Lecture on undecidability

Michael M. Wolf

June 27, 2012

Contents

1 Historical bits and pieces **4**
1.1 Useful and paradoxical self-references 4

2 What's an algorithm? **6**

3 Turing machines and busy beavers **9**
3.1 Functions computable by Turing machines 10
3.2 Composition of Turing machines 11
3.3 Rado's theorem and busy beavers 11
3.4 First occurrence of the halting problem 13
3.5 Variations 13

4 Primitive recursion **14**
4.1 Primitive recursive relations, sets and predicates 15
4.2 Bounded operations 16

5 Gödel numbers, codes, indices **18**
5.1 Multiplicative encoding 19
5.2 Pairing functions 19

6 Limitations of primitive recursion **20**
6.1 Relation to modern programming languages 23

7 Recursive function vs. Turing computability **23**
7.1 Wang encoding of the tape & head configuration 24
7.2 Encoding the Turing machine instructions 25
7.3 Evolution of the Turing machine configuration 26
7.4 Halting & output 26

8 Basic theorems of recursion theory **27**
8.1 The recursion theorem and some applications 29

9 Rice's theorem and the Church-Turing thesis **31**
9.1 Church-Turing thesis 32

10 Recursive enumerability **33**

11 The word problem for Thue systems **35**
11.1 Terminology and notation 36

12 Undecidable problems for semigroups **38**

13 Undecidable problems related to groups and topology **40**

14 Post's correspondence problem **41**

15 Undecidable matrix problems **42**
15.1 Matrix mortality 43
15.2 A reachability problem 44

16 Hilbert's tenth problem **46**
16.1 Examples of diophantine predicates 47
16.2 Examples of diophantine functions 47

1 Historical bits and pieces

When tracing back the attempts of axiomatizing mathematics and automatizing proofs and calculations, one comes across a Doctor of Law in the late seventeenth century. In Hannover, in the best of all worlds, he investigated formal logic and at the same time built some of the worlds first computers. About 200 years and 100km further, similar thoughts were then taken on and posed as now famous problems to the community of mathematicians.

Lecture 0

In this course we are interested in four of them and their aftermath:

1. Decide the continuum hypothesis, i.e. the question whether there is a set with cardinality strictly between that of the integers and that of the reals (Hilbert's 1st problem, 1900).
2. Prove consistency of the axioms of arithmetics (Hilbert's 2nd problem, 1900).
3. Find an algorithm to determine whether or not a polynomial equation with integer coefficients has an integral solution (Hilbert's 10th problem, 1900; in Hilbert's formulation there seems no doubt that such an algorithm exists).
4. Is there an algorithm which decides whether or not certain mathematical statements can be proven? (Entscheidungsproblem, 1928).

The (admittedly vaguely formulated) Entscheidungsproblem (German for "decision problem") appears as a meta-problem within this list – expressing the desire that problems like 1.-3. can be solved in principle. Shockingly, the problems 1.-3. all turned out to hit some principle limitations to what is computable or provable, thereby answering the Entscheidungsproblem in the negative.

In 1931 Gödel proved that there are undecidable (in the sense of unprovable) statements in every consistent axiomatic formal system which is rich enough to develop basic arithmetic (Gödel's first incompleteness theorem). In his second incompleteness theorem he then showed that consistency itself belongs to those intrinsically undecidable statements. About five years later, Church and Turing independently provided a negative answer to the Entscheidungsproblem by proving the existence of undecidable (in the sense of uncomputable) decision problems.

These results show that there are two different albeit related flavors of undecidability: axiomatic and algorithmic.

1.1 Useful and paradoxical self-references

An essential concept underlying many undecidability or incompleteness proofs is self-reference. The way it is typically used (and useful) is in proofs by

contradiction which are sometimes reminiscent of logical paradoxes like "this statement is false". There are umpteen similar paradoxes:

- *Russel's paradox* was prominent in mathematics, in particular before the axiomatization of set theory: "the set of all sets which do not contain themselves" can neither contain nor not contain itself.
- *The barber's paradox* (again mentioned by Russel) defines the barber as "one who shaves all those, and those only, who do not shave themselves." Does the barber shave himself?
- *Berry's paradox* (guess by whom?) "the smallest positive integer that cannot be specified in less than a thousand words".

Self-referential techniques in mathematics (like the one Cantor used to prove that the reals are not countable) often come with some form of diagonalisation. The method is thus often called diagonal method or just diagonalisation.

When boiled down to a simple self-referential statement, Gödel's proof is based on formalizing the statement "this sentence is not provable": if the statement is provable, then the underlying theory is inconsistent. If it is not provable, however, the theory is incomplete and there is a true statement which cannot be deduced. The form of opposition between consistency and completeness which appears in Gödel's results is often popularized and applied to other fields where self-referential puzzles appear. On this level (which we'll quickly leave again) it seems tempting to think about consciousness, the quantum measurement problem, etc.

The poet Hans Magnus Enzensberger even devoted a poem "Hommage an Gödel" (in Elexiere der Wissenschaft) to Gödel's results in which he writes (sorry, German):

> In jedem genügend reichhaltigen System
> lassen sich Sätze formulieren,
> die innerhalb des Systems
> weder beweis- noch widerlegbar sind,
> es sei denn das System
> wäre selber inkonsistent.
>
> Du kannst deine eigene Sprache
> in deiner eigenen Sprache beschreiben:
> aber nicht ganz.
> Du kannst dein eigenes Gehirn
> mit deinem eigenen Gehirn erforschen:
> aber nicht ganz.

Notation: Before we proceed, some remarks on the used notation are in order. Here and in the following, $\mathbb{N} := \{0, 1, 2, \ldots\}$ means the natural numbers including zero. When writing $f : X \to Y$ we do not require that the function f is defined on all of X, but rather that $dom(f) \subseteq X$. If the function is defined on all of X, we will call it a *total* function, whereas if $dom(f)$ is allowed (but not required) to be a proper subset of X, we will call it a *partial* function.

2 What's an algorithm?

Lecture 1

Solving problems like Hilbert's 10th problem or the Entscheidungsproblem in the negative required to formalize the notion of an algorithm (for which Hilbert used the German word "Verfahren") since only then one can prove that no algorithm can exist. During the thirties several mathematicians came up with different but in the end equivalent ways of making this notion precise: Gödel, Kleene, Church, Post and Turing invented and studied recursive functions, λ-calculus and, of course, Post-Turing machines. Having formalized the notion of an algorithm, the (negative) answer to the Entscheidungsproblem turned out to be right around the corner.

When saying something like "we have an algorithm which computes this function" or "there is no algorithm computing this function ", what do we mean? What's an algorithm? Here, our primary interest is in functions computable by algorithms rather than in the notion algorithm itself. Occasionally, we will just say "computable" and mean "computable by an algorithm". Before we formalize this, let's try to identify some essential ingredients. We will mostly restrict ourselves to functions of the form $f : \mathbb{N} \to \mathbb{N}$. Computing with reals or other "continuous" objects seems only realistic when they can be encoded into integers, i.e., specified by finite means. Similarly, if the (co-)domain is a countable set different from $\mathbb{N}$ or if the object of interest is a relation rather than a function, we assume that we can invoke an encoding into $\mathbb{N}$, i.e., an enumeration of all possible inputs and outputs.

An algorithm for computing f is supposed to take an input $x \in \mathbb{N}$ to an output $f(x)$ if x is in the domain of f. But how?

Definition (Algorithm - informal)**.** *An algorithm is an effective procedure which allows a finite description. It is based on finitely many instructions which can be carried out in a stepwise fashion.*

This informal definition tries to capture some essential points:

- *Finiteness* of the algorithm in the sense that it has a finite description within a fixed language (for which candidates will be discussed in the following lectures) which in turn is based on a finite alphabet. Note that this does not mean that there is a fixed, finite bound on the number of instructions, or on the size of the input.

- *Discreteness* in the sense that the instructions are carried out one after the other – there is no "continuous" or "analog" process necessary.
- *Effectiveness* meaning that the instructions can be carried out by a "simple" mechanistic machine or fictitious being. Moreover, the algorithm has to return the outcome after a finite number of steps if the function it is supposed to compute is defined for the input under consideration.

There are several points here one might want to argue about, refine or discuss. An important one is whether or not we want to impose bounds on the spatial or temporal resources, i.e., on the amount of scratch paper (memory) and the duration/length of the computation. Do we for instance want to say that "f can be computed by an algorithm" only if for any input we can say beforehand that the computation doesn't require more resources than a certain bound? The set of functions "computable by algorithm" will depend on this point and we will impose no such bounds on spatial or temporal resources. In fact, we allow and even require the computation to loop forever if the input is not in the domain of f.

Needless to say, assuming unbounded resources in time and space is an impractical idealization. However, if one is interested in the ultimate limitations to what is computable, then this idealized point of view makes sense.

One thing we haven't specified yet is whether or not an algorithm is allowed to use probabilistic methods. Regarding this point it turns out that it doesn't really matter: as long as we stick to the above mentioned assumptions, a function computable by a probabilistic procedure will also be computable by a deterministic one – we just have to follow all the (finitely many) branches which appear in the probabilistic scheme.

Before even fixing a framework or language for describing algorithms we can see that requiring the description to be finite already implies the existence of uncomputable functions, i.e., functions which cannot be computed by any algorithm:

Theorem (uncomputable functions)**.** *There exist functions $f\colon \mathbb{N} \to \{0,1\}$ which cannot be computed by any algorithm.*

Proof. Recall that we require the alphabet on which the description of algorithms should be based on to be finite. We can thus introduce a "lexicographic order" which allows us to enumerate all finite descriptions, including those corresponding to valid algorithms: we begin with all description of length one, continue with all of length two, and so one. The set of algorithms has therefore cardinality at most $\aleph_0$ - the one of the natural numbers. On the contrary, the set of all functions $f\colon \mathbb{N} \to \{0,1\}$ can be identified with the set of all subsets of $\mathbb{N}$ and has therefore cardinality $2^{\aleph_0}$. By Cantor's argument

there cannot be a surjection from the former set to the latter (this would enumerate the reals). Hence, there are functions which do not correspond to any algorithm. □

This argument shows that almost every functions is not computable by an algorithm – independent of how cleverly we choose the language in which we describe algorithms, if only the description is finite. However, one has to admit that "almost every function" is of almost no interest. Functions which we encounter are specific rather than random and they are interesting because they have particular properties or relations to other objects. After having introduced a particular framework for describing algorithms in the next lectures, we will come across more specific examples of uncomputable functions...

Before we do so, let's look at the following three examples:

1. A function $f_1 \colon \mathbb{N} \to \{0,1\}$ which is defined as $f_1(x) = 1$ iff exactly x consecutive ones occur somewhere in the decimal expansion of π and $f_1(x) = 0$ otherwise.

2. A function $f_2 \colon \mathbb{N} \to \{0,1\}$ for which $f_2(x) = 1$ iff at least x consecutive ones occur somewhere in the decimal expansion of π.

3. A function $f_3 \colon \mathbb{N} \to \mathbb{N}$ which is computed by going through the decimal expansion of π, looking for the first instance of exactly x consecutive ones, and returning the position of the first digit of this occurrence as the value of $f_3(x)$.

Although f_1 seems to be a properly defined function, it is difficult to imagine an algorithm computing it – unless π is sufficiently random such that all sequences are guaranteed to appear. In fact, if f_1 is not constant it may well be that no algorithm exists since a finite procedure may be doomed with cases where $f_1(x) = 0$.

The function f_2, at first glance, appears to have a similar problem. However, if we distinguish between the existence of an algorithm and the verification of its correctness, we realize that there exists a very simple algorithm for computing f_2: it is either a constant function or a simple step function, solely characterized by a single integer. In both cases there is an algorithm for computing f_2, the problem is just to find the correct one.

Finally, f_3 is already specified by giving an algorithmic procedure. It is, however, not clear whether this really defines a function on all of $\mathbb{N}$. What can we learn from these examples?

1. One can imagine functions whose definition may already be in conflict with the finiteness of any hypothetical algorithm computing it.

2. Finding an algorithm and proving that one exists are two different things. In particular, every function on $\mathbb{N}$ which is piecewise constant and defined using only finitely many regions is computable by algorithm, regardless whether or not we can compute it in practice. Note that this point of view also means that every function with a finite domain is computable.

3. For some statements aiming at defining a function it seems difficult (or impossible) to find out whether they actually do define a function everywhere.

Two final remarks on these examples: it may turn out that π is not a good choice for the first two examples (f_1 and f_2 may be provably constant). In that case replace π by $e + \pi$, etc.

One possible reaction to the above examples, especially to f_1, is to ask whether this is a proper definition to start with. Clearly, it invokes tertium non datur (the law of excluded middle) which we will assume throughout the lecture. Moreover, one has to admit that certain schools of thought, like mathematical constructivism (not exactly a fan club of the law of excluded middle either), are generally reluctant to objects which are defined without a constructive procedure. We are, however, not going to enter this discussion here...

3 Turing machines and busy beavers

Lecture 2

In the following lectures we will introduce two in the end equivalent ways of formalizing the idea of an algorithm: Turing machines and recursive functions.

A *Turing machine* is a (fictitious or Lego build) device consisting out of

- a *tape* which is infinitely extended in both directions, divided into equally spaced cells each of which contains a symbol from a finite alphabet T. We will restrict ourselves to cases where $T = \{0, 1\}$ and say that the cell is "blank" if it contains a zero,

- a *head* which sits on top of a single cell above the tape and is (i) capable of reading and rewriting the symbol of that cell and (ii) moves one cell to the right or left afterwards,

- a finite set of *internal states* $Q = \{0, \ldots, n\}$,

- a list of *instructions* which determine the next step: depending on the symbol read in the current cell and on the current internal state they determine which symbol to write, in which direction to move and which new internal state to take on.

We will use the convention that the initial state is $q = 0$ and the last state $q = n$ is the *halting state* - the only state upon which the machine halts. Since this results in n active states, the machine is called an n-state Turing machine. Mathematically, the set of instructions characterizing a Turing machine is a map

$$M : T \times (Q \backslash \{n\}) \to T \times Q \times \{R, L\},$$

where $\{R, L\}$ are the directions the head can move. We will sometimes write $M^{(n)}$ if we want to make the number of active internal states explicit. Counting the number of possibilities we get that there are $(4n + 4)^{2n}$ n-state Turing machines.

Example 1. A TM $M^{(k+1)}$ which writes k ones onto a blank tape and then halts above the leftmost one. $M\colon (0, q) \mapsto (1, q + 1, L)$ for $q = 0, \ldots, k - 1$ and $M : (0, k) \mapsto (0, k + 1, R)$.

Example 2. A runaway TM which never halts is for instance obtained by setting $(t, q) \mapsto (t, q, R)$ for all t, q.

There are quite a number of variations on the theme "Turing machine", i.e., different (albeit eventually equivalent) definitions. The most common ones are mentioned at the end of this lecture...

