VILNIUS UNIVERSITY
Faculty of Mathematics and Informatics
Department of Computer Science

ALGORITHM THEORY

Lecture Notes

Adomas Birštunas

Vilnius, 2012

Contents

Introduction

Algorithm Theory is a theoretical part of the computer science. It analyses foundations of algorithms and programming. Algorithm Theory tries to find answers to the following questions:

What is an algorithm?

Can we solve any problem?

Can we determine if a problem is solvable?

What is the complexity of the problem?

What algorithms and programs can calculate and what they cannot?

What are the limitations of the computer?

This teaching material is prepared to help students to study Algorithm Theory. Algorithm Theory course is a compulsory course for computer science and software engineering students of the Faculty of Mathematics and Informatics.

Initially, this course was prepared by associated professor S. Norgėla and later adopted by A. Birštunas. This work is mainly based on works presented in [5, 4, 3].

Chapter 1

Calculi for Propositional Logic

In this chapter we present propositional logic and three main calculi for it. These calculi are:

- Hilbert-style calculus,
- Sequent calculus,
- Resolution Method.

Differently from truth tables, these calculi are also defined for the most known logics (predicate logic, temporal logic, modal logics).

Basic concepts of the propositional logic are introduced in the Section 1.1. In the Section 1.2 we present Hilbert-style calculus, and in the Section 1.3 we present useful Deduction Theorem, which is applicable for Hilbert-style calculus. Sequent calculi are defined in the Section 1.4 and Resolution Method in the Section 1.5.

1.1 Propositional Logic

Definition 1 *Proposition is such a statement which may only be right (true) or wrong (false).*

Propositions usually are denoted by variables - $p, q, r, s, w, \ldots$. Logical constants T (logical true) and F (logical false) are the only values propositions may have. Propositions may be joined using logical operations.

Logical operations may be defined using truth tables:

Definition 2 *Logical operations are:*

- *Logical negation (* $\neg$ *):*

p	$\neg p$
F	T
T	F

- *Logical conjunction (* $\&$ *or* $\wedge$ *):*

p	q	$p\&q$
F	F	F
F	T	F
T	F	F
T	T	T

- *Logical disjunction (* $\vee$ *):*

p	q	$p \vee q$
F	F	F
F	T	T
T	F	T
T	T	T

- *Logical implication (* $\rightarrow$ *):*

p	q	$p \rightarrow q$
F	F	T
F	T	T
T	F	F
T	T	T

- *Logical Equality (* $\leftrightarrow$ *):*

p	q	$p \leftrightarrow q$
F	F	T
F	T	F
T	F	F
T	T	T

- *Exclusive Disjunction* ($\oplus$):

p	q	$p \oplus q$
F	F	F
F	T	T
T	F	T
T	T	F

- *Sheffer stroke* ($|$):

p	q	$p\|q$
F	F	T
F	T	T
T	F	T
T	T	F

Definition 3 *Formula of the propositional logic is defined as follows:*

- *any proposition is a formula,*
- *if φ is a formula, then $\neg\varphi$ is also a formula,*
- *if φ and ψ are formulas, then $(\varphi\&\psi), (\varphi \vee \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi), (\varphi \oplus \psi), (\varphi|\psi)$ are also formulas.*

Example 1 *Examples of the atomic propositions are:*
p *- 'Argentina is the 2006 year World basketball champion' and*
q *- '2010 year World basketball championship was organized by Lithuania'.*

Sentence 'What is the time?' is not a proposition, because it does not state anything.

A composite proposition (statement) may be expressed using a formula.

The formula $p \rightarrow \neg q$ denote the proposition 'If Argentina is the 2006 year World basketball champion, then 2010 year World basketball championship was not organized by Lithuania'.

Definition 4 *The interpretation of the set of propositions $A = \{p_1, p_2, \ldots, p_n\}$ is some function ν with domain A and range $\{T, F\}$.*

The interpretation defines values of the atomic propositions. Using logical operations definitions we may determine if the given formula is true or false with particular interpretation.

Any formula φ is true or false with a particular interpretation ν. If formula φ is true with interpretation ν, we will write $\nu(\varphi) = T$. If formula φ is false with interpretation ν, we will write $\nu(\varphi) = F$.

Definition 5 *Formulas φ, ψ are equivalent if $\nu(\varphi) = \nu(\psi)$ for any interpretation ν. We denote this by using the symbol $\sim$: $\varphi \sim \psi$.*

Example 2 *Formulas $\varphi = p \to \neg q$ and $\psi = \neg(p\&q)$ are equivalent ($\varphi \sim \psi$), since:*

p	q	$\neg q$	$p \to \neg q$	$p\&q$	$\neg(p\&q)$
F	F	T	T	F	T
F	T	F	T	F	T
T	F	T	T	F	T
T	T	F	F	T	F

Definition 6 *Formula φ is satisfiable if there is an interpretation ν, that $\nu(\varphi) = T$ (i.e., it is true with at least one interpretation).*

Definition 7 *Formula φ is a tautology if $\nu(\varphi) = T$ for any interpretation ν (i.e., it is true with any interpretation).*

Example 3 *Propositional logic formula $\varphi = \neg(p\&q)$ is satisfiable, since with interpretation $\nu_1 : \nu_1(p) = T, \nu_1(q) = F$, formula is true: $\nu_1(\varphi) = \neg(T\&F) = \neg F = T$.*

$\varphi = \neg(p\&q)$ is not a tautology, since with interpretation $\nu_2 : \nu_2(p) = T, \nu_2(q) = T$, formula is false: $\nu_2(\varphi) = \neg(T\&T) = \neg T = F$.

Definition 8 *Literal is an atom (atomic proposition) or atom with negation - $p, \neg p$.*

Definition 9 *Clause (or disjunct) is a disjunction of literals.*

Definition 10 *Conjunct is a conjunction of literals.*

Example 4 *Examples of clauses are: $p \vee q \vee s \vee w$, $w \vee \neg q \vee p$, $q \vee \neg p$, $\neg p \vee \neg s$. Formulas p, $\neg q$, q are also clauses. These clauses consist of only one literal. Formula $w \vee \neg(q \vee p)$ is not a clause, since negation is placed before parenthesis.*

Examples of conjuncts are: $p\&q\&s\&w$, $w\&\neg q\&p$, $q\&\neg p$, $\neg p\&\neg s$, p, $\neg q$, q. Formula $w\&\neg(q\&p)$ is not a conjunct, since negation is placed before parenthesis.

Definition 11 *Conjunctive normal form (CNF) of formula φ is an equivalent formula having the shape $D_1\&D_2\&\ldots\&D_n$, and, for every $i \in \{1, 2, \ldots, n\}$, D_i is a clause (disjunct).*

Definition 12 *Disjunctive normal form (DNF) of formula φ is an equivalent formula having the shape $C_1 \vee C_2 \vee \ldots \vee C_n$, and, for every $i \in \{1, 2, \ldots, n\}$, C_i is a conjunct.*

Algorithm to find CNF or DNF:

1) Eliminate all operations except $\neg, \&, \vee$ using the following (or other) equivalences:

- $p \to q \sim \neg p \vee q$,
- $p \leftrightarrow q \sim (p \to q)\&(q \to p)$,
- $p \oplus q \sim \neg(p \leftrightarrow q)$,

- $p|q\neg(p\&q)$.

2) Move the negation into parenthesis using the following equivalences:

- $\neg(p \vee q) \sim \neg p\&\neg q$,
- $\neg(p\&q) \sim \neg p \vee \neg q$.

3) Use distribution to get CNF or DNF:

- $p \vee (q\&s) \sim (p \vee q)\&(p \vee s)$ - to get CNF,
- $p\&(q \vee s) \sim (p\&q) \vee (p\&s)$ - to get DNF.

Example 5 *Given a formula $\varphi = (p \leftrightarrow (q \vee s))$.*

CNF of the formula φ is:

$\varphi = (p \leftrightarrow (q \vee s)) \sim (p \rightarrow (q \vee s))\&((q \vee s) \rightarrow p) \sim (\neg p \vee (q \vee s))\&(\neg(q \vee s) \vee p) \sim$
$\sim (\neg p \vee q \vee s)\&((\neg q\&\neg s) \vee p) \sim (\neg p \vee q \vee s)\&(\neg q \vee p)\&(\neg s \vee p)$.

It is a CNF of the formula φ and it consists of three clauses (disjuncts):
$(\neg p \vee q \vee s),\ (\neg q \vee p),\ (\neg s \vee p)$.

1.2 Hilbert-style Calculus for Propositional Logic

Definition 13 *Hilbert-style calculus for propositional logic is a calculus with Modus Ponens rule and axioms as follows:*

1.1. $A \to (B \to A)$

1.2. $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

2.1. $(A\&B) \to A$

2.2. $(A\&B) \to B$

2.3. $(A \to B) \to ((A \to C) \to (A \to (B\&C)))$

3.1. $A \to (A \vee B)$

3.2. $B \to (A \vee B)$

3.3. $(A \to C) \to ((B \to C) \to ((A \vee B) \to C))$

4.1. $(A \to B) \to (\neg B \to \neg A)$

4.2. $A \to \neg\neg A$

4.3. $\neg\neg A \to A$

Modus Ponens rule: $\dfrac{A \qquad A \to B}{B}$ (MP)

here, A, B, C - some formulas of the propositional logic.

There are 11 different axiom schemes (from 1.1. to 4.3) used in Hilbert-style calculus for propositional logic. When A, B, C are replaced by some particular formulas, we get particular axioms. Therefore, every axiom scheme defines an infinite set of axioms.

Example 6 *From the axiom scheme 1.1 we can get:*

- *an axiom $p \to (q \to p)$ (substitution: A - p, B - q), or*
- *an axiom $(p \vee q) \to ((\neg w \to p) \to (p \vee q))$ (substitution: A - $(p \vee q)$, B - $(\neg w \to p)$).*

Definition 14 *We say that formula φ is provable in Hilbert-style calculus for propositional logic if there exists such a sequence of the formulas, that every formula in this sequence is*

- *an axiom, or*
- *it is obtained by the Modus Ponens rule application from the formulas listed above,*

and this sequence ends with formula φ.

If formula φ is provable, we will write $\vdash \varphi$. The sequence of the formulas which leads to $\vdash \varphi$ will be called the formula φ proof in Hilbert-style calculus.

Example 7 *We will show that formula $p \to p$ is provable in Hilbert-style calculus for propositional logic. We may construct such a sequence of the formulas:*

1) $(p \to ((p \to p) \to p)) \to ((p \to (p \to p)) \to (p \to p))$ *- axiom 1.2 (substitution:* A *-* p*,* B *-* $(p \to p)$*,* C *-* p*),*

2) $p \to ((p \to p) \to p)$ *- axiom 1.1 (substitution:* A *-* p*,* B *-* $(p \to p)$*),*

3) $(p \to (p \to p)) \to (p \to p)$ *- using MP rule from 1 and 2,*

4) $p \to (p \to p)$ *- axiom 1.1 (substitution:* A *-* p*,* B *-* p*),*

5) $p \to p$ *- using MP rule from 3 and 4.*

We constructed a formula $p \to p$ *proof in Hilbert-style calculus, i.e. we show that* $\vdash p \to p$*.*

Theorem 1 *Formula* φ *is provable in Hilbert-style calculus for propositional logic if and only if* φ *is a tautology.*

Theorem 1 says that Hilbert-style calculus is some method which determines if a particular propositional logic formula φ is a tautology. However, this method has some disadvantage. If we success in sequence construction (for some formula φ), we know that the formula is a tautology, but if we fail in sequence construction, we cannot say that the formula is not a tautology. In such a case, we only know that one of the following cases holds:

- formula is not a tautology,
- formula is a tautology, but we didn't find the correct sequence.

This is a reason why Hilbert-style calculus isn't suitable to be a base for an automatic proof search.

Definition 15 *We say that some axiom* K *of some calculus* $\mathfrak{L}$ *is independent, if* K *is not provable in the calculus* $\mathfrak{L}'$*, which is constructed from calculus* $\mathfrak{L}$ *by removing only one axiom* K*.*

Theorem 2 *Every axiom of Hilbert-style calculus for propositional logic is independent.*

1.3 Deduction Theorem

Proof in Hilbert-style calculus for propositional logic is a formula construction from the axioms. If the goal formula is large, this construction becomes rather complicated. There is a Deduction Theorem, which may help to simplify the proof for some large formulas.

To use Deduction Theorem we have to make some generalization.

Definition 16 *We say that formula φ is derivable from premises $A_1, A_2, \ldots, A_n$ in Hilbert-style calculus for propositional logic if there exists such a sequence of the formulas, that every formula in this sequence is*

- *a premise (one of $A_1, A_2, \ldots, A_n$), or*
- *an axiom, or*
- *it is obtained by the MP rule application from the formulas listed above,*

and this sequence ends with formula φ.

If formula φ is derivable from premises $A_1, \ldots, A_n$, we will write $A_1, \ldots, A_n \vdash \varphi$. The sequence of the formulas which leads to $A_1, \ldots, A_n \vdash \varphi$ will be called the formula φ derivation from $A_1, \ldots, A_n$ in Hilbert-style calculus.

It is obvious that if some formula φ is derivable from an empty set of the premises, then formula φ is provable.

Example 8 *We will show that $w\&p, w \to p \vdash (w \vee (p\&q)) \to p$.*

Formula $(w \vee (p\&q)) \to p$ derivation from $w\&p, w \to p$ is:

1) $(w \to p) \to (((p\&q) \to p) \to ((w \vee (p\&q)) \to p))$ - axiom 3.3 (substitution: A - w, B - $(p\&q)$, C - p),

2) $w \to p$ - a premise,

3) $((p\&q) \to p) \to ((w \vee (p\&q)) \to p)$ - using MP rule from 1 and 2,

4) $(p\&q) \to p$ - axiom 2.1 (substitution: A - p, B - q),

5) $(w \vee (p\&q)) \to p$ - using MP rule from 3 and 4.

Theorem 3 (Deduction Theorem) $\Gamma, A \vdash B$ *if and only if* $\Gamma \vdash A \to B$.

Proof.

$\Leftarrow$) We have that $\Gamma \vdash A \to B$. So, there exists such a sequence of the formulas:

1) φ_1,
2) φ_2,
$\ldots$
n) $\varphi_n = A \to B$,

and every formula φ_i ($i \in \{1, 2, \ldots, n\}$) is a premise from Γ, or it is an axiom, or it is obtained by the Modus Ponens rule application.

Using this sequence of formulas we construct a formula B derivation from Γ, A:

1) φ_1,
2) φ_2,
…

n) $\varphi_n = A \to B$,
n+1) $\varphi_{n+1} = A$ - premise,
n+2) $\varphi_{n+2} = B$ - using MP rule from φ_n and φ_{n+1}.

Every formula φ_i ($i \in \{1, 2, \ldots, n+2\}$) in this sequence is a premise from Γ, or it is a premise A, or it is an axiom, or it is obtained by the Modus Ponens rule application.

Therefore, we show that $\Gamma, A \vdash B$.

$\Rightarrow$) We have that $\Gamma, A \vdash B$. So, there exists such a sequence S_1 of the formulas:

1) φ_1,
2) φ_2,
…

n) $\varphi_n = B$,

and every formula φ_i ($i \in \{1, 2, \ldots, n\}$) in this sequence is:

a) a premise A,
b) a premise from Γ,
c) an axiom,
d) it is obtained by the MP rule application from φ_j and φ_k listed above.

We will show how to construct formula $A \to B$ derivation from Γ.

At first, we take a sequence of the formulas S_2, which ends with formula $A \to B$:

1) $A \to \varphi_1$,
2) $A \to \varphi_2$,
…

n) $A \to \varphi_n = A \to B$,

S_2 is not $A \to B$ derivation from Γ, since formulas (listed in sequence S_2) are neither premises, neither axioms, nor obtained by the MP rule application.

We will construct sequence of the formulas S_3 from sequence S_2. Every formula $A \to \varphi_i$ will be replaced by its derivation.

This replacement is performed as follows:

a) If formula φ_i was a premise A in sequence S_1, then formula $A \to \varphi_i = A \to A$ and it is replaced by such a sequence:

   i.1) $(A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A))$ - axiom 1.2,
   i.2) $A \to ((A \to A) \to A)$ - axiom 1.1 (substitution: A - A, B - $(A \to A)$),
   i.3) $(A \to (A \to A)) \to (A \to A)$ - using MP rule from $(i.1)$ and $(i.2)$,
   i.4) $A \to (A \to A)$ - axiom 1.1 (substitution: A - A, B - A),
   i.5) $A \to A$ - using MP rule from $(i.3)$ and $(i.4)$.

b) If formula φ_i was a premise from Γ in sequence S_1, then formula $A \to \varphi_i$ is replaced by such a sequence:

   i.1) $\varphi_i \to (A \to \varphi_i)$ - axiom 1.1 (substitution: A - φ_i, B - A),

   i.2) φ_i - a premise from Γ,

   i.3) $A \to \varphi_i$ - using MP rule from $(i.1)$ and $(i.2)$.

c) If formula φ_i was an axiom in sequence S_1, then formula $A \to \varphi_i$ is replaced by such a sequence:

   i.1) $\varphi_i \to (A \to \varphi_i)$ - axiom 1.1 (substitution: A - φ_i, B - A),

   i.2) φ_i - an axiom (the same as in sequence S_1),

   i.3) $A \to \varphi_i$ - using MP rule from $(i.1)$ and $(i.2)$.

d) If formula φ_i was obtained by the Modus Ponens rule from φ_j and φ_k $(j, k < i)$, then $\varphi_j = \varphi_k \to \varphi_i$ (or $\varphi_k = \varphi_j \to \varphi_i$ and we get an analogous case).

   We already have formulas $j.x)$ $A \to \varphi_j = A \to (\varphi_k \to \varphi_i)$ and $k.y)$ $A \to \varphi_k$ listed above in the sequence S_3 (since $j, k < i$).

   Formula $A \to \varphi_i$ is replaced by such a sequence:

   i.1) $(A \to (\varphi_k \to \varphi_i)) \to ((A \to \varphi_k) \to (A \to \varphi_i))$ - axiom 1.2, (substitution: A - A, B - φ_k, C - φ_i),

   i.2) $(A \to \varphi_k) \to (A \to \varphi_i)$ - using MP rule from $(i.1)$ and $(j.x)$ (already listed above).

   i.3) $A \to \varphi_i$ - using MP rule from $(i.2)$ and $(k.y)$ (already listed above).

Every formula in the sequence S_3 is a premise from Γ, or it is an axiom, or it is obtained by the Modus Ponens rule application. Note, that premise A is not used in sequence S_3. Sequence S_3 ends with formula $A \to B$.

Therefore, we show that $\Gamma \vdash A \to B$.

Deduction Theorem (Theorem 3) usage (deriving $\Gamma, A \vdash B$ instead of $\Gamma \vdash A \to B$) has two advantages:

- we can use one more premise (A) in the derivation sequence,
- we have to construct a sequence which ends with a simpler formula (formula B instead of the formula $A \to B$).

Example 9 *We will show that $w\&p, w \to p \vdash (w \vee (p\&q)) \to p$.*
It is taken from Example 8, but at this time we will use Deduction Theorem:

$w\&p, w \to p \vdash (w \vee (p\&q)) \to p$

$\Updownarrow$ *(by Deduction Theorem)*

$w\&p, w \to p, w \vee (p\&q) \vdash p$

Formula p derivation from $w\&p, w \to p, w \vee (p\&q)$ is:

1) $(w\&p) \to p$ - axiom 2.2 (substitution: A - w, B - p),

2) $w\&p$ - a premise,

3) p - using MP rule from 1 and 2.

1.4 Sequent Calculi

Hilbert-style calculus for propositional logic is not applicable for an automatic proof search.

In 1930 G. Gentzen presented another calculus (sequent calculus), which is applicable for an automatic proof search. The main idea is to reverse the proof search direction. In Hilbert-style calculus a derivation is performed by going from axioms to the goal formula. In sequent calculus the derivation will be performed by going from the goal formula to axioms.

Sequent calculus operates with sequents instead of formulas.

Definition 17 *If $A_1, A_2, \dots A_n, B_1, B_2, \dots, B_m$ are some formulas and $(n+m) > 0$, then expression $A_1, A_2, \dots A_n \vdash B_1, B_2, \dots, B_m$ is a sequent;*
$A_1, A_2, \dots A_n$ is the sequent's antecedent, and $B_1, B_2, \dots, B_m$ is the sequent's succedent.

