Ján Komara and Paul J. Voda

Lecture Notes in Theory of Computability

Institute of Informatics, Comenius University, Bratislava

December 5, 2001

Preface

See the preface of [Vod00].

Table of Contents

List of Figures

1. Functions

1.1 Introduction

See the paragraphs 1.1.10 - 1.1.16 in [Vod00].

1.2 Basic Functions

1.2.1 Functions. If we do not state explicitly n-ary functions are over the domain of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$ with $n \geq 1$.

1.2.2 Terms. We will deal with syntactic objects called terms. *Terms* are expressions formed as sequences of concrete symbols. The terms we are interested in are formed from variables and constants by applications of function symbols and simple conditionals (see below). We use $\tau, \rho, \dots$ as *syntactic* variables ranging over terms. We write $\tau_1 \equiv \tau_2$ if the two terms are identical sequences of symbols. We denote by $\|\tau\|$ the *size*, i.e. the number of operations, of the term τ.

Simple conditionals are terms of a form $\mathcal{D}_s(\tau_1, \tau_2, \tau_3)$ which can be visualized as

$$\textbf{if } \tau_1 \neq 0 \textbf{ then } \tau_2 \textbf{ else } \tau_3$$

by the notation common in computer programming.

We say that the term τ is in $x_1, \dots, x_n$ if all its free variables are from among the variables $x_1, \dots, x_n$. *Closed* terms do not have free variables.

We will write $\vec{x}$ in contexts like $f(\vec{x})$ where f is an n-ary function as an abbreviation for a sequence of n variables, i.e. $f(x_1, \cdots, x_n)$. Generally, $f(\vec{\tau})$ will be an abbreviation for $f(\tau_1, \dots, \tau_n)$ where $\tau_1, \dots, \tau_n$ is a sequence of terms. We will also write $f\,g(\vec{\tau})$ instead of $f(g(\vec{\tau}))$.

1.2.3 Substitution in terms. When we write $\tau[f; \vec{x}]$ we *indicate* that the term τ may apply the n-ary function symbol f and variables from among the m-variables $\vec{x}$. For an n-ary function symbol g and for an m-tuple of terms $\vec{\rho}$ we write $\tau[g; \vec{\rho}]$ for the term obtained from τ by the *substitution* of terms $\vec{\rho}$ for the corresponding variables of $\vec{x}$ as well as by the replacement of all applications $f(\vec{\tau})$ by applications $g(\vec{\tau})$.

We will also use the *special lambda notation* $\tau[\dot{\lambda}\vec{y}.\rho[\vec{y}];\vec{x}]$ where $\vec{y}$ are n variables for the term obtained from τ by the replacement of all applications $f(\vec{\tau})$ by terms $\rho[\vec{\tau}]$. Note that we have $\tau[g;\vec{x}] \equiv \tau[\dot{\lambda}\vec{y}.g(\vec{y});\vec{x}]$.

1.2.4 Formulas. We let all binary propositional connectives in formulas group to the right. We assign the highest precedence to the quantifiers $\forall x$, $\exists y$, and to the negation $\neg$. Next lower precedence has the conjunction $\wedge$ and then the disjunction $\vee$. The connectives of implication $\rightarrow$ and equivalence $\leftrightarrow$ have the lowest precedence. The propositional constants are $\top$ (true) and $\bot$ (falsehood). By $\forall\phi$ we denote the universal closure of the formula ϕ.

Bounded quantifiers are formulas of forms $\exists x \leq \tau\, \phi$ and $\forall x \leq \tau\, \phi$, where the variable x does not occur freely in τ. Bounded quantifiers are abbreviations for the formulas $\exists x(x \leq \tau \wedge \phi)$ and $\forall x(x \leq \tau \rightarrow \phi)$ respectively. We permit also strict bounds on existential quantifiers: $\exists x < \tau\, \phi$ as abbreviation for $\exists x < \tau\, (x \neq \tau \wedge \phi)$. Strict bounds on universal quantifiers are similar.

1.2.5 Modified subtraction. The domain of natural numbers is not closed under the operation of subtraction. Instead of subtraction we use the binary *modified subtraction* function $x \dot{-} y$ which is over $\mathbb{N}$ and satisfies:

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

1.2.6 Integer division and remainder. The domain of natural numbers is not closed under the operation of division. Instead of division we use the binary *integer division* $x \div y$ function, which together with the binary *remainder* function $x \bmod y$, satisfies:

$$y > 0 \rightarrow x = (x \div y) \cdot y + x \bmod y \wedge x \bmod y < y$$
$$x \div 0 = x \bmod 0 = 0.$$

1.2.7 Successor and predecessor functions. The *successor* function S satisfies $S(x) = x + 1$. The *predecessor* function Pr satisfies $Pr(x) = x \dot{-} 1$. We will usually write $\tau + 1$ instead of $S(\tau)$ and $\tau \dot{-} 1$ instead of $Pr(\tau)$.

1.2.8 Identity functions. For each $n \geq 1$ and $1 \leq i \leq n$ the n-ary *identity* function I_i^n yields its i-th argument:

$$I_i^n(x_1, \ldots, x_i, \ldots, x_n) = x_i.$$

As a special case we set $I = I_1^1$ and so we have $I(x) = x$.

1.2.9 Constant functions. For each $n \geq 1$ the n-ary *constant* functions are the n-ary functions C_m^n yielding m, i.e. $C_m^n(\vec{x}) = m$ for every natural numbers $\vec{x}$. We customarily denote the unary constant functions C_m^1 by C_m, i.e. we have $C_m(x) = m$. As a special case when $m = 0$ we set $Z = C_0$ and call it the *zero* function. So we have $Z(x) = 0$.

1.2.10 Case discrimination function. The ternary *case discrimination* function D satisfies the following identities:

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

1.2.11 Boolean functions. The boolean *negation* function $\neg_* x$ is such that

$$\neg_* x = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{otherwise.} \end{cases}$$

The boolean *conjunction* function $x \wedge_* y$ is such that

$$x \wedge_* y = \begin{cases} 1 & \text{if } x > 0 \text{ and } y > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The remaining boolean functions $x \vee_* y$ (*disjunction*), $x \rightarrow_* y$ (*implication*), and $x \leftrightarrow_* y$ (*equivalence*) are defined similarly.

1.2.12 Constant iteration of functions. For an unary function g and a constant c we will write $g^c(\tau)$ as an abbreviation for the c-fold *iteration* of the application of g:

$$\overbrace{g\, g \cdots g}^{c}(\tau).$$

We do not exclude the case $c = 0$ when $g^0(\tau)$ is the same term as τ.

1.2.13 Predicates. If we do not state the domain of a *predicate* we are concerned with n-ary predicates over $\mathbb{N}$ ($n \geq 1$), i.e subsets of the cartesian product $\mathbb{N}^n$. Unary predicates are subsets of $\mathbb{N}$.

We will denote by R^c the *complement* of an n-ary predicate R satisfying $R^c(\vec{x}) \leftrightarrow \neg R(\vec{x})$. The *graph* of an n-ary function f is the $(n+1)$-ary predicate $f(\vec{x}) = y$.

Characteristic function of an n-ary predicate R is the n-ary function R_* such that

$$R_*(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}), \\ 0 & \text{otherwise.} \end{cases}$$

We denote by $x =_* y$, $x \neq_* y$, $x \leq_* y$, $x <_* y$, $x \geq_* y$, and $x >_* y$ the characteristic functions of the binary predicates $x = y$, $x \neq y$, $x \leq y$, $x < y$, $x \geq y$, and $x > y$, respectively.

1.3 Pairing Functions

1.3.1 Introduction. Computer programming, in addition to the standard numerical types, involves a large number of *data structures* such as n-tuples, multidimensional arrays (vectors and matrices), lists, stacks, tables, trees, graphs, etc. Standard programming languages (both imperative and functional ones) therefore work with quite complicated domains obtained by solutions of recursive identities. We think that this is an unnecessary complication and we look for a solution to the programming language LISP which offers excellent coding of symbolic data structures into the domain of *S-expressions*. This domain is freely generated from the set of countably many *atoms* by a binary operation *cons*. There is no advantage in having infinitely many atoms, just one, say, *nil* suffices. There is also no advantage of having S-expressions as a separate domain. Nothing is lost and much is gained by the arithmetization of the domain of S-expressions in $\mathbb{N}$.

1.3.2 Pairing function. We obtain the coding convenience of LISP in the domain of natural numbers by arithmetizing the domain from the preceding paragraph with the help of a suitable binary *pairing* function (x, y). The function, which will be defined in Par. 1.3.8, has the following properties:

$$(x_1, y_1) = (x_2, y_2) \rightarrow x_1 = x_2 \wedge y_1 = y_2 \tag{1}$$

$$x < (x, y) \wedge y < (x, y) \tag{2}$$

$$x = 0 \vee \exists y \exists z\, x = (y, z). \tag{3}$$

Property (1) is called the *pairing property* and it ensures that for every number n in the range of the pairing function, i.e. such that $n = (x, y)$ for some x and y, the numbers x and y, called the *first* and *second projections of* n respectively, are uniquely determined. The pairing property says that the pairing function is an injection. From the property (2) we get $0 \leq x < (x, y)$. This means that 0 is not in the range of the pairing function and so it has no projections, i.e.

$$0 \neq (x, y). \tag{4}$$

Thus the number 0 is an *atom* and plays the role of the atom *nil* of LISP. Property (3) asserts that the pairing function is onto the set $\mathbb{N} \setminus \{0\}$, i.e. that 0 is the only atom.

1.3.3 Pair induction. For any unary predicate $R(x)$ we have the principle of *pair induction*:

$$R(0) \wedge \forall x \forall y \big(R(x) \wedge R(y) \rightarrow R((x, y))\big) \rightarrow \forall x\, R(x),$$

i.e. in order to prove that R holds for all natural numbers it suffices to prove $R(0)$ and $R((x, y))$ under the inductive hypotheses $R(x)$ and $R(y)$.

The principle of pair induction is easily reduced to (mathematical) induction as follows. Assume $R(0)$ and

$$\forall x \forall y \big(R(x) \land R(y) \to R((x,y))\big) \tag{1}$$

and prove

$$\forall x (x < n \to R(x)) \tag{2}$$

by induction on n. In the base case, when $n = 0$, there is nothing to prove.

In the inductive case take any x such that $x < n+1$ and consider two cases. If $x = 0$ then we have $R(x)$ from the assumption $R(0)$. If $x > 0$ then we have $x = (v, w)$ for some numbers v and w by 1.3.2(3). Since $v < x$ and $w < x$ by 1.3.2(2), we obtain $R(v)$ and $R(w)$ from IH (2). We then get $R((v,w))$ from (1). This ends the proof of (2) and we obtain $\forall x R(x)$ by substituting $x+1$ for n in (2).

Instead of the unary predicate $R(x)$ we can also use $(n+1)$-ary predicates $R(x, y_1, \ldots, x_n)$ with parameters.

1.3.4 Pair recursion. Pair induction is used to prove properties of functions defined by *pair recursion.* The schema of pair recursion introduces an $(n+1)$-ary function f from two functions g (which is n-ary) and h (which is $(n+4)$-ary). The function f satisfies the following recurrences:

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}) \\ f((v,w), \vec{y}) &= h(v, w, \vec{y}, f(v, \vec{y}), f(w, \vec{y})). \end{aligned}$$

The schema is justified because its recursion decreases the first-argument in the relation $<$.

We also admit *parameterless* pair recursion defining an unary f from a constant n and a four argument function h to satisfy:

$$\begin{aligned} f(0) &= n \\ f(v,w) &= h(v, w, f(v), f(w)). \end{aligned}$$

1.3.5 Pair numerals. It can be easily proved by pair induction that every natural number n can be uniquely presented as a term called *pair numeral*. The class of pair numerals is the least class of terms containing 0 and with every two terms τ_1 and τ_2 also the term (τ_1, τ_2). We call this the *pair representation* of $\mathbb{N}$.

1.3.6 Pair size. The length of the pair numeral τ denoting the number x is $5 \cdot n + 1$ where n is the number of pairing operations used in the construction of the term τ. The arithmetization of the length function is the *pair size* function $|x|_p$ yieding the number of pairing operations needed for the construction of the pair numeral denoting x. The function is defined by parameterless pair recursion to satisfy:

$$|0|_p = 0$$
$$|(x,y)|_p = |x|_p + |y|_p + 1.$$

$J(x,y)$	0	1	2	3	4	5	6	$\cdots$
0	1_1	2_2	4_3	7_3	11_4	16_4	22_4	$\cdots$
1	3_2	5_3	8_4	12_4	17_5	23_5	30_5	$\cdots$
2	6_3	9_4	13_5	18_5	24_6	31_6	39_6	$\cdots$
3	10_3	14_4	19_5	25_5	32_6	40_6	49_6	$\cdots$
4	15_4	20_5	26_6	33_6	41_7	50_7	60_7	$\cdots$
5	21_4	27_5	34_6	42_6	51_7	61_7	72_7	$\cdots$
6	28_4	35_5	43_6	52_6	62_7	73_7	85_7	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Fig. 1.1. Cantor's pairing function $J(x,y)$

1.3.7 Cantor's pairing function. Without fixing the pairing function (x,y) we do not know the pair representation of any natural number except 0 and our next step is to determine the function. We first note that the standard *diagonal* function J of Cantor (see [Dav58]) when offset by one (to account for the atom 0), i.e. the function

$$J(x,y) = (x+y) \cdot (x+y+1) \div 2 + x + 1,$$

satisfies the properties 1.3.2(1),(2),(3). The initial segment of J is tabulated in Fig. 1.1. The subscripts of values for $J(x,y)$ give the pair size $|J(x,y)|_p$.

Unfortunately, the pairing function J is not suitable for us because it cannot be used for the development of small functional computational classes such as polynomial time, polynomial space, linear space etc. This was demonstrated in [Vod95]. The reason why J is not suitable can be seen when we enumerate the natural numbers in the J-pair representation. The numbers with the same pair size are not grouped together as it is the case with the dyadic size of numbers enumerated in the dyadic notation. We can see from Fig. 1.1 that in the middle of the group of consecutive numbers 4 through 10 we have two numbers with the pair size 4, namely $8 = J(J(0,0), J(0, J(0,0)))$ and $9 = J(J(0, J(0,0)), J(0,0))$, while the remaining numbers have the pair size three.

We will see in Par. 1.3.8 that as a consequence of keeping the numbers with the same pair size together we will have $|x|_p = \Theta(\lg(x))$, i.e. $|x|_p = O(\lg(x))$ and $\lg(x) = O(|x|_p)$. This property assures a natural characterization by pairing of computational complexity classes such as polynomial time or space (see [Vod95]).

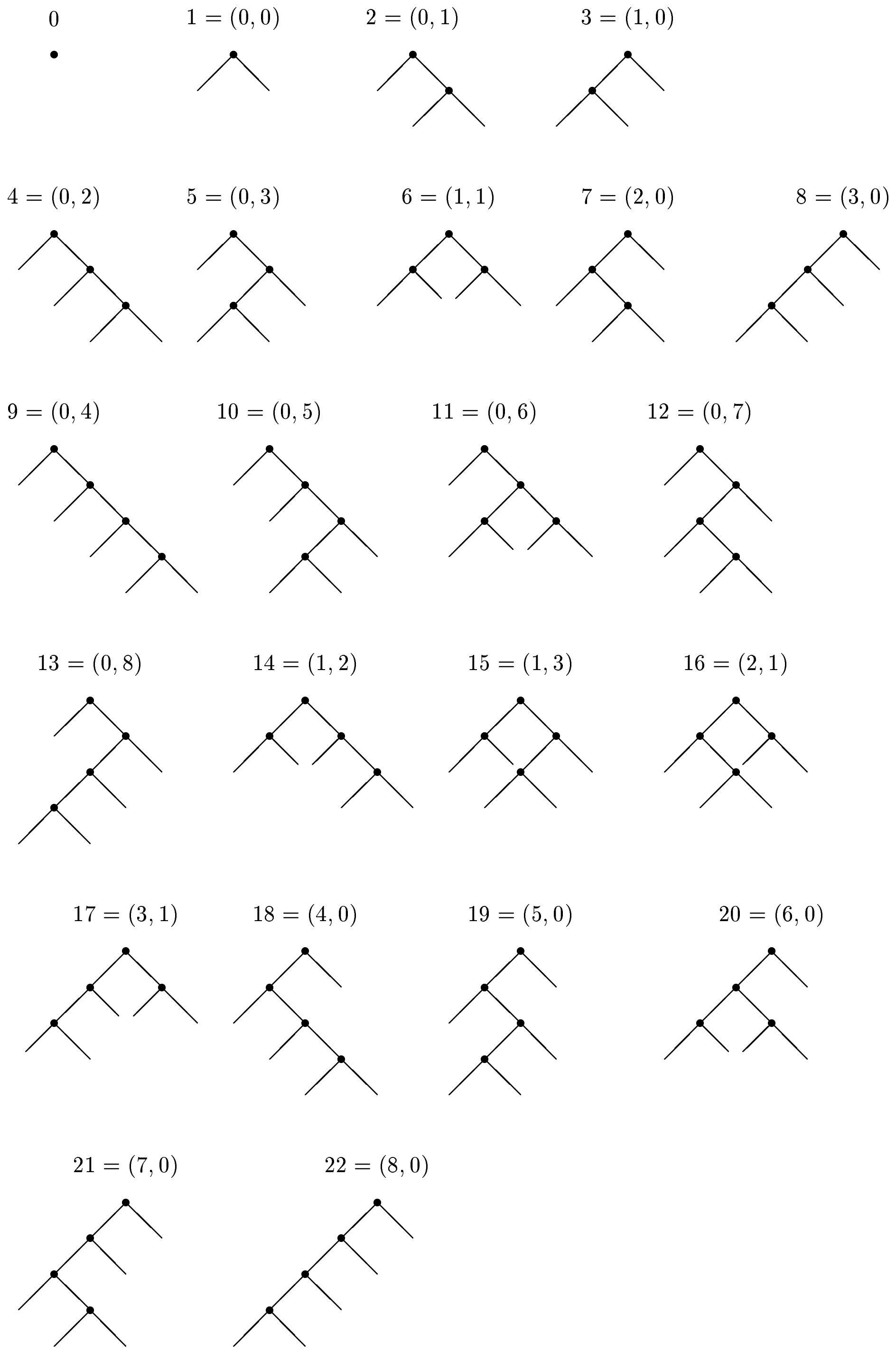

Fig. 1.2. Enumeration of binary trees

1.3.8 Suitable pairing function. We obtain a suitable pairing function (x, y) by keeping the numbers with the same pair size together. For that we note that every natural number in pair representation can be viewed as a *binary tree*. Here 0 is represented by the *empty* tree and the number (x, y) is represented by a tree with the left subtree representing x and the right subtree representing y. Note that that the number of inner nodes of the tree representing x is $|x|_p$. We *enumerate* all binary trees by listing the binary trees with the lesser number of inner nodes before the ones with larger number of inner nodes. Two different binary trees t_1 and t_2 with the same number of inner nodes are listed *lexicographically*. This means that t_1 is listed before t_2 if its left subtree is listed before that of t_2, or if the left subtrees are identical, the right subtree of t_1 is listed before that of t_2. An initial segment of the enumeration is given in Fig. 1.2.

The enumeration of binary trees uniquely fixes the pairing function (x, y). Namely, for two numbers x and y we take the x-th and y-th binary trees t_1 and t_2 (counting from zero). The position of the binary tree $\langle t_1, t_2\rangle$ is the value of (x, y). Fig. 1.3 lists the initial segment of values of (x, y) in a tabular form. The subscripts give the pair size $|(x, y)|_p$.

(x, y)	0	1	2	3	4	5	6	$\cdots$
0	1_1	2_2	4_3	5_3	9_4	10_4	11_4	$\cdots$
1	3_2	6_3	14_4	15_4	37_5	38_5	39_5	$\cdots$
2	7_3	16_4	42_5	43_5	121_6	122_6	123_6	$\cdots$
3	8_3	17_4	44_5	45_5	126_6	127_6	128_6	$\cdots$
4	18_4	46_5	131_6	132_6	399_7	400_7	401_7	$\cdots$
5	19_4	47_5	133_6	134_6	404_7	405_7	406_7	$\cdots$
6	20_4	48_5	135_6	136_6	409_7	410_7	411_7	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Fig. 1.3. Suitable pairing function (x, y)

1.3.9 Projections. The pairing property (3) asserts that every non-zero number x has a form y, z for some numbers y and z which are uniquely determined by (1). *Projection* functions are such that $H(x) = y$ and $T(x) = z$.

The *first projection* function H (head) is a unary function which satisfies the following identities:

$$
\begin{aligned}
H(0) &= 0 \\
H((y, z)) &= y.
\end{aligned}
$$

The *second projection* function T (tail) is a unary function which satisfies the following identities:

$$T(0) = 0$$
$$T((y, z)) = z.$$

It should be clear that we have

$$x = (H(x), T(x)) \leftrightarrow x \neq 0. \tag{1}$$

1.3.10 Ordering properties of (x, y). From the enumeration of binary trees given in Par. 1.3.8 we immediately obtain:

$$x < y \rightarrow |x|_p \leq |y|_p \tag{1}$$
$$|x|_p < |y|_p \rightarrow x < y \tag{2}$$
$$|(x_1, x_2)|_p = |(y_1, y_2)|_p \rightarrow ((x_1, x_2) < (y_1, y_2) \leftrightarrow x_1 < y_1 \vee x_1 = y_1 \wedge x_2 < y_2). \tag{3}$$

The pairing function is strictly monotone in both arguments:

$$x_1 < x_2 \rightarrow (x_1, y) < (x_2, y) \tag{4}$$
$$y_1 < y_2 \rightarrow (x, y_1) < (x, y_2). \tag{5}$$

Indeed, the property (4) holds because we get $|x_1|_p \leq |x_2|_p$ from $x_1 < x_2$ by (1). We then consider two cases. If $|x_1|_p < |x_2|_p$ then also $|(x_1, y)|_p < |(x_2, y)|_p$ and we get the conclusion from (2). If $|x_1|_p = |x_2|_p$ then also $|(x_1, y)|_p = |(x_2, y)|_p$ and we get the conclusion from (3). Property (5) is proved similarly.

1.3.11 Functions σ and C. We denote by $\sigma(n)$ the first number in the enumeration of pairs with with the pair size n and by $C(n)$ the total number of numbers with the pair size n, i.e. the total number of binary trees with n inner nodes. The function $C(n)$ is known as the *Catalan* function (see [GKP89]) and satisfies the obvious *convolution* recurrences:

$$C(0) = 1 \tag{1}$$
$$C(n+1) = \sum_{i \leq n} C(i) \cdot C(n-i) \tag{2}$$

because every binary tree with $n+1$ inner nodes has the left son with $i \leq n$ inner nodes and the right son with $n-i$ inner nodes. We clearly have

$$\sigma(0) = 0 \tag{3}$$
$$\sigma(n+1) = \sigma(n) + C(n). \tag{4}$$

We will need the following properties:

$$\sigma(|x|_p) \leq x < \sigma(|x|_p + 1) \tag{5}$$
$$|\sigma(n)|_p = n \tag{6}$$

$$\sigma(n) \leq C(n) \tag{7}$$
$$2^n \leq \sigma(n+1). \tag{8}$$

Properties (5) and (6) hold directly from the definition of σ. Property (7) holds trivially for $n = 0$. If $n > 0$ then the property holds because every number $x < \sigma(n)$, i.e. such that $|x|_p < n$, can be injectively mapped into the number $(\sigma(n-1-|x|_p), x)$ with the pair size n. Property (8) is proved by induction on n. In the base case we have $2^0 = 1 = \sigma(0+1)$. In the inductive case we have

$$2^{n+1} = 2 \cdot 2^n \overset{\text{IH}}{\leq} 2 \cdot \sigma(n+1) \overset{(7)}{\leq} \sigma(n+1) + C(n+1) = \sigma(n+2).$$

1.3.12 Some properties of the pairing function. We will need the following properties:

$$2^{|x+1|_p} \leq 2 \cdot (x+1) \tag{1}$$
$$2^{|x|_p} \leq 2 \cdot x + 1 \tag{2}$$
$$C(n+1) \leq 4^n \tag{3}$$
$$x < 4^{|x|_p} \tag{4}$$
$$|x|_p \leq x \tag{5}$$
$$(x,y) \leq 4 \cdot (2 \cdot x + 1)^2 \cdot (2 \cdot y + 1)^2. \tag{6}$$

Property (1): We have $|x+1|_p = n+1$ for some n and so

$$2^{|x+1|_p} = 2 \cdot 2^n \overset{1.3.11(8)}{\leq} 2 \cdot \sigma(n+1) = 2 \cdot \sigma(|x+1|_p) \overset{1.3.11(5)}{\leq} 2 \cdot (x+1).$$

Property (2) follows from (1) and from $2^{|0|_p} = 1 = 2 \cdot 0 + 1$.

