RECURSIVE FUNCTIONS AND REGISTER MACHINES

1. Introduction

Our goal in these notes is twofold: first, to rigorously explicate what constitutes an effective procedure, enabling us to study such procedures and deduce their essential properties and limitations; second, to introduce an abstract framework for a class of rudimentary, assembly-like languages known as register machines.

2. Notions related to partial functions on natural numbers

This section introduces terminology used in throughout the notes.

By numbers we mean natural numbers. The set of all numbers is $\mathbb{N}$. Note that $\mathbb{N}$ is well ordered by the standard relation $\leq$. By initial interval of $\mathbb{N}$ we mean initial segment with respect to $\leq$.

If $k \in \mathbb{N}$, then the set of k-tuples of numbers is the Cartesian product $\mathbb{N}^k$. We denote k-tuples of numbers by vector notation $\vec{x}$ and refer to the number k as the length of $\vec{x}$. In the case $k = 0$ the set $\mathbb{N}^0$ contains a unique element: the empty tuple.

Definition 2.1. Let $k \in \mathbb{N}$. Let f be a function with domain a subset of $\mathbb{N}^k$ and taking values in $\mathbb{N}$. Then f is a *k-ary partial function*.

Definition 2.2. A *partial function* is a k-ary partial function for some $k \in \mathbb{N}$.

Definition 2.3. Let f be a k-ary partial function. If $\vec{x} \in \mathbb{N}^k$ is in the domain of f, then we say that f is *defined at $\vec{x}$*.

Remark 2.4. Let f be a k-ary partial function and let $\vec{x} \in \mathbb{N}^k$. We write $f(\vec{x}) \downarrow$ to denote that f is defined at $\vec{x}$. If f is not defined at $\vec{x}$, then we use $f(\vec{x}) \uparrow$.

Definition 2.5. Let $k \in \mathbb{N}$ and let f be a k-ary partial function such that $f(\vec{x}) \downarrow$ for every $\vec{x} \in \mathbb{N}^k$. Then f is called a *k-ary total function*.

Definition 2.6. Let g be a k-ary partial function and let $f_1, ..., f_k$ be n-ary partial functions. Consider all $\vec{x} \in \mathbb{N}^n$ such that $f_1(\vec{x}) \downarrow, ..., f_k(\vec{x}) \downarrow$ and $g\big(f_1(\vec{x}), ..., f_k(\vec{x})\big) \downarrow$. Then the map

$$\vec{x} \mapsto g\big(f_1(\vec{x}), ..., f_k(\vec{x})\big)$$

defines a well defined n-ary partial function. We refer to this partial function as the *composition of g with $f_1, ..., f_k$*.

Definition 2.7. Let g be a $(k+2)$-ary partial function and let f be a k-ary partial function for some $k \in \mathbb{N}$. Fix $\vec{x} \in \mathbb{N}^k$. We define $h(\vec{x}, y)$ for numbers y in some initial interval of $\mathbb{N}$. We set

$$h(\vec{x}, 0) = f(\vec{x})$$

if $f(\vec{x}) \downarrow$. Now suppose that $h(\vec{x}, y) \downarrow$ for some number y and that $g\big(\vec{x}, y, h(\vec{x}, y)\big) \downarrow$. Then we set

$$h(\vec{x}, y+1) = g\big(\vec{x}, y, h(\vec{x}, y)\big)$$

Then h is a $(k+1)$-ary partial function obtained by *primitive recursion* from g and f.

Definition 2.8. Let g be a $(k+1)$-ary partial function for some $k \in \mathbb{N}$. Fix $\vec{x} \in \mathbb{N}^k$. Suppose that there exists number t such that

$$g(\vec{x},0)\downarrow, g(\vec{x},1)\downarrow, ..., g(\vec{x},t-1)\downarrow$$

are defined, nonzero numbers and $g(\vec{x},t) = 0$. Then we define $\mu_y\left[g(\vec{x},y) = 0\right] = t$. If such number t does not exists, then $\mu_y\left[g(\vec{x},y) = 0\right]$ is undefined. This gives rise to a k-ary partial function $\mu_y\left[g(\vec{x},y) = 0\right]$. The function $\mu_y\left[g(\vec{x},y) = 0\right]$ is the result of *unbounded minimization* of g.

3. Primitive recursive functions

Definition 3.1. The function $x \mapsto x+1$ is the *successor* function.

Definition 3.2. The class of *primitive recursive functions* is the smallest class of partial functions such that the following conditions hold.

(1) The zero k-ary total function is primitive recursive for every $k \in \mathbb{N}$.

(2) The successor function is primitive recursive.

(3) Let $k, n \in \mathbb{N}$ and $1 \leq k \leq n$. The n-ary function

$$I_n^k(x_1, ..., x_n) = x_k$$

is primitive recursive.

(4) Let $k, n \in \mathbb{N}$. If g is a k-ary primitive recursive function and $f_1, ..., f_k$ are n-ary primitive recursive functions, then the composition of g and $f_1, ..., f_k$ is a primitive recursive function.

(5) Let $k \in \mathbb{N}$. If g is a $(k+1)$-ary primitive recursive function and f is a k-ary primitive recursive function, then also the function obtained by primitive recursion from g and f is primitive recursive.

Proposition 3.3. *All primitive recursive functions are total.*

Proof. Note that the class of total functions is closed under composition and primitive recursion. Since the class of primitive recursive functions is defined by this operations, it follows that all primitive recursive functions are total. □

The remainder of this section is devoted to proving that most well-known total functions are primitive recursive.

Proposition 3.4. *Let $k \in \mathbb{N}$. Then all k-ary total constant functions are primitive recursive.*

Proof. Let c be a number. By composing the succesor function with itself c-times we obtain the function $x \mapsto x+c$, which is primitive recursive. Next by composing this function with the zero k-ary function, we derive the k-ary total constant function with c as the only value. Hence this function is also primitive recursive. This completes the proof. □

Proposition 3.5. *The addition function $(x,y) \mapsto x+y$ is primitive recursive.*

Proof. Note that I_1^1 is the identity function. Hence identity is primitive recursive. Similarly the function $(x,y,z) \mapsto z+1$ is primitive recursive. Indeed, it is the composition of I_3^3 and the successor. Next the function $(x,y) \mapsto x+y$ is obtained by primitive recursion from the two functions described above. Hence it is primitive recursive and the proof is completed. □

Proposition 3.6. *The multiplication function $(x,y) \mapsto x \cdot y$ is primitive recursive.*

Proof. Note that the function $(x,y,z) \mapsto z+x$ is the composition of addition function with I^3_1 and I^3_3. Hence it is primitive recursive by Proposition 3.5. Next observe that the function $(x,y) \mapsto x \cdot y$ is obtained by primitive recursion from $(x,y,z) \mapsto x+z$ and the identity function. Hence it is primitive recursive and this completes the proof. □

Proposition 3.7. *The exponential function $(x,y) \mapsto x^y$ is primitive recursive.*

Proof. Note that the function $(x,y,z) \mapsto z \cdot x$ is the composition of multiplication with I^3_1 and I^3_3. Hence it is primitive recursive by Proposition 3.6. Now the function $(x,y) \mapsto x^y$ is obtained by primitive recursion from $(x,y,z) \mapsto z \cdot x$ and constant unary function with value 1. Since both these functions are primitive recursive, the proof is completed. □

Definition 3.8. The function

$$x \mapsto \begin{cases} x-1 & \text{if } x>0 \\ 0 & \text{if } x=0 \end{cases}$$

is the *predecessor* function.

