Computability Theory: A Very Short Introduction

Amir Tabatabai
Faculty of Humanities, Utrecht University

February 2021

1 Introuction

Let $a, b, c \in \mathbb{Z}$ be three given integers. Is there any *finitistic method*[1] to decide whether the quadratic equation $ax^2 + bx + c = 0$ has an integer solution? Although the phrase *finitistic method* is vague and open to different interpretations, we can all agree that the answer to this question is positive. The following is a high-level explanation of *a finitistic method* to solve the problem. It is based on the explicit formula for the roots of the quadratic equations:

> *Compute $b^2 - 4ac$. Check whether it is a perfect square or not.*[2] *If not, the equation does not have an integer root. If yes, call it δ and check whether $2a$ divides $-b + \delta$ or $-b - \delta$. If yes, the equation has an integer root, if not, it doesn't.*

The same finitistic method also works for the polynomials with the degree three and four. The reason simply is that these equations are solvable by

[1] We emphasize on being finitistic because we expect our methods to be reasonable for us as finite mortal human beings. For instance, one way to answer the problem is checking all possible numbers which is clearly not an option for us.

[2] There are so many *finitistic methods* to check wether a given number x is a perfect square or not and if yes, computes its square root. Some of these finitistic methods are significantly effective and some extremely slow and impractical. For instance, the following is an example of the latter case: Compute the squares of all the numbers less than or equal to x and compare them with x. If x has an integer square root, the finitistic method finally finds it because the root, whatever it is, stands below the number x itself. There are also more reasonable approaches. One of them is based on some kind of recursion. You can find the recursion yourself. Begin with the situation in which x is even and then ask yourself that what is the relationship between $\sqrt{x}$ and $\sqrt{x/4}$.

the four basic operations and the radicals with degrees two, three and four, respectively.

Now let us go one level further to investigate the same problem for any polynomial with one variable $p(x)$ with degree five. As the nineteen century mathematics has already shown, when it comes to the equations with higher degrees than four, there can not be an explicit root formula consisting of the four basic operations and the radicals. Therefore, our previous finitistic method breaks down. However, it is clear that this failure does not mean that we can not find another *finitistic method* to solve our problem. In fact, there exists a uniform finitistic method to handle all polynomials with one variable with any degree. To explain how, we first need an observation. Let us assume that $a \in \mathbb{Z}$ is an integer root for the polynomial $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$ with integer coefficients. Then, it is clear that there exist a polynomial $q(x)$ such that $p(x) = (x-a)q(x)$, where $q(x) = b_{n-1}x^{n-1} + \ldots + b_0$ and $b_i \in \mathbb{Z}$ for all $i \leq n-1$. Therefore, $a_0 = ab_0$, meaning that $a | a_0$. This simple observation leads to the following *finitistic method*:

> *By checking all the numbers less than or equal to a_0, find all the divisors of a_0 and then check whether any of them is a root of $p(x)$ or not.*

This procedure is a uniform finitistic method applicable to any polynomial with any degree including the quadratics, the cubics and the quartics that we addressed before. But note that this uniformity comes at a price. The radical-based approach is more *effective* and far faster than this *brute force* approach of checking all the possible divisors. The reason is simple. The former has some access to an explicit root formula that only needs checking some *easy properties* rather than *searching for some bounded numbers with some easy properties.*

Now we can go even one step further. What do we know about the polynomials with two variables? For quadratics the answer is again positive. This time the idea is more sophisticated than what we had for the one variable case and unfortunately we do not have enough space to explain even the basics. For the degree three, the problem gets already beyond our full understanding. Here, even if we have access to all possible mathematical methods, including infinitary ones, we are far from the full understanding of the behavior of the zeros of these cubic polynomials with two variables. You can consult the *theory of elliptic curves* to see how complex the situation can be. As we can observe, by targeting the higher degrees and/or higher number of variables, the problem becomes harder and harder. Therefore, it would be reasonable to conjecture that after some point there should not be a finitistic method to handle the polynomials. And certainly, there should not be a unified finitistic method to handle all the polynomials with any number of variables in any possible degree. This is our conjectured answer to Hilbert's tenth problem.[3] The problem reads:

> *"Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers."*

So far, we have had a conjecture that the answer to Hilbert's tenth problem should be negative. But how to prove such a negative statement? How to show that a given decision problem does not have an acceptable deciding finitistic method? The first clear step is finding a precise definition of what we mean by a *finitistic method*.[4]

Let us start with inspecting our basic expectations from any formalization of a finitistic method. The first point as the reader may expect is the *finitieness* condition. The formalization, whatever it will be, should make a difference between the finitistic methods and some infinitary ones. For instance, one infinitary method to solve Hilbert's tenth problem is checking all the integers in the equation $p(\vec{x}) = 0$. But this can not be considered as an acceptable method, since it is not possible for us, as human beings, to follow such an infinitary method. Therefore, one reasonable condition on a finitistic method is that *it must take place in finite time.* However, this finitness condition depends on how we actually count the needed time. If the definition is not well-refined enough, then it can consider the whole checking process of all the integers as an instant atomic act, and hence concludes that the

[3] In his lecture in the second international congress of mathematics in Paris in 1900, Hilbert proposed twenty three problems that shaped the whole skeleton of the twentieth century mathematics. Three of these problems had a foundational flavor. The first on the continuum hypothesis, the second on the consistency of arithmetic and the tenth on the Diophantine equations.

[4] The answer came in 1936 in four different flavors developed by four different mathematicians, Alonzo Church, Alan Turing, Stephen Cole Kleene and Emil Post. Here we will follow Turing's formalization and his argument why we have to accept his formalization as a complete characterization of what we, humans, can perform. This completeness claim is usually called the *Church-Turing thesis*. It roughly states that whatever is computable intuitively, is also computable by a Turing machine that is Turing's formalization of a computational model.

needed time is just finite. Here, we see the second important ingredient: *The finitistic method should consist of very basic operations that we can consider as the atomic steps of our methods.* The third condition is that these *basic operations must be meaningless and mechanical operations on a finite set of symbols* to avoid using any hidden information encoded inside of the symbols. For instance, if we use a function symbol f with the hidden interpretation that f reads a polynomial and decides whether it has an integer root or not, then it is clear that using such f is not acceptable in any finitistic solution for Hilbert's tenth problem.

Now, the only question is that what are these basic meaningless operations that we can do on a piece of paper by some given symbols. There are only four of them and it is totally reasonable to believe that these operations exhaust all possible basic operations. The first three operations are reading, erasing and writing the symbols, while the fourth is moving over the paper to find the other symbols. Note that by moving, we mean moving step by step, following what the rules dictate. Therefore, everything happens locally and there is no jump from one point in the paper to some other point. Following this argument, we can informally define a finitistic method or as we will call it a *computation* as *a method that is implementable in finite time via the local operations of reading, writing and erasing the symbols on a piece of paper and moving throughout the paper, all governed by a given finite set of rules.* This is essentially what Turing defined as his well-known *Turing machine*. We will use the words Turing machine and/or algorithm informally in the rest of this section to refer to such mechanical machines.

Now, we have a formalization of the *finitistic method* as what is performable by a Turing machine. But how to prove that there is no computable way to check the existence of the integer roots? This is not an easy problem to solve. Therefore, it may be reasonable to generalize the problem to first develop a reasonable theory around the notion of computablility and second to have harder problems that makes the impossibility results easier.

For this purpose, let S be a set and $L \subseteq S$ be one of its subsets. The decision problem of L is checking whether a given element $x \in S$ is in L or not. For Hilbert's tenth problem, the set S is the set of all polynomials with integer coeffiecients and L is the set of all such polynomials that have integer roots. For another example, S can be the set of all mathematical statements and $L = L_{Truth}$ can be the set of all *true* mathematical statements. In this case, we are looking for a finitistic method to read a mathematical statement and check wether it is true or not. Here there is a very insightful observation.

Reading this new problem, we may not be able to solve it in an absolute way, but we can at least locate it by comparing its difficulty with a problem that we already know it is hard. The idea is as follows: Assume that you have a finitistic method for the harder problem and show it leads to a finitistic method for the easier one. For instance, as we can easily see, any finitistic method to solve L_{Truth} leads to a decision finitistic method for Hilbert's tenth problem. (Just check the truth of the statement $\exists \vec{x} \in \mathbb{Z}\ [p(\vec{x}) = 0]$.) Therefore, this problem is even harder than the Hilbert's tenth problem and hence it would be reasonable to assume that this problem is not solvable by the finitistic methods.

One other example may help. Let S be the set of all mathematical statements again and let $L = L_{TAUT}$ be the set of all logical tautologies. This set seems easier to decide than what we saw before. After all, being a tautology is admittedly a syntactical property, checkable by looking into the syntactical form of the statement. Surprisingly, in reality, this problem is not even easier but actually harder than the previous examples. Here is the reason. Assume that L_{TAUT} is solvable by a *finitistic method*. Then we will show how to solve the decision problem of L_{Truth} using that finitistic method. Pick M as a finite set of axioms, axiomatizing the whole mathematics. Then, A is a true mathematical statement iff it is provable in M. But the latter is equivalent to the statement that "$\bigwedge M \to A$" is a logical tautology. Therefore, you can use the finitistic method for L_{TAUT} on $\bigwedge M \to A$ to decide if $A \in L_{Truth}$.[5]

The last example is one from the computability theory itself. First, note that Turing machines, may behave in a strange way on some of their inputs. For instance, a reasonable finitistic method to find the least possible number with some property is checking the property first for zero, then for one, two and so on till finding the first number with the property. If there is a number with that property, we can find the least possible one. However, if there is no number with the property, then the algorithm does not understand it and goes on forever. As this algorithm shows, we expect Turing machines to halt

[5]The decision problem for L_{TAUT}, called Entsheidungsproblem, is one of the famous foundational problems of the last century. It has been proposed by David Hilbert and motivated, among many other things, the emergence of the whole theory of computation. Hilbert himself thaugt the answer to this problem should be positive and hence it may lead to a *finitistic method* to decide the truth of any statement in the whole mathematics. Exactly like his tenth problem, here there were also some special cases for which Hilbert's school found some decision finitistic methods. They tried to find a finitistic method for the general problem but it remained unsolved. After 1931 and under the influence of Gödel's first incompleteness theorem, that positive hope declined dramatically.

on some inputs and not halt on the others. Let S be the set of all pairs (M, x), where M is a Turing machine and x is an input. Define L_{Halt} as the set of such pairs that the machine M halts on the input x. Deciding such a problem is crucially important for a programmer. It would be very convenient to have a mechanical finitistic method to check whether a machine halts on a given input without running the actual program, as it is possible that the not-halting case happens as an unintended bug in the design of the algorithm. We will show that the problem L_{Halt} is easier than L_{Truth}. Assume we have a finitistic method to decide the truth of a mathematical statement. Then, pick the statement "The Turing machine M halts on the input x." Its truth is nothing but halting condition and hence we can decide the halting problem.

So far, we have stated a list of decision problems and some natural negative conjectures about them. Now, it is time to explain the main idea behind the proof of undecidability of these problems. First, let us begin with the halting problem. The idea is the usual self-referential argument you may have seen in Russell's or the liar paradox. The moral is that our notion of computation is so much powerful that it can talk about itself and this leads to self-referential and paradoxical behavior. The reason simply is that Turing machines are finite syntactical objects and we can encode them via natural numbers which makes Turing machines powerful enough to even run on themselves as their inputs. Let us explain it a bit more. Assume that there exists a Turing machine $H(M, x)$ that decides whether the Turing machine M halts on the input x or not, i.e., if M halts on x, we have $H(M, x) = 1$ and otherwise, $H(M, x) = 0$. Let's denote the code for the machine M by m. Then, think about the machine N as the machine that first computes $M(x, x)$ and then if it is one, it does not halt by doing something unboundedly and if it is zero, it halts and outputs zero. Now, let us check $N(n)$, where n is the code of N. If N halts on n, then by definition of N, we have $M(n, n) = 0$ which means that N does not halt on n. If N does not halt on n, then again by definition $M(n, n) = 1$ which means N halts on n. You can recognize Russell's paradox. Right? The algorithm N talks about its halting behavior in a negative way.

Now, following the way we located the problems before, we know that the halting is easier than the truth problem and it is easier than the Entsheidungsproblem. Therefore, by proving that the halting problem is undecidable, we actually proved that all these three problems are undecidable, as well. But what about the tenth problem? This problem is easier than the truth problem which does not imply its undecidability. In fact, the tenth problem is also undecidable. However, its proof is extremely harder than what we saw for halting and the others. Its undecidability proved by Matiasevich, Davis, Putnam and Robinson by showing that the halting problem is equivalent to the existence of integer roots for a specific polynomial with integer coefficients. We will mention this problem later in the lectures.

2 Deterministic Finite Automata

As we have explained in Introduction, computation is a mechanical manipulation of symbols via the local operations of reading, writing and erasing, all governed by a finite set of given rules. In this section we limit ourselves to a very restricted notion of computation in which we are only allowed to read the input without writing and erasing any symbol anywhere. One may hesitantly ask that even if some kind of computation is possible in such a limited situation, how the machine is supposed to report the result of the computation without the minimum power of writing. Because of this very reason, in this section, the computation is limited only to the decision procedures through which the machine computes the truth value of the input statement. The mechanism is as follows: The machine has finitely many internal states that change throughout the computation process. Some of these states are called the accepting states. The machine reads the input, letter by letter, and in each step modifies its current state according to what it just has read and what its rules dictate. The whole process ends when the machine reads the last letter of the input and it accepts the input if it lands in a final state.

Definition 2.1. Let Σ be a finite set. By Σ^* we mean the set of all finite strings (including the zero-length string ϵ) consisting of the elements of Σ.

Definition 2.2. By a language over the alphabet Σ we mean any subset of the set Σ^*.

For instance, the English language is a language over the set $\{a, b, c, \dots, z\}$ while natural numbers can be considered as a language on the binary set $\{0, 1\}$ via the binary expansion or the unary set $\{1\}$ via identifying n by 1^n as a sequence of n many 1's. The other examples include the set $\{a^n b^n \mid n \geq 0\}$ over the alphabets $\{a, b\}$ or the set $\{1^p \mid p \text{ is a prime number}\}$ over the alphabet $\{1\}$.

Definition 2.3. A deterministic finite automaton, DFA, M on the alphabet $\Sigma = \{s_1, \dots, s_n\}$ with states $Q = \{q_1, \dots, q_m\}$ is given by a function $\delta : Q \times \Sigma \to Q$ which maps each pair (q_i, s_j), $1 \leq i \leq m$, $1 \leq j \leq n$, into a state q_k, together with a set $F \subseteq Q$. One of the states, usually q_1, is singled out and called the initial state. The states belonging to the set F are called the final or accepting states. δ is called the transition function.

It is usually useful to represent a DFA by a directed graph with the nodes representing the states and edges with the alphabet labels to represent the transition function. If the transition function sends p to q reading the letter s, there will be an edge from p to q with label s. The start state is shown by a node with a small unlabelled in-edge and the accepting states are indicated by double circles.

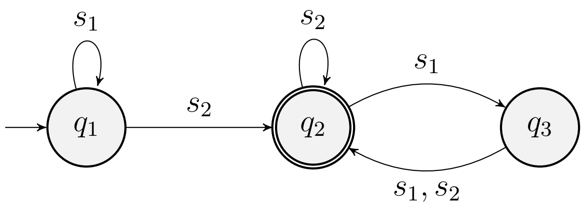


Figure 1: A DFA with the states $\{q_1, q_2, q_3\}$, alphabet $\{s_1, s_2\}$ and $F = \{q_2\}$.

Definition 2.4. Let $q \in Q$ be a state and $w \in \Sigma^*$ be a finite string. By $\delta^*(q, w)$ we mean the state resulting from iterating δ starting from q and reading the letters of w from left to right till it ends. Then M accepts a word w if $\delta^*(q_1, w) \in F$. It rejects w otherwise. Finally, the language accepted (recognized) by M, written $L(M)$, is the set of all $w \in \Sigma^*$ accepted by M, i.e.,:

$$L(M) = \{w \in \Sigma^* \mid \delta^*(q_1, w) \in F\}.$$

A language is called *regular* if there exists a deterministic finite automaton which accepts it.

Notation. From now on, in the examples, we denote the states by p, q, r and so on. The letter p is always reserved for the initial state. For the alphabet letters we usually use a, b, c and so on. We also use 0 and 1 later in the course.

Example 2.5. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q\}$ such that $F = \{p\}$. Define δ as p for the input (p, a) and q otherwise. Then $L(M) = \{a^n \mid n \geq 0\}$. See Figure 2. The reasoning is simple. If M reads a^n, for some n, (including $n = 0$), it goes through the loop over p, n many times. Therefore, it ends up in p and hence M accepts it. But if it includes at least one b, the machine must cross the edge between p and q at some point and after that it will remain in q till the end. Hence, M cannot accept these strings.

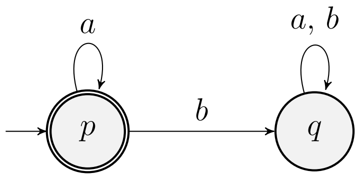


Figure 2: Example 2.5

2.1 Some Examples

Example 2.6. Consider the DFA M on the alphabet $\{a, b\}$ with the state $\{p\}$ such that $F = \emptyset$. Define δ as p for the both inputs (p, a) and (p, b). Then $L(M) = \emptyset$. See Figure 3.

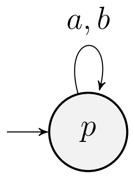


Figure 3: Example 2.6

Example 2.7. Consider the DFA M on the alphabet $\{a, b\}$ with the state $\{p\}$ such that $F = \{p\}$. Define δ as p for the both inputs (p, a) and (p, b). Then $L(M) = \{a, b\}^*$. See Figure 4.

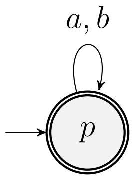


Figure 4: Example 2.7

Example 2.8. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r\}$ such that $F = \{q\}$. Define δ as q for the input (p, a) and r otherwise. Then $L(M) = \{a\}$. See Figure 5.

Example 2.9. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r, t\}$ such that $F = \{r\}$. Define δ as q for the input (p, a) and r for the input (q, b) and t otherwise. Then $L(M) = \{ab\}$. See Figure 21.

Exercise 2.10. Let $u \in \{a, b\}^*$. Define a DFA M such that $L(M) = \{u\}$.

