



(d) Define a function  $\text{issortingfun} : \text{String} \rightarrow \{0, 1\}$ ,

$$\text{issortingfun}(p) := \begin{cases} 1 & \text{if } p \text{ is a syntactically correct} \\ & \text{Pascal program, which has as input a list} \\ & \text{and returns a sorted list,} \\ 0 & \text{otherwise.} \end{cases}$$

So  $\text{issortingfun}$  checks whether its input is a sorting function. Of course it has to be specified precisely what it means for a program to take a list as input and returns it (e.g. input from console and output on console).

If we could decide this problem, we could decide the previous one: Take a Pascal program for which we have to check whether it is a sorting function. Create a new pascal program, which takes as input a list, then runs as the original one, until that one terminates. If it terminates, the new program takes the original list, sorts it and returns it. Otherwise the new program never terminates. The new program is a sorting function if and only if the original program terminated. Since the problem, whether a program terminates, is undecidable, the problem of verifying, whether we have a sorting function, is undecidable as well.

### 1.3 Problems in Computability

In order to properly understand and answer the questions raised in the previous examples we have to:

- First give a precise definition of what “*computable*” means.
  - That will be a *mathematical definition*.
    - \* For showing that a function is computable, it suffices to show how it can be computed by a computer. For this an intuitive understanding of “computable” suffices.
    - \* In order to show that a function is non-computable, however, we need to show that it cannot be computed in principle. For this we need a very precise notion of computable.
- Then provide evidence that our definition of “*computable*” is the correct one.
  - That will be a *philosophical argument*.
- Develop methods for proving that certain functions are computable and certain functions are non-computable.

Questions related to the above are the following:

- Given a function  $f : A \rightarrow B$  can be computed, can it be done *effectively*? (Complexity theory.)
- Can the task of deciding a given problem  $P1$  be reduced to deciding another problem  $P2$ ? (Reducibility theory).

Other interesting questions, which are beyond the scope of this lecture are:

- Can the notion of computability be extended to computations on infinite objects (like e.g. streams of data, real numbers, higher type operations)? (Higher and abstract computability theory).
- What is the relationship between *computing* (producing actions, data etc.) and *proving*.

Computability theory involves **three areas**:

- **Mathematics.**
  - To give a precise definition of computability and analyse this concept.
- **Philosophy.**
  - to verify that notions found, like “computable”, are the correct ones.
- **Computer science.**
  - To investigate the relationship of these theoretical concepts and computing in the real world.

**Remark:** In computability theory, one usually abstracts from limitations on time and space. A problem will be computable, if it can be solved on an *idealised computer*, even if it would take longer than the life time of the universe to actually carry out the computation.

#### 1.4 Remarks on the History of Computability Theory

- **Leibnitz (1670)** Leibnitz built a first mechanical calculator. He was thinking about building a machine that could manipulate symbols in order to determine the truth values of mathematical statements. He noticed that a first step would be to introduce a precise formal language, and he was working on defining such a language.



**Gottfried Wilhelm  
von Leibnitz (1646 – 1716)**

- **Hilbert (1900)** poses in his famous list “Mathematical Problems” as 10th problem to decide Diophantine equations.



**David Hilbert (1862 – 1943)**

- **Hilbert (1928)** poses the “Entscheidungsproblem” (decision problem).
  - He has (already 1900) developed a theory for formalising mathematical proofs, and believes that it is complete and sound, i.e. that it shows exactly all true mathematical formulae.
  - Hilbert asks, whether there is an algorithm, which decides whether a mathematical formula is a consequence of his theory. Assuming that his theory is complete and sound, such an algorithm would decide the truth of all mathematical formulae, expressible in the language of his theory.
  - The question, whether there is an algorithm for deciding the truth of mathematical formulae is later called the “Entscheidungsproblem”.
- **Gödel, Kleene, Post, Turing (1930s)** introduce different models of computation and prove that they all define the same class of computable functions.



**Kurt Gödel (1906 – 1978)**



**Stephen Cole Kleene  
(1909 – 1994)**



**Emil Post (1897 – 1954)**



**Alan Mathison Turing**  
(1912 – 1954)

- **Gödel (1931)** proves in his incompleteness theorem that that any reasonable recursive is incomplete, i.e there is a formula which is neither provable nor its negation. By the Church-Turing thesis to be established later (see below), it will follow that the recursive functions are the computable ones. Therefore no such theory proves all true formulae (true in the sense of being true in a certain model, which could be standard structure assumed in mathematics). Therefore, the “Entscheidungsproblem” is unsolvable – an algorithm for deciding the truth of mathematical formulae would give rise to a complete and sound theory fulfilling Gödel’s conditions.
- **Church, Turing (1936)** postulate that the models of computation established above define exactly the set of all computable functions (Church-Turing thesis).
- Both establish undecidable problems and conclude that the Entscheidungsproblem is unsolvable, even for a class of very simple formulae.
  - Church shows the undecidability of equality in the  $\lambda$ -calculus.
  - Turing shows the unsolvability of the halting problem.  
That problem turns out to be the most important undecidable problem.



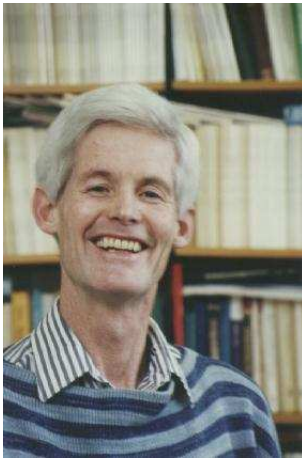
**Alonzo Church (1903 - 1995)**

- **Post (1944)** studies degrees of unsolvability. This is the birth of degree theory.
- **Matiyasevic (1970)** solves Hilbert’s 10th problem negatively: The solvability of Diophantine equations is undecidable.



**Yuri Vladimirovich  
Matiyasevich (\* 1947)**

- **Cook (1971)** introduces the complexity classes **P** and **NP** and formulates the problem, whether  $\mathbf{P} \neq \mathbf{NP}$ .



**Stephen Cook  
(Toronto)**

- **Today:**
  - The problem  $\mathbf{P} \neq \mathbf{NP}$  is still open. Complexity theory has become a big research area.
  - Intensive study of computability on infinite objects (e.g. real numbers, higher type functionals) is carried out (e.g. Dr. Berger in Swansea).
  - Computability on inductive and co-inductive data types is studied.
  - Research on program synthesis from formal proofs (e.g. Dr. Berger in Swansea).
  - Concurrent and game-theoretic models of computation are developed (e.g. Prof. Moller in Swansea).
  - Automata theory further developed.
  - Alternative models of computation are studied (quantum computing, genetic algorithms).
  - ...
- **Remarks on the Name “Computability Theory”:**
  - The original name was *recursion theory*, since the mathematical concept claimed to cover exactly the computable functions is called “recursive function”.
  - This name was changed to computability theory during the last 10 years.
  - Many books still have the title “recursion theory”.

## 1.5 Administrative Issues

**Lecturer:**

Dr. A. Setzer  
Dept. of Computer Science  
University of Wales Swansea  
Singleton Park  
SA2 8PP  
UK

**Room:** Room 211, Faraday Building

**Tel.:** (01792) 513368

**Fax.** (01792) 295651

**Email** a.g.setzer@swansea.ac.uk

**Home page:** <http://www.cs.swan.ac.uk/~csetzer/>

**Assessment:**

- 80% Exam.
- 20% Coursework.

The **course home page:** is located at

<http://www.cs.swan.ac.uk/~csetzer/lectures/computability/03/index.html>

There is an open version of the slides and a, for copyright-reasons, password-protected version.

The password is \_\_\_\_\_.

## 1.6 Plan for this Module.

(Might be changed, since the lecturer is teaching this module for the first time).

### 1. Introduction.

- This section.

### 2. Encoding of data types into $\mathbb{N}$ .

- We show how to encode elements of some data types as elements of  $\mathbb{N}$ , the set of natural numbers.
- This shows that by studying computability of natural numbers we treat computability on many other data types as well.
- We discuss as well the notions of countable vs. uncountable sets.

### 3. The Unlimited Register Machine (URM) and the halting problem.

- We study one model of computation, the URM. It is one of the easiest to work with.
- We show that the halting problem is undecidable.

### 4. Turing machines.

- Turing machines as a more intuitive model of computation.

### 5. Algebraic view of computability.

- We study the notion of primitive recursive functions.
- We introduce the concept of a partial recursive function, a third model of computation.

**6.** Lambda-definable functions.

- We study a fourth model of computation.
- We show that lambda-definable functions and partial recursive functions are the same class.

**7.** Equivalence theorems and the Church-Turing thesis.

- We show that the four notions of computation above coincide.
- We discuss the Church-Turing thesis, namely that in an intuitive sense computable functions are exactly those which can be computed by one of the equivalent models of computation.

**8.** Enumeration of the computable functions.

- We show how to enumerate in a computable way all computable functions.

**9.** Recursively enumerable predicates and the arithmetic hierarchy.

- We study a notion, which covers relations which are undecidable, but which can still be computed: we can in a computable way enumerate the elements fulfilling it.
- We study the arithmetic hierarchy.

**10.** Reducibility.

- We study the concept of reducibility.
- We show Rice's theorem, which allows to show the undecidability of many sets.

**11.** Computational complexity.

- We study basic complexity theory.

**1.7 Aims of this Module**

- To become familiar with fundamental models of computation and the relationship between them.
- To develop an appreciation for the limits of computation and to learn techniques for recognising unsolvable or unfeasible computational problems.
- To understand the historic and philosophical background of computability theory.
- To be aware of the impact of the fundamental results of computability theory to areas of computer science such as software engineering and artificial intelligence.
- To understand the close connection between computability theory and logic.
- To be aware of recent concepts and advances in computability theory.
- To learn fundamental proving techniques like induction and diagonalisation.

## 1.8 Literature:

- N J Cutland: *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1984.
  - Main course book
- H. R. Lewis and C. H. Papadimitriou: *Elements of the Theory of Computation*. 2nd Edition, Prentice Hall, 1998.
- M. Sipser: *Introduction to the Theory of Computation*. PWS, 1997.
- J. Martin: *Introduction to Languages and the Theory of Computation*. McGraw-Hill, 2003.
- J. E. Hopcroft, R. Motwani, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000.
  - Book on automata theory and context free grammars.
- J. R. Hindley: *Basic Simple Type Theory*. Cambridge University Press, Cambridge Tracts in Theoretical Computer Science 42, 1997.
  - Best book on the  $\lambda$ -calculus.
- D. J. Velleman: *How To Prove It*. Cambridge University Press, 1994.
  - Basic mathematics. Recommended to fill any gaps in your mathematical background.
- E Griffor (Ed.): *Handbook of Computability Theory*. North-Holland, 1999.
  - Very expensive. State of the art of computability theory, postgraduate level.

## 2 Encoding of Data Types into $\mathbb{N}$

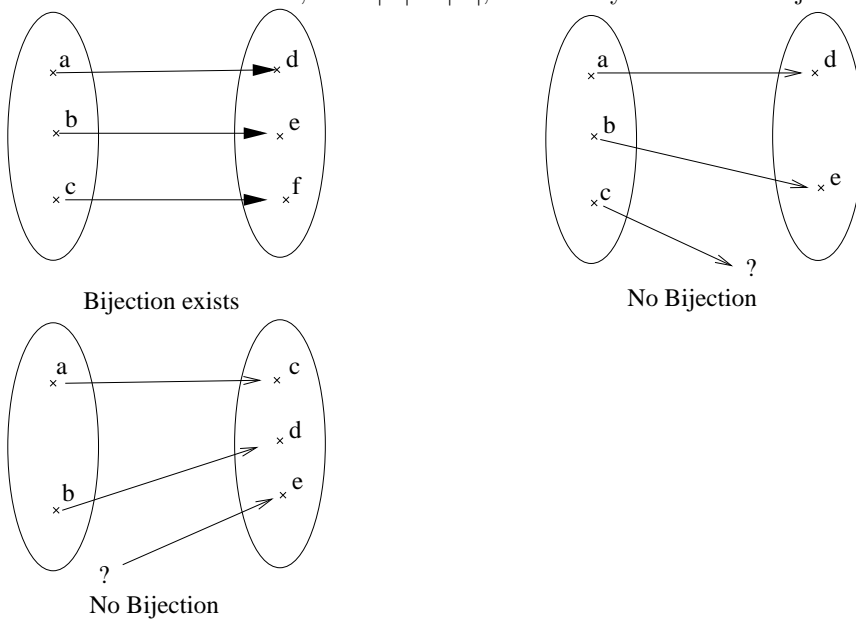
In this section, we show how to encode elements of some data types as elements of  $\mathbb{N}$ , the set of natural numbers. This shows that by studying computability of natural numbers we treat computability on many other data types as well. We discuss as well the notions of countable vs. uncountable sets.

### 2.1 Countable and Uncountable Sets

**Notation 2.1** If  $A$  is a finite set, let  $|A|$  be the number of elements in  $A$ .

**Remark 2.2** One sometimes writes  $\#A$  for  $|A|$ .

If  $A$  and  $B$  are finite sets, then  $|A| = |B|$ , if and only if there is a bijection between  $A$  and  $B$ :



For arbitrary (possibly infinite) sets, the above generalises as follows:

**Definition 2.3** Two sets  $A$  and  $B$  have the **same cardinality**, written as  $A \simeq B$ , if there exists a bijection between  $A$  and  $B$ , i.e. if there exists functions  $f : A \rightarrow B$  and  $g : B \rightarrow A$  which are inverse with each other:  $\forall x \in A. g(f(x)) = x$  and  $\forall x \in B. f(g(x)) = x$ .

It follows immediately:

**Remark 2.4** If  $A$  and  $B$  are finite sets, then  $A \simeq B$  if and only if  $A$  and  $B$  have the same number of elements.

**Lemma 2.5**  $\simeq$  is an equivalence relation, i.e. for all sets  $A, B, C$  we have:

- (a) **Reflexivity.**  $A \simeq A$ .
- (b) **Symmetry.** If  $A \simeq B$ , then  $B \simeq A$ .
- (c) **Transitivity.** If  $A \simeq B$  and  $B \simeq C$ , then  $A \simeq C$ .

**Proof:**

- (a): The function  $\text{id} : A \rightarrow A, \text{id}(a) = a$  is a bijection.
- (b): If  $f : A \rightarrow B$  is a bijection, so is its inverse  $f^{-1}$ .
- (c): If  $f : A \rightarrow B$  and  $g : B \rightarrow C$  are bijections, so is the composition  $g \circ f : A \rightarrow C$ .

**Theorem 2.6** A set  $A$  and its power set  $\mathcal{P}(A) := \{B \mid B \subseteq A\}$  never have the same cardinality:

$$A \not\approx \mathcal{P}(A)$$

**Proof:**

This is a typical diagonalisation argument.

We first consider the case  $A = \mathbb{N}$  and show that there is no bijection between  $\mathbb{N}$  and  $\mathcal{P}(\mathbb{N})$ .

Assume we have such a bijection  $f : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$ . We define a set  $C \subseteq \mathbb{N}$  s.t.  $C \neq f(n)$  for every  $n \in \mathbb{N}$ .  $C = f(n)$  will be violated at element  $n$ :

- If  $n \in f(n)$ , we add  $n$  not to  $C$ , therefore  $n \in f(n) \wedge n \notin C$ .
- If  $n \notin f(n)$ , we add  $n$  to  $C$ , therefore  $n \notin f(n) \wedge n \in C$ .

**Example:** We take an arbitrary function  $f$ , and show how  $C$  is defined in this case:

$$\begin{aligned} f(0) &= \{ 0, 1, 2, 3, 4, \dots \} \\ f(1) &= \{ 0, 2, 4, \dots \} \\ f(2) &= \{ 1, 3, \dots \} \\ f(3) &= \{ 0, 1, 3, 4, \dots \} \\ &\dots \\ C &= \{ 1, 2, \dots \} \end{aligned}$$

We were considering above the  $n$ th element of  $f(n)$ , so we were going through the diagonal in the above matrix. Therefore this argument is called a *diagonalisation argument*.

So we obtain the following definition

$$C := \{n \in \mathbb{N} \mid n \notin f(n)\} .$$

Now  $C = f(n)$  is violated at element  $n$ : If  $n \in C$ , then  $n \notin f(n)$ . If  $n \notin C$ , then  $n \in f(n)$ .  $C$  is not in the image of  $f$ , a contradiction.

In short, the above argument reads as follows:

Assume  $f : \mathbb{N} \rightarrow \mathcal{P}(A)$  is a bijection. Define  $C := \{n \in \mathbb{N} \mid n \notin f(n)\}$ .  $C$  is in the image of  $f$ . Assume  $C = f(n)$ . Then

$$\begin{aligned} n \in C &\stackrel{\text{Definition of } C}{\Leftrightarrow} n \notin f(n) \\ &\stackrel{C=f(n)}{\Leftrightarrow} n \notin C \\ &\text{a contradiction} \end{aligned}$$

**General Situation:** The proof for the general situation is almost identical:

Assume  $f : A \rightarrow \mathcal{P}(A)$  is a bijection.

We define a set  $C$ , s.t.  $C = f(a)$  is violated for  $a$ :

$$C := \{a \in A \mid a \notin f(a)\}$$

$C$  is in the image of  $f$ . Assume  $C = f(a)$ . Then we have

$$\begin{aligned} a \in C &\stackrel{\text{Definition of } C}{\Leftrightarrow} a \notin f(a) \\ &\stackrel{C=f(a)}{\Leftrightarrow} a \notin C \\ &\text{a contradiction} \end{aligned}$$

**Definition 2.7** • A set  $A$  is countable, if it is finite or  $A \simeq \mathbb{N}$ .

- A set, which is not countable, is called uncountable.

**Examples:**

- $\mathbb{N}$  is countable.
  - Since  $\mathbb{N} \simeq \mathbb{N}$ .
- $\mathbb{Z} := \{\dots, -2, -1, 0, 1, 2, \dots\}$  is countable.
  - We can enumerate the elements of  $\mathbb{Z}$  in the following way:  
 $0, -1, +1, -2, +2, -3, +3, -4, +4, \dots$   
 So we have the following map:  
 $0 \mapsto 0, 1 \mapsto -1, 2 \mapsto 1, 3 \mapsto -2, 4 \mapsto 2$ , etc.  
 This map can be described as follows:  
 $g : \mathbb{N} \rightarrow \mathbb{Z}$ ,

$$g(n) := \begin{cases} \frac{n}{2} & \text{if } n \text{ is even,} \\ -\frac{n+1}{2} & \text{if } n \text{ is odd.} \end{cases}$$

**Exercise:** Show that  $g$  is bijective.

- $\mathcal{P}(\mathbb{N})$  is uncountable.
  - $\mathcal{P}(\mathbb{N})$  is not finite.
  - $\mathbb{N} \not\approx \mathcal{P}(\mathbb{N})$ .
- $\mathcal{P}(\{1, \dots, 10\})$  is finite, therefore countable.

**Lemma 2.8** *A set  $A$  is countable, if and only if there is an injective map  $g : A \rightarrow \mathbb{N}$ .*

**Remark 2.9** *Intuitively, Lemma 2.8 expresses:  $A$  is countable, if we can assign to every element  $a \in A$  a unique code  $f(a) \in \mathbb{N}$ . It is not required that each element of  $\mathbb{N}$  occurs as a code.*

**Proof of Lemma 2.8:**

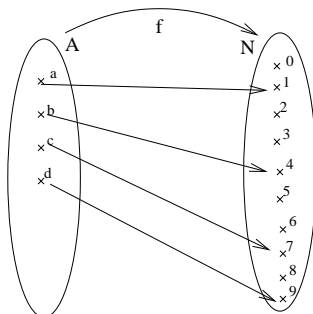
“ $\Rightarrow$ ”: Assume  $A$  is countable. Show there is an injective  $f : A \rightarrow \mathbb{N}$ .

- Case  $A$  is finite. Assume  $A = \{a_0, \dots, a_n\}$  with  $a_i \neq a_j$  for  $i \neq j$ . Define  $f : A \rightarrow \mathbb{N}$ ,  $a_i \mapsto i$ .  $f$  is injective.
- Case  $A$  is infinite.  $A$  is countable, so there exists a bijection from  $A$  into  $\mathbb{N}$ , which is therefore injective.

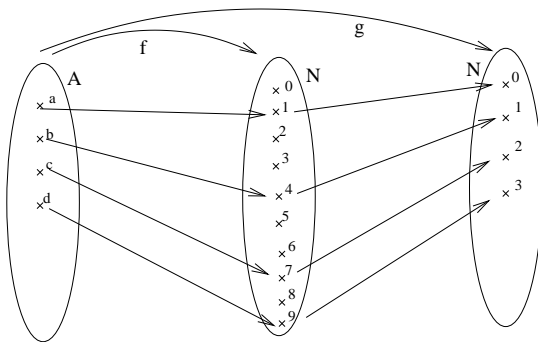
“ $\Leftarrow$ ”: Assume  $f : A \rightarrow \mathbb{N}$  is injective. Show  $A$  is countable.

If  $A$  is finite, we are done.

Assume  $A$  is infinite. Then  $f$  is for instance something like the following:



In order to obtain a bijection  $g : A \rightarrow \mathbb{N}$ , we have to jump over the gaps in the image of  $f$ :



So we have

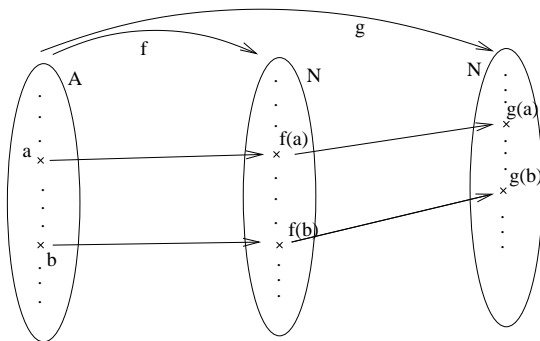
- $f(a) = 1$ , which is the element number 0 in the image of  $f$ .  $g$  should instead map  $a$  to 0.
- $f(b) = 4$ , which is the element number 1 in the image of  $f$ .  $g$  should instead map  $b$  to 1.
- etc.

1 is element number 0 in the image of  $f$ , because the number of elements  $f(a')$  below  $f(a)$  is 0. 4 is element number 1 in the image of  $f$ , because the number of elements  $f(a')$  below  $f(b)$  is 1. So in general we define  $g : A \rightarrow \mathbb{N}$ .

$$g(a) := |\{a' \in A \mid f(a') < f(a)\}|$$

$g$  is well defined, since  $f$  is injective, so the number of  $a' \in A$  s.t.  $f(a') < f(a)$  is finite. We show that  $g$  is a bijection:

- $g$  is injective:  
 Assume  $a, b \in A, a \neq b$ . Show  $g(a) \neq g(b)$ .  
 By the injectivity of  $f$  we have  $f(a) \neq f(b)$ . Let for instance  $f(a) < f(b)$ .



Then

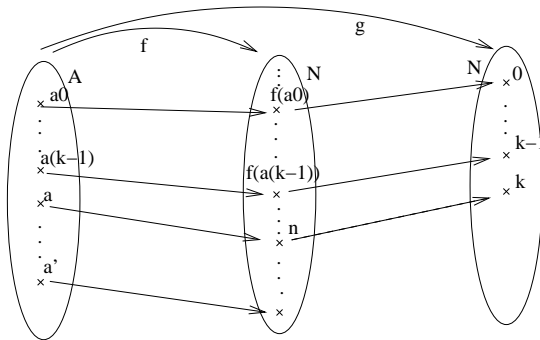
$$\{a' \in A \mid f(a') < f(a)\} \subsetneq \{a' \in A \mid f(a') < f(b)\} ,$$

therefore

$$g(a) = |\{a' \in A \mid f(a') < f(a)\}| < |\{a' \in A \mid f(a') < f(b)\}| = g(b) ,$$

$$g(a) \neq g(b).$$

- $g$  is surjective:  
 We define by induction on  $k$  for  $k \in \mathbb{N}$  an element  $a_k \in A$  s.t.  $g(a_k) = k$ . Then the assertion follows:  
 Assume we have defined already  $a_0, \dots, a_{k-1}$ .



There exist infinitely many  $a' \in A$ ,  $f$  is injective, so there must be at least one  $a' \in A$  s.t.  $f(a') > f(a_{k-1})$ . Let  $n$  be minimal s.t.  $n = f(a)$  for some  $a \in A$  and  $n > f(a_{k-1})$ . Let  $a$  be the unique element of  $A$  s.t.  $f(a) = n$ . Then we have

$$\{a'' \in A \mid f(a'') < f(a)\} = \{a'' \in A \mid f(a'') < f(a_{k-1})\} \cup \{a_{k-1}\} .$$

Therefore

$$\begin{aligned} g(a) &= |\{a'' \in A \mid f(a'') < f(a)\}| \\ &= |\{a'' \in A \mid f(a'') < f(a_{k-1})\}| + 1 \\ &= g(a_{k-1}) + 1 \\ &= k - 1 + 1 \\ &= k . \end{aligned}$$

Let  $a_k := a$ . Then  $g(a_k) = k$ .

**Corollary 2.10** (a) If  $B$  is countable and  $g : A \rightarrow B$  injective, then  $A$  is countable.

(b) If  $A$  is uncountable and  $g : A \rightarrow B$  injective, then  $B$  is uncountable.

(c) If  $B$  is countable and  $A \subseteq B$ , then  $A$  is countable.

(d) If  $A$  is uncountable and  $A \subseteq B$ , then  $B$  is uncountable.

**Proof:**

(a): If  $f : B \rightarrow \mathbb{N}$  is an injection, so is  $f \circ g : A \rightarrow \mathbb{N}$ .

(b): By (a). Why? (Exercise).

(c): By (a). (What is  $g$ ?; exercise).

(d): By (c). Why? (Exercise).

**Lemma 2.11** Let  $A$  be a non-empty set.  $A$  is countable, if and only if there is a surjection  $h : \mathbb{N} \rightarrow A$ .

**Proof:**

“ $\Rightarrow$ ”: Assume  $A$  is non-empty and countable. Show there exists a surjection  $f : \mathbb{N} \rightarrow A$ .

- Case  $A$  is finite. Assume  $A = \{a_0, \dots, a_n\}$ . Define  $f : \mathbb{N} \rightarrow A$ ,

$$f(k) := \begin{cases} a_k & \text{if } k \leq n \\ a_0 & \text{otherwise} . \end{cases}$$

$f$  is clearly surjective.

- Case  $A$  is infinite.  $A$  is countable, so there exists a bijection from  $\mathbb{N}$  to  $A$ , which is therefore surjective.

“ $\Leftarrow$ ”: Assume  $h : \mathbb{N} \rightarrow A$  is surjective. Show  $A$  is countable.

Define  $g : A \rightarrow \mathbb{N}$ ,  $g(a) := \min\{n \mid h(n) = a\}$ .  $g(a)$  is well-defined, since  $h$  is surjective: there exists some  $n$  s.t.  $h(n) = a$ , therefore the minimal such  $n$  is well-defined.

It follows that for all  $a \in A$ ,  $h(g(a)) = a$ . Therefore  $g$  is injective: If  $a, a' \in A$ ,  $g(a) = g(a')$ , then  $a = h(g(a)) = h(g(a')) = a'$ . By Lemma 2.8,  $A$  is countable.

**Lemma 2.12** *The following sets are uncountable:*

- (a)  $\mathcal{P}(\mathbb{N})$ .
- (b)  $F := \{f \mid f : \mathbb{N} \rightarrow \{0, 1\}\}$ .
- (c)  $G := \{f \mid f : \mathbb{N} \rightarrow \mathbb{N}\}$ .
- (d) *The set of real numbers  $\mathbb{R}$ .*

**Proof:**

(a):  $\mathcal{P}(\mathbb{N})$  is not finite and  $\mathcal{P}(\mathbb{N}) \not\cong \mathbb{N}$  by Theorem 2.6.

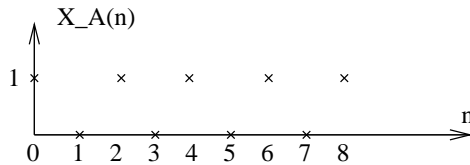
(b): We introduce a bijection between  $F$  and  $\mathcal{P}(\mathbb{N})$ . Then by  $\mathcal{P}(\mathbb{N})$  uncountable it follows that  $F$  is uncountable.

Define for  $A \in \mathcal{P}(\mathbb{N})$   $\chi_A : \mathbb{N} \rightarrow \{0, 1\}$ ,

$$\chi_A(n) := \begin{cases} 1 & n \in A, \\ 0 & \text{otherwise.} \end{cases}$$

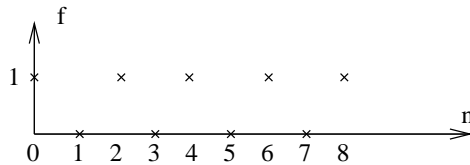
$\chi_A$  is called the *characteristic function of  $A$* . For instance, if  $A = \{2, 4, 6, 8, \dots\}$ , then  $\chi_A$  is as follows:

$A = \{0 \quad 2, \quad 4, \quad 6, \quad 8, \quad \dots$



$\chi$  is a function from  $\mathcal{P}(\mathbb{N})$  to  $\mathbb{N} \rightarrow \{0, 1\}$ , where we write the application of  $\chi$  to an element  $A$  as  $\chi_A$  instead of  $\chi(A)$ .  $\chi$  has an inverse, namely the function  $\chi^{-1}$  from  $\mathbb{N} \rightarrow \{0, 1\}$  into  $\mathcal{P}(\mathbb{N})$ , where for  $f : \mathbb{N} \rightarrow \{0, 1\}$ ,  $\chi^{-1}(f) := \{n \in \mathbb{N} \mid f(n) = 1\}$ .

For instance, if  $f : \mathbb{N} \rightarrow \mathbb{N}$ ,  $f(n) = \begin{cases} 1 & \text{if } n \text{ is even,} \\ 0 & \text{otherwise,} \end{cases}$  (i.e.  $f = \chi_A$  for the set  $A$  of even numbers as above), then  $\chi^{-1}(f)$  is as follows



$X^{-1}(f) = \{0 \quad 2, \quad 4, \quad 6, \quad 8, \quad \dots$

We show that  $\chi$  and  $\chi^{-1}$  are in fact inverse:

$\chi^{-1} \circ \chi$  is the identity: If  $A \subseteq \mathbb{N}$ , then

$$\begin{aligned} \chi^{-1}(\chi_A) &= \{n \in \mathbb{N} \mid \chi_A(n) = 1\} \\ &= \{n \in \mathbb{N} \mid n \in A\} \\ &= A \end{aligned}$$

$\chi \circ \chi^{-1}$  is the identity: If  $f : \mathbb{N} \rightarrow \{0, 1\}$ , then

$$\begin{aligned} \chi_{\chi^{-1}(f)}(n) = 1 &\Leftrightarrow n \in \chi^{-1}(f) \\ &\Leftrightarrow f(n) = 1 \end{aligned}$$

and

$$\begin{aligned} \chi_{\chi^{-1}(f)}(n) = 0 &\Leftrightarrow n \notin \chi^{-1}(f) \\ &\Leftrightarrow f(n) \neq 1 \\ &\Leftrightarrow f(n) = 0 . \end{aligned}$$

Therefore  $\chi_{\chi^{-1}(f)} = f$ .

It follows that  $\chi$  is bijective and  $F$  is uncountable.

(c): By (c), since  $F \subseteq G$ .

(d): A first idea is to define a function  $f_0 : F \rightarrow \mathbb{R}$ ,  $f_0(g) = (0.g(0)g(1)g(2)\cdots)_2$ , where the right hand side is a number in binary format. If  $f_0$  were injective, then by  $F$  uncountable we could conclude  $\mathbb{R}$  is uncountable. The problem is that  $(0.a_0a_1\cdots a_k01111\cdots)_2$  and  $(0.a_0a_1\cdots a_k10000\cdots)_2$  denote the same real number, so  $f_0$  is not injective.

In order to avoid this problem, we modify  $f_0$ , so that we don't obtain any binary numbers of the form  $(0.a_0a_1\cdots a_k1111\cdots)_2$ . We define instead  $f : F \rightarrow \mathbb{R}$ ,  $f(g) = (0.g(0) 0 g(1) 0 g(2) 0\cdots)_2$ , i.e.  $f(g) = (0.a_0a_1a_2\cdots)_2$ , where

$$a_k := \begin{cases} 0 & \text{if } k \text{ is odd,} \\ g(\frac{k}{2}) & \text{otherwise.} \end{cases}$$

If two sequences  $(b_0b_1\cdots)$  and  $(c_0c_1\cdots)$  don't end in  $1111\cdots$ , i.e. are not of the form  $(d_0d_1\cdots d_l1111\cdots)$ , then

$$(0.b_0b_1\cdots)_2 = (0.c_0c_1\cdots)_2 \Leftrightarrow (b_0b_1b_2\cdots) = (c_0c_1c_2\cdots)$$

Therefore

$$\begin{aligned} f(g) = f(g') &\Leftrightarrow (0.g(0) 0 g(1) 0 g(2) 0\cdots)_2 = (0.g'(0) 0 g'(1) 0 g'(2) 0\cdots)_2 \\ &\Leftrightarrow (g(0) 0 g(1) 0 g(2) 0\cdots) = (g'(0) 0 g'(1) 0 g'(2) 0\cdots) \\ &\Leftrightarrow (g(0) g(1) g(2) \cdots) = (g'(0) g'(1) g'(2) \cdots) \\ &\Leftrightarrow g = g' , \end{aligned}$$

$f$  is injective.

### Remark on the Continuum Hypothesis:

One can show that the four sets above all have the same cardinality.

So there are two infinite  $\mathbb{N}$  and  $\mathbb{R}$ , which have different cardinality. The question arises whether there is a set  $B$  with cardinality in between, i.e. s.t. there are injections from  $\mathbb{N}$  into  $B$  and from  $B$  into  $\mathcal{P}(\mathbb{N})$ , but s.t.  $B \not\cong \mathbb{N}$  and  $B \not\cong \mathcal{P}(\mathbb{N})$ . This question was Hilbert's first problem.

That there is no such set is called the Continuum Hypothesis. ( $\mathbb{R}$  is called the continuum). Paul Cohen has shown that this question is independent from ordinary set theory, i.e. we can neither show that such a  $B$  exists, nor that such a  $B$  doesn't exist.



**Paul Cohen**  
(\* 1934)

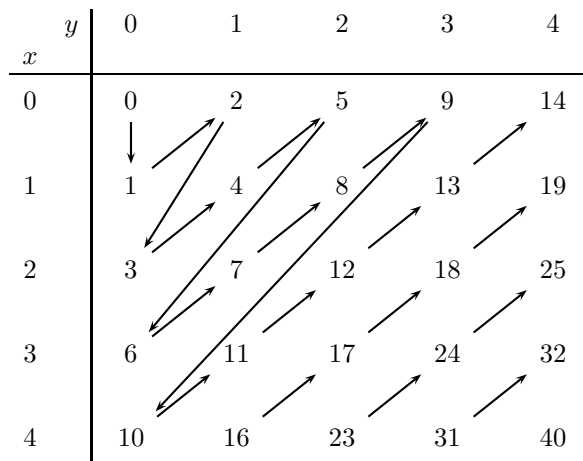
## 2.2 Encoding of Sequences into $\mathbb{N}$

**Definition 2.13** *Let  $A, B$  be sets.*

- $A^k := \{(a_1, \dots, a_k) \mid a_1, \dots, a_k \in A\}$ .  $A^k$  is the set of  $k$ -tuples of elements of  $A$ , and is called the  $k$ -fold Cartesian product.  
Especially  $A^0 = \{()\}$ .
- $A^* := \bigcup_{n \in \mathbb{N}} A^n$ .  
 $A^*$  is the set of tuples of  $A$  of arbitrary length, and called the Kleene-Star.
- $A \rightarrow B := \{f \mid f : A \rightarrow B\}$ .  
 $A \rightarrow B$  is the set of functions from  $A$  to  $B$  and is called the function space.

**Remark:**  $A^*$  can be considered as the set of strings having letters in the alphabet  $A$ .

We want to define a bijective function  $\pi : \mathbb{N}^2 \rightarrow \mathbb{N}$ .  $\pi$  will be called the pairing function. To define such a bijection means, that we have to enumerate all elements of  $\mathbb{N}^2$ . Pairs of natural numbers can be enumerated in the following way:



- We start in the top left corner and define  $\pi(0, 0) = 0$ .
- Then we go to the next diagonal.  $\pi(1, 0) = 1$  and  $\pi(0, 1) = 2$ .
- Then we go to the next diagonal.  $\pi(2, 0) = 3$ ,  $\pi(1, 1) = 4$ ,  $\pi(0, 2) = 5$ .
- etc.