3.1 Functions computable by Turing machines

In order to talk about Turing machines as devices which compute functions of the form $f\colon \mathbb{N}^k \to \mathbb{N}$, we need to specify some conventions about how input and output are represented. We will use "unary" encoding for both of them. That is, a number $x \in \mathbb{N}$ will be represented by $x + 1$ consecutive 1s on the tape with the rest of the tape blank (e.g., 2 would correspond to $0 \ldots 01110 \ldots 0$). Similarly, $(x_1, \ldots, x_k) \in \mathbb{N}^k$ will be represented by k such blocks of 1s separated by single zeros (e.g. $(0, 2)$ would be $0 \ldots 0101110 \ldots 0$).

A Turing machine $M_f^{(n)}$ is then said to compute the function $f\colon \mathbb{N}^k \to \mathbb{N}$ iff the machine starting with the head placed on the leftmost 1 of the unary encoding of $x \in \mathbb{N}^k$ eventually halts on the leftmost 1 of the encoded $f(x)$ if $x \in \mathrm{dom}(f)$ and it never halts if $x \notin \mathrm{dom}(f)$.

Definition (Turing computable functions)**.** *A function $f\colon \mathbb{N}^k \to \mathbb{N}$ is called* Turing computable *iff there is an n-state Turing machine (TM) for some finite n which computes f in the sense that the Turing machine halts for every input $x \in \mathrm{dom}(f)$ with the tape eventually representing $f(x)$ and it doesn't halt if $x \notin \mathrm{dom}(f)$. Here, input and output are supposed to be encoded in the above specified unary way and at the start and (potential) end of the computation the head of the TM should be positioned above the leftmost non-blank symbol of the tape.*

The crucial point here is finiteness of the number n of internal states. This quantifies the length of the algorithm - which we require to be finite.

Example 3. the *successor function* $s(x) = x + 1$ can be computed by the following 2-state Turing machine $M^{(2)}_{x+1}$. $(1,0) \mapsto (1,0,R)$, $(0,0) \mapsto (1,1,L)$, $(1,1) \mapsto (1,1,L)$, $(0,1) \mapsto (0,2,R)$.

Example 4. the *zero function* $z(x) = 0$ can be computed by a 2-state TM: $(1,0) \mapsto (0,0,R)$, $(0,0) \mapsto (1,1,R)$, $(0,1) \mapsto (0,2,L)$.

Example 5. the following 5-state TM $M^{(5)}_{2x}$ implements $x \mapsto 2x$: $(0,0) \mapsto (0,3,R)$, $(1,0) \mapsto (0,1,L)$, $(0,1) \mapsto (1,2,R)$, $(1,1) \mapsto (1,1,L)$, $(0,2) \mapsto (1,0,R)$, $(1,2) \mapsto (1,2,R)$, $(0,3) \mapsto (0,3,L)$, $(1,3) \mapsto (0,4,L)$, $(0,4) \mapsto (0,5,R)$, $(1,4) \mapsto (1,4,L)$.

3.2 Composition of Turing machines

Let $M^{(n_f)}_f$ and $M^{(n_g)}_g$ be two TMs with n_f, n_g internal states, computing functions f and g respectively. Then we can a define a new $(n_f + n_g)$-state TM $M^{(n_f+n_g)}_{gf}$ via

$$M_{gf}(t,q) := \begin{cases} M_f(t,q), & q < n_f \\ M_g(t, q - n_f), & q \geq n_f \end{cases}$$

Its action will be such that it first computes $f(x)$ and then uses the resulting output as an input for g. Hence, M_{gf} computes the concatenation corresponding to $x \mapsto g(f(x))$ for which we will also write $gf(x)$. Note that the possibility of concatenating two TMs in this way builds up on our requirements that the output of a computation has to be encoded in unary on the tape and that the TM (if ever) halts with the head positioned on the leftmost 1.

3.3 Rado's theorem and busy beavers

Let us assign a number $B(M) \in \mathbb{N}$ to every Turing machine M by considering its behavior when run on an initially blank tape. We set $B(M) := 0$ if M never halts and $B(M) := b$ if it halts and the total number of (not necessarily consecutive) 1s eventually written on the tape is b. Based on this we can define the busy beaver function

$$\mathrm{BB}\colon \mathbb{N} \to \mathbb{N},\ \mathrm{BB}(n) := \max\{B(M) \mid M \in \{M^{(n)}\}\}$$

as the largest number of 1s eventually written on an initially blank tape by any n-state TM which halts. Note that the function is well-defined since the maximum is taken over a finite set. Not surprisingly, BB is strictly increasing in n:

Lemma (Monotonicity of busy beavers)**.** $\mathrm{BB}(n+1) > \mathrm{BB}(n)$ *for all* $n \in \mathbb{N}$*.*

Proof. Denote the TM which achieves $\mathrm{BB}(n)$ by $M^{(n)}$. Based on this we can define a $(n+1)$-state TM $M^{(n+1)}$ whose instructions equal those of $M^{(n)}$ for all internal states $q < n$ and which in addition follows the rule $(t, n) \mapsto (1, n+1-t, R)$. By construction $\mathrm{BB}(n) + 1 = B(M^{(n+1)}) \leq \mathrm{BB}(n+1)$. □

This leads us to a common property of all Turing computable functions – they cannot grow as fast as (or faster than) BB:

Theorem (Rado '62)**.** *Let* $f : \mathbb{N} \to \mathbb{N}$ *be any function which is Turing computable by a* k*-state TM* M_f*. Then for all* $x > 2k + 16$ *for which* f *is defined we have* $f(x) < \mathrm{BB}(x)$*.*

Proof. We utilize concatenation of the above discussed examples and define a $(k+n+9)$-state TM (slightly abusing notation) via $M_{f(2n+1)} := M_f^{(k)} M_{x+1}^{(2)} M_{2x}^{(5)} M^{(n+2)}$ with $M^{(n+2)}$ being the TM which writes $n+1$ consecutive 1s. Running $M_{f(2n+1)}$ on the blank tape then produces $f(2n+1)+1$ consecutive ones before halting. Thus, $f(2n+1) < \mathrm{BB}(k+n+9)$. Moreover, monotonicity of BB implies $\mathrm{BB}(k+n+9) \leq BB(2n+1)$ if $k+7 < n$. Similar applies if we construct a TM $M_{f(2n)}$ in which case we obtain $f(2n) < \mathrm{BB}(2n)$ if $k+6 < n$. □

From the proof of Rado's theorem we see that an analogous statement would still hold true if we would require a single block of consecutive 1s in the definition of BB, rather than counting all 1s on the tape.

Corollary (Busy beavers are elusive)**.** *The busy beaver function is not Turing computable.*

Proof. If it were, then there would be a $k \in \mathbb{N}$ and a k-state TM computing BB so that by Rado's theorem $\mathrm{BB}(x) < \mathrm{BB}(x)$ for all sufficiently large x. □

This is the formalized version of the following more vague statement: if $\mathrm{BB}(x)$ is the largest finite number which can be written by an algorithm of length x, then there cannot be a single, finite algorithm which computes $\mathrm{BB}(x)$ for all x.

The fact that BB is not Turing computable doesn't mean that $\mathrm{BB}(x)$ cannot be computed for given x. Rado's theorem just tells us that the complexity of the TM has to increase unboundedly with x. In fact, $\mathrm{BB}(x)$ is known for (very) small values of x: for $x = 1, 2, 3, 4$ we get $\mathrm{BB}(x) = 1, 4, 6, 13$. While this doesn't look very impressive yet, for $x = 5, 6$ one only knows lower bounds which are $\mathrm{BB}(5) \geq 4098$ and $\mathrm{BB}(6) \geq 3.5 \times 10^{18267}$.

3.4 First occurrence of the halting problem

Lecture 3

Assume we had an algorithm (using the informal sense of the word) which could decide whether or not a given TM eventually halts upon a given input, e.g., the blank tape. This would give us an algorithm for computing the busy beaver function: just go through all the halting TMs (which we can identify by assumption), count the number of 1s written by each of them and take the maximum.

So if there is no algorithm for computing BB, then there cannot be an algorithm which decides whether or not a given TM is going to halt. The latter decision problem is known as the halting problem - a repeating encounter in the following lectures.

Before we continue, here is another (the 'standard') heuristic argument for why the halting problem cannot be decidable: assume there is an algorithm (again in the informal sense) which is capable of deciding whether some algorithm C halts upon input I. Using this as a subroutine we can then construct another algorithm P which, given the code of an algorithm C as input, halts iff the algorithm C runs forever upon input of its own source code $I = C$. Now let P run with input $C = P$. Then P halts iff it runs forever—showing that the assumed algorithmic decidability of the halting problem leads to a contradiction.

3.5 Variations

Finally, we list some common variations regarding the definition of Turing machines or Turing computable functions:

1. The tape is infinite only in one direction.
2. The TM can only either write a new symbol onto the tape or move one step.
3. The output is not encoded in unary but given by the total number of ones written after halting.
4. There is more than one halting state (e.g. called accept and reject state).
5. There is more than one non-blank symbol available for each cell.

All these variations do not alter the class of functions which are Turing computable. One which does, is for instance the constraint that the TM head is only allowed to move to the right. This leads to an effective restriction of the memory and thus restricts the set of computable functions.

4 Primitive recursion

Early pieces of recursion theory can be found in the work of Dedekind with the beautiful title "*Was sind und was sollen die Zahlen*".

In this section we will begin to develop another formalization of the notion of "effectively computable functions", this time without referring to a particular type of machine or automaton. The basic idea will be to consider a class of functions which can be obtained in finitely many steps from a small set of "simple" basic functions by following "simple" rules.

Consider for instance the chain successor function (i.e., addition by one), addition, multiplication and exponentiation. These functions can all be recursively defined in terms of the previous, simpler function. So, the successor functions seems to be a reasonable basic function and recursion seems to be a reasonable rule for constructing more sophisticated functions.

As before we will restrict ourselves to functions of the form $f\colon \mathbb{N}^k \to \mathbb{N}$. The set of basic functions comprises the following three:

1. The *successor function* $s(x) := x + 1$ for all $x \in \mathbb{N}$.
2. The *zero function* $z(x) := 0$ for all $x \in \mathbb{N}$.
3. The *projection* or *identity functions* defined for all $n > 0$ and $k \in \{1, \dots, n\}$ as $\mathrm{id}_k^n(x_1, \dots, x_n) := x_k$.

From these we want to construct other functions using the following rules:

Composition/substitution Given a set of functions $g_k\colon \mathbb{N}^n \to \mathbb{N}$ with $k = 1, \dots, m$ and $f\colon \mathbb{N}^m \to \mathbb{N}$ we define a new function $\mathrm{Cn}[f, g_1, \dots, g_m]\colon \mathbb{N}^n \to \mathbb{N}$ by $\mathrm{Cn}[f, g_1, \dots, g_m](x) := f\big(g_1(x), \dots, g_m(x)\big)$.

Primitive recursion Given $f\colon \mathbb{N}^m \to \mathbb{N}$ and $g\colon \mathbb{N}^{m+2} \to \mathbb{N}$ we define a new function $\mathrm{Pr}[f, g]\colon \mathbb{N}^{m+1} \to \mathbb{N}$ by $\mathrm{Pr}[f, g](x, 0) := f(x)$ and $\mathrm{Pr}[f, g](x, y + 1) := g\big(x, y, Pr[f, g](x, y)\big)$.

Definition (Primitive recursive functions)**.** *A function $f\colon \mathbb{N}^k \to \mathbb{N}$ is called* primitive recursive *iff it can be obtained from the above basic functions by finitely many applications of the above rules* Cn *and* Pr.

By definition the class of primitive recursive functions, which we will denote by PR, is the smallest class of functions which is (i) closed under primitive recursion and composition and (ii) contains the basic functions s, z and id. Moreover, primitive recursive functions are all total functions (i.e., defined for every input) since the basic functions are total and this property is preserved under composition and primitive recursion.

Undeniably, primitive recursive function are "effectively computable". At first glance, the definition of PR may seem to be rather restrictive. At

second glance, however, the class contains pretty much everything and only a third glance will reveal... ah, sorry this is part of the next lecture. Anyhow, talking about "primitive" recursion already suggests that there is something beyond. Let us discuss some examples:

Example 2 (Addition). add: $\mathbb{N}^2 \to \mathbb{N}$ can be defined by primitive recursion since $\text{add}(x, 0) := x,\ \text{add}(x, y+1) := \text{add}(x, y) + 1$. A formal way of defining it would be $\text{add} := \text{Pr}\left[\text{id}_1^1, \text{Cn}[s, \text{id}_3^3]\right]$.

Example 3 (Multiplication). We can recursively define a "product function" by $\text{prod}(x, 0) := 0,\ \text{prod}(x, y+1) := \text{prod}(x, y) + x$. More formally, $\text{prod} := \text{Pr}\left[z, \text{Cn}[\text{add}, \text{id}_1^3, \text{id}_3^3]\right]$. Since $\text{add} \in \text{PR}$ we have that $\text{prod} \in \text{PR}$.

Example 4 (Exponentiation). This can be seen to be primitive by defining it via $\exp(x, 0) := 1,\ \exp(x, y+1) := \text{prod}(\exp(x, y), x)$ and using that $\text{prod} \in \text{PR}$.

Example 5 (Predecessor). This one we define via $\text{pred}(0) := 0,\ \text{pred}(y+1) := y$.

Example 6 (Cut-off subtraction).

$$x \dot{-} y := \begin{cases} x - y & if x \geq y \\ 0 & if x < y \end{cases}$$

can be defined recursively via $f(x, 0) := x,\ f(x, y+1) := \text{pred}(f(x, y))$.

Example 7 (Sign function). $\text{sgn}(x) := 1 \dot{-} (1 \dot{-} x)$ defines a function sgn: $\mathbb{N} \to \{0, 1\}$ which is 1 iff the argument is strictly positive.

Note that the formal way of writing a primitive recursive function (using Pr[...] and Cn[...] etc.) provides a single-line expression – this is called derivation. Clearly, there are infinitely many derivations for each function. So it makes sense to distinguish between a function and its derivation (which is nothing but one out of many algorithms for computing the function).

4.1 Primitive recursive relations, sets and predicates

A subset $S \subseteq \mathbb{N}^k$ can be characterized by its characteristic function $\chi: \mathbb{N}^k \to \{0, 1\}$ for which $\chi(x) = 1$ iff $x \in S$. For $k > 1$ we can regard S as a relation between k objects. Similarly, we can identify any predicate which assigns a truth value (TRUE or FALSE) to a k-tuple of numbers in $\mathbb{N}$ with a relation which is satisfied iff the predicate is TRUE. Hence, we can again assign a characteristic function to any such predicate. The predicate $x > y$ for instance would then correspond to the characteristic function

$$\chi_>(x, y) := \begin{cases} 1 & \text{if } x > y, \\ 0 & \text{if } x \leq y \end{cases}.$$

Characteristic functions of relations and predicates will throughout be total functions, i.e., we will in particular refrain from considering cases where predicates are not applicable.