Definition 18 *Sequent calculus G is the calculus with axiom $\Gamma', A, \Gamma'' \vdash \Delta', A, \Delta''$ and with the following rules:*

$$\frac{\Gamma', \Gamma'' \vdash \Delta', A, \Delta''}{\Gamma', \neg A, \Gamma'' \vdash \Delta', \Delta''} \ (\neg \vdash) \qquad \frac{\Gamma', A, \Gamma'' \vdash \Delta', \Delta''}{\Gamma', \Gamma'' \vdash \Delta', \neg A, \Delta''} \ (\vdash \neg)$$

$$\frac{\Gamma', A, B, \Gamma'' \vdash \Delta}{\Gamma', A\&B, \Gamma'' \vdash \Delta} \ (\& \vdash) \qquad \frac{\Gamma \vdash \Delta', A, \Delta'' \qquad \Gamma \vdash \Delta', B, \Delta''}{\Gamma \vdash \Delta', A\&B, \Delta''} \ (\vdash \&)$$

$$\frac{\Gamma', A, \Gamma'' \vdash \Delta \qquad \Gamma', B, \Gamma'' \vdash \Delta}{\Gamma', A \vee B, \Gamma'' \vdash \Delta} \ (\vee \vdash) \qquad \frac{\Gamma \vdash \Delta', A, B, \Delta''}{\Gamma \vdash \Delta', A \vee B, \Delta''} \ (\vdash \vee)$$

$$\frac{\Gamma', \Gamma'' \vdash \Delta', A, \Delta'' \qquad \Gamma', B, \Gamma'' \vdash \Delta', \Delta''}{\Gamma', A \to B, \Gamma'' \vdash \Delta', \Delta''} \ (\to\vdash) \qquad \frac{\Gamma', A, \Gamma'' \vdash \Delta', B, \Delta''}{\Gamma', \Gamma'' \vdash \Delta', A \to B, \Delta''} \ (\vdash\to)$$

here A, B - some formulas of the propositional logic,
$\Gamma, \Gamma', \Gamma'', \Delta, \Delta', \Delta''$ - finite (may be empty) sequences of the formulas.

Definition 19 *Tree is a sequent S inference tree in the sequent calculus G (or just sequent S tree) if the following conditions are satisfied:*

- *every tree node contains some sequent,*
- *in the root node there is a sequent S,*
- *if node N contains the sequent S and it has children N_1, N_2 (or only N_1) those contain sequents S_1, S_2 (or only S_1) respectively, then the sequent S is a conclusion and sequents S_1, S_2 (or only S_1) are all premises of some sequent calculus G rule application.*

Definition 20 *Sequent S inference tree in the sequent calculus G is the sequent S derivation tree in the sequent calculus G if every tree leaf contains some axiom of the sequent calculus G.*

Definition 21 *Sequent S is derivable in sequent calculus G if and only if there exists sequent S derivation tree.*

Example 10 *Suppose we have an initial sequent*

$S = p \vee (q\&\neg w) \vdash (\neg q\&\neg w) \vee p$,

then sequent S inference tree may be the following:

$$\dfrac{\dfrac{\overset{\oplus}{p \vdash \neg q\&\neg w, p}}{p \vdash (\neg q\&\neg w) \vee p}\,(\vdash \vee) \qquad \dfrac{\dfrac{\dfrac{q \vdash w, \neg q\&\neg w, p}{q, \neg w \vdash \neg q\&\neg w, p}\,(\neg \vdash)}{q\&\neg w \vdash \neg q\&\neg w, p}\,(\& \vdash)}{q\&\neg w \vdash (\neg q\&\neg w) \vee p}\,(\vdash \vee)}{p \vee (q\&\neg w) \vdash (\neg q\&\neg w) \vee p}\,(\vee \vdash)$$

Inference tree is an intermediate tree obtained during sequent derivation. This tree may not end with axioms. Therefore, if we have inference tree, in general, we cannot say whether an initial sequent S is derivable or not.

Example 11 *Suppose we have an initial sequent*

$S = \neg p \to (q\&\neg w) \vdash (q\&\neg w) \vee p$,

then sequent S derivation tree may be the following:

$$\dfrac{\dfrac{\dfrac{\overset{\oplus}{p \vdash \neg q\&\neg w, p}}{p \vdash (\neg q\&\neg w) \vee p}\,(\vdash \vee)}{\vdash \neg p, (\neg q\&\neg w) \vee p}\,(\vdash \neg) \qquad \dfrac{\dfrac{\dfrac{\overset{\oplus}{q, \neg w \vdash q, p} \qquad \overset{\oplus}{q, \neg w \vdash \neg w, p}}{q, \neg w \vdash q\&\neg w, p}\,(\vdash \&)}{q, \neg w \vdash (q\&\neg w) \vee p}\,(\vdash \vee)}{q\&\neg w \vdash (q\&\neg w) \vee p}\,(\& \vdash)}{\neg p \to (q\&\neg w) \vdash (q\&\neg w) \vee p}\,(\to\vdash)$$

It is a derivation tree, since it is inference tree and every leaf contains an axiom. If we found a derivation tree, we know that an initial sequent S is derivable.

If sequent $A_1, A_2, \dots A_n \vdash B_1, B_2, \dots, B_m$ is derivable, then we know that from the premises $A_1, A_2, \dots A_n$ follows some conclusion $B_1, B_2, \dots, B_m$, but we cannot say which exactly.

In practice, we check if sequent $A_1, A_2, \dots A_n \vdash \varphi$ is derivable or not. In this case, if sequent $A_1, A_2, \dots A_n \vdash \varphi$ is derivable, we know that from the premises $A_1, A_2, \dots A_n$ follows particular conclusion φ.

Theorem 4 *Formula φ is a tautology if and only if sequent $\vdash \varphi$ is derivable in sequent calculus G.*

In sequent calculus, proof search direction is reversed. During derivation we take a goal sequent S and apply rules in bottom-up direction. Such a reverse rule application has some advantages:

- every premise of every rule has less operations then rule conclusion,
- every rule is applied according to a non-atomic formula placed in bottom sequent,

- for the particular sequent, there exists a finite number of possible rule applications (equals to the number of non-atomic formulas in sequent).

It means that such a sequent inference tree construction (applying rules in the bottom-up direction) always terminates. After all, we get a sequent derivation tree (sequent is derivable), or we get an inference tree whose leaves contain only atomic formulas (no rules can be applied).

Every calculus rule satisfies: if all premises of the rule are derivable, then conclusion of the rule is also derivable.

Definition 22 *Sequent calculus rule is invertible if the rule conclusion is derivable if and only if all its premises are derivable.*

Theorem 5 *Every sequent calculus G rule is invertible.*

According to Theorem 5, if we have sequent S inference tree in sequent calculus G, and there exists some leaf, which does not contain an axiom, then sequent S is non-derivable in sequent calculus G. So, we may construct sequent S inference tree in bottom-up direction, and:

- in the case we get derivation tree (all leaves contain axioms) - sequent S is derivable,
- in the case we get some non-axiom leaf containing only atomic formulas - sequent S is non-derivable.

Therefore, we may construct an efficient automatic proof search based on sequent calculus G, since it always terminates, and always return result (sequent is derivable or non-derivable).

There are known some more sequent calculi for propositional logic. These calculi differ in axioms and rules, but all of them uses the main idea of bottom-up rule application. One of such a calculus is Gentzen's Natural Deduction system ([6]). Therefore, sometimes sequent calculi are called Gentzen calculi.

1.5 Resolution Method

Another popular and widely used calculus for propositional logic is Resolution Method. Resolution Method is based on clause deduction system, which operates with formulas having a special shape - clauses. Clause deduction system may be used to determine if a set of clauses is satisfiable or not.

According to Definition 9, a clause is a disjunction of literals $(l_1 \vee l_2 \vee \ldots \vee l_n)$. A clause may contain several, one, or even zero literals. An empty clause (clause consisting of zero literals) is denoted by $\Box$.

Definition 23 *Resolution rule (RR) is:*

$$\frac{C' \vee p \vee C'' \quad D' \vee \neg p \vee D''}{C' \vee C'' \vee D' \vee D''}\ (RR)$$, here C', C'', D', D'' - some (may be empty) clauses.

The resolution rule is a very simple rule and it may be applied for any two clauses if one clause contains some variable p, and the other clause contains the same variable with negation - $\neg p$. In such a case, we say that the resolution rule is applied in respect to variable p. As a result we get a new clause, which consists of all literals from both clauses except literals p and $\neg p$. If the resulting clause contains several occurrences of some literal, then we may simplify the clause and leave only one occurrence of that literal: clause $\neg p \vee w \vee q \vee \neg p \vee q$ is simplified into clause $\neg p \vee w \vee q$.

Definition 24 *We say that clause C is derivable from the set of clauses S, if there exists such a sequence of the clauses, that every clause in this sequence is*

- *from initial set S, or*
- *it is obtained by the resolution rule application from the clauses listed above,*

and this sequence ends with clause C.

If C is derivable from the set of clauses S we will write $S \vdash C$.

Example 12 *Clause $q \vee r$ derivation from the set of the clauses $S = \{\neg p \vee q \vee s, \neg q \vee r, p \vee q \vee r, r \vee \neg s, \neg r \vee q\}$ is:*

1) $\neg p \vee q \vee s$ - from S,

2) $\neg q \vee r$ - from S,

3) $\neg p \vee r \vee s$ - resolution rule from 1 and 2,

4) $p \vee q \vee r$ - from S,

5) $q \vee r \vee s$ - resolution rule from 3 and 4,

6) $r \vee \neg s$ - from S,

7) $q \vee r$ - resolution rule from 5 and 6.

Therefore, we show that $\{\neg p \vee q \vee s, \neg q \vee r, p \vee q \vee r, r \vee \neg s, \neg r \vee q\} \vdash q \vee r$.

Definition 25 *Set of the formulas S is satisfiable, if there exists such an interpretation ν that every formula from set S is true with interpretation ν.*

And vice versa, set of the formulas S is non-satisfiable, if for any interpretation ν holds: at least one formula from the set S is false with interpretation ν.

Example 13 *Set of the formulas $S = \{\neg p\&q, \neg q \to p, \neg p \vee q\}$ is satisfiable since with interpretation ν_2 all formulas are true:*

Interpretation			*Formula*		
	p	q	$\neg p\&q$	$\neg q \to p$	$\neg p \vee q$
ν_1	F	F	F	F	T
ν_2	F	T	T	T	T
ν_3	T	F	F	T	F
ν_4	T	T	F	T	T

Example 14 *Set of the formulas $S = \{p\&\neg q, q \to p, \neg p \vee q\}$ is non-satisfiable since:*

- *with interpretations ν_1 and ν_4 formula $p\&\neg q$ is false,*
- *with interpretation ν_2 formulas $p\&\neg q$ and $q \to p$ are false,*
- *with interpretation ν_3 formula $\neg p \vee q$ is false:*

Interpretation			*Formula*		
	p	q	$p\&\neg q$	$q \to p$	$\neg p \vee q$
ν_1	F	F	F	T	T
ν_2	F	T	F	F	T
ν_3	T	F	T	T	F
ν_4	T	T	F	T	T

Theorem 6 *If non-satisfiable clause C is derivable from the set of clauses S, then S is non-satisfiable.*

Proof.

Let S be a satisfiable set of clauses and $S \vdash C$.

So, there exists an interpretation ν, that $\nu(H) = T$ for every clause $H \in S$.

We will show that every clause derivable from S is true with the same interpretation ν. For this purpose we use mathematical induction according to clause D derivation length l.

If $l = 1$, the D is from the set S, and, therefore, $\nu(D) = T$.

Mathematical induction assumption: $D_1, D_2, D_3, \ldots D_n = D$ is clause D derivation from the set S and $\nu(D_i) = T$ for every $i < m$ ($m \in \mathbb{N}$).

There are 2 possible cases:

- D_m is from S, then $\nu(D_m) = T$, since $D_m \in S$.

- D_m is obtained from $D_j = D'_j \vee p$ and $D_k = D'_k \vee \neg p$ by resolution rule application, and $D_m = D'_j \vee D'_k$.
  $\nu(D_j) = \nu(D'_j \vee p) = T$ and $\nu(D_k) = \nu(D'_k \vee \neg p) = T$, since $j, k < m$.
  If $\nu(p) = T$, then $\nu(D'_k) = T$, and $\nu(D_m) = \nu(D'_j \vee D'_k) = T$.
  If $\nu(p) = F$, then $\nu(D'_j) = T$, and $\nu(D_m) = \nu(D'_j \vee D'_k) = T$.

According to mathematical induction, every clause D derivable from S is true with interpretation ν. Therefore, also $\nu(C) = T$. And we got a contradiction, since C is a non-satisfiable clause.

The only non-satisfiable clause is an empty clause $\Box$. Therefore, if $S \vdash \Box$, then S is non-satisfiable.

Theorem 7 *If set of the clauses S is non-satisfiable, then an empty clause $\Box$ is derivable from S.*

Proof.

Any clause containing both literals p and $\neg p$ may be removed from the set S without loosing $\Box$ derivability, since any clause containing literals p and $\neg p$ is true with every interpretation (for any interpretation ν holds: $\nu(p \vee \neg p \vee C) = T$). For this reason, we consider that set S does not contain such a clause.

We use mathematical induction according to the l - the count of the different variables used in set the S. For example, if $S = \{p \vee \neg q \vee s, \neg p \vee s, \neg s \vee q\}$, then $l = 3$ - set S contains 3 different variables: p, q, s.

If $l = 1$, then it is possible to construct only three different sets of the clauses:

a) $S = \{p\}$,

b) $S = \{\neg p\}$,

c) $S = \{p, \neg p\}$.

Only in the case c), set S is non-satisfiable, and in this case $S \vdash \Box$ (theorem statement is satisfied).

Mathematical induction assumption: if S is non-satisfiable and contains $l < m$ ($m \in \mathbb{N}$) different variables, then an empty clause $\Box$ is derivable from S.

Suppose that set of the clauses S is non-satisfiable and contains $l = m$ different variables. Let p be one of the variables used in S.

We allocate all clauses from the set S into 3 disjointed subsets:

- S_p^+ will contain all clauses from the set S, those contain literal p,
- S_p^- will contain all clauses from the set S, those contain literal $\neg p$,
- S_0 will contain all clauses from the set S, those do not contain neither literal p nor literal $\neg p$.

Then $S = S_p^+ \cup S_p^- \cup S_0$, and $S_p^+ \cap S_p^- \cap S_0 = \emptyset$.

For every clause from S_p^+ and every clause from S_p^- we apply resolution rule in respect to variable p. $at(S_p)$ is a set of such clauses (obtained by resolution rule application).

Now we construct a new set of the clauses $S_1 = S_0 \cup at(S_p)$. Set S_1 does not contain variable p.

We will show that sets S and S_1 are both satisfiable or both non-satisfiable:

- Suppose, that the set of the clauses S is satisfiable.
  There exists an interpretation ν that every clause $H \in S$ is true with interpretation ν (i.e., $\nu(H) = T$, if $H \in S$). Every clause in S_1 is from set S or it is obtained by resolution rule application from clauses from set S . Therefore, every clause in S_1 is true with interpretation ν (see proof of the Theorem 6). So, set of the clauses S_1 is also satisfiable.

- Suppose that set of the clauses S_1 is satisfiable.
  There exists an interpretation ν, that every clause $H \in S_1$ is true with interpretation ν (i.e., $\nu(H) = T$, if $H \in S_1 = S_0 \cup at(S_p)$). Interpretation ν does not define if variable p is $true$ or $false$. We will extend interpretation ν by defining variable p. The new extended interpretation will be ν_1.

  For every clause $H \in S_0$, $\nu_1(H) = T$, since $S_0 \subseteq S_1$.

  Let $S_p^+ = \{C_1 \vee p, C_2 \vee p, \ldots, C_n \vee p\}$, $S_p^- = \{D_1 \vee \neg p, D_2 \vee \neg p, \ldots, D_k \vee \neg p\}$.

  Then
  $$\begin{array}{lllllll} at(S_p) = \{ & C_1 \vee D_1, & C_1 \vee D_2, & C_1 \vee D_3, & \ldots, & C_1 \vee D_k & \\ & C_2 \vee D_1, & C_2 \vee D_2, & C_2 \vee D_3, & \ldots, & C_2 \vee D_k & \\ & \ldots & \ldots & \ldots & \ldots, & \ldots & \\ & C_n \vee D_1, & C_n \vee D_2, & C_n \vee D_3, & \ldots, & C_n \vee D_k & \}. \end{array}$$

  If $\nu(C_i) = F$ for some $i \in \{1, 2, \ldots, n\}$, then we define $\nu_1(p) = T$. Since every clause $C_i \vee D_j \in at(S_p)$ is true with interpretation ν_1, the following clauses are also true: $\nu(C_i \vee D_1) = \nu(C_i \vee D_2) = \ldots = \nu(C_i \vee D_k) = T$. Therefore, $\nu(D_1) = \nu(D_2) = \ldots = \nu(D_k) = T$ and $\nu_1(H) = T$ for every clause $H \in S_p^-$.

  For every clause $H \in S_p^+$, $\nu_1(H) = T$, since $\nu_1(p) = T$.

  If $\nu(C_i) = T$ for every $i \in \{1, 2, \ldots, n\}$, then we define $\nu_1(p) = F$. Therefore, $\nu_1(H) = T$, for every clause $H \in S_p^+$. For every clause $H \in S_p^-$, $\nu_1(H) = T$, since $\nu_1(p) = F$.

  So, set of the clauses S is also satisfiable (with extended interpretation ν_1).

Sets S and S_1 are both satisfiable or both non-satisfiable. Mathematical induction assumption is held for the set S_1 (since S_1 contains $< m$ different variables). So, if S is non-satisfiable, then an empty clause $\Box$ is derivable from S.

Corollary 1 *Set of the clauses S is non-satisfiable if and only if an empty clause $\Box$ is derivable from S.*

Clause deduction system is suitable to determine if a set of the clauses is satisfiable or not. In practice we do not need such an algorithm. We need a method to determine if a conclusion formula follows from premises. For this goal we have the Resolution Method.

It is known that $A_1, A_2, \ldots, A_n \vdash \varphi$ if and only if a set of the formulas $B = \{A_1, A_2, \ldots, A_n, \neg\varphi\}$ is non-satisfiable. Here $A_1, A_2, \ldots, A_n, \varphi$ are some formulas (unnecessary clauses) of propositional logic.

For every formula in B, a conjunctive normal form (CNF) may be found. Every CNF consists of clauses. Let S be a set of all clauses obtained from NCFs (found for formulas from B).

Set of the formulas B is non-satisfiable if and only if a set of the clauses S is non-satisfiable. And the clause deduction system lets us to determine if a set of the clauses S is non-satisfiable.

Resolution method:

1) $A_1, A_2, \ldots, A_n \vdash \varphi$

$\Updownarrow$

2) $B = \{A_1, A_2, \ldots, A_n, \neg\varphi\}$ is non-satisfiable

$\Updownarrow$ (find CNF for every formula in B)

3) $S = \{D_1, D_2, \ldots, D_m\}$ is non-satisfiable

$\Updownarrow$

4) $S \vdash \Box$

Example 15 *We will use Resolution Method to determine if from the premise $\neg p \to (q\&\neg w)$ follows the conclusion $(q\&\neg w) \vee p$.*

$\neg p \to (q\&\neg w) \vdash (q\&\neg w) \vee p$

$\Updownarrow$

$B = \{\neg p \to (q\&\neg w), \neg((q\&\neg w) \vee p)\}$ *is non-satisfiable.*

Find CNF for every formula in B:

$\neg p \to (q\&\neg w) \sim p \vee (q\&\neg w) \sim (p \vee q)\&(p \vee \neg w)$,

$\neg((q\&\neg w) \vee p) \sim \neg(q\&\neg w)\&\neg p \sim (\neg q \vee w)\&\neg p$.

$B = \{\neg p \to (q\&\neg w), \neg((q\&\neg w) \vee p)\}$ *is non-satisfiable*

$\Updownarrow$

$S = \{p \vee q, p \vee \neg w, \neg q \vee w, \neg p\}$ *is non-satisfiable.*

Empty clause $\Box$ derivation from S:

1) $p \vee q$ *- from S,*

2) $\neg p$ *- from S,*

3) q *- resolution rule from 1 and 2,*

4) $p \vee \neg w$ *- from S,*

5) $\neg w$ *- resolution rule from 2 and 4,* *6)* $\neg q \vee w$ *- from* S*,*

7) $\neg q$ *- resolution rule from 5 and 6.*

8) $\Box$ *- resolution rule from 3 and 7.*

$S \vdash \Box \Rightarrow S$ *is non-satisfiable* $\Rightarrow B$ *is non-satisfiable* $\Rightarrow$ $\neg p \rightarrow (q \& \neg w) \vdash (q \& \neg w) \vee p$*.*

Chapter 2

Algorithm Formalizations

In this chapter we present basic algorithm formalizations, those are based on some recursive functions or on the idealistic mathematical machines. These formalizations are the main known successful attempts to describe formally algorithmically computable functions (algorithms).

