(ADVANCED) PROOF AND COMPUTATION

IMMI HALUPCZOK

Semester II 2015/2016

1. Introduction

1.1. **Hilbert's Program.**

At the end of the 19th century, Cantor and Frege developed "naive set theory", which was supposed to allow to treat sets as actual mathematical objects (in particular infinite sets). Whether "infinity" really makes sense was controversial. Concrete arguments against it:

- One can prove that $\mathbb{N}$ has the same number of elements as $\mathbb{N} \setminus \{0\}$; so $\infty = \infty - 1$, so $0 = -1$?? (See "Hilbert's hotel"; e.g. on youtube: `http://www.youtube.com/watch?v=faQBrAQ87l4`)
- **Russell's paradox:**

  According to Cantor's definition, for any property, there exists a set whose elements are exactly those objects that have this property.

  Examples: "Being an even number" is a property, so there is the set which contains all even numbers (and nothing else). And "Being a set" is a property, so there is a set which contains all sets. In particular, that set contains itself.

  Now consider the property "being a set which does not contain itself". According to Cantor, we can form the set A of all things x which have that property, i.e., $x \in A$ iff x is a set and $x \notin x$. Now the question is: Is $A \in A$? (It seems that neither $A \in A$ nor $A \notin A$ can be true.)

Your intuition might tell you that "$x \in x$" does not make sense: how can something contain itself? This depends on what one means by "set". Cantor and Frege took the point of view that a set is not an actual object which "contains" its elements, but just an abstract concept to speak about a bunch of things which have some common property. If that sounds too counter-intuitive to you, one can reformulate the paradox using "properties" instead of sets:

Properties also have properties; for example the property "being an even number" has the property that it only holds for numbers. Thus it makes sense whether a property has itself as a property. Now consider the property (let me call it P) that a property does not have itself as a property. Does P have itself as a property?

This shows that at least, one should be really careful when working with sets. How can one be sure that one does not get contradictions? In general, how can we make sure that a proof is really valid? Often, when it is unclear whether a proof is valid, one can add more details / fill in more intermediate steps, but at some point, one has to stop and simply believe certain things.

To get rid of any uncertainities, at the beginning of the 20th century, Hilbert proposed to give a precise definition of what a valid proof should be. Roughly, the idea is the following:

- Fix a precise list of **axioms**: statements which are basic enough so that we don't doubt that they are true.
- Fix **inference rules**: rules which tell us precisely how, from some true statements, we can derive other true statements. (Again, these rules should be basic enough that we don't doubt them.)
- A **formal proof** is a proof which uses only these axioms and inference rules, i.e. it has to start with some of the axioms, and is allowed to use only the inference rules to produce new statements.

Thus once we agreed on axioms and inference rules, given any proof, we can try to insert additional steps until we obtain a formal proof, i.e., until the only things we're using are the axioms and the inference rules. If we manage to do that, we can be very sure that our proof was correct (i.e., as sure as we are that the axioms and inference rules are ok).

Such a list of axioms and inference rules is called a **deductive system**. You have seen deductive systems in Logic 1: one for propositional logic and one for predicate logic.

The problem is now to find good axioms and inference rules. If we are not careful, two things might happen:

(a) We might have too few axioms/rules, so that certain true statements cannot be proven in the deductive system.

(b) It might be possible to prove false statements – either because we one of our axioms is false, or because one of the inference rules is wrong, in the sense that it makes it possible to deduce false statements from true ones.

There's no way to make entirely sure that this does not happen (since we don't know which statements are true and which are false; after all, that's what we want to find out).

However, we might, for a start, try to ensure that our deductive system is **consistent**, i.e., that there is no statement such that one can prove that it is true *and* that it is false. Obviously, if we could prove that a statement is both true and false, then something must be wrong with our axioms and rules. In a consistent deductive system, at least things like Russel's paradox are ruled out.

However, it's easy to specify consistent but completely useless deductive systems, namely if one simply cannot prove anything. Instead, one would like that for every statement, we can either prove that it's true or that it's false (but not both); a deductive system with that property is called **complete**.

Hilbert's plan had been to choose a deductive system D as above and to *prove* that

(*) D is consistent and complete.

It might seem that we didn't gain anything, since how can we ensure that the proof of (*) is valid? Hilbert's plan was to prove (*) using only a much more elementary part of mathematics than what can be stated using D:

- First, we fix a precise notation for statements we want to talk about: We fix a finite list of symbols, and each statement has to be written using only those symbols, according to a precise grammar. (The grammar specifies which strings of symbols "make sense": e.g. "$\forall x(x = 1)$" does, whereas "$\forall = x(($" doesn't. Note that the grammar doesn't say anything about the truth of the statement.)
- Using that grammar, each axiom just becomes a string of symbols, and each inference rule just becomes a recipe on how to turn certain strings into other ones. Example: One rule could be: If you have two statements (which you already proved to be true), then you are allowed to concatenate these two statements, putting the symbol "$\wedge$" in between.
- From that point of view, (*) is only a statement about strings of symbols: For example, consistency states that if ϕ is any string of symbols which makes sense according to the grammar, then it is not possible to construct both, ϕ and $\neg\phi$ by repeatedly applying the rules to the axioms.

  Thus, while the *meaning* of the statements could be about very complicated mathematics, the proof of (*) only needs "mathematics of strings of symbols".

With Hilbert's decuctive system, we still couldn't be 100 % sure whether it's exactly the true things that can be proven, but we'd be much saver than before: For each statement, we get a definitive answer about whether it's true or not.

More about Hilbert's program: `http://plato.stanford.edu/entries/hilbert-program/`

1.2. **Gödel's Theorems.** Unfortunately, in the 1930s, Gödel proved that it is impossible to realize Hilbert's program: He proved that "useful" complete deductive systems don't exist. I need to make precise what I mean by "useful".

One way in which a deductive system could be useless is that in its grammar, only boring statements can be expressed. For example, if the only statements one can express are whether two numbers are equal or not (e.g. $15 = 15$, $2 \neq 9$), then it is easy to specify a complete deductive system, but that's not really useful to do mathematics. So Gödel assumed that the deductive system is "rich enough to do some amount of mathematics". What this exactly means will be made precise much later in this course.

There's a second way in which a deductive system can be useless. For example, consider the deductive system which has all true statements as axioms and no inference rule at all.

Obviously, it is complete (and even better: something can be "deduced" iff it's a true statement), but this is of course completely useless, since given a potential proof of a statement ϕ (which would simply consist of ϕ itself), we can't find out whether it's correct or not, since we don't know whether ϕ is an axiom. So the deductive system Hilbert was looking for should have the additional property that there's an "effective method" to find out whether a proof is valid or not. In modern terms, one would ask that there exists an *algorithm* for this. Or, in yet other words, it should be possible to program a computer so that it checks whether a proof is valid or not. A deductive system with this property is called **effective**. (Also this notion will be made more precise later in this course.)

Now we can state Gödel's results. Suppose that D is an effective deductive system which is "rich enough to do some amount of mathematics".

- Gödel's First Incompleteness Theorem (**GINC1**) says that such a D can never be complete.
  In other words, no matter how we choose the axioms and rules, either there are statements which can neither be proven nor disproven, or there are statements such which can be both, proven and disproven.
- If we could at least find a D as above and prove that it is consistent, then this would at least partly save Hilbert's program: even though there may be statements that can neither be proven nor disproven, if we are lucky, most of the "interesting" (true) statements can be proven. And if we proved that D is consistent, we're at least sure to not run into things like Russel's paradox.
  As mentioned before, proving that D is consistent is only useful if the concistency proof needs only more elementary mathematics than what can be expressed using D; otherwise, we didn't gain anything. However, Gödel proved that a consistency cannot be more elementary. Even worse, Gödel's Second Incompleteness Theorem (**GINC2**) says that if D is consistent, then even using the full power of D, one cannot prove that D is consistent.

The big goal of this course is: make all this precise and prove Gödel's Incompleteness Theorems. (The second one will be in Advanced Proof and Computation.)

And now? Does it mean that in mathematics is "unsave"? To some extend yes, but:

- We do have several deductive systems which work quite well in practice: no one ever found an inconsistency, and they are complete enough for most of usual mathematics.
- For these systems, there exist "Relativized Hilbert Programs" (started by Gentzen in the 1930s): proofs of consistency using elementary methods plus one single non-elementary axiom, but which does seem plausible. (However, that axiom typically is "very infinitary": it says that certain very big infinite sets exist.)

Two of the most important such deductive systems are:

- **PA = Peano Arithmetic**: In PA, one can formulate many kinds of statements about natural numbers, in particular many famous mathematical problems like Fermat's last Theorem, the twin prime conjecture, the Goldbach conjecture. Thus quite a lot of interesting mathematics already appears there.
  One can also speak about integers or arbitrary rationals by coding them appropriately, but not about reals.

- **ZFC** (named after its inventors Zermelo and Fraenkel, and "C" stands for "Choice", an axiom that was added later): IN ZFC, one can formulate statements about sets. A priori, statements speak about nothing else than sets, but one can encode other mathematical objects using sets (e.g.: represent a natural number n by an n-element set). In this way, everything that can be expresssed in PA can also be expressed in ZFC and much more. In fact, almost all of mathematics can be carried out in ZFC.

  To avoid Russell's paradox, ZFC is somewhat restrictive about what is a set and what isn't. There's a bunch of axioms allowing to construct sets, so whenever one has something like $\{x \mid x \text{ has a certain property}\}$, one first has to prove that this is a set before one can work with it. This feels slightly unsatisfactory, because it's not so clear whether some important sets might be missing. (Indeed, the axiom of choice was added because of some missing sets.) However, in practice, ZFC works very well.

Some "practical remarks": So PA or ZFC provide notions of "formal proofs", i.e., proofs which start with some precscribed axioms and only use some specific rules, and a computer can check whether such a proof is valid. In practice however, a formal proof is horribly long and nobody would ever write down a proof in so many details. Instead, mathematicians content themselves with writing down proofs to a certain level of detail, where they are reasonably sure that filling out the missing steps would be possible. One might also hope that in the future, computers might be able to automatically fill in the missing steps and in this way formally verify the proofs. Indeed, there already exist computer programs to do this ("coq", "isabelle"), but they are still rather bad at guessing, so one still has quite a lot of manual work.

1.3. **Overview over the course.** The course consists roughly of 3 parts:

- Algorithms and Computability:

  For Gödel's Theorems, it will be important to know exactly what it means that the validity of proofs in our deductive systems can be verified by an algorithm. In this chapter, we will make that notion and related ones precise. (We will do this using something called "register machines", which are some kind of theoretical primitive computers.)

  On our way, we will prove another surprising result: There exist (precisely specified) mathematical problems which cannot be solved by any algorithm (the example we will see is the "halting problem").

  The definition of algorithm using register machines is rather intuitive, but not very handy when it comes to relating that to formulas. Therefore, we will prove that it is equivalent to another notion called "recursive functions".
- The first Incompleteness Theorem:

  We will start by recalling Predicate Logic and Gödel's Completeness Theorem.

  The main part of the chapter consists in formulating a precise (first) version of GINC1 and in proving it.

  Next, we will have a look at Peano Arithmetic, to see that after all, this is not too bad to do mathematics.

  After that, we will prove a stronger version of GINC1. The difference between the two versions is: how "rich" do we require the deductive system to be? (Recall that a prerequisite of GINC1 is that the deductive system has to be "rich enough to do some amount of mathematics".)

  (Depending on time, this stronger version of GINC1 might only be in Advanced Proof and Computation.)
- The second Incompleteness Theorem (only Advanced Proof and Computation):

  Here, it is already pretty difficult to even state the theorem precisely. The claim is that using a deductive system D, one cannot prove that D is consistent, but for this to make sense, one needs to formulate "D is consistent" as a statement from D. The problem is how to do that.

  We will not do everything in full detail, but at least get an overview over how things work.

Reading:

I'd mostly recommend these lecture notes, but here is some additional literature:

- Any introductory book to logic, for example:
  Enderton: A Mathematical Introduction to Logic
- Cooper: Computability Theory
  Not all books about computability theory also contain the stuff about Gödel and PA; this one does.

2. Algorithms and Computability

Recall: The various rules in Hilbert's program should be specified by algorithms. To be able to prove Gödel's theorem (and even to formulate it precisely), we first have to formalize this notion.

2.1. **Algorithms and Computable Functions – informally.**

An algorithm is a set of instructions explaining how, from some "input data", one can compute some "output" or "answer".

Example 1: Input: two natural numbers x and y.
Instructions: to get the output, multiply x by 2 and add y.

Example 2: Input: a natural number x
Instructions: Go through all natural numbers which are bigger than 1 and less than x. If x is divisible by one of these numbers, then output ”no” otherwise, output "yes".

For simplicity, in the following, we will only consider algorithms which take a fixed number of natural numbers as input and which yield a single natural number as output. (We will see later that this restriction is much less severe than it looks.) In Example 2, we can simply replace "yes" by 0 and "no" by 1. [1]

An algorithm taking k natural numbers as input **computes** a function $f\colon \mathbb{N}^k \to \mathbb{N}$: $f(x_1, \dots, x_k)$ is the output of the algorithm when it gets $x_1, \dots, x_k$ as inputs. A function is called **computable** if there exists an algorithm computing it.

In Example 1, the function is $f(x, y) = 2x + y$; in Example 2, it is

$$f(x) = \begin{cases} 0 & \text{if } x \text{ is prime} \\ 1 & \text{if } x \text{ is composite} \end{cases}$$

Note: A function and an algorithm are two very different things. A function is a mathematical object which one can think of as a black box, which, if one plugs in the arguments, yields the corresponding value. The value only depends on the arguments, but apart from that, anything is allowed. Moreover, if two different black boxes always spit out the same value if they get the same input, they are considered as the same function. On the contrary, there may be different algorithms which compute the same function; for example, "add x to y and then add x once more" is a different algorithm which computes the same function as the on in Example 1. And there might be functions for which no algorithm exist (namely non-computable functions).

To obtain a precise notion of computable functions, we still have to define precisely what an algorithm is. First some more details of what it isn't.

- Everything must be entirely specified (not allowed: "choose a number")
- What one is supposed to do according to the instruction is only allowed to depend on the input (and on nothing else); in particular, for the same input, one must always obtain the same output (not allowed: "output the number of students in the room"; or "roll a dice"),
- it's not enough to specify a value, one has to say how to find the value (not allowed: "find out how many times the digit 5 appears in the decimal expansion of π")
- it should take only a finite amount of time to carry out the instructions (not allowed: "go through the decimal expansion of π and count all the digits 5 appearing there")

Now, how to make the notion of algorithm formal? (This should include specifying what a given algorithm computes.)

A modern idea: choose your favorite programming language. An algorithm is a valid program in that language. Write down a formal specification of the language.

Problem: this might depend on the programming language. (Maybe there are functions which can be programmed in some language but not in another one)

Different people gave are several quite different definitions of computability. It turned out that all of these definitions are equivalent; moreover, nobody was able to come up with a

[1] Computer scientists would rather do it the other way round.

function that intuitively should be computable but which is not computable in the sense of any of these definitions. Therefore, one now believes that this is the "right" definition. This belief is known as **Church's Thesis**.

Here are some of the possible (equivalent) definitions of being computable:

- A **Turing machine** is a very simple kind of theoretical computer (invented before real computers existed); one can consider functions which can be computed by suitably programmed Turing machines.
- A **register machine** is something which looks more like a normal programming language (but a really very simple one); one can consider functions which can be computed by such programs
- The precise mathematical definition of a Turing machine or a register machine is a bit long and cumbersome. A notion which is much shorter to define precisely is that of **recursive functions**. Therefore, that notion is often more handy for mathematical applications. On the other hand, recursive functions are much less intuitive.

In this lecture, we will start with the notion of register machines, since they are most intuitive. However, at a later point, it will be more handy to work with recursive functions, so we will prove that these two approaches yield equivalent notions of computability.

2.2. **Register Machines.** Let me first explain informally what a register machine is. A register machine has finitely many registers $r_1, \dots, r_\ell$, each of which holds a natural number. At the beginning, the first k registers contain the inputs ($k \le \ell$), and all remaining registers are set to 0. Then there is a list of instructions $I_1, \dots, I_m$ saying what the machine does. There are only three types of instructions:

- "INC(j, n)": Add 1 to register r_j and then go to instruction I_n.
- "DEC(j, n, n')": If $r_j > 0$, then subtract 1 from register r_j and then go to instruction I_n; otherwise, leave r_j as it is and go to instruction $I_{n'}$.
- "HALT": Stop the computation

When the machine reaches the HALT instruction, the output of the computation is the content of r_1. (The content of the other registers doesn't matter.)

Stupid example:
We have three registers r_1, r_2, r_3, where r_1 and r_2 are the input registers; the instructions are:
1: INC$(3, 2)$: Add 1 to r_3 and go to 2.
2: DEC$(3, 3, 4)$: Try to subtract 1 from r_3; go to 3 if that's possible and to 4 if not.
3: INC$(1, 2)$: Add 1 to r_1 and go to 2.
4: HALT
Now let's suppose we want to use this machine with inputs $(5, 3)$. Then at the beginning, we are at instruction 1 and the contents of the registers are $r_1 = 5, r_2 = 3, r_3 = 0$. We write this shortly as $(\underset{\text{instruction}}{\underset{\uparrow}{1}}\,; \underbrace{5, 3, 0}_{\text{Content of registers}})$; we call this the state of the register machine.

Instruction 1 tells us to increase r_3 by 1 and to go to instruction 2. After that, the state is $(2; 5, 3, 1)$. Next, since $r_3 = 1 > 0$, we can subtract 1 from it, so we do that and go to 3; new state: $(3; 5, 3, 0)$. Then: $(2; 6, 3, 0)$. Then we can't decrease r_3, so we leave it as it is and go to 4; new state: $(4; 6, 3, 0)$. And then we stop the computation. The result of the computation is r_1, i.e., 6 in this case. Thus for the function $f\colon \mathbb{N}^2 \to \mathbb{N}$ computed by this register machine, we have $f(5, 3) = 6$.

Now let us make this formal.

Definition 2.2.1. *A **register machine** M is a list $M = (k, I_1, \dots, I_m)$ where k is a natural number (the "number of inputs") and where each I_i is an "instruction": it is either* INC(j, n) *or* DEC(j, n, n') *or* HALT *for some natural numbers j, n, n' with $j \ge 1$ and $1 \le n, n' \le m$.*

*Let ℓ be the maximal value of j appearing in the instructions, or $\ell = k$ if k is bigger than that maximum. We call ℓ the **number of registers** of M.*

Note that since there are only finitely many instructions, the maximum ℓ always exists.

We also have to make formal how register machine works, i.e., how, given a machine and some input numbers, one gets an output number. Note that a register machine could get into an infinite loop (e.g. "1: INC(1, 1)") and thus never yield a result. We will define what a register machine computes only for machines which, no matter what the input is, reach a HALT instruction after finitely many steps.

Definition 2.2.2. *A* ***state*** *of a register machine $M = (k, I_1, \dots, I_m)$ is a tuple $S = (i; r_1, \dots, r_\ell) \in \mathbb{N}^{1+\ell}$, where $1 \leq i \leq m$ and ℓ is the number of registers of M.*

The successor state S' of a state $S = (i; r_1, \dots, r_\ell)$ is the following.
If $I_i = \mathrm{INC}(j, n)$: $S' = (n; r_1, \dots, r_{j-1}, r_j + 1, r_{j+1}, \dots, r_\ell)$.
If $I_i = \mathrm{DEC}(j, n, n')$ and $r_j > 0$: $S' = (n; r_1, \dots, r_{j-1}, r_j - 1, r_{j+1}, \dots, r_\ell)$.
If $I_i = \mathrm{DEC}(j, n, n')$ and $r_j = 0$: $S' = (n'; r_1, , \dots, r_\ell)$.
If $I_i = \mathrm{HALT}$, then S has no successor state.

We say that M ***computes*** *the function $f\colon \mathbb{N}^k \to \mathbb{N}$ if for every $x_1, \dots, x_k \in \mathbb{N}$, the following holds. Set $S_0 := (1; x_1, \dots, x_k, 0, \dots, 0)$ and for every $t \in \mathbb{N}$, let S_{t+1} be the successor state of S_t (if it exists).*
We require that there exists a $t_0 \in \mathbb{N}$ such that S_{t_0} has no successor state and that this S_{t_0} is of the form $S_{t_0} = (\underset{\substack{\uparrow \\ \text{arbitrary instruction}}}{*}; \underbrace{f(x_1, \dots, x_k)}_{\substack{\text{first register:} \\ \text{result of computation}}}, \underbrace{*, \dots, *}_{\text{other registers arbitrary}})$.

Question: Is it possible that a register machine does not compute any function?

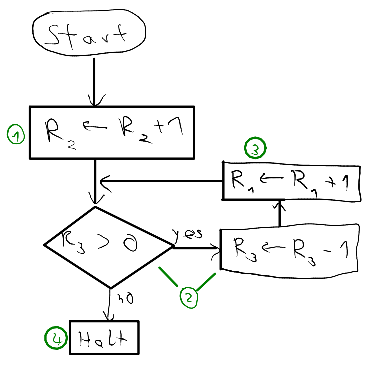
Answer: Yes. The register machine might get into an endless loop. For example:
    1: INC(1, 1)
For a register machine to compute a function, we require that it does output something (in a finite amount of time) no matter what the input is.

Now let's get started constructing some very basic register machines.