Note, that the following naïve attempt to enumerate the pairs, fails:

| $y$ | 0          | 1                      | 2                      | 3                      | 4                      |                        |                     |
|-----|------------|------------------------|------------------------|------------------------|------------------------|------------------------|---------------------|
| $x$ |            |                        |                        |                        |                        |                        |                     |
| 0   | $\pi(0,0)$ | $\rightarrow \pi(0,1)$ | $\rightarrow \pi(0,2)$ | $\rightarrow \pi(0,3)$ | $\rightarrow \pi(0,4)$ | $\rightarrow \pi(0,5)$ | $\rightarrow \dots$ |
| 1   |            | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow \dots$ |
| 2   |            | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow \dots$ |
| 3   |            | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow \dots$ |
| 4   |            | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow$          | $\rightarrow \dots$ |

In this enumeration, the first pair is  $\pi(0,0)$ , which gets number 0. The next one is  $\pi(0,1)$ , and it gets number 1.  $\pi(0,2)$  gets number 2 etc. But this way we never reach the pair  $(1,0)$ , since it takes infinitely long to enumerate the elements of the first row – this enumeration does not work.

We will investigate the first solution, which worked. If we look at the diagonals we see that following:

- For the pairs in the diagonal we have the property that  $x + y$  is constant.
  - The first diagonal, consisting of  $(0,0)$  only, is given by  $x + y = 0$ .
  - The second diagonal, consisting of  $(1,0), (0,1)$ , is given by  $x + y = 1$ .
  - The third diagonal, consisting of  $(2,0), (1,1), (0,2)$ , is given by  $x + y = 2$ .
  - Etc.s
- The diagonal given by  $x + y = n$ , consists of  $n + 1$  pairs:
  - The first diagonal, given by  $x + y = 0$ , consists of  $(0,0)$  only, i.e. of 1 pair.
  - The second diagonal, given by  $x + y = 1$ , consists of  $(1,0), (0,1)$ , i.e. of 2 pairs.
  - The third diagonal, given by  $x + y = 2$ , consisting of  $(2,0), (1,1), (0,2)$ , i.e. of 3 pairs.
  - etc.

| $y$ | 0 | 1 | 2 | 3 | 4 |
|-----|---|---|---|---|---|
| $x$ |   |   |   |   |   |
| 0   | 0 |   |   |   |   |
| 1   | 1 |   |   |   |   |
| 2   | 3 |   |   |   |   |
| 3   | 6 |   |   |   |   |

If we look at how many elements are before the pair  $(x_0, y_0)$  in the above order we have the following:

- We have to count all elements of the previous diagonals. These are those given by  $x + y = n$  for  $n < x_0 + y_0$ .
  - In the above example for the pair  $(2,1)$ , these are the diagonals given by  $x + y = 0, x + y = 1, x + y = 2$ .
  - The diagonal, given by  $x + y = n$ , has  $n + 1$  elements, so in total we have  $\sum_{i=0}^{x+y-1} (i + 1) = 1 + 2 + \dots + (x + y) = \sum_{i=1}^{x+y} i$ .

- Gauß showed already as a student at school that  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ . Therefore the above is  $\frac{(x+y)(x+y+1)}{2}$ .
- Further, we have to count all pairs in the current diagonal, which occur in this ordering before the current one. These are  $y$  pairs.
  - Before  $(2, 1)$  there is only one pair, namely  $(3, 0)$ .
  - Before  $(3, 0)$  there are 0 pairs.
  - Before  $(0, 2)$  there are 2 pairs, namely  $(2, 0), (1, 1)$ .
- Therefore we get that there are in total  $\frac{(x+y)(x+y+1)}{2} + y$  pairs before  $(x, y)$ , therefore the pair  $(x, y)$  is the pair number  $(\frac{(x+y)(x+y+1)}{2} + y)$  in this order. We have now the following definition:

**Definition 2.14**

$$\pi(x, y) := \frac{(x+y)(x+y+1)}{2} + y \quad (= (\sum_{i=1}^{x+y} i) + y)$$

**Exercise:** Prove that  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ .

**Lemma 2.15**

$\pi$  is bijective.

**Proof:**

We show  $\pi$  is **injective**: We prove first that, if  $x + y < x' + y'$ , then  $\pi(x, y) < \pi(x', y')$ :

$$\begin{aligned} \pi(x, y) &= (\sum_{i=1}^{x+y} i) + y < (\sum_{i=1}^{x+y} i) + x + y + 1 = \sum_{i=1}^{x+y+1} i \\ &\leq (\sum_{i=1}^{x'+y'} i) + y' = \pi(x', y') \end{aligned}$$

Assume now  $\pi(x, y) = \pi(x', y')$  and show  $x = x'$  and  $y = y'$ . We have by the above  $x + y = x' + y'$ . Therefore

$$y = \pi(x, y) - (\sum_{i=1}^{x+y} i) = \pi(x', y') - (\sum_{i=1}^{x'+y'} i) = y'$$

and  $x = (x + y) - y = (x' + y') - y' = x'$ .

We show  $\pi$  is **surjective**: Assume  $n \in \mathbb{N}$ . Show  $\pi(x, y) = n$  for some  $x, y \in \mathbb{N}$ . The sequence  $(\sum_{i=1}^{k'} i)_{k' \in \mathbb{N}}$  is strictly existing. Therefore there exists a  $k$  s.t.

$$a := \sum_{i=1}^k i \leq n < \sum_{i=1}^{k+1} i \quad (*)$$

So, in order to obtain  $\pi(x, y) = n$ , we need  $x + y = k$ . By  $y = \pi(x, y) - \sum_{i=1}^{x+y} i$ , we need to define  $y := n - a$  and, by  $k = x + y$ , therefore  $x := k - y$ . By (\*) it follows  $0 \leq y < k + 1$ , and therefore  $x, y \geq 0$ . Further,  $\pi(x, y) = (\sum_{i=1}^{x+y} i) + y = (\sum_{i=1}^k i) + (n - \sum_{i=1}^k i) = n$ .

Since  $\pi$  is bijective, we can define  $\pi_0, \pi_1$  as follows:

**Definition 2.16** Let  $\pi_0 : \mathbb{N} \rightarrow \mathbb{N}$  and  $\pi_1 : \mathbb{N} \rightarrow \mathbb{N}$  be s.t.  $\pi_0(\pi(x, y)) = x, \pi_1(\pi(x, y)) = y$ .

**Remark:**  $\pi, \pi_0, \pi_1$  are computable in an intuitive sense.

**“Proof:”**  $\pi$  is obviously computable.  $\pi_0(n), \pi_1(n)$  can be computed, by first searching for a  $k$  s.t.  $\frac{k(k+1)}{2} \leq n < \frac{(k+1)(k+2)}{2}$  and then defining as in the proof of the surjectivity of  $\pi$  (Lemma 2.15)  $\pi_1(n) := n - \frac{k(k+1)}{2}$  and  $\pi_0(n) = k - \pi_1(n)$ .

**Remark 2.17** For all  $z \in \mathbb{N}$ ,  $\pi(\pi_0(z), \pi_1(z)) = z$ .

**Proof:** Assume  $z \in \mathbb{N}$  and show  $z = \pi(\pi_0(z), \pi_1(z))$ .  $\pi$  is surjective, so there exists  $x, y$  s.t.  $\pi(x, y) = z$ . Then  $\pi(\pi_0(z), \pi_1(z)) = \pi(\pi_0(\pi(x, y)), \pi_1(\pi(x, y))) = \pi(x, y) = z$ .

We want to encode now elements of  $\mathbb{N}^k$  as natural numbers. In order to encode an element  $(l, m, n) \in \mathbb{N}^3 = ((\mathbb{N} \times \mathbb{N}) \times \mathbb{N})$ , we encode first  $l, m$  as  $\pi(l, m) \in \mathbb{N}$ , and then the complete triple as  $\pi(\pi(l, m), n)$ . Similarly we encode  $(l, m, n, p) \in \mathbb{N}^4$  as  $\pi(\pi(\pi(l, m), n), p)$ . So  $\pi^3(l, m, n) = \pi(\pi(\pi(l, m), n), n)$ ,  $\pi^4(l, m, n, p) = \pi(\pi(\pi(\pi(l, m), n), p), p)$ , etc. The corresponding decoding function  $\pi_i^k : \mathbb{N} \rightarrow \mathbb{N}$  should return the  $i$ th component. For  $k = 3$  we see that, if  $x = \pi^3(l, m, n) = \pi(\pi(\pi(l, m), n), n)$ , then  $\pi_0(\pi_0(x)) = l$ ,  $\pi_1(\pi_0(x)) = m$ ,  $\pi_1(x) = n$ , so we define  $\pi_0^3(x) = \pi_0(\pi_0(x))$ ,  $\pi_1^3(x) = \pi_1(\pi_0(x))$ ,  $\pi_1^3(x) = \pi_1(x)$ . Similarly, for  $k = 4$  we have to define  $\pi_0^4(x) = \pi_0(\pi_0(\pi_0(x)))$ ,  $\pi_1^4(x) = \pi_1(\pi_0(\pi_0(x)))$ ,  $\pi_2^4(x) = \pi_1(\pi_0(x))$ .  $\pi_3^4(x) = \pi_1(x)$ .

In general one defines for  $k \geq 1$   $\pi^k : \mathbb{N}^k \rightarrow \mathbb{N}$ ,  $\pi^k(x_0, \dots, x_{k-1}) = \pi(\dots \pi(\pi(x_0, x_1), x_2) \dots x_{k-1})$ , and for  $i < k$   $\pi_i^k : \mathbb{N} \rightarrow \mathbb{N}$ ,  $\pi_0^k(x) = \underbrace{\pi_0(\dots \pi_0(x))}_{k-1 \text{ times}}$  and for  $0 < i < k$ ,  $\pi_i^k(x) = \pi_1(\underbrace{\pi_0(\pi_0(\dots \pi_0(x))}_{k-i-1 \text{ times}}))$ .

An induction definition of  $\pi^k, \pi_i^k$  is as follows:

**Definition 2.18** (a) A bijective function  $\pi^k : \mathbb{N}^k \rightarrow \mathbb{N}$ .

We define by induction on  $k$  for  $k \in \mathbb{N}, k \geq 1$

$$\pi^k : \mathbb{N}^k \rightarrow \mathbb{N}$$

$$\pi^1(x) := x$$

$$\text{For } k > 0 \quad \pi^{k+1}(x_0, \dots, x_k) := \pi(\pi^k(x_0, \dots, x_{k-1}), x_k)$$

(b) The inverse of  $\pi^k$ .

We define by induction on  $k$  for  $i, k \in \mathbb{N}$  s.t.  $1 \leq k, 0 \leq i < k$

$$\pi_i^k : \mathbb{N} \rightarrow \mathbb{N}$$

$$\pi_0^1(x) := x$$

$$\pi_i^{k+1}(x) := \pi_i^k(\pi_0(x)) \text{ for } i < k$$

$$\pi_k^{k+1}(x) := \pi_1(x)$$

**Lemma 2.19** (a) For  $(x_0, \dots, x_{k-1}) \in \mathbb{N}^k, i < k, x_i = \pi_i^k(\pi^k(x_0, \dots, x_{k-1}))$ .

(b) For  $x \in \mathbb{N}, x = \pi^k(\pi_0^k(x), \dots, \pi_{k-1}^k(x))$ .

**Proof:** Induction on  $k$ .

Base case  $k = 0$ : Proof of (a): Let  $(x_0) \in \mathbb{N}^1$ . Then  $\pi_0^1(\pi^1(x_0)) = x_0$ .

Proof of (b): Let  $x \in \mathbb{N}$ . Then  $\pi^1(\pi_0^1(x)) = x$ .

Induction step  $k \rightarrow k + 1$ : Assume the assertion has been shown for  $k$ .

Proof of (a): Let  $(x_0, \dots, x_k) \in \mathbb{N}^{k+1}$ . Then

$$\begin{aligned} \text{for } i < k \quad \pi_i^{k+1}(\pi^{k+1}(x_0, \dots, x_k)) &= \pi_i^k(\pi_0(\pi(\pi^k(x_0, \dots, x_{k-1}), x_k))) \\ &= \pi_i^k(\pi^k(x_0, \dots, x_{k-1})) \\ &\stackrel{\text{IH}}{=} x_i \end{aligned}$$

$$\begin{aligned} \text{and } \pi_k^{k+1}(\pi^{k+1}(x_0, \dots, x_k)) &= \pi_1(\pi(\pi^k(x_0, \dots, x_{k-1}), x_k)) \\ &= x_k \end{aligned}$$

*Proof of (b):* Let  $x \in \mathbb{N}$ .

$$\begin{aligned}
\pi^{k+1}(\pi_0^{k+1}(x), \dots, \pi_k^{k+1}(x)) &= \pi(\pi^k(\pi_0^{k+1}(x), \dots, \pi_{k-1}^{k+1}(x)), \pi_k^{k+1}(x)) \\
&= \pi(\pi^k(\pi_0^k(\pi_0(x)), \dots, \pi_{k-1}^k(\pi_0(x))), \pi_1(x)) \\
&\stackrel{\text{IH}}{=} \pi(\pi_0(x), \pi_1(x)) \\
&\stackrel{\text{Rem. 2.17}}{=} x
\end{aligned}$$

We want to encode now  $\mathbb{N}^*$  into  $\mathbb{N}$ .  $\mathbb{N}^* = \mathbb{N}^0 \cup \bigcup_{n \geq 1} \mathbb{N}^n$ .  $\mathbb{N}^0 = \{\langle \rangle\}$ , and we can encode its only element  $\langle \rangle$  as 0. For  $n \geq 1$ , we have already defined an encoding of  $\pi^n : \mathbb{N}^n \rightarrow \mathbb{N}$ . Note that a natural number can (and in fact will) both be of the form  $\pi^n(a_0, \dots, a_{n-1})$  and  $\pi^k(b_0, \dots, b_{n-1})$  for  $n \neq k$ . In order to distinguish between those elements we have to add the length to that code and encode  $(a_0, \dots, a_{n-1})$  as an element of  $\bigcup_{n \geq 1} \mathbb{N}^n$  as  $\pi(n-1, \pi^n(a_0, \dots, a_{n-1}))$ . Note that we can use  $n-1$  instead of  $n$  as first component, since we are referring here to  $n \geq 1$ . In order to obtain a code for elements of  $\bigcup_{n \geq 1} \mathbb{N}^n$ , which is different from the code for  $\langle \rangle$ , we add 1 to the above, so  $(a_0, \dots, a_{n-1})$  is encoded as  $\pi(n-1, \pi^n(a_0, \dots, a_{n-1})) + 1$ . One can see now that we have obtain in fact a bijection from  $\mathbb{N}^*$  to  $\mathbb{N}$ . The complete definition is as follows:

**Definition 2.20** (a) **A bijective function**  $\lambda x. \langle x \rangle : \mathbb{N}^* \rightarrow \mathbb{N}$ .

Define for  $x \in \mathbb{N}^*$ ,  $\langle x \rangle : \mathbb{N}$  as follows:

$$\begin{aligned}
\langle \rangle &:= 0, \\
\langle x_0, \dots, x_k \rangle &:= 1 + \pi(k, \pi^{k+1}(x_0, \dots, x_k))
\end{aligned}$$

(b) **The length of a code of an element of  $\mathbb{N}^*$ :** We define

$$\begin{aligned}
\text{lh} &: \mathbb{N} \rightarrow \mathbb{N}, \\
\text{lh}(0) &:= 0, \\
\text{lh}(x) &:= \pi_0(x-1) + 1 \text{ if } x > 0.
\end{aligned}$$

$\text{lh}(x)$  is called the length of  $x$ .

(c) We define for  $x \in \mathbb{N}$  and  $i < \text{lh}(x)$ ,  $(x)_i \in \mathbb{N}$  as follows:

$$(x)_i := \pi_i^{\text{lh}(x)}(\pi_1(x-1))$$

For  $\text{lh}(x) \leq i$ , let  $(x)_i := 0$ .

**Lemma 2.21** (a)  $\text{lh}(\langle \rangle) = 0$ ,  $\text{lh}(\langle x_0, \dots, x_k \rangle) = k + 1$ .

(b) For  $i \leq k$ ,  $(\langle x_0, \dots, x_k \rangle)_i = x_i$ .

(c) For  $x \in \mathbb{N}$ ,  $x = \langle (x)_0, \dots, (x)_{\text{lh}(x)-1} \rangle$ .

**Proof:** (a)  $\text{lh}(\langle \rangle) = \text{lh}(0) = 0$ .

$$\begin{aligned}
\text{lh}(\langle x_0, \dots, x_k \rangle) &= \pi_0(\langle x_0, \dots, x_k \rangle - 1) + 1 \\
&= \pi_0(\pi(k, \dots) + 1 - 1) + 1 \\
&= k + 1
\end{aligned}$$

(b)  $\text{lh}(\langle x_0, \dots, x_k \rangle) = k + 1$ . Therefore

$$\begin{aligned} \langle x_0, \dots, x_k \rangle_i &= \pi_i^{k+1}(\pi_1(\langle x_0, \dots, x_k \rangle - 1)) \\ &= \pi_i^{k+1}(\pi_1(1 + \pi(k, \pi^{k+1}(x_0, \dots, x_k)) - 1)) \\ &= \pi_i^{k+1}(\pi^{k+1}(x_0, \dots, x_k)) \\ &\stackrel{\text{Lem 2.19(a)}}{=} x_i \end{aligned}$$

(c) If  $x = 0$ ,  $\text{lh}(x) = 0$ , and therefore  $\langle (x)_0, \dots, (x)_{\text{lh}(x)-1} \rangle = \langle \rangle = 0 = x$ .

If  $x > 0$ , let  $x - 1 = \pi(y, l)$ . Then  $\text{lh}(x) = l + 1$ ,  $(x)_i = \pi_i^{l+1}(y)$  and therefore

$$\begin{aligned} \langle (x)_0, \dots, (x)_{\text{lh}(x)-1} \rangle &= \langle \pi_0^{l+1}(y), \dots, \pi_l^{l+1}(y) \rangle \\ &= \pi(\pi^{l+1}(\pi_0^{l+1}(y), \dots, \pi_l^{l+1}(y)), l) + 1 \\ &\stackrel{\text{Lem 2.19(a)}}{=} \pi(y, l) + 1 \\ &= x \end{aligned}$$

**Theorem 2.22** (a) If  $A$  is countable, so are  $A^k$ ,  $A^*$ .

(b) If  $A, B$  are countable, so are  $A \times B$ ,  $A \cup B$ .

(c) If  $A_n$  are countable sets for  $n \in \mathbb{N}$ , so is  $\bigcup_{n \in \mathbb{N}} A_n$ .

(d)  $\mathbb{Q}$ , the set of rational numbers, is countable.

**Proof:**

(a) Assume  $A$  is countable.

We show first that  $A^*$  is countable: there exists  $f : A \rightarrow \mathbb{N}$ ,  $f$  injective. Define  $f^* : A^* \rightarrow \mathbb{N}^*$ ,  $f^*(a_0, \dots, a_n) = (f(a_0), \dots, f(a_n))$ . Then  $f$  is injective.

$\lambda x. \langle x \rangle : \mathbb{N}^* \rightarrow \mathbb{N}$  is bijective. Therefore  $(\lambda x. \langle x \rangle) \circ f^* : A^* \rightarrow \mathbb{N}$  is injective,  $A^*$  is countable.

$A^k \subseteq A^*$ , therefore  $A^k$  is countable as well.

(b) Assume  $A, B$  countable. Show  $A \times B$ ,  $A \cup B$  are countable.

If  $A = \emptyset$  or  $B = \emptyset$ , then  $A \times B = \emptyset$ , therefore countable, and  $A \cup B = A$  or  $A \cup B = B$ , therefore countable.

Assume  $A, B \neq \emptyset$ . Then there exist surjections  $f : \mathbb{N} \rightarrow A$  and  $g : \mathbb{N} \rightarrow B$ . Then  $f \times g : \mathbb{N}^2 \rightarrow A \times B$ ,  $(f \times g)(n, m) := (f(n), g(m))$  is surjective as well. Further  $\lambda x. (\pi_0(x), \pi_1(x)) : \mathbb{N} \rightarrow \mathbb{N}^2$  is surjective, therefore as well  $(f \times g) \circ (\lambda x. (\pi_0(x), \pi_1(x))) : \mathbb{N} \rightarrow A \times B$ . Therefore  $A \times B$  is countable.

Further  $h : \mathbb{N}^2 \rightarrow A \cup B$ ,  $h(0, n) := f(n)$ ,  $h(k, n) := g(n)$  for  $k > 0$ , is surjective, therefore as well  $h \circ (\lambda x. (\pi_0(x), \pi_1(x))) : \mathbb{N} \rightarrow A \cup B$ .

(c) Assume  $A_n$  are countable for  $n \in \mathbb{N}$ . Show  $A := \bigcup_{n \in \mathbb{N}} A_n$  is countable as well.

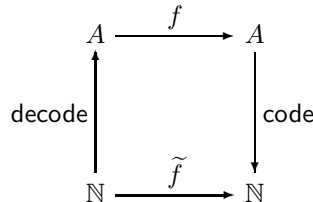
If all  $A_n$  are empty, so is  $\bigcup_{n \in \mathbb{N}} A_n$ . Assume  $A_k$  is non-empty for some  $k$ . By replacing all  $A_l$  which are empty by  $A_k$  we get a sequence of sets, s.t. their union is the same as  $A$ . So without loss of generality we can assume that  $A_n \neq \emptyset$  for all  $n$ .  $A_n$  are countable, so there exist  $f_n : \mathbb{N} \rightarrow A_n$  surjective. Then  $f : \mathbb{N}^2 \rightarrow A$ ,  $f(n, m) := f_n(m)$  is surjective as well. Therefore  $f \circ (\lambda x. (\pi_0(x), \pi_1(x))) : \mathbb{N} \rightarrow A$  is as well surjective,  $A$  is countable.

(d)  $\mathbb{Z} \times \mathbb{N}$  is countable, since  $\mathbb{Z}$  and  $\mathbb{N}$  are countable. Let  $A := \{(z, n) \in \mathbb{Z} \times \mathbb{N}, n \neq 0\}$ .  $A \subseteq \mathbb{Z} \times \mathbb{N}$ , therefore  $A$  is countable as well. Since  $A \neq \emptyset$  and countable, there exists a surjection  $f : \mathbb{N} \rightarrow A$ . Define  $g : A \rightarrow \mathbb{Q}$ ,  $g(z, n) := \frac{z}{n}$ .  $g$  is surjective, therefore as well  $g \circ f : \mathbb{N} \rightarrow \mathbb{Q}$ . So we have defined a surjective function from  $\mathbb{N}$  onto  $\mathbb{Q}$ ,  $\mathbb{Q}$  is countable.

### 2.3 Reduction of Computability on some Data Types to Computability on $\mathbb{N}$

We want to reduce computability on some data types  $A$  to computability on  $\mathbb{N}$ . This avoids having to introduce the notion of computability for each of those types individually. In order to obtain

this, we need to encode elements of  $A$  as elements of  $\mathbb{N}$ , i.e. we need a function  $\text{code} : A \rightarrow \mathbb{N}$ , and we need to decode them again, i.e. we need a decoding function  $\text{decode} : \mathbb{N} \rightarrow A$ . If these functions are computable, then from a computable function  $f : A \rightarrow A$  we can obtain a computable  $\tilde{f} : \mathbb{N} \rightarrow \mathbb{N}$ , by taking an element  $n \in \mathbb{N}$ , decoding it into  $A$ , applying then  $f$  and then encoding it back into a natural number:



We want as well to recover from  $\tilde{f}$  the original function  $f$ . For this we need that, if we first encode an element  $a \in A$  as a natural number and then decode it again, we obtain the original value  $a$ , i.e. we need  $\text{decode}(\text{code}(a)) = a$ .

If we have these properties, we say that  $A$  has a computable encoding into  $\mathbb{N}$ . Since we are referring to the notion of “computable” in an intuitive sense, and since this is not a precise mathematical definition, we call the following an “informal definition”, and lemmata referring to it “informal lemmata”.

**Informal Definition:**

A data type  $A$  has a computable encoding into  $\mathbb{N}$ , if there exist in an intuitive sense computable functions  $\text{code} : A \rightarrow \mathbb{N}$ , and  $\text{decode} : \mathbb{N} \rightarrow A$  such that  $\text{decode}(\text{code}(a)) = a$  for all  $a \in A$ .

**Remark:** Note that we **do not require** that  $\text{code}(\text{decode}(n)) = n$  for every  $n \in \mathbb{N}$ .

So every  $a \in A$  has a unique code in  $\mathbb{N}$ , i.e.  $\text{code}$  is injective, as expressed by  $\text{decode}(\text{code}(a)) = a$ . However, not every  $n \in \mathbb{N}$  needs to be a code for an element of  $A$ , (which would imply  $\text{code}(\text{decode}(n)) = n$ ),  $\text{code}$  need not be surjective.

The most common data used is the data type of finite strings over a finite alphabet  $A$ , and the set of such strings is  $A^*$ . Now  $A^*$  has a computable encoding into  $\mathbb{N}$ :

**Informal Lemma:**

If  $A$  is a finite set, then  $A^*$  has a computable encoding into  $\mathbb{N}$ .

**“Proof”:**

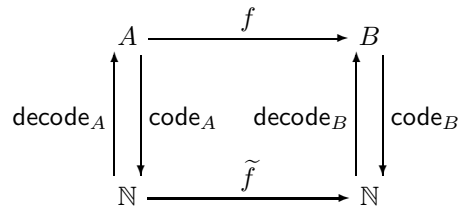
If  $A$  is empty, then  $A^* = \{\langle \rangle\}$ , and we can define  $\text{code} : A^* \rightarrow \mathbb{N}$ ,  $\text{code}(a) = 0$ , and  $\text{decode} : \mathbb{N} \rightarrow A^*$ ,  $\text{decode}(n) = \langle \rangle$ . Both function are clearly computable, and if  $a \in A^*$ , then  $a = \langle \rangle$  and  $\text{decode}(\text{code}(a)) = \text{decode}(0) = \langle \rangle = a$ .

Assume now  $A \neq \emptyset$ ,  $A = \{a_0, \dots, a_n\}$ , where  $a_i \neq a_j$  for  $i \neq j$ . Define  $f : A \rightarrow \mathbb{N}$ ,  $f(a_i) = i$ , and  $g : \mathbb{N} \rightarrow A$ ,  $g(i) = a_i$  for  $i \leq n$ ,  $g(i) = a_0$  for  $i > n$ .  $f$  and  $g$  are in an intuitive sense computable, and therefore as well  $\text{code} : A^* \rightarrow \mathbb{N}$ ,  $\text{code}(a_1, \dots, a_n) = \langle f(a_1), \dots, f(a_n) \rangle$  and the function  $\text{decode} : \mathbb{N} \rightarrow A^*$ ,  $\text{decode}(n) := (g((n)_0), \dots, g((n)_{\text{lh}(n)-1}))$ .

We show  $\text{decode}(\text{code}(x)) = x$  for  $x \in A^*$ : Let  $x = (a_1, \dots, a_n)$

$$\begin{aligned}
 \text{decode}(\text{code}(x)) &= \text{decode}(\text{code}(a_1, \dots, a_n)) \\
 &= \text{decode}(\langle f(a_1), \dots, f(a_n) \rangle) \\
 &= (g(f(a_1)), \dots, g(f(a_n))) \\
 &= (a_1, \dots, a_n) \\
 &= x .
 \end{aligned}$$

**Remark:** Assume  $A, B$  have computable encodings  $\text{code}_A : A \rightarrow \mathbb{N}$ ,  $\text{decode}_A : \mathbb{N} \rightarrow A$  and  $\text{code}_B : B \rightarrow \mathbb{N}$ ,  $\text{decode}_B : \mathbb{N} \rightarrow B$ . If  $f : A \rightarrow B$  is in an intuitive sense computable, then we can define an in an intuitive sense computable function  $\tilde{f} : \mathbb{N} \rightarrow \mathbb{N}$ ,  $\tilde{f} := \text{code}_B \circ f \circ \text{decode}_A$ :



From  $\tilde{f}$  we can recover  $f$ , since  $f = \text{decode}_B \circ \tilde{f} \circ \text{code}_A$  (easy exercise). So by considering computability on  $\mathbb{N}$  we cover computability on data types with a computable encoding into  $\mathbb{N}$  as well.

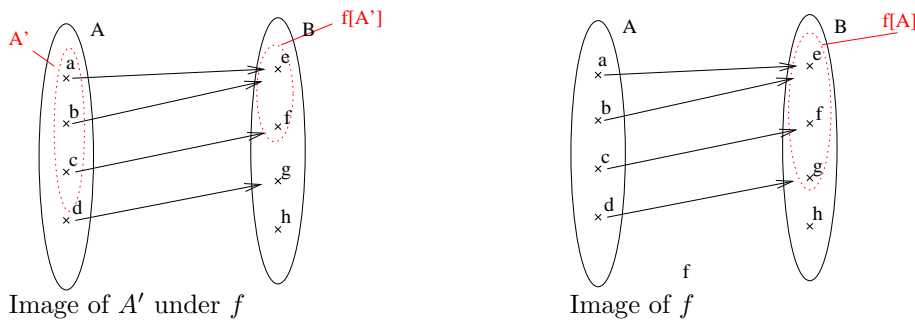
## 2.4 Appendix: Some Mathematical Background

### 2.4.1 Injective – Surjective – Bijective

#### Definition

Let  $f : A \rightarrow B$ ,  $A' \subseteq A$ .

- (a)  $f[A'] := \{f(a) \mid a \in A'\}$  is called the image of  $A'$  under  $f$ .
- (b) The image of  $A$  under  $f$  is called the image of  $f$ .



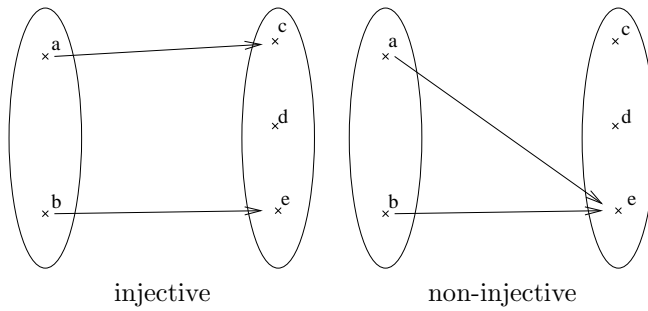
#### Definition 2.23

Let  $A, B$  be sets,  $f : A \rightarrow B$ .

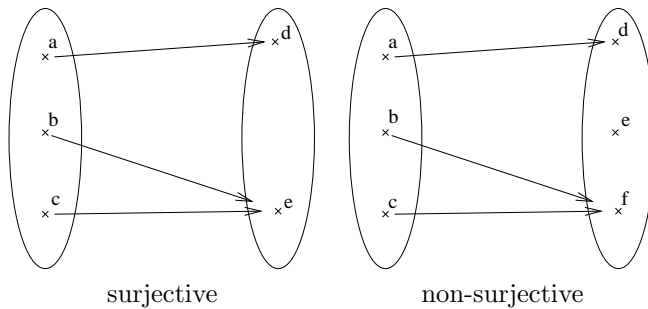
- (a)  $f$  is injective or an injection or one-to-one, if  $f$  applied to different elements of  $A$  has different results:  $\forall a, b \in A. a \neq b \rightarrow f(a) \neq f(b)$ .
- (b)  $f$  is surjective or a surjection or onto, if every element of  $B$  is in the image of  $f$ :  $\forall b \in B. \exists a \in A. f(a) = b$ .
- (c)  $f$  is bijective or a bijection or a one-to-one correspondence iff it is both surjective and injective.

If we visualise a function by having arrows from elements  $a \in A$  to  $f(a) \in B$  then we have the following:

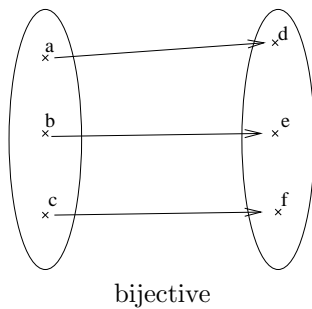
- A function is injective, if for every element of  $B$  there is *at most one arrow pointing to it*:



- A function is surjective, if for every element of  $B$  there is *at least one arrow pointing to it*:



- A function is bijective, if for every element of  $B$  there is *exactly one arrow pointing to it*:



- Note that, since we have a function, for every element of  $A$  there is exactly one arrow originating from there.

### 2.4.2 Sequences vs. Functions $\mathbb{N} \rightarrow A$

A function  $f : \mathbb{N} \rightarrow A$  is nothing but an infinite sequence of elements of  $A$  numbered by elements of  $\mathbb{N}$ , namely  $f(0), f(1), f(2), \dots$ , usually written as  $(f(n))_{n \in \mathbb{N}}$ . We identify functions  $f : \mathbb{N} \rightarrow A$  with infinite sequences of elements of  $A$ . So the following denotes the same mathematical object:

- The function  $f : \mathbb{N} \rightarrow \mathbb{N}$ ,  $f(n) = \begin{cases} 0 & \text{if } n \text{ is odd,} \\ 1 & \text{if } n \text{ is even.} \end{cases}$
- The sequence  $(1, 0, 1, 0, 1, 0, \dots)$ .
- The sequence  $(a_n)_{n \in \mathbb{N}}$  where  $a_n = \begin{cases} 0 & \text{if } n \text{ is odd,} \\ 1 & \text{if } n \text{ is even.} \end{cases}$

### 2.4.3 Partial Functions

A partial function  $f : A \rightsquigarrow B$  is the same as a function  $f : A \rightarrow B$ , but it need not be defined for all values, so we do not require that for every  $a \in A$  there is a value  $f(a)$ .

The key example of a partial function is the function computed by a computer program, which has an input  $a \in A$  and possibly returns an output  $f(a) \in B$ . (We assume that the program always returns the same result, which can be guaranteed by forbidding references to variables, which are defined outside the scope of this function.) If the program applied to some  $a \in A$  returns a value  $b \in B$ , then  $f(a)$  is defined and equal to  $b$ . If the program applied to  $a$  does not terminate, then  $f(a)$  is undefined.

Other examples of partial functions are the function  $f : \mathbb{R} \rightsquigarrow \mathbb{R}$ ,  $f(x) = \frac{1}{x}$ , which is not defined for  $x = 0$ ; or  $g : \mathbb{R} \rightsquigarrow \mathbb{R}$ ,  $g(x) = \sqrt{x}$ , which is only defined for  $x \geq 0$ .

#### Definition 2.24

- Let  $A, B$  be sets. A partial function  $f$  from  $A$  to  $B$ , written  $f : A \rightsquigarrow B$ , is a function  $f : A' \rightarrow B$  for some  $A' \subseteq A$ .  $A'$  is called the domain of  $f$ , written as  $A' = \text{dom}(f)$ .
- Let  $f : A \rightsquigarrow B$ .
  - $f(a)$  is defined, written as  $f(a) \downarrow$ , if  $a \in \text{dom}(f)$ .
  - $f(a) \simeq b$  ( $f(a)$  is partially equal to  $b$ )  $:\Leftrightarrow f(a) \downarrow \wedge f(a) = b$ .

We want to work with terms like  $f(g(2), h(3))$ , where  $f, g, h$  are partial functions. The question is here what happens if  $g(2)$  or  $h(3)$  is undefined. There is a theory of partial functions, in which  $f(g(2), h(3))$  might be defined, even if  $g(2)$  or  $h(3)$  is undefined. This makes sense for instance for the function  $f : \mathbb{N}^2 \rightsquigarrow \mathbb{N}$ ,  $f(x, y) = 0$ . Such functions, which are defined, even if some of its arguments are undefined, are called non-strict, whereas functions, which are defined only if all of its arguments are defined are called strict. In this lecture, functions will always be strict. Therefore a term like  $f(g(2), h(3))$  is defined only, if  $g(2)$  and  $h(3)$  are defined, and if  $f$  applied to the results of evaluating  $g(2)$  and  $h(3)$  is defined.  $f(g(2), h(3))$  is then evaluated as for ordinary functions: We first compute  $g(2)$  and  $h(3)$ , and then evaluate  $f$  applied to the results of those computations.

#### Definition 2.25

- For expressions  $t$  formed from constants, variables and partial functions we define  $t \downarrow$  and  $t \simeq b$  as follows:
  - If  $t = a$  is a constant, then  $t \downarrow$  is always true and  $t \simeq b :\Leftrightarrow a = b$ .
  - If  $t = x$  is a variable, then  $t \downarrow$  is always true and  $t \simeq x :\Leftrightarrow a = x$ .
  - 
  - $f(t_1, \dots, t_n) \simeq b :\Leftrightarrow \exists a_1, \dots, a_n. t_1 \simeq a_1 \wedge \dots \wedge t_n \simeq a_n \wedge f(a_1, \dots, a_n) \simeq b$ .
  - $f(t_1, \dots, t_n) \downarrow :\Leftrightarrow \exists b. f(t_1, \dots, t_n) \simeq b$
- $s \uparrow :\Leftrightarrow \neg(s \downarrow)$ .
- We define for expressions  $s, t$  formed from constants and partial functions  $s \simeq t :\Leftrightarrow (s \downarrow \leftrightarrow t \downarrow) \wedge (s \downarrow \rightarrow \exists a, b. s \simeq a \wedge t \simeq b \wedge a = b)$ .
- $t$  is total means  $t \downarrow$ .
- A function  $f : A \rightsquigarrow B$  is total, iff  $\forall a \in A. f(a) \downarrow$ .

**Remark:** Total partial functions are ordinary (non-partial) functions.

**Remark:** Quantifiers always range over defined elements. So by  $\exists m.f(n) \simeq m$  we mean: there exists a defined  $m$  s.t.  $f(n) \simeq m$ . So from  $f(n) \simeq g(k)$  we cannot conclude  $\exists m.f(n) \simeq m$  unless  $g(k) \downarrow$ .