In this chapter three main algorithm formalizations and their implications are presented:

- Turing machines,
- Partial recursive functions,
- Lambda calculus.

In the Section 2.1 algorithm definition and famous Church's Thesis are discussed. Turing machines, algorithm formalization based on the idealistic mathematical machines, are described in the Section 2.2. Further (in the Section 2.3) finite automata (specific variant of the Turing machine) are presented. Complexity of the algorithms, based on Turing machines, is defined in the Section 2.4.

In the auxiliary Section 2.5 Cantor enumerations of the tuples are introduced. Primitive, general and partial recursive functions are introduced in the Sections 2.6 and 2.7. In the Section 2.8 Ackermann functions are defined. In the Section 2.9, recursive and recursively enumerable sets are discussed. The Halting problem, one of the main problems of the Algorithm Theory, and its implications are presented in the Section 2.10. Universal functions and several important theorems may be found in the Section 2.11. In the Section 2.12, the last algorithm formalization - Lambda calculus is introduced.

2.1 Conception of the Algorithm. The Church's Thesis

We use algorithms in our everyday life. Even if we do not think about them, we use them. We prepear dishes according to recipes, we use instructions to assemble different stuff, we go somewhere using the public transport, or we just want to drive our car. Of course, there are a lot of more specific algorithms we use to solve different (mathematical or non-mathematical) problems. We know algorithms but what can we say about the algorithm itself? What is the main characteristic of the algorithm? What makes particular instructions to become an algorithm?

We can achieve the same result using different algorithms and it is not clear which way is better. Sometimes we even do not know if there exists an algorithm which solves some particular problem at all. In the beginning of the XX century researches started to research algorithms in a more scientific way. One of the main problems was the algorithm definition which was not strict enough. Generally, an intuitive definition of the algorithm is the following:

Definition 26 (Intuitive Algorithm definition) *Algorithm is a strict sequence of the instructions, those allow to solve some (mathematical) problem.*

Unfortunately, this definition does not allow us to answer questions about algorithms. We cannot compare algorithms using such a definition. Therefore, researchers started to attempt to define algorithm in a more formal way. First of all, the main algorithm features were defined. These features are:

1) Algorithm is **discrete**. Algorithm consists of some ordered sequence of actions. The next action is performed only when the previous action was finished. Algorithm actions are called steps.

2) Algorithm is **deterministic**. After finishing one step we know which step shall be performed next.

3) Algorithm steps shall be **elementary**. Algorithm consists of steps, those are simple enough to describe and perform them.

4) Algorithm is **applicable for the set of problems**. In the other words, every algorithm has some input. The same algorithm may be performed with different input.

Most scientists agree that these features are the main algorithm features. There were proposed several algorithm formalizations which satisfy these features. Two main directions are known for algorithm formalization:

1) to create idealistic mathematical machine, which allows to compute any algorithmically computable function,

2) to define formal function set, which consists of every algorithmically computable function.

Intuitively we understand an algorithmically computable function as follows: we know the way how to compute the function value with given parameters. Note that we can define a lot of functions, that are non-algorithmically computable functions. For example, we can define such a function:

$$f(x) = \begin{cases} 1 & \text{, if number } \pi = 3,141592653\ldots \text{ expression contains an } x\text{-length digit row,} \\ & \text{which has only the digit 7 (i.e. } 3,1415\ldots\underbrace{777\ldots77}_{x}\ldots\text{)} \\ 0 & \text{, in the other case.} \end{cases}$$

It is not clear if this function is an algorithmically computable function or not, since we still do not know how to compute it.

Several formalizations of the algorithmically computable functions were introduced. The most famous formalizations were presented in A. Turing, E. Post (idealistic mathematical machine), K. Gödel, A. Church and S. Kleen (formal function set) works. They choose different ways to define algorithmically computable functions, but actually they define the same set of the functions, since the following theorem was proven:

Theorem 8 *Set of the partial recursive functions and set of the functions computable by Turing machines are equal.*

In 1936, A. Church declared his famous thesis:

Theorem 9 (The Church's Thesis) *A set of the algorithmically computable functions and a set of the partial recursive functions are equal.*

A set of the algorithmically computable functions is an informal concept. A set of the recursive functions is a formally defined set of the functions (one of the algorithm formalizations). The Church's thesis is not a theorem. It is a thesis only and, therefore, it cannot be proven. In spite of this, most scientists agree that this thesis is correct. The main arguments are the following:

1) All known algorithm formalizations define the same set of functions (including recursive functions and a set of functions computed by idealistic mathematical machines).

2) Nobody can present an algorithmically computable function, which is not a recursive function (and all recursive functions actually are algorithmically computable functions).

In the later sections, we introduce several algorithm formalizations and various implications of them.

2.2 Turing Machines and Their Variants

In this section we present idealistic (hypothetical) mathematical machine introduced by Alan Turing. Computer is just some kind of the Turing machine which is extended with several additional features. Unfortunately, Turing machine cannot be fully implemented since it uses infinite memory. Therefore, computer is the Turing machine with the restricted capability. What can be computed with Turing machine that also can be computed with a computer, if only you have enough memory. Since, Turing machine can calculate every algorithmically computable function, the saying 'with computer we can calculate everything' is nearly the truth.

Definition 27 *1-tape deterministic Turing machine is a tuple* $< \Sigma, Q, q_0, F, \delta >$*, where*

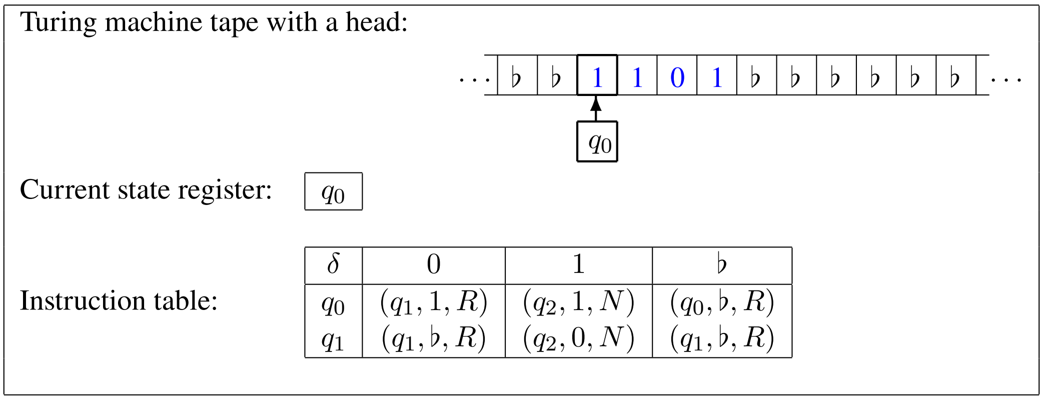
- Σ *- an alphabet, a finite set of symbols; generally,* $\Sigma = \{0, 1, \flat\}$*, where* $\flat$ *means 'Blank cell';*
- Q *- a finite non-empty set of Turing machine states,* $Q = \{q_0, q_1, q_2, \ldots, q_n\}$*;*
- $q_0 \in Q$ *is an initial state;*
- $F \subseteq Q$ *- a set of the final states;*
- δ *- a transition function, and* $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, N\}$*,*
  where L *means 'to the Left',* R *- 'to the Right',* N *- 'No move'.*

We interpret the Turing machine as a physical machine (see Figure 2.1), which:

- Has one infinite tape which is divided into cells. Every cell contains exactly one symbol from alphabet Σ at a particular moment of time.
- Has a head which looks into exactly one cell at a particular time. A head may read and write one symbol into that cell. A head may move by one cell to the left, or to the right.
- Has a register which contains the current state of the machine.
- Has an instruction table, which contains the Turing machine instructions (transition function).

Now we will explain how 1-tape Turing machine works.
Before start:

- The Turing machine input (initial data) is some word from Σ symbols. The input data is placed somewhere on the tape. Other cells contain only the blank symbol ($\flat$).
- A head is placed on the first non-blank symbol from the left in the input tape.


Turing machine tape with a head:

Current state register: q_0

Instruction table:

δ	0	1	$\flat$
q_0	$(q_1, 1, R)$	$(q_2, 1, N)$	$(q_0, \flat, R)$
q_1	$(q_1, \flat, R)$	$(q_2, 0, N)$	$(q_1, \flat, R)$

Figure 2.1: 1-tape deterministic Turing machine

- The Turing machine state is an initial state q_0.

Working:

- The Turing machine performs an instruction according to the Turing machine current state and current symbol.

  The turing machine may change the current machine state, and it may change the symbol of the current cell, and may move a head by one cell to the left or to the right. A performed instruction is defined by the transition function δ.

- After performing one instruction (step), the next instruction (step) is performed according to new Turing machine current state and current symbol.

End of work:

- Turing machine stops working if its current state is one of the final states declared in F.

- The output of the Turing machine is a word of Σ symbols. The output word begins at a current cell and ends just before a first blank cell (in right direction).

Example 16 *Let we have a Turing machine with:*

- *alphabet* $\Sigma = \{0, 1, \flat\}$*,*
- *set of Turing machine states* $Q = \{q_0, q_1, q_2\}$*,*
- *initial state is* q_0*,*
- *set of the final states* $F = \{q_2\}$*,*
- *transition function is defined as follows:*

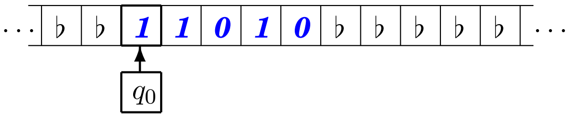
  $$\begin{array}{lll} \delta(q_0, 0) = (q_1, 1, R), & \delta(q_0, 1) = (q_2, 1, N), & \delta(q_0, \flat) = (q_0, \flat, R) \\ \delta(q_1, 0) = (q_1, \flat, R), & \delta(q_1, 1) = (q_2, 0, N), & \delta(q_1, \flat) = (q_1, \flat, R). \end{array}$$

  or presented as a table:

δ	0	1	$\flat$
q_0	$(q_1, 1, R)$	$(q_2, 1, N)$	$(q_0, \flat, R)$
q_1	$(q_1, \flat, R)$	$(q_2, 0, N)$	$(q_1, \flat, R)$

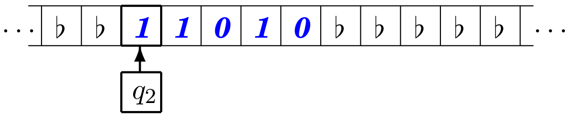
1) *The Turing machine input is a word* 11010*:*

Before work:



The current state is q_0 *and the current symbol is* 1*. Therefore, the Turing machine performs an instruction* $\delta(q_0, 1) = (q_2, 1, N)$*, which instructs to change the current state into* q_2*, to change the current symbol to* 1 *(leave the same), and to leave the head on the same cell.*

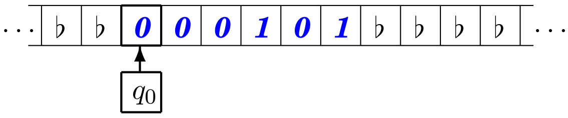
After the first step:



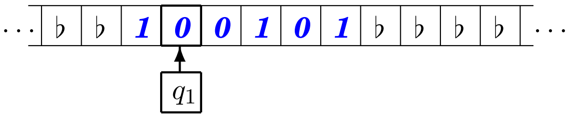
Since the current state is $q_2 \in F$ *(it is a final state), the Turing machine stops working, and its output is a word* 11010 *(the same as the input).*

2) *The Turing machine input is a word* 000101*:*

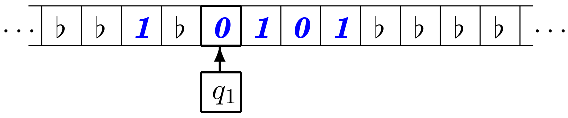
Before work:



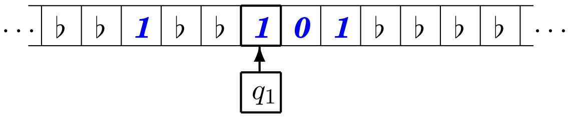
The current state is q_0 *and the current symbol is* 0*. Therefore, the Turing machine performs an instruction* $\delta(q_0, 0) = (q_1, 1, R)$*, which instructs to change the current state into* q_1*, to change the current symbol to* 1*, and to move the head by one cell to the right direction:*



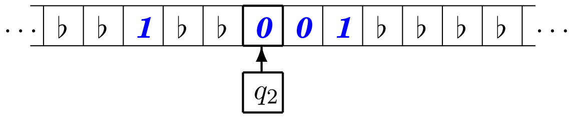
After the Turing machine performs an instruction $\delta(q_1, 0) = (q_1, \flat, R)$*:*



The Turing machine performs an instruction $\delta(q_1, 0) = (q_1, \flat, R)$*:*



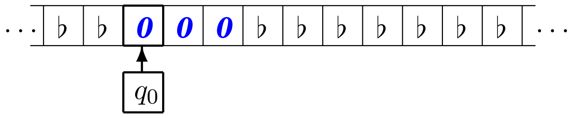
The Turing machine performs an instruction $\delta(q_1, 1) = (q_2, 0, N)$*:*



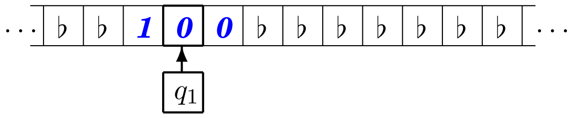
Since the current state is $q_2 \in F$ *(it is a final state), the Turing machine stops working, and its output is a word* 001.

3) *The Turing machine input is a word* 000*:*

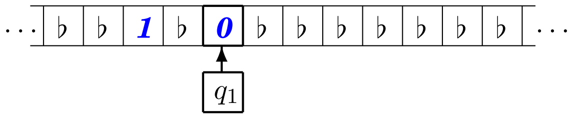
Before work:



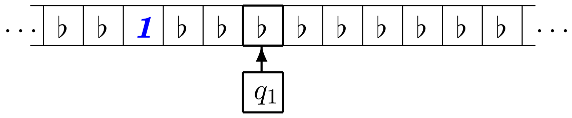
At first, the Turing machine performs an instruction $\delta(q_0, 0) = (q_1, 1, R)$*:*



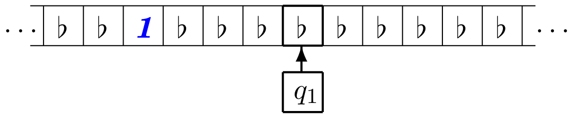
The Turing machine performs an instruction $\delta(q_1, 0) = (q_1, \flat, R)$*:*



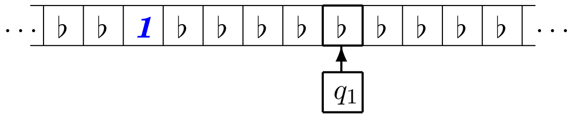
The Turing machine performs an instruction $\delta(q_1, 0) = (q_1, \flat, R)$*:*



The Turing machine performs an instruction $\delta(q_1, \flat) = (q_1, \flat, R)$*:*



The Turing machine performs an instruction $\delta(q_1, \flat) = (q_1, \flat, R)$*:*



and so on...

Since all cells in the right direction contain only the blank symbol $\flat$*, the Turing machine will never stop its work with a given input.*

Definition 28 *We say that the Turing machine is defined with input* x*, if the Turing machine with input data* x *terminates its work and the Turing machine state is one of the final states after all.*

Definition 29 *Language in alphabet Σ is a set of words $L \subseteq \Sigma^*$.*

Definition 30 *The Turing machine $M =< \Sigma, Q, q_0, F, \delta >$ language L is a set of words $w \in \Sigma^*$, with which the Turing machine M is defined (terminates in the final state).*

Definition 31 *We will write $f(x_1^0, x_2^0, \ldots, x_n^0) < \infty$ if function $f(x_1, x_2, \ldots, x_n)$ is defined with arguments $(x_1^0, x_2^0, \ldots, x_n^0)$.*

And we will write $f(x_1^0, x_2^0, \ldots, x_n^0) = \infty$ if function $f(x_1, x_2, \ldots, x_n)$ is undefined with arguments $(x_1^0, x_2^0, \ldots, x_n^0)$.

Definition 32 *Turing machine M computes a partial function $f(x_1, x_2, \ldots, x_n)$ if there exists such a data encoding cod that the following conditions are satisfied:*

- *if $f(x_1, x_2, \ldots, x_n) = y$, then Turing machine M with input $cod(x_1, x_2, \ldots, x_n)$ is defined, and Turing machine M output with this input is $cod(y)$;*
- *if $f(x_1, x_2, \ldots, x_n) = \infty$ (undefined), then Turing machine M with input $cod(x_1, x_2, \ldots, x_n)$ is also undefined.*

Definition 33 *The m-tape deterministic Turing machine is a tuple $< \Sigma, Q, q_0, F, \delta >$, where*

- *Σ - an alphabet, a finite set of symbols; generally, $\Sigma = \{0, 1, \flat\}$, where $\flat$ means 'blank cell';*
- *Q - a finite non-empty set of the Turing machine states, $Q = \{q_0, q_1, q_2, \ldots, q_n\}$;*
- *$q_0 \in Q$ is an initial state;*
- *$F \subseteq Q$ - a set of the final states;*
- *δ - a transition function, and*
  $\delta : Q \times \underbrace{\Sigma \times \Sigma \times \ldots \times \Sigma}_{m} \to Q \times \underbrace{\Sigma \times \Sigma \times \ldots \times \Sigma}_{m} \times \underbrace{\{L, R, N\} \times \ldots \times \{L, R, N\}}_{m}$,
  where L means 'to the Left', R - 'to the Right', N - 'No move'.

We interpret the m-tape deterministic Turing machine as a physical machine (see Figure 2.2), which:

- Has m infinite tapes that are divided into cells. Every cell contains exactly one symbol from alphabet Σ at a particular moment of time. The first tape is called the input tape, the last tape is called the output tape, and all other tapes are called the working tapes.
- Has m heads (one head for every tape). A head looks into exactly one cell at a particular time. A head may read and write one symbol into that cell. Any head may independently move by one cell to the left, or to the right.

- Has a register which contains the current state of the machine. There is only one state for the whole m-tape Turing machine.
- Has an instruction table, which contains the Turing machine instructions (transition function).

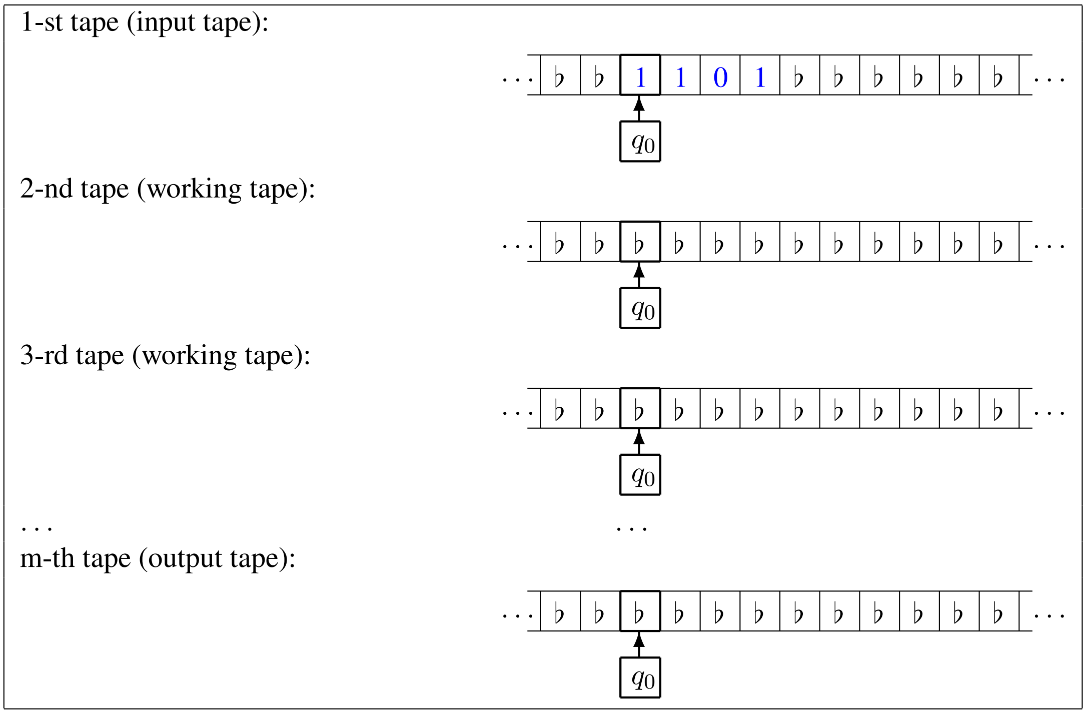


Figure 2.2: m-tape deterministic Turing machine

Now we explain how m-tape Turing machine works.