Exercise 2.2.3. Write down a register machines which do the following:

(1) Clear register r_1, i.e., set it to 0.
(2) Set r_1 to 2 (no matter what it was before).
(3) If, at the beginning, r_1 is arbitrary and r_2 is 0, then at the end, r_1 should be 0 and r_2 should have the previous value of r_1.
(4) Copy the content of r_1 to r_2. (I.e., at the end, both registers r_1 and r_2 should have the value which r_1 had at the beginning.)

Instead of writing down the register machines as lists of instructions, we often draw diagrams. Here is the example from Section 2.1:



Some explanations:

- The notation "$r_j \leftarrow$ *something*" means "Replace r_j by *something*".
- Normal boxes are commands, diagonal boxes are conditionals.

Sometimes, I will call a list of instructions as in Definition 2.2.1 a "formal register machine". In contrast, any other description of a register machine, like a diagram, will be called an "informal register machine". (It just has to be clear that it is somehow possible to turn the informal description into a formal register machine.) Here's another example of how an informal register machine could be written down:

Input: a
1. $b \leftarrow a$
2. If $b = 0$, then halt with output a.
3. Decrease b by 1 and increase a by 2.
4. Go to 2.

Notes:

- Instead of writing r_1, r_2 for the different registers, I wrote a and b. That's ok, since it doesn't really matter which number is stored in which register. (However, one should then specify which ones are the inputs and which one is the output.)
- As before, "$b \leftarrow a$" means: copy the content of a to b. Note that a should not be cleared while doing this, so to turn this into a formal register machine, one will have to use Exercise 2.2.3 (4).
- In those informal machines, if not specified otherwise, the instructions are executed one after the other.

Exercise 2.2.4. (1) What function does this register machine compute?
(2) Write down the corresponding formal register machine.

Let's check that register machines can do useful things:

Lemma 2.2.5. *The following functions can be computed by register machines.*

(1) $f(x, y) = x + y$
(2) $f(x, y) = 0$ *if* $x = y$ *and* 1 *otherwise.*
(3) $f(x, y) = 0$ *if* $x \geq y$ *and* 1 *otherwise.*
(4) $f(x, y) = x - y$ *if* $x \geq y$ *and* 0 *otherwise.*
(5) $f(x, y) = x \cdot y$
(6) $f(x, y) = 0$ *if* $y \neq 0$ *and* x *is divisible by* y*, and* 1 *otherwise.*
(7) $f(x, y) = \lfloor \frac{x}{y} \rfloor$ *if* $y \neq 0$ *and* 0 *if* $y = 0$*.* *(*$\lfloor \frac{x}{y} \rfloor$ *means:* $\frac{x}{y}$ *rounded down.)*

Exercise 2.2.6. Prove the lemma.

For that proof, note that once we know that a function $f\colon \mathbb{N}^k \to \mathbb{N}$ can be computed by a register machine (say, M_f computes f), we can use f in informal descriptions of other register machines, by e.g. writing "$r_j \leftarrow f(r_{j_1}, \dots, r_{j_k})$". Indeed, to turn this into a formal register machine, one simply has to insert M_f at the appropriate place, modified in such a way that is uses different registers than the remainder of the register machine (and with the HALT instruction removed).

And in a similar way, if we have a register machine which tests whether something holds and outputs (e.g.) 0 for yes and 1 for no, then we can use that as a condition in informal register machines. As an example, once Lemma 2.2.5 (3) is proven, we can write "if $r_4 \geq r_7$ then do xxx".

Here is an example of this: We have a formal register machine computing $f(x) = 2x$; using that, we can easily write down an informal register machine computing $g(x) = 2^x$. We then turn that into a formal register machine:

M_f computes $f(x) = 2x$:

Start state: $(1; x, 0)$; first, we move x out of the way

1: $\mathrm{DEC}(1, 2, 3)$
2: $\mathrm{INC}(2, 1)$

State now: $(3; 0, x)$; now put two times r_2 into r_1

3: $\mathrm{DEC}(2, 4, 6)$

```
    4: INC(1, 5)
    5: INC(1, 3)
Final state: (6; 2x, 0)
    6: HALT
```

M_g computes $g(x) = 2^x$, informal machine:

```
Input: $x$
1. $y \leftarrow 1$
2. If $x = 0$:
3.     Halt with output $y$.
4. Decrease $x$ by 1.
5. $y \leftarrow 2y$
6. Go to 2.
```

To turn this into a formal register machine, we need to do the following:

- Let's say we use r_1 for x and r_2 for y. (Then the input is in the right register at the beginning.)
- In line 2, we're supposed to output y; so before halting, we'll have to move that to r_1.
- In line 4, we'll use M_f from above, but with register numbers modified: instead of using registers r_1 and r_2, we'll have it use registers r_2 and r_3 (so that the input and ouput of M_f are r_2, which is y, as desired).

Here's the result: M_g, formally:

```
Line 1 (Set y to 1):
    1: INC(2, 2)
Lines 2 and 4:
    2: DEC(1, 3, 8)
Line 5: The modified M_f:
    3: DEC(2, 4, 5)
    4: INC(3, 3)
    5: DEC(3, 6, 2)        (After computing f(y), go directly back to 2)
    6: INC(2, 7)
    7: INC(2, 5)
Line 3: Need to output the content of r_2, so move that to r_1:
    8: DEC(2, 9, 10)
    9: INC(1, 8)
    10: HALT
```

Two more remarks:

- When moving r_2 to r_1 (in 8, 9), we used that we know that r_1 is already 0 at that moment. (If it wouldn't be, we'd first have to clear it.)
- The machine computing f only works properly if the non-input register (which now is r_3) is 0 at the beginning. Since the machine is run several times to compute 2^x, one should, a priori, set r_3 to zero again after each time M_f is used. However, in this particular example, it happens that r_3 is zero anyway when M_f has finished its computation.

To end this section, here are two trick questions (but an important ones):

Exercise 2.2.7. (1) Does there exist a register machine which takes no input at all and outputs the smallest even number ≥ 4 which cannot be written as a sum of two primes, or 0 if no such number exists. (Note that the Goldbach Conjecture – which is still open – states that no such number exists.)

(2) You will certainly agree that there exists a register machine (which takes no input and) which outputs the number $0 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$. But can you also find a very short register machine doing that?

Note that both questions only specify what the *output* of the register machines whould be. Neither question specifies *how* the machine is supposed to produce that output. (For example, the second question was *not*: Specify a register machine which *carries out* the computation $0 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$.)

2.3. **Computability and Decidability.** Suppose we are given a function $f\colon \mathbb{N}^k \to \mathbb{N}$. Then we can look for an algorithm (i.e., a register machine) computing that function. Maybe for certain functions f, such an algorithm is difficult to find, but do there exist functions for which no algorithm at all exists? We will see that the answer is yes. We introduce a corresponding notion.

Definition 2.3.1. *A function $f\colon \mathbb{N}^k \to \mathbb{N}$ is called **computable** if there exists a register machine M which computes it.*

It will also be handy to have a notion of whether a yes-no-question (like: "Is x a prime?") can be answered by a register machine, so let's also introduce a notion for that.

Recall that a k-ary relation on $\mathbb{N}$ is a property which may or may not hold for k-tuples of natural numbers. (For example "less than" is a binary relation which holds for pairs a, b iff a is less than b.) More formally:

Definition 2.3.2. *A k-**ary relation** R (on $\mathbb{N}$) is just a subset of $\mathbb{N}^k$ (for some $k \geq 1$). If $x_1, \dots, x_k$ are natural numbers, we say that "R holds for $x_1, \dots, x_k$" if the tuple $(x_1, \dots, x_k)$ lies in the set R. Often, we also say (and write) that "$R(x_1, \dots, x_k)$ holds" or "$R(x_1, \dots, x_k)$ is true".*

(Of course, one can also consider relations on other sets than $\mathbb{N}$, but we won't need anything else in this lecture.)

Now, given a relation, we can ask whether a register machine can decide whether the relation holds. We use the convention that an output of 0 means "yes" and an output of 1 means "no". Thus we get the following definition.

In the following, I will use $\vec{x}$ as a short hand notation for $x_1, \dots, x_k$.

Definition 2.3.3. *The **representing function** of a k-ary relation R (on $\mathbb{N}$) is the function*

$$K_R\colon \mathbb{N}^k \to \mathbb{N}, \vec{x} \mapsto \begin{cases} 0 & \text{if } R(\vec{x}) \text{ holds} \\ 1 & \text{if } R(\vec{x}) \text{ does not hold.} \end{cases}$$

*A relation R is **decidable** if its representing function K_R is computable.*

Given that a relation is just a subset of $\mathbb{N}^k$, we will sometimes also speak of sets being decidable or not; i.e.: we say that a set $X \subseteq \mathbb{N}^k$ is decidable if there exists a register machine which outputs 0 if the input tuple lies in X and 1 otherwise.

Now let's consider a few examples concerning computability and decidability.

Exercise 2.3.4. Check that the following functions/relations are computable/decidable.

(1) $R(x)$ holds if and only if x is 2, 3 or 8.
(2) $f(x) = 2^x$.
(3) $R(x)$ holds if and only if x is prime.
(4) Any constant function.

Exercise 2.3.5. Can you come up with functions or relations where it is a challenge to check that they are computable/decidable, or which might be not computable/decidable at all?

And now let's get to questions similar to the trick questions at the end of the previous section:

Exercise 2.3.6. Which of the following are clearly computable/decidable, which are clearly not computable/decidable, and where is it not clear?

(1) $R(x, y)$ holds iff every multiple of x is also a multiple of y.
(2) $f(x)$ = the smallest even number ≥ 4 which cannot be written as a sum of two primes, or 0 if no such number exists (cf. Exercise 2.2.7.)
    Note that this is a constant function.

(3) $R(x)$ holds iff $x < 10$ and the digit x appears infinitely many times in the decimal expansion of π. (Note: It is an open conjecture that this is indeed the case for all digits.)

Some trials to describe what "being computable" means intuitively:

- I'd like to say that intuitively, a function f is computable if you are able to "somehow" determine $f(x)$ when given x. However, some of the examples above show that this is not entirely true. And here is a more extreme example:
  Let $f(x) = n$ be a constant function, where n is a number I chose secretly. Then you cannot determine $f(x)$ as long as I didn't tell you what n is. However, whether a function is computable should obviously not depend on what you happen to know. (Otherwise, it would have to be a notion of "computable by some particular person" or so.) And indeed: f is a constant function, so it is computable even if you can't find out n.
- So next trial: f is computable if you are able to determine $f(x)$ when given x *after somebody has given you any knowledge you might need to know.*
  This also doesn't really work (or at least, it's not yet precise enough). Somebody might simply give you an infinite list with all the function values; then you could determine $f(x)$ by looking it up in the list.
  In fact, a register machine could also do this, e.g.:
  ```
      Input: $x$
   1. If $x = 0$ then halt with output 7.
   2. If $x = 1$ then halt with output 0.
   3. If $x = 2$ then halt with output 9.
   etc.
  ```
  No matter what x is, this machine will output the desired value. Does that mean that *any* function is computable?
  No. If we look back at our definition, we see that a register machine should have only *finitely many instructions*; here, this is obviously violated.
- So here's a third trial to describe computability in an intuitive way: f is computable if you are able to determine $f(x)$ when given x after somebody has given you *a finite amount of knowledge you might need to know.*
  That's of course rather vague, but at least it's correct. (I'm happy about feedback on whether you find this useful.)

Note that up to now, we did not find a single example of something which is clearly not computable. Indeed, that's not so easy. However, there is an easy proof that non-computable functions exist.

Proposition 2.3.7. *Not every function is computable. And not every relation is decidable.*

There is a simple proof of this, namely: there are "more" different functions in one variable than there are register machines. However, since this proof does not say anything about how to find an explicit non-computable function, I will not go too much into details. (We'll see later how to obtain some non-computable functions explicitly.)

Proof of 2.3.7. There are (1) only *countably* many register machines but (2) *uncountably* many functions from $\mathbb{N}$ to $\mathbb{N}$; thus there cannot be a register machine for every such function.

To prove (1), one needs to find an injective map from the set of all register machines to $\mathbb{N}$, i.e., one has to "encode" register machines by numbers so that no two different machines have the same number. We'll see later how one can do this.

To prove (2), note for example that the real interval $[0, 1]$ is uncountable and that each number z in that interval yields a function which sends n to the nth digit of z; in this way, we obtain uncountably many functions. $\square$

Later in the lecture, we will explicitly specify a function that is not computable. Once one has one such function, it becomes easier to also find other ones.

Example: If $f\colon \mathbb{N} \to \mathbb{N}$ is not computable, then neither is $g(x) := 2 \cdot f(x)$.

The formal way to prove such statements is always by contradiction: We suppose that g is computable. Then we (try to) deduce that f is also computable, hence contradicting the assumption that f is not computable.

Deducing computability of f from computability of g also works in the same way: We specify a register machine for f using the register machine for g. In the example, a register machine for f can be obtained by taking the one for g and appending instructions at the end dividing by 2. I.e., an informal machine for f is:

```
Input: $x$
1. $y \leftarrow g(x)$     (we can do that, since by assumption, $g$ is computable.)
2. $y \leftarrow y/2$     (Note that we know that $y$ is even, by definition of $g$.)
3. Halt with output $y$.
```

One can easily specify many other examples where non-computability of f implies non-computability of g, e.g. $g(x) = f(x) + 1$, $g(x) = f(x) \cdot f(x)$; the point is that from the value of g, one can get back to the value of f. However, one has to be careful:

Example: There does exist a non-computable function $f\colon \mathbb{N} \to \mathbb{N}$ such that nevertheless $g(x) := f(2x)$ is computable.

Let me first give an intuitive explanation: Using g (i.e., assuming that g is computable), we can compute $f(x)$ for even x; however, we don't have any control over $f(x)$ for odd x, and that might be a problem. To make this more precise, let's formulate the situation as a lemma:

Lemma 2.3.8. *Suppose that $f, g_1, g_2\colon \mathbb{N} \to \mathbb{N}$ are functions with $g_1(x) = f(2x+1)$ and $g_2(x) = f(2x)$. Then f is computable iff both, g_1 and g_2 are computable.*

Intuitively, the reason is that "the information contained in f is the same as the one from g_1 and from g_2 together". But let's prove this formally:

Proof. Suppose first that f is computable. Then we can obviously compute g_1 and g_2; for example, for g_1:

```
Input: $x$
1. $y \leftarrow 2x + 1$
2. Output $f(y)$.
```

Now suppose that g_1 and g_2 are computable. Then f can be computed as follows:

```
Input: $x$
1. If $x$ is even, then halt with output $g_2(x/2)$
2. Otherwise, halt with output $g_1((x-1)/2)$.
```

□

Using this lemma, we can prove the claim from the example with $g(x) = f(2x)$: Let h be *any* non-computable function (we know that they exist, by Proposition 2.3.7), and choose (for example)

$$f(x) = \begin{cases} 0 & \text{if } x \text{ is even} \\ h(2x-1) & \text{if } x \text{ is odd.} \end{cases}$$

Then f is non-computable by Lemma 2.3.8, but g is constant 0 and hence computable.

To get more used to this, let's look at a few more (somewhat tricky) examples:

Exercise 2.3.9. Prove or disprove the following claims. (Note: Disproving means proving that a counter-example exists, as we just did above.)

(1) If $f\colon \mathbb{N} \to \mathbb{N}$ is not computable, then $g(x) := f(x+2)$ is not computable.
(2) If $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is not computable, then the function $g\colon \mathbb{N} \to \mathbb{N}, g(x) = f(x, 12)$ is not computable either.

(3) Let a function $f\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ be given, and suppose that the function $g(x) := f(x, 12)$ is not computable, then f is not computable either.
(4) If $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is a function such that for every b, the function $g_b\colon \mathbb{N} \to \mathbb{N}, x \mapsto f(x, b)$ is computable. Then f is also computable.
(5) If $R(x)$ and $R'(x)$ are two undecidable unary relation, then $R \wedge R'$ is undecidable, too. (Here, $R \wedge R'$ is the relation which holds for x if both, R and R' hold for x.)

Note: In (4), the difference between f being computable and all g_b being computable is the following: For f to be computable, we need a single register machine taking x and y as input (and yielding the right output). All g_b being computable means that we use different register machines for different b (and each one only takes x as input).

2.4. **Encoding sequences.** Sometimes an algorithm needs to deal with other things than natural numbers. In particular, sometimes, we want an algorithm which takes more than just a fixed number of natural numbers as input. Let us consider an example.

Question: Does there exist an algorithm which, given k numbers $x_1, \dots, x_k \in \mathbb{N}$, yields the maximum of these numbers?

Intuitively, the answer should obviously be yes.

If one *fixes* k, one can easily construct a register machine which takes k numbers as input and outputs their maximum. However, let me make the question more precise: what we want is a *single* algorithm which works for all k.

A first attempt to do this using register machines would be that the input consists of $r_1 = k$ and then $r_2 = x_1, r_3 = x_2, \dots, r_{k+1} = x_k$. However, there exists no register machine which for such an input yields the maximum of the x_i. The reason is that any fixed register machine also uses only a fixed number of registers, so no matter what register machine we choose (which is supposed to compute the maximum), one can then take a k which is bigger than the number of register that machine uses, so the machine does not even look at x_k and hence cannot produce the right output.

The solution to this problem consists in encoding the whole sequence $(x_1, \dots, x_k)$ in a single number and let that single number be the input of our register machine. Then the machine can (hopefully) decode the number, i.e., find the values of all the x_i, and determine the maximum.

There are various ways in which one could encode a sequence of numbers in a single number. One possibility consists in using the prime factor decomposition, i.e. a number whose prime factor decomposition is $2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3} \cdots p_k^{x_k}$ (where p_i is the i-th prime) could encode the sequence $(x_1, \dots, x_k)$. This does not yet quite work, since zeros at the end of the sequence get lost; for example, both, $(5, 1, 3)$ and $(5, 1, 3, 0)$ would be encoded by $2^5 \cdot 3^1 \cdot 5^3$. A solution consists in increasing each of the powers by 1; then $(5, 1, 3)$ is encoded by $2^6 \cdot 3^2 \cdot 5^4$ and $(5, 1, 3, 0)$ is encoded by $2^6 \cdot 3^2 \cdot 5^4 \cdot 7^1$. So let's use this as a definition.

Definition 2.4.1. *For numbers $x_1, \dots, x_k \in \mathbb{N}$, we write $\langle x_1, \dots, x_k \rangle$ for the* **code** *of the sequence $(x_1, \dots, x_k)$. That code is a natural number which is defined as follows:*

$$\langle x_1, \dots, x_k \rangle = 2^{x_1+1} \cdot 3^{x_2+1} \cdot 5^{x_3+1} \cdots p_k^{x_k+1},$$

where p_i is the ith prime. (We define the code $\langle\rangle$ of the empty sequence to be 1.*)*

So in our above example, we're now looking for a register machine which takes a code $\langle x_1, \dots, x_k \rangle$ as input in register r_1, and which outputs the maximum of all the x_i. For this, the register machine needs to be able to "decode" the number in r_1, i.e., it needs to be able to find out what k is and what the various x_i are. The following proposition says that this is possible.

It will also be useful that a register machine can check whether a given number does at all encode a sequence; and finally, we will also need register machines to be able to *encode* sequences, i.e., given some numbers $x_1, \dots, x_k$, find the code $\langle x_1, \dots, x_k \rangle$. Now here we have a problem again: as we have already seen, we cannot expect a register machine to be able to take a variable number of input numbers and do something with them. Therefore, to construct the code of a sequence $(x_1, \dots, x_k)$, we will start with the code of the empty sequence and append the numbers one at a time: $\langle\rangle \rightsquigarrow \langle x_1 \rangle \rightsquigarrow \langle x_1, x_2 \rangle \rightsquigarrow \cdots \rightsquigarrow \langle x_1, \dots, x_k \rangle$. To be able to do this, we just need to know that it is possible, for a register machine, to append a single new number to a sequence given by a code (and output the code for the longer sequence).

Definition 2.4.2. *We define the following functions and relations related to sequences:*

(1) $\mathrm{IsSeq}(z)$ *is a unary relation which holds iff z is the code of a sequence.*
(2) *Given $z = \langle x_1, \dots, x_k \rangle$, we set* $\mathrm{len}(z) := k$.
(3) *Given $z = \langle x_1, \dots, x_k \rangle$ and a number i ($1 \le i \le k$), we set $[z]_i := x_i$.*
(4) *Given $z = \langle x_1, \dots, x_k \rangle$ and $y \in \mathbb{N}$, set* $\mathrm{append}(z, y) := \langle x_1, \dots, x_k, y \rangle$.

This definition doesn't specify what e.g. $\text{len}(z)$ should be if z is not the code of a sequence. Mathematically, it's fine to have a function that is defined only on a subset of $\mathbb{N}$, but if we want to have register machines computing them and want to be very formal, they should be defined everywhere, so let's just have a convention about that:

Convention 2.4.3. *Whenever we define a function f on a subset $A \subseteq \mathbb{N}^k$, we consider it as a function on $\mathbb{N}^k$ by settting $f(\vec{x}) = 0$ for all $\vec{x} \notin A$.*

For example, concerning the functions from Definition 2.4.2:

- If z is not the code of a sequence, we set $\text{len}(z) = 0$, $[z]_i = 0$, $\text{append}(z, y) = 0$.
- We also set $[z]_i = 0$ if $i = 0$ or $i > \text{len}(z)$.