Definition (Primitive recursive relations, sets and predicates)**.** *A relation, set or predicate is called* primitive recursive *iff the corresponding characteristic function is a primitive recursive function.*

Using the above examples of primitive recursive functions we can now easily see that the predicates "$>$" and "$=$" are primitive recursive, since $\chi_>(x, y) = \operatorname{sgn}\big(x \dot{-} y\big)$ and $\chi_=(x, y) = 1 \dot{-} \big(\operatorname{sgn}(x \dot{-} y) + \operatorname{sgn}(y \dot{-} x)\big)$.

Proposition. *Let P, Q be two primitive recursive predicates. Then $P \vee Q$, $P \wedge Q$ and $\neg Q$ are primitive recursive as well.*

Proof. We have to show that the characteristic functions of the new predicates are in PR if the initial ones were. This is seen from

$$\chi_{P\wedge Q} = \chi_P \cdot \chi_Q, \chi_{P\vee Q} = (\chi_P + \chi_Q) \dot{-} (\chi_P \cdot \chi_Q) \text{ and } \chi_{\neg Q} = 1 \dot{-} \chi_Q.$$

□

Proposition (Definition by cases)**.** *Let $g, h \colon \mathbb{N}^k \to \mathbb{N}$ be primitive recursive functions and P a primitive recursive predicate on $\mathbb{N}^k$. Then $f \colon \mathbb{N}^k \to \mathbb{N}$ defined as follows is primitive recursive as well:*

$$f(x) := \begin{cases} g(x) & \text{if } P(x) \\ h(x) & \text{if } \neg P(x) \end{cases}$$

Proof. $f(x) = g(x)\chi_P(x) + h(x)\chi_{\neg P}(x)$ is obtained from primitive recursive building blocks. □

Clearly, the same works for more than two cases.

4.2 Bounded operations

Lecture 4

Proposition (Iterated products and sums)**.** *If $f \colon \mathbb{N} \times \mathbb{N}^k \to \mathbb{N}$ is primitive recursive, then so are the functions $g(y, x) := \sum_{t=0}^{y} f(t, x)$ and $h(y, x) := \prod_{t=0}^{y} f(t, x)$.*

Proof. Both g and h can be obtained by primitive recursion: $g(0, x) := f(0, x)$, $g(y + 1, x) := g(y, x) + f(y + 1, x)$ and similarly $h(0, x) := f(0, x)$, $h(y + 1, x) = h(y, x)f(y + 1, x)$. □

Note that setting $g(0, x) = 0$ and $h(0, x) = 1$ in the above proof would have led to a sum/product starting at $t = 1$.

Corollary (Bounded quantifiers)**.** *If the predicate P is primitive recursive on $\mathbb{N} \times \mathbb{N}^k$, then so are the predicates $\forall t \le y \colon P(t, x)$ and $\exists t \le y \colon P(t, x)$.*

Proof. The corresponding characteristic functions are $\prod_{t=0}^{y} \chi_P(t, x)$ and $\operatorname{sgn}\left(\sum_{t=0}^{y} \chi_P(t, x)\right)$ respectively. These are primitive recursive by the preceding proposition. □

We can now use these observations in order to show that "bounded minimization" preserves the property of being primitive recursive.

Proposition (Bounded minimization)**.** *Let P be a primitive recursive predicate on $\mathbb{N} \times \mathbb{N}^k$. The function $\mu[P] \colon \mathbb{N} \times \mathbb{N}^k \to \mathbb{N}$ defined by $\mu[P](y, x) := \min\{t : P(t, x) \wedge t < y\}$, with the minimum over an empty set set to y, is primitive recursive.*

Proof. Consider the function $f(y, x) := \sum_{n=0}^{y} \prod_{t=0}^{n} \chi_{\neg P}(t, x)$. This is primitive recursive since P is. Suppose there is a smallest t, call it t_x, for which $P(t_x, x)$ is true. Then $\prod_{t=0}^{n} \chi_{\neg P}(t, x) = 1$ iff $n < t_x$. Summing over n then yields t_x so that a primitive recursive construction of $\mu[P]$ can be given using definition by cases and a bounded existential quantifier by

$$\mu[P](y, x) = \begin{cases} f(y, x) & \text{if } \exists t < y \colon P(t, x) \\ y & \text{otherwise} \end{cases}$$

□

In a similar vein, we can show that bounded maximization $\max\{t : t < y \wedge P(t, x)\}$ is primitive recursive if P is. In that case we define the maximum over an empty set to be zero. We will finally apply these findings to some more examples which will be of use later:

Example 8 (Prime numbers). the predicate primality of a number $x \in \mathbb{N}$, which we denote by $\pi(x)$, can be expressed by $(1 < x) \wedge \forall u < x \forall v < x \colon uv \neq x$ and is thus primitive recursive. The function $f \colon \mathbb{N} \to \mathbb{N}$ which outputs the next prime is then primitive recursive as well since we can define it by bounded minimization $f(x) := \min\{y : (y \leq x! + 1) \wedge (y > x) \wedge \pi(y)\}$ (using that the factorial function is in PR). This in turn allows us to give a primitive recursive construction of a function $f_\pi \colon n \mapsto n$th prime number. We just set $f_\pi(0) = 2$ and $f_\pi(y + 1) = f\big(f_\pi(y)\big)$.

Example 9 (Logarithms). We will use two logarithm-type functions of the form $\mathbb{N}^2 \to \mathbb{N}$ both of which are primitive recursive by construction:

$$\operatorname{lo}(x, y) := \begin{cases} \max\{z \leq x \mid \exists c \leq x \colon cy^z = x\} & \text{if } x, y > 1 \\ 0 & \text{otherwise} \end{cases}$$

Example 10.

$$\operatorname{lg}(x, y) = \begin{cases} \max\{z \leq x \mid y^z \leq x\} & \text{if } x, y > 1 \\ 0 & \text{otherwise} \end{cases}$$

Example 11 (Quotients and remainders). Given $x, y \in \mathbb{N}$ we can find $q, r \in \mathbb{N}$ so that $x = qy + r$. This become unambiguous if we choose the largest such q. Both, the quotient $q =: \text{quo}(x, y)$ and the remainder $r =: \text{rem}(x, y)$ can be obtained in a primitive recursive way via:

$$\text{quo}(x, y) = \begin{cases} \max\{z \leq x \mid yz \leq x\} & \text{if } y \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

and $\text{rem}(x, y) = x \dot{-} \big(\text{quo}(x, y)y\big)$.

5 Gödel numbers, codes, indices

Nonnegative integers, i.e., $\mathbb{N}$, build the framework for recursion theory/computability theory (and, as we will see later, parts of mathematical logic). Isn't this too narrow? After all, most things we really want to compute and deal with are based on real or even complex numbers, matrices, operators, functions, not to mention pictures, audio files and minesweeper configurations... in other words: not very natural numbers. The reason why dealing with $\mathbb{N}$ appears to be sufficient, is that the objects we are eventually interested in are countable so that we can assign a number (which one calls index, code or Gödel number) $n \in \mathbb{N}$ to each object in a way which allows us to effectively identify the object from a given n. This will be the content of this lecture.

Complaint from the backseats: "the reals aren't countable!"

True. But when we're using a specific real number $x \in \mathbb{R}$ as an input for a function f say, then in order to be able to compute the value $f(x)$, we tacitly assume that x is computable in the first place. That is, although f may abstractly be defined on all of $\mathbb{R}$ we will usually evaluate it only on computable numbers (i.e., those whose decimal expansion can be effectively computed to arbitrary precision) or, when we're in a more surreal mood, we will act with it on definable numbers. In any case this is a countable set. Expressed differently, the input of a computation should be an object which we can specify by finite means. Arguing like this makes the restriction to $\mathbb{N}$ much less restrictive since any set of practical interest then seems to reveal a countable nature. In fact, one may, as some constructivists do, even think about using the computable numbers instead of the reals as the basis for mathematical analysis. We wont go in this direction and not even justify further the focus on countable sets – we will just deal with countable sets only.

So consider a countable set A which we call the alphabet together with an injective assignment $A \to \mathbb{N}$ of a number to each element in A. This assignment maps any finite string $a_1 \cdots a_k$ where each $a_j \in A$ to an element of $\mathbb{N}^k$. As we will see below, the latter can be mapped injectively into $\mathbb{N}$. Such mappings, together with their inverses are the main content of the

remaining part of this lecture. The main point will be that these mappings can be chosen primitive recursive. Before we discuss two prominent examples, a remark concerning terminology: in our context, an injective mapping into $\mathbb{N}$ is often (but not entirely coherently) called an encoding and the final number the code.

Similarly, a surjective map from $\mathbb{N}$ tends to be called an indexing, Gödel numbering or enumeration (the latter is sometimes reserved for bijective maps) and the corresponding number is called an index or Gödel number. We will further specify the meaning of "index" in the context of general recursive functions.

5.1 Multiplicative encoding

Any string of numbers of the form $(a_0, \dots, a_{n-1}) = a \in \mathbb{N}^n$ can be encoded into a single number $\alpha \in \mathbb{N}$ in an injective way by exploiting the fundamental theorem of arithmetic, i.e., the uniqueness of prime-number factorization. One variant of such an encoding is $(a_0, \dots, a_{n-1}) \mapsto \alpha = 2^n \prod_{i=1}^{n} f_\pi(i)^{a_{i-1}}$, where $f_\pi(i)$ is the i'th prime number starting with $f_\pi(0) = 2$. As a consequence of the results of the last lecture, the mapping $a \mapsto \alpha$ is a primitive recursive function. Conversely, if $\alpha \in \mathbb{N}$ is such an encoding, then $\alpha \mapsto n = \mathrm{lo}(\alpha, 2)$ as well as $\alpha \mapsto a_j = \mathrm{lo}(\alpha, f_\pi(j+1))$, i.e., deducing the length and any of the original components, is achievable by primitive recursive functions. In a similar primitive recursive vein, we can expand the string on the level of its code for instance by $\alpha \mapsto 2\alpha f_\pi(\mathrm{lo}(\alpha, 2) + 1)^b$ which results in an encoding of $(a_0, \dots, a_{n-1}, b)$.

There are several variants of this method many of which differ by how or whether the length n is encoded for convenience.

```
6 .
3 7 .
1 4 8 .
0 2 5 9
```

Figure 1: Enumeration of $\mathbb{N}^2$ giving rise to a bijective and monotonic pairing function.

5.2 Pairing functions

Another frequently used encoding is based on pairing functions $J\colon \mathbb{N}^2 \to \mathbb{N}$ which can be chosen bijective and strictly monotonic in both arguments.

Consider for instance $\mathbb{N}^2$ as a quadrant of a two-dimensional square lattice whose points with coordinates $(x, y) \in \mathbb{N}^2$ are enumerated as depicted in Fig.1. This enumeration defines a pairing function which is then given by $J(x, y) = 1 + 2 + \ldots + (x + y) + x = (x + y)(x + y + 1)/2 + x$ and thus clearly primitive recursive. Conversely, given an encoding $j \in \mathbb{N}$ we can obtain the corresponding components via bounded minimization:

$$X(j) := \min\{x \mid x \leq j \wedge \exists y \leq j \colon J(x, y) = j\},$$
$$Y(j) := \min\{y \mid y \leq j \wedge \exists x \leq j \colon J(x, y) = j\}.$$

Any longer string can then be encoded by successive use of pairing functions, e.g., $(x, y, z) \mapsto J(x, J(y, z))$. This leads to a useful Lemma:

Lemma (Enumeration of $\mathbb{N}^k$)**.** *For any $k \in \mathbb{N}$ there is a bijective function $E \colon \mathbb{N} \to \mathbb{N}^k$ so that*

1. *each of the k component functions $E_i \colon \mathbb{N} \to \mathbb{N},\ i = 1, \ldots, k$ is primitive recursive and*
2. *the inverse function $E^{-1} \colon \mathbb{N}^k \to \mathbb{N}$ is a primitive recursive function.*

6 Limitations of primitive recursion

Lecture 4

After this lecture it should be clearer why "primitive recursion" is called "primitive". We will see two arguments – one based on diagonalisation and one based on (too) rapidly growing functions – which reveal computable functions which are not primitive recursive.

The first, diagonalisation based argument is somehow less constructive, but worthwhile stating since it shows that every class of computable total functions on $\mathbb{N}$ (i.e., functions defined on all of $\mathbb{N}$) is necessarily incomplete in the sense that there are computable total functions which are not contained in it.

Let $\mathcal{F}$ be a class of functions which are

1. total functions from $\mathbb{N}$ to $\mathbb{N}$,
2. enumerable in the sense that there is a surjective map $F : \mathbb{N} \to \mathcal{F}$. We write F_x for the image of $x \in \mathrm{dom}(F)$ under F.

Proposition (Limitations of classes of enumerable, total functions)**.** *Let $\mathcal{F}$ be a class of functions as described above. The following function $g \colon \mathbb{N} \to \mathbb{N}$ is total but not contained in $\mathcal{F}$.*

$$g(x) := \begin{cases} F_x(x) + 1 & \text{if } x \in \mathrm{dom}(F), \\ 0 & \text{otherwise} \end{cases}$$

Proof. Suppose $g \in \mathcal{F}$. Then there exists a $y \in \mathbb{N}$ such that $F_y = g$. This, however, leads to the contradiction $F_y(y) = g(y) = F_y(y) + 1$, so that g cannot be contained in $\mathcal{F}$. $\square$

Since any class of "computable" functions is countable there is always an enumerating map F. Whether g is computable then depends on the computability of F. In the case of $\mathcal{F}$ being the class PR of primitive recursive functions we can obtain a computable enumeration by considering the formal representation of functions (using Pr[...] and Cn[...] etc.) together with the techniques discussed in the previous lecture. This may lead to a cumbersome but certainly computable enumerating map F, so that the foregoing proposition implies the existence of a computable total function which is not primitive recursive. For a second we may be tempted to enlarge our class of functions by just adding g to it, until we realize that the proposition will remain applicable to the extended class of functions.

Consequently, if we aim at a formalism which incorporates all computable functions we have either to accept uncomputable enumerations and/or we have to deal with partial (i.e., not necessary total) functions. The latter choice is able to circumventing the above contradiction since F_y may not be defined for y.

A different and more explicit way to see that there are computable total functions which are not primitive recursive goes back to the work of Wilhelm Ackermann. We will consider the simplified version provided by Rozsa Peter. The following three equations define a total function $A\colon \mathbb{N}^2 \to \mathbb{N}$, the *Ackermann-Peter function*:

$$\begin{aligned} A(0,y) &= y+1, \\ A(x+1,0) &= A(x,1), \\ A(x+1,y+1) &= A\big(x, A(x+1,y)\big). \end{aligned}$$

Some useful properties of the Ackermann-Peter function are:

1. Monotonicity in both arguments: $A(x,y) \le A(x,y+1)$ and $A(x,y) \le A(x+1,y)$,
2. Faster growth in first argument: $A(x,y+1) \le A(x+1,y)$,
3. $A(2,y) \ge 2y$.