**Remark 2.26**

- (a) If  $s \simeq a$ ,  $s \simeq b$ , then  $a = b$ .
- (b) For all terms we have  $t \downarrow \Leftrightarrow \exists a.t \simeq a$ .
- (c)  $f(t_1, \dots, t_n) \downarrow \Leftrightarrow \exists a_1, \dots, a_n.t_1 \simeq a_1 \wedge \dots \wedge t_n \simeq a_n \wedge f(a_1, \dots, a_n) \downarrow$ .

**Examples:**

Assume  $f : \mathbb{N} \xrightarrow{\simeq} \mathbb{N}$ ,  $\text{dom}(f) = \{n \in \mathbb{N} \mid n > 0\}$ ,  $f(n) := n + 1$  for  $n \in \text{dom}(f)$ . Let  $g : \mathbb{N} \xrightarrow{\simeq} \mathbb{N}$ ,  $\text{dom}(g) = \{0, 1, 2\}$ ,  $g(n) := n$ . Then we have

- $f(1) \downarrow$ ,  $f(0) \uparrow$ ,  $f(1) \simeq 2$ ,  $f(0) \not\simeq n$  for all  $n$ .
- $\underbrace{g(f(0))}_{\uparrow} \uparrow$ , since  $f(0) \uparrow$ .
- $\underbrace{g(f(1))}_{\simeq 2} \downarrow$ , since  $f(1) \downarrow$ ,  $f(1) \simeq 2$ ,  $g(2) \downarrow$ .
- $\underbrace{g(f(2))}_{\simeq 3} \uparrow$ , since  $f(2) \downarrow$ ,  $f(2) \simeq 3$ , but  $g(3) \uparrow$ .
- $\underbrace{g(f(2))}_{\uparrow} \simeq \underbrace{f(0)}_{\uparrow}$ , since both expressions are undefined.
- $\underbrace{g(f(1))}_{\simeq 2} \simeq \underbrace{f(g(1))}_{\simeq 2}$ , since both sides are defined and equal to 2.
- $\underbrace{g(f(2))}_{\uparrow} \not\simeq \underbrace{f(g(1))}_{\downarrow}$ , since the left hand side is undefined, the right hand side is defined.
- $\underbrace{f(f(1))}_{\simeq 3} \not\simeq \underbrace{f(1)}_{\simeq 2}$ , since both sides evaluate to different (defined) values.
- $+$ ,  $\cdot$  etc. can be treated as partial functions. So for instance
  - $\underbrace{f(1) + f(2)}_{\downarrow} \downarrow$ , since  $f(1) \downarrow$ ,  $f(2) \downarrow$ , and  $+$  is total.
  - $\underbrace{f(1) + f(2)}_{\simeq 2 \quad \simeq 3} \simeq 5$ .
  - $\underbrace{f(0) + f(1)}_{\uparrow} \uparrow$ , since  $f(0) \uparrow$ .

**Remark:** Strict evaluation of functions corresponds to the so called “call-by-value” evaluation of functions, as it is done in most imperative and some functional programming languages: when evaluating  $f(t_1, \dots, t_n)$ , one first evaluates  $t_1, \dots, t_n$ . Then one evaluates  $f$  applied to the results of this evaluation. Undefined values corresponds to an infinite computation, and if one of the  $t_i$  is undefined, the computation of  $f(t_1, \dots, t_n)$  doesn’t terminate.

In Haskell we have “call-by-name” evaluation, which means  $t_i$  are evaluated only if they are needed in the computation of  $f$ . For instance, if we have  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $f(x, y) = x$ , and  $t$  is an undefined term, then with call-by-need, the term  $f(2, t)$  can be evaluated to 2, since we never need to evaluate  $t$ . So functions in Haskell are non-strict. In our setting, functions are strict, so  $f(2, t)$  is undefined.

#### 2.4.4 $\lambda$ -Notation

When using in an informal context (i.e. outside the  $\lambda$ -calculus to be introduced later) by  $\lambda x.t$  we mean the function mapping  $x$  to  $t$ . E.g.  $\lambda x.x + 3$  is the function  $f$  s.t.  $f(x) = x + 3$ ,  $\lambda x.\sqrt{x}$  is the function mapping  $x$  to  $\sqrt{x}$ . This notation will be used if we want to introduce such a function without giving it a name. Note that we do not specify the domain and codomain – when this notation is used, it will either not matter or clear from the context.

#### 2.4.5 Some Standard Sets

- $\mathbb{N}$  is the set of natural numbers, i.e.

$$\mathbb{N} := \{0, 1, 2, \dots\} .$$

- Note that 0 is a natural number.
- When elements of a sequence, e.g. when counting the arguments of a function, we will usually start with 0:
  - \* The 0th element of a sequence is what is usually called the first element. (So, when considering variables  $x_0, \dots, x_{n-1}$ ,  $x_0$  is the 0th variable).
  - \* The first element of a sequence is what is usually called the second element. (So, when considering variables  $x_0, \dots, x_{n-1}$ ,  $x_1$  is the first variable).
  - \* etc.

- $\mathbb{Z}$  is the set of integers:

$$\mathbb{Z} := \mathbb{N} \cup \{-x \mid x \in \mathbb{N}\} .$$

- $\mathbb{Q}$  is the set of rationals, e.g.

$$\mathbb{Q} := \left\{ \frac{x}{y} \mid x, y \in \mathbb{Z}, y \neq 0 \right\} .$$

- $\mathbb{R}$  is the set of real numbers.

- If  $A, B$  are sets, then  $A \times B$  is the product of  $A$  and  $B$ , i.e.

$$A \times B := \{(x, y) \mid x \in A \wedge y \in B\}$$

- If  $A$  is a set,  $k \in \mathbb{N}$ ,  $k > 0$ , then  $A^k$  is the set of  $k$ -tuples of elements of  $A$ , i.e.

$$A^k := \{(x_0, \dots, x_{k-1}) \mid x_0, \dots, x_{k-1} \in A\} .$$

We identify  $A^1$  (which is formally correct  $\{(x) \mid x \in A\}$ ) with  $A$ . (Note that, if one is pedantic,  $(x)$  is different from  $x$ :  $(x)$  is the tuple consisting of one element  $x$ , whereas  $x$  is one element, from which we haven't formed a tuple. Ignoring this difference usually doesn't cause problems).

#### 2.4.6 Relations, Predicates and Sets

- A predicate on a set  $A$  is a property  $P$  of elements of  $A$ . In this lecture,  $A$  will usually be  $\mathbb{N}^k$  for some  $k \in \mathbb{N}$ ,  $k > 0$ .
- We write  $\underline{P(a)}$  for “predicate  $P$  is true for the element  $a$  of  $A$ ”.
- We often write “ $P(x)$  holds” for “ $P(x)$  is true”.
- We can use  $P(a)$  in formulas. Therefore:

- $\neg P(a)$  (“not  $P(a)$ ”) means that “ $P(a)$  is not true”.
  - $P(a) \wedge Q(a)$  means that “both  $P(a)$  and  $Q(a)$  are true”.
  - $P(a) \vee Q(a)$  means that “ $P(a)$  or  $Q(a)$  is true” (especially, if both  $P(a)$  and  $Q(a)$  are true, then  $P(a) \vee Q(a)$  is true as well).
  - $\forall x \in B.P(x)$  means that “for all elements  $x$  of the set  $B$   $P(x)$  is true”.
  - $\exists x \in B.P(x)$  means that “there exists an element  $x$  of the set  $B$  s.t.  $P(x)$  is true”.
- In this lecture, “relation” is another word for “predicate”.
  - We identify a predicate  $P$  on a set  $A$  with  $\{x \in A \mid P(x)\}$ . Therefore predicates and sets will be identified. E.g., if  $P$  is a predicate,
    - $x \in P$  stands for  $x \in \{x \in A \mid P(x)\}$ , which is equivalent to  $P(x)$ ,
    - $\forall x \in P.\varphi(x)$  for a formula  $\varphi$  stands for  $\forall x.P(x) \rightarrow \varphi(x)$ .
    - etc.
  - An  $n$ -ary relation or predicate on  $\mathbb{N}$  is a relation  $P \subseteq \mathbb{N}^n$ . A unary, binary, ternary relation on  $\mathbb{N}$  is a 1-ary, 2-ary, 3-ary relation on  $\mathbb{N}$ , respectively.
  - An  $n$ -ary function on  $\mathbb{N}$  is a function  $f : \mathbb{N}^n \rightarrow \mathbb{N}$ . A unary, binary, ternary function on  $\mathbb{N}$  is a 1-ary, 2-ary, 3-ary function on  $\mathbb{N}$ , respectively.

### 2.4.7 Other Notations

**The “dot”-notation.** When writing expressions such as  $\forall x.A(x) \wedge B(x)$ , the convention is that if we have a dot after the quantifier together with a variable (in the example the dot after “ $\forall x$ ”), then the scope of the quantifier includes as much as possible to the right. So in the example  $\forall x.$  refers to  $A(x) \wedge B(x)$ , and the formula expresses that for all  $x$  both  $A(x)$  and  $B(x)$  hold, or, using brackets,  $\forall x.(A(x) \wedge B(x))$ . However, in  $(A \rightarrow \forall x.B(x) \wedge C(x)) \vee D(x)$ ,  $\forall x$  refers only to  $B(x) \wedge C(x)$ , since this is the maximum scope possible – it doesn’t make sense to include into the scope of  $\forall x$  as well “ $\vee D(x)$ ”.

Similarly, in  $\exists x.A(x) \wedge B(x)$ ,  $\exists x$  refers to  $A(x) \wedge B(x)$ , whereas in  $(A \wedge \exists x.B(x) \vee C(x)) \wedge D(x)$ ,  $\exists x$  refers to  $B(x) \vee C(x)$ .

This applies as well to  $\lambda$ -expressions, so  $\lambda x.x + x$  is the function taking an  $x$  and returning  $x + x$ .

$\vec{x}, \vec{y}$  etc. We will often refer to arguments of functions and relations, to which we don’t refer explicitly. An example are the variables  $x_0, \dots, x_{n-1}$  in the definition of a function  $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ ,

$$f(x_0, \dots, x_{n-1}, y) = \begin{cases} g(x_0, \dots, x_{n-1}), & \text{if } y = 0, \\ h(x_0, \dots, x_{n-1}), & \text{if } y > 0. \end{cases}$$

In order to avoid having to write in such kind of examples often  $x_0, \dots, x_{n-1}$ , we write  $\vec{x}$  for it. In general,  $\vec{x}$  stands for  $x_0, \dots, x_{n-1}$ , where  $n$ , i.e. how many variables are meant by  $\vec{x}$ , is usually clear from the context.

#### Examples:

- $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ , then in  $f(\vec{x}, y)$ ,  $\vec{x}$  needs to stand for  $n$  arguments, therefore  $\vec{x} = x_0, \dots, x_{n-1}$ .
- If  $f : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ , then in  $f(\vec{x}, y)$ ,  $\vec{x}$  needs to stand for  $n + 1$  arguments, so  $\vec{x} = x_0, \dots, x_n$ .
- If  $P$  is an  $n + 4$ -ary relation, then in  $P(\vec{x}, y, z)$ ,  $\vec{x}$  stands for  $x_0, \dots, x_{n+1}$ .

Similarly, we write  $\vec{y}$  for  $y_0, \dots, y_{n-1}$ , where  $n$  is clear from the context, similar for  $\vec{z}, \vec{n}, \vec{m}$ , etc.

### 3 The Unlimited Register Machine (URM) and the Halting Problem

A model of computation consists of a set of partial functions together with methods, which describe, how to compute those functions. We will usually consider models of computation, which contain all computable functions. One usually aims at describing an as simple model of computation as possible, i.e. to minimise the constructs used, while still being able to represent all computable functions. This makes it easier to show for another model of computation, that the first model can be interpreted in it. Furthermore, models of computations are more used for showing that something is not computable rather than showing that something is computable, and this is easier to show if the model of computation is simple.

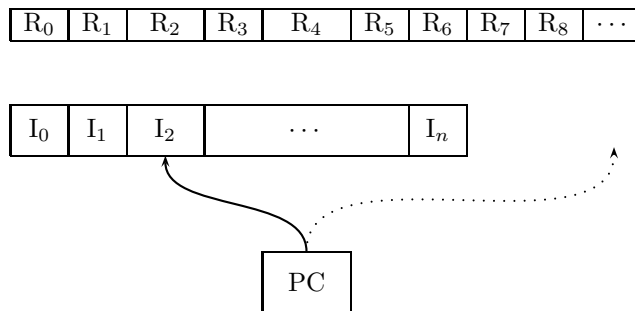
The URM (the unlimited register machine) is one model of computation, which is particularly easy to understand. It is a virtual machine, i.e. it is the description of, how a computer would execute its program, but it is not intended to be actually implemented. It is an abstract virtual machine, so we do not intend to simulate this machine on another computer (although there are implementations) – the machine serves as a mathematical model, which is then investigated mathematically. For instance, one usually doesn't write programs in it – instead one shows that in principal there is a way of writing a certain program in this language. In fact it turns out that it is difficult to write actual programs for the URM. We will have a very low level programming language – there will be no functions, objects, etc., not even while loops. The only loop construct will be the goto. So the URM does not support structured programming in any sense.

A URM is an idealised machine, as expressed by the word “unlimited”. So we don't assume any bounds on the amount of memory or execution time – however all values will be finite.

There exist various variants of URMs, we will here introduce a URM which is particularly simple.

#### 3.1 Syntax and Semantics of the URM

- The URM consists of
  - infinitely many registers  $R_i$ , each of which can store an arbitrarily big natural number;
  - a finite sequence of instructions  $I_0, I_1, I_2, \dots, I_n$ ;
  - and a program counter  $PC$ , which can store a natural number. If it contains a number  $0 \leq i \leq n$ , this means that it points to instruction  $I_i$ . If the PC is set to another number, the program will stop.



Solid line: Execute instruction  
 Dotted line: Program has terminated

- There are three kinds of URM instructions:

- The successor instruction  $\text{succ}(n)$  where  $n \in \mathbb{N}$ .  
If the PC points to this instruction, the URM performs the following operation: It adds 1 to register  $R_n$ , and increments then the PC by one. So the PC points then either to the next instruction, if there is one, or the program stops, if there is none.
- The predecessor instruction  $\text{pred}(n)$ , where  $n \in \mathbb{N}$ .  
If the PC points to this instruction, the URM performs the following operation: If  $R_n$  contains a value  $> 0$ , then it subtracts 1 from it, otherwise it leaves it as it is. Then the PC is incremented by 1.
- The conditional jump instruction  $\text{ifzero}(n, q)$ ,  $1 \leq n \leq N$ ,  $q \in \mathbb{N}$ . If the PC points to this instruction, the URM performs the following operation:
  - \* If  $R_n$  contains value 0, then the PC is set to value  $q$  – so there is an instruction  $I_q$ , it continues executing that instruction; if there is no such instruction the program will stop.
  - \* If  $R_n$  contains a value  $> 0$ , then the PC is incremented by 1, so the program continues executing the next instruction or it stops, if the current instruction is the last one.
- Note that a URM program  $I_0, \dots, I_n$  will refer only to finitely many registers, namely those explicitly occurring in  $I_0, \dots, I_n$ .
- If  $P = I_0, \dots, I_n$  is a URM program, it computes a function  $P^{(k)} : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$ .  $P^{(k)}(a_0, \dots, a_{k-1})$  is computed as follows:
  - **Initialisation:** Set PC to 0, store  $a_0, \dots, a_{k-1}$  in registers  $R_0, \dots, R_{k-1}$ , respectively, and set all other registers to 0 (it suffices to do this for those registers, referenced in the program).
  - **Iteration:** As long as the PC holds a value  $j \in \{0, \dots, n\}$ , execute Instruction  $I_j$ . Continue with the next instruction as given by the PC.
  - **Output:** If the PC value is bigger than  $n$ , the program stops, and the function returns the value  $b$  contained in register  $R_0$ :  $P^{(k)}(a_0, \dots, a_{k-1}) := b$ . If the program never stops, then  $P^{(k)}(a_0, \dots, a_{k-1})$  is undefined.
- A partial function  $f : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$  is URM-computable, if  $f = P^{(k)}$  for some  $k \in \mathbb{N}$  and some URM program  $P$ .

**Remark:**

1. A URM program  $P$  defines many URM-computable functions:
  - As a unary function  $P^{(1)} : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$ , one stores its argument in  $R_0$ , sets all other registers to 0 and then executes  $P$ .
  - As a binary function  $P^{(2)} : \mathbb{N}^2 \xrightarrow{\sim} \mathbb{N}$ , one stores its two arguments in  $R_0, R_1$ , sets all other registers to 0 and then executes  $P$ .
  - As a ternary function  $P^{(3)} : \mathbb{N}^3 \xrightarrow{\sim} \mathbb{N}$ , one stores its three arguments in  $R_0, R_1, R_2$ , sets all other registers to 0 and then executes  $P$ .
  - etc.
2. For a *partial* function  $f$  to be computable we do not need to determine, whether  $f(n)$  is defined or not. We only need to determine in case  $f(n) \downarrow$  after finite amount of time that  $f(n)$  is actually defined and to compute the value of  $f(n)$ . In case  $f(n) \uparrow$ , we will wait infinitinally long for an answer.

The Turing halting problem, which will be discussed problem, is the problem to decide, whether  $f(n) \downarrow$  or  $f(n) \uparrow$ . We will see later that this problem is *undecidable*.

If one wants to make sure that we always get an answer, one has to consider *total* computable functions  $f$ , i.e. computable functions, which are always defined.

In order to introduce total computable functions, we have to introduce first the partial computable functions and then take the subset of those functions, which are total. There is no programming language (in which it is decidable whether a program is actually a program), in which all functions are total and which allows to define all computable functions.

**Example:** The function  $f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$ ,  $f(x) \simeq 0$  is URM-computable. We derive a URM-program for computing it in several steps.

*Step 1:* The program should, if initially  $R_0$  contains  $x$  and all other registers contain 0, terminate with value 0 in register 0. A higher level program (not in the language of the URM) would be as follows:

$R_0 := 0$

*Step 2:* Since we have only a successor, and predecessor operation available, we replace the program by the following:

**while**  $R_0 \neq 0$  **do**  $\{R_0 := R_0 \dot{-} 1\}$

Here  $x \dot{-} y := \max\{x - y, 0\}$ , i.e.  $x \dot{-} y = \begin{cases} x - y & \text{if } y \leq x, \\ 0 & \text{otherwise.} \end{cases}$

*Step 3:* We replace the while-loop by a goto:

**LabelBegin :** **if**  $R_0 = 0$  **then goto** **LabelEnd;**  
 $R_0 := R_0 \dot{-} 1;$   
**goto** **LabelBegin;**

**LabelEnd :**

*Step 4:* The last goto will be replaced by a conditional goto, depending on  $R_1 = 0$ . Since  $R_1$  is initially true, and never changed during the program, this jump will always be carried out.

**LabelBegin :** **if**  $R_0 = 0$  **then goto** **LabelEnd;**  
 $R_0 := R_0 \dot{-} 1;$   
**if**  $R_1 = 0$  **then goto** **LabelBegin;**

**LabelEnd :**

*Step 5:* We translate the program into a URM program  $I_0, I_1, I_2$ :

$I_0 =$  **ifzero**(0,3)  
 $I_1 =$  **pred**(0)  
 $I_2 =$  **ifzero**(1,0)

## 3.2 Translating Higher-Level Constructs into the Language of URM

**Remark on jump addresses** In the following, we will often insert URM programs  $P$  as part of another URM program  $P'$ , for instance as follows:

**succ**(0)  
 $P$   
**pred**(0)

By this we mean that all jump addresses in  $P$  are adapted accordingly, so that they fit to the new numbers of instructions. In the example above, for instance, one has to add 1 to each jump address, since the new numbers of the instructions is the old number of it, increased by 1.

Furthermore, when inserting a piece of program, we assume that whenever  $P$  terminates, it terminates with PC equal to the number of the first instruction following it. So in the example above, if the execution of  $P$  as part of  $P'$  terminates, then the next instruction is the instruction **succ**(0). This can be achieved by renaming jump addresses, which jump outside of  $P$ , accordingly.

In order to introduce more complex URM programs, we introduce some constructions for forming URM programs:

**Labelled URM programs:** We replace numbers as jump addresses by labels. A label is a symbol we assign to certain program lines. If `mylabel` refers to  $I_k$ , then `ifzero( $n$ ,mylabel)` stands for `ifzero( $n$ , $k$ )`. It is trivial to translate a program with labels into an ordinary URM program. `LabelEnd` will be a special label, denoting the first instruction following the program.

So the above program reads with labels as follows:

```
LabelBegin: I0 = ifzero(0,LabelEnd)
            I1 = pred(0)
            I2 = ifzero(1,LabelBegin)
```

We can now omit the notations  $I_k$  and write the program as follows:

```
LabelBegin: ifzero(0,LabelEnd)
            pred(0)
            ifzero(1,LabelBegin)
```

**Replacement of registers by variable names.** Similarly we write variable names instead of registers. So, if we write `x` for  $R_0$ , `y` for  $R_1$ , the above program reads as follows:

```
LabelBegin: ifzero(x,LabelEnd)
            pred(x)
            ifzero(y,LabelBegin)
```

**Writing of statements in a more readable way:**

- `x := x + 1`; stands for `succ(x)`.
- `x := x - 1`; stands for `pred(x)`.
- `if x = 0 then goto mylabel`;  
stands for `ifzero(x,mylabel)`.

The above program reads now as follows:

```
LabelBegin: if x = 0 then goto LabelEnd;
            x := x - 1;
            if y = 0 then goto LabelBegin;
```

**Introduction of more complex statements.** We introduce now some more complex statements, which translate back into URM programs, using any of the new statements introduced before. Most new statements will require some extra registers, which are not used elsewhere in the program and are not used for storing the input and the output of the function. This is not a problem, since we have arbitrary many registers available. The auxiliary registers will at the end of the instruction always be set to 0. Since all registers, except of the argument registers, are 0, and since other statements don't change those registers, they will be always zero, when this statement is started. By "let `aux` be a new variable" we mean, that `aux` is a variable denoting a new register. `LabelEnd` will in the following the next instruction, following the instructions forming the complex statement.

(a) The statement

```
goto mylabel;
```

executes an unconditional jump to statement with label `mylabel`. It stands for the (labelled) URM statement:

```
if aux = 0 then goto mylabel;
```

where `aux` is new variable.

- (b) If  $\langle Instructions \rangle$  is a sequence of instructions, the following statement
- ```
while x ≠ 0 do {
   $\langle Instructions \rangle$ };
```

stands for the following URM program:

```
LabelLoop:  if x = 0 then goto LabelEnd;
              $\langle Instructions \rangle$ 
             goto LabelLoop;
```

- (c) The statement

```
x := 0
```

sets register associated with  $x$  to 0. It stands for the following program:

```
while x ≠ 0 do {x := x - 1};
```

- (d) The statement

```
y := x;
```

sets register associated with  $y$  to the value  $x$ . All other registers (except for auxiliary registers), including  $x$  will remain unchanged. Let  $aux$  be a new variable.

In case  $x$  and  $y$  denote the same variable, it stands for the empty URM program (no instructions). Otherwise the program is executed in two steps: First we decrement  $x$  and increment simultaneously  $aux$ , until  $x$  is 0. At the end  $x$  contains 0, and  $aux$  contains the original value of  $x$ . Then we set  $y$  to 0 and then, in a loop, we decrement  $aux$  and increment simultaneously  $x$  and  $y$ , until  $aux = 0$ . At the end,  $x$  and  $y$  contain the previous value of  $aux$ , which is the original value of  $x$ , and  $aux = 0$ . The complete program is as follows:

```
while x ≠ 0 do {
  x := x - 1;
  aux := aux + 1; };
y := 0;
while aux ≠ 0 do {
  aux := aux - 1;
  x := x + 1;
  y := y + 1; };
```

- (e) Assume  $x, y, z$  denote different registers. The statement

$x := y + z$ ; computes in  $x$  the sum of  $y$  and  $z$ . All other registers, including  $y, z$ , except for auxiliary ones, remain as they are. It is computed as follows ( $aux$  is an additional variable):

```
x := y;
aux := z;
while aux ≠ 0 do {
  aux := aux - 1;
  x := x + 1; };
```

- (f) Assume  $x, y, z$  denote different registers. The statement

```
x := y - z;
```

computes in  $x$  the difference between  $y$  and  $z$ . If  $z$  is bigger than  $y$ , then  $x$  becomes 0. All other registers, including  $y, z$ , except for auxiliary ones, remain as they are. It is computed as follows ( $\mathbf{aux}$  is an additional variable):

```
x := y;
aux := z;
while aux ≠ 0 do {
  aux := aux - 1;
  x := x - 1;};
```

- (g) Assume  $x, y$  denote different registers, and let  $\langle \text{Statements} \rangle$  be a sequence of statements. The statement

```
while x ≠ y do {
  ⟨Statements⟩};
```

executes statements, as long as  $x \neq y$ . It can be defined as follows ( $\mathbf{aux}, \mathbf{aux}_0, \mathbf{aux}_1$  are new variables):

```
aux_0 := x - y;
aux_1 := y - x;
aux := aux_0 + aux_1;
while aux ≠ 0 do {
  ⟨Statements⟩
  aux_0 := x - y;
  aux_1 := y - x;
  aux := aux_0 + aux_1;};
```

We could now continue and introduce more and more complex statements into the language of URM, and it should be clear by now, that in principle one could translate languages of high-level programming languages into URM. We will stop here, since we have now enough material in order to prove the next lemmata.

### 3.3 Constructions for Introducing URM-Computable Functions

We introduce notations for some partial functions

**Definition 3.1** (a) Define the zero function  $\mathbf{zero} : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$ ,  $\mathbf{zero}(x) = 0$ .

(b) Define the successor function  $\mathbf{succ} : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$ ,  $\mathbf{succ}(x) = x + 1$ .

(c) Define for  $0 \leq i < n$  the projection function  $\mathbf{proj}_i^n : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$ ,  $\mathbf{proj}_i^n(x_0, \dots, x_{n-1}) = x_i$ .

(d) Assume  $g : (B_0 \times \dots \times B_{k-1}) \xrightarrow{\sim} C$ , and  $h_i : A \xrightarrow{\sim} B_i$  ( $i = 0, \dots, k-1$ ). Then define the composition of  $g$  with  $h_0, \dots, h_{k-1}$  as  $g \circ (h_0, \dots, h_{k-1}) : A \xrightarrow{\sim} C$ ,

$$(g \circ (h_0, \dots, h_{k-1}))(a) := g(h_0(a), \dots, h_{k-1}(a))$$

(e) Assume  $g : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$ ,  $h : \mathbb{N}^{k+2} \xrightarrow{\sim} \mathbb{N}$ . Then we define the function defined by primitive recursion from  $g$  and  $h$ , namely  $\mathbf{primrec}(g, h) : \mathbb{N}^{k+1} \xrightarrow{\sim} \mathbb{N}$ , as follows: Let  $f := \mathbf{primrec}(g, h)$ .

$$\begin{aligned} f(n_0, \dots, n_{k-1}, 0) &:= g(n_0, \dots, n_{k-1}) \\ f(n_0, \dots, n_{k-1}, m+1) &:= h(n_0, \dots, n_{k-1}, m, f(n_0, \dots, n_{k-1}, m)) \end{aligned}$$

In the special case  $k = 0$ , it doesn't make sense to use  $g()$  – in this case let  $g$  be a natural number instead. So the case  $k = 0$  reads as follows:

We define for  $n \in \mathbb{N}$ ,  $h : \mathbb{N}^2 \xrightarrow{\sim} \mathbb{N}$ ,  $f := \text{primrec}(n, h) : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$ :

$$\begin{aligned} f(0) & \simeq n \\ f(m+1) & \simeq h(m, f(m)) \end{aligned}$$

(f) Let  $g : \mathbb{N}^n + 1 \xrightarrow{\sim} \mathbb{N}$ . We define  $\underline{\mu}(g) : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$ ,

$$\underline{\mu}(g)(x_0, \dots, x_{n-1}) \simeq (\mu y. g(x_0, \dots, x_{n-1}, y) \simeq 0)$$

where we define the partial expression  $\mu y. g(x_0, \dots, x_{n-1}, y) \simeq 0$  as follows

$$(\mu y. g(x_0, \dots, x_{n-1}, y) \simeq 0) \simeq \begin{cases} \text{the least } y \in \mathbb{N} \text{ s.t. } g(x_0, \dots, x_{n-1}, y) \simeq 0 \text{ and} \\ g(x_0, \dots, x_{n-1}, y') \downarrow \text{ for } 0 \leq y' < y & \text{if such } y \text{ exists,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

### Examples for definition by primitive recursion.

- Addition can be defined using primitive recursion: Let  $f(x, y) := x + y$ . We have

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

Therefore

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

where  $g : \mathbb{N} \rightarrow \mathbb{N}$ ,  $g(x) := x$ , and  $h : \mathbb{N}^3 \rightarrow \mathbb{N}$ ,  $h(x, y, z) := z + 1$ . So  $f = \text{primrec}(g, h)$ .

- Multiplication can be defined using primitive recursion: Let  $f(x, y) := x \cdot y$ . We have

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

Therefore, we have

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

where  $g : \mathbb{N} \rightarrow \mathbb{N}$ ,  $g(x) := 0$ , and  $h : \mathbb{N}^3 \rightarrow \mathbb{N}$ ,  $h(x, y, z) := z + x$ . So  $f := \text{primrec}(g, h)$ .

- Define  $\underline{\text{pred}} : \mathbb{N} \rightarrow \mathbb{N}$ ,  $\underline{\text{pred}}(n) := n - 1 = \begin{cases} n - 1 & \text{if } n > 0, \\ 0 & \text{otherwise.} \end{cases}$   
 $\underline{\text{pred}}$  can be defined using primitive recursion:

$$\begin{aligned} \underline{\text{pred}}(0) & = 0 , \\ \underline{\text{pred}}(x + 1) & = x . \end{aligned}$$

Therefore, we have

$$\begin{aligned} \underline{\text{pred}}(0) & = 0 , \\ \underline{\text{pred}}(x + 1) & = h(x, \underline{\text{pred}}(x)) , \end{aligned}$$

where  $h : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $h(x, y) := y$ . Therefore,  $\underline{\text{pred}} = \text{primrec}(0, h)$ .

- $x \dot{-} y$  can be defined using primitive recursion: Let  $f(x, y) := x \dot{-} y$ . We have

$$\begin{aligned} f(x, 0) &= x \dot{-} 0 = x , \\ f(x, y + 1) &= x \dot{-} (y + 1) = (x \dot{-} y) \dot{-} 1 = \text{pred}(f(x, y)) . \end{aligned}$$

Therefore,

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

where  $g : \mathbb{N} \rightarrow \mathbb{N}$ ,  $g(x) := x$ , and  $h : \mathbb{N}^3 \rightarrow \mathbb{N}$ ,  $h(x, y, z) := \text{pred}(z)$ . Therefore,  $f = \text{primrec}(g, h)$ .

**Examples for definition by  $\mu$ .**

- Let  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $f(x, y) := x \dot{-} y$ . Then  $\mu(f)(x) \simeq (\mu y. f(x, y) \simeq 0) \simeq x$ .
- Let  $f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$ ,  $f(0) \uparrow$ ,  $f(n) = 0$  for  $n > 0$ . Then  $(\mu y. f(y) \simeq 0) \uparrow$ .
- Let  $f : \mathbb{N} \rightarrow \mathbb{N}$ ,  $f(n) := \begin{cases} 1 & \text{if there exist primes } p, q < 2n + 4 \text{ s.t. } 2n + 4 = p + q, \\ 0 & \text{otherwise} \end{cases}$   
 $\mu y. f(y) \simeq 0$  is the first  $n$  s.t. there don't exist primes  $p, q$  s.t.  $2n + 4 = p + q$ . Goldbach's conjecture says that every even number  $\geq 4$  is the sum of two primes. Goldbach's conjecture is equivalent to  $(\mu y. f(y) \simeq 0) \uparrow$ . It is one of the most important open problems in mathematics to show (or refute) Goldbach's conjecture. If we could decide whether a partial computing function is defined (which we can't), we could decide Goldbach's conjecture.

We want to show that the set of URM-computable functions is closed under the just introduced operations. The notion of URM-computable function is not directly suitable, since it refers to specific registers and, since the values of the arguments are destroyed by running it.

**Remark on  $\mu$ .** We need in the definition of  $\mu$  the condition " $g(x_0, \dots, x_{n-1}, y') \downarrow$  for  $0 \leq y' < y$ ". If we defined instead

$$(\mu' y. g(x_0, \dots, x_{n-1}, y) \simeq 0) \simeq \begin{cases} \text{the least } y \in \mathbb{N} \text{ s.t. } g(x_0, \dots, x_{n-1}, y) \simeq 0 & \text{if such } y \text{ exists,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

then in general  $t := (\mu' y. g(x_0, \dots, x_{n-1}, y) \simeq 0)$  would be non-computable: Assume for instance  $g(x_0, \dots, x_{n-1}, 1) \simeq 0$ . Before we have finished computing  $g(x_0, \dots, x_{n-1}, 0)$ , we don't know yet whether the value is 0, non-zero or undefined. But we have  $t \simeq 0 \Leftrightarrow g(x_0, \dots, x_{n-1}, 0) \simeq 0$ ,  $t \simeq 1 \Leftrightarrow g(x_0, \dots, x_{n-1}, 0) \not\simeq 0$ , where the latter includes the case  $g(x_0, \dots, x_{n-1}, 0) \uparrow$ .

The above was only a heuristics, and does not show yet that we cannot compute  $t$  by using any trick. However, one can reduce the decidability of the Turing halting problem to the problem of determining the value of  $t$ , and therefore show that  $t$  is for some functions  $g$  non-computable.

**Lemma and Definition 3.2** *Assume  $f : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$  is URM-computable. Assume  $\mathbf{x}_0, \dots, \mathbf{x}_{k-1}, \mathbf{y}, \mathbf{z}_0, \dots, \mathbf{z}_l$  are variable names for different registers. Then one can define a URM program, which, computes  $f(\mathbf{x}_0, \dots, \mathbf{x}_{k-1})$  and stores the result in  $\mathbf{y}$  in the following sense: If  $f(\mathbf{x}_0, \dots, \mathbf{x}_{k-1}) \downarrow$ , then the program ends at the first instruction following this program, and stores the result in  $\mathbf{y}$ . If  $f(\mathbf{x}_0, \dots, \mathbf{x}_{k-1}) \uparrow$ , the program never terminates. Further the program can be defined so that it doesn't change the arguments  $\mathbf{x}_0, \dots, \mathbf{x}_{k-1}$  and the variables  $\mathbf{z}_0, \dots, \mathbf{z}_l$ .*

*For  $P$  we say it is a URM program which computes  $\mathbf{y} \simeq f(\mathbf{x}_0, \dots, \mathbf{x}_{k-1})$  and avoids  $\mathbf{z}_0, \dots, \mathbf{z}_l$ .*

**Proof:**

Let  $P$  be a URM program s.t.  $P^{(k)} = f$ . Let  $u_0, \dots, u_{k-1}$  be registers different from the above. By renumbering of registers and of jump addresses, we obtain a program  $P'$ , which computes the result of  $f(u_0, \dots, u_{k-1})$  in  $u_0$  (if it is defined – if not it does not terminate), leaves the registers mentioned in the lemma unchanged, and which, if it terminates, terminates in the first instruction following  $P'$ . The following is a program as intended:

```

u0 := x0;
...
uk-1 := xk-1;
P'
y := u0;

```

**Lemma 3.3** (a) zero, succ and  $\text{proj}_i^n$  are URM-computable.

(b) If  $f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$ ,  $g_i : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$  are URM-computable, so is  $f \circ (g_0, \dots, g_{n-1})$ .

(c) If  $g : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$ , and  $h : \mathbb{N}^{n+2} \xrightarrow{\sim} \mathbb{N}$  are URM-computable, so is  $\text{primrec}(g, h)$ .

(d) If  $g : \mathbb{N}^{n+1} \xrightarrow{\sim} \mathbb{N}$  is URM-computable, so is  $\mu(g)$ .

**Proof:**

Let  $x_i$  denote register  $R_i$ .

**Proof of (a)**

- zero is computed by the following program:

```
x0 := 0.
```

- succ is computed by the following program:

```
x0 := x0 + 1.
```

- $\text{proj}_k^n$  is computed by the following program:

```
x0 := xk.
```

**Proof of (b)**

Assume  $f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$ ,  $g_i : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$  are URM-computable. Show  $f \circ (g_0, \dots, g_{n-1})$  is computable.

A plan for the program is as follows:

- The input is stored in registers  $x_0, \dots, x_{k-1}$ . Let  $\vec{x} := x_0, \dots, x_{k-1}$ .
- First we compute  $g_i(\vec{x})$  for  $i = 0, \dots, k-1$  and store the result in registers  $y_i$ .
- Then we compute  $f(y_0, \dots, y_{n-1})$  (which is  $\simeq f(g_0(\vec{x}), \dots, g_{n-1}(\vec{x}))$ ), and store the result in the output register  $x_0$ .