(3): If we write a number with the pair size $n+1$ in the pair representation fully parenthesized and omit the left parentheses and commas we obtain a string of)'s and 0's of the form $00\alpha)$ where the length of α is $2n$. This proves (3) because there cannot be more strings α than $2^{2 \cdot n} = 4^n$.

(4): We have

$$x \overset{1.3.11(5)}{<} \sigma(|x|_p + 1) \overset{1.3.11(7)}{\leq} C(|x|_p + 1) \overset{(3)}{\leq} 4^{|x|_p}.$$

Property (5) is proved by induction on x. The base case is trivial. In the inductive case we get $x + 1 \leq \sigma(|x|_p + 1)$ from 1.3.11(5) and so we have

$$|x+1|_p \overset{1.3.10(1)}{\leq} |\sigma(|x|_p + 1)|_p \overset{1.3.11(6)}{=} |x|_p + 1 \overset{\text{IH}}{\leq} x + 1.$$

(6): We have

$$(x,y) \overset{(4)}{<} 4^{|(x,y)|_p} = 4^{|x|_p + |y|_p + 1} = 4 \cdot (2^{|x|_p})^2 \cdot (2^{|y|_p})^2 \overset{(2)}{\leq} 4 \cdot (2 \cdot x + 1)^2 \cdot (2 \cdot y + 1)^2.$$

1.3.13 The pairing function (x, y) is suitable. We will now demonstrate that the pairing function (x, y) is a suitable pairing function, i.e. that $|x|_p = \Theta(\lg(x))$. Indeed, for $x > 0$ we have

$$2^{|x|_p} \overset{1.3.12(1)}{\leq} 2 \cdot x \leq 2^{\lg(x)+2}$$

and so $|x|_p \leq \lg(x) + 2$, i.e. $|x|_p = O(\lg(x))$. On the other hand, we have

$$x \overset{1.3.12(4)}{<} 4^{|x|_p} = 2^{2 \cdot |x|_p}$$

and so for $x > 0$ we have $\lg(x) < 2 \cdot |x|_p$, i.e. $\lg(x) = O(|x|_p)$.

1.3.14 Contraction to unary functions. We now establish a natural correspondence between n-ary and unary functions and predicates over $\mathbb{N}$.

If f is an n-ary function then we denote by $\langle f \rangle$ its *contraction*, which is an unary function defined as follows:

$$\langle f \rangle(x) = \begin{cases} f(x_1, \ldots, x_{n-1}, x_n) & \text{if } x = (x_1, \ldots, x_{n-1}, x_n), \\ 0 & \text{otherwise,} \end{cases}$$

where we write $(x_1, \ldots, x_{n-1}, x_n)$ as an abbreviation for

$$(x_1, \ldots, (x_{n-1}, x_n) \ldots).$$

From the above we get

$$f(x_1, \ldots, x_{n-1}, x_n) = \langle f \rangle((x_1, \ldots, x_{n-1}, x_n)). \tag{1}$$

We note that the contraction $\langle f \rangle$ of an unary function f is the function f itself. For $n > 1$ we clearly have

$$\begin{aligned} &\exists x_1 \ldots \exists x_{n-1} \exists x_n \, x = (x_1, \ldots, x_{n-1}, x_n) \Leftrightarrow \\ &x = (H\,T^0(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x)) \Leftrightarrow \\ &T^{n-2}(x) > 0 \end{aligned}$$

and thus we have

$$\langle f \rangle(x) = \begin{cases} f(H\,T^0(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x)) & \text{if } T^{n-2}(x) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

If R is an n-ary relation then we denote by $\langle R \rangle$ its *contraction* which is an unary predicate, satisfying:

$$\langle R \rangle(x) \leftrightarrow \exists x_1 \ldots \exists x_{n-1} \exists x_n \big(x = (x_1, \ldots, x_{n-1}, x_n) \land R(x_1, \ldots, x_{n-1}, x_n)\big).$$

From this we get

$$R(x_1, \ldots, x_{n-1}, x_n) \leftrightarrow \langle R \rangle((x_1, \ldots, x_{n-1}, x_n)).$$

Thus in the presence of pairing there is no essential difference between n-ary and unary functions, except that the expression $f(x)$ is meaningless for an n-ary function whenever $n > 1$ but well-defined for its contraction $\langle f \rangle(x)$.

1.3.15 Conventions for the symbol comma. We will usually omit the outermost pairing parentheses around pairing (τ_1, τ_2). Thus, for instance, we can write $f(x) = g(x), h(x)$ instead of $f(x) = (g(x), h(x))$. We postulate that the pairing operator ',' groups to the right, i.e. that (τ_1, τ_2, τ_3) abbreviates $(\tau_1, (\tau_2, \tau_3))$. We assign the lowest precedence to pairing. Thus $x + y \cdot v, z$ is an abbreviation for $((x + (y \cdot v)), z)$.

The omission of parentheses around pairing can lead to ambiguities in situations where commas could be confused with the separators of arguments of n-ary functions. In such situations when we write $f(\tau_1, \ldots, \tau_n, \ldots, \tau_{n+m})$ we (obviously) treat the commas as separators of arguments if $m = 0$ and we understand the expression as an abbreviation for $f(\tau_1, \ldots, (\tau_n, \ldots, \tau_{n+m}))$ if $m > 1$. Thus the first $n-1$ commas separate the arguments while the remaining ones are the infix pairing operators. We adopt similar comma conventions for n-ary predicates.

For instance, if f is binary then $f((\tau_1, \tau_2), (\tau_3, \tau_4, \tau_5))$ can be abbreviated by the dropping of the outermost pairing parentheses to $f((\tau_1, \tau_2), \tau_3, \tau_4, \tau_5)$ The reader will note that we cannot drop the parentheses around τ_1, τ_2. If R is an unary predicate then $R((\tau_1, \tau_2))$ can be written as $R(\tau_1, \tau_2)$.

We extend the dual role of commas to sequences like $\vec{x}$ and $\vec{\tau}$. For instance, when f is an n-ary function then the basic property of its contraction $\langle f \rangle$ is:

$$\langle f \rangle(x_1, \ldots, x_n) = f(x_1, \ldots, x_n).$$

The commas on the left of the identity stand for the pairing operator while they are separators on the right. We can abbreviate this to $\langle f \rangle(\vec{x}) = f(\vec{x})$ with the same understanding about the commas in the sequence $\vec{x}$.

1.4 Partial Functions

1.4.1 Partial functions. A *partial* n-ary function f is a mapping with the domain S which is a subset of the cartesian product $\mathbb{N}^n$ and with the range a subset of $\mathbb{N}$. The partial function f is *total*, or just a function, if $S = \mathbb{N}^n$.

For every number n we denote by $\emptyset^{(n)}$ the nowhere defined n-ary partial function, i.e. the partial function with the empty domain $\emptyset \subseteq \mathbb{N}^n$.

1.4.2 Graphs of partial functions. The *graph* of an n-ary partial function f is the $(n+1)$-ary relation denoted by $f(\vec{x}) \asymp y$ which holds if f is defined for $\vec{x}$ with the value y. We clearly have for (total) functions f:

$$f(\vec{x}) \asymp y \leftrightarrow f(\vec{x}) = y.$$

1.4.3 Completion of partial functions. We say that the (total) function g is the *completion* of a partial function f if the following holds:

$$g(\vec{x}) = \begin{cases} y & \text{if } f(\vec{x}) \asymp y, \\ 0 & \text{otherwise.} \end{cases}$$

1.4.4 Chains of partial functions. For two partial functions f_1 and f_2 of the same arity we write $f_1 \subseteq f_2$ if the inclusion holds set-theoretically, i.e. if we have

$$f_1(\vec{x}) \asymp y \rightarrow f_2(\vec{x}) \asymp y$$

for all $\vec{x}$ and y. Note that $\emptyset^{(n)} \subseteq f$ for every n-ary partial function f.

An infinite sequence of partial functions $f_0, f_1, f_2, \ldots$ of the same arity is a *chain* if we have $f_i \subseteq f_{i+1}$ for all i. It should be clear that the set-theoretical union of this chain, which is written as $\bigcup_i f_i$, is a partial function for which we have

$$(\bigcup_i f_i)(\vec{x}) \asymp y \leftrightarrow \exists i\, f_i(\vec{x}) \asymp y \tag{1}$$

$$f_i \subseteq \bigcup_i f_i. \tag{2}$$

We also have

$$\forall i\, f_i \subseteq f \rightarrow \bigcup_i f_i \subseteq f \tag{3}$$

because if $(\bigcup_i f_i)(\vec{x}) \asymp y$ holds then we have $f_i(\vec{x}) \asymp y$ for some i by 1.4.4(1) and $f(\vec{x}) \asymp y$ from the assumption.

1.4.5 Partial denotation of terms. *Interpretation* of a term is an assignment of concrete partial functions to the function symbols applied in the term. Closed terms obtain meaning by interpretation and they *partially denote* natural numbers. Terms with free variables partially denote natural numbers only subject to an *assignment* of natural numbers as values of their variables.

A term $\tau[\vec{x}]$ applying partial functions, i.e. with some partial functions assigned to the function symbols of τ, may fail to denote and in this respect it acts as a partial function of $\vec{x}$ arguments. For this reason we introduce for each term τ the formula $\tau \asymp y$, called the *graph of* τ, defined by induction on the structure of τ:

$$x \asymp v \equiv x = v \tag{1}$$

$$n \asymp v \equiv n = v \tag{2}$$

$$f(\tau_1, \ldots, \tau_n) \asymp v \equiv \exists v_1 \cdots \exists v_n (\bigwedge_{i=1}^{n} \tau_i \asymp v_i \wedge f(v_1, \ldots, v_n) \asymp v) \tag{3}$$

$$\mathcal{D}_s(\tau_1, \tau_2, \tau_3) \asymp v \equiv \exists v_1 \big(\tau_1 \asymp v_1 \wedge (v_1 \neq 0 \wedge \tau_2 \asymp v \vee v_1 = 0 \wedge \tau_3 \asymp v)\big). \tag{4}$$

Straightforward induction on the structure of terms τ yields the following property of graphs of τ:

$$\tau[\vec{x}] \asymp y \wedge \tau[\vec{x}] \asymp z \rightarrow y = z.$$

A closed interpreted term τ is assigned as its *partial denotation* the number y if $\tau \asymp y$ holds. The term τ is *defined*, or it *converges*, if $\exists y\, \tau \asymp y$ holds, which we write as $\tau\downarrow$. Otherwise τ is *undefined*, or it *diverges* which we write as $\tau\uparrow$. A term is *total* if it applies only total functions. In such case we clearly have $\tau \asymp v \leftrightarrow \tau = v$.

Note that the interpretation of simple conditionals (4) is *non-strict* because the term $\tau \equiv \mathcal{D}_s(\tau_1, \tau_2, \tau_3)$ denotes only if τ_1 denotes and then if $\tau_1 \neq 0$ the term τ denotes the same number as the term τ_2 and if $\tau_1 = 0$ then the term τ denotes the same number as the term τ_3. Note that in the first case the term τ_3, and in the second case the τ_2, may diverge without affecting the denotation of τ. Note also that if all partial functions applied in τ are total then we have $\mathcal{D}_s(\tau_1, \tau_2, \tau_3) = D(\tau_1, \tau_2, \tau_3)$.

1.4.6 Use and mention of terms. In situations where we discuss the syntax and semantic of languages containing terms it is important to distinguish between their *use* and *mention*, i.e. between *extensions* and *intensions*, of terms. When a term τ is *mentioned* then we understand under τ the term itself, i.e. a finite sequence of symbols. Terms are most often mentioned in contexts like 'the term τ'. When the term τ is used then we understand it as the name of its denotation (subject to an intepretation and assignment).

When the term τ is defined, i.e. when $\tau \asymp n$ holds for some n, then we can **use** τ as a name for n. If τ is total we have:

$$\tau[\vec{x}] \asymp y \leftrightarrow \tau[\vec{x}] = y.$$

Note that the first occurrence of the term τ is mentioned whereas the second one is used.

We make the same distinction between the use and mention of function symbols. They are mentioned in contexts 'the function symbol f' and then we mean the sequence of symbols constituting the function symbol. Function symbols are used in contexts 'the function f' and then we mean the function interpreting the symbol f.

1.4.7 Non-strict identity. In the presence of partial functions we need the *non-strict* identity $\tau_1 \simeq \tau_2$ abbreviating the formula $\forall y(\tau_1 \asymp y \leftrightarrow \tau_2 \asymp y)$. Here y is a new variable. In other words, non-strict identity holds iff either both terms converge and denote the same number or else both diverge. It should be clear that if τ_1 and τ_2 are total then we have $\tau_1 \simeq \tau_2 \leftrightarrow \tau_1 = \tau_2$.

1.5 Exercises

2. Function Operators

2.1 Inductively Defined Function Classes

2.1.1 Classes of partial functions. We will denote by $\mathcal{F}$, $\mathcal{G}$, $\mathcal{H}$, and $\mathcal{I}$ (possibly with subscripts) *classes*, i.e. sets, of partial functions over $\mathbb{N}$. The class of all partial functions over $\mathbb{N}$ is denoted by $\mathcal{P}$. For arbitrary class $\mathcal{F}$ we denote by $\mathcal{F}^{(n)}$ the set of all n-ary partial functions of $\mathcal{F}$. Thus $\mathcal{P}^{(n)}$ is the set of all n-ary partial functions over $\mathbb{N}$.

For a class of partial functions $\mathcal{F}$ we say that the predicate R is an $\mathcal{F}$-*predicate* if its characteristic function R_* is in $\mathcal{F}$. By $\mathcal{F}_*$ we denote the class of all $\mathcal{F}$-predicates.

2.1.2 Function operators. We will encounter in the following many different kinds of function operators which are *functionals*, i.e. functions operating over partial functions. An n-ary *function operator* $\mathcal{O}$ is a functional, which takes n partial functions $f_1, \ldots, f_n$ as arguments and yields the partial function $f = \mathcal{O}(f_1, \ldots, f_n)$ as the result. We also admit $n = 0$ and then the operator $\mathcal{O}$ can be identified with the partial function it yields.

For an n-ary function operator $\mathcal{O}$ and a class of partial functions $\mathcal{F}$ we say that $\mathcal{F}$ *closed under* $\mathcal{O}$ if for any partial functions $f_1, \ldots, f_n$ from $\mathcal{F}$ also the partial function $f = \mathcal{O}(f_1, \ldots, f_n)$ yielded by the operator is in $\mathcal{F}$.

We are interested mostly in *effective* function operators. That is, a function operator is called effective (in intuitive sense) if when applied to effective partial functions from its domain the result of the application is also an effective partial function.

2.1.3 Inductively defined classes of functions. For a class of partial functions $\mathcal{I}$ and for a family of function operators $\{\mathcal{O}_i\}_{i\in I}$ we say that the class $\mathcal{F}$ of partial functions is *(inductively) generated from* $\mathcal{I}$ *by* $\{\mathcal{O}_i\}_{i\in I}$ if the following holds for every partial function f:

> $f \in \mathcal{F}$ iff f is obtained from $\mathcal{I}$ by finitely many applications of the function operators $\{\mathcal{O}_i\}_{i\in I}$.

The partial functions in $\mathcal{I}$ are called the *initial partial functions* of $\mathcal{F}$.

2.1.4 Theorem. *The class $\mathcal{F}$ generated from $\mathcal{I}$ by $\{\mathcal{O}_i\}_{i\in I}$ is the set smallest class containing $\mathcal{I}$ and closed under the function operators from $\{\mathcal{O}_i\}_{i\in I}$.*

Proof. Let us denote by $\{\mathcal{G}_\alpha\}_{\alpha\in J}$ the set of all classes containing $\mathcal{I}$ and closed under the function operators $\{\mathcal{O}_i\}_{i\in I}$. The set is non-empty since it contains at least the class of all partial functions $\mathcal{P}$. Let $\mathcal{G} = \bigcap_{\alpha\in J}\mathcal{G}_\alpha$. We wish to show that $\mathcal{G} = \mathcal{F}$.

The inclusion $\mathcal{G} \subseteq \mathcal{F}$ is obvious since it is easy to see that the class $\mathcal{F}$ contains $\mathcal{I}$ and it is closed under the function operators $\{\mathcal{O}_i\}_{i\in I}$. On the other hand every class $\mathcal{G}_\alpha$ contains each partial function obtained from $\mathcal{I}$ by finitely many applications of the function operators $\{\mathcal{O}_i\}_{i\in I}$. Thus $\mathcal{F} \subseteq \mathcal{G}_\alpha$ and hence $\mathcal{F} \subseteq \mathcal{G}$. □

2.1.5 Induction on the construction of inductively defined classes. The property of an inductively defined class $\mathcal{F}$ that it is the smallest class containing $\mathcal{I}$ and closed under the operators $\{\mathcal{O}_i\}_{i\in I}$ will be often used in the following to show that all functions f from $\mathcal{F}$ posses a property $\mathcal{P}(f)$ by *induction on the construction of functions f in $\mathcal{F}$*. For this it is sufficient to consider two cases. If $f \in \mathcal{I}$ we have to show $\mathcal{P}(f)$. If $f = \mathcal{O}(f_1,\ldots,f_n)$ then we have to show $\mathcal{P}(f)$ under the inductive hypotheses that the functions $f_1,\ldots,f_n$ posses the property.

2.1.6 Relatively defined function classes. Suppose that the class $\mathcal{F}$ is generated from $\mathcal{I}$ by the function operators $\{\mathcal{O}_i\}_{i\in I}$. For a class $\mathcal{G}$ we say that $\mathcal{G}$ *is $\mathcal{F}$-closed* if $\mathcal{I} \subseteq \mathcal{G}$ and $\mathcal{G}$ is closed under all operators $\{\mathcal{O}_i\}_{i\in I}$. Note that we then have $\mathcal{F} \subseteq \mathcal{G}$ by the minimality of $\mathcal{F}$.

For a class of partial functions $\mathcal{H}$ we say that the class $\mathcal{G}$ *is $\mathcal{F}$ in $\mathcal{H}$* if $\mathcal{G}$ is generated from $\mathcal{H}$ and $\mathcal{I}$ by the function operators $\{\mathcal{O}_i\}_{i\in I}$, or equivalently by Thm. 2.1.4, if $\mathcal{G}$ is the smallest $\mathcal{F}$-closed class containing $\mathcal{H}$.

2.1.7 Universal functions. For a given binary partial function $U(e,x)$ we say that the number e is an *index* of an n-ary partial function f if we have

$$\forall x_1 \ldots \forall x_n\, f(x_1,\ldots,x_n) \simeq U(e,(x_1,\ldots,x_n)). \tag{1}$$

We say that U is *universal* for a class $\mathcal{F}$ if the following holds

$$f \in \mathcal{F} \text{ iff } f \text{ has an index} \tag{2}$$

for every partial function f. Note that this is equivalent to

$$\mathcal{F} = \bigcup_{n\geq 1}\{\lambda x_1,\ldots,x_n.U(e,(x_1,\ldots,x_n)) \mid e \in \mathbb{N}\}. \tag{3}$$

2.1.8 Theorem. *Let $\mathcal{F}$ be a class of partial functions closed under explicit definitions containing a unary function g such that $g(x) \neq x$ for all x. Then there is no universal partial function for $\mathcal{F}$ such that its completion is in $\mathcal{F}$. In particular, the class $\mathcal{F}$ does not contain its own universal (total) function.*

Proof. Suppose that there is a universal partial function U for $\mathcal{F}$ such that its completion U_1 is in $\mathcal{F}$. Then the unary function f explicitly defined by

$$f(x) \simeq g\,U_1(x,x) \tag{1}$$

is in $\mathcal{F}$ and thus there is an index e of f w.r.t. U. Since U_1 is the completion of U then we have

$$f(x) = U_1(e,x) \tag{2}$$

for all x by 2.1.7(1). We obtain

$$f(e,e) \overset{(1)}{=} g\,U_1(e,e) \overset{(2)}{=} g\,f(e,e).$$

Contradiction. □

2.2 Explicit Definitions

2.2.1 Operator of composition. For every pair of numbers $m \geq 1$ and $n \geq 1$ the operator of *composition* takes an m-ary partial function h and m n-ary partial functions $g_1, \ldots, g_m$ and yields an n-ary partial function f satisfying:

$$f(\vec{x}) \simeq h(g_1(\vec{x}), \ldots, g_m(\vec{x})). \tag{1}$$

If the argument functions are total so is the function yielded by the composition operator. Note that then (1) is equivalent to

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_m(\vec{x})). \tag{2}$$

2.2.2 Conditional operator. For every $n \geq 1$ the *conditional* operator takes three n-ary partial functions g_1, g_2, and g_3, and yields an n-ary partial function f satisfying:

$$f(\vec{x}) \simeq \mathcal{D}_s(g_1(\vec{x}), g_2(\vec{x}), g_3(\vec{x})). \tag{1}$$

If the argument functions are total so is the function yielded by the conditional operators. Note that then (1) is equivalent to

$$f(\vec{x}) = D(g_1(\vec{x}), g_2(\vec{x}), g_3(\vec{x})). \tag{2}$$

2.2.3 Explicit definitions. *Explicit definitions of partial functions* are of a form

$$f(\vec{x}) \simeq \tau[\vec{x}], \tag{1}$$

where the term $\tau[\vec{x}]$ contains at most the indicated variables free and does not apply f. Clearly there is a unique partial function f satisfying (1). If all partial functions applied in $\tau[\vec{x}]$ are total so is the function yielded by the explicit definition. Note that then (1) is equivalent to

$$f(\vec{x}) = \tau[\vec{x}]. \tag{2}$$

Equations of the form (2) are called *explicit definitions of functions*.

Explicit definition (1) (and similarly (2)) can be viewed as a function operator taking all partial functions applied in τ and yielding the partial function f.

2.2.4 Unary contractions. Let $\mathcal{F}$ be a class closed under explicit definitions of functions containing the pairing function (x, y) and its projections $H(x)$ and $T(x)$. Then for every n-ary function f we have:

$$f \text{ is in } \mathcal{F} \text{ iff its unary contraction } \langle f \rangle \text{ is in } \mathcal{F}.$$

The claim trivially holds for unary functions. If $n > 1$ then we consider two cases. If $f \in \mathcal{F}$ then by 1.3.14(2) we obtain

$$\langle f \rangle(x) = \mathcal{D}_s(T^{n-2}(x), (H\,T^0(x), H\,T^1(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x)), 0)$$

and we can take the identity as an explicit definition of $\langle f \rangle$ in $\mathcal{F}$. The reverse implication follows from 1.3.14(1).