Proposition 3.9. *The predecessor function is primitive recursive.*

Proof. The predecessor function is obtained by primitive recursion from I^2_1 and the zero nullary function. Hence it is primitive recursive. □

Definition 3.10. The 2-ary function

$$x \dot{-} y = \begin{cases} x-y & \text{if } x>y \\ 0 & \text{otherwise} \end{cases}$$

is the *monus* function.

Proposition 3.11. *The monus function is primitive recursive.*

Proof. Note that the function $(x,y,z) \mapsto z \dot{-} 1$ is the composition of the predecessor with I^3_3. By Proposition 3.9 it is primitive recursive. The monus function is obtained from $(x,y,z) \mapsto z \dot{-} 1$ and the identity by primitive recursion. Thus it is primitive recursive. □

Proposition 3.12. *Let $k \in \mathbb{N}$ and let f be a $(k+1)$-ary primitive recursive function. Then functions given by formulas*

$$(\vec{x},y) \mapsto \sum_{z<y} f(\vec{x},z),\ (\vec{x},y) \mapsto \prod_{z<y} f(\vec{x},z)$$

are primitive recursive.

Proof. We prove the result for sum. The proof for product is analogous.

First observe that the function $(\vec{x},y,z) \mapsto f(\vec{x},y)$ is the composition of f and $I^{k+2}_1, ..., I^{k+2}_{k+1}$. Hence it is primitive recursive. Next note that the function $(\vec{x},y,z) \mapsto z + f(\vec{x},y)$ is primitive recursive as the composition of addition which is primitive recursive by Proposition 3.5 with the function described above and I^{k+2}_{k+2}. Finally the function

$$(\vec{x},y) \mapsto \sum_{z<y} f(\vec{x},z)$$

is obtained by primitive recursion from the function described above and the k-ary zero function. □

Corollary 3.13. *Let $k \in \mathbb{N}$ and let f be a $(k+1)$-ary primitive recursive function. Then functions given by formulas*

$$(\vec{x}, y) \mapsto \sum_{z \leq y} f(\vec{x}, z),\ (\vec{x}, y) \mapsto \prod_{z \leq y} f(\vec{x}, z)$$

are primitive recursive.

Proof. Left for the reader as an exercise. □

Definition 3.14. The function

$$\mathrm{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

is the *sign* function.

Proposition 3.15. *The sign function is primitive recursive.*

Proof. The sign function is obtained by primitive recursion from the constant unary function with value 1 and the unary zero function. Since both these functions are primitive recursive, we obtain the statement. □

4. Primitive recursive relations

Definition 4.1. Let $k \in \mathbb{N}$ and let R be a function that assigns to each $\vec{x} \in \mathbb{N}^k$ a proposition $R(\vec{x})$. Then R is a *k-ary relation*.

Remark 4.2. Let $k \in \mathbb{N}$ and let R be a k-ary relation. Then depending on the context $R(\vec{x})$ can denote either a proposition or a k-ary relation. This is reflects Frege's distinction between complete and incomplete symbols.

Remark 4.3. Let $k \in \mathbb{N}$ and let R be a k-ary relation. Then R defines a subset

$$\{\vec{x} \in \mathbb{N}^k \,|\, R(\vec{x})\} \subseteq \mathbb{N}^k$$

This correspondence gives rise to a bijection between the class of all k-ary relations and the class of all subsets of $\mathbb{N}^k$.

Definition 4.4. Let $k \in \mathbb{N}$ and let R be a k-ary relation. The k-ary function

$$\mathbb{1}_R(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \\ 0 & \text{otherwise} \end{cases}$$

is called the *indicator function* of R.

Definition 4.5. Let $k \in \mathbb{N}$ and let R be a k-ary relation. Suppose that the indicator function of R is primitive recursive. Then R is *primitive recursive*.

Proposition 4.6. *Let $k \in \mathbb{N}$ and let R, S be k-ary primitive recursive relation. Then the following assertions hold.*

(1) *$R(\vec{x}) \vee S(\vec{x})$ is primitive recursive.*

(2) *$R(\vec{x}) \wedge S(\vec{x})$ is primitive recursive.*

(3) *$\neg R(\vec{x})$ is primitive recursive.*

Proof. For the proof of **(1)** note that the function obtained by applying sign to the sum $\mathbb{1}_R + \mathbb{1}_S$ is primitive recursive. Since this function coincides with $\mathbb{1}_{R\cup S}$, we derive that **(1)** holds.

Since $\mathbb{1}_{R\cap S} = \mathbb{1}_R \cdot \mathbb{1}_S$, we deduce that **(2)** holds.

Let $\mathbb{1}_{\mathbb{N}^k}$ be a constant k-ary function with value 1. By composing monus function with $\mathbb{1}_{\mathbb{N}^k}, \mathbb{1}_R$ we deduce that

$$\mathbb{1}_{\mathbb{N}^k \setminus R} = \mathbb{1}_{\mathbb{N}^k} \dot{-} \mathbb{1}_R$$

is primitive recursive. This proves **(3)**. □

Proposition 4.7. *Let $k \in \mathbb{N}$ and let R be a $(k+1)$-ary primitive recursive relation. Then the following $(k+1)$-ary relations*

$$\forall_{t<y}\, R(\vec{x}, t),\ \forall_{t\leq y}\, R(\vec{x}, t),\ \exists_{t<y}\, R(\vec{x}, t),\ \exists_{t\leq y} R(\vec{x}, t)$$

are primitive recursive.

Proof. This follows immediately from Proposition 3.12 and Corollary 3.13 combined with the fact that sign is primitive recursive. These imply that bounded quantification of primitive recursive relations produces primitive recursive relations. □

Proposition 4.8. *The binary relations $x < y$, $x = y$, $x \leq y$ are primitive recursive.*

Proof. The relation $x < y$ is primitive recursive because its indicator can be expressed according by combination of monus and sing which are primitive recursive functions. By Proposition 4.6 the relation $x \geq y$ is primitive recursive as well. Since $x \leq y$ is obtained by swapping variables in $x \leq y$, it is also primitive recursive. Finally using Proposition 4.6 and the equivalence

$$(x \leq y) \wedge (y \leq x) \ \Leftrightarrow\ x = y$$

we conclude that $x = y$ is primitive recursive. □

Now we prove some useful results regarding primitive recursive relations and functions.