Let's look at some special cases in order to get a feeling for what this function is: it yields $A(1,y) = y+2$, $A(2,y) = 2y+3$ and $A(3,y) = 2^{y+3}-3$. That is, for $x = 1,2,3$ we essentially get addition, multiplication and exponentiation w.r.t. base 2. This continues with $A(4,y)$ being $2^{(2^{(\ldots(2^{(2^2)}))})}-3$ where the "power tower" involves $y+3$ levels of 2's. So, increasing the first

argument of A basically corresponds to climbing up the ladder addition-multiplication-exponentiation-power towers, etc. The rapid growth of the function will in fact be the cornerstone for constructing a function which is not primitive recursive. To this end one defines a class $\mathcal{A}$ of functions which is "dominated" by A. More precisely, $f\colon \mathbb{N}^k \to \mathbb{N}$ is an element of $\mathcal{A}$ iff $\exists y \in \mathbb{N}\, \forall z \in \mathbb{N}^k\colon f(z) < A(z', y)$ where $z' := \max\{z_1, \ldots, z_k\}$.

Proposition (Ackermann-Peter function is not primitive recursive)**.** *Let $\mathcal{A}$ be the class of functions which is dominated by the Ackermann-Peter function as described above and define $\alpha(x) := A(x, x)$. Then* PR $\subseteq \mathcal{A}$ *but $\alpha \notin \mathcal{A}$ (and therefore $\alpha \notin$* PR*).*

Proof. Let us first check that $\alpha \notin \mathcal{A}$. If it were, there would be a $y \in \mathbb{N}$ so that $\alpha(x) = A(x, x) < A(x, y)$ for all x. Setting $x = y$ then leads to the sought contradiction which implies that $\alpha \notin \mathcal{A}$.

That PR $\subseteq \mathcal{A}$ can be shown by proving (i) the three basic functions (zero, successor and identity) are in $\mathcal{A}$, (ii) $\mathcal{A}$ is closed under composition, and (iii) $\mathcal{A}$ is closed w.r.t. primitive recursion. Here (i) is rather obvious and (ii) and (iii) follow after a couple of elementary steps by invoking the above mentioned properties of the Ackermann-Peter function. A detailed proof for (ii) and (iii) can be found here. □

Loosely speaking, functions which grow too rapidly (at least as fast as α) cannot be in PR. This is reminiscent of the busy beaver function, but with a crucial difference: although the Ackermann-Peter function is not defined in a primitive recursive way, its definition is somehow "recursive" and allows us to compute the function. That is, α grows too fast for PR but slow enough to remain computable.

If we want a formal framework for all computable functions we have thus to go beyond the set PR of primitive recursive functions. This is done by adding a third rule in addition to composition and primitive recursion, namely minimization which we describe in the next lecture. Extending the list of rules in this way then leads to the class of recursive functions (as opposed to the class of "primitive" recursive functions which is not closed w.r.t. minimization – only w.r.t. bounded minimization). That the function α is indeed a recursive function will only be shown later, when we've developed some recursive function toolbox.

How can we intuitively understand that the Ackermann-Peter function is not primitive recursive? One possibility is to consider the number of uses of primitive recursion (i.e., occurrences of Pr[...] in the formal characterization) in a primitive recursive derivation of the function $y \mapsto A(x, y)$ for a fixed value of x. Recall that for $x = 1, 2, 3, \ldots$ this essentially amounts to addition, multiplication, exponentiation, etc. for which we would use Pr[...] exactly x times. In other words, the first argument of A appears to provide a lower bound on the length of any primitive recursive derivation. Hence, the fact

that there has to be a finite such derivation for each function in PR is in conflict with $x \mapsto A(x, x)$.

6.1 Relation to modern programming languages

One can express the limitation of primitive recursion and its overcoming by general recursive functions in more modern terms of computer programming languages: the class PR corresponds to what can be computed by algorithms using the basic arithmetic operations, IF THEN ELSE, AND, OR, NOT, =, > and FOR loops of the form they are allowed in Fortran for instance (just in case anybody is familiar with ancient languages). In the PR framework IF THEN ELSE would for instance correspond to definition by cases, FOR loops to bounded optimization or quantification, etc. Functions which are computable by such algorithms are sometimes called LOOP-computable and these are exactly the primitive recursive ones.

The crucial point which is missing in the above list are WHILE loops or, equivalently, a GOTO instruction. Adding those leads to the set of WHILE-computable functions (or GOTO-computable functions, which is essentially the same). The Ackermann-Peter function turns out to be WHILE computable but, as we have seen, it is not LOOP-computable. The analog of the WHILE loop will be the (unbounded) minimization operation to be introduced in the next lecture...

7 Recursive function vs. Turing computability

In order to overcome the limitations of primitive recursion we add a third rule in addition to composition and primitive recursion: Lecture 5

Minimization For any function $f\colon \mathbb{N} \times \mathbb{N}^k \to \mathbb{N}$ define $\mathrm{Mn}[f]\colon \mathbb{N}^k \to \mathbb{N}$ via

$$\mathrm{Mn}[f](x) := \begin{cases} y & \text{if } f(y,x) = 0 \wedge \big(\forall j < y\colon f(j,x) \in \mathbb{N} \setminus \{0\}\big), \\ \text{undefined} & \text{if no such } y \text{ exists.} \end{cases}$$

Note that there are two possible reasons for $\mathrm{Mn}[f](x)$ to be undefined: (i) there is no y for which $f(y,x) = 0$ and (ii) there is such a y but $f(j,x)$ is undefined for some $j < y$.

For later use we define the closely related minimization operator $\mu : \mathbb{N}^k \to \mathbb{N}$ for any predicate P on $\mathbb{N} \times \mathbb{N}^k$ via

$$\mu[P](x) := \begin{cases} y & P(y,x) \wedge \big(\forall j < y\colon \neg P(j,x)\big) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

That is, $\mu[P](x) = \mathrm{Mn}[\chi_{\neg P}](x)$ and we will occasionally use the notation $\mu_y\big(P(y,x)\big) := \mu[P](x)$ to specify which of the variables of P is considered.

Definition (Recursive functions and relations)**.** *A partial or total function* $f\colon \mathbb{N}^k \to \mathbb{N}$ *is called* recursive *iff it can be obtained from the basic functions (zero, successor and identity/projection) by finitely many applications of the rules* Cn*,* Pr *and* Mn*.*

A relation, set or predicate is called recursive *iff the corresponding characteristic function is a recursive total function.*

We will denote the class of recursive partial functions with k arguments by $R^{(k)}$ and write $R := \bigcup_k R^{(k)}$ for the class of all recursive partial functions. One remark concerning terminology is in order: in the literature "recursive function" is often used in the sense of "recursive total function". We wont follow this practice but rather prefer to specify if a function is total or partial. Moreover, we will use the notion "partial function" so that it includes all total functions.

Note that R is the smallest class of functions which is (i) closed w.r.t. composition, primitive recursion and minimization and (ii) contains the basic functions z, s and id. The minimization operator, when applied to a recursive predicate, leads to a recursive partial function. In fact, we will see later that R can be obtained from PR by solely supplementing it with μ acting on primitive recursive predicates.

Needless to say, PR $\subset R$. Moreover, this inclusion is strict as PR only contains total functions whereas R clearly allows for functions which are not everywhere defined. We will see, however, that there are also total functions outside PR (like the Ackermann-Peter function) contained in R.

The content of this lecture is the following equivalence:

Theorem (Recursive = Turing computable)**.** *A partial function* $f\colon \mathbb{N}^k \to \mathbb{N}$ *is recursive iff it is Turing computable.*

In order to show that every recursive partial function is Turing computable, we could begin with showing that this holds for z, s and id and then prove that the class of Turing computable functions is closed w.r.t. Cn, Pr and Mn. In fact, we essentially did the first four of these steps. The last two, however, require a little bit more effort which we will skip and believe the result... We will rather spend some time on the reverse direction which will turn out to lead to additional insight into normal forms and universal functions and machines. In the remainder of this lecture we will assume that there is a TM M which computes a function $F_M\colon \mathbb{N} \to \mathbb{N}$ and we aim at showing that $F_M \in R$.

7.1 Wang encoding of the tape & head configuration

The Wang encoding is a one-to-one map between two natural numbers and any tape & head configuration.

We will assign two numbers $l, r \in \mathbb{N}$ to every configuration of tape and head (disregarding the internal states for the moment and only taking care of

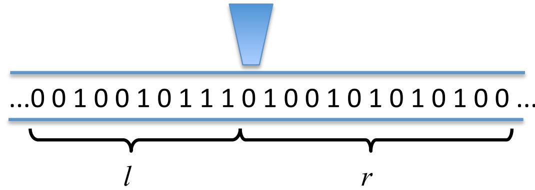

Figure 2: The Wang encoding is a bijective map between two natural numbers (l, r) and any tape and head configuration of a Turing machine.

the position of the head above the tape). To this end, we regard everything left from the head position as a binary representation of a number l and everything right from that (including the head position itself) as a binary representation which is read backwards for a number r. So if a Turing machine (TM) computes $x \mapsto f(x)$ this translates to $(l, r) = (0, 2^{x+1} \dot{-} 1) \mapsto (0, 2^{f(x)+1} \dot{-} 1)$. Recall that by the convention we chose, every admissible initial and final configuration has to be of the form $(l, r) = (0, 2^{x+1} \dot{-} 1)$ for some $x \in \mathbb{N}$. If the TM halts with the tape in a different configuration, then the function value will be undefined. So checking whether or not a Wang encoding (l, r) corresponds to a halting configuration with a defined function value, can be done by the predicate

$$l = 0 \wedge r = 2^{lg(r,2)+1} \dot{-} 1.$$

From the pair (l, r) we can determine the scanned symbol beneath the head as $\mathrm{rem}(r, 2)$, i.e., the remainder when we divide r by 2. If the head replaces the scanned symbol and writes some $t \in \{0, 1\}$ instead, the Wang encoding changes according to $l \mapsto l$ and $r \mapsto (r + t) \dot{-} \mathrm{rem}(r, 2)$.

Moving the head one site to the left leads to $l \mapsto \mathrm{quo}(l, 2)$ and $r \mapsto 2r + \mathrm{rem}(l, 2)$. Moving it one site to the right amounts to the same expressions with r and l interchanged.

To summarize, the Wang encoding describes the tape & head configuration of a TM in a way so that every change of that configuration due to writing or moving can be expressed by primitive recursive functions. Similarly, the validity of a final configuration is expressed in terms of a primitive recursive predicate.

7.2 Encoding the Turing machine instructions

Recall that the set of instructions of a TM is represented by a map $M\colon (t, q) \mapsto (t', q', d)$ where $t, t' \in \{0, 1\}$ are tape symbols, $q, q' = 0, \ldots, n$ are internal states and $d = R, L$ is the direction in which the head moves one step. Let us encode the pair (t', d) which describes one out of four instantaneous actions

of the TM by a number $a = 1, \dots, 4$. The map M will be encoded in a string $m = (m_0, \dots, m_{4n-1}) \in \mathbb{N}^{4n}$ such that if (t, q) are scanned symbol and current internal state respectively, then $m_{4q+2t} \in \{1, \dots, 4\}$ is the action a, and $m_{4q+2t+1} \in \{0, \dots, n\}$ is the new internal state q'.

Finally, by using the tools developed in lecture 3 we encode the entire string m in a single number which by abuse of notation we call $M \in \mathbb{N}$, the code of the TM.

7.3 Evolution of the Turing machine configuration

Let us characterize an entire configuration of the TM (i.e., tape, head and internal state) by a single number $c = 2^l 3^q 5^r$. Note that the map $(l, q, r) \leftrightarrow c$ is primitive recursive in both directions. Let $\mathrm{conf}(M, x, \tau)$ denote such an entire configuration which is obtained after running a TM specified by a code $M \in \mathbb{N}$ on an input $x \in \mathbb{N}$ for $\tau \in \mathbb{N}$ steps. The crucial point is now that the function $\mathrm{conf}\colon \mathbb{N}^3 \to \mathbb{N}$ turns out to be primitive recursive. In order to see this we sketch a definition by primitive recursion:

First note that $\mathrm{conf}(M, x, 0) = 5^r = 5^{2^{x+1} \dot{-} 1}$. Furthermore, $\mathrm{conf}(M, x, \tau + 1)$ can be obtained in a primitive recursive way from M, x and $\mathrm{conf}(M, x, \tau)$ via:

1. extract the corresponding triple (r, q, l) from $\mathrm{conf}(M, x, \tau)$,
2. determine the action m_k with $k = 4q + 2\,\mathrm{rem}(r, 2)$ from the code M,
3. determine the new triple (r', q', l'),
4. set $\mathrm{conf}(M, x, \tau + 1) := 2^{l'} 3^{q'} 5^{r'}$.

Since all building blocks are primitive recursive, we get $\mathrm{conf} \in \mathrm{PR}$.

7.4 Halting & output

The TM halts once the internal state is n (which was our convention for the halting state). Given an entire configuration c this is expressed by the primitive recursive predicate $\mathrm{lo}(c, 3) = n$. The conjunction with the above primitive recursive predicate for (l, r) describing a valid final configuration gives a primitive recursive predicate $P(c)$ which is true iff the TM has reached a halting state in a valid final configuration. So, for given M and x, $\tau_{\mathrm{halt}} := \mu_y\big(P(\mathrm{conf}(M, x, y))\big)$ gives the time step when and if this happens and it is undefined otherwise. The output of the function computed by the TM with code M and input x is thus $F_M(x) = \mathrm{lg}(r, 2)$, where $r = \mathrm{lo}\left(\mathrm{conf}(M, x, \tau_{\mathrm{halt}}), 5\right)$.

This shows us not only that F_M is indeed a recursive function, but it also reveals two extremely useful facts which we will discuss in greater detail in the following lecture. Loosely speaking, we have seen that

1. a single application of the minimization operation is sufficient,
2. by regarding the code M as a variable we obtain a function $F(M,x) := F_M(x)$ which is universal in the sense that for every Turing computable function f there is an $M \in \mathbb{N}$ such that $F(M,x) = f(x)$ for all x.

8 Basic theorems of recursion theory

The proof of the equivalence between Turing computability and recursiveness, which we sketched in the last lecture, reveals a lot of additional structure which we will now discuss in more detail. Lecture 6

Theorem (Kleene's normal form). *For any $k \in \mathbb{N}$ there is a primitive recursive predicate T on $\mathbb{N}^{k+2}$ and a primitive recursive function $U\colon \mathbb{N} \to \mathbb{N}$ such that for any recursive partial function $f\colon \mathbb{N}^k \to \mathbb{N}$ there is an $M \in \mathbb{N}$ for which*

$$f(x) = U\Big[\mu_y\big(T(y,M,x)\big)\Big]$$

for all $x \in \mathbb{N}^k$.