Proposition 2.4.4. *The relation* IsSeq *is decidable; the functions* len, $(z, i) \mapsto [z]_i$ *and* append *are computable.*

Exercise 2.4.5. Using the proposition, show that our above example works, i.e., that there exists a register machine, which, given a sequence $\langle x_1, \dots, x_k\rangle$, outputs the maximum of the x_i. Specify your register machine by a diagram which uses the functions from the proposition.

Proof of Proposition 2.4.4. We first solve some intermediate problems:

A: The relation "x is prime" is decidable:
Use a loop to test for all n between 2 and $x - 1$ whether x is divisible by n; divisibility is decidable by Lemma 2.2.5.

B: The function $i \mapsto p_i$ (where p_i is the ith prime) is computable.
In a loop, consider the natural numbers one after another, checking each time whether it is prime; each time a prime is found, decrease i by one; when i reaches 0, we found the prime we were looking for.

Concering IsSeq, note: A number $z \geq 1$ is a sequence if there's no "gap" in its prime factor decomposition, i.e.: there should be a k such that z is divisible by $p_1, \dots, p_k$ and by no other primes.

Deciding IsSeq(z):
First check whether $z = 0$. If not, divide z by $p_1 = 2$ as often as possible. Then do the same for p_2, p_3, etc., until we find a p_i which does not divide z. If at that point, z is 1, then z is a code of a sequence; otherwise, it isn't.

Computing $\text{len}(z)$:
Go through the primes $p_1, p_2, \dots$ until you find one which does not divide the input.

Computing $(z, i) \mapsto [z]_i$:
Divide z as often as possible by p_i (and count how many times it is possible).

To compute $\text{append}(z, y)$:
Compute $k := \text{len}(z)$; then multiply y by p_{k+1}^{y+1}. (Use a loop to multiply it $y + 1$ times by p_{k+1}.)

More precisely, the machines for the three functions should first check whether the arguments are valid (e.g. using IsSeq) and output 0 otherwise (to comply with Convention 2.4.3). □

Exercise 2.4.6. Write out the machines appearing in this proof as informal register machines, but somewhat more precisely.

When writing down informal register machines dealing with sequences, instead of using how exactly sequences are encoded, we prefer to only use the relation and functions from Proposition 2.4.4; in this way, a reader will understand the informal machines even if he/she forgot (or never knew) how we encode sequences.

It should be clear how, using the append function from the proposition, we can construct a register machine which, say, takes a number x and returns the code for the list of squares $(1, 4, 9, \dots, x^2)$. However, we sometimes also might want to modify a given list. Here are some examples of things we might want to do:

(1) Given $z = \langle x_1, \dots, x_k \rangle$ where $k \geq 1$, output the list with the last entry removed: $\langle x_1, \dots, x_{k-1} \rangle$.
(2) Given $z = \langle x_1, \dots, x_k \rangle$ and a numbers $y \in \mathbb{N}$, return the list where y has been inserted at the beginning: $\langle y, x_1, \dots, x_k \rangle$.
(3) Given $z = \langle x_1, \dots, x_k \rangle$, return the same list backwards: $\langle x_k, \dots, x_1 \rangle$.

How can this be done *using only Proposition 2.4.4*, and not using how specifically we encoded sequences? The only way provided by the proposition to construct codes consists in appending elements to a sequence, so whatever we want to do, we'll have to construct the output sequence by starting with an empty sequence and appending elements one after the other. As an example, here is how (1) works:

```
    Input: A code of a sequence $z$
We build up the list we want to output by starting from an empty one:
    1: $r \leftarrow \langle\rangle$        (According to Definition 2.4.1, $\langle\rangle = 1$.)
$i$ is a counter which will iterate through all but the last element of $z$:
    2: $i \leftarrow 1$.
    3: If $i \geq \mathrm{len}(z)$:
    4:        Halt with output $r$
Append the $i$-th element of $z$ to the output list:
    5: $x \leftarrow [z]_i$
    6: $r \leftarrow \mathrm{append}(r, x)$
    7: $i \leftarrow i + 1$
    8: Go to 3.
```

Note: I was not careful about what the machine does if z is "invalid", i.e., not the code of a sequence. Since this is not a big deal, I will often skip this in the future.

Exercise 2.4.7. Do (2) and (3) in a similar way.

2.5. **The Halting Problem.** A famous difficult problem is the following:

Definition 2.5.1. *The **Halting Problem** is the following: given any register machine which takes no inputs at all, find out whether it will stop after some time or whether it will continue running forever (without ever reaching a* HALT *instruction).*

Easy example: The machine consisting only of the instruction "1: INC$(1,1)$" will run forever.

Difficult example: Consider a machine which systematically searches for numbers $a \geq 1, b \geq 1, c \geq 1, k \geq 3$ satisfying the equation $a^k + b^k = c^k$. As soon as it finds a solution, it stops. (Otherwise, it goes on searching.)

If a solution exists, then this machine will eventually find it and halt; otherwise, it will run forever. Nowadays, we know that the machine will run forever, since Fermat's Last Theorem says that no solution exists. However, one sees that this was quite difficult to find out.

Now one can wonder: Is there a general method (i.e., an algorithm) to find out whether register machines run forever or not? The answer is (as one might already guess) no. The goal of this section is to prove this. To make this precise, we need a way in which a register machine can take another register machine as an input; to this end, we will encode register machines by numbers. We will then prove that there exists no register machine which takes the code of a register machine M as input and decides whether M would run forever or not.

Given that we already know how to encode sequences by numbers, it is not so difficult to also encode register machines. Here is one possibility:

Definition 2.5.2. *We encode instructions as follows:*
$\ulcorner\mathrm{HALT}\urcorner := \langle\rangle$
$\ulcorner\mathrm{INC}(j,n)\urcorner := \langle j,n\rangle$
$\ulcorner\mathrm{DEC}(j,n,n')\urcorner := \langle j,n,n'\rangle$
Let $M = (k, I_1, \dots, I_m)$ be a register machine. (Recall: k is the number of inputs.) Then M is encoded by the list consisting of k and of the codes of its instructions:
$\ulcorner M\urcorner := \langle k, \ulcorner I_1\urcorner, \dots, \ulcorner I_m\urcorner\rangle$.

Let's get used a bit to register machines which operate on register machines.

Exercise 2.5.3. Check that the following functions/relations are computable/decidable:

(1) The relation IsRM(x) which holds iff x is the code of a register machine.
(2) The function sending $\ulcorner M\urcorner$ (for a register machine M) to the number of INC-instructions appearing in the machine.
(3) A function sending x to the code of a register machine M_x which takes no inputs and outputs x.
(4) A function sending the code of a register machine M to the code of a register machine M' which does the same as M, except that at the very beginning, it adds 1 to the first register.

Now let's come to the main result. We first prove a slightly different version than announced at the beginning of Section 2.5:

Theorem 2.5.4. *Consider the following relation:*
$R_{\mathrm{HALT1}}(m,x)$ holds if $m = \ulcorner M\urcorner$ for some register machine M taking a single input and M eventually halts when run with input x.
This relation R_{HALT1} is not decidable.

Proof. Suppose the contrary, i.e., suppose that M_{HALT1} is a register machine which decides R_{HALT1}. I claim that then, using M_{HALT1}, we can construct another register machine M_{par} which is paradoxical in a similar way as Russel's paradox at the beginning of the lecture.

Here is M_{par}:

```
Input: $m = \ulcorner M\urcorner$
1. If $R_{\mathrm{HALT1}}(m,m)$ holds:      (test this using $M_{\mathrm{HALT1}}$)
```

```
2.     Go to 2.     (endless loop)
3. HALT
```

The question is now: If this machine M_{par} gets its own code $m := \ulcorner M_{\text{par}} \urcorner$ as input: Does it halt or does it get into an infinite loop? We will see that neither is possible. This means that M_{par} cannot exist, and hence M_{HALT1} cannot exist.

Let us check what M_{par} does on input m. First it checks whether $R_{\text{HALT1}}(m, m)$ holds. So to see what happens next, we have to do a case distinction:

Case 1: $R_{\text{HALT1}}(m, m)$ does hold.
Then the machine goes to the right, into the infinite loop. Thus in this case M_{par} does not halt. However, by definition of R_{HALT1}, if $R_{\text{HALT1}}(m, m)$ holds, then the machine encoded by m (which is just M_{par} itself) should halt on input m. So Case 1 is not possible.

Case 2: $R_{\text{HALT1}}(m, m)$ does not hold.
Then the machine goes to the bottom and halts. However, again by definition of R_{HALT1}, if $R_{\text{HALT1}}(m, m)$ does not hold, then the machine encoded by m should not halt on input m. Again, we have a contradiction. □

Now that we have a concrete relation $R = R_{\text{HALT1}}$ which is not decidable, it is much easier to prove that also some other relations R' are non-decidable (and some functions are non-computable). As explained before, the idea for such proofs is always to prove the contrapositive, namely that if R' would be decidable, then so is R.

Exercise 2.5.5. Use this proof method to show that the following functions/relations are not computable/decidable:

(1) $f(\ulcorner M \urcorner, x) = \begin{cases} 0 & \text{if on input } x,\ M \text{ yields output } 0 \\ 1 & \text{if on input } x,\ M \text{ yields an output } > 0 \\ 2 & \text{if on input } x,\ M \text{ runs forever} \end{cases}$

(2) $R(\ulcorner M \urcorner, x, y)$ holds iff, on input x, M reaches instruction number y after some time.

(3) $f(\ulcorner M \urcorner, x) = \begin{cases} y & \text{if on input } x,\ M \text{ yields output } y \\ 0 & \text{if on input } x,\ M \text{ runs forever} \end{cases}$

Now let us use this method to deduce the version of the halting problem from the beginning of Section 2.5:

Theorem 2.5.6. *Consider the following relation: $R_{\text{HALT}}(m)$ holds iff m is the code of a register machine taking no inputs and which eventually halts.*
This relation R_{HALT} is not decidable.

Proof. As usual, we prove the contrapositive, namely: We assume that R_{HALT} is decidable and (try to) construct a machine M_{HALT1} deciding R_{HALT1}.

Thus: M_{HALT1} gets $m = \ulcorner M \urcorner$ and $x \in \mathbb{N}$ as input and it should find out whether m halts on input x; when doing so, it is allowed to use R_{HALT}. However, R_{HALT} only tests whether a machine taking no input halts, so how can it be helpful if we have an input x to deal with?

The idea is that M_{HALT1} uses m and x to construct a new machine m' that takes no input and that does the same as m would do on input x. After that, M_{HALT1} runs M_{HALT} with input m, and the output of M_{HALT} will be exactly what M_{HALT1} is supposed to output.

To obtain M from M', we simply need to insert x many INC instructions at the beginning. We have seen in Exercise 2.5.3 (4) that it is possible to do that. Thus, here is our M_{HALT1}:

```
Inputs: $m = \ulcorner M \urcorner$ and $x$
1: $m' \leftarrow \ulcorner M' \urcorner$, where $M'$ is obtained from $M$ by inserting $x$ INC-instructions
      at the beginning.
2. Modify $m'$ to change the number of inputs of $M'$ from 1 to 0
3: If $R_{\text{HALT}}(m')$ holds, then output 0, otherwise output 1.
```

Now let's check carefully that this does the right thing; this means that we have to check that $R_{\mathrm{HALT}}(m')$ holds iff $R_{\mathrm{HALT1}}(m, x)$ holds. In other words, we have to check that M' halts (on no input) iff M halts on input x.

So let's check what M' does. At its beginning, there are x many INC-instructions; after those, we have $r_1 = x$. The remainder of the machine M' is just the machine M, which now gets x as input. Thus indeed M' halts if M halts on input x. □

2.6. **Semi-Decidability.** We have classified relations into decidable and undecidable ones. We will now see that among the undecidable ones, there are some which are "at least half-way decidable" in the following sense:

Definition 2.6.1. *A k-ary relation R is* ***semi-decidable*** *if there exists a register machine M such that for every $\vec{x} \in \mathbb{N}^k$:*
M halts on input $\vec{x}$ iff $R(\vec{x})$ holds.
We say that such a machine M ***semi-decides*** *R.*

Instead of semi-decidable, it is also common to use the term "**recursively enumerable**".

Thus: Suppose we have a relation R and a machine M semi-deciding R, and suppose we want to find out whether $R(x)$ holds for some x. If M would decide R, we could just run M with intput x and would get the answer. Now we can also let M run with input x. If we're lucky, M halts and we know that R holds. However, if R does not hold, then M will never halt, and we'll never now for sure that R does not hold. From that point of view, M is "half as useful as a machine deciding R": It does give us the information we want if R holds, but it doesn't give us the information if R does not hold. (This is only supposed to be an informal explanation, but we'll later make this more precise.)

Typical examples of semi-decidable relations are relations which hold if some complicated equation with parameters has a solution. Here is a concrete example: $R(x)$ holds iff there exist two primes p, q whose difference $p-q$ is equal to x. It is easy to come up with a machine M semi-deciding R: Given x, M just needs to go through all primes q (in an infinite loop), and check whether $q + x$ is also prime. If it is, then it halts. In this way, it will halt if $R(x)$ holds and run forever otherwise.

Note: Any decidable relation is obviously also semi-decidable: If M decides a relation R, one can obtain a machine which semi-decides it by replacing the HALT instruction by instructions which go into an infinite loop if r_1 is 1 and halt only if r_1 is 0. For further reference, I'm formulating this as a lemma:

Lemma 2.6.2. *Any decidable relation is semi-decidable.*

Now let's have a slightly more complicated example of a semi-decidable relation:

Exercise 2.6.3. Prove that the following relation is semi-decidable:

$R(x_1, x_2)$ holds iff there exist y_1, y_2 such that $x_1^{y_1} + 1 = x_2^{y_2}$.

Note: A machine semi-deciding R could work by having an infinite loop going through all possible y_1 and y_2. However: How to make sure that the loop really tries all pairs?

A more general example is the following. Instead of the equation $x_1^{y_1} + 1 = x_2^{y_2}$, we can use an arbitrary decidable relation $Q(\vec{x}, \vec{y})$, and we can consider the following relation $R(\vec{x})$:

$R(\vec{x})$ holds iff there exists a $\vec{y}$ such that $Q(\vec{x}, \vec{y})$ holds.

Any $R(\vec{x})$ obtained in this way is semi-decidable: As in Exercise 2.6.3, a machine to semi-decide R simply needs to systematically go through all possible $\vec{y}$, check whether $Q(\vec{x}, \vec{y})$ holds and halt if it does.

It turns out that the converse is also true and that even a single variable y suffices for the converse:

Proposition 2.6.4. *A k-ary relation $R(\vec{x})$ is semi-decidable iff there exists a decidable $(k+1)$-ary relation $Q(\vec{x}, y)$ such that for all $\vec{x} \in \mathbb{N}^k$, we have:*
$R(\vec{x})$ holds iff there exists an $y \in \mathbb{N}$ such that $Q(\vec{x}, y)$ holds.

Proof. We have to prove the two directions of the "iff" in the first line of the proposition.

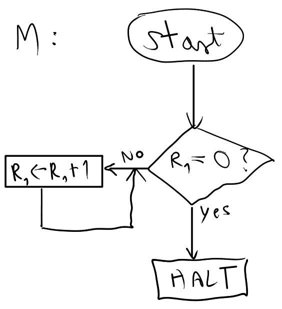
"$\Leftarrow$": see the argument above the proposition.

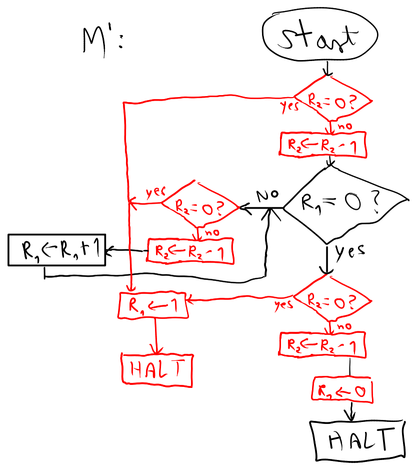
"$\Rightarrow$" The idea is as follows. We suppose that M is a machine semi-deciding R. We modify M as follows to a machine M':

M' takes an additional input y, and does the same as M, except that it halts after at most y steps, no matter what M would have done at that point. The output of M' is 1 if it "had to interrupt M" and 0 otherwise. More precisely: If M halts after at most y steps, then M' also halts and outputs 0. If M did not reach a HALT-instruction in the first y steps, then M' outputs 1 when it halts.

This machine M' halts no matter what the input is (since it carries out at most y steps of M); in particular it decides a $(k+1)$-ary relation Q. Let us now verify that $R(\vec{x})$ holds iff there exists an $y \in \mathbb{N}$ such that $Q(\vec{x}, y)$ holds. To this end, fix a tuple $\vec{x} \in \mathbb{N}^k$.
If $R(\vec{x})$ holds, then M will halt after a certain number y_0 of steps (on input $\vec{x}$), and hence M' outputs 0 on input $(\vec{x}, y)$ whenever $y \geq y_0$. Thus $Q(\vec{x}, y)$ holds for those y.
If, on the other hand, $R(\vec{x})$ does not hold, then M never halts, which means that M' outputs 1 on input $(\vec{x}, y)$ for any y. (And $Q(\vec{x}, y)$ holds for no y.)

It remains to check that we can indeed construct such an M' from M. The idea is simply to insert, between any two instructions of M, an instruction which decreases y and halts with output 1 if $y = 0$. Moreover, before any HALT-instruction of M, we set the output to 0. Instead of writing this down formally, here is an example (where M takes a single input):





□

Now let's make the claim precise that "semi-decidable is half of decidable":

Proposition 2.6.5. *A relation R is decidable iff both, R and $\neg R$ are a semi-decidable.*

Recall that the negation $\neg R$ of a relation R is the relation which holds exactly for those $\vec{x}$ for which R does not hold. Thus $\neg R$ being semi-decidable means that there exists a machine which halts if R does not hold and runs forever otherwise.

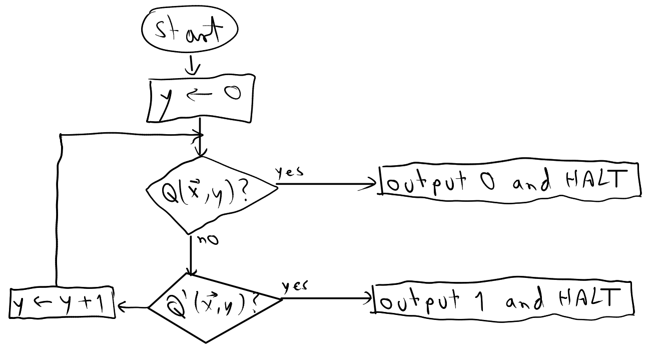
Proof. Here is an intuitive argument: Suppose that M semi-decides R and M' semi-decides $\neg R$. To decide whether R holds, we run both, M and M' in parallel. One of them will run forever and the other one will halt. As soon as one of them halts, we stop the computation, and we know whether R holds or not, depending on whether it was M or M' which halted.

This argument can be made formal: a machine deciding M'' could alternatingly carry out one step of M and one step on M'. It would of course have to use different registers for M and M', and when doing one step of M', it would have to recall at which instruction of M it has been (and vice versa). Writing this out is a bit cumbersome.

Here is another proof using Proposition 2.6.4 (which is less cumbersome to make completely formal).

By the proposition, there exist decidable $Q(\vec{x}, y)$ and $Q'(\vec{x}, y)$ such that:
  $R(\vec{x})$ holds iff $\exists y\, Q(\vec{x}, y)$ and
  $\neg R(\vec{x})$ holds iff $\exists y\, Q'(\vec{x}, y)$.

The diagram below describes a machine deciding R. Note that it halts on every input, since for every $\vec{x}$ either there exists a y such that $Q(\vec{x}, y)$ holds or there exists a y such that $Q'(\vec{x}, y)$ holds.



□

Exercise 2.6.6. Prove the following (where R, R' are k-ary relations).

- If R and R' are semi-decidable, then so is $R \wedge R'$.
- If R and R' are semi-decidable, then so is $R \vee R'$.

What are all the possibilities for R and $\neg R$ concerning decidability and semi-decidability: Make a list of all possible combinations.

2.7. **Universal Register Machines.** Question: Is R_{HALT} semi-decidable? I.e.: Does there exist a register machine M_0 which takes a code $\ulcorner M \urcorner$ as input and halts iff $\ulcorner M \urcorner$ would halt on input 0?

And: Same question for R_{HALT1}.

Answer: Yes; M_0 can "simulate" M, by doing exactly what we had written in Definition 2.2.2. And in fact, then M_0 can even do a bit better: when it halts, it can output the value which M would have given. In other words: every computation which can be carried out by any register machine M can also be carried out by this particular register machine M_0: simply use $\ulcorner M \urcorner$ as the first input for M_0. Such an M_0 is then called a **universal register machine**. Now let us make this a bit more formal.