A problem could be that, when computing  $y_i \simeq g_i(\vec{x})$ ,

- we might change one of the registers  $\vec{x}$ , which are still to be used by later computations of  $y_j \simeq g_j(\vec{x})$  for  $j > i$
- we might change  $y_j$  for  $j < i$  which contains the result from previous computations of  $g_j(\vec{x})$ .

We have already seen in Lemma and Definition 3.2 that we can define programs, which avoid changing their arguments and certain other registers. Therefore programs  $P_i$  as follows do exist.

- Let  $P_i$  be a URM program ( $i = 0, \dots, n-1$ ), which computes  $y_i \simeq g_i(\vec{x})$  and avoids  $y_j$  for  $j \neq i$ . (Note that our definition includes that the arguments are changed neither).

- Let  $Q$  be a URM program, which computes  $\mathbf{x}_0 \simeq f(\mathbf{y}_0, \dots, \mathbf{y}_{n-1})$ .

A URM program  $R$  for computing  $f \circ (g_0, \dots, g_{n-1})$  is defined as follows:

$P_0$   
 $\dots$   
 $P_{n-1}$   
 $Q$

We show  $R^{(n)}(\vec{\mathbf{x}}) \simeq (f \circ (g_0(\vec{\mathbf{x}}), \dots, g_{n-1}(\vec{\mathbf{x}})))$ .

- Case 1: For one  $i$   $g_i(\vec{\mathbf{x}}) \uparrow$ . The program will loop in program  $P_i$  for the first such  $i$ , therefore  $R^{(n)}(\vec{\mathbf{x}}) \uparrow$ . Further,  $(f \circ (g_0, \dots, g_{n-1}))(\vec{\mathbf{x}}) \uparrow$ , the program computes correctly.
- Case 2:  $g_i(\vec{\mathbf{x}}) \downarrow$  for all  $i$ . The program will execute  $P_i$ , and execute  $\mathbf{y}_i \simeq g_i(\mathbf{x}_0, \dots, \mathbf{x}_{k-1})$  and reaches the beginning of program  $Q$ .
  - Case 2.1:  $f(g_0(\vec{\mathbf{x}}), \dots, g_{n-1}(\vec{\mathbf{x}})) \uparrow$ . The program  $Q$  will loop, and we have  $R^{(n)}(\vec{\mathbf{x}}) \uparrow$  and  $(f \circ (g_0, \dots, g_{n-1}))(\vec{\mathbf{x}}) \uparrow$ .
  - Case 2.2: Otherwise, the program will reach the end of program  $Q$  and result in  $\mathbf{x}_0 \simeq f(g_0(\vec{\mathbf{x}}), \dots, g_{n-1}(\vec{\mathbf{x}}))$ , so  $R^{(n)}(\vec{\mathbf{x}}) \simeq (f \circ (g_0, \dots, g_{n-1}))(\vec{\mathbf{x}})$ .

In all cases, we have  $R^{(n)}(\vec{\mathbf{x}}) \simeq (f \circ (g_0, \dots, g_{n-1}))(\vec{\mathbf{x}})$ .

**Proof of (c)**

Assume  $g : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$ , and  $h : \mathbb{N}^{n+2} \xrightarrow{\sim} \mathbb{N}$  are URM-computable. Let  $f := \text{primrec}(g, h)$ . Show  $f$  is URM-computable.

Note that the defining equations for  $f$  are as follows (let  $\vec{n} := n_0, \dots, n_{n-1}$ ):

- $f(\vec{n}, 0) \simeq g(\vec{n})$ ,
- $f(\vec{n}, k + 1) \simeq h(\vec{n}, k, f(\vec{n}, k))$ .

So in order to compute  $f(\vec{n}, l)$  for  $l > 0$  we have to make the following computations:

- Compute  $f(\vec{n}, 0)$  as  $g(\vec{n})$ .
- Compute  $f(\vec{n}, 1)$  as  $h(\vec{n}, 0, f(\vec{n}, 0))$ , using the previous result.
- Compute  $f(\vec{n}, 2)$  as  $h(\vec{n}, 1, f(\vec{n}, 1))$ , using the previous result.
- ...
- Compute  $f(\vec{n}, l)$  as  $h(\vec{n}, l - 1, f(\vec{n}, l - 1))$ , using the previous result.

A plan for the program is as follows:

- Let  $\vec{\mathbf{x}} := \mathbf{x}_0, \dots, \mathbf{x}_{n-1}$ . Let  $\mathbf{y}, \mathbf{z}, \mathbf{u}$  be new registers.
- We will compute  $f(\vec{\mathbf{x}}, \mathbf{y})$  for  $\mathbf{y} = 0, 1, 2, \dots, \mathbf{x}_n$ , and store the result in  $\mathbf{z}$ .
  - Initially we start with  $\mathbf{y} = 0$  (which is the case since all registers except of  $\mathbf{x}_i$  for  $i = 0, \dots, n$  contain initially 0), and need to compute  $\mathbf{z} \simeq f(\vec{\mathbf{x}}, 0)$ . This is achieved by computing  $\mathbf{z} \simeq g(\vec{\mathbf{x}})$ . Then  $\mathbf{y} = 0$  and  $\mathbf{z} \simeq f(\vec{\mathbf{x}}, \mathbf{y})$ .
  - In the step from  $\mathbf{y}$  to  $\mathbf{y} + 1$  we assume that when entering one iteration of the loop we have initially  $\mathbf{z} \simeq f(\vec{\mathbf{x}}, \mathbf{y})$ . We want to achieve that after increasing of  $\mathbf{y}$  by 1 we still have  $\mathbf{z} \simeq f(\vec{\mathbf{x}}, \mathbf{y})$ . Then the loop-invariant  $\mathbf{z} \simeq f(\vec{\mathbf{x}}, \mathbf{y})$  is preserved. This is obtained as follows:
    - \* We first compute  $\mathbf{u} \simeq h(\vec{\mathbf{x}}, \mathbf{y}, \mathbf{z}) (\simeq h(\vec{\mathbf{x}}, \mathbf{y}, f(\vec{\mathbf{x}}, \mathbf{y})) \simeq f(\vec{\mathbf{x}}, \mathbf{y} + 1))$ .

- \* Then execute  $z := u$  ( $\simeq f(\vec{x}, y + 1)$ ).
- \* Finally execute  $y := y + 1$ .
- \* Then we have  $z \simeq f(\vec{x}, y)$  for the new value of  $y$ .
- This loop is iterated as long as  $y$  hasn't reached  $x_n$ .
- Once  $y$  has reached  $x_n$ ,  $z$  contains  $f(\vec{x}, y) \simeq f(\vec{x}, x_n)$ .
- Execute  $x_0 := z$ .

Let therefore

- $P$  be a URM program, which computes  $z \simeq g(\vec{x})$ , and avoids  $y$ .
- $Q$  be a program, which computes  $u \simeq h(\vec{x}, y, z)$ .

The following program  $R$  computes  $f$  (everything following  $\%$  is a comment):

```

P                                % Compute  $u \simeq g(\vec{x}, y, z)$ 
while  $x_n \neq y$  do {
  Q                                % Compute  $u \simeq h(\vec{x}, y, z)$ 
                                % will be  $\simeq h(\vec{x}, y, f(\vec{x}, y)) \simeq f(\vec{x}, y + 1)$ 
  z := u;
  y := y + 1;
}
x_0 := z;
```

**Correctness of this program:** When  $P$  is terminated, we have  $y = 0$  and  $z \simeq g(\vec{x}) \simeq f(\vec{x}, y)$ . After each iteration of the while loop, we have  $y := y' + 1$  and  $z \simeq h(\vec{x}, y', z')$ , where  $y', z'$  are the previous values of  $y, z$ , respectively. This amounts to having  $z \simeq f(\vec{x}, y)$ . The loop terminates, when  $y$  has reached  $x_n$ , and then  $z$  contains  $f(\vec{x}, y)$ , which is then stored in  $x_0$ .

If  $P$  loops for ever, or in one of the iterations  $Q$  loops for ever, then  $R$  loops as well,  $R^{(n+1)}(\vec{x}, x_n) \uparrow$ . But then we have as well  $f(\vec{x}, k) \uparrow$  for some  $k \leq x_n$ , and therefore  $f(\vec{x}, l)$  is undefined for all  $l > k$  ( $f(\vec{x}, k + 1) \simeq h(\vec{x}, k, f(\vec{x}, k)) \uparrow$ ,  $f(\vec{x}, k + 2) \simeq h(\vec{x}, k + 1, f(\vec{x}, k + 1)) \uparrow$ , etc.) Especially,  $f(\vec{x}, x_n) \uparrow$ . Therefore  $f(\vec{x}, x_n) \simeq R^{(n+1)}(\vec{x}, x_n)$  (since both sides are undefined).

### Proof of (d)

Assume  $g : \mathbb{N}^{n+1} \xrightarrow{\sim} \mathbb{N}$  is URM-computable. Show  $\mu(g)$  is URM-computable as well.

Note  $\mu(g)(x_0, \dots, x_{k-1})$  is the minimal  $z$  s.t.  $g(x_0, \dots, x_{k-1}, z) \simeq 0$ . Let  $\vec{x} := x_0, \dots, x_{k-1}$  and let  $y, z$  be registers different from  $\vec{x}$ .

Plan for the program:

- Compute  $g(\vec{x}, 0), g(\vec{x}, 1), \dots$  until we find a  $k$  s.t.  $g(\vec{x}, k) \simeq 0$ . Then return  $k$ .
- This is carried out by executing  $z \simeq g(\vec{x}, y)$  and successively increasing  $y$  by 1 until we have  $z = 0$ .
- Since we haven't introduced a repeat-until-loop, we replace this by a while loop as follows:
  - Initially we set  $z$  to something not 0.
  - As long as  $z \neq 0$ , compute  $z \simeq g(\vec{x}, y)$  and then increase  $y$  by 1.
  - If the while-loop terminates, we have  $y = n + 1$  for the minimal  $n$  s.t.  $g(\vec{x}, n) \simeq 0$ .
  - Decrease  $y$  once, and store result in  $x_0$ .

Let  $P$  be a program which computes  $z \simeq g(x_0, \dots, x_{k-1}, y)$ . Then the following program  $R$  computes  $\mu(g)$  (note that initially  $y = 0, z = 0$ ).

```

z := z + 1;
while z ≠ 0 do {
  P
  y := y + 1; };
y := y - 1;
x0 := y;

```

By setting  $z$  initially to 1, we guarantee that the while-loop is executed at least once. Further initially  $y = 0$ . After each iteration of the while loop, we have  $y := y' + 1$  and  $z \simeq g(x_0, \dots, x_{k-1}, y')$ , where  $y'$  is the value of  $y$  before starting this iteration. If the loop terminates, then we have therefore  $z \simeq 0$  and  $y = y' + 1$  for the first value, such that  $g(x_0, \dots, x_{k-1}, y') \simeq 0$ . At the end  $x_0$  is set to that value. If ever  $P$  runs into a loop, then for some  $k$  we have  $g(\vec{x}, k) \uparrow$  and for all  $l < k$   $g(\vec{x}, l) \neq 0$ , and therefore  $\mu(g)(\vec{x}) \uparrow$ , and  $R^{(n)}(\vec{x}) \uparrow$ . If  $P$  always terminates, but the while-loop doesn't terminate, then there is no  $k$  s.t.  $g(\vec{x}, k) \simeq 0$ , again  $\mu(g)(\vec{x}) \uparrow$  and  $R^{(n)}(\vec{x}) \uparrow$ .

### 3.4 The Undecidability of the Halting Problem

The undecidability of the Halting Problem was first proved 1936 by Alan Turing in this paper “*On computable numbers, with an application to the Entscheidungsproblem*”. We recast his proof using URMs instead of Turing machines.

In the following, by “computable” means “URM-computable”. This will be justified later when we will argue that URM-computability does coincide with the intuitive notion of computability (the Church-Turing-thesis).

**Definition 3.4** (a) A problem is an  $n$ -ary predicate  $M(x_0, \dots, x_{n-1})$  of natural numbers, i.e. a property of  $n$ -tuples of natural numbers.

(b) A problem  $M$  is decidable, if the characteristic function of  $M$  defined by

$$\chi_M(x_0, \dots, x_{n-1}) := \begin{cases} 1 & \text{if } M(x_0, \dots, x_{n-1}) \text{ holds,} \\ 0 & \text{otherwise} \end{cases}$$

is computable.

**Example** the binary predicate

$$\text{Multiple}(x, y) :\Leftrightarrow x \text{ is a multiple of } y$$

is a problem.

$\chi_M(x_0, \dots, x_{n-1})$  decides, whether  $M(x_0, \dots, x_{n-1})$  holds (then it returns 1 for yes), or not (then it returns 0 for no). For instance,

$$\chi_{\text{Multiple}}(x, y) = \begin{cases} 1 & \text{if } x \text{ is a multiple of } y, \\ 0 & \text{if } x \text{ is not a multiple of } y. \end{cases}$$

This function is intuitively computable (and one can show in fact that it is URM-computable), therefore **Multiple** is decidable.

URM-programs can be written as a string of ASCII-symbols, which is an element of  $A^*$ , where  $A$  is the set of ASCII-symbols, and which can be encoded as a natural number. Let for a URM program  $P$ ,  $\text{code}(P)$  be the number encoding  $P$ . It is intuitively decidable, whether a string of ASCII symbols is a URM-program, and therefore it is as well intuitively decidable, whether  $n = \text{code}(P)$  for a URM-program  $P$ . With some effort one can show that this property can be decided by a URM-program. However, the following problem is undecidable:

**Definition 3.5** The Halting Problem is the following binary predicate:

$$\text{Halt}(x, y) :\Leftrightarrow \begin{cases} 1 & \text{if } x = \text{code}(P) \text{ for a URM program } P \text{ and } P^{(1)}(y) \downarrow \\ 0 & \text{otherwise} \end{cases}$$

**Example:** Let  $P$  be a code for the URM program `ifzero(0,0)`

If the input is  $> 0$ , the program terminates immediately, and  $R_0$  remains unchanged, so  $P^{(0)}(k) \simeq k$  for  $k > 0$ .

If the input is  $= 0$ , the program loops for ever. Therefore,  $P^{(0)}(0) \uparrow$ .

Let  $e = \text{code}(P)$ . Then  $\text{Halt}(e, n)$  holds for  $n > 0$  and does not hold for  $n = 0$ .

**Remark:** We will see below that  $\text{Halt}$  is undecidable. However, the following function is computable:

$$\text{WeakHalt}(x, y) :\simeq \begin{cases} 1 & \text{if } x = \text{code}(P) \text{ for a URM program } P \text{ and } P^{(1)}(y) \downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

A program for computing  $\text{WeakHalt}(x, y)$  can be defined as follows: One first checks whether  $x$  encodes a valid URM program. If this is not the case, the program enters an infinite loop. Otherwise, one simulates the corresponding URM program  $P$  with input  $y$ . This can be done – it is an programming exercise to write a program which simulates a URM, and therefore this simulation is intuitively computable. It is not too complicated to show that there exists in fact a URM program for carrying out the simulation. If the simulation stops, we output 1, otherwise the program loops for ever.

**Theorem 3.6** *The halting problem is undecidable.*

**Proof:**

Assume there exist a URM  $P$  s.t.  $P^{(2)}$  decides the Halting Problem. Therefore we have

$$P^{(2)}(x, y) \simeq \begin{cases} 1 & \text{if } x \text{ codes a URM program } Q \text{ s.t. } Q^{(1)}(y) \downarrow \\ 0 & \text{otherwise.} \end{cases}$$

We argue now similar to the proof that  $\mathbb{N} \not\cong \mathcal{P}(\mathbb{N})$ : We define a computable function  $f : \mathbb{N} \rightarrow \mathbb{N}$ , which cannot be computed by the URM with code  $e$  – this will be violated on input  $e$ :

- If  $P^{(2)}(e, e) \simeq 1$ , i.e.  $e$  encodes a URM  $Q$  and  $Q^{(1)}(e) \downarrow$ , then we let  $f(e) \uparrow$ . Therefore,  $Q^{(1)} \neq f$ .
- If  $P^{(2)}(e, e) \simeq 0$ , i.e.  $e$  doesn't encode a URM, or it encodes a URM  $Q$  and  $Q^{(1)}(e) \uparrow$ , then we let  $f(e) \downarrow$  and define  $f(e) \simeq 0$ . (We could have defined  $f(e) \simeq n$  for any other natural number  $n$ , it only matters that  $f(e)$  is a defined.) Therefore, if  $e$  encodes a URM  $Q$ , we have  $Q^{(1)} \neq f$ .

The complete definition is therefore as follows:

$$f(e) \simeq \begin{cases} 0 & \text{if } P^{(2)}(e, e) \simeq 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$f$  is not URM-computable, for  $f = R^{(1)}$  is violated by  $f(\text{code}(R)) \neq R^{(1)}(\text{code}(R))$ . Assume  $f$  were computed by  $R$ , i.e.  $f = R^{(R)}$ . Then:

$$\begin{array}{lcl} R^{(1)}(\text{code}(R)) \downarrow & \begin{array}{c} \text{property of } P \\ \Leftrightarrow \\ \text{Def of } f \\ \Leftrightarrow \\ f=R^{(1)} \\ \Leftrightarrow \end{array} & P^{(2)}(\text{code}(R), \text{code}(R)) \simeq 1 \\ & & f(\text{code}(R)) \uparrow \\ & & R^{(1)}(\text{code}(R)) \uparrow , \end{array}$$

a contradiction. However,  $f$  can be intuitively computed (the program makes use of program  $P$ ), and it can easily be shown that it is URM-computable. So we get a contradiction, and the assumption that  $P^{(2)}$  decides the Halting Problem is wrong. Hence, the Halting Problem is undecidable.

**Remark:** The above proof can easily be adapted to any reasonable programming language, in which one can define all computable functions. Such programming languages are called Turing-complete languages. For instance Babbage's machine was, if one removes the restriction to finite memory, Turing-complete, since it had a conditional jump.

Applied to standard programming language, which are Turing complete, the unsolvability of the Turing-halting problem means: it is not possible to write a program, which checks, whether a program on given input terminates.

## 4 Turing Machines

### 4.1 Motivation

URMs is a model of computation which is very easy to understand. The model of URMs has however two drawbacks:

- (1) The execution of a single URM instruction, e.g.  $\text{succ}(n)$ , might take arbitrarily long:

For instance, if  $R_n$  contains the binary number  $\underbrace{111 \cdots 111}_{k \text{ times}}$ , we have to replace it by the binary number  $1 \underbrace{000 \cdots 000}_{k \text{ times}}$ , i.e. we have to replace  $k$  ones by zeros. Since  $k$  is arbitrary, this single step might take arbitrarily long time.

Therefore URMs are unsuitable as a basis for defining the complexity of algorithms. Of course, there exist meta theorems which relate the complexity of URM programs to the actual URM programs. But for defining the complexity, a different notion is needed, and the Turing machine is the currently most widely accepted model of computation used for this purpose.

- (2) We are aiming at a notion of computability, which covers all possible ways of computing something, independently of any concrete machine. URMs are a model of computation which covers computers, as they are currently used. However, there might be completely different notions of computability, based on symbolic manipulations of a sequence of characters, where it might be more complicated to see directly that all such computations can be simulated by a URM. It is more easy to see that such notions are covered by the Turing machine model of computation.

We will therefore introduce a second model of computation, the Turing machine (TM). We will later show that the sets of Turing-computable and of URM-computable functions coincide. The definition of computability proposed by Turing in 1936 is based on an analysis of how a human being (called agent) carries out a computation on a piece of paper.

|           |
|-----------|
| 15 . 16 = |
| 15        |
| 90        |
| 240       |

In order to formulate this, we will make the following steps:

- An algorithm should be deterministic, and therefore we assume that the agent uses only finitely many symbols, which he puts at discrete positions on the paper.

|  |  |   |   |   |   |   |   |  |   |   |   |
|--|--|---|---|---|---|---|---|--|---|---|---|
|  |  | 1 | 5 | . | 1 | 6 | = |  |   |   |   |
|  |  |   |   |   |   |   |   |  | 1 | 5 |   |
|  |  |   |   |   |   |   |   |  |   | 9 | 0 |
|  |  |   |   |   |   |   |   |  | - | - | - |
|  |  |   |   |   |   |   |   |  | 2 | 4 | 0 |
|  |  |   |   |   |   |   |   |  |   |   |   |
|  |  |   |   |   |   |   |   |  |   |   |   |

**Sideremark:** If one has doubts, whether this is always possible (we might squeeze a small symbol between two existing ones, which might violate having a grid of symbols), one can









- **Case 1:** *The TM stops.*  
 At any time, only finitely many cells contain a non-blank symbol: initially this is the case, and in finitely many steps, only finitely many cell contents are changed.  
 Therefore there cannot be an infinite sequence of symbols in  $\{0, 1\}$  starting from the head position to the right, and let the  $k$ th cell be the first one not containing 0 or 1. Therefore the TM contains, starting from the head position, symbols  $b_0b_1 \cdots b_{k-1}c$ , where  $b_i \in \{0, 1\}$  and  $c \notin \{0, 1\}$ .  $k$  might be 0 (if the symbol at the position of the head itself is  $\neq 0, 1$ ).  
 Let  $a = (b_0, \dots, b_{k-1})_2$  (in case  $k = 0$ ,  $a = 0$ ). Then  $M^{(k)}(a_0, \dots, a_{k-1}) \simeq a$ .
- **Case 2:** *Otherwise.* Then  $M^{(k)}(a_0, \dots, a_{k-1}) \uparrow$ .

**Example:** Let  $\Sigma = \{0, 1, a, b, \sqcup\}$ .

- If tape contains at the end, starting with the head position,  $01010\sqcup_0101\sqcup$  or  $01010a\sqcup$ , the output is  $(01010)_2 = 10$ .
- If tape contains at the end, starting with head position,  $ab\sqcup$ ,  $a$  or  $\sqcup$ , the output is 0.

**Definition 4.2**  $f : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$  is Turing-computable, in short TM-computable, if  $f = M^{(k)}$  for some TM  $M$ , the alphabet of which contains  $\{0, 1\}$ .

**Example:** The above example of a TM treated in detail shows that  $\text{succ} : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$  is Turing-computable.

**Theorem 4.3** If  $f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$  is URM-computable, then it is as well Turing-computable by a TM with alphabet  $\{0, 1, \sqcup\}$ .

**Proof:**

We first introduce the following notation:

By saying the tape of a TM contains  $a_0, \dots, a_l$  we mean that the tape contains, starting with the head position,  $a_0, \dots, a_l$ , and all other cells of the tape contain  $\sqcup$ .

Furthermore, in this proof,  $\text{bin}(n)$  will stand for any binary representation of  $n$ , for instance  $\text{bin}(2)$  could be any of 10, 010, 0010 etc. This is needed, since when performing operations, we don't want to bother with normalising the representation of numbers on the tape, i.e. we don't want to deal with having to delete possible leading zeros.

Assume  $f$  is URM-computable by URM  $P$ , i.e.  $f = P^{(n)}$ . Assume  $P$  refers only to registers  $R_0, \dots, R_{l-1}$  and that the input registers (i.e.  $R_0, \dots, R_{n-1}$ ) of  $P$  are among  $R_0, \dots, R_{l-1}$ , i.e. that  $l > n$

We will define a Turing machine  $M$ , which simulates  $P$ . This will be done as follows:

- If the registers  $R_0, \dots, R_{l-1}$  of the URM  $P$  contain  $a_0, \dots, a_{l-1}$ , this is modelled in the TM as the tape containing  $\text{bin}(a_0)\sqcup \cdots \sqcup \text{bin}(a_{l-1})$ .
- An instruction  $I_j$  will be simulated by finitely many states  $q_{j,0}, \dots, q_{j,i}$  of the TM with instructions for those states.

The instructions and states will be defined in such a way that the following holds:

- Assume for the URM  $P$ 
  - $R_0, \dots, R_{l-1}$  contain  $a_0, \dots, a_{l-1}$ ,
  - the URM is about to execute  $I_j$ .
- Assume that after executing  $I_j$  the URM  $P$  arrives at a state where

- $R_0, \dots, R_{l-1}$  contain  $b_0, \dots, b_{l-1}$ ,
- the PC contains  $j'$ .
- Then, if the configuration of the TM  $P$  is s.t.
  - the tape contains  $\text{bin}(a_0)\sqcup\text{bin}(a_1)\sqcup\cdots\sqcup\text{bin}(a_{l-1})$ ,
  - and the state of is  $q_{j,0}$ ,
- then the TM will, after executing the corresponding instructions, arrive at a configuration, in which
  - the tape contains  $\text{bin}(b_0)\sqcup\text{bin}(b_1)\sqcup\cdots\sqcup\text{bin}(b_{l-1})$ ,
  - the state is  $q_{j',0}$ .

For instance, if we simulate the instruction  $I_4 = \text{pred}(2)$  then we have the following:

- Assume the URM is about to execute instruction  $I_4$  with register contents

|       |       |       |
|-------|-------|-------|
| $R_0$ | $R_1$ | $R_2$ |
| 2     | 1     | 3     |

So PC is initially 4

- Then the URM will end with the PC containing 5 and register contents

|       |       |       |
|-------|-------|-------|
| $R_0$ | $R_1$ | $R_2$ |
| 2     | 1     | 2     |

So, when simulating this in the TM we want the following:

- If the TM is in state  $q_{4,0}$ , with the tape containing  $\text{bin}(2)\sqcup\text{bin}(1)\sqcup\text{bin}(3)$
- it should reach state  $q_{5,0}$  with the tape containing  $\text{bin}(2)\sqcup\text{bin}(1)\sqcup\text{bin}(2)$

Furthermore, we need an initialisation for the TM consisting of states  $q_{\text{init},0}, \dots, q_{\text{init},j}$  and corresponding instructions, s.t.

- if the TM initially contains the configuration corresponding to arguments  $b_0, \dots, b_{n-1}$  of the function to be defined, namely  $\text{bin}(b_0)\sqcup\text{bin}(b_1)\sqcup\cdots\sqcup\text{bin}(b_{n-1})$ ,
- it will reach state  $q_{0,0}$  with the tape containing  $\text{bin}(b_0)\sqcup\text{bin}(b_1)\sqcup\cdots\sqcup\text{bin}(b_{n-1})\sqcup\underbrace{0\sqcup 0\sqcup\cdots\sqcup 0}_{l-n \text{ times}}$ .

**Remark:** We assume here that 0 is represented on the tape by 0, enclosed by blanks separating the binary numbers of the registers. We could as represent it as the empty string, which means that we have the two enclosing blanks with no symbol in between. Then the initial configuration of the TM would represent already  $\text{bin}(b_0)\sqcup\text{bin}(b_1)\sqcup\cdots\sqcup\text{bin}(b_{n-1})\sqcup\underbrace{\text{bin}(0)\sqcup\text{bin}(0)\sqcup\cdots\sqcup\text{bin}(0)}_{l-n \text{ times}}$ , since

$\underbrace{\text{bin}(0)\sqcup\text{bin}(0)\sqcup\cdots\sqcup\text{bin}(0)}_{l-n \text{ times}}$  is just  $\underbrace{\sqcup\sqcup\cdots\sqcup}_{l-n \text{ times}}$ . No initialisation would be needed.

If we have achieved the above simulation steps, then a computation of  $P^{(n)}(a_0, \dots, a_{n-1})$  is simulated as follows: Assume the run of the URM, starting with  $R_i$  containing  $a_{0,i} = a_i$   $i = 0, \dots, n-1$ ,

and  $a_{0,i} = 0$  for  $i = n, \dots, l - 1$  is as follows:

|             |           |           |         |             |           |         |             |
|-------------|-----------|-----------|---------|-------------|-----------|---------|-------------|
| Instruction | $R_0$     | $R_1$     | $\dots$ | $R_{n-1}$   | $R_n$     | $\dots$ | $R_{l-1}$   |
| $I_0$       | $a_0$     | $a_1$     | $\dots$ | $a_{n-1}$   | $0$       | $\dots$ | $0$         |
| =           | =         | =         |         | =           | =         |         | =           |
| $I_{k_0}$   | $a_{0,0}$ | $a_{0,1}$ | $\dots$ | $a_{0,n-1}$ | $a_{0,n}$ | $\dots$ | $a_{0,l-1}$ |
| $I_{k_1}$   | $a_{1,0}$ | $a_{1,1}$ | $\dots$ | $a_{1,n-1}$ | $a_{1,n}$ | $\dots$ | $a_{1,l-1}$ |
| $I_{k_2}$   | $a_{2,0}$ | $a_{2,1}$ | $\dots$ | $a_{2,n-1}$ | $a_{2,n}$ | $\dots$ | $a_{2,l-1}$ |
|             |           | $\dots$   |         |             |           |         |             |

Then the corresponding TM will successively reach the following configurations:

|              |                                                                                                                                          |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------|
| State        | Tape contains                                                                                                                            |
| $q_{init,0}$ | $\text{bin}(a_0)\_ \_ \text{bin}(a_1)\_ \_ \dots \_ \_ \text{bin}(a_{n-1})\_ \_ \_$                                                      |
| $q_{k_0,0}$  | $\text{bin}(a_0)\_ \_ \text{bin}(a_1)\_ \_ \dots \_ \_ \text{bin}(a_{n-1})\_ \_ \_ \text{bin}(0)\_ \_ \dots \_ \_ \text{bin}(0)\_ \_ \_$ |
| =            | =                                                                                                                                        |
| $q_{0,0}$    | $\text{bin}(a_{0,0})\_ \_ \text{bin}(a_{0,1})\_ \_ \dots \_ \_ \text{bin}(a_{0,l-1})\_ \_ \_$                                            |
| $q_{k_1,0}$  | $\text{bin}(a_{1,0})\_ \_ \text{bin}(a_{1,1})\_ \_ \dots \_ \_ \text{bin}(a_{1,l-1})\_ \_ \_$                                            |
| $q_{k_2,0}$  | $\text{bin}(a_{2,0})\_ \_ \text{bin}(a_{2,1})\_ \_ \dots \_ \_ \text{bin}(a_{2,l-1})\_ \_ \_$                                            |
|              | $\dots$                                                                                                                                  |

**Example:**

We take the URM program  $P$

$I_0 = \text{ifzero}(0, 3)$

$I_1 = \text{pred}(0)$

$I_2 = \text{ifzero}(1, 0)$

The corresponding unary function  $P^{(1)}$  operates as follows:  $R_1$  is initially zero and never changed, therefore  $R_1$  is always 0 and the jump in  $I_1$  is always executed. Therefore a run of the program is as follows: It checks whether  $R_0$  contains 0. If yes, it will stop. Otherwise, it will reduce  $R_0$  by 1, and then jump back to the beginning. So the program always terminates with  $R_0$  containing 0, and we have  $P^{(1)}(n) \simeq 0$ .

A run of this program corresponding to a computation of  $P^{(1)}(2)$  is as follows:

|             |       |       |
|-------------|-------|-------|
| Instruction | $R_0$ | $R_1$ |
| $I_0$       | 2     | 0     |
| $I_1$       | 2     | 0     |
| $I_2$       | 1     | 0     |
| $I_0$       | 1     | 0     |
| $I_1$       | 1     | 0     |
| $I_2$       | 0     | 0     |
| $I_0$       | 0     | 0     |
| $I_3$       | 0     | 0     |
| URM Stops   |       |       |

We start with instruction  $I_0$  and  $R_0$  containing 2 and  $R_1$  containing 0. Jump  $I_0$  is not executed, so we arrive at  $I_1$  with register values unchanged. Then  $R_0$  is reduced by 1 and we move to  $I_2$ . Then we jump back to  $I_0$  with the register values unchanged. Etc.

The corresponding Turing machine  $M$  should simulate this program. A run of  $M^1(2)$  will start in state  $q_{init,0}$  with initial configuration  $\text{bin}(2)\_ \_$ . Then in the initialisation part, the TM expands this to  $\text{bin}(2)\_ \_ \text{bin}(0)\_ \_$  and arrives at state  $q_{0,0}$ . Then each of the steps of the URM is simulated in the TM. So, when simulating  $I_0$ , the TM checks whether the first binary number on the tape is 0 or not. If it is 0, it switches to  $q_{3,0}$ , otherwise, it switches to  $q_{1,0}$ . When simulating  $I_1$ , it decreases the first binary number on the tape by one. When simulating  $I_2$ , it checks, whether the second binary number on the tape is 0 or not. If it is zero, it switches to  $q_{0,0}$ , otherwise to  $q_{3,0}$ .

Assuming that we have introduced such a TM, we can write the configuration of the URM and of the TM in one table and obtain the following:

| Instruction    | R <sub>0</sub> | R <sub>1</sub> | State of TM         | Content of Tape                                           |
|----------------|----------------|----------------|---------------------|-----------------------------------------------------------|
|                |                |                | $q_{\text{init},0}$ | $\text{bin}(2) \sqcup \sqcup$                             |
| I <sub>0</sub> | 2              | 0              | $q_{0,0}$           | $\text{bin}(2) \sqcup \sqcup \text{bin}(0) \sqcup \sqcup$ |
| I <sub>1</sub> | 2              | 0              | $q_{1,0}$           | $\text{bin}(2) \sqcup \sqcup \text{bin}(0) \sqcup \sqcup$ |
| I <sub>2</sub> | 1              | 0              | $q_{2,0}$           | $\text{bin}(1) \sqcup \sqcup \text{bin}(0) \sqcup \sqcup$ |
| I <sub>0</sub> | 1              | 0              | $q_{0,0}$           | $\text{bin}(1) \sqcup \sqcup \text{bin}(0) \sqcup \sqcup$ |
| I <sub>1</sub> | 1              | 0              | $q_{1,0}$           | $\text{bin}(1) \sqcup \sqcup \text{bin}(0) \sqcup \sqcup$ |
| I <sub>2</sub> | 0              | 0              | $q_{2,0}$           | $\text{bin}(0) \sqcup \sqcup \text{bin}(0) \sqcup \sqcup$ |
| I <sub>0</sub> | 0              | 0              | $q_{0,0}$           | $\text{bin}(0) \sqcup \sqcup \text{bin}(0) \sqcup \sqcup$ |
| I <sub>3</sub> | 0              | 0              | $q_{3,0}$           | $\text{bin}(0) \sqcup \sqcup \text{bin}(0) \sqcup \sqcup$ |
| URM Stops      |                |                |                     | TM Stops                                                  |

Once we have introduced such a simulation we can see that  $P^{(n)} = M^{(n)}$ :

- If  $P^n(a_0, \dots, a_{n-1}) \downarrow$ ,  $P^n(a_0, \dots, a_{n-1}) \simeq j$ , then  $P$  will eventually stop with  $R_i$  containing some values  $b_i$ , where  $b_0 = j$ .  
Then, the TM  $M$  starting with  $\text{bin}(a_0) \sqcup \dots \sqcup \text{bin}(a_{n-1})$  will eventually terminate in a configuration  $\text{bin}(b_0) \sqcup \dots \sqcup \text{bin}(b_{l-1})$  and therefore  $M^n(a_0, \dots, a_{n-1}) \simeq b_0 = j$ .
- If  $P^n(a_0, \dots, a_{n-1}) \uparrow$ , the URM  $P$  will loop and the TM  $M$  will carry out the same steps as the URM and loop as well, therefore  $M^n(a_0, \dots, a_{n-1}) \uparrow$ , again  $P^n(a_0, \dots, a_{n-1}) \simeq M^n(a_0, \dots, a_{n-1})$ .

Therefore, we have  $P^{(n)} = M^{(n)}$  and we are done, provided we have defined the simulation. We describe informally, how the instructions of the URM are simulated and the initialisation is obtained.

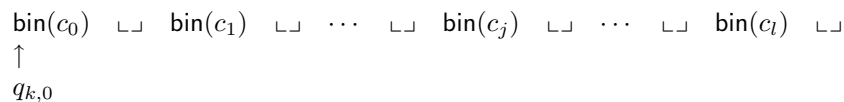
- **Initialisation.** We start with the tape containing  $\text{bin}(a_0) \sqcup \dots \sqcup \text{bin}(a_{n-1})$ , and have to extend this to  $\text{bin}(a_0) \sqcup \dots \sqcup \text{bin}(a_{n-1}) \sqcup \underbrace{\text{bin}(0) \sqcup \dots \sqcup \text{bin}(0)}_{l-n \text{ times}}$ .

This is achieved by moving the head to the end of the initial position (by moving to the  $n$ th blank to the right), and then inserting, starting from the next blank,  $l - n$ -times  $0 \sqcup$ , and then moving back to the beginning (by moving to the  $l$ th blank to the left).

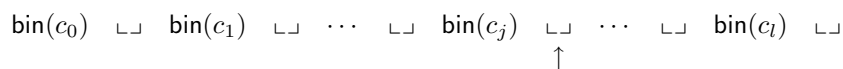
• **Simulation of URM instructions.**

- **Simulation of instruction  $I_k = \text{succ}(j)$ .** We have to increase the  $(j + 1)$ st binary number on the tape by 1 (note that register number 0 is the first number on the tape, register number 1 is the second number on the tape, etc.).

Initially, the configuration is:



- \* We first move to the  $(j + 1)$ st blank to the right. Then we are at the end of the  $(j + 1)$ st binary number.



