roads of Europe summer and winter … an indefatigable worker, a patriot and cosmopolitan, a great scientist, and one of the most powerful spirits of Western civilisation.*" (from the mathematician's biographies collection). This most powerful spirit had a dream: not having to do all that thinking anymore all by himself, but implement a universal reasoning mechanism in a (mechanical) machine which would capture and embody all truth and valid reasoning. He called this the *characteristica universalis*:

"All our reasoning is nothing but the joining and substituting of characters, whether these characters be words or symbols or pictures… all inquiries that depend on reasoning would be performed by the transposition of characters and by a kind of calculus.... And if someone would doubt my results, I should say to him: `let us calculate, Sir,' and thus by taking to pen and ink, we should soon settle the question." (from Grant Malcolm's webpage).

About the task to mechanically deduce all truths from first principles, Leibniz wrote:
"I believe that a number of chosen men can complete the task within five years; within two years they will exhibit the common doctrines of life, that is, metaphysics and morals, in an irrefutable calculus." (from Grant Malcolm's webpage).

[8] http://en.wikipedia.org/wiki/Scholasticism
[9] http://en.wikipedia.org/wiki/The_Name_of_the_Rose
[10] http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Al-Khwarizmi.html
[11] http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Leibniz.html

George Boole[12] (1815 – 1864), mathematician, found that elementary logical operations "and" (in mathematical notation: $\wedge$), "or" ($\vee$), "not" ($\neg$) have similar operational structure as has arithmetics modolo 2 or set-theoretic manipulations with the empty and the all set. For instance, "a $\wedge$ b" (where a and b are propositions that can be true or false) behaves in the same was as "$X \cap Y$" (where X and Y are either the empty set or an "all" set), and both behave in the same way as the multiplication "$x * y$" modolo 2 (where $x, y \in \{0,1\}$). These insights are published in a slim book with a bold title, *An investigation into the Laws of Thought*. *Boolean logic* or *Boolean algebra* is nowadays implemented in terms of "transistor on / off" in logical gate arrays which form the basis of digital computer chips. Boolean algebra establishes a first rigorous connection between reasoning and calculation. Boole also found out that two of the 19 classical Aristotelian syllogisms were not valid.

David Hilbert[13] (1862 – 1943). Possibly the most renown mathematician of his time, with contributions to (and connections between) many fields in mathematics, Hilbert at the turn of the century was bold enough to declare in a famous speech *The open problems of Mathemtics* what would be the most important questions for Mathematics in the upcoming (20th) century. At that time, the foundations of formal logic (Gotthold Frege[14] 1845 – 1925, Bertrand Russell[15] 1872 – 1970, Alfred N. Whitehead[16] 1861 – 1947) and set theory (Georg Cantor[17], 1845 – 1918) were being developed, and there was hope to reduce *all* mathematics to a few first principles, and to prove mathematically that these foundations would be coherent, i.e. never lead to contradictions. Hilbert was convinced that this could be achieved. A central aspect of this program was to prove that the theory of arithmetics (as defined by the axioms usually named after Guiseppe Peano[18] (1858 – 1932)) was free of internal contradictions. Proving this meta-theorem was indeed one (the third) of Hilbert's famous Open Problems, called by him *"the compatibility of the arithmetic axioms"*. On his tombstone, Hilbert had to be written 6 intellectually aggressive words, *We must know, we shall know (Wir müssen wissen, wir werden wissen)*.

Ludwig Wittgenstein[19] (1889 – 1951) was a philosopher/logician who led an intensely unhappy, isolated life as a renown but introverted genius. His fame rests on a small, highly compressed, enigmatic logical-philosophical investigation into the nature of nature and logic, the *Tractatus Logico Philosophicus* (1922). This book of crystalline mysteriousness contains sentences like *"The riddle does not exist. If a question can be put at all, then it can also be answered." "Everything that can be said, can be said clearly."*, and, most famous citation of all, *"Whereof one cannot speak, thereof one must be silent."* The book starts (and continues) like a hymn on reasoning (my translation from the German original; there are more scholarly correct translations available):

1 *The world is everything which is the case.*
1.1 *The world is the totality of facts, not of things.*
1.2 *The world is determined by all facts and by the fact that those are all facts.*
...
3. *The logical image of facts are thoughts.*

[12] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Boole.html
[13] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Hilbert.html
[14] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Frege.html
[15] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Russell.html
[16] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Whitehead.html
[17] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Cantor.html
[18] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Peano.html
[19] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Wittgenstein.html

3.01 The totality of true thoughts is an image of the world.
3.02 The thought contains the possibility of the circumstance it thinks. What is thinkable is possible.
3.03 We cannot think anything unlogical, because then we would have to think unlogically.
...
3.032 Representing something "contradictory to logic" by language is just as impossible as rendering by its coordinates a figure which contradicts the laws of geometry.

It is difficult to understand the Tractatus, but struggling with it makes your mind think more beautifully. My interpretation is that it claims that the physical world follows the same structural rules as does language, and that the truth of sentences derives from a structural agreement between the symbolic structure of the sentence with the circumstance (Sachverhalt) described. Wittgenstein and his Tractatus lead from mathematical logic into the philosophy of science and language and has been enormously influential for our current "positivistic[20]" understanding of scientific methods and insight.

Kurt Gödel[21] (1906 – 1978) in 1931 published an article *On Formally Undecidable Propositions* [22](*Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*). This paper was a mind-boggling flash of lightning which shattered the hopes that ran from Aristotle through Leibniz to Hilbert. Loosely speaking, the Gödel incompleteness theorems state that no mathematical theory (which includes at least basic arithmetics) can prove all theorems that are true in this theory. Or, stated more generously and somewhat mysteriously, you cannot catch all mathematical truths from within mathematics. Interestingly, *this* can be proved from within mathematics. The basic idea of the proof runs as follows:

1. Assume you have an algorithm *A* that gives true answers to *all* questions that have true answers.
2. Write down the statement "*A* will never say that this sentence is true". Call this statement *G* (honoring Gödel).
3. Ask *A* whether *G* is true.
4. What should *A* report? If *A* says, *G* is true, then *G* is false. So *A* can't say *G* is true, because *A* always answers correctly. So *A* won't say *G* is true. So *G* is true!
5. We have found a true statement, *G*, which cannot be asked from *A*. Thus, our initial assumption that we have an algorithm that gives true answers to all questions is false.

The Gödel incompleteness theorems can be seen from many angles[23] and always retain a certain dizzifying quality. For instance, the second Gödel incompleteness theorem implies that we cannot disprove 0 = 1 (and that this is basically the same why Microsoft Word or airplane autopilots cannot be certified never to crash). The incompleteness theorem has been much abused in philosophical and popular discussions, as in statements like "you can never completely know yourself".

Alan Turing[24] (1912 – 1954), code-breaking English war hero, marathon athlete, and cruelly harassed homosexual, was a machine-oriented mathematician if there ever was one. He reduced syllogisms and every other computation to a little read/write device that wandered up

[20] http://en.wikipedia.org/wiki/Logical_positivism
[21] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Godel.html
[22] http://www.research.ibm.com/people/h/hirzel/papers/canon00-goedel.pdf
[23] http://www.miskatonic.org/godel.html
[24] http://www.turing.org.uk/turing/index.html

and down a long tape with zeros and ones on it. This was the Turing Machine (TM for short), which we will get to know extremely well in this lecture. In 1950 he published Computing Machinery and Intelligence[25], a paper where he claims that machines can, in principle, think like humans. As a criterium to decide whether a machine can really think he proposed an *imitation game* (later known as Turing Test): if an external (human) judge cannot decide whether one partner in an overheard conversation is a computer or a human, and it is indeed a computer, then this computer cannot be denied the quality of human thinking. Turing addresses the obvious critique implied by the Gödel theorems by putting the notion of "human intelligence" into a human perspective: true human intelligence can come up with errors and only draws from finite resources. Therefore, Gödel's (and actually Turing's own) mathematical results on incompleteness and undecidability do not apply, because they refer to a 100%-errorless notion of truth derived by idealized mechanisms with infinite resources. Note that this is a fundamental turn in perspective: whereas the old, pre-Gödel dream went something like this:

- True is what can be derived by correct reasoning
- Humans (and machines) can do correct reasoning
- Therefore, humans (and machines) can discover all truths

the new, post-Turing dream goes like this:

- Human intelligence is produced by a finite mechanism
- Computers are finite mechanisms that can simulate other finite mechanisms
- Therefore, computers can *perfectly simulate* (i.e., *be*) intelligence

There is a direct road from Turing's theses to the claims of Artificial Intelligence, which have been criticised for over-reaching. But Turing's original claims are, in fact, quite moderate in that they gracefully accept the Gödel blow and ascribe the same kind of modest, finite, error-prone intelligence to both humans and machines.

With this I will end my historical reflections. What has come after Turing is the theory of complexity. It is not yet history and we will devote much of this course to it, starting where history ends: with the Turing machine. I hope that you have a better feeling where you and I stand in the big sway of history: namely, on the shoulders of giant eagles, with little sparrow's wings to try some fluttering of our own.

[25] A. M. Turing (1950) Computing Machinery and Intelligence. Mind 59 (236): 433-460. http://www.wps.com/projects/Computing-machinery.html

3 Turing Machines and Computability

3.1 How to work with Turing machines

By and large, our Turing machine (TM) notation follows the Papadimitriou book.

A TM is a deterministic automaton model like deterministic finite automata (DFA) or pushdown automata (PDA). In the lecture "Advanced CS 1" we used automata for describing languages: with every DFA or PDA we associated a formal language, the set of all words accepted by the automaton. Turing machines can be used to define languages, too. But this is not the only, and indeed not the most common, purpose of TMs. Here is a list of what TMs are used for:

1. **Computing numerical functions.** A numerical function is any function $f: \mathbb{N}^k \to \mathbb{N}$ (where $k \geq 0$), not necessarily defined everywhere. Examples range from simple addition to functions as strange as "f_G is the function that returns 1 on every input if the Goldbach conjecture holds, else it returns 0 on every input". [The Goldbach conjecture claims that every even number is the sum of two primes; it has resisted all efforts to be proven so far]. If you could compute $f_G(1)$, you'd be instantaneously famous. TMs can compute some numerical functions, and cannot compute some others. We'll see that *every* function that can be computed *at all* can also be computed by a TM.
2. **Solving problems.** A "problem", in the language of theoretical computer science, is a family of similar questions that can be rigorously formalized. A standard textbook example is the problem of REACHABILITY (we will use all-uppercase words to denote problems in this lecture): given a graph with a particular starting node *A* and a particular goal node *B*, the question is whether there is a path in the graph from *A* to *B*. This is not a single concrete question but a whole family of questions, because for every graph and choice of start and goal node you get another concrete question. Each such concrete question is an *instance* of the problem. TMs are used to solve problems as follows. First, code a problem instance to make it TM-readable – that is, convert it to a symbol string. Then input this string to the TM, and start it. Then wait... if after some time the TM stops and writes "yes" on its output tape, the answer to the problem instance is "yes". If the TM stops and writes "no", the answer is "no". If the TM does not stop at all, then you have chosen an inappropriate TM, or this problem is undecidable, that is, no TM exists that can give correct yes/no answers on all problem instances. Notice that in problem solving, you re-use the same TM for all instances of the problem.
3. **Defining languages by enumeration.** Just like DFAs and PDAs, TMs come equipped with an accepting state and can be used to define a language by acceptance: a word *w* belongs to the language of the TM if and only if the TM ends up in the accepting state on input *w*. For reasons that will later become clear this method of defining a language by a TM leads to the class of *recursively enumerable* languages. It turns out that this is the same class of languages that we obtain through type-0 grammars (aka unrestricted grammars).
4. **Deciding languages.** Another way of defining a language by a TM is to require that the TM is equipped with two special states, an accepting state and a rejecting state. Then we say that such a TM decides a language, if on every word input *w* it either ends in the accepting state or in the rejecting state. The word *w* belongs to the language of such a TM iff the TM ends in the accepting state. This method leads to the class of *recursive* (or *decidable*) languages, which is a proper subclass of the recursively enumerable languages.

Computing totally defined functions, solving problems, and deciding languages will turn out to be essentially equivalent tasks. If you know how to cope with either of these three, you automatically know how to solve the remaining types of tasks. We will soon learn much more about this. Here is how these three types of tasks can be transformed into each other:

- **Language decision problems as numerical function computation:** Given: a language decision task, that is, a language L and the task to decide whether a given word w is in L or not. Transform this into an equivalent numerical function computation by (i) coding words w by integers through a *codefunction*, (ii) defining f to be f (*codefunction* (w)) = 1 iff w is in L, else 0. Then if you can compute the function f with a TM, you have also solved the word membership decision problem.
- **Numerical function computation as problem solving:** Given: a numerical function f that you want to compute. Transform this task into a problem solving task by (i) coding queries "$f(n) = m$?" into input strings that can be read by a TM, (ii) use this TM to give answers to the series of queries "$f(n) = m$?" for m = 0, 1, 2, ... , waiting until for some m the answer is "yes". Then you have computed the value of $f(n)$.
- **Problem solving as language decision:** Given: a set of problem instances q in TM-readable format. Transform this into a language decision problem by (i) interpreting the questions q as words (inputs to TMs are in fact words over some alphabet), (ii) solving the problem by deciding the language L = {q | answer to q is "yes"}.

In a similar way, the tasks of defining languages by recursive enumeration, of computing partially defined functions, and of solving problems that do not always have yes/no answers, are essentially identical.

Theoretical computer scientists freely jump from one variant of these tasks to another, taking for granted that a problem first stated in one variant can be "coded into", "transformed into", "reduced to", "represented in" the other variants. The theory of computability is mostly expressed in terms of function computation and language decision. The reason is that, historically, the theory was started in the 1930'ies by mathematicians with an interest in functions, and that later it became part of theoretical CS with its preference for formal languages – which allows to see TMs in the same light as DFAs and PDAs. The reason why the theory of complexity instead is usually spelled out for problems is practical: it's ultimately mostly problems what computer programs are designed for.

3.2 Definition of TMs

A DFA is just a deterministic, finite state transition mechanism with no further "memory" except the current state. A PDA is a DFA equipped with an additional memory of a simple stack organiziation. TMs come next (and last) in this sequence: TMs are DFAs equipped with a more powerful additional memory, namely, a re-writable tape. We saw in the ACS 1 lecture that many limitations of PDAs arise from the fact that the automaton can only "see" the topmost memory element; in order to see what is below, it has to erase the topmost element. By contrast, a TM may inspect any part of its memory without erasing anything else.

A basic TM consists of three components:

1. a right-infinite tape which serves as input medium, output medium and memory (the TM can read/write symbols from/to it),
2. a read/write head (or "cursor") that can move over this tape and read/write symbols, and
3. a control unit that is actually a DFA, controlling the actions of the read-write head.

Graphically, a TM looks like this:

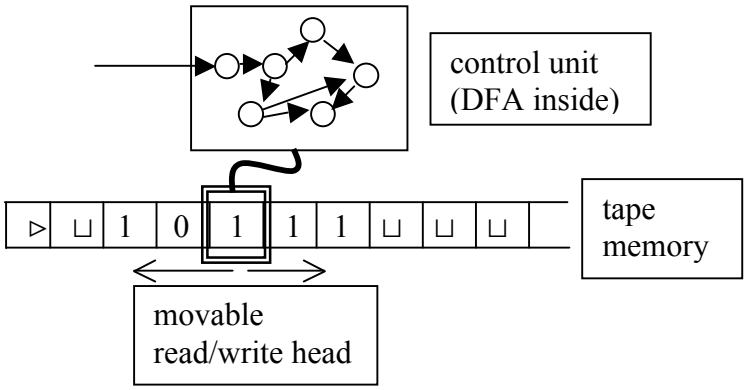


Figure 3.1 Schema of a TM

In the beginning of a TM run, input may be written on the tape as a finite word. Then the TM works by repeating a simple operating cycle. One operating cycle runs as follows:

1. read the tape symbol under the r/w head
2. use this as input to the control DFA
3. DFA transits to next state according to symbol input
4. as a side effect of this transition,
    a. the symbol is overwritten
    b. the r/w head may move one tape cell to the right or left or just remain on its current position
5. stop if the control DFA enters a special state (there are several types of stopping states, *accepting* or *rejecting* or just *halting* states)

After stopping, what is left on the tape may be used as the TM's output (not all variants of operating TMs lead to output).

Before we proceed to the formal definition, a historical note. Turing arrived at the notion of TMs by reasoning from intuitive first principles concerning the nature of (machine-executable) operations on symbols, for instance:

"... The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B *to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused...."*
(From the original paper where Turing introduces his machine model, "On computable numbers, with an application to the Entscheidungsproblem[26]" (1936-37))

[26] http://www.abelard.org/turpap2/tp2-ie.asp

Now we give the formal definition. There are many (equivalent) ways of how to define TMs. We use the definition from the Papadimitriou book.

Definition 3.1: A Turing machine is a structure $M = (K, \Sigma, \delta, s)$, where K is a finite set of *states*, $s \in K$ is the *initial state*, the *alphabet* Σ is a set of (tape) *symbols*, and where K and Σ are disjoint. We assume that Σ always contains the special symbols $\sqcup$ and $\triangleright$, the *blank* and the *first* symbol. Finally, δ is a *transition function*, where

$$\delta: K \times \Sigma \rightarrow (K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}.$$

We assume that h (the *halting state*), "yes" (the *accepting state*), "no" (the *rejecting state*), and the *cursor directions* $\leftarrow$ for "left", $\rightarrow$ for "right" and $-$ for "stay", are extra symbols not in $K \cup \Sigma$.

This might seem a bit awkward, but if you consider (as we shall see) that this definition gives you "total computing power", it's really very compact.

Because the special symbols $\sqcup$ and $\triangleright$ are always assumed to be in Σ, we sometimes will not explicitly mention them for simplicity and, for instance, say that a TM has alphabet $\{1, 0\}$ although technically speaking, the alphabet is $\{1, 0, \sqcup, \triangleright\}$.

The function δ is the hardwired operating rule of the machine. It specifies, for each combination of current state $q \in K$ and current symbol $\sigma \in \Sigma$, a triple $\delta(q, \sigma) = (p, \rho, D)$, where p is the next state, ρ is the symbol to be overwritten on σ, and $D \in \{\leftarrow, \rightarrow, -\}$ is the direction in which the cursor will move. Special rules apply to our special symbols. For $\triangleright$ we require that upon reading $\triangleright$, the same symbol $\triangleright$ is re-printed and the head moves right. In this way, $\triangleright$ works as a "left end mark" which is never erased. The symbol $\sqcup$ marks an empty tape cell.

For initialization we agree that the initial state is s. The tape is initialized to $\triangleright$, followed on the right by a finite (possibly empty) word $x \in (\Sigma - \{\sqcup, \triangleright\})^*$, followed by an infinite sequence of $\sqcup$ symbols marking an empty tape. We say that x is the *input.* The cursor is pointing to the first symbol, always a $\triangleright$. This ends our special agreements.

Although our agreements ensure that the cursor never trips over the initial $\triangleright$ to the left, it may wander arbitrarily far to the right.

This definition of a TM has some convenience features that are not strictly required from a theoretical perspective. One could omit the left-end delimiter symbol $\triangleright$ and use a left-right-infinite tape; one could omit the "yes" and "no" states and replace them by a terminal subroutine where these results would be explicitly printed on the tape just before halting. Also the stand-still cursor motion "–" could be replaced by a subroutine of two moves $\rightarrow \leftarrow$.

In order to illustrate the intuitions behind our definition, we consider an example.

Example 3.1 (from the Papdimitriou book)**:** The following table specifies a 4-state TM SHIFTRIGHT with alphabet {0,1}, which shifts the input one place to the right. To the right of the table you see a run of this TM. Underscores indicate current cursor position.

$p \in K$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s	0	$(s,0, \rightarrow)$
s	1	$(s,1, \rightarrow)$
s	⊔	$(q, ⊔, \leftarrow)$
s	▷	$(s, ▷, \rightarrow)$
q	0	$(q_0, ⊔, \rightarrow)$
q	1	$(q_1, ⊔, \rightarrow)$
q	⊔	$(q, ⊔, -)$
q	▷	$(h, ▷, \rightarrow)$
q_0	0	$(s,0, \leftarrow)$
q_0	1	$(s,0, \leftarrow)$
q_0	⊔	$(s,0, \leftarrow)$
q_0	▷	$(h, ▷, \rightarrow)$
q_1	0	$(s,1, \leftarrow)$
q_1	1	$(s,1, \leftarrow)$
q_1	⊔	$(s,1, \leftarrow)$
q_1	▷	$(h, ▷, \rightarrow)$

0.	s	$\underline{▷}010$
1.	s	$▷\underline{0}10$
2.	s	$▷0\underline{1}0$
3.	s	$▷01\underline{0}$
4.	s	$▷010\underline{⊔}$
5.	q	$▷01\underline{0}⊔$
6.	q_0	$▷01⊔\underline{⊔}$
7.	s	$▷01\underline{⊔}0$
8.	q	$▷0\underline{1}⊔0$
9.	q_1	$▷0⊔\underline{⊔}0$
10.	s	$▷0\underline{⊔}10$
11.	q	$▷\underline{0}⊔10$
12.	q_0	$▷⊔\underline{⊔}10$
13.	s	$▷\underline{⊔}010$
14.	q	$\underline{▷}⊔010$
15.	h	$▷\underline{⊔}010$

Note that we provided a completely specified lookup table for δ here. However, some rules are redundant because they will never be executed (e.g., rule $\delta(q, ⊔) = (q, ⊔, -)$). We shall often leave out rules that we know to be redundant (but note that we will not be generally able to decide for an arbitrary rule whether it is redundant! – because that would solve the halting problem which we will soon get to know).

Like with the definition of DFAs and PDAs (in the lecture ACS 1), the raw definition 3.1 of TMs does not tell us how to use them – the previous example reflected merely our intuitions. To make these precise in a very similar way as with DFAs and PDAs, we introduce "snapshot" descriptions of TM runs (*configurations*), and then specify how δ induces sequences of configurations.

Definition 3.2: A *configuration* of a TM M is a triple (q, w, u), where $q \in K$ is a state, and w and u are words from Σ^*.

The intended interpretation of a configuration is the following: q is the current state that the TM is in at the beginning of a working cycle, w is the tape symbol string to the left of the cursor, including the symbol currently scanned by the cursor, and u is the symbol string to the right of the cursor (the finite string without the infinitely many trailing ⊔'s); u may be the empty string ε. The symbol string to the right of the cursor contains all the symbols that have been printed by the cursor during the previous (and including the current) steps of the computation. Blank symbols ⊔ are included in a configuration only if they have been printed (that is, the default blank symbols that inhabit the tape at the start of the computation are not part of a configuration). For instance, the configuration $▷01\underline{0}⊔$ encountered at the fifth step in

Example 3.1 would be written as $(q, \triangleright 010, \sqcup)$. Using configurations, we shall describe the workings of a TM, as follows:

Definition 3.3:

(a) Let (q, w, u), (q', w', u') be two configurations of a TM M, where $w = a_1...a_n$, $u = b_1...b_m$, where only u is possibly empty. Then, if either of the three following cases holds:

1. $\delta(q, a_n) = (q', c, \leftarrow)$ and $w' = a_1...a_{n-1}$ and $u' = cb_1...b_m$
2. $\delta(q, a_n) = (q', c, \rightarrow)$ and $w' = a_1...a_{n-1}cb_1$ and $u' = b_2...b_m$
3. $\delta(q, a_n) = (q', c, -)$ and $w' = a_1... a_{n-1}c$ and $u' = b_1b_2...b_m$

we say that δ carries M from configuration (q, w, u) to configuration (q', w', u') in one step, and we write $(q, w, u) \xrightarrow{M} (q', w', u')$. (Notice that we should add a special case when the cursor moves right and u was empty – such trifles are commonly neglected in the TM literature, where a spirit of "details will sort themselves out anyway" rules.)

(b) If M transits from (q, w, u) to (q', w', u') in k steps, we write $(q, w, u) \xrightarrow{M^k} (q', w', u')$. If there exists a finite k such that $(q, w, u) \xrightarrow{M^k} (q', w', u')$, we write $(q, w, u) \xrightarrow{M^*} (q', w', u')$.

(c) If $(s, \triangleright, x) \xrightarrow{M^*} (q, \triangleright w, u)$, where $q \in \{h, \text{"yes"}, \text{"no"}\}$, we agree on the following terminology:

- If $q = h$, we say the TM has halted with output $y = wu$. We write $M(x) = y$.
- If $q =$ "yes", we say the machine accepts its input and write $M(x) =$ "yes".
- If $q =$ "no", we say the machine rejects its input and write $M(x) =$ "no".

(d) If for no k, $(s, \triangleright, x) \xrightarrow{M^k} (q, \triangleright w, u)$, where $q \in \{h, \text{"yes"}, \text{"no"}\}$, we say that the TM does not halt on input x, and write $M(x) = \nearrow$.

These different possibilities reflect the different uses one can make of a TM: the case $M(x) = y$ is used for computing functions and certain problems that have output beyond a simple yes/no answer, the cases $M(x) =$ "yes" and $M(x) =$ "no" are used for language decision and problem solving, and the case $M(x) =$ %is encountered when a function is only partially defined or a problem is undecidable for some inputs.

Example 3.2 (from the Papadimitriou book)**:** Consider the two-state TM PSEUDO-ADD1 with alphabet {0,1} specified by the following table.

$p \in K$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s	0	$(s,0, \rightarrow)$
s	1	$(s,1, \rightarrow)$
s	⊔	$(q, ⊔, \leftarrow)$
s	▷	$(s, ▷, \rightarrow)$
q	0	$(h,\ 1,\ -)$
q	1	$(q,\ 0,\ \leftarrow)$
q	▷	$(h, ▷, \rightarrow)$

0.	s	▷010
1.	s	▷0̲10
2.	s	▷01̲0
3.	s	▷010̲
4.	s	▷010⊔̲
5.	q	▷01̲0⊔
6.	h	▷011̲⊔
0.	s	▷̲011
1.	s	▷0̲11
2.	s	▷01̲1
3.	s	▷011̲
4.	s	▷011⊔̲
5.	q	▷01̲1⊔
6.	q	▷0̲10⊔
7.	q	▷0̲00⊔
8.	h	▷1̲00⊔

On the right side you see two runs of this TM. It implements the numerical "add 1" operation on its binary input, by first moving to the rightmost end of the input (in state s) and then working its way leftward again (in state q), flipping 1's to 0's. The first 0 it encounters is flipped to 1 and the machine halts. However, the machine has a bug: it will transform input 1^n to output 0^n. A possible remedy would be to combine PSEUDO-ADD1 with the TM SHIFTRIGHT from example 2.1. Roughly speaking, build a combined TM ADD1 that first executes SHIFTRIGHT. When SHIFTRIGHT halts, the cursor is one position right from the left end-marker on an empty cell symbol: ▷⊔̲. Modify SHIFTRIGHT such that in this situation it doesn't halt but prints a 0 and moves the cursor left, entering the starting state of PSEUDO-ADD1. Then, execute PSEUDO-ADD1. Because some of the states of PSEUDO-ADD1 and SHIFTRIGHT have the same names (s and q), this requires that the states of PSEUDO-ADD1 be renamed (say, to s' and q').

This is a painfully low-level way of programming; assembler programming is luxury compared with this. But you saw that it is not difficult to build more complex TMs from simpler ones in a modular way, exactly like we assembled complex DFAs from simple ones or complex computer programs from pre-programmed modules. The basic idea to dynamically connect "module" TMs is to redefine the halting rules of all non-terminal module TMs such that they switch not to a halting state but into the starting state of the module TM whose turn is next. We will not bother about details – working with TMs is the art of not bothering about computational detail, and after your previous experience with DFAs and PDAs you are supposed to be able to fill in such detail for yourself.

Many TM simulators (Java applets) can be found on the Internet, for instance at http://morphett.info/turing/turing.html . You will see there are some minor differences to our basic definition: rules are denoted in a different format; for instance **6,A 12,_,<** means "if symbol **A** is read in state **6**, then enter state **12** and print a blank and go left". Generally, TMs come in many variants; they are all bascially equivalent as we shall see.

Example 3.3 (from the Papdimitriou book)**:** SHIFTRIGHT is a TM that performs a simple memory re-allocation operation, ADD1 is a TM that computes a numerical function. We will now give an example of a 6-state TM PALINDROME that decides a language over $\Sigma = \{0,1\}$, namely, the language of palindromes $L = \{w \in \{0,1\}^* \mid w = w^R\}$.

$q \in K$	$\sigma \in \Sigma$	$\delta(q, \sigma)$
s	0	$(q_0, \triangleright, \rightarrow)$
s	1	$(q_1, \triangleright, \rightarrow)$
s	$\sqcup$	("yes", $\sqcup$, –)
s	$\triangleright$	$(s, \triangleright, \rightarrow)$
q_0	0	$(q_0, 0, \rightarrow)$
q_0	1	$(q_0, 1, \rightarrow)$
q_0	$\sqcup$	$(q'_0, \sqcup, \leftarrow)$
q_1	0	$(q_1, 0, \rightarrow)$
q_1	1	$(q_1, 1, \rightarrow)$
q_1	$\sqcup$	$(q'_1, \sqcup, \leftarrow)$

$q \in K$	$\sigma \in \Sigma$	$\delta(q, \sigma)$
q'_0	0	$(q, \sqcup, \leftarrow)$
q'_0	1	("no", 1, –)
q'_0	$\triangleright$	("yes", $\sqcup$, $\rightarrow$)
q'_1	0	("no", 1, –)
q'_1	1	$(q, \sqcup, \leftarrow)$
q'_1	$\triangleright$	("yes", $\sqcup$, $\rightarrow$)
q	0	$(q, 0, \leftarrow)$
q	1	$(q, 1, \leftarrow)$
q	$\triangleright$	$(s, \triangleright, \rightarrow)$

The machine PALINDROME works as follows. In state s, it searches the tape for the first input symbol. When it finds it, it turns it into a $\triangleright$ and remembers it in its state (entering q_0 if symbol is 0 and entering q_1 if it is 1). Remembering symbols through states is a common trick when programming TMs. PALINDROME then moves to the right until the first $\sqcup$ is met, upon which it switches to q'_0 or q'_1 (still remembering the first symbol). Then it goes one to the left and checks whether the symbol seen is the one remembered in q'_0 / q'_1. If not, stop in the rejecting state. If yes, overwrite the rightmost symbol with $\sqcup$, and enter state q. In state q, the TM just moves leftwards until it sees a $\triangleright$. Then the entire routine is started all over again. Note that one cycle of this routine replaces the leftmost symbol of the (remaining) input word with $\triangleright$ and the rightmost with a blank symbol. When the entire input has been used up in this way without the TM entering the rejecting state, it reaches the accepting state. At http://morphett.info/turing/turing.html you can run a TM on the PALINDROME problem to experience how it works.

For example, on input 0010 (not a palindrome), PALINDROME will pass through the following configurations:

$$(s, \triangleright, 0010) \overset{M^5}{\rightarrow} (q_0, \triangleright\triangleright 010\sqcup, \varepsilon) \overset{M}{\rightarrow} (q'_0, \triangleright\triangleright 010, \sqcup) \overset{M}{\rightarrow} (q, \triangleright\triangleright 01, \sqcup\sqcup) \overset{M^2}{\rightarrow} (q, \triangleright\triangleright, 01\sqcup\sqcup)$$
$$\overset{M}{\rightarrow} (s, \triangleright\triangleright 0, 1\sqcup\sqcup) \overset{M}{\rightarrow} (q_0, \triangleright\triangleright\triangleright, 1\sqcup\sqcup) \overset{M^2}{\rightarrow} (q_0, \triangleright\triangleright\triangleright 1\sqcup, \sqcup) \overset{M}{\rightarrow} (q'_0, \triangleright\triangleright\triangleright 1, \sqcup\sqcup) \overset{M}{\rightarrow}$$
("no", $\triangleright\triangleright\triangleright 1$, $\sqcup\sqcup$).

We have informally noted that TMs can be applied in various tasks. The following definition makes this precise:

Definition 3.4 (= adapted from Papadimitriou Def. 2.3):

- Let $L \subseteq (\Sigma - \{\sqcup, \triangleright\})^*$ be a language and M a TM which on input $x \in (\Sigma - \{\sqcup, \triangleright\})^*$ halts with "yes" if $x \in L$ and with "no" if $x \notin L$. Then we say that *M decides L*. If L is decided by some TM, L is called a *recursive language*.
- We say that *M accepts L* if on input $x \in (\Sigma - \{\sqcup, \triangleright\})^*$ M halts with "yes" if $x \in L$, but if $x \notin L$, then $M(x) = \nearrow$. We say that L is *recursively enumerable*.
- Let f: $(\Sigma - \{\sqcup, \triangleright\})^* \to \Sigma^*$ be a totally defined string function, and let M be a TM with alphabet Σ. Then we say that *M computes f*, if for all $x \in (\Sigma - \{\sqcup, \triangleright\})^*$, $M(x) = f(x)$. If such M exists, f is called a *total*[*ly*] *recursive* function.
- Let f: $(\Sigma - \{\sqcup, \triangleright\})^* \to \Sigma^*$ be a partially defined string function, and let M be a TM with alphabet Σ. Then we say that *M computes f*, if for all $x \in (\Sigma - \{\sqcup, \triangleright\})^*$, $M(x) = f(x)$ if $f(x)$ is defined and $M(x) = \nearrow$ if $f(x)$ is undefined. f is then called a [*partial*[*ly*]] *recursive* function.

Note that this definition does not include our case of problem solving, and that the task of computing functions is here stated not only for numerical functions but for any functions from words (= "strings") to words. Don't be irritated by such mismatches and variations. Every author uses his/her own versions of definitions. I used Papadimitriou's version here to stay consistent with that book. All such definition variants can be transformed into one another. We will become quite experienced in such transformations in this lecture. For the time being, just observe that problem solving can be considered as a case of language decision in a natural way (as outlined above).

We point out a few basic facts about the relationships between recursively enumerable and recursive languages.

Proposition 3.1.

1. If L is recursive, then L is recursively enumerable.
2. If L is recursive, then $L^c = \Sigma^* \setminus L$ (the complement language of L), is recursive.
3. L is recursive if and only if both L and L^c are recursively enumerable.

Proof. 1. If M is a TM deciding L, we obtain from M another TM M' that accepts all words from L with a "yes" and goes into an infinite loop on other inputs, by changing the "no" producing transitions in M's table into transitions that enter some trivial infinite loop.
2. If M is a TM deciding L, we obtain from M another TM M^c deciding L^c by simply reversing the "yes" and "no" answers in M.
3. " $\Rightarrow$ ": Follows from 1. and 2. " $\Leftarrow$ ": If L and L^c are recursively enumerable, we have a TM M which reports "yes" on every input $w \in L$ and doesn't halt on input $w \notin L$, and we have another TM M^c which reports "yes" for $w \notin L$ and runs forever for $w \in L$. Combine M and M^c into a new TM K which decides L, by making K emulate both M and M^c simultaneously (K alternatively emulates steps of M and M^c). On any input w, either the M-emulating or the M^c-emulating "branch" of K produces its "yes". According to which of the two does, K terminates in "yes" or "no" state. □

The word "recursively enumerable" should be explained at this point. By definition a language L is recursively enumerable iff there is a deterministic TM M that accepts L, that is, M halts with "yes" on all words of L and runs forever for inputs not in L. We can construct from M another TM N which "enumerates" L, that is, on empty input writes a (possibly infinite) output of the form $w_1 \sqcup w_2 \sqcup w_3 \sqcup \ldots$ such that $L = \bigcup_i \{w_i\}$ (repetitions in N's output are admissible). N systematically simulates M on all possible input words x, by generating these

words in lexicographical order and simulating M on each of the inputs. A suitable "dovetailing" (or time-sharing) scheme ensures that arbitrarily long runs on M on any input x are captured. For instance, N might first simulate M for maximally 10 steps on the first 10 inputs (in lexicographical order), then simulate M for maximally 100 steps on the first 100 inputs, then for 1000, ... etc. Whenever in one of these (possibly truncated) simulation runs the simulated machine M accepts its current simulated input, N writes this input to its output tape.

3.3 Busy Beavers

It seems that TMs are not very efficient computing devices – awkwardly low-level, and bound to stumble over their own symbols because they only have a *linear* string as memory. But this perception is misguided and only reflects our prejudices that programming should consist in convenient access to a RAM and if-then-else statements. In fact, even very small TMs can perform astounding acts which could not easily be replicated with equally small "ordinary" programs.

TMs for the **Busy Beaver** (BB) task are a point in case. Wikipedia gives a comprehensive introduction. A standard and very readable scientific introduction paper[27] has been written by one of the busiest Beaverists, Heiner Marxen. The TM simulator at http://morphett.info/turing/turing.html has the 4-state BB pre-configured. The BB task is spelled out for a variant of TMs with two-sided infinite tape and alphabet $\Sigma = \{0,1\}$ (no special tape-boundary symbol $\triangleright$). The TM is started on an all-0 tape in its starting state. The goal is to write as many 1's as possible and halt. The maximal number of 1's possible (there may be 0's interspersed between them) with an n-state TM is called $\sigma(n)$. The function $\sigma(n)$ is also known as the Rado function, after the mathematician who first introduced the busy beaver problem. An n-state TM that achieves this length is called a Busy Beaver. The number $\sigma(n)$ increases very rapidly with n. Busy Beavers are known only for $n \leq 5$. Here is a table which gives the current state of the art (from Heiner Marxen's BB webpage[28]):

n	$\sigma(n)$ = # of 1's	# of steps	Authors
1	1	1	Lin & Rado
2	4	6	Lin & Rado
3	6	21	Lin & Rado
4	13	107	Brady
5	>= 4098	>= 47,176,670	Marxen & Buntrock
6	$> 3.514 * 10^{18276}$	$> 7.412 * 10^{36534}$	Kropitz

The Rado function has some other fascinating properties beyond being fast-growing. For instance, within the framework of our standard mathematical proof methods we can determine only finitely many values of the Rado function (so far, 4 values have been determined). This and related facts have recently been proven; for the mathematically inclined here here is a little sniplet from one of the authors' blogs[29]:

[27] Heiner Marxen, Jürgen Buntrock, Attacking the Busy Beaver 5, Bulletin of the EATCS, Number 40, February 1990, pp. 247-251, http://www.drb.insel.de/~heiner/BB/mabu90.html

[28] http://www.drb.insel.de/~heiner/BB/

[29] Scott Aaronson: The 8000th Busy Beaver number eludes ZF set theory: new paper by Adam Yedidia and me http://www.scottaaronson.com/blog/?p=2725 . Copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/AaronsonBlogpage.zip (retrieved Feb 3, 2017)

"But the BB function has a second amazing property: namely, it's a perfectly well-defined integer function, and yet once you fix the axioms of mathematics, only finitely many values of the function can ever be *proved*, even in principle. To see why, consider again a Turing machine M that halts if and only if there's a contradiction in ZF set theory. Clearly such a machine could be built, with some finite number of states k. But then ZF set theory can't possibly determine the value of BB(k) (or BB(k+1), BB(k+2), etc.), unless ZF is inconsistent! For to do so, ZF would need to prove that M ran forever, and therefore prove its own consistency, and therefore be inconsistent by Gödel's Theorem."

3.4 TMs with multiple tapes

Designing a single-tape TM for a given task is often awkward because there is only one read/write head which must move on a linearly organized memory. If portions of data have to be shifted in memory, this leads to annoying low-level subroutines. Machines with several tapes, each of which is read/written by its own head, are more convenient to use. In this section we will see that by introducing such augmented versions of TMs, we obtain machines that can compute the same functions as single-tape TMs. (Remember that this is different for pushdown automata: adding stack memories properly increases the number of acceptable languages).

While in the first half of this lecture we will concentrate on the question which functions can be computed by machines *at all* (the question that constitutes the *theory of computation*) we will also already introduce some terminology and elementary insights relating to the question of *how fast* functions can be computed by machines (the topic of the *theory of complexity*). We will see that multi-tape TMs typically run faster than single-tape TMs, but not *much* faster.

Definition 3.5. A k-tape TM (where $k \geq 1$) is a structure $M = (K, \Sigma, \delta, s)$, where K, Σ and s are like in single-tape TMs. The transition function δ is augmented to cope with k tapes simultaneously by putting $\delta: K \times \Sigma^k \to (K \cup \{h, \text{"yes"}, \text{"no"}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$. The semantics is obvious. Initialization conventions: all tapes are left-bounded by $\triangleright$, and initially are blank except for the first tape which contains the input word. The output of a k-tape TM which stops in the halting state h is, by convention, the symbol string that is written on the last tape.

A *configuration* of a k-tape TM is a $(2k+1)$-tuple $(q, w_1, u_1, ..., w_k, u_k)$, where w_i, u_i describes the contents of the i-th tape with the convention that the i-th cursor points to the rightmost symbol of w_i. Single-step transitions $(q, w_1, u_1, ..., w_k, u_k) \xrightarrow{M} (q', w'_1, u'_1, ..., w'_k, u'_k)$, t-step transitions $\xrightarrow{M^t}$ and finite-length transitions $\xrightarrow{M^*}$ are defined as in the 1-tape case. The notions of acceptance and rejection of input are defined in the same way as for single-tape TMs.

In the second half of this lecture, we will use k-tape TMs as our reference model for measuring computational complexity. Here we introduce the requisite terminology:

Definition 3.6 If for a k-tape TM M and input w we have $(s, \triangleright, w, \triangleright, \varepsilon, ..., \triangleright, \varepsilon) \xrightarrow{M^t} (H, w_1, u_1, ..., w_k, u_k)$, where $H \in \{h, \text{"yes"}, \text{"no"}\}$, then the *time required by M on input w* is t. Let n denote the word length of inputs w. We say that *M operates within time f*(n), if for any input

word w of length n the time required by M on input w of is at most $f(n)$. $f(n)$ is a *time bound* for M.

Note that f need not be a tight bound: if $f'(n) \geq f(n)$ for all n, and M operates within time $f(n)$, then M also operates within time $f'(n)$.

Next comes a fundamental definition of complexity theory.

Definition 3.7 Suppose that a language $L \subseteq (\Sigma - \{\sqcup, \triangleright\})^*$ is decided by a some multi-tape TM within time $f(n)$. We say that $L \in \mathbf{TIME}(f(n))$.

For a given f, $\mathbf{TIME}(f(n))$ is a *set of languages*, namely, the set of languages that can be decided by some multi-tape TM within time $f(n)$. $\mathbf{TIME}(f(n))$ is a *compexity class*. We will write $L \in \mathbf{TIME}_k(f(n))$ if L is decided by a k-tape TM within the time bound $f(n)$.