Before start:

- The Turing machine input (initial data) is some word from Σ symbols. The input data is placed somewhere in the input (first) tape. All other cells (on all tapes) contain only the blank symbol ($\flat$).
- The first head is placed on the first non-blank symbol from the left in the input tape. All other heads are placed on some blank symbols in every other tape.
- The Turing machine state is an initial state q_0.

Working:

- The Turing machine performs an instruction according to the Turing machine current state and current symbols placed in every tape.

  The Turing machine may change the current machine state, and, in every tape, it may change the symbol of the current cell, and may move a head by one cell to the left or to the right independently. The performed instruction is defined by the transition function δ.

- After performing one instruction (step), the next instruction (step) is performed according to the new Turing machine current state and current symbols placed in every tape.

End of work:

- The Turing machine stops working if its current state is one of the final states declared in F.
- The output of the Turing machine is a word of Σ symbols. The output word begins at a current cell and ends just before first blank cell (in right direction). The output word is always placed in the output (the last) tape.

Example 17 *We will construct a* 3*-tape Turing machine, which calculates function* $f(x) = x^*$*, here* x^* *defines the word* x *written in the opposite order.*

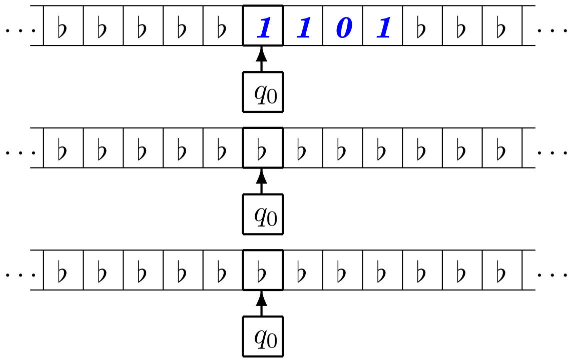
The Turing machine is defined as follows:

- *alphabet* $\Sigma = \{0, 1, \flat\}$*,*
- *set of Turing machine states* $Q = \{q_0, q_1\}$*,*
- *initial state is* q_0*,*
- *set of the final states* $F = \{q_1\}$*,*
- *transition function is defined as follows:*

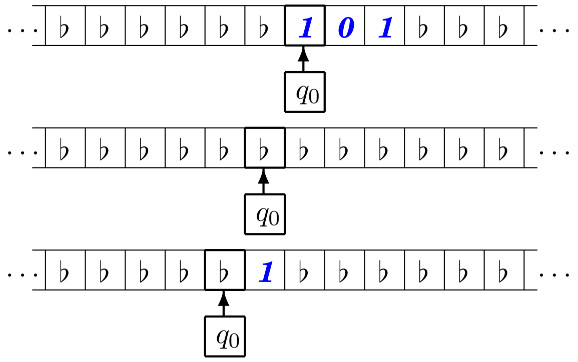
$$\delta(q_0, 0, \flat, \flat) = (q_0, \flat, \flat, 0, R, N, L)$$
$$\delta(q_0, 1, \flat, \flat) = (q_0, \flat, \flat, 1, R, N, L)$$
$$\delta(q_0, \flat, \flat, \flat) = (q_1, \flat, \flat, \flat, N, N, R)$$

Suppose that the Turing machine input is a word 1101*:*

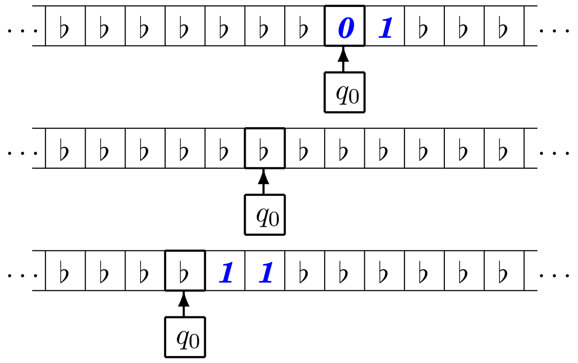
Before work:



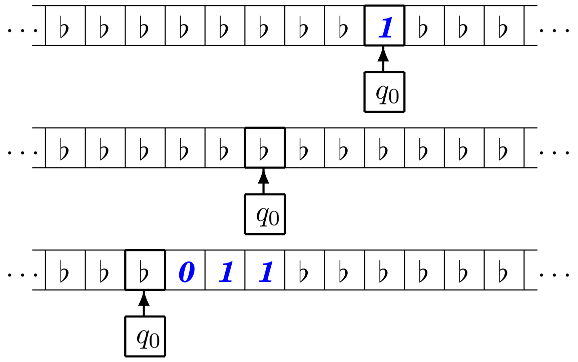
Performing instruction $\delta(q_0, 1, \flat, \flat) = (q_0, \flat, \flat, 1, R, N, L)$:



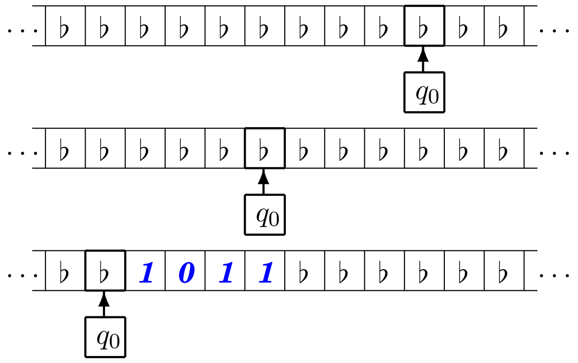
Performing instruction $\delta(q_0, 1, \flat, \flat) = (q_0, \flat, \flat, 1, R, N, L)$:



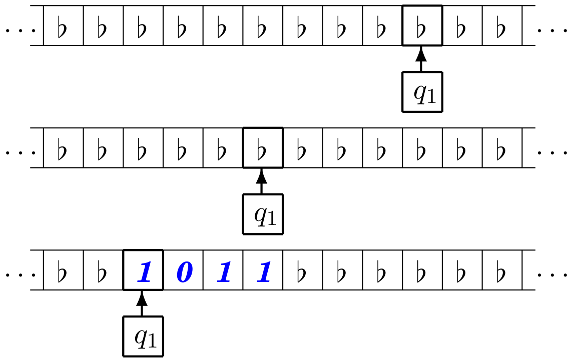
Performing instruction $\delta(q_0, 0, \flat, \flat) = (q_0, \flat, \flat, 0, R, N, L)$:



Performing instruction $\delta(q_0, 1, \flat, \flat) = (q_0, \flat, \flat, 1, R, N, L)$:



Performing instruction $\delta(q_0,\flat,\flat,\flat)=(q_1,\flat,\flat,\flat,N,N,R)$*:*



Since the current state is $q_1 \in F$ *(it is a final state), the Turing machine stops working, and its output is a word* 1011.

The deterministic Turing machine always gets the same result with the same input data. There is another variant of the Turing machine, which may return different result even with the same input data.

Definition 34 *The Non-deterministic* m*-tape Turing machine is such a* m*-tape Turing machine where* δ *is a transition relation:*

$$\delta \subseteq: \left(Q\times\underbrace{\Sigma\times\ldots\times\Sigma}_{m}\right)\times\left(Q\times\underbrace{\Sigma\times\ldots\times\Sigma}_{m}\times\underbrace{\{L,R,N\}\times\ldots\times\{L,R,N\}}_{m}\right).$$

Non-deterministic Turing machine may have several instructions in the same situation. For example, δ may be defined in such a way:
$\delta(q_0,0,\flat,\flat)=(q_0,\flat,\flat,0,R,N,L)$,
$\delta(q_0,0,\flat,\flat)=(q_1,1,0,\flat,R,R,N)$,
$\delta(q_0,1,\flat,\flat)=(q_3,1,\flat,0,N,L,R)$,
$\ldots$

In such a situation, the non-deterministic Turing machine chooses (at random) one of the possible instructions to be performed. Therefore, its result may differ after every execution (using the same input data).

2.3 Finite Automata

Definition 35 *Deterministic finite automaton (DFA) is a 1-tape deterministic Turing machine with the following features:*

- *Transition function satisfies $\delta(q_i, a) = (q_j, a, R)$, where $q_i, q_j \in Q$ are some states of the Turing machine, $a \in \Sigma$ is some symbol from alphabet.*
- *DFA stops working when the blank-symbol ($\flat$) is reached. It does not stop working if the final state is reached (unlike Turing machine).*

Finite automaton is one special Turing machine variant. It is a very simple, but useful machine. Since every instruction has the shape $\delta(q_i, a) = (q_j, a, R)$, the head always moves to the right and, therefore, always reaches a blank cell. DFA always stops its work (that is why it is called finite). Transition function never changes symbols in the cells. Therefore, DFA does not change anything except a state. DFA state obtained after stopping the work is the only result of DFA.

Definition 36 *Word $w \in \Sigma^*$ is accepted by DFA, if DFA stops working having the state $q_i \in F$ (terminates in one of the final states).*

Definition 37 *DFA language $L \subseteq \Sigma^*$ is a set of all words accepted by DFA.*

We say that language L is a DFA language if there exists a DFA, the language of which is L. Therefore, DFA is some kind of a tool, which allows to identify if a particular word belongs to the language or not.

In every DFA, the resulting symbol and direction are fixed in instruction. This lets us store the transition function in a more simple table, which defines only resulting states (symbols and the direction are omitted).

For the same reason, DFA transition function may be presented as a labelled directed graph $D = (Q, A)$. Vertices of the graph (defined by set Q) are states of the DFA. Directed edges (or arrows) of the graph (defined by set A) present instructions defined by the transition function and A contains the directed edge (q_i, q_j) with label a if and only if $\delta(q_i, a) = (q_j, a, R)$.

Example 18 *We will present a DFA, which has language $K = \{10, 11\}$.*

DFA will have 4 states: $Q = \{q_0, q_1, q_2, q_3\}$.

Set of the final states is $F = \{q_2\}$.

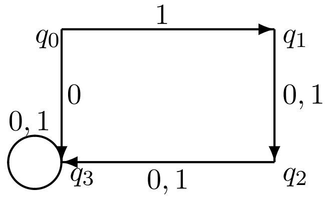
DFA transition function δ in a 'Turing machine' style:

$\delta(q_0, 0) = (q_3, 0, R)$, $\delta(q_1, 0) = (q_2, 0, R)$,
$\delta(q_0, 1) = (q_1, 1, R)$, $\delta(q_1, 1) = (q_2, 1, R)$,
$\delta(q_2, 0) = (q_3, 0, R)$, $\delta(q_3, 0) = (q_3, 0, R)$,
$\delta(q_2, 1) = (q_3, 1, R)$, $\delta(q_3, 1) = (q_3, 1, R)$.

DFA transition function δ in a "‘table"’ style:

	0	1
q_0	q_3	q_1
q_1	q_2	q_2
q_2	q_3	q_3
q_3	q_3	q_3

DFA transition function δ in a "‘graph"’ style:

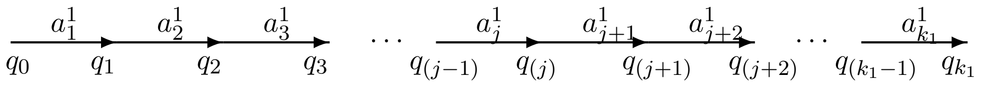


Theorem 10 *Any finite language is a DFA language.*

Proof.

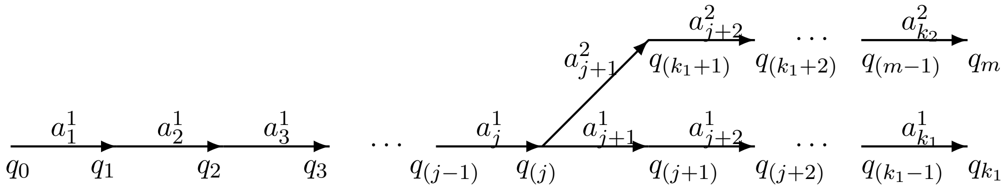
Suppose we have the finite language
$L = \{a_1^1 a_2^1 \dots a_{k_1}^1,\ a_2^2 a_2^2 \dots a_{k_2}^2,\ \dots,\ a_1^n a_2^n \dots a_{k_n}^n\}$

For the first word $a_1^1 a_2^1 \dots a_{k_1}^1$ we can construct a fragment of the graph, which assures, that word $a_1^1 a_2^1 \dots a_{k_1}^1$ is accepted by DFA:



Here $q_{k_1} \in F$.

Take the second word $a_1^2 a_2^2 \dots a_{k_1}^{12} \in K$. Suppose that $a_i^1 = a_i^2$ for all $i \leq j$ for some $j \geq 1$ (i.e. $a_1^2 a_2^2 \dots a_j^2 a_{j+1}^2 \dots a_{k_2}^2 = a_1^1 a_2^1 \dots a_j^1 a_{j+1}^2 \dots a_{k_2}^2$). We may extend the existing fragment of the graph in such a way, that word $a_1^2 a_2^2 \dots a_{k_1}^2$ is also accepted by DFA:



Here $q_{k_1}, q_m \in F$.

Similarly we can expand a fragment of the graph where all words from L are accepted by DFA (for every word one branch is created and one special final state is declared). To complete the graph, all missing arrows go to special vertex q_l just to make sure that there are no more words accepted by DFA.

The constructed graph defines the transition function of the DFA, which accepts only words from language L. Therefore, L is a DFA language.

Theorem 11 *If L is a DFA language, then L complement $\overline{L}$ is also a DFA language.*

Proof.

Suppose DFA language is L and DFA is defined by tuple $< \Sigma, Q, q_0, F, \delta >$ (here Q is a set of the states, and F is a set of the final states.

Then tuple $< \Sigma, Q, q_0, F', \delta >$, where $F' = Q \setminus F$, defines DFA with language $\overline{L}$.

Definition 38 *Suppose $D_1 = (Q_1, A_1)$ and $D_2 = (Q_2, A_2)$ are two labelled directed graphs. Graph Cartesian product $D = D_1 \times D_2$ is a Graph $D = (Q, A)$, where $Q = Q_1 \times Q_2$, and the set of directed edges A is defined as follows:*
an edge $(\ (q_i, q_k),\ (q_j, q_m)\ )$ with label a is presented in A if and only if

- *there is an edge (q_i, q_j) with label a in graph D_1, and*
- *there is an edge (q_k, q_m) with label a in graph D_2.*

Theorem 12 *If L_1 and L_2 are some DFA languages, then sets $A = L_1 \cup L_2$ and $B = L_1 \cap L_2$ are also DFA languages.*

Proof.

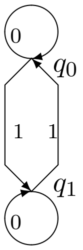
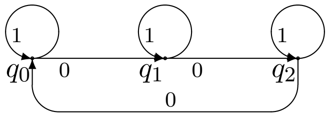
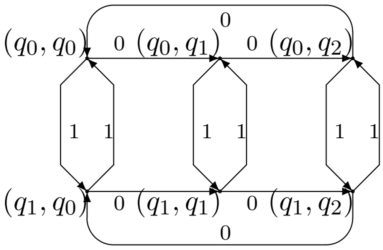
Suppose, that DFA $T_1 =< \Sigma, Q_1, q_0, F_1, \delta_1 >$ language is L_1 and transition function δ_1 is presented by graph $D_1 = (Q_1, A_1)$.

Suppose, that DFA $T_2 =< \Sigma, Q_2, q_0, F_2, \delta_2 >$ language is L_2 and transition function δ_2 is presented by graph $D_2 = (Q_2, A_2)$.

We define DFA $T_A =< \Sigma, Q_1 \times Q_2, (q_0, q_0), F_A, \delta_A >$, whose transition function δ_A is presented by graph $D_A = D_1 \times D_2$ and $F_A = \{(q_i, q_j) : q_i \in F_1$ **or** $q_j \in F_2\}$. Then DFA T_A language is $A = L_1 \cup L_2$. Therefore, $A = L_1 \cup L_2$ is a DFA language.

We define DFA $T_B =< \Sigma, Q_1 \times Q_2, (q_0, q_0), F_B, \delta_B >$, whose transition function δ_B is presented by graph $D_B = D_1 \times D_2$ and $F_B = \{(q_i, q_j) : q_i \in F_1$ **and** $q_j \in F_2\}$. Then DFA T_B language is $B = L_1 \cap L_2$. Therefore, $B = L_1 \cap L_2$ is a DFA language.

Example 19 *Suppose we have two languages:*

<table>
<tr><td><i>L_1 - set of the words $w \in \{0,1\}^*$, whose digit 1 count is divisible by 2.</i></td><td><i>L_2 - set of the words $w \in \{0,1\}^*$, whose digit 0 count is divisible by 3.</i></td></tr>
<tr><td><b><i>DFA</i></b> <i>$T_1 =< \Sigma, Q_1, q_0, F_1, \delta_1 >$,
$Q_1 = \{q_0, q_1\}$, $F_1 = \{q_0\}$,
and transition function δ_1
is presented by graph D_1:</i>

 </td><td><b><i>DFA</i></b> <i>$T_2 =< \Sigma, Q_2, q_0, F_2, \delta_2 >$,
$Q_2 = \{q_0, q_1, q_2\}$, $F_2 = \{q_0\}$,
and transition function δ_2
is presented by graph D_2:</i>

 </td></tr>
<tr><td colspan="2"><i>Then the graphs Cartesian product $D = D_1 \times D_2$ is:</i>


0
(q_0, q_0)
0
(q_0, q_1)
0
(q_0, q_2)
1
1
1
1
1
1
(q_1, q_0)
0
(q_1, q_1)
0
(q_1, q_2)
0

</td></tr>
</table>

***DFA** presented by graph D and with a final state set*
$F_\cup = \{(q_0, q_0), (q_0, q_1), (q_0, q_2), (q_1, q_0)\}$ *is a **DFA** with language* $L_1 \cup L_2$.

***DFA** presented by graph D and with a final state set*
$F_\cap = \{(q_0, q_0)\}$ *is a **DFA** with language* $L_1 \cap L_2$.

2.4 Algorithm Complexity

Definition 39 *Turing machine M time complexity is a function:*
$T_M(n) = max\,\{t(v) : i(v) = n\}$,
here $i(v)$ - length of the input data v,
$t(v)$ - how many steps Turing machine M performs with input data v.

Definition 40 *Turing machine M space complexity is function:*
$S_M(n) = max\,\{s(v) : i(v) = n\}$,
here $i(v)$ - length of the input data v,
$s(v)$ - how many cells Turing machine M uses during its work with input data v.

For the non-deterministic Turing machines, $t(v)$ it is the count of the maximum steps performed by Turing machine M in every possible branch, $s(v)$ it is the count of the maximum cells used by Turing machine M in every possible branch.

Definition 41 *We say that set (problem) A is decidable with Turing machine M if its language is set A.*

We say that time/space deterministic/non-deterministic complexity of the set (problem) A is $f(n)$ if there exists a deterministic/non-deterministic Turing machine with language A and whose time/space complexity $\leq c \cdot f(n)$ ($c > 0$ is some constant).

Definition 42 *Set (problem) A belongs to the complexity class $DTIME(f(n))$ if and only if there exists a multi-tape deterministic Turing machine whose language is A and its time complexity is $f(n)$.*

Definition 43 *Set (problem) A belongs to the complexity class $DSPACE(f(n))$ if and only if there exists a multi-tape deterministic Turing machine whose language is A and its space complexity is $f(n)$.*

Definition 44 *Set (problem) A belongs to the complexity class $NTIME(f(n))$ if and only if there exists a multi-tape non-deterministic Turing machine whose language is A and its time complexity is $f(n)$.*

Definition 45 *Set (problem) A belongs to the complexity class $NSPACE(f(n))$ if and only if there exists a multi-tape non-deterministic Turing machine whose language is A and its space complexity is $f(n)$.*

Defined complexity classes are meaningful only if function $f(n)$ is defined, since they define only the type of the used complexity. We may say that the problem belongs to the complexity class $DSPACE(n^2)$ and to the complexity class $DTIME(2^n)$ at the same time. It means that the problem deterministic space complexity is quadratic, but deterministic time complexity is exponential.