2.2.5 Boolean functions. If $\mathcal{F}$ is a class closed under explicit definitions of functions then $\mathcal{F}$ contains all boolean functions since they can be derived in $\mathcal{F}$ by the following explicit definitions:

$$\neg_* x = D(x, 0, 1) \tag{1}$$
$$x \wedge_* y = D(x, D(y, 1, 0), 0) \tag{2}$$
$$x \vee_* y = \neg_*(\neg_* x \wedge_* \neg_* y) \tag{3}$$
$$x \rightarrow_* y = \neg_* x \vee_* y \tag{4}$$
$$x \leftrightarrow_* y = (x \rightarrow_* y) \wedge_* (y \rightarrow_* x). \tag{5}$$

2.2.6 Bounded formulas. *Bounded* formulas are formulas constructed from *atomic* formulas: $\tau_1 < \tau_2$, $\tau_1 \leq \tau_2$, $\tau_1 > \tau_2$, $\tau_1 \geq \tau_2$, $\tau_1 = \tau_2$, and $Q(\vec{\tau})$ by propositional connectives and by bounded quantification. The terms of bounded formulas are composed from variables and constant symbols by applications of (total) functions.

2.2.7 Explicit definitions of predicates with bounded formulas. *Explicit definitions* of predicates *with bounded formulas* are of a form

$$R(\vec{x}) \leftrightarrow \phi[\vec{x}], \tag{1}$$

where ϕ is a bounded formula with at most the indicated n-tuple of variables free and without any application of the predicate symbol R.

If the equivalence (1) is such that the list of $g_1, \ldots, g_k$ ($k \geq 0$) of functions is minimal such that every function f applied in ϕ is in the list and if for every predicate Q applied in ϕ its characteristic function Q_* is in the list then the equivalence is *explicit definition of the predicate R from $g_1, \ldots, g_k$*. Every such explicit definition can be viewed as a k-place function operator yielding the characteristic function R_* of the predicate R from the functions $g_1, \ldots, g_k$.

2.2.8 Operator of bounded minimalization. For every $n \geq 1$ the operator of *bounded minimalization* takes an $(n+1)$-ary function g and yields an $(n+1)$-ary function f satisfying:

$$\begin{aligned} f(z, \vec{x}) = y \leftrightarrow y \leq z \wedge g(y, \vec{x}) = 1 \wedge \forall y_1 < y\, g(y_1, \vec{x}) \neq 1 \vee \\ \forall y \leq z\, g(y, \vec{x}) \neq 1 \wedge y = 0. \end{aligned} \tag{1}$$

The function f defined by (1) is such that $f(z, \vec{x}) = y$ where y is the smallest number $y \leq z$ such that $\phi[\vec{x}, y]$ holds. We have $f(z, \vec{x}) = 0$ if there is no such number.

In the sequel we abbreviate (1) to

$$f(z, \vec{x}) = \mu_{y \leq z}[g(y, \vec{x}) = 1].$$

2.2.9 Definitions by bounded minimalization. Definitions of functions by *bounded minimalization* are of a form

$$\begin{aligned} f(\vec{x}) = y \leftrightarrow y \leq \tau[\vec{x}] \wedge \phi[\vec{x}, y] \wedge \forall z < y \neg\phi[\vec{x}, z] \vee \\ \forall z \leq \tau[\vec{x}] \neg\phi[\vec{x}, z] \wedge y = 0, \end{aligned} \tag{1}$$

where $\tau[\vec{x}]$ is a term and $\phi[\vec{x}, y]$ is a bounded formula with at most the indicated variables free, both without any application of f. We require that the term τ is built up only from variables and constants by applications of functions. The function f defined by (1) is such that $f(\vec{x})$ is the smallest number $y \leq \tau[\vec{x}]$ such that $\phi[\vec{x}, y]$ holds. We have $f(\vec{x}) = 0$ if there is no such number.

In the sequel we abbreviate (1) to

$$f(\vec{x}) = \mu_{y \leq \tau[\vec{x}]}[\phi[\vec{x}, y]].$$

We permit also strict bounds in (1). That is, we allow definitions of a form

$$f(\vec{x}) = \mu_{y < \tau[\vec{x}]}[\phi[\vec{x}, y]]$$

as abbreviation for

$$f(\vec{x}) = \mu_{y \leq \tau[\vec{x}]}[y < \tau[\vec{x}] \wedge \phi[\vec{x}, y]].$$

2.2.10 Theorem. *A class $\mathcal{F}$ is closed under explicit definitions of partial functions iff $\mathcal{F}$ contains the identity functions I_i^n, the constant functions C_m, and it is closed under composition and conditional.*

Proof. We prove first the implication ($\rightarrow$). Suppose that $\mathcal{F}$ is closed under explicit definitions of partial functions. Then each identity function I_i^n has the following explicit definition in $\mathcal{F}$: $I_i^n(x_1, \ldots, x_i, \ldots, x_n) = x_i$. Similarly, each constant function C_m can be explicitly defined in $\mathcal{F}$ by $C_m(x) = m$. If the n-ary partial function f is defined by composition from the partial functions $h, g_1, \ldots, g_m$ in $\mathcal{F}$ then we define f in the class $\mathcal{F}$ by the explicit definition 2.2.1(1). Finally, if the n-ary partial function f is defined by the conditional operator from the partial functions g_1, g_2, g_3 in $\mathcal{F}$ then we define f in $\mathcal{F}$ by the explicit definition 2.2.2(1).

We prove the implication ($\leftarrow$) by assuming that $\mathcal{F}$ contains the identity functions I_i^n, the constant functions C_m and that it is closed under the composition and conditional of partial functions. We show that $\mathcal{F}$ is closed under explicit definitions $f(\vec{x}) \simeq \tau[\vec{x}]$ of n-ary functions f by induction on the structure of terms τ.

If $\tau \equiv x_i$ with $1 \leq i \leq n$ then f is the identity function $I_i^n \in \mathcal{F}$.

If $\tau \equiv m$ then f is the constant function C_m^n which can be defined in $\mathcal{F}$ by the composition $C_m^n(\vec{x}) = C_m\, I_1^n(\vec{x})$ of the unary constant function $C_m \in \mathcal{F}$ and the identity function $I_1^n \in \mathcal{F}$.

If $\tau \equiv h(\rho_1[\vec{x}], \ldots, \rho_m[\vec{x}])$, where $h \in \mathcal{F}$, then we explicitly define the n-ary partial functions $g_1(\vec{x}) \simeq \rho_1[\vec{x}], \ldots, g_m(\vec{x}) \simeq \rho_m[\vec{x}]$ and get $g_1, \ldots, g_m$ in $\mathcal{F}$ by IH. We now obtain $f \in \mathcal{F}$ from $h, g_1, \ldots, g_m$ by the composition 2.2.1(1).

Finally, if $\tau \equiv \mathcal{D}_s(\rho_1[\vec{x}], \rho_2[\vec{x}], \rho_3[\vec{x}])$ then we explicitly define the n-ary partial functions $g_1(\vec{x}) \simeq \rho_1[\vec{x}], g_2(\vec{x}) \simeq \rho_2[\vec{x}], g_3(\vec{x}) \simeq \rho_3[\vec{x}]$ and get g_1, g_2, g_3 in $\mathcal{F}$ by IH. We obtain $f \in \mathcal{F}$ from g_1, g_2, g_3 by the conditional 2.2.2(1). $\square$

2.2.11 Remark. In the sequel we will use the following special cases of Thm. 2.2.10 which can be proved by similar means:

(i) A class $\mathcal{F}$ of functions is closed under explicit definitions iff $\mathcal{F}$ contains the identity functions I_i^n, the constant functions C_m, the case discrimination function D and it is closed under composition.

2.2.12 Theorem. *Every class $\mathcal{F}$ closed under explicit definitions of functions, under bounded minimalization, and containing $\leq_*$ is closed under explicit definitions of predicates with bounded formulas.*

Proof. We show that $\mathcal{F}$ is closed under explicit definitions $R(\vec{x}) \leftrightarrow \phi[\vec{x}]$ of n-ary predicates by induction on the structure of formulas ϕ.

If ϕ is one of $\tau_1 \leq \tau_2$, $\tau_1 < \tau_2$, $\tau_1 \geq \tau_2$, $\tau_1 > \tau_2$, or $\tau_1 = \tau_2$ we define the characteristic function R_* of R in $\mathcal{F}$ by the corresponding explicit definition:

$$
\begin{aligned}
R_*(\vec{x}) &= \tau_1[\vec{x}] \leq_* \tau_2[\vec{x}] \\
R_*(\vec{x}) &= \neg_* \tau_2[\vec{x}] \leq_* \tau_1[\vec{x}] \\
R_*(\vec{x}) &= \tau_2[\vec{x}] \leq_* \tau_1[\vec{x}] \\
R_*(\vec{x}) &= \tau_2[\vec{x}] <_* \tau_1[\vec{x}] \\
R_*(\vec{x}) &= \tau_1[\vec{x}] \leq_* \tau_2[\vec{x}] \wedge_* \tau_2[\vec{x}] \leq_* \tau_1[\vec{x}].
\end{aligned}
$$

If $\phi \equiv Q(\vec{\rho})$ then, since $Q_* \in \mathcal{F}$, we define R_* in $\mathcal{F}$ by explicit definition: $R_*(\vec{x}) = Q_*(\vec{\rho}[\vec{x}])$.

If $\phi \equiv \neg\psi$ we use IH and define an auxiliary n-ary predicate $Q \in \mathcal{F}$ by explicit predicate definition: $Q(\vec{x}) \leftrightarrow \psi[\vec{x}]$. We then define R_* in $\mathcal{F}$ by explicit definition: $R_*(\vec{x}) = \neg_* Q_*(\vec{x})$.

If $\phi \equiv \phi_1 \wedge \phi_2$ we obtain in $\mathcal{F}$ two auxiliary n-ary predicates $Q_1(\vec{x}) \leftrightarrow \phi_1[\vec{x}]$ and $Q_2(\vec{x}) \leftrightarrow \phi_2[\vec{x}]$ by IH. We then define R_* in $\mathcal{F}$ by explicit definition: $R_*(\vec{x}) = Q_{1*}(\vec{x}) \wedge_* Q_{2*}(\vec{x})$.

If $\phi \equiv \exists y \leq \tau\, \psi$ we use IH and define an auxiliary $(n+1)$-ary predicate $Q \in \mathcal{F}$ by explicit predicate definition: $Q(y, \vec{x}) \leftrightarrow \psi[y, \vec{x}]$. Then we define an auxiliary *witnessing function* $f \in \mathcal{F}$ by bounded minimalization:

$$
f(z, \vec{x}) = \mu_{y \leq z}[Q_*(y, \vec{x}) = 1].
$$

The characteristic function R_* of the predicate R has then explicit definition in $\mathcal{F}$ by: $R_*(\vec{x}) = Q_*[f(\tau[\vec{x}], \vec{x}), \vec{x}]$.

If $\phi \equiv \forall y \leq \tau\, \psi$ we use IH and define an auxiliary $(n+1)$-ary predicate $Q \in \mathcal{F}$ by explicit predicate definition: $Q(y, \vec{x}) \leftrightarrow \psi[y, \vec{x}]$. Then we define an auxiliary *counterexample function* $f \in \mathcal{F}$ by bounded minimalization of the function $\neg_* Q_* \in \mathcal{F}$:

$$
f(z, \vec{x}) = \mu_{y \leq z}[\neg_* Q_*(y, \vec{x}) = 1].
$$

The characteristic function R_* of the predicate R has then explicit definition in $\mathcal{F}$ by: $R_*(\vec{x}) = Q_*[f(\tau[\vec{x}], \vec{x}), \vec{x}]$.

We leave the cases with the remaining propositional connectives and with strict bounded quantifiers to the reader. $\square$

2.2.13 Theorem. *Every class $\mathcal{F}$ closed under explicit definitions of functions, under bounded minimalization, and containing $\leq_*$ is closed under definitions of functions with bounded minimalization.*

Proof. Suppose that the n-ary function f is defined by the bounded minimalization

$$
f(\vec{x}) = \mu_{y \leq \tau[\vec{x}]}[\phi[\vec{x}, y]]
$$

from the functions and predicates from $\mathcal{F}$. We can explicitly define f in $\mathcal{F}$ by

$$
\begin{aligned}
R(y, \vec{x}) &\leftrightarrow \phi[\vec{x}, y] \\
g(z, \vec{x}) &= \mu_{y \leq z}[R_*(y, \vec{x}) = 1] \\
f(\vec{x}) &= g(\tau[\vec{x}], \vec{x})
\end{aligned}
$$

since the characteristic function R_* of R is in $\mathcal{F}$ by Thm. 2.2.12. $\square$

2.3 Recursive Definitions

2.3.1 Functional equations with non-strict identity. Let $\tau[f; \vec{x}]$ be a term in the n-variables $\vec{x}$ containing applications of function symbols $g_1, \ldots, g_k$, where $k \geq 0$, and at least one application of the n-ary function symbol f. We are interested in solving *function equations* of the form

$$f(\vec{x}) \simeq \tau[f; \vec{x}] \tag{1}$$

for all arguments $\vec{x}$. More precisely, if we assign k partial functions $g_1, \ldots, g_k$ to the corresponding function symbols then we say that a partial function f *solves* (1) if the property holds for all $\vec{x}$. The following theorem asserts that there is a solution of the function equation (1) which is minimal in certain sense.

2.3.2 Graphs of terms are monotone. Graphs of terms $\tau[f] \asymp y$ are *monotone in* f in the following sense:

$$f_1 \subseteq f_2 \wedge \tau[f_1] \asymp y \rightarrow \tau[f_2] \asymp y. \tag{1}$$

This is proved by assuming $f_1 \subseteq f_2$, taking any $\vec{x}$ and then proving

$$\forall y(\tau[f_1] \asymp y \rightarrow \tau[f_2] \asymp y)$$

by induction on the construction of τ. So taky any y, assume $\tau[f_1] \asymp y$, and continue by the case analysis of τ.

If τ is a constant or a variable then $\tau[f_2] \asymp y$ holds trivially.

If $\tau \equiv f(\rho_1, \ldots, \rho_n)$ then from the assumption $f_1(\rho_1[f_1], \ldots, \rho_n[f_1]) \asymp y$ we get that there are numbers $y_1, \ldots, y_n$ such that $\rho_i[f_1] \asymp y_i$ for all $1 \leq i \leq n$ and $f_1(y_1, \ldots, y_n) \asymp y$. Hence also $f_2(y_1, \ldots, y_n) \asymp y$ and from n IH's we get $\rho_i[f_2] \asymp y_i$ for all $1 \leq i \leq n$. But this means that $f_2(\rho_1[f_2], \ldots, \rho_n[f_2]) \asymp y$ holds. The case when $\tau \equiv g(\vec{\rho})$ with $g \not\equiv f$ has a similar proof.

Finally, if $\tau \equiv \mathcal{D}_s(\rho_1, \rho_2, \rho_3)$ then from $\mathcal{D}_s(\rho_1[f_1], \rho_2[f_1], \rho_3[f_1]) \asymp y$ we get $\rho_1[f_1] \asymp y_1$ for some y_1 and we have $\rho_1[f_2] \asymp y_1$ from IH. We now consider two cases. If $y_1 > 0$ then we have $\rho_2[f_1] \asymp y$ and we obtain $\rho_2[f_2] \asymp y$ from another IH. But then $\mathcal{D}_s(\rho_1[f_2], \rho_2[f_2], \rho_3[f_2]) \asymp y$ holds. The case $y_1 = 0$ is proved similarly.

2.3.3 Graphs of terms are continuous. Graphs of terms $\tau[f] \asymp y$ are *continuous in* f in the sense that for any chain of n-ary functions $f_0, f_1, \ldots$ we have

$$\tau[\bigcup_i f_i] \asymp y \rightarrow \exists i\, \tau[f_i] \asymp y. \tag{1}$$

Indeed, we take any $\vec{x}$ and prove $\forall y(1)$ by induction on the construction of the term τ. So take any y, assume $\tau[\bigcup_i f_i] \asymp y$, and continue by case analysis of τ.

If τ is a constant or a variable then $\tau[f_0] \asymp y$ holds trivially.

If $\tau \equiv f(\rho_1, \ldots, \rho_n)$ then we have for all $1 \leq j \leq n$ numbers y_j such that $\rho_j[\bigcup_i f_i] \asymp y_j$ and $(\bigcup_i f_i)(y_1, \ldots, y_n) \asymp y$ hold. From the last we have $f_{i_0}(y_1, \ldots, y_n) \asymp y$ for some i_0. From n IH's there are numbers i_j such that $\rho_j[f_{i_j}] \asymp y_i$ holds. For $i = \max(i_0, i_1, \ldots, i_n)$ we have $\rho_j[f_i] \asymp y_i$ as well as $f_i(y_1, \ldots, y_n) \asymp y$ by 2.3.2(1) and so we also have $\tau[f_i] \asymp y$. The case when $\tau \equiv g(\vec{\rho})$ with $g \not\equiv f$ has a similar proof.

If $\tau \equiv \mathcal{D}_s(\rho_1, \rho_2, \rho_3)$ then we have $\rho_1[\bigcup_i f_i] \asymp y_1$ for some y_1 and we obtain $\rho_1[f_j] \asymp y_1$ for some j from IH. We now consider two cases. If $y_1 > 0$ then we have $\rho_2[\bigcup_i f_i] \asymp y$ and we obtain $\rho_2[f_k] \asymp y$ for some k from another IH. For $i = \max(j, k)$ we have $\rho_1[f_i] \asymp y_1$ and $\rho_2[f_i] \asymp y$ by 2.3.2(1) and so $\mathcal{D}_s(\rho_1[f_i], \rho_2[f_i], \rho_3[f_i]) \asymp y$ holds. The case $y_1 = 0$ is proved similarly.

2.3.4 First recursion theorem (Kleene). *Every functional equation*

$$f(\vec{x}) \simeq \tau[f; \vec{x}]$$

in an n-ary function symbol f has a solution $\bigcup_i f_i$, where the partial functions f_i are explicitly defined to satisfy

$$f_0 = \emptyset^{(n)}$$
$$f_{i+1}(\vec{x}) \simeq \tau[f_i; \vec{x}]$$

form a chain. The solution is minimal in the sense that we have $\bigcup_i f_i \subseteq f$ for any solution f.

Proof. That the partial functions f_i form a chain, i.e. that $f_i \subseteq f_{i+1}$ holds, is proved by induction on i. The base case $f_0 = \emptyset^{(n)} \subseteq f_1$ holds trivially. In the inductive case we assume $f_{i+1}(\vec{x}) \asymp y$ and obtain $\tau[f_i; \vec{x}] \asymp y$ by the definition of f_{i+1}. We get $\tau[f_{i+1}; \vec{x}] \asymp y$ from IH by 2.3.2(1) and so we have $f_{i+2}(\vec{x}) \asymp y$ by the definition of f_{i+2}.

We now prove that the partial function $\bigcup_i f_i$ is a solution of the functional equation by showing

$$(\bigcup_i f_i)(\vec{x}) \asymp y \leftrightarrow \tau[\bigcup_i f_i; \vec{x}] \asymp y$$

In the direction $(\to)$ we assume $(\bigcup_i f_i)(\vec{x}) \asymp y$ and obtain $f_i(\vec{x}) \asymp y$ from 1.4.4(1) for some i. Because the partial functions f_i form a chain we may assume that i is positive and so we have $\tau[f_{i-1}; \vec{x}] \asymp y$ by the definition of f_i. We then obtain $\tau[\bigcup_i f_i; \vec{x}] \asymp y$ from 1.4.4(2) and 2.3.2(1). In the direction $(\leftarrow)$ we assume $\tau[\bigcup_i f_i; \vec{x}] \asymp y$ and from 2.3.3(1) we get $\tau[f_i; \vec{x}] \asymp y$ for some i and so $f_{i+1}(\vec{x})$ by the definition of f_{i+1}. We obtain $(\bigcup_i f_i)(\vec{x}) \asymp y$ by 1.4.4(2).

If f is any solution of the functional equation we prove first $f_i \subseteq f$ by induction on i. The base case $\emptyset^{(n)} \subseteq f$ holds trivially. In the inductive case we assume $f_{i+1}(\vec{x}) \asymp y$ and obtain $\tau[f_i; \vec{x}] \asymp y$ by the definition of f_{i+1}. Hence we have $\tau[f; \vec{x}] \asymp y$ by IH and 2.3.2(1). Thus $f(\vec{x}) \asymp y$ holds. We then obtain $\bigcup_i f_i \subseteq f$ from 1.4.4(3). □

2.3.5 Recursive definitions. Let the term τ of the functional equation

$$f(\vec{x}) \simeq \tau[f; \vec{x}] \tag{1}$$

be such that it applies the function symbols f and also $g_1, \ldots, g_k$. If the last symbols are interpreted by the corresponding partial functions $g_1, \ldots, g_k$ then the the functional equation (1) is a *recursive definition from the partial functions* $g_1, \ldots, g_k$. The definition *defines* the partial function f which is its minimal solution guaranteed to exist by Thm. 2.3.4.

Recursive definition (1) can be viewed as a function operator taking all partial functions $g_1, \ldots, g_k$ applied in τ and yielding the partial function f defined by it.

2.3.6 Remark. Note that if

$$f(\vec{x}) \simeq \tau[f; \vec{x}] \tag{1}$$

is a recursive definition from (total) functions defining a (total) function f, then the function f is the unique solution of the functional equation (1). We then have

$$f(\vec{x}) = \tau[f; \vec{x}]. \tag{2}$$

Note also that f is also the unique function satisfying the identity (2).

2.3.7 Effectivity of recursive definitions. We will show in the subsequent paragraphs that the recursive definitions are effective function operators. That is, we provide an effective process by which we can compute the function f defined by a recursive definition $f(\vec{x}) \simeq \tau[f; \vec{x}]$ from partial functions $g_1, \ldots, g_k$ provided we have algorithms for computing the partial functions $g_1, \ldots, g_k$. Our model of computability is based on reductions (simplifications) of a certain kind of terms - here called recursive terms. That the recursive definitions are effective follows from Thm. 2.3.14.

2.3.8 Recursive terms. A *language* $\mathcal{L}$ of recursive terms is given by a set *oracle function symbols*. All oracle function symbols of arity $n > 0$ will come from the sequence

$$g_0^n, g_1^n, \ldots, g_i^n, \ldots .$$

The set of *recursive terms* (R-terms for short) of the language $\mathcal{L}$ is the smallest set of terms satisfying

– the variables $x_1, x_2, \ldots$ and the constant 0 are R-terms,
– if τ is a R-term so are $(\tau + 1)$ and $(\tau \dot{-} 1)$,
– if τ_1, τ_2, and τ_3 are R-terms so is $\mathcal{D}_s(\tau_1, \tau_2, \tau_3)$,
– if $\vec{\tau}$ is a non-empty n-tuple of R-terms then $\mathfrak{f}_n(\vec{\tau})$ is a R-term,
– if τ is a R-term and $\vec{\rho}$ is a non-empty n-tuple of R-terms then $(\lambda_n.\tau)(\vec{\rho})$ is a R-term subject to restrictions below,
– if $\vec{\tau}$ is a non-empty n-tuple of R-terms and $g_i^n \in \mathcal{L}$ then $g_i^n(\vec{\tau})$ is a R-term.

Function symbols $\mathfrak{f}_n$ are called *n-ary recursors*. Function symbols $\lambda_n.\tau$ are called *n-ary defined functions*.

An occurrence of a variable x in a R-term τ is *free* if it does not occur in the term ρ of a defined function symbol $\lambda_n.\rho$ which is applied in τ. The occurrence is *bound* otherwise.

Likewise, an occurrence of a recursor $\mathfrak{f}_n$ is *free* if it does not occur in the term ρ of a defined function symbol $\lambda_n.\rho$ which is applied in τ. The occurrence is *bound* otherwise.