Make sure that you understand the following box statements perfectly.

$\mathbf{TIME}(f(n))$ is a set of languages. It contains exactly those languages which can be decided by *some* multiple-tape TM within the time bound $f(n)$.

Conventions / traditions / imperfections. In complexity theory, one is interested in the asymptotic computation time that some TM needs as the input word length grows longer. How a particular TM performs on the few possible inputs of short length does not matter. However, from a rigorous mathematical perspective, the small-input-length behaviour sometimes is troublesome. For instance, an interesting class of languages is $\mathbf{TIME}(n^2)$, the languages decidable in quadratic time. But what happens when the input is the empty word ε? It has length 0 and therefore, because $0^2 = 0$, must be accepted or rejected in zero steps, which means that the starting state of the TM must already be accepting or rejecting state, which would make this poor machine *always* stop before it even has started. To avoid this and similar pitfalls, various strategies are employed in the literature:

1. Papadimitriou makes a convention (Definition 1.1. in his book!) that all numerical functions, regardless of their standard mathematical definition, always return the smallest nonnegative integer larger or equal the correct mathematical value. This would mean that, for instance, $0^2 = 1$ or $\log(0) = 1$.
2. In some textbooks, $\mathbf{TIME}(f(n))$ is defined in a slightly different way, namely, $\mathbf{TIME}(f(n))$ is there defined as the set of languages that can be decided by some multiple-tape TM within the time bound $g(n)$, where $g(n) = O(f(n))$. If we wish to use this convention in these lecture notes, we will write $\mathbf{TIME}(O(f(n)))$.
3. In the first (much more detailed and rigorous than the second) edition of the Hopfcroft/Ullman book on formal languages and computability[30], it is stated explicitly that $\mathbf{TIME}(f(n))$ should be read as $\mathbf{TIME}(\max\{n + 1, \lceil f(n) \rceil\})$ and $\mathbf{SPACE}(f(n))$ as $\mathbf{SPACE}(\max\{1, \lceil f(n) \rceil\})$.
4. Some authors admit only inputs of length > 0. In complexity theory that makes sense, because there the languages to be decided typically code instances of some problem, and such codewords never have length 0.

[30] Hopcroft, J.E., Ullman, J.D., Introduction to Automata Theory, Languages and Computation. Addison-Wesley 1979

5. Most authors simply don't care, don't comment on this issue and tacitly assume that any statements about computational complexities should be understood as referring only to large enough n. Papadimitriou is a good (or bad) example in this respect!

In this script, I will generally assume that languages or problems to be decided do not contain the empty word or a zero-length problem coding. – The problem of zero input length is just one instance of a technical complication that arises for small input lengths. In this script, I try to be as precise as possible. At various places, where other such complications turn up, I make them explicit and describe ways to circumvent them.

Multiple vs. single tape time complexity. Obviously, a multitape TM can solve a given problem at least at fast as a single-tape TM. How large is the speedup that we can gain by using multiple tapes? This question is partially answered by the following propositions.

Proposition 3.2 Given any k-tape Turing machine M operating within time $f(n)$, we can construct a single-tape Turing machine M' operating within time $O(f(n)^2)$ such that, for any input x, $M(x) = M'(x)$. (We say, *M' simulates M* with quadratic loss in efficiency).

Remark. The following proof is typical for many proofs in complexity: the workings of one system are *simulated* by the workings of some other mechanism. Usually these proofs are tedious and provide little insight through their technical detail – that's why the simulations occurring in such proofs are often described very superficially. However, there is always the danger that through being sloppy one brushes away some crucial little snag, and ends up with at faulty proof. I am afraid that this happens more often than one might believe – and because proofreaders also do not bother much about the nitty-gritty detail of simulation proofs, such faults will have a strong tendency to go unnoticed for a long while.

Proof of Prop. 3.2, sloppy version. Set up M' such that for every simulated step of M it has on its tape the contents of the k tapes of M in concatenated form (use extra delimiting symbols between each tape representation). The current head positions of the k heads of M are coded in this representation by extra symbols. For simulating one step of M, M' must traverse its entire tape once in order to assemble the information that goes into the left-hand side of the transition function δ of M, and traverse it a second time to make the changes in each of the k tape copies corresponding to a one-step transition of M. Complication: if M prolongs the inscription on one of its tapes to the right, M' has to shift the remaining copies one place to the right. In the worst case this incurs at most another k back-and-forth traversal of M's tape. Now, since M runs in time $f(n)$, none of its k tape inscriptions ever becomes longer than $f(n)$. Taking all together, M' traverses its tape of length $\le k\, f(n)$ at most $2 + 2\, k$ times for simulating a single step of M, and thus in time $O((2 + 2\, k)\, k\, f(n)^2) = O(f(n)^2)$ [since k is a constant] for the complete simulation of $f(n)$ steps of the simulated TM.

Prop. **3.2**, taken together with the obvious observation that any k-tape TM can simulate a single-tape TM with no loss of time, implies that any k-tape TM can simulate any other l-tape TM ($k, l \ge 1$) with at most quadratic loss in time. Ignoring the subtelties which distinguish one polynomial from another, in the relaxed perspective of complexiticians what counts is the fact that one TM simulating another incurs a *polynomial* loss (they don't care whether it's quadratic or cubic or to-the-tenth!). More formally, we say

Definition 3.8: two functions $f, f': \mathbb{N} \to \mathbb{N}$ are *polynomially related* if there exists a polynomial p such that $f(n) \le p(f'(n))$ and $f'(n) \le p(f(n))$.

Then, what we have learnt is that if one TM solves a problem in time $f(n)$ and another TM in $g(n)$, then g and f are polynomially related if both TMs were designed without stupidity, or stated in yet another way, for large parts of complexity theory it does not matter which kind of TM we use.

We will now consider the question whether the bound of Prop. **3.2** is tight, i.e. whether we can simulate a k-tape TM on a single-tape machine with better than quadratic loss. The answer is, in general, no. We will prove that a particular language, which can be decided by a 2-tape TM in linear time, cannot be decided on a 1-tape TM in better than quadratic time. The proof introduces an elementary scheme for finding lower complexity bounds which can be used for other problems, too.

I follow the book of W. Paul.

We first prove a general-purpose lemma. Consider a single-tape TM M. Note that if M decides some language, M can be modified such that it reaches its "yes" or "no" state always with its head on the left tape delimiter symbol ▷. The additional time can at most double the running time of the original program, so we can assume without loss of generality that M halts with its cursor at the leftmost position.

Consider a particular cell $i > 0$ on the tape and run M on input w (we number cells such that the leftmost has index $i = 0$) Whenever in the ensuing computation the cursor crosses from cell $i - 1$ to cell i or vice versa, note down the state M is in directly after this crossing step. The sequence of states obtained this way is called the *crossing sequence* at position i of M run on w, and is denoted by CS(w, i).

Lemma 3.3: Let M be a single-tape TM deciding a language L. Let u, v, w, x be words, and let CS(uv, $|u|$) = CS(wx, $|w|$). Then $ux \in L$ iff $uv \in L$.

The **proof** follows almost immediately from the following diagrams (Fig. 3.2) which show the head position of our TM on inputs uv, wx, ux in a case where CS(uv, $|u|$) = CS(wx, $|w|$) = s_1, s_2, s_3, s_4.

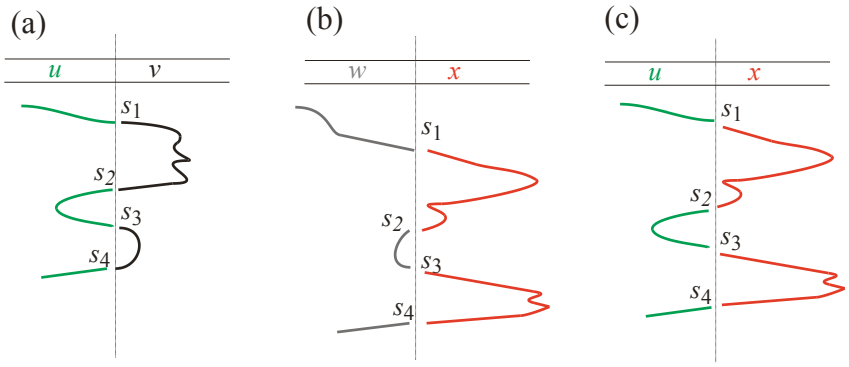


Fig. 3.2 cross-coupling crossing sequences.

If you "cut open the computation" for uv and wx at positions $|u|$ and $|w|$, (Fig. 3.2 (a) and (b)) and reconnect them as in Fig. 3.2 (c), it is clear that you get a run that accepts ux iff M accepts uv. □

We will now consider a language decision problem which demonstrates that the bound in Prop. 3.2 is tight in certain cases.

Proposition 3.4 Let $\Sigma \supseteq \{a, 0, 1\}$ be a tape symbol alphabet. The language $L = \{w a^{|w|} w \mid w \in \{0,1\}^*\}$ has the following properties:

1. $L \in \mathbf{TIME}_1(O(n^2))$
2. $L \in \mathbf{TIME}_2(O(n))$
3. $L \notin \mathbf{TIME}_1(f(n))$ for $f(n) = o(n^2)$

Note: the notation $f(n) = o(g(n))$ [pronounce "small-oh"] means that $\lim_{n\to\infty} f(n) / g(n) = 0$, i.e. f grows asymptotically slower than g (note: if $f(n) = o(g(n))$, then also $f(n) = o(C\, g(n))$ for any constant factor C.)

Proof. We only show 3. and leave 1. and 2. as easy exercises. Let M be a single-tape TM deciding L with time bound $f(n)$, with tape alphabet Σ and state set K. Let $m \in \mathbb{N}$ and let $n = 3m$. For any $i \in \{m+1, ..., 2m\}$ and $w, w' \in \{0,1\}^m$, $w \neq w'$, it must hold that $CS(w a^m w, i) \neq CS(w' a^m w', i)$ because otherwise M would accept $w a^m w'$ by Lemma 3.3. Let $p(i)$ be the average length of crossing sequences at position i in accepting runs, i.e.

$$p(i) = \frac{1}{2^m} \sum_{w \in \{0,1\}^m} \left| CS(w a^m w, i) \right| .$$

At least half of the $w \in \{0,1\}^m$ (that is, at least 2^{m-1} many) must satisfy $|CS(w a^m w, i)| \leq 2\, p(i)$. Thus there are at least 2^{m-1} many crossing sequences at position i with length not exceeding $2\, p(i)$. On the other hand, for combinatorial reasons there exist at most $(|K|+1)^{2p(i)}$ different crossing sequences of length less or equal $2\, p(i)$. These two findings imply $(|K|+1)^{2p(i)} \geq 2^{m-1}$ or equivalently, $p(i) \geq (m-1) / (2 \log_2 (|K|+1))$. The running time of M on input $w a^m w$ is certainly at least $\sum_{m<i\leq 2m} |CS(w a^m w, i)|$. The average runtime T_{avacc} of accepting runs (given m) then can be bounded from below by

$$\begin{aligned} T_{avacc} &\geq \frac{1}{2^m} \sum_{w \in \{0,1\}^m} \sum_{m<i\leq 2m} \left| CS(w a^m w, i) \right| \\ &= \sum_{m<i\leq 2m} p(i) \\ &\geq m(m-1)/(2\log(|K|+1)) \\ &= \Omega(n^2) \end{aligned}$$

(We say $f = \Omega(g)$ iff $g = O(f)$). Therefore, the runlength of at least one accepting run must be at least $\Omega(n^2) = \alpha\, n^2$ for some positive number α which depends only on L. □

Space complexity. Next to runtime, the most important efficiency aspect of computations is how much memory they require. We finish our introduction of multitape TMs by considering *space complexity*.

The most straightforward way to measure the space used by a k-tape TM would be to count all the tape cells that are visited during a computation, summed over all tapes. However, this would give the input and output tapes the same weights as the "internal" tapes. This does not conform to our intiutions: Consider one of the old-fashioned desk calculators that printed out their results on a band of paper. It has no RAM to speak of, but during a day's work in a busy office receives a large amount of input (what the operator hacks in) and produces a long string of output. The important observation here is that the input is read-only (the calculator cannot overwrite the typed-in input) and the output is write-only. This motivates to use the following convention for k-tape TMs when we are investigating space complexity:

Definition 3.9: A *TM with input and output* is a k-tape TM with at least 2 tapes. The first tape is designated the *input tape* and the last tape the *output tape*. They are read-only and write-only, respectively, in the sense that every transition rule $\delta(q, a_1, ..., a_k) = (p, b_1, D_1, ..., b_k, D_k)$ must satisfy $b_1 = a_1$ (i.e., an input symbol must be overwritten by itself) and $D_k \neq \leftarrow$ (i.e., the output cursor cannot move left).

It is easy to see that for every k-tape TM operating within time bound $f(n)$ there is a k+2-tape TM with input and output operating in time $O(f(n))$.

Definition 3.10: The *space required* by a k-tape TM M with input and output (on a computation that halts) is the number of cells on tapes 2 to $k - 1$ which are visited during the computation. We say that M operates within space bound $f(n)$ if for any input of length n, M requires space at most $f(n)$. Finally, let L be a language decided by some TM with input and output within space bound $f(n)$. Then L is in the *space complexity class* **SPACE**($f(n)$).

Many polynomial-time computations are in **SPACE**(log n). Because of its practical importance, this complexity class has gotten its own name and is usually referred to simply as **L**.

Linear speedup. In complexity theory one makes liberal use of the O() notation, that is, one does not care about additive and multiplicative constants when it comes to measuring computation costs. This common practice can be justified by the observation that within the realm of multiple-tape TMs, one can always speed up computations linearly by any desired factor:

Proposition 3.5: Let $L \in$ **TIME**($f(n)$). Then, for any $\varepsilon > 0$, also $L \in$ **TIME**($\varepsilon f(n) + n + 2$).

The proof proceeds by showing how a k-tape TM M which decides L within time $f(n)$ can be simulated by a k'-tape TM M' running in time $\varepsilon f(n) + n + 2$, where $k' = \max(2, k)$. The basic idea for the simulation is to use for the alphabet of M' symbols which correspond to blocks of some length m of symbols of M. Processing such "block symbols" instead of the "elementary symbols" of M speeds M' up by a constant factor times m. By suitably selecting m (the larger, the faster the speedup) the required speedup of ε can be ascertained. The n in the speed-up time comes from the fact that the simulating TM M' first has to scan the original input and transform it into a block-symbol version. Papadimitriou provides a semi-sloppy proof which however runs over two printed pages. A thoroughly non-sloppy version of this proof is a

tedious exercise, of the kind "you have to do something like that only once in your lifetime". I assume that you suffered your way through one of the detailed TM programming exercises on the exercise sheet, and therefore neither you nor I have to carry out this odious task here. (This is the way to hell: grossly inaccurate results may accumulate in the literature in this way!).

4 Random access machines

Alan Turing arrived at TMs by reasoning from first principles. His goal was to find the simplest possible model for a computing machine that could perform *all* computations that *any* finite machine (or brain) can do. Today, TMs are universally accepted as such a most simple, but entirely sufficient, model of computation. Why? We have to convince ourselves that TMs actually have "total computation power". A complete understanding of this problem will be obtained in several further parts of this lecture. We will begin our quest by demonstrating that TMs can perform the same computations that your PC can perform. Concretely, in this chapter we will first introduce a simplified, but realistic model of modern PCs, the random access machine (RAM) and then show that RAMs can carry out the same computations as TMs.

A RAM is a simplified and formalized version of our current standard computer CPU architectures. Compared to TMs, RAMs can be "programmed" in the same way as standard PCs (using a kind of mixture programming language between assembler and Basic). Thus, it is generally easier and closer to our everyday programming practice to devise a RAM program for a given task, than it is to find an equivalent TM. The drawback of RAMs is that they are more complicated than TMs and thus are less suited as a machine model for mathematically investigating (and understanding) the truly fundamental properties of computation.

Unlike TMs, which are usually specified for arbitrary symbol sets and are suited for language decision problems just as well as for arithmetics, RAMs are designed to perform numerical computations.

In my rendering of RAMs I follow the book W.J. Paul, Komplexitätstheorie, Teubner Studienbücher Informatik, Teubner 1978. The description of RAMs in the Papadimitriou book is flawed and should be avoided [integers of any size can be read in in constant time in Papadimitriou, but correctly they must be read in in time proportional to input bitlength].

A RAM consists of a central processing unit (CPU), a random access memory, an **input tape**, an **output tape**, and a **program**. The memory consists of countably infinite many **registers**, enumerated by 1, 2, ... The number of a register is called its **address**. Every register can hold an arbitrary natural number. The program is a finite sequence of **commands**. There are only a few basic commands, which we will list below. The CPU consists of two items: (i) a special register called **accumulator**, which has register number 0, and (ii) a **command counter** κ. The accumulator can hold an arbitrary integer or single symbols from the **finite input/output alphabet** Σ. The command counter can hold a positive number which corresponds to the currently executed command line in the program.

The input tape contains a read-only word $w \in \Sigma^*$. The RAM can move its read head only from left to right (i.e. can read each input symbol only once). On the output tape, the RAM

can move its head only from right to left, printing one symbol from Σ at each step: a model of a simple printer.

At each time step, the *configuration* of a RAM is a quintuple (κ, c, w, p, x), where

- κ is the contents of the command counter (initialized to 1),
- c: $\mathbb{N} \to \mathbb{N} \cup \Sigma$ is a function where $c(i)$ is the contents of register i, (they are all initialized to zero). Note that only $c(0)$ can be a symbol from Σ, the other registers must contain integers.
- $w \in \Sigma^*$ is the input word,
- $p \in \{1, ..., |w|+1\}$ is the current position of the read head,
- $x \in \Sigma^*$ is the word written on the output tape.

In each single step, the executed command (i) changes the contents of at most one register, (ii) changes the command counter, (iii) possibly changes the position of the reading head, and (iv) possibly outputs one symbol. Table 4.1 lists all possible commands.

Command	Result
`load i`	$c(0) = c(i)$; $\kappa = \kappa+1$
`store i`	$c(i) = c(0)$; $\kappa = \kappa+1$
`add i`	$c(0) = c(0)+c(i)$; $\kappa = \kappa+1$
`sub i`	$c(0) = \max(c(0) - c(i), 0)$; $\kappa = \kappa+1$
`read`	$c(0) = w_p$; $p = p+1$; $\kappa = \kappa+1$
`print a`	$x = ax$; $\kappa = \kappa+1$ [$a \in \Sigma$]
`shift`	$c(0) = c(0)/2$ if $c(0)/2$ is integer else $(c(0) – 1)/2$; $\kappa = \kappa+1$
`goto i`	$\kappa = i$ [$i > 0$]
`if c(0) = i goto j`	if $c(0) = i$ then $\kappa = j$ else $\kappa = \kappa+1$ ($i \in \mathbb{N} \cup \Sigma$)
`ind load i`	$c(0) = c(c(i))$; $\kappa = \kappa+1$ [indirect addressing]
`ind store i`	$c(c(i)) = c(0)$; $\kappa = \kappa+1$ [indirect addressing]
`ind add i`	$c(0) = c(0) + c(c(i))$; $\kappa = \kappa+1$ [indirect addressing]
`ind sub i`	$c(0) = \max(c(0) - c(c(i)),0)$; $\kappa = \kappa+1$ [indirect addressing]
`c load i`	$c(0) = i$; $\kappa = \kappa+1$ ($i \in \mathbb{N}$) [loading a constant]
`c add i`	$c(0) = c(0) + i$; $\kappa = \kappa+1$ ($i \in \mathbb{N}$)
`c sub i`	$c(0) = \max(c(0) - i, 0)$; $\kappa = \kappa+1$ ($i \in \mathbb{N}$)
`end`	stop

The commands `store`, `add`, `sub`, `shift`, `ind store`, `ind add`, `ind sub` are defined only if the accumulator contains an integer, else they are undefined.

At the start of a run, all registers are initialized to hold 0.

Table 4.1 The command set of a RAM

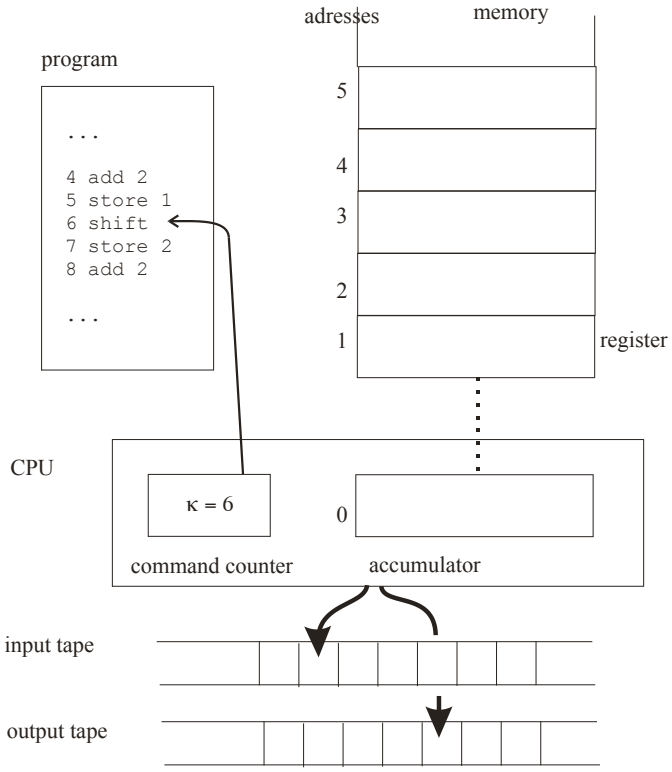


Figure 4.1 Components of a RAM.

Here is a RAM program for addition. The input is of the form *u*#*v*#, where *u* and *v* are binary representations of integers. The RAM has to stop with the binary representation of the sum of these integers on the output tape. The following program does this. The commands 1 – 8 are executed $|u|$ times in a row, whereby the input $u = u_1 \ldots u_{|u|}$ (a string!) is transformed into $\sum_{i=1}^{|u|} u_i 2^{|u|-i}$ (a number!) and saved in register 1. A similar procedure in steps 9 – 16 transforms the symbol string *v* into its corresponding integer, which is held in register 2. The actual addition takes place in steps 17 – 18. The remaining commands (omitted here) serve to write the sum on the output tape.

```
1     read
2     if c(0)= # goto 9
3     store 3
4     load 1
5     add 1
6     add 3
7     store 1
8     goto 1
9     read
10    if c(0)= # goto 17
11    store 3
12    load 2
13    add 2
14    add 3
15    store 2
16    goto 9
17    load 1
18    add 2
19    ...
... end
```

If *P* is a RAM program and $w \in \Sigma^*$, we write $P(w) = u$ if the RAM with program *P* enters the `end` command with output *u* on input *w*, and we write $P(w) = \nearrow$ if the RAM with program *P* on input *w* does never enter the `end` command (which may happen if the program runs infinitely long or enters an undefined situation).

The *uniform computation time* of a RAM which on input w runs k steps and then halts is just k: we simply count the number of steps. Specifically, we do not care in our time measuring how large the integers are which are fetched/stored from/to registers, or are processed in the accumulator. Thus, if a RAM processing inputs of length n runs at most $f(n)$ steps, we say it has (uniform) time complexity $f(n)$.

This may not appear too realistic, because on real machines, the time to access an integer n is asymptotically proportional to $\log(n)$. Therefore one also uses *logarithmic computation time* where the costs of memory access and arithmetic operations scales logarithmically with the size of the integers processed.

However, the time that we measure when we use the simple uniform time complexity is polynomially related with the more realistic logarithmic time. The fact that logarithmic and uniform RAM times are polynomially related is a consequence of the following two propositions, which show that both times are polynomially related with TM time:

Proposition 4.1 (Simulation of TM by RAM) A TM (with arbitrary number k of tapes) that has time complexity $f(n)$ can be simulated by a RAM in uniform time $O(f(n))$ and in logarithmic time $O(f(n) \log f(n))$.

This proposition is not surprising, inasmuch as RAMs are intuitively more powerful than TMs. The proof is a tedious exercise in coding whose detail we omit. The basic idea is that any configuration of the TM can be represented in the registers of a RAM. We illustrate how this can be done for a 2-tape TM with symbol set $\Sigma = \{a_1,..., a_n\}$ and state set $K = \{q_1, ..., q_m\}$. Assume the current configuration of the TM is $(q_{10}, a_3a_7a_8a_9, \varepsilon, a_5a_6, a_1)$. The position of the two cursors are on the 4th and 2nd tape cell, respectively. All this information is contained in the sequence of integers 10 4 2 3 5 7 6 8 1 9 0: the leading 10 indicates that the TM is in state q_{10}, the next two numbers 4 and 2 indicate cursor positions, and the remaining numbers represent the tape contents (they should be interpreted pairwise, so the subsequence 3 5 means that the first symbol on tape 1 is a_3 and the first symbol on tape 2 is a_5). A "0" indicates a blank cell. Call this sequence a *configuration coding sequence*. This sequence of 11 integers is the contents of the registers $c(3)$, $c(4)$, ..., $c(13)$: $c(3) = 10$, $c(4) = 4$, ..., $c(13) = 0$. It is clear that any TM configuration can be coded into registers $c(3)$, $c(4)$, ... in this way. The registers $c(1)$ and $c(2)$ are used for storing certain intermediate results that occur during a simulation run. One update step of the TM deterministically changes the configuration coding sequence in the first position (due to TM state change), the next two positions (due to cursor motion), and/or in two of the remaining positions (due to overwriting tape symbols). If the TM extends the tape used to the right, the configuration coding sequence grows by two integers to the right. It is intuitively clear that such a change of a configuration coding sequence can be "re-programmed" with a RAM. The time bounds claimed in the proposition are obtained from a careful analysis of the simulating RAM program. The factor $\log(f(n))$ in the logarithmic simulating time is due to the fact that address integers (which are needed in indirect addressing) are growing to (at most) $f(n)$.

Proposition 4.2 (Simulation of a RAM by a TM)

1. A RAM that has logarithmic time complexity $f(n)$ can be simulated by a 4-tape TM in $O(f(n)^2)$.
2. A RAM that has uniform time complexity $f(n)$ can be simulated by a 4-tape TM in $O(f(n)^3)$.

Part 1 of this proposition is again a coding exercise whose detail we omit. Analogous to the previous proposition, the main idea is that (i) the complete configuration of a RAM can be coded in three sequences of integers, (ii) these sequences can be written in a binary representation on the first three tapes of the simulating TM. The fourth tape is used for auxiliary computations. Of course, many variations are possible – the four-tape simulation is the one used in the Paul book. The complexity claim is yielded by an analysis of the simulating TM.

However, part 2 of Prop. 4.2 involves more than a straightforward (albeit tedious) coding exercise: how is it possible for a TM (which has to perform all "memory management" of integers on its tape bitwise, and furthermore must perform arithmetic operations bitwise) to simulate a uniform-time RAM (which doesn't care about bitlength of register contents) in polynomial time, considering that integers can become arbitrarily large (and thereby the difference between TM vs. RAM operation times on those integers)? Here an extra idea is called for.

The reason why part 2 of Prop. 4.2 can be proven is that additions are the only operations in our definition of RAM programs which can lead to longer integers – but the integer length can increase at most by 1 per addition. (This would be different if we also had a `multiply` command!). Therfore, the *increase* in time a TM needs to simulate a RAM step is at most linear with the ongoing uniform time of the simulated RAM. This fact must be used in the proof of the time bounds in Prop. 4.2.

5 Recursive functions and Church's thesis

We have seen that single-tape TMs, multi-tape TMs, and RAMs can compute the same (possibly partially defined) functions because any of these machines can simulate the others. In this section, we will characterize this same class of "computable" functions in a completely different way, which is reminiscent of how we defined the regular languages through regular expressions.

Recall that we defined the regular languages in an inductive fashion: as an induction basis, we said that $\boldsymbol{\varepsilon}$, $\boldsymbol{\varnothing}$, $\boldsymbol{a}$ were regular expressions denoting the languages $\{\varepsilon\}$, $\varnothing$, $\{a\}$ (for every $a \in \Sigma$). Then we considered the language operations of union, concatenation, and Kleene closure. We defined the regular languages as those languages that can be obtained from the basis languages $\{\varepsilon\}$, $\varnothing$, $\{a\}$ by applying some (finitely many) of the language operations.

In a similar fashion, we will now characterize a class of functions inductively. This class of functions is called the *recursive functions*. We will see that the class of recursive functions thus defined yields the same functions that can be computed by TMs. Historically, this inductive definition came (a little bit) earlier than Turing Machines. The motivation to define the recursive functions was to find a rigorous definition of "computable": the goal was to characterize exactly those functions that can be computed "effectively", that is, by some algorithm that can be formulated with a finite description and runs in finite time. The inductive basis is provided by some very simple functions which are clearly computable. Then we introduce a few mechanisms which construct more complex computable functions from already existing ones. The recursive functions, then, are all functions which can be obtained by a finite number of application of such mechanisms. (I follow here the book W.J. Paul, Komplexitätstheorie, Teubner Studienbücher Informatik vol. 39, Teubner: Stuttgart

1978). We follow the development of mathematical history in that we first give the definition of the *primitive recursive functions*, which cover only a subset of the recursive functions but which for some time were believed to cover all intuitively computable functions.

Definition 5.1: The class of **primitive recursive functions** $f: \mathbb{N}^k \to \mathbb{N}$ (where $k \geq 0$) is made from all functions which can be obtained by a finite number of applications of the following rules:

1. Introduce the **null constant**: the function $\mathbf{0}: \mathbb{N}^0 \to \mathbb{N}$, $\mathbf{0}() = 0$ is a primitive recursive function.
2. Introduce the **successor** function: the function $\sigma: \mathbb{N} \to \mathbb{N}$, $\sigma(n) = n+1$ is a primitive recursive function.
3. Introduce **projection** functions: the functions $p^r_i: \mathbb{N}^r \to \mathbb{N}$, $p^r_i(n_1, ..., n_r) = n_i$ for all $r \geq 1$ and $1 \leq i \leq r$ are primitive recursive functions.
4. Construct a new function by **substitution**: if $f: \mathbb{N}^r \to \mathbb{N}$ and $g_1,..., g_r: \mathbb{N}^m \to \mathbb{N}$ are primitive recursive (where $r \geq 1$ and $m \geq 0$), then also $h: \mathbb{N}^m \to \mathbb{N}$, $h(x) = f(g_1(x),..., g_r(x))$ is primitive recursive.
5. Construct a new function by **primitive recursion**: if $f: \mathbb{N}^r \to \mathbb{N}$ and $g: \mathbb{N}^{r+2} \to \mathbb{N}$ are primitive recursive (where $r \geq 0$), then also every function $h: \mathbb{N}^{r+1} \to \mathbb{N}$ is primitive recursive which satisfies the following equations:

    a. $h(0, x) = f(x)$,
    b. $h(n + 1, x) = g(n, h(n, x), x)$.

The rules 1. – 3. yield the inductive basis, a set of elementary functions from which new functions can be constructed using the function operations 4. and 5.

The primitive recursion scheme is an augmented version of the familiar principle of induction for natural numbers. A good way to understand it is to view x as *parameters* of the function h that one wants to define from f and g. If there are no parameters (case $r = 0$), the recursion scheme collapses to the more familiar-looking scheme (note that constants can be seen as 0-ary functions):

5'. (stripped-down special case of 5.) Construct a new function h as follows by "ordinary" recursion: if $c \in \mathbb{N}$ and $g: \mathbb{N}^2 \to \mathbb{N}$ is primitive recursive, then also every function $h: \mathbb{N} \to \mathbb{N}$ is primitive recursive which satisfies the following equations:

    a. $h(0) = c$,
    b. $h(n + 1) = g(n, h(n))$.

Note that the concatenation of functions corresponds to the special case $r = 1$ of rule 4: if $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N}^m \to \mathbb{N}$ are primitive recursive (where $m \geq 0$), then also $h: \mathbb{N}^m \to \mathbb{N}$, $h(x) = f(g(x))$, for which we also write $h(x) = f \circ g(x)$.

Example 5.1. We show how the function $add: \mathbb{N}^2 \to \mathbb{N}$, $add(x, y) = x + y$ is primitive recursive. The idea is to use the fact that addition on integers can be defined on the basis of the successor function σ recursively by (1.) for all x: $0 + x = x$ (basis of recursion); (2) for all x, y: $\sigma(y) + x = \sigma(x + y)$. Writing this with exactly the formalism given by the rules above, we get the following line of argument:

1. The identity function $id: \mathbb{N} \to \mathbb{N}$, $id(x) = x$, is primitive recursive by rule 3 because $id = p^1_1$.
2. The function $\varphi: \mathbb{N}^3 \to \mathbb{N}$, $\varphi(x, y, z) = \sigma(y)$ is primitive recursive, because by rule 2, σ is primitive recursive, and by rule 3, p^3_2 is primitive recursive, and therefore by rule 4, $\varphi(x, y, z) = \sigma \circ p^3_2(x, y, z) = \sigma(y)$ is primitive recursive.
3. Now, by rule 5, the function $add\ \mathbb{N}^2 \to \mathbb{N}$ which satisfies
   a. $add(0, x) = id(x)$
   b. $add(y+1, x) = \varphi(y, add(y, x), x) = \sigma(add(y, x))$
   is primitive recursive (where we have used *id* for the *h* that appears in rule 5, and φ for the *g* that appears in rule 5).

Notice how much care was used in this proof to really use *only* what is *exactly* provided by the 5 rules, in order to nail down the intuition of how to introduce addition inductively. Specifically, the primitive recursion rule 5 is more complex than what one is needed for many "ordinary" recursions that one meets in basic functional programming exercises. The complexity of rule 5 is however needed to master inductive function definitions that are more demanding than *add*. Conversely, for inductions that are as simple as is needed for *add*, one has to "project away" the unnecessary add-on features that rule 5 offers.

For some time it was an open issue (raised by Hilbert in 1926) whether the primitive recursive functions covered all functions which are intuitively computable. In 1928 Ackermann provided an example of a function which was clearly computable but not primitive recursive. This Ackermann function is a function which grows exceedingly fast; it is inspired by the idea of turning the sequence of ever-faster growing functions, $f_1(n) = n + 2$; $f_2(n) = 2*n$; $f_3(n) = 2\hat{}n$; ... , into a single, two-argument function which would for instance yield $f(1, n) = n + 2$; $f(2, n) = 2*n$; $f(3, n) = 2\hat{}n$. The actual definition of the **Ackermann function** $A: \mathbb{N}^2 \to \mathbb{N}$ yields slightly different results and installs a wildly recursive carousel, as follows:

Definition 5.2: The Ackermann function $A: \mathbb{N}^2 \to \mathbb{N}$ is defined by

$$A(0, n) = n + 1$$
$$A(m + 1, 0) = A(m, 1)$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

Clearly, the Ackermann function is computable in an intuitive sense: you could easily write a program that computes it. However, it can be shown that it is not primitive recursive. The intuitive reason is that the Ackermann function grows faster than any primitive recursive function ever could because primitive recursive functions can only work from a finite number of recursions, whereas *A* is "recursively recursive" and creates arbitrarily many levels of primitive recursions, according to its first argument.

The first argument in the Ackermann function specifies the function "type", the second argument the number of iterations. We have

$A(0, n) = n + 1$: "successor function",
$A(1, n) = n + 2$: "add 2",
$A(2, n) = 2n + 3$: "essentially, multiply by 2",
$A(3, n) = 2^{n+3} - 3$: "essentially, exponentiate 2 by *n*",

$A(4, n) = 2^{2^{2^{\cdot^{\cdot^{2}}}}} - 3$ (the tower of 2's has height $n + 3$, evaluation order is top-down)
$A(5, n)$ = ... (try to imagine)

A(4,2) is about 10^{19728}. This is reminiscent of busy beavers! (We will soon see that the busy beaver function is actually even worse than the Ackermann function).

Examples of non-primitive-recursive functions prompted an extension of the definition of primitive recursive functions, by introducing another function-construction mechanism, the μ-operator, which intuitively computes a minimum function:

Definition 5.3: For $f: \mathbb{N}^{r+1} \to \mathbb{N}$, $\mu f: \mathbb{N}^r \to \mathbb{N}$ is defined by

$$\mu f(n) = \begin{cases} \min\{m \mid f(m,n) = 0 \text{ and for all } k \le m \ f(k,n) \text{ is defined}\} \ (\text{provided such } m \text{ exists}) \\ \text{not defined, else} \end{cases}$$

Then, augment Definition 5.1 by the following construction rule.

2. Construct a new function by **μ-recursion**: if $f: \mathbb{N}^{r+1} \to \mathbb{N}$ is recursive, then μf is recursive.

The first 5 rules have to be adapted by replacing "primitive recursive" by "recursive" everywhere, and by specifying suitable conditions (in rules 4 and 5) to the effect that conditions of non-definedness are caught, as follows:

4. Construct a new function by **substitution**: if $f: \mathbb{N}^r \to \mathbb{N}$ and $g_1,..., g_r: \mathbb{N}^m \to \mathbb{N}$ are recursive (where $r \ge 1$ and $m \ge 0$), then also $h: \mathbb{N}^m \to \mathbb{N}$, $h(x) = f(g_1(x),..., g_r(x))$ is recursive, where $h(x)$ is defined for all x where all $g_i(x)$ are defined and $f(g_1(x),..., g_r(x))$ is defined.
5. Construct a new function by **primitive recursion**: if $f: \mathbb{N}^r \to \mathbb{N}$ and $g: \mathbb{N}^{r+2} \to \mathbb{N}$ are primitive recursive (where $r \ge 0$), then also every function $h: \mathbb{N}^{r+1} \to \mathbb{N}$ is primitive recursive which satisfies the following equations:

   a. $h(0, x) = f(x)$,
   b. $h(n + 1, x) = g(n, h(n, x), x)$.

   $h(n, x)$ is defined when $f(x)$ is defined, and for all $n' < n$ all $h(n', x)$ are defined, and all $g(n, h(n, x), x)$ are defined.

The new rules 1. – 6. yield the **μ-recursive** (or simply recursive) functions. Note that by using 6. we may arrive at functions which are not defined everywhere – i.e., *partial functions*. They can be shown to be the same class of functions as the TM-computable numerical functions. (Note that also a TM need not terminate on every input – in which case the TM, too, computes only a partial function).

The Ackermann function can be shown to be μ-recursive.

Church's thesis. We have seen by now that there are (at least) three equivalent ways of characterizing the partial functions that are "computable": TMs, RAMs, recursive functions. Soon we will learn about yet another such characterization, lambda calculus, which is as different as can be from TMs, RAMs, or recursive functions. The fact that such outwardly quite different formalizations lead to the same concept is remarkable and suggests that by the notion of partial recursive functions we have hit a truly fundamental concept.

The intuitions that led mathematicians and (later) computer scientists to develop these characterizations were not quite identical but similar:

- Turing invented TMs as a model of everything that any finite (human) intelligence can achieve by way of symbolic thinking.
- Recursive functions were developed by mathematicians in order to make the intuitive notion of "computable" precise.
- RAMs were designed as a simplified yet in principle realistic model of modern computers.
- Lambda calculus was invented by Church as an alternative, axiomatic approach to set theory as a basis for all of mathematics, driven by the idea that the most elementary notion of mathematics should not be static-structural (as in set theory) but dynamical: namely, function evaluation.

Thus, all of these scientific enterprises were driven, in one way or the other, by the desire to achieve a rigorous account of computation. The fact that all of these are equivalent can be proven – one basically has to "simulate" each of the formalisms by any of the others; this is sometimes tedious but not really difficult. But the fact that the notion of recursive functions indeed captures the intuitive notion of "computable" cannot be proven, because such a proof would have to show that an *intuitive* notion ("computability") is equivalent to a *formally defined* notion (recursive functions). But one cannot formally prove something about an intuitive notion. Therefore, the idea that the rigorous concept of recursive functions indeed captures all out intuitions about "computability" is and will remain a *hypothesis.* It is today known as the **Church hypothesis** or as the **Church-Turing hypothesis**.

Among traditional computer scientists it is generally believed to be true. There are, however, frequent attempts to widen the intuitive notion of "computable" in some nonstandard way, for instance incorporating timing of analog computations or even quantum effects. It's an interesting and speculative niche somewhere between philosophy, maths, physics, psychology and computer science. Wikipedia has an extensive entry on "Hypercomputation" with links to dedicated websites.

6 The halting problem and undecidability results

6.1 Universal Turing Machines

You can program your PC to simulate any TM (at least as long as you don't run out of memory, but well, "in principle" you can if you are prepared to buy further external memory whenever needed...). You can also program a TM U to simulate any other TM – such a TM U that simulates all TMs is called a *universal* TM. There are many ways of constructing universal TMs, all depending on some convention of how to code the simulated TMs. There is nothing remarkable or deep about universal TMs, once you have grown accustomed to the intuitions of coding and simulation. I briefly sketch the coding scheme proposed in Papadimitriou's textbook (there: chapter 3).

The objective of coding here is to define a convention that allows one to transform any TM $M = (K, \Sigma, \delta, q_0)$ into a codeword over some fixed coding alphabet – we take $\{0, 1, \#\}$. We use angular brackets to denote the codeword $\langle M \rangle \in \{0, 1, \#\}^*$ of M.

The coding $M \mapsto \langle M \rangle$ should be *effective*, i.e. computable by a machine, and *effectively reversible*, i.e. it must be possible to automatically reconstruct M from $\langle M \rangle$.

To devise a coding, we assume that each TM M is given in tabular form (the transition table contains all the information about M). The only technical difficulty in translating a table into a word is that Turing machines M may have tape alphabets and state sets of arbitrary sizes. This problem can be resolved by coding symbols and states by natural numbers, which are represented in $\langle M \rangle$ as binary strings separated by "#". The transition rules of M can be encoded as 5-tuples of integers in a straightforward way (note that $\delta(q, \sigma) = (p, \rho, D)$ can be coded by 5 integer codes for states, symbols, and cursor action). A moment's thinking will convince you that one can find simple coding schemes for arbitrary M and x using a code alphabet $\{0, 1, \#\}$, which we will assume in the rest of this section.

We notice that if a TM M gets input words x from M's tape alphabet, then one can devise another coding scheme, following the same principles, that translates x into a codeword $\langle x \rangle \in \{0, 1, \#\}^*$.