Some concrete complexity classes (with a fixed function $f(n)$) are widely used in the algorithm complexity evaluation. These concrete complexity classes have their own names.

Definition 46 *Complexity classes:*

- $L = DSPACE(log(n))$*;*
- $NL = NSPACE(log(n))$*;*
- $P = DTIME(n^k)$*, and* $k >= 0$ *is some constant;*
- $NP = NTIME(n^k)$*, and* $k >= 0$ *is some constant;*
- $PSPACE = DSPACE(n^k)$*, and* $k >= 0$ *is some constant;*
- $EXP = DTIME(2^{(n^k)})$*, and* $k >= 0$ *is some constant.*

Although these complexity classes are different type complexity classes (some classes are time complexity, some are space complexity), but some interdependence exists. It is known that the following theorem holds.

Theorem 13 $L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP$*,*
and $NL \neq PSPACE$*,* $P \neq EXP$*.*

There is a great and very well known problem called $P - NP$ problem:
Does complexity class $P = NP$, or $P \neq NP$?

Simply speaking, the problem belongs to the complexity class P, if it can be solved in a polynomial time. The problem belongs to the complexity class NP, if it can be solved in an exponential time, but it is enough the polynomial time to check the result. A lot of cryptography algorithms are based on NP complexity problems.

Most scientists think that $P \neq NP$, but this problem is still unsolved.

More about the $P - NP$ problem may be found in L. Fortnow's work [1].

2.5 Pairing Function

It is known that a set of all natural numbers $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ is an enumerable set. The Cartesian product of two enumerable sets is also an enumerable set. Therefore $\mathbb{N} \times \mathbb{N}$ is an enumerable set. Every enumerable set may be written in a sequence having some order. Therefore, all pairs of the natural numbers ($\mathbb{N} \times \mathbb{N}$) may be written in a sequence.

G. Cantor suggested to enumerate all pairs as follows:

$$\overbrace{(0,0)}^{sum=0}, \overbrace{(0,1),(1,0)}^{sum=1}, \overbrace{(0,2),(1,1),(2,0)}^{sum=2}, \overbrace{(0,3),(1,2),(2,1),(3,0)}^{sum=3}, \overbrace{(0,4),(1,3),(2,2),}^{sum=4}\ldots$$

This ordering of pairs of the natural numbers is called the Cantor enumeration.

Definition 47 *Pair (x, y) number in the Cantor enumeration is a function $\alpha_2(x, y)$. Function $\pi_2^1(n)$ is the left member of the n-th pair in the Cantor enumeration. Function $\pi_2^2(n)$ is the right member of the n-th pair in the Cantor enumeration.*

Example 20 *Cantor functions satisfy:*

- $\alpha_2(0, 0) = 0$; $\alpha_2(0, 1) = 1$; $\alpha_2(1, 0) = 2$; $\alpha_2(1, 3) = 11$*;*
- $\pi_2^1(11) = 1$; $\pi_2^2(11) = 3$*;*
- $\pi_2^1(\alpha_2(x, y)) = x$; $\pi_2^2(\alpha_2(x, y)) = y$*;*
- $\alpha_2(\pi_2^1(n), \pi_2^2(n)) = n$*.*

Theorem 14 $\alpha_2(x, y) = \frac{(x+y)^2+3x+y}{2}$*.*

Theorem 15 $\pi_2^1(n) = n \dot{-} \frac{1}{2}\left[\frac{[\sqrt{8n+1}]+1}{2}\right]\left[\frac{[\sqrt{8n+1}]\dot{-}1}{2}\right]$

and $\pi_2^2(n) = \left[\frac{[\sqrt{8n+1}]\dot{-}1}{2}\right] \dot{-} \left(n \dot{-} \frac{1}{2}\left[\frac{[\sqrt{8n+1}]+1}{2}\right]\left[\frac{[\sqrt{8n+1}]\dot{-}1}{2}\right]\right)$*,*

here $[x]$ - is a floor function, and $x \dot{-} y = \begin{cases} x - y & \text{, if } x \geq y \\ 0 & \text{, otherwise.} \end{cases}$

Using the same strategy we also can enumerate n-tuples.

Definition 48 *Cantor functions (for n-tuples enumeration) are:*

- $\alpha_n(x_1, x_2, x_3, \ldots, x_n)$ *is the number of the n-tuple $(x_1, x_2, x_3, \ldots, x_n)$, and* $\alpha_n(x_1, x_2, x_3, \ldots, x_n) = \alpha_2(x_1, \alpha_{n-1}(x_2, x_3, \ldots, x_n))$*;*
- $\pi_n^i(m)$ *is the $i - th$ element in the m-th n-tuple.*

Note, that $\pi_n^i(m)$ may be calculated as follows:

- $\pi_n^1(m) = \pi_2^1(m)$,

- $\pi_n^i(m) = \pi_{n-1}^{i-1}(\pi_2^2(m))$ if $i > 1$.

Example 21 *We calculate* 4*-tuple* $(1, 0, 1, 2)$ *number in the Cantor enumeration:*

$\alpha_4(1,0,1,2) = \alpha_2(1, \alpha_3(0,1,2)) = \alpha_2(1,28) = \frac{(1+28)^2+3\cdot1+28}{2} = 436$

$\alpha_3(0,1,2)) = \alpha_2(0, \alpha_2(1,2)) = \alpha_2(0,7) = \frac{(0+7)^2+3\cdot0+7}{2} = 28$

$\alpha_2(1,2) = \frac{(1+2)^2+3\cdot1+2}{2} = 7.$

Therefore:

$\pi_4^1(436) = \pi_2^1(436) = 1.$

$\pi_4^2(436) = \pi_3^1(\pi_2^2(436)) = \pi_3^1(28) = \pi_2^1(28) = 0.$

$\pi_4^3(436) = \pi_3^2(\pi_2^2(436)) = \pi_3^2(28) = \pi_2^1(\pi_2^2(28)) = \pi_2^1(7) = 1.$

$\pi_4^4(436) = \pi_3^3(\pi_2^2(436)) = \pi_3^3(28) = \pi_2^2(\pi_2^2(28)) = \pi_2^2(7) = 2.$

Every n-tuple has its unique number ($\alpha_n(x_1, x_2, x_3, \ldots, x_n)$). Therefore, we can treat any n-argument function as 1-argument function, which has the only parameter - the number of the n-tuple.

Example 22 *Suppose we have a* 3*-argument function* $f(x, y, z) = x + 2 \cdot y + 3 \cdot z$.

Function $g(w) = \pi_3^1(w) + 2 \cdot \pi_3^2(w) + 3 \cdot \pi_3^3(w)$
calculates the same results as function $f(x, y, z)$ *if only* $w = \alpha_3(x, y, z)$*:*

$g(w) = g(\alpha_3(x,y,z)) = \pi_3^1(\alpha_3(x,y,z)) + 2 \cdot \pi_3^2(\alpha_3(x,y,z)) + 3 \cdot \pi_3^3(\alpha_3(x,y,z)) = x + 2 \cdot y + 3 \cdot z = f(x,y,z).$

Formally, we will write $g(\alpha_n(x_1, x_2, x_3, \ldots, x_n)) = f(x_1, x_2, x_3, \ldots, x_n)$ to denote 1-argument function $g(w)$, which calculates the same results as n-argument function $f(x_1, x_2, x_3, \ldots, x_n)$.

2.6 Primitive Recursive Functions

Definition 49 *Function $f(x_1, x_2, \dots, x_n)$ is obtained from functions*
$h(x_1, x_2, \dots, x_m)$ *and*
$g_1(x_1, x_2, \dots, x_n),\ g_2(x_1, x_2, \dots, x_n),\ \dots,\ g_m(x_1, x_2, \dots, x_n)$
according to the composition (or superposition) operator if

$$f(x_1, x_2, \dots, x_n) = h(\ g_1(x_1, x_2, \dots, x_n),\ g_2(x_1, x_2, \dots, x_n),\ \dots,\ g_m(x_1, x_2, \dots, x_n)\ ).$$

Example 23 *There are given functions: $f(x, y) = x + y$ and $g(x, y) = x \cdot y$.*

Function $h(x, y, z) = (x+z)\cdot(x+y\cdot z) = f(x, z)\cdot(x+g(y, z)) = f(x, z)\cdot f(x, g(y, z)) = g(\ f(x, z),\ f(x, g(y, z))$ is obtained from $f(x, y)$ and $g(x, y)$ according to the composition operator applied several times.

Definition 50 *Function $f(x_1, x_2, \dots, x_n)$ is obtained from functions*
$g(x_1, x_2, \dots, x_{n-1})$ *and*
$h(x_1, x_2, \dots, x_{n-1}, x_n, x_{n+1})$
according to the primitive recursion operator if:

- $f(x_1, x_2, \dots, x_{n-1}, 0) = g(x_1, x_2, \dots, x_{n-1})$,
- $f(x_1, x_2, \dots, x_{n-1}, y+1) = h(x_1, x_2, \dots, x_{n-1}, y,\ f(x_1, x_2, \dots, x_{n-1}, y)\ )$.

Example 24 *Function $f(x, y, z)$ is obtained according to the primitive recursion operator from $g(x, y) = x + 2 \cdot y$ and $h(x, y, z, w) = (x + y) \cdot (z + w)$.*

We will calculate $f(1, 2, 3)$:
$f(1, 2, 3) = h(1, 2, 2, f(1, 2, 2)) = h(1, 2, 2, 48) = (1 + 2) \cdot (2 + 48) = 150$*, since*
$f(1, 2, 2) = h(1, 2, 1, f(1, 2, 1)) = h(1, 2, 1, 15) = (1 + 2) \cdot (1 + 15) = 48$*, since*
$f(1, 2, 1) = h(1, 2, 0, f(1, 2, 0)) = h(1, 2, 0, 5) = (1 + 2) \cdot (0 + 5) = 15$*, since*
$f(1, 2, 0) = g(1, 2) = 1 + 2 \cdot 2 = 5$.

Definition 51 *Set of the primitive recursive functions (denoted by $PrRF$) is the smallest set of the functions, which contains basic functions:*

- *Zero function:* 0,
- *Successor function:* $s(x) = x + 1$,
- *Projection function:* $pr_n^i(x_1, x_2, \dots, x_n) = x_i$,

and which (set) is closed under the composition and primitive recursion operators.

Definition 52 *Function $f(x_1, x_2, \dots, x_n)$ is a total function if it is defined with any arguments.*

Since, basic primitive recursive functions are total functions and the composition and primitive recursion operators preserve totalness, the following theorem is held:

Theorem 16 *If $f(x_1, x_2, \ldots, x_n) \in PrRF$, then it is a total function.*

Example 25 *We will show that $f(x,y) = x + y$ is a primitive recursive function.*

Suppose that $f(x,y)$ is obtained by the primitive recursion operator from functions $g(x)$ and $h(x,y,z)$.

According to the primitive recursion operator definition:
$f(x,0) = g(x)$, and, on the other hand, from the function definition
$f(x,0) = x + 0 = x$.

Therefore, $g(x) = x$.

According to the primitive recursion operator definition:
$f(x, y+1) = h(x,y,f(x,y))$, and, on the other hand, from the function definition
$f(x,y+1) = x + (y+1) = (x+y) + 1 = f(x,y) + 1$.

Therefore, $h(x,y,f(x,y)) = f(x,y) + 1$, and the simplest function, which satisfies this equation is $h(x,y,z) = z + 1$.

We know that $f(x,y)$ is obtained by the primitive recursion operator from functions $g(x) = x$ and $h(x,y,z) = z + 1$.

$g(x) = x = pr_1^1(x) \in PrRF$, since it is a basic function.

$h(x,y,z) = z + 1 = pr_3^3(x,y,s(z)) \in PrRF$, since it is obtained from basic functions $pr_3^3(x,y,z), s(x)$ by the composition operator.

Since $g(x), h(x,y,z) \in PrRF$ and $f(x,y)$ is obtained by the primitive recursion operator from functions $g(x)$ and $h(x,y,z)$, we have proved that $f(x,y) \in PrRF$.

In a similar manner, we may find more primitive recursive functions. Some of them are widely used (see Theorem 17).

Theorem 17 *The following functions are primitive recursive functions:*

- *constant: $f = k$, and k is a constant,*
- *constant addition: $f(x) = x + k$, and k is a constant,*
- *sign function:* $sg(x) = \begin{cases} 1 & \text{, if } x > 0 \\ 0 & \text{, if } x = 0 \end{cases}$,
- *sign function:* $\overline{sg}(x) = \begin{cases} 0 & \text{, if } x > 0 \\ 1 & \text{, if } x = 0 \end{cases}$,
- *subtraction:* $f(x,y) = x \dot{-} y = \begin{cases} x - y & \text{, if } x \geq y \\ 0 & \text{, otherwise} \end{cases}$,
- *addition: $f(x,y) = x + y$,*
- *multiplication: $f(x,y) = x \cdot y$,*
- *absolute difference: $f(x,y) = |x - y|$,*
- *floor of the division: $f(x,y) = [x/y]$,*

- *modulus:* $f(x, y) = x \, mod \, y$,
- *Cantor functions:* $\alpha_n(x_1, x_2, \ldots, x_n), \pi_n^1(x), \pi_n^2(x), \ldots, \pi_n^n(x)$.

Theorem 18 *If* $\varphi(x_1, x_2, \ldots, x_n) \in PrRF$*, then*
$f(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{x_n} \varphi(x_1, x_2, \ldots, x_{n-1}, i) \in PrRF$.

Proof.

Suppose that $f(x_1, x_2, \ldots, x_n)$ is obtained by the primitive recursion operator from functions $g(x_1, x_2, \ldots, x_{n-1})$ and $h(x_1, x_2, \ldots, x_n, x_{n+1})$.

According to the primitive recursion operator definition:
$f(x_1, x_2, \ldots, x_{n-1}, 0) = g(x_1, x_2, \ldots, x_{n-1})$, and, on the other hand,
$f(x_1, x_2, \ldots, x_{n-1}, 0) = \sum_{i=0}^{0} \varphi(x_1, x_2, \ldots, x_{n-1}, i) = \varphi(x_1, x_2, \ldots, x_{n-1}, 0)$.

Therefore, $g(x_1, x_2, \ldots, x_{n-1}) = \varphi(x_1, x_2, \ldots, x_{n-1}, 0)$.

According to the primitive recursion operator definition:
$f(x_1, \ldots, x_{n-1}, y + 1) = h(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y))$, and
$f(x_1, \ldots, x_{n-1}, y + 1) = \sum_{i=0}^{y+1} \varphi(x_1, \ldots, x_{n-1}, i) =$
$= \sum_{i=0}^{y} \varphi(x_1, \ldots, x_{n-1}, i) + \varphi(x_1, \ldots, x_{n-1}, y + 1) =$
$= f(x_1, \ldots, x_{n-1}, y) + \varphi(x_1, \ldots, x_{n-1}, y + 1)$.

Therefore,
$h(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y)) = f(x_1, \ldots, x_{n-1}, y) + \varphi(x_1, \ldots, x_{n-1}, y + 1)$,
and the simplest function which satisfies this equation is
$h(x_1, \ldots, x_{n-1}, x_n, x_{n+1}) = x_{n+1} + \varphi(x_1, \ldots, x_{n-1}, x_n + 1)$.

$g(x_1, x_2, \ldots, x_{n-1}) = \varphi(x_1, x_2, \ldots, x_{n-1}, 0) \in PrRF$, since it is obtained from primitive recursive functions $\varphi(x_1, x_2, \ldots, x_{n-1}, x_n), 0$ by the composition operator.

$h(x_1, \ldots, x_{n-1}, x_n, x_{n+1}) = x_{n+1} + \varphi(x_1, \ldots, x_{n-1}, s(x_n)) \in PrRF$, since it is obtained from known primitive recursive functions $x + y, s(x), \varphi(x_1, x_2, \ldots, x_{n-1}, x_n)$ by the composition operator.

Since $g(x_1, x_2, \ldots, x_{n-1}), h(x_1, \ldots, x_{n-1}, x_n, x_{n+1}) \in PrRF$, and $f(x_1, x_2, \ldots, x_n)$ is obtained by the primitive recursion operator from them, we have proved that $f(x_1, x_2, \ldots, x_n) \in PrRF$.

Definition 53 *If functions* $\varphi_i(x_1, \ldots, x_n) \in PrRF$ *and* $\beta_j(x_1, \ldots, x_n) \in PrRF$*, for every* $i = 1, \ldots, s, s + 1$ *and every* $j = 1, \ldots, s$*, then*

$$f(x_1, \ldots, x_n) = \begin{cases} \varphi_1(x_1, \ldots, x_n) & \text{, if } \beta_1(x_1, \ldots, x_n) = 0 \\ \ldots & \\ \varphi_s(x_1, \ldots, x_n) & \text{, if } \beta_s(x_1, \ldots, x_n) = 0 \\ \varphi_{s+1}(x_1, \ldots, x_n) & \text{, otherwise} \end{cases} \in PrRF.$$

From the set of all primitive recursive functions we may take a part of them – set of all 1-argument primitive recursive functions. All these functions may be expressed using the composition and primitive recursion operators from the basic primitive recursive functions. It is known that every 1-argument primitive recursive function may be expressed (additionally) using only other 1-argument primitive recursive functions. For this reason we need another operator.

Definition 54 1-*argument function* $f(x)$ *is obtained from function* $h(x)$ *according to the iteration operator if:*

- $f(0) = 0$,
- $f(y+1) = h(\ f(y)\ )$.

We will write $f(x) = h(x)^I$ *to denote this.*

The iteration operator is a simpler kind of the primitive recursive operator adapted only for 1-argument functions.

Definition 55 1-*argument function* $f(x)$ *is obtained from functions* $g(x)$ *and* $h(x)$ *according to the addition operator if* $f(x) = g(x) + h(x)$.

Theorem 19 *The set of* 1-*argument primitive recursive functions is the smallest set of the functions which contains basic functions:*

- $s(x) = x + 1$,
- $q(x) = x \dot{-} [\sqrt{x}]^2$,

and which (set) is closed under the composition, addition and iteration operators.

2.7 Partial Recursive Functions

Definition 56 *Function $f(x_1, x_2, \ldots, x_n)$ is obtained from function $g(x_1, x_2, \ldots, x_n)$ according to the minimisation operator if the following conditions hold:*

- *if there exists $y \in \mathbb{N}$ that $g(x_1, x_2, \ldots, x_{n-1}, y) = x_n$ and $g(x_1, x_2, \ldots, x_{n-1}, i) \neq x_n$ and $g(x_1, x_2, \ldots, x_{n-1}, i) < \infty$ (defined) for every $0 \leq i < y$, $i \in \mathbb{N}$, then $f(x_1, x_2, \ldots, x_n) = y$,*

- *otherwise $f(x_1, x_2, \ldots, x_n) = \infty$ (undefined).*

We will write $f(x_1, x_2, \ldots, x_n) = \mu_y(\ g(x_1, x_2, \ldots, x_{n-1}, y) = x_n\ )$ to denote this.

Simply speaking, a function defined using the minimisation operator returns the smallest natural number y, which satisfies the given equation written in the parenthesis. If an equation does not have natural solutions, the function is undefined.

The minimisation operator is used to define algorithmically computable functions. Therefore, there must be a strict algorithm to compute a function obtained according to the minimisation operator. We do not know how to solve any equation. Fortunately, we do not need this; we only need to find the smallest natural root of the given equation. We can do this using the following algorithm.

The algorithm to compute the function $f(x_1, x_2, \ldots, x_n)$ value if $f(x_1, x_2, \ldots, x_n) = \mu_y(\ g(x_1, x_2, \ldots, x_{n-1}, y) = x_n\ )$:

- check $y = 0$: if $g(x_1, x_2, \ldots, x_{n-1}, 0) = x_n$, then $f(x_1, x_2, \ldots, x_n) = 0$, if not

- check $y = 1$: if $g(x_1, x_2, \ldots, x_{n-1}, 1) = x_n$, then $f(x_1, x_2, \ldots, x_n) = 1$, if not

- check $y = 2$: if $g(x_1, x_2, \ldots, x_{n-1}, 2) = x_n$, then $f(x_1, x_2, \ldots, x_n) = 2$, if not

  $\ldots$

If there exists $y \in \mathbb{N}$ satisfying the given equation, the defined algorithm will find it. If such an y does not exist, the defined algorithm will work infinitely.