\* Now perform the operation for increasing by 1 as above.

At the end we obtain:

$$\text{bin}(c_0) \sqcup \text{bin}(c_1) \sqcup \cdots \sqcup \text{bin}(c_j + 1) \sqcup \cdots \sqcup \text{bin}(c_l) \sqcup$$

↑

However, it might be that we needed to write over the separating blank a 1, in which case we have:

$$\text{bin}(c_0) \sqcup \text{bin}(c_1) \sqcup \cdots \text{bin}(c_{j-1}) \text{bin}(c_j + 1) \sqcup \cdots \sqcup \text{bin}(c_l) \sqcup$$

↑

\* If we are in the latter case, then we have to shift all symbols to the left once left, in order to obtain a separating  $\sqcup$  between the  $l$ th and  $l - 1$ st entry. This can be achieved easily (until one has reached the  $l - 1$ st blank) and we obtain

$$\text{bin}(c_0) \sqcup \text{bin}(c_1) \sqcup \cdots \text{bin}(c_{j-1}) \sqcup \text{bin}(c_j + 1) \sqcup \cdots \sqcup \text{bin}(c_l) \sqcup$$

↑

\* Otherwise, we move the head to the left, until we reached the  $(j + 1)$ st blank to the left, and then move it once to the right. We obtain

$$\text{bin}(c_0) \sqcup \text{bin}(c_1) \sqcup \cdots \sqcup \text{bin}(c_j + 1) \sqcup \cdots \sqcup \text{bin}(c_l) \sqcup$$

↑

– **Simulation of instruction**  $I_k = \text{pred}(j)$ . We have to decrease the  $(j + 1)$ st binary number on the tape by 1.

\* Assume the configuration at the beginning is :

$$\text{bin}(c_0) \sqcup \text{bin}(c_1) \sqcup \cdots \text{bin}(c_j) \sqcup \cdots \sqcup \text{bin}(c_l) \sqcup$$

↑

We want to decrease the  $j$ th number by 1 (or leave it as it is, if it is zero).

So we want to achieve the following configuration:

$$\text{bin}(c_0) \sqcup \text{bin}(c_1) \sqcup \cdots \text{bin}(c_j - 1) \sqcup \cdots \sqcup \text{bin}(c_l) \sqcup$$

↑

Done as follows:

\* We move as before to the end of the  $(j + 1)$ st number.

\* Then we check, if the number consists only of zeros or not.

· If it consists only of zeros, i.e. it represents 0, then  $\text{pred}(j)$  doesn't change anything.

· Otherwise, the number is of the form  $b_0 \cdots b_k 1 \underbrace{00 \cdots 0}_{l' \text{ times}}$ .

$$\text{We have } (b_0 \cdots b_k \underbrace{100 \cdots 0}_{l' \text{ times}})_2 - 1 = (b_0 \cdots b_k 0 \underbrace{11 \cdots 1}_{l' \text{ times}})_2.$$

So, we have to replace the binary string by  $b_0 \cdots b_k 0 \underbrace{11 \cdots 1}_{l' \text{ times}}$ . This can be done

similarly as for the successor operation.

\* Finally we move back to the beginning.

– **Simulation of instruction**  $I_k = \text{ifzero}(j, k')$ .

The URM instruction does the following: if  $R_j$  contains zero, then the next instruction is  $I_{k'}$  (or the program stops if there is no such instruction). Otherwise the next instruction is  $I_{k+1}$  (or the program stops if there is no such instruction).

This is simulated as follows on the TM: It moves to the  $(j + 1)$ st binary number on the tape, checks whether it consists only of zeros. If yes, we switch to state  $q_{k',0}$ , otherwise we switch to state  $q_{k+1,0}$ .

This completes the simulation of the URM  $P$ .

**Remark:** We will later show the other direction, namely that every TM-computable program is URM-computable. Therefore both models of computation define the same functions. This will be done by showing first that every TM-computable function is partial recursive (where the partial recursive functions is a third model of computation), and then that every partial recursive function is URM-computable. One could show as well directly that every TM-computable function is URM-computable, but the way taken in this lecture is probably easier.

**Extension to arbitrary alphabets.** Let  $A$  be a finite alphabet s.t.  $\sqcup \notin A$ , and  $B := A^*$ . To a Turing machine  $T = (\Sigma, S, l, \sqcup, s_0)$  with  $A \subseteq \Sigma$  corresponds a partial function  $T^{(A,n)} : B^n \rightharpoonup B$ , where  $T^{(A,n)}(a_0, \dots, a_{n-1})$  is computed as follows:

- Initially write  $a_0 \sqcup \dots \sqcup a_{n-1}$  on the tape, otherwise blanks. Start in state  $s_0$  on the left most position of  $a_0$ .
- Iterate TM as before.
- In case of termination, the output of the function is  $c_0 \dots c_{l-1}$ , if the tape contains, starting with the head position  $c_0 \dots c_{l-1} d$  with  $c_i \in A$ ,  $d \notin A$ .
- Otherwise, the function value is undefined.

This notion is modulo encoding of  $A^*$  into  $\mathbb{N}$  equivalent to the notion of Turing-computability on  $\mathbb{N}$ . However, when considering complexity bounds, it might be more appropriate, since the encoding and decoding of natural numbers might exceed the intended complexity bounds.

**Remark on Turing-computable predicates:** A predicate  $A$  is Turing-decidable, iff  $\chi_A$  is Turing-computable. However, instead of simulating  $\chi_A$ , which amounts to finally writing the output of  $\chi_A$  (a binary number 0 or 1) on the tape, it is more convenient, to take TM with two additional special states  $s_{\text{true}}$  and  $s_{\text{false}}$  corresponding to truth and falsity of the predicate. Then we can say that a predicate is Turing decidable, if, when we write initially the inputs as before on the tape and start executing the TM, it always terminates in  $s_{\text{true}}$  or  $s_{\text{false}}$ , and it terminates in  $s_{\text{true}}$ , iff the predicate holds for the inputs, and in  $s_{\text{false}}$ , otherwise. This notion, which is equivalent to the first notion, is usually taken as basis for complexity considerations.

## 5 Algebraic View of Computability

The previous models of computations (URMs and TMs) were based on programming languages, which allow to introduce all computable functions. In this section we discuss a model of computation, in which the set of computable functions is generated from basic functions by using certain operations. Thus we describe the set of computable functions as the least algebra of functions containing the basic functions, and which is closed under these operations. This algebraic approach was proposed by Gödel and Kleene in 1936, and is a elegant and mathematically rigorous definition of the set of computable functions. In order to show that a function is computable, it is usually easier to show that it is computable in this model.

We will first introduce the **primitive-recursive functions**. This is a set of **total** functions, which includes all functions which can be realistically computed, and many more, but does not cover all computable functions. Then we will introduce the **partial recursive functions**, which is a complete model of computation. Then we will show that the sets of URM-computable, of TM-computable functions and of partial recursive functions coincide.

### 5.1 The Primitive-Recursive Functions

**Definition 5.1** We define inductively the set of primitive-recursive functions  $f$  together with their arity, i.e. together with the  $k$  s.t.  $f : \mathbb{N}^k \rightarrow \mathbb{N}$ .

We write “ $f : \mathbb{N}^k \rightarrow \mathbb{N}$  is primitive-recursive” for “ $f$  is primitive-recursive with arity  $k$ ”, and  $\mathbb{N}$  for  $\mathbb{N}^1$ .

- The following basic functions are primitive-recursive:

- zero :  $\mathbb{N} \rightarrow \mathbb{N}$ ,
- succ :  $\mathbb{N} \rightarrow \mathbb{N}$ ,
- $\text{proj}_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$  ( $0 \leq i \leq k$ ).

(Remember that these functions have defining equations

- zero( $n$ ) = 0,
- succ( $n$ ) =  $n + 1$ ,
- $\text{proj}(a_0, \dots, a_{k-1}) = a_i$ .)

- If  $g : \mathbb{N}^k \rightarrow \mathbb{N}$  is primitive-recursive, and for  $i = 0, \dots, k - 1$  we have  $h_i : \mathbb{N}^n \rightarrow \mathbb{N}$  is primitive-recursive, then  $g \circ (h_0, \dots, h_{k-1}) : \mathbb{N}^n \rightarrow \mathbb{N}$  is primitive-recursive as well.

(Remember that  $f := g \circ (h_0, \dots, h_{k-1})$  has defining equation:

- $f(\vec{x}) = g(h_0(\vec{x}), \dots, h_{k-1}(\vec{x}))$ .)

Some notations: In the special case  $k = 1$  we write  $g \circ h$  instead of  $g \circ (h)$ , and we write  $g_0 \circ g_1 \circ g_2 \circ \dots \circ g_n$  instead of  $g_0 \circ (g_1 \circ (g_2 \circ \dots \circ g_n))$ .

- If  $g : \mathbb{N}^n \rightarrow \mathbb{N}$ , and  $h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$  are primitive-recursive, then  $\text{primrec}(g, h) : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  is primitive-recursive as well.

(Remember that  $f := \text{primrec}(g, h)$  has defining equations

- $f(\vec{x}, 0) = g(\vec{x})$ ,
- $f(\vec{x}, n + 1) = h(\vec{x}, n, f(\vec{x}, n))$ .)

- If  $k \in \mathbb{N}$  and  $h : \mathbb{N}^2 \rightarrow \mathbb{N}$  is primitive-recursive, then  $\text{primrec}(k, h) : \mathbb{N} \rightarrow \mathbb{N}$  is primitive-recursive as well.

(Remember that  $f := \text{primrec}(k, h)$  has defining equations

- $f(0) = k$ ,
- $f(n + 1) = h(n, f(n))$ .)

**Remark:** That we defined inductively this set means that the set of primitive-recursive functions is the least set closed under the above mentioned operations, or that it is the set generated by the above operations: The primitive-recursive functions are exactly those for which we can introduce a term formed from zero, succ,  $\text{proj}_i^n$ ,  $\circ$  ( $-, \dots, -$ ) (i.e if  $f, g_i$  are terms so is  $f \circ (g_0, \dots, g_{n-1})$ ) and primrec, provided we respect the arities of the functions as above.

Examples:

- $\text{primrec}(\underbrace{\text{proj}_0^1}_{:\mathbb{N} \rightarrow \mathbb{N}}, \underbrace{\text{succ} \circ \text{proj}_2^3}_{:\mathbb{N}^3 \rightarrow \mathbb{N}}) : \mathbb{N}^2 \rightarrow \mathbb{N}$  is prim. rec.

(We will see below that this is the function  $\text{add} : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $\text{add}(x, y) := x + y$ ).

- $\text{primrec}(\underbrace{0}_{\in \mathbb{N}}, \underbrace{\text{proj}_0^2}_{:\mathbb{N}^2 \rightarrow \mathbb{N}}) : \mathbb{N} \rightarrow \mathbb{N}$  is prim. rec..

(We will see below that this is the function  $\text{pred} : \mathbb{N} \rightarrow \mathbb{N}$ ,  $\text{pred}(x) := x - 1$ ).

**Definition 5.2** A relation  $R \subseteq \mathbb{N}^n$  is a primitive-recursive relation, if the characteristic function  $\chi_R : \mathbb{N}^n \rightarrow \mathbb{N}$  is primitive-recursive.

**Examples:**

- The identity function  $\text{id} : \mathbb{N} \rightarrow \mathbb{N}$ ,  $\text{id}(n) = n$  is primitive-recursive, since  $\text{id} = \text{proj}_0^1$ ;  $\text{proj}_0^1 : \mathbb{N}^1 \rightarrow \mathbb{N}$ ,  $\text{proj}_0^1(n) = n = \text{id}(n)$ .
- The constant function  $\text{const}_n : \mathbb{N} \rightarrow \mathbb{N}$ ,  $\text{const}_n(k) = n$  is primitive-recursive, since  $\text{const}_n = \underbrace{\text{succ} \circ \dots \circ \text{succ}}_{n \text{ times}} \circ \text{zero}$ :

$$\begin{aligned} \underbrace{\text{succ} \circ \dots \circ \text{succ}}_{n \text{ times}} \circ \text{zero}(k) &= \underbrace{\text{succ}(\text{succ}(\dots \text{succ}(\text{zero}(k))))}_{n \text{ times}} \\ &= \underbrace{\text{succ}(\text{succ}(\dots \text{succ}(0)))}_{n \text{ times}} \\ &= \underbrace{0 + 1 + 1 \dots + 1}_{n \text{ times}} \\ &= n \\ &= \text{const}_n(k) . \end{aligned}$$

- Addition is primitive-recursive. We have seen previously that  $\text{add} : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $\text{add}(x, y) = x + y$  follows the rules

$$\begin{aligned} \text{add}(x, 0) &= x + 0 \\ &= x \\ &= g(x) , \\ \text{add}(x, y + 1) &= x + (y + 1) \\ &= (x + y) + 1 \\ &= \text{add}(x, y) + 1 \\ &= h(x, y, \text{add}(x, y)) . \end{aligned}$$

where

$$g : \mathbb{N} \rightarrow \mathbb{N}, \quad g(x) = x, \quad \text{therefore } g = \text{id} = \text{proj}_0^1 \text{ prim. rec. ,}$$

and

$$h : \mathbb{N}^3 \rightarrow \mathbb{N} \quad h(x, y, z) := z + 1 .$$

We have  $h = \text{succ} \circ \text{proj}_0^3$  and therefore prim. rec. :

$$\begin{aligned} (\text{succ} \circ \text{proj}_2^3)(x, y, z) &= \text{succ}(\text{proj}_2^3(x, y, z)) \\ &= \text{succ}(z) \\ &= z + 1 \\ &= h(x, y, z) . \end{aligned}$$

Therefore  $\text{add} = \text{primrec}(\text{proj}_0^1, \text{succ} \circ \text{proj}_2^3)$ .

- Multiplication is primitive-recursive. We have seen previously that  $\text{mult} : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $\text{mult}(x, y) = x \cdot y$  follows the rules

$$\begin{aligned} \text{mult}(x, 0) &= x \cdot 0 \\ &= 0 \\ &= g(x) , \\ \text{mult}(x, y + 1) &= x \cdot (y + 1) \\ &= x \cdot y + x \\ &= \text{mult}(x, y) + x \\ &= \text{add}(\text{mult}(x, y), x) \\ &= h(x, y, \text{mult}(x, y)) , \end{aligned}$$

where

$$g : \mathbb{N} \rightarrow \mathbb{N} , \quad g(x) = 0 \text{ and therefore } g = \text{zero prim. rec.}$$

and

$$h : \mathbb{N}^3 \rightarrow \mathbb{N} , \quad h(x, y, z) = \text{add}(z, x) .$$

We have  $h = \text{add} \circ (\text{proj}_2^3, \text{proj}_0^3)$  and therefore prim. rec. :

$$\begin{aligned} (\text{add} \circ (\text{proj}_2^3, \text{proj}_0^3))(x, y, z) &= \text{add}(\text{proj}_2^3(x, y, z), \text{proj}_0^3(x, y, z)) \\ &= \text{add}(z, x) \\ &= h(x, y, z) . \end{aligned}$$

Therefore  $\text{mult} = \text{primrec}(\text{zero}, \text{add} \circ (\text{proj}_2^3, \text{proj}_0^3))$  is primitive-recursive.

#### Remark:

- Unless a direct reduction to the principle of primitive recursion is demanded, for showing that a function  $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  is primitive-recursive defined by using  $\text{primrec}$  it suffices to show informally that

- $f(\vec{x}, 0)$  can be defined by an expression built from previously defined primitive-recursive functions, parameters  $\vec{x}$ , and constants.

Example:

$$f(x_0, x_1, 0) = (x_0 + x_1) \cdot 3 .$$

- $f(\vec{x}, y + 1)$  can be defined by an expression built from previously defined primitive-recursive functions, parameters  $\vec{x}$ , the *recursion argument*  $y$ , the *recursion hypothesis*  $f(\vec{x}, y)$ , and constants.

Example:

$$f(x_0, x_1, y + 1) = (x_0 + x_1 + y + f(x_0, x_1, y)) \cdot 3 .$$

- Similarly, if one wants to verify that a function is primitive-recursive by giving a direct definition of it in terms of other primitive-recursive functions (i.e. by using previously defined primitive-recursive functions and composition), it suffices to show that  $f(\vec{x})$  can be defined by an expression built from previously defined primitive-recursive functions, parameters  $\vec{x}$  and constants.

Example:

$$f(x, y, z) = (x + y) \cdot 3 + z .$$

All the following proofs will be given in this style and are therefore examples for this principle.

We continue with the introduction of primitive-recursive functions:

- The predecessor function  $\text{pred}$  is primitive-recursive, because of the equations

$$\begin{aligned}\text{pred}(0) &= 0, \\ \text{pred}(x+1) &= x,\end{aligned}$$

where the induction step doesn't refer to the recursion hypothesis, but only to the recursion argument.

- The function  $f(x, y) = x \dot{-} y$  is primitive-recursive, because of the equations

$$\begin{aligned}f(x, 0) &= x, \\ f(x, y+1) &= \text{pred}(f(x, y)).\end{aligned}$$

- The signum-function  $\text{sig} : \mathbb{N} \rightarrow \mathbb{N}$ ,

$$\text{sig}(x) := \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{if } x = 0 \end{cases}$$

is primitive-recursive, since  $\text{sig}(x) = x \dot{-} (x \dot{-} 1)$ :

- For  $x = 0$  we have

$$\begin{aligned}x \dot{-} (x \dot{-} 1) &= 0 \dot{-} (0 \dot{-} 1) \\ &= 0 \dot{-} 0 \\ &= 0 \\ &= \text{sig}(x).\end{aligned}$$

- For  $x > 0$  we have

$$\begin{aligned}x \dot{-} (x \dot{-} 1) &= x - (x - 1) \\ &= x - x + 1 \\ &= 1 \\ &= \text{sig}(x).\end{aligned}$$

Alternatively, one can show that  $\text{sig}$  is primitive-recursive by using the principle of primitive recursion and the equations

$$\begin{aligned}\text{sig}(0) &= 0, \\ \text{sig}(x+1) &= 1.\end{aligned}$$

- The relation “ $x < y$ ”, i.e.  $A(x, y) :\Leftrightarrow x < y$  is primitive-recursive, since  $\chi_A(x, y) = \text{sig}(y \dot{-} x)$ :

- If  $x < y$ , then

$$y \dot{-} x = y - x > 0,$$

therefore

$$\text{sig}(y \dot{-} x) = 1 = \chi_A(x, y).$$

- If  $\neg(x < y)$ , i.e.  $x \geq y$ , then

$$y \dot{-} x = 0,$$

therefore

$$\text{sig}(y \dot{-} x) = 0.$$

- Consider the sequence of definitions of addition, multiplication, exponentiation:

– **Addition:**

$$\begin{aligned} n + 0 &= n , \\ n + (m + 1) &= (n + m) + 1 , \end{aligned}$$

Therefore, if we write  $(+1)$  for the function  $\mathbb{N} \rightarrow \mathbb{N}$ ,  $(+1)(n) = n + 1$ , then

$$n + m = (+1)^m(n) .$$

– **Multiplication:**

$$\begin{aligned} n \cdot 0 &= 0 , \\ n \cdot (m + 1) &= (n \cdot m) + n , \end{aligned}$$

Therefore, if we write  $(+n)$  for the function  $\mathbb{N} \rightarrow \mathbb{N}$ ,  $(+n)(k) = k + n$ , then

$$n \cdot m = (+n)^m(0) .$$

– **Exponentiation:**

$$\begin{aligned} n^0 &= 1 , \\ n^{m+1} &= (n^m) \cdot n , \end{aligned}$$

Therefore, if we write  $(\cdot n)$  for the function  $\mathbb{N} \rightarrow \mathbb{N}$ ,  $(\cdot n)(m) = n \cdot m$ , then

$$n^m = (\cdot n)^m(1) .$$

We can extend this sequence further, by defining

– **Superexponentiation:**

$$\begin{aligned} \text{superexp}(n, 0) &= 1 , \\ \text{superexp}(n, m + 1) &= n^{\text{superexp}(n, m)} , \end{aligned}$$

Therefore, if we write  $(n \uparrow)$  for the function  $\mathbb{N} \rightarrow \mathbb{N}$ ,  $(n \uparrow)(k) = n^k$ , then

$$n^m = (n \uparrow)^m(1) .$$

– **Supersuperexponentiation:**

$$\begin{aligned} \text{supersuperexp}(n, 0) &= 1 , \\ \text{supersuperexp}(n, m + 1) &= \text{superexp}(n, \text{supersuperexp}(n, m)) , \end{aligned}$$

– Etc.

This way we obtain a sequence of extremely fast growing functions.

Traditionally, instead of considering such binary functions, one considers a sequence of unary functions. which have a similar growth rate. These functions are called the Ackermann functions, and will exhaust all primitive recursive functions (in the sense of Lemma 5.9):

- For  $n \in \mathbb{N}$ , the  $n$ -th branch of the Ackermann function  $\text{Ack}_n : \mathbb{N} \rightarrow \mathbb{N}$ , is defined by

$$\begin{aligned} \text{Ack}_0(y) &= y + 1 , \\ \text{Ack}_{n+1}(y) &= (\text{Ack}_n)^{y+1}(1) := \underbrace{\text{Ack}_n(\text{Ack}_n(\dots \text{Ack}_n(1)))}_{y+1 \text{ times}} . \end{aligned}$$

$\text{Ack}_n$  is primitive-recursive **for fixed**  $n$ . We show this by induction on  $n$ :

- **Base-case:**  $\text{Ack}_0 = \text{succ}$  is primitive-recursive.
- **Induction step:** Assume  $\text{Ack}_n$  is primitive-recursive. Show  $\text{Ack}_{n+1}$  is primitive-recursive. We have:

$$\begin{aligned}\text{Ack}_{n+1}(0) &= \text{Ack}_n(1) , \\ \text{Ack}_{n+1}(y+1) &= \text{Ack}_n(\text{Ack}_{n+1}(y)) .\end{aligned}$$

Therefore  $\text{Ack}_{n+1}$  is primitive-recursive.

**Remark:**

$$\begin{aligned}\text{Ack}_0(n) &= n + 1 . \\ \text{Ack}_1(n) &= \text{Ack}_0^{n+1}(1) \\ &= \underbrace{1+1+\cdots+1}_{n+1 \text{ times}} \\ &= 1 + n + 1 = n + 2 . \\ \text{Ack}_2(n) &= \text{Ack}_1^{n+1}(1) \\ &= \underbrace{1+2+\cdots+2}_{n+1 \text{ times}} \\ &= 1 + 2(n+1) \\ &= 2n + 3 > 2n . \\ \text{Ack}_3(n) &= \text{Ack}_2^{n+1}(1) \\ &> \underbrace{2 \cdot 2 \cdots 2 \cdot 1}_{n+1 \text{ times}} \\ &= 2^{n+1} > 2^n . \\ \text{Ack}_4(n) &= \text{Ack}_3^{n+1}(1) \\ &> \underbrace{2 \cdots 2^1}_{n+1 \text{ times}} .\end{aligned}$$

$\text{Ack}_5(n)$  will iterate  $\text{Ack}_4$   $n+1$  times, etc.

So for small  $n$ ,  $\text{Ack}_5(n)$  will exceed the number of particles in the universe, and therefore  $\text{Ack}_5$  is not realistically computable.

Whereas  $\text{Ack}_n$  for fixed  $n$  is primitive-recursive, we will show below that a uniform version of the Ackermann function, i.e.  $\text{Ack} : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $\text{Ack}(x, y) := \text{Ack}_x(y)$ , is **not** primitive-recursive.

## 5.2 Closure Properties of the Primitive-Recursive Functions

- The primitive-recursive relations are closed under union, intersection, and complements:  
If  $R, S \subseteq \mathbb{N}^n$  are primitive-recursive, so are  $R \cup S$ ,  $R \cap S$ ,  $\mathbb{N}^n \setminus R$ .  
Note that
- $(R \cup S)(\vec{x}) \Leftrightarrow R(\vec{x}) \vee S(\vec{x})$ :

$$\begin{aligned}(R \cup S)(\vec{x}) &\Leftrightarrow \vec{x} \in R \cup S \\ &\Leftrightarrow \vec{x} \in R \vee \vec{x} \in S \\ &\Leftrightarrow R(\vec{x}) \vee S(\vec{x})\end{aligned}$$

- $(R \cap S)(\vec{x}) \Leftrightarrow R(\vec{x}) \wedge S(\vec{x})$ :

$$\begin{aligned} (R \cap S)(\vec{x}) &\Leftrightarrow \vec{x} \in R \cap S \\ &\Leftrightarrow \vec{x} \in R \wedge \vec{x} \in S \\ &\Leftrightarrow R(\vec{x}) \wedge S(\vec{x}) \end{aligned}$$

- $(\mathbb{N}^n \setminus R)(\vec{x}) \Leftrightarrow \neg R(\vec{x})$ :

$$\begin{aligned} (\mathbb{N}^n \setminus R)(\vec{x}) &\Leftrightarrow \vec{x} \in (\mathbb{N}^n \setminus R) \\ &\Leftrightarrow \vec{x} \notin R \\ &\Leftrightarrow \neg R(\vec{x}) \end{aligned}$$

Therefore, the primitive-recursive predicates are essentially closed under  $\vee$ ,  $\wedge$ ,  $\neg$ .

Proof that primitive-recursive predicates are closed under  $\cup$ ,  $\cap$  and complement:

- $\chi_{R \cup S}(\vec{x}) = \text{sig}(\chi_R(\vec{x}) + \chi_S(\vec{x}))$  (and therefore primitive-recursive):

- \* If  $R(\vec{x})$  holds, then

$$\underbrace{\underbrace{\text{sig}(\underbrace{\chi_R(\vec{x})}_{=1} + \underbrace{\chi_S(\vec{x})}_{\geq 0})}_{\geq 1}}_{=1} = 1 = \chi_{R \cup S}(\vec{x}) .$$

- \* Similarly, if  $S(\vec{x})$  holds, then

$$\underbrace{\underbrace{\text{sig}(\underbrace{\chi_R(\vec{x})}_{\geq 0} + \underbrace{\chi_S(\vec{x})}_{=1})}_{\geq 1}}_{=1} = 1 = \chi_{R \cup S}(\vec{x}) .$$

- \* If neither  $R(\vec{x})$  nor  $S(\vec{x})$  holds, then we have

$$\underbrace{\underbrace{\text{sig}(\underbrace{\chi_R(\vec{x})}_{=0} + \underbrace{\chi_S(\vec{x})}_{=0})}_{=0}}_{=0} = 0 = \chi_{R \cup S}(\vec{x}) .$$

- $\chi_{R \cap S}(\vec{x}) = \chi_R(\vec{x}) \cdot \chi_S(\vec{x})$  (and therefore  $R \cap S$  is primitive-recursive):

- \* If  $R(\vec{x})$  and  $S(\vec{x})$  hold, then

$$\underbrace{\underbrace{\chi_R(\vec{x})}_{=1} \cdot \underbrace{\chi_S(\vec{x})}_{=1}}_{=1} = 1 = \chi_{R \cap S}(\vec{x}) .$$

- \* If  $\neg R(\vec{x})$  holds, then  $\chi_R(\vec{x}) = 0$ , therefore

$$\underbrace{\underbrace{\chi_R(\vec{x})}_{=0} \cdot \chi_S(\vec{x})}_{=0} = 0 = \chi_{R \cap S}(\vec{x}) .$$

\* Similarly, if  $\neg S(\vec{x})$ , we have

$$\underbrace{\chi_R(\vec{x}) \cdot \underbrace{\chi_S(\vec{x})}_{=0}}_{=0} = 0 = \chi_{R \cap S}(\vec{x}) .$$

–  $\chi_{\mathbb{N}^n \setminus R}(\vec{x}) = 1 \dot{-} \chi_R(\vec{x})$  (and therefore primitive-recursive):

\* If  $R(\vec{x})$  holds, then  $\chi_R(\vec{x}) = 1$ , therefore

$$\underbrace{1 \dot{-} \underbrace{\chi_R(\vec{x})}_{=1}}_{=0} = 0 = \chi_{\mathbb{N}^n \setminus R}(\vec{x}) .$$

\* If  $R(\vec{x})$  does not hold, then  $\chi_R(\vec{x}) = 0$ ,

$$\underbrace{1 \dot{-} \underbrace{\chi_R(\vec{x})}_{=0}}_{=1} = 1 = \chi_{\mathbb{N}^n \setminus R}(\vec{x}) .$$

• The predicates “ $x \leq y$ ” and “ $x = y$ ” are primitive-recursive:

–  $x \leq y \Leftrightarrow \neg(y < x)$ .

The righthand of this equivalence is primitive-recursive, since ‘ $y < x$ ’ is primitive-recursive, and primitive-recursive predicates are closed under  $\neg$ .

–  $x = y \Leftrightarrow x \leq y \wedge y \leq x$ .

The righthand side of this equivalence is primitive-recursive, since “ $x \leq y$ ” and “ $y \leq x$ ” are primitive-recursive, and primitive-recursive predicates are closed under  $\wedge$ .

• The primitive-recursive functions are closed under *definition by cases*: Assume  $g_1, g_2 : \mathbb{N}^n \rightarrow \mathbb{N}$  are primitive-recursive, and  $R \subseteq \mathbb{N}^n$  is primitive-recursive, then the function  $f : \mathbb{N}^n \rightarrow \mathbb{N}$ ,

$$f(\vec{x}) := \begin{cases} g_1(\vec{x}), & \text{if } R(\vec{x}), \\ g_2(\vec{x}), & \text{if } \neg R(\vec{x}), \end{cases}$$

is primitive-recursive, since

$$f(\vec{x}) = g_1(\vec{x}) \cdot \chi_R(\vec{x}) + g_2(\vec{x}) \cdot \chi_{\mathbb{N}^n \setminus R}(\vec{x}) \quad :$$

– If  $R(\vec{x})$  holds, then  $\chi_R(\vec{x}) = 1$ ,  $\chi_{\mathbb{N}^n \setminus R}(\vec{x}) = 0$ ,

$$\underbrace{g_1(\vec{x}) \cdot \underbrace{\chi_R(\vec{x})}_{=1}}_{=g_1(\vec{x})} + \underbrace{g_2(\vec{x}) \cdot \underbrace{\chi_{\mathbb{N}^n \setminus R}(\vec{x})}_{=0}}_{=0} = g_1(\vec{x}) = f(\vec{x}) .$$

– If  $\neg R(\vec{x})$  holds, then  $\chi_R(\vec{x}) = 0$ ,  $\chi_{\mathbb{N}^n \setminus R}(\vec{x}) = 1$ ,

$$\underbrace{g_1(\vec{x}) \cdot \underbrace{\chi_R(\vec{x})}_{=0}}_{=0} + \underbrace{g_2(\vec{x}) \cdot \underbrace{\chi_{\mathbb{N}^n \setminus R}(\vec{x})}_{=1}}_{=g_2(\vec{x})} = g_2(\vec{x}) = f(\vec{x}) .$$

- The primitive-recursive functions are closed under bounded sums:

If  $g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  is primitive-recursive, so is

$$f : \mathbb{N}^{n+1} \rightarrow \mathbb{N} , \quad f(\vec{x}, y) := \sum_{z < y} g(\vec{x}, z) ,$$

where

$$\sum_{z < 0} g(\vec{x}, z) := 0 ,$$

and for  $y > 0$ ,

$$\sum_{z < y} g(\vec{x}, z) := g(\vec{x}, 0) + g(\vec{x}, 1) + \cdots + g(\vec{x}, y - 1) .$$

Proof that  $f$  is primitive-recursive:

This follows by the equations:

$$\begin{aligned} f(\vec{x}, 0) &= 0 , \\ f(\vec{x}, y + 1) &= f(\vec{x}, y) + g(\vec{x}, y) . \end{aligned}$$

Here, the last equations follows from

$$\begin{aligned} f(\vec{x}, y + 1) &= \sum_{z < y+1} g(\vec{x}, z) \\ &= \left( \sum_{z < y} g(\vec{x}, z) \right) + g(\vec{x}, y) \\ &= f(\vec{x}, y) + g(\vec{x}, y) . \end{aligned}$$

- The primitive-recursive functions are closed under bounded products:

If  $g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  is primitive-recursive, so is

$$f : \mathbb{N}^{n+1} \rightarrow \mathbb{N} , \quad f(\vec{x}, y) := \prod_{z < y} g(\vec{x}, z) ,$$

where

$$\prod_{z < 0} g(\vec{x}, z) := 1 ,$$

and for  $y > 0$ ,

$$\prod_{z < y} g(\vec{x}, z) := g(\vec{x}, 0) \cdot g(\vec{x}, 1) \cdot \cdots \cdot g(\vec{x}, y - 1) .$$

Proof that  $f$  is primitive-recursive:

This follows by the equations:

$$\begin{aligned} f(\vec{x}, 0) &= 1 , \\ f(\vec{x}, y + 1) &= f(\vec{x}, y) \cdot g(\vec{x}, y) . \end{aligned}$$

Here, the last equations follows by

$$\begin{aligned} f(\vec{x}, y + 1) &= \prod_{z < y+1} g(\vec{x}, z) \\ &= \left( \prod_{z < y} g(\vec{x}, z) \right) \cdot g(\vec{x}, y) \\ &= f(\vec{x}, y) \cdot g(\vec{x}, y) . \end{aligned}$$

**Example for closure under bounded products:**

$f : \mathbb{N} \rightarrow \mathbb{N}$ ,

$$f(n) := n! = 1 \cdot 2 \cdot \dots \cdot n$$

is primitive recursive, since

$$f(n) = \prod_{i < n} (i + 1) = \prod_{i < n} g(i) ,$$

where  $g(i) := i + 1$  is prim.-rec..

(Note that in the special case  $n = 1$  we have

$$f(0) = 0! = 1 = \prod_{i < 0} (i + 1) .$$

- The primitive-recursive relations are closed under bounded quantification:  
If  $R \subseteq \mathbb{N}^{n+1}$  is primitive-recursive, so are the relations

$$\begin{aligned} R_1(\vec{x}, y) &:\Leftrightarrow \forall z < y. R(\vec{x}, z) , \\ R_2(\vec{x}, y) &:\Leftrightarrow \exists z < y. R(\vec{x}, z) . \end{aligned}$$

**Proof for  $R_1$ :**

$$\chi_{R_1}(\vec{x}, y) = \prod_{z < y} \chi_R(\vec{x}, z) :$$

- If  $\forall z < y. R(\vec{x}, z)$  holds, then  $\forall z < y. \chi_R(\vec{x}, z) = 1$ , therefore

$$\prod_{z < y} \chi_R(\vec{x}, z) = \prod_{z < y} 1 = 1 = \chi_{R_1}(\vec{x}, y) .$$

- If for one  $z < y$  we have  $\neg R(\vec{x}, z)$ , then for this  $z$  we have  $\chi_R(\vec{x}, z) = 0$ , therefore

$$\prod_{z < y} \chi_R(\vec{x}, z) = 0 = \chi_{R_1}(\vec{x}, y) .$$

**Proof for  $R_2$ :**

$$\chi_{R_2}(\vec{x}, y) = \text{sig}\left(\sum_{z < y} \chi_R(\vec{x}, z)\right) :$$

- If  $\forall z < y. \neg R(\vec{x}, z)$ , then

$$\begin{aligned} \text{sig}\left(\sum_{z < y} \chi_R(\vec{x}, z)\right) &= \text{sig}\left(\sum_{z < y} 0\right) \\ &= \text{sig}(0) \\ &= 0 \\ &= \chi_{R_2}(\vec{x}, y) . \end{aligned}$$

- If for one  $z < y$  we have  $R(\vec{x}, z)$ , then for this  $z$  we have  $\chi_R(\vec{x}, z) = 1$ , therefore

$$\sum_{z < y} \chi_R(\vec{x}, z) \geq \chi_R(\vec{x}, z) = 1 ,$$

therefore

$$\text{sig}\left(\sum_{z < y} \chi_R(\vec{x}, z)\right) = 1 = \chi_{R_2}(\vec{x}, y) .$$

- The primitive-recursive functions are closed under *bounded search*, i.e. if  $R \subseteq \mathbb{N}^{n+1}$  is a primitive-recursive predicate, so is  $f(\vec{x}, y) := \mu z < y. R(\vec{x}, z)$  where

$$\mu z < y. R(\vec{x}, z) := \begin{cases} \text{the least } z \text{ s.t. } R(\vec{x}, z) \text{ holds,} & \text{if such } z \text{ exists,} \\ y & \text{otherwise.} \end{cases}$$

**Proof:** Define

$$\begin{aligned} Q(\vec{x}, y) & :\Leftrightarrow R(\vec{x}, y) \wedge \forall z < y. \neg R(\vec{x}, z) , \\ Q'(\vec{x}, y) & :\Leftrightarrow \forall z < y. \neg R(\vec{x}, z) . \end{aligned}$$

$Q$  and  $Q'$  are primitive-recursive.  $Q(\vec{x}, y)$  holds exactly, if  $y$  is the minimal  $z$  s.t.  $R(\vec{x}, z)$  holds (i.e. if  $R(\vec{x}, y)$  holds, but for all  $z < y$   $R(\vec{x}, z)$  is false).