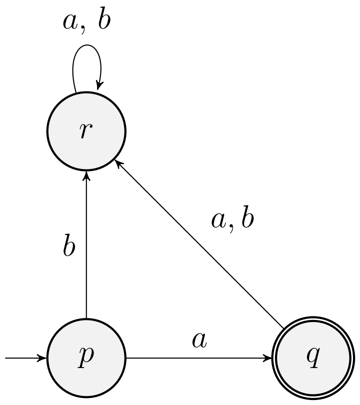


Figure 5: Example 2.8

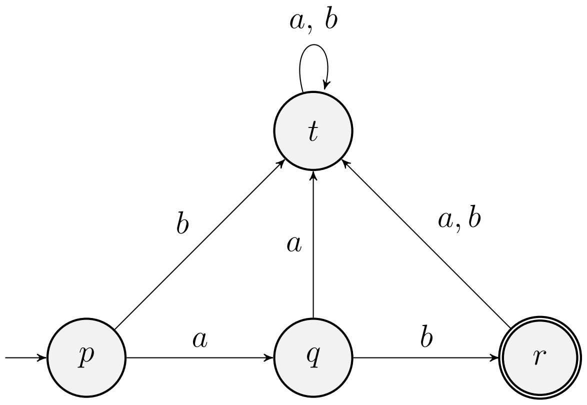


Figure 6: Example 11.29

Example 2.11. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r, t\}$ such that $F = \{q, r\}$. Define δ as q for (p, a); r for (p, b) and t otherwise. Then $L(M) = \{a, b\}$. See Figure 7.

Example 2.12. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r\}$ such that $F = \{r\}$. Define δ as q for the input (p, b), r for the input (p, a), q for both (q, a) and (q, b) and finally the same for r, meaning r for both (r, a) and (r, b). Then $L(M) = \{aw \mid w \in \{a, b\}^*\}$. See figure 22.

Exercise 2.13. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r, t\}$ such that $F = \{r\}$. Define δ as q for the input (p, a); r for the input (q, b); r for both (r, a) and (r, b) and t for the rest. Then, show that $L(M) = \{abw \mid w \in \{a, b\}^*\}$.

Exercise 2.14. Let $u \in \{a, b\}^*$. Define a DFA M such that $L(M) = \{uw \mid w \in \{a, b\}^*\}$.

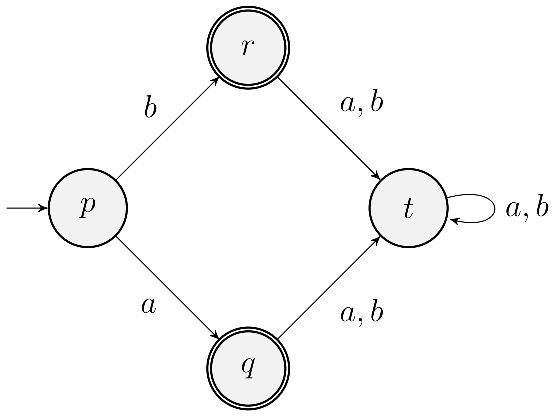


Figure 7: Example 2.11

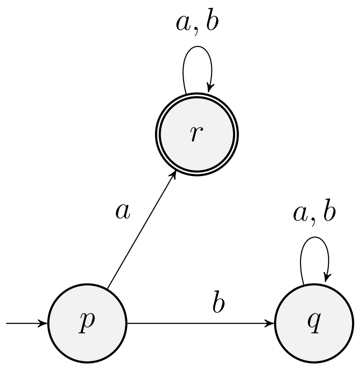


Figure 8: Example 11.31

Example 2.15. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q\}$ such that $F = \{q\}$. Define δ as q for the input (p, a) and (q, a) and p otherwise. Then $L(M) = \{wa \mid w \in \{a, b\}^*\}$. See Figure 9.

Exercise 2.16. Define a DFA M such that $L(M) = \{wba \mid w \in \{a, b\}^*\}$.

Exercise 2.17. Let $u \in \{a, b\}^*$. Define a DFA M such that $L(M) = \{wu \mid w \in \{a, b\}^*\}$.

Example 2.18. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q\}$ such that $F = \{q\}$. Define δ as p for the input (p, a) and q otherwise. Then $L(M) = \{w \mid w \text{ has at least one } b\}$. See Figure 10.

Exercise 2.19. Compare Examples 2.5 and 2.18 and answer the following question: Let L be a regular language over the alphabet Σ. Is $\Sigma^* - L$ also regular?

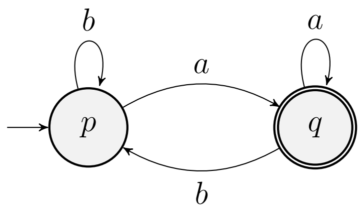


Figure 9: Example 5.11

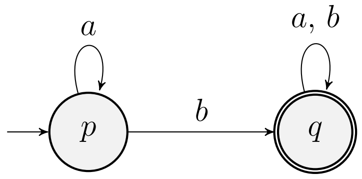


Figure 10: Example 2.18

Exercise 2.20. Describe a DFA M that recognizes the strings over $\{a, b\}$ with at least two a's. What about the condition "at most one a"?

Example 2.21. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r, t\}$ such that $F = \{r\}$. Define δ as q for the input (p, a); r for (q, b) and (r, b); q for (q, a) and t otherwise. Then $L(M) = \{a^n b^m \mid m, n \geq 1\}$. See Figure 11.

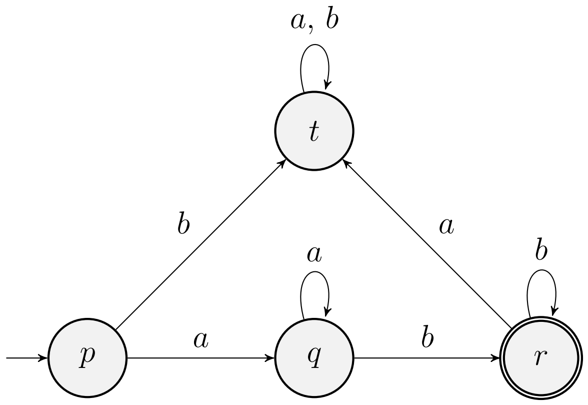


Figure 11: Example 2.21

Example 2.22. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r\}$ such that $F = \{p\}$. Define δ as p for the input (q, b); q for (p, a) and r otherwise. Then $L(M) = \{(ab)^n \mid n \geq 0\}$. See Figure 12.

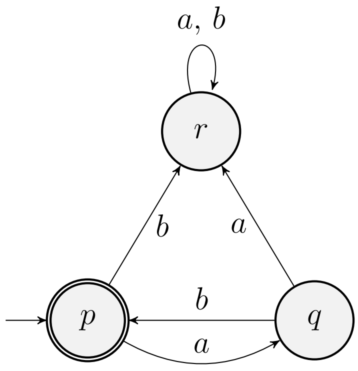


Figure 12: Example 2.22

Exercise 2.23. Let $u \in \{a, b\}^*$. Define a DFA M such that $L(M) = \{u^n \mid n \geq 0\}$.

Example 2.24. Consider the DFA M on the alphabet $\{1\}$ with the states $\{p, q, r\}$ such that $F = \{p\}$. Define δ as q for the input $(p, 1)$ and r for $(q, 1)$ and p for $(r, 1)$. Then $L(M) = \{1^{3n} \mid n \geq 0\}$. Interpreting $\{1\}^*$ as $\mathbb{N}$, the machine recognizes the multiples of three. See Figure 13.

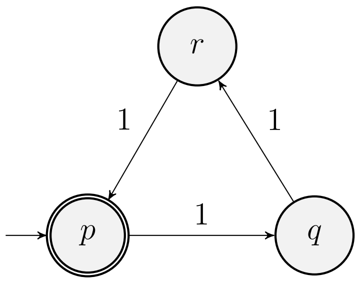


Figure 13: Example 2.24

Exercise 2.25. Define a DFA M on the alphabet $\{1\}$ such that $L(M) = \{1^{3n+1} \mid n \geq 0\}$.

Exercise 2.26. Let m and r be two fixed natural numbers such that $m \geq 1$ and $r < m$. Define a DFA M on the alphabet $\{1\}$ such that $L(M) = \{1^{mn+r} \mid n \geq 0\}$.

3 Non-deterministic Automata and Regular Operations

In this section we will introduce a generalization of a DFA called a non-deterministic finite automaton or NFA, for short. These models may seem more powerful than the DFAs but they are actually equivalent to them and accept the same languages as the DFAs do. Their advantage, though, lies in the fact that the NFAs are easier and more well-behaved to use. For instance, the class of the new machines is closed under some simple gluing operations that help us to construct more regular languages from the old. We can simulate these operations over the DFAs, as well, but then they are not as transparent as they were for NFAs.

Definition 3.1. A non-deterministic finite automaton, NFA, M on the alphabet $\Sigma = \{s_1, \dots, s_n\}$ with states $Q = \{q_1, \dots, q_m\}$ is given by a transition relation $\delta \subseteq Q \times \Sigma \times Q$, together with a set of accepting states $F \subseteq Q$ and the initial state q_1.

Remark 3.2. Note that in a run of an NFA, after reading a letter from the input, the machine may have many possible next states (including zero possibilities), leading to many possible futures. This is why it is called non-deterministic.

Definition 3.3. Let $q \in Q$ be a state and $w \in \Sigma^*$ be a string. By $\delta^*(q, w)$ we mean the *set of all states* resulting from any iteration of δ starting from q and reading the letters of w from left to right till they end. Then M accepts a word w if $\delta^*(q_1, w) \cap F \neq \emptyset$. In other words, it accepts w if there *exists a path of states* following the relation δ and the letters of w from q_1 to F. Finally, the language accepted by M, written $L(M)$, is:

$$L(M) = \{w \in \Sigma^* \mid \delta^*(q_1, w) \cap F \neq \emptyset\}.$$

3.1 Some Examples

Example 3.4. Consider the NFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r\}$ such that $F = \{r\}$. Define δ as the following relation: $\{(p, a, q), (q, b, r)\}$. Then $L(M) = \{ab\}$. See Figure 14.

Exercise 3.5. Let $u \in \Sigma^*$. Define an NFA M such that $L(M) = \{u\}$.

Example 3.6. Consider the NFA M on the alphabet $\{a, b, c\}$ with the states $\{p, q, r, t\}$ such that $F = \{t\}$. Define δ as the following relation:

$$\{(p, a, q), (p, a, r), (q, b, t), (r, c, t)\}.$$

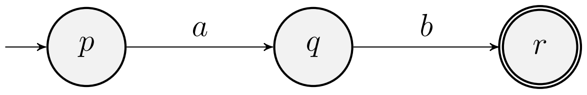


Figure 14: Example 3.4

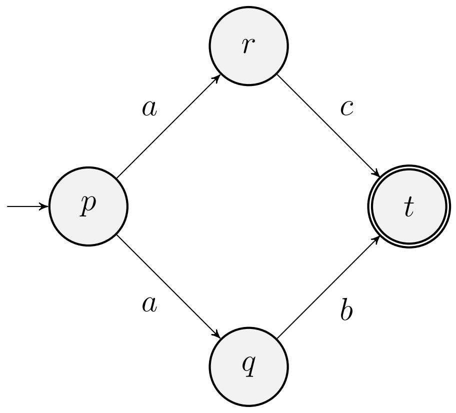


Figure 15: Example 3.6

Then $L(M) = \{ab, ac\}$. See Figure 15.

Exercise 3.7. Let $u_1, \dots, u_k \in \Sigma^*$. Define an NFA M such that $L(M) = \{u_1, \dots, u_k\}$.

Exercise 3.8. Let M and N be two NFAs that recognize the languages L_1 and L_2, respectively. Is there an NFA to recognize the language $L_1 \cup L_2$?

Example 3.9. Consider the DFA M on the alphabet $\{a\}$ with the states $\{p, q\}$ such that $F = \{q\}$. Define δ as the following relation: $\{(p, a, q), (q, a, q)\}$. Then $L(M) = \{a^n \mid n \geq 1\}$. See Figure 16.

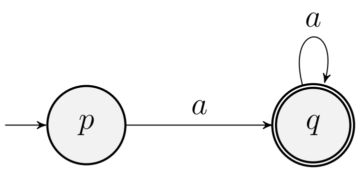


Figure 16: Example 3.9

Now we will use M to construct the NFA N on the alphabet $\{a, b\}$ with the states $\{p, q, q', r', t'\}$ such that $F = \{t'\}$. Define δ as the following relation:

$$\{(p, a, q), (q, a, q)\} \cup \{(q, b, q'), (q', a, t'), (q, a, r'), (r', b, t')\}.$$

Then $L(M) = \{a^n ba \mid n \geq 1\} \cup \{a^n ab \mid n \geq 1\}$. See Figure 17. Note that this machine is the result of gluing the previous machine M to a machine similar to the machine of Example 3.6, in the node $q = p'$.

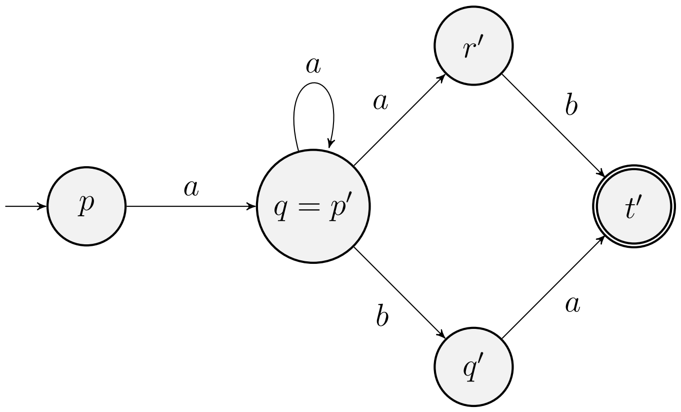


Figure 17: Example 3.9

Exercise 3.10. Let M and N be two NFAs that recognize the languages L_1 and L_2, respectively. Is there an NFA to recognize the language $L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}$?

Example 3.11. Consider the NFA M on the alphabet $\{a, b, c\}$ with the states $\{p, q, r\}$ such that $F = \{p\}$. Define δ as the following relation:

$$\{(p, a, q), (q, b, p), (p, a, r), (r, c, p)\}.$$

Then $L(M) = \{w_1 w_2 \dots w_n \mid w_i \in \{ab, ac\}, n \geq 0\}$. See Figure 18. Note that this machine is the result of gluing the initial and the final states of the machine of Example 3.6.

Exercise 3.12. Let M be an NFA that recognizes the language L. Is there an NFA to recognize the language $L^* = \{w_1 w_2 \dots w_n \mid n \geq 0, w_i \in L\}$?

3.2 The Equivalence Theorem

Theorem 3.13. *A language is accepted by an NFA if and only if it is regular. Equivalently, a language is accepted by an NFA if and only if it is accepted by a DFA.*

Proof. Obviously, any language accepted by a DFA is also accepted by an NFA. Conversely, let $L = L(M)$, where M is an NFA with the transition relation δ, set of states $Q = \{q_1, \dots, q_m\}$, and the set of final states F. We will

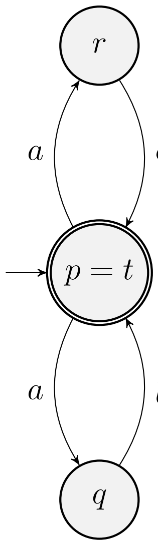


Figure 18: Example 3.11

construct a DFA N such that $L(N) = L(M)$. The idea is that the individual states of N will be the subsets of M denoted by $Q' = \{Q_1, Q_2, \ldots, Q_{2^m}\}$, where in particular $Q_1 = \{q_1\}$ is to be the initial state of M. The set G of final states of N is given by:

$$G = \{Q_i \in Q' \mid Q_i \cap F \neq \emptyset\}$$

The transition function σ of N is then defined by:

$$\sigma(Q_i, s) = \{r \in Q \mid \exists q \in Q_i \; \delta(q, s, r)\}$$

It is easy to see that N accepts exactly the strings for which there exists an accepting path in M. Hence, $L(M) = L(N)$. $\square$

Remark 3.14. Simulating the parallel nature of non-determinism by some deterministic moves usually comes at a huge price. For instance, note that in the previous theorem, the deterministic simulator is exponentially bigger than the original non-deterministic machine.

3.3 Regular Operations

In this section we will explain some high-level methods to construct different regular languages, without the need to describe the actual machine every time. In fact, we will present a descriptive criterion that characterizes all regular languages based on their common description form.

Lemma 3.15. *For any NFA M, there exists an equivalent non-restarting NFA N. Non-restarting means that the machine has no edge entering the start state i.e., there is no pair $(q, s) \in Q \times \Sigma$ such that $\delta(q, s, q_1)$.*

Proof. Add one new state q to M as the new start state and add an edge from q to r with label $s \in \Sigma$ if there is already an edge from the starting state of M to r with the same label. Finally, make the state q accepting iff the initial state of M was accepting. It is easy to see that N is non-restarting and equivalent to M, i.e., $L(M) = L(N)$. □

Theorem 3.16. *Let L_1 and L_2 be regular languages. Then all of the languages $L_1 \cup L_2$, $L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}$ and $L_1^* = \{w_1 w_2 \dots w_n \mid w_i \in L_1, n \geq 0\}$ are regular.*

Proof. Let M and N be two DFAs accepting the languages L_1 and L_2, respectively. To show the mentioned languages are regular, we describe the construction of an accepting NFA K for each case. Checking that these machines work is easy and left to the reader.

- For $L_1 \cup L_2$, put M above N, add one new start state q mimicking the out edges of both start states of M and N. Note that this new machine can be non-deterministic.
- For $L_1 \cdot L_2$, put N after M, connecting all final states of M to some states of N mimicking the in-edges of the start state of N.
- For L_1^*, use M but for any edge with the label s from q to r, where r is a final state in M, draw an edge with the label s from q to the initial state of M and make the initial state of M also final (if it is not final already). For this machine to work, M should be non-restarting and we can assume that condition for M, thanks to Lemma 3.15.

□

Exercise 3.17. In the third part of the proof of Theorem 3.16, for the closure under star, run the algorithm on a restarting M (i.e., there is a pair $(q, s) \in Q \times \Sigma$ such that $\delta(q, s) = q_1$). Show by an example that the resulting automaton does not recognize L_1^*.

Definition 3.18. Any combination of the symbols $\{\cup, \cdot, (-)^*\}$ on the elements of Σ is called a regular expression. By the language of a regular expression E, we mean the language defined recursively by interpreting the element $s \in \Sigma$ as the singleton language $\{s\}$ and the symbols in $\{\cup, \cdot, (-)^*\}$ by their corresponding operations. If L is the language of the regular expression E, we say that E is a description of L (or L is described by E).

Example 3.19. Let $\Sigma = \{a, b\}$. Using the convention of omitting the points in concatenations, the regular expressions a^*, $(aa^*)(bb^*)$ and $(ab)^*$ are descriptions of the regular languages $\{a^n \mid n \geq 0\}$, $\{a^n b^m \mid m, n \geq 1\}$ and $\{(ab)^n \mid n \geq 0\}$, respectively.