Theorem 2.7.1. *There exists a register machine M_{univ} which takes two inputs $\ulcorner M \urcorner$ and $\langle x_1, \dots, x_k \rangle$ and which does the following: If M would halt on input $x_1, \dots, x_k$ and would give the output y, then M_{univ} halts and outputs y. Otherwise, i.e., if M would not halt on input $x_1, \dots, x_k$, then M_{univ} does not halt either.*

Proof of Theorem 2.7.1. Here is a somewhat more precise description of how M_{univ} works: we use one register to hold the current instruction from our simulation (called i below) and one to hold the values of all registers of M, encoded as a sequence (called y below).

```
Input: $m = \ulcorner M \urcorner$ and $y = \langle x_1, \dots, x_k \rangle$.
    $\ell \leftarrow$ number of registers of $M$
    if $\ell > k$, then append as many 0s to $y$ as necessary to get a sequence of length $\ell$.
    $i \leftarrow 1$
    Repeat the following:
        If the $i$th instruction of $M$ is INC$(j, n)$:
            Increase $x_j$ by 1 (i.e., modify $y$ accordingly, by rebuilding the sequence)
            $i \leftarrow n$
        If the $i$th instruction of $M$ is DEC$(j, n, n')$ and $x_j \neq 0$:
            Decrease $x_j$ by 1 (i.e., modify $y$ accordingly)
            $i \leftarrow n$
        If the $i$th instruction of $M$ is DEC$(j, n, n')$ and $x_j = 0$:
            $i \leftarrow n'$
```

```
Otherwise, i.e., if the $i$th instruction of $M$ is HALT:
    $r_1 \leftarrow x_1$
    HALT
```
□

Note: Both, in the proof of Proposition 2.6.5 and in the proof of Theorem 2.7.1, we are given some register machine M and we do some stuff with it. However, there is a fundamental difference between these two proofs. In Proposition 2.6.5 we just needed to show that there exists a machine M' with some properties. (Even though we constructed this M' explicitly, just knowing that it exists would have been enough.) In Theorem 2.7.1, we need to know that *a register machine* can do the stuff with M.

Corollary 2.7.2. *R_{HALT} and R_{HALT1} are semi-decidable.*

Proof. The following machine semi-decides R_{HALT1}:

```
Inputs: $m = \ulcorner M \urcorner$ and $x$.
1. $z \leftarrow \langle x \rangle$
2. Run $M_{\text{univ}}$ with inputs $m$ and $z$; denote its output by $y$ (if it halts).
3. Output $y$ and halt.
```

For R_{HALT}, it works similarly, but with $z \leftarrow \langle\rangle$. □

Exercise 2.7.3. (1) Show that $\neg R_{\text{HALT}}$ is not semi-decidable.
(2) Find a relation R such that neither R nor $\neg R$ is semi-decidable.

2.8. **Recursive Functions.** Register machines make it relatively easy to implement typical computable functions, but they are not so easy to handle mathematically. Therefore, we now introduce the notion of *recursive functions*. We will prove that the recursive functions are exactly the same as the functions computable by register machines. Similarly, we will define *recursive relations*, which will be the same as decidable relations.

In the following, all functions will be functions from $\mathbb{N}^k \to \mathbb{N}$ for some k.

Notation: As before, we will often write $\vec{x}$ for a tuple of variables $x_1, \dots, x_k$.

One of the key ingredient in the definition of recursive functions is the following operation:

Definition 2.8.1. *Suppose that $R(\vec{x}, y)$ is a $(k+1)$-ary relation and that for every $\vec{x} \in \mathbb{N}^k$, there exists a natural number y such that $R(\vec{x}, y)$ holds. Then we write $\mu y\,(R(\vec{x}, y))$ for the smallest y such that $R(\vec{x}, y)$ holds.*

Example: $\mu x\,(x \text{ is prime}) = 2$

Another example: $\mu y\,(x < y) = x + 1$

Recall that we write $K_R(\vec{x})$ for the representing function of a relation $R(\vec{x})$: $K_R(\vec{x})$ is 0 if $R(\vec{x})$ holds and 1 otherwise. In particular,

$$K_<(x, y) = \begin{cases} 0 & \text{if } x < y \\ 1 & \text{if } x \geq y. \end{cases}$$

Definition 2.8.2. *A function $f\colon \mathbb{N}^k \to \mathbb{N}$ is **recursive** if $f(\vec{x})$ can be written using constants (in $\mathbb{N}$), the variables x_i $(1 \leq i \leq k)$, addition, multiplication, the function $K_<$ and "$\mu y(g(\vec{x}, y) = 0)$", where g is also recursive. More formally:*

- *$f(\vec{x}) = n$ is recursive for any $n \in \mathbb{N}$*
- *$f(\vec{x}) = x_i$ is recursive for $i = 1, \dots, k$.*
- *If g and h are recursive, then so are $g(\vec{x}) + h(\vec{x})$, $g(\vec{x}) \cdot h(\vec{x})$ and $K_<(g(\vec{x}), h(\vec{x}))$*
- *Suppose that $g(\vec{x}, y)$ is a recursive function which has the property that for every $\vec{x}$, there exists a y such that $g(\vec{x}, y) = 0$. Then $f(\vec{x}) = \mu y(g(\vec{x}, y) = 0)$ is recursive. (This is called "**μ-recursion**")*

*A k-ary relation R is **recursive** if its representing function K_R is recursive. (Recall that K_R is the function on $\mathbb{N}^k$ which is 0 if R holds and 1 otherwise.)*

Example (rather useless): $f(x, y) = x \cdot y + K_<(5, x)$.

Another example: $f(x) = \mu y\,(K_<(x, y + 2) = 0)$.

This can be written in a slightly simpler way: $K_<(x, y + 2) = 0$ just means that $x < y + 2$, so we have: $f(x) = \mu y\,(x < y + 2)$.

Exercise 2.8.3. (1) Give a simpler description of the function $f(x) = \mu y\,(x < y + 2)$.
(2) Using a similar idea, show that the following function is recursive:

$$f(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$$

The word "recursion" usually means that one repeats a computation a certain number of times. Here, "μ-recursion" means that one computes $g(\vec{x}, y)$ for $y = 0, 1, 2, \dots$ until one finds a y such that $g(\vec{x}, y) = 0$.

To me, it feels a bit unsatisfactory that in Definition 2.8.2, one allows addition and multiplication; I would have preferred to work using only really basic functions, like the INC- and DEC-instructions in register machines. It is possible to define recursive functions in such a way, but then one needs an additional, other form of recursion. For our purposes, such a definition would only make things more complicated, so let's stick to Definition 2.8.2.

The main goal of this section is to prove:

Theorem 2.8.4. *A function is recursive if and only if it is computable (by a register machine).*

One direction is not very difficult, namely that every recursive function is computable.

Exercise 2.8.5. Sketch a proof that every recursive function is computable.

Remark: In the μ-recursion, some people omit the condition on $g(\vec{x}, y)$ that for every $\vec{x}$, there should exist a y such that $g(\vec{x}, y) = 0$; if no y exists, one declares $\mu y(g(\vec{x}, y) = 0)$ to be "undefined". This corresponds exactly to register machines which do not halt on input $\vec{x}$. Indeed, a register machine looking for the smallest y such that $g(\vec{x}, y) = 0$ would get into an infinite loop.

Now let's construct some more recursive functions and relations which we will need for Theorem 2.8.4.

Lemma 2.8.6. *Suppose that $R(\vec{x})$, $R'(\vec{x})$, $R''(\vec{x}, y)$ are recursive relations. Then the following functions and relations are also recursive:*

(1) $f(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$
(2) $Q(\vec{x}) \iff R(\vec{x}) \vee R'(\vec{x})$
(3) $Q(\vec{x}) \iff R(\vec{x}) \wedge R'(\vec{x})$
(4) $Q(\vec{x}) \iff \neg R(\vec{x})$
(5) $Q(x, y) \iff x = y$
(6) $Q(\vec{x}, z) \iff \exists (y \leq z)\, R''(\vec{x}, y)$
(7) $Q(\vec{x}, z) \iff \forall (y \leq z)\, R''(\vec{x}, y)$
(8) $\mathrm{Rem}(x, y) =$ *the remainder of division of x by y. (We set $\mathrm{Rem}(x, 0) := 0$ to make this function defined everywhere.)*

Exercise 2.8.7. Prove this lemma. For that proof, recall that "$R(\vec{x})$ is recursive" just means "the function $K_R(\vec{x})$ is recursive".

Note that (6) and (7) cannot be expected to work without the bound $y \leq z$. Indeed, once we'll know that recursive relations are the same as decidable ones, (6) without bound would say that if $R''(\vec{x}, y)$ is decidable, then so is $Q(\vec{x}) :\iff \exists y\, R''(\vec{x}, y)$. However, we saw that such $Q(\vec{x})$ usually are only semi-decidable. (More precisely, any semi-decidable relation $Q(\vec{x})$ can be written in the form $\exists y\, R''(\vec{x}, y)$, so if (6) would be true without bound, then every semi-decidable relation would already be decidable. . . which is not the case, as we have seen.)

Note also that this lemma gives us quite some flexibility when constructing recursive functions and relations. If, for example, $f(x)$ and $g(x)$ are recursive functions, then "$f(x) = g(x)$" is a recursive relation. (Formally, one considers the composition $K_=(f(x), g(x))$; by the lemma, $K_=$ is recursive, and from the way recursive functions have been defined, it is clear that compositions of recursive functions are again recursive.)

And once we know that a relation is recursive, we can apply μ-recursion to it: if a relation $R(\vec{x}, y)$ is recursive, then $f(\vec{x}) = \mu y\, (R(\vec{x}, y))$ is a recursive function, since this can also be written as $f(\vec{x}) = \mu y\, (K_R(\vec{x}, y) = 0)$.

The idea of the proof that computable functions are recursive is to somehow turn Definition 2.2.2 (which made precise how a register machine works) into a recursive function. Recall that in that definition, the state of a register machine was defined as a tuple $S = (i; r_1, \ldots, r_\ell)$, where i is the next instruction to be executed and $r_1, \ldots, r_\ell$ are the values of the registers. Let us already check that finding the successor state of a given state is recursive. More precisely:

Lemma 2.8.8. *Suppose that M is a fixed register machine with ℓ registers. Then the following relation is recursive:*
$R_{\mathrm{succ}_M}(i, r_1, \ldots, r_\ell, i', r'_1, \ldots, r'_\ell) :\iff$ *$(i'; r'_1, \ldots, r'_\ell)$ is the successor state of $(i; r_1, \ldots, r_\ell)$ for the machine M.*

(In particular, R_{succ_M} should always be false if $(i, r_1, \ldots, r_\ell)$ has no successor state.)

Proof. Instead of writing down the proof formally, let me show it on an example. Suppose that M is the following:

1: INC$(1,2)$
2: DEC$(2,1,3)$
3: HALT

This has two registers. The relation $R_{\mathrm{succ}_M}(i, r_1, r_2, i', r_1', r_2')$ can be expressed as follows (which makes it clear that it is recursive, using Lemma 2.8.6):

$$\begin{aligned}
&\big(i = 1 \wedge (r_1' = r_1 + 1 \wedge r_2' = r_2 \wedge i' = 2)\big)\\
&\quad\vee\\
&\big(i = 2 \wedge (\\
&\qquad (r_2 > 0 \wedge r_2' + 1 = r_2 \wedge r_1' = r_1 \wedge i' = 1)\\
&\qquad\qquad \vee\\
&\qquad (r_2 = 0 \wedge r_2' = 0 \wedge r_1' = r_1 \wedge i' = 3)\\
&\qquad)\big)
\end{aligned}$$

Explanation: What the successor state is depends on the current instruction. If, for example, $i = 1$, then the current instruction is "INC$(1,2)$", so register 1 should be incremented (i.e. $r_1' = r_1 + 1$), register 2 should stay the same ($r_2' = r_2$) and the next instruction number should be 2 ($i' = 2$). The case of a DEC-instruction is similar, but a bit more complicated since what happens depends on whether the register is 0 or not. We entirely forbid $i = 3$, since this would mean that we are at the HALT instruction and that there is no successor state.

The same strategy as in this example works for arbitrary register machines $M = (I_1, \dots, I_m)$. In general R_{succ_M} is something like

$$(i = 1 \wedge Q_{I_1}) \vee (i = 2 \wedge Q_{I_2}) \vee \cdots \vee (i = m \wedge Q_{I_m})$$

where Q_I is a relation describing what should happen if the machine is at instruction I, and where we omit $(i = j \wedge Q_{I_j})$ if I_j is a halt instruction. □

Remark: Note that our proof somehow resembles the construction of the universal register machine, but it is also different in a significant way: The universal register machine had to be able to treat arbitrary machines which it got as input. In contrast, our R_{succ_M} corresponds to one specific register machine M which we know in advance. This makes the construction much easier.

Now let's try to finish the proof that computable functions are recursive.

Proof that computable functions are recursive, first part. Suppose that M is a given register machine which computes a function $f(\vec{x})$. Let me recall the entire Definition 2.2.2. More precisely, let me formulate it as a relation $R_M(\vec{x}, y)$ which holds iff $f(\vec{x}) = y$.

$R_M(\vec{x}, y) :\Longleftrightarrow$ There exist states $S_1, \dots, S_n$ such that
(i) S_1 is the start state (i.e., $S_1 = (1; x_1, \dots, x_k, 0, \dots, 0)$),
(ii) for each i, S_{i+1} is the successor state of S_i,
(iii) S_n has no successor state, and
(iv) y is the content of register 1 in state S_n

If we can show that $R_M(\vec{x}, y)$ is a recursive relation, then we can easily deduce that $f(\vec{x})$ is a recursive function. Indeed, for each $\vec{x}$, there is a unique y such that $R_M(\vec{x}, y)$ holds, so in particular that unique y is also the minimal y for which $R_M(\vec{x}, y)$ holds. Thus $f(\vec{x}) = \mu y(R_M(\vec{x}, y))$.

If we would know n (the number of steps), then it would not be very difficult to show that $R_M(\vec{x}, y)$ is recursive, using 2.8.8. But how can we speak about a whole sequence of states without knowing in advance how many states there are? Let's have a little break in the proof.
not yet □

Again, we need a way to encode sequences of numbers in a single number. In principle, this could be done exactly as we did it for register machines in Proposition 2.4.4; however, working with recursive functions is much harder than with register machines, and proving directly that $\operatorname{len}(z)$, $[z]_i$, and $\operatorname{append}(z, y)$ are recursive is quite hard. Therefore, we will use another encoding of sequences, which is not as nice as the one from Proposition 2.4.4, but which is good enough for our purposes.

Lemma 2.8.9. *There exists a recursive function $\beta\colon \mathbb{N}^3 \to \mathbb{N}$ ("**Gödel's Beta Function**") with the property that for every sequence of numbers $x_1, \dots, x_k \in \mathbb{N}$, there exist $a, b \in \mathbb{N}$ such that $\beta(a, b, i) = x_i$ for $1 \leq i \leq k$.*

The idea of how to apply this lemma is that if we want to encode a sequence $x_1, \dots, x_k$, then we choose a and b as in the lemma, and we consider this pair of numbers as a code for the sequence. Then the function β plays the role of $[z]_i$.

Here is a bunch of differences to the encoding from Proposition 2.4.4:

- First of all, a sequence is not encoded by one single number, but by two numbers (namely a and b). This sounds a bit strange, but it's not really harmful and it makes the function β much simpler.
- The lemma just tells us that there exist such a and b. However, there might be several ones and the lemma doesn't tell us which one to choose if we want to encode some given sequence.
- Finally, a and b don't know the length of the sequence, i.e., the same pair of numbers a, b might be used to encode the sequences $(5, 1, 2)$ and $(5, 1, 2, 9)$. In general, if we found some a, b such that, say $\beta(a, b, 1) = 5$, $\beta(a, b, 2) = 1$, $\beta(a, b, 3) = 2$, then we have no control what happens with $\beta(a, b, i)$ for $i \geq 4$.

To prove Lemma 2.8.9, I will need a result from number theory (which we will not prove here), namely the Chinese Remainder Theorem. Recall that we write $\operatorname{Rem}(x, y)$ for the remainder of the division of x by y.

Lemma 2.8.10 (The Chinese Remainder Theorem)**.** *If $x_1, \dots, x_k$ and $y_1, \dots, y_k$ are non-negative integers such that:*
$x_i < y_i$ for each i and
y_i and y_j are coprime for each i, j with $i \neq j$
then there exists an $a \in \mathbb{N}$ such that
$\operatorname{Rem}(a, y_1) = x_1, \operatorname{Rem}(a, y_2) = x_2, \dots, \operatorname{Rem}(a, y_k) = x_k$.

Example: Suppose we are looking for a number whose last digit is 3 and which is divisible by 7. The lemma tells us that such a number a exists: our conditions are: $\operatorname{Rem}(a, 10) = 3$ and $\operatorname{Rem}(a, 7) = 0$. (Note that 10 and 7 are coprime.) (Exercise: find such an a.)

Non-example: if, instead of requiring divisibility by 7, we require divisibility by 8, then the lemma doesn't tell us anything, since 8 and 10 are not co-prime. And indeed, no number divisible by 8 can end in 3, since the number has to be even.

Now we can prove that Gödel's Beta Function exists.

Proof of Lemma 2.8.9. The definition of β is very simple:

$$\beta(a, b, i) = \operatorname{Rem}(a, b \cdot i + 1).$$

It is clear that this function is recursive (we saw that $\operatorname{Rem}(x, y)$ is recursive in Lemma 2.8.6), so it remains to prove that this β has the property claimed in the lemma. Thus suppose that $x_1, \dots, x_k$ are given; we want to find suitable a and b. (Note that we are completely free concerning how we find a and b, i.e., there doesn't need to exist a recursive function which yields them.)

The idea is to use the Chinese Remainder Theorem. It tells us: if the numbers $y_1 = b \cdot 1 + 1, \dots, y_k = b \cdot k + 1$ are pairwise coprime and $y_i > x_i$ for all i, then we can find an a such that $\operatorname{Rem}(a, b \cdot i + 1) = x_i$ for all i.

I claim that for this to work, it is sufficient to choose $b = \ell! = 1 \cdot 2 \cdots \ell$ such that $\ell \geq k$ and $b > x_i$.

That b is bigger than x_i already ensures that y_i (which is bigger than b) is bigger than x_i, so it remains to check that the y_i are pairwise coprime.

Suppose that for some $i < j$, $y_i = b \cdot i + 1$ and $y_j = b \cdot j + 1$ have a common prime divisor p. First of all, b is divisible by all numbers $\leq \ell$, hence $b \cdot i$ is also divisible by all these numbers and hence $b \cdot i + 1$ is not divisible by any of these numbers. Thus $p > \ell$. Now consider the difference

$$y_j - y_i = (b \cdot j + 1) - (b \cdot i + 1) = b \cdot (j - i).$$

If both, y_j and y_i are divisible by p, then so is this difference. However, all prime factors of $b \cdot (j - i)$ are less or equal to ℓ (by definition of b and since $j - i \leq k$). □

Finally, we can finish the proof that all computable functions are recursive.

Proof that computable functions are recursive, second part. Recall that we have a register machine M (which computes a function, i.e., which halts on every input) and that we want to prove that the relation $R_M(\vec{x}, y)$ is recursive, which was defined (in the first part of the proof) using a sequence of states

$$\begin{aligned} S_1 &= (i_1; r_{1,1}, \dots, r_{\ell,1}) \\ S_2 &= (i_2; r_{1,2}, \dots, r_{\ell,2}) \\ &\vdots \\ S_n &= (i_n; r_{1,n}, \dots, r_{\ell,n}) \end{aligned}$$

The idea is to use one pair of numbers to encode each "vertical sequence of numbers" in this sequence of states:

$$\begin{aligned} &a_0, b_0 \text{ encodes } i_1, \dots, i_n \\ &a_1, b_1 \text{ encodes } r_{1,1}, \dots, r_{1,n} \\ &\vdots \\ &a_\ell, b_\ell \text{ encodes } r_{\ell,1}, \dots, r_{\ell,n} \end{aligned}$$

("Encode" means e.g. that $\beta(a_0, b_0, 1) = i_1, \dots, \beta(a_0, b_0, n) = i_n$.)

Using this we can write down $R_M(\vec{x}, y)$:

(1) $\exists n, a_0, b_0, \dots, a_\ell, b_\ell\colon$

S_1 is start state, i.e., $S_1 = (1; x_1, \dots, x_k, 0, \dots, 0)$:

(2) $\beta(a_0, b_0, 1) = 1 \wedge \beta(a_1, b_1, 1) = x_1 \wedge \dots \wedge \beta(a_\ell, b_\ell, 1) = 0$

$\wedge$

S_{t+1} is successor of S_t for $t = 1 \dots, n-1$:

(3) $\forall t \leq n - 1 : (t > 0 \to R_{\mathrm{succ}_M}(\beta(a_0, b_0, t), \dots, \beta(a_\ell, b_\ell, t),$
$\beta(a_0, b_0, t+1), \dots, \beta(a_\ell, b_\ell, t+1)))$

$\wedge$

S_n has no successor:

(4) $(\beta(a_0, b_0, n) = \underset{\uparrow}{h_1} \vee \beta(a_0, b_0, n) = \underset{\uparrow}{h_2} \vee \dots)$

Numbers of the HALT instructions in M

$\wedge$

y is the output value:

(5) $y = \beta(a_1, b_1, n)$

All of this is recursive (including the "$\forall t \leq n - 1$"), except for the "$\exists n, a_0, b_0, \dots, a_\ell, b_\ell\colon$" at the beginning. This would be recursive if we could specify a bound s on $n, a_0, \dots, b_\ell$ (since writing $\exists n \leq s$ is allowed), but how can we find such an s? It has to be a recursive function of $\vec{x}$ and y, and it should be bigger than the number n of steps the machine M will run and also bigger than all the code numbers a_j, b_j we will need to encode the sequence of states.