U has to be able to simulate how any given TM M acts on some input x. Technically, U gets the input $\langle M \rangle;\langle x \rangle$ (two binary coding words for M and x, respectively, separated by a special separator ";"). U is a multi-tape TM which simulates the action of M in a step-by-step fashion. U has a special tape on which the configurations of M are written in coded form. A sequence of steps of U which simulates one step of M transforms a coded configuration of M into a coded version of M's next configuration. When M halts in h, "yes" or "no", so does U. The result of M's computation can then be decoded from the coded configuration found on U's special tape.

We write $U(\langle M \rangle;\langle x \rangle) = M(x)$ to denote that U simulates M on input x. The value of $U(\langle M \rangle;\langle x \rangle)$ is therefore the value of $M(x)$, that is "yes", "no", or %for decision problems and a coded version of the output string of M (or %in the case of function computation problems.

Side notice. Universal TMs need not be terribly complex. There exists a universal Turing Machine with just 2 states and 3 symbols (rather more correctly, a 1-dim cellular automaton emulating a universal TM). Read more at www.wolframprize.org.

6.2 The halting problem and some immediate consequences

In the following treatment of the halting problem we depart from the Papadimitriou book (which falls in the sloppyness trap here and contains some technical errors / inaccuracies). We consider only a particular kind of TMs, namely those whose tape alphabet coincides with our coding alphabet $\{0, 1, \#\}$ (plus, of course, $\sqcup, \triangleright$}). Note that in this case we can "code" inputs x by the identity code, i.e. $\langle x \rangle = x$.

We now describe *the* classical example of an undecidable language. H consists of all words $\langle M \rangle;x \in \{0, 1, \#, ;\}^*$ which code a halting run of some TM M with tape alphabet $\{0, 1, \#\}$ in the following way:

H = $\{ \langle M \rangle; x \mid M(x) \neq \nearrow \}$ [shorthand]

= $\{ \langle M \rangle; x \in \{0, 1, \#, ;\}^* \mid \langle M \rangle$ is a code of some TM M which uses the tape alphabet $\{0, 1, \#\}$, $x \in \{0, 1, \#\}^*$, and $M(x) \neq \nearrow \}$ [more precise version]

The following is easy to establish:

Proposition 6.1: H is recursively enumerable.

Proof. We need a deterministic TM that accepts H, i.e. halts with "yes" on every input $\langle M \rangle; x \in H$. It is easy to modify U to work this way. □

The problem of deciding (not merely accepting) H is the famous *halting problem*, to which we will refer by HALTING. This problem is not decidable:

Proposition 6.2: H is not recursive.

Proof. The proof works by a method which is used for many undecidability results, a *diagonalization* argument. Suppose, for the sake of contradiction, that some TM M_H decides H. Modify M_H to obtain another TM D (with tape alphabet $\{0, 1, \#\}$) that behaves as follows on inputs $\langle M \rangle$. D first simulates M_H on input $\langle M \rangle;\langle M \rangle$. Since M_H is supposed to decide H, M_H on any input will eventually halt with "yes" or "no". Specifically, M_H on input $\langle M \rangle;\langle M \rangle$ will eventually halt with "yes" or "no", too. Arrange D such that if M_H on input $\langle M \rangle;\langle M \rangle$ halts with "yes", D enters an infinite loop that moves its cursor to the right indefinitely. If M_H on input $\langle M \rangle;\langle M \rangle$ halts with "no", D halts with "yes". Formally,

$D(\langle M \rangle)$ = if $M_H(\langle M \rangle;\langle M \rangle)$ = "yes" then %else "yes".

Now what is $D(\langle D \rangle)$? If $D(\langle D \rangle)$ = %then by definition of D we have that $M_H(\langle D \rangle;\langle D \rangle)$ = "yes", that is $\langle D \rangle;\langle D \rangle \in H$, that is, $D(\langle D \rangle) \neq \nearrow$. Conversely, if we assume $D(\langle D \rangle) \neq \nearrow$, then by definition of D we have $D(\langle D \rangle)$ = "yes", that is, M_H rejects $\langle D \rangle;\langle D \rangle$, that is $\langle D \rangle;\langle D \rangle \notin H$, that is, $D(\langle D \rangle) = \nearrow$. Thus our first assumption that some TM M_H can decide H leads into a contradiction and must be false. □

This type of proof is called a proof by diagonalization for the following reason. We may think of an infinite table whose rows and columns each are indexed by *all* TMs M_1, M_2, M_3,... with tape alphabet $\{0, 1, \#\}$. In cell (M, N) indexed by row TM M and column TM N of this table we write the result $M(\langle N \rangle)$ of the computation of M on input $\langle N \rangle$. On the diagonal we have all results $M_i(\langle M_i \rangle)$ of runs of TMs M started on their own codewords as input:

	M_1	M_2	M_3	...
M_1	$M_1(\langle M_1 \rangle)$	$M_1(\langle M_2 \rangle)$	$M_1(\langle M_3 \rangle)$	
M_2	$M_2(\langle M_1 \rangle)$	$M_2(\langle M_2 \rangle)$	$M_2(\langle M_3 \rangle)$	
M_3	$M_3(\langle M_1 \rangle)$	$M_3(\langle M_2 \rangle)$	$M_3(\langle M_3 \rangle)$	
...				

We know that every TM M (with tape alphabet $\{0, 1, \#\}$) corresponds to one row. The row entries in row M show the results of the computations of the M on input words which are codes of other such TMs. Now we construct a TM D with tape alphabet $\{0, 1, \#\}$. Assuming that M_H exists, we define D such that for any TM M, $D(\langle M \rangle)$ is different from $M(\langle M \rangle)$, that is, we make the computations $D(\langle M_1 \rangle)$, $D(\langle M_2 \rangle)$,... of D different from the diagonal

elements $M_1(<M_1>)$, $M_2(<M_2>)$, ... of our table. But D itself must correspond to some row of our table. We consider the diagonal element $D(<D>)$ in that row. By construction, we know that $D(<D>)$ is different from $D(<D>)$ – a crying-out-loud contradiction!

Proofs by diagonalization are a common strategy in the field of computability and complexity. The general schema of such proofs is the following

1. Goal: show that some machine M (or recursive language L) lacks some property P.
2. Assume that M (or L) has property P.
3. Set up a table like the above whose rows and columns are indexed by *all* machines (or recursive languages), and whose cells are functions of the row/column indices.
4. Using assumption 2., construct a machine M' or language L' such that on the diagonal of the table, the result of the row/column function is different from the original table. At this point, often some kind of "self-referentiality" is exploited.
5. Conclude that M' (or L') cannot be a row or column index. Contradiction, because these rows / columns range over *all* machines or languages.
6. Thus, the assumption 2. must be false.

From our undecidable language H we can derive other undecidable languages by a method called *reduction*. In fact, thousands of formal languages have been shown to be undecidable by reduction. A reduction proof of undecidability works as follows.

Given: a language $L_1 \subseteq \Sigma_1^*$ that one wants to show to be undecidable, and another language $L_2 \subseteq \Sigma_2^*$ which one already knows is undecidable (for instance, the halting language). Note that L and K may be defined over different alphabets.

Ansatz: find an effective mapping r: $\Sigma_2^* \to \Sigma_1^*$ such that $w \in L_2$ iff $r(w) \in L_1$. This mapping is called a *reduction* and one says, L_2 *is reduced to* L_1.

Why this works: if one has found such a reduction r, then L_1 cannot be decidable. Because if we assume that L_1 were decidable, L_2 could be decded too: in order to decide whether $w \in L_2$, map w on $r(w)$ and use the (assumed) decision method for L_1 to decide whether $r(w) \in L_1$. Since $w \in L_2$ iff $r(w) \in L_1$, this also would decide the question whether $w \in L_2$. But since L_2 is known to be undecidable, this cannot be possible, so the assumption that L_1 was decidable must be wrong.

An important aspect of this story is that r must be an "effective" mapping. When researchers of theoretical CS use that word, they imply that there is an algorithmic procedure (that is, ultimately, a TM) which computes r.

If one has established by reduction that L is undecidable, further languages can be shown to be undecidable by reducing L to them, etc. This gives a tree of undecidable languages, with the halting language at its root, and going down the tree one link from some language L to another language L' means that L can be reduced to L'. In fact, thousands of formal languages have been shown to be undecidable by reduction chains that ultimately lead back to H. These languages span all areas of mathematics and applied modeling techniques. Whenever in your professional carreer you meet a problem that you want to decide but can't find a simple decision method for, it is a good idea to consult the literature of the respective field where this problem arises and see whether you can find a problem that is similar to yours and which already has been shown to be undecidable.

Here is a collection of languages that are close relatives of H which can be shown to undecidable by reducing H to these languages in fairly simple ways:

Proposition 6.3. The following languages are not recursive:

$H_0 = \{\langle M\rangle \mid Code(\langle M\rangle)$ and M halts on the empty input$\}$
$H_1 = \{\langle M\rangle \mid Code(\langle M\rangle)$ and M halts on all inputs$\}$
$H_2 = \{\langle M\rangle;x \mid Code(\langle M\rangle)$ and $x \in \{0, 1, \#\}^*$ and there exists some $y \in \{0, 1, \#\}^*$ such that $M(x) = y\}$
$H_3 = \{\langle M\rangle;x \mid Code(\langle M\rangle)$ and $x \in \{0, 1, \#\}^*$ and the computation of M on input x uses all states of $M\}$

Here, $Code(\langle M\rangle)$ denotes the condition "$\langle M\rangle$ codes a TM with tape alphabet $\{0, 1, \#\}$".

A host of other languages of a similar flavour can be shown not to be recursive. We prove undecidability of the first two and the last language by reducing HALTING to those problems, to illustrate the pattern of a reduction argument.

Undecidability of H_0: For any word $\langle N\rangle;x$ with $Code(\langle N\rangle)$ and $x \in \{0, 1, \#\}^*$ we can effectively construct a TM $K_{\langle N\rangle;x}$ with tape alphabet $\{0, 1, \#\}$, which started on empty input first prints x on one of its tapes and then behaves as N on input x. Consider the language

$L = \{\langle K_{\langle N\rangle;x}\rangle \mid \langle K_{\langle N\rangle;x}\rangle$ is a code of $K_{\langle N\rangle;x}$ and $K_{\langle N\rangle;x}(\varepsilon)$ halts$\}$.

It is clear that $\langle K_{\langle N\rangle;x}\rangle \in L$ iff $\langle N\rangle;x \in H$. The reduction r maps $\langle N\rangle;x$ on $\langle K_{\langle N\rangle;x}\rangle$.

Therefore H_0 is not decidable. □

Undecidability of H_1: Take any word $\langle N\rangle;x$ with $Code(\langle N\rangle)$ and $x \in \{0, 1, \#\}^*$. We can effectively construct a TM $K_{\langle N\rangle;x}$ with tape alphabet $\{0, 1, \#\}$ which, for all inputs $y \in \{0, 1, \#\}^*$, yields the following result:

$$K_{\langle N\rangle;x}(y) \quad = \quad \text{if } x = y \text{ then } N(x) \text{ else "yes".}$$

Obviously, $K_{\langle N\rangle;x}$ halts on all inputs iff $\langle N\rangle;x \in H$. The reduction map here maps $\langle N\rangle;x$ on $\langle K_{\langle N\rangle;x}\rangle$. □

Undecidability of H_3: same pattern as in the two previous examples. For a TM N with tape alphabet $\{0, 1, \#\}$, we construct a TM K_N with tape alphabet $\{0, 1, \#\}$ which on input x starts to simulate $N(x)$. If this simulation reaches a halting configuration of N, then K_N enters a subroutine which takes K_N through all its states (that can be done, e.g., by printing a single 1 on a special tape of K_N when some halting condition of the simulated N is reached; if this 1 is present, K_N cycles through its states). If the simulation does not enter the accepting state of N, then K_N does not use all of its states (because it does not use its own halting state). Therefore we have:

$K_N(x)$ uses all states iff $\langle N\rangle;x \in H$,

that is, we have devised a reduction r which maps $\langle N\rangle;x$ on $\langle K_N\rangle;x$ □

With a similar reduction to H we can prove an undecidability theorem of surprising generality. In essence, it states that for any non-trivial property C of recursively enumerable languages over some alphabet Σ, the language

$L_C = \{<M> \mid Code(<M>)$ and M is a TM that accepts a language $L \subseteq \Sigma^*$ which has property $C\}$

is undecidable. In other words, for no nontrivial language property C there is a general method to effectively check whether the language $L(M)$ accepted by some TM M has property C.

Proposition 6.4 (Rice's Theorem): Suppose that C is a nontrivial subset of the class of recursively enumerable languages over some alphabet Σ. ("Nontrivial" means: not empty and not the class of recursively enumerable languages itself). Then L_C is undecidable.

Proof. We first consider the case that the empty language $L_\varnothing$ is not in C. Because C is nonempty, there must be some nonempty language $L \in C$ which is accepted by some TM M_L.

Remember that H is recursively enumerable (Prop. 6.1), that is, there is a TM K which accepts all inputs of the form $<M>;x$ where $M(x) \neq \nearrow$. We don't know what may happen when K is given some input z not of this form, but we can modify K such that it continues to work like before on inputs of the form $<M>;x$, and on inputs z of a different form either runs forever or accepts. We assume that K has been modified this way, that is, K is a TM that accepts a language over $\{0, 1, \#, ;\}$. Now, for every $z \in \Sigma^* = \{0, 1, \#, ;\}^*$, a TM M_z can be effectively constructed whose accepted language is either L or $L_\varnothing$. On input $y \in \Sigma^*$, M_z first simulates K on z. If $K(z)$ = "yes", then M_z, instead of halting, goes on to simulate M_L on input y: it either halts and accepts, or never halts, according to the behavior of M_L on input y. And of course, if $K(z) = \nearrow$, then $M_z(y) = \nearrow$ as well. Schematically, M_z is the machine

$M_z(y)$ = if $K(z)$ = "yes" then $M_L(y)$ else $\nearrow$.

Now we claim that $L(M_z) \in C$ iff $z \in L(K)$. To show this claim, first assume $z \in L(K)$, that is, $K(z)$ = "yes". Then by construction of M_z, M_z behaves like M_L, that is, M_z accepts L which is in C. Now assume $z \notin L(K)$, that is, $K(z) = \nearrow$. In this case M_z never halts, that is, M_z accepts $L_\varnothing$, known not to be C.

But the claim actually is a reduction of the halting problem to the problem of deciding L_C. Because if we can decide L_C, we can decide for any z whether $L(M_z) \in C$, which by the claim entails that we can decide for any z whether $z \in L(K)$, which by construction of K entails that we can decide H, which is impossible.

This completes the proof of the claim for the case that the empty language $L_\varnothing$ is not in C. In the other case ($L_\varnothing$ in C) repeat the proof for the complement language L_Cc = $\{<M> \mid M$ accepts a language *without* property $C\}$ and note that L_Cc is undecidable iff L_C is undecidable.

6.3 Busy Beavers revisited: $\sigma(n)$ is not recursive

We have derived several undecidability results in this section, starting by a diagonalization argument which gave us undecidability of the halting problem, and then creating from this

result numerous spinoffs by reduction. You might argue that the halting problem is a rather artificial one, and the self-referential nature of the diagonalization argument is nothing that you would encounter in real-life problems. But fact, most of the undecidable problems found in the literature ultimately rest on a diagonalization argument (because they are derived from such a problem by reduction). It is rare that one encounters an undecidability proof which is not carried out by diagonalization or reduction.

A charming little example of such an undecidable problem comes from the world of busy beavers. Namely, the Rado function $\sigma(n)$, which gives the maximal number of 1's that an n-state 1-tape TM can leave on its tape before stopping, is not a recursive function. (This claim can be transformed into an equivalent language decision claim. Consider the language L_{Rado} = {<n>;<m> | <n> and <m> are binary representations of integers n and m, and $m < \sigma(n)$}. It is clear that L_{Rado} is decidable iff $\sigma(n)$ is recursive.)

Proposition 6.5. $\sigma(n)$ is not recursive.

Proof. (Taken from Arno Schwarz's [German] paper on the limits of computability[31] http://www.zum.de/Faecher/Inf/Saar/material/grenzen/grenzen1.htm). We start with the observation that $\sigma(n)$ is a strictly growing function (argument: from an n-state busy beaver BB_n writing $\sigma(n)$ construct an $(n+1)$-state beaver that writes one more 1 than BB_n). Now assume there is a TM B that computes $\sigma(n)$. We may assume that B reads in n in binary representation and outputs $\sigma(n)$ in unary representation as a sequence of $\sigma(n)$ many 1's. Let B have m states. Let $n = 2^k-1$ for some natural number k. Let A_n be a 1-tape TM which started on the empty tape writes n in binary representation on its tape – that is, a sequence of k 1's. This can obviously be achieved with an A_n that has $k = \log(n+1)$ states. Note that A_n is a beaver TM.

If you couple A_n with B into a combined TM A_nB which first acts like A_n and then like B, and start A_nB on the empty tape, then A_nB writes exactly $\sigma(n)$ many 1's and stops. That is, AB is a beaver with $\log(n+1) + m$ states which writes as many 1's as the busy beaver with n states. But if n is sufficiently large, we have $\log(n+1) + m < n$. This means that the beaver A_nB which has fewer states than the n-state busy beaver writes as many ones as the latter. This contradicts the strict growth of $\sigma(n)$. □

So, the search for busy beavers will go on forever...

6.4 Undecidability of first-order logic

One of the most important implications of the undecidability of the halting problem is that first-order logic is undecidable, that is, there exists no algorithm that on input <Φ>; <φ> (where <Φ>, <φ> are suitably coded versions of a finite set of FOL formulae Φ and a single formula φ, respectively) can decide whether $\Phi \models \varphi$. This was first proven by A. Church[32] in 1936, and in his honour the undecidability of FOL is sometimes referred to as Church's theorem. Church's proof rests on the λ-calculus which he invented; only a few months later, Alan Turing, a student of Church at that time, published a proof of the undecidability of FOL

[31] http://www.saarland.de/dokumente/thema_bildung/Prinzipielle_Grenzen_der_Berechenbarkeit.pdf, lcoal copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/Grenzen.pdf (in German)

[32] http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Church.html

which rested on Turing machines and the halting problem. In modern textbooks, proofs of the undecidability of FOL usually work by reduction from the halting problem.

Proposition 6.6.[33] Let $\Sigma = \{\triangleright, \sqcup, 0, 1, \#\}$ be our standard Turing tape alphabet, and $S = \{c, f_{\triangleright}, f_{\sqcup}, f_{\#}, f_0, f_1, R\}$ be a signature, where c is a constant symbol, f_i are unary function symbols for every $i \in \Sigma$, and R is a ternary predicate symbol (the reason for choosing such S will become clear in the proof below). Then, the language

$L_{\mathrm{FOL}} = \{\langle\varphi\rangle \in \{0,1,\#\}^* \mid \varphi$ is a formula of FOL with symbols from S, and $\models \varphi$, that is, φ is a tautology$\}$

is undecidable.

Here $\langle\varphi\rangle$ is a coding of φ as a word in the alphabet $\{0,1,\#\}$ according to some (arbitrary) coding scheme. Notice that if L_{FOL} is undecidable, then the more general question whether $\Phi \models \varphi$ holds for any Φ, φ (over any signature) is *a fortiori* undecidable.

Proof. (By reduction from the HALTING problem). We will effectively assign to every input $\langle M\rangle;x$ to a universal TM a FOL S-sentence $\varphi_{M,x}$ such that $\models \varphi_{M,x}$ iff $M(x) \neq \nearrow$. Then, if L_{FOL} were decidable, H would also be decidable, contradiction. The main work to be done is the construction of $\varphi_{M,x}$.

Let M be a single-tape Turing machine with tape alphabet $\Sigma = \{\triangleright, \sqcup, 0, 1, \#\}$ which has states $q_0, \ldots, q_n$, where for convenience we assume q_0 to be the initial state and q_n to be the halting state. Recall that the configurations of M are specified by 3-tuples (q, w, u), where q denotes the current state, and w and u are words from Σ^* (with the head placed at the end of w). Furthermore, the transition function δ can be specified as a function on possible configurations obeying certain rules.

We assign to the pair (M, x) an S-structure $\mathcal{M} = (\Sigma^*, c^{\mathcal{M}}, f_{\triangleright}^{\mathcal{M}}, \ldots, f_1^{\mathcal{M}}, R^{\mathcal{M}})$, in which we can model the run $M(x)$, in the following way:

Let $c^{\mathcal{M}} = \varepsilon$ (where ε denotes the empty word), $f_i^{\mathcal{M}}(v) = iv$ for any word $v \in \Sigma^*$ and $i \in \Sigma$ and

$$R^{\mathcal{M}} = \{(\#^k, w, u) \mid (q_k, w^{\mathrm{rev}}, u) \textit{ is a configuration reached in the run } M(x)\},$$

where w^{rev} denotes the reverse of w and $\#^k$ denotes the word consisting of k times the symbol '#'. To simplify our notation we write $\underline{k} := f_{\#}^k(c)$ for numbers $k \in \mathbb{N}$ and $\underline{x} := f_{x_1} \circ \quad \circ f_{x_m}(c)$ for words $x = x_1 \ldots x_m \in \Sigma^*$ (notice that $\underline{k}$ and $\underline{x}$ are S-terms).

We then have $\mathcal{M} \models R(c, f_{\triangleright}c, \underline{x})$, since the initial configuration is surely reachable, and furthermore $\mathcal{M} \models \exists u\, \exists v : R(\underline{n}, u, v)$ iff $M(x) \neq \nearrow$.

Next, we want to code the state transition rules that determine the workings of M in terms of S-formulas (compare Def. 3.3):

---

[33] The following proof was adapted by Michael Thon from lecture notes of Robert van Glabbeek, http://kilby.stanford.edu/~rvg/154/handouts/fol.html

- If M has a transition rule $\delta(q, a) = (q', b, \leftarrow)$, then
  $\mathcal{M} \models \forall u\ \forall v : R(\underline{q}, f_a u, v) \rightarrow R(\underline{q'}, u, f_b v)$.

- If M has a transition rule $\delta(q, a) = (q', b, -)$, then
  $\mathcal{M} \models \forall u\ \forall v : R(\underline{q}, f_a u, v) \rightarrow R(\underline{q'}, f_b u, v)$.

- If M has a transition rule $\delta(q, a) = (q', b, \rightarrow)$, then
  $\mathcal{M} \models \forall u\ \forall v : R(\underline{q}, f_a u, f_i v) \rightarrow R(\underline{q'}, f_i f_b u, v)$ for all $i \in \Sigma$ and
  $\mathcal{M} \models \forall u\ \forall v : R(\underline{q}, f_a u, c) \rightarrow R(\underline{q'}, f_\sqcup f_b u, c)$.

Let φ_M be the conjunction of all formulas occurring under the three cases above, i.e. the conjunction of all $\forall u\ \forall v : R(\underline{q}, f_a u, v) \rightarrow R(\underline{q'}, u, f_b v)$ etc. As M has only finitely many transition rules, this conjunction φ_M is finite as well, and thus a formula of first-order logic. It codes which next configuration is obtained from a previous one under the operations of δ. Now consider the formula

$$\varphi_{M,x} = R(c, f_\triangleright c, \underline{x}) \wedge \varphi_M \rightarrow \exists u\ \exists v : R(\underline{n}, u, v).$$

Then it holds that $\models \varphi_{M,x}$ iff $M(x) \neq \nearrow$. To see this, assume first $\models \varphi_{M,x}$. Since $\varphi_{M,x}$ is a tautology, we have $\mathcal{M} \models \varphi_{M,x}$, which implies $\mathcal{M} \models \exists u\ \exists v : R(\underline{n}, u, v)$ because $\mathcal{M} \models R(c, f_\triangleright c, \underline{x}) \wedge \varphi_M$ by the construction of M and φ_M . But we have already seen that $\mathcal{M} \models \exists u\ \exists v : R(\underline{n}, u, v)$ means that $M(x) \neq \nearrow$.

Conversely, assuming $M(x) \neq \nearrow$, then the run of M on input x must be leading to a configuration (q_n, w, u) for some $w, u \in \Sigma^*$ after k steps. Let us denote the configurations of this run as

$$(q_0, x, \varepsilon) =: (q^0, u^0, v^0) \overset{M}{\rightarrow} (q^1, u^1, v^1) \overset{M}{\rightarrow} (q^2, u^2, v^2) \overset{M}{\rightarrow} \ldots \overset{M}{\rightarrow} (q^k, u^k, v^k) = (q_n, u, v)$$

We have to check that the formula $\varphi_{M,x}$ is provable, i.e. $\vdash \varphi_{M,x}$, and therefore $\models \varphi_{M,x}$. The logical derivation simply repeats the run of the TM. Concretely, deriving $\vdash \varphi_{M,x}$ with the sequent calculus (or any other FOL calculus) is equivalent to deriving

$$\vdash\ R(c, f_\triangleright c, \underline{x}) \wedge \varphi_M \rightarrow \exists u\ \exists v : R(\underline{n}, u, v),$$

which in turn is equivalent to deriving

$$R(c, f_\triangleright c, \underline{x}) \wedge \varphi_M \vdash \exists u\ \exists v : R(\underline{n}, u, v).$$

The obvious way to construct a derivation is the following (several substeps omitted):

1. $R(c, f_\triangleright c, \underline{x}) \wedge \varphi_M \vdash R(\underline{q}^1, \underline{u}^1, \underline{v}^1)$; use the appropriate rule coded in φ_M
2. $R(\underline{q}^1, \underline{u}^1, \underline{v}^1) \wedge \varphi_M \vdash R(\underline{q}^2, \underline{u}^2, \underline{v}^2)$; (same)

   ...

k. $R(\underline{q}^{k-1}, \underline{u}^{k-1}, \underline{v}^{k-1}) \wedge \varphi_M \vdash R(\underline{n}, u, v)$

k+1. $R(c, f_\triangleright c, \underline{x}) \wedge \varphi_M \vdash R(\underline{n}, u, v)$; essentially a k fold application of chain rule

k+2. $R(c, f_\triangleright c, \underline{x}) \wedge \varphi_M \vdash \exists u\ \exists v\ R(\underline{n}, u, v)$

k+3. $\vdash R(c, f_{\triangleright}c, \underline{x}) \wedge \varphi_M \rightarrow \exists u\, \exists v\ \ R(\underline{n}, u, v)$

In sum, it holds that $\models \varphi_{M,x}$ iff $M(x) \neq \nearrow$. Since the latter is undecidable, $\models \varphi_{M,x}$ is also undecidable. □

Note. This is the shortest proof of undecidability of FOL that I am aware of. Each textbook will do it in its author's own special way – using different versions of TMs (or RAMs), different encodings of TM configurations in FOL formulae, etc. But the basic idea is always the same: code termination of a run of a TM in FOL.

7 Combinatorial algebras and Lambda calculus

TMs, RAMs and recursive functions are three ways to formalize the intuitive notion of "computable" or "computable function". TMs are "essentials-of-symbol-manipulation-oriented", RAMs are "machine-oriented", and recursive functions are "arithmetics-oriented". We will now introduce yet another way of approaching "computable functions", which one might declare as "functional" or "Herbert's favourite". Bear with me if I spend some time on the beauties of: *combinatorial algebras* and the *Lambda calculus* (let's call it CALC).

Warning: CALC is deep and strange. Be prepared that you have to twist your brain. Besides being deep, beautiful, and strange, CALC is also very practical: it lies at the heart of functional programming languages, which we will introduce after the maths part. The most venerable of these languages is LISP, *the* language of artificial intelligence programming. You know Standard ML from the Gen CS lectures: that is another functional programming language.

CALC was developed in the 1920's and 1930's by Moses Schönfinkel (first invented combinators 1924), Alonso Church[34] (1903-1995, he invented the lambda calculus, gave name to half of the Church-Turing hypothesis, and founded the Journal of Symbolic Logics), and by Haskell B. Curry[35] (1900-1982, invented combinators independently of Schönfinkel, worked out combinatorial logics toward a foundation for mathematics, and posthumously lent his first name to a functional programming language).

A very recommendable introductory text on the lambda calculus[36] as a foundation for functional programming was written by L. C. Paulson. It is more advanced than our treatment but very accessible. Another beautiful text on CALC is the third part of the book "Foundations of Mathematics: Questions of Analysis, Geometry and Algebra" by E. Engeler (Springer-Verlag 1993). Some of the following is adapted from these two sources.

7.1 Computation in the spirit of CALC: evaluation of functional expressions

If you are doing practical maths, say, "calculating" an everyday formulae like $\sin(\pi + 1)$, what do you do? Well, you first add 1 to π and then take the sine of what you have gotten so far. You *evaluate* functions on their arguments.

The CALC way of thinking about maths and computation starts from this everyday observation and makes a dramatic claim: computation (and mathematics) is, basically, *nothing but* function evaluation. The single basic concept of CALC is the notion of function application. This is an *action-centered perspective*: the CALC view of the world (and maths) is dynamical and centered on change. This is in stark contrast to the perspective of set theory, whose *object-centered perspective* traces everything back to the static concept of a set. Since computation is an activity, an action concept seems a good starting point for a theory of computation.

[34] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Church.html
[35] http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Curry.html
[36] originally from http://www.cl.cam.ac.uk/users/lcp/papers/Notes/Founds-FP.pdf local copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/Founds-FP.pdf

We start our expedition into the exotic (you will soon agree I am sure!) and beautiful (you will soon agree I hope!) world of CALC by fixing a few notational conventions.

A basic convention in the world of CALC is to use prefix notation. Thus, $\sin^n(y + \pi)$ becomes $\hat{}n(\sin(+(y, \pi)))$. See a little essay[37] by W. M. Leach for historical background on prefix ("Polish") and postfix ("reverse Polish") notation – it once played a role as the notation in HP's first pocket calculators.

Another convention is to use only unary functions. Schönfinkel discovered (and Curry re-discovered) that unary functions always suffice. The trick to convert expressions containing multiple-argument functions into expressions made up only of unary functions is referred to as "Currying". Currying the binary function + in the expression $+ (y, \pi)$ would work by first introducing a new unary functional object $(+ y)$ ("add y") and then apply this to π, changing $+ (y, \pi)$ into $(+ y)$ (π). Note that the functional object $(+y)$ contains itself a variable; it depends on what value is assigned to y. Thus, Currying is just the trick to turn arguments of a multi-argument function into parameters of unary functions. Our example expression $\sin^n(y + \pi)$ would become $((\hat{}n)(\sin))((+y)\ (\pi))$. In this formula we find another functional object that depends on a variable, namely, $\hat{}n$. This obviously just looks like an instance of a syntactical rewriting, but we will see that it has mind-boggling consequences.

Another convention is to agree on left-associative binding for saving brackets. That is, a functional expression of the form $fxyz$ would be read as $((fx)y)z$. Using this convention, $((\hat{}n)(\sin))((+y)\ (\pi))$ turns into $(\hat{}n\ \sin)(+y\ \pi)$.

Generally, Currying an m-ary function expression of the form $f(x_1, \ldots, x_m)$ is done by successively splitting away arguments from the right:

$$f(x_1,\ldots,x_m) \to (f'(x_1,\ldots,x_{m-1}))(x_m) \to ((f''(x_1,\ldots,x_{m-2}))(x_{m-1}))(x_m) \to \ldots$$
$$\ldots \to ((\ldots((f^{(m-1)}(x_1))(x_2))\ldots(x_{m-2}))(x_{m-1}))(x_m),$$

which could be written without brackets, observing left-associative binding, as

$$f^{(m-1)}x_1x_2\ldots x_m.$$

One characteristic feature of CALC is implicit in Currying: the evaluation of a functional expression like $fx_1x_2\ldots x_m$ may yield a function as a result. For instance, $f^{(m-1)}x_1$ is a function that can be applied to x_2; what you then get is a function that can be applied to x_3, etc. In fact, CALC dispenses with the type distinction between functions and arguments; it would be in the spirit of CALC to write $f_0f_1 \ldots f_m$ or $x_0x_1\ldots x_m$. (Type distinctions can later be re-introduced).

This deserves a comment. In the world view of basic CALC, the *only* objects in computations and in mathematics are functions: you apply functions to functions and get functions (just as in the world of set theory, the *only* objects from which all other mathematical objects are made are sets – you put together sets and get new sets, and nothing else). Notice that unary functions, which always evaluate to the same result regardless of the argument, can be considered as constants.

[37] originally taken from http://users.ece.gatech.edu/~mleach/revpol/, local copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/ReversePolishNotation.html

Expressions like $f^{(m-1)}x_1x_2...x_m$ or $(\hat{}n \sin)(+y\ \pi)$ are the subject matter of CALC approaches to computation. We call such basic expressions *terms*. Given some fixed finite set of constant symbols (for instance 0 [a unary function that always returns 0] and +1 [the unary function of adding 1]) and an infinite set of variables (we will often use x, y, z, f, g, ..., possibly with indices), terms are words over the symbol set {(,)} ∪ <set of constants> ∪ <set of variables>, which can be generated with this grammar:

Definition 7.1 (grammar of "pure combinatorial algebra (CA)" terms)

```
<term> ::= <variable> | <constant> | (<term> <term>)
```

Note that the rule `<term> ::= (<term> <term>)` corresponds to writing down a function application. Bracket-saving conventions allow us to drop brackets when they conform to left-associative binding, e.g. we may write $xy(uv)$ instead of $((x\ y)(u\ v))$. Constants are usually written by uppercase symbols, e.g. X, Y, S, C.

Consider some term t', for instance

$$t' = z(x(zu(Avu))).$$

This term is combined from variables u, v, x, z and a constant A. Instead of variables, other terms could have been combined into the form of t', for instance if you replace u by $W(r\ W)$, then

$$t'' = z(x(z\ (W\ (r\ W))\ (Av\ (W(r\ W)))))$$

would have, at some level of description, the same structure as t'. Combining the variables u, v, x, z into t' is itself a mathematical action – an application of the *term-building function* "`combine terms` t_1, t_2, t_3, t_4 `into the structure of` $t = t_4\ (t_3\ (t_4\ t_1\ (A\ t_2\ t_1)))$" applied to the arguments u, v, x, z. Such term-building functions that assemble terms of a given form from arguments are called *combinators*, and interpreting term-building operations as functions is called *functional abstraction*. CALC is very much concerned with combinators. In fact, "pure" combinatorial algebra considers a mathematical universe which is populated *only* by combinators (dual to set theory which interprets all mathematical objects as sets).

It is easy to define a combinator, giving it a name at the same time. All that is needed is to write down, on the left-hand side of a combinator specification, a constant symbol that will serve as the name of the combinator to be defined, followed by its arguments, and on the r.h.s. the term whose structure is constructed by the combinator. For example,

$$Cuvxz := z(x(zu(Avu)))$$

declares C as the (name of the) constructor of terms of the form $z(x(zu(Avu)))$. Remember that $Cuvxz$, by the convention of left-associativity, means $(((Cu)v)x)z$.

We write $t_n[x_1, \ldots, x_n]$ to indicate that a term made from variables and constants contains at most variables from the set $\{ x_1, \ldots, x_n \}$. An n-ary combinator A, then, is a function associated with some term $t_n[x_1, \ldots, x_n]$ such that when A is applied to n terms $s_1,..., s_n$, it yields the term $As_1...s_n = t_n[s_1 / x_1, \ldots, s_n / x_n]$, that is the term obtained from $t_n[x_1, \ldots, x_n]$ by

replacing every occurrence of x_i by the term s_i ($i = 1,..., n$). We say that *A represents* the term t_n.

We will stick to pure CA for a while and introduce more familiar objects (like numbers or Boolean functions) into our world later. The world of pure CA certainly needs some time to get used to. In pure CA, if you use a constant symbol within a term, it must refer to a combinator! And variables, of course, can only be replaced by other terms – and ultimately, by combinators.

Let's play around with terms of pure CA a little to get a feeling for them. We define three (famous) combinators *S*, *K*, *I* by the following equations:

Definition 7.2 (Combinators *S*, *K*, *I*)

for all terms x, y, z: $Sxyz = xz(yz)$
for all terms x, y: $Kxy = x$
for all terms x: $Ix = x$

Note that *S* is – like all combinators considered in CA – a unary operator, so it would also be allowed to write down the term *Sx*. A fully written-out version of *Sxyz* would spell $((Sx)y)z)$ and would be read as "apply *S* to *x*, then apply the obtained combinator *Sx* to *y*, then apply that obtained combinator *Sxy* obtained to *z*. The equation $Sxyz = xz(yz)$ does not imply that *S* is a ternary operator, but rather makes a statement about an iterated application of first *S* to *x*, then of *Sx* to *y*, then of *Sxy* to *z*.

We can use equations like $Sxyz = xz(yz)$ to *reduce* a CA term. For instance, consider the term $SK(KSS)t$. Using the definitions of *S* and *K*, we can reduce this term as follows:

(1)	$SK(KSS)t$	$\rightarrow Kt((KSS)t)$	[definition of *S*]
		$\rightarrow t$	[apply leftmost *K* according to def. of *K*]

We find that $SK(KSS)$ turns out to be the identity combinator *I*, so we could have defined *I* in terms of *S* and *K* by putting $I = SK(KSS)$. (There are simpler ways to define *I* in terms of *S* and *K*.) We say that $SK(KSS)t$ can be *reduced* to *t.*

A precise definition of when some term *t* can be derived from some term *s* via a sequence of reductions comes in the form of a *calculus*. We already met such a calculus in first-order logic. It came in the form of a sequent calculus, in which we could write down lines ("sequents") of the kind $\Phi \; \varphi$ according to certain rules, which either allowed us to write down certain such sequents a priori, or allowed us to write down a sequent given that certain other sequents had already been established. A sequence of sequents produced according to the rules can then be considered a proof of the statement of the last sequent written.

In a similar spirit, we will now describe a calculus for specifying derivations of CS terms. Like in the FOL sequent calculus, this calculus allows one to write down a sequence of lines, where each line (i) is either justified by some starting conditions that allow us to write down the line per se, or (ii) is justified by using lines that were previously written down. Where a FOL sequent $\Phi \; \varphi$ should be understood as "there exists a derivation $\vdash$ for φ from Φ, that is, $\Phi \vdash \varphi$ ", the lines in our reduction calculus for CA have the format $t_1 \rightarrow t_2$, with t_1, t_2 being CA terms, and should be understood as "there exists a sequence of reductions that carries t_1 to t_2".

Definition 7.3: (reduction calculus for CA terms with constants K and S):

Axioms: you may immediately write down the following lines:

(A1)	$t \to t$	for single symbol terms (variables or constants)
(A2)	$S\,t_1\,t_2\,t_3 \to t_1\,t_3\,(t_2\,t_3)$	for any terms t_i
(A3)	$K\,t_1\,t_2 \to t_1$	for any terms t_i

Inference rules: if lines corresponding to the "enumerators" in the following schemas have already been obtained, you may write down the lines in the "denominators":

$$(I1)\quad \frac{t \to s}{rt \to rs} \qquad \textit{for any term } r$$

$$(I2)\quad \frac{t \to s}{tr \to sr} \qquad \textit{for any term } r$$

$$(I3)\quad \frac{\begin{array}{c} t \to r \\ r \to s \end{array}}{t \to s}$$

A sequence $t \to s$ is understood as "s can be obtained from t by a reduction sequence of any finite number of reduction steps (including zero steps)". You can use this calculus to find out whether some term s is the product of an (iterated) reduction of some term t, if you find a "proof" for $t \to s$. A proof is a sequence of formulae $u \to v$, where each formula can be either written down immediately, justified by one of the three axioms, or can be derived from previous lines by one of the inference rules.

Side remark: it is undecidable whether for two CA terms t, s with constants K and S it holds that $t \to s$. The proof is not simple and is based on the fact that every recursive function can be represented by a combinator term (see the Engeler book for more).

Our example reduction $SK(KSS)t \to t$ can be formally shown to be correct w.r.t. this calculus by the proof

(2)

1. $SK(KSS)t \to Kt((KSS)t)$		(apply A2)
2. $Kt((KSS)t) \to t$		(apply A3)
3. $SK(KSS)t \to t$		(apply I3 on 1. and 2.)

Note that deriving a formal proof of $t \to s$ is not the same thing as invoking an evaluation algorithm. Our calculus does not tell you how, exactly, you should proceed in order to reduce a given starting term t. But if you wish to delegate this task to a computer, you have to give a deterministic, completely specified algorithm. The question of how one can set up a deterministic algorithm that carries out reduction sequences brings us to the important theme of *evaluation strategies*.

To start our discussion of evaluation strategies, note that our derivation (1) of $SK(KSS)t \to t$ proceeded by evaluating combinators "from the outside" in that we did not start by evaluating the inner bracket level (KSS), but passed this term (KSS) to the leftmost S in unevaluated form. Evaluating terms "from the outside" of the bracket structure is called *normal-order* evaluation. In normal order evaluation, the term tree of the expression is searched from the root downwards until an evaluable operator is found, which is then evaluated; the process is then repeated. The derivation (1) was a normal-order evaluation. Normal-order evaluation is also called *call-by-name* evaluation (because we pass the arguments to the function as they are written in the formula, i.e. as they are "named").

There exists no law about evaluation order, so we could also have evaluated $SK(KSS)t$ "from the inside", as follows:

$SK(KSS)t$	$\to SKSt$	[apply def. of K to (KSS)]
	$\to Kt(St)$	[apply definition of S]

At this point we cannot go further by evaluation from the inside, because the inner term (St) cannot be further processed. By switching to call-by-name evaluation we would get t. Proceeding from the inside to the outside of bracket levels is called *greedy* evaluation, or *call-by-value*, or *strict* evaluation. "Mixed" evaluation strategies are possible as well, for instance *call-by-need* (aka *lazy* evaluation). Call-by-need evaluation follows the strategy "prefer call-by-name evaluation but switch to call-by-value opportunistically whenever this leads to shorter expressions".

A little warning is in place here: the usage of "call-by-value", "call-by-name", etc., is not clearly defined. Most authors introduce these terms in a hand-waving way (just as we did here). But you run into challenging subtelties if you really get concrete. For instance, consider the combinator D defined by $Dx = xx$. If you evaluate the term $Kx(DD)$ by call-by-value, you will immediately run into the loop $Kx(DD) \to Kx(DD) \to Kx(DD) \to ...$ But now replace D by SII. Observing that $D = SII$ in the sense that $SIIx = xx$, you would expect that $Kx(SII\ (SII)) \to Kx(SII\ (SII)) \to Kx(SII\ (SII))$ just like before. But actually, if you strictly adhere to innermost-bracket-level-first evaluation, you will find that the innermost SII cannot be evaluated, so you are immediately stuck instead of entering a loop. One way to dissolve this inconsistency would be to interpret "call-by-value" as "always evaluate the innermost bracket level which admits an evaluation". Making this precise is not quite trivial, by the way! The only detailed specification of various kinds of evaluation order which I have spotted in the literature is in an article[38] written by Peter Sestoft that explicitly deals with this topic. That article, however, deals with the more complicated lambda calculus formulae, a superset of combinator formulae. A very good introduction and overview on evaluation strategies can be found at http://en.wikipedia.org/wiki/Evaluation_strategy.