Proposition 4.9. *Let $k, n \in \mathbb{N}$. Let $R_1, ..., R_n$ be k-ary primitive recursive relations and let $f_1, ..., f_n$ be k-ary primitive recursive functions. Suppose that for each $\vec{x} \in \mathbb{N}^k$ exactly one*

$$R_1(\vec{x}), ..., R_n(\vec{x})$$

holds. Then the k-ary function defined by

$$\vec{x} \mapsto \begin{cases} f_1(\vec{x}) & \text{if } R_1(\vec{x}) \\ ... & \\ f_n(\vec{x}) & \text{if } R_n(\vec{x}) \end{cases}$$

is primitive recursive.

Proof. Indeed, we have

$$f(\vec{x}) = f_1(\vec{x}) \cdot \mathbb{1}_{R_1}(\vec{x}) + ... + f_n(\vec{x}) \cdot \mathbb{1}_{R_n}(\vec{x})$$

for every $\vec{x} \in \mathbb{N}^k$. □

Definition 4.10. Let $k \in \mathbb{N}$ and let R be a $(k+1)$-ary relation. We define

$$\mu_{t<y}\, R\,(\vec{x}, t) = \begin{cases} \min\{t \,|\, t < y \text{ and } R\,(\vec{x}, t)\} & \text{if } R\,(\vec{x}, t) \text{ for some } t < y \\ y & \text{otherwise} \end{cases}$$

Then $\mu_{t<y}\, R\,(\vec{x}, t)$ is said to be obtained by *bounded minimization* from R.

Finally we prove important result that bounded minimization of primitive recursive relation is primitive recursive.

Proposition 4.11. *Let $k \in \mathbb{N}$ and let R be a $(k+1)$-ary primitive recursive relation. Then the function*

$$(\vec{x}, y) \mapsto \mu_{t<y}\, R\,(\vec{x}, t)$$

is primitive recursive.

Proof. Define a relation

$$N(\vec{x}, t) = \forall_{z<t}\, \neg R\,(\vec{x}, z)$$

which states that there are no z such that $R(\vec{x}, z)$ and $z < t$. Then N is the $(k+1)$-ary primitive recursive relation by Proposition 4.7.

Next define

$$S(\vec{x}, t) = R(\vec{x}, t) \wedge N(\vec{x}, t)$$

which holds precisely when t is the smallest z such that $R(\vec{x}, z)$. Hence S is primitive recursive $(k+1)$-ary relation by Proposition 4.6.

Now observe that

$$\mu_{t<y}\, R\,(\vec{x}, t) = y \cdot \mathbb{1}_N\,(\vec{x}, y) + \sum_{t<y} \mathbb{1}_S\,(\vec{x}, t) \cdot t$$

for every $\vec{x} \in \mathbb{N}^k$ and $y \in \mathbb{N}$. This function is primitive recursive by Proposition 3.12, the fact that sum is primitive recursive and our observations that N, S are primitive recursive $(k+1)$-ary relations. □

5. Number theory and primitive recursion

In this section we focus on some elementary number-theoretic properties and their relation to primitive recursion.

Proposition 5.1. *The divisibility relation $y|x$ is primitive recursive.*

Proof. By definition $y|x$ is

$$\exists_{z\leq x}\, z \cdot y = x$$

Hence it is primitive recursive by Proposition 4.7. □

Proposition 5.2. *Let $\mathbb{P}(x)$ denote the property that x is prime. Then $\mathbb{P}$ is primitive recursive.*

Proof. By definition

$$\mathbb{P}(x) = \forall_{y\leq x}\, \big(\neg\,(y|x) \vee (y = 1) \vee (y = x)\,\big)$$

Hence it is primitive recursive by results of previous section. □

Theorem 5.3. *Let p_x denote the x-th prime number. Then the function*

$$x \mapsto p_x$$

is primitive recursive.

For the proof we need some elementary result.

Lemma 5.3.1. *The factorial function*

$$x \mapsto x!$$

is primitive recursive.

Proof of the lemma. Left for the reader as an exercise. □

Proof of the theorem. Consider the total 2-ary function

$$f(x,y) = \mu_{t<y}\left((x<t)\wedge \mathbb{P}(t)\right)$$

According to Proposition 5.2 and Proposition 4.11 we infer that f is primitive recursive. By Lemma 5.3.1 we derive that

$$g(x) = f(x, x!+2)$$

is primitive recursive. Note that $g(x)$ is the smallest prime number greater than x.

Now we have $p_0 = 2$ and

$$p_{x+1} = g(p_x)$$

for every x. Hence $x \mapsto p_x$ is obtained by primitive recursion from $g \cdot I^2_2$ and constant function equal to 2. Therefore, it is primitive recursive. □

Proposition 5.4. *For numbers x, y we define*

$$e_{p_y}(x) = \begin{cases} \max\left\{t \,|\, p_y^t \text{ divides } x\right\} & \text{if } x>0 \\ 0 & \text{if } x=0 \end{cases}$$

Then the function $(x,y)\mapsto e_{p_y}(x)$ is primitive recursive.

Proof. Consider the 3-ary relation

$$N(x,y,t) = \neg\left(p_y^t | x\right)$$

Then N is primitive recursive by previous results. Note that

$$e_{p_y}(x) = \begin{cases} \mu_{t<y} N(x,y,t) \dot{-} 1 & \text{if } x>0 \\ 0 & \text{otherwise} \end{cases}$$

Then results of the previous section and the fact that the monus is primitive recursive imply that $(x,y)\mapsto e_{p_y}(x)$ is primitive recursive. □

6. Sequences as primitive recursive data structures

In this section we explore how finite sequences of natural numbers can be represented and manipulated within the framework of primitive recursive functions. We will define encoding schemes for sequences and demonstrate that common operations on sequences are primitive recursive.

Definition 6.1. We define

$$[\vec{x}] = \begin{cases} p_0^{1+x_0}\cdot \ldots \cdot p_k^{1+x_k} & \text{if } \vec{x} = (x_0,...,x_k)\in\mathbb{N}^{k+1} \text{ for some } k \\ 1 & \text{if } \vec{x} \text{ is empty tuple} \end{cases}$$

Then $[\vec{x}]$ is the *sequence number of* $\vec{x}$.

Proposition 6.2. *Let* $\mathrm{Seq}(x)$ *denote that x is the sequence number of some tuple. Then* Seq *is primitive recursive unary relation.*

Proof. Consider binary relation

$$R(x,y) = \forall_{z\le y}\Big(\left(e_{p_z}(x)>1\right)\vee(z=0)\Big)$$

which states that $e_{p_z}(x) > 1$ for all nonzero $z \le y$. Hence R is primitive recursive. Next consider binary relation

$$N(x,y) = \forall_{z<x}\Big((y<z)\wedge\neg(p_z|x)\Big)$$

which states that x is not divisible by p_z for all z such that $y<z<x$. Clearly N is primitive recursive.