Let me write $Q(\vec{x}, n, a_0, \dots, b_\ell)$ for lines (2), (3) and (4). We know that Q is recursive, and the crucial point is that we also know the following:

(*) For every $\vec{x}$, there exist $n, a_0, \dots, b_\ell$ such that Q holds.

Indeed, we assumed that our machine M stops for every input $\vec{x}$. If it stops after n steps, then Q will hold for that n, and for some $a_0, \dots, b_\ell$ which encode the states up to that n.

Now (*) implies that for every $\vec{x}$ there exists an s such that

$Q'(\vec{x}, s) :\Longleftrightarrow \exists n \le s, \exists a_0 \le s, \ldots, \exists b_\ell \le s : Q(\vec{x}, a_0, \ldots, b_\ell)$

holds. This, together with the fact that Q' is recursive, implies that we can apply μ-recursion and obtain a recursive function

$S(\vec{x}) = \mu s(Q'(\vec{x}, s))$.

What's the meaning of this S? Well, $S(\vec{x})$ is a number which is bigger than n, where n is the number of steps that M runs on input $\vec{x}$, and moreover, there do exist codes $a_0, \ldots, b_\ell$ all less than $S(\vec{x})$ which encode the states of this run of M. In other words, $S(\vec{x})$ is exactly the function we needed above. Thus we can replace line (1) by

$\exists n < S(\vec{x}), a_0 \le S(\vec{x}), b_0 \le S(\vec{x}), \ldots, a_\ell \le S(\vec{x}), b_\ell \le S(\vec{x}):$ □

3. Gödel's First Incompleteness Theorem

3.1. **Recall Predicate Logic.** The purpose of this section is only to recall predicate logic and to fix notations and conventions; the definitions are not complete and precise.

In this lecture, the (essentially) only language we will use is the **language of Peano Arithmetic L_{PA}**.

Recall that a "**first order formula** in the language of Peano Arithmetic" – or "**L_{PA}-formula**", for short – is a "syntactically correct expression" built out of the following symbols:

(1) $\wedge$, $\vee$, $\to$, $\neg$,$\forall$, $\exists$, (,), $=$
(2) variables (x, y, z, $\dots$)
(3) 0, S, $+$, $\cdot$, $<$

Also recall that even though most of the symbols look very suggestive, if we just have an L_{PA}-formula, the meaning of 0, S, $+$, $\cdot$, $<$ is not yet fixed. Fixing the meaning of these symbols is done by choosing an **L_{PA}-structure**.

The L_{PA}-structure we will mostly be interested in is the natural numbers $\mathbb{N}$. There, 0, $+$, $\cdot$ and $<$ have their usual meaning, and $S\colon \mathbb{N} \to \mathbb{N}$ is the successor function, i.e., $S(n) = n + 1$.

However, it will be important that other L_{PA}-structures also exist. In general, an L_{PA}-structure is a set A together with **interpretations** of 0, S, $+$, $\cdot$, $<$: 0 is an element of A, S is a function $A \to A$, $+$ and $\cdot$ are functions $A^2 \to A$ and $<$ is a binary relation on A. There are no condition whatsoever on these interpretations to be as one would expect from what the symbols look like.

Example 3.1.1. *One can consider the L_{PA}-structure A which is exactly the same as $\mathbb{N}$, except that $+$ is interpreted as multiplication, and "$x < y$" is interpreted as: x and y are co-prime.*

Note however that the meaning of the symbols in (1) above is fixed (for example, "$\forall$" always means "for all", no matter what A is).

Convention 3.1.2. *In this lecture, we fix that "$=$" always means equality, no matter what the structure A is.*

(Some authors also allow "$=$" to be interpreted differently in different structures; I think this was the point of view taken in Logic I.)

Convention 3.1.3. *From now on, "formula" and "structure" will always mean "L_{PA}-formula" and "L_{PA}-structure".*

If ϕ is a formula with free variables $x_1, \dots, x_k$, we often write "$\phi(x_1, \dots, x_k)$" to emphasize what the free variables are. Recall that a **sentence** is a formula without free variables.

Once a structure A is fixed, we can also **interpret** formulas in A:

- If ϕ is a sentence, we write "$A \models \phi$" to say that ϕ holds in A. (We also say that A is a **model** of ϕ or that ϕ is **true** in A.)
- If $\phi(x_1, \dots, x_k)$ is a formula (with k free variables), and $a_1, \dots, a_k \in A$, we write "$A \models \phi(a_1, \dots, a_k)$" to say that ϕ holds in A when one substitutes $a_1, \dots, a_k$ for the free variables $x_1, \dots x_k$.
  (Note: This notation differs a bit from the one in Logic I.)

Since we will mainly be interested in the structure $\mathbb{N}$, we define:

Definition 3.1.4. *We say that a sentence ϕ is **true** if it is true in $\mathbb{N}$, i.e., if $\mathbb{N} \models \phi$.*

Here are some examples of formulas:

(1) $\phi_1 = \forall x\,(x + 0 = 0)$
This is a sentence which is not true, but there are structures in which is it true, e.g.

the one from Example 3.1.1, since $+$ is interpreted as multiplication, and indeed, $x \cdot 0 = 0$ for all x.

(2) $\phi_2 = \forall x \forall y (x = y \lor x < y \lor y < x)$
This is a true sentence, but there are other structures in which it is not true, e.g. the one from Example 3.1.1: Take $x = 2$ and $y = 4$: "$2 = 4 \lor 2 < 4 \lor 2 < 4$" is false in that structure, since "$2 < 4$" stands for "2 and 4 are co-prime", which is not the case.

(3) $\phi_3 = (0 = 0)$
This sentence is true in any structure.

(4) $\phi_4(x, y) = \exists z\, (x + z = y)$
We have, for example, $\mathbb{N} \models \phi_4(2, 3)$ but not $\mathbb{N} \models \phi_4(7, 4)$. In general, for $a, b \in \mathbb{N}$, we have $\mathbb{N} \models \phi_4(a, b)$ iff $a \leq b$.
(Recall: When interpreting a formula in a structure A, the quantifiers run over A; so in this example, since we are interpreting the formula in $\mathbb{N}$, z has to be a natural number.)

We will use various shortcut notations in formulas, for example:

- "$x \neq y$" means "$\neg x = y$"
- "$x \leq y$" means "$x < y \lor x = y$"
- "$\forall x, y$" means "$\forall x\, \forall y$"
- "4" means "$S(S(S(S(0))))$"

Essentially, any shortcut is allowed as long as it is clear how it can be turned into a precise formula.

Exercise 3.1.5. We will write "$\exists^{=1} x : \phi(x)$" to say that there exists exactly on x for which $\phi(x)$ holds. How can this be expressed as a (first order) formula?

Once we fix a structure A, a formula with k free variables yields a k-ary relation on A. For example, the above $\phi_4(x, y)$ yields the relation $x \leq y$ on $\mathbb{N}$.

We'd like to also be able to describe functions $f(\vec{x})$ by formulas. Since formulas can only be true or false (and don't yield values), we cannot directly use a formula $\phi(\vec{x})$ to describe f. Instead, we use a formula $\phi(\vec{x}, y)$ which holds iff $y = f(\vec{x})$.

Here is a more formal definition.

Definition 3.1.6. *Fix a structure A.*

*We say that a formula $\phi(x_1, \dots, x_k)$ **defines** a k-ary relation R on A if:*
For all $\vec{a} \in A^k$, we have $A \models \phi(\vec{a}) \iff R(\vec{a})$.

*We say that a formula $\phi(x_1, \dots, x_k, y)$ **defines** a function $f\colon A^k \to A$ if:*
For all $\vec{a} \in A^k$ and all $b \in A$, we have $A \models \phi(\vec{a}, b) \iff f(\vec{a}) = b$.

*We call a relation on $\mathbb{N}$ or a function $\mathbb{N}^k \to \mathbb{N}$ **arithemtical** if there exists a formula defining it.*

Remark: In the same way as for computable functions, it is easy to see that not all functions can be arithmetical, since there are only countably many arithmetical ones. (And similarly for relations.)

Later, we will prove that every computable function is arithmetical (by passing through recursive functions). However, this time, the converse is not true: There are arithmetical functions that are not computable. (Details later.)

Examples (in $\mathbb{N}$):

- The function $(x, y) \mapsto x + y$ is arithmetical; it is defined by the formula $\phi(x, y, z) = (x + y = z)$.
- The function
$$f(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$$
is aritmetical, e.g. by the formula $\phi(x, y, z) = (x \geq y \land z + y = x) \lor (x < y \land z = 0)$.

Note: Every formula with k free variables defines a k-ary relation, but not every formula with $k+1$ free variables defines a function $\mathbb{N}^k \to \mathbb{N}$. A formula $\phi(\vec{x}, y)$ defines a function iff for every $\vec{x}$, there exists exactly one y such that $\phi(\vec{x}, y)$ holds.

Exercise 3.1.7. (1) Does the formula $\phi(x, y) = (x = 2 \cdot y)$ define a function? (If yes, which one?)
(2) What about $\phi'(x, y) = (x = 2 \cdot y \vee x = 2 \cdot y + 1)$?

Once we proved that a function or relation is arithmetical, we can use them as shortcuts in formulas to prove that other things are arithmetical. Here are some examples:

Examples:

- The formula $\phi(x) = \exists y\colon 2y = x$ defines the relation "x is even".
  Now that we know this, the following is a valid shortcut:
  $\psi = \forall x : (x \text{ is even} \quad \vee \quad x + 1 \text{ is even})$
  Indeed, we can to replace "x is even" by $\phi(x)$ to obtain we obtain:
  $\psi = \forall x : (\exists y\, (2y = x) \quad \vee \quad \exists y\, (2y = x + 1))$
- For functions, this is a bit less obvious. If $\phi(x, y, z)$ is a formula defining a function $f(x, y)$, we might e.g. want to consider the formula
  $\psi(x) = \forall y : (y + f(x, S(y)) > 0)$
  To get rid of the f (and use ϕ instead), we need to introduce a new variable z for the value of $f(x, y)$:
  $\psi(x) = \forall y : (\exists z : \phi(x, S(y), z) \wedge y + z > 0)$

Exercise 3.1.8. (1) Find an L_{PA}-formula defining the relation $P(x) :\Longleftrightarrow x$ is prime.
(2) Formulate the Goldbach Conjecture as a first order sentence, namely that every even number ≥ 4 can be written as a sum of two primes. When doing this, use the relations $P(x)$ from (1) and "x is even" as shortcuts.

3.2. **Gödel's Completeness Theorem.** Recall from Logic I:

Definition 3.2.1. *Let T be a set of sentences and let ψ be a sentence.*

A structure A is called a **model** *of T if $A \models \phi$ for all $\phi \in T$. We write this as "$A \models T$".*

We say that T **logically implies** *ψ if ψ holds in every model $A \models T$. We write this as "$T \models \psi$".*

We say that ψ **is provable** *from T if ψ can be deduced from T using the rules of natural deduction. We write this as "$T \vdash \psi$".*

Example: $T = \{\forall x : x + 0 = x\}$ logically implies $\phi = (S(0) + 0 = S(0))$.

Exercise 3.2.2. (1) Does the above T also logically imply $\phi_2 = (0 + S(0) = S(0))$?
(2) Can the empty set logically imply something? Give an example.

Theorem 3.2.3 (Gödel's Completeness Theorem)**.** *For every set of sentences T and every sentence ϕ, one has:*
$T \models \phi$ *iff* $T \vdash \phi$.

Thus from now on, I will not distinguish between $T \models \phi$ and $T \vdash \phi$. I will also use various formulations synonymously for this: "T (logically) implies ψ", "ϕ can be deduced from T", "ψ is provable from T" (or similar formulations).

Here are some more properties of theories:

Definition 3.2.4. *We say that a set of sentences T is* **consistent** *if it has a model, i.e., if there exists A such that $A \models T$. We say that T is* **complete** *if it is consistent and moreover, for every sentence ϕ, either $T \vdash \phi$ or $T \vdash \neg\phi$.*

Note: Here, the word "complete" is used in a somewhat different sense than in "Gödel's Completeness Theorem". In both cases, complete means: Something that should be provable can be proven. However, in Gödel's Completeness Theorem, "ϕ should be provable from T" means that ϕ follows logically from T, whereas now, we want T itself to be "big enough that anything can be proved or disproved from T".

Examples:

- Consider $T_1 = \{\forall x, y : x + y = y + x\}$.
  $\mathbb{N}$ is a model of T_1 (since this sentence does hold in $\mathbb{N}$).
  T_1 is consistent (since a model exists).
  T_1 is not complete, since we can (e.g.) neither deduce $0 < 0$ or $\neg 0 < 0$.
- $T_2 = \{S(0) = 0\}$: We don't have $\mathbb{N} \models T_2$, but if A is a structure where S is interpreted e.g. as the identity function, then A is a model of T_2. Thus T_2 is also consistent.
- $T_3 = \{S(0) = 0, \neg S(0) = 0\}$ is not consistent: For any structure A, if $A \models S(0) = 0$, then $A \neg \models \neg S(0) = 0$.
  More generally, for any sentence ψ, if $\psi \in T$ and $\neg\psi \in T$, then T is inconsistent.
- The set of all true sentences (i.e., sentences that hold in $\mathbb{N}$) is complete.
  (Indeed: If ϕ is not true in $\mathbb{N}$, then by definition of the interpretation of the negation of a formula, $\neg\phi$ is true in $\mathbb{N}$.)

Note: Gödel's Incompleteness Theorem will tell us that we (or, more precisely, a register machine) cannot find out which sentences exactly hold in $\mathbb{N}$ and which don't. However, this does *not* mean that there exist sentences ϕ that are neither true nor false in $\mathbb{N}$.

3.3. **Computable functions are arithmetical.**

Proposition 3.3.1. *Every computable function is arithmetical. Every decidable relation is arithmetical.*

We have already seen that computable functions and decidable relations are recursive, so it now suffices to prove that recursive implies arithmetical. This is easy, because the expression describing a recursive function already looks almost like a formula. I will skip the details of the proof.

The converse of the proposition is not true: The halting problem $R_{\mathrm{HALT}}(x)$ is not decidable, but it is arithemetical. More generally:

Corollary 3.3.2. *Every semi-decidable relation is arithmetical.*

Proof. Recall: If $R(\vec{x})$ is semi-decidable, then there exists a decidable $Q(\vec{x}, y)$ such that $R(\vec{x})$ holds iff there exists a y such that $Q(\vec{x}, y)$ holds. (This was Proposition 2.6.4.)

Since Q is arithmetical, it is defined by a formula $\phi_Q(\vec{x}, y)$. This implies that R is defined by the formula $\exists y : \phi(\vec{x}, y)$. □

Exercise 3.3.3. That the halting problem is arithmetical means that there exists a formula $\phi(x)$ such that for all $a \in \mathbb{N}$, we have $\mathbb{N} \models \phi_{\mathrm{HALT}}(a)$ iff $R_{\mathrm{HALT}}(a)$ holds. How could one find such a formula, using what we have seen in this lecture?

3.4. **Encoding formulas.** We will need a way to feed formulas into register machines, so we have to encode them by numbers. There are various ways to do this – some more handy than others if one wants to actually write down the register machines operating on them. Let's not bother too much about technical details and simply encode them as follows.

Definition 3.4.1. *To each symbol of the language, we associate a number, for example:*

- $\ulcorner\wedge\urcorner = 0$, $\ulcorner\vee\urcorner = 1$, $\ulcorner\rightarrow\urcorner = 2$, $\ulcorner\neg\urcorner = 3$, $\ulcorner\forall\urcorner = 4$, $\ulcorner\exists\urcorner = 5$, $\ulcorner(\urcorner = 6$, $\ulcorner)\urcorner = 7$, $\ulcorner=\urcorner = 8$;
- $\ulcorner 0\urcorner = 9$, $\ulcorner S\urcorner = 10$, $\ulcorner+\urcorner = 11$, $\ulcorner\cdot\urcorner = 12$, $\ulcorner<\urcorner = 13$;
- *each number* ≥ 14 *stands for a different variable symbol. (When x is a variable, we will write $\ulcorner x\urcorner$ without thinking about which number is actually used to encode x.)*

If ϕ is a formula or a term consisting of symbols $s_1 s_2 \dots s_\ell$, we define $\ulcorner\phi\urcorner := \langle\ulcorner s_1\urcorner, \dots, \ulcorner s_\ell\urcorner\rangle$.

Here, we assume that in $s_1 s_2 \dots s_\ell$, no shortcut is used; for example, we might write $\ulcorner x \geq 0\urcorner$, but by this, we mean $\langle\ulcorner 0\urcorner, \ulcorner<\urcorner, \ulcorner x\urcorner, \ulcorner\vee\urcorner, \ulcorner x\urcorner, \ulcorner=\urcorner, \ulcorner 0\urcorner\rangle$.

It is tedious but not particularly difficult to check that register machines can do all sorts of natural operations with these formulas and terms; for example, they can:

- check whether a given number x is the code of a syntactically correct formula;
- find out what the free variables of a formula are;
- substitute a term for a free variable;
- given a natural number n, output $\ulcorner n\urcorner$.

The last example might need some explanation: Recall that if we write n in a formula, this is a shortcut for $\underbrace{S(S(\dots S}_{n \text{ times}}(0)))$. So by "$\ulcorner n\urcorner$", what I really mean is $\ulcorner\underbrace{S(S(\dots S}_{n \text{ times}}(0)))\urcorner$. Since it might sometimes be confusing if one just writes n for the term, I will sometimes use the following notation.

Notation 3.4.2. *If n is a natural number, then $\underline{n}$ stands for the term $\underbrace{S(S(\dots S}_{n \text{ times}}(0)))$.*

3.5. **Gödel's First Incompleteness Theorem.** Using our encoding of formulas, we define the following.

Definition 3.5.1. *Given a set of sentences T, consider the relation*
$R_T(x) :\Longleftrightarrow x = \ulcorner\phi\urcorner$ *for some* $\phi \in T$.
T is called ***decidable*** *iff R_T is decidable, and T is called* ***semi-decidable*** *if R_T is semi-decidable.*

Now we can finally state Gödel's First Incompleteness Theorem precisely. Here is a first version:

Theorem 3.5.2 (Gödel's First Incompleteness Theorem, Version 1)**.** *The set of true sentences is not decidable.*

In other words, there exists no algorithm to find out whether a given sentence is true or not.

This is not exactly the formulation I promised at the beginning of the lecture. We'll see other versions afterwards.

Proof of 3.5.2. Suppose the set of true sentences is decidable. I claim that then, we can use this to decide the halting problem. (Hence this yields a contradiction.)

Recall that R_{HALT} is arithmetical, i.e., there exists a formula $\phi_{\text{HALT}}(x)$ be a formula such that $\mathbb{N} \models \phi_{\text{HALT}}(x)$ iff $R_{\text{HALT}}(x)$ holds. The following register machine decides whether $R_{\text{HALT}}(x)$ holds:

Input: n
1. Construct (a code for) the sentence $\psi = \phi_{\text{HALT}}(\underline{n})$. (Recall that $\underline{n} = \underbrace{S(S(\dots S}_{n \text{ times}}(0))))$
2. If ψ is true, then output 0; otherwise output 1 □

From Theorem 3.5.2, we can easily directly deduce a stronger statement:

Theorem 3.5.3 (Gödel's First Incompleteness Theorem, Version 2)**.** *The set of true sentences is not semi-decidable.*

Proof. Suppose
$R_T(\ulcorner\phi\urcorner) :\Longleftrightarrow \phi$ is true
would be semi-decidable.

Then $\neg R_T$ is also semi-deciable, since $\neg R_T(\ulcorner\phi\urcorner) \iff R_T(\ulcorner\neg\phi\urcorner)$.

However, if both R_T and $\neg R_T$ are semi-decidable, then R_T is decidable (by Proposition 2.6.5). □

What I promised at the beginning of the lecture was to prove:

Corollary 3.5.4. *There exists no effective deductive system D which is complete in the following sense:*

ψ *can be proven within* $D \iff \psi$ *is true*

Recall: "Effective" means that there exists an algorithm which can check whether something is a valid proof within D or not. To make this more formal, we assume that proofs within D can be encoded by numbers. Then we can consider the relation

$\text{proof}_D(\ulcorner\phi\urcorner, y) :\Longleftrightarrow y$ is the code of a proof within D of ϕ

D being effective means: proof_D is decidable.

Now a register machine could try to find out whether ϕ holds by checking whether $\text{proof}_D(\ulcorner\phi\urcorner, y)$ holds for one y after the other (in other words, by trying out all potential proofs). This shows: If D is an effective deductive system, then the set of sentences provable within D is semi-decidable. This shows that Theorem 3.5.3 indeed implies the corollary.

A side remark: Corollary 3.5.4 might sound like contradicting Gödel's Completeness Theorem, which states that natural deduction is complete. However, this is completeness in a slighly different sense:

$$\{\} \vdash \psi \quad \Longleftrightarrow \quad A \models \psi \text{ for every structure } A$$

(I.e.: only sentences that are true in *every* structure can be proved using natural deduction, whereas the incompleteness theorems are about sentences true in $\mathbb{N}$.)

One way to deal with the problem that there's no way to decide what's true and what's false is to fix a set of "axtioms" T_0, i.e., a set of true sentences, and then try to see what one can prove using those axioms.