Note that any regular expression describes a regular language by Theorem 3.16 and Exercise 3.5.

Exercise 3.20. For each language L described in the exercises 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, give a description by a regular expression.

Exercise 3.21. Let $L = \{w \in \{a, b\}^* \mid w \neq \epsilon \text{ and } bb \text{ is not a substring of } w\}$.

(i) Find a regular expression describing L.

(ii) Show that L is regular by constructing an NFA that accepts L.

The next theorem provides the converse of the previous statement. It can be read as a connection between the computability via certain models and being described by a certain form.

Theorem 3.22. *(Kleene) A language is regular iff it has a description by a regular expression.*

Proof. One side is proved. For the converse, we will explain the idea behind the proof. First, we have to introduce a generalized version of NFAs called GNFAs, with the same structure but this time using regular expressions as the labels of the edges. Acceptance in these machines is defined in a natural way. The machine reads the input, but not letter by letter. If it is in the state q and read the input up to w_i in $w = w_1 w_2 \dots w_n$, it can read any segment onwards such as $w_{i+1} \dots w_j$ and check if this segment is in the language of the label of an eadge from q to some other state, say q'. If yes, it can go to q' and jump on the input to w_{j+1}. It is clear that the usual DFAs are the special case of these machines. There are two things to prove. First, one should prove that these generalized machines only accept regular languages and then to show that any GNFA is equivalent (accepting the same language) to a smaller GNFA. This procedure reduces a usual DFA (read as a GNFA) to a single regular expression which completes the proof. $\square$

4 Pumping Lemma

Consider the language $L = \{a^n b^n \mid n \geq 0\}$ over the alphabets $\{a, b\}$. Is L regular? Let us assume its regularity for a moment and try to construct a

DFA to accept it. How should this machine work? The natural candidate for the algorithm is the following: First read all the a's till you reach the first b and remember the numbers of a's that you have read. Then do the same for b's, meaning that you have to read all the block of b's and remember their number again. Then, if you read an a again, reject, otherwise check whether the number of a's and b's that you have read so far are equal or not. The problem with this very natural algorithm is that it goes far beyond the power of the DFAs. The reason is simple: DFAs are not allowed to write and hence they do not have a reasonable memory (except maybe for a fixed finite amount of data, encoded through the finite states of the machine) and hence they cannot remember the variable number of a's or b's that they have read in the algorithm. This lack of memory limits the power of DFAs dramatically. In this section we will explain how to use this weakness of DFAs to show that a given language is not regular. The main idea is the following: If you have a bounded memory, you will repeat yourself, eventually.

Theorem 4.1. *(Pumping Lemma). Let $L = L(M)$, where M is a DFA with n states. Let $x \in L$, where $|x| \geq n$. Then we can write $x = uvw$, where $|v| \neq 0$, $|uv| < n$ and $uv^iw \in L$ for all $i \geq 0$.*

Proof. Since x consists of at least n symbols, x must go through at least n state transitions as it scans x. Including the initial state, this requires atleast $n+1$ (not necessarily distinct) states. But since there are only n states in all, we conclude that M must be in at least one state more than once. Let q be the first state in which M finds itself at least twice. Then we can write $x = uvw$, where $\delta^*(q_1, u) = q$, $\delta^*(q, v) = q$, $\delta^*(q, w) \in F$. That is, M arrives in state q for the first time after scanning the last (right-hand) symbol of u and then again after scanning the last symbol of v. Since this "loop" can be repeated any number of times, it is clear that $\delta^*(q_1, uv^iw) = \delta^*(q_1, uvw) \in F$. Hence $uv^iw \in L$. Moreover, since q is the first repeated state, we have $|uv| < n$. $\square$

Example 4.2. The language $L = \{a^nb^n \mid n \geq 0\}$ on the alphabet $\{a, b\}$ is not regular. Assume otherwise. Then there exists a DFA M such that $L(M) = L$. Set $|Q_M| = n$. Then by the pupming lemma, since the length of $x = a^nb^n$ is bigger than n, it has a partition uvw such that $|uv| < n$. Therefore, v only consists of a's and since $|v| \neq 0$, the number of a's and b's in uv^2w cannot be equal which means $uv^2w \notin L$. The same argument also works for the language $L = \{x \in \{a, b\}^* \mid N_a(x) = N_b(x)\}$, where $N_a(x)$ and $N_b(x)$ are the number of a's and b's in x.

Exercise 4.3. Show that the language $L = \{ww^R \in| \ w \in \{a, b\}^*\}$ is not regular, where w^R is the reverse of w, i.e., the word w written from right to left. For instance, $(abb)^R = bba$.

Example 4.4. The language $L = \{1^p \mid p \text{ is a prime number}\}$ on the alphabet $\{1\}$ is not regular. Assume otherwise. Then there exists a DFA M such that $L(M) = L$. Set $|Q_M| = n$ and pick $p \geq n$. Then by the pupming lemma, since the length of $x = 1^p$ is bigger than or equal to n, it has a partition uvw such that $|v| \neq 0$. Since the alphabet consists only of one element, any string can be identified with its length. Set $|u| = a$, $|v| = b$ and $|w| = c$. Therefore, $a + ib + c$ for any $i \geq 0$ is prime. Put $i = 0$, then $a + c$ is prime. Now put $i = a + c$, then $a + ib + c = (a + c)(1 + b)$. We know that $1 + b \geq 2$ and $a + c$ is prime. Therefore, their product cannot be a prime. Hence $uv^{a+c}w \notin L$.

Exercise 4.5. Show that the language $L = \{1^{n^2} \mid n \geq 0\}$ on the alphabet $\{1\}$ is not regular. Interpreting $\{1\}^*$ as $\mathbb{N}$, it means that the set of all perfect squares is not regular.

Exercise 4.6. Show that the language $L = \{1^{2^n} \mid n \geq 0\}$ on the alphabet $\{1\}$ is not regular. It means that the set of all powers of two is not regular.

Exercise 4.7. Show that the language $L = \{ww \mid w \in \{a, b\}^*\}$ on the alphabet $\{a, b\}$ is not regular.

5 Turing Machines

Let us recall our informal notion of computation as a mechanical manipulation of symbols via the local operations of reading, writing and erasing, all governed by a finite set of given rules. In the previous sections we studied the limited power of computability when only reading is allowed. Now, we are ready to talk about the full story.

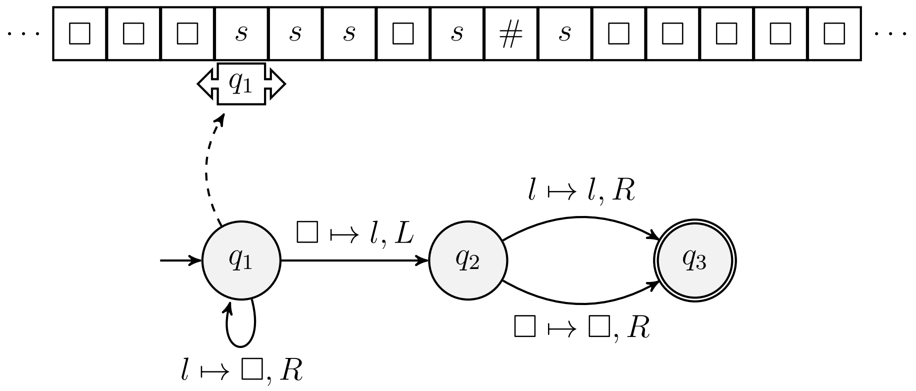
Definition 5.1. A Turing machine M is described by a tuple $M = (Q, \Sigma, \delta, q_1, F)$ containing:

- A finite set Q of possible states of M. We assume that Q contains a designated start state q_1, and $F \subseteq Q$ as the *halting* states.
- A finite set Σ of the input alphabet such that $\Box, \# \notin \Sigma$ where $\Box$ is a symbol for blank and $\#$ is a symbol for separating comma that we will see later.
- A transition function $\delta : (Q - F) \times (\Sigma \cup \{\Box, \#\}) \to Q \times (\Sigma \cup \{\Box, \#\}) \times \{L, R\}$.

We also need a formal version of *a piece of paper* on which the computation takes place. For that matter, the Turing machine $M = (Q, \Sigma, \delta, q_1, F)$

uses an unbounded linear tape, divided into infinite discrete cells. Initially, M receives its input $w = w_1 w_2 \dots w_n \in (\Sigma \cup \{\square, \#\})^*$ on the tape with the rest of the tape filled with the blank symbol $\square$. The machine also has a read/write head. By convention, we assume that the head always starts on the leftmost square of the input. If the input is empty, then it starts somewhere. Once M has started, the computation proceeds according to the rules described by the transition function, meaning that if the machine is in the state p reading the letter s, it goes to the state q, changes the content of the cell to s' and go one cell right or left according to D where $\delta(p, s) = (q, s', D)$ and $D \in \{L, R\}$. The computation continues until it enters one of the halting states. If it does not occur, M goes on forever.

Just like a finite automaton, we can also represent a Turing machine by a directed graph with the nodes representing the states and edges with labels $s \mapsto s', D$ where $s, s' \in \Sigma \cup \{\square\}$ and $D \in \{L, R\}$, representing the transition function. If the transition function sends p to q reading the letter s with the instruction to erase s and write s' and move the head to D, there will be an edge from p to q with the label $s \mapsto s', D$. Similar to what we had for automata, the start state is shown by a node with a small unlabelled in-edge and the halting states are indicated by double circles.



$\cdots$ $\square$ $\square$ $\square$ s s s $\square$ s $\#$ s $\square$ $\square$ $\square$ $\square$ $\square$ $\cdots$
q_1
q_1
$\square \mapsto l, L$
q_2
$l \mapsto l, R$
q_3
$\square \mapsto \square, R$
$l \mapsto \square, R$

Figure 19: A TM with the states $\{q_1, q_2, q_3\}$, alphabet $\{s\}$ and $F = \{q_3\}$.

Definition 5.2. A partial function[6] $f : (\Sigma^*)^k \to \Sigma^*$ is *computable by the Turing machine* M, if for any $(w_1, w_2, \dots, w_k) \in (\Sigma^*)^k$, the machine halts on the input $w_1 \# w_2 \# \dots \# w_k$ iff $(w_1, \dots, w_k)$ is in the domain of the function

[6]Unlike the usual practice of mathematics, when we write $f : A \to B$ we mean a partial function with $dom(f) \subseteq A$.

f. Moreover, when it halts, the value of f must appear on the tape between two lines of blanks while the head is on its first letter. A partial function is called *computable* if it is computable by some Turing machine.

To represent the truth values over any alphabet set Σ, fix an element $a \in \Sigma$ and represent $\mathbf{0}$ and $\mathbf{1}$ by the empty string and the string a, respectively.

Definition 5.3. A relation $R \subseteq (\Sigma^*)^k$ is called *semi-decidable* or *computably enumerable*[7], c.e. for short, if there exists a Turing machine M such that if $\vec{w} \in R$ it halts on the input $w_1\#w_2\#\dots\#w_k$ and outputs $\mathbf{1}$ and if $\vec{w} \notin R$, it does not halt. It is called *decidable* or *computable* if there exists a Turing machine M such that it always halts on the input $w_1\#w_2\#\dots\#w_k$ and outputs $\mathbf{1}$ if $\vec{w} \in R$ and $\mathbf{0}$ if $\vec{w} \notin R$.

Remark 5.4. Note that the relation R is decidable iff its characteristic function χ_R:

$$\chi_R(\vec{w}) = \begin{cases} \mathbf{1} & \vec{w} \in R \\ \mathbf{0} & \vec{w} \notin R \end{cases}$$

is computable and it is c.e. if the function χ'_R with the domain R and the constant value $\mathbf{1}$ is computable.

5.1 Some Examples

In this subsection we will present some basic examples of Turing machines. We will first focus on the computable functions to see how to compute a concrete function. Then we will move to the decision procedures and their corresponding decidable and c.e. relations.

5.1.1 Computable Functions

Example 5.5. Consider the Turing machine M on the alphabet Σ with the states $\{p, q\}$ such that $F = \{q\}$. Define δ as $(p, \Box, R)$ for the input (p, l) for any $l \in \Sigma \cup \{\#\}$ and $(q, \Box, R)$ for $(p, \Box)$. Then, M erases its input, if it is in Σ^*. See Figure 20. For $\Sigma = \{1\}$, it computes the constant zero function.

Exercise 5.6. Let $u \in \Sigma^*$ be a fixed string. Describe a Turing machine that computes the function C_u defined by $C_u(w) = u$ for any $w \in \Sigma^*$.

[7]The terminology here may seem a bit mysterious, simply because the machine is apparently implementing a sort of decision procedure and not enumeration. We will explain the reason behind this terminology in the following sections.

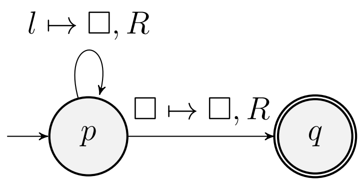


Figure 20: Example 11.29

Example 5.7. Consider the Turing machine M on the alphabet Σ with the states $\{p, q, r\}$ such that $F = \{r\}$. Pick $a \in \Sigma$ and define δ as (p, l, R) for the input (p, l) for any $l \in \Sigma \cup \{\#\}$; (q, a, L) for $(p, \square)$; (q, l, L) for (q, l) for any $l \in \Sigma \cup \{\#\}$ and $(r, \square, R)$ for $(q, \square)$. Then M computes the total function $f : \Sigma^* \to \Sigma^*$ that sends w to wa. See Figure 21. For $\Sigma = \{1\}$, it computes the successor function.

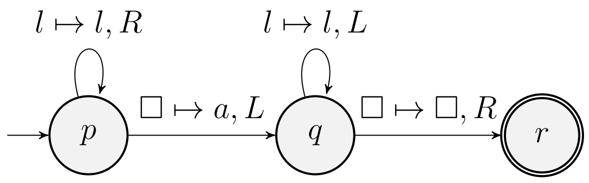


Figure 21: Example 11.30

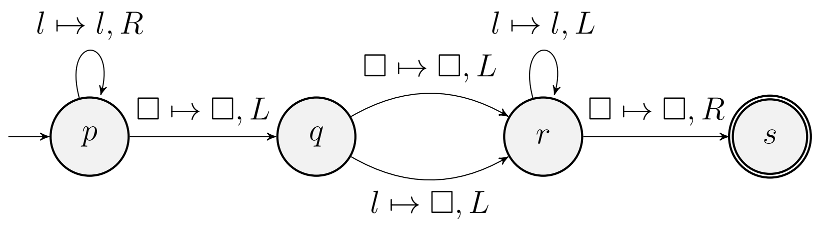
Exercise 5.8. Let $u \in \Sigma^*$ be a fixed string. Describe a Turing machine that computes the function S_u defined by $S_u(w) = wu$ for any $w \in \Sigma^*$.

Example 5.9. Consider the Turing machine M on the alphabet Σ with the states $\{p, q, r, s\}$ such that $F = \{s\}$. Define δ as (p, l, R) for the input (p, l) for any $l \in \Sigma \cup \{\#\}$; $(q, \square, L)$ for $(p, \square)$; $(r, \square, R)$ for (q, l) for any $l \in \Sigma \cup \{\#\}$; $(r, \square, L)$ for $(q, \square)$; (r, l, L) for (r, l) for any $l \in \Sigma \cup \{\#\}$ and $(s, \square, R)$ for $(r, \square)$. This machine erases the right-most letter of $w \in \Sigma^*$, if there is any, otherwise, it returns ϵ. Therefore, M computes the function:

$$\begin{cases} Pred(\epsilon) = \epsilon \\ Pred(wa) = w \end{cases}$$

See figure 22. For $\Sigma = \{1\}$, it computes the predecessor function.

As we can observe even with these very simple examples, spelling out all the details of a Turing machine and drawing its graph representation can be very complicated and time-consuming. Therefore, from now on, we will explain the *algorithm* behind the machine in words and in a more high-level



$l \mapsto l, R$
$\square \mapsto \square, L$
$\square \mapsto \square, L$
$l \mapsto l, L$
$\square \mapsto \square, R$
$l \mapsto \square, L$
p
q
r
s

Figure 22: Example 11.31

manner. It helps to see the main ideas of the algorithm rather than the details of the implementation. However, one has to be aware that every high-level operation must be reducible to the basic operations of the Turing machines, controlled by their finite number of states.

Example 5.10. (*Projection*) Consider the Turing machine `Projection` on any set of alphabets Σ:

(I) Go right and erase everything till you see the $(i-1)$th #. (Don't erase this #). Then, go right but do nothing till you see the next #. Then, again go right and erase the tape till you read blank. Come back till you see #. Erase # and go one cell to the right.

This machine reads the input $w_1\#w_2\#\dots\#w_k$, where $w_j \in \Sigma^*$ and outputs w_i for $i \leq k$. Therefore, it computes the projection function $I_k^i(w_1, \dots, w_k) = w_i$.

Example 5.11. (*Moving Right*) Consider the Turing machine `MovingRight` on any set of alphabets $\Sigma \cup \{\#\}$:

(I) If you read blank, go one step right and halt. If you read any letter l go right till you reach a blank. Then, go to (II).

(II) Go one step left, read the content. If it is in Σ, remember it, change it to blank. Go one step right, write the letter, go left and go to (II). If it is blank, go two steps right and halt.

This machine moves the input string in Σ^* one step to the right.

Example 5.12. (*Concatenation*) Consider the following `Concatenation` algorithm:

(I) Go right till reaching the first #. Erase it and go left till you see the first blank. Then, go one step right and apply the `MovingRight` algorithm as in Example 5.11.

If we apply the `Concatenation` to the input $u\#v$, it moves u one step to the right to produce the concatenation uv. Therefore, the machine computes the concatenation function $f : \Sigma^* \times \Sigma^* \to \Sigma^*$ mapping (u, v) to uv. Note that if the algorithm `MoveRight` reads $u\Box v$, it does not care what is in the right hand side of the middle blank. The `Concatenation` uses this property, substantially. Note that for the language $\Sigma = \{1\}$ and identifying $\{1\}^*$ by $\mathbb{N}$, the function f is the usual addition.

Exercise 5.13. (*Sequence Concatenation*) Let Σ be a set of alphabets. Write a Turing machine `SeqConcatenation` on Σ such that if it reads $u_1\#u_2\#\dots\#u_n$ where $u_i \in \Sigma^*$, it returns the concatenation $u_1u_2\dots u_n$.

Exercise 5.14. (*Copy*) Let Σ be a set of alphabets. Write a Turing machine `Copy` on Σ such that if it reads $u\#w$, where $u \in (\Sigma \cup \{\#\})^*$ and $w \in \Sigma^*$, it returns $u\#w\#w$ and when it halts the head is on the first letter of the first w.