Note that the only dependence on f is in M which we call an index of the function f. In the foregoing equivalence proof M was the code of the Turing machine. Since there are many Turing machines computing the same function, the index of a partial recursive function is not unique. Even more, the indexing itself is ambiguous – we could have chosen a different encoding of TMs, leading to a different indexing. In the following we will work with one fixed indexing/encoding in order to obtain compatible statements. This may be the one chosen in the equivalence proof or any other one as long as they can be interconverted by primitive recursive means. The particular choice of the indexing wont be important.

If we define the index of a function as an $M \in \mathbb{N}$ such that the above normal form holds, we get that a function has an index iff it is in R. In that case we will write $f = F_M$ to indicate that we're considering the function specified by the index M. By considering the index itself as an input/argument we obtain the following:

Corollary (Universal Turing machines & functions). *For any $k \in \mathbb{N}$ there exists a partial recursive function $F\colon \mathbb{N}^{k+1} \to \mathbb{N}$ such that for any $f \in R^{(k)}$ there exists an $M \in \mathbb{N}$ for which $f(x) = F(M,x)$ for all $x \in \mathbb{N}^k$.*

Proof. This is an immediate consequence of the normal form theorem which gives $F(M,x) = U\Big[\mu_y\big(T(y,M,x)\big)\Big]$. □

We will call such a function F a universal function and a respective Turing machine a universal Turing machine. What we have said about the

ambiguity of indexings/encodings holds as well for universal functions and machines: there are many; we choose a fixed one; with a little bit of care about convertibility, the particular choice doesn't matter.

Note that F formally depends on k, i.e., the number of arguments, although we will suppress this dependence in the notation (hopefully for the benefit of readability). In fact, if we express the same result in terms of Turing machines, then universal Turing machines do no longer depend on k, i.e., it makes sense to talk about a universal TM without specifying the number of its inputs.

Universal Turing machines may be considered the origin of the idea of a "universally programmable computer" reflecting the fact that we don't have to buy different devices for running different programs.

For the case of a tape with binary alphabet (symbols 0 and 1), there are explicit constructions of universal TMs with $n = 15$ internal states. On the other hand, for $n < 4$ it is known that no such universal machine can exists (since $n < 4$ gives rise to a solvable halting problem). As for now, the intermediate regime $4 \leq n < 15$ is uncharted territory – so feel free...

Another consequence of the normal-form theorem is that the range of a function does not care about whether the function is primitive recursive or merely partial recursive:

Theorem (Ranges of partial recursive functions)**.** *Let* $f\colon \mathbb{N}^k \to \mathbb{N}$ *be any partial recursive function with non-empty range. Then, there exists a primitive recursive function* $g\colon \mathbb{N} \to \mathbb{N}$ *such that* $\mathrm{range}(f) = \mathrm{range}(g)$*.*

Proof. Exploiting the normal-form theorem we can define a function $G\colon \mathbb{N}^{k+1} \to \mathbb{N}$,

$$G(x,y) := \begin{cases} U(y) & \text{if } T(y,M,x) \wedge \forall j < y\colon \neg T(j,M,x) \\ s_0 \in \mathrm{range}(f) & \text{otherwise.} \end{cases}$$

By construction G is primitive recursive and $\mathrm{range}(G) = \mathrm{range}(f)$. Now we exploit the existence of primitive recursive functions $E_i\colon \mathbb{N} \to \mathbb{N}$, $i = 1, \ldots, k+1$ which are such that the mapping $x \mapsto (E_1(x), \ldots, E_{k+1}(x))$ is a surjection from $\mathbb{N}$ onto $\mathbb{N}^{k+1}$ (see the lecture on Gödel numbers). Putting these things together we can construct a function $g(x) := G\big(E_1(x), \ldots, E_{k+1}(x)\big)$ which is primitive recursive and satisfies $\mathrm{range}(g) = \mathrm{range}(G) = \mathrm{range}(f)$. □

The following is useful whenever we want to consider some variables of a function as fixed parameters:

Theorem (Parameter/S-m-n-theorem)**.** *For all* $m, n \in \mathbb{N}$ *there is a primitive recursive function* $S\colon \mathbb{N}^{m+1} \to \mathbb{N}$ *such that* $\forall M \in \mathbb{N}, \forall x = (x_1, x_2) \in \mathbb{N}^m \times \mathbb{N}^n$

$$F\big(S(M, x_1), x_2\big) = F(M, x).$$

One way of proving the parameter theorem is to exploit again the equivalence between recursiveness and Turing computability: construct a TM which (i) writes x_1 in front of the current input (separated by a blank and stopping on top of the leftmost 1) and then (ii) runs the TM with code M. The code of this TM will then be $S(M, x_1)$ and the theorem follows from the fact that the code of a concatenation of two TMs is a primitive recursive function of their respective codes.

8.1 The recursion theorem and some applications

Theorem (Recursion theorem)**.** *Let $f\colon \mathbb{N} \to \mathbb{N}$ be any total recursive function. Then, for any $k \in \mathbb{N}$ there is an $n \in \mathbb{N}$ such that $F(n, x) = F\big(f(n), x\big)$ for all $x \in \mathbb{N}^k$.*

Proof. Define a function $g\colon \mathbb{N}^k \times \mathbb{N} \to \mathbb{N}$ by $g(x, y) := F\big(f(F(y, y)), x\big)$. Since g is recursive we can assign an index to it and then apply the parameter theorem in order to obtain $g(x, y) = F(s(y), x)$ for some primitive recursive and thus total function s. For the latter there exists in turn an index $m \in \mathbb{N}$ so that $s(y) = F(m, y)$. Putting things together we obtain

$$F\Big(f\big(F(y, y)\big), x\Big) = F\big(F(m, y), x\big)$$

for all x, y.

The sought result is then obtained by setting $y = m$ and $n := F(m, m)$, where the latter is possible since s is a total function. □

A closer look at the recursion theorem reveals that it does not only guarantee one such fixed point n, but infinitely many:

Corollary (Recursion theorem – refined)**.** *Let $f\colon \mathbb{N} \to \mathbb{N}$ be any total recursive function. Then, for any $k \in \mathbb{N}$ there are infinitely many $n \in \mathbb{N}$ such that*

$$F(n, x) = F\big(f(n), x\big)$$

for all $x \in \mathbb{N}^k$.

Proof. We have to show that there is no largest such n, i.e., that for any $l \in \mathbb{N}$ there is an $n > l$ fulfilling the above equation. To this end, pick a $c \in \mathbb{N}$ so that $F_c \notin \{F_0, \ldots, F_l\}$ and define

$$g(x) := \begin{cases} c & \text{if } x \leq l \\ f(x) & \text{if } x > l \end{cases}$$

Then g is a total recursive function to which we can apply the recursion theorem. For any $n \leq l$, however, $F_{g(n)} = F_c \neq F_n$ so that there must be a $n > l$ for which $F_n = F_{g(n)}$. Observing that in this case $F_{g(n)} = F_{f(n)}$ then completes the proof. □

If we regard the indexing of partial recursive functions as a (non-injective) enumeration, the recursion theorem says something about repetitions of functions. For instance, that there is an n so that $F_n = F_{n+1}$, etc. It might seem desirable to have an enumeration without repetition. Naively, this should be obtainable by just going through the list and omitting repeating instances. As the following corollary shows, this is, however, not possible by recursive means.

Corollary (Enumeration without repetition is not recursive)**.** *Suppose $f\colon \mathbb{N} \to \mathbb{N}$ is an increasing total function so that $m \neq n$ implies $F_{f(m)} \neq F_{f(n)}$ and so that $f(n)$ is the smallest index of the function $F_{f(n)}$. Then f is not recursive.*

Proof. It follows from the constraints imposed on f together with the fact that there are repetitions in the previously used indexing that there is a $k \in \mathbb{N}$ such that $\forall n \geq k\colon f(n) > n$. For those n's we thus have $F_{f(n)} \neq F_n$ since otherwise n would be a smaller index. This is, however, in conflict with the refined version of the recursion theorem unless f is not recursive. □

Our next aim is to prove the existence of functions which provide their own index as output. For that the following intermediate step is useful:

Corollary. *For all $k \in \mathbb{N}$ and $f \in R^{(k+1)}$ there are infinitely many $m \in \mathbb{N}$ such that $\forall y \in \mathbb{N}^k\colon f(m, y) = F(m, y)$.*

Proof. Since f is recursive we can express it in terms of a universal function and then apply the parameter theorem so that $f(x, y) = F(n, x, y) = F(s(x), y)$ for some primitive recursive function $s\colon \mathbb{N} \to \mathbb{N}$. Applying the refined recursion theorem to the latter leads to infinitely many m's for which $F(s(m), y) = F(m, y)$ and thus $f(m, y) = F(m, y)$ for all y. □

Note that a simple consequence (which we know already) is that every partial recursive function g has infinitely many indices: just apply the above corollary to any function with a trivial m-dependence, i.e., $f(m, y) := g(y)$.

Corollary ("Self-reflexive" functions)**.** *There is a partial recursive function $f\colon \mathbb{N} \to \mathbb{N}$ so that $\forall x \in \mathbb{N}\colon f(x) = m$ where m is an index of f, i.e., $f = F_m$.*

Proof. This follows from the previous corollary when applied to $f(m, y) := m$. □

Phrased in terms of algorithms, this corresponds to a program which prints its own source code – something one occasionally encounters in the unpleasant costume of a computer virus. Loosely speaking, the source code of such a print-your-own-code-program can be of the form

```
Print the following twice, the second time in quotes:
"Print the following twice, the second time in quotes:"
```

Finally, we will see that the Ackermann-Peter function is indeed recursive:

Corollary (Ackermann-Peter is recursive)**.** *The Ackermann-Peter function* $A\colon \mathbb{N}^2 \to \mathbb{N}$ *is recursive.*

Proof. Consider an auxiliary function $g\colon \mathbb{N}^3 \to \mathbb{N}$ which we define by $g(m,0,y) := y+1$, $g(m,x+1,0) := F(m,x,1)$ and $g(m,x+1,y+1) := F\big(m,x,F(m,x+1,y)\big)$. By construction g is recursive, so we can make use of the parameter theorem and obtain a primitive recursive function s for which $g(m,x,y) = F_{s(m)}(x,y)$. Applying the recursion theorem to s, we know that there is an n so that $F_{s(n)} = F_n$. Using this one as first argument of g and defining $A(x,y) := g(n,x,y)$ we obtain a function A which satisfies exactly the three defining equations of the Ackermann-Peter function. □

9 Rice's theorem and the Church-Turing thesis

Lecture 7

Recall that a recursive set is defined as one whose characteristic function is a total recursive function. A decision problem or predicate with domain $\mathbb{N}$ is called *recursively undecidable* iff the corresponding characteristic function is not a total recursive function. The following theorem states that basically any property of functions leads to such an undecidable problem:

Theorem (Rice)**.** *Let* $\mathcal{A}$ *be any class of functions such that* $\emptyset \subset \mathcal{A} \subset R^{(1)}$ *where both inclusions are supposed to be strict. Then* $A := \{x \in \mathbb{N} \mid F_x \in \mathcal{A}\}$ *is not a recursive set, i.e.,* $x \in A$ *is a recursively undecidable predicate.*

Proof. Let χ_A be the characteristic function of the set A. Take $a \in A$, $b \notin A$ and define $f(x) := \chi_A(x)b + \chi_{\neg A}(x)a$. If A is recursive, then f is a total recursive function. Hence, we can apply the recursion theorem and obtain an n such that $F_{f(n)} = F_n$. This means that $n \in A$ iff $f(n) \in A$. This is, however, in conflict with the construction of f which implies $x \in A$ iff $f(n) \notin A$. So, A cannot be recursive. □

Now we can choose to which property to apply Rice's theorem...

Corollary (Some undecidable problems)**.** *The following predicates are recursively undecidable for any* y*:*

1. Halting type problem*:* $z \in \{x \mid F_x(y) \text{ is defined}\}$,
2. Equivalence of functions*:* $z \in \{x \mid F_x = F_y\}$,
3. Constrained range*:* $z \in \{x \mid y \in F_x(\mathbb{N})\}$.

Let us rephrase the first two in terms of every-day computer programming language. The halting problem tells us that there is no general method for finding out whether a program eventually returns or continues running forever.

That is, any debugger necessarily has its limitations. The non-recursiveness of the second set implies that there is no general way of determining whether two algorithms actually do the same.

Some additional remarks on the halting problem are in order since this is somehow the paradigm for a recursively undecidable problem. If we stick to the conventions we chose in the discussion of Turing computability, then the above problem (nr.1) is not evidently a formulation of the halting problem. Recall that in our framework of Turing computability there were two possibilities for $F_x(y)$ to be not defined: (i) the TM doesn't halt, and (ii) the TM halts but in an ill-defined configuration. Due to the latter, the halting problem isn't exactly matched by the non-recursiveness of $\{x \mid F_x(y) \text{ is defined}\}$. However, the undecidability of the above problem clearly implies undecidability of the halting problem. Moreover, we are free to use a different (but computationally equivalent) convention for which there are no ill-defined final configurations.

The set $\{x \mid F_y(x) \text{ is defined}\}$ may or may not be recursive depending on y. One way to see that there are y's for which it is not recursive, is to make use of universal functions again. For instance, regard $F_x(z)$ as a function with arguments (x, z), so that $F_x(z) = F(v, x, z)$ for some index v. Employing the parameter theorem we obtain a $y(v, z)$ so that $F(v, x, z) = F(y, x)$ implying that $\{x \mid F_x(z) \text{ is defined}\} = \{x \mid F_y(x) \text{ is defined}\}$ is non-recursive.

9.1 Church-Turing thesis

So far we have seen that two ways of formalizing the notion "computable by algorithm"(synonymously "effectively computable", "computable in the physical world" or just "computable"), namely Turing computability and recursiveness are equivalent. We could have considered other ways of formalizing this: cellular automata (like Conway's Game of Life or a Rule110 automaton), register machines (which are closer to the computers we actually use), λ-calculus, grammars for formal languages, the Shakespeare programming language, etc.. For all of them we would in principle be able to formally prove that they are no more (and also generally no less) powerful than Turing machines. The same holds true for quantum computers which act on a finite Hilbert space. The Church-Turing thesis states that this is always true, irrespective of the model we choose, as long as it is a somewhat *realistic* one:

Thesis (Church-Turing thesis). *Anything which is effectively computable, is computable by a Turing machine.*

The thesis seems to be commonly accepted by computer scientist and mathematicians. Nevertheless, machines, so-called hypercomputers, whose capabilities go beyond Turing machines are studied (theoretically :-)). Note that from a purely operational perspective, if we consider computation as a

black box process, we may not be able to distinguish hypercomputers from not-so-hyper computers if we are restricted to finite data. It is only on the level of the theoretical description where we might realize the difference.

Note that one motivation for studying hypercomputation could be to specify the degree (actually called Turing-degree) of unsolvability of a problem...

We will in the following invoke the Church-Turing thesis whenever we drop “recursively” in “recursively undecidable” or when we omit “Turing” in “Turing computable”.