We show

$$f(\vec{x}, y) = \left( \sum_{z < y} \chi_Q(\vec{x}, z) \cdot z \right) + \chi_{Q'}(\vec{x}, y) \cdot y :$$

- If there exists  $z < y$  s.t.  $R(\vec{x}, z)$  holds, then for the minimal such  $z$  we have  $Q(\vec{x}, z)$ , therefore

$$\chi_Q(\vec{x}, z) \cdot z = z .$$

For all other  $z'$  we have  $\neg Q(\vec{x}, z')$ , therefore

$$\text{for } z' \neq z \text{ } \chi_Q(\vec{x}, z') \cdot z' = 0 .$$

Furthermore,  $\neg Q'(\vec{x}, y)$ , therefore

$$\chi_{Q'}(\vec{x}, y) \cdot y = 0 .$$

It follows

$$\left( \sum_{z < y} \chi_Q(\vec{x}, z) \cdot z \right) + \chi_{Q'}(\vec{x}, y) \cdot y = z = f(\vec{x}, y) .$$

- If there exists no  $z < y$  s.t.  $R(\vec{x}, z)$  holds, then we have  $\neg Q(\vec{x}, z)$  for all  $z < y$ , therefore

$$\forall z < y. \chi_Q(\vec{x}, z) \cdot z = 0 .$$

Furthermore,  $Q'(\vec{x}, y)$ , therefore

$$\chi_{Q'}(\vec{x}, y) \cdot y = y .$$

It follows

$$\left( \sum_{z < y} \chi_Q(\vec{x}, z) \cdot z \right) + \chi_{Q'}(\vec{x}, y) \cdot y = y = f(\vec{x}, y) .$$

We show that the functions, which encode fixed length and arbitrary length sequences as natural numbers or decode them, are primitive-recursive:

**Lemma 5.3** *The following functions are primitive-recursive:*

- The pairing functions  $\pi : \mathbb{N}^2 \rightarrow \mathbb{N}$ .  
(Remember, that  $\pi(n, m)$  encodes two natural numbers as one.)
- The projections  $\pi_0, \pi_1 : \mathbb{N} \rightarrow \mathbb{N}$ .  
(Remember that  $\pi_0(\pi(n, m)) = n$ ,  $\pi_1(\pi(n, m)) = m$ , so  $\pi_0, \pi_1$  invert  $\pi$ .)
- $\pi^k : \mathbb{N}^k \rightarrow \mathbb{N}$  ( $k \geq 1$ ).  
(Remember that  $\pi^k(n_0, \dots, n_{k-1})$  encodes the sequence  $(n_0, \dots, n_k)$ .)

(d)  $f : \mathbb{N}^3 \rightarrow \mathbb{N}$ ,

$$f(x, k, i) = \begin{cases} \pi_i^k(x), & \text{if } i < k, \\ x, & \text{otherwise.} \end{cases}$$

(Remember that  $\pi_i^k(\pi^k(n_0, \dots, n_{k-1})) = n_i$  for  $i < k$ .)

We write  $\pi_i^k(x)$  for  $f(x, k, i)$ , even if  $i \geq k$ .

(e) The function  $f_k : \mathbb{N}^k \rightarrow \mathbb{N}$ ,  $f_k(x_0, \dots, x_{k-1}) = \langle x_0, \dots, x_{k-1} \rangle$ .

(Remember that  $\langle x_0, \dots, x_{k-1} \rangle$  encodes the sequence  $x_0, \dots, x_{k-1}$  as one natural number. Note that it doesn't make sense to say that  $\lambda x. \langle x \rangle : \mathbb{N}^* \rightarrow \mathbb{N}$  is primitive-recursive, since each primitive-recursive function has to have domain  $\mathbb{N}^k$  for some  $k$ .)

(f)  $\text{lh} : \mathbb{N} \rightarrow \mathbb{N}$ .

(Remember that  $\text{lh}(\langle x_0, \dots, x_{k-1} \rangle) = k$ .)

(g)  $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $g(x, i) = (x)_i$ .

(Remember that  $(\langle x_0, \dots, x_{k-1} \rangle)_i = x_i$  for  $i < k$ .)

**Proof:**

(a)

$$\begin{aligned} \pi(n, m) &= \left( \sum_{i \leq n+m} i \right) + m \\ &= \left( \sum_{i < n+m+1} i \right) + m \end{aligned}$$

is primitive-recursive.

(b) One can easily show that  $n, m \leq \pi(n, m)$ . Therefore we can define

$$\begin{aligned} \pi_0(n) &:= \mu k < n + 1. \exists l < n + 1. n = \pi(k, l) , \\ \pi_1(n) &:= \mu l < n + 1. \exists k < n + 1. n = \pi(k, l) . \end{aligned}$$

Therefore  $\pi_0, \pi_1$  are primitive-recursive.

(c) Proof by induction on  $k$ .

$k = 1$ :  $\pi^1(x) = x$ , so  $\pi^1$  is primitive-recursive.

$k \rightarrow k + 1$ : Assume that  $\pi^k$  is primitive-recursive. Show that  $\pi^{k+1}$  is primitive-recursive as well:

$$\pi^{k+1}(x_0, \dots, x_k) = \pi(\pi^k(x_0, \dots, x_{k-1}), x_k) .$$

Therefore  $\pi^{k+1}$  is primitive-recursive (using that  $\pi, \pi^k$  are primitive-recursive).

(d) We have

$$\begin{aligned} \pi_0^1(x) &= x , \\ \pi_i^{k+1}(x) &= \pi_i^k(\pi_0(x)), \text{ if } i < k, \\ \pi_i^{k+1}(x) &= \pi_1(x), \text{ if } i = k, \end{aligned}$$

Therefore

$$\pi_i^k(x) = \begin{cases} \pi_1((\pi_0)^{k-i}(x)), & \text{if } i > 0, \\ (\pi_0)^k(x), & \text{if } i = 0. \end{cases}$$

and

$$f(x, k, i) = \begin{cases} x, & \text{if } i \geq k, \\ \pi_1((\pi_0)^{k-i}(x)), & \text{if } 0 < i < k, \\ (\pi_0)^k(x), & \text{if } i = 0 < k. \end{cases}$$

Define  $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,

$$\begin{aligned} g(x, 0) &:= x, \\ g(x, k+1) &:= \pi_0(g(x, k)), \end{aligned}$$

which is primitive-recursive. Then we get  $g(x, k) = (\pi_0)^k(x)$ , therefore

$$f(x, k, i) = \begin{cases} x, & \text{if } i \geq k, \\ \pi_1(g(x, k \dot{-} i)), & \text{if } 0 < i < k, \\ g(x, k), & \text{if } i = 0 < k. \end{cases}$$

So  $f$  is primitive-recursive.

(e)

$$f_k(x_0, \dots, x_{k-1}) = 1 + \pi(k \dot{-} 1, \pi^k(x_0, \dots, x_{k-1}))$$

is primitive-recursive.

(f)

$$\text{lh}(x) = \begin{cases} 0, & \text{if } x = 0, \\ \pi_0(x \dot{-} 1) + 1, & \text{if } x \neq 0. \end{cases}$$

(g)

$$\begin{aligned} (x)_i &= \pi_i^{\text{lh}(x)}(\pi_1(x \dot{-} 1)) \\ &= f(\pi_1(x \dot{-} 1), \text{lh}(x), i) \end{aligned}$$

is primitive-recursive.

**Lemma and Definition 5.4** *There exists primitive-recursive functions with the following properties:*

- (a) A function  $\text{snoc} : \mathbb{N}^2 \rightarrow \mathbb{N}$  s.t.  $\text{snoc}(\langle x_0, \dots, x_{n-1} \rangle, x) = \langle x_0, \dots, x_{n-1}, x \rangle$ .  
*(The name snoc is derived from inverting the letters in cons. Our definition of the encoding of sequences corresponds to appending new elements at the end rather than at the beginning).*
- (b) Functions  $\text{last} : \mathbb{N} \rightarrow \mathbb{N}$  and  $\text{beginning} : \mathbb{N} \rightarrow \mathbb{N}$  s.t.

$$\begin{aligned} \text{last}(\text{snoc}(x, y)) &= y, \\ \text{beginning}(\text{snoc}(x, y)) &= x. \end{aligned}$$

**Proof:**

(a) Define

$$\text{snoc}(x, y) = \begin{cases} \langle y \rangle, & \text{if } x = 0, \\ 1 + \pi(\text{lh}(x), \pi(\pi_1(x \dot{-} 1), y)), & \text{otherwise,} \end{cases}$$

so  $\text{snoc}$  is primitive-recursive.

We have

$$\begin{aligned} \text{snoc}(\langle \rangle, y) &= \text{snoc}(0, y) \\ &= \langle y \rangle, \\ \text{snoc}(\langle x_0, \dots, x_k \rangle, y) &= \text{snoc}(1 + \pi(k, \pi^{k+1}(x_0, \dots, x_k)), y) \\ \text{lh}(\langle x_0, \dots, x_k \rangle) &= k + 1 \\ &= 1 + \pi(k + 1, \pi(\pi_1((1 + \pi(k, \pi^{k+1}(x_0, \dots, x_k))) \dot{-} 1), y)) \\ &= 1 + \pi(k + 1, \pi(\pi_1(\pi(k, \pi^{k+1}(x_0, \dots, x_k))), y)) \\ &= 1 + \pi(k + 1, \pi(\pi^{k+1}(x_0, \dots, x_k), y)) \\ &= 1 + \pi(k + 1, \pi^{k+2}(x_0, \dots, x_k, y)) \\ &= \langle x_0, \dots, x_k, y \rangle. \end{aligned}$$

(b)

**Proof for beginning:**

Define

$$\text{beginning}(x) := \begin{cases} \langle \rangle, & \text{if } \text{lh}(x) \leq 1, \\ \langle (x)_0 \rangle & \text{if } \text{lh}(x) = 2, \\ 1 + \pi((\text{lh}(x) \dot{-} 1) \dot{-} 1, \pi_0(\pi_1(y \dot{-} 1))), & \text{otherwise.} \end{cases}$$

Let  $x = \text{snoc}(y, z)$ . Show  $\text{beginning}(x) = y$ .**Case**  $\text{lh}(y) = 0$ : Then

$$x = \text{snoc}(y, z) = \langle z \rangle$$

therefore  $\text{lh}(x) = 1$ , and

$$\begin{aligned} \text{beginning}(x) &= \langle \rangle \\ &= y \end{aligned}$$

**Case**  $\text{lh}(y) = 1$ : Then  $y = \langle y' \rangle$  for some  $y'$ ,  $\text{snoc}(y, z) = \langle y', z \rangle$ ,

$$\begin{aligned} \text{beginning}(x) &= \langle (x)_0 \rangle \\ &= \langle (\langle y', z \rangle)_0 \rangle \\ &= \langle y' \rangle \\ &= y \end{aligned}$$

**Case**  $\text{lh}(y) > 1$ : Let  $\text{lh}(y) = n + 2$ ,

$$y = \langle y_0, \dots, y_{n+1} \rangle = 1 + \pi(n + 1, \pi^{n+2}(y_0, \dots, y_{n+1})) .$$

Then

$$\text{snoc}(y, z) = 1 + \pi(n + 2, \pi(\pi_1(y \dot{-} 1), z)) .$$

Therefore

$$\begin{aligned} \text{beginning}(\text{snoc}(y, z)) &= 1 + \pi(((\text{lh}(x) \dot{-} 1) \dot{-} 1), \pi_0(\pi_1(\text{snoc}(y, z) \dot{-} 1))) \\ &= 1 + \pi(n, \pi_0(\pi_1((1 + \pi(n + 2, \pi(\pi_1(y \dot{-} 1), z))) \dot{-} 1))) \\ &= 1 + \pi(n, \pi_0(\pi_1(\pi(n + 2, \pi(\pi_1(y \dot{-} 1), z)))))) \\ &= 1 + \pi(n, \pi_0(\pi(\pi_1(y \dot{-} 1), z))) \\ &= 1 + \pi(n, \pi_1(y \dot{-} 1)) \\ &= 1 + \pi(n, \pi_1((1 + \pi(n + 1, \pi^{n+2}(y_0, \dots, y_{n+1}))) \dot{-} 1)) \\ &= 1 + \pi(n, \pi_1(\pi(n + 1, \pi^{n+2}(y_0, \dots, y_{n+1})))) \\ &= 1 + \pi(n, \pi^{n+2}(y_0, \dots, y_{n+1})) \\ &= y . \end{aligned}$$

**Proof for last:**

Define

$$\text{last}(x) := (x)_{\text{lh}(x) \dot{-} 1}$$

If  $y = \langle y_0, \dots, y_{n-1} \rangle$ , then

$$\begin{aligned} \text{last}(\text{snoc}(y, z)) &= \text{last}(\langle y_0, \dots, y_{n-1}, z \rangle) \\ &= (\langle y_0, \dots, y_{n-1}, z \rangle)_{\text{lh}(\langle y_0, \dots, y_{n-1}, z \rangle) \dot{-} 1} \\ &= (\langle y_0, \dots, y_{n-1}, z \rangle)_n \\ &= z . \end{aligned}$$

**Lemma 5.5** *The primitive-recursive functions are closed under course-of-values recursion:*

*If  $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$  is primitive-recursive, then  $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  is primitive-recursive as well, where  $f$  is defined by*

$$f(\vec{x}, k) = g(\vec{x}, k, \langle f(\vec{x}, 0), f(\vec{x}, 1), \dots, f(\vec{x}, k-1) \rangle) .$$

**Remark:** Informally, this means: If we can define  $f(\vec{x}, y)$  by an expression which uses previously defined primitive-recursive functions, constants,  $\vec{x}$ ,  $y$  and any  $f(\vec{x}, z)$  s.t.  $z < y$ , then  $f$  is primitive-recursive – if we have such an expression, we can form a function  $g$  as in the lemma above.

**Example:** The Fibonacci numbers are primitive-recursive, i.e. the function  $\text{fib} : \mathbb{N} \rightarrow \mathbb{N}$ , defined by

$$\begin{aligned} \text{fib}(0) &:= 1 , \\ \text{fib}(1) &:= 1 , \\ \text{fib}(n) &:= \text{fib}(n-1) + \text{fib}(n-2), \text{ if } n > 1, \end{aligned}$$

is primitive-recursive, as shown by course-of-values recursion. (This is the inefficient implementation, the more efficient solution can be seen directly to be primitive-recursive).

**Proof** that primitive-recursive functions are closed under course-of-values recursion:

Let

$$f(\vec{x}, y) := g(\vec{x}, y, \langle f(\vec{x}, 0), f(\vec{x}, 1), \dots, f(\vec{x}, y-1) \rangle)$$

Show  $f$  is prim. rec.

Define  $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ ,

$$h(\vec{x}, y) := \langle f(\vec{x}, 0), f(\vec{x}, 1), \dots, f(\vec{x}, y-1) \rangle .$$

(Especially,  $h(\vec{x}, 0) = \langle \cdot \rangle$ .)

It follows

$$\begin{aligned} h(\vec{x}, 0) &= \langle \cdot \rangle , \\ h(\vec{x}, y+1) &= \langle f(\vec{x}, 0), f(\vec{x}, 1), \dots, f(\vec{x}, y-1), f(\vec{x}, y) \rangle \\ &= \text{snoc}(\underbrace{\langle f(\vec{x}, 0), f(\vec{x}, 1), \dots, f(\vec{x}, y-1) \rangle}_{=h(\vec{x}, y)}, f(\vec{x}, y)) \\ &= \text{snoc}(h(\vec{x}, y), g(\vec{x}, y, \underbrace{\langle f(\vec{x}, 0), f(\vec{x}, 1), \dots, f(\vec{x}, y-1) \rangle}_{=h(\vec{x}, y)})) \\ &= \text{snoc}(\text{snoc}(h(\vec{x}, y), g(\vec{x}, y, h(\vec{x}, y)))) . \end{aligned}$$

Therefore  $h$  is primitive-recursive.

Now we have that

$$\begin{aligned} f(\vec{x}, y) &= (\langle f(\vec{x}, 0), \dots, f(\vec{x}, y) \rangle)_{y+1} \\ &= (h(\vec{x}, y+1))_{y+1} \end{aligned}$$

is primitive recursive.

We show that functions, which manipulate codes for sequences of natural numbers, are primitive-recursive.

**Lemma and Definition 5.6** *There exists primitive-recursive functions with the following properties:*

(a) A function  $\text{append} : \mathbb{N}^2 \rightarrow \mathbb{N}$  s.t.

$$\text{append}(\langle n_0, \dots, n_{k-1} \rangle, \langle m_0, \dots, m_{l-1} \rangle) = \langle n_0, \dots, n_{k-1}, m_0, \dots, m_{l-1} \rangle .$$

We write

$$n * m$$

for

$$\text{append}(n, m) .$$

(b) A function  $\text{subst} : \mathbb{N}^3 \rightarrow \mathbb{N}$  s.t. if  $i < n$  then

$$\text{subst}(\langle x_0, \dots, x_{n-1} \rangle, i, y) = \langle x_0, \dots, x_{i-2}, y, x_i, x_{i+1}, \dots, x_{n-1} \rangle ,$$

and if  $i \geq n$ , then

$$\text{subst}(\langle x_0, \dots, x_{n-1} \rangle, i, y) = \langle x_0, \dots, x_{n-1} \rangle .$$

This means that  $\text{subst}(x, i, y)$  will be the result of substituting in the sequence  $x$  the  $i$ th element by  $y$ , if such an element exist – if not,  $x$  remains unchanged.

We write  $x[i/y]$  for  $\text{subst}(x, i, y)$ .

(c) A function  $\text{substring} : \mathbb{N}^3 \rightarrow \mathbb{N}$  s.t., if  $i < n$ ,

$$\text{substring}(\langle x_0, \dots, x_{n-1} \rangle, i, j) = \langle x_i, x_{i+1}, \dots, x_{\min(j-1, n-1)} \rangle ,$$

and if  $i \geq n$ ,

$$\text{substring}(\langle x_0, \dots, x_{n-1} \rangle, i, j) = \langle \rangle .$$

(d) A function  $\text{half} : \mathbb{N} \rightarrow \mathbb{N}$ , s.t.  $\text{half}(n) = k$  if  $n = 2k$  or  $n = 2k + 1$ .

(e) The function  $\text{bin} : \mathbb{N} \rightarrow \mathbb{N}$ , s.t.  $\text{bin}(n) = \langle b_0, \dots, b_k \rangle$ , for  $b_i$  in normal form (no leading zeros, unless  $n = 0$ ), s.t.  $n = (b_0, \dots, b_k)_2$

(f) A function  $\text{bin}^{-1} : \mathbb{N} \rightarrow \mathbb{N}$ , s.t.  $\text{bin}^{-1}(\langle b_0, \dots, b_k \rangle) = n$ , if  $(b_0, \dots, b_k)_2 = n$ .

**Proof:**

(a) We have

$$\begin{aligned} \text{append}(\langle x_0, \dots, x_n \rangle, 0) &= \text{append}(\langle x_0, \dots, x_n \rangle, \langle \rangle) \\ &= \langle x_0, \dots, x_n \rangle , \\ &\text{and for } m > 0 \\ \text{append}(\langle x_0, \dots, x_n \rangle, \langle y_0, \dots, y_m \rangle) &= \langle x_0, \dots, x_n, y_0, \dots, y_m \rangle \\ &= \text{snoc}(\langle x_0, \dots, x_n, y_0, \dots, y_{m-1} \rangle, y_m) \\ &= \text{snoc}(\text{append}(\langle x_0, \dots, x_n \rangle, \langle y_0, \dots, y_{m-1} \rangle), y_m) \\ &= \text{snoc}(\text{append}(\langle x_0, \dots, x_n \rangle, \text{beginning}(\langle y_0, \dots, y_m \rangle)), \text{last}(\langle y_0, \dots, y_m \rangle)) . \end{aligned}$$

Therefore we have

$$\begin{aligned} \text{append}(x, 0) &= x , \\ \text{append}(x, y) &= \text{snoc}(\text{append}(x, \text{beginning}(y)), \text{last}(y)) , \end{aligned}$$

One can see that  $\text{beginning}(x) < x$  for  $x > 0$ , therefore the last equations give a definition of  $\text{append}$  by course-of-values recursion, therefore  $\text{append}$  is primitive-recursive.

(b) We have

$$\text{subst}(x, i, y) := \begin{cases} x, & \text{if } \text{lh}(x) \leq i, \\ \text{snoc}(\text{beginning}(x), y), & \text{if } i + 1 = \text{lh}(x), \\ \text{snoc}(\text{subst}(\text{beginning}(x), i, y), \text{last}(x)) & \text{if } i + 1 < \text{lh}(x). \end{cases}$$

Therefore `subst` is definable by course-of-values recursion.

(c) We can define

$$\text{substring}(x, i, j) = \begin{cases} \langle \rangle, & \text{if } i \geq \text{lh}(x), \\ \text{substring}(\text{beginning}(x), i, j), & \text{if } i < \text{lh}(x) \text{ and } j < \text{lh}(x), \\ \text{snoc}(\text{substring}(\text{beginning}(x), i, j), \text{last}(x)) & \text{if } i < \text{lh}(x) \leq j, \end{cases}$$

which is a definition by course-of-values recursion.

(d)  $\text{half}(x) = \mu y < x. (2 \cdot y = x \vee 2 \cdot y + 1 = x)$ .

(e)

$$\text{bin}(x) = \begin{cases} \langle 0 \rangle, & \text{if } x = 0, \\ \langle 1 \rangle & \text{if } x = 1, \\ \text{snoc}(\text{half}(x), x \dot{-} (2 \cdot \text{half}(x))), & \text{if } x > 1. \end{cases}$$

therefore definable by course-of-values recursion.

(f)

$$\text{bin}^{-1}(x) = \begin{cases} 0, & \text{if } \text{lh}(x) = 0, \\ (x)_0 & \text{if } \text{lh}(x) = 1, \\ \text{bin}^{-1}(\text{beginning}(x)) \cdot 2 + \text{last}(x) & \text{if } \text{lh}(x) > 1, \end{cases}$$

therefore definable by course-of-values recursion.

### 5.3 Not All Computable Functions are Primitive-Recursive

All primitive-recursive functions are computable. However, there are computable functions which are not primitive-recursive. One such function is the Ackermann-function:

**Definition 5.7** *The Ackermann function  $\text{Ack} : \mathbb{N}^2 \rightarrow \mathbb{N}$  is defined as  $\text{Ack}(n, m) := \text{Ack}_n(m)$ . Therefore  $\text{Ack}$  has the following recursion equations:*

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

**Lemma 5.8** *For each  $n, m$ , the following holds:*

- (a)  $\text{Ack}(m, n) > n$ .
- (b)  $\text{Ack}_m$  is strictly monotone, i.e.  $\text{Ack}(m, n + 1) > \text{Ack}(m, n)$ .
- (c)  $\text{Ack}(m + 1, n) > \text{Ack}(m, n)$ .
- (d)  $\text{Ack}(m, \text{Ack}(m, n)) < \text{Ack}(m + 2, n)$ .
- (e)  $\text{Ack}(m, 2n) < \text{Ack}(m + 2, n)$ .
- (f)  $\text{Ack}(m, 2^k \cdot n) < \text{Ack}(m + 2k, n)$ .

**Proof:**

(a) Induction on  $m$ .

$m = 0$ :

$$\text{Ack}(0, n) = n + 1 > n.$$

$m \rightarrow m + 1$ : Side-induction on  $n$

$n = 0$ :

$$\text{Ack}(m + 1, 0) = \text{Ack}(m, 1) \stackrel{\text{IH}}{>} 1 > 0.$$

$n \rightarrow n + 1$ :

$$\begin{aligned} \text{Ack}(m + 1, n + 1) &= \text{Ack}(m, \text{Ack}(m + 1, n)) \\ &\stackrel{\text{Main IH}}{>} \text{Ack}(m + 1, n) \\ &\stackrel{\text{Side IH}}{>} n, \end{aligned}$$

therefore

$$\text{Ack}(m + 1, n + 1) > n + 1 .$$

(b) Case  $m = 0$ :

$$\text{Ack}(0, n + 1) = n + 2 > n + 1 = \text{Ack}(0, n) .$$

Case  $m = m' + 1$ :

$$\begin{aligned} \text{Ack}(m' + 1, n + 1) &= \text{Ack}(m', \text{Ack}(m' + 1, n)) \\ &\stackrel{(a)}{>} \text{Ack}(m' + 1, n) . \end{aligned}$$

(c) Induction on  $m$ .

$m = 0$ :

$$\text{Ack}(1, n) = n + 2 > n + 1 = \text{Ack}(0, n)$$

$m \rightarrow m + 1$ : Side-induction on  $n$ :

$n = 0$ :

$$\text{Ack}(m + 2, 0) = \text{Ack}(m + 1, 1) \stackrel{(b)}{>} \text{Ack}(m + 1, 0) .$$

$n \rightarrow n + 1$ :

$$\begin{aligned} \text{Ack}(m + 2, n + 1) &= \text{Ack}(m + 1, \text{Ack}(m + 2, n)) \\ &\stackrel{\text{main-IH}}{>} \text{Ack}(m, \text{Ack}(m + 2, n)) \\ &\stackrel{\text{side-IH} + (b)}{>} \text{Ack}(m, \text{Ack}(m + 1, n)) = \text{Ack}(m + 1, n + 1) . \end{aligned}$$

(d) Case  $m = 0$ :

$$\text{Ack}(0, \text{Ack}(0, n)) = n + 2 < 2n + 3 = \text{Ack}(2, n) .$$

Assume now  $m > 0$ . Proof of the assertion by induction on  $n$ :

$n = 0$ :

$$\begin{aligned} \text{Ack}(m + 2, 0) &= \text{Ack}(m + 1, 1) \\ &= \text{Ack}(m, (\text{Ack}(m + 1, 0))) \\ &> \text{Ack}(m, \text{Ack}(m, 0)) . \end{aligned}$$

$n \rightarrow n + 1$ :

$$\begin{aligned} \text{Ack}(m + 2, n + 1) &= \text{Ack}(m + 1, \text{Ack}(m + 2, n)) \\ &\stackrel{\text{IH}, (b)}{>} \text{Ack}(m + 1, \text{Ack}(m, \text{Ack}(m, n))) \\ &\stackrel{(b), (c)}{>} \text{Ack}(m, \text{Ack}(m - 1, \text{Ack}(m, n))) \\ &= \text{Ack}(m, \text{Ack}(m, n + 1)) . \end{aligned}$$

(e) Case  $m = 0$ :

$$\begin{aligned} \text{Ack}(m, 2n) &= \text{Ack}(0, 2n) \\ &= 2n + 1 \\ &< 2n + 3 \\ &= \text{Ack}(2, n) = \text{Ack}(m + 2, n) . \end{aligned}$$

$m = m' + 1$ :

Induction on  $n$ :

$n = 0$ :

$$\text{Ack}(m' + 1, 2n) = \text{Ack}(m' + 1, 0) < \text{Ack}(m' + 3, 0) = \text{Ack}(m' + 3, n) .$$

$n \rightarrow n + 1$ :

$$\begin{aligned} \text{Ack}(m' + 1, 2n + 2) &= \text{Ack}(m', \text{Ack}(m', \text{Ack}(m' + 1, 2n))) \\ &\stackrel{\text{(d)}}{<} \text{Ack}(m' + 2, \text{Ack}(m' + 1, 2n)) \\ &\stackrel{\text{IH}}{<} \text{Ack}(m' + 2, \text{Ack}(m' + 3, n)) \\ &= \text{Ack}(m' + 3, n + 1) \end{aligned}$$

(f) Induction on  $k$ :

$k = 0$ : trivial.

$k \rightarrow k + 1$ :

$$\begin{aligned} \text{Ack}(m, 2^{k+1} \cdot n) &= \text{Ack}(m, 2 \cdot 2^k \cdot n) \\ &\stackrel{\text{(e)}}{<} \text{Ack}(m + 2, 2^k \cdot n) \\ &\stackrel{\text{IH}}{<} \text{Ack}(m + 2 + 2k, n) \\ &= \text{Ack}(m + 2(k + 1), n) . \end{aligned}$$

**Lemma 5.9** *Every primitive-recursive function  $f : \mathbb{N}^n \rightarrow \mathbb{N}$  can be majorised by one branch of the Ackermann function, i.e. there exists an  $N$  s.t.*

$$f(x_0, \dots, x_{n-1}) < \text{Ack}_N(x_0 + \dots + x_{n-1})$$

for all  $x_0, \dots, x_{n-1} \in \mathbb{N}$ .

*Especially, if  $f : \mathbb{N} \rightarrow \mathbb{N}$  is primitive-recursive, then there exists an  $N$  s.t.*

$$\forall x \in \mathbb{N}. f(x) < \text{Ack}_N(x)$$

**Proof:**

We write  $\sum(\vec{x})$  for  $x_0 + \dots + x_{n-1}$ , if  $\vec{x} = x_0, \dots, x_{n-1}$ .

We proof the assertion by induction on the definition of primitive-recursive functions.

**Basic functions:**

zero:

$$\text{zero}(x) = 0 < x + 1 = \text{Ack}_0(x) .$$

succ:

$$\text{succ}(x) = \text{Ack}(0, x) < \text{Ack}(1, x) = \text{Ack}_1(x) .$$

$\text{proj}_i^n$ :

$$\begin{aligned} \text{proj}(x_0, \dots, x_{n-1}) &= x_i \\ &< x_0 + \dots + x_{n-1} + 1 \\ &= \text{Ack}_0(x_0 + \dots + x_{n-1}) . \end{aligned}$$

**Composition:** Assume assertion holds for  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  and  $g_i : \mathbb{N}^n \rightarrow \mathbb{N}$ . Show assertion for  $h := f \circ (g_0, \dots, g_{k-1})$ .

Assume

$$f(\vec{y}) < \text{Ack}_l(\sum(\vec{y})) ,$$

and

$$g_i(\vec{x}) < \text{Ack}_{m_i}(\sum(\vec{x})) .$$

Let  $N := \max\{l, m_0, \dots, m_{k-1}\}$ . By 5.8 (c) it follows

$$f(\vec{y}) < \text{Ack}_N(\sum(\vec{y})) ,$$

$$g_i(\vec{x}) < \text{Ack}_N(\sum(\vec{x})) .$$

Then, with  $M$  s.t.  $l < 2^M$ , we have

$$\begin{aligned} h(\vec{x}) &= f(g_0(\vec{x}), \dots, g_{k-1}(\vec{x})) \\ &< \text{Ack}_N(g_0(\vec{x}) + \dots + g_{k-1}(\vec{x})) \\ &\stackrel{(b)}{<} \text{Ack}_N(\text{Ack}_N(\sum(\vec{x})) + \dots + \text{Ack}_N(\sum(\vec{x}))) \\ &= \text{Ack}_N(\text{Ack}_N(\sum(\vec{x})) \cdot k) \\ &< \text{Ack}_N(\text{Ack}_N(\sum(\vec{x})) \cdot 2^M) \\ &< \text{Ack}_{N+2M}(\text{Ack}_N(\sum(\vec{x}))) \\ &\leq \text{Ack}_{N+2M}(\text{Ack}_{N+2M}(\sum(\vec{x}))) \\ &< \text{Ack}_{N+2M+2}(\sum(\vec{x})) . \end{aligned}$$

**Primitive recursion,  $n > 1$ :** Assume assertion holds for  $f : \mathbb{N}^n \rightarrow \mathbb{N}$  and  $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ . Show assertion for  $h := \text{primrec}(f, g) : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ .

Assume

$$\begin{aligned} f(\vec{x}) &< \text{Ack}_l(\sum(\vec{x})) , \\ g(\vec{x}, y, z) &< \text{Ack}_r(\sum(\vec{x}) + y + z) . \end{aligned}$$

Let  $N := \max\{l, r\}$ ,  $k < 2^M$ . Then

$$\begin{aligned} f(\vec{x}) &< \text{Ack}_N(\sum(\vec{x})) , \\ g(\vec{x}, y, z) &< \text{Ack}_N(\sum(\vec{x}) + y + z) . \end{aligned}$$

We show

$$h(\vec{x}, y) < \text{Ack}_{N+3}(\sum(\vec{x}) + y)$$

by induction on  $y$ :  $y = 0$ :

$$\begin{aligned}
h(\vec{x}, 0) &= g(\vec{x}) \\
&< \text{Ack}_N(\sum(\vec{x})) \\
&< \text{Ack}_{N+3}(\sum(\vec{x}) + 0) .
\end{aligned}$$

$y \rightarrow y + 1$ :

$$\begin{aligned}
h(\vec{x}, y + 1) &= g(\vec{x}, y, h(\vec{x}, y)) \\
&< \text{Ack}_N(\sum(\vec{x}) + y + h(\vec{x}, y)) \\
&\stackrel{\text{IH}}{<} \text{Ack}_N(\sum(\vec{x}) + y + \text{Ack}_{N+3}(\sum(\vec{x}) + y)) \\
&< \text{Ack}_N(\text{Ack}_{N+3}(\sum(\vec{x}) + y) + \text{Ack}_{N+3}(\sum(\vec{x}) + y)) \\
&= \text{Ack}_N(2 \cdot \text{Ack}_{N+3}(\sum(\vec{x}) + y)) \\
&< \text{Ack}_{N+2}(\text{Ack}_{N+3}(\sum(\vec{x}) + y)) \\
&= \text{Ack}_{N+3}(\sum(\vec{x}) + y + 1)
\end{aligned}$$

**Primitive recursion  $n = 0$ :** Assume  $l \in \mathbb{N}$ ,  $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ . Show assertion for  $h := \text{primrec}(l, g) : \mathbb{N}^1 \rightarrow \mathbb{N}$ .

Define

$$\begin{aligned}
f' &: \mathbb{N} \rightarrow \mathbb{N}, & f'(x) &= l , \\
g' &: \mathbb{N}^3 \rightarrow \mathbb{N}, & g'(x, y, z) &= g(y, z) , \\
h' &: \mathbb{N}^2 \rightarrow \mathbb{N}, & h &:= \text{primrec}(f', g') .
\end{aligned}$$

Using the constructions already shown and IH it follows that

$$h'(x, y) < \text{Ack}_N(x + y)$$

for some  $N$ .

Therefore

$$h(y) = h'(0, y) < \text{Ack}_N(y) .$$

**Lemma 5.10** *The Ackermann function is not primitive-recursive.*

**Proof:** Assume Ack were primitive-recursive. Then  $f : \mathbb{N} \rightarrow \mathbb{N}$ ,  $f(n) := \text{Ack}(n, n)$  would be primitive-recursive as well. Then there exists an  $N \in \mathbb{N}$ , s.t.  $f(n) < \text{Ack}(N, n)$  for all  $n$ , especially

$$\text{Ack}(N, N) = f(N) < \text{Ack}(N, N) ,$$

a contradiction.

**Remark:** We can show more directly that there are non-primitive-recursive computable functions: Assume all computable functions are primitive-recursive.

Define  $h : \mathbb{N}^2 \rightarrow \mathbb{N}$  as follows:

$$h(e, n) = \begin{cases} f(n), & \text{if } e \text{ encodes a string in ASCII} \\ & \text{which is a term denoting a unary} \\ & \text{primitive-recursive function } f, \\ 0, & \text{otherwise.} \end{cases}$$

So  $h(e, n)$  computes, if  $e$  is a code for a primitive-recursive function  $f$ , the result of applying  $f$  to  $n$ . For other  $e$ , the result of  $h(e, n)$  doesn't matter as long as it is defined, we have set it to 0.

$h$  can be considered as an *interpreter* for the primitive-recursive functions.

So if  $e$  is a code for  $f$ , then  $\forall n. f(n) = h(e, n)$ ,  $f = \lambda n. h(e, n)$ . Therefore for each primitive-recursive function  $f : \mathbb{N} \rightarrow \mathbb{N}$  there exists an  $e$  s.t.  $f = \lambda n. h(e, n)$  (choose  $e$  to be the code for  $n$ ).

$h$  is computable, since the defining laws for primitive-recursive functions give us a way of computing  $h(e, n)$ . Using the assumption, that all computable recursive functions are primitive-recursive, it follows that  $h$  is primitive-recursive.

Define

$$f : \mathbb{N} \rightarrow \mathbb{N}, \quad f(n) := h(n, n) + 1 .$$

$f$  is primitive-recursive, since  $h$  is primitive-recursive. But  $f$  is chosen in such a way that it cannot be of the form  $\lambda n. h(e, n)$ . This is violated at input  $e$ :

$$f(e) = h(e, e) + 1 \neq h(e, e) = (\lambda n. h(e, n))(e) .$$

But since  $f$  is primitive-recursive, there exists a code  $e$  for  $f$  and therefore  $f = \lambda e. h(e, n)$  for some  $e$ , which cannot be by the above, so we get a contradiction. This completes the proof.

A **short version of the proof** reads as follows:

Assume all computable functions were primitive-recursive. Define  $h$  as above.  $h$  is computable, therefore primitive-recursive. Define  $f$  as above, and let  $e$  be a code for  $f$ . Then we have

$$f(n) = h(e, n)$$

for all  $n$ . But then

$$h(e, e) = f(e) = h(e, e) + 1 ,$$

a contradiction.