Example 5.15. (*Multiplication*) Consider the Turing machine `Multiplication` on any set of alphabets Σ:

(I) Read the current cell. If it is #, then go right and erase everything till the first blank. Then, go left till reaching the # and erase it and halt. If the content of the current cell is in Σ, erase it, go one step to the right and go to (II).

(II) Read the content of the current cell, if it is #, erase it, go one step to the right and go to (III). If the content of the current cell is a letter in Σ, erase it and go right till reaching the first blank. Then, go left until reaching the first #. Go one cell right and run `Copy`. Go left till reaching the first blank. Go one step right. Go to (II).

(III) Run `SeqConcatenation`.

This machine computes $f : \Sigma^* \times \Sigma^* \to \Sigma^*$ that sends (u, v) to $v^{|u|}$. Note that for the language $\Sigma = \{1\}$ and identifying $\{1\}^*$ by $\mathbb{N}$, the function f is the usual multiplication.

Example 5.16. (*Boolean Operations*) Consider the Turing machine `Disjunction` defined as follows: Run `Concatenation`. Scan the output. If it consists of two a's, erase one a, move the head over that a and halt. Otherwise, just halt. This machine computes the disjunction function $\vee : \{\mathbf{0}, \mathbf{1}\} \times \{\mathbf{0}, \mathbf{1}\} \to \{\mathbf{0}, \mathbf{1}\}$. For the conjunction function $\wedge : \{\mathbf{0}, \mathbf{1}\} \times \{\mathbf{0}, \mathbf{1}\} \to \{\mathbf{0}, \mathbf{1}\}$ use `Multiplication`. For negation, $\neg : \{\mathbf{0}, \mathbf{1}\} \to \{\mathbf{0}, \mathbf{1}\}$, use the following `Negation` machine: Scan the input. If it consists of one a, erase the a and halt. If the input is empty, write one a and halt. Otherwise, just halt.

Exercise 5.17. Show that the following function $C : \{\mathbf{0}, \mathbf{1}\} \times \Sigma^* \times \Sigma^* \to \Sigma^*$ is computable:

$$C(u, v, w) = \begin{cases} v & u = \mathbf{0} \\ w & u = \mathbf{1} \end{cases}$$

Exercise 5.18. Let $S \subseteq \Sigma^*$ be a finite set and $f : S \to \Sigma^*$ be any function. Show that f is computable. Note that it implies that any function with finite domain is computable.

5.1.2 Computably Enumerable and Decidable Relations

Example 5.19. Consider the Turing machine M on the alphabet $\Sigma = \{1\}$ with the states $Q = \{p, q, r, s\}$ such that $F = \{s\}$. Define δ as $(q, 1, R)$ for the input $(p, 1)$; (q, l, R) for the input (q, l) for any $l \in \{1, \Box\}$; $(r, 1, R)$ for $(p, \Box)$; (s, l, L) for (r, l) for any $l \in \{1, \Box\}$. The machine halts if it reads the empty string and outputs 1 but if we feed the machine any other string, then it goes right forever. Therefore, M computes the partial function $f : \{1\}^* \to \{1\}^*$ with domain $\{\epsilon\}$ and the value $\mathbf{1} = 1$. In other words, it shows that the set $\{\epsilon\}$ is computably enumerable. See Figure 23.

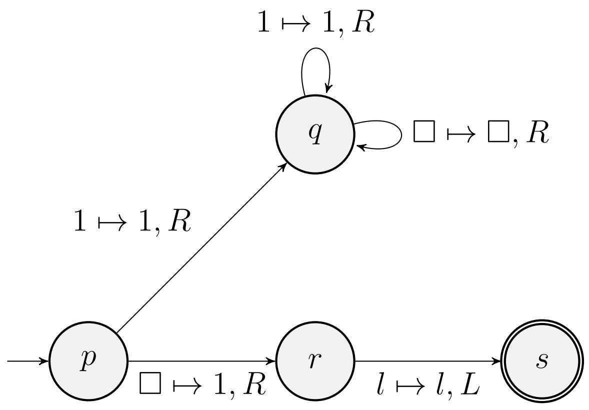


Figure 23: Example 5.19

Exercise 5.20. By constructing two Turing machines, show that the equality relation $\{(u, v) \in \Sigma^* \times \Sigma^* | u = v\}$ is both decidable and computably enumerable.

Exercise 5.21. By constructing two Turing machines, show that the first segment relation $\{(u, v) \in \Sigma^* \times \Sigma^* | \exists w(uw = v)\}$ is both decidable and computably enumerable.

Example 5.22. Any regular language L is also decidable. Let us assume that the DFA M recognizes L. We will construct a Turing machine N deciding L. The algorithm N first runs M till it reads all the input. If it accepts the input, N erases everything, writes **1** and halts. If it rejects, N erases everything, writes **0** and halts.

The previous example shows that the class of all decidable languages extends the class of all regular languages. This extension is proper as we will see in the following example:

Example 5.23. Consider the following Turing machine:

(I) If you read blank, write **1** and halt. If you read b, erase everything and right **0**. If you read a, erase it and then go right till you reach the blank. Go left and check whether the last letter is b or not. If not, erase everything, write **0** and halt, if yes, erase the last letter and go left till reaching the blank. Go right and go to (I).

This machine computes the function $f : \{a, b\}^* \to \{\mathbf{0}, \mathbf{1}\}$ that sends all strings of the form $a^n b^n$ to **1** and the rest to **0**. Therefore, it decides the language $L = \{a^n b^n | n \geq 0\}$ and hence there are some decidable non-regular languages.

Exercise 5.24. Show that the language $\{1^{2^n} | n \geq 0\}$ over the alphabet $\{1\}$ is decidable.

Example 5.25. Any decidable language is also computably enumerable. Let M be the decision procedure for R. Then define the machine N as first running M, if it outputs **1** halt. If it outputs **0**, go to right forever.

Remark 5.26. The inclusion of the Example 5.25 is also proper. We will provide a c.e. but undecidable language later in the course.

6 Variants of Turing Machines

In the previous section we introduced the deterministic Turing machines with one linear unbounded tape, and we have defined the notion of computability based on that definition. Fortunately, the notion of computability is extremely *robust* and independent from the implementing details of the Turing machines. In the following we will mention some variants of the Turing machines with different structural details but with the same power of computation:

- Usually Turing machines are defined without the set F and the transition function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$ as a *partial function.* In these machines, halting means reaching a configuration outside of the domain of δ.

- Just like the case for NFAs, it is also possible for Turing machines to be *non-deterministic.* The idea, again, is allowing the function δ to be a relation.

- For computably enumerable and decidable relations, sometimes people use the machines with two specified states called *the accept state* and *the reject state.* If the machine reaches the accept state (reject state) it accepts (rejects) the input, otherwise the machine does not halt on the input. In these machines, accepting or rejecting the language has nothing to do with the outputs of the machine.

- It is possible to use a greater set of *working alphabets* than the the set of the *input alphabets* Σ. This set is usually denoted by Γ and it should be an extension of Σ.

- The machines can have access to *more local moves* than the two basic moves of left and right. For instance in some cases it is useful to have a machine with three moves Left, Right and *Stay.*

- It is also possible to change the details of *the tape* of the machine. The machine could have only *one semi-unbounded tape* that is unbounded only on one direction and bounded on the other or it can have *fixed number of tapes with separate heads for any tape* or even *a plane* type of tape with one head with four moves for four directions.

- The machine can receive its input in many different ways. For instance, the head can be on its right-most end of the input or when the machine has a one-sided tape, the head can be on the first cell of the bounded side of the tape.

7 A Bit of Recursion Theory

As we have observed for finite automata and their relation with regular expressions, in some cases, it is possible to develop a machine independent characterization for computability with respect to a given notion of computation. Turing machines are no exception in this regard. In this section we will introduce a machine-independent characterization for computable functions and computably enumerable and decidable relations. As we can expect, such a characterization is helpful in proving the computability of functions and relations without providing the actual possibly complex Turing machines. However, we have to pay the price for handling these complicated details at some point and this section and its equivalence theorem may be the reasonable point to handle them. However, we have to confess that in this section we will only use the computability in its very high-level sense, avoiding the implementation details, altogether. Admittedly, this is a sort of cheating, but in our defence, let us emphasize that for such a short lecture, it may be reasonable to explain the main ideas behind the *algorithms* rather than getting lost in the implementation details.

Theorem 7.1. *(Basic Functions) The basic functions $Z(w) = \epsilon$, $S_a(w) = wa$ for $a \in \Sigma$ and $I^i_k(w_1, w_2, \dots, w_k) = w_i$ for $1 \leq i \leq k$ are all computable.*

Proof. See the examples of the previous lecture on Turing machines. □

Theorem 7.2. *(Composition) If $f : (\Sigma^*)^k \to \Sigma^*$ and $g_i : (\Sigma^*)^l \to \Sigma^*$ for $1 \leq i \leq k$ are computable, then so is $h = f(g_1, \dots, g_k)$.*

Proof. Use a Turing machine with k tapes to compute the function h. First, for each $i \leq k$, run g_i on the ith tape, one after another. If one of them does not halt, then as we expect, the new machine does not halt, as well. If all halt with the outputs $g_i(\vec{w})$, then copy all the elements of all the tapes in the first tape in the form $g_1(\vec{w})\#g_2(\vec{w})\#\dots\#g_k(\vec{w})$ and then run the computing algorithm for f on the first tape.

□

Application 7.3. *(Substitution)* If $R \subseteq (\Sigma^*)^k$ is a c.e. relation and for any $1 \leq i \leq k$ the function $g_i : (\Sigma^*)^l \to \Sigma^*$ is computable, then, the relation $S = R(g_1, \dots, g_k)$ is also c.e. If R is decidable and all g_i's are total, S is also decidable.

Proof. It is a consequence of the closure of the class of computable functions under composition and the fact that $\chi'_S = \chi'_R(g_1, \dots, g_k)$. Note that if R is decidable, then by the totality of g_i's, we have $\chi_S = \chi_R(g_1, \dots, g_k)$ and hence the result follows. □

Exercise 7.4. Let $f : (\Sigma^*)^k \to \Sigma^*$ be a computable function. Then, the graph of f, i.e., $\{(\vec{x}, y) \mid f(\vec{x}) = y\}$ is a c.e. relation. If f is also total, its graph is decidable.

Theorem 7.5. *(Primitive Recursion) Assume the functions $g : (\Sigma^*)^k \to \Sigma^*$ and for any $a \in \Sigma$, $h_a : \Sigma^* \times (\Sigma^*)^k \times \Sigma^* \to \Sigma^*$ are computable. Then the function $f : \Sigma^* \times (\Sigma^*)^k \to \Sigma^*$ with the following definition is also computable:*

$$\begin{cases} f(\epsilon, \vec{v}) = g(\vec{v}) \\ f(ua, \vec{v}) = h_a(u, \vec{v}, f(u, \vec{v})) \end{cases}$$

Proof. First, note that to compute f on any input in the form $(wa, \vec{v})$, it is enough to know the value $f(w, \vec{v})$. Using this observation, we can develop a possible computing procedure: We reduce computing f on $(w, \vec{v})$ to computing f on some $(w', \vec{v})$, where w' is shorter than w and we keep following this procedure till reaching $f(\epsilon, \vec{v})$ which is computable by $g(\vec{v})$. Now, the strategy is following the procedure that we have constructed, backwardly. First, compute $f(\epsilon, \vec{v})$. Then, compute f for the previous element using the appropriate h_a, till you reach the input u. More formally: Use a 4-taped machine to compute f. Write u on the first tape and $\vec{v}$ on the third tape. First, copy $\vec{v}$ on the fourth tape, run g on $\vec{v}$. Then, read the elements of u one by one and in each step, copy u on the second tape, keep the part of u that you have read and erase the rest, and apply the predecessor. If you have read a, copy the content of the second and the third tape to the fourth tape in the order second, third, fourth and then apply h_a on the fourth tape. Erase the second tape and do it again till you scan all the elements of u. Then, halt and output the content of the fourth tape. □

Example 7.6. The concatenation function $Con(u, v) = uv$ is computable. Use recursion on v to define $Con(u, v)$:

$$\begin{cases} Con(u, \epsilon) = u = I_1^1(u) \\ Con(u, va) = S_a(Con(u, v)) \end{cases}$$

Since both I_1^1 and S_a are computable, the so is Con. Note that identifying $\mathbb{N}$ with $\{1\}^*$, Con becomes the usual addition function, denoted by Add.

Example 7.7. The *conditional* function $C : \Sigma^* \times \Sigma^* \times \Sigma^* \to \Sigma^*$ with the definition:

$$C(u, v, w) = \begin{cases} v & u = \epsilon \\ w & u \neq \epsilon \end{cases}$$

is computable. It is enough to use primitive recursion on u to define $C(\epsilon, v, w) = I_2^1(v, w)$ and $C(ua, v, w) = I_3^3(u, v, w)$.

Exercise 7.8. As usual, identify the set of natural numbers with $\{1\}^*$. Then, show that the functions $Sum(m, n) = m+n$, $Prod(m, n) = mn$, $Exp(m, n) =$

m^n, $Fact(n) = n!$, the predecessor $Pred(n) = max\{0, n-1\}$, the proper subtraction $PSub(m, n) = max\{0, m-n\}$, the division $div(m, n) = \lfloor \frac{m}{n+1} \rfloor$ and the remainder $r(m, n) = m - (n+1)div(m, n)$ are all computable.

Exercise 7.9. Show that over the alphabets $\{1\}^*$, the function $Sign(n)$ with the following definition:

$$Sign(n) = \begin{cases} 1 & n = 0 \\ 0 & n \geq 1 \end{cases}$$

is computable. Then, use $Sign$ and $PSub$ to prove that the inequality relation $\{(m, n) \in \{1\}^* \times \{1\}^* | m \leq n\}$ is decidable. Finally, show that the equality relation and the strict inequality are also decidable.

Definition 7.10. Any function constructed from the basic functions and the operations of composition and primitive recursion is called *primitive recursive.*

Application 7.11. *(Boolean Combinations)* Let $R, S \subseteq (\Sigma^*)^k$ be two computably enumerable relations. Then $R \cap S$ is also computably enumerable. If R is also decidable, then so is R^c. Therefore, any boolean combinations of decidable relations is also decidable.

Proof. For the first part, note that $\chi'_{R\cap S} = C_{\mathbf{1}}(\chi'_R, \chi'_S)$ where $C_{\mathbf{1}} : \Sigma^* \times \Sigma^* \to \{\mathbf{1}\}$ is the constant function, constructible via composition and basic functions as $C_{\mathbf{1}}(u, v) = S_{\mathbf{1}}(Z(I^1_2(u, v)))$. Therefore, by the closure under composition, we see that if χ'_R and χ'_S are computable, then so is $\chi'_{R\cap S}$.
For the second part, the claim is a consequence of the closure under composition and the fact that $\chi_{R\cap S} = C(\chi_R, \mathbf{0}, \chi_S)$ and $\chi_{R^c} = C(\chi_R, \mathbf{1}, \mathbf{0})$, where C is the conditional function. □

Remark 7.12. The complement of a c.e. relation is not necessarily c.e. In the last section we will see an interesting counter-example.

Theorem 7.13. *(Union) Let $R, S \subseteq (\Sigma^*)^k$ be two computably enumerable relations. Then, $R \cup S$ is also computably enumerable.*

Proof. Let M and N be the algorithms for R and S. Then, for an algorithm K for $R \cup S$, run M and N in parallel. More formally, use a 2-taped Turing machine and copy the input on the second tape, as well. Then, run one step of M on the first tape and then one step of N on the second tape and keep implementing both algorithms at the same time. The whole process halts if at least one of M or N halts and it outputs $\mathbf{1}$ when it halts. This algorithm clearly computes $\chi'_{R\cup S}$. Note that at the first glance, it can be really tempting to simplify the algorithm by running M first and then applying N. The problem with this algorithm is that it ignores the possibility in which M does not halt while N halts on the input. So the algorithm get stuck in running M and can not see that N actually halts on the input. $\square$

Application 7.14. *(Definition by cases)* Let $R \subseteq (\Sigma^*)^k$ be a decidable relation and $g, h : (\Sigma^*)^k \to \Sigma^*$ be two computable functions. Then, the function $f : (\Sigma^*)^k \to \Sigma^*$ defined by:

$$f(\vec{u}) = \begin{cases} g(\vec{u}) & \vec{u} \in R \\ h(\vec{u}) & \vec{u} \notin R \end{cases}$$

is also computable.

Proof. The theorem is a consequence of the closure under composition and the fact that we have $f(\vec{u}) = C(\chi_R(\vec{u}), h(\vec{u}), g(\vec{u}))$. $\square$

Example 7.15. Let Σ be a set of alphabets. Then the length function $|\cdot| : \Sigma^* \to \{\mathbf{1}\}^*$ sending w to $\mathbf{1}^n$ where n is the length of w, is computable. It is enough to use recursion on u to define $|u|$ as $|\epsilon| = \epsilon$ and $|ua| = S_{\mathbf{1}}(|u|)$.

Lemma 7.16. *For any primitive recursive function over $\{1\}$, the function $\tilde{f}$ over Σ with the definition $\tilde{f}(\vec{u}) = \mathbf{1}^{f(|\vec{u}|)}$ is primitive recursive over Σ.*

Proof. The proof is easy and uses induction on the structure of f. $\square$

Exercise 7.17. Use Lemma 7.16 to show that both of the length-inequality relation

$$LIneq(u, v) = \{(u, v) \in \Sigma^* \times \Sigma^* |\ |u| \leq |v|\}$$

and the length-equality relation

$$Leq(u, v) = \{(u, v) \in \Sigma^* \times \Sigma^* |\ |u| = |v|\}$$

are decidable.

Exercise 7.18. First show that the set $\{\epsilon\}$ is decidable. Then, prove that for any $a \in \Sigma$, the set $\{a\}$ is also decidable. We will denote this predicate with Eq_a.

Exercise 7.19. Show that the function $Seg(u, v)$ computing the first segment of u consisting of the leftmost $|v|$ elements of u, is computable. Then, show that the $|v|$-th component function u_v computing the $|v|$'th letter of u (for $|u| < |v|$ answers ϵ) is computable.