10 Recursive enumerability

Definition (Recursively enumerable sets)**.** *A set $S \subseteq \mathbb{N}^k$ is called recursively enumerable iff there is a partial recursive function f such that* $\mathrm{dom}(f) = S$, *i.e., $f(x)$ is defined iff $x \in S$.*

Equivalently we could define recursively enumerable sets as those for which there exists a Turing machine which halts upon input x iff $x \in S$.

Note that if S is recursively enumerable, then $x \in S$ can be verified by recursive means but in general it cannot be falsified. A paradigmatic example of a recursively enumerable set which is not recursive is $\{M \in \mathbb{N} \mid F(M,x) \text{ is defined}\}$ for any x.

Proposition (Characterization via primitive recursive predicates)**.** *A set $S \subseteq \mathbb{N}^k$ is recursively enumerable iff there exists a primitive recursive predicate P on $\mathbb{N} \times \mathbb{N}^k$ such that $S = \{x \in \mathbb{N}^k \mid \exists y\colon P(y,x)\}$.*

Proof. The “only if” direction follows from Kleene’s normal form theorem: let f be the the partial recursive function whose domain is S. Then $f(x) = U\Big[\mu_y\big(T(y,M,x)\big)\Big]$ where M is any index of f. Since U is total we have that $x \in \mathrm{dom}(f)$ iff $\exists y\colon T(y,M,x)$. Setting $P(y,x) := T(y,M,x)$ and using that T is a primitive recursive predicate then proves the claim.

For the “if” direction define $f(x) := \mu_y\big(P(y,x)\big)$. Then $\exists y\colon P(y,x)$ iff $x \in \mathrm{dom}(f)$. □

Now we can show that the property of being recursively enumerable is closed w.r.t. intersections and finite unions:

Proposition. *If $A, B \subseteq \mathbb{N}^k$ are recursively enumerable sets, then so are $A \cup B$ and $A \cap B$.*

Proof. For the union we use the foregoing proposition together with the fact that disjunction preserves primitive recursiveness: let P_A, P_B be the primitive recursive predicates characterizing A and B respectively, then we can write $A \cup B = \{x \mid \exists y\colon P_A(y,x) \lor P_B(y,x)\}$.

For the intersection we exploit the defining property of recursively enumerable sets and note that the domain of the function $f_{AB} = f_A + f_B$ is the intersection of the domains of f_A and f_B. □

While unions and intersections preserve recursive enumerability, the complement does not necessarily do so:

Proposition. *A set $S \subseteq \mathbb{N}^k$ and its complement $\bar{S}$ are both recursively enumerable iff they are both recursive.*

Proof. Clearly, if S is recursive, then so is its complement and both are recursively enumerable: to see the latter, just define a recursive function $f(x) := \mu_y(x \in S)$. Then $f(x)$ is undefined iff $x \notin S$. Hence, $\mathrm{dom}(f) = S$.

Conversely, if we assume that $S, \bar{S}$ are both recursively enumerable with characterizing primitive recursive predicates $P_S, P_{\bar{S}}$ respectively, then we can define a recursive function $f(x) := \mu_y\big(P_S(y,x) \vee P_{\bar{S}}(y,x)\big)$ which is total because $\forall x\colon x \in S \cup \bar{S}$ and thus $\forall x \exists y\colon \big(P_S(y,x) \vee P_{\bar{S}}(y,x)\big)$. From here we can construct the characteristic function $\chi_S(x)$ by setting it to 1 iff $P_S(f(x), x)$. Recursiveness of χ_S then implies (by definition of recursive sets) that S is recursive. □

Lecture 8

The following proposition makes clear why "recursively enumerable" sets are called like that:

Proposition (recursive enumeration)**.** *Let $S \subseteq \mathbb{N}$ be a non-empty set. Then the following are equivalent:*

1. *S is recursively enumerable.*
2. *There is a primitive recursive function $g\colon \mathbb{N} \to \mathbb{N}$ such that* $\mathrm{range}(g) = S$.
3. *There is a $k \in \mathbb{N}$ and a partial recursive function $G\colon \mathbb{N}^k \to \mathbb{N}$ such that* $\mathrm{range}(G) = S$.

Proof. The equivalence of 2. and 3. was proven in a previous lecture using the normal-form theorem. In order to prove 1. → 3. we use that there is a primitive recursive predicate such that $S = \{x \mid \exists y\colon P(y,x)\}$. Based on this we define a function $G(x,y)$ so that it equals x if $P(y,x)$ is true and it is equal to some $s_0 \in S$ otherwise. By construction, G is recursive and $\mathrm{range}(G) = S$.

Conversely, if we have a function g with $\mathrm{range}(g) =: S$, then $f(x) := \mu_y\big(g(y) = x\big)$ defines a partial recursive function for which $\mathrm{dom}(f) = \mathrm{range}(g)$ so that S is indeed recursively enumerable. □

So, any recursively enumerable set $S \subseteq \mathbb{N}$ is of the form $S = f(\mathbb{N})$ for some recursive function f. The latter can thus be thought of "enumerating" the elements of S, albeit not necessarily in an increasing way. In fact, if the enumeration is an increasing sequence, then S is recursive:

Proposition (ordered recursive enumeration)**.** *Let $S \subseteq \mathbb{N}$ be an infinite set. Then the following are equivalent:*

1. *S is recursive.*

2. *There is a partial recursive function $g\colon \mathbb{N} \to \mathbb{N}$ which is (i) total, (ii) strictly increasing and (iii) such that* $\text{range}(g) = S$.

Proof. For 1. $\to$ 2. define a function $g(0) := \mu_y(y \in S)$, $g(n+1) := \mu_y(y \in S \wedge y > g(n))$. Then g is strictly increasing, total recursive and so that $\text{range}(g) = S$.

For the converse direction note that $y = g(n)$ implies that $n \leq y$. Therefore, the characteristic function χ_S of S equals the characteristic function of the recursive predicate $\exists n \leq y\colon g(n) = y$. □

Finally, let us use the characterization via ranges together with the old diagonalisation trick in order to show that the set of total recursive functions itself is not recursively enumerable:

Proposition (total recursive functions are not enumerable)**.** *The set $S := \{M \in \mathbb{N} \mid F_M \text{ is total}\}$ is not recursively enumerable.*

Proof. Suppose $S = \text{range}(g)$ for some primitive recursive $g\colon \mathbb{N} \to \mathbb{N}$. Define $f(x) := F(g(x), x) + 1$. By construction f is a total recursive function. However, there cannot be an $n \in \mathbb{N}$ so that $F_{g(n)} = f$ since then $F_{g(n)}(n) = F_{g(n)}(n)+1$. Consequently, g cannot be recursive, i.e., S cannot be recursively enumerable. □

11 The word problem for Thue systems

The halting problem can serve as the origin for a chain of provably undecidable problems. A standard and in fact the by far most common way for proving that a problem is undecidable, is to show that the halting problem can be reduced to it in the sense that any algorithmic solution of the considered problem would imply a solution for the halting problem. In many cases there is in fact a longer chain of reductions through intermediate problems.

In this lecture we will provide a first (and second) link in such chains by proving a reduction of the halting problem to the word problem for semi-Thue systems (and finally for Thue systems). Here and in the following lectures (until we say something different) undecidable will always mean algorithmically undecidable by recursive means. Having in mind the Church-Turing thesis we will drop "by recursive means" and since we haven't talked about axiomatic undecidability yet, we will be even more sloppy and also omit "algorithmically"...

11.1 Terminology and notation

In principle we could talk about sets, their elements, ordered pairs and n-tuples, but in this business a different terminology is commonly used: we consider a finite set $A = \{a_1, \ldots, a_N\}$ which we call *alphabet*, the elements of which are called symbols. A finite sequence of symbols $a_{i_1} a_{i_2} \ldots$ is called a *word* over the alphabet, and the set of all words of length n will be denoted by $A^{(n)}$. It is convenient to use an extra symbol, say e, for the word of zero length and to include this into the set of all words $A^* := \bigcup_{n \in \mathbb{N}} A^{(n)}$.

Definition (semi-Thue system)**.** *A* semi-Thue system *is a pair* (A, Π) *of a finite alphabet* A *and a finite set of ordered pairs* $\Pi \subset A^* \times A^*$ *called* substitution rules*.*

Furthermore, we will write $u \Rightarrow v$ for a pair of words $u, v \in A^*$ iff there is a substitution rule $(P_1, P_2) \in \Pi$ and words $x, y \in A^*$ (possibly of zero length) so that $u = xP_1y$ and $v = xP_2y$ (hence, the name substitution rule). Note that here we make use of the natural operation on words, namely concatenation which is, needless to say, associative.

In a similar vein, we will write $u \Rightarrow^* v$ iff there is a finite sequence of words $u_1, \ldots, u_m$ so that $u = u_1 \Rightarrow \ldots \Rightarrow u_m = v$ which for $m = 1$ means $u = v$. Keep in mind that all this is always w.r.t. a given semi-Thue system. The word problem for a fixed semi-Thue system is then the problem of deciding whether or not $u \Rightarrow^* v$ for given words $u, v \in A^*$. The following specifies in which sense the word problem is undecidable:

Proposition (Undecidable word problem for semi-Thue systems)**.** *There is a semi-Thue system* (A, Π) *and a word* $v \in A^*$ *such that there is no algorithm which upon input* $u \in A^*$ *is capable of deciding whether or not* $u \Rightarrow^* v$*. (That is, every algorithm attempting to compute this predicate will necessarily fail on an infinite subset of inputs.)*

Proof. As mentioned before, the idea is to reduce the halting problem to this word problem. So we will construct a semi-Thue process for every Turing machine for which the corresponding map $M \colon T \times Q \to T \times Q \times \{R, L\}$ is defined for any internal state apart from the halting state. Let us denote the internal states as $q_0, \ldots, q_n$ where q_0 and q_n are the starting state and halting state, respectively. In order to encode a Turing machine into a semi-Thue system we characterize the entire tape&head configuration by a string of the form $\mid xqy \mid$ where $q \in Q$ is the current internal state, $xy \in T^*$ is the content of the tape (in a sufficiently large region to be specified in a moment) and the scanned symbol t is the left most symbol of y. If we denote the rightmost symbol of x as s, then a single step in the TM evolution only alters the triple sqt in the way

$$sqt \to \begin{cases} q'st' & \text{if } L \\ st'q' & \text{if } R \end{cases}.$$

Here the case to be chosen (R or L) and the primed = new symbols for the internal state and the tape cell depend on t and q from which they are determined by the map M which characterizes the TM. Hence, the evolution of a TM corresponds to a finite set of substitution rules so that we almost have a semi-Thue process assigned to any TM. What has yet to be taken care of is the fact that the tape is infinite while a word has to be finite. In order to cure this apparent conflict we introduce the marker $|$ which does nothing but setting a virtual mark at the outmost points of the "relevant" part of the tape. Since we do not know beforehand how big this is to be chosen, we allow the marks to move. This is done by adding the substitution rules

$$q \,|\!\rightarrow q0\,| \qquad \text{and} \qquad |\, q \rightarrow |\, 0q$$

for all $q \in Q \setminus \{q_n\}$.

Finally, we want to assure that if the halting state appears, then there is a unique squence of substitutions which collaps the entire head&tape configuration to a unique word. To this end we add a new symbol ω and the substitution rules

$$q_n t \rightarrow q_n, \quad q_n \,|\!\rightarrow \omega\,| \quad \text{and} \quad t\omega \rightarrow \omega$$

for all $t \in T$. Taking it all together we have achieved that the TM halts upon starting with the head on the leftmost symbol of a string $u \in T^*$ iff

$$|\, q_0 u \,| \Rightarrow^* |\, \omega \,|$$

holds for the constructed semi-Thue system with alphabet $A = Q \cup T \cup \{\omega, |\}$. Applying this to a universal TM we see that a solution to the above word problem would imply a solution to the halting problem. Hence, there cannot be an algorithm deciding the word problem. □

The construction in the proof is such that an n-state TM with binary tape alphabet gives rise to a semi-Thue system with $6n + 5$ substitution rules and an alphabet with $n + 4$ symbols. While the latter number can easily always be reduced to 2, the size of Π is more tricky – the current record is $|\Pi| = 3$.

Lecture 9

Definition (Thue system)**.** *A* Thue system *is a semi-Thue system for which* $(P_1, P_2) \in \Pi$ *implies that* $(P_2, P_1) \in \Pi$*.*

It follows directly from the definition of a Thue system that the relation $u \Rightarrow^* v$ becomes an equivalence relation since it is reflexive, symmetric and transitive. We will thus write $u \sim v$ in this case.

Proposition (Undecidable word problem for Thue systems)**.** *There is a Thue system* (A, Π) *and a word* $v \in A^*$ *such that there is no algorithm which upon input* $u \in A^*$ *is capable of deciding whether or not* $u \sim v$*.*

Proof. We will exploit the semi-Thue system constructed from a universal Turing machine in the proof of the undecidability of the word problem for semi-Thue systems and reduce the latter to the one of Thue systems.

Denote by (A, Π_s) the semi-Thue system and construct a Thue system (A, Π) as a minimal extension thereof in the sense that $\Pi := \{(P_1, P_2) \mid (P_1, P_2) \in \Pi_s \vee (P_2, P_1) \in \Pi_s\}$. Now suppose that $u \sim v$. Then there is a sequence of say N words such that $u = u_1 \sim u_2 \sim \ldots \sim u_N = v$ where each step involves only a single substitution, i.e., w.r.t. Π_s we have either $u_k \Leftarrow u_{k+1}$ or $u_k \Rightarrow u_{k+1}$. Suppose the latter is not true for all steps in the chain. Then let i be the largest index of a word u_i in this chain for which $u_i \Leftarrow u_{i+1}$ w.r.t. Π_s. Since we chose the final word $u_N =$ "$|\omega|$" such that there is no Π_s-substitution starting with u_N, the last step in the chain has to be $u_{N-1} \Rightarrow u_N$ w.r.t. Π_s, which implies that we have $i \leq N - 2$. By the construction of Π and the definition of i we get that both $u_{i+1} \Rightarrow u_i$ and $u_{i+1} \Rightarrow u_{i+2}$ w.r.t. Π_s. The construction of the semi-Thue system into which we encoded the TM was, however, such that no more than one substitution rule applies to any word which contains exactly one symbol from the set $Q \cup \{\omega\}$. Therefore $u_i = u_{i+2}$ and we can eliminate these two steps from the chain. By induction we are then left with a sequence of words where $u_k \Rightarrow u_{k+1}$ holds for all of them w.r.t. Π_s. So $u \sim v$ holds iff $u \Rightarrow^* v$ holds for the semi-Thue system. Consequently, the word problem for Thue systems cannot be decidable either. □

12 Undecidable problems for semigroups

Quoting Calvin (from a Calvin and Hobbes comic) "The living dead don't need to solve word problems". As we saw in the previous lecture, this makes life (or whatever) easier. In the following we will discuss some corollaries and consequences of the undecidability of the word problem for Thue systems. The basic line of thought will be to realize that the set of words forms a semigroup which then allows us to extend the undecidability result to presentations of semigroups and groups. The latter in turn enables us to obtain similar results in the realm of algebraic topology when for instance homotopy groups are considered.