A term is said to be in *normal form* when it cannot be further evaluated, i.e. no evaluation rule is applicable. Our little example has shown us that the evaluation strategy (call-by-name, lazy, or strict, or yet others) does practically matter: Evaluations which terminate in a normal form can be obtained by some evaluation strategy but not by another. Can it happen that two different evaluation strategies lead to different normal forms? This would be disastrous, but it cannot happen, as stated in the fundamental Church-Rosser theorem:

[38] copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/2159_Sestoft01.pdf , originally fetched from http://www.dina.dk/~sestoft/papers/mfps2001-sestoft.pdf

Theorem 7.1 (Church-Rosser): If two evaluation strategies transform a term into normal form, then the two normal forms are identical.

A supplementary theorem says that if any strategy terminates, then so does call-by-name:

Proposition 7.2 (Standardization theorem): If a term can be evaluated to normal form by any evaluation strategy, then this can also be done by means of call-by-name evaluation.

(These theorems are usually stated for the lambda calculus, an extension of pure CA which we will learn about soon.) By the Church-Rosser theorem, it appears that one should always use call-by-name; however, call-by-name is in practice often very inefficient. If it terminates in a normal form, strict evaluation is usually faster (has shorter derivation sequences). Different functional programming languages have different default evaluation strategies, and generally evaluation strategies is a major topic in the design of programming languages.

We saw that the *I* combinator can be made up from *S* and *K*. In fact, *S* and *K* suffice for all purposes:

Proposition 7.3 (combinatorial completeness) Every combinator representing a term made from variables and symbols *S*, *K* can be constructed from *S* and *K*.

Note: we will prove a variant, *"Every combinator representing a term made from variables and symbols S, K, I can be constructed from S and K and I."* Because *I* can be expressed through *S* and *K*, this is equivalent. When applying the proposition, this version is often more convenient to use.

We write $t_n[x_1, \ldots, x_n]$ to indicate that a term made from variables and constants contains at most variables from the set $\{ x_1, \ldots, x_n \}$. An *n*-ary combinator *A*, then, is a function associated with some term $t_n[x_1, \ldots, x_n]$ such that when *A* is applied to *n* terms $s_1,..., s_n$, it yields the term $As_1...s_n = t_n[s_1 / x_1, \ldots, s_n /x_n]$, that is the term obtained from $t_n[x_1, \ldots, x_n]$ by replacing every occurrence of x_i by the term s_i ($i = 1,..., n$). We say that *A represents* the term t_n.

Proof. We first show an auxiliary statement. Claim: for every $n \geq 1$, for every term $t_n[x_1, \ldots, x_n]$ made from (possibly not all) variables $x_1, \ldots, x_n$ and constants *S*, *K*, *I* there exists a term $t_{n-1}[x_1, \ldots, x_{n-1}]$ such that $(t_{n-1}[x_1, \ldots, x_{n-1}])(x_n) = t_n[x_1, \ldots, x_n]$. Proof of claim: By induction over the composition structure of t_n. If t_n is non-composite and of the form $t_n = S$ or $t_n = K$ or $t_n = I$ or $t_n = x_i$, where $i \neq n$, we put $t_{n-1} = K\, t_n$. Then it holds that $t_{n-1}[x_1, \ldots, x_{n-1}]\, x_n = (K\, t_n\, [x_1, \ldots, x_n])\, x_n = t_n[x_1, \ldots, x_n]$. The remaining possibility for t_n to be non-composite is $t_n = x_n$. Then we put $t_{n-1} = I$ and again have $t_{n-1}[x_1, \ldots, x_{n-1}]\, x_n = t_n[x_1, \ldots, x_n]$. Now assume that t_n is composite, that is $t_n = t_n'\, t_n''$. By induction we have $t_n' = t_{n-1}'\, x_n$ and $t_n'' = t_{n-1}''\, x_n$. We put $t_{n-1}[x_1, \ldots, x_{n-1}] = S\, t_{n-1}'\, [x_1, \ldots, x_{n-1}]\, t_{n-1}''\, [x_1, \ldots, x_{n-1}]$ and verify

$$
\begin{aligned}
t_{n-1}[x_1,\ldots,x_{n-1}]x_n &= St'_{n-1}[x_1,\ldots,x_{n-1}]t''_{n-1}[x_1,\ldots,x_{n-1}]x_n \\
&= t'_{n-1}[x_1,\ldots,x_{n-1}]x_n(t''_{n-1}[x_1,\ldots,x_{n-1}]x_n) \\
&= t'_n[x_1,\ldots,x_n]t''_n[x_1,\ldots,x_n] \\
&= t_n[x_1,\ldots,x_n]
\end{aligned}
$$

This proves our claim (and, incidentally, explains the reason for *S*). Combinatorial completeness is proven by repeated application of this claim:

$$t_n[x_1,\ldots,x_n] = t_{n-1}[x_1,\ldots,x_{n-1}]x_n = \ldots = t_0[]x_1\ldots x_n\,.$$

Notice that $t_0[\,]$ contains no variables and hence must consist solely of *K*, *S*, and *I*. □

Example. Applying the above theorem in practice is quite difficult, in fact a torture for humans (one would rather write a computer program implementing the procedure than doing it by hand). The difficulty comes from the fact that we are confronted with two nested inductions. For a demonstration, consider the combinator *A* defined by $Axy = xx$. Formally the action of *A* is defined for two arguments *x,y*, that is, *A* is formally represented by a term $t_2[x,y] = xx$. Our task is to find a term $t_0[]$ made solely from *S* and *K* and *I*, such that $t_0[]xy$ evaluates to *xx*. In order not to become confused during the construction, we will strictly adhere to the terminology used in the proof – using lower indices t_i to denote the maximal number *i* of variables in *t*, and use primes ′ and double primes ″ to denote the two halves of a composite term.

We start with $t_2[x,y] = xx$. This is composite: $t_2[x,y] = t'_2[x,y]\,t''_2[x,y]$ with $t'_2[x,y] = t''_2[x,y] = x$. Let us consider $t'_2[x,y] = x$. This is non-composite and thus by the method of the proof of the theorem can be represented by $t'_2[x,y] = t'_1[x]\,y = (K\,x)y$. Similarly, $t''_2[x,y] = t''_1[x]\,y = (K\,x)y$. Turning back to the top level $t_2[x,y] = t'_2[x,y]\,t''_2[x,y]$, we have $t_1[x] = S\,t'_1[x]\;t''_1[x] = S\,(Kx)\,(Kx)$. That is, we have managed to transform $t_2[x,y]$ into $t_1[x]\,y$.

Now we have to transform $t_1[x]$ to $t_0[]$ such that $S\,(Kx)\,(Kx) = t_1[x] = t_0[]\,x$.
$S\,(Kx)\,(Kx)$ is composite: $t_1[x] = t'_1[x]\;t''_1[x] = (S\,(Kx))\,(Kx)$, that is, $t'_1[x] = S\,(Kx)$ and $t''_1[x] = Kx$. Our next step is to find $t'_0[]$ and $t''_0[]$ such that $t'_0[]\,x = t'_1[x] = S\,(Kx)$ and $t''_0[]\,x = t''_1[x] = Kx$.

The second is easier, so let us start with this. $t''_1[x] = Kx$ is composite and we have $t''_1[x] = t'^{(\prime)}_1[x]\,t'^{(\prime\prime)}_1[x]$ with $t'^{(\prime)}_1[x] = K$ and $t'^{(\prime\prime)}_1[x] = x$. Again by the method of the proof we obtain $t'^{(\prime)}_0[] = KK$ and $t'^{(\prime\prime)}_0[] = I$. Now we can turn the composite $t''_1[x]$ into $t''_0[] = S\,t'^{(\prime)}_0\,t'^{(\prime\prime)}_0 = S\,(KK)\,I$.

The more involved $t'_1[x] = S\,(Kx)$ comes next. It is again composite and we have $t^{(\prime)}_1[x] = S$ and $t^{(\prime\prime)}_1[x] = Kx$. Applying the proof method on the former we get $t^{(\prime)}_0[] = KS$. For the

latter, we re-use our finding from before and put $t''_0[] = S\,(KK)\,I$. Using the "*S*-trick", we conclude by $t'_0[] = S\,t'_0\,t''_0 = S\,(KS)\,(S\,(KK)\,I)$.

Now we may return to $t_1[x] = t'_1[x]\ \ t''_1[x]$ and apply the *S*-trick, which finally gives us $A = t_0[] = S\,t'_0[]\,t''_0[] = S\,(S\,(KS)\,(S\,(KK)\,I))\,(S\,(KK)\,I)$.

Checking: $S\,(S\,(KS)\,(S\,(KK)\,I))\,(S\,(KK)\,I)\,x\,y = (S\,(KS)\,(S\,(KK)\,I))\,x\,((S\,(KK)\,I)\,x)\,y =$
$= KSx\,((S\,(KK)\,I)\,x)\,(Kx)\,y = S\,((S\,(KK)\,I)\,x)\,(Kx)\,y = ((S\,(KK)\,I)\,x)y\,((Kx)\,y) =$
$= (KK\,x)\,(I\,x)\,y\,x = K\,x\,y\,x = xx$. Relief ! we did not err on our tangled path.

7.2 Lambda abstraction

Combinatorial completeness shows us that we could in principle define any combinator through *S* and *K*. However, this leads to quite complex formulas even for simple combinators. We would prefer to have a way to formally express something like "let *A* be the combinator which has the property that $Axy = yx$". The tricky part in this statement is "the combinator which has the property that $Axy = yx$". Why is this tricky? Well, how would you write down formally "the arithmetic operation which adds two numbers and squares the result"? The standard way of introducing functions in mathematics is to write something like

$f\colon \mathbb{N}^2 \to \mathbb{N}\ ,\ f(n, m) = (n + m)^2$

and in programming to define a function by something like

```
definefunction (type integer) (name f) (arguments x y) (x+y)^2
```

In any case, you introduce a function symbol, and whenever you use that function later in proofs or programs, you have to follow up a (mental or programmed) pointer to the definition. But when you are living in a world made up exclusively of functions (the world of CALC), this is cumbersome, because during computations often there will be functions created "on the fly" for local use, and you would have to leave the ongoing computation, create that function definition, and return with a pointer. In addition, you would have to bother (in programming) about mechanisms for the automated creation of unambiguous function identifiers. Fortunately, Church invented *lambda abstraction*, or *anonymous functions* (though at his time he surely did not have programming in his mind). A lambda expression is a way to write down a function (in CALC: a combinator) without naming it (and without combining it explicitly from *S* and *K*). A lambda expression for our $(n + m)^2$ example would look like this:

$\lambda nm.\ ((n + m)^2)$

In order to write down that this function is to be applied to two arguments, say arithmetic terms t_1 and t_2, you would write

$\lambda nm.\ ((n + m)^2)\ t_1\ t_2$

For example, you would have $\lambda nm.\ ((n + m)^2)\ (1+2)\ 1 = 16$, or $\lambda nm.\ ((n + m)^2)\ k\ 1 = (k + 1)^2$. Or in a combinator example, $\lambda uv.\ (vu)\ xy = yx$. We now make this formal and extend our grammar of pure combinators into the grammar of lambda calculus:

Definition 7.4 (syntax of λ-terms)

```
<term> ::= <variable> | <constant> | (<term> <term>) |
           (λ <variable> . <term>)
```

The terms derivable in this grammar are called λ-terms. Since combinators are unary operators, this grammar specifies only unary λ-terms. We will use shorthand notation to save brackets and nested lambdas, for instance we write $\lambda xy.\ t$ for $(\lambda x.\ (\lambda y.\ t))$, etc. *Note that this bracket-saving convention for nested lambdas relies on* ***right-associative*** *binding.* This is a powerful source of confusion since otherwise we retain the left-associative binding convention. The term t in $\lambda x.\ t$ can, but need not, contain the variable x.

To make this formal, in an expression $(\lambda x.\ t)$, t is called the *body* of the λ-term. Every occurrence of x in the body t is *bound* by the lambda abstraction. An occurrence of a variable is *free* if it is not bound by some enclosing λ-abstraction. For example, x occurs bound and y free and z both bound and free in $(\lambda xz.\ xz)\ zy$. Here is a formal specification of which variables are bound and which are free in a lambda term (following Paulson's lecture notes[39], section 1.2.):

Definition 7.5: (free and bound variables in lambda terms).

(a) The set $BV(t)$ of the bound variables in a lambda term t is defined by induction over the structure of lambda terms as follows:

$BV(x) = \varnothing$
$BV(C) = \varnothing$
$BV(t\ t') = BV(t) \cup BV(t')$
$BV(\lambda x\ t) = BV(t) \cup \{x\}$

(b) The set $FV(t)$ of the free variables in a lambda term t is defined by induction over the structure of lambda terms as follows:

$FV(x) = \{x\}$
$FV(C) = \varnothing$
$FV(t\ t') = FV(t) \cup FV(t')$
$FV(\lambda x\ t) = FV(t) \setminus \{x\}$

Notice that according to this definition, a variable x which is bound in t need not occur in the body of t. For instance, x is bound in $(\lambda x.\ Syy)$.

A λ-term without a free variable corresponds to a combinator, and vice versa (think about it). Examples: our I combinator could be expressed by the lambda expression $\lambda x.\ x$, the S

[39] Originally from http://www.cl.cam.ac.uk/users/lcp/papers/Notes/Founds-FP.pdf; copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/Founds-FP.pdf

combinator by $\lambda xyz.\ xz(yz)$. The λ-term $\lambda y.\ x$ contains a free variable and thus does not represent a combinator; instead, it represents a function which returns the variable x when applied to any argument. (Hey! isn't that weird? a function that returns a variable?? yes, it's weird – and smells as if it is strangely powerful, too...) Note that, due to cases with free variables, λ-terms are more expressive than combinators made from S and K: such combinators cannot "return" terms containing variables that are not supplied to the combinator as an argument.

When we apply a λ-term $(\lambda x.\ t)$ to another term s, what we actually do in terms of symbol manipulation is to substitute s for all free occurrences of x in t, and drop the λx-enclosure. For instance, the application $(\lambda x.\ (xKy))\ (\lambda z.\ Szzz)$ yields $(\lambda z.\ Szzz)Ky$. To make this precise, we define variable substitution as follows:

Definition 7.6: (variable substitution). $t[s/y]$, the result of substituting a term s for all free occurrences of y in t, is defined per induction over the structure of terms by

$$
\begin{aligned}
x[s/y] &\equiv \begin{cases} s & \text{if } x \equiv y \\ x & \text{otherwise} \end{cases} \\
C[s/y] &\equiv C \\
(\lambda x.t)[s/y] &\equiv \begin{cases} (\lambda x.t) & \text{if } x \equiv y \\ (\lambda x.t[s/y]) & \text{otherwise} \end{cases} \\
(t_1 t_2)[s/y] &\equiv (t_1[s/y]t_2[s/y])
\end{aligned}
$$

This definition is part of our meta-language about the lambda calculus, not a part of lambda calculus itself, which is why we write "≡" and not "=". "$x \equiv y$" should be understood as "the identifier x is identically the same symbol (or symbol sequence) as the identifier y". By contrast, when we write "$x = y$" we want to say: "the mathematical object denoted by x is the same mathematical object as denoted by y", which would be a statement within the theory of lambda calculus.

Substitution must not disturb variable binding. Consider the combinator $(\lambda x.\ (\lambda y.\ x))$. It should represent the function that, when applied to an argument s, returns the constant function $(\lambda y.\ s)$ [which, when applied to anything, returns s]. Unfortunately, this does not work with argument $s \equiv y$, because then we would get $(\lambda x.\ (\lambda y.\ x))\ y = (\lambda y.\ y) = I$. The argument y is "captured" by the bound of λy. To avoid such *variable capture* pitfalls we have to make sure that when we apply a lambda term t to an argument s, building ts, no variable which occurs bound within t is free in s. We can always rename the bound variables in t to make this condition true. For example, instead of $(\lambda x.\ (\lambda y.\ x))\ y$ we would use $(\lambda x.\ (\lambda z.\ x))\ y$. We will tacitly use such variable renamings whenever required; a compiler for a programming language based on λ–calculus would have to do this automatically. Renaming variables is sometimes called "α-reduction" in lambda calculus.

β-*reductions,* η-*reductions and* λ-*calculus proper*

The process of applying a λ-term to its arguments (= evaluation of a function) is called β-reduction (or β-conversion). In formal terms, a β-reduction leads from $(\lambda x.t)\ s$ to $t[s/x]$, written $(\lambda x.t)\ s \rightarrow_\beta t[s/x]$. When carrying out a β-reduction, one must make sure (by

appropriate variable renaming if required) that no variable which is free in s is bound in t, in order to avoid variable capture.

Example: $(\lambda z.zy)\lambda x.x \rightarrow_\beta (\lambda x.x)y \rightarrow_\beta y$.

If you apply a λ-term of the form $(\lambda x.\, fx)$ to any argument s, you get $(\lambda x.\, fx)\ s \rightarrow_\beta fs$. That is, $(\lambda x.\, fx)$ behaves exactly like f itself, and we may therefore reduce terms of the form $(\lambda x.\, fx)$ to f. This is called η-reduction: $(\lambda x.\, fx) \rightarrow_\eta f$.

β or η reductions can be applied to subterms, yielding overall correct reductions. We write $s \rightarrow t$ if t can be obtained from s by a single β or η reduction, possibly applied to a subterm of s. Similar to TMs, where we introduced the transitive closure $\overset{M*}{\rightarrow}$ of single-step state update rule applications, we write $t \rightarrow* s$ if s can be obtained from t through a finite number of single-step reductions $\rightarrow$.

When $t \rightarrow* s$, our insight into the intuitions behind β and η reductions would justify that we claim "s and t refer to the same mathematical object, they are *equal*" (just as we say, "2 is equal to (1+1)"). A bit more generally, we would also say that if s and t are equal, one is connected to the other by a sequence of reductions and/or *expansions* (where an expansion is just a reduction read backwards), like this:

$$s \rightarrow * u_1 * \leftarrow u_2 \rightarrow * u_3 * \leftarrow \cdots \rightarrow * u_{k-1} * \leftarrow u_k \rightarrow * t$$

To make our notion of equality formal, we introduce a symbol "=" to denote it, and a complete calculus which allows us to carry out formal proofs for term equality:

Definition 7.7: (calculus for equality of λ-terms, aka "λ-conversion calculus")

Axioms:

(A1) $s = s$ for any term s

(A2) $(\lambda x.t) = (\lambda y.t[y/x])$ if y is not free in t (variable renaming, or α - reduction)

(A3) $(\lambda x.t)\ s = t[s/x]$ if no free variable of s is bound in t (β – reduction)

(A4) $\lambda x.\, fx = f$ if x is not free in f (η-reduction)

Inference rules:

$$\text{(I1)}\ \frac{t = s}{s = t} \qquad \text{(I2)}\ \frac{s = u \quad u = t}{s = t}$$

$$\text{(I3)}\ \frac{t = s}{(\lambda x.t) = (\lambda x.s)} \qquad \text{(I4)}\ \frac{t = s}{(tu) = (su)} \qquad \text{(I5)}\ \frac{t = s}{(ut) = (us)}$$

This calculus is (like the simpler reduction calculus for CA we met before) a tool for *reasoning about* (equality of) λ-terms, not a machinery to actually *compute with* λ-terms. For example, the following is a proof for the equality claim $(\lambda uv.\, v)\ st = t$:

1. $(\lambda uv.\ v)\ s\ =\ (\lambda v.\ v)$ (A3)
2. $(\lambda v.\ v)\ t\ =\ t$ (A3)
3. $(\lambda uv.\ v)\ st\ =\ (\lambda v.\ v)\ t$ (I4) on 1.
4. $(\lambda uv.\ v)\ st\ =\ t$ (I2) on 2., 3.

7.3 Basic programming constructs in λ-calculus

If we want to do Boolean logic, construct standard data structures, or do arithmetics within λ-calculus, we have to define the respective basic mathematical objects (like truth values, lists, numbers and addition) as combinators – because combinators are the only objects we have at our disposal. This is similar to the re-construction of integers as sets (you may have learnt in the early math lectures that 0 is represented by the empty set {}, 1 by {{}}, 2 by {{{}},{}}, etc.) This involves some coding, i.e. we must agree on some conventions about what combinators will mean what mathematical object. The beauty of the λ-calculus resides to a great deal in the fact that codings have been found which, in a sense, are not arbitrary: they carry with them the "symbol manipulation operations" which are typically performed on the respective mathematical object.

Booleans

We start with the booleans, where this becomes particularly clear. The most important usage of truth values TRUE and FALSE in most programming languages is in IF clauses, which basically look like this:

IF "*this is true*" THEN "*do this*" ELSE "*do that*",

or if you simplify the syntax a bit,

IF "*this is true*" "*do this*" "*do that*".

The evaluation of such a program statement should result in "*do this*" if "*this is true*" evaluates to TRUE, and it should result in "*do that*" if "*this is true*" evaluates to FALSE. Written out, we want to have

IF TRUE "*do this*" "*do that*" → "do this",
IF FALSE "*do this*" "*do that*" → "do that".

Well, "*do this*" and "*do that*" refer to some mathematical action – clearly a combinator in the world of CALC. FALSE and TRUE also must be combinators – because in the CALC world there is nothing but. The same holds for IF. Considering this after a having emptied a glass of good wine, it becomes clear that all what we need to encode this piece of Boolean logic in the λ-calculus are three combinators **true**, **false**, **if** which satisfy

(*) **if true** $s\ t \to^* s$,
if false $s\ t \to^* t$

for all λ-terms s and t. The usual way to encode **true**, **false**, **if** such that this desired behavior is achieved is

true ≡ λ*xy*. *x* (= *K* !)
false ≡ λ*xy*. *y*
if ≡ λ*pxy*. *pxy*

It is easy to see that (*) is indeed obtained with this definition. For instance, let us check **if true** *s t* →* *s*. Inserting the definitions of **if** and **true**, we get

(λ*pxy*. *pxy*) (λ*xy*. *x*) *s t*

	→*	(λ*uvw*. *uvw*) (λ*xy*. *x*) *s t*	[variable renamings]
	→*	(λ*xy*. *x*) *s t*	[β-reductions]
	→	*s*	[β-reduction]

The operations of Boolean logics can be defined from here, for instance

not ≡ λ*p*. **if** *p* **false true**
and ≡ λ*pq*. **if** *p q* **false**
or ≡ λ*pq*. **if** *p* **true** *q*

In order to verify these, you have to check the truth table for the particular operator. For instance for **and** it should hold

and true true →* **true**
and false true →* **false**
and true false →* **false**
and false false →* **false**

This is easily verified.

Pairs and lists

What is the essence of *lists*? Strange question... from the Algorithms and Datatypes lecture, and maybe from exposure to object-oriented programming, you are familiar with the idea that in order to define a datatype or object class (such as *list*), and in order to make it useful, one usually defines operations for *constructing* and for *accessing* instances of the datatype. This is also the spirit how datatypes are designed in lambda calculus. We start with the simplest kind of lists, namely *pairs*. A *constructor* for pairs is a combinator **pair** which takes two arguments and binds them into a pair – that is, the lambda term (**pair** *s t*) should represent the ordered pair made from terms *s* and *t.* The core operations that one should be able to perform with pairs are – well, just to select the first element of a pair if one wishes, and to select the second if one wishes (think about it – that's what "pair-ness" is all about). So, in addition to the constructor **pair** which binds *s* and *t* into the pair (**pair** *s t*) we also want to have *selectors* **first** and **second**, such that for any lambda terms *s*, *t* we have

first (**pair** *s t*) →* *s*
second (**pair** *s t*) →* *t*

There are many ways to design lambda-terms **pair**, **first**, and **second** which work out like that, for instance

$$\textbf{pair} \equiv \lambda xyf.\, fxy$$
$$\textbf{first} \equiv \lambda p.\, p\ \textbf{true}$$
$$\textbf{second} \equiv \lambda p.\, p\ \textbf{false}$$

do the job (easy to check). Magic can be so simple.

It is sometimes convenient to write (s, t) for **pair** $s\ t$. Lists are construed by most authors as nested pairs, with some extra conventions. The basic idea is to conceive of a list $[x_1, ..., x_k]$ as the nested structure of pairs $(x_1, (x_2, (.... (x_{k-1}, (x_k, \textbf{nil})) ...)))$, which in the world of lambda calculus is also often written as $x_1 :: x_2 :: .... :: x_k :: \textbf{nil}$. **nil** is a special end-of-list marker which is also treated as the empty list, i.e. [] corresponds to **nil**. The central operation for constructing a list is to make a pair from the new head element of a list and the rest (tail) of the list. This operation is usually called "cons", and in λ-calculus it is concretely the λ-term

$$\textbf{cons} \equiv \lambda xy.\, (\textbf{pair}\ xy)$$

Example: the list $[x_1, x_2]$ corresponds to the λ-term $(x_1, (x_2, \textbf{nil}))$, which is obtained from reducing **cons** x_1 (**cons** x_2 **nil**). If you think about it, you will find that **cons** $\rightarrow^*$ **pair** by η-reductions, so the definition of **cons** above gives us nothing new. **cons** is just another name for **pair**, introduced as an extra because (i) the definition $\lambda xy.$ (**pair** xy) of **cons** makes it visible how **pair** is employed to *construct* a pair from arguments, and (ii) because **cons** has a place in history (and present time) as a fundamental construct in LISP. Some authors define **cons** $\equiv \lambda xyf.\, fxy$ and avoid the introduction of **pair**; other authors yet (such as Paulson) use a different definition for cons (and lists), namely, **cons** $\equiv \lambda xy.$ **pair false** (**pair** xy).

A way to define **nil** (derived from "**n**ot **i**n **l**ist") which is useful for later purposes is to put

$$\textbf{nil} \equiv \lambda x.\, \textbf{true}$$

nil is useful to determine whether the end of list has been reached in recursive definitions on lists (see below), together with a predicate **null** which checks whether a list is the empty list, that is, a combinator satisfying

$$\textbf{null nil} \rightarrow \textbf{true}$$
$$\textbf{null}\ (\textbf{cons}\ x\ y) \rightarrow \textbf{false}$$

One way to define **null** is via

$$\textbf{null} \equiv \lambda p.\, (p\ (\lambda xy.\, \textbf{false}).$$

The selectors head and tail, which pick the first element and the rest of a list, respectively, can be defined by

$$\textbf{head} \equiv \lambda x.\, (\textbf{first}\ x)$$
$$\textbf{tail} \equiv \lambda x.\, (\textbf{second}\ x)$$

You can easily check that **head** (**cons** *s t*) →* *s* and **tail** (**cons** *s t*) →* *t*. Again, **head** and **tail** η-reduce to **first** and **second**, thus (like **cons**) they are superfluous, but illuminating "syntactic sugar[40]" items.

This way of constructing lists directly models a frequent implementations of list data structures by "cons cells", where each element x_i of the list is implemented by a pair of pointers, the first of which points to x_i and the second to the next pair of pointers in line, as in Figure 7.1. Functional programming languages like Lisp or Scheme root in this way of handling lists (the name Lisp derives from "list processing").

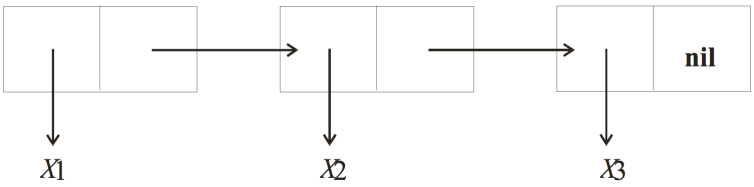


Figure 7.1: Implementing the list $[x_1, x_2, x_3]$ by "cons cells".

Elementary arithmetics

Church encoded the natural numbers in lambda terms now referred to as *Church numerals*:

$\underline{0} \equiv \lambda fx.\ x$
$\underline{1} \equiv \lambda fx.\ fx$
$\underline{2} \equiv \lambda fx.\ f(fx)$
...
$\underline{n} \equiv \lambda fx.\ f(...(fx)...)$ [*n* occurrences of *f* in body of λ-term]

Thus, $\underline{n}\ gz$ evaluates to $g^n z$, i.e. the Church numeral $\underline{n}$ is a general *n*-fold function iterator.

The elegance of λ-calculus comes to the surface when we see how addition, multiplication and exponentiation work:

add ≡ $\lambda mnfx.\ mf(nfx)$ ["*m*-fold application of *f* followed by *n*-fold application of *f* gives *m*+*n*-fold application of *f*"]

mult ≡ $\lambda mnfx.\ m(nf)x$ ["*m*-fold application of *n*-fold application of *f* gives *mn*-fold application of *f*"]

exp ≡ $\lambda mnfx.\ nmfx$ ["iterating *m*-fold application *n* times gives an m^n-fold iterator, which applied to *f* gives m^n-fold application of *f*"]

We check addition here as an example:

[40] see http://en.wikipedia.org/wiki/Syntactic_sugar for the meaning of "syntactic sugar", who coined this term, and what syntactic salt and syntactic saccharine is!

$$\begin{aligned}
\textbf{add}\ \underline{m}\ \underline{n} \quad &\to^* \lambda fx.\ \underline{m}\, f(\underline{n}\, f\, x) \\
&\to^* \lambda fx.\ f^m(f^n\, x) \\
&\equiv \quad \lambda fx.\ f^{m+n}\, x \\
&\equiv \quad \underline{m+n}
\end{aligned}$$

The successor function and the test for being zero can be coded as follows:

$$\textbf{succ} \equiv \lambda nfx.\ f(nfx)$$
$$\textbf{iszero} \equiv \lambda n.\ n(\lambda x.\ \textbf{false})\ \textbf{true}$$

Defining the predecessor function **pre** $(\underline{n+1}) \to^* \underline{n}$ and the subtraction **sub** is tricky; one way to do it is shown in Paulson's lecture notes[41] on page 15. Church himself struggled a long time to find a way to get **sub.** Interestingly, recovering the Ackermann function is easier than subtraction:

$$\textbf{ack} \equiv \lambda m.\ m(\lambda f\, n.\ nf(f\,\underline{1}))\ \textbf{succ}$$

does it. It is altogether one page of straightforward calculation to show that the defining equations of the Ackermann function hold:

$$\textbf{ack}\ \underline{0}\ \underline{n} = \underline{n+1}$$
$$\textbf{ack}\ (\underline{m+1})\ \underline{0} = \textbf{ack}\ \underline{m}\ \underline{1}$$
$$\textbf{ack}\ (\underline{m+1})\ (\underline{n+1}) = \textbf{ack}\ \underline{m}\ (\textbf{ack}\ (\underline{m+1})\ \underline{n})$$

Defining functions by recursion

The reconstruction of the Ackermann function by a lambda expression unfortunately comes "out of the blue" (like a good busy beaver program) and does not illuminate us toward a general-purpose recipe for creating recursive functions in the λ-calculus. But such a general recipe exists, is simple, mathematically transparent, and is indeed one of the main reasons why you should know about CALC at all.

This general recipe has one super-power ingredient: a *fixed point combinator Y*. A fixed-point combinator is any combinator Y with the property that for any term f,

$$Yf = f(Yf)$$

Y is called fixed point combinator because for every term f, Y creates a fixed point of f, namely Yf. There are many ways to specify a fixed-point combinator by a λ-term. The most frequently used one goes back to Curry:

$$Y \equiv \lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx))$$

The following derivation shows us that Y thus defined does what we want:

$$\begin{aligned}
Y\,F \quad &\to\ (\lambda x.\ F\,(xx))\ (\lambda x.\ F\,(xx)) \\
&\to\ F\,((\lambda x.\ F\,(xx))\ (\lambda x.\ F\,(xx)))
\end{aligned}$$

[41] Originally from http://www.cl.cam.ac.uk/users/lcp/papers/Notes/Founds-FP.pdf; copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/Founds-FP.pdf

$$= \quad F\,(Y\,F)$$

The fixed point combinator can be expressed in terms of S, K, I as follows (check it if you are of a playful disposition):

$$Y = S\,(K\,(S\,I\,I))\,(S\,(S\,(K\,S)\,K)\,(K\,(S\,I\,I)))$$

To demonstrate how recursion is realized using Y, we shall encode the factorial function and the infinite list [0,0,...] in λ-calculus, realising the recursion equations

fact N = **if** (**iszero** N) $\underline{1}$ (**mult** N (**fact** (**pre** N))),
zeroes = **cons** $\underline{0}$ **zeroes**,

which of course are no valid definitions, because the to-be-defined combinator appears on both sides of the "=" – but that's just the essence of recursive definitions! However, with a little help from Y we can turn these equations into valid definitions:

fact ≡ $Y\,(\lambda gn.$ **if** (**iszero** n) $\underline{1}$ (**mult** n (g (**pre** n))))
zeroes ≡ $Y(\lambda g.$ **cons** $\underline{0}$ g)

In each of these definitions, the recursive call is replaced by the variable g. We verify that this works for **zeroes** and leave **fact** as an exercise.

zeroes = $Y\,(\lambda g.$ **cons** $\underline{0}$ g)	[by definition]
= $(\lambda g.$ **cons** $\underline{0}$ g) ($Y(\lambda g.$ **cons** $\underline{0}$ g))	[use fixed point property of Y]
= $(\lambda g.$ **cons** $\underline{0}$ g) **zeroes**	[plug in def. of **zeroes**]
= **cons** $\underline{0}$ **zeroes**	[β-reduction]

This is a fast, human-insight based derivation. The result **zeroes** = **cons** $\underline{0}$ **zeroes** could also be derived with 100% formal rigor using our λ-conversion calculus from Def. 7.6:

1.	$Y = \lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx))$	premise
2.	**zeroes** = $Y\,(\lambda g.$ **cons** $\underline{0}$ g)	premise
3.	$Y\ (\lambda g.$ **cons** $\underline{0}$ g) = $(\lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx)))\ (\lambda g.$ **cons** $\underline{0}$ g)	I4 on 1.
4.	$(\lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx)))\ (\lambda g.$ **cons** $\underline{0}$ g) = $((\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx))\ (\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx)))$	A3 on 3.
5.	$(\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx))\ (\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx))$ = $(\lambda g.$ **cons** $\underline{0}$ $g)((\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx))\ (\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx)))$	A3 on 4.
6.	**zeroes** = $((\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx))\ (\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx)))$	I2 on 2., 3., 4. (2 times)
7.	$(\lambda g.$ **cons** $\underline{0}$ $g)((\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx))\ (\lambda x.\ (\lambda g.$ **cons** $\underline{0}$ $g)(xx)))$ = $(\lambda g.$ **cons** $\underline{0}$ $g)$ **zeroes**	I5 on 6.
8.	**zeroes** = $(\lambda g.$ **cons** $\underline{0}$ $g)$ **zeroes**	I2 on 2., 3., 4., 5. (3 times)
9.	$(\lambda g.$ **cons** $\underline{0}$ $g)$ **zeroes** = **cons** $\underline{0}$ **zeroes**	A3 on 8.
10.	**zeroes** = **cons** $\underline{0}$ **zeroes**	I2 on 8., 9.

The combinator **zeroes** is a mathematical object that has the (defining) property **zeroes = cons** $\underline{0}$ **zeroes.** We can "visualize" this property by writing

zeroes = $\underline{0}$ :: $\underline{0}$ :: $\underline{0}$ :: $\underline{0}$ :: $\underline{0}$ …

But this does not mean that **zeroes** is something like an "infinite-size λ-expression". The representation of **zeroes** as a right-infinite sequence is just our human-intuitional way to visualize the property **zeroes = cons** $\underline{0}$ **zeroes**. Combinators that can be "visualized" in this way as right-infinite sequences are called *streams*. Streams are perfectly normal citizens of the data type universe, and one can define and apply perfectly normal selector operations for them, like **first** or **second** or **third**, with the obvious semantics.

7.4 λ-calculus and recursive functions

Given that we can realize recursion in λ-calculus, it is not surprising that we can easily represent all primitive recursive functions in λ-calculus:

- For 0 use $\underline{0}$.
- For the successor function use **succ**.
- For the projection p^r_i use $\lambda x_1...x_r.\ x_i$
- To get the substitution scheme, assume that the functions f: $\mathbb{N}^r \to \mathbb{N}$ and $g_1,..., g_r$: $\mathbb{N}^m \to \mathbb{N}$ are represented by λ-terms F, G_1, ..., G_r. Their composition is declared in λ-calculus by

  $H \equiv \lambda x_1...x_m.\ F\ (G_1\ x_1...x_m)...(\ G_r\ x_1...x_m)$

- To get primitive recursion, assume that the functions f: $\mathbb{N}^r \to \mathbb{N}$ and g: $\mathbb{N}^{r+2} \to \mathbb{N}$ are represented by λ-terms F, G. The primitive recursive function H: $\mathbb{N}^{r+1} \to \mathbb{N}$ is defined by

$$H \equiv Y\ (\ \lambda h y x_1...x_r.\ \textbf{if}\ (\textbf{iszero}\ y)\ (F\ x_1...x_r)\ (G\ (\textbf{pre}\ y)\ (h\ (\textbf{pre}\ y)\ x_1...x_r)\ x_1...x_r)\ )$$

The μ-recursion scheme can likewise be described within λ-calculus, but it is more involved and we omit it here (see the Paulson lecture notes for a simplified version which covers μ-recursion from totally defined functions, and the Engeler book for a complete but very condensed treatment).

Thus, the recursive functions can be characterized within the λ-calculus. The converse is also true: every (partial or total) function definable within the λ-calculus is a recursive function. The proof proceeds in two steps. First, one describes an interpreter mechanism for evaluation of λ-terms, by specifying term transformation rules in the spirit of our reduction calculus (but including detailed specifications of evaluation order to end up with a deterministic mechanism). Second, the rules of this mechanism must be translated into the rules of a TM which mimicks the rules. This is yet another simulation exercise and shows that the functions definable within the λ-calculus are TM-computable, which is equivalent to being recursive.

7.5 A simple functional programming language

We now show how the λ-calculus can be directly turned into a programming language. This yields a *functional* programming language. The programming language that we will use here was written and implemented by Lloyd Allison from Monash University, Australia. He hasn't named it, I will call it L5, which of course stands for LLoyd's Little Lambda Language. Much of the following description of L5 I have just copied from Lloyd Allison's lambda calculus webpage http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/ (or http://www.allisons.org/ll/FP/Lambda/). Lloyd Allison has backed up this webpage with an interpreter for L5, so you can type in and run little L5 programs there.

L5 is very similar to ISWIM ("If you see what I mean"), an early functional programming language devised by Peter Landin in the mid-60'ies to analyse Algol 60, a then dominating programming language (and, allow me a flash of nostalgia, the language that I was trained on during my university studies, using punchcards as input medium). ISWIM dominated the early literature on functional programming and was the model for ML, a functional programming language still in widespread use – as you know.

Here is the central part of the syntax of L5:

```
<program> ::= <Exp>

<Exp> ::= <ident> | <numeral> | '<letter>' | () | true | false | nil |
          ( <Exp> ) | <unopr> <Exp> | <Exp> <binopr> <Exp> |
          if <Exp> then <Exp> else <Exp> |
          lambda <param> . <Exp> |  <Exp> <Exp> |
          let [rec] <decs> in <Exp>

<decs>  ::= <dec>,<decs> | <dec>
<dec>   ::= <ident>=<Exp>

<param> ::= () | <ident>

<unopr> ::= hd | tl | null | not | -
<binopr> ::= and | or | = | <> | < | <= | > | >= | + | - | * | / | ::

priorities:   ::               1  cons list (right associative)
              or               2
              and              3
              = <> < <= > >=   4  scalars only
              + -              5  (binary -)
              * /              6
              application      7
              - hd tl null not 8  (unary -)
```

Some comments:

The basic construct of a L5 program is an expression `<Exp>`. If you look at the grammar, you will find that L5-expressions are essentially λ-terms, with predefined constants `true`, `false`, `nil`, atomic data types `<numeral>` (which means integer; L5 only does integer arithmetics) and `'<letter>'` (i.e., alphanumeric symbols). Binary operators can be inputted in infix format.

The above syntax is not quite complete. For completion it would need subgrammars that declare what strings are admissible for variable names (i.e. the "identifiers" `<ident>`) and what letters are allowed (the `<letter>`). Lloyd's website does not provide these specs. If you use for identifiers alphanumeric sequences that start with letters (like `var12`, `x`, `t`, `f`, `x1`, `x2`) you are on the safe side.

The `let [rec] <decs> in <Exp>` construct serves to pass values to the variables of a function. It directly corresponds to the evaluation of λ-terms, that is, to β-reduction, as follows:

`let x = M in N`	≡	$(\lambda x.\ N)\ M$	[single declaration example]
`let x = M, y = K in N`	≡	$(\lambda xy.\ N)\ M\ K$	[multiple declarations example]

As an important special case, if *M* is a function, this reads

`let f = lambda x1. ... lambda xn. M in N` ≡ $(\lambda f.\ N)\ (\lambda x_1 \ldots x_n.\ M)$

The optional `rec` argument in `let` is for recursive function definitions and relates to λ-terms like this:

`let rec f = lambda x1. ... lambda xn. M in N` ≡ $(\lambda f.\ N)\ (Y\ (\lambda g x_1 \ldots x_n.\ M))$

As a little starter demo, this is how one can write the factorial function in L5:

```
let rec fact = lambda n. if n=0 then 1 else n*fact(n-1)
in fact 10
```

Nicely compact, isn't it? In λ-calculus, this program would correspond to the term

$(\lambda fact.\ fact\ \underline{10})\ (Y(\lambda gn.\ \textbf{if}\ (\textbf{iszero}\ n)\ \underline{1}\ (\textbf{mult}\ n\ (g\ (\textbf{pre}\ n)))))$

As can be seen from this example, in L5 you cannot first define a function and then later independently use the definition. (There is no "`define function`" operator in L5). A function can only be defined within a `let` environment (which at its end typically evaluates the function). The fundamental structure `<program> ::= <Exp>` of a L5 program implicitly says "a program corresponds to a λ-term, there is no way of sequencing commands like in imperative programming languages".

A more interesting example is the `hamm` program given at http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/03.Prog2.Hamm.shtml. Like the **zeroes** lambda expression that we considered earlier, `hamm` is best understood as representing a stream. In this case, `hamm` can be visualized as an "infinite list" which contains in ascending order the Hamming integers, i.e. all integers which are either 1 or some multiple of 2, 3 and/or 5. The sequence begins 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... Note that 7, 11, 13, 14 are *not* in the list.