Note that there are 2 possible cases when the function obtained according to the minimisation operator ($f(x_1, x_2, \ldots, x_n) = \mu_y(\ g(x_1, x_2, \ldots, x_{n-1}, y) = x_n\ )$) is undefined:

- if $g(x_1, x_2, \ldots, x_{n-1}, y) \neq x_n$ for every $y \in \mathbb{N}$,

- if there exists $w \in \mathbb{N}$ that $g(x_1, x_2, \ldots, x_{n-1}, w) = \infty$ (undefined) and $g(x_1, x_2, \ldots, x_{n-1}, i) \neq x_n$ for every $0 \leq i < w$, $i \in \mathbb{N}$.

Note that according to the second case, function may be undefined even if there exists $y \in \mathbb{N}$ satisfying the given equation. This happens if there is such a $w \in \mathbb{N}$, $w < y$, that the given equation is undefined with w.

Definition 57 *Set of the partial recursive functions (denoted by $PaRF$) is the smallest set of the functions, which contains the basic functions:*

- *Zero function:* 0,
- *Successor function:* $s(x) = x + 1$,
- *Projection function:* $pr_n^i(x_1, x_2, \ldots, x_n) = x_i$,

and which (set) is closed under the composition, primitive recursion and minimisation operators.

From the definition we see that $PrRF \subseteq PaRF$. Since every primitive recursive function is a total function and a partial function may not be a total function, we get that $PrRF \subset PaRF$.

Example 26 *We will show that a partial subtraction function*
$$f(x,y) = x - y = \begin{cases} x - y & \text{, if } x \geq y \\ \infty & \text{, otherwise} \end{cases}$$
is a partial recursive function.

$f(x,y) = x-y = \mu_z(y+z = x) \in PaRF$*, since it is obtained from known primitive recursive function* $x + y$ *by composition and minimisation operators.*

To get confidence, we will calculate $f(7,5)$ *and* $f(3,4)$.

$f(7,5) = 7 - 5 = \mu_z(\,5 + z = 7\,)$ *and*
if $z = 0$, *then* $5 + 0 = 5 \neq 7$, *so*
if $z = 1$, *then* $5 + 1 = 6 \neq 7$, *so*
if $z = 2$, *then* $5 + 2 = 7 = 7$, *and, therefore,* $f(7,5) = 2$.

$f(3,4) = 3 - 4 = \mu_z(\,4 + z = 3\,)$ *and*
if $z = 0$, *then* $4 + 0 = 4 \neq 3$, *so*
if $z = 1$, *then* $4 + 1 = 5 \neq 3$, *so*
if $z \geq 2$, *then* $4 + z \geq 6 \neq 3$, *and, therefore,* $f(3,4) = \infty$.

According to Church's Thesis (Theorem 9), a partial recursive functions is a good algorithm formalization, since a set of the algorithmically computable functions and set of the partial recursive functions are equal.

Definition 58 *A set of the general recursive functions (denoted by $GeRF$) is a set of all partial recursive functions, that are total functions.*

Theorem 20 $PrRF \subseteq GeRF \subset PaRF$.

Proof.

The proof goes straightforward from the definitions of the primitive, partial and general recursive functions.

Later, we will show that $PrRF \neq GeRF$. Therefore, $PrRF \subset GeRF \subset PaRF$.

2.8 Ackermann Functions

In this section, we will show that there exists a general recursive function (which is also a total partial recursive function), which is not a primitive recursive function. In other words, we will show that even if a function is total, the minimisation operator may be inevitable.

The main idea is to define the total function, which grows faster than any primitive recursive function.

Analyse these well known functions:

$B_0(a, x) = a + x,$

$B_1(a, x) = a \cdot x,$

$B_2(a, x) = a^x.$

It is known that $B_1(a, x)$ grow faster than $B_0(a, x)$ and $B_2(a, x)$ grows faster than $B_1(a, x)$.

We may notice, that the following equations hold:

$B_1(a, x + 1) = a \cdot (x + 1) = a + a \cdot x = a + B_1(a, x) = B_0(a, B_1(a, x)),$

$B_2(a, x + 1) = a^{x+1} = a \cdot a^x = a \cdot B_2(a, x) = B_1(a, B_2(a, x)).$

We may extend this to get functions that grow even more faster.

Definition 59 *Ackermann functions are:*

$B_{n+1}(a, x + 1) = B_n(a, B_{n+1}(a, x))$, $B_{n+1}(a, 1) = a$ *and* $B_{n+1}(a, 0) = 1$, *if* $n \geq 1$.

$B_1(a, x) = a \cdot x$, $B_0(a, x) = a + x$.

Definition 60 *A version of Ackermann function with* $a = 2$ *is function:*

$A(n, x) = B_n(2, x).$

Further we will write just Ackermann function $A(n, x)$ to define the version of Ackermann function with $a = 2$.

From the definition we get the main features of the Ackermann function $A(n, x)$:

Theorem 21 *The following equations hold:*

1) $A(n + 1, x + 1) = A(n, A(n + 1, x))$,

2) $A(n, 1) = 2$, *if* $n \geq 1$,

3) $A(n, 0) = 1$, *if* $n \geq 2$,

4) $A(1, x) = 2 \cdot x$,

5) $A(0, x) = 2 + x$.

Theorem 22 *If* $n, x \geq 2$, *then the following inequalities hold:*

1) $A(n, x) \geq 2^x$,

2) $A(n + 1, x) > A(n, x)$,

3) $A(n, x+1) > A(n, x)$*,*

4) $A(n+1, x) > A(n, x+1)$*.*

These inequalities show that an increase of any argument also increases the function value, an increase of the first argument will increase the function value more, than the increase of the second argument.

Definition 61 *Function $f(x)$ majorizes function $g(x)$ (or $g(x)$ is majorized by function $f(x)$) if there exists such a $n_0 \in \mathbb{N}$, that $g(x) < h(x)$ for every $x > n_0$.*

Theorem 23 *For every 1-argument function $f(x) \in PrRF$ there exists such a $n \in \mathbb{N}$, that $f(x) < A(n, x)$, if only $x \geq 2$.*

Proof.

According to Theorem 19, every 1-argument primitive recursive function $f(x)$ may be expressed through $s(x) = x+1$, $q(x) = x \dot{-} [\sqrt{x}]^2$ and the addition, composition and iteration operators.

We use mathematical induction according to the function expression through functions $s(x)$ and $q(x)$.

If $f(x) = s(x)$, then $f(x) = x+1 < 2^x = A(2, x)$, and, therefore, there exists $n = 2 \in \mathbb{N}$, that $f(x) < A(n, x)$ (if $x \geq 2$).

If $f(x) = q(x)$, then $f(x) = x \dot{-} [\sqrt{x}]^2 \leq x < 2^x = A(2, x)$, and, therefore, there exists $n = 2 \in \mathbb{N}$, that $f(x) < A(n, x)$ (if $x \geq 2$).

Suppose, that for functions $g(x), h(x) \in PrRF$ there exists $n_1, n_2 \in \mathbb{N}$, that $g(x) < A(n_1, x)$ and $h(x) < A(n_2, x)$.

- If $f(x) = g(x) + h(x)$ (the addition operator). We define $n' = max\,\{2, n_1, n_2\}$. With $n = n'+1 = max\,\{2, n_1, n_2\} + 1 \in \mathbb{N}$, we have that $f(x) < A(n, x)$, since $f(x) = g(x) + h(x) < A(n_1, x) + A(n_2, x) \leq 2 \cdot A(n', x) = A(1, A(n', x)) <$ $< A(n'-1, A(n', x)) = A(n', x+1) < A(n'+1, x) = A(n, x)$, if $x \geq 2$.

- If $f(x) = g(h(x))$ (the composition operator). We define $n' = max\,\{n_1, n_2\}$. With $n = n'+2 = max\,\{n_1, n_2\} + 2 \in \mathbb{N}$, we have that $f(x) < A(n, x)$, since $f(x) = g(h(x)) < A(n_1, h(x)) \leq A(n_1, A(n_2, x)) < A(n_1, A(n'+1, x)) \leq$ $\leq A(n', A(n'+1, x)) = A(n'+1, x+1) < A(n'+2, x) = A(n, x)$, if $x \geq 2$.

- If $f(x) = g(x)^I$ (the iteration operator). We use (inner) mathematical induction to show that with $n = n_1 + 1 \in \mathbb{N}$ we have that $f(x) < A(n, x)$.

  If $x = 0$, then $f(x) = 0$ and $f(x) = 0 \leq g(x) < A(n_1, x) < A(n_1+1, x) = A(n, x)$.

  Suppose that $f(x) < A(n_1+1, x)$ for every $x \leq y \in \mathbb{N}$.
  According to our assumption we have that $f(y) < A(n_1+1, y)$.
  If $x = y+1$, then
  $f(y+1) = g(f(y)) < A(n_1, f(y)) \leq A(n_1, A(n_1+1, y)) = A(n_1+1, y+1)$, if $y \geq 2$.
  Therefore, $f(x) < A(n_1+1, x) = A(n, x)$ for every $x \geq 2$.

According to mathematical induction we have proved that if $f(x) \in PrRF$ there exists such a $n \in \mathbb{N}$ that $f(x) < A(n, x)$ if $x \geq 2$.

Theorem 24 *There exists a general recursive function which is not a primitive recursive function.*

Proof.

Analyze function $h(x) = A(x, x)$. $h(x) \in GeRF$, since it is obtained from Ackermann function by composition (Ackermann function is a total and algorithmically computable function).

Take any 1-argument primitive recursive function $f(x) \in PrRF$.

According to Theorem 23, there exists $n \in \mathbb{N}$ that $f(x) < A(n, x)$ if $x \geq 2$.
Then $f(n + x) < A(n, n + x) < A(n + x, n + x) = h(n + x)$, if $x \geq 2$.

Therefore, there exists $n' = n + 2 \in \mathbb{N}$ that $f(x) < h(x)$, if $x \geq n'$.

We have got that $h(x)$ majorizes any primitive recursive function $f(x)$. It is evident that $h(x)$ cannot be majorized by itself, so $h(x) \notin PrRF$.

2.9 Recursive and Recursively Enumerable Sets

In this section, we analyse one of the main problems of the Algorithm theory: how to determine if a given set contains a particular element or not? Does an algorithm (which determines the element existence in the set) exists for any set? Can we implement these algorithms in practice?

Definition 62 *Characteristic function of the set A is a function*

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

Definition 63 *Set A is a recursive (or decidable) set if its characteristic function is algorithmically computable and total.*

It means, if set A is a recursive set, then there is an algorithm, which may determine if the set contains a particular element or not, and it always terminates.

Unfortunately, for some sets we cannot find such algorithms. However, (for such a set) there is an algorithm, which satisfies:

- if the set contains a particular element, then algorithm terminates and it asserts, that the set contains that element,
- if the set does not contain the element, then the algorithm does not terminate at all.

In other words, we have only a partial algorithm to determine if a set contains the element or not. Such sets are called recursively enumerable sets. We present three equivalent definitions:

Definition 64 *Set A is a recursively enumerable set (or semidecidable) if there exists a partial recursive function whose domain is the set A.*

Definition 65 *A non-empty set A is a recursively enumerable set if there exists a primitive recursive function whose range (or image) is set A.*

Definition 66 *Set A is a recursively enumerable set if there exists a primitive recursive function $f(a, x)$ that satisfies:*

equation $f(a, x) = 0$ has solution if and only if element $a \in A$.

Theorem 25 *Definitions 64, 65 and 66 are equivalent for a non-empty set.*

Proof.

We will present partial proof only: from Definition 65 to Definition 64,
from Definition 65 to Definition 66 and from Definition 66 to Definition 65.

(65 ⇒ 64) Suppose that the range of the function $h(x) \in PrRF$ is set A (Definition 65). So, $A = \{h(0), h(1), h(2), \ldots\}$.

Function $f(x) = \mu_y(\ h(y) = x\ ) \in PaRF$, since it is obtained from a primitive recursive function $h(x)$ according to the minimisation operator.

- If $a \in A = \{h(0), h(1), h(2), \ldots\}$, then there exists $y \in \mathbb{N}$, that $h(y) = a$, and, therefore, $f(a) = \mu_y(\ h(y) = a\ ) < \infty$ (defined).
- If $b \notin A = \{h(0), h(1), h(2), \ldots\}$, then, for every $i \in \mathbb{N}$, we have $h(i) \neq a$, and, therefore, $f(b) = \mu_y(\ h(y) = b\ ) = \infty$ (undefined).

Therefore, the domain of the function $f(x)$ is set A, and set A is a recursively enumerable set according to Definition 64.

(65 ⇒ 66) Suppose that the range of the function $h(x) \in PrRF$ is set A (Definition 65). So, $A = \{h(0), h(1), h(2), \ldots\}$.

Function $f(a, x) = |h(x) - a| \in PrRF$, since it is obtained from primitive recursive functions $h(x), |x - y|$ according to the composition operator.

Analyse equation $f(a, x) = 0 \Rightarrow |h(x) - a| = 0$.

- If $a \in A = \{h(0), h(1), h(2), \ldots\}$, then there exists $x_0 \in \mathbb{N}$, that $h(x_0) = a$, and, therefore, $f(a, x_0) = |h(x_0) - a| = 0$ (a solution exists).
- If $b \notin A = \{h(0), h(1), h(2), \ldots\}$, then, for every $x \in \mathbb{N}$, we have $h(x) \neq b$, and, therefore, $f(b, x) = |h(x) - b| > 0$ (a solution does not exist).

Therefore, equation $f(a, x) = 0$ has a solution if and only if $a \in A$, and set A is recursively enumerable set according to Definition 66.

(66 ⇒ 65) Suppose that function $f(a, x)$ is a primitive recursive function and equation $f(a, x) = 0$ has a solution if and only if $a \in A$ (Definition 66).

Let d be some particular element from the set A.

Analyse function $h(t) = \pi_2^1(t) \cdot \overline{sg}(\ f(\pi_2^1(t), \pi_2^2(t))\ ) + d \cdot sg(\ f(\pi_2^1(t), \pi_2^2(t))\ )$.

Function $h(t) \in PrRF$, since it is obtained from primitive recursive functions $f(a, x), \pi_2^1(x), \pi_2^2(x), sg(x), \overline{sg}(x), x + y, x \cdot y$ according to the composition operator.

With $t_0 = \alpha_2(a, x_0)$ we have
$h(t_0) = h(\alpha_2(a, x_0)) = \pi_2^1(\alpha_2(a, x_0)) \cdot \overline{sg}(\ f(\pi_2^1(\alpha_2(a, x_0)), \pi_2^2(\alpha_2(a, x_0)))\ ) +$
$d \cdot sg(\ f(\pi_2^1(\alpha_2(a, x_0)), \pi_2^2(\alpha_2(a, x_0)))\ ) = a \cdot \overline{sg}(\ f(a, x_0)\ ) + d \cdot sg(\ f(a, x_0)\ )$.

The following cases are possible:

- If $f(a, x_0) = 0$ (x_0 is a solution), then $a \in A$ and
  $h(t_0) = a \cdot 1 + d \cdot 0 = a \in A$.

  Therefore, the range of the function $h(t)$ contains all elements from A, since for every $a \in A$ there exists such a x_0, that $f(a, x_0) = 0$.

- If $f(a,x_0) \neq 0$ (x_0 is not a solution), then
  $h(t_0) = a \cdot 0 + d \cdot 1 = d \in A$.

  Therefore, the range of the function $h(t)$ does not contain any other elements except from the set A.

Therefore, set A is a range of the function $h(t)$, and set A is a recursively enumerable set according to Definition 65.

Example 27 *Some examples of the recursively enumerable sets:*

- *an empty set ($\emptyset$) is a recursively enumerable set, since it is the domain of the partial recursive function $f(x) = \mu_z(\ z + (x+1) = 0\ )$ (Definition 64);*
- $\mathbb{N}^+ = \{1,2,3,4,\dots\}$ *is a recursively enumerable set, since it is the range of the primitive recursive function $s(x) = x+1$ (Definition 65);*

Theorem 26 *If A_1 and A_2 are recursively enumerable sets, then sets $B = A_1 \cup A_2$ and $C = A_1 \cap A_2$ are also recursively enumerable sets.*

Proof.

According to Definition 66, there exist primitive recursive functions $f_1(a,x)$ and $f_2(a,x)$, that

- equation $f_1(a,x) = 0$ has a solution if and only if $a \in A_1$ and
- equation $f_2(a,x) = 0$ has a solution if and only if $a \in A_2$.

Function $f_b(a,x) = f_1(a,\pi_2^1(x)) \cdot f_2(a,\pi_2^2(x)) \in PrRF$, since it is obtained from primitive recursive functions $f_1(a,x), \pi_2^1(x), \pi_2^2(x), x \cdot y$ according to the composition operator.

- if $a \in A_1$, then there exists x_1, that $f_1(a,x_1) = 0$,
  and with $x = \alpha_2(x_1,x_2)$:

  $f_b(a,x) = f_b(a,\alpha_2(x_1,x_2)) = f_1(a,\pi_2^1(\alpha_2(x_1,x_2))) \cdot f_2(a,\pi_2^2(\alpha_2(x_1,x_2))) =$
  $= f_1(a,x_1) \cdot f_2(a,x_2) = 0 \cdot f_2(a,x_2) = 0$ (has a solution).

- if $a \in A_2$, then there exists x_2, that $f_2(a,x_2) = 0$,
  and with $x = \alpha_2(x_1,x_2)$:

  $f_b(a,x) = f_b(a,\alpha_2(x_1,x_2)) = f_1(a,\pi_2^1(\alpha_2(x_1,x_2))) \cdot f_2(a,\pi_2^2(\alpha_2(x_1,x_2))) =$
  $= f_1(a,x_1) \cdot f_2(a,x_2) = f_1(a,x_1) \cdot 0 = 0$ (has a solution).

- if $a \notin A_1$ and $a \notin A_2$, then, for any x_1, x_2, $f_1(a,x_1) \neq 0$, $f_2(a,x_2) \neq 0$,
  and with $x = \alpha_2(x_1,x_2)$:

  $f_b(a,x) = f_b(a,\alpha_2(x_1,x_2)) = f_1(a,\pi_2^1(\alpha_2(x_1,x_2))) \cdot f_2(a,\pi_2^2(\alpha_2(x_1,x_2))) =$
  $= f_1(a,x_1) \cdot f_2(a,x_2) \neq 0$ (does not have a solution).

Therefore, equation $f_b(a,x)=0$ has a solution if only $a \in B = A_1 \cup A_2$, and set B is a recursively enumerable set (Definition 66).

Using function $f_c(a,x) = f_1(a,\pi_2^1(x)) + f_2(a,\pi_2^2(x)) \in PrRF$ it is simple to show, that C is also a recursively enumerable set.

Theorem 27 *If set A is recursive, then it is also a recursively enumerable set.*

Proof.

Since A is a recursive set, there exists a general recursive characteristic function $\chi_A(x) = \begin{cases} 1 & \text{, if } x \in A \\ 0 & \text{, otherwise.} \end{cases}$

Function $h(x) = \chi_A(x) - 1 \in PaRF$, since it is obtained from partial recursive functions $\chi_A(x), x - y$ by the composition operator. Set A is a domain of the function $h(x)$, since $0 - 1 = \infty$. Therefore, set A is a recursively enumerable set (Definition 64).

Theorem 28 *If set A is finite, then it is a recursive set*
(and also it is a recursively enumerable set).

Proof.

Suppose $A = \{a_1, a_2, a_3, \ldots, a_n\}$.

Function $\chi_A(x) = \overline{sg}(\,|x-a_1|\cdot|x-a_2|\cdot\ldots\cdot|x-a_n|\,) = \begin{cases} 1 & \text{, if } x \in A \\ 0 & \text{, otherwise.} \end{cases}$

Function $\chi_A(x) \in PrRF \subseteq GeRF$, since it is obtained from primitive rucursive functions $\overline{sg}(x), |x-y|, x \cdot y$ according to the composition operator.

Therefore, set A is a recursive set.

We can summarize that as follows:

Recursive set	Recursively enumerable set
$\chi_A(x) = \begin{cases} 1 & \text{, if } x \in A \\ 0 & \text{, otherwise} \end{cases} \in GeRF.$	$\chi_A(x) = \begin{cases} 1 & \text{, if } x \in A \\ \infty & \text{, otherwise} \end{cases} \in PaRF.$
There is a total algorithm to determine if a set contains an element	There is a partial algorithm to determine if a set contains an element
Algorithm always terminates	Algorithm terminates if a set contains the given element

Theorem 29 *If set A is a recursively enumerable set and it is not recursive, then set A complement $\overline{A}$ is neither a recursively enumerable set nor a recursive set.*

Proof.