Consider a Thue system (A, Π). As mentioned previously, the symmetric appearance of the substitution rules makes $u \Rightarrow^* v$ an equivalence relation for which we will write $u \sim v$. The quotient $M := A^*/\sim$ is then a *monoid* (i.e., a semigroup with identity) if we use concatenation of words as associative binary operation. The Thue system (A, Π) then becomes a *presentation* of M where A is the set of generators. Hence, $u \sim v$ means that u and v are actually the same element of M albeit possibly expressed differently in terms of the generators. Since both A and Π are supposed to be finite sets, the resulting semigroup is said to be *finitely presented.* Given a presentation

of a monoid, the corresponding word problem is about deciding whether or not u and v correspond to the same element of the monoid. It was proven undecidable independently by Post and Markov in the late 40ies. In our context it is a corollary or rather a reformulation of the result for Thue systems:

Corollary (Undecidable word problem for monoid presentations)**.** *There is a presentation (A, Π) of a monoid M and a word v over the set of generators A such that there is no algorithm which upon input $u \in A^*$ is capable of deciding whether or not $u \sim v$.*

The minimal number $|\Pi|$ of relations for which the word problem for monoid presentations is known to be undecidable is three. Remarkably, it is not known if the word problem becomes decidable if we restrict ourselves to cases with only a single relation.

There are plenty properties of finitely presented semigroups which turn out to be undecidable. A standard and generally useful approach to prove such an undecidability is to show that the property under consideration is a so called Markov property.

Definition (Markov property)**.** *Let $\mathcal{P}$ be a property of semigroups which is preserved under isomorphisms. $\mathcal{P}$ is called a* Markov property *iff it satisfies the following conditions:*

1. *There is a finitely presented semigroup S_1 with property $\mathcal{P}$,*
2. *There exists a finitely presented semigroup S_2 which does not embed into any finitely presented semigroup with property $\mathcal{P}$.*

Examples of Markov properties include being finite, being a group and many others. All those turn out to be generally undecidable for finitely presented semigroups:

Proposition (Markov properties are undecidable)**.** *Let $\mathcal{P}$ be a Markov property of semigroups. There is no algorithm which upon input of a finite semigroup presentation decides whether or not the given semigroup has property $\mathcal{P}$.*

Proof. (sketch) Let M be a finitely presented monoid with an undecidable word problem and let S_1 and S_2 be two monoids as in the definition of "Markov property". Let $S := M * S_2$ be the "free monoid product" which defines the monoid $S =: (A, \Pi)$ in terms of a Thue system whose alphabet A is the union of the disjoint alphabets of M and S_2 and whose substitutions rules Π are the union of those corresponding to M and S_2. Then S has an undecidable word problem, it is finitely presented (since M and S_2 are) and it does not have property $\mathcal{P}$ (since this would contradict the assumption on S_2).

For any pair $u, v \in A^*$ define a monoid $S_{u,v}$ via the Thue system

$$\big(B, \Pi \cup \{(bxvy, xvy)\}_{\forall b \in B} \cup \{(xuy, e)\}\big)$$

with alphabet $B := A \cup \{x, y\}$ and the additional symbols x, y are understood not to be already in A.

If $u \sim v$, then $S_{u,v}$ is trivial since in that case $b \sim be \sim bxuy \sim bxvy \sim xvy \sim xuy \sim e$ for all $b \in B$. If $\neg(u \sim v)$, then the additional relations can be seen to be useless so that S_2 embeds into $S_{u,v}$ and consequently $S_{u,v}$ cannot have property $\mathcal{P}$. Hence, the monoid $S_1 * S_{u,v}$ has property $\mathcal{P}$ iff $u \sim v$, which is the sought reduction. $\square$

13 Undecidable problems related to groups and topology

For groups, or more precisely presentations of groups, one can now follow similar albeit technically more demanding lines and prove that the word problem is again undecidable for finitely presented groups. This is known as the Novikov-Boone theorem. In our context a Thue-system gives rise to a group presentation if A is a set of disjoint pairs a_i, a_i' such that $(a_i a_i', e) \in \Pi$ for all i. In the formulation of the word problem one can fix one of the elements to be the identity, so that the (generally undecidable) question becomes whether or not any given element equals the identity.

Similar to the semigroup case, many properties which one typically encounters when dealing with groups turn out to be undecidable. A meta-theorem from which many of these follow is the Adjan-Rabin theorem. Some of its corollaries are that properties like being finite, being trivial (i.e., containing only one element), being cyclic, being commutative, and being solvable are all generally undecidable from a group presentation.

Group presentations play an important role in algebraic topology where groups which specify topological properties and invariants are typically given in terms of generators and relations.

A simple topological consequence of the undecidability of triviality of a group follows from a result by Haken who constructed a 2-dimensional simplicial complex from any group presentation such that the given group is the fundamental group of the simplicial complex. Since the fundamental group is trivial iff the simplicial complex is simply connected, we obtain that being simply connected is a generally undecidable property for 2-dimensional simplicial complexes.

Another topological problem which was proven to be undecidable (in this case by Markov) is the question whether or not two manifolds of dimension ≥ 4 are homeomorphic. For dimension two, decidability goes back to Riemann and for dimension three it is open whether or not the question is decidable.

14 Post's correspondence problem

Lecture 10

In this lecture we will show that Post's correspondence problem (PCP) is undecidable. The undecidability of PCP is frequently used as a final step in undecidability proofs since many problems can more easily be related to PCP, than for instance to the halting problem. So what is Post's correspondence problem?

Let A be a finite alphabet for which we consider the set of words A^* as a monoid with respect to which $X, Y : K \to A^*$ are two homomorphisms from $K := \{1, \ldots, k\}$. Post's correspondence problem is then to decide whether or not there is a non-empty word $w \in K^*$ such that $X(w) = Y(w)$.

An equivalent, possibly more intuitive, depiction of the problem is in terms of "dominos": consider a set of k dominos of the form

$$\begin{bmatrix} X(1) \\ Y(1) \end{bmatrix}, \ldots, \begin{bmatrix} X(k) \\ Y(k) \end{bmatrix},$$

where each $X(i)$ and $Y(i)$ is a word over A. The question is now whether or not there is a finite sequence of those dominos (where each domino may occur multiple times) for which the first and second row are equal, i.e., for which $X(w_1)X(w_2)\ldots = Y(w_1)Y(w_2)\ldots$ for some $w \in K^*$ which is at least of length one.

Example 1. Take $A = \{0, 1\}$ and two dominos of the form $(X(1), Y(1)) = (0, 00)$, $(X(2), Y(2)) = (01, 1)$. Then PCP has a solution in the sense that $X(1)X(2) = Y(1)Y(2) = 001$.

Example 2. For $(X(1), Y(1)) = (0, 01)$, $(X(2), Y(2)) = (10, 01)$ there cannot be such a solution since any word in $X(K^*)$ ends with a zero while every word in $Y(K^*)$ ends with a one.

By reducing the word problem for semi-Thue systems to PCP we will now show that there is no general recipe for deciding PCP:

Proposition (PCP is undecidable)**.** *Let A be a binary (or larger) alphabet. There is a $k \in \mathbb{N}$ such that there exists no algorithm which upon input of two monoid homomorphisms $X, Y : K := \{1, \ldots, k\} \to A^*$ decides whether or not there exists a non-empty word $w \in K^*$ for which $X(w) = Y(w)$.*

Proof. We will show that a hypothetical algorithm capable of deciding PCP would also allow for deciding the word problem $x \Rightarrow^* y$ for semi-Thue systems. To this end, consider any semi-Thue system (A, Π) with alphabet A and substitution rules Π. In order to encode this into a PCP, we enlarge the alphabet first by introducing an alphabet A' which contains symbols which are in one-to-one correspondence to the ones of A. That is, for every $a \in A$ there is exactly one $a' \in A'$. In addition, we introduce the symbol I, so that the alphabet $A_{PCP} := A \cup A' \cup \{I\}$ contains $2|A| + 1$ symbols.

Now we use $k = 2+2|A|+|\Pi|$ "dominos" which we will write as $(X(i), Y(i))$ and refer to its entries, somewhat confusingly, as first and second "row" (imagining $(X(i), Y(i))$ to be rotated clockwise by 90 degrees). We choose dominos (Ix, I), (I, yI) and (a, a') and (a', a) for all $|A|$ pairs of symbols as well as (v', u) for all substitutions $(u, v) \in \Pi$. In the latter case v' means that each symbol in v is replaced by its primed twin from A'. We will assume that each u, v as well as x and y are all non-empty words in A^*. This can be done w.l.o.g. since the semi-Thue system with an undecidable word problem stemming from the halting problem satisfied this assumption.

Any PCP solution has to start with (Ix, I) since this is the only domino where the left most symbols coincide in both rows. The only way to catch up with x is then the use of dominos of the types (a', a) and (v', u). In any case, completing x in the second row, will generate a $x_1' \in A'^*$ in the first row whose unprimed twin satisfies $x \Rightarrow^* x_1$. Catching up with x_1' in turn can only involve the (a, a') dominos which produce an unprimed copy x_1 in the first row. At this point our argument continues like before and we see that forcing the two rows to lead to equal words implies (if it is at all possible) a sequence $x \Rightarrow^* x_1 \Rightarrow^* x_2 \Rightarrow^* \ldots$ which (if at all) necessarily ends with $\ldots \Rightarrow^* y$ since (I, yI) is the only possibility for a right most domino. Hence a solution $X(w) = Y(w)$ implies a solution for $x \Rightarrow^* y$. Since the converse holds as well, we have reduced the word problem for the semi-Thue systems to PCP. The alphabet A_{PCP} can always be reduced to a binary one – just by binary encoding. □

Using a semi-Thue system with binary alphabet and an undecidable word problem based on $|\Pi|$ substitution rules (for which we know that $|\Pi| = 3$ suffices), the above proof leads to an undecidable PCP with $k = |\Pi| + 6$ relations. There is a more efficient encoding which achieves the same with $k = |\Pi| + 4$, so $k = 7$ is currently the smallest number of dominos for which PCP is known to be undecidable. Conversely, PCP is known to be decidable for $k = 2$.

Finally, note that if we constrain the size of the alphabet, the number of dominos as well as the length of the words $X(i)$, $Y(i)$, then PCP becomes a finite set of decision problems and thus recursively decidable (even though we might have no idea how the actual algorithm looks like).

15 Undecidable matrix problems

Lecture 8

In this lecture we exploit PCP in order to prove that two problems involving products (i.e., words) of small matrices from a small alphabet are undecidable: the matrix mortality problem (I guess I would have chosen a different name) and the reachability problem.

15.1 Matrix mortality

Consider a finite set of $d \times d$ matrices $S = \{M_1, \dots, M_n\} \subset \mathcal{M}_d(\mathbb{Z})$ with integer entries. We call S mortal iff there is a non-empty word $w \in \{1, \dots, n\}^*$, of length m say, such that for the corresponding product of matrices:

$$M_{w_1} \cdots M_{w_m} = 0.$$

Example 1. Consider a set consisting of two matrices

$$\begin{pmatrix} 0 & 1 \\ -3 & 2 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 2 & 0 \\ -0 & -1 \end{pmatrix}$$

This cannot be mortal since the matrices have non-zero determinant and the determinant of any product is just the product of determinants.

Example 2. The two matrices

$$\begin{pmatrix} 0 & 0 \\ -0 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

form a mortal set since their product is a nilpotent matrix whose square vanishes.

Before we show that unlike in these simple examples there cannot be a general recipe for deciding mortality, we will introduce some tools for encoding words into products of matrices:

For words $w = a_1 \dots a_m$ over the alphabet $A := \{1, 2, 3\}$ define an injective map $W(w) := \sum_{k=1}^{m} a_k 4^{m-k}$ from A^* to $\mathbb{N}$. Denote by $|w|$ the length of a word and define a map from $A^* \times A^*$ into the set of 3×3 integer matrices by

$$M(u, w) := \begin{pmatrix} 4^{|u|} & 0 & 0 \\ 0 & 4^{|w|} & 0 \\ W(u) & W(w) & 1 \end{pmatrix}.$$

If we use concatenation of words and matrix multiplication as binary operations in the domain and codomain respectively, then $(u, w) \mapsto M(u, w)$ is an injective monoid homomorphism. That is, in particular $M(u_1, v_1)M(u_2, v_2) = M(u_1 u_2, v_1 v_2)$.

In addition we will need the matrix

$$B := \begin{pmatrix} 1 & 0 & 1 \\ -1 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}$$

which satisfies $B^2 = B$ and $BM(u, v)B = \big(4^{|u|} + W(u) - W(v)\big)B$.

The latter implies that $BM(u, v)B = 0$ iff $W(v) = W(1u)$ which in turn is equivalent to $v = 1u$. Now let us exploit these relations to prove the following:

Proposition (Matrix mortality is undecidable)**.** *Let $k \in \mathbb{N}$ be such that PCP with k "dominos" is undecidable. Then there is no algorithm which upon input of a set $S \subset \mathcal{M}_3(\mathbb{Z})$ of $2k+1$ integer matrices decides whether or not S is mortal.*

Proof. Consider an undecidable PCP with k dominos and choose $\{2,3\}$ as a binary alphabet for it. Denote by (x_i, y_i) with $i = 1, \ldots, k$ the pairs of words appearing in the PCP. For each of these k dominos we define two matrices $M_i := M(x_i, y_i)$ and $M_i' := M(x_i, 1y_i)$. So together with B these form a set S of $2k+1$ integer matrices.

Now assume that PCP has a solution $w \in \{1, \ldots, k\}^*$. Then

$$BM'_{w_1} M_{w_2} \cdots M_{w_{|w|}} B = 0$$

so the set S is mortal. Conversely, if S is mortal, then there is a product so that $BM(u_1, v_1)BM(u_2, v_2)B \cdots B = 0$. Since $B^2 = B$ and each $BM(u_i, v_i)B$ is a multiple of B, the product can only be zero if for at least one i we have $1u_i = v_i$. Observing that $u_i \in \{2,3\}^*$ this implies a solution for PCP. □

Using that PCP is known to be undecidable for seven dominos, we obtain that matrix mortality is undecidable for sets of fifteen 3×3 matrices. One can trade the number of matrices with their dimension and show that matrix mortality is undecidable as well for two 24×24 matrices. On the positive side, it is known that it is decidable for two 2×2 matrices and for instance for an arbitrary number of upper triangular 2×2 matrices. Without such an additional constraint decidability is, however, not known already for three 2×2 matrices with integer coefficients.

15.2 A reachability problem

We will now have a look at a problem which can be motivated by problems arising in engineering, in particular in the context of control theory. Assume that the state of a system/machine is characterized by a vector of parameters which can be changed by means of a finite number of controls all of which act on this "state vector" in a linear or affine way. A natural question is then whether or not there is a sequence of controls which map a given initial state onto a given final state.