Exercise 7.20. Let $R, S \subseteq (\Sigma^*)^k$ be two decidable relations and $g_1, g_2, g_3, g_4 : (\Sigma^*)^k \to \Sigma^*$ be four computable functions. Show that

$$f(\vec{u}) = \begin{cases} g_1(\vec{u}) & \vec{u} \in R \text{ and } \vec{u} \in S \\ g_2(\vec{u}) & \vec{u} \in R \text{ and } \vec{u} \notin S \\ g_3(\vec{u}) & \vec{u} \notin R \text{ and } \vec{u} \in S \\ g_4(\vec{u}) & \vec{u} \notin R \text{ and } \vec{u} \notin S \end{cases}$$

Exercise 7.21. Let $f(m, \vec{n})$ be a computable function over $\mathbb{N}$. Show that the functions $g(p, \vec{n}) = \sum_{m \leq p} f(m, \vec{n})$ and $h(p, \vec{n}) = \prod_{m \leq p} f(m, \vec{n})$ are computable.

Exercise 7.22. Let $\pi(n)$ be the number of prime numbers less than or equal to n. Show that π is a primitive recursive function over $\mathbb{N}$.

Exercise 7.23. *(Bounded Concatenation)* Assume that $f : \Sigma^* \times (\Sigma^*)^k \to \Sigma^*$ is computable. Show that the function $\bar{f} : \Sigma^* \times (\Sigma^*)^k \to \Sigma^*$ with the following definition is computable, as well:

$$\bar{f}(u, \vec{v}) = f(\mathbf{1}^0, \vec{v}) f(\mathbf{1}^1, \vec{v}) \ldots f(\mathbf{1}^{|u|}, \vec{v})$$

Exercise 7.24. Show that the function $Mirror : \Sigma^* \to \Sigma^*$ mapping w to its mirror image is computable. By the mirror image of w, we mean a word by reading w from right to left. For instance, $Mirror(abb) = bba$.

Exercise 7.25. Show that the function $e : (\{a, b, ;\})^* \to (\{a, b\})^*$ mapping any letter a, b and $;$ in w to aa, bb and ab, respectively, is computable. For instance, $en(a;b) = aaabbb$.

Application 7.26. *(Bounded Search)* If $f : \Sigma^* \times (\Sigma^*)^k \to \Sigma^*$ is a total computable function, then so is $\mu.|u| \leq |w|\ [f(u, \vec{v}) = \mathbf{1}]$, where $\mu.|u| \leq |w|\ [f(u, \vec{v}) = \mathbf{1}]$ is defined as $\mathbf{1}^k$, where $k \leq |w|$ is the length of the shortest u such that $f(u, \vec{v}) = \mathbf{1}$. If there is no such u, then the output is $\mathbf{1}^{|w|+1}$.

Proof. Denote $\mu|u| \leq |w|\ [f(u, \vec{v}) = \mathbf{1}]$ by $g(w, \vec{v})$. We will use primitive recursion on w to show that g is computable. For $w = \epsilon$, the only possible u is $u = \epsilon$. Hence, it is just enough to check whether $f(\epsilon, \vec{v}) = \mathbf{1}$ or not. We can do it as $\{\mathbf{1}\}$ is decidable. Then, using the definition by cases operation, if $f(\epsilon, \vec{v}) = \mathbf{1}$, set $g(\epsilon, \vec{v}) = \mathbf{1}^0 = \epsilon$. Otherwise, set $g(\epsilon, \vec{v}) = \mathbf{1}$. To compute $g(wa, \vec{v})$ from $g(w, \vec{v})$, again we use definition by cases. If the shortest possible u with the condition $|u| \leq |w|$ exists (check it by checking the decidable $|g(w, \vec{v})| = |w| + 1$), then we have found our answer. If not, we have to check all possible $f(wa, \vec{v}) = \mathbf{1}$, for any $a \in \Sigma$, because the intended u may be one of these finitely many possibilities. If we find one, the answer is $\mathbf{1}^{|w|+1}$. Otherwise, it is $\mathbf{1}^{|w|+2}$. □

Remark 7.27. Note that for any decidable relation $R(u, \vec{v})$, the function $g(w, \vec{v}) = \mu|u| \leq |w|.R(u, \vec{v})$ is computable, where $\mu|u| \leq |w|.R(u, \vec{v})$ is $\mathbf{1}^n$ where $n \leq |w|$ is the length of the shortest possible u such that $R(u, \vec{v})$. And if there is no such u, then $\mu|u| \leq |w|.R(u, \vec{v})$ is $|w| + 1$. The reason is simply the fact that $R(u, \vec{v})$ is equivalent to $\chi_R(u, \vec{v}) = \mathbf{1}$.

Example 7.28. The function $NextP(x)$ that finds the least prime $p > x$ is computable, because it is definable via bounded search $NextP(x) = \mu y \leq 2x.[Prime(y) \wedge x < y]$. (We will see how to define $Prime(y)$ in a moment). Note that we are using Bertrand's postulates that states between any number $x \geq 1$, there exists a primes number $x < p \leq 2x$. Moreover, it is also possible to use $NextP$ to define the function p_x that computes the x-th prime number. It is enough to use primitive recursion to define p_x by $p_0 = SS(0) = 2$ and $p_{x+1} = NextP(p_x)$.

Example 7.29. The equality relation over the alphabet Σ is decidable because it has the following construction: $(|u| = |v|) \wedge |x_{(u,v)}| = |u| + 1$ where $x_{(u,v)} = \mu|w| \leq |u| \; \neg[u_w \equiv v_w]$ and $y \equiv z$ is an abbreviation for $\bigvee_{a \in \Sigma}(Eq_a(y) \wedge Eq_a(z))$.

Exercise 7.30. Let $R \subseteq \Sigma^*$ be a finite set. Show that χ_R is primitive recursive.

Application 7.31. *(Bounded Quantifiers)* If $R \subseteq \Sigma^* \times (\Sigma^*)^k$ is a decidable relation, then so are $S(w, \vec{v}) = \forall|u| \leq |w| \; R(u, \vec{v})$ and $T(w, \vec{v}) = \exists|u| \leq |w| \; R(u, \vec{v})$.

Proof. By closure under booleans, it is enough to prove the theorem for T. By bounded search, $f(w, \vec{v}) = \mu.|u| \leq |w| \; [\chi_R(u, \vec{v}) = \mathbf{1}]$ is computable and total. Now, check whether $|f(w, \vec{v})| = |w| + 1$ or not. If $|f(w, \vec{v})| = |w| + 1$, then $\exists|u| \leq |w| \; R(u, \vec{v})$ does not hold . If $|f(w, \vec{v})| \neq |w| + 1$, then we have $\exists|u| \leq |w| \; R(u, \vec{v})$. □

Example 7.32. The set of all prefect squares in the language $\{1\}^*$ is decidable, because it is definable by

$$Square(x) = [\exists y \leq x(x = y^2)]$$

The latter is decidable as multiplication is total and computable and the equality relation is decidable. Hence, by substitution the relation $x = y^2$ is decidable. Finally, since decidability is closed under bounded quantifiers, the claim follows.

Example 7.33. The function $GP(n)$ that computes m such that p_m is the greatest prime that $p_m | n+2$, is computable, because it is constructible via bounded search $\mu i \leq n+2.\ [\exists k \leq n+2(p_{n+2-i}k = n+2)]$. Here, we use the fact that if $p_m | n+2$, then $m \leq p_m \leq n+2$.

Example 7.34. The set of all prime numbers in the language $\{1\}^*$ is decidable, because it is definable by

$$Prime(x) = [(x > 1) \wedge \forall yz \leq x((x = yz) \to (y = x \vee y = 1))]$$

The latter is decidable as the equality and inequality relations are decidable. Hence, by substitution the relations $x > 1$, $x = yz$, $y = x$ and $y = 1$ are decidable. Finally, since decidability is closed under boolean operations and bounded quantifiers, the claim follows.

Remark 7.35. Note that the languages in the previous examples, namely $\{1^{n^2} \mid n \geq 0\}$ and $\{1^p \mid p \text{ is prime}\}$ are not regular. Therefore, we can also use the above argument for their decidability to show that the inclusion of regular languages in decidable languages is proper.

Exercise 7.36. Show that the function $exp(m, n)$ computing the exponent of $m+2$ in $n+1$, meaning the greatest k such that $(m+2)^k$ devides $n+1$, is computable.

Exercise 7.37. Show that the function $ind(m, n)$ computing the exponent of p_m in $n+1$ is computable.

Exercise 7.38. Use bounded search to show that the functions $PSub$ and div are computable.

So far, we have shown that all primitive recursive functions are computable. The natural question is that whether these functions exhaust the whole class of computable functions? The answer is of course not. Because there are so many partial computable functions while all primitive recursive functions are total by construction. However, the partiality may not be an important problem. Therefore, let us revise the question: Is a total computable function primitive recursive? The answer is again negative:

Example 7.39. Consider the Ackerman function defined recursively by:

$$A(m, n) = \begin{cases} n+1 & m = 0 \\ A(m-1, 1) & m > 0, n = 0 \\ A(m-1, A(m, n-1)) & m > 0, n > 0 \end{cases}$$

This function is not primitive recursive as it grows faster than any primitive recursive function. More precisely, it is not hard to prove that for any primitive recursive function $f(\vec{x})$, there exists a number t such that for any $\vec{x}$, we have $f(\vec{x}) < A(t, max\{\vec{x}\})$. Therefore, if A is primitive recursive, there is t such that for any x, y we have $A(x, y) < A(t, max\{x, y\})$. Hence $A(t, t) < A(t, t)$. On the other hand, it is very clear that this function is computable. What is your strategy to compute it?

To capture the full power of computability and its partial nature, we need another operation. But before introducing that operator let us state a technical lemma and a theorem on the existential quantifiers. It helps to understand how the final operator works:

Lemma 7.40. *(Enumerability of the Total Space) There is a total computable function $en : \Sigma^* \to \Sigma^*$ whose restriction to $\{\mathbf{1}\}^*$ is surjective.*

Proof. Let us first explain the enumeration by an example. Let $\Sigma = \{s_1, s_2, s_3\}$. Then, enumerate the set Σ^* in the following way:

$$\epsilon \mid s_1 \mid s_2 \mid s_3 \mid s_1s_1 \mid s_1s_2 \mid s_1s_3, s_2s_1 \mid s_2s_2 \mid s_2s_3 \mid s_3s_1 \mid s_3s_2 \mid s_3s_3$$

$$s_1s_1s_1 \mid s_1s_1s_2 \mid s_1s_1s_3 \mid s_1s_2s_1 \mid ... \mid s_3s_3s_3 \mid s_1s_1s_1s_1 \mid ...$$

The algorithm is clear. Right? Here is the formal version. First, fix an order over the elements of Σ as $\{s_1 < s_2 < \ldots < s_m\}$. Use a 2-taped machine. The input is on the first tape. Now, in each step, read a letter from the input, go one step to right on the input and do the following: Check the second tape. If the rightmost elements is not s_m, change it from blank to s_1 or from s_i to s_{i+1}. If it is s_m, find the leftmost cell whose right side consists only of s_m's. Change the content of the this cell from s_i to s_{i+1} and if it is blank to s_1. Then, change the content of every cell in the right hand side from s_m to s_1. Keep iterating this procedure till reading all the input. Then halt. This algorithm computes a total function and its application even on the elements of $\{\mathbf{1}\}^*$ is enough to cover the whole Σ^*. □

Theorem 7.41. *(Unbounded Existential Quantifier) If $R \subseteq \Sigma^* \times (\Sigma^*)^k$ is a c.e. relation, then so is $S(\vec{v}) = \exists u R(u, \vec{v})$.*

Proof. Let M be the algorithm for R and we want to define an algorithm N for S. The idea is reading the input $\vec{v}$ and then computing $M(u, \vec{v})$ for all possible u's to check whether there exists a u such that $M(u, \vec{v}) = \mathbf{1}$ or not. For this purpose, first order all u's in a length increasing order u_0, u_1, ... Consider the following tempting search algorithm: Apply M on $(u_0, \vec{v})$, then on $(u_1, \vec{v})$ and so on, till one of them outputs $\mathbf{1}$, otherwise do not halt.

The problem with this algorithm is that it is possible that $M(u_0, \vec{v})$ does not halt, while $M(u_1, \vec{v}) = \mathbf{1}$. Therefore, the machine gets stuck in u_0 and never reaches the working u_1. To overcome this problem, we run all these algorithms in parallel and step by step in the following manner and we call the algorithm N. Let $en : \Sigma^* \to \Sigma^*$ be a total computable function whose restriction to $\{\mathbf{1}\}^*$ is surjective. In the stage n, the algorithm N applies M on $(en(n_0), \vec{v})$ for n_1 many steps where $n+1 = 2^{n_0}(2n_1+1)$. Then, N checks whether it reached a halting state or not. If it does, it also halts. Otherwise, it erases everything and go to the next stage. This algorithm captures $S(\vec{v})$. The reason is the following: If N halts and outputs $\mathbf{1}$, then it means that it halts in some stage, say n. Then, it means that the machine M applied on the input $(en(n_0), \vec{v})$ halts after n_1 many steps. Hence, we have $R(en(n_0), \vec{v})$ which implies $S(\vec{v})$. Conversely, if $S(\vec{v})$, then there exists some string u such that $R(u, \vec{v})$. Then, if M runs on $(u, \vec{v})$, it halts after say m many steps. Since the restriction of en to $\{\mathbf{1}\}^*$ is surjective, there exists k such that $en(k) = u$. Then, the machine N on the stage n where $n+1 = 2^k(2m+1)$ simulates the work of M on $(u, \vec{v})$ and hence it halts and outputs $\mathbf{1}$. □

Application 7.42. *(Bounded Existential Quantifier)* If $R \subseteq \Sigma^* \times (\Sigma^*)^k$ is a c.e. relation, then so is $T(w, \vec{v}) = \exists |u| \leq |w| \; R(u, \vec{v})$.

Proof. Since $R(u, \vec{v})$ is c.e. and $|u| \leq |w|$ is decidable, then $|u| \leq |w| \wedge R(u, \vec{v})$ is also c.e. Hence, $\exists u(|u| \leq |w| \wedge R(u, \vec{v}))$ is c.e. □

Example 7.43. Let $p(\vec{x}, \vec{y})$ be a polynomial with non-negative integer coefficients. Then, the relation $R(\vec{x})$ defined as $\exists \vec{y} p(\vec{x}, \vec{y}) = 0$ is computably enumerable, because all polynomials are compositions of addition and multiplications and hence computable and then by decidability of the equality and by substitution, the relation $p(\vec{x}, \vec{y}) = 0$ is decidable and by closure of the c.e. relations under unbounded existential quantifiers, the claim follows.

Example 7.44. Let $\mathcal{L}$ be a first-order language. Then, the set of all $\mathcal{L}$-tautologies over the alphabets $\mathcal{L} \cup \{;\}$ is computably enumerable. Note that we use the symbol ";" to separate formulas and hence to represent the proofs as words in the new language $\mathcal{L} \cup \{;\}$. To show why this set is c.e., note that it has the following description: $\exists \pi Prf(\pi, A)$, where $Prf(\pi, A)$ is the relation "π is a proof of A". Intuitively, it is clear that checking whether something is a proof of some statement is decidable. We only need to check some basic syntactical properties of π to check whether it is really a proof or not.

Exercise 7.45. The domain and the range of any computable function is computably enumerable.

Theorem 7.46. *(Unbounded Search) Let $g : \Sigma^* \times (\Sigma^*)^k \to \Sigma^*$ be a computable function. Then, the function $f : (\Sigma^*)^k \to \Sigma^*$ with the following definition is also computable:*

$$f(\vec{v}) = \mu|u|.\ [g(u, \vec{v}) = \mathbf{1}]$$

where $\mu|u|.\ [g(u, \vec{v}) = \mathbf{1}]$ means the string $\mathbf{1}^n$ where n is the length of the shortest possible u such that $g(u, \vec{v})$ is defined and $g(u, \vec{v}) = \mathbf{1}$ and for any w shorter than u, the value $g(w, \vec{v})$ is defined and $g(w, \vec{v}) \neq \mathbf{1}$. If there is no such u, $f(\vec{v})$ is not defined.

Proof. The most natural algorithm is computing $g(u, \vec{v})$ for the given $\vec{v}$ and all possible u's, one after another in a length increasing order, till we see the first output $\mathbf{1}$ and then we can compute the length of the shortest u. The problem with this algorithm is that it ignores the situation in which the shortest possible length is n and we are checking u before w, both with length n while $g(u, \vec{v})$ is not defined and $g(w, \vec{v})$ is. To overcome this issue, we check all possible lengths but running the machine for g on $(u, \vec{v})$ for all u's with the same length, in parallel. More formally, let $en : \Sigma^* \to \Sigma^*$ be a total computable function whose restriction to $\{\mathbf{1}\}^*$ is surjective and it enumerates all strings in a length increasing order and M be the machine computing g. Then, define the algorithm N in the following way. It consists of some stages that itself is partitioned to some steps. In the stage n, the machine N first computes all $en(i)$'s, one after another, to find the first number N_n such that the length of $en(N_n)$ becomes n. This is possible because en enumerates the elements in a length increasing order. Then, we know that all elements with length n are $en(N_n)$, $en(N_n + 1)$ till $en(N_n + 2^n)$. Now in each step m, the algorithm N first computes m_0 and m_1 where $m + 1 = 2^{m_0}(2m_1 + 1)$ and then if $m_0 > 2^n$, the machine goes to the next step, otherwise N applies M on $(en(N_n + m_0), \vec{v})$ for m_1 many steps and then checks whether M reached a halting state or not. If it does, N checks whether the output of M is $\mathbf{1}$ or not. If it is $\mathbf{1}$, it halts and outputs n. Otherwise, it erases everything, writes m_0 somewhere to remember it and goes to the next step. The machine goes to its next stage, if it gets the halting state of M with output not equal to $\mathbf{1}$ for all possible $0 \leq m_0 \leq 2^n$.
If there exists a shortest possible u such that $f(u, \vec{v}) = \mathbf{1}$, then M halts on any shorter input and outputs something different than $\mathbf{1}$. Hence, N passes all the stages before n, and in the stage n finally meets the answer u at some point since en is surjective. If there is no shortest u, then there is a number n such that M halts on any $(u, \vec{v})$ where $|u| < n$ and for u with length n at least at one point it does not halt and wherever it halts it does not output $\mathbf{1}$. Therefore, N passess all the stages before n but in the stage n since there

is no u such that $g(u, \vec{v}) = \mathbf{1}$ and at least there is one u for which M never halts, N does not halt. $\square$

Remark 7.47. Note that for any decidable relation $R(u, \vec{v})$, the function $f(\vec{v}) = \mu|u|.R(u, \vec{v})$ computing $\mathbf{1}^n$ where n is the length of the shortest possible u such that $R(u, \vec{v})$. The reason is that $R(u, \vec{v})$ is equivalent to $\chi_R(u, \vec{v}) = \mathbf{1}$.

Example 7.48. The empty function $\mathit{empty}(w)$ with the empty domain is computable because $\mathit{empty}(w) = \mu|u|.\ [Z(u) = \mathbf{1}]$.