A defined function symbol $\lambda_n.\tau$ is legal if all free occurrences of variables in τ are of variables from among the n-tuple of variables $x_1, \ldots, x_n$ and if all free occurrences of recursors in τ are of the form $\mathfrak{f}_n$.

A R-term is *closed* if it does not have free occurrences of recursors or of any variable.

2.3.9 Standard interpretation of recursive terms. R-terms of a language $\mathcal{L}$ obtain the standard partial denotations once we interprete the oracle function symbols g_i^n of $\mathcal{L}$ as partial functions. This is done with the help of a mapping $\mathfrak{I}$, called the *interpretation of* $\mathcal{L}$, from oracles of $\mathcal{L}$ into partial functions, where we interprete the n-ary oracle function symbol $g_i^n \in \mathcal{L}$ by the n-ary partial function $\mathfrak{I}(g_i^n)$. We will usually abbreviate $\mathfrak{I}(g_i^n)$ by g_i^n and called the partial function g_i^n an *oracle*. An interpretation is called *total* if oracle function symbols are interpreted by (total) functions.

Let $\mathfrak{I}$ be a interpretation of the language $\mathcal{L}$. By induction on the R-terms τ of $\mathcal{L}$ we extend the interpretation $\mathfrak{I}$ to defined function symbols $\lambda_n.\tau$ as follows. Suppose that $\tau[\mathfrak{f}_n; \vec{x}]$ is a term such that $\lambda_n.\tau$ is a legal defined function symbol and it has all applied defined function symbols interpreted by IH and we interprete the function symbol $\lambda_n.\tau$ by the partial function f defined by the definition

$$f(\vec{x}) \simeq \tau[f; \vec{x}]$$

which is explicit if f is not applied in τ and recursive otherwise.

2.3.10 Notation for n-tuples of monadic numerals. We will need a notation for n-tuples of monadic numerals, which are closed R-terms, and we define for $n \geq 1$:

$$\underline{x_1, \ldots, x_n} \equiv S^{x_1}(0), \ldots, S^{x_n}(0).$$

Note that $\underline{x} \equiv S^x(0)$.

2.3.11 Reductions of closed recursive terms. For any interpretation $\mathfrak{I}$ of the language $\mathcal{L}$ we now describe a possibly infinite process by which we can *reduce* (compute, calculate, simplify) a closed R-term of $\mathcal{L}$ until we obtain a (monadic) numeral which cannot be further reduced.

If a closed R-term τ is not a numeral then it must contain at least one occurrence, called the *redex* (for *reducible expression*), of one of the closed R-terms listed below on the left hand side:

$$\begin{aligned}
(\underline{x} \dot{-} 1) &\rhd_1 \underline{x \dot{-} 1} \\
\mathcal{D}_s(\underline{0}, \tau_2, \tau_3) &\rhd_1 \tau_3 \\
\mathcal{D}_s(\underline{x+1}, \tau_2, \tau_3) &\rhd_1 \tau_2 \\
(\lambda_n.\tau[\mathfrak{f}_n; \vec{x}])(\underline{\vec{x}}) &\rhd_1 \tau[\lambda_n.\tau; \underline{\vec{x}}] \\
g_i^n(\underline{\vec{x}}) &\rhd_1 \underline{g_i^n(\vec{x})}.
\end{aligned}$$

One step reduction consists of locating the *leftmost* redex in τ and replacing it by its *contraction*, which is the closed term on the corresponding right-hand side. By the replacement we obtain again a closed R-term ρ. We note that the term ρ is uniquely determined by τ.

We say that τ_1 *reduces to* τ_2 *in* k *steps*, in symbols $\tau_1 \rhd_k \tau_2$, if there is a finite one step reduction sequence of length k of closed R-terms $\rho_0, \rho_1, \ldots, \rho_k$ such that $\rho_0 \equiv \tau_1$, $\rho_k \equiv \tau_2$, and for each $i < k$ we obtain the term ρ_{i+1} by one step reduction from the term ρ_i. We write $\tau_1 \rhd_{\leq k} \tau_2$ if $\tau_1 \rhd_n \tau_2$ for some $n \leq k$. We write $\tau_1 \rhd \tau_2$ if $\tau_1 \rhd_k \tau_2$ for some k.

It is not difficult to see that for every closed R-terms τ, ρ_1, and ρ_2 we have

$$\tau \rhd \rho_1 \wedge \tau \rhd \rho_2 \rightarrow \rho_1 \rhd \rho_2 \vee \rho_2 \rhd \rho_1 .$$

Since we have $\underline{x} \rhd \rho$ iff $\rho \equiv \underline{x}$, we can see that if τ reduces to a monadic numeral then the numeral is uniquely determined.

2.3.12 Effectivity of reductions. We can always effectively determine if a sequence of symbols is a closed R-term. We can also effectively recognize a monadic numeral. If a closed R-term is not a monadic numeral we can effectively locate in it its leftmost redex and produce a new closed term τ_1

by copying the term τ and replacing the redex by its contraction provided that the redex is not an application of a function from $\mathcal{F}$. We can effectively repeat one step reductions until we obtain a monadic numeral which can be reduced only to itself.

Thus if $\mathcal{L} = \emptyset$, the reductions are effective. If $\mathcal{L}$ is not empty then we can perform the reductions only *relatively* in a sense that we obtain the monadic numeral to the right of $\rhd_1$ of a redex-contraction pair

$$g_i^n(\underline{x_1, \ldots, x_n}) \rhd_1 \underline{g_i^n(x_1, \ldots, x_n)}$$

as an answer to a question to an *oracle* for g_i^n. There is no need to consider oracles as mystical objects. We can think of the above redex contraction pairs as elements of an infinite enumerable sequence ordered by the non-decreasing numbers $i + x_1 + \cdots + x_n$ in which the answer is looked up by searching. Note that the answer will be always found after finitely many steps because the functions g_i^n are total. However, the process of searching through the sequence is not effective because we cannot in general effectively present the sequence.

2.3.13 Denotational versus operational semantics. Function symbols $\lambda_n.\tau$ can be both used and mentioned. The function symbol $\lambda_n.\tau$ is used as a name denoting an n-ary function f over $\mathbb{N}$. The same symbol is mentioned as a *program* (rule) for the computation of the function f by reductions.

The next theorem asserts that such functions can be computed by reductions in exactly those points in which they are defined. This establishes the equivalence of so-called *denotational* (definitional) semantics with the *operational* (computational) semantics.

2.3.14 Theorem. *We have*

$$(\lambda_n.\tau)(\vec{x}) \asymp y \leftrightarrow (\lambda_n.\tau)(\underline{\vec{x}}) \rhd \underline{y}. \tag{1}$$

Proof. The $(\rightarrow)$-direction of (1) is proved by triple induction. The outer induction is on the structure of defined function symbols $\lambda_n.\tau$:

$$\forall\vec{x}\forall y((\lambda_n.\tau)(\vec{x}) \asymp y \rightarrow (\lambda_n.\tau)(\underline{\vec{x}}) \rhd \underline{y}). \tag{2}$$

So take any $\lambda_n.\tau[\mathfrak{f}_n;\vec{x}]$. Let $f_0 = \emptyset^{(n)}$ and $f_{i+1}(\vec{x}) \simeq \tau[f_i;\vec{x}]$. We prove by (the middle) induction on i:

$$\forall\vec{x}\forall y(f_i(\vec{x}) \asymp y \rightarrow (\lambda_n.\tau)(\underline{\vec{x}}) \rhd \underline{y}). \tag{3}$$

In the base case there is nothing to prove since f_0 is nowhere defined. In the inductive case we take any $\vec{x}$ and prove by (the inner) induction on the structure of subterms $\rho[\mathfrak{f}_n;\vec{x}]$ of τ:

$$\forall y(\rho[f_i;\vec{x}] \asymp y \rightarrow \rho[\lambda_n.\tau;\underline{\vec{x}}] \rhd \underline{y}). \tag{4}$$

We take any subterm ρ of τ, any y, assume $\rho[f_i;\vec{x}] \asymp y$ and continue by the case analysis of ρ.

If $\rho \equiv x_i$ then $y = x_i$ and thus

$$x_i[\lambda_n.\tau;\underline{\vec{x}}] \equiv \underline{x_i} \rhd_0 \underline{x_i}.$$

If $\rho \equiv 0$ then $y = 0$ and thus

$$0[\lambda_n.\tau;\underline{\vec{x}}] \equiv 0 \rhd_0 \underline{0}.$$

If $\rho \equiv \rho_1 + 1$ then $y = y_1 + 1$ for some y_1 such that $\rho_1[f_i;\vec{x}] \asymp y_1$. We have

$$(\rho_1 + 1)[\lambda_n.\tau;\underline{\vec{x}}] \equiv \rho_1[\lambda_n.\tau;\underline{\vec{x}}] + 1 \overset{\text{inner IH}}{\rhd} \underline{y_1} + 1 \equiv \underline{y_1 + 1}.$$

If $\rho \equiv \rho_1 \dot{-} 1$ then $y = y_1 \dot{-} 1$ for some y_1 such that $\rho_1[f_i;\vec{x}] \asymp y_1$. We have

$$(\rho_1 \dot{-} 1)[\lambda_n.\tau;\underline{\vec{x}}] \equiv \rho_1[\lambda_n.\tau;\underline{\vec{x}}] \dot{-} 1 \overset{\text{inner IH}}{\rhd} \underline{y_1} \dot{-} 1 \rhd_1 \underline{y_1 \dot{-} 1}.$$

If $\rho \equiv \mathcal{D}_s(\rho_1,\rho_2,\rho_3)$ then $\rho_1[f_i;\vec{x}] \asymp y_1$ for some y_1. We consider two subcases. If $y_1 \neq 0$ then $\rho_2[f_i;\vec{x}] \asymp y$ and we have

$$\mathcal{D}_s(\rho_1,\rho_2,\rho_3)[\lambda_n.\tau;\underline{\vec{x}}] \equiv \mathcal{D}_s(\rho_1[\lambda_n.\tau;\underline{\vec{x}}],\rho_2[\lambda_n.\tau;\underline{\vec{x}}],\rho_3[\lambda_n.\tau;\underline{\vec{x}}]) \overset{\text{inner IH}}{\rhd}$$
$$\mathcal{D}_s(\underline{y_1},\rho_2[\lambda_n.\tau;\underline{\vec{x}}],\rho_3[\lambda_n.\tau;\underline{\vec{x}}]) \rhd_1 \rho_2[\lambda_n.\tau;\underline{\vec{x}}] \overset{\text{inner IH}}{\rhd} \underline{y_1}.$$

The subcase when $x = 0$ is similar.

If $\rho \equiv \mathfrak{f}_n(\vec{\rho})$ then for all $j = 1,\dots,n$ there are numbers y_j such that $\rho_j[f_i;\vec{x}] \asymp y_j$ and $f_i(\vec{y}) \asymp y$. We have

$$\mathfrak{f}_n(\vec{\rho})[\lambda_n.\tau;\underline{\vec{x}}] \equiv (\lambda_n.\tau)(\vec{\rho}[\lambda_n.\tau;\underline{\vec{x}}]) \overset{\text{inner IH's}}{\rhd} (\lambda_n.\tau)(\underline{\vec{y}}) \overset{\text{middle IH}}{\rhd} \underline{y}.$$

If $\rho \equiv (\lambda_m.\sigma)(\vec{\rho})$ then for all $k = 1,\dots,m$ there are numbers y_k such that $\rho_k[f_i;\vec{x}] \asymp y_k$ and $(\lambda_m.\sigma)(\vec{y}) \asymp y$. We have

$$(\lambda_m.\sigma)(\vec{\rho})[\lambda_n.\tau;\underline{\vec{x}}] \equiv (\lambda_m.\sigma)(\vec{\rho}[\lambda_n.\tau;\underline{\vec{x}}]) \overset{\text{inner IH's}}{\rhd} (\lambda_m.\sigma)(\underline{\vec{y}}) \overset{\text{outer IH}}{\rhd} \underline{y}.$$

If $\rho \equiv g^m_j(\vec{\rho})$ then for all $k = 1,\dots,m$ there are numbers y_k such that $\rho_k[f_i;\vec{x}] \asymp y_k$ and $g^m_j(\vec{y}) \asymp y$. We have

$$g^m_j(\vec{\rho})[\lambda_n.\tau;\underline{\vec{x}}] \equiv g^m_j(\vec{\rho}[\lambda_n.\tau;\underline{\vec{x}}]) \overset{\text{inner IH's}}{\rhd} g^m_j(\underline{\vec{y}}) \rhd_1 \underline{g^m_j(\vec{y})} \equiv \underline{y}.$$

This ends the proof of the inner induction.

We now finish the inductive case of the middle induction by taking any y such that $f_{i+1}(\vec{x}) \asymp y$ holds. We thus have $\tau[f_i;\vec{x}] \asymp y$ and we obtain

$$(\lambda_n.\tau)(\underline{\vec{x}}) \rhd_1 \tau[\lambda_n.\tau;\underline{\vec{x}}] \overset{(4)}{\rhd} \underline{y}$$

by taking $\rho \equiv \tau$ in the inner induction formula.

We can now finish the proof of the outer induction (2). We take any $\vec{x}$ and y such that $(\lambda_n.\tau)(\vec{x}) \asymp y$ holds. Then $f_i(\vec{x}) \asymp y$ for some i from Thm. 2.3.4 and thus, by (3), we obtain $(\lambda_n.\tau)(\underline{\vec{x}}) \rhd \underline{y}$.

The ($\leftarrow$)-direction of (1) follows from the next generalised property:

for every closed R-term τ we have

$$\forall y(\tau \rhd_k \underline{y} \to \tau \asymp y), \tag{5}$$

which is proved by complete induction on k. We take any closed R-term τ and prove (5) by (the inner) induction on the structure of the term τ. So take any y such that $\tau \rhd_k \underline{y}$ holds and consider two cases. If the term τ is a numeral then $k = 0$ and thus $\tau \equiv \underline{y}$. Consequently, $\tau = y$. So suppose that τ is not a numeral. Then $k > 0$ and we continue by the case analysis on the closed R-term τ.

If $\tau \equiv \rho + 1$ then there is a y_1 such that $\rho \rhd_k \underline{y_1}$ and

$$(\rho + 1) \rhd_k (\underline{y_1} + 1) \equiv \underline{y_1 + 1} \equiv \underline{y}.$$

We have $\rho \asymp y_1$ by the inner IH and, since $y_1 + 1 = y$, we obtain $\rho + 1 \asymp y$.

If $\tau \equiv \rho \dot{-} 1$ then there is a y_1 such that $\rho \rhd_{k-1} \underline{y_1}$ and

$$(\rho \dot{-} 1) \rhd_{k-1} (\underline{y_1} \dot{-} 1) \rhd_1 \underline{y_1 \dot{-} 1} \equiv \underline{y}.$$

We have $\rho \asymp y_1$ by the outer IH and, since $y_1 \dot{-} 1 = y$, we obtain $\rho \dot{-} 1 \asymp y$.

If $\tau \equiv \mathcal{D}_s(\rho_1, \rho_2, \rho_3)$ then there is a $k_1 < k$ and a y_1 such that $\rho_1 \rhd_{k_1} \underline{y_1}$. By the outer IH we have $\rho_1 \asymp y_1$ and we consider two subcases. If $y_1 \neq 0$ then we have

$$\mathcal{D}_s(\rho_1, \rho_2, \rho_3) \rhd_{k_1} \mathcal{D}_s(\underline{y_1}, \rho_2, \rho_3) \rhd_1 \rho_2 \rhd_{k-k_1-1} \underline{y}.$$

We have $\rho_2 \asymp y$ by another outer IH and so $\mathcal{D}_s(\rho_1, \rho_2, \rho_3) \asymp y$. The subcase when $y_1 = 0$ is similar.

If $\tau \equiv (\lambda_n.\sigma)[\mathfrak{f}_n;\vec{x}](\vec{\rho})$ then for all $i = 1, \ldots, n$ there are numbers k_i and y_i such that $\rho_i \rhd_{k_i} \underline{y_i}$ and $\sum_i k_i < k$. We then have

$$(\lambda_n.\sigma)(\vec{\rho}) \rhd_{\sum_i k_i} (\lambda_n.\sigma)(\underline{\vec{y}}) \rhd_1 \sigma[\lambda_n.\sigma;\underline{\vec{y}}] \rhd_{k-1-\sum_i k_i} \underline{y}.$$

We have $\sigma[\lambda_n.\sigma;\underline{\vec{y}}] \asymp y$ by the outer IH and from $(\lambda_n.\sigma)(\underline{\vec{y}}) \simeq \sigma[\lambda_n.\sigma;\underline{\vec{y}}]$ we obtain $(\lambda_n.\sigma)(\underline{\vec{y}}) \asymp y$. We have $\rho_i \asymp y_i$ by outer IH's and so $(\lambda_n.\sigma)(\vec{\rho}) \asymp y$ holds.

If $\tau \equiv g_i^n(\vec{\rho})$ then for all $j = 1, \ldots, n$ there are numbers k_j and y_j such that $\rho_j \rhd_{k_j} \underline{y_j}$ and $\sum_j k_j < k$. We then have

$$g_i^n(\vec{\rho}) \rhd_{\sum_i k_i} g_i^n(\underline{\vec{y}}) \rhd_1 \underline{g_i^n(\vec{y})} \equiv \underline{y}$$

We have $\rho_j \asymp y_j$ by outer IH's and, since $g_i^n(\vec{y}) \asymp y$, we obtain $g_i^n(\vec{\rho}) \asymp y$. $\square$

2.3.15 Well-founded relations over $\mathbb{N}$. In this paragraph we discuss binary relations denoted by $\prec$ over the set of natural numbers. We use the infix notation and write $x \prec y$ as an abbreviation for $\prec(x, y)$, i.e. for $\langle x, y\rangle \in \prec$. We conventionally write $y \succ x$ as an abbreviation for $x \prec y$.

For a non-empty subset K of $\mathbb{N}$ we say that its element $m \in K$ is $\prec$-*minimal* if

$$\forall x(x \prec m \to x \notin K).$$

A binary relation $x \prec y$ is *well-founded* if every non-empty subset K of $\mathbb{N}$ has a $\prec$-minimal element.

We call a (finite or infinite) sequence $x_0, x_1, x_2, \ldots, x_n, \ldots$ of numbers $\prec$-descending if $x_{i+1} \prec x_i$ for all i, i.e.

$$x_0 \succ x_1 \succ x_2 \succ \cdots \succ x_n \succ \cdots .$$

A binary relation $x \prec y$ is *noetherian* if every $\prec$-descending sequence is finite.

Let $\phi[x]$ be a property of natural numbers. We say that the property is $\prec$*-progressive* if

$$\forall y(y \prec x \to \phi[y]) \to \phi[x]$$

holds for all numbers x.

For a binary relation $x \prec y$ the following conditions are equivalent:

(i) $\prec$ is a well-founded relation,
(ii) $\prec$ is a noetherian relation,
(iii) ($\prec$-induction principle): for every $\prec$-progressive property $\phi[x]$ we have $\forall x\, \phi[x]$.

The proof is standard and left to the reader.

Let $x \prec y$ be a well-founded relation, $\phi[\vec{x}]$ a property of natural numbers, and $\mu[\vec{x}]$ a total term. We then have

$$\forall \vec{x}\Big(\forall \vec{y}\big(\mu[\vec{y}] \prec \mu[\vec{x}] \to \phi[\vec{y}]\big) \to \phi[\vec{x}]\Big) \to \phi[\vec{x}]$$

for every $\vec{x}$ and and call it $\prec$*-induction on* $\vec{x}$ *with measure* $\mu[\vec{x}]$.

Let $x \prec y$ be a binary relation over $\mathbb{N}$. We say that it is an *order* of $\mathbb{N}$ if the following holds:

$$\begin{gathered} x \not\prec x \\ x \prec y \land y \prec z \to x \prec z \\ x \prec y \lor x = y \lor x \succ y. \end{gathered}$$

An order $x \prec y$ is a *well-order* if it is also a well-founded relation. For instance, the standard order $x < y$ of natural numbers is a well-order.

2.3.16 Functional equations. Let $\tau[f;\vec{x}]$ be a term in the n-variables $\vec{x}$ containing applications of function symbols $g_1,\ldots,g_k$, where $k \geq 0$, and at least one application of the n-ary function symbol f. We are interested in solving *functional equations* of the form

$$f(\vec{x}) = \tau[f;\vec{x}] \tag{1}$$

for all arguments $\vec{x}$. More precisely, if we assign k functions $g_1,\ldots,g_k$ to the corresponding function symbols then we say that a function f *solves* (1) if the property holds for all $\vec{x}$. If the functional equation (1) has a unique solution f then we say that the function f is *defined* by (1).

We are interested in functional equations defining functions obeying the following condition of effectivity. We say that the function defined by Par. 2.3.19 satisfies the *effectivness condition* if the function f is also a minimal solution of the functional equation

$$f(\vec{x}) \simeq \tau[f;\vec{x}]$$

with non-strict identity. In Par. 2.3.19 we describe sufficient and necessary conditions on functional equations to define functions satisfying the above effectiveness condition.

2.3.17 Bounded recursion. By a functional equation with *bounded recursion* we mean a functional equation of a form

$$f(\vec{x}) = \tau[\dot{\lambda}\vec{y}.\mathcal{D}_s(\mu[\vec{y}] \prec_* \mu[\vec{x}], f(\vec{y}), 0);\vec{x}], \tag{1}$$

where $\mu[\vec{x}]$ is a total term in $\vec{x}$ called *measure* and $\prec$ is a well-founded relation.

Note that every recursive application in (1) is surrounded by a *guard* guaranteeing the decrease of recursive arguments in the measure μ. This means that every recursive application $f(\vec{\rho})$ in $\tau[f;\vec{x}]$ is replaced (starting from the bottom) by the term

$$\mathcal{D}_s(\mu[\vec{\rho}] \prec_* \mu[\vec{x}], f(\vec{\rho}), 0).$$

In the sequel we will usually abbreviate the equation (1) into the form

$$f(\vec{x}) = \tau[[f]^{\mu,\prec}_{\vec{x}};\vec{x}]. \tag{2}$$

We will omit the qualifier $\prec$ from (2) when $\prec$ is the standard well-order $<$ of natural numbers.

The following theorem says that there is a unique function f satisfying the functional equation (2). We say that f is defined by $\prec$-*bounded recursion with measure* μ.

2.3.18 Theorem (Bounded recursion). *Every functional equation with bounded recursion has a unique solution.*

Proof. We claim that f defined by

$$f(\vec{x}) \simeq \tau[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}] \tag{1}$$

is a unique solution of the functional equation 2.3.17(2).

We first prove by $\prec$-induction on $\vec{x}$ with measure $\mu[\vec{x}]$ that $f(\vec{x})\downarrow$ holds. So take any $\vec{x}$. By induction on the structure of terms we prove an auxiliary property:

$$\text{if } \rho[f; \vec{x}] \text{ is a subterm of } \tau \text{ then } \rho[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}]\downarrow. \tag{2}$$

Take any subterm ρ of τ and continue by the case analysis on the structure of the term ρ. If $\rho \equiv \mathcal{D}_s(\rho_1[f;\vec{x}], \rho_2[f;\vec{x}], \rho_3[f;\vec{x}])$ then by the inner IH $\rho_1[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}] \asymp y_1$ for some y_1. We consider two subcases. If $y_1 \neq 0$ then by the inner IH again $\rho_2[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}] \asymp y_2$ for some y_2. We thus have

$$\mathcal{D}_s(\rho_1[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}], \rho_2[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}], \rho_3[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}]) \asymp y_2.$$

The subcase when $y_1 = 0$ is similar. If $\rho \equiv g(\vec{\rho}[f;\vec{x}])$, where g is a m-ary (total) function, then by the inner IH there are numbers $\vec{y}$ such that $\rho_i[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}] \asymp y_i$ for all $i = 1, \ldots, m$. We obtain

$$g(\vec{\rho}[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}]) \asymp g(\vec{y})$$

since g is total. If $\rho \equiv f(\vec{\rho}[f;\vec{x}])$ then by the inner IH there are numbers $\vec{y}$ $\rho_i[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}] \asymp y_i$ for all $i = 1, \ldots, n$. We consider two subcases. If $\mu[\vec{y}] \prec \mu[\vec{x}]$ then by the outer IH $f(\vec{\rho}[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}]) \asymp y$ for some y. We clearly have

$$\mathcal{D}_s(\mu[\vec{\rho}[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}]] \prec_* \mu[\vec{x}], f(\vec{\rho}[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}]), 0) \asymp y.$$

The subcase when $\mu[\vec{y}] \prec \mu[\vec{x}]$ does not hold is obvious. The remaining cases when $\rho \equiv x_i$ or $\rho \equiv n$ are straightforward and left to the reader.