It follows that

$$\mathrm{Seq}(x) = (x > 0) \wedge \exists_{y<x}\Big(R(x,y) \wedge N(x,y)\Big)$$

Hence $\mathrm{Seq}(x)$ is primitive recursive. □

Definition 6.3. We define

$$\mathrm{lh}(x) = \begin{cases} k & \text{if } x = [\vec{x}] \text{ for some } \vec{x} \in \mathbb{N}^k \\ 0 & \text{otherwise} \end{cases}$$

Then $\mathrm{lh}(x)$ is the *length* of x.

Proposition 6.4. *The function $x \mapsto \mathrm{lh}(x)$ is primitive recursive.*

Proof. Binary relation

$$\mathrm{PDiv}(x,z) = (z > 0) \wedge (p_z | x)$$

is primitive recursive. It follows that the function $x \mapsto \sum_{z<x} \mathbb{1}_{\mathrm{PDiv}}(x,z)$ is primitive recursive. Then

$$\mathrm{lh}(x) = \begin{cases} \sum_{z<x} \mathbb{1}_{\mathrm{PDiv}}(x,z) & \text{if } \mathrm{Seq}(x) \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive and the proposition is proved. □

Proposition 6.5. *For numbers x, y define*

$$x[y] = \begin{cases} x_y & \textit{if } x = [\vec{x}] \textit{ for some } \vec{x} \in \mathbb{N}^{k+1} \textit{ and } 0 \leq y \leq k \\ 0 & \textit{otherwise} \end{cases}$$

Then the function $(x,y) \mapsto x[y]$ is primitive recursive.

Proof. We have

$$x[y] = \begin{cases} e_{p_y}(x) \dot{-} 1 & \text{if } \mathrm{Seq}(x) \text{ and } 0 \leq y \leq \mathrm{lh}(x) \\ 0 & \text{otherwise} \end{cases}$$

and hence the proposition follows. □

Proposition 6.6. *Let $k \in \mathbb{N}$. Then the k-ary function*

$$\mathbb{N}^k \ni \vec{x} \mapsto [\vec{x}] \in \mathbb{N}$$

is primitive recursive.

Proof. Left for the reader. □

7. Register machines and programs

We now define abstract computing machine.

Definition 7.1. A *register machine* consists of the following components.

- A set of *registers* numbered by elements of $\mathbb{N}$. Each register is can store a number in $\mathbb{N}$. If $r \in \mathbb{N}$ is a register, then we denote by $c_r \in \mathbb{N}$ its content.
- A *program counter* capable of storing a number in $\mathbb{N}$. We denote its content by pc.
- A *halt* state.
- Three types of *instructions*.

  INC r for $r \in \mathbb{N}$: This instruction increments c_r and pc.

DEC r for $r \in \mathbb{N}$:

If $c_r > 0$, then the instruction decrements c_r and increments pc by 2.

If $c_r = 0$, then the instruction increments pc.

JUMP q for $q \in \mathbb{Z}$:

If $q > 0$, then the instruction increments pc by q.

If $q < 0$ and $|q| \leq pc$, then the instruction decrements pc by $|q|$.

If $q < 0$ and $pc < |q|$, then the instruction causes the machine to halt.

Remark 7.2. The reader familiar with modern computers will recognize that a register machine is a mathematical model of a Harvard architecture central processing unit with random-access memory.

Definition 7.3. A finite sequence of instructions is a *register-machine program*.

If $\mathcal{P}$ is a register-machine program, then we denote by $\mathrm{lh}(\mathcal{P})$ the number of instructions in $\mathcal{P}$. Moreover, if $0 \leq i < \mathrm{lh}(\mathcal{P})$, then $\mathcal{P}[i]$ denotes i-th instruction in $\mathcal{P}$.

Definition 7.4. A *state* of a register machine consists of all but finitely many registers storing zeros. Moreover, if the program counter is set to zero, then the state is *initial*.

Definition 7.5. Let $\mathcal{P}$ be a register-machine program. Suppose that a register machine starts in some initial state. Then *execution of $\mathcal{P}$ starting from this initial state* follows iterative prcedure described below:

At the beginning of each iteration the machine reads the content of the program counter.

If the content of program counter is smaller than $\mathrm{lh}(\mathcal{P})$, then the machine modifies contents of its registers and the program counter according to $\mathcal{P}[pc]$. Note that $\mathcal{P}[pc]$ may cause the machine to halt. If the instruction does not cause the machine to halt, then the next iteration begins.

If the content of program counter is greater or equal to $\mathrm{lh}(\mathcal{P})$, then the machine halts.

Remark 7.6. Note that for every register-machine program and every initial state of registers execution of the program starting from this initial state may either continue indefinitely or halt at some iteration.

We now provide examples of register-machine programs, which we use later as subprograms of larger programs.

Example 7.7. We define CLEAR r for $r \in \mathbb{N}$ as the following program.

DEC r

JMP 2

JMP -2

Note that the effect of executing this program on a register machine is replacing the content of register r by zero.

Example 7.8. We define MOV r, s for distinct $r, s \in \mathbb{N}$ as the following program.

CLEAR s

DEC r

```
JMP 3
INC s
JMP −3
```

Note that the effect of executing this program on a register machine is twofold. First it replaces the content of register s by the initial content of register r. Second it replaces the initial content of r by zero.

Example 7.9. We define COPY r, s, t for distinct $r, s, t \in \mathbb{N}$ as the following program.

```
CLEAR t
CLEAR s
DEC r
JMP 4
INC t
INC s
JMP −4
MOV t, r
```

Note that the effect of executing this program on a register machine is threefold. First it replaces the content of register s by the initial content of register r. It preserves the initial content of register r. Finally it replaces the content of register t by zero.

Now we explain the relation between register-machine programs and partial functions.

Definition 7.10. Let $\mathcal{P}$ be a register-machine program and suppose that a register-machine with some initial state is given. Assume that $\mathcal{P}$ is executed on the machine. If the machine halts with program counter equal to $\mathrm{lh}(\mathcal{P})$, then the execution *terminates*.

Definition 7.11. Let $\vec{x} \in \mathbb{N}^k$ for some k and let $r_1, ..., r_k \in \mathbb{N}$ be registers. Consider an initial state of a register machine given by $c_{r_i} = x_i$ for $i \in \{1, ..., k\}$ and potentially some other registers storing nonzero numbers. As usual assume that the machines program counter is set to zero. Then the machine is *initialized with $\vec{x}$ in registers $r_1, ..., r_k$*.

Definition 7.12. Let $\mathcal{P}$ be a register-machine program and let f be a k-ary partial function for some number k. Suppose that the following assertions hold.

(1) If $\vec{x} \in \mathbb{N}^k$ is any tuple in the complement of the domain of f, then executing $\mathcal{P}$ on a register machine initialized with $\vec{x}$ in registers $1, ..., k$ and with all other registers storing zero results in the machine running indefinitely.

(2) If $\vec{x} \in \mathbb{N}^k$ is any k-tuple in the domain of f, then executing $\mathcal{P}$ on a register machine initialized with $\vec{x}$ in registers $1, ..., k$ and with all other registers storing zero terminates the execution and results in $f(\vec{x})$ stored in register 0 upon termination.