The Hamming program (see further below) defines a stream which begins with the first Hamming number `1':

```
hamm = 1 :: ...
```

Besides 1, the Hamming numbers are all numbers whose factor decomposition only has prime factors 2, 3, 5. Stated differently, the Hamming numbers is the smallest set of integers containing 1 which is closed under multiplication with 2, 3, and 5. The program therefore continues from `1 :: ...` by *merging* the results of multiplying the members of the list by 2, 3 and 5.

```
merge (mul 2 hamm) (merge (mul 3 hamm) (mul 5 hamm))
```

Merging two increasing lists of integers means to compute the sorted union of the two lists (weeding out duplicates). The merge and multiply functions can be implemented quite straightforwardly. The program below never terminates but continues to print Hamming numbers until it runs out of space or is killed.

```
let rec
   merge = lambda a. lambda b.
      if hd a < hd b then (hd a)::(merge (tl a) b)
      else if hd b < hd a then (hd b)::(merge a (tl b))
           else (hd a)::(merge (tl a) (tl b)),

   mul = lambda n. lambda l. (n * hd l)::(mul n tl l)

in let rec
   hamm = 1::(merge (mul 2 hamm) (merge (mul 3 hamm) (mul 5 hamm)))

   in hamm
```

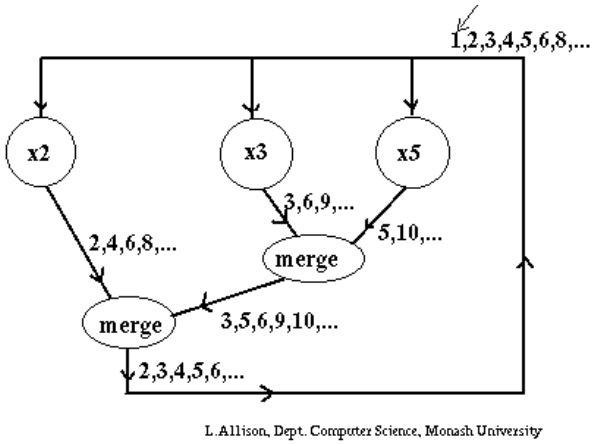


Figure 7.2: data streams in `hamm`.

The merge and multiplication functions can be thought of as processes communicating by streams of values. The initial value '1' has to be injected to start the calculation. Many operating systems and other useful programs have similar structures where graphs of processes communicate through streams.

General remarks about functional programming languages

There are many functional programming languages, some of them very close to λ-calculus, others with extensive add-ons. The functional programming languages FAQ at

http://www.cs.nott.ac.uk/~gmh/faq.html[42] lists about 20 such languages – the best known are probably Haskell, ML, Miranda, Scheme, and Lisp. (However, Lisp is not mentioned in that FAQ list! the reason is that Lisp is (in)famous for not being a "clean" functional programming language; in fact, Lisp is arguably the programming language with the largest set of command primitives, and like a chameleon, a Lisp program can on the surface mimic almost every programming style. However, the *Common Lisp* standard has a rigorous functional evaluation scheme underneath the gaudy surface.)

Generally, quoting from the FAQ

> "Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style."

Functional programming should be contrasted to imperative programming styles (and languages, such as C). In a C (or Pascal or Basic or Java... they are all imperative languages) program, you (the programmer) specify the operations and sequence of actions that are carried out when the program executes. You want, and get, *control* over the ticks of the computational clockwork. By contrast, in a functional programming style, you don't care (or pretend not to care) about the actual mechanism and sequence of manipulations that your computer does when it carries out the program. You just declare your problem to the machine, essentially by defining a function that results in the answer of your problem, and leave all the details of function evaluation to the interpreter and/or compiler. This is the basic idea of *declarative* programming styles: writing a program is "just" specifying the problem; the operations that derive a solution from your problem declaration are hidden from you and you don't even want to bother. Functional programming is one approach to declarative programming; the other main approach is logic programming (where you declare your problem as a logical formula that you wish to have proved by a mechanism which you don't want to care about).

To put our little treatment of lambda calculus and functional programming into perspective, you might enjoy a condensed overview about history, types and principles of programming languages[43] and at http://people.ku.edu/~nkinners/LangList/Extras/classif.htm[44] where you find a short explanation of about 20 basic categories of programming languages and styles. You might also wish to get a feeling for history by taking a glance at the genealogical tree of the 150 most important programming languages compiled by Pascal Rigaux, which I found at http://rigaux.org/language-study/diagram.html (Figure 7.3).

A note on the practical usefulness of functional programming languages. They have one major disadvantage: execution of functional programs is slow, and they have one big

[42] local Jacobs copy (retrieved April 5 2010) at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/faq_compLangFunctional.html

[43] http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/ProgrammingParadigmsHistory.html (fetched from http://www.cs.qub.ac.uk/~J.Campbell/myweb/oop/oophtml/node4.html in about 2005, this website is not longer active)

[44] local Jacobs copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/classifyingProgLangs.html , the original URL is now unavailable

advantage: functional programs are transparent, and their behavior can be described rigorously using lambda calculus. The first property renders functional programming languages often unsuited for industrial and business applications. But the second property makes them a premier choice for program and compiler verification tools. For instance, the HOL system[45] (written in ML) "is a powerful and widely used computer program for constructing formal specifications and proofs [about λ-calculus statements] in higher order logic. The system is used in both industry and academia to support formal reasoning in many different areas, including hardware design and verification, reasoning about security, proofs about real-time systems, semantics of hardware description languages, compiler verification, program correctness, modelling concurrency, and program refinement. HOL is also used as an open platform for general theorem-proving research." (from http://homepages.inf.ed.ac.uk/wadler/realworld/ where real-world applications of functional programming languages are listed). And, of course, their mathematical transparency also makes functional programming languages a frequent choice as a first language for students in university computer science courses.

[45] http://hol.sourceforge.net/

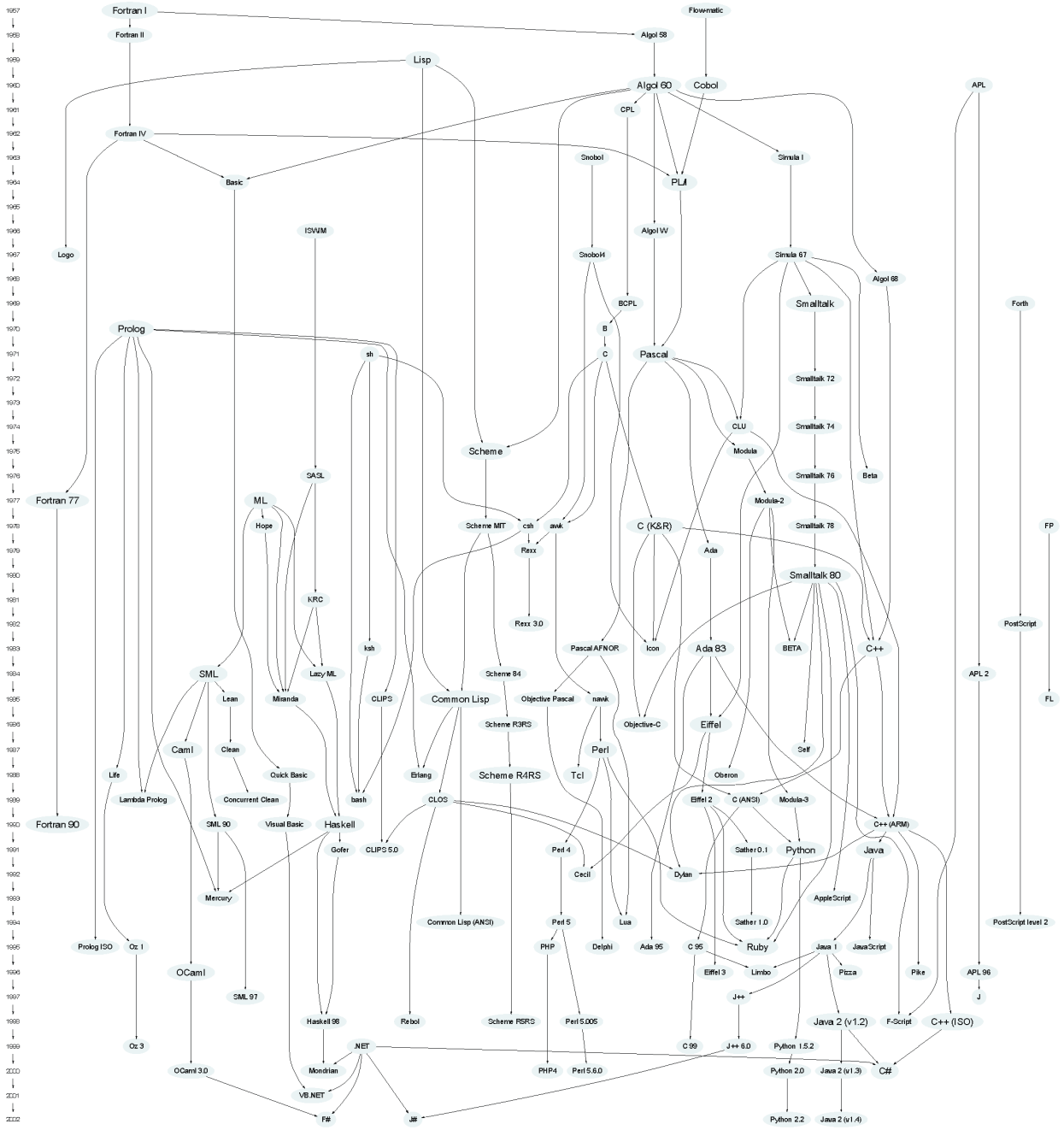


Figure 7.3: a genealogy of programming languages. From Pascal Rigaux's stunning website on programming languages (rigaux.org)

Part 2: Complexity

8 Two elementary problems: REACHABILITY and TSP

We learnt earlier in this lecture that TMs (or algorithms in general) can be used to (i) compute numerical functions, (ii) decide / accept languages, (iii) solve "problems". These three aspects are basically equivalent (Section 3.1). For the theory of computational complexity, the third perspective is most commonly adopted.

Remember that a "problem", in the language of theoretical computer science, is a bag of questions that can be rigorously formalized and be offered to a computer (or a TM or a RAM or to lambda calculus...), typically in the form of yes/no questions. To get started on the theme of complexity, we will present two concrete problems. The first of these problems, "finding out whether in a given graph a certain node is reachable from another given node", can be solved in polynomial time, whereas the other, the "traveling salesman problem", is famous for the fact that (very likely) it needs exponential time.

8.1 The graph reachability problem

Let $G = (V, E)$ be a (directed) graph, i.e. V is a set $\{v_1, \ldots, v_n\}$ of nodes (or vertices) and $E \subseteq V \times V$ a set of edges. For ease of notation we often identify nodes with their indices, i.e. write n for v_n and (i,j) for (v_i, v_j). The most basic problem on graphs is this: given nodes 1 and n, is there a path from 1 to n? This problem is called REACHABILITY.

Remarks:

- We will generally give problems names that we write in SMALLCAPS. These names are rather standardized today and allow you to recognize problems across different articles and books.
- REACHABILITY is a *decision problem*, in that we want a yes/no answer. This is the prevailing kind of problem. Other types of problems are for example optimization problems or function problems, which we will encounter soon.
- Each problem comes in an infinite number of *instances*. For example, each graph and pair of nodes in this graph yields one instance of REACHABILITY. An algorithm that solves the problem must deal with all instances.
- In order to offer a problem instance to a computer, it first must be *coded* into the input format required for that computer. For instance, when we use TMs for solving instances of REACHABILITY, the first thing to do is to design a coding scheme that transforms any graph $G = (V, E)$ into a string made from a finite (input tape) alphabet. One possibility would be to use as tape alphabet the set $\{0, 1, (,), ;\}$, code the nodes $\{v_1, \ldots, v_n\}$ by the binary representations of the integers 1, ..., n:

  $v_1 \rightarrow \langle v_1 \rangle = 1$
  $v_2 \rightarrow \langle v_2 \rangle = 10$
  ...
  $v_n \rightarrow \langle v_n \rangle =$ [binary representation of integer n],

  code edges as pairs of codes of nodes, for instance

  $(1, 3) \rightarrow (\langle v_1 \rangle; \langle v_3 \rangle) = (1;11)$

and code the entire graph as a sequence that starts with the code of n, followed by the codes of edges. For instance, this would be a code of a three-node graph with edges (1,1) and (2,3):

$$<(V, E)> = 11(1;1)(10;11)$$

Such a coding scheme is arbitrary – you may invent yours according to taste. However, a coding scheme must have certain properties in order to qualify:

a. Given any string from the coding alphabet (here: {0, 1, (,), ;}), it must be *decidable* whether the string is a well-formed code of a problem instance.
b. Given a code of a problem instance, the problem instance must be *uniquely characterized* by the code.
c. The code of a problem instance must be *concise* in the sense that the length of the codeword reflects the bit information needed to describe the instance. Note that this requirement cannot be formulated rigorously because the original description of the problem instance can itself be formulated in various ways. For instance, an n-node graph can be represented by a list of the edges or as an n by n matrix with entries 1 at positions (i,j) corresponding to edges and entries 0 at other positions. Thus, this requirement is actually an informal appeal not to use a "stupid" representation of the problem instances.

A suitable coding scheme is typically not described explicitly but just taken as granted.

REACHABILITY can be solved by a simple algorithm: maintain a set S of nodes. Initially, $S = \{1\}$. Each node can be *marked* or *unmarked*. Initially only 1 is marked. At each iteration of the algorithm, choose one node i from S and remove it from S. Then process all edges (i,j) going out from i. If node j is unmarked, mark it and add it to S. Continue until S is empty. Answer "yes" if at this point n is marked, else "no".

This is of course an informal description of an algorithm, and it has gaps (we have left unspecified how the node i is chosen from S – e.g., whether by depth-first or breadth-first choice). It could be turned into a formal description by specifying a TM with input tape alphabet {0, 1, (,), ;} that carries out the computations described above. In complexity theory, one very often works and argues with such informal descriptions, relying on the assumption that the problem *could* be coded into a TM format.

Let's carry out a quick investigation of the number of steps required by this algorithm. It is clear that the algorithm processes every connection (i,j) at most once. There are at most n^2 edges in G. Assume that the other computations associated with an edge processing (choosing the edge, marking etc.) can be done in constant time. Under this assumption, the overall processing time is at most a constant factor times n^2, that is, it runs within time $O(n^2)$. Note that this finding is not only quick, but also dirty, because the assumption of constant time for edge processing operations is unrealistic – the larger the graph, the more time intensive will memory access operations become, for instance.

If we carry out a similar investigation using TMs and our coded version of the problem, things look a bit different. First, we have learnt earlier in the lecture that the computational performance of a TM is measured w.r.t. the length m of input words. If we use our coding scheme introduced above to turn a graph into a word over {0, 1, (,), ;}, then the length m of the code is related to the size n of the graph as follows. The length of codes $(< v_i >; < v_j >)$ for edges is $O(\log(n))$. In the worst case, when we have a fully connected graph, we have n^2

edges. In addition we have the leading code for n which adds another $\log(n)$ symbols. All in all, the code $<(V, E)>$ for a graph with n nodes may reach a length of $m = O(\log(n)\, n^2)$, or, more generously, $m = O(n^3)$. Next, we have to procure us with a TM that on input $<(V, E)>$ computes the correct yes/no answer. We skip this here and assume we have a TM M that in the worst case has to traverse its input tape n times back and forth (I can easily think of a multi-tape TM that solves REACHABILITY in this way). This machine would therefore run within time $O(n^4)$. However, the time consumed by a TM should be measured w.r.t. the length m of its input word, not w.r.t. n. Thus, we should rather say that our TM has a time complexity of $O(m^{4/3})$.

This appears confusing, and indeed it is. The bad news is that we are confronted with at least three different ways of fixing a time bound for computing REACHABILITY:

1. Using a natural measure n for the "size" of a problem instance and an informal algorithm, we get a time bound of $O(n^2)$.
2. Using the same measure n for the problem size, but a concrete TM algorithm, including a proper coding scheme, we get $O(n^4)$.
3. Using the TM algorithm and the length m of the codeword $<(V, E)>$ for a problem instance as a basis for measuring the "size" of the instance, we get $O(m^{4/3})$.

Further complications arise from the arbitrariness of our coding scheme and the fact that we might design our TM in different ways. But now comes the good news:

All our time bounds $O(n^2)$, $O(n^4)$, $O(m^{4/3})$ are polynomially related. This reflects a general experience: switching between different bases of measuring problem size, different encodings, different concrete TM designs and even between TMs and RAMs and PCs will always (if we don't do really stupid things) lead to time bounds that are polynomially related. Therefore, as long as we are interested only in characterizing the computational complexity of problems up to polynomial relatedness, it is not important which of the options 1.—3. we choose. Some notes:

- In the remainder of this lecture, we will be interested only in analyzing complexities up to polynomial relatedness.
- In textbooks, these difficulties are sometimes brushed over, and sometimes you find that authors switch between the three perspectives without ever explaining.
- Once one gets interested in a detailed analysis of linear or even sublinear computation times, this issue of coding, choosing a basis for measuring problem size, etc., becomes really important.

8.2 The Traveling Salesman problem (TSP)

This is one of the most intensely researched computational problems, and very likely you know it already. Problem instance: a set C of n cities (denote them simply by $1,\ldots,n$) and a distance function d: $C \times C \to \mathbb{N}_+$. of pairwise inter-city travel distances (symmetric of course). Wanted: a travel route of minimal total length which starts in city 1, visits all cities exactly once and returns to 1.

A naïve algorithm for TSP enumerates all possible travel routes, computes their lengths, and returns the shortest one. Because there are ½ $(n-1)!$ tours to be considered, this naïve algorithm has time complexity $\Omega((n-1)!)$, which is even far worse than exponential.

A less naïve algorithm with better time complexity roughly goes like this. For each subset S of cities *excluding city* 1, and for each $j \in S$, define $c[S, j]$ to be the shortest path that starts from city 1, visits all cities in S and ends in j. By skilfully calculating $c[S, j]$ from smaller sets $S' \subset S$, one arrives at an algorithm with time complexity $O(n^2\, 2^n)$, "only" exponential.

Note that TSP is not a decision problem – its answers are not of the yes/no kind. Rather, TSP is an *optimization* problem, where the result is a data structure (here: a graph path) that optimizes some evaluation criterion.

A closely related variant of the TSP is however a decision problem: Given C and d, and additionally a "mileage budget" B, the yes-no-question is whether there is a route of length at most B. This decision-type version is denoted TSP(D). It is easy to see that the time complexity of TSP(D) is at most the time complexity of TSP. But also the other direction holds in a certain sense: if we had a polynomial-time algorithm for TSP(D), we could use it to obtain a polynomial-time algorithm for TSP. This is not quite trivial (see Chapter 5 in Garey/Johnson); we mention this fact here to motivate that we will mainly work on decision problems in this course.

After half a century of intense investigations, no better than exponential-time algorithms for TSP have been found. Today most researchers are convinced that they don't exist. We'll learn why.

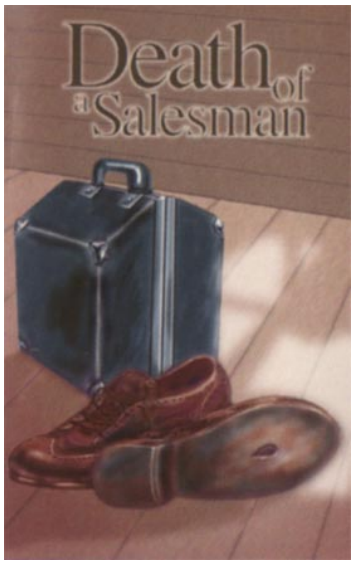


Figure 7.1[46]: What $O(n^2\, 2^n)$ implies in practice.

[46] http://www.bte.org/shows/2001-2002/salesman.html

8.3 On polynomial vs. exponential time complexities

REACHABILITY can be solved in polynomial time, and TSP (for all we know) only in exponential time. Polynomial vs. exponential time complexity is a fundamental distinction for algorithms. Roughly speaking, problems with exponential time complexity quickly outgrow all our technical means, present and future, to solve them, whereas with polynomial-time problems there is always hope. The awe-inspiring growth of exponentials becomes clear from the following table with a few examples (from M. R. Garey & D. S. Johnson, Computers and Intractability, Freeman: New York 1979):

n $f(n)$	10	20	30	40	50	60
n^2	0.0001 sec	0.0004 sec	0.0009 sec	0.0016 sec	0.0025 sec	0.0036 sec
n^5	.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13.0 min
3^n	.059 sec	58 min	6.5 years	3855 centuries	2e8 centuries	1300000 billion yrs.

Table 8.1. Some running times for algorithms of different time complexities $f(n)$. An execution time of one microsecond per operation is assumed.

Table 8.1 drastically shows that we will very likely never be able to execute an $O(3^n)$ algorithm for a problem instance with $n = 60$. In contrast, the polynomial time complexity $O(n^5)$ looks almost harmless. Here are some commonly stated beliefs of the field:

- Algorithms with exponential time complexity are practically impossible to use except for very small problem instances.
- Algorithms with polynomial time complexity are deemed "practically useful".
- Experience tells that if an algorithm is found that solves a problem with polynomial time complexity of high order (say, $O(n^{10})$), then it usually does not take long until another polynomial algorithm is found with low order (say, $O(n^{2.5})$). A good demonstration of this point is the discovery of a polynomial-time algorithm for deciding primality of an integer. The first algorithm (the now famous AKS-algorithm[47], named after their inventors Manindra Agrawal, Neeraj Kayal and Nitin Saxena) had a complexity of $O(n^{12})$. Only some months after its discovery in 2002, other researchers brought this down to $O(n^6)$ for a version that covers all integers. Later Agrawal proposed an $O(n^3)$ algorithm which however is only correct if a certain mathematical conjecture is true, which remains to be proven[48].

Taking all these facts and feelings together, the polynomial vs. exponential divide in time complexity is considered as *the* "continental divide" that separates useful from impractical algorithms.

[47] http://en.wikipedia.org/wiki/AKS_primality_test
[48] https://en.wikipedia.org/wiki/Primality_test

However, this wisdom sometimes has to be received with some caution:

- If typical instances of a problem have large n (say, in the order of 1000), and if there are high demands on speed, then even an $O(n^2)$ algorithm can be painfully slow, and much ingenuity would be spent on improving it. Important example: DNA sequence alignment.
- There are problems when the "arbitrary" constant hidden in the big-Oh notation actually becomes large. A very relevant example is matrix- matrix multiplication. Standard algorithms have time complexity $O(n^3)$ where n is the matrix size. For very big matrices of sizes $n \sim 1{,}000{,}000$ (which standardly appear in the computations of theoretical physics, for example), $O(n^3)$ becomes cumbersome. But, $O(n^{2.8})$ would still be practical (it would run about 20 times faster). Indeed, $O(n^{2.8})$ algorithms for matrix-matrix multiplication have been devised. However, they are very complicated and their "overhead" leads to a large constant coefficient in the $O(n^{2.8})$, which renders this type of algorithms useful only for very large matrices. The current record is an algorithm with $O(n^{2.38})$, but its constant overhead is so large that this algorithm would become relevant only for matrix sizes that cannot be handled on today's computing machinery (see http://en.wikipedia.org/wiki/Matrix_multiplication_algorithm).

9 Nondeterminism and the first appearance of N = NP

A not very stupid and very common way to attack a difficult problem is to first guess candidate solutions and then check whether they are indeed correct solutions. If the guessing and checking can be done in polynomial time, we witness a problem in one of the most (in)famous complexity classes, namely, **NP**. In this section we will formally introduce NP by describing nondeterministic ("guessing") TMs, and explain intuitively that, in a sense, many practical problems that you might encounter as a computer scientist, are in **NP**.

A nondeterministic TM looks by and large like an ordinary (deterministic) TM, with the sole exception that at each time step, the nondeterministic machine can randomly select between a number of alternative next actions. Formally, this turns the transition *function* into a *relation*. I follow here the definition given by Papadimitriou (like always in complexity theory, other authors have other definitions, which may look different but are all equivalent – the powers of mutual simulation!)

Definition 9.1 (nondeterministic TM). A nondeterministic (single-tape) Turing machine is a structure $M = (K, \Sigma, \Delta, s)$, where K, Σ, s are defined like in deterministic TMs, just as we also keep our conventions concerning the special symbols $\sqcup$ and $\triangleright$. Δ is a relation

$$\Delta \subseteq (K \times \Sigma) \times [(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}]$$

That is, for every state-symbol combination (q, a) there may be several possible next steps – or none at all. Multi-tape TMs are defined accordingly.

A *configuration* of a nondeterministic TM M is a triple (q, w, u), where q is the current state, w is the symbol string to the left of the cursor, including the symbol currently scanned by the cursor, and u is the symbol string to the right of the cursor, as we had it with deterministic TMs. The workings of a nondeterministic TM are formalized, as follows:

Definition 9.2 If there is a rule in Δ which carries M from configuration (q, w, u) to configuration (q', w', u') in one step, we write $(q, w, u) \xrightarrow{M} (q', w', u')$. Furthermore, $(q, w, u) \xrightarrow{M^k} (q', w', u')$ and $(q, w, u) \xrightarrow{M^*} (q', w', u')$ are derived from $(q, w, u) \xrightarrow{M} (q', w', u')$ as k-step iteration and as transitive closure, respectively, like usual.

A nondeterministic TMs M accepts a language in a way that is reminiscent of how pushdown automata accept languages:

Definition 9.3: A nondeterministic TM M *accepts* a word $w \in L$, if $(s, \triangleright, w) \xrightarrow{M^*} (''yes'', u', v')$, that is, there exists an accepting run for input w. The language L *accepted by* M is the set of words for which an accepting run of M exists.

Note that we don't care whether there are other runs that take w into the rejecting state, or don't halt. The only criterium for M to accept w is that there exists *some* run which ends in the accepting state.

A note on terminology: I used the term "*L accepted by M*" in agreement with W. Paul. Other authors speak of "*L decided by M*" (Papadimitriou) or of "*L recognized by M*" (Garey/Johnson). This terminological mess has a reason. When we say, a deterministic TM M *decides* a language L, then M gives a yes or a no answer for every input. Specifically, if a word w is not in L, then M tells us so. By contrast, if w is not in L, our nondeterministic TM cannot tell us. Therefore, it seems unsuitable to speak of a nondeterministic TM *deciding* a language. However, as we will see shortly, nondeterministic TMs *whose time complexity is known* can in fact be used to decide languages – and that is actually the way they are most often used. Therefore, again, it might seem appropriate to speak of nondeterministic TMs deciding languages. Deep down, the confusion arises from the fact that nondeterministic TMs are not really models of computation: they are models of guessing / verification mechanisms. But "decide" is a computational notion, so there is an inbuilt misfit of intuitions here. As it turns out, this misfit is very productive.

The computation time of a nondeterministic TM is given by the time needed by an accepting run:

Definition 9.4, Version a: A nondeterministic TM M *accepts a language L in time* $f(n)$, if M accepts L, and moreover for any $w \in \Sigma^*$, if $(s, \triangleright, w) \xrightarrow{M^k} (q, u, v)$ for any (q, u, v), then $k \leq f(|w|)$. That is, we require that on no input w, M ever runs longer than $f(|w|)$.

This definition is taken from Papadimitriou, and we will use it. Other authors use a different definition:

Definition 9.5, Version b: A nondeterministic TM M *accepts a language L in time* $f(n)$, if M accepts L, and moreover for any $w \in L$, there exists an accepting run $(s, \triangleright, w) \xrightarrow{M^k} (''yes'', u, v)$ with $k \leq f(|w|)$.

The two definitions are by and large equivalent (up to $O(f(|w|))$): if a nondeterministic TM M accepts a language L in time $f(n)$ according to Version a, then there exists another TM M' accepting L in time $f(n)$ according to Version b, and vice versa (under certain conditions on f, see Definition 10.1; exercise!).

For space, we have the following definition:

Definition 9.6: A nondeterministic TM *M* [with input and output] *accepts a language L in space f*(*n*), if *M* accepts *L*, and moreover for any $w \in L$, there exists an accepting run that uses at most *f*(*n*) tape cells on its working tapes [remember that when dealing with space complexity, we use TMs that have working tapes separate from the input and the output tape; space consumption is measured only on the former].

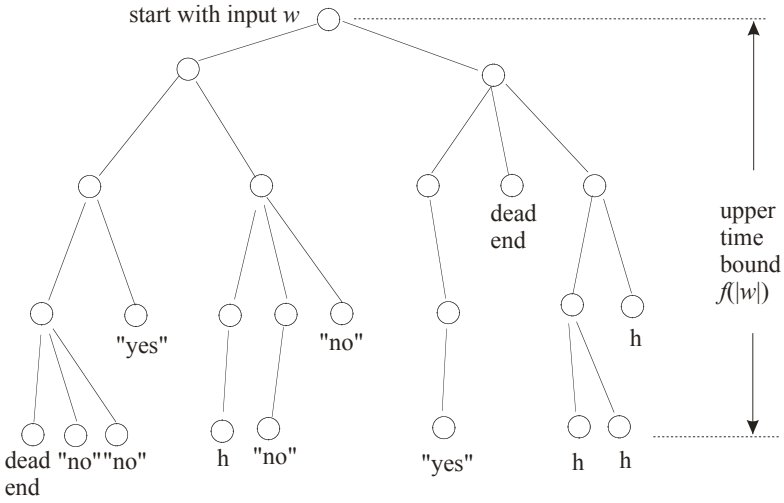


Figure 9.1: schematic diagram of the branching possible computation paths that a nondeterministic TM accepting some *L* in time *f*(*n*) (version a. of Def. 9.4) can take on a given input word *w*. The input word *w* is accepted if one of these many branches ends in "yes"; others may end in "h" or "no" or may simply come to a dead end because Δ provides no continuation.

The class of languages accepted by nondeterministic TMs in time *f*(*n*) is called **NTIME**(*f*(*n*)). The union of all polynomial such classes is **NP**,

$$\mathbf{NP} = \bigcup_{k \geq 1} \mathbf{NTIME}(n^k)$$

Note that it doesn't matter for fixing the class NP how many tapes the TMs have that we use, because the computation times of TMs with different numbers of tapes are polynomially related.

Example: We claim that the problem TSP(D) is in **NP**. (Remember that TSP(D) was the following "decision" variant of the travelling salesman problem: Given a set of cities *C* and distance function *d*, and additionally a "mileage budget" *B*, the yes-no-problem is whether there is a route of length at most *B*. You can interpret this as a language decision problem by interpreting the inputs [i.e. the problem instance descriptions] as words). We can show TSP(D) ∈ **NP**, as follows. We design a multitape nondeterministic TM *M* that accepts TSP(D) in time $O(n^2)$. As a preparation, we invent some coding for travelling routes as symbol sequences, such that the codeword of a travelling route that satisfies the TSP conditions (circular, not hitting any city twice) is not longer than the problem instance description used in the input. *M* works in several stages. First, it writes a completely random (!) sequence of symbols on its second tape, of length no longer than its input. After that it

checks whether this sequence is a valid codeword of a TSP route. If not, it halts with "no". If yes, next it checks whether the total length of route is below the budget bound B. If yes, halt with "yes", if not, halt with "no". Both tasks can easily be carried out in time $O(n^2)$ [says Papadimitriou without blushing]. It is clear that M accepts TSP(D): if for a given input C and d there exists a tour of total mileage below B, then it can be guessed by the TM in its first stage and checked for "mileage below B" in the rest of the computation. Conversely, if there are no tours with mileage below B, then M will never stop with "yes".

The TM in this example works according to the schema, "first guess a candidate solution, then check". We will now show that all language decision problems in class **NP** can be solved by this approach. In fact, some authors introduce nondeterministic TMs as two-stage processing machines, where first a candidate solution is guessed by a random string generation mechanism, and in a second deterministic stage this candidate solution is checked whether it is in L.

Following the treatment from Papdimitriou (chapter 9), as a preparation for stating our main insight, we introduce the concepts of polynomial decidability and polynomial balance.

Definition 9.7 Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on words. R is called *polynomially decidable* if there is a deterministic TM deciding the language $\{x;y \mid (x, y) \in R\}$ in polynomial time. We say that R is *polynomially balanced* if there is some $k \geq 1$, such $(x, y) \in R$ implies $|y| \leq |x|^k$. That is, the length of the second component is bounded by a polynomial of the length of the first component.

Here comes our formal version of the "guess-and-check" strategy. The claim is stated a bit more generally than merely as a version of guess-and-check.

Proposition 9.1 Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomially decidable and polynomially balanced relation R, such that $L = \{w \mid (w, y) \in R \text{ for some } y\}$.

Proof. "$\Leftarrow$ ". Suppose that such an R exists. Then L is decided by a nondeterministic TM M as follows. On input w, M guesses a y of length at most $|w|^k$ (the polynomial balance bound for R) and then uses the polynomial test algorithm on $w;y$ to check whether $(w, y) \in R$. If so, it accepts, otherwise, it rejects. Clearly an accepting computation exists if and only if $w \in L$. "$\Rightarrow$" Suppose that $L \in$ **NP.** Then there is a nondeterministic TM M that accepts L in time $|n|^k$ for some $k \geq 1$. Define R by $(w, y) \in R$ if and only if y is the encoding of an accepting computation of M on input w. (Spelled out in detail, this would need to specify an encoding scheme for computation sequences of M; it is clear for all of you who have once in their lives specified a TM simulation in detail that this can be done). It is clear that R is polynomially balanced and polynomially decidable. Furthermore, by our assumption that M accepts L, we have that $L = \{w \mid (w, y) \in R \text{ for some } y\}$.

An y that comes by R with an accepted input w is called a *polynomial witness*, or a *succinct certificate*, of w being a "yes" instance. The most intuitive way to think about polynomial witnesses is to consider them as "candidate solutions" (as in the TSP(D) problem from above), but other types of polynomial witnesses are also intuitive, for instance the y's encountered in the proof of Proposition 9.1, which were encodings of accepting runs of an accepting nondeterministic TM (these encodings of accepting runs can also be considered as encodings of positive solutions, so the difference to the "candidate solution" interpretation is not that large).

It now becomes clear why so many practical problems are in **NP**. Many real-life tasks require the construction of a "solution" object, for instance a good VLSI layout, a transportation fleet schedule, a cost-efficient allocation of resources, you name it. The solution objects are the "witnesses" of a problem. The solution objects in themselves are typically not too large in comparison to the size of the task description – in our terminology, the witnesses are typically polynomial in size. Furthermore, it is typically not too difficult (that is, requires only polynomial time) to check whether a given solution object indeed is a solution (check for consistency and check for meeting the task's specification for a solution). But this is just another way of saying that the problem is in **NP**. This is why this class is so important, although its underlying mathematical -"computational" model (nondeterministic TMs) is clearly not a practical method. However, the nondeterministic TM model yields a powerful and transparent tool for a mathematical analysis of **NP**.

If you have a nondeterministic TM M which "solves" your problem (i.e., accepts some language L) in its weird nondeterministic fashion in some time bound $f(n)$, you can transform M into a standard deterministic TM M' which solves the same problem – and which you could actually run because it's deterministic. The obvious way to derive M' from M is to make M' go through all of the paths that M might take up to its allotted time horizon $f(n)$, that is, to make M' build trees of the kind shown in Fig. 9.1, and to arrange things such that M' either terminates in the "yes" state if it finds some path of the "nondeterministic tree" that ends with "yes", or else terminates in the "no" state. Note that M' actually decides L in the usual sense of the word, i.e. you always either get a "yes" or a "no" for an answer.

This transformation from M into an equivalent deterministic M' works only because we know that M accepts L *in time bound* $f(n)$. Without that knowledge, we wouldn't know how deep the simulating deterministic TM M' should construct the "run tree" of M.

Given a particular nondeterministic TM M, then there exists a maximum branching factor d such that in all "run trees" of M every node has at most d child nodes. This d depends only on the transition relation Δ of M. As a consequence, the size S (number of nodes) of run trees grows exponentially with exponent $f(n)$ and base of d, that is, $S \leq d^{f(n)}$. Since such a tree can be constructed by a suitably designed (3-tape) M' in time proportional to its size S, we have that M' can decide L in time $O(d^{f(n)})$.

In terms of complexity classes, we can state our last observation as follows:

Proposition 9.2. $\mathbf{NTIME}\big(f(n)\big) \subseteq \bigcup_{d>1} \mathbf{TIME}\big(d^{f(n)}\big)$.

Our first investigations into the complexity of computations have so far led us to the following two intuitions:

- The class of problems which are "practically" solvable is made of the problems for which deterministic polynomial-time algorithms exist, i.e. it is the class **P.**
- Many real-life problems lie in the class **NP.**

The world of computer scientists would be a very comfortable place if most real-life problems would be practically solvable, that is, if **P** = **NP**. Unfortunately, all that we know today is that $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP} = \bigcup_{k\geq 1} \mathbf{TIME}(2^{kn})$**.** The first inclusion follows immediately from the fact that every deterministic TM is a special case of a nondeterministic one. The second inclusion is a special case of Proposition 9.2. Nobody knows whether **P** = **NP**, almost everybody

believes that **P** ≠ **NP**, and one person yet unknown will gain eternal fame by settling this question.

10 Relationships between complexity classes

10.1 Introduction

In this section we will develop two basic insights, one positive and one negative:

- The positive one: There exists proper hierarchies of complexity classes, that is, sequences of the form $\mathbf{TIME}(f_1(n)) \subsetneq \mathbf{TIME}(f_2(n)) \subsetneq \mathbf{TIME}(f_3(n)) \subsetneq \ldots$, where $f_1(n) < f_2(n) < \ldots$. That is, by admitting more time for computation, one can compute properly more problems. Such theorems can be obtained relatively easily and with fine granularity, that is, even rather small increments in computation time lead to properly larger solvable problem classes. Such *hierarchy theorems* are established within a particular type of complexity classes, for instance **TIME**($f(n)$) or **SPACE**($f(n)$).
- The negative one: it is very difficult to establish similar proper inclusion relations *across* different types of complexity classes, for instance when comparing deterministic with nondeterministic complexity classes. To wit, all of the following inclusions are suspected of being proper, but none of them has yet been proven to be so: $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$. Only few such "cross-type" results are known, and we will see some of them.

When reasoning about complexity classes of the form **TIME**($f(n)$) or **SPACE**($f(n)$), one has to make some assumptions about the functions f used to measure the complexity. Specifically, f should not be more difficult to compute (use more time or space) than the (time or space) complexity class it describes.

Definition 10.1: A function $f: \mathbb{N} \to \mathbb{N}$ is a *proper complexity function* (or simply *proper*) iff

1. f is nondecreasing, that is $f(n + 1) \geq f(n)$,
2. f can be computed by some deterministic multitape TM M_f with input and output which on input n (in binary) produces as output a sequence of $f(n)$ many ⊓ (a "yardstick" representation of $f(n)$ using the special quasi-blank symbol ⊓),
3. M_f uses time $O(n + f(n))$,
4. on each of its tapes (except the input and output tape) M_f uses at most space $O(f(n))$.

Using proper complexity functions f, we will investigate the following types of complexity classes[49]:

- **TIME**($f(n)$): deterministic time
- **SPACE**($f(n)$): deterministic space
- $\mathbf{L} = \mathbf{SPACE}(\log(n))$
- **NTIME**($f(n)$): nondeterministic time
- **NSPACE**($f(n)$): nondeterministic space (a nondeterministic TM M uses space S on input x if no possible run on this input covers more than S cells on M's internal tapes)

[49] Keep in mind the remarks after Def. 3.7 concerning the imperfections of notation for complexity functions!

- **NL** = **NSPACE**(log(n))
- **P** = $\bigcup_{k>0}$ **TIME**(n^k)
- **NP** = $\bigcup_{k>0}$ **NTIME**(n^k)
- **PSPACE** = $\bigcup_{k>0}$ **SPACE**(n^k)
- **NPSPACE** = $\bigcup_{k>0}$ **NSPACE**(n^k)
- **EXP** = $\cup_{k\geq 1}$ **TIME**(2^{kn})

10.2 Hierarchy theorems

Let $f(n)$ be a proper complexity function which is at least linear, that is $f(n) \geq n$. Consider the following time-bound version of the HALTING language H (adapted & corrected from Papadimitriou):

$$H_f = \{<M>;x \mid Code(<M>) \text{ and } Standard(x) \text{ and } |<M>| \geq |x| \text{ and if } M(x) = \text{"yes", then } M \text{ on input } x \text{ runs at most } f(2|x|+1) \text{ steps}\}$$

(We use the same convention as in Section 5, that is, we consider only TMs M that operate with symbol set {0,1,#}, and $<M>$ denotes a codeword for M made of symbols {0,1,#}.) We will first show that H_f can be decided essentially by simulation of M in time f^3, and secondly we will use a diagonalization argument to see that H_f cannot be decided in time f. Both insights taken together create a hierarchy of complexity classes.

Proposition 10.1 $H_f \in$ **TIME**($f(m)^3$).

Proof. Let m denote the length $|<M>;x|$ throughout. We build a simulating machine U_f that on input $<M>;x$ checks in time $f(m)^3 = f(|<M>;x|)^3$ whether M accepts x in time $f(2|x|+1)$. We adapt here the proof given in Papadimitriou.

U_f works in several stages. Let $g(n) = f(2n+1)$.

1. U_f first behaves as M_g, the TM associated with g according to Def. 10.1, and writes a "yardstick" of $g(|x|) = f(2|x|+1)$ many quasi-blanks ⊓ on a special tape. Time needed: O($f(2|x|+1)$)
2. Next, U_f checks whether $Code(<M>)$ and $Standard(x)$ and $|<M>| \geq |x|$. If not, reject input. If yes, continue. Time needed when U_f devotes two tapes to this task: O(m) [says Papadimitriou].
3. Next, U_f simulates the run $M(x)$. After each simulated step, U_f ticks off one mark from the "yardstick", thus keeping record of the simulated time. If the simulation of $M(x)$ reaches M's accepting state before the end of the yardstick is reached, U_f accepts; else it rejects. This can be done in time O($f(m)^2$) per simulation step. $f(2|x|+1)$ such simulation steps are needed, which makes a (generously measured) total of O($f(m)^3$) [use $|<M>| \geq |x|$].

Taking 1.–3. together, we see that U_f takes $O(f(2|x|+1)) + O(m) + O(f(m)^3) = O(f(m)^3)$ steps. By treating several symbols as one like in the proof of the linear speedup theorem, this $O(f(m)^3)$ can be sharpened to an exact $f(m)^3$. □

Proposition 10.2 $H_f \notin$ **TIME**($f(m)$).