With the auxiliary property proved we take $\rho \equiv \tau$ in (2) and obtain that $\tau[[f]^{\mu,\prec}_{\vec{x}}; \vec{x}]\downarrow$ holds. From (1) we thus get $f(\vec{x})\downarrow$.

We see that f is a (total) function and thus, by the discussion in Par. 2.3.6, the function f is the unique solution of the functional equation 2.3.17(2). $\square$

2.3.19 Regular recursion. We can drop the guards around the recursive applications in definitions with bounded recursion 2.3.17(2) if we restrict the recursive applications in τ to *regular* applications. For that we need the concepts of governing conditions and conditions of regularity.

To every term τ and an occurrence of a subterm ρ of τ we define by induction on $\|\tau\| - \|\rho\|$ the *governing conditions* Γ^τ_ρ *of* ρ *in* τ. If $\rho \equiv \tau$ then

$\Gamma^\tau_\rho \equiv \top$. If ρ is either a variable or a constant then $\Gamma^\tau_\rho \equiv \top$. If $\rho \equiv f(\rho_1, \ldots, \rho_n)$ then $\Gamma^\tau_{\rho_i} \equiv \Gamma^\tau_\rho$. If $\rho \equiv \mathcal{D}_s(\rho_1, \rho_2, \rho_2)$ then

$$\Gamma^\tau_{\rho_1} \equiv \Gamma^\tau_\rho, \quad \Gamma^\tau_{\rho_2} \equiv \Gamma^\tau_\rho \wedge \rho_1 \neq 0, \quad \Gamma^\tau_{\rho_3} \equiv \Gamma^\tau_\rho \wedge \rho_1 = 0.$$

Consider a functional equation of a form

$$f(\vec{x}) = \tau[f; \vec{x}]. \tag{1}$$

Assume further we are given a measure term $\mu[\vec{x}]$ in $\vec{x}$ and a well-founded relation $\prec$. We assign to each recursive occurrence $\rho \equiv f(\vec{\rho}[f;\vec{x}])$ in τ governed by $\Gamma^\tau_\rho[f;\vec{x}]$ in τ the following *condition of regularity*:

$$\Gamma^\tau_\rho[f;\vec{x}] \rightarrow \mu[\vec{\rho}[f;\vec{x}]] \prec \mu[\vec{x}]. \tag{2}$$

Regularity asserts the same as the informal phrase that the *recursion decreases in the measure μ in the well-founded relation $\prec$.*

We say that the functional equation (1) is *regular* if the function g defined by bounded recursion:

$$g(\vec{x}) = \tau[[g]^{\mu,\prec}_{\vec{x}}; \vec{x}]$$

satisfies its conditions of regularity (2), i.e. the following holds

$$\Gamma^\tau_\rho[g;\vec{x}] \rightarrow \mu[\vec{\rho}[g;\vec{x}]] \prec \mu[\vec{x}]$$

for each condition of regularity (2) of the functional equation (1).

2.3.20 Theorem (Regular recursion). *Every functional equation with regular recursion has a unique solution satisfying its condition of effectivity.*

Proof. Suppose that the function g defined by bounded recursion:

$$g(\vec{x}) = \tau[[g]^{\mu,\prec}_{\vec{x}}; \vec{x}] \tag{1}$$

satisfies the conditions of regularity w.r.t. μ and $\prec$ of the functional equation

$$f(\vec{x}) = \tau[f; \vec{x}]. \tag{2}$$

We show that g is the unique solution of (2) and the minimal solution of the functional equation

$$f(\vec{x}) \simeq \tau[f; \vec{x}] \tag{3}$$

with non-strict identity.

We first prove that g is a solution of (2), i.e. we have

$$g(\vec{x}) = \tau[g; \vec{x}]. \tag{4}$$

By induction on the structure of subterms $\rho[f;\vec{x}]$ of τ we prove an auxiliary property:

$$\Gamma^\tau_\rho[g;\vec{x}] \to \rho[[g]^{\mu,\prec}_{\vec{x}};\vec{x}] = \rho[g;\vec{x}], \tag{5}$$

where $\Gamma^\tau_\rho[f;\vec{x}]$ governs ρ in τ. So assume $\Gamma^\tau_\rho[g;\vec{x}]$ and continue by the case analysis of ρ. If $\rho \equiv f(\vec{\rho}[f;\vec{x}])$ then we obtain

$$\begin{aligned}&\mathcal{D}_s(\mu[\vec{\rho}[[g]^{\mu,\prec}_{\vec{x}};\vec{x}]] \prec_* \mu[\vec{x}], g(\vec{\rho}[[g]^{\mu,\prec}_{\vec{x}};\vec{x}]), 0) \stackrel{\text{IH}}{=} \\ &\quad = \mathcal{D}_s(\mu[\vec{\rho}[g;\vec{x}]] \prec_* \mu[\vec{x}], g(\vec{\rho}[g;\vec{x}]), 0) \stackrel{(*)}{=} g(\vec{\rho}[g;\vec{x}]).\end{aligned}$$

The step marked by $(*)$ follows from regularity since the terms $\vec{\rho}$ are governed by Γ^τ_ρ in τ. The remaining cases are straightforward and left to the reader. With the auxiliary property proved we obtain (4) from

$$g(\vec{x}) \stackrel{(1)}{=} \rho[[g]^{\mu,\prec}_{\vec{x}};\vec{x}] \stackrel{(5)}{=} \tau[g;\vec{x}]$$

by noting that $\Gamma^\tau_\tau \equiv \top$.

We now show that the function g is the minimal solution of (3). So take any solution f of (3) We prove by $\prec$-induction on the measure $\mu[\vec{x}]$ of $\vec{x}$ that $f(\vec{x}) \simeq g(\vec{x})$ holds. Take any $\vec{x}$. By induction on the structure of subterms ρ of τ we prove an auxiliary property:

$$\Gamma^\tau_\rho[g;\vec{x}] \to \rho[f;\vec{x}] \simeq \rho[g;\vec{x}] \tag{6}$$

where $\Gamma^\tau_\rho[f;\vec{x}]$ governs ρ in τ. So assume $\Gamma^\tau_\rho[g;\vec{x}]$ and continue by the case analysis of ρ. If $\rho \equiv f(\vec{\rho}[f;\vec{x}])$ then $\Gamma^\tau_{\rho_i}[g;\vec{x}]$ holds and thus by the inner IH $\rho_i[f;\vec{x}] \simeq \rho_i[g;\vec{x}]$ for every $i = 1, \ldots, n$. We obtain from regularity that $\mu[\vec{\rho}[g;\vec{x}]] \prec \mu[\vec{x}]$ and so $f(\vec{\rho}[g;\vec{x}]) \simeq g(\vec{\rho}[g;\vec{x}])$ by the outer IH. Consequently

$$f(\vec{\rho}[f;\vec{x}]) \simeq g(\vec{\rho}[g;\vec{x}]).$$

The remaining cases are straightforward and left to the reader. With the auxiliary property proved we obtain the inductive case of the outer induction from

$$f(\vec{x}) \stackrel{(3)}{\simeq} \tau[f;\vec{x}] \stackrel{(6)}{\simeq} \tau[g;\vec{x}] \stackrel{(4)}{=} g(\vec{x})$$

by noting that $\Gamma^\tau_\tau \equiv \top$.

Since g is the minimal solution of (3) then it is the unique solution of (2) (see also Par. 2.3.6). □

2.3.21 Regular recursive definitions. Let the term $\tau[f;\vec{x}]$ of the functional equation with regular recursion

$$f(\vec{x}) = \tau[f;\vec{x}]. \tag{1}$$

be such that they applies the function symbols f and also $g_1, \dots, g_k$. If the last symbols are interpreted by the corresponding functions $g_1, \dots, g_k$ then the functional equation (1) is a *regular recursive definition from the functions* $g_1, \dots, g_k$. The definition *defines* the function f which is its unique solution guaranteed to exist by Thm. 2.3.20.

Recursive definition (1) can be viewed as a function operator taking all functions $g_1, \dots, g_k$ applied in τ and yielding the function f defined by it. Note that we do not count into the arguments of the function operator functions occurring in the measure term $\mu[\vec{x}]$.

Our measure terms $\mu[\vec{x}]$ are almost always into the well-order $<$ of natural numbers. In such cases we say that the definition is *regular in the i-th argument* if $\mu[\vec{x}] \equiv x_i$.

2.3.22 Theorem. *Every functional equation defining a function satisfying its condition of effectivity is a regular recursive definition of the same function.*

Proof. Suppose that the functional equation

$$f(\vec{x}) = \tau[f; \vec{x}] \tag{1}$$

has a unique solution f and the n-ary function f is also the minimal solution of the functional equation

$$f(\vec{x}) \simeq \tau[f; \vec{x}] \tag{2}$$

with non-strict identity.

Let $f_0 = \emptyset^{(n)}$ and $f_{i+1}(\vec{x}) \simeq \tau[f_i; \vec{x}]$. We claim that (1) is a recursive definition of f regular in the measure $m(\vec{x})$, where

$$m(\vec{x}) \text{ is the minimal number } i \text{ such that } f_{i+1}(\vec{x})\downarrow,$$

which is into the standard well-order $<$ of natural numbers. Note that

$$m(\vec{x}) < i \leftrightarrow f_i(\vec{x}) \asymp f(\vec{x}) \tag{3}$$
$$m(\vec{x}) \geq i \leftrightarrow f_i(\vec{x})\uparrow \tag{4}$$

since the partial functions f_i form a chain and $f = \bigcup_i f_i$ by Thm. 2.3.4.

We first prove for every subterm $\rho[f; \vec{x}]$ of τ that

$$\forall y(\rho[f_{m(\vec{x})}; \vec{x}] \asymp y \rightarrow \rho[f; \vec{x}] \asymp y) \tag{5}$$

by induction with measure $m(\vec{x})$.

Now we prove for every subterm $\rho[f; \vec{x}]$ of τ that

$$\Gamma^\tau_\rho[f; \vec{x}] \rightarrow \rho[f_{m(\vec{x})}; \vec{x}] \simeq \rho[f; \vec{x}], \tag{6}$$

where Γ^τ_ρ governs ρ in τ. Property (6) is proved by induction on $\|\tau\| - \|\rho\|$.

Next we prove for every recursive application $f(\vec{\rho}[f;\vec{x}])$ in τ that

$$\Gamma^{\tau}_{f(\vec{\rho})}[f;\vec{x}] \to m(\vec{\rho}[f;\vec{x}]) < m(\vec{x}), \tag{7}$$

where $\Gamma^{\tau}_{f(\vec{\rho})}$ governs $f(\vec{\rho})$ in τ.

Finally we prove

$$f(\vec{x}) = \tau[[f]^{m(\vec{x})}_{\vec{x}};\vec{x}]. \tag{8}$$

This concludes the proof of the theorem. □

2.3.23 Functions estimating the length of reductions. For every R-function symbol $\lambda_n.\tau$ we define the estimating function $d_{\lambda_n.\tau}(z)$ with the help of auxiliary terms δ_ρ estimating the length of reductions needed to reduce the subterms ρ of τ to monadic numerals. The auxiliary terms are defined by recursion on the construction of ρ:

$$\begin{aligned}
\delta_{x_i} &\equiv 0\\
\delta_0 &\equiv 0\\
\delta_{(\rho+1)} &\equiv \delta_\rho\\
\delta_{(\rho \dot{-} 1)} &\equiv \delta_\rho + 1\\
\delta_{\mathcal{D}_s(\rho_1,\rho_2,\rho_3)} &\equiv \delta_{\rho_1} + \delta_{\rho_2} + \delta_{\rho_3} + 1\\
\delta_{\mathfrak{f}_n(\rho_1,\dots,\rho_n)} &\equiv \delta_{\rho_1} + \ldots + \delta_{\rho_n} + f(z)\\
\delta_{(\lambda_n.\tau)(\rho_1,\dots,\rho_n)} &\equiv \delta_{\rho_1} + \cdots + \delta_{\rho_n} + d_{\lambda_n.\tau}(z)\\
\delta_{g^m_i(\rho_1,\dots,\rho_m)} &\equiv \delta_{\rho_1} + \cdots + \delta_{\rho_m} + 1.
\end{aligned}$$

We now define the unary estimating functions $d_{\lambda_n.\tau}$ by induction on the construction of R-terms τ. The estimating functions for all R-function symbols applied in τ are defined by IH. We note that f is applied in τ iff it is also applied in the term $\delta_\tau[f;z]$ as an unary function symbol. We define the estimating function by

$$\begin{aligned}
d_{\lambda_n.\tau}(0) &= 0\\
d_{\lambda_n.\tau}(z+1) &= \delta_\tau[d_{\lambda_n.\tau};z] + 1.
\end{aligned}$$

2.3.24 Theorem. *Suppose that $(\lambda_n.\tau)(\vec{x}) = \tau[\lambda_n.\tau;\vec{x}]$ is a regular recursive definition with a measure $\mu[\vec{x}]$ into the well-order $<$ of natural numbers. Assume further that the term τ does not apply any defined function symbols. Then we have*

$$(\lambda_n.\tau)(\vec{x}) = y \leftrightarrow (\lambda_n.\tau)(\underline{\vec{x}}) \rhd_{\leq d_{\lambda_n.\tau}(\mu[\vec{x}]+1)} \underline{y}. \tag{1}$$

Proof. Let $f_0 = \emptyset^{(n)}$ and $f_{i+1}(\vec{x}) \simeq \tau[f_i;\vec{x}]$. We prove by induction on i:

$$\forall\vec{x}\forall y(f_i(\vec{x}) \asymp y \to (\lambda_n.\tau)(\underline{\vec{x}}) \rhd_{\leq d_{\lambda_n.\tau}(i)} \underline{y}). \tag{2}$$

In the base case there is nothing to prove since f_0 is nowhere defined. In the inductive case we take any $\vec{x}$ and prove by (the inner) induction on the structure of subterms $\rho[\mathfrak{f}_n;\vec{x}]$ of τ:

$$\forall y(\rho[f_i;\vec{x}] \asymp y \rightarrow \rho[\lambda_n.\tau;\underline{\vec{x}}] \rhd_{\leq \delta_\rho[\lambda_n.\tau;i]} \underline{y}). \tag{3}$$

We take any subterm ρ of τ, any y, assume $\rho[f_i;\vec{x}] \asymp y$ and continue by the case analysis of ρ.

If $\rho \equiv x_i$ then $y = x_i$ and thus

$$x_i[\lambda_n.\tau;\underline{\vec{x}}] \equiv \underline{x_i} \rhd_0 \underline{x_i},$$

where $\delta_{x_i}[d_{\lambda_n.\tau};i] = 0$.

If $\rho \equiv 0$ then $y = 0$ and thus

$$0[\lambda_n.\tau;\underline{\vec{x}}] \equiv 0 \rhd_0 \underline{0},$$

where $\delta_0[d_{\lambda_n.\tau};i] = 0$.

If $\rho \equiv \rho_1 + 1$ then $y = y_1 + 1$ for some y_1 such that $\rho_1[f_i;\vec{x}] \asymp y_1$. We have

$$(\rho_1 + 1)[\lambda_n.\tau;\underline{\vec{x}}] \equiv \rho_1[\lambda_n.\tau;\underline{\vec{x}}] + 1 \overset{\text{inner IH}}{\rhd_{\leq \delta_{\rho_1}[d_{\lambda_n.\tau};i]}} \underline{y_1} + 1 \equiv \underline{y_1 + 1},$$

where $\delta_{(\rho_1+1)}[d_{\lambda_n.\tau};i] = \delta_{\rho_1}[d_{\lambda_n.\tau};i]$.

If $\rho \equiv \rho_1 \dot{-} 1$ then $y = y_1 \dot{-} 1$ for some y_1 such that $\rho_1[f_i;\vec{x}] \asymp y_1$. We have

$$(\rho_1 \dot{-} 1)[\lambda_n.\tau;\underline{\vec{x}}] \equiv \rho_1[\lambda_n.\tau;\underline{\vec{x}}] \dot{-} 1 \overset{\text{inner IH}}{\rhd_{\leq \delta_{\rho_1}[d_{\lambda_n.\tau};i]}} \underline{y_1} \dot{-} 1 \rhd_1 \underline{y_1 \dot{-} 1},$$

where $\delta_{(\rho_1 \dot{-} 1)}[d_{\lambda_n.\tau};i] = \delta_{\rho_1}[d_{\lambda_n.\tau};i] + 1$.

If $\rho \equiv \mathcal{D}_s(\rho_1,\rho_2,\rho_3)$ then $\rho_1[f_i;\vec{x}] \asymp y_1$ for some y_1. We consider two subcases. If $y_1 \neq 0$ then $\rho_2[f_i;\vec{x}] \asymp y$ and we have

$$\mathcal{D}_s(\rho_1,\rho_2,\rho_3)[\lambda_n.\tau;\underline{\vec{x}}] \equiv \mathcal{D}_s(\rho_1[\lambda_n.\tau;\underline{\vec{x}}],\rho_2[\lambda_n.\tau;\underline{\vec{x}}],\rho_3[\lambda_n.\tau;\underline{\vec{x}}]) \overset{\text{inner IH}}{\rhd_{\leq \delta_{\rho_1}[d_{\lambda_n.\tau};i]}}$$
$$\mathcal{D}_s(\underline{y_1},\rho_2[\lambda_n.\tau;\underline{\vec{x}}],\rho_3[\lambda_n.\tau;\underline{\vec{x}}]) \rhd_1 \rho_2[\lambda_n.\tau;\underline{\vec{x}}] \overset{\text{inner IH}}{\rhd_{\leq \delta_{\rho_2}[d_{\lambda_n.\tau};i]}} \underline{y_1},$$

where $\delta_{\mathcal{D}_s(\rho_1,\rho_2,\rho_3)}[d_{\lambda_n.\tau};i] \geq \delta_{\rho_1}[d_{\lambda_n.\tau};i] + 1 + \delta_{\rho_2}[d_{\lambda_n.\tau};i]$. The subcase when $x = 0$ is similar.

If $\rho \equiv \mathfrak{f}_n(\rho_1,\dots,\rho_n)$ then there are numbers $y_1,\dots,y_n$ such that $\rho_j[f_i;\vec{x}] \asymp y_j$ for $j = 1,\dots,n$ and $f_i(y_1,\dots,y_n) \asymp y$. We have

$$\mathfrak{f}_n(\rho_1,\dots,\rho_n)[\lambda_n.\tau;\underline{\vec{x}}] \equiv (\lambda_n.\tau)(\rho_1[\lambda_n.\tau;\underline{\vec{x}}],\dots,\rho_n[\lambda_n.\tau;\underline{\vec{x}}]) \overset{\text{inner IH's}}{\rhd_{\leq \sum \delta_{\rho_j}[d_{\lambda_n.\tau};i]}}$$
$$(\lambda_n.\tau)(\underline{y_1},\dots,\underline{y_n}) \overset{\text{outer IH}}{\rhd_{\leq d_{\lambda_n.\tau}(i)}} \underline{y},$$

where $\delta_{\mathfrak{f}_n(\rho_1,\dots,\rho_n)}[d_{\lambda_n.\tau};i] = \sum \delta_{\rho_j}[d_{\lambda_n.\tau};i] + d_{\lambda_n.\tau}(i)$.

If $\rho \equiv g_j^m(\rho_1, \dots, \rho_m)$ then there are numbers $y_1, \dots, y_m$ such that $\rho_k[f_i; \vec{x}] \asymp y_k$ for $k = 1, \dots, m$ and $g_j^m(y_1, \dots, y_m) \asymp y$. We have

$$g_j^n(\rho_1, \dots, \rho_m)[\lambda_n.\tau; \underline{\vec{x}}] \equiv g_j^n(\rho_1[\lambda_n.\tau; \underline{\vec{x}}], \dots, \rho_m[\lambda_n.\tau; \underline{\vec{x}}]) \overset{\text{inner IH's}}{\rhd_{\leq \sum \delta_{\rho_k}[d_{\lambda_n.\tau}; i]}}$$
$$g_j^n(\underline{y_1}, \dots, \underline{y_m}) \rhd_1 \underline{g_j^m(y_1, \dots, y_m)} \equiv \underline{y}.$$

This ends the proof of the inner induction.

We now finish the inductive case of the outer induction by taking any y such that $f_{i+1}(\vec{x}) \asymp y$ holds. We thus have $\tau[f_i; \vec{x}] \asymp y$ and we obtain

$$(\lambda_n.\tau[\mathfrak{f}_n; \vec{x}])(\underline{\vec{x}}) \rhd_1 \tau[\lambda_n.\tau; \underline{\vec{x}}] \overset{(2)}{\rhd_{\leq \delta_\tau[\lambda_n.\tau; i]}} \underline{y},$$

where $d_{\lambda_n.\tau}(i+1) = 1 + \delta_\tau[\lambda_n.\tau; i]$.

The $(\leftarrow)$-direction of (1) follows from Thm. 2.3.14. $\square$

2.4 Clausal Definitions

2.4.1 Introduction. We extend the language of terms with new constructs - generalized conditionals. Some of them by being *variable binding* constructs are well-known from LISP and from functional languages. Generalized conditionals will not be used for the presentation purposes. We will use clausal definitions instead (see Par. 2.4.25). Definitions in clausal form allow *pattern matching* style of definitions which is very readable. Clausal definitions are presented to humans in the form of clauses which are formulas derived from generalized regular recursive definitions $f(\vec{x}) = \alpha[f; \vec{x}]$ of functions (see Par. 2.4.20) by eliminating generalized conditionals in α via first-order constructs.

2.4.2 Patterns. Consider a formula

$$\phi[x_1, \dots, x_n; y_1, \dots, y_m] \tag{1}$$

with all its free variables among the indicated. The variables $\vec{x} \equiv x_1, \dots, x_n$ and $\vec{y} \equiv y_1, \dots, y_m$ are called respectively the *input* and the *output* variables of (1). The formula is called a *pattern* if it satisfies the following *pattern's uniqueness condition*:

$$\phi[\vec{x}; y_1, \dots, y_m] \land \phi[\vec{x}; z_1, \dots, z_m] \to y_1 = z_1 \land \cdots y_m = z_m. \tag{2}$$

A term $\rho[\vec{x}]$ in $\vec{x}$ is called a *recognizer* of the pattern (1) if we have

$$\rho[\vec{x}] = 0 \lor \rho[\vec{x}] = 1 \tag{3}$$
$$\exists \vec{y}\, \phi[\vec{x}; \vec{y}] \leftrightarrow \rho[\vec{x}] = 1. \tag{4}$$

Terms $\delta_1[\vec{x}], \ldots, \delta_m[\vec{x}]$ in $\vec{x}$ are called *destructors* of the pattern (1) if have

$$\exists y_1 \ldots \exists y_m \, \phi[\vec{x}; y_1, \ldots, y_m] \leftrightarrow \phi[\vec{x}; \delta_1[\vec{x}], \ldots, \delta_m[\vec{x}]]. \tag{5}$$

We require that the recognizers and the destructors of patterns are are built up only from variables and constants by applications of (total) functions.