Here's a version of Gödel's First Incompleteness Theorem concerning that approach:

Corollary 3.5.5. *There exists no decidable set of true sentences T_0 such that every true sentence ψ can be deduced from T_0 ("deduced" in the sence $T_0 \vdash \psi$, or equivalently, $T_0 \models \psi$).*

Proof. Suppose otherwise. Then T_0 together with natural deduction would yield an effective proof system D_0 contradicting Corollary 3.5.4. Indeed, we would then define a proof of ψ within D_0 to be a natural deduction proof $T_0 \vdash \psi$. Whether such a proof is valid can be checked by a register machine:

Recall that a proof using natural deduction consists of applying some rules finitely many times. This means that we may encode the entire proof by a number, e.g., as the list of intermediate formulas.

Then a register machine can check whether each formulas in that list either is a sentence from T_0 (here, we use the assumption that T_0 is decidable), or whether it can be obtained from previous formulas in the list by applying the rules of natural deduction. □

4. More Gödel Incompleteness

Recall: Before the Easter Break, we proved various versions of Gödel's First Incompleteness Theorem (GInc1). One way to state it is that the set of true sentences (i.e., sentences which hold in $\mathbb{N}$) is not decidable (Thm 3.5.2). And from this, we then deduced that the set of true sentences is not even semi-decidable (Thm 3.5.3), and this in turn implies that there is no effective deductive system which can prove exactly the true sentences Corollary 3.5.4). Or, in informal terms, there is no way to formalize a notion of proof in a way that

- exactly the true sentences can be proven and
- correctness of proofs can be checked algorithmically.

As I also mentioned, to be able to sensibly do mathematics nevertheless, one can choose a small set of sentences ("axioms") which may not be strong enough to imply all true sentences, but which are hopefully good enough to imply a lot of what one is really interested in. One quite successful set of axioms is PA (Peano Arithmetic).

When using PA, a sentence can fall into one of three categories: either it is provable, or it is disprovable (its negation is provable) or neither nor. The set of provable sentences is semi-decidable, and similarly the set of disprovable sentences, but it would be nice if those sets would be decidable, i.e., if there would exist an algorithm deciding into which of the three categories a given sentence falls. Our first goal in Advanced Proof and Computation is to prove a stronger version of GInc1 which implies that this is not possible.

The other thing we will see in the lecture is Gödel's second incompleteness theorem (GInc2), which roughly states that using the axioms of PA, one cannot prove that PA is consistent. A bit more formally:

PA $\not\models$ "PA is consistent".

To make sense of this, one of course has to formulate "PA is consistent" as an L_{PA}-sentence. It is not too difficult to do this in some intuitive way. However, it will turn out that it is not so clear whether different intuitive ways to formulate "PA is consistent" as an L_{PA}-sentence are really equivalent. Thus even formulating GInc2 precisely will give us some headache.

For our two goals, we will have to carry out certain proofs inside PA (i.e., using only the axioms of PA). In principle, we could just do the entire lecture again, but doing all the proofs only using the axioms from PA. However, a lot of that work can be avoided using the *Representability Theorem*, which states that a whole bunch of sentences can be proved in PA (so one does not need to check each of these individually).

A second key tool (for both of our goals) will be the *Fixed Point Theorem.* In some intuitive sense, the theorem allows a sentence to speak about itself. In particular, it will allow us to turn sentences like

"This sentence cannot be proven in PA."

into a precise mathematical statement.

4.1. Peano Arithmetic.

As I mentioned before, Peano Arithmetic is a set of axioms which works quite well for mathematics in $\mathbb{N}$. I also introduce a weaker variant of it, called Robinson Arithmetic (which will also be useful). (Note: the language of Peano Arithmetic is called that way because it is the language in which those axioms are formulated.)

Definition 4.1.1. ***Robinson Arithmetic*** *(ROB) is the following set of axioms:*

(S1) $\forall x : S(x) \neq 0$
(S2) $\forall x, y : (S(x) = S(y) \to x = y)$
(A1) $\forall x : x + 0 = x$
(A2) $\forall x, y : x + S(y) = S(x + y)$
(M1) $\forall x : x \cdot 0 = 0$
(M2) $\forall x, y : x \cdot S(y) = (x \cdot y) + x$
(L1) $\forall x : \neg(x < 0)$

(L2) $\forall x, y : (x < S(y) \leftrightarrow (x < y \vee x = y))$
(L3) $\forall x, y : (x < y \vee x = y \vee y < x)$

Peano Arithmetic *(PA) consists of the same axioms, and additionally one axiom for every formula $\phi(x)$:*

(IND) $\phi(0) \wedge \ \forall x : (\phi(x) \to \phi(S(x))) \quad \to \quad \forall x : \phi(x)$

((IND) is called an "axiom scheme".)

(Note: ROB consists only of finitely many axioms, whereas PA consists of infinitely many axioms.)

One way to understand what those axioms imply (and what they don't imply) is to consider an arbitrary model A of ROB or PA and see what we can say about it.

Lemma 4.1.2. *Suppose $A \models \mathrm{ROB}$; in the following, I write $+^A$ for the interpretation of $+$ in A, to distinguish them from addition in $\mathbb{N}$, and similarly for the other symbols of L_{PA}. Consider the map $\alpha\colon \mathbb{N} \to A$ sending $n \in \mathbb{N}$ to $\underbrace{S^A(S^A(\dots S^A}_{n \text{ times}}(0^A)))$.*

(1) *This map α is injective.*
(2) *It is compatible with addition, multiplication and the order, i.e., $\alpha(m+n) = \alpha(m) +^A \alpha(n)$, $\alpha(m \cdot n) = \alpha(m) \cdot^A \alpha(n)$, and $m < n \iff \alpha(m) <^A \alpha(n)$.*
(3) *For any $n \in \mathbb{N}$ and any $a \in A \setminus \alpha(\mathbb{N})$, we have $n < a$.*

In other words, (1) and (2) say that we can (and will) identify $\mathbb{N}$ with its image in A under α (for any $A \models \mathrm{ROB}$), and (3) says that A can differ from $\mathbb{N}$ only by having "infinite" elements, i.e., elements bigger than any $n \in \mathbb{N}$.

The proof of that lemma will also make the purpose of the axioms of ROB clear. Intuitively, one can think of (S1) and (S2) as defining the successor function, (A1) and (A2) as (recursively) defining addition, (M1) and (M2) as defining multiplication and (L1) and (L2) as defining "<".

Proof of 4.1.2. (1)

(S1) says that 0^A is not the successor of anything, so this already proves $\alpha(0) \neq \alpha(n)$ for all $n \geq 1$.

To prove $\alpha(m) \neq \alpha(n)$ for general $m < n$, we use (S2) and induction over m (the case $m = 0$ having already been done): We have $\alpha(m) = S^A(\alpha(m-1))$ and $\alpha(n) = S^A(\alpha(n-1))$. By the induction hypothesis, $\alpha(m-1) \neq \alpha(n-1)$. Now (S2) states that S^A is injective, so we get $\alpha(m) \neq \alpha(n)$, as desired.

(2)

Addition: We prove $\alpha(m+n) = \alpha(m) +^A \alpha(n)$ by induction over n. The case $n = 0$ ($\alpha(m) = \alpha(m) +^A 0^A$) follows from (A1). For $n \geq 1$, we have

$$\alpha(m+n) = S^A(\alpha(m+n-1)) \overset{i.h.}{=} S^A(\alpha(m) +^A \alpha(n-1)) \overset{(A2)}{=} \alpha(m) +^A S^A(\alpha(n-1))$$
$$= \alpha(m) +^A \alpha(n).$$

Multiplication: $\alpha(m \cdot n) = \alpha(m) \cdot^A \alpha(n)$ follows from (M1) in the case $n = 0$, and again, the case $n \geq 1$ is done by induction using (M2):

$$\alpha(m \cdot n) = \alpha(m \cdot (n-1) + m) = \alpha(m \cdot (n-1)) + \alpha(m) \overset{i.h.}{=} \alpha(m) \cdot \alpha(n-1) + \alpha(m)$$
$$\overset{(M2)}{=} \alpha(m) \cdot^A S^A(\alpha(n-1)) = \alpha(m) \cdot^A \alpha(n).$$

Less than: We want to prove that for all m, n we have $m < n \iff \alpha(m) <^A \alpha(n)$. We do an induction over n. The case $n = 0$ follows from (L1), so now assume $n \geq 1$. Then

$$\alpha(m) <^A \alpha(n) \overset{(L2)}{\iff} \underbrace{\alpha(m) <^A \alpha(n-1)}_{\iff m < n-1} \vee \underbrace{\alpha(m) = \alpha(n-1)}_{\iff m = n-1}$$
$$\iff m \leq n-1 \iff m < n.$$

(From now on, I will identify $\mathbb{N}$ with its image in A.)

(3)

To obtain $n < a$, by (L3), it suffices to prove $\neg a < n$ and $\neg a = n$. The latter is clear, and the proof of $\neg a < n$ is almost the same as the one of compatibility of α with $<$:
We prove it by induction over n. By (L1), we have $\neg a < 0$, and for $n \geq 1$, we have $\neg a < (n-1)$ (by the induction hypothesis) and $\neg a = n - 1$, which, by (L2), together imply $\neg a < n$. $\square$

Now let us try to prove something more general: Does ROB imply $(*)$ $\forall x : 0 + x = x$?

Using (A1) and (A2), we can prove:
$0 + 0 = 0$
$0 + 1 = 1$
$0 + 2 = 2$
$\vdots$

We could try to prove $(*)$ by induction over x. However, this only yields that $0+x = x$ holds for $x \in \mathbb{N}$. However, for ROB to imply $(*)$, we would need it to hold for all $x \in A$ (for all $A \models$ ROB). It turns out that $(*)$ does *not* follow from ROB. This is where PA and (IND) comes in: (IND) postulates that inductive arguments are valid within any model of PA. More formally: if we want to prove that some formula $\phi(x)$ holds for all $x \in A$, then it is enough to prove that ϕ holds for 0 and that if it holds for x, then also for $x + 1$.

Lemma 4.1.3. *PA implies: $(*)$ $\forall x : 0 + x = x$*

Proof. We use (IND) with the formula $\phi(x) = \quad 0 + x = x$:

$$\underbrace{0 + 0 = 0}_{\text{(a)}} \wedge \underbrace{\forall x : (0 + x = x \rightarrow 0 + S(x) = S(x))}_{\text{(b)}} \quad \rightarrow \quad \forall x : 0 + x = x$$

Hence to obtain $(*)$, we have to prove (a) and (b). (a) follows from (A1), and (b) follows from (A2), since (A2) implies $\forall x : 0 + S(x) = S(0 + x)$. $\square$

Exercise 4.1.4. In a similar way, prove that PA implies
$(**)$ $\forall x, y, z : (x + y) + z = x + (y + z)$.

Hint: This works best to do "induction over z".

In a similar way, one can prove all the basic properties of addition, multiplication and the $<$ relation. After that, it becomes easier to work with PA, and after some time, it becomes clear that indeed almost everything one would want to do can be done in PA. In fact, almost all proofs we have seen in this lecture could have been done in PA.

As an example, we could have proven using only PA that the halting problem is not decidable. For this to make sense, we have to formulate this as an L_{PA}-sentence. This would involve even more encoding of register machines by numbers: We want to express: "There exists no register machine deciding $R_{\mathrm{HALT}}(x)$". Since L_{PA}-sentences only speak about numbers and not register machines, we formulate this as:

$\neg\exists y$: "y is the code of a register machine deciding $R_{\mathrm{HALT}}(x)$"

Etc.

4.2. **Representability.** Recall that a function $f\colon \mathbb{N}^k \to \mathbb{N}$ is arithmetical if it can be defined by a formula ϕ_f, i.e., $\phi_f(\vec{x}, y)$ holds in $\mathbb{N}$ iff $f(\vec{x}) = y$. Now we'd like to have something stronger, namely we want to be able to *prove this in ROB* (or in PA). This means that it should not only be true in $\mathbb{N}$ that $\phi_f(x, y)$ holds iff $f(x) = y$, but also in any other model of ROB. Let's make this precise.

In this entire subsection, we assume that T is a set of true sentences containing ROB (e.g. $T =$ ROB itself or $T =$ PA or $T =$ all true sentences). In particular, any model of T contains $\mathbb{N}$. We will mainly need the case $T =$ ROB.

Definition 4.2.1. *Suppose that $f\colon \mathbb{N}^k \to \mathbb{N}$ is a function. A formula $\phi(\vec{x}, y)$* ***represents*** *f in T if for all $\vec{a} \in \mathbb{N}^k$ and for $b = f(\vec{a})$, we have:*

$$T \vdash \forall y : (\phi(\vec{a}, y) \leftrightarrow y = b).$$

Suppose that $R(\vec{x})$ is a k-ary relation. A formula $\phi(\vec{x})$ ***represents*** *$R(\vec{x})$ in T if for all $\vec{a} \in \mathbb{N}^k$, we have:*

If $R(\vec{a})$ holds, then $T \vdash \phi(\vec{a})$;
If $R(\vec{a})$ does not hold, then $T \vdash \neg\phi(\vec{a})$

A function or a relation is called ***representable*** *in T if there exists a formula representing it.*

The definition might look a bit strange; why not, for relations for example, ask that $T \vdash \forall\vec{x} : (R(\vec{x}) \leftrightarrow \phi(\vec{x}))$? The reason is that "$R(\vec{x})$" is not a formula, so this does not make sense.

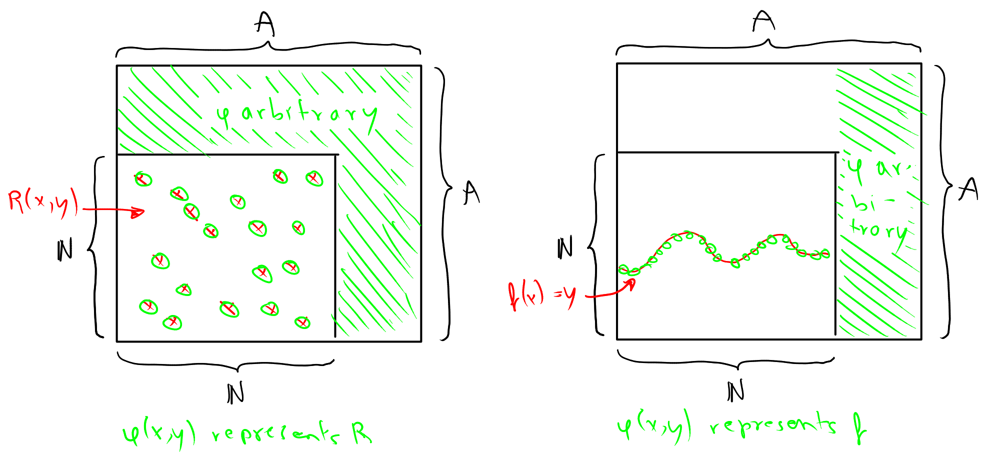
To get a better intuition of the meaning of representability, let me reformulate this in terms of models.

Lemma 4.2.2. *Let T be as above.*

A formula $\phi(\vec{x}, y)$ represents $f\colon \mathbb{N}^k \to \mathbb{N}$ in T if for every model $A \models T$, we have:

$$\text{For every } \vec{a} \in \mathbb{N}^k \text{ and } \underbrace{\text{every } b \in A\text{: } A \models \phi(\vec{a}, b) \text{ iff } b = f(\vec{a})}_{\text{This says: } A \models \forall y : (\phi(\vec{a}, y) \leftrightarrow y = f(\vec{a}))}.$$

A formula $\phi(\vec{x})$ represents a relation $R(\vec{x})$ if for all models $A \models T$, we have:
For every $\vec{a} \in \mathbb{N}^k$, $A \models \phi(\vec{a})$ iff $R(\vec{a})$ holds.



In Lemma 4.2.2, it is a bit odd that in the representability of a function, the quantifier for b runs over A and not just over $\mathbb{N}$. (If it would run over $\mathbb{N}$, then a formula $\phi(\vec{x}, y)$ would represent a function $f\colon \mathbb{N}^k \to \mathbb{N}$ iff it represents the relation on $\mathbb{N}^{k+1}$ defined by $f(\vec{x}) = y$.) The reason to define it like this is that this is what we will need in applications. However, in reality, it doesn't make a difference: If f is a function and $\phi(\vec{x}, y)$ is a formula representing the relation $f(\vec{x}) = y$, then we easily obtain a formula $\phi'(\vec{x}, y)$ representing f: For any $\vec{x}$, we define $\phi'(\vec{x}, y)$ to hold iff y is the smallest value for which $\phi(\vec{x}, y)$ holds.

Examples:

- The relation $R(x,y) :\iff x < y$ is represented in ROB by the formula $x < y$. Indeed, one only has to check that for any model $A \models$ ROB and any $a, b \in \mathbb{N}$, we have $a < b$ (as natural numbers) iff $a <^A b$. This is indeed the case (by Lemma 4.1.2).
- The function $f(x) = 3x^2 + 5$ is represented (in ROB) the formula $\phi(x,y) = \ulcorner 3x^2 + 5 = y \urcorner$; this follows from the fact that the embedding $\mathbb{N} \subseteq A$ respects addition and multiplication.
- Let $R(x)$ be the relation which holds iff x is a square. In that case, it is not clear whether the formula $\phi(x) = \exists y : y^2 = x$ represent R in ROB. The problem is that for some non-square $a \in \mathbb{N}$, there might exists an element $b \in A \setminus \mathbb{N}$ such that $b^2 = a$; in that case, we would have $A \models \phi(a)$, but not $R(a)$.

Exercise 4.2.3.
- In the last of the above examples, how could one modify $\phi(x)$ such that it represents $R(x)$ in ROB?
- Also in that example: how could one prove that $\phi(x)$ represents $R(x)$ in PA? (Only give the idea of the proof.)
- Let $R(x)$ be the relation which is always true. Does there exist a formula $\phi(x)$ which defines the relation, but which does not represent it in PA? (Hint: Recall that PA is incomplete.)
- Does there exist a formula which defines the identity function $f(x) = x$, but which does not represent it in PA?

Now we are ready to prove the first ingredient to our main goals.

Theorem 4.2.4 (Representability Theorem)**.** *Every computable function is representable in* ROB.

Proof. As in the proof that every computable function is arithmetical (Proposition 3.3.1), we use that every computable function is recursive, and we go through the various ingredients of recursive functions. (To check that a formula represents a function, we will use Lemma 4.2.2.)

- The constant function $f(\vec{x}) = n$ is represented by the formula $\phi(\vec{x}, y) = \ulcorner y = \underline{n} \urcorner$. (Indeed, that formula defines the constant-n-function in *every* model or ROB.)
- The function $f(\vec{x}) = x_i$ is represented by the formula $\phi(\vec{x}, y) = \ulcorner y = x_i \urcorner$.
- Addition: Suppose that $g_1(\vec{x})$, $g_2(\vec{x})$ are represented by formulas $\phi_1(\vec{x}, y)$ and $\phi_2(\vec{x}, y)$. I claim that the following formula represents $f(\vec{x}) := g_1(\vec{x}) + g_2(\vec{x})$: $\phi(\vec{x}, y) = \exists y_1, y_2 : \phi_1(\vec{x}, y_1) \land \phi_2(\vec{x}, y_2) \land y_1 + y_2 = y$.

  Indeed, fix any model $A \models$ ROB and any tuple $\vec{a} \in \mathbb{N}^k$. Then the only $y_1 \in A$ for which $A \models \phi_1(\vec{a}, y_1)$ holds is $g_1(\vec{a})$ (since ϕ_1 represents g_1), and similarly for y_2. This implies $y = g_1(\vec{a}) + g_2(\vec{a})$.
- Multiplication: works in exactly the same way as addition.
- $K_<(g_1(\vec{x}), g_2(\vec{x}))$: Use the formula

  $\phi(\vec{x}, y) = \ulcorner \exists y_1, y_2 : \phi_1(\vec{x}, y_1) \land \phi_2(\vec{x}, y_2) \land (y_1 < y_2 \to y = 0) \land (\neg y_1 < y_2 \to y = 1) \urcorner$

  Again, in any model A, y_1 and y_2 can only be the "right values", and the end of the formula ensures that y is the right value.
- Suppose that $g\colon \mathbb{N}^{k+1} \to \mathbb{N}$ is represented by $\phi(\vec{x}, y, z)$ and that for every $\vec{x}$, there exists a y such that $g(\vec{x}, y) = 0$. I claim that $f(\vec{x}) = \mu y(g(\vec{x}, y) = 0)$ is represented by the formula

  $\psi(\vec{x}, y) = \ulcorner \phi(\vec{x}, y, 0) \land \forall y' : (y' < y \to \neg\phi(\vec{x}, y', 0)) \urcorner$.

  Fix $\vec{a} \in \mathbb{N}^k$ and fix a model $A \models$ ROB. We have to check that we have $A \models \psi(\vec{a}, b)$ iff $b = f(\vec{a})$, i.e. iff

  (1) b is the smallest element of $\mathbb{N}$ with $g(\vec{a}, b) = 0$.

  Since ϕ represents g, this is equivalent to:

  (2) b is the smallest element of $\mathbb{N}$ with $A \models \phi(\vec{a}, b, 0)$.

  Since all elements in $A \setminus \mathbb{N}$ are bigger than all elements of $\mathbb{N}$, (2) is equivalent to:

  (3) b is the smallest element of A with $A \models \phi(\vec{a}, b, 0)$.

  Now (3) is exactly what ψ expresses.

□

Corollary 4.2.5. *Every decidable relation is representable in* ROB.