**Remark 2:** The above proof can be used in other contexts as well. It shows as well that there is no programming language, which computes all computable functions and such that all functions definable in this language are total. If we had such a language, then we could define codes  $e$  for programs in this language and therefore we could define  $h : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,

$$h(e, n) = \begin{cases} f(n), & \text{if } e \text{ is a program in this language} \\ & \text{for a unary function } f, \\ 0, & \text{otherwise.} \end{cases}$$

If  $f$  is a unary function, computable in this language, then  $f = \lambda n. h(e, n)$  for the code  $e$  for the program of  $f$ . Now define as above  $f(n) = h(n, n) + 1$ .  $h$  is computable, therefore as well  $f$ , therefore  $f$  is computable in this language. Let  $f = \lambda n. h(e, n)$ . Then we obtain

$$h(e, e) + 1 = f(e) = (\lambda n. h(e, n))(e) = h(e, e) ,$$

a contradiction.

**Remark 3:** The above proof shows as well for instance that if we extend the primitive-recursive functions by adding the Ackermann function as basic function, or any other functions, we still won't obtain all computable functions – the above argument can be used for for such sets of functions in just the same way as for the set of primitive-recursive functions.

## 5.4 The Partial Recursive Functions

The primitive-recursive functions will allow to define functions which

- compute the result of running  $n$  steps of a URM or TM,

- check whether a URM or TM has stopped,
- obtain, depending on  $n$  arguments the initial configuration for computing  $P^{(n)}$  for a URM  $P$  or  $M^{(n)}$  for a TM  $M$ ,
- extract from a configuration of  $P$  and  $M$ , in which this machine has stopped, the result obtained by  $P^{(n)}$ ,  $M^{(n)}$ .

We will show the existence of such functions for TMs in Subsection 5.6. However, TMs and URMs do not allow to compute an  $n$ , s.t. after  $n$  steps the URM or TM has stopped. In order to obtain such an  $n$ , we will extend the set of primitive-recursive function by the principle of being closed as well under  $\mu$ . The resulting set will be called the set of partial recursive functions.

Using the  $\mu$ -operator, we will be able to find the first  $n$  s.t. a TM or URM stops. Together with the above we will then be able to show that all TM- and URM-computable functions are partial recursive.

Note that  $\mu(f)$  might be partial, even if  $f$  is total, therefore we will necessarily obtain a set of *partial* functions rather than a set of total functions, therefore the name **partial** recursive functions. The recursive functions will be the partial recursive functions, which are total.

**Definition 5.11** We define inductively the set of partial recursive functions  $f$  together with their arity, i.e. together with the  $k$  s.t.  $f : \mathbb{N}^k \rightarrow \mathbb{N}$ .

We write “ $f : \mathbb{N}^k \rightarrow \mathbb{N}$  is partial recursive” for “ $f$  is partial recursive with arity  $k$ ”, and  $\mathbb{N}$  for  $\mathbb{N}^1$ .

- The following basic functions are partial recursive:
  - zero :  $\mathbb{N} \rightarrow \mathbb{N}$ ,
  - succ :  $\mathbb{N} \rightarrow \mathbb{N}$ ,
  - $\text{proj}_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$  ( $0 \leq i \leq k$ ).
- If  $g : \mathbb{N}^k \rightarrow \mathbb{N}$  is partial recursive, and for  $i = 0, \dots, k - 1$  we have  $h_i : \mathbb{N}^n \rightarrow \mathbb{N}$  is partial recursive, then  $g \circ (h_0, \dots, h_{k-1}) : \mathbb{N}^n \rightarrow \mathbb{N}$  is partial recursive as well.  
As before we write  $g \circ h$  instead of  $g \circ (h)$ , and  $g_0 \circ g_1 \circ g_2 \circ \dots \circ g_n$  for  $g_0 \circ (g_1 \circ (g_2 \circ \dots \circ g_n))$ .
- If  $g : \mathbb{N}^n \rightarrow \mathbb{N}$ , and  $h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$  are partial recursive, then  $\text{primrec}(g, h) : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  is partial recursive as well.
- If  $k \in \mathbb{N}$  and  $h : \mathbb{N}^2 \rightarrow \mathbb{N}$  is partial recursive, then  $\text{primrec}(k, h) : \mathbb{N} \rightarrow \mathbb{N}$  is partial recursive as well.
- If  $k \in \mathbb{N}$  and  $g : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$  is partial recursive, then  $\mu(g) : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  is partial recursive as well.

(Remember that  $f := \mu(g)$  has defining equation

$$f(\vec{x}) \simeq \begin{cases} \min\{k \in \mathbb{N} \mid g(\vec{x}, k) \simeq 0 \wedge \forall l < k. g(\vec{x}, l) \downarrow\}, & \text{if such a } k \text{ exists,} \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

**Definition 5.12** (a) A recursive function is a partial recursive function, which is total.

(b) A recursive relation is a relation  $R \subseteq \mathbb{N}^n$  s.t.  $\chi_R$  is recursive.

**Example:** One can show that the Ackermann function is recursive. Note that it is not primitive-recursive.

## 5.5 Closure Properties of the Recursive Functions

- Every primitive-recursive function (relation) is recursive.
- The recursive functions and relations have the same closure properties as those discussed for the primitive-recursive functions and relations in Sect. 5.2, using essentially the same proofs.
- Let for a predicate  $U \subseteq \mathbb{N}^{n+1}$

$$\mu z.U(\vec{n}, z) := \begin{cases} \min\{z \mid U(\vec{n}, z)\}, & \text{if such a } z \text{ exists,} \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

Then we have that, if  $U$  is recursive, so is the function

$$f(\vec{n}) := \mu z.U(\vec{n}, z) .$$

**Proof:**

$$f(\vec{n}) \simeq \mu z.(\chi_{\mathbb{N}^n \setminus U}(\vec{n}, z) \simeq 0) .$$

## 5.6 Equivalence of URM-Computable, TM-Computable and Partial Recursive Functions

**Lemma 5.13** *All partial recursive functions are URM computable.*

**Proof:** By Lemma 3.3.

### Turing-computable functions are partial recursive.

We are going to show that TM-computable functions are partial recursive. We will show an even stronger result: We will encode Turing machines  $M$  as natural numbers  $\text{code}(M)$  and show that there exists for every  $n \in \mathbb{N}$  partial recursive functions  $f_n : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  s.t. for every TM  $M$  we have

$$\forall \vec{m} \in \mathbb{N}^n . f_n(\text{code}(M), \vec{m}) \simeq M^{(n)}(\vec{m}) .$$

These functions  $f_n$  will be called universal partial recursive functions. The functions  $f_n$  are essentially interpreters for Turing machines –  $f_n$  essentially evaluates (simulates) a Turing machine (given by its code  $e$ ) applied to arguments  $\vec{m}$ .

We will later write  $\{e\}^n(\vec{m})$  for  $f_n(e, \vec{m})$ . The brackets in  $\{e\}^n$  are called “Kleene-brackets”, since Kleene introduced this notation.

### Encoding of Turing Machines as Natural Numbers.

We will encode TMs using the following steps:

- In general we use sometimes Gödel-brackets in order to denote the code of some object. E.g. one writes  $\lceil x \rceil$  for the code of  $x$ ,  $\lceil y \rceil$  for the code of  $y$ .
- We assume some encoding  $\lceil x \rceil$  for the symbols  $x$  of the alphabet  $\Sigma$  as natural numbers, s.t.
  - $\lceil 0 \rceil = 0$ ,
  - $\lceil 1 \rceil = 1$ ,
  - $\lceil \perp \rceil = 2$ .
- We assume that each states  $q$  is encoded as natural number  $\lceil q \rceil$ , where we have  $\lceil q_0 \rceil = 0$  for the initial state  $q_0$  of the TM.
- We encode the directions in the instructions by

- $\lceil L \rceil = 0$ ,
- $\lceil R \rceil = 1$ .
- We assume that the alphabet consists of  $\{0, 1, \sqcup\}$  and those symbols mentioned in the instructions. Any other symbols of the TM will never occur during a run of the TM, therefore omitting those symbols doesn't change the behaviour of the TM. Therefore we don't need to write down the alphabet explicitly.
- Similarly, we can assume that the set of states consists of  $q_0$  and all states mentioned in the instructions, therefore we don't need to write down the set of states explicitly. Again, omitting any other states won't change the behaviour of the TM.
- Since we assumed that  $q_0$  is always the initial state with code 0, and that  $\sqcup$  has code 2, we don't need to mention those symbols. Therefore a TM is given by its set of instructions.
- An instruction  $I = (q, a, q', a', D)$ , where  $q, q'$  are states,  $a, a'$  are symbols,  $D \in \{L, R\}$ , will be encoded as
 
$$\text{code}(I) = \pi^5(\lceil q \rceil, \lceil a \rceil, \lceil q' \rceil, \lceil a' \rceil, \lceil D \rceil) .$$
- A set of instructions  $\{I_0, \dots, I_{k-1}\}$  will be encoded as  $\langle \text{code}(I_0), \dots, \text{code}(I_{k-1}) \rangle$ . This number is as well the encoding  $\text{code}(M)$  of the corresponding TM.

### Encoding of the Configuration of a TM as a Natural number.

The configuration of a TM can be given by:

- The code  $\lceil q \rceil$  of its state  $q$ .
- A segment (i.e. a consecutive sequence of cells) of the tape, which includes the cell, the head is pointing to, and all cells which are not blank. (Note that during a run of a TM, only finitely many cells contain a symbol which is non-blank: initially this is the case and in finitely many steps only finitely many cells are changed from blank to non-blank. Therefore there exists always a segment as just stated.) A segment  $a_0, \dots, a_{n-1}$  is encoded as  $\text{code}(a_0, \dots, a_{n-1}) := \langle \lceil a_0 \rceil, \dots, \lceil a_{n-1} \rceil \rangle$ .
- The position of the head on this tape. This can be given as a number  $i$  s.t.  $0 \leq i < n$ , if the segment represented is  $a_0, \dots, a_{n-1}$ .

A configuration of a TM, given by a state  $q$ , a segment  $a_0, \dots, a_{n-1}$  and a head position  $i$ , will be encoded as  $\pi^3(\lceil q \rceil, \text{code}(a_0, \dots, a_{n-1}), i)$ .

Note that the segment represented is not uniquely defined, but that doesn't cause any problems. We only require that the functions generate and use arbitrary codes for configurations.

### Primitive-Recursive Functions Simulating the Steps of a Turing Machine

We will now introduce primitive-recursive function, as outlined in the beginning of Section 5.4, which create the initial configuration of a TM, make one step of the TM, check whether the TM has stopped and extract the result from its configuration.

- We define primitive-recursive functions  $\text{symbol}, \text{state} : \mathbb{N} \rightarrow \mathbb{N}$ , which extract the symbol at the head and the state of the TM from the current configuration:

$$\begin{aligned} \text{symbol}(a) &= (\pi_1^3(a))_{\pi_2^3(a)} \\ \text{state}(a) &= \pi_0^3(a) \end{aligned}$$

Note that  $a = \pi^3(\lceil q \rceil, \langle \lceil a_0 \rceil, \dots, \lceil a_{n-1} \rceil \rangle, i)$ .

- We define a primitive-recursive function  $\text{lookup} : \mathbb{N}^3 \rightarrow \mathbb{N}$ , s.t. if  $e$  is the code for a TM,  $q$  state,  $a$  a symbol, then  $\text{lookup}(e, q, a)$  is a number  $\pi^3(\lceil q' \rceil, \lceil a' \rceil, \lceil D \rceil)$  for the first instruction of the TM  $e$  of the form  $(q, a, q', a', D)$ , if it exists. If no such instruction exists, the result will be  $0 (= \pi^3(0, 0, 0))$ .

$\text{lookup}$  is defined as follows:

- First find using bounded search the index of the first instruction starting with  $\lceil q \rceil, \lceil a' \rceil$ .
- Then extract the corresponding values from this instruction.

Formally the definition is as follows:

- Define an auxiliary primitive-recursive function  $g : \mathbb{N}^3 \rightarrow \mathbb{N}$ ,

$$g(e, q, a) = \mu i \leq \text{lh}(e) \cdot \pi_0^5((e)_i) = q \wedge \pi_1^5((e)_i) = a ,$$

which finds the index of the first instruction starting with  $q, a$ .

- Now define

$$\text{lookup}(e, q, a) := \pi^3(\pi_2^5((e)_{g(e, q, a)}), \pi_3^5((e)_{g(e, q, a)}), \pi_4^5((e)_{g(e, q, a)})) .$$

- There exists a primitive-recursive relation  $\text{hasinstruction} \subseteq \mathbb{N}^3$ , s.t.  $\text{hasinstruction}(e, \lceil q \rceil, \lceil a \rceil)$  holds, iff the TM corresponding to  $e$  has a instruction  $(q, a, q', a', D)$  for some  $q', a', D$ .

- Defined, using the function  $g$  from the previous item, as

$$\chi_{\text{hasinstruction}}(e, q, a) = \text{sig}(\text{lh}(e) \dot{-} g(e, q, a)) .$$

If there is such an instruction,  $g(e, q, a) < \text{lh}(e)$ , therefore  $\text{lh}(e) \dot{-} g(e, q, a) > 0$ . Otherwise  $g(e, q, a) \geq \text{lh}(e)$ ,  $\text{lh}(e) \dot{-} g(e, q, a) = 0$ .

- There exists a primitive-recursive function  $\text{next} : \mathbb{N}^2 \rightarrow \mathbb{N}$  s.t., if  $e$  encodes a TM and  $c$  is a code for a configuration, then  $\text{next}(e, c)$  is a code for the configuration after one step of the TM is executed, or equal to  $c$ , if the TM has halted.

Informal description of  $\text{next}$ :

- Assume the configuration is  $c = \pi^3(q, \langle a_0, \dots, a_{n-1} \rangle, i)$ .
- Check using  $\text{hasinstruction}$ , whether the TM has stopped. If it has stopped, return  $c$ .
- Otherwise, use  $\text{lookup}$  to obtain the codes for the next state  $q'$ , next symbol  $a'$  and direction  $D$ .
- Replace in  $\langle a_0, \dots, a_{n-1} \rangle$ , the  $i$ th element by  $a'$ . Let the result be  $x$ .
- It might be that the head in the next step will leave the current segment. Then we have to extend the segment by one blank to the left or right:
  - \* If the direction is  $D = \lceil L \rceil$  and  $i = 0$ , we have to extend the segment to the left by one blank. So the result of  $\text{next}$  is

$$\pi^3(q', \langle \lceil \_ \rceil \rangle * x, 0) .$$

- \* If the direction is  $D \neq \lceil L \rceil$  and  $i \geq n - 1$ , we have to extend the segment to the right by one blank. Then the result of  $\text{next}$  is

$$\pi^3(q', x * \langle \lceil \_ \rceil \rangle, n) .$$

- \* Otherwise let  $i' = i - 1$ , if  $D = \lceil L \rceil$ , and  $i' = i + 1$ , if  $D \neq \lceil L \rceil$ . Then the result of next is

$$\pi^3(q', x, i') .$$

The above can easily be formalised as a primitive recursive function, using the previously defined functions, list operations (especially substitution and concatenation of lists) and case distinction.

- There exists a primitive-recursive predicate  $\text{terminate} \subseteq \mathbb{N}^2$ , s.t.  $\text{terminate}(e, c)$  holds, iff the TM  $e$  with configuration  $c$  stops.

$$\text{terminate}(e, c) :\Leftrightarrow \neg \text{hasinstruction}(e, \text{state}(c), \text{symbol}(c)) .$$

- There exists a primitive-recursive function  $\text{iterate} : \mathbb{N}^3 \rightarrow \mathbb{N}$  s.t.  $\text{iterate}(e, c, n)$  is the result of iterating TM  $e$  with initial configuration  $c$  for  $n$  steps.

The definition is as follows

$$\begin{aligned} \text{iterate}(e, c, 0) &= c , \\ \text{iterate}(e, c, n + 1) &= \text{next}(e, \text{iterate}(e, c, n)) . \end{aligned}$$

- There exists a primitive-recursive function  $\text{init}^n : \mathbb{N}^n \rightarrow \mathbb{N}$ , s.t.  $\text{init}^n(\vec{m})$  is the initial configuration for computing  $M^{(n)}(\vec{m})$  for of any TM  $M$  containing alphabet  $\{0, 1, \lceil \_ \rceil\}$ :

$$\text{init}^n(m_0, \dots, m_{n-1}) = \pi^3(\lceil q_0 \rceil, \text{bin}(m_0) * \langle \lceil \_ \rceil \rangle * \text{bin}(m_1) * \langle \lceil \_ \rceil \rangle * \dots * \langle \lceil \_ \rceil \rangle * \text{bin}(m_{n-1}), 0) .$$

- There exists a primitive-recursive function  $\text{extract} : \mathbb{N} \rightarrow \mathbb{N}$ , s.t. if  $c$  is a configuration in which the TM stops,  $\text{extract}(c)$  is the natural number corresponding to the result returned by the TM.

- Assume  $c = \pi^3(q, x, i)$ .
- We need to find first the largest subsequence of  $x$  starting at  $i$  consisting of zeros and ones only: This is  $j$  defined as

$$j = \mu z < \text{lh}(x).z \geq i \wedge (x)_z \neq 0 \wedge (x)_z \neq 1 .$$

Now

$$\text{extract}(c) = \text{bin}^{-1}(\text{substring}(x, i, j)) .$$

- There exist primitive-recursive predicates  $\mathbb{T}^n \subseteq \mathbb{N}^{n+2}$  s.t., if  $e$  encodes a TM  $M$ , then  $\mathbb{T}^n(e, m_0, \dots, m_{n-1}, k)$  holds, iff  $M$ , started with the initial configuration corresponding to a computation of  $M^{(n)}(m_0, \dots, m_{n-1})$ , stops after  $\pi_0(k)$  steps and has final configuration  $\pi_1(k)$ .

$$\mathbb{T}^n(e, \vec{m}, k) :\Leftrightarrow \text{terminate}(e, \text{iterate}(e, \text{init}^n(\vec{m}), \pi_0(k))) \wedge \text{iterate}(e, \text{init}^n(\vec{m}), \pi_0(k)) = \pi_1(k) .$$

- There exists a primitive-recursive function  $\mathbb{U} : \mathbb{N} \rightarrow \mathbb{N}$  s.t. , if  $M$  is a TM with  $\text{code}(M) = e$ , and  $\mathbb{T}^n(e, \vec{m}, k)$  holds, then  $\mathbb{U}(k)$  is the result of  $M^{(n)}(\vec{m})$ .

$$\mathbb{U}(k) = \text{extract}(\pi_1(k)) .$$

- Now it follows that, if  $e$  encodes a TM  $M$ , then

$$M^{(n)}(\vec{m}) \simeq \mathbb{U}(\mu z. \mathbb{T}^n(e, \vec{m}, z)) .$$

So we have shown the following theorem:

**Theorem 5.14 (Kleene's Normal Form Theorem)**

(a) There exists partial recursive functions  $f_n : \mathbb{N}^{n+1} \xrightarrow{\sim} \mathbb{N}$  s.t. if  $e$  is the code of TM  $M$  then

$$f_n(e, \vec{n}) \simeq M^n(\vec{n}) .$$

(b) There exist a primitive-recursive function  $U : \mathbb{N} \rightarrow \mathbb{N}$  and primitive recursive predicates  $T^n \subseteq \mathbb{N}^{n+2}$  s.t. for the function  $f_n$  introduced in (a) we have

$$f_n(e, \vec{n}) \simeq U(\mu z. T^n(e, \vec{n}, z)) .$$

**Definition 5.15** Let  $U, T^n$  as in Kleene's Normal Form Theorem 5.14. Then we define for  $e \in \mathbb{N}$  the partial recursive function  $\{e\}^n : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$  by

$$\{e\}^n(\vec{m}) \simeq U(\mu y. T^n(e, \vec{m}, y)) .$$

$\{e\}^n(\vec{m})$  is pronounced Kleene-brackets- $e$  in  $n$  arguments applied to  $\vec{m}$ .

**Corollary 5.16**

(a) For every Turing-computable function  $f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$  there exists an  $e \in \mathbb{N}$  s.t.

$$f = \{e\}^n .$$

(b) Especially, all Turing-computable functions are partial recursive.

**Proof:** (a) Immediate. (b) by (a), since  $\{e\}^n$  is partial recursive.

**Theorem 5.17** The sets of URM-computable, of Turing-computable, and of partial recursive functions coincide.

**Proof:**

By Theorem 4.3, every URM computable function is TM-computable. By Corollary 5.16, every TM computable function is partial recursive. By Lemma 5.13, all partial recursive functions are URM computable.

**Remark:**

- By Kleene's Normal Form Theorem every Turing computable function  $g : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$  is of the form  $\{e\}^n$ . Since the Turing computable functions are the partial recursive functions, we obtain the following:

The partial recursive functions  $g : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$  are exactly the functions  $\{e\}^n$ .

This means:

- $\{e\}^n$  is partial recursive for every  $e \in \mathbb{N}$ .
- For every partial recursive function  $g : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$  there exists an  $e$  s.t.  $g = \{e\}^n$ .

- Therefore we can say

$$f_n : \mathbb{N}^{n+1} \xrightarrow{\sim} \mathbb{N} , \quad f_n(e, \vec{x}) \simeq \{e\}^n(\vec{x})$$

forms a **universal  $n$ -ary partial recursive function**, since it encodes all  $n$ -ary partial recursive function.

- So we can assign to each partial recursive function  $g$  a number, namely an  $e$  s.t.  $g = \{e\}^n$ .
  - Each number  $e$  denotes one partial recursive function  $\{e\}^n$ .
  - However, several numbers denote the same partial recursive function, i.e. there are  $e, e'$  s.t.  $e \neq e'$  but  $\{e\}^n = \{e'\}^n$ .
 

(**Proof:** There are several algorithms for computing the same function. Therefore there are several Turing machines which compute the same function. These Turing machines have different codes  $e$ ).

**Lemma 5.18** *The set  $F$  of partial recursive functions (and therefore as well of Turing-computable and of URM-computable functions), i.e.*

$$F := \{f : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N} \mid k \in \mathbb{N} \wedge f \text{ partial recursive}\}$$

*is countable.*

**Proof:** Every partial recursive function is of the form  $\{e\}^n$  for some  $e, n \in \mathbb{N}$ . Therefore

$$f : \mathbb{N}^2 \rightarrow F , \quad f(e, n) := \{e\}^n$$

is surjective, therefore as well

$$g : \mathbb{N} \rightarrow F , \quad g(e) := f(\pi_0(e), \pi_1(e)) .$$

Therefore  $F$  is countable.

## 6 The Church-Turing Thesis

We have introduced three models of computations:

- The URM-computable functions.
- The Turing-computable functions.
- The partial recursive functions.

Further we have shown that all three models compute the same partial functions.

Lots of other models of computation have been studied:

- The while programs.
- Symbol manipulation systems by Post and by Markov.
- Equational calculi by Kleene and by Gödel.
- The  $\lambda$ -definable functions.
- Any of the programming languages Pascal, C, C++, Java, Prolog, Haskell, ML (and many more).
- Lots of other models of computation.

One can show that the partial functions computable in these models of computation are again exactly the partial recursive functions. So all these attempts to define a complete model of computation result in the same set of partial recursive functions. Because of this, one states the

**Church-Turing Thesis:** *The (in an intuitive sense) computable partial functions are exactly the partial recursive functions (or equivalently the URM-computable or Turing-computable functions).*

Note that this thesis is **not a mathematical theorem**, but a **philosophical thesis**. Therefore the Church-Turing thesis **cannot be proven**, but we can only provide **philosophical evidence** for it. This evidence comes from the following **considerations and empirical facts**:

- All complete models of computation suggested by researchers (including those mentioned above) define the same set of partial functions.
- Many of these models were carefully designed in order to capture intuitive notions of computability:
  - The Turing machine model captures the intuitive notion of **computation on a piece of paper** in a general sense.
  - The URM machine model captures the general notion of **computability by a computer**.
  - Symbolic manipulation systems capture the general notion of computability by **manipulation of symbolic strings**.
- No intuitively computable partial function, which is not partial recursive, has been found, despite lots of researchers trying it.
- Using metamathematical investigations, a strong intuition has been developed that in principal programs in any programming language can be simulated by Turing machines and URMs, and therefore partial functions computable in such a language are partial recursive.

Because of this, only very few researchers seriously doubt the correctness of the Church-Turing thesis.

**Remark:** Because of the equivalence of the models of computation it follows that the halting problem for any of the above mentioned models of computation is undecidable. Especially it is undecidable, whether a program in one of the programming languages mentioned terminates. If we had a decision procedure which decides whether or not say a Pascal program terminates for given input, then we could, using a translation of URMs into Pascal programs, decide the halting problem for URMs, which is impossible.

## 7 Kleene's Recursion Theorem

The main theorem in this section is Kleene's Recursion Theorem, which expresses that the partial recursive functions are closed under a very general form of recursion. In order to prove this we use the S-m-n theorem, which is an important lemma that is used in many proofs in computability theory.

### 7.1 The S-m-n Theorem

If  $f : \mathbb{N}^{n+m} \rightrightarrows \mathbb{N}$  is a partial recursive, and we fix the first  $m$  arguments (say  $l_0, \dots, l_{m-1}$ ), we obtain a function of  $n$  arguments

$$g : \mathbb{N}^n \rightrightarrows \mathbb{N}, \quad g(x_0, \dots, x_{n-1}) = f(l_0, \dots, l_{m-1}, x_0, \dots, x_{n-1}),$$

which is again partial recursive. The following theorem says that we can compute a Kleene index of  $g$  (i.e. an  $e'$  s.t.  $g = \{e'\}^n$ ) from a Kleene index of  $f$  and  $l_0, \dots, l_{m-1}$  primitive-recursively: There exists a primitive-recursive function  $S_n^m$  s.t., if  $f = \{e\}^{m+n}$ , then  $g = \{S_n^m(e, l_0, \dots, l_{m-1})\}^n$ :

**Theorem 7.1 (S-m-n Theorem).**

For all  $m, n \in \mathbb{N}$  there exists a primitive-recursive function

$$S_n^m : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$$

s.t. for all  $l_0, \dots, l_{m-1}, x_0, \dots, x_{n-1} \in \mathbb{N}$

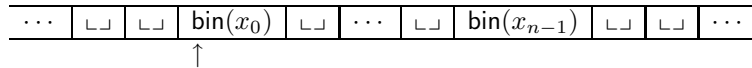
$$\{S_n^m(e, l_0, \dots, l_{m-1})\}^n(x_0, \dots, x_{n-1}) \simeq \{e\}^{m+n}(l_0, \dots, l_{m-1}, x_0, \dots, x_{n-1}).$$

**Proof:** We write  $\vec{x}$  for  $x_0, \dots, x_{n-1}$  and  $\vec{l}$  for  $l_0, \dots, l_{m-1}$ . Let  $M$  be a Turing machine encoded as  $e$ . The Turing machine  $M'$  corresponding to  $S_n^m(e, \vec{l})$  should be such that

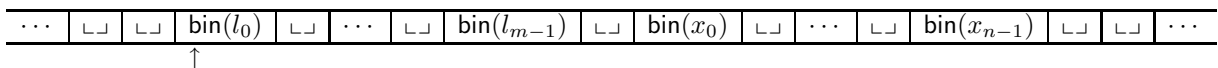
$$M'^n(\vec{x}) \simeq M^{n+m}(\vec{l}, \vec{x}).$$

Such a Turing machine can be defined as follows:

1. The initial configuration is that  $x_0, \dots, x_{m-1}$  are written on the tape and the head is pointing to the left most bit:



2.  $M'$  writes first the binary representation of  $l_0, \dots, l_{m-1}$  separated by blanks in front of this, and terminates this step with the head pointing to the most significant bit of  $\text{bin}(l_0)$ . So we obtain after this first step the following configuration:



3. Then  $M'$  will run  $M$ , starting in this configuration, and will terminate, if  $M$  terminates. The result will be

$$\simeq M^{m+n}(\vec{l}, \vec{x}),$$

and in total we get therefore

$$M'^n(\vec{x}) \simeq M^{m+n}(\vec{l}, \vec{x})$$

as desired.

A code for  $M'$  can be obtained from a code for  $M$  and from  $\vec{l}$  as follows:

- One takes a Turing machine  $M''$ , which writes the binary representations of  $l_0, \dots, l_{m-1}$  in front of its initial position (separated by a blank and with a blank at the end), and terminates at the left most bit. It's a straightforward exercise to write a code for the instructions of such a Turing machine, depending on  $\vec{l}$ , and show that the function defining it is primitive-recursive. Assume, the terminating state of  $M''$  has Gödel number (i.e. code)  $s$ , and that all other states have Gödel numbers  $< s$ .

- Then one appends to the instructions of  $M''$  the instructions of  $M$ , but with the states shifted, so that the new initial state of  $M$  is the final state  $s$  of  $M''$  (i.e. we add  $s$  to all the Gödel numbers of states occurring in  $M$ ). This can be done as well primitive-recursively.

So a code for  $M''$  can be defined primitive-recursively depending on a code  $e$  for  $M$  and  $\vec{l}$ , and  $S_n^m$  is the primitive-recursive function computing this. With this function it follows now that, if  $e$  is a code for a TM, then

$$\{S_n^m(e, \vec{l})\}^n(\vec{x}) \simeq \{e\}^{n+m}(\vec{l}, \vec{x}) .$$

This equation holds, even if  $e$  is not a code for a TM: In this case  $\{e\}^{m+n}$  interprets  $e$  as if it were the code for a valid TM  $M$  (A code for such a valid TM is obtained by

- deleting any instructions  $\text{code}(q, a, q', a', D)$  in  $e$  s.t. there exists an instruction  $\text{code}(q, a, q'', a'', D')$  occurring before it in the sequence  $e$ ,
- and by replacing all directions  $> 1$  by  $[R] = 1$ .)

$e' := S_n^m(e, \vec{l})$  will have the same deficiencies as  $e$ , but when applying the Kleene-brackets, it will be interpreted as a TM  $M'$  obtained from  $e'$  in the same way as we obtained  $M$  from  $e$ , and therefore

$$\{e'\}^n(\vec{x}) \simeq M'^n(\vec{x}) \simeq M^{n+m}(\vec{l}, \vec{x}) \simeq \{e\}^{n+m}(\vec{l}, \vec{x}) .$$

So we obtain the desired result in this case as well.

**Notation:** We will in the following usually omit the superscript  $n$  in  $\{e\}^n(m_0, \dots, m_{n-1})$ , i.e. we will write  $\{e\}(m_0, \dots, m_{n-1})$  instead of  $\{e\}^n(m_0, \dots, m_{n-1})$ . Further  $\{e\}$  not applied to arguments and without superscript means usually  $\{e\}^1$ .

## 7.2 Kleene's Recursion Theorem

**Theorem 7.2 (Kleene's Recursion Theorem).** *Let  $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  be a partial recursive function. Then there exists an  $e \in \mathbb{N}$  s.t.*

$$\{e\}^n(x_0, \dots, x_{n-1}) \simeq f(e, x_0, \dots, x_{n-1}) .$$

Before proving this, we give some examples for the use of this theorem.

### Examples:

1. There exists an  $e$  s.t.

$$\{e\}(x) \simeq e + 1 .$$

For showing this take in the Recursion Theorem  $f(e, n) := e + 1$ . Then we get:

$$\{e\}(x) \simeq f(e, x) \simeq e + 1 .$$

**Note** that it would be rather difficult to find directly such an  $e$  (try it yourself). Such an  $e$  would be a code for a Turing machine, which, independent of its input writes its own code on the tape.

**Remark:** Such applications of the Recursion Theorem are usually not very useful. Usually, when using it, one doesn't use the index  $e$  directly, but only the application of  $\{e\}$  to some arguments.

2. The function `fib` computing the Fibonacci numbers can be seen to be recursive by using Kleene's Recursion theorem. (This is a weaker result than what we obtained above, where we showed that `fib` is even primitive-recursive. However, it is a nice example which demonstrates very well the use of the recursion theorem.)

Remember the defining equations for `fib`:

$$\begin{aligned}\text{fib}(0) &= 1, \\ \text{fib}(1) &= 1, \\ \text{fib}(n+2) &= \text{fib}(n) + \text{fib}(n+1).\end{aligned}$$

From these equations we obtain

$$\text{fib}(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1, \\ \text{fib}(n-2) + \text{fib}(n-1), & \text{otherwise.} \end{cases}$$

We show using the recursion theorem that there exists a recursive function  $g : \mathbb{N} \rightarrow \mathbb{N}$ , which fulfils the same equations, i.e. s.t.

$$g(n) \simeq \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1, \\ g(n-2) + g(n-1), & \text{otherwise.} \end{cases}$$

This can be shown as follows:

Define a recursive  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  s.t.

$$f(e, n) \simeq \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1, \\ \{e\}(n-2) + \{e\}(n-1), & \text{otherwise.} \end{cases}$$

Now let  $e$  be s.t.

$$\{e\}(n) \simeq f(e, n).$$

Then  $e$  fulfills the equations

$$\{e\}(n) \simeq \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1, \\ \{e\}(n-2) + \{e\}(n-1), & \text{otherwise.} \end{cases}$$

Let  $g = \{e\}$ . Then  $g$  fulfills the equations as desired:

$$g(n) \simeq \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1, \\ g(n-2) + g(n-1), & \text{otherwise.} \end{cases}$$

It is not a priori clear that  $g(x) = \text{fib}(x)$ , since there might be several ways of solving the same recursion equation.

(For instance, the recursion equation  $f(x) \simeq f(x)$  can be solved by any definition of  $f$ , e.g. by  $f(x) = 0$ , by  $f(x) = 1$ , and by  $f(x) \uparrow$ ).

Therefore we show by induction on  $n$  that  $\forall n \in \mathbb{N}. g(n) \simeq \text{fib}(n)$ , and that therefore `fib` is recursive:

- In the base cases  $n = 0, 1$  this is clear.
- In the induction step  $n \rightarrow n + 1$  with  $n \geq 1$  we have

$$g(n+1) \simeq g(n-1) + g(n) \stackrel{\text{IH}}{\simeq} \text{fib}(n-1) + \text{fib}(n) = \text{fib}(n+1).$$

In a similar way one can introduce arbitrary partial recursive functions  $g$ , where  $g(\vec{n})$  refers to arbitrary other values  $g(\vec{m})$ .

Such definitions correspond to the recursive definition of functions in many programming languages. For instance, in Java one defines the Fibonacci numbers recursively as follows:

```

public static int fib(int n){
    if (n == 0 || n == 1){
        return 1;}
    else{
        return fib(n-1) + fib(n-2);
    }
};

```

When programming functions recursively, we obtain functions which might terminate, or might not terminate. Similarly, when defining functions using the recursion theorem, we obtain partial recursive functions, which might not be always be defined, as in the following two examples:

3. There exists a partial recursive function  $g : \mathbb{N} \rightharpoonup \mathbb{N}$  s.t.

$$g(x) \simeq g(x) + 1 .$$

(Take in Kleene's Recursion Theorem

$$f(e, x) \simeq \{e\}(x) + 1 ,$$

so we obtain an  $e$  s.t.

$$\{e\}(x) \simeq f(e, x) \simeq \{e\}(x) + 1 .$$

Then let  $g = \{e\}$ .)

It follows that

$$g(x) \uparrow .$$

For, if  $g(x)$  were defined, say  $g(x) = n$ , we would get  $n = n + 1$ .

Note that, if  $g(x)$  is undefined,  $g(x) + 1$  is undefined as well, so

$$g(x) \simeq g(x) + 1$$

holds in this case.

The definition of  $g$  corresponds to the following definition in Java:

```

public static int g(int n){
    return g(n) + 1;
};

```

When executing  $g(x)$ , Java loops (actually it will terminate with a stack overflow).

(Note that a recursion equation for a function  $f$  can not always be solved by setting  $f(x) \uparrow$ . E.g. the recursion equation for fib above can't be solved by setting  $\text{fib}(n) \uparrow$ .)

4. There exists a partial recursive function  $g : \mathbb{N} \rightharpoonup \mathbb{N}$  s.t.

$$g(x) \simeq g(x + 1) + 1 .$$

Again  $g(x)$  is undefined, for if  $g(x)$  were defined, it need to be infinitely big.

A corresponding Java function will again loop for ever.

5. An interesting example, where the Recursion Theorem is of great help in order to show that a function is partial recursive, is the Ackermann Function  $\text{Ack}$ .

Remember that  $\text{Ack}$  has the following defining equations:

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

So we have

$$\text{Ack}(x, y) = \begin{cases} y + 1, & \text{if } x = 0, \\ \text{Ack}(x - 1, 1), & \text{if } x > 0 \text{ and } y = 0, \\ \text{Ack}(x - 1, \text{Ack}(x, y - 1)), & \text{otherwise.} \end{cases}$$

In order to show that  $\text{Ack}$  is recursive, we use Kleene's Recursion Theorem in order to introduce a partial recursive function  $g$  which fulfils the same equations, i.e.

$$g(x, y) \simeq \begin{cases} y + 1, & \text{if } x = 0, \\ g(x - 1, 1), & \text{if } x > 0 \wedge y = 0, \\ g(x - 1, g(x, y - 1)), & \text{if } x > 0 \wedge y > 0. \end{cases}$$

(In detail this is shown as follows:

There exists an  $e$  s.t.

$$\{e\}(x, y) \simeq \begin{cases} y + 1, & \text{if } x = 0, \\ \{e\}(x - 1, 1), & \text{if } x > 0 \wedge y = 0, \\ \{e\}(x - 1, \{e\}(x, y - 1)), & \text{if } x > 0 \wedge y > 0. \end{cases}$$

Let  $g := \{e\}^2$ . Then  $g$  fulfils those equations.)