Proof. A diagonalization argument! Assume that there exists a TM M_{Hf} that decides H_f in time $f(m)$, where m denotes length of input for M_{Hf}: $m = |<M>;x|$. From M_{Hf} we can construct a machine D_f with the following behavior:

$D_f(<M>)$ = if $M_{Hf}(<M>;<M>)$ = "yes" then "no" else "yes"

D_f can be constructed such that on input $<M>$, where $|<M>| \geq |<D_f>|$, it runs in at most the time as M_{Hf} on input $<M>;<M>$, that is, within time $f(2|<M>|+1)$.

Now consider the following implications:

$D_f(<D_f>)$ = "yes"
⇔ $M_{Hf}(<D_f>;<D_f>)$ = "no"
⇔ $<D_f>;<D_f> \notin H_f$
⇔ D_f does not accept $<D_f>$ within $f(2|<D_f>|+1)$ steps
⇔ $D_f(<D_f>)$ = "no",

a contradiction. □

From Propositions 10.1 and 10.2 we get

Proposition 10.3 (a time hierarchy theorem): If f is a proper complexity function, then **TIME**($f(m)$) $\subsetneq$ **TIME**($f(m)^3$).

Iterated application of this proposition leads to a hierarchy of properly included time complexity classes **TIME**($f(m)$) $\subsetneq$ **TIME**($f(m)^3$) $\subsetneq$ **TIME**($f(m)^9$)...

This time hierarchy is not the sharpest possible. Much tighter results are known, for instance **TIME**($f(m)$) $\subsetneq$ **TIME**($f(m)$ log ($f(m)$)). The proofs rely on simulations that are more sophisticated than our simulation in Proposition 10.1.

The following is a little helper observation that is often used in proofs of complexity theory without explicit mention. Essentially it states that for determining inclusion of time classes we may often ignore the time used for inputs of short length, and concentrate on the asymptotic growth of the time used in the two classes that we want to compare.

Proposition 10.4 If f, g are proper complexity functions and $g(n) \geq f(n) + 2$ n_0 for all $n > n_0$, and $f(n) \geq n$ and $g(n) \geq n$ for all n, then **TIME**($f(n)$) $\subseteq$ **TIME**($g(n)$).

Proof: Let M be a TM that decides some language L within time $f(n)$. Modify M into M' such that M' first scans the input up to a length n_0, and immediately accepts / rejects if the input

word x has length $\leq n_0$ (this requires that M' possesses a lookup table for the accepted/rejected inputs of length $\leq n_0$). If M' finds that the input is longer than n_0, it moves its input tape cursor back to the leftmost position and then continues like M from the start of $M(x)$. It is clear that M' decides L in time $g(n)$.

From Propositions 10.3 and 10.4 we can conclude

Proposition 10.5 $\mathbf{P} \subsetneq \mathbf{EXP}$.

Proof: We show first that **P** is a subset of $\mathbf{TIME}(2^n)$. Because any polynomial n^k will ultimately become smaller than 2^n, we have that **P** is a subset of $\mathbf{TIME}(2^n)$ [use Prop. 10.4]. By Proposition 10.3, $\mathbf{TIME}(2^n) \subsetneq \mathbf{TIME}((2^n)^3)$. Now conclude $\mathbf{P} \subseteq \mathbf{TIME}(2^n) \subsetneq \mathbf{TIME}((2^n)^3) = \mathbf{TIME}(2^{(3n)}) \subseteq \mathbf{EXP}$. □

Similar hierarchy theorems hold for space. We give a standard result without proof:

Proposition 10.6 For proper complexity functions f,

$$\mathbf{SPACE}(f(m)) \subsetneq \mathbf{SPACE}\,(f(m) \log (f(m))).$$

We now turn to the more difficult part of this Section, relations between complexity classes of different types. The following proposition collects some of the more obvious facts.

Proposition 10.7 For proper complexity functions f,

1. $\mathbf{SPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n))$ and $\mathbf{TIME}(f(n)) \subseteq \mathbf{NTIME}(f(n))$,
2. $\mathbf{NTIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$,
3. $\mathbf{NSPACE}(f(n)) \subseteq \bigcup_{k>0} \mathbf{TIME}(k^{\log n + f(n)})$.

Proof. 1. is trivial. 2. is almost trivial. Idea: Let $L \in \mathbf{NTIME}(f(n))$ be accepted by nondeterministic TM M in time $f(n)$. Then L can be decided by a deterministic TM K, which generates all branches of the "run-tree" of M (see Fig. 9.1) and checks whether one of them accepts. Because each branch has length at most $f(n)$, it can be dealt with by K using space $f(n)$. K can treat all branches one by one, re-using the same space of size $f(n)$.

The only not so obvious part is 3. Let $L \in \mathbf{NSPACE}(f(n))$, that is, there exists an m-tape nondeterministic TM M with input and output that accepts L within space $f(n)$. We have to find a deterministic TM N which decides L within time $k^{\log n + f(n)}$, where $n = |x|$ is the length of input words x presented to K, and k is some constant that depends on N and M.

The key observation is that because M works within space $f(n)$, the number of configurations that might appear in a computation of M on input x is bounded by a constant that depends only on $|x|$. We can determine an upper bound as follows. Each configuration of M that might turn up in the computation $M(x)$ is determined by the machine state $q \in K$, by the cursor position i of the input reading head, and for each of the m–2 internal tapes by two words u, v, where u is the tape inscription left from (and including) the cursor and v the inscription to the

cursor's right. Because none of the internal tape inscriptions may exceed $f(|x|)$ in total length, both u and v have length at most $f(|x|)$. Taking all these components of a configuration together, we find that we can bound their number a (generously) by
$a \le |K|\ (|x|+1)\ |\Sigma|^{2(m\text{-}2)f(|x|)} \le |x|\ (2\ |K|\ |\Sigma|)^{2(m\text{-}2)f(|x|)} =: |x|\ c_1{}^{f(|x|)} = c_1{}^{\log|x| + f(|x|)}$.

These maximally $c_1{}^{\log|x| + f(|x|)}$ many configurations that might possibly occur in a run $M(x)$ can be used as nodes of a directed *configuration graph* $G(M, x)$, which has an edge from configuration C to another configuration C' iff $C \overset{M}{\to} C'$. Obviously, $|x| \in L$ iff there is a path in $G(M, x)$ that runs from the configuration C_0, which corresponds to the starting configuration of $M(x)$, to some configuration whose state q is "yes". That is, we can reduce the problem of deciding whether $|x| \in L$ to an instance of REACHABILITY (with multiple goal nodes) on a graph with $c_1{}^{\log|x| + f(|x|)}$ nodes.

Technically, we can cast this reduction into a deterministic TM N in many ways, for instance by first letting N construct the adjacency matrix of $G(M, x)$. That can be done in at most time $|x| + f(|x|)$ per pair C, C' to be checked, that is in time $(|x| + f(|x|))\ c_1{}^{2(\log|x| + f(|x|))}$ altogether.

We know that REACHABILITY can be solved in polynomial time n^p by a deterministic TM, where n is the number of nodes of the graph. Thus N first uses at most $f(|x|)^2\ c_1{}^{2(\log|x| + f(|x|))}$ steps to construct $G(M, x)$ and then uses at most another $c_1{}^{p(\log|x| + f(|x|))}$ steps to solve REACHABILITY. By some generous bounding estimates using simple arithmetic inequalities we can find a k such that $f(|x|)^2\ c_1{}^{2(\log|x| + f(|x|))} + c_1{}^{p(\log|x| + f(|x|))} \le k^{\log|x| + f(|x|)}$. □

Combining the results from Proposition 10.7, we get the following sequence of inclusions:

(10.1) $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$

The only not immediately obvious inclusion is $\mathbf{NL} \subseteq \mathbf{P}$, which follows easily from

$$
\begin{aligned}
\mathbf{NL} &= \mathbf{NSPACE}(\log(n)) \\
&\subseteq \mathbf{TIME}(k^{\log(n)+\log(n)}) \text{ for some } k \quad \text{[use Prop. 10.7 (3)]} \\
&= \mathbf{TIME}((2^{\log(k)})^{2\log(n)}) \\
&= \mathbf{TIME}(n^{2\log(k)}) \qquad \text{[elementary arithmetics; we use log to base 2]} \\
&\subseteq \mathbf{P}
\end{aligned}
$$

We know from the space hierarchy theorem that $\mathbf{L} \subsetneq \mathbf{PSPACE}$, so at least one of the inclusions in Eq. (10.1) must be proper. However, it is not known which – in fact, *all* of the inclusions are suspected to be proper.

The "reachability method" we witnessed in the proof of Prop. 10.7 (3) is just the simple idea that all possible runs of a space-bounded nondeterministic TM can be represented in the (finite) configuration graph, and that the existence of accepting runs is thereby translated into instances of REACHABILITY. This basic method can be refined to yield some of the more precious results connecting different types of complexity classes.

We describe first a space-efficient implementation of REACHABILITY, which is also interesting for its own sake:

Proposition 10.8 (Savitch's theorem [1977], here basically copied from Papadimitriou). REACHABILITY $\in$ **SPACE**($\log^2 n$), where n refers to the number of graph nodes (not length of input word coding the graph!).

Proof. Here we understand REACHABILITY in the version given at the beginning of Section 8, that is, the question is whether in a graph with vertices v_1, ..., v_n, there is a path from v_1 to v_n. We identify as usual the vertices with the integers 1, ..., n.

Let G be a graph with n nodes, let x and y be nodes of G, and let $i \geq 0$. We say that the predicate PATH(x, y, i) holds if there is a path from x to y in G, of length at most 2^i.

Notice that because a path in G need have at most length n to solve REACHABILITY, we can solve that problem if we can compute whether PATH(x, y, $\lceil \log n \rceil$) for any two nodes in G.

We design a multiple-tape TM M with two working tapes besides the input tape, which decides whether PATH(1, n, $\lceil \log n \rceil$). G is presented to M at the input tape in the form of the adjacency matrix, turned into a word in a suitable fashion. Note that for multiple-tape machines with read-only input tapes, we defined their space complexity in terms of the space consumption on the working tapes only (Definition 3.9), so we don't have to think about the size of the codeword used to input G.

M starts its work by writing the triple (1, n, $\lceil \log n \rceil$) on its first working tape. In the following computation, the first working tape of M will typically contain several triples, of which (1, n, $\lceil \log n \rceil$) is (and remains) the leftmost. The other working tape serves as scratch space (size O(log n) is enough, says Papadimitriou) and we will not give a detailed account of it.

We now describe how M decides recursively whether PATH(x, y, i), for any triple (x, y, i) which appears as the rightmost triple on the first working tape (in the beginning, there is only one such triple, which is (1, n, $\lceil \log n \rceil$)). If $i = 0$, we can tell whether x and y are connected by a path of length at most $2^0 = 1$ by simply checking whether $x = y$ or whether $x \to y$. If $i \geq 1$, then we decide PATH(x, y, i) by the following recursive algorithm:

`For all nodes` z `test whether` PATH(x, z, i–1) `and` PATH(z, y, i–1).

We may conclude that PATH(x, y, i) is true if our test finds some such z – the reason for this is that a path from x to y of length at most 2^i exists iff there is a node z halfways on this path, that is, a node z with distances at most 2^{i-1} from x and from y.

To implement this idea in a space-efficient manner, we generate the binary indices of all nodes z, one after the other, re-using space. Once a new z is generated, we add a triple (x, z, i–1) to the first working tape, and start working on this problem recursively. If a negative answer to PATH(x, z, i–1) is found, we erase this triple and move on to the next z. If a positive answer is found, we replace the triple (x, z, i–1) by (z, y, i–1), and work on deciding whether PATH(z, y, i–1). If this is negative, we erase the triple and try the next z; if it is positive, we return a positive answer to the call PATH(x, y, i).

It is clear that this recursive algorithm decides PATH(1, n, $\lceil \log n \rceil$). The first working tape contains at any time $\lceil \log n \rceil$ or fewer triples, each of length at most 3 $\log n + c$, where c is a constant accounting for extra symbols (like semicolons or brackets) we might have used for organizing our triples. Thus the maximum tape consumption is $\lceil \log n \rceil$ (3 $\log n + c$) = $O(\log^2 n)$. This can be further linearly compressed to a tape consumption of $\log^2 n$, by using a version M' of M with a "blocked" alphabet as introduced in the linear speedup theorem. □

Because the simulation of a space-bound nondeterministic TM by a deterministic TM in decision tasks can be recast as a version of REACHABILITY, Savitch's theorem is a pillar for understanding nondeterministic space vs. deterministic space complexity. One immediate consequence of Savitch's theorem is

Proposition 10.9 NSPACE($f(n)$) $\subseteq$ **SPACE**($f(n)^2$) for any proper complexity function $f(n) \geq \log n$.

Proof: To simulate an $f(|x|)$-space-bound nondeterministic TM run $K(x)$ by a deterministic TM, we run Savitch's TM on the configuration graph of $K(x)$. Since the configuration graph has $k^{\log |x| + f(|x|)} \leq k^{2 f(|x|)} \leq k'^{f(|x|)}$ nodes, Savitch's TM needs $\log^2(k'^{f(|x|)}) = O(f(|x|)^2)$ space, or, possibly after alphabet compression, $f(|x|)^2$ space. □

This result immediately implies

Proposition 10.10 PSPACE = **NPSPACE**,

which in the sense of treating all polynomial complexities as equal means that nondeterminism does not give any advantage over determinism with respect to space – quite a contrast to what we believe nondeterminism yields with respect to time.

Savitch's theorem is also instrumental in the proof of the following strengthening of the obvious **TIME**($f(n)$) $\subseteq$ **SPACE**($f(n)$), which we give here without proof (a sketch of the proof can be found in the Papadimitriou book as exercise 7.4.17):

Proposition 10.11 [Hopcroft, Paul, Valiant 1975]: **TIME**($f(n)$) $\subseteq$ **SPACE**($f(n)$ / $\log(f(n))$) for proper complexity functions f.

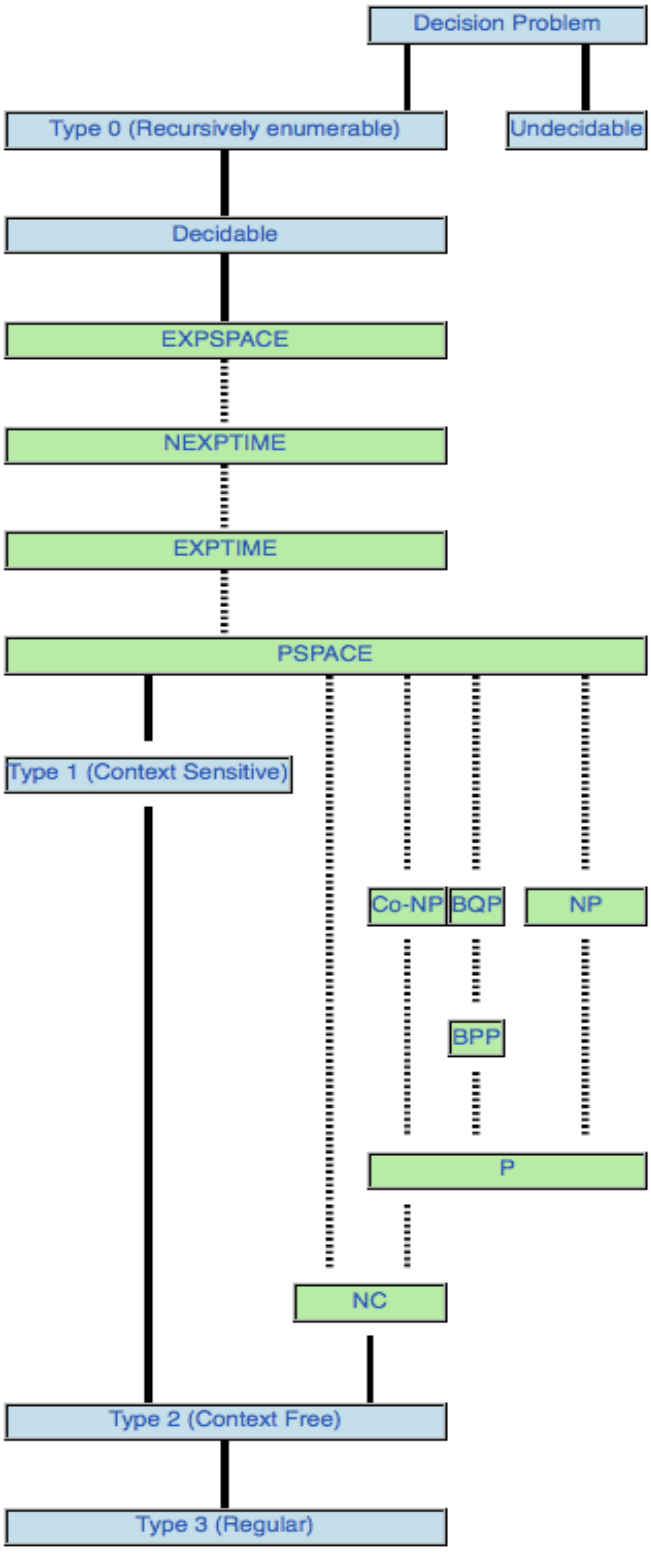
The "reachability method", that is, transforming a nondeterministic, space-bound TM computation into the graph problem REACHABILITY, is also used to prove the following important theorem (proof in Papadimitriou):

Proposition 10.12 For a proper complexity function $f(n) \geq \log n$, **NSPACE**($f(n)$) = **coNSPACE**($f(n)$).

Here **coNSPACE**($f(n)$) is the class $\{L \mid L^c \in \mathbf{NSPACE}(f(n))\}$.

Generally, for some complexity class **C**, one denotes by **coC** the class $\{L \mid L^c \in \mathbf{C}\}$. For deterministic time or space complexity classes **C**, one always has **C** = **coC**, because if any language L can be decided deterministically within some time- or space-bounded TM M_L, then L^c can be decided by the same machine within the same bounds by reversing the "yes" and "no" answers. Proposition 10.12 says that also nondeterministic space classes are closed

under complement. However, it is an open problem whether nondeterministic time classes are closed under complement. It is strongly believed that **NP ≠ coNP.** Again, like **PSPACE** = **NPSPACE**, **NSPACE**(*f*(*n*)) = **coNSPACE**(*f*(*n*)) reflects that nondeterminism appears to be more powerful in time than in space.



The figure (from http://en.wikipedia.org/wiki/Complexity_class) illustrates some known relationships between basic complexity classes, and the classes of the Chomsky hierarchy. Solid lines mean "properly contained in", dashed lines mean "contained in – and conjectured that the inclusion is proper". The naming of complexity classes here sometimes differs from ours. Explanation: EXPSPACE = exponential deterministic space; NEXPTIME = exponential nondeterministic time; EXPTIME = what we called EXP; BQP = this is something exotic, namely "bounded error quantum polynomial time", the class of decision problems solvable by a quantum computer in polynomial time, with an error probability of at most 1/3 for all instances; BPP = a little less exotic (in fact, practically useful), namely "**bounded-error probabilistic polynomial** time", the class of decision problems solvable by a probabilistic Turing machine in polynomial time, with an error probability of at most 1/3 for all instances; NC = "Nick's Class", see http://en.wikipedia.org/wiki/NC_(complexity) if you want to learn more (it measures "tractable on parallel computers" complexity).

11 Propositional logic (again!, but used now as a pillar for complexity theory)

11.1 A quick recap

Since you should already be familiar with propositional (= Boolean) logic, I start with a condensed recap of the basic definitions, following the notation of the Papadimitriou book.

Definition 11.1 (essentials of Boolean logic):

1. *Syntax* of Boolean expression: terms whose atoms are Boolean variables x_i, from which more complex terms are formed with the operators $\neg$, $\wedge$, $\vee$. Terms of the form x_i or $\neg x_i$ are *literals*. $\varphi \rightarrow \psi$ is shorthand for $(\neg\varphi) \vee \psi$. Precedence: $\neg$ before $\wedge$ before $\vee$ before $\rightarrow$.
2. *Semantics* of a Boolean expression: a truth assignment T is a mapping from (a finite subset of) the variables X to {**true**, **false**}. T *satisfies* a Boolean expression φ, written $T \models \varphi$, if (if $\varphi = x_i$ then $T(x_i)$ = **true**) and (if $\varphi = \neg\psi$ then $T \not\models \psi$) and (if $\varphi = \psi \wedge \psi'$ then $T \models \psi$ and $T \models \psi'$) and (if $\varphi = \psi \vee \psi'$ then $T \models \psi$ or $T \models \psi'$).
3. *Normal forms*: every Boolean expression is equivalent to one in conjunctive normal form (CNF), e.g. $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_4)$, and to one in disjunctive normal form (DNF), e.g. $x_1 \vee (x_2 \wedge \neg x_4)$. The disjunctions appearing in a CNF are called the *clauses* of the expression.
4. Numerous laws for equivalence, e.g.
   a. $(x_1 \vee (x_2 \wedge x_3)) \equiv (x_1 \vee x_2) \wedge (x_1 \vee x_3)$; $(x_1 \wedge (x_2 \vee x_3)) \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3)$ [distributivity]
   b. $\neg (x_1 \wedge x_2) \equiv (\neg x_1 \vee \neg x_2)$; $\neg (x_1 \vee x_2) \equiv (\neg x_1 \wedge \neg x_2)$ [deMorgan's laws]
   c. $\neg\neg x_1 \equiv x_1$ [double negation]
5. φ is *satisfiable* iff there exists T such that $T \models \varphi$. It is *valid* or *a tautology* iff it is satisfied by every T, written $\models \varphi$. It is *unsatisfiable* or a *contradiction* iff it is not satisfied by any T.
6. An n-ary *Boolean function* is a function f: {**true**, **false**}$^n \rightarrow$ {**true**, **false**}.
   a. Every Boolean expression represents a Boolean function in a natural way.
   b. Conversely, every n-ary Boolean function f can be expressed as a Boolean expression φ_f with variables $x_1, x_2, \ldots x_n$.
   c. One simple way to construct φ_f is to make it the disjunction of subexpressions that represent the truth value combinations that f makes true. For instance, consider the binary f defined by the following truth table:

x_1	x_2	f
true	**true**	**true**
true	**false**	**true**
false	**true**	**false**
false	**false**	**false**

   then the corresponding φ_f is

   $\varphi_f = (x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$.

Note that φ_f is in DNF. In a similar way, one can get another φ'_f for f in CNF, by creating a conjunction of disjunctions, each of which corresponding to one negated truth value combination that makes f false. For our example, we get

$\varphi'_f = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$.

7. A *Boolean circuit* (or simply circuit) is a directed graph whose nodes i are of sort $s_i \in \{\text{true}, \text{false}, \neg, \wedge, \vee, x_1, x_2, x_3, \ldots\}$. Nodes are called *gates*. The indegree (= number of ingoing edges) of a gate is 0 for gates of sort **true**, **false** or x_i; 1 for gates of sort $\neg$ and 2 for gates of sort $\wedge$ or $\vee$. Gates are numbered such that if (i, j) is an edge, then $i < j$. Gates of sort x_i are *input* gates. The gate with largest index n is the *output* gate of the circuit. Given a truth assignment T, truth values are propagated through the circuit C as truth values of its gates and result in a truth value of the output gate, which is the final *value* $T(C)$ of the circuit. See Figs. 11.1 and 1.2 for examples of circuits.
8. The notions of satisfiability etc. carry over from Boolean expressions to circuits in an obvious way.

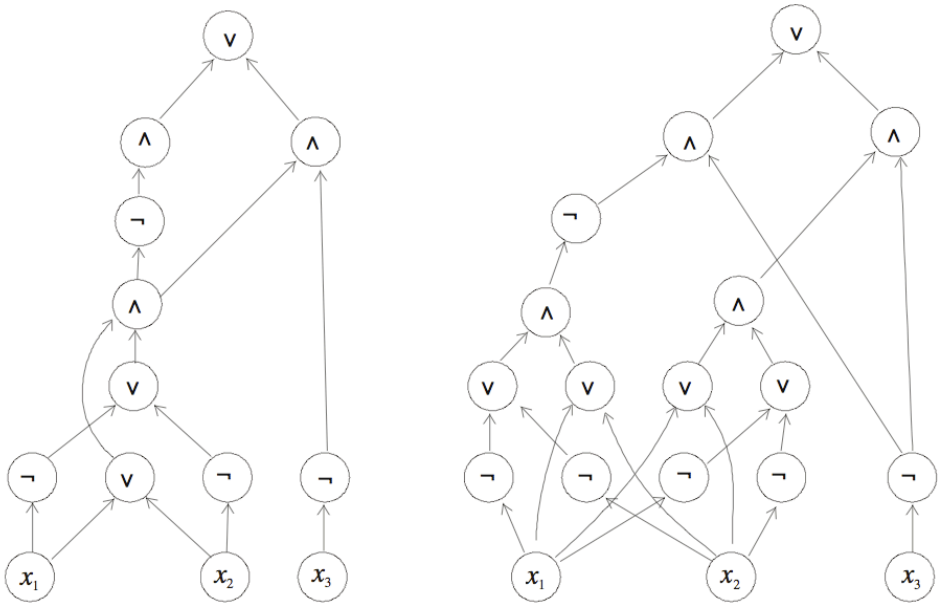


Figure 11.1: Two circuits, both representing the expression $(\neg x_3 \wedge \neg((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))) \vee (\neg x_3 \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))$

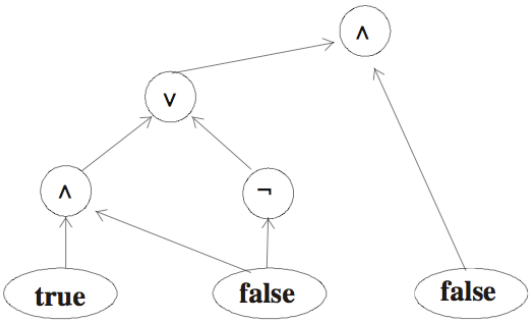


Figure 11.2: A circuit with no variable gates, representing the expression $(x_3 \wedge ((x_1 \wedge x_2) \vee \neg x_2))$ together with the assignment $T(x_1) =$ **true**, $T(x_2) = T(x_3) =$ **false**.

11.2 Some complexity considerations concerning Boolean logic

The computational problem of whether a given Boolean expression or a circuit is satisfiable will turn out to be an important pillar for complexity theory. This problem SATISFIABILITY or just SAT for short is the following: given a Boolean expression φ in CNF, is it satisfiable?

Posing this question from the perspective of complexity, we need to standardize how an expression is presented to a TM. As a tape alphabet we take $\Sigma = \{0, 1, \neg, \wedge, \vee, (,)\}$. It is clear that we can directly rewrite a Boolean expression in CNF using this alphabet if we code the variables x_i by the binary representation of i. The *length* of any Boolean expression φ is then the length of the symbol string that we obtain from the CNF of φ.

A deterministic TM can check whether a given truth assignment is satisfied by φ in time $O(m^2)$ [m denotes length of input]. Using truth assignments as polynomial witnesses, it is directly seen that SAT is in **NP**. It is not known whether SAT is in **P**. All known deterministic algorithms to decide SAT need exponential time; they boil down to checking all possible truth assignments.

A special case of SAT is by far easier to compute, namely, if all clauses of φ are *Horn clauses*, that is, they contain at most one positive literal. We say that φ is a Horn formula, and name the corresponding satisfiability problem HORNSAT. For instance,

$$(11.1) \qquad x_3 \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee \neg x_1 \vee x_2)$$

is a Horn formula. We will see how satisfiability of (11.1) can be checked without considering all possible truth assignments. The clauses that are not purely negative (like the second clause in (11.1) is) can be re-written as implications. (11.1) would become

$$(11.2) \qquad (\mathbf{true} \rightarrow x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \wedge x_1 \rightarrow x_2)$$

Note that (11.2) is not actually a Boolean expression but could be transformed into one if one replaces **true** by $(y \vee \neg y)$. Now, to check whether (11.2) is satisfiable, we proceed in two stages. In the initial stage we consider only the clauses that are implications – here, the first and third clause. We start from the all-false truth assignment T_0, which we incrementally change to T_1, T_2, ... by going through the implication clauses repeatedly from left to right. Each T_{i+1} contains exactly one more positive assignment than T_i.

Assume that we have an assignment T_{i-1}. We inspect all implication clauses from left to right. There are two possibilities.

1. If all implications are satisfied by T_{i-1}, we leave the first stage and call T_i the *critical assignment* T_{critical}.
2. If we find some implication of the form $(\mathbf{true} \rightarrow x_j)$ or $(x_{n1} \wedge \ldots \wedge x_{nk} \rightarrow x_j)$, and this implication is not satisfied by T_{i-1}, (which implies that $T_{i-1}(x_j) = \mathbf{false}$), we change T_{i-1} to T_i by flipping the value for x_j from **false** to **true**.

Because in each update from T_{i-1} to T_i we increase the number of positive assignements, this first stage must terminate. The assignment T_{critical} is minimal in the sense that if T' is another assignment that satisfies the implications in φ, then T' assigns **true** to all variables that T_{critical} assigns **true** to, for which we write $T_{\text{critical}} \subseteq T'$. Because, if not $T_{\text{critical}} \subseteq T'$, consider the first i

where T_i ceases to be a subset of T'. The clause that caused a flip in T_i cannot be satisfied by T'.

In our example (11.2), we get

T_0 = (**false**, **false**, **false**)
T_1 = (**false**, **false**, **true**) [after considering (**true** $\rightarrow x_3$)]

Because T_1 satisfies all implications, we put $T_{\text{critical}} = T_1$.

The truth assignment T_{critical} thus obtained satisfies all implication clauses in φ, and every truth assignment which satisfies all implication clauses in φ must have at least the positive assignments of T_{critical}. It remains to be checked in a second stage whether T_{critical} satisfies the (possibly) remaining all-negative clauses of φ. In our case, T_{critical} does satisfy the clause ($\neg x_1 \vee \neg x_2$), so we can conclude that φ is satisfiable. The entire procedure is apparently polynomial in the length of φ, so we have HORNSAT $\in$ **P**.

SAT is one (important) example of a problem which is in **NP**, and which can be turned into a tractable **P** problem by requiring a special restriction, in our case, Horn clauses. It is a very general phenomenon that intractable problems have tractable special cases, and often indeed the practical task of a program designer / engineer / computer scientist is to find a special case of an intractable problem which can be practically solved – this being *one* way to deal with intractability.

It is intuitively clear that the problem of whether a given circuit is satisfiable, called CIRCUIT SAT, is closely related to SAT. It is easy to see that CIRCUIT SAT is also in **NP** (argument: guess a truth value assignment for the input gates, then check whether the circuit comes out as **true**). Finally consider the problem of whether a given truth assignment satisfies a given circuit, known as CIRCUIT VALUE. Fig. 11.2 shows an example. Obviously an instance of CIRCUIT VALUE can be computed by going through the circuit once, which can be done in polynomial time.

We collect our findings:

Proposition 11.1 SAT $\in$ **NP**, HORNSAT $\in$ **P**, CIRCUIT SAT $\in$ **NP**, CIRCUIT VALUE $\in$ **P**.

Comparing the two circuits in Fig. 11.1, which both represent the same Boolean expression, we see that the left circuit is economical in that it uses only a single copy of the clauses ($x_1 \vee x_2$) and of ($\neg x_1 \vee \neg x_2$) which appear several times in the original expression. This raises the question whether in principle the construction of "economical" circuits could lead us away from the exponential complexity of SAT. However, this is not so:

Proposition 11.2 For any $n > 1$ there is an n-ary Boolean function f such that no circuit with at most $m = 2^n/2n$ gates can compute it.

Proof (sketch): There are 2^{2^n} different n-ary Boolean functions (why?). But how many circuits with at most m gates exist? A rough estimate yields that there are less than $((n+5)\, m^2)^m$ such circuits [full argument in the Papadimitriou book; idea: there are n+5 sorts,

each gate is determined by sort and at most 2 input arrows (that is, at most m^2 choices for input arrows)]. Elementary arithmetics shows that $((n+5)\ m^2)^m < 2^{2^n}$ if $n > 1$ and $m = 2^n/2n$.

12 Reductions and Completeness, and P =? NP

In Section 10 we considered inclusion relations between complexity classes, that is, we asked questions like "what kind of problems can we solve when we allow so-and-so much time / space?". The underlying intuition was that with more resources one can solve "harder" problems. In this section we will show how one can rigorously compare the "hardness" of two *problems* (and not of two complexity classes as before).

12.1 Reductions

The tool for comparing two problems is the concept of *reduction*. Intuitively, a problem B can be reduced to another problem A, if

1. we have an algorithm (a TM) N that solves A,
2. we have a computable function R to "translate" instances x of B into instances $y = R(x)$ of A that have the same yes/no answer.

"Translation" here means: R is a TM that takes the description x of an instance of B as an input argument and yields an instance $y = R(x)$ of A. And the overall computation $N(R(x))$ must solve the problem B. Figure 12.1 sketches this idea.

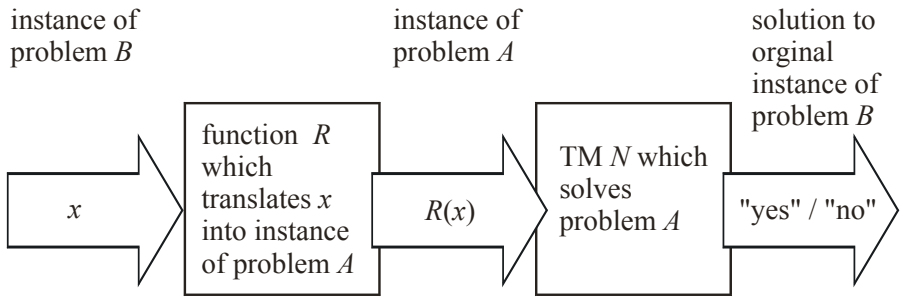


Fig. 12.1: Reduction of problem B to problem A

Now we have a sharper intiution of the relative hardness of problems: we would say, problem A is at least as hard as problem B, if we can translate B into A as in Fig. 12.1. However, there is a caveat: The function R should not itself be so powerful as to significantly contribute to the solution of B. To illustrate this point, think of B as the TSP(D) problem, which (we think) is hard, and of A as REACHABILITY, which we know is easy. Now if R were an algorithm that first solves TSP(D), then in the case of a "yes" answer passes to REACHABILITY as $R(x)$ a two-node graph with two connected nodes and in the case of a "no" answer a two-node graph with two unconnected nodes, then obviously R would qualify as a reduction in the sense of Fig. 12.1. This example teaches us to restrict the computational power of R. It must be less than the computational complexity of the problem we are reducing. For our purposes, it will be enough to consider *polynomial-time* reductions R when we are reducing some problem B to

A, where A is in **NP**; and to use *logarithmic-space* reductions when the target problem A is in **P**.[50]

Definition 12.1 A language L_B is *ptime-reducible* (or *logspace-reducible,* respectively) to another language L_A, if there is a word-to-word function R computable by a deterministic TM M_R in polynomial time (in logarithmic space, respectively) such that for all inputs x the following is true: $x \in L_B$ iff $R(x) \in L_A$. R is called a *ptime-reduction* (or *logspace-reduction*) from L_B to L_A.

When the context is clear ($A \in$ **NP** vs. $A \in$ **P**), we may omit the qualification *ptime/logspace* and simply speak of reductions.

Reducibility is transitive: if L_C is ptime/logspace reducible to L_B and L_B to L_A, then L_C is ptime/logspace reducible to L_A. (exercise!).

We now inspect a simple, instructive example of a reduction. Let HAMILTONIAN PATH be the problem to decide whether in a given directed graph there exists a path p which visits every node exactly once. This "smells" like a difficult combinatorial search problem. Indeed, one can show that HAMILTONIAN PATH is essentially (that is, up to polynomial-time transformations) as difficult as SAT: each of the two problems can be reduced to the other. We show only the easy direction:

Proposition 12.1 HAMILTONIAN PATH is ptime-reducible to SAT.

Proof. We must find a polynomial-time reduction R which transforms the question, "is there a Hamiltonian path p in graph G?", into a question of the kind "is the CNF Boolean formula φ satisfiable?". The idea is to construct $\varphi = R(G)$ such that it describes the existence of a Hamiltonian path. Suppose that G has n nodes 1, 2, ..., n. We construct φ with n^2 Boolean variables x_{ij}. Informally, variable x_{ij} denotes the statement "p is a sequence of graph nodes and node j appears at the i-th position in it" (which may be true or false). We now give a list of requirements that are clearly equivalent to stating that p is a Hamiltonian path, and show how each requirement can be described by certain clauses of φ.

- Requirement 1: "Every node j of G must appear in p". For each $1 \le j \le n$, this is expressed by a clause $(x_{1j} \vee x_{2j} \vee ... \vee x_{nj})$ of φ.
- Requirement 2: "No node may appear at two different positions i and k in p". To ensure this, for every pair $i \neq k$, and every node j, we add the clause $(\neg x_{ij} \vee \neg x_{kj})$ to φ.
- Requirement 3: "For every position i of p there must be a node j": add the clause $(x_{i1} \vee x_{i2} \vee ... \vee x_{in})$ for every i.
- Requirement 4: "No two nodes j, k can appear at the i-th position of p": for every i and pair of nodes $j \neq k$, add the clause $(\neg x_{ij} \vee \neg x_{ik})$.
- Requirement 5: "If (j, k) is not an edge in G, then node k cannot be the direct successor of node j in p": for every position $1 \le i \le n-1$, and every (j, k) which is not an edge in G, add the clause $(\neg x_{ij} \vee \neg x_{(i+1)k})$.

It is clear that if there exists a truth assignment that makes φ true, then there exists a Hamiltonian path, and vice versa. To see this, note that requirements 1 – 4 boil down to

[50] This choice is standard in the literature. Papadimitriou differs from the standards in his book in that he uses logspace reductions also for problems in **NP**.

stating that there exists some bijection between n positions and the n nodes, and requirement 5 enforces that any such bijection is a path that is in agreement with the existing graph arcs.

Note that φ is in CNF. It is not difficult to construct a TM for R which on input G constructs all these clauses and writes them on the output tape, separated by $\wedge$. Furthermore, there are only polynomially many such clauses, and computing each clause clearly requires only polynomial time, so the overall polynomial time requirement for R is easily satisfied. (Of course this is a dangerously sloppy argument...). □

Expressing HAMILTONIAN PATH in terms of SAT is easy because Boolean logic is all about expressing combinations of discrete facts. We will soon see that the expressive powers of Boolean logic and SAT are much greater than this simple example demonstrates: in fact, *every* problem in **NP** can be reduced to SAT.

Before we come to that, we will give another simple example of a reduction, namely, from CIRCUIT SAT to SAT. Recall from Section 11 that a circuit is a directed graph whose n nodes ("gates") i are of sort $s_i \in \{\textbf{true}, \textbf{false}, \neg, \wedge, \vee, x_1, x_2, x_3, \ldots x_k\}$; that gates are numbered such that if (i, j) is an edge, then $i < j$; that gates of sort x_i are *input* gates; that gate n is the *output* gate of the circuit; and that given a truth assignment T, truth values are propagated through the circuit C as truth values of its gates and result in a truth value of the output gate, which is the final value $T(C)$ of the circuit. The problem CIRCUIT SAT is to decide whether a given circuit has a satisfying truth assignment.

Proposition 12.2 CIRCUIT SAT ptime-reduces to SAT.

Proof. For a given circuit with gates $1 \leq i \leq n$, and k Boolean input variables $x_1, \ldots, x_k$, we have to construct a Boolean expression φ in CNF such that the circuit is satisfiable iff φ is satisfiable. We build φ from n Boolean variables y_i (which correspond to the gates of the circuit), plus variables $x_1, x_2, \ldots x_k$ in the following way:

- Start with empty φ.
- For $i = 1, \ldots, n$, add clauses to φ, to encode the circuit, as follows:
  - If i is a **true** gate, add the clause y_i. It is clear that only a positive truth assignment to y_i can satisfy this clause, so y_i reflects a **true** gate.
  - If i is a **false** gate, add the clause $\neg y_i$. It is clear that only a negative truth assignment to y_i can satisfy this clause, so y_i reflects a **false** gate.
  - If i is of sort x_j, add the clauses $(\neg y_i \vee x_j)$ and $(y_i \vee \neg x_j)$. It is clear that any truth assignment to y_i which satisfies these clauses must assign the same truth value to y_i and to x_j.
  - If i is a NOT-gate, with predecessor j, add the clauses $(\neg y_i \vee \neg y_j)$ and $(y_i \vee y_j)$. It is clear that $y_i \Leftrightarrow \neg y_j$.
  - If i is an AND-gate with predecessors j and j', add the clauses $(\neg y_i \vee y_j)$, $(\neg y_i \vee y_{j'})$ and $(y_i \vee \neg y_j \vee \neg y_{j'})$. Then in any truth assignment T which satisfies these three clauses, $T(y_i)$ must be $T(y_j) \wedge T(y_{j'})$.
  - If i is an OR-gate with predecessors j and j', add the clauses $(y_i \vee \neg y_j)$, $(y_i \vee \neg y_{j'})$ and $(\neg y_i \vee y_j \vee y_{j'})$. Then in any truth assignment T which satisfies these three clauses, $T(y_i)$ must be $T(y_j) \vee T(y_{j'})$.
  - Finally, add the clause y_n to ensure that the entire circuit must be satisfied.

It is clear from this construction that the circuit can be satisfied if and only if φ can be satisfied, and that φ can be constructed in polynomial time in the size of (a reasonable coding of) the circuit. □

12.2 Completeness, especially NP-completeness

Overview. We have seen in the previous section that some problems can be conveniently reduced to SAT. All we had to do was to "describe" the original problem through a suitably constructed Boolean formula. In this section we'll learn that in fact *every* problem in **NP** can be "rephrased" as a SAT problem. The basic argument runs through the following three steps:

1. A problem in NP can be described by the nondeterministic TM by which it is solved. This TM has an accepting run of polynomial time.
2. An accepting, polynomial-time run of a nondeterministic TM can be described as a polynomial-length path in the configuration graph of the TM.
3. The existence of such paths can be coded in a Boolean expression φ in a way that is in principle similar to what we did in the reduction of HAMILTONIAN PATH to SAT.

The ptime-reducibility of any NP-problem to SAT is one of the pillars of today's complexity theory. It was first shown by Stephen Cook in 1971. Before we embark on the proof, we introduce some terminology.