Proof. Let $R(\vec{x})$ be decidable, i.e., the representing function

$$K_R\colon \mathbb{N}^k \to \mathbb{N}, \vec{x} \mapsto \begin{cases} 0 & \text{if } R(\vec{x}) \text{ holds} \\ 1 & \text{if } R(\vec{x}) \text{ does not hold} \end{cases}$$

is computable. Then, by Theorem 4.2.4, K_R is representable in ROB, i.e., there exists a formula $\phi(\vec{x}, y)$ such that for $A \models \mathrm{ROB}$, $\vec{a} \in \mathbb{N}^k$ and $b \in A$, we have:
$A \models \phi(\vec{a}, b)$ iff $K_R(\vec{a}) = b$.

In particular, this implies:
$A \models \phi(\vec{a}, 0)$ iff $R(\vec{a})$ holds.

Thus $\psi(\vec{x}) := \phi(\vec{x}, 0)$ represents $R(\vec{x})$. □

4.3. The Fixed Point Theorem.

4.3. The Fixed Point Theorem. One central ingredient to the proof of both Incompleteness results we are aiming for is to find a sentence θ which expresses something like "θ cannot be proven in PA". It seems that this does not make sense: How can a sentence speak about itself? We will now prove that this is indeed possible in general, i.e., for any formula $\phi(x)$, there exists a sentence θ expressing that $\phi(\ulcorner\theta\urcorner)$ holds. (Thus θ expresses that itself has the property defined by ϕ.)

At first sight, this seems impossible, since the sentence $\theta' = \phi(\ulcorner\theta\urcorner)$ will always be longer than θ itself. (Recall that the "$\ulcorner\theta\urcorner$" in θ' stands for $S(S(\dots S(0)))$ and already this part of θ' is much longer than θ.) So θ will be different from θ', but the point will be that θ and θ' will be equivalent. And in fact, this equivalence will already follow from ROB. Here is the precise theorem.

Theorem 4.3.1 (Fixed Point Theorem)**.** *Let $\phi(x)$ be any formula. Then there exists a sentence θ such that* ROB $\vdash \theta \leftrightarrow \phi(\ulcorner\theta\urcorner)$*.*

Remark: What if, in the Fixed Point Theorem, we would only ask that θ and $\phi(\ulcorner\theta\urcorner)$ are equivalent, but not that the equivalence can be proven in ROB? Does this make sense? Does the result become trivial? That θ and $\phi(\ulcorner\theta\urcorner)$ are equivalent in $\mathbb{N}$ just means that either both are true in $\mathbb{N}$ or both are false in $\mathbb{N}$. Thus in that version, the FPT states that there is no formula $\phi(x)$ such that $\phi(\ulcorner\theta\urcorner)$ holds iff θ is false. By replacing ϕ by its negation, we obtain that even this weak version of the FPT implies the following strengthening of GInc1:

Corollary 4.3.2. *The set of true sentences is not arithmetical, i.e., there is no formula $\phi(x)$ such that for sentences θ, $\phi(\ulcorner\theta\urcorner)$ holds iff $\mathbb{N} \models \theta$.*

Proof of Theorem 4.3.1. Let $\text{subs}(x, y)$ be the following function: if $x = \ulcorner\phi\urcorner$ where ϕ is a formula with a single free variable, then $\text{subs}(x, y) = \ulcorner\phi(\underline{y})\urcorner$. (where $\underline{y} = \underbrace{S(S(\dots S}_{y}(0))))$.

The idea is to choose $\psi(x) = \phi(\text{subs}(x, x))$ and $\theta = \psi(\ulcorner\psi\urcorner)$. These are not yet exactly formulas, but let me first explain intuitively why this θ should be equivalent to $\phi(\theta)$.

$\psi(x)$ says: ϕ holds for the sentence obtained by applying the formula x to itself.

Thus θ says: ϕ holds for the $\underbrace{\text{sentence obtained by applying the formula } \psi(x) \text{ to itself}}_{(*)}$.

Now what is (*)? If we apply $\psi(x)$ to itself, we obtain exactly θ. Hence θ is equivalent to: ϕ holds for θ.

It remains to do two things:
(1) Turn ψ into a precise formula
(2) Carry out the above proof of equivalence inside ROB

(1)
It is not difficult to check that $\text{subs}(x, y)$ is computable. Therefore, by the representability theorem, there exists a formula $\phi_{\text{subs}}(x, y, z)$ which holds iff $\text{subs}(x, y) = z\dots$ and such that for all $a, b \in \mathbb{N}$ and $c = \text{subs}(a, b)$, we have:
(**) ROB $\vdash \forall y : (\phi_{\text{subs}}(a, b, y) \leftrightarrow y = c)$.

We use ϕ_{subs} to define ψ:

$$\psi(x) :\Longleftrightarrow \exists z : (\phi_{\text{subs}}(x, x, z) \wedge \phi(z)).$$

(2)
Using (**), the above proof directly works in ROB. Let's look at this slowly.

θ is $\psi(\ulcorner\psi\urcorner)$, which in turn is $\exists z : (\underbrace{\phi_{\text{subs}}(\ulcorner\psi\urcorner, \ulcorner\psi\urcorner, z)}_{(+)} \wedge \phi(z))$.

By (**), (+) is equivalent to $z = \text{subs}(\ulcorner\psi\urcorner, \ulcorner\psi\urcorner) = \ulcorner\psi(\ulcorner\psi\urcorner)\urcorner = \ulcorner\theta\urcorner$. Thus the whole of θ is equivalent to $\exists z : (z = \ulcorner\theta\urcorner \wedge \phi(z)))$, which can be simplified to $\phi(\ulcorner\theta\urcorner)$. $\square$

The above FPT says that a sentence can speak about itself. For some applications, one needs a variant of this for formulas:

Theorem 4.3.3. *Let $\phi(x, \vec{y})$ be any formula. Then there exists a formula $\theta(\vec{y})$ such that* $\mathrm{ROB} \vdash \forall \vec{y} : (\theta(\vec{y}) \leftrightarrow \phi(\ulcorner\theta\urcorner, \vec{y}))$.

The proof works in exactly the same way as the proof of Theorem 4.3.1, except that in various places, one has these additional variables, which makes it look more messy.

Exercise 4.3.4. Does there exist a formula $\phi(x)$ which holds only for one x, namely when x is the number of symbols ϕ is made of?

Use the Fixed Point Theorem to prove that such a ϕ exists.

4.4. The generalized First Incompleteness Theorem.

We now can prove our first Advanced-Proof-And-Computation goal, namely the following generalized version of GInc1. First a definition:

Definition 4.4.1. *If T is any set of sentences, we write*

$$T^* := \{\phi \mid T \vdash \phi\}$$

of sentences which can be deduced from T.

(As an example, if T is inconsistent, then T^* consists of all sentences; but that's of course an uninteresting case.)

Theorem 4.4.2. *If T is any consistent set of sentences containing ROB, then T^* is not decidable.*

In particular, if we take $T = \mathrm{PA}$, we get that there's no algorithm which, given a sentence ϕ, decides whether ϕ can be proven from PA, can be disproven, or neither-nor.

Note: Our previous version of the first incompleteness theorem is the special case where T is the set of all true sentences.

Proof of 4.4.2. Suppose that T^* would be decidable. Then by the representability theorem, it is representable in T, i.e., there exists a formula $\tau(x)$ such that:

(i) If $\psi \in T^*$ (i.e., if $T \vdash \psi$), then $T \vdash \tau(\ulcorner\psi\urcorner)$;
(ii) If $\psi \notin T^*$ (i.e., if $T \nvdash \psi$), then $T \vdash \neg\tau(\ulcorner\psi\urcorner)$.

(The Representability Theorem yields representability in ROB, but T contains ROB, so everything provable in ROB is also provable in T.)

Apply the fixed point theorem to $\phi(x) = \neg\tau(x)$. This yields a sentence θ such that

(*) $T \vdash \theta \leftrightarrow \neg\tau(\ulcorner\theta\urcorner)$.

In other words, θ claims that itself is not provable in T.

We will now check that both, $T \nvdash \theta$ and $T \vdash \theta$, lead to a contradiction, hence showing that $\tau(x)$ cannot exist.

Claim 1: $T \nvdash \theta$ is not possible.

Suppose otherwise, i.e., $T \nvdash \theta$.
Then by (ii), we have $T \vdash \neg\tau(\ulcorner\theta\urcorner)$.
By (*), we obtain $T \vdash \theta$, contradicting the assumption.

Claim 2: $T \vdash \theta$ is not possible.

Suppose otherwise, i.e., $T \vdash \theta$.
Then by (i) we have $T \vdash \tau(\ulcorner\theta\urcorner)$.
By (*), this is equivalent to $T \vdash \neg\theta$.
So now we got both, $T \vdash \theta$ and $T \vdash \neg\theta$. However, this is not possible since T was assumed to be consistent. □

From this, we can now deduce another nice result: In some sense, even if one starts with nothing at all, one gets incompleteness:

Corollary 4.4.3. *The set $\{\phi \mid \emptyset \vdash \phi\}$ of sentences which hold in every structure is not decidable.*

Proof. The key ingredient here is that ROB only consists of finitely many axioms. Let ψ_{ROB} be the conjunction of all those axioms. Then $\mathrm{ROB} \vdash \phi$ iff $\emptyset \vdash (\psi_{\mathrm{ROB}} \to \phi)$. Thus if we had a register machine which can decide for which sentences ϕ' we have $\emptyset \vdash \phi'$, then by applying that machine to $\psi_{\mathrm{ROB}} \to \phi$, we could find out whether $\mathrm{ROB} \vdash \phi$. □

Exercise 4.4.4. From Theorem 4.4.2 and Corollary 4.4.3, one starts getting the feeling that for *any* consistent set T of sentences, the set

$$T^* := \{\phi \mid T \vdash \phi\}$$

is not decidable.

Why does this not follow from Theorem 4.4.2 and Corollary 4.4.3? Give an example of a set of sentences T such that T^* is decidable.

Exercise 4.4.5. Deduce Rosser's incompleteness Theorem:

Theorem 4.4.6. *If T is a consistent, semi-decidable set of sentences containing* ROB*, then T is not complete.*

(Note: If we would impose T to consist only of true sentences, this would be just a reformulation of GIncl, in the version of Theorem 3.5.3. So the point here is that T is allowed to contain sentences which are false in the usual natural numbers.)

Theorem 4.4.2 says that T^* is not decidable for certain T. (Recall: $T^* = \{\phi \mid T \vdash \phi\}$.) What about semi-decidability? To end this section, we prove:

Proposition 4.4.7. *If T is semi-decidable, then so is T^*.*

We essentially already saw how to prove this, in the proof of Corollary 3.5.5. Since this was a bit quick and sketchy, let me prove it again in a bit more detail:

Proof of Prop. 4.4.7. Recall: $T \vdash \phi$ means that ϕ follows from T using natural deduction.

Also recall: We can somehow fix a way to encode proofs in natural deduction, and then checking whether something is a valid proof can be done by a register machine. More formally, the relation $\mathrm{proof}(x, y, z)$ is decidable, where $\mathrm{proof}(\langle\ulcorner\phi_1\urcorner, \dots, \ulcorner\phi_n\urcorner\rangle, \ulcorner\psi\urcorner, z)$ holds iff z is the code of a proof of ψ using the assumption $\phi_1, \dots, \phi_n$.

Let M be a machine semi-deciding T. Using this, one can semi-decide T^* as follows. Let $\ulcorner\psi\urcorner$ be given.

Go systematically through all triples $(x, z, u) \in \mathbb{N}^3$ and for each of them, do the following:

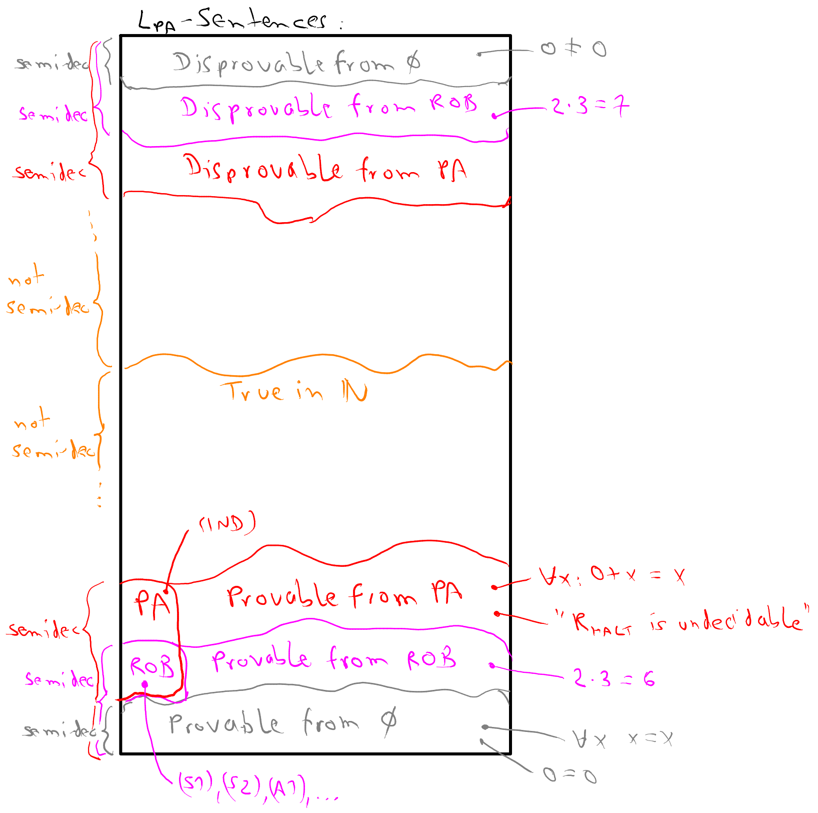
```
    Check whether $x$ is a code of a sequence of sentences $(\phi_1, \dots, \phi_n)$
    Check whether $\mathrm{proof}(x, \ulcorner\psi\urcorner, z)$ holds.
    For each $i \leq n$, check whether $M$ with input $\ulcorner\phi_i\urcorner$ halts after at most $u$ steps.
    If all this is true, then halt. (Otherwise go on trying other $x$, $z$, $u$)
```

I claim that this algorithm halts iff $\psi \in T^*$. Indeed:

If it does halt, then it means that all ϕ_i are in T and that a proof of ψ from those ϕ_i exists (namely the one encoded by z).

Vice versa, if ψ follows from T, then there exist finitely many $\phi_1, \dots, \phi_n \in T$ from which ψ follows, and for each of those ϕ_i, the machine M halts after a finite amount of time on input $\ulcorner\phi_i\urcorner$. Thus the above algorithm halts when u is the maximum of those times (for $\phi_1, \dots, \phi_n$), $x = \{\phi_1, \dots, \phi_n\}$, and z is a code of the proof of ψ using those ϕ_i. □

4.5. **Overview over things we have seen.** Here's a picture summarizing some of the sets of sentences, and which of them are (not) (semi-)decidable:



Suppose that T is a consistent set of sentences and set $T^* = \{\phi \mid T \vdash \phi\}$. The following table gives an overview over what we can say about T^* under assumptions on T.

$T = \emptyset$	T^* not decidable (but semi-decidable)	(Cor. 4.4.3)
T contains ROB	T^* not decidable	(Thm 4.4.2)
T = all true sentences	T^* ($= T$) not arithmetical	(Thm 4.3.2)
T semi-decidable	T^* semi-decidable	(Prop 4.4.7)
T semi-decidable and complete	T^* decidable	(Prop 4.4.7)

Note: The last line follows from the fact that complete and semi-decidable implies decidable.

4.6. **The Statement of Gödel's Second Incompleteness Theorem.** Now let's move on to Gödel's Second Incompleteness Theorem (GInc2). First, let's state it precisely. Recall that the statement is supposed to be that

PA $\nvdash$ "PA is consistent".

To make GInc2 precise, we have to express "PA is consistent" as a sentence.

Definition 4.6.1. *Let $\bot$ be any sentence which is never true, e.g., $\bot = (0 \neq 0)$.*

One easily checks that a set of sentences T is consistent iff $T \nvdash \bot$, so we can take this as the definition of T being consistent. Now let $\mathrm{prov}_{\mathrm{PA}}(x)$ be the relation which holds iff x is the code of a sentence which follows from PA. By Proposition 4.4.7, $\mathrm{prov}_{\mathrm{PA}}(x)$ is semi-decidable; thus it is arithmetical, i.e., $\mathrm{prov}_{\mathrm{PA}}(x)$ can be defined by a formula, which I will denote by $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$. Using this, we can express "PA is consistent" as follows:

$\neg\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\bot\urcorner)$

So now it seems that we can state GInc2 precisely:

Theorem 4.6.2 (Gödel's Second Incompleteness Theorem)**.** PA $\nvdash \neg\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\bot\urcorner)$.

However, there is a problem with this formulation, namely: what *exactly* is the formula $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$? If we just fix an arbitrary formula $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ defining the relation $\mathrm{prov}_{\mathrm{PA}}(x)$, then two problems might arise:

(1) It might be that the theorem becomes wrong. For example, let $\psi(x)$ be any formula defining the relation $\mathrm{prov}_{\mathrm{PA}}(x)$, and consider $\psi'(x) = \psi(x) \wedge x \neq \ulcorner\bot\urcorner$. Then we have PA $\vdash \neg\psi'(\ulcorner\bot\urcorner)$ for trivial reasons.

(2) It might be that the theorem becomes "trivially true": Again, let $\psi(x)$ define $\mathrm{prov}_{\mathrm{PA}}(x)$, and choose a sentence θ which is false but such that $\neg\theta$ cannot be proven from PA. Then $\psi''(x) := \psi(x) \vee \theta$ also defines $\mathrm{prov}_{\mathrm{PA}}(x)$, but we have trivially

PA $\nvdash \neg\psi''(\ulcorner\bot\urcorner)$,

since if PA $\vdash \neg(\psi(\ulcorner\bot\urcorner) \vee \theta)$ would in particular imply PA $\vdash \neg\theta$.

There exists a good solution to problem (1): we will specify precise properties which the $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ should satisfy so that we can prove Theorem 4.6.2.

For (2), no good solution seems to exist. Philosophically, one could say that even though $\psi''(x)$ *defines* $\mathrm{prov}_{\mathrm{PA}}(x)$, it *expresses* something different, namely "x is provable *or θ is false*". However, no precise mathematical definition exists of what we mean by "express".

The more or less only solution is to specify explicitly which formula $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ we want to take, in such a way that we intuitively think that it expresses the right thing. For example, we can define:

Definition 4.6.3. *Let $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ be the formula defining $\mathrm{prov}_{\mathrm{PA}}(x)$ obtained by going through the proof of definability of $\mathrm{prov}_{\mathrm{PA}}(x)$ given in this lecture.*

It then remains to prove that this specific formula $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ satisfies the properties mentioned above (in the solution to problem (1)). Carrying this out entirely in all details is very long and tedious: it would require redoing large parts of this lecture within PA. Instead of doing that, we will believe that PA works as expected and skip most of that work.

Exercise 4.6.4. What about solving problem (2) by taking a formula $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ which *represents* the relation $\mathrm{prov}_{\mathrm{PA}}(x)$? Would this help? Do we know that such a formula exists? (Or maybe we know that it does not exist?)

4.7. **Provability Conditions.** As announced, instead of proving GInc2 for a specific formula $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$, we will prove it for any formula $P(x)$ which has certain properties which express that in some sense, $P(x)$ behaves like a notion of provability. The conditions on $P(x)$ are the following.

Definition 4.7.1. *Let T be a set of sentences and $P(x)$ be a formula. We say that P satisfies the* ***(Hilbert–Bernays) Provability Conditions*** *(for T) if for every sentence ϕ and ψ,* *the following hold:*

(P1) If $T \vdash \phi$, then $T \vdash P(\ulcorner\phi\urcorner)$
(P2) $T \vdash (P(\ulcorner\phi\urcorner) \wedge P(\ulcorner\phi \to \psi\urcorner)) \to P(\ulcorner\psi\urcorner)$
(P3) $T \vdash P(\ulcorner\phi\urcorner) \to P(\ulcorner P(\ulcorner\phi\urcorner)\urcorner)$

Note: Here, when I wrote e.g. $P(\ulcorner\phi\urcorner)$, what I really mean is $P(S(S(\dots S(0))))$, where the number of S's is $\ulcorner\phi\urcorner$. So I could have written this as $P(\underline{\ulcorner\phi\urcorner})$, but this would have made things look rather messy.

Here is some intuition about Definition 4.7.1; I will write $\mathrm{prov}_T(x)$ for the relation which holds if x is a code of a sentence provable in T. To make it less confusing, in the intuitive description, I will assume that T only contains true sentences. (However, we will obtain an abstract version of GInc2 for any consistent T containing ROB, without the condition that all sentences are true.)

(P1) says that if $\mathrm{prov}_{\mathrm{PA}}(\ulcorner\phi\urcorner)$ holds, then $P(\ulcorner\phi\urcorner)$ should hold and even be provable in T. This is half of the statement that $P(x)$ represents $\mathrm{prov}_{\mathrm{PA}}(x)$.

The other half would be that if $\mathrm{prov}_{\mathrm{PA}}(\ulcorner\phi\urcorner)$ does not hold, then $\neg P(\ulcorner\phi\urcorner)$ should be provable in T; however, we have seen that we cannot expect this other direction to hold.