We show by induction on  $x$  that  $g(x, y)$  is defined and equal to  $\text{Ack}(x, y)$  for all  $x, y \in \mathbb{N}$ :

- Base case  $x = 0$ .

$$g(0, y) = y + 1 = \text{Ack}(0, y).$$

- Induction Step  $x \rightarrow x + 1$ . Assume

$$g(x, y) = \text{Ack}(x, y).$$

We show

$$g(x + 1, y) = \text{Ack}(x + 1, y)$$

by side-induction on  $y$ :

- \* Base case  $y = 0$ :

$$g(x + 1, 0) \simeq g(x, 1) \stackrel{\text{Main-IH}}{=} \text{Ack}(x, 1) = \text{Ack}(x + 1, 0).$$

- \* Induction Step  $y \rightarrow y + 1$ :

$$\begin{aligned} g(x + 1, y + 1) &\simeq g(x, g(x + 1, y)) \\ &\stackrel{\text{Main-IH}}{\simeq} g(x, \text{Ack}(x + 1, y)) \\ &\stackrel{\text{Side-IH}}{\simeq} \text{Ack}(x, \text{Ack}(x + 1, y)) \\ &= \text{Ack}(x + 1, y + 1). \end{aligned}$$

**Proof of Kleene's Recursion Theorem:**

We write  $\vec{x}$  for  $x_0, \dots, x_{n-1}$ . Assume

$$f : \mathbb{N}^{n+1} \xrightarrow{\simeq} \mathbb{N} .$$

We have to find an  $e$  s.t.

$$\forall \vec{x} \in \mathbb{N}. \{e\}^n(\vec{x}) \simeq f(e, \vec{x}) .$$

The idea is to define  $e = S_n^1(e_1, e_2)$  for some (yet unknown)  $e_1, e_2$ . Then we get

$$\{e\}^n(\vec{x}) \simeq \{S_n^1(e_1, e_2)\}^n(\vec{x}) \simeq \{e_1\}^{n+1}(e_2, \vec{x}) .$$

So, in order to fulfil our original equation, we need to find  $e_1, e_2$  s.t.

$$\forall \vec{x}. \underbrace{\{e_1\}^{n+1}(e_2, \vec{x})}_{\simeq \{e\}^n(\vec{x})} \simeq \underbrace{f(S_n^1(e_1, e_2), \vec{x})}_{\simeq f(e, \vec{x})} .$$

Now let  $e_1$  be an index for the partial function function

$$(y, \vec{x}) \mapsto f(S_n^1(y, y), \vec{x}) ,$$

i.e. let  $e_1$  s.t.

$$\{e_1\}^{n+1}(y, \vec{x}) \simeq f(S_n^1(y, y), \vec{x}) .$$

Using this equation, the above equation becomes

$$\underbrace{f(S_n^1(e_1, e_2), \vec{x})}_{\simeq \{e_1\}^{n+1}(e_2, \vec{x})} \simeq f(S_n^1(e_2, e_2), \vec{x}) .$$

We can fulfill this equation by defining

$$e_2 := e_1 .$$

So, an index solving the problem is

$$e = S_n^1(e_1, e_2) = S_n^1(e_1, e_1) ,$$

and we are done.

In short the complete proof reads as follows:

Let  $e_1$  be s.t.

$$\{e_1\}^{n+1}(y, \vec{x}) \simeq f(S_n^1(y, y), \vec{x}) .$$

Let  $e := S_n^1(e_1, e_1)$ . Then we have

$$\begin{aligned} \{e\}^n(\vec{x}) & \stackrel{e = S_n^1(e_1, e_1)}{\simeq} \{S_n^1(e_1, e_1)\}^n(\vec{x}) \\ & \stackrel{\text{S-m-n theorem}}{\simeq} \{e_1\}^{n+1}(e_1, \vec{x}) \\ & \stackrel{\text{Def of } e_1}{\simeq} f(S_n^1(e_1, e_1), \vec{x}) \\ & \stackrel{e = S_n^1(e_1, e_1)}{\simeq} f(e, \vec{x}) . \end{aligned}$$

## 8 Recursively Enumerable Predicates

### 8.1 Introduction

In this section we are studying predicates  $P \subseteq \mathbb{N}^n$ , which are non-decidable, but “half decidable”. The official name is semi-decidable, or recursively enumerable. (The latter name will be explained later).

Remember that a predicate  $A$  is recursive (or, using the Church-Turing thesis computable), if its characteristic function  $\chi_A$  is recursive, so we have a “full” decision procedure:

$$\begin{aligned} P(x_0, \dots, x_{n-1}) &\Leftrightarrow \chi_A(x_0, \dots, x_{n-1}) = 1, \text{ i.e. answer yes ,} \\ \neg P(x_0, \dots, x_{n-1}) &\Leftrightarrow \chi_A(x_0, \dots, x_{n-1}) = 0, \text{ i.e. answer no .} \end{aligned}$$

A predicate  $P$  will be semi-decidable, if there exists a partial recursive function  $f$  s.t.

$$P(x_0, \dots, x_{n-1}) \Leftrightarrow f(x_0, \dots, x_{n-1}) \downarrow .$$

Therefore

- If  $P(x_0, \dots, x_{n-1})$  holds, we will eventually know it – the algorithm for computing  $f$  will finally terminate, and then we know that  $P(x_0, \dots, x_{n-1})$  holds.
- If  $P(x_0, \dots, x_{n-1})$  doesn't hold, then the algorithm computing  $f$  will loop for ever, and we never get an answer.

So we have:

$$\begin{aligned} P(x_0, \dots, x_{n-1}) &\Leftrightarrow f(x_0, \dots, x_{n-1}) \downarrow \text{ i.e. answer yes ,} \\ \neg P(x_0, \dots, x_{n-1}) &\Leftrightarrow f(x_0, \dots, x_{n-1}) \uparrow \text{ i.e. no answer returned by } f . \end{aligned}$$

It seems at first sight that recursively enumerable sets are not interesting from a computational point of view. But it turns out that such sets occur in many practical applications. A typical example is a set  $A$  of the form

$$A(x) \Leftrightarrow \text{there exists a proof in a certain formal system of a property } \varphi(x) .$$

For instance,  $\varphi(x)$  might express the correctness of an algorithm, e.g. that the result of a digital circuit with  $x$ -bit inputs for multiplying two binary numbers is actually the product of its inputs. Under certain conditions (“soundness and completeness”; see other modules) we have:

- If we find a proof for  $\varphi(x)$ , then  $\varphi(x)$  holds.
- If we don't find a proof, then  $\varphi(x)$  doesn't hold.

There are many formal systems such that the corresponding function  $f$  s.t.

$$A(x) \Leftrightarrow f(x) \downarrow$$

terminates in most practical cases very fast, despite the fact that for theoretical reasons it is known that it can take arbitrarily long before this happens.

If one under these circumstances runs  $f(x)$ , one usually assumes that, if one hasn't obtained an answer after a certain amount of time,  $A(x)$  is probably false, and then uses other means in order to determine the reason why  $A(x)$  is false.

If  $A(x)$  holds, one expects the algorithm for  $f(x)$  to terminate after a certain amount of time. If this algorithms actually terminates within the expected amount of time, one knows that  $\varphi(x)$  is true.

## 8.2 Recursively Enumerable Predicates and their Canonical Numbering

**Definition 8.1** Assume  $f : \mathbb{N}^n \rightsquigarrow \mathbb{N}$  is a partial function.

(a) The domain of  $f$ , in short  $\text{dom}(f)$  is defined as follows:

$$\text{dom}(f) := \{(x_0, \dots, x_{n-1}) \in \mathbb{N}^n \mid f(x_0, \dots, x_{n-1}) \downarrow\} .$$

(b) The range of  $f$ , in short  $\text{ran}(f)$  is defined as follows:

$$\text{ran}(f) := \{y \in \mathbb{N} \mid \exists x_0, \dots, x_{n-1}. (f(x_0, \dots, x_{n-1}) \simeq y)\} .$$

(c) The graph of  $f$  is the set  $G_f$  defined as

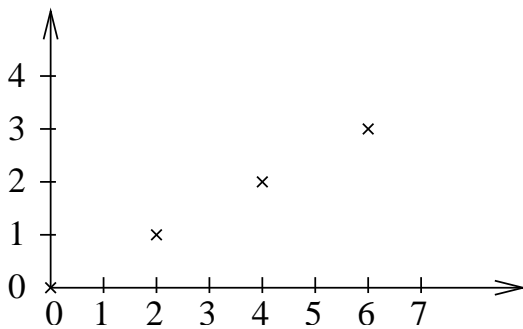
$$G_f := \{(\vec{x}, y) \in \mathbb{N}^{n+1} \mid f(\vec{x}) \simeq y\} .$$

**Remark:**

- The notion “graph” used here has nothing to do with the notion of “graph” in graph theory.
- The graph of a function is essentially the graph we draw when visualising  $f$ . Take as an example

$$f : \mathbb{N} \rightsquigarrow \mathbb{N} , \quad f(x) = \begin{cases} \frac{x}{2}, & \text{if } x \text{ even,} \\ \text{undefined,} & \text{if } x \text{ is odd.} \end{cases}$$

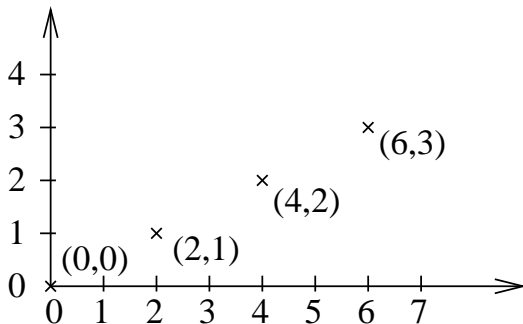
Then we can draw  $f$  as follows:



In this example we have

$$G_f = \{(0, 0), (2, 1), (4, 2), (6, 3), \dots\}$$

These are exactly the coordinates of the crosses in the picture:



**Definition 8.2** • A predicate  $A \subseteq \mathbb{N}^n$  is recursively enumerable, in short r.e., if there exists a partial recursive function  $f : \mathbb{N}^n \rightrightarrows \mathbb{N}$  s.t.

$$A = \text{dom}(f) .$$

- Sometimes recursive predicates are as well called
  - semi-decidable or
  - semi-computable or
  - partially computable.

**Lemma 8.3** (a) Every recursive predicate is r.e.

(b) The halting problem, defined as

$$\text{Halt}^n(e, \vec{x}) := \{e\}^n(\vec{x}) \downarrow ,$$

is r.e., but not recursive.

**Proof:**

(a) Assume  $A \subseteq \mathbb{N}^k$  is decidable. Then  $\mathbb{N}^k \setminus A$  is recursive, therefore its characteristic function  $\chi_{\mathbb{N}^k \setminus A}$  is recursive as well.

Define

$$f : \mathbb{N}^k \rightrightarrows \mathbb{N}, f(\vec{x}) := (\mu y. \chi_{\mathbb{N}^k \setminus A}(\vec{x}) \simeq 0) .$$

Note that  $y$  doesn't occur in the body of the  $\mu$ -expression. Then we have

- If  $A(\vec{x})$ , then

$$\chi_{\mathbb{N}^k \setminus A}(\vec{x}) \simeq 0 ,$$

so

$$f(\vec{x}) \simeq (\mu y. \chi_{\mathbb{N}^k \setminus A}(\vec{x}) \simeq 0) \simeq 0 ,$$

especially

$$f(\vec{x}) \downarrow .$$

- If  $(\mathbb{N}^k \setminus A)(\vec{x})$ , then

$$\chi_{\mathbb{N}^k \setminus A}(\vec{x}) \simeq 1 ,$$

so there exists no  $y$  s.t.

$$\chi_{\mathbb{N}^k \setminus A}(\vec{x}) \simeq 0 .$$

therefore

$$f(\vec{x}) \simeq (\mu y. \chi_{\mathbb{N}^k \setminus A}(\vec{x}) \simeq 0) \simeq \text{undefined} ,$$

especially

$$f(\vec{x}) \uparrow .$$

So we get

$$A(\vec{x}) \Leftrightarrow f(\vec{x}) \downarrow \Leftrightarrow \vec{x} \in \text{dom}(f) ,$$

$$A = \text{dom}(f) \text{ is r.e. .}$$

(b) We have

$$\text{Halt}^n(e, \vec{x}) := f_n(e, \vec{x}) \downarrow ,$$

where  $f_n$  is partial recursive as in Sect. 5 s.t.

$$\{e\}^n(\vec{x}) \simeq f_n(e, \vec{x}) .$$

So

$$\text{Halt}^n = \text{dom}(f_n) \text{ is r.e. .}$$

We have seen above that  $\text{Halt}^n$  is non-computable, i.e. not recursive.

**Theorem 8.4 (The sets  $W_e^n$ .)**

There exist r.e. predicates  $W^n \subseteq \mathbb{N}^{n+1}$  s.t., if we define

$$W_e^n := \{(x_0, \dots, x_{n-1}) \in \mathbb{N}^n \mid W^n(e, x_0, \dots, x_{n-1})\},$$

then we have the following:

- Each of the predicates  $W_e^n \subseteq \mathbb{N}^n$  is r.e.
- For each r.e. predicate  $P \subseteq \mathbb{N}^n$  there exists an  $e \in \mathbb{N}$  s.t.  $P = W_e^n$ , i.e.

$$\forall \vec{x} \in \mathbb{N}. P(\vec{x}) \Leftrightarrow W_e^n(\vec{x}).$$

In other words, the r.e. sets  $P \subseteq \mathbb{N}^n$  are exactly the sets  $W_e^n$ , where  $e$  ranges over all natural numbers.

**Remark:**

- $W_e^n$  is therefore a **universal recursively enumerable sets**, which encodes all other recursively enumerable predicates.
- The theorem expresses that we can assign to every recursively enumerable predicate  $A$  a natural number, namely the  $e$  s.t.  $A = W_e^n$ .
  - Each number denotes one predicate.
  - But several numbers denote the same predicate, i.e. there are  $e, e'$  s.t.  $e \neq e'$  but  $W_e^n = W_{e'}^n$ .  
(This is since there are  $e, e'$  s.t.  $e \neq e'$  but  $\{e\}^n = \{e'\}^n$ ).

**Proof idea for Theorem 8.4:**

If  $A$  is r.e., then  $A = \text{dom}(f)$  for some partial rec.  $f$ . Let  $f = \{e\}^n$ . Then  $A = W_e^n$ .

**Proof of Theorem 8.4:** Let  $f_n$  s.t.

$$\forall e, \vec{n} \in \mathbb{N}. f_n(e, \vec{x}) \simeq \{e\}(\vec{x}).$$

Define

$$W^n := \text{dom}(f_n).$$

$W^n$  is r.e. We have

$$\begin{aligned} \vec{x} \in W_e^n &\Leftrightarrow (e, \vec{x}) \in W^n \\ &\Leftrightarrow f_n(e, \vec{x}) \downarrow \\ &\Leftrightarrow \{e\}(\vec{x}) \downarrow \\ &\Leftrightarrow \vec{x} \in \text{dom}(\{e\}^n). \end{aligned}$$

Therefore

$$W_e^n = \text{dom}(\{e\}^n).$$

$W^n$  is r.e., since  $f_n$  is partial recursive.

Furthermore, we have for any set  $A \subseteq \mathbb{N}^n$

$$\begin{aligned} A \text{ is r.e.} &\text{ iff } A = \text{dom}(f) \text{ for some partial recursive } f \\ &\text{ iff } A = \text{dom}(\{e\}^n) \text{ for some } e \in \mathbb{N} \\ &\text{ iff } A = W_e^n \text{ for some } e \in \mathbb{N}. \end{aligned}$$

This shows the assertion.

### 8.3 Characterisations of the Partial Recursive Functions

**Theorem 8.5 (Characterisation of the recursively enumerable predicates).** *Let  $A \subseteq \mathbb{N}^n$ . The following is equivalent:*

(i)  $A$  is r.e.

(ii)

$$A = \{\vec{x} \mid \exists y.R(\vec{x}, y)\}$$

for some primitive-recursive predicate  $R$ .

(iii)

$$A = \{\vec{x} \mid \exists y.R(\vec{x}, y)\}$$

for some recursive predicate  $R$ .

(iv)

$$A = \{\vec{x} \mid \exists y.R(\vec{x}, y)\}$$

for some recursively enumerable predicate  $R$ .

(v)  $A = \emptyset$  or

$$A = \{(f_0(x), \dots, f_{n-1}(x)) \mid x \in \mathbb{N}\}$$

for some primitive-recursive functions

$$f_i : \mathbb{N} \rightarrow \mathbb{N} .$$

(vi)  $A = \emptyset$  or

$$A = \{(f_0(x), \dots, f_{n-1}(x)) \mid x \in \mathbb{N}\}$$

for some recursive functions

$$f_i : \mathbb{N} \rightarrow \mathbb{N} .$$

**Remark:**

(a) We can summarise Theorem 8.5 by saying that there are essentially three equivalent ways of defining that  $A \subseteq \mathbb{N}^n$  is r.e.:

- $A = \text{dom}(f)$  for some partial recursive  $f$ ;
- $A = \emptyset$  or  $A$  is the image of primitive recursive/recursive functions  $f_0, \dots, f_{n-1}$ ;
- $A = \{\vec{x} \mid \exists y.R(\vec{x}, y)\}$  for some primitive-recursive/recursive/r.e.  $R$ .

(b) Consider the special case  $n = 1$ . Taking (i), (v), (vi) we get the following equivalence:

- Let  $A \subseteq \mathbb{N}$ . Then the following is equivalent:

- $A$  is r.e.
- $A = \emptyset$  or

$$A = \text{ran}(f) \text{ for some primitive-recursive } f : \mathbb{N} \rightarrow \mathbb{N} .$$

(This is (v)).

- $A = \emptyset$  or

$$A = \text{ran}(f) \text{ for some recursive } f : \mathbb{N} \rightarrow \mathbb{N} .$$

(This is (vi))

This means that  $A \subseteq \mathbb{N}$  is r.e. if  $A = \emptyset$  or there exists a (primitive-)recursive function  $f$ , which enumerates all its elements. This explains the name “recursively enumerable predicate”.

**Proof of Theorem 8.5:**

**(i) → (ii):**

**Proof idea for (i) → (ii):** If  $A$  is r.e.,  $A = \text{dom}(f)$ , then

$$\begin{aligned} A(\vec{x}) &\Leftrightarrow f(\vec{x}) \downarrow \\ &\Leftrightarrow \exists y. \text{the Turing machine for computing } f(\vec{x}) \text{ terminates after } y \text{ steps} \\ &\Leftrightarrow \exists y. R(\vec{x}, y) \end{aligned}$$

where

$$R(\vec{x}, y) \Leftrightarrow \text{the Turing machine for computing } f(\vec{x}) \text{ terminates after } y \text{ steps .}$$

$R$  is primitive-recursive.

**Detailed proof of (i) → (ii):**

(The actual predicate  $R$  we will take will be slightly differently from that in the proof idea – it is technically easier to prove the theorem this way.)

If  $A$  is r.e., then for some partial recursive function  $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$  we have

$$A = \text{dom}(f) .$$

Let  $f = \{e\}^n$ .

By Kleene’s Normal Form Theorem there exist a primitive-recursive function  $U : \mathbb{N} \rightarrow \mathbb{N}$  and a primitive-recursive predicate  $T_n \subseteq \mathbb{N}^{n+1}$  s.t.

$$\{e\}^n(\vec{x}) \simeq U(\mu y. T_n(e, \vec{x}, y)) .$$

Therefore

$$\begin{aligned} A(\vec{x}) &\Leftrightarrow \vec{x} \in \text{dom}(f) \\ &\Leftrightarrow \vec{x} \in \text{dom}(\{e\}^n) \\ &\Leftrightarrow U(\mu y. T_n(e, \vec{x}, y)) \downarrow \\ U \text{ prim.-rec., therefore total} &\Leftrightarrow \mu y. T_n(e, \vec{x}, y) \downarrow \\ &\Leftrightarrow \exists y. T_n(e, \vec{x}, y) \\ &\Leftrightarrow \exists y. R(\vec{x}, y) . \end{aligned}$$

where

$$R(\vec{x}, y) \Leftrightarrow T_n(e, \vec{x}, y) .$$

Now  $R$  is primitive-recursive, and

$$A = \{\vec{x} \mid \exists y. R(\vec{x}, y)\} .$$

**(ii) → (iii):** Trivial.

**(iii) → (iv):** By Lemma 8.3.

**(iv) → (ii):**

Assume

$$A = \{\vec{x} \mid \exists y. R(\vec{x}, y)\} ,$$

where  $R$  is r.e. By “(i) → (ii)” there exists a primitive-recursive predicate  $S$  s.t.

$$R(\vec{x}, y) \Leftrightarrow \exists z. S(\vec{x}, y, z) .$$

Therefore

$$\begin{aligned} A &= \{\vec{x} \mid \exists y. \exists z. S(\vec{x}, y, z)\} \\ &= \{\vec{x} \mid \exists y. S(\vec{x}, \pi_0(y), \pi_1(y))\} \\ &= \{\vec{x} \mid \exists y. R'(\vec{x}, y)\} , \end{aligned}$$

where

$$R'(\vec{x}, y) :\Leftrightarrow S(\vec{x}, \pi_0(y), \pi_1(y)) \text{ is primitive-recursive.}$$

(ii)  $\rightarrow$  (v):

**Proof idea for (ii)  $\rightarrow$  (v), special case  $n = 1$ :**

Assume

- $A = \{x \in \mathbb{N} \mid \exists y. R(x, y)\}$ , where  $R$  is primitive recursive,
- $A \neq \emptyset$ ,
- $y \in A$  fixed.

Define  $f : \mathbb{N} \rightarrow \mathbb{N}$  recursive,

$$f(x) = \begin{cases} \pi_0(x), & \text{if } R(\pi_0(x), \pi_1(x)), \\ y & \text{otherwise.} \end{cases}$$

Then  $A = \text{ran}(f)$ .

**Detailed proof for (ii)  $\rightarrow$  (v):**

Assume  $A$  is not empty and  $R$  is primitive-recursive s.t.

$$A = \{\vec{x} \mid \exists y. R(\vec{x}, y)\} .$$

Let  $\vec{y} = y_0, \dots, y_{n-1}$  be some fixed elements s.t.  $A(\vec{y})$  holds. Define for  $i = 0, \dots, n-1$

$$f_i(x) := \begin{cases} \pi_i^{n+1}(x), & \text{if } R(\pi_0^{n+1}(x), \pi_1^{n+1}(x), \dots, \pi_{n-1}^{n+1}(x), \pi_n^{n+1}(x)), \\ y_i, & \text{otherwise.} \end{cases}$$

$f_i$  are primitive-recursive. We show

$$A = \{(f_0(x), \dots, f_{n-1}(x)) \mid x \in \mathbb{N}\} .$$

“ $\supseteq$ ”:

Assume  $x \in \mathbb{N}$ , and show

$$A(f_0(x), \dots, f_{n-1}(x)) .$$

- If  $R(\pi_0^{n+1}(x), \pi_1^{n+1}(x), \dots, \pi_{n-1}^{n+1}(x), \pi_n^{n+1}(x))$ , then

$$\exists z. R(\pi_0^{n+1}(x), \pi_1^{n+1}(x), \dots, \pi_{n-1}^{n+1}(x), z) ,$$

therefore

$$(\pi_0^{n+1}(x), \pi_1^{n+1}(x), \dots, \pi_{n-1}^{n+1}(x)) \in A ,$$

therefore

$$A(f_0(x), \dots, f_{n-1}(x)) .$$

- If  $(\mathbb{N}^k \setminus R)(\pi_0^{n+1}(x), \pi_1^{n+1}(x), \dots, \pi_{n-1}^{n+1}(x), \pi_n^{n+1}(x))$ , then

$$f_i(x) = y_i ,$$

therefore by  $A(\vec{y})$

$$A(f_0(x), \dots, f_{n-1}(x)) .$$

So in both cases we get that

$$A(f_0(x), \dots, f_{n-1}(x)) ,$$

so

$$\{(f_0(x), \dots, f_{n-1}(x)) \mid x \in \mathbb{N}\} \subseteq A .$$

“ $\subseteq$ ”:

Assume

$$A(x_0, \dots, x_{n-1}) ,$$

and show

$$\exists z.(f_0(z) = x_0 \wedge \dots \wedge f_{n-1}(z) = x_{n-1}) .$$

We have for some  $y$

$$R(x_0, \dots, x_{n-1}, y) .$$

Let

$$z = \pi^{n+1}(x_0, \dots, x_{n-1}, y) .$$

Then we have

$$x_i = \pi_i^{n+1}(z) , \quad y = \pi_n^{n+1}(z) ,$$

therefore

$$R(\pi_0^{n+1}(z), \pi_1^{n+1}(z), \dots, \pi_{n-1}^{n+1}(z), \pi_n^{n+1}(z)) ,$$

therefore for  $i = 0, \dots, n-1$

$$f_i(z) = \pi_i^{n+1}(z) = x_i ,$$

therefore

$$(x_0, \dots, x_{n-1}) = (f_0(z), \dots, f_{n-1}(z)) \in \{(f_0(x), \dots, f_{n-1}(x)) \mid x \in \mathbb{N}\} ,$$

and we have

$$A \subseteq \{(f_0(x), \dots, f_{n-1}(x)) \mid x \in \mathbb{N}\} .$$

Therefore we have shown

$$A = \{(f_0(x), \dots, f_{n-1}(x)) \mid x \in \mathbb{N}\} ,$$

and the assertion follows.

(v)  $\rightarrow$  (vi): Trivial.

(vi)  $\rightarrow$  (i):

**Proof idea for (vi)  $\rightarrow$  (i), special case  $n = 1$ :**

If  $A = \text{ran}(f)$ , where  $f$  is recursive, then  $A = \text{dom}(g)$  where

$$g(x) \simeq (\mu y.f(y) \simeq x) .$$

$g$  is partial recursive.

**Detailed proof for (vi)  $\rightarrow$  (i):**

If  $A$  is empty, then  $A$  is recursive, therefore r.e.

Assume

$$A = \{(f_0(x), \dots, f_{n-1}(x)) \mid x \in \mathbb{N}\} .$$

for some recursive functions  $f_i$ .

Define

$$f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N} ,$$

s.t.

$$f(x_0, \dots, x_{n-1}) \simeq \mu x.(f_0(x) \simeq x_0 \wedge \dots \wedge f_{n-1}(x) \simeq x_{n-1}) .$$

$f$  can be written as

$$f(x_0, \dots, x_{n-1}) := \mu x. (((f_0(x) \dot{-} x_0) + (x_0 \dot{-} f_0(x))) + ((f_1(x) \dot{-} x_1) + (x_1 \dot{-} f_1(x))) + \dots + ((f_{n-1}(x) \dot{-} x_{n-1}) + (x_{n-1} \dot{-} f_{n-1}(x))) \simeq 0) ,$$

therefore  $f$  is partial recursive.  
Further we have

$$\begin{aligned} A(x_0, \dots, x_{n-1}) &\Leftrightarrow \exists x \in \mathbb{N}. x_0 = f_0(x) \wedge \dots \wedge x_{n-1} = f_{n-1}(x) \\ &\Leftrightarrow f(x_0, \dots, x_{n-1}) \downarrow , \end{aligned}$$

therefore

$$A = \text{dom}(f) \text{ is r.e. .}$$

**Theorem 8.6 (Relationship between recursive and r.e. predicates).**  
 $A \subseteq \mathbb{N}^k$  is recursive iff both  $A$  and  $\mathbb{N}^k \setminus A$  are r.e.

**Proof ideas:**

“ $\Rightarrow$ ” is easy.

For “ $\Leftarrow$ ”: Assume  $R, S$  primitive-recursive s.t.

$$\begin{aligned} A(\vec{x}) &\Leftrightarrow \exists y. R(\vec{x}, y) \\ (\mathbb{N}^k \setminus A)(\vec{x}) &\Leftrightarrow \exists y. S(\vec{x}, y) \end{aligned}$$

In order to decide  $A$ , search simultaneously for a  $y$  s.t.  $R(\vec{x}, y)$  and for a  $y$  s.t.  $S(\vec{x}, y)$  holds.  
If we find a  $y$  s.t.  $R(\vec{x}, y)$  holds, then  $A(\vec{x})$  holds.  
If we find a  $y$  s.t.  $S(\vec{x}, y)$  holds, then  $\neg A(\vec{x})$  holds

**Detailed Proof:**

“ $\Rightarrow$ ”:

If  $A$  is recursive, then both  $A$  and  $\mathbb{N}^k \setminus A$  are recursive, therefore as well r.e.

“ $\Leftarrow$ ”:

Assume  $A, \mathbb{N}^k \setminus A$  are r.e.

Then there exist primitive-recursive predicates  $R$  and  $S$  s.t.

$$\begin{aligned} A &= \{ \vec{x} \mid \exists y. R(\vec{x}, y) \} , \\ \mathbb{N}^k \setminus A &= \{ \vec{x} \mid \exists y. S(\vec{x}, y) \} . \end{aligned}$$

By

$$A \cup (\mathbb{N}^k \setminus A) = \mathbb{N}^k ,$$

it follows

$$\forall \vec{x}. ((\exists y. R(\vec{x}, y)) \vee (\exists y. S(\vec{x}, y))) ,$$

therefore as well

$$\forall \vec{x}. \exists y. (R(\vec{x}, y) \vee S(\vec{x}, y)) . \quad (*)$$

Define

$$h : \mathbb{N}^n \rightarrow \mathbb{N} , h(\vec{x}) := \mu y. (R(\vec{x}, y) \vee S(\vec{x}, y)) .$$

$h$  is partial recursive. By (\*) we have  $h$  is total, so  $h$  is recursive.

We show

$$A(\vec{x}) \Leftrightarrow R(\vec{x}, h(\vec{x})) .$$

If

$$A(\vec{x})$$

then

$$\exists y.R(\vec{x}, y)$$

and

$$\vec{x} \notin (\mathbb{N}^k \setminus A) ,$$

therefore

$$\neg \exists y.S(\vec{x}, y) .$$

Therefore we have for the  $y$  found by  $h(\vec{x})$  that  $R(\vec{x}, y)$  holds, i.e.

$$R(\vec{x}, h(\vec{x})) .$$

On the other hand, if  $R(\vec{x}, h(\vec{x}))$  holds then

$$\exists y.R(\vec{x}, y) ,$$

therefore

$$A(\vec{x}) .$$

Now we get

$$A = \{\vec{x} \mid R(\vec{x}, h(\vec{x}))\} \text{ is recursive.}$$

**Theorem 8.7 (Characterisation of partial recursive functions).**

Let  $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ .

Then

$$f \text{ is partial recursive} \Leftrightarrow G_f \text{ is r.e.} .$$

**Proof idea for “ $\Leftarrow$ ”:**

Assume  $R$  primitive recursive s.t.

$$G_f(\vec{x}, y) \Leftrightarrow \exists z.R(\vec{x}, y, z) .$$

In order to compute  $f(\vec{x})$ , search for a  $y$  s.t.  $R(\vec{x}, \pi_0(y), \pi_1(y))$  holds.  $f(\vec{x})$  will be the first projection of this  $y$ .

**Detailed Proof:**

“ $\Rightarrow$ ”:

Assume  $f$  is partial recursive. Then  $f = \{e\}^n$  for some  $e \in \mathbb{N}$ . By Kleene’s Normal Form Theorem we have

$$f(\vec{x}) \simeq U(\mu y.T_n(\vec{x}, y)) ,$$

for some primitive-recursive relation  $T_n \subseteq \mathbb{N}^{n+1}$  and some primitive-recursive function  $U : \mathbb{N} \rightarrow \mathbb{N}$ . Therefore

$$(\vec{x}, y) \in G_f \Leftrightarrow (f(\vec{x}) \simeq y) \Leftrightarrow \exists z.(T_n(\vec{x}, z) \wedge (\forall z' < z.T_n(\vec{x}, z')) \wedge U(z) = y) ,$$

therefore  $G_f$  is r.e.

“ $\Leftarrow$ ”:

If  $G_f$  is r.e., then there exists a primitive-recursive predicate  $R$  s.t.

$$f(\vec{x}) \simeq y \Leftrightarrow (\vec{x}, y) \in G_f \Leftrightarrow \exists z.R(\vec{x}, y, z) .$$

Therefore for any  $z$  s.t.  $R(\vec{x}, \pi_0(z), \pi_1(z))$  holds we have that

$$f(\vec{x}) \simeq \pi_0(z) .$$

Therefore

$$f(\vec{x}) \simeq \pi_0(\mu u.R(\vec{x}, \pi_0(u), \pi_1(u))) ,$$

$f$  is partial recursive.

## 8.4 Closure Properties of the Recursively enumerable Predicates

**Lemma 8.8** *The recursively enumerable predicates are closed under the following operations:*

(a) **Union:**

If  $A, B \subseteq \mathbb{N}^n$  are r.e. so is  $A \cup B$ .

(b) **Intersection:**

If  $A, B \subseteq \mathbb{N}^n$  are r.e. so is  $A \cap B$ .

(c) **Substitution by recursive functions:**

If  $A \subseteq \mathbb{N}^n$  is r.e.,  $f_i : \mathbb{N}^k \rightarrow \mathbb{N}$  are recursive for  $i = 0, \dots, n$ , so is

$$C := \{(y_0, \dots, y_{k-1}) \in \mathbb{N}^k \mid A(f_0(y_0, \dots, y_{k-1}), \dots, f_{n-1}(y_0, \dots, y_{k-1}))\} .$$

(d) **(Unbounded) existential quantification:**

If  $D \subseteq \mathbb{N}^{n+1}$  is r.e., so is

$$E := \{(x_0, \dots, x_{n-1}) \in \mathbb{N}^n \mid \exists y. D(x_0, \dots, x_{n-1}, y)\} .$$

(e) **Bounded universal quantification:**

If  $D \subseteq \mathbb{N}^{n+1}$  is r.e., so is

$$F := \{(x_0, \dots, x_{n-1}, z) \in \mathbb{N}^{n+1} \mid \forall y < z. D(x_0, \dots, x_{n-1}, z)\} .$$

**Proof:**

Let  $A, B \subseteq \mathbb{N}^n$  be r.e.

Then there exist primitive-recursive relations  $R, S$  s.t.

$$\begin{aligned} A &= \{(x_0, \dots, x_{n-1}) \in \mathbb{N}^n \mid \exists y. R(x_0, \dots, x_{n-1}, y)\} , \\ B &= \{(x_0, \dots, x_{n-1}) \in \mathbb{N}^n \mid \exists y. S(x_0, \dots, x_{n-1}, y)\} . \end{aligned}$$

(a), (b):

One can easily see that

$$\begin{aligned} A \cup B &= \{(x_0, \dots, x_{n-1}) \in \mathbb{N}^n \mid \exists y. (R(x_0, \dots, x_{n-1}, y) \vee S(x_0, \dots, x_{n-1}, y))\} , \\ A \cap B &= \{(x_0, \dots, x_{n-1}) \in \mathbb{N}^n \mid \exists y. (R(x_0, \dots, x_{n-1}, \pi_0(y)) \wedge S(x_0, \dots, x_{n-1}, \pi_1(y)))\} . \end{aligned}$$

therefore  $A \cup B$  and  $A \cap B$  are r.e.

(c):

$$\begin{aligned} C &= \{(\vec{y}) \mid A(f_0(\vec{y}), \dots, f_{n-1}(\vec{y}))\} \\ &= \{(\vec{y}) \mid \exists z. R(f_0(\vec{y}), \dots, f_{n-1}(\vec{y}), z)\} \text{ is r.e.} \end{aligned}$$

(d) follows from 8.5.

(e):

Assume  $T$  is a primitive-recursive predicate s.t.

$$D = \{(x_0, \dots, x_{n-1}, y) \in \mathbb{N}^{n+1} \mid \exists z. T(x_0, \dots, x_{n-1}, y, z)\} .$$

Then we get

$$\begin{aligned} F &= \{(\vec{x}, y) \mid \forall y' < y. D(\vec{x}, y')\} \\ &= \{(\vec{x}, y) \mid \forall y' < y. \exists z. T(\vec{x}, y', z)\} \\ &= \{(\vec{x}, y) \mid \exists z. \forall y' < y. T(\vec{x}, y', (z)_{y'})\} \text{ is r.e.,} \end{aligned}$$

where in the last line we used that

$$\{(\vec{x}, z) \mid \forall y' < y. T(\vec{x}, y', (z)_{y'})\} \text{ is primitive-recursive .}$$

**Lemma 8.9** *The r.e. predicates are not closed under complement, i.e. there exists an r.e. predicate  $A \subseteq \mathbb{N}^n$  s.t.  $\mathbb{N}^n \setminus A$  is not r.e.*

**Proof:**

$\text{Halt}^n$  is r.e.. If  $\mathbb{N}^n \setminus \text{Halt}^n$  were r.e., then by Theorem 8.6  $\text{Halt}^n$  were recursive, which is not the case.

## 9 Lambda-definable Functions

## 10 Reducibility.

## 11 Computational Complexity