Definition 12.2 Let C be a complexity class and L a language. Then L is *C-hard with respect to ptime/logspace reductions* iff every language $L' \in C$ can be ptime- (logspace-, respectively) reduced to L. If L is C-hard and furthermore L is itself in C, we say that L is *C-complete* with respect to ptime/logspace reductions.

One application of the completeness concept is that it can be used to show that two complexity classes C and C' coincide. This is important in advanced complexity theory, where one defines complexity classes by different mechanisms, and often does not easily know whether a class C defined in one way is actually identical to another already studied class C'. To see how completeness helps here, we first introduce the concept of *closure under reduction*: a complexity class C is closed under reduction if, whenever L is reducible to L' and $L' \in C$, then also L is in C. The classes **P** and **NP** and **EXP** are of this kind (for logspace / ptime / ptime reductions, respectively). Now it holds that if both C and C' are closed under reductions, and there is a language L which is complete for both C and C', then $C = C'$. The argument runs as follows: if L^* is some language in C, then L^* can be reduced to L and hence (because C' is closed under reduction) $L^* \in C'$. That is, $C \subseteq C'$. The other direction follows by symmetry.

We will consider only C = **NP** in the remainder of our course. An **NP**-complete language captures, in a sense, "the essence" of the class **NP**: If we have a decision algorithm for L (that is, a nondeterministic TM that accepts L in polynomial time), and we want to find a decision algorithm for some other language $L' \in$ **NP**, then all we have to do is to reduce L' to L. Then the TM for L can be re-used to decide L'.

We will presently show that SAT is **NP**-complete. By reducing SAT to other languages, and these other languages to yet others, a host of languages have been found that are **NP**-complete, too. In a sense, SAT is the mother of all **NP**-complete languages.

However, it is not at all evident that **NP**-complete problems should exist at all! Thus it comes as no surprise that showing SAT to be **NP**-complete requires some work.

Theorem 12.3 (Cook 1971): SAT is **NP**-complete.

Proof. We follow the proof given in the Papadimitriou book. We already know from Section 11 that SAT is in **NP**. Therefore, all we have to show is that SAT is **NP**-hard, that is, if L is some language in **NP**, we can find a reduction R from L to SAT. Of course showing hardness is the hard part!

So, let $L \in \mathbf{NP}$. We know from Prop. 11.2 that CIRCUIT SAT reduces to SAT, and we know that reducibility is transitive. Therefore, it suffices to show that L can be reduced to CIRCUIT SAT. Let M be a nondeterministic TM (K, Σ, Δ, s) which accepts L in polynomial time, and $x \in \Sigma^*$ a candidate word. We may assume that M is a single-tape TM. We have to construct a circuit $R(x)$ which is satisfiable iff $x \in L$. For this construction, we may use our knowledge of M. (Furthermore, R must be computable in polynomial time – but that is a relatively easy side-issue, which we will deal with at the end). Several ideas have to be combined to achieve the construction of $R(x)$. All these ideas aim at a standardized representation of the workings of M, wherein the actions of M can be described by 0-1-valued (i.e. Boolean) variables. Once we have arrived at a standardized "0-1-description" of how M works, a way to cast this as a Boolean circuit is immediately found.

Idea 1: Describe the working of a *deterministic* polynomial-time TM N by a *computation table*. (We will later transform the nondeterministic working of M into a deterministic decision problem). A computation table is a standardized representation of runs of the TM in a table form.

Consider the general case of some language L' which is decided by a deterministic single-tape TM $N = (K, \Sigma, \delta, s)$ in time $|x|^k$. The computation table A of a run $N(x)$ is a matrix of size $|x|^{k+1}$ by $|x|^{k+1}$ and in each row shows one configuration of the run. Figure 12.1. shows the computation table of a run $N(0110)$ for some such N that runs in quadratic time.

▷	0_s	1	1	0	⊔	⊔	⊔	⊔	... (repeat up to column 4^3) ...	⊔
▷	▷	1_{q0}	1	0	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	1	1_{q0}	0	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	1	1	0_{q0}	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	1	1	0	$⊔_{q0}$	⊔	⊔	⊔	... ...	⊔
▷	▷	1	1	$0_{q'0}$	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	1	1_q	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	1_q	1	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	$▷_q$	1	1	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	1_s	1	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	▷	1_{q1}	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	▷	1	$⊔_{q1}$	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	▷	$1_{q'1}$	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	$▷_q$	⊔	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	▷	$⊔_s$	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	▷	$⊔_{yes}$	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	▷	yes	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	▷	yes	⊔	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	yes	⊔	⊔	⊔	⊔	⊔	⊔	⊔	... ...	⊔
▷	yes	⊔	⊔	⊔	⊔	⊔	⊔	⊔	... ...	⊔
					...					
				(repeat up to row 4^3)						
					...					
▷	yes	⊔	⊔	⊔	⊔	⊔	⊔	⊔	... ...	⊔

Figure 12.1
A computation table for a run $N(0110)$ (after Papadimitriou)

How a computation table is made in principle should become clear from the figure. A few details must be explained:

1. The symbol set Γ used to annotate the cells of the table is $\Gamma = \{▷, ⊔\} \cup \Sigma \cup (\{▷, ⊔\} \cup \Sigma) \times K \cup \{\text{yes, no}\}$, that is, N's standard tape symbols, plus these standard symbols indexed by N's states, plus two special new symbols "yes" and "no".
2. A table size of $|x|^k$ by $|x|^k$ is enough to accomodate all configurations (both in number and in length) that may occur in a run of $N(x)$, if N runs within time $|x|^k$.
3. We will later need computation tables where the leftmost "▷" is never visited by the cursor. This can be achieved by modifying N such that (i) it is started not from the leftmost "▷" but from the first input symbol, and (ii) if N visits the leftmost "▷", the corresponding row is simply dropped from the table.
4. We will also need that the rightmost ⊔ is never visited by the cursor, and that the machine halts before the last line in the table. This can be ensured by increasing k: a TM that runs within time $|x|^k$ also runs within time $|x|^{k+1}$. The TM in Fig. 12.1. actually runs in time $|x|^2$, however, the table has been written in size $|x|^3$ by $|x|^3$.
5. After the original machine halts (with "yes" or "no" because it decides a language), a special symbol "yes" (or "no") is written in the corresponding row of the table, and (by an extension of N) subsequently shifted to the left until it reaches the second position. (Again, by using table size $k + 1$ it is ensured that there are enough rows left, after the TM has ended in its "yes" or "no" state, to propagate the special "yes" or "no" symbol to the left margin of the table.

6. When the "yes" (or "no") symbol has reached the second position, the remaining rows of the table are filled with copies of this final configuration.

We say that *the table A is accepting* if $A(|x|^{k+1}, 2) =$ yes. It should be clear that A is accepting iff $x \in L'$.

Idea 2: encode the "local transition function" of a computation table by a Boolean circuit C.

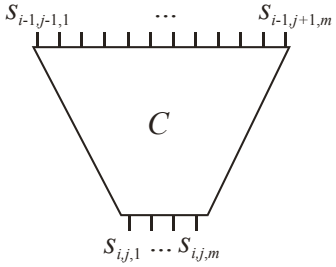
First I will explain what the "local transition function" is. For $2 \le j \le |x|^{k+1} - 1$ and $i > 1$, the value of the cell $A(i, j)$ depends only on the values $A(i-1, j-1)$, $A(i-1, j)$, $A(i-1, j+1)$ of the three adjacent cells in the previous row (as in a 1-dimensional cellular automaton, if you are familiar with them). (The leftmost and rightmost cells of each row simply are constant $\triangleright$ or $\sqcup$, respectively, thanks to our special agreement spelled out in points 2. and 3. above). That is, the changes in A from each row to the next are governed by a function f: $\Gamma^3 \to \Gamma$.

Encode the symbols from Γ as binary numbers of length $m = \lceil \log |\Gamma| \rceil$. We introduce the notation $b_{i,j,k}$ for the k-th binary digit of the binary code for the symbol in cell $A(i, j)$. Then the three table symbols $A(i-1, j-1)$, $A(i-1, j)$, $A(i-1, j+1)$ become a 0-1-sequence of length $3m$, and f can be interpreted as a function f: $\{0,1\}^{3m} \to \{0,1\}^m$ (on arguments which do not code symbol triples from the table, f can be defined arbitrarily). We know that such a function can be realized by some Boolean circuit C with $3m$ inputs and m outputs. For each $2 \le j \le |x|^{k+1} - 1$ and $i > 1$, we have therefore

$$C(s_{i-1,j-1,1}\ , s_{i-1,j-1,2}\ , \ldots, s_{i-1,j+1,m}) = s_{i,j,1}\ , \ldots, s_{i,j,m}\ .$$

Graphically, we might wish to represent the circuit C as in Figure 11.2 (a).

(a)



(b)

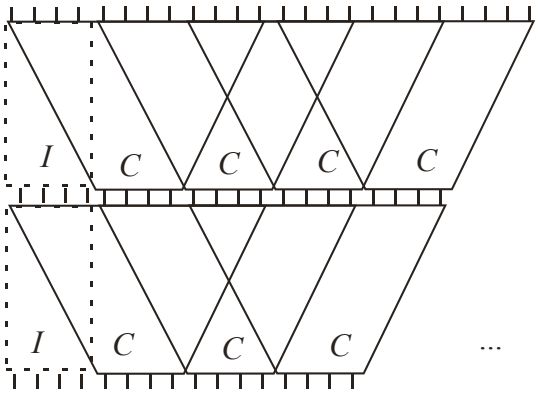


Figure 12.2: (a) The circuit C for f. (b) How to cast the computation tables of computations $N(x)$ (for given length $|x|$) as a complex circuit made from copies of C for each local "triple transition" of A. The upper leftmost portion of the overall complex circuit is shown. The dashed rectangles I mark "identity circuits" which simply copy their m inputs to their m outputs.

Notice that C depends only on N, not on input queries x. The construction of C does not enter the runtime of R!

Idea 3: Code all runs of N on inputs of a given length l as a circuit made from copies of C.

Consider the computation tables A that arise when N is started on inputs x of some given length l. They are all of size l^{k+1} by l^{k+1}. Take $(l^{k+1}\text{-}2)\ l^{k+1}$ copies of C and "stack them together" into a composite circuit as indicated in Figure 11.2. (b), stuffed with l^{k+1} "identity circuits" on the left side and another l^{k+1} "identity circuits" on the right side. To make this composite circuit simulate the run $N(x)$, first code the input x of N as a binary string of length $m\ |x|^{k+1}$, which becomes the input to this composite circuit. This input is passed through the composite circuit and leads to a (coded) "yes" or "no" in the second m output gates, according to whether $x \in L'$. We are free to agree on a binary coding for "yes" which has a leading 1, and on a binary coding for "no" which has a leading 0. By assigning the m+1-st output gate of the composite circuit as its relevant output gate, we can read off from this gate the decision for $x \in L'$.

In fact, what we have done so far in ideas 1 to 3 is to reduce the problem of deciding $L' \in \mathbf{P}$ to CIRCUIT VALUE (remember, this is the problem of whether a given truth assignment satisfies a given circuit): given x (and knowing N), we have constructed a circuit which yields 1 iff $x \in L'$. It is a tedious exercise to show that the construction of this composite circuit can be done in logarithmic space... So, we briefly pause in our multi-step proof of Cooks theorem and note, as an interesting side-result:

Proposition 12.4 CIRCUIT VALUE is **P**-complete with respect to logspace-reductions.

Now we resume our proof of Cook's theorem. Remember that we want to reduce decidability of some language $L \in \mathbf{NP}$ to CIRCUIT SAT, and not of some language $L' \in \mathbf{P}$ to CIRCUIT VALUE! However, we have done most of the work. Remember that M is a nondeterministic single-tape TM (K, Σ, Δ, s) which accepts L in polynomial time.

Idea 4: Transform M into a nondeterministic TM M' which at each step has exactly two nondeterministic choices.

I give here only the basic idea without details, by way of a diagram: if the original M for a given read symbol and state has, say, 5 possible actions, spread this 5-fold alternative into a little binary alternative-action tree as in Figure 12.3:

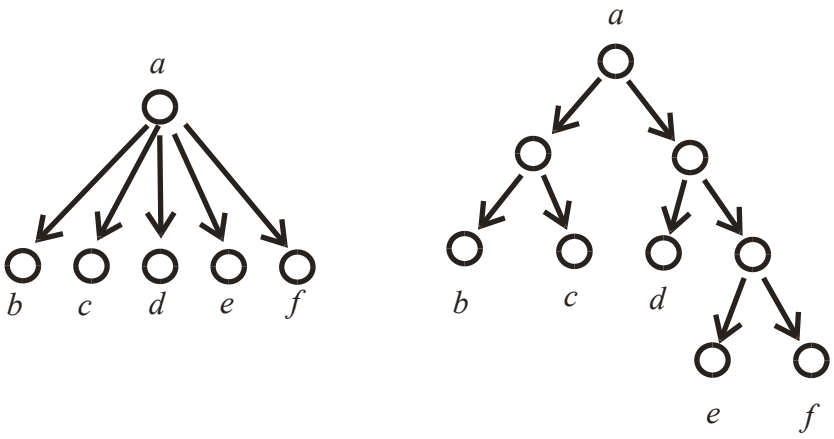


Figure 12.3: Reducing the degree of nondeterminism.

(Exercise: show that if M is polynomial-time, then so is M').

Idea 5: Transform the nondeterministic binary-alternative TM M' into a deterministic TM with an extra input specifying the sequence of action alternatives.

Our TM M' runs in time $|x|^k$. At each time, there is a binary choice of alternatives; call them 0 and 1. A given run with a given sequence of choices is thus determined by a bitstring c_1, c_2, ..., $c_{|x|^k-1}$ of length $|x|^k - 1$, which represents the choices. Repeat the construction of a computation table and a composite circuit for runs of M' on input of some length l, as we did in ideas 1 – 3, with one additional gadget. The circuit C is equipped with one extra input to account for the choice arguments c_i: if this argument is 1, the new circuit C' carries out the local function as conforming to the first action alternative; if it is 0, conforming to the other alternative. Each choice variable c_i becomes a Boolean variable, which is fed to all circuits C' in the i-th row of the new composite circuit. Figure 12.4 shows one such circuit picked from the i-th row:

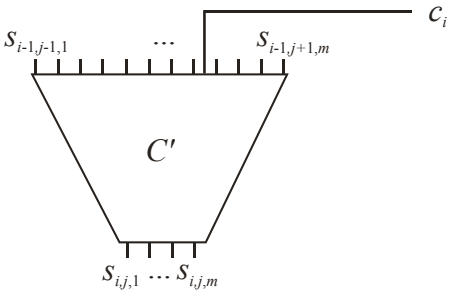


Figure 12.4: An augmented local circuit C' taken from the i-th row of the new composite circuit.

To decide whether $x \in L$, we construct our augmented composite circuit and write our encoding of x into its input gates on the first row. Then by construction it holds that $x \in L$ iff there exists a choice sequence c_1, c_2, ..., $c_{|x|^k-1}$ such that the composite circuit yields a 1 answer. But this is just another way of asking whether the composite circuit can be satisfied by some choice of c_1, c_2, ..., $c_{|x|^k-1}$, that is, by some truth assignment for these variables. Again, by a bold hand-waving argument we declare that the construction of the composite circuit from x and M can be done in polynomial time. Thus we have finally reduced the question of deciding L to CIRCUIT SAT and hence to SAT, and have proven Cook's theorem. □

We now discuss why NP-completeness is such an important and pervasive issue. I strongly lean on the standard reference book on NP-complete problems, "Computers and Intractability" by M. R. Garey and D. S. Johnson (included in the course references at the IRC).

The first observation that *very many* problems are NP-complete. *Hundreds* of problems have been shown to have this property! The majority of them come from the following fields:

- Graph theory
  - Examples: TSP(D), HAMILTONIAN PATH
  - Example: VERTEX COVER: given a graph G and a positive integer $K \leq |V|$, is there a vertex cover of size K or less for G, that is, a subset V' of V of size at most K, such that for each edge (x, y) of G, x or y is in V' ?
- Networks and flows
  - Example: MAX CUT, that is, decide whether in a given network there is a cut with capacity exceeding some given threshold
- Sets and partitionings
  - Example 1: 3-DIMENSIONAL MATCHING, aka 3DM: given a set $M \subseteq W \times X \times Y$, where W and X and Y are disjoint and $|W| = |X| = |Y| = q$, is there a subset M' of M of size q, such that each element of W and X and Y appears exactly once in M' ?
  - Example 2: PARTITION, that is, given a finite set A and an integer "size" $s(a) > 0$ for every element $a \in A$, decide whether A can be divided up into two disjoint subsets of identical size (where size of a set is sum of element sizes)
- Propositional logic
  - Example 1: SAT
  - Example 2: 3SAT: Like SAT, with the additional requirement that all clauses in the Boolean CNF expression have exactly three literals.
- Game theory
  - Examples: many generalized versions of e.g. checkers and Go, where the question is whether player 1 has a winning strategy from a given starting configuration, are known to be NP-hard
- Automata and languages
  - Example: Given a single-tape nondeterministic TM M which may use only the tape portion where the input is written on (including two end-marking extra symbols), and given an input x, does M accept x? This is known to be NP-hard.
- Program optimization
- Algebra and number theory
  - Example: QUADRATIC DIOPHANTINE EQUATIONS: given positive integers a, b, c, are there positive integers x and y such that $ax^2 + by = c$?
- Sequencing and scheduling
  - Examples: many problems concerning the construction of timetables and schedules (for workers, processors, trucks...) under time or other constraints
- Data storage and retrieval
  - Example: BIN PACKING: Given a finite set U of items, a size $s(u)$ for every item, a positive integer bin capacity B, a number K of bins, can the items be packed into the bins, that is, is there a partition of U into K subsets U_i ($i = 1, ..., K$) such that the sum of the sizes of items in each U_i is B or less?

The problem types collected in this list cover a very large variety of problems that one is likely to encounter in the daily practice of a computer scientist, especially if one considers the fact that many problems can easily be re-cast as graph, flow, or logical problems. If one is confronted with a novel problem Π (say, in VLSI design or fleet scheduling or code optimization...), which appears to be difficult at first sight, then one can do basically two things:

1. try to find a solution algorithm anyway, or
2. try to prove that the problem is NP-complete (or NP-hard).

Often, option 2 is the more intelligent way to proceed. Proving NP-completeness of a given problem is often not too difficult! This is illustrated by the very fact that hundreds of problems have already been shown to be NP-complete. The most common strategy for proving NP-hardness or NP-completeness is simple:

1. find a problem Π' which is already known to be NP-complete, and which looks similar to Π,
2. reduce Π' to Π.

This works because the following fact:

Proposition 12.5 If an NP-complete problem Π' can be reduced to a problem Π, then Π is NP-hard.

This follows immediately from the definition of NP-hardness and NP-completeness.

Finding a reduction from Π' to Π only establishes NP-hardness of Π. To show NP-completeness, in addition one has to verify that Π lies in NP, that is, that correctness of a guessed solution can be verified in polynomial time. This is often (but not always) relatively simple.

In fact, almost all of the known NP-complete or NP-hard problems have been dervived by reduction from previously known such problems. The "mother of all NP-complete problems" is SAT. This problem thus stands at the root of a huge "NP-completeness-reduction" tree. It is something like an art form which requires some practice to find in this huge tree the problem Π which is best suited to be used in a reduction to your particular new problem Π, and to carry out the reduction. For beginners in this reduction game, a small number of particularly versatile NP-complete problems are recommended as "mother problems" Π': 3SAT, 3DM, VERTEX COVER, PARTITION, HAMILTONIAN CIRCUIT, CLIQUE. Their NP-completeness can be derived from SAT in several ways. The Garey/Johnson book proves reductions as shown in Figure 12.5. The same book also explains numerous standard "tricks of the trade" of how reduction proofs can be achieved. If you should come into a situation where you have to convince yourself of NP-completeness of "your" problem Π, you should read chapter 3 of that book.

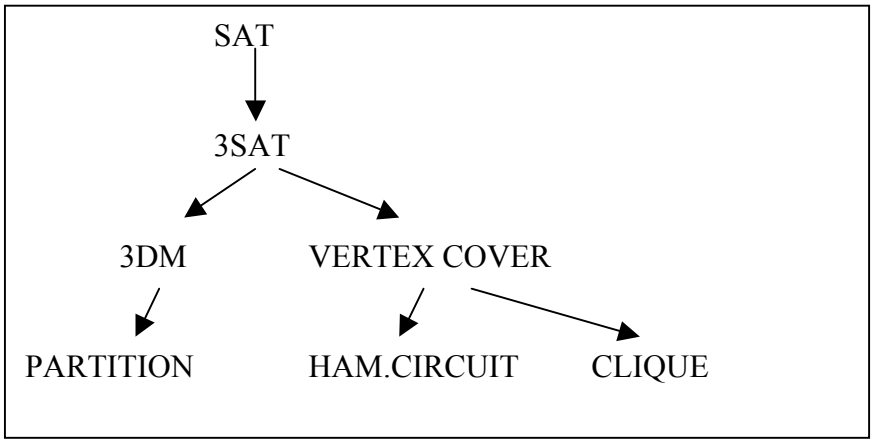


Figure 12.5: Diagram of reductions used to establish NP-completeness by (transitive) reduction from SAT.

After one has shown that one's current problem Π is NP-complete, one has likely saved a lot of time by not trying to do the impossible, namely, finding a practical algorithm for Π. However, one has not solved the problem. This is a typical situation to be in: one needs to solve a problem Π, but one knows that no tractable solution exists. What can one do? There are a number of options.

Option 1: Simplify the problem, by adding constraints or considering only special cases which are tractable.

This approach is extremely common and useful. The literature (e.g. the Garey/Johnson book) contains many examples of known restrictions (not reductions...) of NP-complete problems which turn exponential problems into polynomial ones. SAT is a point in case. We have already seen that HORNSAT is tractable. Another tractable subproblem of SAT is 2SAT: satisfiability of CNF Boolean formulas where each clause contains at most 2 literals.

Option 2: Be contended with approximate solutions.

This applies specifically to optimization problems, where the task is to find solution which optimizes some cost criterium, as for instance path length in TSP. Here one faces a *search* problem: in a multitude of solution candidates, find one which is close to the optimum. There is no single general method of how to obtain good approximate algorithms. A host of general-purpose techniques for search over "cost landscapes" is available, for instance genetic algorithms, simulated annealing, gradient descent methods, recurrent neural networks. They all require a good deal of specialized experience. In some cases, problem-specific approximate algorithms exist which have provable suboptimality bounds. An example is the optimal bin packing problem. This problem is a NP-hard variant of BIN PACKING and consists in finding a packing for a given set of items in a minimal number of bins. The following simple approximate algorithm is polynomial and can be shown never to result in a number of bins that is larger than the optimal number by at most a factor of 11/9:

1. sort the items into a sequence u_1, u_2,..., u_n, such that their size is decreasing
2. number your available bins b_1, b_2, ...
3. sort the items into bins in the order u_1, u_2,..., u_n, putting each item into the first available bin whose capacity still holds the current item.

Option 3: Use a randomized "Monte Carlo" algorithm which gives the correct answer most of the times.

Monte Carlo algorithms (MCA) work like follows on a given decision problem Π, with an instance x, for which a yes-or-no answer is wanted:

- Initialize the MCA with a random number.
- Apply the initialized MCA on x. In polynomial time, MCA answers "yes" or "no".
- If the MCA answer is "yes", then the true answer is also "yes", that is, the MCA produces no false positives.
- If the MCA answer is "no", then the true answer is "yes" with a probability $p < 1$. That is, the probability of false negatives is bounded away from 1.
- If you desire an answer that is false with a small residual error probability q, run the MCA with different initializations at most k times, where k is such that $p^k < q$. If one of these runs yields "yes", stop: the true answer is found. If all k runs yield "no", the probability that the true answer is "yes" is $< q$.

Monte Carlo algorithms are not easy to come by, because they require deep insight into the problem to prove the bound p. An important class of MCAs arises in number theory: highly efficient MCAs are known for determining whether an integer number is composite (= not prime). The Papadimitriou book (chapter 11) gives a MCA for integer compositeness which has $p = 1/2$.

12.3 P vs NP

The classes **P** and **NP** appear to be intuitively quite different:

- **P** contains those languages/problems for which there exists a tractable solution *finding* algorithm, whereas **NP** contains those languages/problems for which there exists a tractable solution *checking* algorithm.
- CIRCUIT VALUE and CIRCUIT SAT, two problems that are complete for the two classes, appear to be of a quite different nature – that is, purely intuitively speaking.
- Problems in **NP** have a strong flavour of "combinatorial search" – for all we know, there are not better ways of solving problems in this class other than by searching an exponentially branching search tree. In contrast, problems in **P** lack the element of search.

In spite of these striking surface differences, it is currently not known whether **P** is, in fact, different from **NP**. The question, **P** =? **NP**, is one of the most famous unsolved mathematical problems of the present. There is a 1 Mio Dollar prize set on settling this question (http://www.claymath.org/millennium/). The majority of mathematicians believes that the answer is **P** ≠ **NP**, but there is a non-negligible minority of about 10% who favour the opposite idea. Plus, there are a few people who suspect the problem is ill-posed and undecidable... [51]

[51] See http://www.claymath.org/millennium/P_vs_NP/Official_Problem_Description.pdf for an overview on current approaches and opinions on **P** =? **NP.**

Why is the **P** =? **NP** so famous, why should it be important except for a few specialists in computational complexity? There are a number of reasons that make this problem so tantalizing:

1. The problem can be easily grasped. Among the great unsolved problems of mathematics, the **P** ≠ **NP** problem is most easily understood by non-specialists, and can even be explained to mathematical laymen quite easily.
2. The question appears to have an almost obvious answer, namely, **P** ≠ **NP**. It is very painful for mathematicians (and computer scientists) that they have been unable to prove the apparently obvious for more than 40 years.
3. The problem is economically very relevant. If it would turn out that **P** = **NP**, and if the proof of this fact would be constructive (that is, provide a polynomial-time algorithm for some **NP**-complete problem), then the currently used public-key encryption algorithms would become breakable.
4. The question has deep philosophical aspects. In essence, if **P** = **NP**, then "creativity" and "intuition" [= guessing a solution for a problem] could be replaced by "mechanism", at least for problems in **NP**.
5. The problem reaches close to the heart of mathematics, namely, finding proofs of claims. If it would hold that **P** = **NP**, then for any logic (say, FOL) and any proof calculus (say, the sequent calculus) and any class of conjectures expressible in the logic, the language L_a = {<*c*> | <*c*> is a code of a conjecture expressed in the logic, and *c* has a proof within the calculus of length at most $|<c>|^a$} would be decidable in polynomial time. This circumstance would yield a family of polynomially cheap decision algorithms for whether conjectures *c* can be proven in polynomial-length proofs. Many open questions in mathematics might then be mechanically settled.

If one sets out to prove **P** = **NP**, the strategy is straightforward: simply find a way to solve one of the many NP-complete problems in polynomial time. The fact that no such method has yet been found, in conjunction with the counterintuitive implications that **P** = **NP** would have, leads most mathematicians to believe that **P** ≠ **NP**.

However, the world of algorithms is always good for surprises. Consider the problem PRIMES, whose instances are the natural numbers *n* and the question is whether *n* is composite. The naive method to check whether a natural number is composite is exponentially expensive: the "sieve of Eratosthenes check". However, PRIMES is easily seen to be in **NP**. The problem has much of the flavour of combinatorial search which is typical for NP-complete problems. However, NP-completeness of PRIMES could not be derived[52] – and for a good reason: in 2004, M. Agrawal, N. Kayal and N. Saxena found a polynomial-time algorithm for deciding primality of natural numbers[53], soon named the AKS algorithm. This earthquake result exploits deep insights from number theory and makes one at least faintly suspicious of the possibility that maybe, after all, **P** = **NP** might be ascertained by some "deep" algorithm solving some number-theoretic, NP-complete problem in polynomial time.

If you are interested in the futility of mathematical effort, check out the hilarious website http://www.win.tue.nl/~gwoegi/P-versus-NP.htm (pointed out to me by Dmitri Cucleschin). Besides some serious overview articles, and youtube videos and two Homer Simpson cartoons

[52] In fact, most researchers in that field suspected PRIMES to be in **P** since long, -- for their reasons, check out http://primes.utm.edu/prove/prove4_3.html

[53] **M. Agrawal**, **N. Kayal** and **N. Saxena**, "PRIMES in P," *Ann. of Math. (2)*, 160:2 (2004) 781--793. Available from http://www.cse.iitk.ac.in/users/manindra/.

concerning **P = NP**, this page also lists about 90 published (!) research articles, divided into two groups of roughly same size, where papers in the first group present proofs of **P = NP** and papers in the other group present proofs of **P ≠ NP**. Plus, there is a small group of papers that prove in various ways and senses why/how neither **P = NP** nor **P ≠ NP** can be proven.

13 Descriptive Complexity

In this final chapter I want to introduce you to a recent new approach to computational complexity, which takes a quite different perspective than the "classical", execution-and-implementation-oriented perspective of counting CPU cycles or measuring memory usage. This alternative approach, often called "descriptive complexity", characterizes complexity classes (like **P** or **NP**) not in terms of computational resources but in terms of the power of a logic needed to characterize the languages within one class. For instance, very roughly speaking, the class of languages that can be defined by propositional logic should be intuitively "less complex", or "simpler", than the class of languages definable within first-order logic. This approach has led to fascinating ramifications into other fields of logic, game theory and the theory of database languages. I follow closely a survey work[54] by Erich Grädel, a pioneer of the field.

The main results that I will report are a landmark theorem[55] by Fagin, which characterizes **NP** in terms of logical descriptive complexity, and which in retrospect was the starting shot for the field; and a more recent theorem by Grädel (1991) which gives a (partial, as we will see) characterization of **P** through a logic.

I find this a very satisfying capstone chapter for the two-semester courses on theoretical CS (formal languages, logics, computability and complexity), because it brings together logics and computational complexity, and is elegant and modern.

13.1 A recap of first-order logic

We start with a recap of first-order predicate logic (FOL), which I essentially copy from the Adv. CS 1 lecture notes. Check out those lecture notes at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/lectureNotes/LN_FLL_Fall10.pdf for details, explanations and examples. You probably may skip most of this subsection, except the definitions 13.11 through 13.14 at the end, which introduce concepts not covered in the Adv. CS 1 lecture (namely, prenex normal forms and Horn formulae in FOL).

Definition 13.1. Each FOL language uses the following sorts of *symbols*:

1. Symbols which are shared by all FOL languages:
    a. variables x_1, x_2, x_3, ... (for convenience we will also use y, z, and others),
    b. logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$,
    c. quantifiers $\exists$ and $\forall$,
    d. the identity symbol $=$,
    e. brackets (and).
2. Domain-specific symbols:
    a. for each $n \geq 1$, a set (possibly empty) of n-ary predicate symbols,
    b. for each $n \geq 1$, a set (possibly empty) of n-ary function symbols,
    c. a set (possibly empty) of constant symbols.

[54] E. Grädel, Finite Model Theory and Descriptive Complexity. Online manuscript, fetched from www-mgi.informatik.rwth-aachen.de, 2002. Copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/2160_Graedel07.pdf

[55] R. Fagin, *Generalized first order spectra and polynomial time recognizable sets*, in Complexity of Computation. SIAM-AMS proceedings 7 (R. Karp, ed.), 1974, 43-73

When one talks about the *symbol set S* of a FOL language, one typcially refers only to the collection of all its domain-specific symbols (the shared symbols are tacitly taken as granted). *S* is also referred to as the *signature* or the *vocabulary* of a FOL language.

Definition 13.2. (Syntax of FOL terms with symbols from *S*). Given a symbol set *S*, the *terms* of the FOL language over *S* are defined inductively, as follows:

1. Each variable is an *S*-term.
2. Each constant from *S* is an *S*-term.
3. If $t_0, ..., t_{n-1}$ are *S*-terms, and *f* is an *n*-ary function symbol from *S*, then $f\, t_0...t_{n-1}$ is an *S*-term.

Definition 13.3 (Syntax of *S*-expressions; FOL language over *S*) Given a symbol set *S*, the *expressions* of the FOL language over *S* are defined inductively, as follows:

1. For *S*-terms t_0 and t_1, $t_0 = t_1$ is an *S*-expression.
2. For *S*-terms $t_0, ..., t_{n-1}$, and an *n*-ary relation symbol *R*, $Rt_0...t_{n-1}$ is an *S*-expression.
3. For an *S*-expression φ, $\neg\, \varphi$ is an *S*-expression.
4. For *S*-expressions φ and ψ, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, $(\varphi \leftrightarrow \psi)$ are *S*-expressions.
5. For an *S*-expression φ and a variable *x*, $\exists\, x\, \varphi$ and $\forall\, x\, \varphi$ are *S*-expressions.

The set of *S*-expressions is denoted by L^S, that is, the first-order *language* over *S*.

Definition 13.4 (free occurrence of variables in *S*-expressions). We inductively define the set free(φ) of variables that occur *free* in an expression φ. For any term *t*, let var(*t*) denote the variables that occur in *t*. Then define

$$\text{free}(t_0 = t_1) = \text{var}(t_0) \cup \text{var}(t_1)$$
$$\text{free}(Rt_0...t_{n-1}) = \text{var}(t_0) \cup ... \cup \text{var}(t_{n-1})$$
$$\text{free}(\neg\, \varphi) = \text{free}(\varphi)$$
$$\text{free}((\varphi * \psi)) = \text{free}(\varphi) \cup \text{free}(\psi) \qquad \text{for } * = \wedge, \vee, \rightarrow, \leftrightarrow$$
$$\text{free}(\exists x\, \varphi) = \text{free}(\varphi) \setminus \{x\}$$
$$\text{free}(\forall x\, \varphi) = \text{free}(\varphi) \setminus \{x\}$$

We say that *x* is *bound* by a quantifier in an expression $\forall x\, \varphi$ or $\exists x\, \varphi$ if *x* occurs free in φ. *S*-expressions which do not contain free variables are called *propositions* or *sentences*. We write $\varphi[x_1, ..., x_n]$ to indicate that φ contains at most the free variables $x_1, ..., x_n$.

So much on the *syntax* of FOL. The *semantics* of FOL establishes in what sense *S*-expressions hold true (or not) for suitable mathematical objects, that is, how *S*-expressions *describe* mathematical entities. Here we adopt the set-theoretic view on mathematics, by which all mathematical objects can be seen as sets. Given a symbol set *S*, the *S*-structures are the objects which *S*-expressions can describe:

Definition 13.5 (*S*-structures).

Let *S* be a set of predicate, function, and constant symbols. An *S*-structure $\mathcal{A}$ is a non-empty set *A* (called the *domain* of $\mathcal{A}$ – note that "domain" is here a technical term), wherein the following conditions are declared:

1. For each constant symbol c of S, there is one fixed element of A, denoted by $c^{\mathcal{A}}$. For instance, if we talk about the real numbers, our symbol set S will very likely contain the constant symbol 1 [a formal symbol from our language], and $1^{\mathcal{A}}$ then is the real number of unit size [a mathematical "thing"].
2. For each unary predicate symbol P of S, there is a (possibly empty) subset A_P of A, denoted by $P^{\mathcal{A}}$. For instance, when describing ancient Athens, $\text{is-dog}^{\mathcal{A}}$ is the set of all dogs in Athens. That is, in predicate logic we adhere to the so-called *extensional* interpretation of properties: a property "is" just the set of all things which have that property. One also speaks of the *class* or a *type* of things that have the property.
3. For each binary predicate symbol R of S, there is a (possibly empty) subset A_P of $A \times A$, denoted by $R^{\mathcal{A}}$. That is, $R^{\mathcal{A}}$ is a set of pairs in A. Again, note the extensional perspective. For instance, when talking about the real numbers, $<^{\mathcal{A}}$ is the set of all pairs (x, y) of real numbers where $x < y$. Predicates of higher arity are treated in a similar fashion.
4. For each unary function symbol f of S, $f^{\mathcal{A}}$ is a unary function in A, that is, a function that takes arguments and values from A. Technically, one can define such a function again as a set of pairs: $f^{\mathcal{A}} = \{(x, y) \mid f^{\mathcal{A}}(x) = y\}$. Functions of higher arity are treated in a similar fashion.

Instead of $f^{\mathcal{A}}$ one often writes f^{A} etc. In many important examples of (mathematical) structures, the symbol set S is finite, even small. A convenient and much-used way to write down such a structure is to put all its ingredients into a tuple, the domain first. For instance, the standard structure of arithmetics is described using the symbol set $S_{\text{ar}} = \{+, \cdot, 0, 1\}$ and is denoted by

$$\mathcal{A}_{\text{ar}} = (\mathbb{N}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}}).$$

Definition 13.6 (variable assignments). An *assignment* in an S-structure is a mapping $\beta: \{x_0, x_1, \ldots\} \to A$.

Intuitively, an assignment specifies to what things variables point. If we fix an assignment, we can interpret the (free) variables in S-expressions by the assignees, and we can proceed further toward defining what S-expressions "mean". The technical term for the combined interpretation of symbols from S and variables under a fixed assignment is an S-interpretation:

Definition 13.7 (S-interpretations). An *S-interpretation* $\mathcal{I}$ is a pair $(\mathcal{A}, \beta)$, where $\mathcal{A}$ is an S-structure and β is an assignment.

Note that with $\mathcal{A}$ we know how to interpret symbols from S and with β we know how to interpret variables.

Often (namely, when we have to cope with variables unwantedly getting captured by quantifiers) we will want to rename variables. Renaming variables must be done carefully, because under a given assignment a variable is already "attached" to some element of A. To deal with this, we need a technical tool for adjusting assignments in one variable. We introduce the following notation:

Definition 13.8 (re-assignment of a single variable). Let β be an assignment, $a \in A$ and x a variable. By $\beta\frac{a}{x}$ denote the assignment which acts like β on all variables except x, and assigns x to a:

$$\beta\frac{a}{x} = \begin{cases} \beta(y) & \text{for } x \neq y \\ a & \text{for } x = y \end{cases}$$

If $\mathcal{I}$ is $(\mathcal{A}, \beta)$, let $\mathcal{I}\frac{a}{x}$ denote $(\mathcal{A}, \beta\frac{a}{x})$.

Definition 13.9 (interpretation of terms). Let $\mathcal{I}$ be an S-interpretation $(\mathcal{A}, \beta)$.

1. For each variable x, $\mathcal{I}(x) = \beta(x)$.
2. For each constant symbol $c \in S$, $\mathcal{I}(c) = c^{\mathcal{A}}$.
3. For n-ary $f \in S$ and S-terms $t_0, ..., t_{n-1}$, $\mathcal{I}(f t_0...t_{n-1}) = f^{\mathcal{A}}(\mathcal{I}(t_0), ..., \mathcal{I}(t_{n-1}))$.

Now we have everything prepared to define in rigorous terms what it means for some S-expression φ to "mean" something in an S-structure. The goal is to specify when a given φ is true and when it is false. The truth or falseness of some φ is relative to a given interpretation, because interpretations declare what the individual symbols occurring in φ mean. Therefore, φ can be true or false only relative to an interpretation $\mathcal{I}$. We say that $\mathcal{I}$ is a *model* of φ, or that $\mathcal{I}$ *satisfies* φ, or that φ *holds* in $\mathcal{I}$, and write $\mathcal{I} \vDash \varphi$, if φ is true relative to $\mathcal{I}$.

Technically, $\mathcal{I} \vDash \varphi$ is defined by induction over the structure of φ:

Definition 13.10 (model relation between an interpretation and an S-expression or a set of S-expressions).

a. Let $\mathcal{A}$ be an S-structure. For all $\mathcal{I} = (\mathcal{A}, \beta)$ we define

$\mathcal{I} \vDash t_0 = t_1$	iff	$\mathcal{I}(t_0) = \mathcal{I}(t_1)$
$\mathcal{I} \vDash Rt_0...t_{n-1}$	iff	$R^{\mathcal{A}}\,\mathcal{I}(t_0)\,...\mathcal{I}(t_{n-1})$ [that is, in extensional view, $(t_0, ..., t_{n-1}) \in R^{\mathcal{A}}$]
$\mathcal{I} \vDash \neg\varphi$	iff	not $\mathcal{I} \vDash \varphi$
$\mathcal{I} \vDash (\varphi \wedge \psi)$	iff	$\mathcal{I} \vDash \varphi$ and $\mathcal{I} \vDash \psi$
$\mathcal{I} \vDash \forall x\,\varphi$	iff	for all $a \in A$, $\mathcal{I}\frac{a}{x} \vDash \varphi$
$\mathcal{I} \vDash \exists x\,\varphi$	iff	for at least one $a \in A$, $\mathcal{I}\frac{a}{x} \vDash \varphi$

This definition can be extended by clauses dealing with $\vee$, $\rightarrow$, and $\leftrightarrow$ in an obvious manner.

b. If Φ is a set of S-expressions (possibly empty or infinite), we write $\mathcal{I} \vDash \Phi$ iff $\mathcal{I} \vDash \varphi$ for all $\varphi \in \Phi$.

Note that if φ is a *proposition*, the model relation does not depend on the assignment, that is, for $\mathcal{I}_1 = (\mathcal{A}, \beta_1)$ and $\mathcal{I}_2 = (\mathcal{A}, \beta_2)$, it holds that $\mathcal{I}_1 \vDash \varphi$ iff $\mathcal{I}_2 \vDash \varphi$. That is, for propositions φ we may write $\mathcal{A} \vDash \varphi$ instead of $\mathcal{I} \vDash \varphi$.

This ends our recapitulation of FOL basics from the earlier lecture Adv. CS 1.

13.2 Some more concepts of logic

For our purpose of characterizing **P** and **NP** in terms of logics, we need a few more concepts from logics, which are standard constructions in logics at large and worth knowing for their own sake.

Definition 13.11. A FOL S-expression φ is in *prenex normal form* if $\varphi \equiv Q_1 x_1 \ldots Q_n x_n \xi$, where $n \geq 0$, $Q_i \in \{\forall, \exists\}$, and ξ is an S-expression that does not contain any quantifier. In other words, expressions in prenex normal form start with a (possibly empty) block of quantifiers followed by a "body" expression which is quantifier-free.

Proposition 13.12. For each S-expression ϕ there exists an equivalent S-expression φ in prenex normal form.

Proof (sketch): an easy exercise. Use induction over the structure of S-expressions, plus the rules "$\forall \equiv \neg\exists\neg$" and "$\exists \equiv \neg\forall\neg$", to move quantifiers in ϕ to the left and negations around quantifiers to the right.