Example 7.49. Let $R \subseteq \mathbb{N}$ be a decidable relation and $f : \mathbb{N} \to \mathbb{N}$ be a computable function over the alphabets $\{1\}$. Then $f|_R$, the restriction of f to the relation R is also computable, because $f|_R(v)$ is definable by $f(\mu|w|.\ [R(w) \wedge (w = v)])$.

Example 7.50. Let $R \subseteq \Sigma^*$ be a decidable relation. Then, it is also c.e., because $\chi'_R(v)$ is definable by $C_\mathbf{1}(\mu|w|.\ [R(w) \wedge (w = v)])$.

Exercise 7.51. Use the μ operator to show that the set $\{w \in \Sigma^* \mid |w| \leq 3\}$ is c.e.

Example 7.52. The function $Twin(n)$ that computes the least $p \geq n$ such that both p and $p + 2$ are primes is computable, because it is definable via unbounded search $Twin(n) = \mu|m|.[Prime(m) \wedge Prime(m + 2) \wedge n \leq m]$. Note that the totality of the function $Twin$ is equivalent to the twin prime conjecture stating the existence of infinitely many pairs of primes $(p, p + 2)$.

Theorem 7.53. *(Kleene)*

(i) A function is computable iff it is constructible from the basic functions, composition, primitive recursion and unbounded search.

(ii) A relation is c.e. iff it is in the form $\exists u[f(u, \vec{v}) = \mathbf{1}]$, where f is primitive recursive.

Over the language $\{1\}^*$, it is also possible to prove a far more powerful characterization of c.e. relations using polynomials instead of arbitrary primitive recursive functions. This characterization was proved by Yuri Matiyasevich, Julia Robinson, Martin Davis and Hilary Putnam, hence its acronym:

Theorem 7.54. *(MRDP) Over the language $\{1\}^*$, a relation is c.e. iff it is in the form $\exists \vec{y} p(\vec{x}, \vec{y}) = 0$ where p is a polynomial.*

So far, we have provided a machine-independent characterization of computable functions and computably enumerable relations. In the following theorem, we reduce the notion of decidability to the notion of computably enumerability to provide a characterization for decidable relations, as well:

Theorem 7.55. *A relation R is decidable iff both R and R^c are c.e. Therefore, R is decidable if there are primitive recursive functions f, g such that $R(\vec{v})$ iff $\exists u[f(u, \vec{v}) = \mathbf{1}]$ iff $\forall w[g(w, \vec{v}) = \mathbf{1}]$.*

Proof. It is enough to prove the first part. The rest follows. For the first part, one direction is proved previously. For the other direction, if both R and R^c are c.e., they have algorithms M and N. For a decision algorithm K for R, run M and N in parallel. More formally, use a 2-taped Turing machine and copy the input on the second tape, as well. Then, run one step of M on the first tape and one step of N on the second tape, alternately, till one of them halts. Since for any $\vec{v}$, either $\vec{v} \in R$ or $\vec{v} \in R^c$. Therefore, one and exactly one of M and N halts and outputs $\mathbf{1}$. If M halts, output $\mathbf{1}$, if N halts, output $\mathbf{0}$. □

8 The Stability Theorem

Turing machines are defined in a way that depends on the given set of alphabets and computability inherits this unintended dependence from them. However, as the reader may expect, we want the notion of comutability independent of all these details. For instance, this would be unsatisfactory if we have a numeral function computable over the binary expansions of the numbers but uncomputable over the usual unary representation. The problem, though, is with the functions themselves. A function is defined over a fixed set of alphabets and it is somewhat meaningless to compare functions with different possible inputs and outputs. To solve this problem, we need a transfering method to transfer functions over one set of alphabets to the functions over another set, respecting computability in both ways. This section is devoted to this kind of *stability*.

Let Σ be a set of alphabets with at least two elements $a, b \in \Sigma$.[8] And let $x \notin \Sigma$ be a new symbol. We want to present a translation from the expressions over the alphabet $\Gamma = \Sigma \cup \{x\}$ into the expressions over the set Σ, in a way that the computable functions over Γ make a one to one

[8]This is not a real restriction. All the results of this section also hold for a singleton Σ. But, the needed techniques must change dramatically and we want to avoid such complicacies in this very short lecture.

correspondence with the computable functions over Σ. For this matter we need two translation functions; one for the encoding process $e : \Gamma^* \to \Sigma^*$ and the other for the decoding process $d : \Sigma^* \to \Gamma^*$. For the encoding function $e : \Gamma^* \to \Sigma^*$, define $e(w_1 w_2 \dots w_n)$ as $u_1 u_2 \dots u_n$ where u_i is defined in the following manner: If $w_i \in \Sigma$, then $u_i = w_i w_i$. If $w_i = x$, then u_i is ab. For instance, if $\Sigma = \{a, b\}$, then $e(axb) = aaabbb$. Note that the function e is total and computable if we consider it as a function over Γ. It simply reads the input and make any letter double. Moreover, if we restrict e to Σ^*, it will again be a very easy computable function. This restricted function is definable over Σ and we denote it by $e' : \Sigma^* \to \Sigma^*$. For the decoding process $d : \Sigma^* \to \Gamma^*$, read a string over Σ. If the length is odd, then just go right forever, and do not halt. Otherwise, Split the input into blocks with length two and compute the inverse of the process that we defined above for each block. If all the blocks behave in an expected manner you can recover the original string over Γ. Otherwise, go right forever and do not halt. The function d is not a total function. It works only on the image of the function e and is computable if we consider it as a function over Γ. Moreover, note that e and d are inverses of each other, i.e., for any $w \in \Gamma^*$ we have $d(e(w)) = w$ and for any $u \in \Sigma^*$ for which d is defined, $e(d(u)) = u$. Restricting d to strings without ab lands in Σ^* and is computable. This function that is clearly the inverse of the function e' is denoted by $d' : \Sigma^* \to \Sigma^*$.

Note that the functions of e and d are intuitively easy, syntactical and "computable". However, they cannot be computable in their technical sense, because the notion of computability is defined over one fixed language and we do not have computability of functions between strings over different alphabets. However, it is possible to formalize this informal computability of e and d via the bridge role they play in transferring the computable functions over Γ to the computable functions over Σ and vice verse:

Theorem 8.1. *(Stability Theorem) Let $\Gamma = \Sigma \cup \{x\}$ where $x \notin \Sigma$. Then a function $f : (\Gamma^*)^k \to \Gamma^*$ is computable iff the function $\tilde{f} : (\Sigma^*)^k \to \Sigma^*$ with the definition $\tilde{f}(\vec{w}) = e(f(d(\vec{w})))$ is computable.*

Proof. Let us assume that $\tilde{f} : (\Sigma^*)^k \to \Sigma^*$ is computable. Then, we show that f is also computable. Since $\tilde{f}$ is computable, thre exists a Turing machine M over Σ that computes $\tilde{f}$. First, we want to change this machine to a machine over Γ. The only thing to do is defining a dummy work for the machine when it reads x. We require the machine to stay in the same state with the same head position. Then clearly, this new machine ignores any letter x in the input and does what $\tilde{f}$ demands. Now, combine this new machine with the machines for e and d when we consider them as computable functions over Γ.

For the converse, assume that f is computable by a machine N over Γ and we want to show that $\tilde{f}$ is computable over Σ. We will define the machine M simulating N in the following way: Read and write everything in pairs. If you read ww for $w \in \Sigma$, then do what N does for w, and if it writes $u \in \Sigma$, write uu, if you read ab, then do what N does for x and if it writes x, write ab. In any case, if you cannot follow this pattern, go right forever. This is easily possible by making all the states of N double. Then, this new machine computes $\tilde{f}$. □

Remark 8.2. Using the stability theorem, it is possible to transfer any computable function over Γ to a computable function over Σ. Sometimes we use this theorem loosely to forget which language we are computing over. For instance, adding one new symbol to the alphabet makes the language powerful enough to talk about all sequences of strings over the original alphabet. It is enough to add the symbol ";" to Σ to represent a sequence $w_1, w_2, \ldots w_n$, where $w_i \in \Sigma^*$, by one string $w_1; w_2; \ldots; w_n$ over $\Sigma \cup \{;\}$. In this case, sometimes, we silently move to a bigger alphabet to have the power to encode sequences. But we actually mean going to the bigger language, doing the computation and then using the stability theorem to come back.

Remark 8.3. The map $e' = e|_{\Sigma^*}$ translates the set $\Sigma^* \subseteq \Gamma^*$ into the set A, consisting of all strings in the form $w_1 w_1 w_2 w_2 \ldots w_n w_n$ where $w_i \in \Sigma$ and $d' = d|_A$ acts as the inverse of e'. Moreover, we know that both of the functions e' and d' are computable over Σ. Therefore, in transferring the functions over Γ to the functions over Σ, if the domain or the range of any variable is $\Sigma^* \subseteq \Gamma^*$, we can use the computable equivalence of A and Σ^* to keep those variables unchanged. For instance, if $f : \Gamma^* \to \Gamma^*$ is computable over Γ, it implies that $\tilde{f}$ is computable over Σ^*. Define $g : \Sigma^* \to \Sigma^*$ as $d' \tilde{f} e'$. Since e' and d' are computable over Σ, then g is also computable over Σ. But $g(w) = d' e \tilde{f} d(e'(w))$. Since $de' = d'e = id$, we have $g(w) = f(w)$. Hence, f is also computable over Σ, meaning that in transfering the function we could keep the variable $w \in \Sigma^*$ and the range $f(w) \in \Sigma^*$ unchanged. Moreover, it shows that in computing a function over Σ, it is possible to extend the language temporarily for conveninece in the computation. It does not change the status of computability of the function.

Remark 8.4. The stability theorem provides a translation pair (e, d), for any pair of alphabet sets. The reason is that using the stability theorem we know how to add or eliminate an element from the alphabet. With these two operations at hand, we can clearly reach any alphabet from any other one.

9 The Universal Machine

Any computable task can be implemented by a Turing machine and hence it is mechanizable. However, this mechanization is not necessarily performed in a uniform manner and for different tasks, we need to develop different machines; one machine for addition, one for multiplication, one for exponentiation and so on. To use the everyday life examples, the situation is similar to the pre-digital era of the single-task machines such as telephones, typing machine, the television and so on. What Alan Turing's game-changing insight added to that story was the existence of one multi-task machine capable of implementing every possible computable task. This machine is called the universal machine and it can be rightfully called the heart of the modern digital age.

A universal machine is a Turing machine that reads a program (a Turing machine itself) and its input and simulates the work of that machine on that input. In this sense, we have one machine to do every possible computation and to mechanize every computable task, it is enough to design one universal machine, once and for all. To have a machine like this, we have to first feed the machine with a program, which means that our first task in this section is encoding Turing machines into strings of the language. Our strategy is the following. We first extend the alphabet from the original set Σ to the enriched alphabet $\bar{\Sigma} = \Sigma \cup \{1, \mapsto, ;, +, -, L, R, (,), [,], \langle, \rangle\}$ to have the power to freely talk about programs and computations. Then, we will use the Stability theorem to transfer whatever we have constructed over this new language to the original language. Therefore, throughout this section, we work in the extended language carelessly. Now, let us formalize all the primitive notions of computations via this new language. There are three fundamental notions that we want to handle; algorithms, configurations and computations:

- Any Turing machine is nothing but a finite table consisting of its basic data and its basic rules. Therefore, it is reasonable to think of the possibility of encoding these machines by some strings over a suitable set of alphabet. The following is one possible way to do that which is by no means canonical. Let M be a Turing machine. Represent M by the string $(I_1; I_2; \ldots; I_n)$ over $\bar{\Sigma}^*$ where I_i's are strings in the form $1^r; a \mapsto 1^s; b; D$ where $r, s \in \mathbb{N}$, $a, b \in \Sigma$ and $D \in \{L, R\}$, encoding the rule that if the machine in the state q_r reads a, it changes the content of the cell to b, goes to the state q_s and moves its head to left or right according to D.

- By the *configuration* of a machine at the moment n, we mean all the data of the machine at the n-th instant of time, namely, the tuple consisting of the *state of the machine*, *its head position* and t*he whole non-blank data over its tape*, at the moment n. The configuration at the moment 0 is called the *starting configuration* and a configuration whose state is in the halting states is called a *halting configuration*. A configuration is also a finite data and hence representable. For instance, we can represent a configuration by $[1^r; 1^s; x; E_1; E_2; \dots, E_m]$ where E_i is a string in the form $(1^j; y; a)$ where $r, s, j \in \mathbb{N}$, $x, y \in \{+, -\}$ and $a \in \Sigma$, encoding that the state is q_r, the head is in the position $+s$ or $-s$ depending on x and the content of the $(+j)$-th cell or $(-j)$-th cell is a, depending on y. For any j not occurring in E_i's, we assume that the content of that cell is blank.

- A *computation* is a sequence of configurations of the machine, starting with the initial configuration and following the rules of the machine. Represent a computation by $\langle C_1; \dots; C_m \rangle$ where any C_i is a representation of the configuration of the machine in the i-th step.

Theorem 9.1. (i) *The relation $T(m, \vec{u}, w, v)$ stating "the machine m on the input $\vec{u}$ has the halting computation w leading to the output v" over the alphabet $\bar{\Sigma}$ is decidable.*

(ii) *There exists a universal Turing machine $U(m, \vec{u})$ simulating all Turing machines, i.e., for any machine M with description m and any input $\vec{u}$ we have $U(m, \vec{u}) = M(\vec{u})$.*

Proof. For (i), define the function $F(x, m, \vec{u})$ as a function over $\bar{\Sigma}$ that sends $(x, m, \vec{u})$ to the computation of m on $\vec{u}$ after $|x|$ many steps. This function is computable via the following natural `Simulation` algorithm that uses recursion on x. For the base case, `Simulation` generates the initial configuration by printing the initial state, the input itself and the head position over the first letter of the input. Then, reading any element of x, first the machine finds the last configuration of m encoded in the output of the last step and then goes to m to see what the rules in m say about changing this last configuration. It then applies what m states to compute the next configuration and finally adds this new configuration to the whole string of computation that it has produced so far. The machine halts when it reads all the elements of x. Now, computing F, we can define $T'(m, \vec{u}, w)$ stating that "the machine m halts on the input $\vec{u}$ with the computation w" as

$$\exists |x| \leq |w| \; [F(x, m, \vec{u}) = w] \wedge hState(m, w)$$

where $hState(m, w)$ means that the last state of the computation w is halting according to m. Note that $hState(m, w)$ is computable because for its decision it is enough to scan both m and w. Then, define the predicate $T(m, \vec{u}, w, v)$ as $T'(m, \vec{u}, w) \wedge (out(w) = v)$ where $out(w)$ computes the output of the computation w. Again note that $out(w)$ is computable because it is enough to scan the computation w to extract the output of the computation m from it as the tape data of the last configuration. For (ii), define $U(m, \vec{u}) = out(F(x_{(m,\vec{u})}, m, \vec{u}))$ where $x_{(m,\vec{u})} = \mu|x|.\ hState(m, F(x, m, \vec{u}))$. □

Remark 9.2. Using the Stability under the language extensions, both of the predicate T and the universal machine U can be transferred over Σ via the suitable encoding-decoding process.

The previous theorem can be rewritten more carefully to serve also as a proof for the Kleene's machine-independent characterization theorem:

Proof. For (i), one direction is already proved. For the other direction, note that a more detailed investigation shows that what we explained before is actually a primitive recursive construction for the predicate T. Therefore, the universal machine is constructible via Kleene's operations, because it applies one unbounded search on the predicate T. Therefore, if a function $f(\vec{u})$ is computable via a Turing machine M with the code m, then $f(\vec{u}) = U(m, \vec{u})$. Since U is constructible, f is also constructible. For (ii), again one direction is proved. For the other direction, assume that the machine for R is M with the code m. Then, note that $R(\vec{u})$ if $\exists w\ T(m, \vec{u}, w, \mathbf{1})$. This completes the proof. Note that this proof works over the extended language $\bar{\Sigma}$. To move it over the original language Σ, we also need a primitive recursive version of the translations in the Stability theorem. This is also possible, but it needs more work. □

On Computably Enumerable Relations

Using what we have developed in this section, we can explain the terminology that we use for c.e. relations:

Theorem 9.3. *A set $A \subseteq \Sigma^*$ is c.e. iff it is either empty or the range of a total computable function $f : \{\mathbf{1}\}^* \to \Sigma^*$.*

Proof. For simplicity, identify $\{\mathbf{1}\}^*$ with $\mathbb{N}$ and represent $\mathbf{1}^n$ by n. Then, assume $A \neq \emptyset$ and M is the machine for A with the code m. Pick $a \in A$ and consider the predicate $R(u, w) = T(m, u, w, \mathbf{1})$ representing that "w is a computation of the machine M on the input u that halts and outputs $\mathbf{1}$". Let $en : \Sigma^* \to \Sigma^*$ be a total computable function whose restriction to $\{\mathbf{1}\}^*$ is surjective. Now, define $f : \{\mathbf{1}\}^* \to \Sigma^*$ as

$$f(n) = \begin{cases} u_n & R(u_n, v_n) \\ a & \neg R(u_n, v_n) \end{cases}$$

where $en(n_0) = u_n$ and $en(n_1) = v_n$ where $n + 1 = 2^{n_0}(2n_1 + 1)$. Firstly, the range of f is clearly a subset of A, because in the first case, v_n is the computation of the machine M on u_n with answer $\mathbf{1}$, meaning that $u_n \in A$. The second case is clear because $a \in A$. Conversely, if $b \in A$, then M halts on b and outputs $\mathbf{1}$. Take w as the computation of M on b. Then, since the restriction of en to $\{\mathbf{1}\}^*$ is surjective, there exist m, k such that $en(m) = b$ and $en(k) = w$. Therefore, $f(n) = b$ where $n + 1 = 2^m(2k + 1)$. □

Remark 9.4. The Theorem 9.3 states that a set A is c.e. if it is either empty or there exists a total computable function with range A, surjective even when it is restricted to $\{\mathbf{1}\}^*$. In other words, f reads a number n and produces an element of A. Since $f|_{\{\mathbf{1}\}^*}$ is surjective, it covers A (with possible repetitions). This means that $A = \{f(0), f(1), f(2), \ldots\}$ meaning that f enumerates all the elements of A in a computable manner; hence the name computably enumerable.

Exercise 9.5. Let $f : \mathbb{N} \to \mathbb{N}$ be a strictly increasing total computable function. Then, show that the set $\{f(n) \mid n \in \mathbb{N}\}$ is decidable.