Note that we do not even require that $P(x)$ defines $\mathrm{prov}_{\mathrm{PA}}(x)$: it is allowed that $P(\ulcorner\phi\urcorner)$ holds for sentences ϕ which do not follow from T. Thus one should think of the above conditions as stating that $P(x)$ defines some notion of provability which might allow to prove more than T. For this reason, one should really distinguish between "T-provable" and "P-provable". However, to make it more intuitive, let's think of the case where $P(x)$ defines $\mathrm{prov}_T(x)$. (After all, that's the situation we will really apply this to.)

In (P2), "$(P(\ulcorner\phi\urcorner) \wedge P(\ulcorner\phi \to \psi\urcorner)) \to P(\ulcorner\psi\urcorner)$ says that if ϕ and $\phi \to \psi$ can be proven, then ψ can also be proven; in other words, this just says that modus ponens holds. (P2) says that the fact that modus ponens holds should be provable in T.

In (P3), "$P(\ulcorner\phi\urcorner) \to P(\ulcorner P(\ulcorner\phi\urcorner)\urcorner)$" is just (P1), formulated as a formula (using P), i.e., "P half-represents $\mathrm{prov}_{\mathrm{PA}}(x)$". (P3) says that this can be proven in T.

Exercise 4.7.2. (1) Does the formula $P(x)$ which is always true satisfy the above conditions?
(2) Check whether these conditions exclude "Problem (2)" from Section 4.6. More precisely:
Suppose that $P(x)$ defines $\mathrm{prov}_{\mathrm{PA}}(x)$ and satisfies the Provability Conditions, take a sentence θ such that $\mathbb{N} \models \neg\theta$ but PA $\not\vdash \neg\theta$, and define $P'(x) := \quad P(x) \vee \theta$. Does $P'(x)$ satisfy the Provability Conditions?
Hint: Be careful especially about (P3). The following Lemma might be useful.

A priori, it seems unclear whether the Provability Conditions really imply that $P(x)$ behaves like a notion of provability, i.e.: Suppose that we can deduce a sentence ψ from some other sentences $\phi_1, \dots, \phi_\ell$ in some easy way. Then we'd want that $P(x)$ reflects this, i.e., if $P(\ulcorner\phi_1\urcorner) \dots, P(\ulcorner\phi_k\urcorner)$ hold, then $P(\ulcorner\psi\urcorner)$ should also hold. By the following lemma, this is indeed true. More precisely, "in some easy way" can be dropped; we just need to be able to carry out that deduction within T.

Lemma 4.7.3. *Suppose that $P(x)$ satisfies the Provability Conditions, and suppose that $\phi_1, \dots, \phi_\ell$ and ψ are sentences such that $T \vdash (\phi_1 \wedge \dots \wedge \phi_\ell) \to \psi$. Then $T \vdash (P(\ulcorner\phi_1\urcorner) \wedge \dots \wedge P(\ulcorner\phi_\ell\urcorner)) \to P(\ulcorner\psi\urcorner)$.*

Proof. Let's first do the case $\ell = 1$:

Applying (P1) to $T \vdash \phi_1 \to \psi$ yields $T \vdash P(\ulcorner\phi_1 \to \psi\urcorner)$.

Now consider (P2): $T \vdash (\underbrace{P(\ulcorner\phi_1\urcorner)}_{(*)} \wedge \underbrace{P(\ulcorner\phi_1 \to \psi\urcorner)}_{(**)}) \to \underbrace{P(\ulcorner\psi\urcorner)}_{(***)}$. Since (**) holds (in T), we have the implication (*) $\to$ (***) (also in T), which is exactly what we want.

Now let's do the case $\ell = 2$:

The assumption $T \vdash (\phi_1 \wedge \phi_2) \to \psi$ is equivalent to: $T \vdash \phi_1 \to (\phi_2 \to \psi)$. Applying the $\ell = 1$ case to that formula yields

$T \vdash P(\ulcorner \phi_1 \urcorner) \to P(\ulcorner \phi_2 \to \psi \urcorner)$.

By (P2), we have

$T \vdash (P(\ulcorner \phi_2 \urcorner) \wedge P(\ulcorner \phi_2 \to \psi \urcorner)) \to P(\ulcorner \psi \urcorner)$.

Combining these two yields the desired result.

Finally, suppose that $\ell > 2$. We apply the $\ell = 2$ case to the following implications:

$$\begin{aligned}
&\phi_1 \wedge \phi_2 \to (\phi_1 \wedge \phi_2)\\
&(\phi_1 \wedge \phi_2) \wedge \phi_3 \to (\phi_1 \wedge \phi_2 \wedge \phi_3)\\
&\quad\vdots\\
&(\phi_1 \wedge \cdots \wedge \phi_{\ell-2}) \wedge \phi_{\ell-1} \to (\phi_1 \wedge \cdots \wedge \phi_{\ell-1})\\
&(\phi_1 \wedge \cdots \wedge \phi_{\ell-1}) \wedge \phi_\ell \to \psi
\end{aligned}$$

This yields

$$\begin{aligned}
&T \vdash P(\ulcorner \phi_1 \urcorner) \wedge P(\ulcorner \phi_2 \urcorner) \to P(\ulcorner \phi_1 \wedge \phi_2 \urcorner)\\
&T \vdash P(\ulcorner \phi_1 \wedge \phi_2 \urcorner) \wedge P(\ulcorner \phi_3 \urcorner) \to P(\ulcorner \phi_1 \wedge \phi_2 \wedge \phi_3 \urcorner)\\
&\quad\vdots\\
&T \vdash P(\ulcorner \phi_1 \wedge \cdots \wedge \phi_{\ell-2} \urcorner) \wedge P(\ulcorner \phi_{\ell-1} \urcorner) \to P(\ulcorner \phi_1 \wedge \cdots \wedge \phi_{\ell-1} \urcorner)\\
&T \vdash P(\ulcorner \phi_1 \wedge \cdots \wedge \phi_{\ell-1} \urcorner) \wedge P(\ulcorner \phi_\ell \urcorner) \to P(\ulcorner \psi \urcorner)
\end{aligned}$$

All this together yields the desired result □

4.8. **A formula satisfying the Provability Conditions.**

Proposition 4.8.1. $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ *satisfies the Provability Conditions for* PA.

I will give a precise proof of (P1); for (P2), I will do a lot of hand-waving, and the proof of (P3), will essentially be skipped.

The key for (P2) and (P3) is that "almost everything we have proven in this lecture can also be proven within PA". To be more precise, to be able to prove something within PA, we need to encode by numbers all objects arising in the proof. This means that we can formulate proofs in PA which involve e.g. sequences, register machines and formulas, but the proof should (for example) not involve structures: Since there are uncountably many of them, they cannot be encoded by numbers.

Proof of Proposition 4.8.1, (P1). To get a simple precise proof of (P1), we first need to think a bit more precisely about which formula $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ we choose in Definition 4.6.3.

Recall (from the proof of Proposition 4.4.7) that the relation

$$\mathrm{proof}(\langle\ulcorner\phi_1\urcorner,\dots\ulcorner\phi_k\urcorner\rangle,\ulcorner\psi\urcorner,z) :\Longleftrightarrow z \text{ encodes a proof of } \psi \text{ using the assumptions } \phi_1,\dots,\phi_k$$

is decidable. Since PA is decidable, we also get that the following is decidable:

$$\mathrm{proof}_{\mathrm{PA}}(\ulcorner\psi\urcorner,z) :\Longleftrightarrow z \text{ encodes a proof of } \psi \text{ using axioms from PA as assumptions.}$$

"Going through the lecture" (according to Definition 4.6.3) means: Construct a formula $\underline{\mathrm{proof}}_{\mathrm{PA}}(x,z)$ defining $\mathrm{proof}_{\mathrm{PA}}(x,z)$, and then define

$$\underline{\mathrm{prov}}_{\mathrm{PA}}(x) :\Longleftrightarrow \exists z : \underline{\mathrm{proof}}_{\mathrm{PA}}(x,z).$$

And actually, $\underline{\mathrm{proof}}_{\mathrm{PA}}(x,z)$ even represents $\mathrm{proof}_{\mathrm{PA}}(x,z)$ in PA (or even in ROB). This is what we need now to prove that $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ satisfies (P1).

What we need to prove is: If PA $\vdash \phi$, then PA $\vdash \underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\urcorner)$.

Suppose that PA $\vdash \phi$. Then there exists a proof of ϕ using the axioms of PA and natural deduction. Let $c \in \mathbb{N}$ be a code of that proof. The fact that c is a code of a proof of ϕ means that $\mathrm{proof}_{\mathrm{PA}}(\ulcorner\phi\urcorner,c)$ holds. Since $\underline{\mathrm{proof}}_{\mathrm{PA}}(x,z)$ represents $\mathrm{proof}_{\mathrm{PA}}(x,z)$, we get that PA $\vdash \underline{\mathrm{proof}}_{\mathrm{PA}}(\ulcorner\phi\urcorner,c)$.

In particular, this implies PA $\vdash \exists z : \underline{\mathrm{proof}}_{\mathrm{PA}}(\ulcorner\psi\urcorner,z)$; this is exactly what we had to prove.

□

Sketch of proof of Proposition 4.8.1 (P2). We want to prove:
$$\mathrm{PA} \vdash (\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\urcorner) \wedge \underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\rightarrow\psi\urcorner)) \rightarrow \underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\psi\urcorner).$$

Let ϕ and ψ be given. We have to prove within PA:

(*) If $\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\urcorner)$ and $\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\rightarrow\psi\urcorner)$ hold, then $\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\psi\urcorner)$ also holds.

Let's first prove that (*) is true (in $\mathbb{N}$), without bothering that it should be provable within PA. For this, the argument is the following: That $\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\urcorner)$ holds means that PA $\vdash \phi$. Similarly, we have PA $\vdash \phi \rightarrow \psi$. Thus PA $\vdash \psi$, and hence $\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\psi\urcorner)$ holds.

Now how can this proof be carried out within PA? Before thinking about that, let us reformulate (*):

(*') If there exists a formal proof of ϕ and a formal proof of $\phi \rightarrow \psi$, then there also exists a formal proof of ψ.

To prove this implication, we can just specify the formal proof of ψ: Put the formal proofs of ϕ and $\phi \rightarrow \psi$ together, and add one step at the end, namely

$$\frac{\phi \qquad \phi\rightarrow\psi}{\psi}.$$

This proof of (*') can be carried out within PA. Recall: At an even more formal level, "$\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\urcorner)$" is "$\exists z : \underline{\mathrm{proof}}_{\mathrm{PA}}(\ulcorner\phi\urcorner,z)$", where $\underline{\mathrm{proof}}_{\mathrm{PA}}$ expresses that z encodes a formal proof of ϕ. So by the left hand side of (*'), we have a z_1 which encodes a formal proof of ϕ and a z_2 which encodes a formal proof of $\phi \to \psi$. From these two codes, we can compute the code z_3 of the formal proof of ψ mentioned above. This computation can be expressed by a formula. What we then really have to prove within PA is that this computation really yields the desired result, i.e., that the z_3 obtained in that way really does satisfy the formula $\underline{\mathrm{proof}}_{\mathrm{PA}}(\ulcorner\psi\urcorner, z_3)$. This involves quite some work, but since everything is expressed in terms of numbers, it can be done. $\square$

Not really a sketch of proof of Proposition 4.8.1 (P3). We want to prove:
$\mathrm{PA} \vdash \underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\urcorner) \to \underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\phi\urcorner)\urcorner)$

As already mentioned, (P3) simply claims that (P1) can be proved within PA, so we "just" have to carry out that proof within PA. $\square$

4.9. **Proof of G2 using the provability conditions.**

Theorem 4.9.1 (Gödel's Second Incompleteness Theorem, abstract version)**.** *Suppose that T is a consistent set of sentences containing ROB and suppose that there exists a formula $P(x)$ satisfying the Provability Conditions (Defn 4.7.1). Then $T \nvdash \mathrm{CON}_T$, where*

$$\mathrm{CON}_T := \quad \neg P(\ulcorner \bot \urcorner).$$

Remark: By Proposition 4.8.1, this in particular implies the concrete version of the second incompleteness theorem for PA (Theorem 4.6.2).

Proof of 4.9.1. We start as in the proof of Thm 4.4.2 (the generalized First Incompleteness Theorem): We apply the Fixed Point Theorem to $\phi(x) = \neg P(x)$ to obtain a sentence θ such that
(1) $T \vdash \theta \leftrightarrow \neg P(\ulcorner \theta \urcorner)$.

I.e., again, θ claims that itself is not provable.

(Note: It is to be able to apply the Fixed Point Theorem that in 4.9.1 we required that T contains ROB.)

By (P3), we have
(2) $T \vdash P(\ulcorner \theta \urcorner) \to P(\ulcorner P(\ulcorner \theta \urcorner) \urcorner)$

(1) in particular implies $T \vdash P(\ulcorner \theta \urcorner) \to \neg\theta$; applying Lemma 4.7.3 to that yields
$T \vdash P(\ulcorner P(\ulcorner \theta \urcorner) \urcorner) \to P(\ulcorner \neg\theta \urcorner)$.

Combining that with (2) yields
(3) $T \vdash P(\ulcorner \theta \urcorner) \to P(\ulcorner \neg\theta \urcorner)$

This says: if θ is provable (according to P), then $\neg\theta$ is provable. This means that if θ is provable, we get a contradiction. Let's make this formal:

Applying Lemma 4.7.3 to $\theta \wedge \neg\theta \to \bot$ yields
$T \vdash P(\ulcorner \theta \urcorner) \wedge P(\ulcorner \neg\theta \urcorner) \to P(\ulcorner \bot \urcorner)$.

Combining this with (3) yields
(4) $T \vdash P(\ulcorner \theta \urcorner) \to P(\ulcorner \bot \urcorner)$.

Now suppose that the theorem is wrong, i.e., that $T \vdash \neg P(\ulcorner \bot \urcorner)$.

By (4), we get
(5) $T \vdash \neg P(\ulcorner \theta \urcorner)$.

Now (1) implies
$T \vdash \theta$,
and then (P1) implies
$T \vdash P(\ulcorner \theta \urcorner)$,
which contradicts (5).

□

4.10. **What's next?** To end the lecture, let me mention a few continuations in proof theory.

Löb's Theorem:

Using a similar proof as the one of GInc2, one can obtain:

Theorem 4.10.1. *If T contains ROB and $P(x)$ satisfies the Provability Conditions, then for every sentence θ, we have the following equivalence:*
(*) $\quad T \vdash P(\ulcorner \theta \urcorner) \to \theta \quad \iff \quad T \vdash \theta$

Let me explain the meaning a bit; let's assume that T is PA and $P(x)$ is $\underline{\mathrm{prov}}_{\mathrm{PA}}(x)$ (to make things more intuitive).

The direction from right to left is trivial anyway. The direction from left to right is saying: If we can prove θ using that θ is provable, then we can also prove it without using it. In other words, knowing that θ is provable doesn't help to prove θ. At first sight, this seems pretty strange: one would almost think that "$P(\ulcorner\theta\urcorner) \to \theta$" should be trivially true: if θ is provable, then it should obviously hold. However, this argument assumes that our theory is consistent; after all, if it's not, then anything is provable. Now we might know that PA is consistent (and hence we know that $P(\ulcorner\theta\urcorner) \to \theta$ holds); however, in PA, we cannot prove that PA is consistent (by GInc2), so the argument does not imply "PA $\vdash P(\ulcorner\theta\urcorner) \to \theta$".

Theorem 4.10.1 implies Theorem 4.9.1 by taking $\theta = \bot$: Since $T \nvdash \bot$, using (*) we get $T \nvdash P(\ulcorner\bot\urcorner) \to \bot$, and $P(\ulcorner\bot\urcorner) \to \bot$ is equivalent to $\neg P(\ulcorner\bot\urcorner)$.

Theorem 4.10.1 is almost what one calls Löb's Theorem: Löb's Theorem says that (*) can be proven in T:

Theorem 4.10.2 (Löb's Theorem). *If T contains ROB and $P(x)$ satisfies the Provability Conditions, then for every sentence θ, we have the following:*
$T \vdash P(\ulcorner P(\ulcorner\theta\urcorner) \to \theta\urcorner) \leftrightarrow P(\ulcorner\theta\urcorner)$

Modal logic:

Formulas in modal logic consist of usual first-order formulas together with a new symbol $\Box$ expressing "how" something is true. If ϕ is a sentence, then $\Box\phi$ could mean "I believe ϕ is true" or "ϕ will always be true" or "ϕ is trivially true" or something like that. Depending on what $\Box$ is supposed to mean, one adds suitable axioms and inference rules. One possible meaning of $\Box\phi$ is: "ϕ can be proven (in some fixed theory)." In that case, one would use (P1), (P2), (P3) from Definition 4.7.1 (and maybe some more), where (P1) becomes a rule and (P2) and (P3) become axioms:

(P1) From ϕ, one is allowed to deduce $\Box\phi$.
(P2) $(\Box\phi \wedge \Box(\phi \to \psi)) \to \Box\psi$
(P3) $\Box\phi \to \Box\Box\phi$.

In that notation, the conclusion of Löb's Theorem becomes:
$T \vdash \Box(\Box\theta \to \theta) \leftrightarrow \Box\theta$

(Note however that the axioms (P1)–(P3) alone are not enough to imply Löb's Theorem; one also needs the Fixed Point Theorem.)

Set Theory:

In PA, one can do quite a lot of mathematics, but there are also a lot of things one can't do. For example, one cannot speak about real numbers: encoding real numbers by natural numbers is not possible since there are uncountably many. A set of axioms which is strong enough to do essentially everything one might be interested in is ZFC, which describes what one can do with sets. Whereas the elements of a model of PA are numbers (or better: are supposed to be considered as numbers), all elements of a model of ZFC are (supposed to be) sets. Instead of $L_{\mathrm{PA}} = \{0, S, +, \cdot, <\}$, it uses a language which has only one single binary relation, namely "$\in$", which says that one set is an element of another one. In the same way as we encoded lots of stuff by numbers to speak about it within PA, one can encode lots of stuff as sets; a natural number n can, for example, be encoded by a set with n elements. In this way, it is not very difficult to see that everything which can be done in PA can also be done in ZFC. However, we can do more. For example, in ZFC, we can speak about the set $\mathbb{N}$, define addition and multiplication on it and check that it satisfies all the axioms of PA. In other words, we can prove that $\mathbb{N}$ is a model of PA, which means: ZFC $\vdash \neg\,\mathrm{prov}_{\mathrm{PA}}(\ulcorner\bot\urcorner)$. On the other hand, Gödel's Incompleteness Theorems can also be proven for ZFC (it's even a bit easier), i.e., ZFC $\nvdash \neg\,\mathrm{prov}_{\mathrm{ZFC}}(\ulcorner\bot\urcorner)$.

And then one can come up with an even stronger set of axioms from which one can prove that ZFC is consistent, etc.; this yields an entire hierarchy of systems of axioms.

Goodstein's Theorem:

We've seen an explicit sentence which is true but can't be proven in PA, namely $\neg\underline{\mathrm{prov}}_{\mathrm{PA}}(\ulcorner\bot\urcorner)$. However, one can ask: Do there also exist such sentences (i.e., which are true but not provable in PA) which don't look that artificial, i.e., which would seem natural also to a non-logician? Goodstein's Theorem is such a sentence.

Definition 4.10.3. *A **Goodstein Sequence** is a sequence of natural numbers obtained as follows. Start with any number $a_1 \in \mathbb{N}$ and then define a_k recursively using a_{k-1}, as follows.*
Write a_{k-1} "hereditarily in base k" (to be explained below)
Replace the base by $k+1$
Subtract one from the resulting number.

This is best explained on an example: let's take $a_1 = 10$. This can be written in base 2:
$10 = 1 \cdot 2^3 + 1 \cdot 2^1$.
Now "hereditarily" means that also the exponents should be written in base 2 (and, if necessary, also the exponents of the exponents, etc.). Thus:
$10 = 1 \cdot 2^{1 \cdot 2^{1 \cdot 2^0} + 1 \cdot 2^0} + 1 \cdot 2^{1 \cdot 2^0}$.
Now we're supposed to replace the base 2 by 3, i.e.:
$1 \cdot 3^{1 \cdot 3^{1 \cdot 3^0} + 1 \cdot 3^0} + 1 \cdot 3^{1 \cdot 3^0}$.
This is equal to 84, so $a_2 = 83$. Let's repeat this once more, but for simplicity, let me omit all these 3^0 (after all, they'll stay equal to 1 anyway).
$a_2 = 1 \cdot 3^{1 \cdot 3^1 + 1} + 2$
so $a_3 = (1 \cdot 4^{1 \cdot 4^1 + 1} + 2) - 1 = 1025$

As one can see, the sequence grows quite rapidly. However, Goodstein proved:

Theorem 4.10.4. *For any n, the Goodstein sequence starting with n will eventually reach 0.*

This theorem cannot be proven from the axioms of PA. (Intuitively, its proof requires an inductive argument which is more complicated than what can be done using (IND).)

And to end this lecture...

...let me mention a puzzle (invented by Raymond Smullyan):

Recall that on the Island of Knights and Knaves, the Knights always tell the truth and the Knaves always lie.

Mr. L. has some disease and goes to a doctor. The doctor tells him: "If you believe that you will be cured, then you will be cured." Even though doctors always tell the truth, Mr. L. doesn't know what to do with this information. A bit later, L. goes on holiday to the Island of Knights and Knaves, where he visits a shaman. The shaman (who might be a Knight or a Knave) says: "If you believe that I'm a knight, you will be cured."

When Mr. L. hears this, he's very happy.

Exercise 4.10.5. Why? And what does this have to do with Theorem 4.10.1?

THE END