We give here examples of typical patterns we will use later. We will indicate only the output variables of patterns.

2.4.3 Monadic patterns. *Monadic patterns* are formulas of a form

$$\tau = y + 1 \tag{1}$$

with the output variable y. The pattern's uniqueness condition

$$\tau = y + 1 \land \tau = z + 1 \rightarrow y = z$$

holds trivially. Note that we have

$$\exists y \, \tau = y + 1 \leftrightarrow \tau \neq 0$$

and thus $\rho \equiv (\tau \neq_* 0)$ is the recognizer of the pattern (1). Note also that

$$\exists y \, \tau = y + 1 \leftrightarrow \tau = \tau \dot{-} 1 + 1$$

and so $\delta \equiv (\tau \dot{-} 1)$ is the destructor of the pattern (1).

2.4.4 Pair patterns. *Pair patterns* are formulas of a form

$$\tau = y_1, y_2 \tag{1}$$

with the output variables y_1 and y_2. The pattern's uniqueness condition

$$\tau = y_1, y_2 \land \tau = z_1, z_2 \rightarrow y_1 = z_1 \land y_2 = z_2$$

follows from the pairing property 1.3.2(1) of the pairing function. Note that we have

$$\exists y_1 \exists y_2 \, \tau = y_1, y_2 \leftrightarrow \tau \neq 0$$

and thus $\rho \equiv (\tau \neq_* 0)$ is the recognizer of the pattern (1). Note also that

$$\exists y_1 \exists y_2 \, \tau = y_1, y_2 \leftrightarrow \tau = H(\tau), T(\tau)$$

and so $\delta_1 \equiv H(\tau)$ and $\delta_2 \equiv T(\tau)$ are the destructors of the pattern (1).

2.4.5 Pair constructors patterns. A *pair constructor* is either a constant $K_c = c, 0$ or an n-ary function $K_c(\vec{y}) = c, (\vec{y})$ where c is a constant called the *tag* of the constructor.

Constant pair constructor patterns are formulas of a form

$$\tau = K_c \tag{1}$$

with empty output variables, where K_c is a constant pair constructor. Note that we have

$$\tau = K_c \leftrightarrow \tau \neq 0 \land H(\tau) = c \land T(\tau) = 0$$

and thus $\rho \equiv (x \neq_* 0 \land_* H(\tau) =_* c \land_* T(\tau) =_* 0)$ is the recognizer of the pattern (1).

Functional pair constructor patterns are formulas of a form

$$\tau = K_c(\vec{y}), \tag{2}$$

with the output variables $\vec{y}$, where $K_c(\vec{y})$ is an n-ary functional pair constructor. Note that we have

$$\exists \vec{y}\, \tau = K_c(\vec{y}) \leftrightarrow H(\tau) = c \land T^{n-1}(\tau) \neq 0$$

and thus $\rho \equiv (H(\tau) =_* c \land_* T^{n-1}(\tau) \neq_* 0)$ is the recognizer of the pattern (2). Note also that

$$\exists \vec{y}\, \tau = K_c(\vec{y}) \leftrightarrow \tau = K_c(H\,T(\tau), \ldots, H\,T^{n-1}(\tau), T^n(\tau))$$

and so the terms $\vec{\delta} \equiv \delta_1, \ldots, \delta_n$ defined by

$$\delta_i \equiv \begin{cases} H\,T^{i-1}(\tau) & \text{if } 1 \leq i < n \\ T^n(\tau) & \text{if } i = n \end{cases}$$

are the destructors of the pattern (2).

2.4.6 Generalized terms. We now describe the syntax of generalized terms which, unlike the complex syntax which seems to be de rigueur in the definitions of programming languages, is in the usual simple style of logic. The reader should bear on mind that this syntax is not meant for presentation to humans. We use $\alpha, \beta, \ldots$, as syntactic variables ranging over generalized terms.

For a given n-tuple of variables $\vec{x}$ we define the set of *generalized terms in* $\vec{x}$ as the smallest set of expressions satisfying the following:

– Every term in $\vec{x}$ which is built up only from variables and constants by applications of (total) functions is a generalized term in $\vec{x}$.

– The expression of the form ($m \geq 1$):

$$\alpha[\vec{x}] \equiv \mathcal{D}^{\vec{\delta}_1,\ldots,\vec{\delta}_m}_{\rho_1,\ldots,\rho_m}(\phi_1[\vec{x};\vec{y}_1],\beta_1[\vec{x},\vec{y}_1],\ldots,\phi_m[\vec{x};\vec{y}_m],\beta_m[\vec{x},\vec{y}_m]), \qquad (1)$$

where $\phi_i[\vec{x};\vec{y}_i]$ are patterns with the recognizer ρ_i and the destructors $\vec{\delta}_i$, and $\beta_i[\vec{x},\vec{y}_i]$ are generalized terms in $\vec{x},\vec{y}_i$. Moreover, the patterns satisfies the following *disjointness* and *completeness* conditions respectively:

$$\bigwedge_{\substack{i,j=1\\ i\neq j}}^{m} \neg(\exists\vec{y}_i\,\phi_i[\vec{x};\vec{y}_i] \wedge \exists\vec{y}_j\,\phi_j[\vec{x};\vec{y}_j]) \qquad (2)$$

$$\bigvee_{i=1}^{m} \exists\vec{y}_i\,\phi_i[\vec{x};\vec{y}_i]. \qquad (3)$$

Generalized terms of a form (1) are called *generalized conditionals* or *case discrimination* terms. The variables $\vec{y}_i$ are said to be *bound* in the term (1).

We extend the notion of the graphs of terms for generalized conditionals as follows. The graph $\alpha \asymp v$ of the generalized conditional 2.4.6(1) is the formula of the form

$$\bigvee_{i=1}^{m} \exists\vec{y}_i\big(\phi_i[\vec{x};\vec{y}_i] \wedge \beta_i[\vec{x},\vec{y}_i] \asymp v\big). \qquad (4)$$

2.4.7 Notational conventions. In the sequel we usually omit recognizers and destructors from the notation of generalized conditionals. We will visualize the conditionals of a form 2.4.6(1) by the notation known from functional programming languages as

$$\begin{array}{l}\textbf{case}\\ \quad \phi_1[\vec{x};\vec{y}_1] \Rightarrow_{\vec{y}_1} \beta_1[\vec{x},\vec{y}_1]\\ \quad \vdots\\ \quad \phi_m[\vec{x};\vec{y}_m] \Rightarrow_{\vec{y}_m} \beta_m[\vec{x},\vec{y}_m]\\ \textbf{end}\end{array}$$

We will write the conditional 2.4.6(1) even as

$$\begin{array}{l}\textbf{case}\\ \quad \phi_1[\vec{x};\vec{y}_1] \Rightarrow_{\vec{y}_1} \beta_1[\vec{x},\vec{y}_1]\\ \quad \vdots\\ \quad \phi_{m-1}[\vec{x};\vec{y}_{m-1}] \Rightarrow_{\vec{y}_{m-1}} \beta_{m-1}[\vec{x},\vec{y}_{m-1}]\\ \quad \textbf{otherwise} \Rightarrow \beta_m[\vec{x}]\\ \textbf{end}\end{array}$$

when the last pattern ϕ_m is of the form $\bigwedge_{i=1}^{m-1} \neg\exists\vec{y}_i\phi_i$.

We give here examples of typical generalized conditionals we will use later. We will indicate only the output variables of patterns.

2.4.8 Equality tests. *Equality tests* are conditionals of a form

$$\mathcal{D}^{\emptyset,\emptyset}_{\rho_1,\rho_2}(\tau_1 = \tau_2, \beta_1, \tau_1 \neq \tau_2, \beta_2), \tag{1}$$

where $\rho_1 \equiv (\tau_1 =_* \tau_2)$ and $\rho_2 \equiv (\tau_1 \neq_* \tau_2)$. We visualize the conditional (1) as

$$\begin{array}{l} \textbf{case} \\ \quad \tau_1 = \tau_2 \Rightarrow \beta_1 \\ \quad \tau_1 \neq \tau_2 \Rightarrow \beta_2 \\ \textbf{end} \end{array}$$

2.4.9 Negation discrimination. *Negation discrimination* terms are conditionals of a form

$$\mathcal{D}^{\emptyset,\emptyset}_{\rho_1,\rho_2}(R(\vec{\tau}), \beta_1, \neg R(\vec{\tau}), \beta_2), \tag{1}$$

where $\rho_1 \equiv R_*(\vec{\tau})$ and $\rho_2 \equiv (\neg_* R_*(\vec{\tau}))$. We visualize the conditional (1) as

$$\begin{array}{l} \textbf{case} \\ \quad R(\vec{\tau}) \Rightarrow \beta_1 \\ \quad \neg R(\vec{\tau}) \Rightarrow \beta_2 \\ \textbf{end} \end{array}$$

2.4.10 Dichotomy discrimination. *Dichotomy discrimination* terms are conditionals of a form

$$\mathcal{D}^{\emptyset,\emptyset}_{\rho_1,\rho_2}(\tau_1 \leq \tau_2, \beta_1, \tau_1 > \tau_2, \beta_2), \tag{1}$$

where $\rho_1 \equiv (\tau_1 \leq_* \tau_2)$ and $\rho_2 \equiv (\tau_1 >_* \tau_2)$. We visualize the conditional (1) as

$$\begin{array}{l} \textbf{case} \\ \quad \tau_1 \leq \tau_2 \Rightarrow \beta_1 \\ \quad \tau_1 > \tau_2 \Rightarrow \beta_2 \\ \textbf{end} \end{array}$$

2.4.11 Trichotomy discrimination. *Trichotomy discrimination* terms are conditionals of a form

$$\mathcal{D}^{\emptyset,\emptyset,\emptyset}_{\rho_1,\rho_2,\rho_3}(\tau_1 < \tau_2, \beta_1, \tau_1 = \tau_2, \beta_2, \tau_1 > \tau_2, \beta_3), \tag{1}$$

where $\rho_1 \equiv (\tau_1 <_* \tau_2)$, $\rho_2 \equiv (\tau_1 =_* \tau_2)$, and $\rho_3 \equiv (\tau_1 >_* \tau_2)$. We visualize the conditional (1) as

$$\begin{array}{l} \textbf{case} \\ \quad \tau_1 < \tau_2 \Rightarrow \beta_1 \\ \quad \tau_1 = \tau_2 \Rightarrow \beta_2 \\ \quad \tau_1 > \tau_2 \Rightarrow \beta_3 \\ \textbf{end} \end{array}$$

2.4.12 Discrimination on constant patterns. *Discrimination on constants* are conditionals of a form

$$\mathcal{D}^{\emptyset,\dots,\emptyset,\emptyset}_{\rho_1,\dots,\rho_k,\rho_{k+1}}(\tau = c_1, \beta_1, \dots, \tau = c_k, \beta_k, \bigwedge_{i=1}^{k} \tau \neq c_i, \beta_{k+1}), \tag{1}$$

where $c_1, \dots, c_k$ are constants denoting pairwise different numbers, for each $1 \leq i \leq k$ we have $\rho_i \equiv (\tau =_* c_i)$, and $\rho_{k+1} \equiv \tau \neq_* c_1 \wedge_* \cdots \wedge_* \tau \neq_* c_k$. We visualize the conditional (1) as

$$\begin{array}{l} \textbf{case} \\ \quad \tau = c_1 \Rightarrow \beta_1 \\ \qquad \vdots \\ \quad \tau = c_k \Rightarrow \beta_k \\ \quad \textbf{otherwise} \Rightarrow \beta_{k+1} \\ \textbf{end} \end{array}$$

2.4.13 Discrimination on monadic patterns. *Monadic discrimination* terms are conditionals of a form

$$\mathcal{D}^{\emptyset,\delta_2}_{\rho_1,\rho_2}(\tau = 0, \beta_1, \tau = y + 1, \beta_2[y]), \tag{1}$$

where $\rho_1 \equiv (\tau =_* 0)$, $\rho_2 \equiv (\tau \neq_* 0)$, and $\delta_2 \equiv (\tau \dot{-} 1)$. We visualize the conditional (1) as

$$\begin{array}{l} \textbf{case} \\ \quad \tau = 0 \Rightarrow \beta_1 \\ \quad \tau = y + 1 \Rightarrow_y \beta_2[y] \\ \textbf{end} \end{array}$$

2.4.14 Discrimination on pair patterns. *Pair discrimination* terms are conditionals of a form

$$\mathcal{D}^{\emptyset,(\delta_2^1,\delta_2^2)}_{\rho_1,\rho_2}(\tau = 0, \beta_1, \tau = y_1, y_2, \beta_2[y_1, y_2]), \tag{1}$$

where $\rho_1 \equiv (\tau =_* 0)$, $\rho_2 \equiv (\tau \neq_* 0)$, $\delta_2^1 \equiv H(\tau)$, and $\delta_2^2 \equiv T(\tau)$. The disjointness condition 2.4.6(2) and the completeness condition 2.4.6(3) of the conditional (1) follow from the properties 1.3.2(4) and 1.3.2(3) of the pairing function respectively. We visualize the conditional (1) as

$$\begin{array}{l} \textbf{case} \\ \quad \tau = 0 \Rightarrow \beta_1 \\ \quad \tau = y_1, y_2 \Rightarrow_{y_1,y_2} \beta_2[y_1, y_2] \\ \textbf{end} \end{array}$$

2.4.15 Discrimination on pair constructors patterns. *Pair constructor discrimination* terms are conditionals of a form

$$\mathcal{D}^{\vec{\delta}_1,\ldots,\vec{\delta}_m,\emptyset}_{\rho_1,\ldots,\rho_m,\rho_{m+1}}\big(\tau = K_{c_1}(\vec{y}_1), \beta_1[\vec{y}_1], \ldots, \tau = K_{c_m}(\vec{y}_m), \beta_m[\vec{y}_m], \\ \bigwedge_{i=1}^{m} \neg\exists\vec{y}_i\, \tau = K_{c_i}(\vec{y}_i), \beta_{m+1}\big), \tag{1}$$

where $K_{c_i}(\vec{y}_i)$ are pair constructors with pairwise different tags c_i. The recognizer ρ_i and the destructors $\vec{\delta}_i$ of the pattern $\tau = K_{c_i}(\vec{y}_i)$ are defined as in Par. 2.4.5. The recognizer of the last pattern satisfies

$$\rho_{m+1} \equiv (\rho_1 =_* 0 \wedge_* \cdots \wedge_* \rho_m =_* 0).$$

The disjointness condition 2.4.6(2) of the conditional (1) follows from the pairing property 1.3.2(1) of the pairing function and its completeness condition 2.4.6(3) holds trivially. We visualize the conditional (1) as

$$\begin{array}{l}
\textbf{case} \\
\quad \tau = K_{c_1}(\vec{y}_1) \Rightarrow_{\vec{y}_1} \beta_1[\vec{y}_1] \\
\qquad \vdots \\
\quad \tau = K_{c_m}(\vec{y}_m) \Rightarrow_{\vec{y}_m} \beta_m[\vec{y}_m] \\
\quad \textbf{otherwise} \Rightarrow \beta_{m+1} \\
\textbf{end}
\end{array}$$

2.4.16 Assignments. *Assignments* are conditionals of a form

$$\mathcal{D}^{\delta}_{\rho}(\tau = y, \beta[y]), \tag{1}$$

where $\rho \equiv 1$ and $\delta \equiv \tau$. We visualize the conditional (1) as

$$\textbf{let } \tau = y \textbf{ in } \beta[y]$$

2.4.17 Translation of generalized terms. We now describe an effective process which for a given generalized term α finds an equivalent term $\alpha^\star$, i.e. we have

$$\alpha^\star = \alpha, \tag{1}$$

and without generalized conditionals. The idea is based on the fact that the generalized conditional 2.4.6(1) has the same denotation as the term

$$\overbrace{\mathcal{D}_s(\rho_1, \beta_1[\vec{x}, \vec{\delta}_1], \ldots \mathcal{D}_s}^{(m-1)\text{-times}}(\rho_{m-1}, \beta_{m-1}[\vec{x}, \vec{\delta}_{m-1}], \beta_m[\vec{x}, \vec{\delta}_m]) \ldots).$$

The mapping $\alpha^\star$ is defined inductively on the structure of generalized terms as follows. If α is without conditionals then $\alpha^\star \equiv \alpha$. Otherwise, we have

$$\mathcal{D}^{\vec{\delta}_1,\ldots,\vec{\delta}_m}_{\rho_1,\ldots,\rho_m}(\phi_1,\beta_1[\vec{y}_1],\ldots,\phi_m,\beta_m[\vec{y}_m])^\star \equiv$$
$$\equiv \overbrace{\mathcal{D}_s(\rho_1,\beta_1^\star[\vec{\delta}_1],\ldots\mathcal{D}_s}^{(m-1)\text{-times}}(\rho_{m-1},\beta_{m-1}^\star[\vec{\delta}_{m-1}],\beta_m^\star[\vec{\delta}_m])\ldots).$$

2.4.18 Generalized explicit definitions. Generalized explicit definitions of functions are of a form

$$f(\vec{x}) = \alpha[\vec{x}], \tag{1}$$

where the generalized term $\alpha[\vec{x}]$ contains at most the indicated variables free and does not apply f. Clearly there is a unique function f satisfying (1).

Generalized explicit definition (1) can be viewed as a function operator taking all functions applied in α and yielding the function f.

2.4.19 Theorem. *Every class $\mathcal{F}$ closed under explicit definitions of functions is closed under generalized explicit definitions.*

Proof. The theorem follows directly from the property 2.4.17(1) of the translation function $\alpha^\star$. □

2.4.20 Generalized regular recursive definitions. Suppose that the following equation

$$f(\vec{x}) = \alpha[f;\vec{x}] \tag{1}$$

is such that its translation

$$f(\vec{x}) = \alpha^\star[f;\vec{x}] \tag{2}$$

is a regular recursive definition of f. By 2.4.17(1) the function f is a unique function satisfying the equation (1) We call the equation (1) a *generalized regular recursive definition* of f.

Generalized regular recursive definition (1) can be viewed as a function operator taking all functions applied in α and yielding the function f. The following theorem is then straightforward.

2.4.21 Theorem. *Every class $\mathcal{F}$ closed under regular recursive definitions is closed under generalized regular recursive definitions.*

2.4.22 Remark. We can check directly whether the equation 2.4.20(1) is a generalized regular recursive definition with the help of conditions of regularity which are defined in the same way as the conditions of regularity of recursive definitions with ordinary terms. For that we need only to extend the concept of governing conditions onto generalized terms.

To every generalized term α and an occurrence of a subterm β of α we define by induction on $\|\alpha\| - \|\beta\|$ the *governing condition $\mathit{\Gamma}^\alpha_\beta$ of β in α.*

If $\beta \equiv \alpha$ then $\Gamma^{\alpha}_{\beta} \equiv \top$. Otherwise, if β is a term without conditionals then $\Gamma^{\alpha}_{\beta} \equiv \top$. Finally, if $\beta \equiv \mathcal{D}^{\vec{\delta}_1,\ldots,\vec{\delta}_m}_{\rho_1,\ldots,\rho_m}(\phi_1, \beta_1, \ldots, \phi_m, \beta_m)$ then

$$\Gamma^{\alpha}_{\rho_i} \equiv \Gamma^{\alpha}_{\delta^j_i} \equiv \Gamma^{\alpha}_{\beta}$$
$$\Gamma^{\alpha}_{\beta_i} \equiv \Gamma^{\alpha}_{\beta} \land \phi_i.$$

It is easy to see that for a given measure $\mu[\vec{x}]$ and a well-founded relation $\prec$ the conditions of regularity of the equations 2.4.20(1) and 2.4.20(2) are equivalent. Consequently, the equation 2.4.20(1) is a generalized regular recursive definition iff there is a measure $\mu[\vec{x}]$ and a well-founded relation $\prec$ such that the function g defined by bounded recursion:

$$g(\vec{x}) = \alpha^{\star}[[g]^{\mu,\prec}_{\vec{x}}; \vec{x}]$$

satisfies all its conditions of regularity, i.e. the following holds

$$\Gamma^{\alpha}_{\beta}[g; \vec{x}] \to \mu[\vec{\alpha}[g; \vec{x}]] \prec \mu[\vec{x}]$$

for every recursive occurrence $\beta \equiv f(\vec{\beta}[f; \vec{x}])$ in α governed in α by $\Gamma^{\alpha}_{\beta}[f; \vec{x}]$.

2.4.23 Clauses. *Clauses* are *Horn* formulas, i.e. implications with formulas in the consequent. Every clause can be presented in a form

$$\psi_1 \land \cdots \land \psi_k \to f(\vec{\rho}) = \alpha, \tag{1}$$

where α is a generalized term. Clauses used in definitions are in logic programming customarily written with converse implications:

$$f(\vec{\rho}) = \alpha \leftarrow \psi_1 \land \cdots \land \psi_k. \tag{2}$$

We adopt this custom and treat such a formula only as a notational variant of (1). The identity $f(\vec{\rho}) = \alpha$ is the *head* of the clause (2) and the conjunction on the right hand side constitutes the *body* of the clause. We do not exclude the case when $k = 0$ when the body of the clause is *empty* and then the clause is written as $f(\vec{\rho}) = \alpha$.

The clause (2) is *terminal* if it the term α does not contain conditionals, i.e. it is built up only from variables and constans by applications of functions. Otherwise, the clause is *non-terminal*.

2.4.24 Clausal form of equations with generalized terms. We now describe the transformation called *unfolding* which leads from an equation

$$f(\vec{x}) = \alpha[\vec{x}], \tag{1}$$

where the generalized term α may apply f, to a finite set of terminal clauses

$$\{\phi_1, \ldots, \phi_m\} \tag{2}$$

satisfying the following:

$$\forall \vec{x}\, f(\vec{x}) = \alpha[\vec{x}] \leftrightarrow \forall \phi_1 \wedge \cdots \wedge \forall \phi_m. \tag{3}$$

The set of clauses (2) is called a *clausal form* of the equation (1).

In one unfolding step we take a non-terminal clause ϕ of a form

$$f(\vec{x}) = \mathcal{D}^{\vec{\delta}_1,\ldots,\vec{\delta}_m}_{\rho_1,\ldots,\rho_m}(\chi_1, \beta_1, \ldots, \chi_m, \beta_m) \leftarrow \psi_1 \wedge \cdots \wedge \psi_k$$

and unfold the clause to the set of clauses $\{\phi_1, \ldots, \phi_m\}$, where every its clause ϕ_i is of the form $(1 \leq i \leq m)$:

$$f(\vec{x}) = \beta_i \leftarrow \psi_1 \wedge \cdots \wedge \psi_k \wedge \chi_i$$

Clearly, the clauses $\{\phi_1, \ldots, \phi_m\}$ satisfies the following *unfolding invariant*:

$$\forall \phi \leftrightarrow \forall \phi_1 \wedge \cdots \wedge \forall \phi_m. \tag{4}$$

The unfolding process for the equation (1) is started from the *initial* clause (1) and eventually leads to the set of terminal clauses (2). Property (3) follows from the unfolding invariant (4).

2.4.25 Clausal definitions. Suppose that

$$f(\vec{x}) = \alpha \tag{1}$$

is a generalized explicit definition of f if α does not apply f or a generalized regular recursive definition of f otherwise. Suppose further that the set of clauses

$$\{\phi_1, \ldots, \phi_m\} \tag{2}$$

is a clausal form of the equation (1). By 2.4.24(3) the function f is a unique function satisfying the clauses (2). We call the clauses (2) a *clausal definition* of the function f. The clausal definition is called *explicit* if the definition (1) is explicit and *recursive* otherwise.