10 The Uncomputable World

In this section we will present some negative results including an uncomputable function, an undecidable relation and a relation that is c.e. but not decidable. Our method is essentially different from what we did before in the pumping lemma for the regular languages. Here, the computability notion is extremely powerful. Therefore, it seems impossible to prove an undecidability result by investigating all the possible patterns of computation. However, this full notion of computation is so powerful that it can speak about itself, as we observed in the previous section. This power then leads to the usual self-referential paradoxes, this time encoded via computations. This self-referential type of argument is the main technique of proving undecidabilities. Let us recall the *diagonalization method* by reproving the fact that there is no surjective function from $\mathbb{N}$ to the set of all infinite sequences of natural numbers. Assume that such function exists and call it f. Then, we can use it to enumerate all possible sequences as in the following diagram, where the sequence in the ith row is $f(i)$:

1	n_{11}	n_{12}	n_{13}	n_{14}	n_{15}	n_{16}	$\cdots$
2	n_{21}	n_{22}	n_{23}	n_{24}	n_{25}	n_{26}	$\cdots$
3	n_{31}	n_{32}	n_{33}	n_{34}	n_{35}	n_{36}	$\cdots$
4	n_{41}	n_{42}	n_{43}	n_{44}	n_{45}	n_{46}	$\cdots$
5	n_{51}	n_{52}	n_{53}	n_{54}	n_{55}	n_{56}	$\cdots$
6	n_{61}	n_{62}	n_{63}	n_{64}	n_{65}	n_{66}	$\cdots$
7	n_{71}	n_{72}	n_{73}	n_{74}	n_{75}	n_{76}	$\cdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Now define the sequence $s = \{s_i\}_{i \in \mathbb{N}}$ by $s_i = f(n)_i + 1 = n_{ii} + 1$. This sequence can not be any of $f(i)$'s, because if it is then $s = f(n)$ and hence $s_n = f(n)_n$, while s_n is defined as $f(n)_n + 1$. The method is called diagonalization as it uses the diagonal in the previous diagram to produce a violating sequence. In the following theorem we use the same technique to show that the halting predicate is not decidable:

Theorem 10.1. *The halting relation $H \subseteq \Sigma^* \times \Sigma^*$ defined as $H(m, u) = \exists wv T(m, u, w, v)$ is c.e. but not decidable. Hence, the total function χ_H is not computable.*

Proof. It is c.e. because it has the description $\exists wv T(m, u, w, v)$ where the predicate $T(m, u, w, v)$ is the decidable relation that states "w is the computation of the machine m on the input u halting and outputting v". For undecidability, let us assume that there exists a Turing machine M to decide H. Therefore, M always halts. Define the machine $N(x)$ as the machine that first compute $M(x, x)$ and then if it is one, it does not halt and if it is zero, it halts and outputs zero. Now, let us check $N(n)$ where n is the code of N. If N halts on n, then by definition of N, we have $M(n, n) = 0$ which means that N does not halt on n. If N does not halt on n, then again by definition $M(n, n) = 1$ which means N halts on n. $\square$

Corollary 10.2. *The set H^c is not computably enumerable.*

Proof. If it is computably enumerable, then since H is also computably enumerable, by Theorem 7.55, H will be decidable which we know it is not the case. $\square$

Remark 10.3. Note that the previous corollary shows that the class of c.e. relations is not closed under complement.

Exercise 10.4. Prove or disprove: If $f : \mathbb{N}^k \to \mathbb{N}$ is a total function and there exists a number $k \in \mathbb{N}$ such that for any $\vec{n} \in \mathbb{N}^k$ we have $f(\vec{n}) \leq k$, then f is computable.

Exercise 10.5. Is there a total computable function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ such that an algorithm M halts on the input $\vec{n}$ iff it halts on $\vec{n}$ in $f(m, \vec{n})$ steps, where m is the description of the algorithm M?

Exercise 10.6. Prove that there is no total computable function $f : \mathbb{N} \to \mathbb{N}$ such that $\{f(n) \mid n \in \mathbb{N}\}$ becomes the set of the descriptions of all Turing machines computing total computable functions on $\mathbb{N}$.

Exercise 10.7. Show that the relation $\{m \mid m \text{ never halts}\}$ is not decidable.

Exercise 10.8. Show that the relation

$$\{(m, n) \mid m \text{ and } n \text{ halts on the same elements}\}$$

is not decidable.

Exercise 10.9. Show that the relation

$$\{m \mid m \text{ computes } \chi'_R \text{ for a finite } R \subseteq \Sigma^* \}$$

is not decidable.

In the following, we will generalize what we had for the halting property of a machine to a very general setting. We will show that any non-trivial property of the machines that depends on the function that the machine computes and not the implementation details (like its number of states) is not decidable.

Definition 10.10. Let C be a set of Turing machines. It is called *extensional* if for any computable function f, either C has all the machines for f or none of them.

Theorem 10.11. *(Rice Theorem) Let C be an extensional set of Turing machines. Then, C is either undecidable or trivial, i.e., either empty or the set of all Turing machines.*

Proof. Assume on the contrary that C is decidable and non-trivial. Since C is extensional, either C has all the machines computing the function with the empty domain or C has none of them. W.l.o.g. we assume the latter, because in the first case we can use C^c in the rest of the argument. Now, we use the decidability of C to show that the halting relation H is also decidable. Since C is not trivial, there is at least one $K \in C$. The decision algorithm for halting is the following: Read a machine M and an input w. Then construct the machine $[M, w]$ with the input u and the following algorithm: Run M on w. If it halts, run K on u. It is clear that $[M, w]$ computes the same function as K does if M halts on w. Otherwise, the domain of $[M, w]$ is empty. Since C has none of the machines computing the function with the empty domain and $K \in C$, we have $[M, w] \in C$ iff M halts on w. Finally, we can run the decision procedure for C on $[M, w]$. $\square$

Example 10.12. The class of all Turing machines halting on all of their inputs is not decidable as it is extensional and non-trivial.

Exercise 10.13. Show that the class of all Turing machines computing the constant functions is not decidable.

It is satisfying to end this section by the negative solution for Hilbert's tenth problem. It heavily depends on the elegant MRDP theorem that we stated before:

Theorem 10.14. *(Unsolvability of Hilbert's Tenth Problem) There is no algorithm to decide whether a polynomial with integer coefficients has an integer root or not.*

Proof. By MRDP, the halting relation $H(x, y)$ is equivalent to a statement like $\exists \vec{z} \in \mathbb{N}\, p(x, y, \vec{z}) = 0$. Define $q(x, y, \vec{z}, \vec{w})$ as

$$p(x, y, \vec{z})^2 + \Sigma_i (z_i - w_{i1}^2 - w_{i2}^2 - w_{i3}^2 - w_{i4}^2)^2$$

Then $\exists \vec{z} \in \mathbb{N}\ p(x, y, \vec{z}) = 0$ iff $\exists \vec{z}\vec{w} \in \mathbb{Z}\ q(x, y, \vec{z}, \vec{w}) = 0$. The main reason behind this equivalence is the Lagrange theorem stating that any natural number is representable as the sum of four perfect squares. Finally, if there exists an algorithm to decide the latter, then it means we can decide the former and hence the halting problem which is impossible. $\square$

Finally, let us mention some other concrete undecidable problems. There are many of them scattered in different fields of mathematics, from topology and analysis to algebra and combinatorics. In the following we only mention two problems of this kind:

Example 10.15. (*The Matrix Mortality Problem*) Use a reasonable language to talk about matrices with integer entries, for instance $\{1, ;\}$. Then, in this language, pick the set of all finite sets of $n \times n$ matrices with integer entries such that there exists a multiplication of them in some order, possibly with repetitions, to yield the zero matrix. This set is c.e. as it is definable via the unbounded existential quantifier (the list of multiplications) over a decidable relation "checking whether the multiplication is zero". The latter is decidable, because it only needs to follow the list of multiplications, compute them and then check whether it is zero or not. The set is undecidable, but its proof is beyond the scope of our lectures.

Example 10.16. (*Post Correspondence Problem*) Fix Σ with at least two elements and use $\Sigma \cup \{1, ;\}$. Then, pick the set of all pairs of finite lists of strings over Σ such as $\{a_i\}_{i=1}^{n}$ and $\{b_i\}_{i=1}^{n}$ with the same length such that there exists a list of indices like $\{i_j\}_{j=1}^{m}$ where $1 \leq i_j \leq n$ such that $a_{i_1}a_{i_2}\dots a_{i_m} = b_{i_1}b_{i_2}\dots b_{i_m}$. This set is c.e. as it is definable by an unbounded existential quantifier on the list of indices over a decidable relation "over the given list $\{i_j\}_{j=1}^{m}$, we have $a_{i_1}a_{i_2}\dots a_{i_m} = b_{i_1}b_{i_2}\dots b_{i_m}$". The latter is decidable, because it only needs to follow the list to put a_{i_j}'s and b_{i_j}'s and then check whether they are equal or not that needs only a simple scanning. The relation is undecidable, but its proof is beyond the scope of our lectures.

11 Lambda Calculus

We saw that the universal machine is powerful enough to simulate all Turing machines. Now, the natural question is: Is it possible to develop a machine-independent language using just the universal machine? The answer is yes and the result is called *lambda calculus*.

Definition 11.1. Let V be a countable set of variable symbols. Then, the set of λ-terms is defined inductively in the following way:

- Any element of V is a λ-term. These are called the *variables*.
- If M is a λ-term and x is a variable, then $(\lambda x.M)$ is also a λ-term. The λ-term $(\lambda x.M)$ is called the *λ-abstraction of M with respect to x.*
- If M and N are both λ-terms, then (MN) is also a λ-term. The λ-term (MN) is called the *application of M to N.*

We can read the λ-terms as the codes for Turing machines, the application MN as the universal machine $U(M, N)$ and the abstraction as a program that reads x and returns M. Note that in this setting there is not a priori data type. Everything by design is both an algorithm and an input as you can observe in the application (MN).

Example 11.2. The expressions $(\lambda x.x)$, (xx) and $(\lambda f.(\lambda x.(fx)))$ are all λ-terms.

Convention. To read the λ-terms easily, we use the following conventions:

- Omit the outermost parentheses. For instance, we write MN for (MN).

- Associate the applications to the left. For instance, MNP means $(MN)P$. This is convenient when applying a function to a number of arguments, as in $fxyz$, which means $((fx)y)z$.
- The body of a lambda abstraction (the part after the dot) extends as far to the right as possible. In particular, $\lambda x.MN$ means $\lambda x.(MN)$, and not $(\lambda x.M)N$.
- Multiple lambda abstractions can be contracted; thus $\lambda xyz.M$ will abbreviate $\lambda x.\lambda y.\lambda z.M$.
- We use the symbol $\equiv$ as the syntactical equality meaning that a λ-term is just equal to itself.

Definition 11.3. Define the set of free variables of a λ-term M, denoted by $FV(M)$, recursively by:

- $FV(x) = \{x\}$,
- $FV(MN) = FV(M) \cup FV(N)$,
- $FV(\lambda x.M) = FV(M) - \{x\}$.

Definition 11.4. A relation $\sim$ on λ-terms is called a congruence if it is an equivalence and has the property that if $M \sim N$, then $MK \sim NK$, $KM \sim KN$ and $\lambda x.M \sim \lambda x.N$, for any λ-term K and any variable x.

Definition 11.5. We define α-equivalence to be the smallest congruence relation on λ-terms, such that for all λ-terms M and all variables y that do not occur in M, the λ-terms $\lambda x.M$ and $\lambda y.M[x/y]$ are considered congruent, i.e., $\lambda x.M =_\alpha \lambda y.M[x/y]$

Clearly, the name of the variable x in the λ-term $\lambda x.M$ is not important. The α-equivalence reads terms up to the name change of these occurrences of the variables.

Definition 11.6. Let M, P be two λ-term and $x \in FV(M)$. Define the substitution of a λ-term N for x in M inductively as:

- $x[x/N] \equiv N$ and $y[x/N] \equiv y$, when $y \neq x$,
- $(MP)[x/N] \equiv M[x/N]P[x/N]$,
- $(\lambda x.M)[x/N] \equiv \lambda x.M$.
- $(\lambda y.M)[x/N] \equiv \lambda y.(M[x/N])$, if $y \neq x$ and $y \notin FV(N)$,

- $(\lambda y.M)[x/N] \equiv \lambda z.(M[z/y])[x/N]$, if $y \neq x$ and $y \in FV(N)$ and z is a new variable that does not occur in M and N.

Note that the variable z in the last part can be any fresh variable and hence it seems that the substitution operation is not well-defined. However, as we consider the λ-terms up to the α-equivalence, the problem with the name of z disappears.

11.1 Computation in Lambda Calculus

Reading λ-terms as Turing machines, it is reasonable to assume that computation in this setting is the simplification of λ-terms by changing $(\lambda x.M)N$ to $M[x/N]$.

Definition 11.7. (*β-reduction*) By a β-reduction we mean the following transformation to any subterm of a λ-term: $(\lambda x.M)N \to_\beta M[x/N]$. We write $M \twoheadrightarrow_\beta N$ if M reduces to N in zero or more steps of β-reductions. In fact, $\twoheadrightarrow_\beta$ is the reflexive transitive closure of $\to_\beta$. Similarly, define β-equivalence as the reflexive, symmetric and transitive closure of $\to_\beta$ and denote it by $=_\beta$.

Example 11.8. We have

$$(\lambda x.x)M \to_\beta M \qquad (\lambda y.y)N \to_\beta N$$

and hence

$$((\lambda x.x)M)((\lambda y.y)N) \twoheadrightarrow_\beta MN$$

Moreover,

$$((\lambda x.x)M)((\lambda y.y)N) =_\beta M((\lambda y.y)N) =_\beta ((\lambda x.x)M)N =_\beta MN$$

Example 11.9. The λ-term $\mathbf{I} \equiv \lambda x.x$ is a program reading that reads an input x and returns it. For any λ-term M, we also have the constant program $\lambda x.M$ reading x and returning M, regardless of the input. What happens to the multi-variable functions? For instance, if f reads both x and y, then we interpret it as a function that reads x and return the function which reads y to return $f(x, y)$. For instance, the program $\mathbf{I^i_n} \equiv \lambda x_1 \cdots x_n.x_i$ reads x_1 and then x_2 till x_n to return x_i.

A λ-term on which no β-reduction is applicable is called *normal* or *in normal form.* For instance, the λ-terms x, $\lambda x.x$ are normal while $(\lambda x.x)M$ is not. The normal λ-terms can be interpreted as the values and if $M \twoheadrightarrow_\beta N$ where N is normal, we can say that M has been evaluated to the value N.

The natural question to ask is whether all λ-terms have a value? If they do, is this value unique? As we expect from any formalization of computability, there must be some sort of partiality here. It is actually the case. There are some λ-terms that are no reducible to a normal λ-term and their reducing process does not halt. For instance, consider the λ-term $(\lambda x.xx)(\lambda x.xx)$. This λ-term is clearly not normal, becuase we can apply the following reduction:

$$(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$$

However, as this is the only reduction we can peform and the result of the reduction is the λ-term itself, it is clear that there can not be any reduction process to a normal λ-term.
So far, we saw that some of the λ-terms have no values. Here, it is important to mention another point. Even if a λ-term has a value, it does not mean that any reduction strategy halts. For instance, consider the following λ-term $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))z$. By using reduction on the outer lambda we have

$$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))z \to_\beta z$$

Since z is normal, we can claim that the λ-term has a value. However, if we use reduction over the inner lambda we reach itself and if we keep reducing the inner lambda the process never halts.

In general, there are different reduction strategies some of which may lead to a normal λ-term while the others may not. However, the most common strategy, the *natural strategy* that always picks the left-most possible lambda to reduce, leads to a normal λ-term iff the λ-term has a normal form.
The last question is that if a λ-term reduces to a normal λ-term, is this λ-term unique? We come back to this question later. Before that a bit of programming.

Exercise 11.10. Let M and N be two λ-terms in which x, y, a and b do not occur. Simplify the following λ-terms: $(\lambda xy.x)MN$, $(\lambda xy.y)MN$, $(\lambda ab.abM)(\lambda xy.x)N$ and $\lambda xy.(\lambda fx.x)(\lambda z.y)x$.

Exercise 11.11. Write down a λ-term F such that $FMN =_\beta M(NM)N$, for all λ-terms M and N.

11.2 Basic Data Types and Programs

As everything in lambda calculus is a program, we have to encode the data types with the programs to be able to perform the computations. Let us start with the booleans:

Example 11.12. (*Booleans*) To define the booleans, we have to first represent the truth values by some λ-terms and then we have to define some other λ-terms to represent the basic boolean operations including the conjunction and the negation. For the truth values we use: $\mathbf{T} \equiv \lambda xy.x$ and $\mathbf{F} \equiv \lambda xy.y$. Then, if we define $\mathbf{and} \equiv \lambda ab.ab\mathbf{F}$, the following reductions show that **and** actually represents the conjunction:

$$\mathbf{and}\ \mathbf{TT} \twoheadrightarrow_\beta (\lambda ab.ab\mathbf{F})\mathbf{TT} \twoheadrightarrow_\beta \mathbf{TTF} \twoheadrightarrow_\beta (\lambda xy.x)\mathbf{TF} \twoheadrightarrow_\beta \mathbf{T}$$

$$\mathbf{and}\ \mathbf{TF} \twoheadrightarrow_\beta (\lambda ab.ab\mathbf{F})\mathbf{TF} \twoheadrightarrow_\beta \mathbf{TFF} \twoheadrightarrow_\beta (\lambda xy.x)\mathbf{FF} \twoheadrightarrow_\beta \mathbf{F}$$

$$\mathbf{and}\ \mathbf{FT} \twoheadrightarrow_\beta (\lambda ab.ab\mathbf{F})\mathbf{FT} \twoheadrightarrow_\beta \mathbf{FTF} \twoheadrightarrow_\beta (\lambda xy.y)\mathbf{TF} \twoheadrightarrow_\beta \mathbf{F}$$

$$\mathbf{and}\ \mathbf{FF} \twoheadrightarrow_\beta (\lambda ab.ab\mathbf{F})\mathbf{FF} \twoheadrightarrow_\beta \mathbf{FFF} \twoheadrightarrow_\beta (\lambda xy.y)\mathbf{FF} \twoheadrightarrow_\beta \mathbf{F}$$

The negation is also definable as $\mathbf{neg} \equiv \lambda a.a\mathbf{FT}$. Then, we again need the following computations:

$$\mathbf{neg}\ \mathbf{T} \twoheadrightarrow_\beta (\lambda a.a\mathbf{FT})\mathbf{T} \twoheadrightarrow_\beta \mathbf{TFT} \twoheadrightarrow_\beta (\lambda xy.x)\mathbf{FT} \twoheadrightarrow_\beta \mathbf{F}$$

$$\mathbf{neg}\ \mathbf{F} \twoheadrightarrow_\beta (\lambda a.a\mathbf{FT})\mathbf{F} \twoheadrightarrow_\beta \mathbf{FFT} \twoheadrightarrow_\beta (\lambda xy.y)\mathbf{FT} \twoheadrightarrow_\beta \mathbf{T}$$

More generally, it is not hard to see that if B is a boolean, then BPQ is *"if B then P, else Q*. Therefore, we can define $\mathbf{if\text{-}then\text{-}else} \equiv \lambda xyz.xyz$.