Again, before we prove undecidability of such a reachability problem we introduce a way of encoding words into matrices. The basic observation is the structure of the following simple matrix product:

$$\begin{pmatrix} 1 & x \\ 0 & y \end{pmatrix} \begin{pmatrix} 1 & x' \\ 0 & y' \end{pmatrix} = \begin{pmatrix} 1 & x' + xy' \\ 0 & yy' \end{pmatrix}$$

where $x, x', y, y' \in \mathbb{Q}$ (and xy',yy' are ordinary products rather than concatenations of words).

For words $w = a_1 \dots a_m$ over the alphabet $\{0,1\}$ we define a map $\psi(w) := \psi'(a_1) \cdots \psi'(a_m)$ as a product of matrices of the form

$$\psi'(a_i) := \begin{pmatrix} 1 & a_i \\ 0 & 2 \end{pmatrix}.$$

Similarly, define $\phi(w) := [\psi'(a_m)]^{-1} \cdots [\psi'(a_1)]^{-1}$.

Using the above structure of the matrix product one can see that ψ is an isomorphism between the monoid $\{0,1\}^*$ and the one generated by the matrices $\psi'(0), \psi'(1)$ and the identity matrix $\mathbf{1}$ if we set $\psi(e) := \mathbf{1}$.

Proposition (Reachability is undecidable for affine transformations in $\mathbb{Q}^2$)**.** *Let $k \in \mathbb{N}$ be such that PCP with k "dominos" is undecidable. Then there is no algorithm which upon input of a set $\{T_i : \mathbb{Q}^2 \to \mathbb{Q}^2\}$ of $i = 1, \dots, k$ affine transformations decides whether or not there is a finite sequence of these transformations for which $T_{i_n} \cdots T_{i_1}(0,1) = (0,1)$.*

Proof. Consider any PCP instance given by k pairs $(X_1, Y_1), \dots, (X_k, Y_k)$ of words in $\{0,1\}^*$. Exploiting the properties discussed above, we obtain that for some non-empty word $w \in \{1, \dots, k\}^*$, of length n say, we have that $X_{w_1} \cdots X_{w_n} = Y_{w_1} \cdots Y_{w_n}$ iff $\phi(X_{w_n}) \cdots \phi(X_{w_1})\psi(Y_{w_1}) \cdots \psi(Y_{w_n}) = \mathbf{1}$.

This equation can be expressed in terms of an application of affine transformations which we define as $T_i\colon (x,y) \mapsto (x', y')$ via

$$\begin{pmatrix} 1 & x' \\ 0 & y' \end{pmatrix} = \phi(X_i) \begin{pmatrix} 1 & x \\ 0 & y \end{pmatrix} \psi(Y_i).$$

□

The dimension of affine transformations as well as their number (we can choose $k = 7$) is remarkably small.

Note that every affine transformation T on $\mathbb{Q}^2$ can be embedded into a linear transformation G on $\mathbb{Q}^3$ in a way such that $T\colon (x,y) \mapsto (x', y')$ becomes $G\colon (x, y, 1) \mapsto (x', y', 1)$. This leads to the following:

Corollary (Reachability is undecidable for linear transformations in $\mathbb{Q}^3$)**.** *Let $k \in \mathbb{N}$ be such that PCP with k "dominos" is undecidable. Then there is no algorithm which upon input of a set $\{G_i\colon \mathbb{Q}^3 \to \mathbb{Q}^3\}$ of $i = 1, \dots, k$ linear transformations decides whether or not there is a finite sequence of these transformations for which $G_{i_n} \cdots G_{i_1}(0,1,1) = (0,1,1)$.*

As in the case of matrix mortality, one trade the number of matrices by their dimensions: the above reachability problem has also been shown undecidable for two rational 16×16 matrices.

In a similar vein, many other matrix problems with a similar flavor can be shown to be generally undecidability: among them "set-to-point reachability" and the "zero-in-the-upper-right-corner" problem...

16 Hilbert's tenth problem

Hilbert's tenth problem was: find a procedure (German: "Verfahren") which decides whether or not any multivariate polynomial with integer coefficients has an integral root. 70 years after Hilbert formulated this within his now famous list of 23 problems, it was proven to be unsolvable in the sense that no such procedure/algorithm can exist. The undecidability of the existence of integral roots turned out to be the consequence of a deep equivalence which will be the content of this lecture. Lecture 9

Definition (diophantine predicates and relations). *A predicate P on $\mathbb{N}^k$ is called* diophantine *iff there is an $n \in \mathbb{N}$ and a polynomial p with integer coefficients in $k+n$ variables, such that $P(x) \Leftrightarrow \exists y \in \mathbb{N}^n \colon p(x,y) = 0$.*

A set or relation $S \subset \mathbb{N}^k$ is called diophantine *iff $x \in S$ is a diophantine predicate.*

A function $f \colon \mathbb{N}^k \to \mathbb{N}$ is called diophantine *iff its graph $\big(f(x), x\big) \subset \mathbb{N}^{k+1}$ is a diophantine set.*

The definition allows for polynomials of arbitrary (though finite) degree. The following trick by Skolem, however, shows that we can trade the degree with the number of variables to the extent that finally we may restrict ourselves to polynomials of degree at most four:

Lemma (Reduction to degree four). *If $S \subset \mathbb{N}^k$ is a diophantine set, then there is an $m \in \mathbb{N}$ and a polynomial q in $m+k$ variables with integer coefficients and of degree at most four such that $S = \{x \in \mathbb{N}^k \mid \exists z \in \mathbb{N}^m \colon q(x,z) = 0\}$.*

Proof. By assumption there is a polynomial p with integer coefficients such that $S = \{x \in \mathbb{N}^k \mid \exists y \in \mathbb{N}^n \colon p(x,y) = 0\}$. The construction of the new polynomial is then done recursively: for all monomials in p which have degree larger than two, introduce new variables $u_1, u_2, \ldots$ defined as a product of the first two variables of the corresponding monomial. Inserting the new variables then leads to a new polynomial $p_1(x,y,u)$ whose maximal degree is one less than that of p and $p(x,y) = p_1(x,y,u)$ if we impose the defining constraints for the u_is. Iterating this procedure, we can obtain a sequence of polynomial in more and more variables which eventually is at most quadratic in all variables. Suppose p_n is this quadratic polynomial. The imposed constraints on the new variables which guarantee that $p(x,y) = p_n(x,y,u,\ldots)$ can now be formulated in terms of the existence of integral roots of quadratic polynomials with integer coefficients. Denote by $c_1, c_2, \ldots$ those polynomial. That is, if we have for instance $u_1 := x_2 y_7, u_2 := u_1 x_2$, then we define $c_1(u_1, x_2, y_7) := u_1 - x_2 y_7$ and $c_2(u_2, u_1, x_2) := u_2 - u_1 x_2$. In this way, we achieve that $p(x,y) = 0$ iff $\exists u \colon p_n^2 + \sum_i c_i^2 = 0$, where we denote by u the collection of all variables added to x and y. Hence, the polynomial

$q := p_n^2 + \sum_i c_i^2$ leads to the sought quartic characterization of the diophantine set S. □

The following is (the simple) half of this lecture:

Proposition (Recursive enumerability of diophantine sets)**.** *Every diophantine set is recursively enumerable.*

Proof. Let $p(x, y)$ be a characterizing polynomial for the diophantine set. The statement follows from observing that $p(x, y) = 0$ is a primitive recursive predicate which we call $P(x, y)$ and from recalling that a recursively enumerable set S is exactly one for which there is a primitive recursive predicate for which $S = \{x \mid \exists y \colon P(x, y)\}$. □

A basic property of recursively enumerable sets is that the class is closed w.r.t. unions and intersections. This is easily seen to hold also for diophantine sets:

Proposition. *The class of diophantine predicates is closed w.r.t. (i) conjunction, (ii) disjunction and (iii) the use of existencial quantifiers.*

Proof. Let upper case Ps be diophantine predicates and lower case ps their characterizing polynomials. Then

$$\begin{aligned} P_1(x) \wedge P_2(x) &\Leftrightarrow \exists y_1, y_2 \colon p_1(x, y_1)^2 + p_2(x, y_2)^2 = 0, \\ P_1(x) \vee P_2(x) &\Leftrightarrow \exists y_1, y_2 \colon p_1(x, y_1) p_2(x, y_2) = 0, \\ \exists x_2 \colon P(x_1, x_2) &\Leftrightarrow \exists y, x_2 \colon p(x_1, x_2, y) = 0. \end{aligned}$$

□

16.1 Examples of diophantine predicates

Example 1. $x < y$ is diophantine since it holds iff $\exists z \in \mathbb{N} \colon x + z + 1 - y = 0$.

Example 2. $x \leq y$ is diophantine since it holds iff $\exists z \in \mathbb{N} \colon x + z - y = 0$.

Example 3. $(x = y) \mod z$ is diophantine since it holds iff $\exists n \in \mathbb{N} \colon (x + nz - y)(y + nz - x) = 0$.

In a similar vein, we can again use closedness w.r.t. conjunction in order to show that a concatenation of diophantine functions, f and g say, is again diophantine, since the predicate y=f(g(x)) holds iff $y = f(z) \wedge z = g(x)$.

16.2 Examples of diophantine functions

Example 1. The remainder function characterized by the predicate $r = \operatorname{rem}(a, b)$ is diophantine since this holds iff $(r = a) \mod b \wedge r < b$.

Example 2. Similarly, the quotient function which corresponds to the predicate $q = \operatorname{quo}(a, b)$ is diophantine since this holds iff $0 \leq a - qb < b$.

Needless to say, all polynomials with integer coefficients are diophantine functions.

The by far most tricky example of a function which can be shown to be diophantine is the exponential function. In fact, this has for a long time been the bottleneck in the undecidability proof for Hilbert's tenth problem. We wont prove this. Once this achieved, we can go through the proof of the fact that every Turing computable function is recursive. Since the encoding of an arbitrary Turing machine in terms of recursive functions uses only functions for which we now known that they are diophantine, one can with a little bit of effort see that the predicate "the Turing machine halts" is a diophantine predicate. Following these lines leads to two remarkable consequences:

Theorem (Davies, Putnam, Robinson, Matiyasevich). *Every recursively enumerable set is diophantine.*

Theorem (Existence of universal polynomials). *There is an $n \in \mathbb{N}$ and a polynomial p with integer coefficients such that for any recursively enumerable set $S \subseteq \mathbb{N}$ there exists an $s \in \mathbb{N}$ so that*

$$S = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N}^n \colon p(s, x, y) = 0\}.$$

Denoting by (n, d) the number of variables and the maximal degree of a polynomial, then there are universal polynomials known for $(n, d) = (58, 4)$ (note that the possibility of having $d = 4$ follows from Skolem's trick) to $(n, d) = (9, 1.6 * 10^{45})$.

The fact that diophantine sets and recursively enumerable sets are the same leads to the sought undecidability of Hilbert's tenth problem:

Corollary (Undecidability of H10). *Let $\mathcal{P}$ be the class of polynomials with integer coefficients and of degree at most four. (i) There is no algorithm which upon input of any element $p \in \mathcal{P}$ decides whether or not p has an integral root, and (ii) there is no algorithm which upon input of any element $p \in \mathcal{P}$ decides whether or not p has a non-negative integral root.*

Proof. Assume there would be an algorithm for deciding integral roots. Then there would be one for non-negative integral roots as well, since we can exploit the Lagrange four square theorem to the end that $0 \in p(\mathbb{N}^n) \Leftrightarrow 0 \in p'(\mathbb{Z}^{4n})$.

Then for any diophantine set $S = \{x \in \mathbb{N}^k \mid \exists y \in \mathbb{N}^m \colon p(x, y) = 0\}$, the hypothetical algorithm could be used in order to decide $x \in S$ for any x. In other words, every diophantine set would be a recursive set. However, we known that there are non-recursive sets within the recursively enumerable sets. An since the latter are exactly the diophantine sets, the assumption of such an algorithm leads to a contradiction. The fact that we can restrict ourselves to degree at most four follows from Skolem's lemma. □

While for polynomials with maximal degree two, there exists such an algorithm, the case of maximal degree three is still open. Similarly, for

rational (rather than integral) roots, decidability is an open problem. For real roots, on the other hand, a result of Tarski implies that the problem then becomes decidable.

As we will prove in the exercise, one can extend the above undecidability result in the following direction: let C be any set of cardinal numbers $\leq \aleph_0$ which is neither empty nor does it contain all such cardinal numbers. Then the question of whether or not the number of non-negative integral roots of a polynomial is in C turns out to be undecidable as well. The proof is a reduction from $C = \{0\}$ – the undecidability of Hilbert's tenth problem.

Proposition (Prime number producing polynomials)**.** *There is a polynomial $q(y_1, \dots, y_n, x)$ with integer coefficients such that the positive integers in its range are exactly all prime numbers in the sense that*

$$q(\mathbb{N}^{n+1}) \cap \mathbb{N} \setminus \{0\} = \text{ the set of all primes.}$$

Proof. Primes form a recursively enumerable and thus diophantine set S. This implies that there is a polynomial p with integer coefficients such that $S = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N}^n \colon p(x, y) = 0\}$. Defining $q(y, x) := x(1 - p(x, y)^2)$ then gives the sought polynomial since this is positive iff p has a root in which case indeed $q(y, x)$ takes on the value of the corresponding prime. □

Following the remark regarding universal polynomials, we obtain that for prime number producing polynomials ten variables suffice.

A similar construction leads to the following:

Proposition (All computable functions are polynomials - somehow)**.** *Let $f \colon \mathbb{N} \to \mathbb{N}$ be any partial recursive function. There exists a polynomial q with integer coefficients such that for all $x, y \in \mathbb{N}$:*

$$y = f(x) \;\Leftrightarrow\; \exists x_0, \dots, x_n \in \mathbb{N} \colon y = q(x, x_0, \dots, x_n).$$

Proof. The graph of f is recursively enumerable and thus diophantine. So $y = f(x)$ holds iff for a certain polynomial p we have $\exists x_0, \dots, x_n : (1 - p(x_0, \dots, x_n, x)^2) > 0 \wedge x_0 = y$.

This in turn is equivalent to $\exists x_0, \dots, x_n \colon (x_0+1)(1-p(x_0, \dots, x_n, x)^2) = y + 1$. Therefore the sought polynomial can be defined as $q(x, x_0, \dots, x_n) := (x_0 + 1)(1 - p(x_0, \dots, x_n, x)^2) - 1$. □

Many other interesting consequences can be derived from the equivalence of recursively enumerable and diophantine sets. One can for instance write down a polynomial which has an integral root iff the Riemann hypothesis fails to hold.

Another interesting direction of research is whether or not the proof of undecidability of Hilbert's tenth problem can be done within what is called bounded arithmetic. In that case one would be able to solve a longstanding problem in theoretical computer science and prove that NP= co-NP...