Proof by contradiction: Suppose that $\overline{A}$ is a recursively enumerable set.

Since $A, \overline{A}$ are recursively enumerable sets, there exist functions $f(x) \in PrRF$ and $g(x) \in PrRF$, that $A = \{f(0), f(1), f(2), \ldots\}$ and $B = \{g(0), g(1), g(2), \ldots\}$.

Analyse function $h(x) = \mu_z(\,|f(z)-x|\cdot|g(z)-x| = 0\,)$.

Function $h(x) \in PaRF$, since it is obtained from primitive recursive functions $f(x), g(x), |x - y|, x \cdot y$ according to the composition and minimisation operators.

Since $A \cup \overline{A} = \mathbb{N}$ and $A \cap \overline{A} = \emptyset$, for every $x \in \mathbb{N}$, there exists $z \in \mathbb{N}$, that $f(z) = x$ or $g(z) = x$. Therefore, $h(x)$ is a total function and $h(x) \in GeRF$.

- If $x \in A$, then $h(x) = z$, which satisfies $f(z) = x$ and $g(z) \neq x$.
- If $x \notin A$ ($x \in \overline{A}$), then $h(x) = z$, which satisfies $g(z) = x$ and $f(z) \neq x$.

Function $\chi_A(x) = \overline{sg}(\ |f(h(x)) - x|\ ) \in GeRF$, since it is obtained from general recursive functions $h(x), f(x), \overline{sg}(x), |x - y|$ according to the composition operator.

Function $\chi_A(x) =$

$$\overline{sg}(\ |f(h(x)) - x|\ ) = \begin{cases} \overline{sg}(\ |f(z) - x|\ ), f(z) = x & \text{, if } x \in A \\ \overline{sg}(\ |f(z) - x|\ ), f(z) \neq x & \text{, otherwise} \end{cases} = \begin{cases} 1 & \text{, if } x \in A \\ 0 & \text{, otherwise} \end{cases}$$

is a characteristic function of the set A.

Therefore set A is a recursive function. We got a contradiction, since it was given, that A is not a recursive function.

If we have a recursively enumerable set, which is not recursive (according to the last theorem), we may construct a set, which is not a recursively enumerable set. In other words, we may construct such a set that there is no algorithm, which may determine if the set contains particular element or not at all.

2.10 The Halting Problem

Definition 67 *We say that Turing machine is a standard the Turing machine if it is a deterministic 1-tape Turing machine, which satisfies the following conditions:*

- *Alphabet is* $\Sigma = \{0, 1, \flat\}$.
- *There is only one final state (i.e.,* $|F| = 1$*).*
- *After terminating, a head is placed on the first non-blank symbol from the left and tape contains only output of the Turing machine (all symbols outside the result are* $\flat$*).*

Theorem 30 *For every Turing machine* M *there exists a standard Turing machine* M'*, which calculates the same function.*

This theorem says that any Turing machine (deterministic or non-deterministic, 1-tape or m-tape) may be replaced with a standard Turing machine which calculates the same function (usually it uses more time or/and more space).

Theorem 31 *A set of all standard Turing machines is enumerable.*

Proof.

Every standard Turing machine M is a tuple $< \Sigma, Q, q_0, F, \delta >$ and may be written as one long word containing symbols from the symbol set

$A = \{0, 1, \ldots, 9, \flat, \Sigma, Q, F, q, \delta, =, (,), <, >, L, N, R, ; \} \cup \{, \}$.

In other words, every standard Turing machine may be written in a form of word $w \in A^*$. Some words $w \in A^*$ describe Turing machines and some do not. Therefore, a set of all standard Turing machines is a subset of A^*.

A^* is an enumerable set, since A is a finite set. Every subset of the enumerable set is also enumerable.

Therefore, a set of all standard Turing machines is an enumerable set.

It means that all standard Turing machines may be labelled with natural numbers and may be written in the sequence. Suppose, that this sequence of all standard Turing machines is $T_0, T_1, T_2, \ldots$ and these machines calculates partial functions $\varphi_0(x), \varphi_1(x), \varphi_2(x), \ldots$ (Turing machine T_i calculates function φi):

$$\begin{array}{cccccccccc}
T_0, & T_1, & T_2, & T_3, & T_4, & T_5, & \ldots, & T_i, & T_{i+1}, & \ldots \\
\Downarrow & \Downarrow & \Downarrow & \Downarrow & \Downarrow & \Downarrow & & \Downarrow & \Downarrow & \\
\varphi_0(x), & \varphi_1(x), & \varphi_2(x), & \varphi_3(x), & \varphi_4(x), & \varphi_5(x), & \ldots, & \varphi_i(x)\ , & \varphi_{i+1}(x), & \ldots
\end{array}$$

It is interesting if there exists such an algorithm which may calculate every standard Turing machine result according to its number and input data. The simpler problem is to determine if the Turing machine with a given input data terminates its work or it works infinitely.

Definition 68 (The Halting Problem) *If there exists such an algorithm which, for the pair of natural numbers (m, n), determines if a standard Turing machine with number m (i.e., T_m) and input data n (natural number n given in binary notation) terminates its work or not?*

Theorem 32 (The Halting Problem) *The Halting Problem is undecidable:*

There is no such an algorithm which, for the pair of natural numbers (m, n), may determine if a standard Turing machine with number m and input data n finishes its work or not.

Proof.

If the Halting Problem is decidable (solvable), then there is an algorithmically computable function:

$$g(\alpha_2(x,y)) = \begin{cases} 1 & \text{, if } \varphi_x(y) < \infty \\ 0 & \text{, if } \varphi_x(y) = \infty \end{cases} .$$

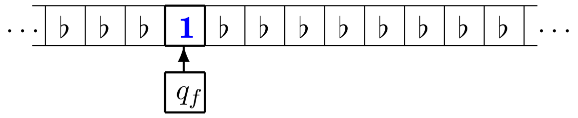
Here $\varphi_x(y)$ is a function calculated by the Turing machine with number x – T_x.

Proof by contradiction: suppose that function $g(\alpha_2(x, y))$ exists and it is algorithmically computable.

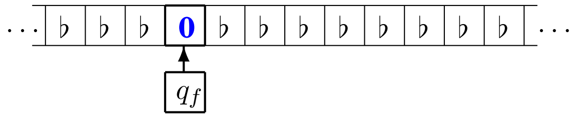
There is a standard Turing machine M_1 which calculates $g(\alpha_2(x, y))$ since it is algorithmically computable.

We will show that there exists such a standard Turing machine M_2, which calculates function $\psi(x) = \begin{cases} 1 & \text{, if } g(\alpha_2(x,x)) = 0 \\ \infty & \text{, if } g(\alpha_2(x,x)) = 1. \end{cases}$

Suppose that $M_1 =< \Sigma, Q, q_0, F, \delta >$ is a standard Turing machine with final state q_f (i.e., $F = \{q_f\}$). Since function $g(\alpha_2(x, y))$ has 2 possible results (0 and 1), after finishing the work a tape will contain the only non-blank symbol (0 or 1), a head will be placed on it, and the Turing machine will have the state q_f:



or



We may construct a Turing machine M_2 by extending the Turing machine M_1:

$M_2 =< \Sigma, Q', q_0, F', \delta' >$, and $Q' = Q \cup \{q^*\}$ (q^* is some new final state), $F' = \{q^*\}$. δ' is the same as δ but extended with the following instructions:

$\delta(q_f, 0) = (q^*, 1, N)$ - return result 1,

$\delta(q_f, 1) = (q_f, 1, R)$ - work infinitely,

$\delta(q_f, \flat) = (q_f, \flat, R)$ - work infinitely.

Therefore, function $\psi(x)$ is also algorithmically computable, since we have a standard Turing machine, which calculates it.

Since M_2 is some standard Turing machine (or $\psi(x)$ is a one argument algorithmically computable function), there exists such a number $l \in \mathbb{N}$, that $\psi(x) = \varphi_l(x)$.

Let's analyse $\psi(l)$:

- if $\psi(l) < \infty$, then also $\varphi_l(l) < \infty \Rightarrow g(\alpha_2(l,l)) = 1 \Rightarrow \psi(l) = \infty$;
- if $\psi(l) = \infty$, then also $\varphi_l(l) = \infty \Rightarrow g(\alpha_2(l,l)) = 0 \Rightarrow \psi(l) < \infty$.

We got a contradiction, since $\psi(l)$ cannot be defined and undefined at the same time.

Therefore, our assumption was incorrect and function $g(\alpha_2(x,y))$ is not algorithmically computable.

The main implication of the Halting Problem:
There does not exist any algorithm (a program or a method) which may determine if some particular program will finish its work or not. In some special occasions we can say that a program is infinite, but generally while a program is working we are not sure if it will finish its work or not.

A more general theorem was proven by H.G. Rice.

Theorem 33 (Rice theorem) *Suppose that X is a set of 1-argument partial recursive functions. If X is a non-empty set and X does not contain all partial recursive functions, then a set of the function numbers $A = \{x : x \in \mathbb{N} \text{ and } \varphi_x \in X\}$ is not a recursive set.*

If set A is a recursive set, then there exists such an algorithm which always finds if set A contains a particular element or not (see Definition 63). If set A is not a recursive set, then such an algorithm does not exist.

According to Rice Theorem 33, we can construct a lot of non-recursive sets:

- If X consists of all functions that are equal to 0. A set of all numbers of functions (Turing machines) from set X is not a recursive function. Therefore, there is no algorithm to determine if some particular Turing machine calculates function $f(x) = 0$ or not.
- If X consists of all functions that are equal to $f(x) = x$. A set of all numbers of functions (Turing machines) from set X is not a recursive function. Therefore, there is no algorithm to determine if some particular Turing machine calculates function $f(x) = x$ or not.

There is no algorithm (program or method), which may determine if some particular program calculates some predefined function or not. And, therefore, there is no algorithm to check if a program works as it was described in its specification.

More about the Halting Problem and its implications may be found in A. Jung's work [2].

2.11 Universal Functions

In this section, we will present universal functions and we will show that there exist recursively enumerable sets, that are not recursive.

Definition 69 *$(n+1)$-argument function $F(x_0, x_1, x_2, \ldots, x_n)$ is a universal function of the set A of the n-argument functions if the following conditions are satisfied:*

- *If $i \in \mathbb{N}$ is some constant, then n-argument function $F(i, x_1, x_2, \ldots, x_n) \in A$.*
- *If n-argument function $g(x_1, x_2, \ldots, x_n) \in A$, then there exists such an $i \in \mathbb{N}$, that $F(i, x_1, x_2, \ldots, x_n) = g(x_1, x_2, \ldots, x_n)$.*

Simply speaking, a universal function represents all functions of the set A. From the definition we also may claim that
$A = \{F(0, x_1, x_2, \ldots, x_n),\ F(1, x_1, x_2, \ldots, x_n),\ F(2, x_1, x_2, \ldots, x_n),\ \ldots\}$

Example 28 *$F(x, y, z) = sg(x) \cdot (y+z) + \overline{sg}(x \dot{-} 1) \cdot (2 \cdot y \cdot z)$ is a universal function of the set A.*

We may find all functions of the set A, since
$A = \{F(0, y, z),\ F(1, y, z),\ F(2, y, z),\ F(3, y, z),\ \ldots\}$.
$F(0, y, z) = sg(0) \cdot (y+z) + \overline{sg}(0 \dot{-} 1) \cdot (2 \cdot y \cdot z) = 2 \cdot y \cdot z = g_1(y, z) \in A$,
$F(1, y, z) = sg(1) \cdot (y+z) + \overline{sg}(1 \dot{-} 1) \cdot (2 \cdot y \cdot z) = (y+z) + 2 \cdot y \cdot z = g_2(y, z) \in A$,
$F(2, y, z) = sg(2) \cdot (y+z) + \overline{sg}(2 \dot{-} 1) \cdot (2 \cdot y \cdot z) = y + z = g_3(y, z) \in A$,
$F(3, y, z) = sg(3) \cdot (y+z) + \overline{sg}(3 \dot{-} 1) \cdot (2 \cdot y \cdot z) = y + z = g_3(y, z) \in A$,
If $x > 3$, then $F(x, y, z) = sg(x) \cdot (y+z) + \overline{sg}(x \dot{-} 1) \cdot (2 \cdot y \cdot z) = y + z = g_3(y, z) \in A$.
Therefore, $A = \{g_1(y, z),\ g_2(y, z),\ g_3(y, z)\} = \{2 \cdot y \cdot z,\ (y+z) + 2 \cdot y \cdot z,\ y+z\}$.

Theorem 34 *The following statements hold:*

1) If A is a set of all n-argument primitive recursive functions, then a set A universal function is not a primitive recursive function,

2) If A is a set of all n-argument general recursive functions, then a set A universal function is not a general recursive function.

Proof.

1) Proof by contradiction. Suppose that $F(x_0, x_1, x_2, \ldots, x_n) \in PrRF$ and it is a universal function of the set A.

Analyse n-argument function $g(x_1, x_2, \ldots, x_n) = F(x_1, x_1, x_2, \ldots, x_n) + 1$.

Function $g(x_1, x_2, \ldots, x_n) \in PrRF$, since it is obtained from the primitive recursive functions $(F(x_0, x_1, x_2, \ldots, x_n), x+y)$ according to the composition operator. Therefore, $g(x_1, x_2, \ldots, x_n) \in A$.

Since $F(x_0, x_1, x_2, \ldots, x_n)$ is a universal function of the set A, there exists a constant $i \in \mathbb{N}$, that $F(i, x_1, x_2, \ldots, x_n) = g(x_1, x_2, \ldots, x_n)$.

With $x_1 = x_2 = \ldots = x_n = i$ we have:

$g(i, i, \ldots, i) = F(i, i, i, \ldots, i) + 1$ and, on the other hand,

$g(i, i, \ldots, i) = F(i, i, i, \ldots, i)$.

Therefore, $F(i, i, i, \ldots, i) + 1 = F(i, i, i, \ldots, i)$. This equation may hold only if $F(i, i, i, \ldots, i) = \infty$ (undefined) but $F(x_0, x_1, x_2, \ldots, x_n) \in PrRF$ and, therefore, it is a total function.

We got a contradiction, and the proof, that $F(x_0, x_1, x_2, \ldots, x_n) \notin PrRF$.

2) The proof is analogous (just $PrRF$ is replaced by $GeRF$).

Definition 70 *Function $D^2(n, x)$ denotes a universal function of all 1-argument primitive recursive functions.*

Theorem 35 *Function $D^2(n, x)$ exists and it is a general recursive function.*

Proof.

Accoring to Theorem 19, every 1-argument primitive recursive function $f(x)$ may be expressed through $s(x) = x + 1$, $q(x) = x \dot{-} [\sqrt{x}]^2$ and the addition, composition and iteration operators.

For every 1-argument primitive recursive function $f(x)$ we will give a number (defined by function $num(f(x))$) constructed according to function expression through $s(x), q(x)$:

- If $f(x) = s(x)$, then $num(f(x)) = num(s(x)) = 1$.
- If $f(x) = q(x)$, then $num(f(x)) = num(q(x)) = 3$.
- If $f(x) = g(x) + h(x)$ (the addition operator), then $num(f(x)) = 2 \cdot 3^{num(g(x))} \cdot 5^{num(h(x))}$.
- If $f(x) = g(h(x))$ (the composition operator), then $num(f(x)) = 4 \cdot 3^{num(g(x))} \cdot 5^{num(h(x))}$.
- If $f(x) = g(x)^I$ (the iteration operator), then $num(f(x)) = 8 \cdot 3^{num(g(x))}$.

We will use these numbers to construct a set A universal function.

With $f_n(x)$ we denote a function which has number n.

It must be mentioned that not every natural number n is a number of some function. The same function may be expressed in several ways and, therefore, every function has several numbers.

For example, $f(x) = s(x) + q(x)$ may also be expressed as $q(x) + s(x)$, and
$num(s(x) + q(x)) = 2 \cdot 3^{num(s(x))} \cdot 5^{num(q(x))} = 2 \cdot 3^1 \cdot 5^3 = 750$, and
$num(q(x) + s(x)) = 2 \cdot 3^{num(q(x))} \cdot 5^{num(s(x))} = 2 \cdot 3^3 \cdot 5^1 = 270$.
Therefore numbers 270 and 750 are numbers of the same function $f(x)$, since

$f_{270}(x) = f_{750}(x) = f(x)$. Number 77 is not a number of any function, since 77 cannot be factorized using prime numbers $2, 3, 5$ only.

If we have a function $f(x)$ number, we may also find the function itself (we may find its expression through $s(x), q(x)$ and, therefore, to get a way to calculate its value):

$$f_n(x) = \begin{cases} s(x) & \text{, if } n = 1 \\ q(x) & \text{, if } n = 3 \\ f_a(x) + f_b(x) & \text{, if } n = 2 \cdot 3^a \cdot 5^b \\ f_a(\, f_b(x)\,) & \text{, if } n = 4 \cdot 3^a \cdot 5^b \\ 0 & \text{, if } n = 8 \cdot 3^a \text{ and } x = 0 \\ f_a(\, f_n(x-1)\,) & \text{, if } n = 8 \cdot 3^a \text{ and } x > 0 \end{cases}$$

We know how to factorize a natural number and, therefore, we know how to determine if a particular number is a number of some function or not.

Function $D^2(n, x) = \begin{cases} f_n(x) & \text{, if } n \text{ is some function number} \\ 0 & \text{, otherwise} \end{cases} \in GeRF$,

since it is an algorithmically computable and total function.

Function $D^2(n, x)$ is a universal function since, with particular n, it generates some 1-argument primitive recursive function and every 1-argument primitive recursive function has its own number.

Definition 71 *Function $D^{n+1}(x_0, x_1, x_2, \ldots, x_n)$ denotes universal function of all n-argument primitive recursive functions.*

Theorem 36 *Universal function $D^{n+1}(x_0, x_1, \ldots, x_n) = D^2(x_0,\ \alpha_n(x_1, x_2, \ldots, x_n))$ and it is a general recursive function.*

Here $D^2(n, x)$ is a universal function of the set of all 1-argument primitive recursive functions.

Proof.

- If $i \in \mathbb{N}$ is some constant, then $D^2(i, x)$ is some primitive recursive function.

  $D^{n+1}(i, x_1, x_2, \ldots, x_n) = D^2(i,\ \alpha_n(x_1, x_2, \ldots, x_n)) \in PrRF$, since it is obtained from primitive recursive functions $D^2(i, x), \alpha_n(x_1, x_2, \ldots, x_n)$ by the composition operator.

- If some n-argument function $g(x_1, x_2, \ldots, x_n) \in PrRF$, then 1-argument function $f(x) = g(\pi_n^1(x),\ \pi_n^2(x),\ \ldots,\ \pi_n^n(x)) \in PrRF$, since it is obtained from primitive recursive functions $g(x_1, x_2, \ldots, x_n), \pi_n^1(x), \pi_n^2(x), \ldots, \pi_n^n(x)$ by the composition operator.

  There exists $i \in \mathbb{N}$, that $D^2(i, x) = f(x)$, since $D^2(n, x)$ is a universal function of the set of all 1-argument primitive recursive functions.

  With $x_0 = i \in \mathbb{N}$ we get, that:

$$D^{n+1}(i, x_1, x_2, \dots, x_n) = D^2(i,\ \alpha_n(x_1, \dots, x_n)) = f(\alpha_n(x_1, \dots, x_n)) =$$
$$g(\pi_n^1(\alpha_n(x_1, \dots, x_n)), \dots, \pi_n^n(\alpha_n(x_1, \dots, x_n))) = g(x_1, x_2, \dots, x_n).$$

According to a definition, function $D^{n+1}(x_0, x_1, x_2, \dots, x_n)$ is a universal function of all n-argument primitive recursive functions.

Function $D^{n+1}(x_0, x_1, x_2, \dots, x_n) \in GeRF$, since it is obtained from a general recursive function $D^2(n, x)$ and a primitive recursive function $\alpha_n(x_1, x_2, \dots, x_n)$ by the composition operator.

Definition 72 *The partial function $f(x_1, x_2, \dots, x_n)$ graph is a set of the points $G = \{(x_1, x_2, \dots, x_n, y) : f(x_1, x_2, \dots, x_n) = y\}$.*

Example 29 *Function $f : \mathbb{N} \to \mathbb{N}$ and $f(x) = x^2 + 1$.*

The function $f(x)$ graph contains points $(0, 1)$, $(8, 65)$, since $f(0) = 1$ and $f(8) = 65$. The function $f(x)$ graph does not contain the point $(1, 1)$, since $f(1) \neq 1$. The function $f(x)$ graph does not contain the point $(\frac{1}{2}, \frac{1}{4})$, since $\frac{1}{2} \notin \mathbb{N}$.