Example 11.13. (*Pairing*) Using the λ-term **if-then-else** it is now easy to encode the pair $\langle M, N\rangle$ as the conditional $\langle M, N\rangle \equiv \lambda z.$ ($\mathbf{if}\ z\ \mathbf{then}\ M\ \mathbf{else}\ N$) $\equiv \lambda z.zMN$. Then, if we define $\mathbf{p_0} \equiv \lambda p.p\mathbf{T}$ and $\mathbf{p_1} \equiv \lambda p.p\mathbf{F}$, we have

$$\mathbf{p_0}\ \langle M, N\rangle \twoheadrightarrow_\beta (\lambda p.p\mathbf{T})\ \langle M, N\rangle \twoheadrightarrow_\beta \langle M, N\rangle\mathbf{T} \twoheadrightarrow_\beta (\lambda z.zMN)\mathbf{T} \twoheadrightarrow_\beta \mathbf{T}MN \twoheadrightarrow_\beta M$$

and

$$\mathbf{p_1}\ \langle M, N\rangle \twoheadrightarrow_\beta (\lambda p.p\mathbf{F})\ \langle M, N\rangle \twoheadrightarrow_\beta \langle M, N\rangle\mathbf{F} \twoheadrightarrow_\beta (\lambda z.zMN)\mathbf{F} \twoheadrightarrow_\beta \mathbf{F}MN \twoheadrightarrow_\beta N$$

The λ-terms $\mathbf{p_0}$ and $\mathbf{p_1}$ act as the projection functions associated to the pairing operation.

Example 11.14. (*Numbers*) To represent the natural numbers by λ-terms, one clever and useful idea is encoding the number n with the *function that iterates a given function n many times*. This encoding is called the nth *Church numeral* defined as $\bar{n} \equiv \lambda fx.f^n(x)$, where $f^n(x)$ is $f(f(\cdots(fx)\cdots))$

with n many f's. Using this encoding for natural numbers, then we can define the successor function as **Succ** $\equiv \lambda nfx.f(nfx)$. By the following reductions we can see that this λ-term captures the behavior of the successor function:

$$\mathbf{Succ}\ \bar{n} \equiv (\lambda nfx.f(nfx))(\lambda fx.f^n x) \to_\beta \lambda fx.f((\lambda fx.f^n x)fx)$$

$$\to_\beta \lambda fx.f(f^n x) \equiv \lambda fx.f^{n+1}x \equiv \overline{n+1}$$

Moreover, we can define the addition and the multiplication functions by the λ-terms **Add** $\equiv \lambda nmfx.nf(mfx)$ **Mult** $\equiv \lambda nmf.n(mf)$. The first interprets the addition of m and n as "*n iterations after m iterations*" and the second interprets the multiplication of m and n as "*n many iterations of the m iterations.*"

Exercise 11.15. Show that the **Add** $\bar{2}\ \bar{3} =_\beta \bar{5}$.

Exercise 11.16. Show that the λ-term **Succ′** $\equiv \lambda nfx.nf(fx)$ represents the successor function.

Exercise 11.17. Find a λ-term that represents the function $Exp(m,n) = m^n$.

Example 11.18. (*Wisdom teeth trick*) How to represent the predecessor function? Actually, Church himself thought that it may be impossible to represent this function, till his student Kleene found out how to do it, while having his wisdom teeth pulled, so his trick for defining the predecessor function is sometimes referred to as the "*wisdom teeth trick*". Let us explain the trick. The idea is that the λ-term $\bar{n}$ represents the operation of n many iterations. So, if we do these many iterations on an appropriate function, we might find a way to represent the predecessor function. Let S be the λ-term that maps a pair (m,l) to $(l, l+1)$. Then, if we iterate S, n many times on $(0,0)$ we get $(n-1,n)$. Then, by projection, we can easily extract $n-1$. More precisely, define $S \equiv \lambda p.\langle \mathbf{p_1}p, \mathbf{Succ}(\mathbf{p_1}p)\rangle$. Then, **Pred** $\equiv \lambda n.\mathbf{p_0}(nS\langle\bar{0},\bar{0}\rangle)$.

Example 11.19. (*Subtraction*) Define **Psub** $\equiv \lambda xy.y$ **Pred** x. Then, we have

$$\mathbf{Psub}\ \bar{m}\ \bar{n} \twoheadrightarrow_\beta \bar{n}\ \mathbf{Pred}\ \bar{m} \twoheadrightarrow_\beta \overline{Psub(m,n)}$$

Example 11.20. Define **IsZero** $\equiv \lambda nxy.n(\lambda z.y)x$ as a λ-term that checks whether a given Church numeral is zero or not. More precisely, we have

$$\mathbf{IsZero}\ \bar{0} \equiv (\lambda nxy.n(\lambda z.y)x)\bar{0} \twoheadrightarrow_\beta \lambda xy.\bar{0}(\lambda z.y)x \twoheadrightarrow_\beta \lambda xy.x \equiv \mathbf{T}$$

and for $n+1$ we have

$$\mathbf{IsZero}\ \overline{n+1} \equiv (\lambda nxy.n(\lambda z.y)x)\overline{n+1} \twoheadrightarrow_\beta \lambda xy.\overline{n+1}(\lambda z.y)x \twoheadrightarrow_\beta \lambda xy.y \equiv \mathbf{F}$$

Example 11.21. (*Equality and inequality*) Define $\mathbf{IsLeq} \equiv \lambda xy.\mathbf{IsZero}\ (\mathbf{Psub}yx)$. This defines $m \leq n$. Therefore, for equality we have

$$\mathbf{Eq} \equiv \lambda xy.(\mathbf{and}\ (\mathbf{IsLeq}xy)\ (\mathbf{IsLeq}yx)).$$

11.3 Representing All Computable Functions

In this section we prove that all computable functions over natural numbers are representable in lambda calculus. To prove that, we use Kleene's machine-independent characterization from the previous section.

Definition 11.22. Suppose $f : \mathbb{N}^k \to \mathbb{N}$ is a k-ary function and F is a λ-term. We say that F represents f when for any $n_1, \ldots, n_k \in \mathbb{N}$, if $(n_1, \cdots, n_k)$ is in the domain of f, we have $F\bar{n_1} \cdots \bar{n_k} =_\beta \overline{f(n_1, \cdots, n_k)}$ and if it is not in the domain, the λ-term $F\bar{n_1} \cdots \bar{n_k}$ has no normal form. In this situation the function f is called representable in lambda calculus.

Theorem 11.23. *All total computable functions are representable in lambda calculus.*

For the basic functions we just use $\mathbf{Z} \equiv \lambda x.\bar{0}$, $\mathbf{Succ} \equiv \lambda nfx.f(nfx)$ and $\mathbf{I^i_n} \equiv \lambda x_1 \cdots x_n.x_i$. For composition, if $h(\vec{x}) = f(g_1(\vec{x}), \cdots, g_k(\vec{x}))$ and $\mathbf{F}$ and $\mathbf{G_i}$'s are the representing λ-terms for f and g_i's, respectively, then it is enough to define $\mathbf{H}$ as $\lambda\vec{x}.\mathbf{F}(\mathbf{G_1}\vec{x})(\mathbf{G_2}\vec{x})\cdots(\mathbf{G_k}\vec{x})$. Now, we have to address the other two operations: the primitive recursion and the unbounded search. Let us first explain the idea with an example. Consider the primitive recursive function $e(n) = 2^n$, defined by the recursive definition: $e(0) = 1$ and $e(n + 1) = e(n) + e(n)$. To find a λ-term $\mathbf{e}$ representing e, it is enough to find $\mathbf{e}$ such that

$$\mathbf{e} =_\beta \lambda x.\mathbf{if}\ \mathbf{IsZero}(x)\ \mathbf{then}\ \bar{1}, \mathbf{else}\ (\mathbf{Add}\ \mathbf{e}(\mathbf{Pred}\ x)\ \mathbf{e}(\mathbf{Pred}\ x))$$

because then we can see that

$$\mathbf{e}\ \bar{0} =_\beta \mathbf{if}\ \mathbf{IsZero}(\bar{0})\ \mathbf{then}\ \bar{1}, \mathbf{else}\ (\mathbf{Add}\ \mathbf{e}(\mathbf{Pred}\ \bar{0})\ \mathbf{e}(\mathbf{Pred}\ \bar{0})) =_\beta \bar{1}$$

and

$$\mathbf{e}\ \overline{n+1} =_\beta \mathbf{if}\ \mathbf{IsZero}(\overline{n+1})\ \mathbf{then}\ \bar{1}, \mathbf{else}\ (\mathbf{Add}\ \mathbf{e}(\mathbf{Pred}\ \overline{n+1})\ \mathbf{e}(\mathbf{Pred}\ \overline{n+1}))$$

$$=_\beta (\mathbf{Add}\ \mathbf{e}(\mathbf{Pred}\ \overline{n+1})\ \mathbf{e}(\mathbf{Pred}\ \overline{n+1}))$$

Now, if we define $M \equiv \lambda x.\mathbf{if}\ \mathbf{IsZero}(x)\ \mathbf{then}\ \bar{1}, \mathbf{else}\ \mathbf{Add}\ (y(\mathbf{Pred}\ x)\ y(\mathbf{Pred}\ x)$ then the λ-term $\mathbf{e}$ can be set as a fixed point of M. The need for such a fixed point leads us to the following theorem:

Theorem 11.24. *(Fixed Point Theorem) Let $A \equiv \lambda xy.y(xxy)$ and define $\Theta \equiv AA$. If F is a λ-term then $N \equiv \Theta F$ is a fixed point of F, i.e., $FN =_\beta N$.*

Proof. This is shown by the following calculation: $N \equiv \Theta F \equiv AAF \equiv (\lambda xy.y(xxy))AF =_\beta F(AAF) \equiv F(\Theta F) \equiv FN$. $\square$

Exercise 11.25. Write down a λ-term F representing the Ackermann function.

Exercise 11.26. Show that the $\mathbf{e}\ \bar{2} =_\beta \bar{4}$.

For primitive recursion, if f is defined by g and h in the following way:

$$\begin{cases} f(0, \vec{y}) = g(\vec{y}) \\ f(x+1, \vec{y}) = h(x, \vec{y}, f(x, \vec{y})) \end{cases}$$

and g and h are representable by $\mathbf{G}$ and $\mathbf{H}$, respectively, then we can represent f by any λ-term $\mathbf{F}$ satisfying

$$\mathbf{F} =_\beta \lambda x\vec{y}.(\textbf{if IsZero}(x) \textbf{ then } \mathbf{G}\vec{y}, \textbf{else } (\mathbf{H}\ x\ \vec{y}\ (\mathbf{F}\ (\textbf{Pred}\ x)\ \vec{y})))$$

This is possible by defining $\mathbf{F}$ as the fixed point of the operator

$$\mathbf{M}z \equiv \lambda x\vec{y}.(\textbf{if IsZero}(x) \textbf{ then } \mathbf{G}\vec{y}, \textbf{else } (\mathbf{H}\ x\ \vec{y}\ (z\ (\textbf{Pred}\ x)\ \vec{y})))$$

Exercise 11.27. Without using the fixed points, prove the primitive recursive case of the theorem.
Hint: Define j as $j(n, \vec{m}) = (n, \vec{m}, f(n, \vec{m}))$ and represent j as n many iterations of some appropriate λ-term.

Finally, it is remained to represent the μ operator. Assume g is representable by the λ-term $\mathbf{G}$ and $f(\vec{y})$ is defined as $\mu x.[g(x, \vec{y}) = 1]$, then

$$\mathbf{H} =_\beta \lambda x\vec{y}.(\textbf{if IsEq}(\mathbf{G}x\vec{y}, \bar{1}) \textbf{ then } x, \textbf{else } \mathbf{H}\ (\textbf{Succ}\ x)\ \vec{y})$$

It is not hard to see that $\mathbf{H}\bar{0}\vec{y}$ checks whether $\textbf{IsEq}(\mathbf{G}\bar{0}\vec{y}, \bar{1})$, if yes it return $\bar{0}$, otherwise it goes one level higher and sets $\mathbf{H}\bar{0}\vec{y}$ as $\mathbf{H}\bar{1}\vec{y}$. Then, it does the same procedure to check whether $\textbf{IsEq}(\mathbf{G}\bar{1}\vec{y}, \bar{1})$, if yes it return $\bar{1}$, otherwise it goes one level higher and sets $\mathbf{H}\bar{1}\vec{y}$ as $\mathbf{H}\bar{2}\vec{y}$ and it keeps going till finding the first n such that $\textbf{IsEq}(\mathbf{G}\bar{n}\vec{y}, \bar{1})$ and then we know that the value $\mathbf{H}\bar{0}\vec{y}$ must be $\mathbf{H}\bar{n}\vec{y}$ that is $\bar{n}$. Therefore, it is enough to define $\mathbf{F}$ as $\lambda\vec{y}.\mathbf{H}\bar{0}\vec{y}$.

11.4 Church-Rosser Property

In this section we study the general behavior of computation namely the reductions in lambda calculus. The main goal is to show the coherency condition that if a λ-term is reducible to a normal form, then the normal λ-terms is unique.

Theorem 11.28. *(Church-Rosser) Suppose M, N, and P are λ-terms such that $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta P$. Then there exists a λ-term Q such that $N \twoheadrightarrow_\beta Q$ and $P \twoheadrightarrow_\beta Q$.*

Corollary 11.29. *If $M =_\beta N$, then there exists some P such that $M, N \twoheadrightarrow_\beta P$.*

Proof. If $M =_\beta N$, then there exists a sequence of λ-terms M_0, M_1, $\cdots$, M_n such that $M \equiv M_0$, $M_n \equiv N$ and for any $0 \leq i < n$, either $M_i \to_\beta M_{i+1}$ or $M_{i+1} \to_\beta M_i$. Use induction on n to prove the claim. For $n = 0$, we have $M \equiv N$ and there is nothing to prove. For $n + 1$, by induction hypothesis there exists Q such that $M \twoheadrightarrow_\beta Q$ and $M_n \twoheadrightarrow_\beta Q$. Then, if $M_{n+1} \to_\beta M_n$, there is nothing to prove as $M_{n+1} \twoheadrightarrow_\beta Q$. If $M_n \to_\beta M_{n+1}$, then by Church Rosser property, there exists R such that $M_{n+1} \twoheadrightarrow_\beta R$ and $Q \twoheadrightarrow_\beta R$. Since $M \twoheadrightarrow_\beta Q$ and $Q \twoheadrightarrow_\beta R$ we have $M \twoheadrightarrow_\beta R$. $\square$

Corollary 11.30. *If N is normal and $M =_\beta N$, then $M \twoheadrightarrow_\beta N$.*

Proof. Since $M =_\beta N$, by Corollary 11.29, there exists a λ-term P such that $M, N \twoheadrightarrow_\beta P$. Since N is normal, $N =_\alpha P$ which implies that $M \twoheadrightarrow_\beta N$. $\square$

Corollary 11.31. *If M and N are normal and $M =_\beta N$, then $M =_\alpha N$.*

Proof. By Corollary 11.30, we have $M \twoheadrightarrow_\beta N$. But M is also normal, hence $M =_\alpha N$. $\square$

This shows the consistency of the system in the sense that it is impossible to show $\bar{0} =_\beta \bar{1}$ in lambda calculus. The reason simply is that both of these two λ-terms are normal and by Corollary 11.31, if $\bar{0} =_\beta \bar{1}$, then $\bar{0} =_\alpha \bar{1}$ which is impossible.

Exercise 11.32. Show that there is no λ-term F such that $F(MN) =_\beta M$, for all λ-terms M and N.

Exercise 11.33. Assume that for some λ-terms M and N and some x not occurring in M and N, we have $Mx =_\beta Nx$. Does it necessarily imply that $M =_\beta N$?

Theorem 11.34. *All representable functions are computable.*

Proof. If $\mathbf{F}$ represents the function f, then f is computable by the algorithm that searches for a series of reductions starting with $F\bar{n}_1 \cdots \bar{n}_k$ and ending in a normal λ-term in the form $\bar{m}$. Then, the algorithm returns the number m. First, note that this m is unique, if it exists, thanks to the Church-Rosser property. Secondly, note that since it is unique, it must be $f(n_1, \cdots, n_k)$ by the assumption and thirdly, $f(n_1, \cdots, n_k)$ is defined iff such a normal λ-terms exists. Therefore, the algorithm computes the function f. □

Theorem 11.35. *(i) The subset of all λ-terms that have a normal form is undecidable.*

(ii) The relation $=_\beta$ is undecidable.

Proof. First, note that if $H(m, n)$ is the halting predicate, then χ'_H is computable and hence representable in lambda calculus by a λ-term $\mathbf{F}$. For (i), if the set is decidable, then there is an algorithm to decide whether the λ-term $\mathbf{F}\bar{m}\bar{n}$ has a normal form or not. By the definition of representability, this is equivalent to the existence of (m, n) in the domain of χ'_H or equivalently to $H(m, n)$. Since halting is undecidable, (i) follows. For (ii), if $=_\beta$ is decidable, then $\mathbf{F}\bar{m}\bar{n} =_\beta \bar{1}$ is decidable while it is equivalent to $H(m, n)$. □

Acknowledgment. The present note is the course material that we designed for the course "Logic and Computation" for master AI students at Utrecht university, taught in 2019 and 2021. We are indebted to Raheleh Jalali for her careful reading of the text that led to a significant modification and also her invaluable suggestions on the presentation style and the useful exercises she provided.

References

[1] Barendregt, Henk P. "Introduction to lambda calculus." (1984).

[2] Davis, Martin, Ron Sigal, and Elaine J. Weyuker. Computability, complexity, and languages: fundamentals of theoretical computer science. Elsevier, 1994.

[3] Enderton, Herbert B. Computability theory: An introduction to recursion theory. Academic Press, 2010.

[4] Odifreddi, Piergiorgio. Classical recursion theory: The theory of functions and sets of natural numbers. Elsevier, 1992.

[5] Selinger, Peter. "Lecture notes on the lambda calculus." arXiv preprint arXiv:0804.3434 (2008).

[6] Sipser, Michael. Introduction to the Theory of Computation. Cengage learning, 2012.