Clausal definition (2) can be viewed as a function operator taking all functions applied in α and yielding the function f. Thus the only requirement imposed on clausal definitions is that they should be obtained by unfolding of some generalized definition of a form (1).

2.4.26 Presentation of clausal definitions. We may further simplify a clausal definition for the purposes of presentation to a human reader. For instance, we may rename variables of one of its clauses. We may eliminate variables of clauses in contexts like $\tau = x$ provided that the variable x does not occur in the term τ by substituting it for the corresponding term. For instance, the clause

$$\phi[x] \leftarrow \phi_1[x] \land \tau = x \land \phi_2[x]$$

is simplified into the equivalent clause

$$\phi[\tau] \leftarrow \phi_1[\tau] \land \phi_2[\tau].$$

We may simplified a clausal definition by omitting from it one or more *default* clauses which are clauses with the heads $f(\vec{\rho}) = 0$. Due to the omitted defaults and because of the writing of implications in the direction $\leftarrow$ only, a clausal definition is more than the statement of the properties asserted by the non-default clauses. In order to distinguish such a clausal definition from the mere assertion of properties we always write clausal definitions *aligned to the left*.

We can define a predicate R by a clausal definition which defines its characteristic function R_* and is such that the heads of the clauses have the form $R_*(\vec{\rho}) = 1$ or $R_*(\vec{\rho}) = 0$ and all applications of R_* in the bodies are one of the following forms: $R_*(\vec{\rho}) = 1$, $R_*(\vec{\rho}) \neq 0$, or $R_*(\vec{\rho}) = 0$. We can present such a definition in a *predicate form* where we replace $R_*(\vec{\rho}) = 1$ and $R_*(\vec{\rho}) \neq 0$ by $R(\vec{\rho})$ and $R_*(\vec{\rho}) = 0$ by $\neg R(\vec{\rho})$. We can also remove all *defaults* which are clauses with the heads $\neg R(\vec{\rho})$.

Clausal definitions are probably best explained with examples. The reader will note that the conditions of regularity of clausal definitions can be easily read off from their recursive clauses. For simplicity we will omit the recognizers and the destructors in the notation of generalized conditionals. We strongly recommend that readers interested in more details in programming and proving with the clausal language download the text [KV01].

2.4.27 Example. Consider the following recursive definition of the remainder function $x \bmod y$:

$$\begin{aligned} x \bmod y = \mathbf{case} & \\ & y = 0 \Rightarrow 0 \\ & y > 0 \Rightarrow \mathbf{case} \\ & \qquad x < y \Rightarrow x \\ & \qquad x \geq y \Rightarrow (x \dot{-} y) \bmod y \\ & \qquad \mathbf{end} \\ & \mathbf{end} \end{aligned} \tag{1}$$

The definition is regular in the first argument x since its condition of regularity

$$y > 0 \land x \geq y \rightarrow x \dot{-} y < y \tag{2}$$

is trivially satisfied.

We unfold the equation (1) to the following three clauses

$x \bmod y = 0 \leftarrow y = 0$ (3)
$x \bmod y = x \leftarrow y > 0 \land x < y$ (4)
$x \bmod y = (x \dot{-} y) \bmod y \leftarrow y > 0 \land x \geq y.$ (5)

Then we omit the first clause (3) by default whereby we obtain

$$x \bmod y = x \leftarrow y > 0 \land x < y \tag{6}$$
$$x \bmod y = (x \dot{-} y) \bmod y \leftarrow y > 0 \land x \geq y. \tag{7}$$

The final clauses (6) and (7) forms the clausal definition of the remainder function $x \bmod y$ derived from the equation (1). Note that the condition of regularity (2) can be easily read off from the last clause (7).

2.4.28 Example. Consider the following recursive definition of the characteristic function $Even_*(x)$ of the predicate $Even(x) \leftrightarrow \exists y\, x = 2 \cdot y$ holding of even numbers:

$$\begin{aligned} Even_*(x) = \mathbf{case}& \\ &x = 0 \Rightarrow 1 \\ &x = y + 1 \Rightarrow_y \mathbf{case} \\ &\qquad\qquad Even_*(y) \neq 0 \Rightarrow 0 \\ &\qquad\qquad Even_*(y) = 0 \Rightarrow 1 \\ &\qquad\quad \mathbf{end} \\ &\mathbf{end} \end{aligned} \tag{1}$$

The definition is regular in x, since its condition of regularity

$$x = y + 1 \rightarrow y < x \tag{2}$$

is trivially satisfied.

We unfold the equation (1) to the clauses

$$Even_*(x) = 1 \leftarrow x = 0 \tag{3}$$
$$Even_*(x) = 0 \leftarrow x = y + 1 \land Even_*(y) \neq 0 \tag{4}$$
$$Even_*(x) = 1 \leftarrow x = y + 1 \land Even_*(y) = 0. \tag{5}$$

We simplify the clauses by eliminating the local variable x from their bodies; in particular, the variable x is substituted for the $y + 1$ in the both clauses (4) and (5). After simplification we obtain the following clauses

$$Even_*(0) = 1 \tag{6}$$
$$Even_*(y + 1) = 0 \leftarrow Even_*(y) \neq 0 \tag{7}$$
$$Even_*(y + 1) = 1 \leftarrow Even_*(y) = 0 \tag{8}$$

The clauses are then transformed into predicate form:

$$Even(0) \tag{9}$$
$$\neg Even(y + 1) \leftarrow Even(y) \tag{10}$$
$$Even(y + 1) \leftarrow \neg Even(y). \tag{11}$$

Finally, we omit the second clause (10) by default whereby we obtain

$$Even(0) \tag{12}$$
$$Even(y + 1) \leftarrow \neg Even(y) \tag{13}$$

The final clauses (12) and (13) forms the clausal definition of the predicate $Even(x)$. Note that the condition of regularity (2) can be easily read off from the last clause (13).

2.5 Exercises

2.5.1 Exercise. Suppose that $\mathcal{F}$ is a class of functions closed under explicit definitions of predicates with bounded formulas. Show that the following predicates are in $\mathcal{F}$:

- the predicate $x \mid y$ holding if x divides y,
- the predicate $Prime(x)$ holding if x is a prime number,
- the predicate $\exists y\, x = 2^y$ holding if x is a power of two,
- the predicate $\exists y\, x = 3^y$ holding if x is a power of three,
- the predicate $\exists y\, x = 4^y$ holding if x is a power of four,
- the predicate $\exists y\, x = p^y$ holding if x is a power of the prime number p,

3. Primitive Recursive Functions

We will in this chapter investigate a class of effectively computable functions which has a particulary simple inductive definition and actually it was the first known class of computable functions. The class of primitive recursive functions will be shown closed in the following sections under three successively stronger special cases of recursive definitions. The strongest special case is that of recursive definitions regular in primitive recursive measures (see Thm. 3.3.11). Contrary to the original expectations, primitive recursive functions are not closed under recursive definitions (see Par. 5.1.2).

3.1 Primitive Recursion

3.1.1 Operator of primitive recursion. The operator of *primitive recursion* takes an n-ary function g and an $(n+2)$-ary function h and yields the $(n+1)$-ary function f defined by

$$f(0,\vec{y}) = g(\vec{y}) \tag{1}$$
$$f(x+1,\vec{y}) = h(x, f(x,\vec{y}), \vec{y}). \tag{2}$$

The argument x is called the *recursive argument* whereas the arguments $\vec{y}$ are *parameters*. Note that we require that the definition has at least one parameter.

3.1.2 Primitive recursive functions. The class of *primitive recursive* functions is generated from the successor function $x+1$, the zero function $Z(x)=0$, and the identity functions $I_i^n(\vec{x}) = x_i$ by composition of functions and primitive recursion.

We denote by PRIMREC the class of primitive recursive functions. We denote by PRIMREC($\mathcal{F}$) primitive recursive functions in the class $\mathcal{F}$. Clearly we have PRIMREC = PRIMREC($\emptyset$).

3.1.3 Unary constant functions are primitive recursive. We show by induction on n that every unary constant function $C_n(x) = n$ is primitive recursive. In the base case we have $C_0 = Z$. In the inductive case we assume that the constant function C_n is primitive recursive by IH and define the constant function C_{n+1} as a primitive recursive function by unary composition:

$$C_{n+1}(x) = C_n(x) + 1.$$

3.1.4 Case discrimination function is primitive recursive. The case analysis function D satisfies $D(0, y, z) = z$ and $D(x+1, y, z) = y$. We derive D as a primitive recursive function by primitive recursion from the idenitity functions I_2^2 and I_2^4:

$$\begin{aligned} D(0, y, z) &= I_2^2(y, z) \\ D(x+1, y, z) &= I_3^4(x, D(x, y, z), y, z). \end{aligned}$$

3.1.5 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under explicit definitions of functions.*

Proof. The claim follows directly from Par. 3.1.3, Par. 3.1.4, and Thm. 2.2.10 (see also Par. 2.2.11(i)). ☐

3.1.6 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under generalized explicit definitions of functions.*

Proof. Directly from Thm. 3.1.5 and Thm. 2.4.19. ☐

3.1.7 Boolean functions are primitive recursive. Primitive recursive functions are closed under explicit definitions and thus, by Par. 2.2.5, the boolean functions are primitive recursive.

3.1.8 Primitive recursive definitions. A *primitive recursive definition* of a function f is of a form

$$f(\vec{y}, 0, \vec{z}) = \tau_1[\vec{y}, \vec{z}] \tag{1}$$

$$f(\vec{y}, x+1, \vec{z}) = \tau_2[\dot{\lambda}\vec{y}_1 x_1 \vec{z}_1.f(\vec{y}, x, \vec{z}); \vec{y}, x, \vec{z}], \tag{2}$$

where $\tau_1[\vec{y}, \vec{z}]$ and $\tau_2[f; \vec{y}, x, \vec{z}]$ are terms containing at most the indicated variables free and the term τ_1 may not applied the function symbol f. Note that every recursive application in the term of the second identity has the form $f(\vec{y}, x, \vec{z})$. Note also that the parameters $\vec{y}$ and $\vec{z}$ may be empty.

The primitive recursive definition can be viewed as a function operator taking all functions applied in τ_1 and τ_2 and yielding the function f.

3.1.9 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under primitive recursive definitions.*

Proof. Let f be defined by the above primitive recursive definition from functions in $\mathcal{F}$. First we explicitly define in $\mathcal{F}$ two auxiliary functions:

$$g(w,\vec{y},\vec{z}) = \tau_1[\vec{y},\vec{z}] \tag{1}$$
$$h(x,a,w,\vec{y},\vec{z}) = \tau_2[\dot{\lambda}\vec{y}_1 x_1 \vec{z}_1.a;\vec{y},x,\vec{z}]. \tag{2}$$

We then define a function $f_1 \in \mathcal{F}$ by primitive recursion:

$$f_1(0,w,\vec{y},\vec{z}) = g(w,\vec{y},\vec{z}) \tag{3}$$
$$f_1(x+1,w,\vec{y},\vec{z}) = h(x,f_1(x,w,\vec{y},\vec{z}),w,\vec{y},\vec{z}). \tag{4}$$

We now prove by induction on x:

$$f(\vec{y},x,\vec{z}) = f_1(x,0,\vec{y},\vec{z}). \tag{5}$$

In the base case we have:

$$f(\vec{y},0,\vec{z}) \overset{3.1.8(1)}{=} \tau_1[\vec{y},\vec{z}] \overset{(1)}{=} g(0,\vec{y},\vec{z}) \overset{(3)}{=} f_1(0,0,\vec{y},\vec{z}).$$

In the inductive case we have:

$$\begin{aligned} f(\vec{y},x+1,\vec{z}) &\overset{3.1.8(2)}{=} \tau_2[\dot{\lambda}\vec{y}_1 x_1 \vec{z}_1.f(\vec{y},x,\vec{z});\vec{y},x,\vec{z}] \overset{(2)}{=} h(x,f(\vec{y},x,\vec{z}),0,\vec{y},\vec{z}) \overset{\mathrm{IH}}{=} \\ &= h(x,f_1(x,0,\vec{y},\vec{z}),0,\vec{y},\vec{z}) \overset{(4)}{=} f_1(x+1,0,\vec{y},\vec{z}). \end{aligned}$$

With (5) proved we can use it as explicit definition of $f \in \mathcal{F}$.

The reader will note that the reason why we have added to the arguments of f_1 the seemingly superfluous parameter w is to cater to the case when $\vec{y} \equiv \vec{z} \equiv \emptyset$ where we need that the function f_1 has at least one parameter as required by the operator of primitive recursion. □

3.1.10 Generalized primitive recursive definitions. In order to obtain the comfort of recursive definitions in the clausal form we introduce a special form of generalized recursive definitions and show in Thm. 3.1.11 that the primitive recursive functions are closed under it.

Generalized primitive recursive definitions are generalized regular clausal definitions of a form

$$\begin{aligned} f(\vec{y},x_1,\vec{z}) = {}& \mathbf{case} \\ & x_1 = 0 \Rightarrow \beta_1[\vec{y},\vec{z}] \\ & x_1 = x+1 \Rightarrow_x \beta_2[\dot{\lambda}\vec{y}_1 x_1 \vec{z}_1.f(\vec{y},x,\vec{z});\vec{y},x,\vec{z}] \\ & \mathbf{end} \end{aligned} \tag{1}$$

regular in x_1, where $\beta_1[\vec{y},\vec{z}]$ and $\beta_2[f;\vec{y},x,\vec{z}]$ are generalized terms containing at most the indicated variables free and the term β_1 may not applied the function symbol f. Note that every recursive application in the generalized term of the second identity has the form $f(\vec{y},x,\vec{z})$. Note also that the function f satisfies

$$f(\vec{y},0,\vec{z}) = \beta_1[\vec{y},\vec{z}] \tag{2}$$
$$f(\vec{y},x+1,\vec{z}) = \beta_2[\dot{\lambda}\vec{y}_1 x_1 \vec{z}_1.f(\vec{y},x,\vec{z});\vec{y},x,\vec{z}]. \tag{3}$$

The generalized primitive recursive definition can be viewed as a function operator taking all functions applied in the generalized terms β_1 and β_2 and yielding the function f.

3.1.11 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under generalized primitive recursive definitions.*

Proof. Let the function f be defined by the generalized primitive recursive definition 3.1.10(1) from functions of $\mathcal{F}$. We have

$$f(\vec{y},0,\vec{z}) = \beta_1^\star[\vec{y},\vec{z}] \tag{1}$$
$$f(\vec{y},x+1,\vec{z}) = \beta_2^\star[\dot{\lambda}\vec{y}_1 x_1 \vec{z}_1.f(\vec{y},x,\vec{z});\vec{y},x,\vec{z}] \tag{2}$$

by 3.1.10(2)(3) and the property 2.4.17(1) of the translation function $\alpha^\star$. By Thm. 3.1.9 we can take the identities (1) and (2) as a derivation of f in $\mathcal{F}$. ☐

3.1.12 Addition is primitive recursive. The addition function $x+y$ is a primitive recursive function by primitive recursive definition:

$0+y = y$
$(x+1)+y = (x+y)+1.$

Note that the last $+$ in the second clause belongs to the successor function.

3.1.13 Predecessor function is primitive recursive. The predecessor function $x \dot{-} 1$ is a primitive recursive function by primitive recursive definition:

$0 \dot{-} 1 = 0$
$(x+1) \dot{-} 1 = x.$

3.1.14 Modified subtraction is primitive recursive. The modified subtraction function $x \dot{-} y$ is a primitive recursive function by primitive recursive definition:

$x \dot{-} 0 = x$
$x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1.$

Note that in the second clause the last $\dot{-}$ belongs to the the predecessor function $x \dot{-} 1$.

3.1.15 Multiplication is primitive recursive. The multiplication function $x \cdot y$ is is a primitive recursive function by primitive recursive definition:

$0 \cdot y = 0$
$(x+1) \cdot y = x \cdot y + y.$

3.1.16 Exponentiation function is primitive recursive. The exponentiation function x^y is a primitive recursive function by primitive recursive definition:

$x^0 = 1$
$x^{y+1} = x^y \cdot x.$

3.1.17 Predicate $x \leq y$ is primitive recursive. The characteristic function of the binary predicate $x \leq y$ is primitive recursive by explicit definition:

$$(x \leq_* y) = \neg_*(x \dot{-} y),$$

since we have $x \leq y \Leftrightarrow x \dot{-} y = 0 \Leftrightarrow \neg_*(x \dot{-} y) = 1$.

3.1.18 The equality predicate is primitive recursive. The characteristic function of the binary predicate $x = y$ is primitive recursive by explicit definition:

$$(x =_* y) = (x \leq_* y) \wedge_* (y \leq_* x).$$

3.1.19 Lemma. *Primitively recursively closed classes $\mathcal{F}$ are closed under the operator of bounded minimalization.*

Proof. If $f(z, \vec{x}) = \mu_{y \leq z}[g(y, \vec{x}) = 1]$ from a function $g \in \mathcal{F}$ then we can define $f \in \mathcal{F}$ by (generalized) primitive recursive definition:

$f(0, \vec{x}) = 0$
$f(z+1, \vec{x}) = f(z, \vec{x}) \leftarrow g(f(z, \vec{x}), \vec{x}) = 1$
$f(z+1, \vec{x}) = z + 1 \leftarrow g(f(z, \vec{x}), \vec{x}) \neq 1 \wedge g(z+1, \vec{x}) = 1$
$f(z+1, \vec{x}) = 0 \leftarrow g(f(z, \vec{x}), \vec{x}) \neq 1 \wedge g(z+1, \vec{x}) \neq 1.$ □

3.1.20 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under explicit definitions of predicates with bounded formulas and under definitions of functions with bounded minimalization.*

Proof. The class $\mathcal{F}$ contains the predicate $\leq$ and so the theorem follows from Thm. 2.2.12 and Thm. 2.2.13 by Thm. 3.1.5 and Lemma 3.1.19. □

3.1.21 Integer division is primitive recursive. We define the integer division $x \div y$ as a primitive recursive function by bounded minimalization:

$$x \div y = \mu_{q \leq x}[x < (q+1) \cdot y].$$

3.1.22 Remainder function is primitive recursive. We define the remainder function $x \bmod y$ as a primitive recursive function by explicit clausal definition:

$x \bmod y = x \dot{-} (x \div y) \cdot y \leftarrow y > 0.$

3.1.23 The operator of iteration. We have encountered in Par. 1.2.12 the iteration notation $g^n(x)$ for constants n. The notation is generalized with the help of the operator of *iteration* which takes a unary function g and yields a binary function f satisfying:

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

We say that f is the *iteration of* g. We will often write $g^{\tau_1}(\tau_2)$ as an abbreviation for the application $f(\tau_1, \tau_2)$. We clearly have:

$$g^0(x) = f(0,x) = x \tag{1}$$

$$g^1(x) = f(1,x) = g(x) \tag{2}$$

$$g^{n+m}(x) = f(n+m,x) = f\big(n, f(m,x)\big) = g^n g^m(x). \tag{3}$$

3.1.24 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under the iteration of functions.*

Proof. If the binary function f is obtained by the iteration of a $g \in \mathcal{F}$ we define $f \in \mathcal{F}$ by primitive recursive definition:

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

□

3.2 Course of Values Recursion

In this section we show that primitive recursive functions are closed under *course of values recursive definitions* which are recursive definitions with the recursive argument going arbitrarily down in the recursive applications and with the parameters unchanged. For that we need to show that the pairing function (x,y) and the projection functions H and T are primitive recursive functions.

Pairing Function is Primitive Recursive

3.2.1 Catalan function is primitive recursive. We have introduced the pairing function (x,y) in Par. 1.3.8 by set theoretical means of enumerating all binary trees. We need a primitive recursive definition of the pairing function. For that we need to know that the Catalan function $C(n)$ (see Par. 1.3.11), which yields the count of numbers x with the pair size n, is a primitive recursive function.

We cannot use the convolution recurrences for C from Par. 1.3.11 because they constitute a course of values recursive definition. We show instead that the Catalan function satisfies:

$$C(n) = \frac{1}{n+1} \cdot \binom{2 \cdot n}{n}. \tag{1}$$

This certainly holds for $n = 0$ because $C(0) = 1$ and so we assume $n > 0$. Let τ be a (fully parenthesized) pair numeral such that $|\tau|_p = n$. If we omit from τ the commas and left parentheses we get a string of a form 0α where α consists of n zeroes and n right parentheses. The string 0α can be viewed as a postfix representation of the pair numeral τ (right parentheses play the role of postfix pairing operators). We have $\binom{2\cdot n}{n}$ different strings 0α out of which $C(n)$ are in the postfix form. The reader will note that when counting from left to right in a postfix form 0α the count of zeroes is always greater than the count of right parentheses. This also holds in the converse when a string 0α satisfies the above counting condition then it is a postfix form. We get all non-postfix strings 0α by choosing $n-1$ zeroes into the $2 \cdot n$ positions of the string α and then filling up the free positions from left to right by n right parentheses. The last free position is filled by the remaning zero. The string 0α thus obtained cannot be a postfix form because there are n right parentheses and at most that much zeroes to the left of the last position filled. Thus there are $\binom{2\cdot n}{n-1}$ non-postfix strings and we have:

$$C(n) = \binom{2 \cdot n}{n} - \binom{2 \cdot n}{n-1} = \binom{2 \cdot n}{n} - \frac{n}{n+1} \cdot \binom{2 \cdot n}{n} = \frac{1}{n+1} \cdot \binom{2 \cdot n}{n}.$$

We have

$$\begin{aligned} C(n+1) &\overset{(1)}{=} \frac{1}{n+2} \cdot \binom{2 \cdot n+2}{n+1} = \frac{1}{n+2} \cdot \frac{(2 \cdot n+2) \cdot (2 \cdot n+1)}{(n+1)^2} \cdot \binom{2 \cdot n}{n} = \\ &= \frac{4 \cdot n+2}{n+2} \cdot C(n). \end{aligned}$$

Hence, the Catalan function C is a primitive recursive function by primitive recursion:

$C(0) = 1$
$C(n+1) = (4 \cdot n + 2) \cdot C(n) \div (n+2).$

3.2.2 Function σ is primitive recursive. The function $\sigma(n)$ which yields the least number with the pair size n was introduced in Par. 1.3.11 by recurrences which constitute its primitive recursive derivation by primitive recursive definition:

$\sigma(0) = 0$
$\sigma(n+1) = \sigma(n) + C(n).$

3.2.3 Pair size function is primitive recursive. The pair size function $|x|_p$ cannot be defined as a primitive recursive function by its natural definition:

$|0|_p = 0$
$|v, w|_p = |v|_p + |w|_p + 1$

because the folding of the pair discrimination introduces the projection functions H and T.

We use instead the property 1.3.11(5) which says that

$$\sigma(|x|_p) \le x < \sigma(|x|_p + 1)$$

and the property 1.3.12(5) which says that $|x|_p \le x$ and define $|x|_p$ as a primitive recursive function by bounded minimalization:

$$|x|_p = \mu_{n \le x}[x < \sigma(n+1)].$$

3.2.4 Relative offset function. We define the unary *relative offset* function $Ro(x)$ as the excess of the number x over the number $\sigma(|x|_p)$, i.e.

$$Ro(x) = x - \sigma(|x|_p)$$

which is a non-negative number by 1.3.11(5). The function Ro is thus primitive recursive by explicit definition:

$$Ro(x) = x \mathbin{\dot{-}} \sigma(|x|_p).$$

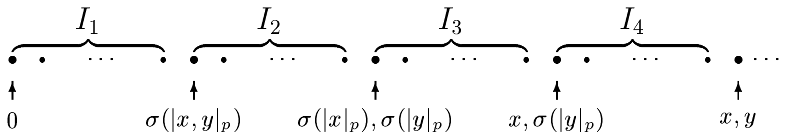

Fig. 3.1. Arithmetic derivation of the pairing function (x, y)

3.2.5 Pairing function is primitive recursive. We are now ready to give an arithmetic derivation of the pairing function as a primitive recursive function. We refer the reader to Fig. 3.1 where we have shown four intervals I_1, I_2, I_3, and I_4 consisting jointly of all numbers preceding the number (x, y).