The function $f(x)$ graph is a set $A = \{(0, 1), (1, 2), (2, 5), (3, 10), (4, 17), (5, 26), \dots\}$.

Definition 73 *Function $\widetilde{D}^{n+1}(x_0, x_1, x_2, \dots, x_n)$ denotes a universal function of all n-argument partial recursive functions*

Theorem 37 *Universal function $\widetilde{D}^{n+1}(x_0, x_1, x_2, \dots, x_n)$ exists and it is a partial recursive function.*

Proof.

Suppose $f(x_1, x_2, \dots, x_n)$ is some n-argument partial recursive function.

1) The function $f(x_1, x_2, \dots, x_n)$ graph G is a domain of the function $h(x_1, x_2, \dots, x_n, y) = \overline{sg}\,(|f(x_1, x_2, \dots, x_n) - y|) - 1$, since $h(x_1, x_2, \dots, x_n, y)$ is defined if and only if $f(x_1, x_2, \dots, x_n) = y$.

2) Function $h(x_1, x_2, \dots, x_n, y) = \overline{sg}\,(|f(x_1, x_2, \dots, x_n) - y|) - 1 \in PaRF$, since it is obtained from the primitive recursive functions $\overline{sg}(x), |x - y|$ and partial recursive functions $f(x_1, x_2, \dots, x_n), x - y$ by the composition operator.

3) According to the recursively enumerable set Definition 64, the function $f(x_1, x_2, \dots, x_n)$ graph $G = \{(x_1, x_2, \dots, x_n, y) : f(x_1, x_2, \dots, x_n) = y\}$ is a recursively enumerable set.

4) According to the recursively enumerable set Definition 66, there exists such a primitive recursive function $g(x_1, x_2, \dots, x_n, y, z) \in PrRF$ that the equation $g(x_1, x_2, \dots, x_n, y, z) = 0$ has a solution if and only if $(x_1, x_2, \dots, x_n, y) \in G$ (i.e., if and only if $f(x_1, x_2, \dots, x_n) = y$).

5) Suppose that $t = \alpha_2(y, z)$. Equation $g(x_1, x_2, \dots, x_n, \pi_2^1(t), \pi_2^2(t)) = 0$ has a solution if and only if $f(x_1, x_2, \dots, x_n) = y$.

6) Function $F(x_1, x_2, \ldots, x_n, t) = g(x_1, x_2, \ldots, x_n, \pi_2^1(t), \pi_2^2(t)) \in PrRF$, since it is obtained from primitive recursive functions $g(x_1, x_2, \ldots, x_n, y, z), \pi_2^1(x), \pi_2^2(x)$ by the composition operator.

In the clauses above (from 1 to 6) we have shown that for any n-argument partial recursive function $f(x_1, x_2, \ldots, x_n)$ there exists a primitive recursive function $F(x_1, x_2, \ldots, x_n, t) \in PrRF$ which satisfies the following condition:

Equation $F(x_1, x_2, \ldots, x_n, t) = 0$ has a solution if and only if $f(x_1, x_2, \ldots, x_n) = y$ (and $t = \alpha_2(y, z)$). Therefore,

$$f(x_1, x_2, \ldots, x_n) = \pi_2^1 \left(\mu_t \left(F(x_1, x_2, \ldots, x_n, t) = 0 \right)\right). \tag{2.1}$$

Equation (2.1) is correct since:

- if $f(x_1, x_2, \ldots, x_n) = y$, then equation $F(x_1, x_2, \ldots, x_n, t) = 0$ has a solution and $\mu_t \left(F(x_1, x_2, \ldots, x_n, t) = 0 \right)$ returns a solution $t = \alpha_2(y, z)$ (and $y = \pi_2^1(t)$).
- if $f(x_1, x_2, \ldots, x_n) = \infty$, then equation $F(x_1, x_2, \ldots, x_n, t) = 0$ has no solution and $\mu_t \left(F(x_1, x_2, \ldots, x_n, t) = 0 \right) = \infty$.

Now we may construct a universal function $\widetilde{D}^{n+1}(x_0, x_1, x_2, \ldots, x_n)$ (a universal function of the set A):

$$\widetilde{D}^{n+1}(x_0, x_1, x_2, \ldots, x_n) = \pi_2^1 \left(\mu_t \left(D^{n+2}(x_0, x_1, x_2, \ldots, x_n, t) = 0 \right)\right), \tag{2.2}$$

($D^{n+2}(x_0, x_1, x_2, \ldots, x_n, t)$ is a universal function
of the set of all $(n + 1)$-argument primitive recursive functions).

Function $\widetilde{D}^{n+1}(x_0, x_1, x_2, \ldots, x_n)$ is a universal function of all n-argument partial recursive functions, since:

- If $i \in \mathbb{N}$ is some constant, then $D^{n+2}(i, x_1, x_2, \ldots, x_n, t)$ is some primitive recursive function and, therefore, $\widetilde{D}^{n+1}(i, x_1, x_2, \ldots, x_n) \in PaRF$, since it is obtained from primitive recursive functions $D^{n+2}(i, x_1, x_2, \ldots, x_n, t), \pi_2^1(x)$ by the composition and minimisation operators.
- If $f(x_1, x_2, \ldots, x_n) \in PaRF$, then there exists a primitive recursive function $F(x_1, x_2, \ldots, x_n, t)$, which satisfies equation (2.1).
  Since, $F(x_1, x_2, \ldots, x_n, t) \in PrRF$, and $D^{n+2}(x_0, x_1, x_2, \ldots, x_n, t)$ is a universal function of all $(n + 1)$-argument primitive recursive functions, there exists such an $i \in \mathbb{N}$, that $D^{n+2}(i, x_1, x_2, \ldots, x_n, t) = F(x_1, x_2, \ldots, x_n, t)$.
  Therefore, with $x_0 = i$ we have:
  $\widetilde{D}^{n+1}(i, x_1, x_2, \ldots, x_n) = \pi_2^1 \left(\mu_t \left(D^{n+2}(i, x_1, x_2, \ldots, x_n, t) = 0 \right)\right) =$
  $= \pi_2^1 \left(\mu_t \left(F(x_1, x_2, \ldots, x_n, t) = 0 \right)\right) = f(x_1, x_2, \ldots, x_n)$.

We also have proved (see equation (2.1)) that any partial recursive function may be expressed through a primitive recursive function applying the minimisation operator only one time.

Definition 74 *The partial function $f(x_1, x_2, \ldots, x_n)$ extension is some total function $g(x_1, x_2, \ldots, x_n)$, which satisfies the following condition:*

for any $x'_1, x'_2, \ldots, x'_n \in \mathbb{N}$, if $f(x'_1, x'_2, \ldots, x'_n) < \infty$ (defined), then $f(x'_1, x'_2, \ldots, x'_n) = g(x'_1, x'_2, \ldots, x'_n)$.

Example 30 *Functions $g(x, y) = x \dot{-} y$ and $h(x, y) = |x - y|$ are different extensions of the partial function $f(x, y) = x - y$, since $g(x, y) = h(x, y) = f(x, y)$, if $f(x, y) < \infty$ (i.e., if $x \geq y$).*

Theorem 38 *Function $V(x) = \overline{sg}\left(\widetilde{D}^{n+1}(x, x, \ldots, x)\right)$ does not have any extension.*

Proof.

Proof by contradiction: suppose that function $V(x)$ has an extension $W(x)$.

Since, $W'(x, x_2, \ldots, x_n) = pr^1_n(W(x), x_2, \ldots, x_n) = W(x)$ is also an n-argument general (and partial) recursive total function, there exists $i \in \mathbb{N}$ that $\widetilde{D}^{n+1}(i, x, x_2, \ldots, x_n) = W'(x, x_2, \ldots, x_n) = W(x)$.

With $x_1 = x_2 = \ldots = x_n = i$, we have $\widetilde{D}^{n+1}(i, i, i, \ldots, i) = W'(i, i, \ldots, i) = W(i)$.

Since function $W(x)$ is a function $V(x)$ extension, it is a total function and consequently $\widetilde{D}^{n+1}(i, i, i, \ldots, i) = W(i) < \infty$ (is also defined).

Therefore, $V(i) = \overline{sg}\left(\widetilde{D}^{n+1}(i, i, \ldots, i)\right) < \infty$ (is also defined).

Since, $V(i) < \infty$ and $W(x)$ is its extension, $W(i) = V(i)$ and it is defined.

From the obtained statements we got that $W(i) = V(i) = \overline{sg}\left(\widetilde{D}^{n+1}(i, i, \ldots, i)\right) = \overline{sg}(W'(i, i, \ldots, i)) = \overline{sg}(W(i))$. Therefore, $W(i) = \overline{sg}(W(i))$ and it is defined.

We got a contradiction, and consequently function $V(x)$ does not have extension.

Theorem 39 *Universal function $\widetilde{D}^{n+1}(x_0, x_1, x_2, \ldots, x_n)$ does not have any extension.*

Proof.

Proof by contradiction: suppose that a universal function $\widetilde{D}^{n+1}(x_0, x_1, x_2, \ldots, x_n)$ has an extension $P(x_0, x_1, x_2, \ldots, x_n)$. Therefore, if $\widetilde{D}^{n+1}(x_0, x_1, x_2, \ldots, x_n) < \infty$, then $\widetilde{D}^{n+1}(x_0, x_1, x_2, \ldots, x_n) = P(x_0, x_1, x_2, \ldots, x_n)$.

Since $P(x_0, x_1, x_2, \ldots, x_n)$ is a total function, function $W(x) = \overline{sg}(P(x, x, \ldots, x))$ is also a total function.

According to Theorem 38, $V(x) = \overline{sg}\left(\widetilde{D}^{n+1}(x, x, \ldots, x)\right)$ does not have extension.

If $V(x) < \infty$ (defined), then also $\widetilde{D}^{n+1}(x, x, x, \ldots, x) < \infty$ and
$V(x) = \overline{sg}\left(\widetilde{D}^{n+1}(x, x, \ldots, x)\right) = \overline{sg}(P(x, x, \ldots, x)) = W(x)$.

Therefore, function $W(x)$ is an extension of the function $V(x)$ - a contradiction for the Theorem 38. So, function $\widetilde{D}^{n+1}(x_0, x_1, x_2, \ldots, x_n)$ does not have any extension.

Theorem 40 *There exists a recursively enumerable set, which is not a recursive set.*

Proof.

Function $\widetilde{D}^2(x_0, x_1)$ is a universal function of all 1-argument partial recursive functions.

According to Theorem 38, function $V(x) = \overline{sg}\left(\widetilde{D}^2(x, x)\right)$ has the following features:

- function $V(x) \in PaRF$,
- function $V(x)$ does not have any extension,
- function $V(x)$ range is $\{0, 1\}$.

Set A is a set of all solutions of the equation $V(x) = 0$.

Set A is a recursively enumerable set, since it is the domain of the partial recursive function $h(x) = \mu_z\,(V(x) + z = 0)$ (Definition 64).

Proof by contradiction: suppose that A is a recursive set.
Therefore, there exists a general recursive characteristic function (Definion 63)

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

Function $W(x) = \overline{sg}(\chi_A(x))$ is an extension of the function $V(x)$,
since if $V(x) < \infty$, then

$$W(x) = \overline{sg}(\chi_A(x)) = \begin{cases} \overline{sg}(1) & \text{, if } V(x) = 0 \\ \overline{sg}(0) & \text{, otherwise} \end{cases} = \begin{cases} 0 & \text{, if } V(x) = 0 \\ 1 & \text{, otherwise} \end{cases} = V(x).$$

We got a contradiction, since $V(x)$ does not have any extension.

2.12 The Lambda Calculus

The Lambda calculus, or simply λ-calculus, is one of the algorithm formalizations. It was introduced by A. Church in 1930.

This calculus operates with special words of the symbols, that are called terms.

Definition 75 *The term of the λ-calculus is defined as follows:*

- *Any variable is a term.*
  We will use small latin letters to denote variables (i.e., $a, b, c, \ldots, w, x, y, z$).
- *If E_1 and E_2 are terms, then $(E_1 E_2)$ is also a term (application).*
- *If x is a variable and E is a term, then $(\lambda x.E)$ is also a term (abstraction).*
  In the term $(\lambda x.E)$, term E is the scope of λx.

Usually, outermost parenthesis will be omitted. Term $\lambda x_1.\lambda x_2.\ldots\lambda x_n.(E)$ denotes $\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.(E)))\ldots))$. Sometimes we will use angle brackets [,] instead of the usual parenthesis (,) just to highlight the alignment of the brackets. And there is no semantic difference between [,] and (,) used.

Example 31 *Some examples of the terms of the λ- calculus are:*
x, u, v,
xu, $(uv)(ux)$, $((xv)u)(yw)$,
$\lambda x.x$, $\lambda u.(\,(uv)(ux)\,)$, $\lambda w.(\,[\,(\lambda x.\lambda u.(ux)\,)u\,](yw)\,)$.

Note, that terms $(xy)z$ and $x(yz)$ are different terms ($(xy)z \neq x(yz)$). Terms $\lambda x.xx$ and $\lambda x.(xx)$ are also differetn terms ($\lambda x.xx \neq \lambda x.(xx)$).

Definition 76 *Variable x occurrence in the term E is bounded if it occurs under the scope of $\lambda x.$. Otherwise, variable x occurrence is free.*

Example 32 *Given λ-calculus term $\lambda x.[\,(yz)(\,\lambda z.((zx)y)\,)\,](x(\,\lambda y.(xy)\,))$.*

In the example, free variables occurrences are marked in the blue colour, and bounded occurrences in the magenta (red-pink) colour: the first x occurrence is bounded, the second and the third x occurrences are free; the first and the second y occurrences are free, the third y occurrence is bounded; the first z occurrence is free, the second z occurrence is bounded.

Definition 77 *λ-calculus terms E_1 and E_2 are α - equivalent if:*

- *term E_2 is obtained from term E_1, where all free variable x occurrences in E_1 are substituted by some new variable y;*
- *term E_2 is obtained from term E_1, where term $\lambda x.(E_1')$ is substituted by term $\lambda y.(E_2')$, and term E_2' is obtained from term E_1', where all free variable x occurrence in E_1' are substituted by some new variable y;*

- *terms E_1 and E_2 are α - equivalent to some term E_3.*

Example 33 *λ-calculus terms $E_1 = \lambda x.[\,(yz)(\,\lambda z.((zx)y)\,)\,](x(\,\lambda y.(xy)\,))$ and $E_2 = \lambda x.[\,(yz)(\,\lambda z.((zx)y)\,)\,](w(\,\lambda y.(wy)\,))$ are α - equivalent.*

Terms $E_2 = \lambda x.[\,(yz)(\,\lambda z.((zx)y)\,)\,](w(\,\lambda y.(wy)\,))$ and $E_3 = \lambda x.[\,(yz)(\,\lambda u.((ux)y)\,)\,](w(\,\lambda y.(wy)\,))$ are α - equivalent.

Therefore, terms E_1 and E_3 are also α - equivalent.

Definition 78 *λ-calculus term of the shape $(\,\lambda x.(E)\,)Y$ is called redex. Reduct of the redex $(\,\lambda x.(E)\,)Y$ is term $E[Y/x]$. $E[Y/x]$ denotes the term obtained from term E, where all free variable x occurrences in E are substituted by term Y.*

Definition 79 *λ-calculus term E β-reduction is a sequence of the terms $E_1 \triangleright E_2 \triangleright E_3 \triangleright \ldots$, which satisfies:*

- *term $E_1 = E$,*
- *term E_{i+1} is obtained from term E_i, where the first redex in the E_i was substituted by its reduct ($i = 1, 2, \ldots$).*

Definition 80 *λ-calculus term E is normalized if it does not contain any redex.*

In other words, term E β-reduction is term E normalization (the way to find a normalized term).

Example 34 *λ-calculus term $(\lambda x.((\lambda x.(xy))x))(\lambda u.u)$ β-reduction is the following:*
$(\lambda x.((\lambda x.(xy))x))(\lambda u.u) \triangleright (\lambda x.(xy))(\lambda u.u) \triangleright (\lambda u.u)y \triangleright y$.

Definition 81 *λ-calculus term E is unnormalized if its β-reduction is infinite.*

Example 35 *λ-calculus term $(\lambda u.(uu))(\lambda u.(uu)))$ is an unnormalized term, since its β-reduction is infinite:*
$(\lambda u.(uu))(\lambda u.(uu))) \triangleright (\lambda u.(uu))(\lambda u.(uu))) \triangleright (\lambda u.(uu))(\lambda u.(uu))) \triangleright \ldots$.

In λ-calculus, logical constants and natural numbers are encoded with special terms. λ-calculus term, which encodes natural number, is called the Church numeral.

Definition 82 *λ-calculus term $\lambda x.\lambda y.x$ encode logical constant* true *and term $\lambda x.\lambda y.y$ encode logical constant* false.

Definition 83 *λ-calculus term $\underline{k} = \lambda f.\lambda x.(f^k x)$ is the k-th Church numeral and it encodes the natural number k.*
Here, $f^k x$ denotes term $\underbrace{f(f(\ldots(f}_{k \text{ times}}\,x)\ldots))$.

Example 36 *Examples of some Church numerals are:*
$\underline{0} = \lambda f.\lambda x.(x)$,
$\underline{1} = \lambda f.\lambda x.(fx)$,
$\underline{2} = \lambda f.\lambda x.(f^2x) = \lambda f.\lambda x.(f(fx))$,
$\underline{3} = \lambda f.\lambda x.(f^3x) = \lambda f.\lambda x.(f(f(fx)))$.

Definition 84 *Partial function $f(x_1, x_2, \ldots, x_n)$ is defined by λ-calculus term E if the following conditions are satisfied:*

- *If $f(k_1, k_2, \ldots, k_n) = k$, then term $(\ldots(((E)\underline{k_1})\underline{k_2})\ldots)\underline{k_n}$ β-reduction finishes with term $\underline{k}$.*
- *If $f(k_1, k_2, \ldots, k_n) = \infty$ (undefined), then term $(\ldots(((E)\underline{k_1})\underline{k_2})\ldots)\underline{k_n}$ is unnormalized (β-reduction is infinite).*

Example 37 *λ-calculus term $E = \lambda n.\lambda f.\lambda x.((nf)(fx))$ defines function $s(x) = x+1$.*

We will calculate $s(2)$ with λ-calculus:
$[\lambda n.\lambda f.\lambda x.(\,(nf)(fx)\,)]\,\underline{2}\ =$
$[\lambda n.\lambda f.\lambda x.(\,(nf)(fx)\,)]\,(\lambda f.\lambda x.(f(fx))) \triangleright$
$\lambda f.\lambda x.(\,([\,\lambda f.\lambda x.(f(fx))\,]\,f)(fx)\,) \triangleright$
$\lambda f.\lambda x.(\,(\lambda x.(f(fx)))(fx)\,) \triangleright$
$\lambda f.\lambda x.(f(f(fx))) \triangleright \underline{3}$.

Therefore, $s(2) = 3$.

Theorem 41 *If $f(x_1, x_2, \ldots, x_n)$ is some partial recursive function, then there exists λ-calculus term E, which defines function $f(x_1, x_2, \ldots, x_n)$.*

Since the set of all partial recursive functions is equal to the set of all algorithmically computable functions (according to the Church's Thesis, Theorem 9), Theorem 41 says that λ-calculus is also a good algorithm formalization.

More about λ-calculus may be found in P. Selinger's work [7].

Bibliography

[1] L. Fortnow, *The status of the P versus NP problem*, Communications of the ACM 52, No. 9, 2009, pp. 78-86.

[2] A. Jung, *The Halting Problem* (adopted by Volker Sorge, Steve Vickers in 2012), The University of Birmingham, 2007.

[3] R. Lassaigne, M. de Rougemont, *Logika ir algoritmų sudėtingumas*, Žodynas, Vilnius, 1999.

[4] S. Norgėla, *Logika ir dirbtinis intelektas*, TEV, 2007.

[5] S. Norgėla, *Matematinė logika*, TEV, 2004.

[6] Lawrence C. Paulson, *Logic and Proof*, Computer Laboratory, University of Cambridge, 2007.

[7] P. Selinger, *Lecture Notes on the Lambda Calculus*, Department of Mathematics and Statistics, Dalhousie University, Halifax, Canada, 2007.