Definition 13.13. **a.** A FOL S-expression ξ is *atomic* if it is of the form $t_0 = t_1$, or of the form $Rt_0 \ldots t_{n-1}$, where the t_i are S-terms.
b. A *literal* is an atomic or negated atomic S-expression. The former are called *positive* literals, the latter *negative* literals.

Examples: $x = y$, Px_3, $Rfaxgy$ are atomic S-expressions for $S = \{P, R, f, g, a\}$ where P is a unary predicate symbol, R a binary predicate symbol, f a binary function symbol, g a unary function symbol, and a a constant symbol. These three expressions are also literals, as are $\neg x = y$, $\neg Px_3$, $\neg Rfaxgy$.

Definition 13.14. A FOL S-expression φ is a *Horn expression* if (i) it is in prenex normal form, (ii) the quantifier block of the prenex normal form only contains universal quantifiers, and (iii) the body is a conjunction of disjunctions of literals, where each disjunction contains at most one positive literal. These disjunctions are called the *clauses* of a Horn expression.

Examples: $x = y$, $Pax \wedge (fz = a \vee \neg Px_3)$, or $\forall x \forall y\, (Pax \wedge (fz = a \vee \neg Px_3))$ are Horn expressions, whereas $\forall x \exists y\, (Pax \wedge (fz = a \vee \neg Px_3))$ or $\forall x \forall y\, (Pax \wedge (fz = a \vee Px_3))$ are not.

Remarks.

1. Disjunctions of literals with at most one positive literal can be rewritten as implications. For instance, $fz = a \vee \neg Px_3$ is equivalent to $\neg Px_3 \rightarrow fz = a$. In general, $\neg\xi_1 \vee \ldots \vee \neg\xi_n \vee \zeta$ is equivalent to $(\xi_1 \wedge \ldots \wedge \xi_n) \rightarrow \zeta$. If one wishes, one can also interpret a single positive literal ξ as the implication **true** $\rightarrow \xi$, where **true** is some tautology (e.g., **true** $\equiv x = x$), and a negative literal $\neg\xi$ as the implication $\xi \rightarrow$ **false**, where for instance **false** $\equiv \neg x = x$.
2. Horn logic is the basis for *logic programming*, instantiated in the PROLOG programming language family, which is widely used in computer linguistics.

Second-order logic (SOL) is a (very far-reaching) generalization of FOL, which is obtained by admitting not only quantification on individuals, but also on predicates and relations. We do not give a formal specification of the extended syntax and semantics of SOL, which is quite straightforward, but rather illustrate it with the following formula:

$$\forall P\,((P0 \wedge \forall\, n\,(Pn \rightarrow P(n{+}1))) \rightarrow \forall\, n\,(Pn))$$

This is a SOL formula which reads, *"for all properties P, if 0 has it and if some n has it, then n+1 has it, too, then all n have this property"* – you should recognize the principle of induction over the natural numbers here.

Full SOL allows existential and universal quantification on predicates and relations of any arity, and is an extremely expressive logic. However, it is also a "monster", -- the other side of its extreme expressive power is that full SOL has almost no "nice" properties. In particular, any syntactic proof calculus for SOL will be far from being complete. Logicians stay away from full SOL as much as possible, and rather extend FOL by timid small steps toward full SOL whenever FOL is too weak for a particular purpose. Of such useful *fragments* of SOL there are plenty. For instance, one may extend FOL by a quantifier $\exists^f$, where $\exists^f x\ \varphi[x]$ would be interpreted to mean *"there exist only finitely many elements x sucht that* $\varphi[x]$*"*. $\exists^f$ can be defined within SOL (how? hint: with SOL it is possible to define the natural numbers.)

For our purposes in descriptive complexity, the following fragment of SOL is of particular interest:

Definition 13.15. *Existential second-order logic*, sometimes denoted Σ^1_1, is the set of SOL expressions of the form $\exists R_1 \ldots \exists R_m\ \varphi$, where φ is a FOL expression and $R_1, \ldots, R_m$ are relation variables of any arity.

Thus Σ^1_1 is the fragment of SOL characterized by the constraints that only existential second-order quantifiers are admitted, and that they all must appear in a negation-free fashion in the beginning of an expression.

13.3 Setting the stage for descriptive complexity

In the remainder of chapter 13, I follow closely the survey paper of Grädel, frequently copying portions verbatim, only adapting Grädel's notation to the one used in the Adv. CS 1 course.

Fagin's and Grädel's theorems characterize **NP** and (partially) **P** as the sets of languages that can be specified by formulae of certain extensions of FOL toward second-order logic. In order to formulate these theorems, we need to make ourselves familiar with the way of thinking in this field.

In traditional complexity theory, a language is a set of strings over some alphabet Σ. In the field of descriptive complexity, the notion of a language is, at first sight, more general: a language is a set of finite S-structures. This perspective is not fundamentally different, though, because strings over some alphabet can be seen as finite S-structures (exercise: how? there are many possibilities. Hint: one way to see symbol strings as S-structures is to interpret the symbols from Σ as unary predicates in S, and the linear order of a string is just this, a linear

order... for which you would need another binary relation symbol in S). Conversely, given some signature S, finite S-structure can be coded in bitstrings under certain circumstances. The next few definitions serve to clarify all of this.

Definition 13.16. For any signature S, we write $Fin(S)$ for the class of finite S-structures and $Ord(S)$ for the class of all structures $(\mathcal{A}, <^A)$, where $\mathcal{A} \in Fin(S)$ and $<^A$ is a binary relation on A which is a linear order on A. In other words, the structures in $Ord(S)$ are finite $S \cup \{<\}$ – structures (we may assume that $<$ is new to S), with the requirement that $<^A$ is a linear order on A. We also say, $(\mathcal{A}, <^A)$ is an *ordered S-structure*.

For any ordered S-structure $(\mathcal{A}, <^A)$ of cardinality $|A| = n$ and any k we can identify A^k with the set $\{0, ..., n^k - 1\}$, by associating each k-tuple with its position in the lexical ordering induced by $<^A$ on A^k. Ordered S-structures can be encoded in strings over some alphabet Σ in many ways; however, an encoding scheme must meet the following requirements:

Definition 13.17. A function $code: Ord(S) \to \Sigma^*$ is an *encoding* of ordered S-structures in words of some alphabet Σ, if

1. it identifies isomorphic structures, that is,
   $code(\mathcal{A}, <^A) = code(\mathcal{B}, <^B)$ if $(\mathcal{A}, <^A) \cong (\mathcal{B}, <^B)$,
2. its values are polynomially bounded, that is, if
   $|\, code(\mathcal{A}, <^A)\,| \leq p(|A|)$ for some polynomial p,
3. it is first-order definable in the following sense: for all $k \in \mathbb{N}$ and all $\alpha \in \Sigma$ there exists a FOL $S \cup \{<\}$ – expression $\beta_\alpha[x_1, ..., x_k]$ such that for all $(\mathcal{A}, <^A) \in Ord(S)$ and all $\bar{a} \in A^k, \bar{a} = a_1 \dots a_k$, with $N(\bar{a})$ being the position of $\bar{a}$ in the lexical ordering of A^k w.r.t. $<^A$, the following equivalence holds:
   $(\mathcal{A}, <^A) \models \beta_\alpha [a_1, ..., a_k]$ iff the $N(\bar{a})$th symbol of $code(\mathcal{A}, <^A)$ is α[56], and
4. given $code(\mathcal{A}, <^A)$, a k-ary relation symbol R of S and a (representation of a) tuple $\bar{a} \in A^k$, one can efficiently decide (in at most linear time) whether $(\mathcal{A}, <^A) \models R\bar{a}$.

Condition 3 is not very easy to understand. The key idea here is that in order to describe the codeword $code(\mathcal{A}, <^A)$ by first-order logic expressions, these expressions have to talk about the i-th symbol in $code(\mathcal{A}, <^A)$, for any i not greater than the length of the word $code(\mathcal{A}, <^A)$. The expression $\beta_\alpha [a_1, ..., a_k]$ "mis-"uses the tuple $(a_1, ..., a_k)$ merely to code the number $N(\bar{a})$.

A convenient encoding is the following. Let $<$ be a linear order on A and let $\mathcal{A} = (A, R_1^A, ..., R_t^A)$ be a relational structure of cardinality n. Let l be the maximal arity of the relation symbols $R_1, ..., R_t$. With each relation R of arity j we associate the binary string $\chi(R) = w_0 \dots w_{n^j-1} 0^{n^l - n^j} \in \{0,1\}^{n^l}$, where $w_i = 1$ if the i-th tuple of A^j belongs to R, and $w_i = 0$ otherwise. Now, set $code(\mathcal{A}, <^A) = 1^n 0^{n^l - n}\, \chi(R_1) \dots \chi(R_t)$. – If S contains function symbols, proceed accordingly, by coding the graph of the function; if S contains constant symbols, code them as if they were unary predicates comprising a single individual from A.

[56] $\beta_\alpha[a_1, ..., a_k]$ is obtained from $\beta_\alpha [x_1, ..., x_k]$ by replacing free occurences of the x_i by the corresponding a_i.

Definition 13.17 and the example suggest that having an order available within an S-structure $\mathcal{A}$ is essential for being able to code $\mathcal{A}$ in a string. In fact, there is no known generic way to encode S-structures $\mathcal{A}$ lacking an order. Therefore, when we want to encode an unordered S-structure $\mathcal{A}$, we associate with $\mathcal{A}$ the *set* of *all* encodings $code(\mathcal{A}, <^A)$, where $<^A$ is a linear order on A. We are now prepared to define in what sense we may algorithmically decide a class $\mathcal{K}$ of S-structures.

Definition 13.18. (deciding classes of S-structures) Let $\mathcal{K}$ be a class of finite S-structures. With $\mathcal{K}$ we associate the language

$$code(\mathcal{K}) = \{ code(\mathcal{A}, <^A) \mid \mathcal{A} \in \mathcal{K} \text{ and } <^A \text{ is a linear order on } A\}.$$

We say that a TM M decides $\mathcal{K}$ iff M decides $code(\mathcal{K})$.

Notes:

1. The task of deciding classes of S-structures is a generalization of the language decision task, since languages are classes of strings over some alphabet, and such strings can be seen as a particularly simple kind of S-structures.
2. If M decides $\mathcal{K}$, then M is invariant w.r.t. $<^A$, that is, for any finite S-structure $\mathcal{A}$ (not necessarily in $\mathcal{K}$), any linear orderings $<_1^A$, $<_2^A$ on A, it holds that $M(code(\mathcal{A}, <_1^A)) = M(code(\mathcal{A}, <_2^A))$. Unfortunately, it is undecidable whether a given TM M is invariant w.r.t. $<^A$. That is, when dealing with decision algorithms M for classes of S-structures, we have to verify on a case-by-case basis whether M is invariant w.r.t. $<^A$. Fortunately, this is usually not difficult.
3. In the light of definition 13.18, classes $\mathcal{K}$ of S-structures can essentially be seen as languages, and it makes sense to ask whether some such $\mathcal{K}$ belongs to a particular complexity class, say, **P** or **NP**. In particular, given some sentence φ in some logical language with signature S, it makes sense to ask what is the complexity of deciding $\mathcal{K}(\varphi) = \{ \mathcal{A} \mid \mathcal{A}$ is a finite S-structure and $\mathcal{A} \models \varphi \}$.

Capturing complexity classes. We are now prepared to understand the general approach taken in descriptive complexity. Recall that logicians freely design logics (plural!) as they deem fit. We have met with only a few logics so far in this lecture, namely propositional logic, FOL, full SOL, existential SOL, and the SOL fragment FOL + $\exists^f$ – but if this gave you the impression that only some few logics exist, then this is quite a false impression; certainly, thousands of logics have been introduced and used on the fly by mathematicians, computer scientists and linguists so far. Now, given *some* logic $\mathcal{L}$, and with it the set of well-formed $\mathcal{L}$-expressions φ (over arbitrary signatures S), descriptive complexity focusses on the question:

Given $\mathcal{L}$, is there a complexity class **C** such that for all $\mathcal{L}$-expressions φ, the complexity of deciding $\mathcal{K}(\varphi)$ is in **C**?

This statement of the mission of descriptive complexity is intuitive only and needs to be made more precise. To this end, we formulate the following definition:

Definition 13.19. (Capturing a complexity class by a logic on a domain of finite structures). Let $\mathcal{L}$ be a logic, **C** a complexity class (in the ordinary sense based on TMs), and **D** a class of

finite structures (later we will consider the class of *all* finite structures with non-empty finite signatures, and the class of *ordered* structures with finite signatures, i.e. such signatures which contain a <-relation which is a linear order on the carrier of the structure). We say that $\mathcal{L}$ captures **C** on **D** if

1. For every non-empty finite signature S and every sentence $\varphi \in \mathcal{L}(S)$, and for every finite S-structure $\mathcal{A}$, the language

   $$\{code(\mathcal{A}, <^A) \mid \mathcal{A} \in \mathcal{K}(\varphi), <^A \text{ is a linear order on } A\}$$

   is in **C**.

2. For every model class $\mathcal{K} \subseteq \mathbf{D}(S)$ [where $\mathbf{D}(S)$ denotes the restriction of **D** on S-structures], where the language $\{code(\mathcal{A}, <^A) \mid \mathcal{A} \in \mathcal{K}, <^A \text{ is a linear order on } A\}$ is in **C**, there exists a sentence $\varphi \in \mathcal{L}(S)$, such that

   $$\mathcal{K} = \{ \mathcal{A} \in \mathbf{D}(S) \mid \mathcal{A} \models \varphi \}.$$

For brevity, we say that $\mathcal{K} \in \mathbf{C}$ if the language $\{code(\mathcal{A}, <^A) \mid \mathcal{A} \in \mathcal{K}, <^A \text{ is a linear order on } A\}$ is in **C**. This gives a short way to state the above definition: that $\mathcal{L}$ captures **C** on **D** if for all $\mathcal{K} \subseteq \mathbf{D}(S)$: $\mathcal{K} \in \mathbf{C}$ iff $\mathcal{K} = \mathcal{K}(\varphi)$ for some $\mathcal{L}$-sentence φ.

In the light of this definition, the quest in descriptive complexity is for *identifying complexity classes with logics, relative to certain classes of finite structures*. The insights achieved in this way of thinking are "deeper", in a certain sense, than the insights achievable in traditional, "algorithmic" complexity theory. In traditional complexity theory, a complexity class is defined in terms of the *computational resources* needed to decide languages (or, in the more general view, classes of S-structures). These computational resources need some machine model, which is in a sense ad hoc – you may have found the ubiquitous use of TMs as the reference machine model somewhat unsatisfying (at least I did), and the justification of settling on TMs by showing that TMs can simulate other machine models in polynomial loss in efficiency (and vice versa) is tedious (and breaks down when linear or even sublinear complexity classes are considered – then the choice of the underlying machine model *does* suddenly matter).

In contrast, in descriptive complexity the fundamental question is not about computational resources to decide classes of objects, but the *descriptive* complexity of the objects themselves. The approach of descriptive complexity is free of ad-hoc choices of machine models!

Thus, a pure descriptive complexity approach would just not care about machine models, but instead focus on logics and the sets of S-structures $\mathcal{K}(\varphi)$ characterizable within particular logics. However, because the traditional approach to complexity is (of course) also very interesting and historically dominating, many results in descriptive complexity aim at the "classical" complexity classes, like **L**, **P** or **NP**, and try to find logics that characterize those traditional classes.

The first landmark result of descriptive complexity, to which we will now direct our attention, is therefore a result that found a logic which captures **NP**.

13.4 Fagin's theorem: capturing NP on finite structures

In 1974, Ronald Fagin (now at the IBM Almaden Research Center[57]) proved Fagin's theorem:

Theorem 13.1. Let $\mathcal{K}$ be an isomorphism-closed class of finite S-structures, where S is any finite, non-empty signature. Then $\mathcal{K} \in$ **NP** iff $\mathcal{K} = \mathcal{K}(\varphi)$ for some existential SOL sentence φ. In other words, existential second-order logic captures **NP** on the class of all finite S-structures (with non-empty signature).

Note. The restriction to non-empty signatures is essential. The reason is that S-structures with empty signature are just bare sets which would be encoded differently (by logarithmically shorter codewords) than with the *code* conventions assumed here, leading to different complexity behavior.

Proof. First we show how to decide $\mathcal{K}(\psi)$ in nondeterministic polynomial time, for any existential SOL sentence $\psi = \exists R_1 \ldots \exists R_m \varphi$. For this we need a nondeterministic polynomial-time algorithm M which, given an encoding $code(\mathcal{A}, <^A)$ of an S-structure $\mathcal{A}$, decides whether $\mathcal{A} \models \psi$. First, M guesses relations $R_1^A, \ldots, R_m^A$ on A. A relation R_i^A is determined by a binary string of length n^{r_i}, where r_i is the arity of R_i^A and $|A| = n$. Then M decides whether $(\mathcal{A}, R_1^A, \ldots, R_m^A) \models \varphi$. Since φ is a first-order expression, this can be done in polynomial time (exercise: use induction on the structure of φ). Hence the computation of M consists of guessing a polynomial number of bits, followed by a deterministic polynomial-time computation.

Conversely, let $\mathcal{K}$ be an isomorphism-closed class of finite S-structures and M a nondeterministic, single-tape TM which, on input $code(\mathcal{A}, <^A)$, decides in polynomial time whether $\mathcal{A}$ belongs to $\mathcal{K}$. We construct an existential second-order sentence φ whose finite models are precisely the structures in $\mathcal{K}$.

Let $M = (Q, \Sigma, q_0, \delta)$ be a single-tape nondeterministic TM with state set Q, tape symbol set Σ, starting state q_0, transition function δ, just as we know it except that here we have a *set* $F^+ \subseteq Q$ of accepting states and $F^- \subseteq Q$ of rejecting states, and that we identify the cursor motion set $\{\rightarrow, \leftarrow, -\}$ with the integers $\{1, -1, 0\}$. Without loss of generality we may assume that all computations of M on input $code(\mathcal{A}, <^A)$ reach an accepting or rejecting state after at most $n^k - 1$ steps, where n is the size of A.

We represent a computation of M on input $code(\mathcal{A}, <^A)$ by a tuple $\bar{X}$ of relations on A, and we will construct a first-order sentence φ_M of signature $S \cup \{<\} \cup \{\bar{X}\}$ such that

$$(\mathcal{A}, <, \bar{X}) \models \varphi_M \qquad \text{iff} \qquad \text{the relations } \bar{X} \text{ represent an accepting computation of } M \text{ on } code(\mathcal{A}, <^A).$$

To represent the n^k time and space parameters of the computation we identify numbers up to $n^k - 1$ with tuples in A^k. Given a linear order, the associated successor relation σ and the least

[57] http://www.almaden.ibm.com/cs/people/fagin/

and greatest element are first-order definable[58]. Note further, that if a successor relation σ and constants 0, *e* for the first and last elements are available, then the induced successor relation $\bar{y} = \bar{x} + 1$ on *k*-tuples (where each member of the tuple is an element from *A* which we may identify with its position index w.r.t. <) is definable by a quantifier-free expression

$$\bigvee_{i<k}\Big(\bigwedge_{j<i}(x_j = e \wedge y_j = 0) \wedge \sigma x_i\, y_i \ \wedge \ \bigwedge_{j>i} x_j = y_j\Big).$$

Hence for any integer *m* (may also be negative), the relation $\bar{y} = \bar{x} + m$ is first-order definable, too.

The description $\bar{X}$ of a computation of *M* on *code*($\mathcal{A}$, $<^A$) consists of the following relations:

1. For each state $q \in Q$, the predicate

   $X_q := \{ \bar{t} \in A^k \mid$ at time $\bar{t}$, *M* is in state $q\}$.

2. For each symbol $\alpha \in \Sigma$, the predicate

   $Y_\alpha := \{ (\bar{t}, \bar{c}) \in A^k \times A^k \mid$ at time $\bar{t}$, tape cell $\bar{c}$ contains symbol $\alpha\}$.

3. The head predicate

   $Z := \{ (\bar{t}, \bar{c}) \in A^k \times A^k \mid$ at time $\bar{t}$, the read/write head is on position $\bar{c}\}$.

The sentence φ_M is the universal closure (w.r.t. first-order universal quantification) of the conjunction START ∧ COMPUTE ∧ END.

The subexpression START enforces that the configuration of *M* at time 0 is $C_0(\mathcal{A}, <^A)$, that is, the input configuration on *code*($\mathcal{A}$, $<^A$). Recall from condition 3 in definition 13.17 that an encoding is represented by first-order $S \cup \{<\}$ - epressions $\beta_\alpha[x_1, ..., x_k] =: \beta_\alpha[\bar{x}]$. We set

$$\text{START} := X_{q_0}(\bar{0}) \wedge Z(\bar{0}, \bar{0}) \wedge \bigwedge_{\alpha\in\Sigma}(\beta_\alpha(\bar{x}) \rightarrow Y_\alpha(\bar{0}, \bar{x})).$$

The subexpression COMPUTE describes the transitions from one configuration to a next one. It is the conjunction of the expressions

$$\text{NOCHANGE} := \bigwedge_{\alpha\in\Sigma}\Big(Y_\alpha(\bar{t}, \bar{x}) \wedge (\bar{y} \neq \bar{x}) \wedge (\bar{t}' = \bar{t} + 1) \wedge Z(\bar{t}, \bar{y}) \rightarrow Y_\alpha(\bar{t}', \bar{x})\Big)$$

and

$$\text{CHANGE} := \bigwedge_{\substack{q\in Q\\ \alpha\in\Sigma}}\Big(PRE[q,\alpha] \rightarrow \bigvee_{(q',\alpha',m)\in\delta(q,\alpha)} POST[q',\alpha',m]\Big),$$

where

[58] Exercise: assuming that < satisfies the axioms of a linear order, provide a FOL {<}-expression *succ*[*x*, *y*] that encodes *succ*[*x*, *y*] ⇔ *y* is the direct successor of *x* under <; and give FOL {<}-expressions *first*[*x*] amounting to "*x* is the minimal element of <" and *last*[*x*] amounting to "*x* is the maximal element of <"

$$PRE[q,\alpha] := X_q(\bar{t}) \wedge Z(\bar{t},\bar{x}) \wedge Y_\alpha(\bar{t},\bar{x}) \wedge (\bar{t}' = \bar{t} + 1),$$
$$POST[q',\alpha',m] := X_{q'}(\bar{t}') \wedge Y_{\alpha'}(\bar{t}',\bar{x}) \wedge (\exists \bar{y}\ \bar{y} = \bar{x} + m) \wedge Z(\bar{t}',\bar{y}).$$

NOCHANGE expresses that the contents of currently not scanned tape cells do not change from one configuration to the next, whiles CHANGE enforces the changes of the relations X_q, Y_α, and Z imposed by the transition function.

Finally we have the expression

$$\text{END} := \bigwedge_{q \in F^-} \neg X_q(\bar{t}),$$

which enforces acceptance by forbidding rejection.

Having captured the workings of M in φ_M, we proceed with the main argument of our proof, by showing two claims.

Claim 1. *If M accepts code*$(\mathcal{A}, <^A)$, *then* $(\mathcal{A}, <^A) \models (\exists \bar{X})\varphi_M$.

This follows immediately from the construction of φ_M, since for any accepting computation of M on *code*$(\mathcal{A}, <^A)$, the intended meaning of $\bar{X}$ satisfies φ_M.

Claim 2. *If* $(\mathcal{A}, <^A, \bar{X}) \models \varphi_M$, *then M accepts code*$(\mathcal{A}, <^A)$.

Expressed in words, this claim says that "if there exist relations $\bar{X} = (\{X_q\}, \{Y_\alpha\}, Z)$ of arities $(k, 2k, 2k)$ on A, such that $(\mathcal{A}, <^A, \bar{X}) \models \varphi_M$, then M accepts *code*$(\mathcal{A}, <^A)$." In order to show the claim, suppose that $(\mathcal{A}, <^A, \bar{X}) \models \varphi_M$. For any M-configuration C with state q, head position p and tape content $\alpha_0 \sqcup \alpha_{n^k-1} \in \Sigma^*$, and for any time $j < n^k$, let CONF$[C, j]$ be the conjunction of the atomic statements that hold for C at time j, that is,

$$\text{CONF}[C,j] := X_q(\bar{j}) \wedge Z(\bar{j},\bar{p}) \wedge \bigwedge_{i=0,\ldots,n^k-1} Y_{\alpha_i}(\bar{j},\bar{i}),$$

where $\bar{j}, \bar{p}, \bar{\imath}$ are the tuples in A^k representing the numbers j, p, i.

a. Let C_0 be the input configuration of M on *code*$(\mathcal{A}, <^A)$. Since $(\mathcal{A}, <^A, \bar{X}) \models$ START, it follows that

$$(\mathcal{A}, <^A, \bar{X}) \models \text{CONF}[C_0, 0].$$

b. Due to the subexpression COMPUTE of φ_M we have for all non-final configurations C and all $j < n^k - 1$ that

$$\varphi_M \wedge \text{CONF}[C,j] \models \bigvee_{C' \in Next(C)} CONF[C', j+1]$$

where $Next(C) = \{C' \mid C \xrightarrow{M} C'\}$ is the set of possible successor configurations of C. It follows that there is a computation

$$C_0(\mathcal{A}, <^A) = C_0 \overset{M}{\rightarrow} C_1 \overset{M}{\rightarrow} \ldots \overset{M}{\rightarrow} C_{n^k-1} = C_{\text{end}}$$

of M on $code(\mathcal{A}, <^A)$ such that for all $j < n^k$,

$$(\mathcal{A}, <^A, \bar{X}) \models \text{CONF}[C_j, j].$$

c. Since $(\mathcal{A}, <^A, \bar{X}) \models \text{END}$, the configuration C_{end} is not rejecting. Thus, M accepts $code(\mathcal{A}, <^A)$.

This proves claim 2. Let $\varphi_<$ be a FOL sentence axiomatizing linear orders (exercise). Taking into account what was said in definition 13.18 concerning that M is invariant w.r.t. $<^A$, we finally obtain from the two claims

$$\mathcal{A} \in \mathcal{K} \quad \text{iff} \quad \mathcal{A} \models \exists < \exists \bar{X} (\varphi_< \wedge \varphi_M),$$

that is, we have characterized $\mathcal{K}$ by an existential SOL sentence $\psi = \exists < \exists \bar{X} (\varphi_< \wedge \varphi_M)$. □

Fagin's theorem is the strongest and "deepest" theorem that we met in this lecture. Some of its implications connect some old-standing questions of logics to complexity – topics beyond the scope of this lecture (if you are interested, check out corollary 2.8 in Grädel's survey). Here we demonstrate the power of Fagin's theorem by deriving Cook's theorem as a simple corollary:

Corollary 13.2. SAT is **NP**-hard.

Proof. Let L be any problem in **NP** – where by "problem" here we refer to the classical interpretation of L as a language of codewords of problem instances, over some alphabet Σ. Since codewords can be considered as finite structures over some suitable signature S (was a teaser exercise at the beginning of section 13.3), we can identify L with some $\mathcal{K} \subseteq Fin(S)$. By Fagin's theorem, there exists a FOL sentence φ such that

$$\mathcal{K} = \{ \mathcal{A} \in Fin(S) \mid \mathcal{A} \models \exists R_1 \ldots \exists R_m \varphi \}.$$

We have to find a transformation from a problem instance $\mathcal{A} \in Fin(S)$ (that is, a word $w \in \Sigma^*$, seen as a finite S-structure) to a Boolean formula. We use the information from $\mathcal{A}$ and adapt φ to our purpose, obtaining a Boolean formula $\varphi_{\mathcal{A}}$ that does what we want. Given $\mathcal{A}$, replace in φ

- all subexpressions $\exists x\, \zeta$ by $\bigvee_{a \in A} \zeta\left[\frac{a}{x}\right]$,[59]
- all subexpressions $\forall x\, \zeta$ by $\bigwedge_{a \in A} \zeta\left[\frac{a}{x}\right]$,
- all (ground) S-atoms by their truth values in $\mathcal{A}$.

After this transformation, the remaining atoms $R_i \bar{a}$ with relation variables R_i are considered as Boolean variables, and we have

[59] $\zeta\left[\frac{a}{x}\right]$ denotes the expression that is identical to ζ, except that free occurrences of x have been replaced by a.

$\mathcal{A} \in \mathcal{K}$ iff $\mathcal{A} \models \exists R_1 \ldots \exists R_m \varphi$ iff $\varphi_{\mathcal{A}}$ is satisfiable. □

13.5 Grädel's theorem: capturing P on ordered structures

It would be nice to have an analog of Fagin's theorem for a logical characterization of finite structures that can be decided by some TM in *deterministic* polynomial time – said simply, we want to capture **P** by some logic. A theorem of Grädel[60] (1992) comes close, but a full solution is unknown – a very interesting fact, see the concluding remarks at the end of this subsection.

Essentially, Grädel's theorem states that if we use existential *Horn* SOL instead of existential SOL (as in Fagin's theorem), then we (almost) capture **P** instead of **NP**. Let us first fix notation:

Definition 13.20. *Existential second-order Horn logic*, sometimes denoted Σ^1_1-HORN, is the set of SOL expressions of the form $\exists R_1 \ldots \exists R_m \varphi$, where φ is a FOL Horn expression (see definition 13.14) and $R_1, \ldots, R_m$ are relation variables of any arity.

The point where Fagin's theorem is more general than what we will be able to get for **P** is that Fagin's theorem was stated for arbitrary S-structures $\mathcal{A} \in Fin(S)$, while now we will have to restrict ourselves to *ordered* S-structures $\mathcal{A} \in Ord(S)$:

Theorem 13.3 (Grädel 1992). Let $\mathcal{K}$ be an isomorphism-closed class of ordered finite S-structures $\mathcal{A}$, where S is any finite, non-empty signature containing $<$ such that $<^A$ is a linear order. Then $\mathcal{K} \in \mathbf{P}$ iff $\mathcal{K} = \mathcal{K}(\varphi)$ for some existential SOL Horn sentence φ. In other words, existential second-order Horn logic captures **P** on ordered structures.

Remarks on the proof. The easy direction (if $\mathcal{K} = \mathcal{K}(\varphi)$ for some existential SOL Horn sentence φ then $\mathcal{K} \in \mathbf{P}$) is essentially a re-use of the idea in the proof of corollary 13.2, using as a final twist that HORNSAT $\in \mathbf{P}$. The hard direction of the proof is a remake of the proof of Fagin's theorem, exploiting determinism of M to make φ_M Horn. The point where the restriction to ordered S-structures comes into play is right at the end of Fagin's original proof, where a FOL sentence $\varphi_<$ describing linear orderings was used – but such a sentence is necessarily non-Horn.

It is conjectured that no logic can capture **P** on general S-structures. Proving this, however, amounts to settling the question **P** =? **NP** (Detailed treatment in Section 5.4 of Grädel's survey).

I think this is a nice capstone for this lecture, which knots together logic, complexity, and **P** =? **NP** in a profound, surprising, and unsettled way, leaving room for your own future breakthrough research (*someone* will do it – why not YOU??).

[60] E. Grädel, Capturing complexity classes by fragments of second-order logic. Theoretical Computer Science 101 (1992), 35-57

14 What did we learn?

Here is the eagle's view of the grand picture of computation and complexity – the big, essential, over-arching and under-lying landscape-shaping insights. For the sake of completeness, I include some hints to material that we did not cover in the lecture (and that will not be tested in the final exam). This material is marked by (*)

It is clear what "computation" and "computability" means.

It is not clear that this should be clear! But in the last 50 years or so (and not earlier), it has indeed become clear. Computation

- is what Turing Machines can do is what RAMs can do
- is what recursive functions can produce
- is what lamda calculus can produce
- is what your PC (under Windows or under Linux or in assembler or even using Word macros) can do
- (*) is what type-0 grammars can do
- (*) is what cellular automata can do (we cast a glance on them in the tutorial)
- (*) is what first order predicate logic can do (we did not enter into this topic)
- is what ...

All of these models of computation (and there are even more), as different as they may appear on the surface, are known to be equivalent. The Church-Turing hypothesis claims that indeed THIS IS COMPUTATION, and there is no way to "compute" beyond this.

So we have a big collection of equivalent *mechanisms* of computation. Likewise, there is a sizable collection of equivalent *task types* of computation. We have seen that

- the question of calculating an integer-valued partial function is essentially the same as
- the question of accepting a language as
- the question of solving a problem (which in full generality would include problems that admit "undecidable" outcomes).

Turing machines (and other computational mechanisms) have "universal simulation power"

We have seen that a universal TM can simulate any other TM. We have seen that RAMs can simulate TMs and vice versa. We have seen how programs written in L5 can be transformed into lambda expressions and we have seen a part of the story how recursive functions can be captured in lambda calculus. (*) We omitted a host of similar mutual simulation methods in the lecture – for instance, how a TM can be specified in first order logic, how a TM can be simulated by a cellular automaton (we saw a glimpse of that in a homework problem and in the proof of Cook's theorem), how a TM can be coded as an arithmetic function, etc. All in all, each of the (equivalent) computation mechanisms is capable of simulating every other.

Note that every such simulation of some mechanism A by another mechanism B involves some coding *convention*. For instance, the representation of integers as Church numerals is just one possible way of expressing numbers as lambda expressions. Likewise, if a universal TM *U* is constructed, you have to fix a scheme for coding other TMs as suitable input strings for *U*. Again, this rests on a convention and can be done in many different ways.

From that universal simulation power, undecidability results from self-simulation paradoxes

A universal TM *U* can simulate itself – and thus run into trouble when it is asked whether it will run forever on the question whether it will run forever. If you assume that this question can be answered by *U* itself, then it should terminate with "yes" iff it runs forever – this is the basic idea of the diagonalization proof of showing undecidability of the halting problem. (*) The same idea of diagonalization underlies many other undecidability results relating to other computation mechanisms – for instance, the famous undecidability theorems concerning first order logics, due to Gödel. While Gödel's original proof did not use TMs (they were not known at that time) but employed a direct FOL-internal diagonalization approach, today most textbooks prove undecidability of FOL by reduction to the halting problem, in that first it is shown that FOL can express TMs.

(*) In a similar vein, it can be shown that one cannot decide whether a lambda expression reduces to normal form or whether a recursive function is defined on a given input argument, etc.

All of these undecidability results have a certain contrived flavour – after all, computational problems involving self-simulations are not very "natural". (*) But there exist also "natural", "useful" problems that can be shown to be undecidable. They are not easy to come by and the few results available are celebrated.

Often one computational problem can be expressed in terms of another – reductions.

Not only computational mechanisms, but also computational problems sometimes can "simulate" another. The basic idea is to construct a *reduction*: Translate some problem A into another problem B such that A can be solved by first translating it into B and then solving B. We used this strategy twice: for deriving undecidability results (reduction of the halting problem to some target language) and for obtaining NP-complete problems (reduction of SAT to some target problem).

The rigorous semantics and proof calculi for lambda-calculus and FOL is the basis for program verification

For lambda calculus and for FOL there exist rigorous *proof calculi*, that is, purely syntactical symbol-manipulation systems that allow you to derive statements of the form "given that the FOL formulae ξ_1, ..., ξ_n are true, φ is true" or "given that equivalences $\xi_1 = \zeta_1$, ..., $\xi_n = \zeta_n$ hold true, the lambda expression φ is equivalent to the lambda expression ψ".

We learnt in ACS 1 what it means that a FOL formula ξ "holds": it holds for a mathematical structure $\mathcal{I}$ (an "interpretation") if $\mathcal{I} \models \xi$ (we defined what that means). "Holding true" is a *semantic* relation between an expression ξ and a (mathematical) structure $\mathcal{I}$. What a structure is we defined using set theory – basically, it is a set where the symbols appearing in ξ are *interpreted*.

(*) In a similar way, one can define mathematical structures suitable for use with lambda calculus. However, one typically does not use sets as the mathematical "substance" from which model structures for lambda calculus are made, but directed hierarchical *graphs* (in a hierarchical graph, sets of nodes can be assembled into "meta-nodes"). We did not explore the issue of *graph models* for lambda calculus in the lecture. But the idea is the same as with the

set-based models used with FOL: first, one interprets the symbols of a lambda expression ξ by subgraphs (just as constants in FOL are interpreted by set elements, relation symbols by relation sets etc). Then, there is a general scheme of how the syntactical operations of concatenation and lambda abstraction (the syntactical constructions that make up lambda expressions from their constitutents) are reflected in the construction of other subgraphs from the constitutent subgraphs. In this way, a lambda expression "means" a subgraph in a graph model. Two lambda expressions are equivalent (semantically) if they mean the same subgraphs in all graph models. The lambda conversion calculus that we introduced in Definition 7.6. is *correct* in the sense that if equivalence of two lambda expressions φ and ψ is derived from premises $\xi_1 = \zeta_1, ..., \xi_n = \zeta_n$, then the subgraphs denoted by φ and ψ are the same in all graph models where the subgraphs denoted by ξ_1 and $\zeta_1, ..., \xi_n$ and ζ_n are the same.

(*) The existence of correct proof calculi for FOL and/or lambda calculus is the reason why FOL and lambda calculus can be used for *program verification*: Generally speaking, the program and its input is represented by some FOL formulae $\xi_1, ..., \xi_n$ or lambda equations $\xi_1 = \zeta_1, ..., \xi_n = \zeta_n$. Then the respective calculus is used to prove that other FOL formulae ψ or lambda equations $\varphi = \psi$ are entailed. The formulae ψ or lambda equations $\varphi = \psi$ are the specifications of the desired (and then guaranteed) result of the program. The technical implementation of program verification tools may look different on the surface from this, but this is the logical essence behind the scene.

Computational complexity classes show that there are problems of qualitatively different "difficulty"

The basic insight of complexity theory is that computational resources matter: the more time or memory space you have, the more problems you can solve. There are two main types of results:

- Complexity hierarchies: results of the kind $\mathbf{C}(f(n)) \subsetneq \mathbf{C}(g(n))$, where **C** defines some way to measure complexity (for instance time or space) and $f(n)$ grows slower than $g(n)$.
- Relations between different ways of measuring complexity: results of the kind $\mathbf{C}(f(n)) \subseteq \mathbf{D}(g(n))$, where **C** and **D** relate to two different ways of measuring complexity (for instance space vs. time or deterministic vs. nondeterministic).
- (*) Another type of complexity results are *tradeoff* insights of the kind $\mathbf{C}(f(n), g(n)) \subseteq \mathbf{C}(f'(n), g'(n))$, where **C** depends on two functions of the input length, each one measuring a different aspect of complexity, for instance $\mathbf{C}(f(n), g(n)) = \mathbf{TIME}(f(n)) + \mathbf{SPACE}(g(n))$.

The theory of computational complexity is more abstract than a practitioner might wish. Most results contain a "big-Oh", that is, complexity claims are made up to a constant factor. Even worse, much of complexity theory does not even care about polynomial differences – classes **P** and **NP** and higher classes like **EXP** don't differentiate between polynomially related complexities. When real-life computation time matters, this is too coarse. Therefore, when it comes to fine-tuning real-life algorithms, the insights of complexity theory are of little help.

There is a continental divide in complexity: tractable vs. intractable

Problems in **P** are called tractable, problems that need exponential resources are called intractable. While this appears plausible at first sight, this perspective should be adopted with caution:

- If a problem is polynomial of high order, it is dubious to call it "tractable". However, experience shows that if a high-order polynomial algorithm is found for some problem, sooner or later lower-order polynomial algorithms will be found. A point in case is the polynomial-time algorithm found for testing primality of integers (see remarks in 7.3).
- Large constants hidden in the big-Oh can be cumbersome. Much ingeniuity may have to be spent to reduce the multiplicative constants. A point in case are certain algorithms for linear adaptive filtering in signal processing (the RLS algorithm family) which has time complexity $O(n^2)$ with constants in the range of 7 for naive implementations. Because these algorithms are often run in time-critical applications, it matters that with an extra effort the time constant can be halved.
- (*) Conversely, it sometimes happens that a problem has an exponential-time algorithm which in practice runs faster than another algorithm that has polynomial time. The reason is that our definition of complexity actually measures the worst-case resource needs. Sometimes the worst case occurs very rarely or not at all in "reasonable" problem instances, and the *average* resource consumption of an exponential algorithm may be quite small. The study of average complexity is a (difficult but important) subfield of complexity theory.

Almost all interesting optimization problems are equivalent, and probably very difficult.

Whether you want to optimize a VLSI layout, find a winning chess strategy, or minimize the expenses when scheduling a fleet of trucks, you invariably end in an **NP**-complete or **NP**-hard problem. **NP**-complete problems are a quite magical class: there are very many of them, and they are all alike. It seems that when real-life problems reach a certain level of complexity, you are always basically in the same situation. You very likely have to test out all possibilities (that is the deterministic-exponential nature of **NP**), and in testing out all possibilities you have to invoke a general-purpose search method that you can re-use on other such problems (that is the completeness aspect).

Nondeterminism is a deep riddle

Is **P** = **NP**? **NP** is the class of problems that apparently need "ingenuity": you first have to guess the solution, then checking it is easy (= polynomial in time). Can the guessing part be captured by an efficient (= polynomial-time) deterministic (= machine-runnable) algorithm? If so, mathematicians or executive decision-makers might become obsolete, because the "gut feeling" that guides human judgement would become efficiently programmable. It appears rather unlikely that this should be possible – but who knows? and why can't we settle this question despite great efforts? Where's the riddle? We can't even put our finger on it. There is something about solving problems that we deeply don't understand.

It is not really clear what "computational complexity" means

While we have a widely shared and transparent understanding of the concept of computability (is a function computable – yes/no), we do not have a likewise transparent and universally agreed notion of complexity (is a function costly to compute – less/more). In the lecture we have dwelled mostly on the most classical way to come to terms with complexity, which is

based on "machine running cost" intuitions about complexity and in its most elementary versions measures TM runtime and tape expenditure. However, truly interesting and relevant questions arise when the "unrealistic" nondeterministic complexities are considered; and there are weirder ramifications of the standard TIME and SPACE complexity measures (such as considering the complexities obtained by TMs equipped with "oracles"...) which are "irrealistic" but yield important insights. We also threw a look on an alternative approach to characterizing the complexity of problems, i.e. descriptive complexity. Let me mention that there are yet other approaches to come to terms with computational complexity, for instance the *axiomatic complexity* framework of Blum (check out Google for "complexity Blum axiom"), which could be seen as the broadest possible abstraction of the "classical" complexity measures, and *Kolmogorov* (or minimal description length) *complexity*, which describes the complexity of stochastic signals in terms of sizes in terms of the required size of generators for these signals.