Then f is *register-computable* and *$\mathcal{P}$ computes f*.

We prove now that some very basic functions are register-computable.

Fact 7.13. *Let k be a number. Then zero constant k-ary function is register-computable.*

Proof. Indeed, it is computed by the empty register-machine program. □

Fact 7.14. *The successor function is register-computable.*

Proof. Indeed, the register-machine program

INC 1

MOV 1, 0

computes the successor function. □

Fact 7.15. *Let k, n be numbers. Then function I^k_n is register-computable.*

Proof. Indeed, the register-machine program

MOV k, 0

computes I^k_n. □

These are relatively simple examples. To demonstrate that the class of register-computable functions includes more complex functions, we introduce the generalized notion of function computation.

Definition 7.16. Let $\mathcal{P}$ be a register-machine program and let f be a k-ary partial function for some number k. Let $r_1, ..., r_k, r$ be distinct registers of a register machine and let l be a number. Suppose that the following assertions hold.

(1) If $\vec{x} \in \mathbb{N}^k$ is in the complement of the domain of f, then a register machine initialized with $\vec{x}$ in registers $r_1, ..., r_k$ and executing $\mathcal{P}$ runs indefinitely.

(2) If $\vec{x} \in \mathbb{N}^k$ is any k-tuple in the domain of f, then execution of $\mathcal{P}$ on a register machine initialized with $\vec{x}$ in registers $r_1, ..., r_k$ terminates, preserves initial contents of registers $0, ..., l-1$ (excluding r if $r < l$) and stores $f(\vec{x})$ in register r upon termination.

Then $\mathcal{P}$ *computes f from $r_1, ..., r_k$ to r preserving the first l registers.*

Now we prove very useful result.

Proposition 7.17. *Let f be a k-ary register-computable function for some number k. Then for every distinct registers $r_1, ..., r_k, r$ and every number l there exists a register-machine program that computes f from $r_1, ..., r_k$ to r preserving the first l registers.*

Proof. Suppose that $\mathcal{P}$ is a register-machine program that computes f. Our goal is to construct a register-machine program $\mathcal{Q}$ that computes f from $r_1, ..., r_k$ to r and preserves the first l registers.

Note that $\mathcal{P}$ is by definition a finite sequence of instructions. Suppose that $M \in \mathbb{N}$ is greater than $r_1, ..., r_k, r, l$ and all registers occurring in instructions of $\mathcal{P}$.

Having defined these numbers we construct $\mathcal{Q}$. We proceed in stages. Each stage is a register-machine subprogram and $\mathcal{Q}$ is the concatenation of these subprograms.

(1) MOV $i, M+i$ for $i = 0, ..., M-1$.

(2) COPY $M + r_i, i, 0$ for $i = 1, ..., k$.

(3) $\mathcal{P}$

(4) MOV $0, r$

(5)

$$\begin{cases} \text{MOV } M+i, i \text{ for } i = 0, ..., l-1 & \text{if } r \geq l \\ \text{MOV } M+i, i \text{ for } i = 0, ..., l-1 \text{ with } r \text{ excluded} & \text{if } r < l \end{cases}$$

We now analyze $\mathcal{Q}$ and show that it satisfies the statement. Fix $\vec{x} \in \mathbb{N}^k$. Assume that a register machine is initialized with $\vec{x}$ in registers $r_1, ..., r_k$. We execute $\mathcal{Q}$ on the machine. After **(1)** and **(2)** all registers $0, ..., M-1$ are zero except that $\vec{x}$ is stored in registers $1, ..., k$. Next **(3)** is executing $\mathcal{P}$. We have two options which we analyze separately.

- Assume that $\mathcal{P}$ runs indefinitely when the machine is initialized with $\vec{x}$ in registers $1, ..., k$ and all other registers storing zero. By the definition of M and considering that registers $0, ..., M-1$ store zeros except for registers $1, ..., k$ which store $\vec{x}$ we infer that **(3)** runs indefinitely.
- Assume that $\mathcal{P}$ terminates when the machine is initialized with $\vec{x}$ in registers $1, ..., k$ and all other registers storing zero. By the definition of M and considering that registers $0, ..., M-1$ store zeros except for registers $1, ..., k$ which store $\vec{x}$ we infer that **(3)** results in $f(\vec{x})$ in register 0. Moreover, the program counter points to **(4)** upon termination of **(3)**. Next **(4)** moves $f(\vec{x})$ to register r. Finally **(5)** recovers the initial contents of the first l registers except for r if $r < l$.

This completes the proof. □

Now we use the proposition above to prove that class of register-computable functions is closed under certain operations.

Proposition 7.18. *The class of register-computable functions is closed under composition.*

Proof. Fix numbers k, n. Let g be a k-ary register-computable function and let $f_1, ..., f_k$ be n-ary register-computable functions. Let h be the composition of g with $f_1, ..., f_k$. Our goal is to prove that h is register-computable.

For every $i = 1, ..., k$ let $\mathcal{P}_i$ be a register-machine program that computes f_i from

$$\underbrace{n \cdot (i-1) + 1, ..., n \cdot i}_{\text{consecutive numbers}}$$

to $n \cdot k + i$ preserving the first $n \cdot k + k + 1$ registers.

Next let $\mathcal{P}$ be a register-machine program that computes g from

$$\underbrace{n \cdot k + 1, ..., n \cdot k + k}_{\text{consecutive numbers}}$$

to 0.

These programs exist according to Proposition 7.17.

Consider now the following register-machine program.

(1) COPY $j, n \cdot i + j, 0$ for $j = 1, ..., n$ for $i = 1, ..., k-1$.

(2) $\mathcal{P}_1, ..., \mathcal{P}_k$

(3) $\mathcal{P}$

Suppose that the register machine is initialized with $\vec{x} \in \mathbb{N}^n$ stored in registers $1, ...n$. Then **(1)** results in

$$\left(\underbrace{\vec{x}, ..., \vec{x}}_{k \text{ times}}\right)$$

stored in registers $1, ..., n \cdot k$. Now we need to distinguish three cases.

- All $f_1(\vec{x}), ..., f_k(\vec{x})$ and $g\,(f_1(\vec{x}), ..., f_k(\vec{x}))$ are defined. Then **(2)** terminates with each $f_i(\vec{x})$ stored in register $n \cdot k + i$ for $i = 1, ..., k$. Next the machine executes **(3)**, which also terminates and results in
$$h(\vec{x}) = g\,(f_1(\vec{x}), ..., f_k(\vec{x}))$$
stored in register 0.
- All $f_1(\vec{x}), ..., f_k(\vec{x})$ are defined but $g\,(f_1(\vec{x}), ..., f_k(\vec{x}))$ is undefined. Then **(2)** terminates and upon termination $f_i(\vec{x})$ is stored in register $n \cdot k + i$ for $i = 1, ..., k$. Next the machine executes **(3)**, which runs indefinitely.
- At least one of $f_1(\vec{x}), ..., f_k(\vec{x})$ is undefined. Then **(2)** runs indefinitely.

This completes the proof. □

Proposition 7.19. *The class of register-computable functions is closed under primitive recursion.*

Proof. Fix a number k. Let f be a k-ary register computable partial function and let g be a $(k+2)$-ary register-computable partial function. Suppose that h is obtained by primitive recursion from g and f. Our goal is to construct a register-machine program which computes h.

Let $\mathcal{P}$ be a register-machine program that computes f from $1, ..., k$ to register $k+3$ by preserving the first $k+4$ registers. Similarly let $\mathcal{Q}$ be a register-machine program that computes g from $\underbrace{1, ..., k}_{\text{consecutive registers}}, k+2, k+3$ to register $k+4$ while preserving the first $k+4$ registers. Both these programs exist according to Proposition 7.17.

Consider now the following register-machine program.

(1) $\mathcal{P}$

(2) DEC $k+1$

(3) JMP $\mathrm{lh}(\mathcal{Q}) + 4$

(4) $\mathcal{Q}$

(5) INC $k+2$

(6) MOV $k+4, k+3$

(7) JMP $-2 - \mathrm{lh}(\mathcal{Q}) - 2$

(8) MOV $k+3, 0$

We now analyze this program and show that it satisfies the statement.

First we analyze the subprogram consisting of block **(2)-(7)**. Suppose that $\vec{x} \in \mathbb{N}^k$ is stored in registers $1, ..., k$, the number $t \in \mathbb{N}$ is stored in register $k+1$, the number $s \in \mathbb{N}$ in register $k+2$ and $h(\vec{x}, s)$ in register $k+3$. Moreover, suppose the program counter is points to instruction **(5)**. We distinguish three cases.

- Assume that $t > 0$ and $h(\vec{x}, s+1)$ is defined. The instruction **(2)** decrements the content of register $k+1$ and execution continues at **(4)**. That is the machine executes $\mathcal{Q}$. Since $h(\vec{x}, s+1)$ is defined, it follows that $g(\vec{x}, s, h\,(\vec{x}, s)\,)$ is defined and $\mathcal{Q}$ terminates. Upon its termination $h(\vec{x}, s+1)$ is stored in register $k+4$ and the first $k+4$ registers are preserved. The program counter points to **(5)**. Hence instructions **(5)**,**(6)** are executed. This results in the following state:

  $\vec{x}$ is stored in registers $1, ..., k$,

  $t-1$ is stored in register $k+1$,

$s+1$ is stored in register $k+2$,

$h(\vec{x}, s+1)$ is stored in register $k+3$.

Finally **(7)** is executed and the program counter points to **(2)**.

- Assume that $t > 0$ and $h(\vec{x}, s+1)$ is undefined. Then **(2)** decrements the content of register $k+1$ and execution continues at **(4)**. That is the machine executes $\mathcal{Q}$. Since $h(\vec{x}, s+1)$ is undefined, this implies that $g\big(\vec{x}, s, h\left(\vec{x}, s\right)\big)$ is undefined. Thus $\mathcal{Q}$ runs indefinitely. Hence **(4)** causes the machine to run indefinitely.
- Assume that $t = 0$. Then **(3)** is executed and the program counter jumps to **(8)**.

Now fix $\vec{x} \in \mathbb{N}^k$ and $y \in \mathbb{N}$. Assume that a register machine is initialized with $\vec{x}$ in registers $1, ..., k$ and with y in register $k+1$. We now execute the program. If f is not defined for $\vec{x}$, then **(1)** runs indefinitely and $h(\vec{x}, 0)$ is undefined. Thus we may assume that f is defined for $\vec{x}$, then **(1)** terminates and $h(\vec{x}, 0) = f(\vec{x})$ is stored in register $k+3$. Moreover, the first $k+4$-registers are preserved. From our analysis of the subprogram consisting of the code block **(2)**-**(7)** we infer the following. Either $h(\vec{x}, y)$ is undefined and then **(4)** runs indefinitely or $h(\vec{x}, y)$ is defined and then the execution of subprogram results in $h(\vec{x}, y)$ stored in register $k+3$ and the program counter pointing to **(8)**. In the latter case $h(\vec{x}, y)$ is moved to register 0 and the whole program terminates. □

Proposition 7.20. *The class of register-computable functions is closed under unbounded minimization.*

Proof. Fix a number k. Let g be a $(k+1)$-ary register computable partial function. Suppose that f is the result of unbounded minimization of g. Our goal is to construct a register-machine program which computes f.

Let $\mathcal{P}$ be a register-machine program that computes g from $1, ..., k, k+1$ to register $k+2$ by preserving the first $k+2$ registers. This program exists according to Proposition 7.17.

Consider now the following register-machine program.

(1) $\mathcal{P}$

(2) DEC $k+2$

(3) JMP 3

(4) INC $k+1$

(5) JMP $-3-\mathrm{lh}(\mathcal{P})$

(6) MOV $k+1, 0$

We now analyze this program and show that it satisfies the statement.

Suppose that $\vec{x} \in \mathbb{N}^k$ is stored in registers $1, ..., k$ and register $k+1$ stores a number $t \in \mathbb{N}$. Moreover, suppose that the program counter points to **(1)**. We distinguish three cases.

- Assume that $g(\vec{x}, t)$ is defined and nonzero. Then the machine executes **(1)** that is program $\mathcal{P}$. Since $g(\vec{x}, t)$ is defined, the subprogram **(1)** terminates preserving the contents of the first $k+2$ registers and stores $g(\vec{x}, t)$ in the register $k+2$. Next by assumption $g(\vec{x}, t)$ is nonzero and thus **(2)** causes a jump to **(4)** which increments register $k+1$. This results in the following state:

  $\vec{x}$ remains in registers $1, ..., k$,

  $t+1$ is stored in register $k+1$

  Finally **(5)** is executed and the program counter points again to **(1)**.

- Assume that $g(\vec{x}, t) = 0$. The machine executes **(1)** that is program $\mathcal{P}$. Since $g(\vec{x}, t)$ is defined, we derive that the subprogram **(1)** terminates and stores $g(\vec{x}, t) = 0$ in register $k + 2$. Thus **(2)** increments the program counter and **(3)** is executed. This causes the machine to jump to **(6)**, which moves t from register $k + 1$ to register 0 and the program terminates.
- Assume that $g(\vec{x}, t)$ is undefined. Then the machine executes **(1)** that is program $\mathcal{P}$. Since $g(\vec{x}, t)$ is undefined, it runs indefinitely.

Suppose that the machine is initialized with $\vec{x} \in \mathbb{N}^k$ in registers $1, ..., k$. We use the analysis above to reason about the program's behavior.

If $f(\vec{x})$ is defined, then the program terminates and stores $f(\vec{x})$ in register 0 upon termination.

Assume now that $f(\vec{x})$ is undefined. Then $t \mapsto g(\vec{x}, t)$ is defined for some initial interval of $\mathbb{N}$, but its range does not include zero.

- If the interval of definition is finite, then the program runs indefinitely with the program counter stuck at instructions within block **(1)**.
- If $t \mapsto g(\vec{x}, t)$ is defined for $t \in \mathbb{N}$, then the entire program runs indefinitely.

This completes the proof. $\square$

8. Gödel encoding and Kleene's T predicate

In this section we introduce core ideas in theory of computation and logic.

Definition 8.1. The numbers

$$\begin{aligned} \#\mathrm{INC}\, r &= [1, r] \\ \#\mathrm{DEC}\, r &= [2, r] \\ \#\mathrm{JMP}\, q &= \begin{cases} [3, q] & \text{if } q \geq 0 \\ [4, |q|] & \text{if } q < 0 \end{cases} \end{aligned}$$

are *Gödel number* of register machine instructions.

Definition 8.2. Let $\mathcal{P}$ be a register machine program. Then the number

$$\#\mathcal{P} = [\#\mathcal{P}[0], ..., \#\mathcal{P}[\mathrm{lh}(\mathcal{P}) - 1]]$$

is the *Gödel number* of $\mathcal{P}$.

Definition 8.3. Let

$$c_0, c_1, c_2, ..., c_k, ...$$

be contents of registers describing a state of some register machine. A *memory number* of the state is

$$2^{c_0} \cdot 3^{c_1} \cdot 5^{c_2} \cdot ... \cdot p_k^{c_k} \cdot ...$$

Note that this number is well defined according to the fact that all but finitely many $c_k = 0$.

Encodings defined above are primitive recursive according to the following result.

Fact 8.4. *Unary relations*

$$\begin{aligned} \mathrm{Ins}(x) &= x \textit{ is the Gödel number of some register machine instruction} \\ \mathrm{Prog}(x) &= x \textit{ is the Gödel number of some register machine program} \end{aligned}$$

are primitive recursive.

Proof. Left for the reader as an exercise. □

Using Gödel and memory numbers we can define functions which describe operation of register machine.

We begin with the following function.

$$\mathrm{mem}(c,m) = \begin{cases} m \cdot p_{c[1]} & \text{if } \mathrm{Ins}(c) \text{ and } c[0] = 1 \\ \lfloor m / p_{c[1]} \rfloor & \text{if } \mathrm{Ins}(c) \text{ and } c[0] = 2 \text{ and } e_{p_{c[1]}}(m) > 0 \\ m & \text{otherwise} \end{cases}$$

Note that mem is primitive recursive.

Proposition 8.5. *Let c be the Gödel number of some instruction and let m be the memory number of a state of some register machine. Then* $\mathrm{mem}(c,m)$ *is the memory number of the state obtained after executing instruction encoded by c on the state of a register machine described by m.*

Proof. Follows immediately from the definition of mem. □

Next we define

$$\mathrm{pc}(pc,e,m) = \begin{cases} pc & \text{if } \mathrm{Prog}(e) \text{ and } pc \geq \mathrm{lh}(e) \\ pc+1 & \text{if } \mathrm{Prog}(e) \text{ and } pc < \mathrm{lh}(e) \text{ and } e[pc][0] = 1 \\ pc+2 & \text{if } \mathrm{Prog}(e) \text{ and } pc < \mathrm{lh}(e) \text{ and } e[pc][0] = 2 \text{ and } e_{p_{e[pc][1]}}(m) > 0 \\ pc+1 & \text{if } \mathrm{Prog}(e) \text{ and } pc < \mathrm{lh}(e) \text{ and } e[pc][0] = 2 \text{ and } e_{p_{e[pc][1]}}(m) = 0 \\ pc + e[pc][1] & \text{if } \mathrm{Prog}(e) \text{ and } pc < \mathrm{lh}(e) \text{ and } e[pc][0] = 3 \\ pc \dot{-} e[pc][1] & \text{if } \mathrm{Prog}(e) \text{ and } pc < \mathrm{lh}(e) \text{ and } e[pc][0] = 4 \text{ and } pc \geq e[pc][1] \\ \mathrm{lh}(e) & \text{otherwise} \end{cases}$$

Clearly pc is a primitive recursive function.

Proposition 8.6. *Let e be the Gödel number of some register-machine program and let m be the memory number of some register-machine state. Then* $\mathrm{pc}(pc,e,m)$ *is the content of the program counter after executing pc-th instruction of a program encoded by e on the state of a register machine described by m.*

Proof. Follows immediately from the definition of pc. □

Next we define $\mathrm{snap}(e,m,t)$ for numbers e,m,t. We set

$$\mathrm{snap}(e,m,0) = [0,m]$$

Suppose now that $\mathrm{snap}(e,m,t)$ is defined for some e,m,t. Then we set

$$\mathrm{snap}(e,m,t+1) =$$

$$= \Big[\mathrm{pc}\big(\mathrm{snap}(e,m,t)[0], e, \mathrm{snap}(e,m,t)[1]\big), \mathrm{mem}\big(e[\mathrm{snap}(e,m,t)[0]], \mathrm{snap}(e,m,t)[1]\big)\Big]$$

Note that the function

$$(e,m,t,z) \mapsto \Big[\mathrm{pc}\big(z[0],e,z[1]\big), \mathrm{mem}\big(e[z[0]], z[1]\big)\Big]$$

is primitive recursive. Since snap is obtained by primitive recursion from the function above and the function $(e,m,0) \mapsto [0,m]$, we derive that snap is primitive recursive.

Proposition 8.7. *Let e be the Gödel number of some register-machine program and let m be the memory number of some state. Then* $\mathrm{snap}(e,m,t)$ *is the sequence number*

$$[\text{program counter}, \text{memory number}]$$

after t-iterations of execution of register-machine program encoded by e on a machine with inital state described by m.

Proof. The proof goes by induction on t. Inductive step is a consequence of the definition of snap, Proposition 8.5 and Proposition 8.6. □

Next for k-tuple $\vec{x} = (x_1, ..., x_k)$ we define

$$\mathrm{memorize}(\vec{x}) = {p_1}^{x_1} \cdot ... \cdot p_k^{x_k}$$

It follows that memorize is a primitive recursive k-ary function.

Proposition 8.8. *Let k be a number and let $\vec{x}$ be a k-tuple. Then* memorize$(\vec{x})$ *is the memory number of the state of a register machine with $\vec{x}$ in registers $1, ..., k$ and all other registers storing zeros.*

Proof. Follows from the definition of memory numbers. □

Fix $k \in \mathbb{N}$. We define

$$T_k(e, s, \vec{x}) =$$

$$= \mathrm{Seq}(s) \wedge \mathrm{Prog}(e) \wedge \big(0 < \mathrm{lh}(s)\big) \wedge \forall_{t<\mathrm{lh}(s)} \Big(s[t] = \mathrm{snap}(e, \mathrm{memorize}(\vec{x}), t)\Big) \wedge$$

$$\wedge \forall_{t<\mathrm{lh}(s) \dot{-} 1} \Big(s[t][0] < \mathrm{lh}(e)\Big) \wedge \big(s[\mathrm{lh}(s) \dot{-} 1][0] = \mathrm{lh}(e)\big)$$

for $e, s \in \mathbb{N}$ and $\vec{x} \in \mathbb{N}^k$.

9. Universal functions and Kleene normal form

In this section we prove theorems which are at heart of computability theory.

Theorem 9.1. *Let f be a k-ary partial recursive function and let e be the Gödel number of a register-machine program that computes f.*

Definition 9.2. Let k be a number and let $\mathcal{C}$ be a class of k-ary partial functions. Consider a $(k+1)$-ary partial function Φ. Suppose that the following assertions hold. for every k-ary partial recursive function f there exists number e such that the following assertions hold.

(1) For every function $f \in \mathcal{C}$ there exists a number e such that

$$\{\vec{x} \in \mathbb{N}^k \,|\, \Phi(e, \vec{x}) \downarrow\} = \{\vec{x} \in \mathbb{N}^k \,|\, f(\vec{x}) \downarrow\}$$

and

$$\Phi(e, \vec{x}) = f(\vec{x})$$

for every $\vec{x} \in \mathbb{N}^k$ for which $f(\vec{x})$ is defined.

(2) For every number e the k-ary partial function

$$\vec{x} \mapsto \Phi(e, \vec{x})$$

is contained in $\mathcal{C}$.

Then Φ is a *universal* for $\mathcal{C}$.

Theorem 9.3 (universal function)**.** *Let k be a number. Then there exists a $(k+1)$-ary partial function Φ such that the following holds.*

(1) *Φ is recursive.*

(2) *Φ is universal for k-ary partial recursive functions.*

(3) *Φ is universal for k-ary partial register-computable functions.*

Proof. We define

$$\Phi(e,\vec{x}) = e_2\Big(\mathrm{snap}\Big(e, \mathrm{memorize}(\vec{x}), \mu_t\left[\mathrm{lh}(e) \dot{-} \mathrm{snap}(e, \mathrm{memorize}(\vec{x}), t)[0] = 0\right]\Big)[1]\Big)$$

Then Φ is $(k+1)$-ary partial recursive function.

Since every partial recursive function is register-machine computable, we infer that Φ is $(k+1)$-ary partial recursive function. Fix number e. Then definition of recursive functions and Proposition 7.18 imply that a k-ary partial function

$$\vec{x} \mapsto \Phi(e,\vec{x})$$

is both recursive and register-computable.

Suppose now that f is a k-ary partial register-computable function. Let $\mathcal{P}$ be a register-machine program that computes f and let $e = \#\mathcal{P}$ be its Gödel number. Consider two cases.

$f(\vec{x})\downarrow$ does not hold. Then

$$0 \leq \mathrm{snap}\big(e, \mathrm{memorize}(\vec{x}), t\big) < \mathrm{lh}(e)$$

for all $t \in \mathbb{N}$. In particular, $\Phi(e,\vec{x})$ is undefined.

$f(\vec{x})$ is defined. Then combining Proposition 8.7 and Proposition 8.8 we infer that $\Phi(e,\vec{x})$ is the content of register 0 when $\mathcal{P}$ terminates execution which starts with $\vec{x}$ in registers $1,...,k$ and all the other registers set to zero. By definition the content of register 0 upon termination is $f(\vec{x})$. Hence $f(\vec{x}) = \Phi(e,\vec{x})$.

This proves that Φ is universal for k-ary register-computable functions. Thus it is also universal for k-ary recursive functions. □

Corollary 9.4. *Classes of register-computable and register functions coincide.*

Corollary 9.5.

10. Rudimentary and ∃-rudimentary relations

In this section we introduce important class of formulas expressing arithmetical relations.

Definition 10.1. First order language with the signature consisting of the following symbols:

the constant 0

binary relation symbols $=, <$

unary function symbol S

2-ary function symbols $+, \cdot$

is the *language of arithmetic*.

The intended model of the language of arithmetic is

$$\mathfrak{N} = (\mathbb{N}, 0, s, <, +, \cdot)$$

with the usual meaning of $0, <, +, \cdot$ and s denoting the succesor function $s(n) = n+1$ for every $n \in \mathbb{N}$.

Remark 10.2. Let ϕ be a first order formula in the language of arithmetic and let x, y be variables. Then we introduce the following notation

$$x \leq y \equiv (x = y) \vee (x < y)$$
$$\forall_{x<y}\, \phi \equiv \forall_x \big((x < y) \wedge \phi\big)$$
$$\forall_{x\leq y}\, \phi \equiv \forall_x \big((x \leq y) \wedge \phi\big)$$

$$\exists_{x<y}\,\phi \equiv \exists_x\,\big(\,(x<y)\wedge\phi\big)$$
$$\exists_{x\leq y}\,\phi \equiv \exists_x\,\big(\,(x\leq y)\wedge\phi\big)$$

Definition 10.3. The class of *rudimentary formulas* is the smallest class of formulas in the language of arithmetic such that the following conditions hold.

(1) Every atomic formula is rudimentary.

(2) If ϕ, ψ are rudimentary formulas, then
$$\phi\wedge\psi, \phi\vee\psi$$
are rudimentary.

(3) If ϕ is a rudimentary formula, then $\neg\phi$ is rudimentary.

(4) if ϕ is a rudimentary formula and x, y are variables, then also
$$\forall_{x<y}\,\phi,\ \exists_{x<y}\phi$$
are rudimentary.

Fact 10.4. *Let ϕ be a rudimentary formula and let x, y be variables. Then formulas*
$$\forall_{x\leq y}\,\phi,\ \exists_{x\leq y}\,\phi$$
are rudimentary.

Proof. Left for the reader as an exercise. □

Definition 10.5. Let ϕ be a rudimentary formula and let x be a variable. Then
$$\forall_x\,\phi$$
is a $\exists$-*rudimentary* formula.

Remark 10.6. Let $n\in\mathbb{N}$. Then the term
$$\underbrace{S...S}_{n\text{ times}}(0)$$
of the language of arithmetic is denoted by $\overline{n}$. If ϕ is a formula in the language of arithmetic with free variables $x_1, ..., x_k$ and $n_1, ..., n_k\in\mathbb{N}$, then
$$\phi^{x_1,...,x_k}_{\overline{n}_1,...,\overline{n}_k}$$
is denoted by $\phi(\overline{n}_1, ..., \overline{n}_k)$.

Definition 10.7. Let $R\subseteq\mathbb{N}^k$ be a k-ary relation. Suppose that there exists rudimentary formula ϕ with k distinct free variables such that
$$R(n_1, ..., n_k)\ \Leftrightarrow\ \mathfrak{N}\vdash\phi(n_1, ..., n_k)$$
for all $n_1, ..., n_k\in\mathbb{N}$.