The interval I_1 consists of $\sigma(|x, y|_p) = \sigma(|x|_p + |y|_p + 1)$ numbers z such that

$$0 \le z < \sigma(|x, y|_p).$$

The interval I_2 consists of numbers v, w such that

$$\sigma(|x, y|_p) \le v, w < \sigma(|x|_p), \sigma(|y|_p).$$

There is $\sum_{i < |x|_p} C(i) \cdot C(|x|_p + |y|_p \mathbin{\dot{-}} i)$ such numbers. The last can be expressed with the help of the binary function $\lambda xz. \sum_{i<x} C(i) \cdot C(z \mathbin{\dot{-}} i)$ which is a primitive recursive function by primitive recursion:

$\sum_{i<0} C(i) \cdot C(z \dot{-} i) = 0$
$\sum_{i<x+1} C(i) \cdot C(z \dot{-} i) = C(x) \cdot C(z \dot{-} x) + \sum_{i<x} C(i) \cdot C(z \dot{-} i)$.

The interval I_3 consists of $Ro(x) \cdot C(|y|_p)$ numbers v, w such that

$$\sigma(|x|_p), \sigma(|y|_p) \leq v, w < x, \sigma(|y|_p).$$

The interval I_4 consists of $Ro(y)$ numbers v, w such that

$$x, \sigma(|y|_p) \leq v, w < x, y.$$

Hence, the number (x, y) is preceded by as many numbers as there are in the intervals I_1, I_2, I_3, and I_4 and we have the following primitive recursive derivation of the pairing function by explicit definition:

$$(x, y) = \sigma(|x|_p + |y|_p + 1) + \sum_{i<|x|_p} C(i) \cdot C(|x|_p + |y|_p \dot{-} i) + \\ + Ro(x) \cdot C(|y|_p) + Ro(y).$$

3.2.6 Projection functions are primitive recursive. Now that we know that the pairing function is primitive recursive we can define the projection functions H and T as primitive recursive functions by bounded minimalization:

$$H(x) = \mu_{v<x}[\exists w < x\, x = v, w]$$
$$T(x) = \mu_{w<x}[\exists v < x\, x = v, w].$$

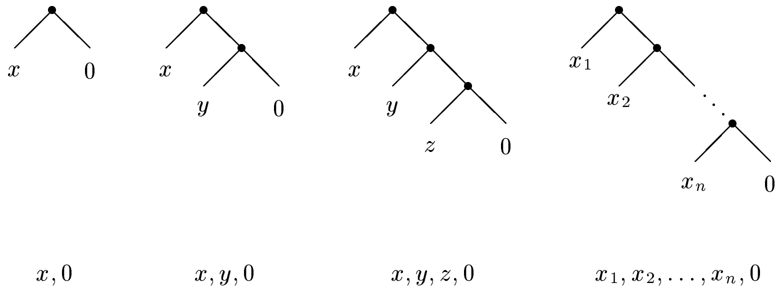

Fig. 3.2. Lists

3.2.7 Lists. There is a simple way of arithmetizing finite sequences over natural numbers. We assign the code 0 to the empty sequence $\emptyset$. The non-empty sequence $x_1\, x_2\, \ldots\, x_n$ is coded by the number $(x_1, x_2, \ldots, x_n, 0)$ (see Fig. 3.2). The reader will note that the assignment of codes is one to one, every finite sequence of natural numbers is coded by exactly one natural number, and vice versa, every natural number is the code of exactly one finite sequence of natural numbers. Codes of finite sequences are called *lists* in computer science and this is how we will be calling them from now on.

3.2.8 List indexing function is primitive recursive. The *list indexing* function $(x)_i$ yields the i-th element of a list x (counting from 0), i.e.

$$(x_0, \ldots, x_i, \ldots, x_{n-1}, 0)_i = x_i. \tag{1}$$

The indexing function has the following recursive definition regular in the second argument:

$$\begin{aligned} &(v, w)_0 = v \\ &(v, w)_{i+1} = (w)_i. \end{aligned}$$

Although the pairing discrimination is admissible, this is not a definition by primitive recursion because the first argument is modified in the recursive application. We can, however, explicitly derive the list indexing function as a primitive recursive function from the iteration of T:

$$(x)_i = H\,T^i(x).$$

3.2.9 List length function is primitive recursive. The *list length* function $L(x)$ yields the length element of a list x, i.e.

$$L(x_0, \ldots, x_i, \ldots, x_{n-1}, 0) = n. \tag{1}$$

The list length function has the following natural regular recursive definition:

$$\begin{aligned} &L(0) = 0 \\ &L(v, w) = L(w) + 1. \end{aligned}$$

We can explicitly derive the list length function as a primitive recursive function by the bounded minimalization:

$$L(x) = \mu_{n \le x}[T^n(x) = 0].$$

Course of Values Recursion

3.2.10 Course of values functions. For an $(n+1)$-ary function f we call the $(n+1)$-ary function $\overline{f}$ defined by primitive recursive definition:

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

the *course of values function for f*. It should be clear that $\overline{f}(x, \vec{y})$ yields the list of all values of f from $x - 1$ to 0:

$$\overline{f}(x, \vec{y}) = f(x-1, \vec{y}), f(x-2, \vec{y}), \ldots, f(1, \vec{y}), f(0, \vec{y}), 0.$$

This property is formally expressed by the following formulas:

$$L\,\overline{f}(x,\vec{y}) = x \tag{1}$$

$$\forall z\big(z < x \to \big(\overline{f}(x,\vec{y})\big)_{x-(z+1)} = f(z,\vec{y})\big). \tag{2}$$

Both properties are proved by induction on x. We prove here only the second one. The base case holds trivially since $z < 0$ cannot be satisfied. In the inductive case we assume $z < x+1$ and consider two cases. If $z = x$ we have:

$$\big(\overline{f}(x+1,\vec{y})\big)_{x+1-(z+1)} = \big(f(x,\vec{y}),\overline{f}(x,\vec{y})\big)_0 = f(z,\vec{y}).$$

If $z < x$ then we have:

$$\begin{aligned}\big(\overline{f}(x+1,\vec{y})\big)_{x+1-(z+1)} &= \big(f(x+1,\vec{y}),\overline{f}(x,\vec{y})\big)_{x+1-(z+1)} = \\ &= \big(\overline{f}(x,\vec{y})\big)_{x-(z+1)} \overset{\text{IH}}{=} f(z,\vec{y}).\end{aligned}$$

3.2.11 Course of values recursive definitions. Recursive definitions where the recursive argument goes arbitraily down in the recursive applications while the parameters stay unchanged are the obvious generalization of primitive recursive definitions.

Suppose that

$$f(\vec{x}) = \tau[f;\vec{x}] \tag{1}$$

is a regular recursive definition of an n-ary function f regular in an i-th argument and is such that every occurrence of a recursive application of f in the term τ has the form

$$f(x_1,\ldots,x_{i-1},\rho_i,x_{i+1},\ldots,x_n)$$

for some ρ_i. The definition (1) is then called the *course of values recursive definition* of f.

The following theorem asserts that primitive recursive functions are closed under course of values recursive definitions.

3.2.12 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under course of values recursive definitions.*

Proof. We prove the claim for the following special case when the function f is defined by course of values recursive definition from the functions of $\mathcal{F}$ regular in the *first* argument:

$$f(x,\vec{y}) = \tau[f;x,\vec{y}].$$

We define its course of values function by primitive recursive definition as a primitive recursive function:

$$\overline{f}(0,\vec{y}) = 0$$
$$\overline{f}(x+1,\vec{y}) = \tau[\dot{\lambda} z,\vec{y}.\big(\overline{f}(x,\vec{y})\big)_{x \dot{-} (z+1)};x,\vec{y}], \overline{f}(x,\vec{y}).$$

We show, by induction on x, that the property 3.2.10(2) holds . In the base case there is nothing to prove. In the inductive case take any $z < x+1$ and consider two cases.

If $z < x$ then we have

$$\big(\overline{f}(x+1,\vec{y})\big)_{x+1 \dot{-} (z+1)} \stackrel{\text{def}}{=} \big(\tau[\dot{\lambda} z,\vec{y}.\big(\overline{f}(x,\vec{y})\big)_{x \dot{-} (z+1)};x,\vec{y}], \overline{f}(x,\vec{y})\big)_{x \dot{-} (z+1)+1} =$$
$$= \big(\overline{f}(x,\vec{y})\big)_{x \dot{-} (z+1)} \stackrel{\text{IH}}{=} f(z,\vec{y}).$$

If $z = x$ then we first prove by (the inner) induction on the structure of subterms $\rho[f;x,\vec{y}]$ of τ an auxiliary property:

$$\Gamma^\tau_\rho[f;x,\vec{y}] \to \rho[\dot{\lambda} z,\vec{y}.\big(\overline{f}(x,\vec{y})\big)_{x \dot{-} (z+1)};x,\vec{y}] = \rho[f;x,\vec{y}], \tag{1}$$

where Γ^τ_ρ governs ρ in τ. So take any subterm ρ of τ, assume $\Gamma^\tau_\rho[f;x,\vec{y}]$, and procced by the case analysis on the structure of ρ. If $\rho \equiv f(\rho_1[f;x,\vec{y}],\vec{y})$ then $\rho_1[f;x,\vec{y}] < x$ by regularity and we obtain

$$\big(\overline{f}(x,\vec{y})\big)_{x \dot{-} (\rho_1[\dot{\lambda} z,\vec{y}.(\overline{f}(x,\vec{y}))_{x \dot{-} (z+1)};x,\vec{y}]+1)} \stackrel{\text{inner IH}}{=}$$
$$= \big(\overline{f}(x,\vec{y})\big)_{x \dot{-} (\rho_1[f;x,\vec{y}]+1)} \stackrel{\text{outer IH}}{=} f(\rho_1[f;x,\vec{y}],\vec{y}).$$

The remaining cases are straightforward and left to the reader. With the auxilliary property proved we prove the inductive step of the outer induction when $z = x$ as follows:

$$\big(\overline{f}(x+1,\vec{y})\big)_{x+1 \dot{-} (x+1)} \stackrel{\text{def}}{=} \big(\tau[\dot{\lambda} z,\vec{y}.\big(\overline{f}(x,\vec{y})\big)_{x \dot{-} (z+1)};x,\vec{y}], \overline{f}(x,\vec{y})\big)_0 =$$
$$= \tau[\dot{\lambda} z,\vec{y}.\big(\overline{f}(x,\vec{y})\big)_{x \dot{-} (z+1)};x,\vec{y}] \stackrel{(1)}{=} \tau[f;x,\vec{y}] \stackrel{\text{def}}{=} f(z,\vec{y}).$$

The general case can be easily obtained from the special case and is left to the reader. □

3.2.13 Generalized course of values recursive definitions. Suppose that

$$f(\vec{x}) = \alpha[f;\vec{x}] \tag{1}$$

is a generalized regular recursive definition of an n-ary function f regular in an i-th argument and is such that every occurrence of a recursive application of f in the generalized term α has the form

$$f(x_1,\ldots,x_{i-1},\beta_i,x_{i+1},\ldots,x_n)$$

for some β_i. The definition (1) is then called the *generalized course of values recursive definition* of f.

Note that the equation (1) is a generalized course of values recursive definition of f iff its translation $f(\vec{x}) = \alpha^\star[f;\vec{x}]$ is a course of values recursive definition of f. As a corollary of Thm. 3.2.12 we obtain the following claim.

3.2.14 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under generalized course of values recursive definitions.*

3.2.15 List concatenation function is primitive recursive. In the sequel we will need the binary *list concatenation* function $x \oplus y$ concatenating two lists, i.e.

$$(x_1, \ldots, x_n, 0) \oplus (y_1, \ldots, y_m, 0) = x_1, \ldots, x_n, y_1, \ldots, y_m, 0. \qquad (1)$$

The function is defined by course of values recursion regular in the first argument as a primitive recursive function:

$$0 \oplus y = y$$
$$(v, w) \oplus y = v, w \oplus y.$$

3.3 Course of Values Recursion with Measure

R. Péter [Pét67] defined classes of *k-recursive* functions with nested recursion. She has proved that the classes form a strictly increasing hierarchy. By the theorem of Tait [Tai61] (see also [Ros82]) k-recursive functions can be defined by $< \omega^{\omega^k}$-ordinal recursion. For instance, the well known Péter-Ackermann's function A (see Par. 5.1.2) is 2-recursive. Péter has proved in [Pét36] that primitive recursive functions are closed for the special case $k = 1$ of *simply nested recursion* where the recursive argument, say the i-th one, goes from $x+1$ to x and the parameters may be arbitrarily substituted for, even with nested recursive applications. Such simply nested recursion is then regular in the i-th argument. We generalize this result in Thm. 3.3.11 where we prove that primitive recursive functions are closed under regular recursive definitions with primitive recursive measures. This is done by the arithmetization of reductions by primitive recursive functions.

Arithmetization of Reductions

3.3.1 Fixing language and interpretation. In the current subsection we fix the language $\mathcal{L}$ of recursive terms and the total interpretation $\mathfrak{I}$ of the language $\mathcal{L}$. By $g(n, i, x)$ we denote the ternary function defined as

$$g(n,i,x) = \begin{cases} g_i^n(\vec{x}) & \text{if } g_i^n \in \mathcal{L} \text{ and } x = (\vec{x}), \\ 0 & \text{otherwise.} \end{cases}$$

We say that the function g *represents* the interpretation $\mathfrak{I}$. We clearly have

$$g_i^n(\vec{x}) = g(n, i, (\vec{x})) \tag{1}$$

for every oracle g_i^n of $\mathcal{L}$.

3.3.2 Arithmetization of recursive terms. We arithmetize R-terms and R-functions symbols with the following pair constructors:

$\boldsymbol{x}_i = 0, i$	*(variables)*
$\mathbf{0} = 1, 0$	*(zero)*
$\boldsymbol{S}(t) = 2, t$	*(successor)*
$\boldsymbol{Pr}(t) = 3, t$	*(predecessor)*
$\boldsymbol{D}_s(t_1, t_2, t_3) = 4, t_1, t_2, t_3$	*(conditional)*
$t_1 \bullet t_2 = 5, t_1, t_2$	*(curried application)*
$e[\![ts]\!] = 6, e, ts$	*(partial application)*
$\boldsymbol{f}_n = 0, n$	*(recursors)*
$\boldsymbol{\lambda}_n\boldsymbol{.}\, t = 1, n, t$	*(defined functions)*
$\boldsymbol{g}_i^n = 2, n, i.$	*(oracles)*

The arities of constructors are as shown in their definitions. We postulate that the binary constructor $\bullet$ groups to the left, i.e. that $t_1 \bullet t_2 \bullet t_3$ abbreviates $(t_1 \bullet t_2) \bullet t_3$.

We assign to every R-term τ and to every R-function symbol f their codes $\ulcorner\tau\urcorner$ and $\ulcorner f\urcorner$ inductively on the structure of R-terms and of R-function symbols:

$$\ulcorner x_i \urcorner = \boldsymbol{x}_i \tag{1}$$

$$\ulcorner 0 \urcorner = \mathbf{0} \tag{2}$$

$$\ulcorner \tau + 1 \urcorner = \boldsymbol{S}(\ulcorner\tau\urcorner) \tag{3}$$

$$\ulcorner \tau \dot{-} 1 \urcorner = \boldsymbol{Pr}(\ulcorner\tau\urcorner) \tag{4}$$

$$\ulcorner \mathcal{D}_s(\tau_1, \tau_2, \tau_3) \urcorner = \boldsymbol{D}_s(\ulcorner\tau_1\urcorner, \ulcorner\tau_2\urcorner, \ulcorner\tau_3\urcorner) \tag{5}$$

$$\ulcorner f(\tau_1, \ldots, \tau_n) \urcorner = \ulcorner f \urcorner [\![\ulcorner\tau_1\urcorner, \ldots, \ulcorner\tau_k\urcorner, 0]\!] \bullet \ulcorner\tau_{k+1}\urcorner \bullet \cdots \bullet \ulcorner\tau_n\urcorner \tag{6}$$

where k is the maximal number such that the terms $\tau_1, \ldots, \tau_k$ are numerals

$$\ulcorner \mathfrak{f}_n \urcorner = \boldsymbol{f}_n \tag{7}$$

$$\ulcorner \lambda_n.\tau \urcorner = \boldsymbol{\lambda}_n\boldsymbol{.}\ulcorner\tau\urcorner \tag{8}$$

$$\ulcorner g_i^n \urcorner = \boldsymbol{g}_i^n. \tag{9}$$

3.3.3 Codes of numerals. Applications of functions are reduced when their arguments are numerals. In order to recognize when the codes of arguments are already reduced we will need a unary predicate Nm holding of the codes of numerals, i.e. $Nm(t) \leftrightarrow \exists x\, t = \ulcorner \underline{x} \urcorner$. The predicate is primitive recursive by parameterless course of values recursive definition:

$Nm(\mathbf{0})$
$Nm\,\boldsymbol{S}(t) \leftarrow Nm(t)$.

We will need a unary *coding* function $\ulcorner \underline{x} \urcorner$ which takes a number x and yields the code of the numeral $\underline{x}$. The function is primitive recursive by primitive recursive definition:

$\ulcorner \underline{0} \urcorner = \mathbf{0}$
$\ulcorner \underline{x+1} \urcorner = \boldsymbol{S}(\ulcorner \underline{x} \urcorner)$.

Its inverse $Dc(t)$, called the *decoding* function, satisfies

$$Dc(\ulcorner \underline{x} \urcorner) = x. \tag{1}$$

The function is primitive recursive by parameterless course of values recursive definition:

$Dc(\mathbf{0}) = 0$
$Dc\,\boldsymbol{S}(t) = Dc(t) + 1$.

We will also need a binary function $Dcs(ts)$ decoding non-empty lists of numerals. The function satisfies

$$n > 0 \to Dcs(\ulcorner \underline{x_1} \urcorner, \dots, \ulcorner \underline{x_n} \urcorner, 0) = x_1, \dots, x_n \tag{2}$$

and it is primitive recursive by course of values recursive definition:

$Dcs(t, ts) = Dc(t) \leftarrow ts = 0$
$Dcs(t, ts) = Dc(t), Dcs(ts) \leftarrow ts \neq 0$.

3.3.4 Contraction function. The binary *contraction* function $t_1 \,\underline{\bullet}\, t_2$ associating to the left satisfies the identity

$$\ulcorner f(\tau_1, \dots, \tau_n) \urcorner = \ulcorner f \urcorner [\ulcorner \tau_1 \urcorner, \dots, \ulcorner \tau_k \urcorner, 0] \,\underline{\bullet}\, \ulcorner \tau_{k+1} \urcorner \,\underline{\bullet} \cdots \underline{\bullet}\, \ulcorner \tau_n \urcorner, \tag{1}$$

where the terms $\tau_1, \dots, \tau_k$ are numerals, and it is defined by explicit definition as a primitive recursive function:

$e[ts] \,\underline{\bullet}\, t_2 = e[ts \oplus (t_2, 0)] \leftarrow Nm(t_2)$
$t_1 \,\underline{\bullet}\, t_2 = t_1 \bullet t_2 \leftarrow \neg\big(\exists e \exists ts\, t_1 = e[ts] \wedge Nm(t_2)\big)$.

3.3.5 Arithmetization of substitution function. The substitution function $\tau[\lambda_n.\sigma; \vec{\underline{x}}]$ is over recursive terms. Its arithmetization $t[e; rs]$ is a ternary function which takes the code t of the R-term $\tau[\mathfrak{f}_n; x_1, \dots, x_n]$ with all free recursors and free variables indicated, the code e of the n-ary function symbol $\lambda_n.\sigma$ and the list $rs = \ulcorner \underline{x_1} \urcorner, \dots, \ulcorner \underline{x_n} \urcorner, 0$ of the codes of the numerals $\underline{x_1}, \dots, \underline{x_n}$, and yields the code of the R-term $\tau[\lambda_n.\sigma; \underline{x_1}, \dots, \underline{x_n}]$, i.e.

$$\ulcorner \tau \urcorner [\ulcorner \lambda_n.\sigma \urcorner ; \ulcorner \underline{x_1} \urcorner, \dots, \ulcorner \underline{x_n} \urcorner, 0] = \ulcorner \tau[\lambda_n.\sigma; \underline{x_1}, \dots, \underline{x_n}] \urcorner. \tag{1}$$

The arithmetized substitution function is primitive recursive by course of values definition regular in the first argument:

$\boldsymbol{x}_i[e; rs] = (rs)_{i \dot{-} 1}$
$\mathbf{0}[e; rs] = \mathbf{0}$
$\boldsymbol{S}(t)[e; rs] = \boldsymbol{S}(t[e; rs])$
$\boldsymbol{Pr}(t)[e; rs] = \boldsymbol{Pr}(t[e; rs])$
$\boldsymbol{D}_s(t_1, t_2, t_3)[e; rs] = \boldsymbol{D}_s(t_1[e; rs], t_2[e; rs], t_3[e; rs])$
$(t_1 \bullet t_2)[e; rs] = t_1[e; rs] \underline{\bullet}\, t_2[e; rs]$
$\boldsymbol{f}_n[ts][e; rs] = e[ts]$
$(\boldsymbol{\lambda}_n.\, t)[ts][e; rs] = (\boldsymbol{\lambda}_n.\, t)[ts]$
$\boldsymbol{g}_i^n[ts][e; rs] = \boldsymbol{g}_i^n[ts]$.

Property (1) is proved by induction on the structure of the R-term τ. If $\tau \equiv x_i$ with $1 \le i \le n$ then we have

$$\ulcorner x_i \urcorner[\ulcorner \lambda_n.\sigma \urcorner; \ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0] = \boldsymbol{x}_i[\ulcorner \lambda_n.\sigma \urcorner; \ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0] \stackrel{\text{def}}{=}$$
$$= (\ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0)_i \stackrel{3.2.8(1)}{=} \ulcorner \underline{x_{i \dot{-} 1}} \urcorner = \ulcorner x_{i \dot{-} 1}[\lambda_n.\sigma; \underline{x_1}, \ldots, \underline{x_n}] \urcorner.$$

If $\tau \equiv \mathfrak{f}_n(\tau_1, \ldots, \tau_k, \ldots, \tau_n)$, where k is the maximal number such that the terms $\tau_1, \ldots, \tau_k$ are numerals, then we have

$$\begin{aligned}
&\ulcorner \mathfrak{f}_n(\tau_1, \ldots, \tau_n) \urcorner[\ulcorner \lambda_n.\sigma \urcorner; \ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0] = \\
&= \big(\boldsymbol{f}_n[\ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_k \urcorner, 0] \bullet \ulcorner \tau_{k+1} \urcorner \bullet \cdots \bullet \ulcorner \tau_n \urcorner\big)[\ulcorner \lambda_n.\sigma \urcorner; \ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0] \stackrel{\text{def}}{=} \\
&= \ulcorner \lambda_n.\sigma \urcorner[\ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_k \urcorner, 0] \,\underline{\bullet} \\
&\qquad \underline{\bullet}\, \ulcorner \tau_{k+1} \urcorner[\ulcorner \lambda_n.\sigma \urcorner; \ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0] \,\underline{\bullet} \cdots \underline{\bullet} \\
&\qquad \underline{\bullet}\, \ulcorner \tau_n \urcorner[\ulcorner \lambda_n.\sigma \urcorner; \ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0] \stackrel{\text{IH's}}{=} \\
&= \ulcorner \lambda_n.\sigma \urcorner[\ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_k \urcorner, 0] \,\underline{\bullet} \\
&\qquad \underline{\bullet}\, \ulcorner \tau_{k+1}[\lambda_n.\sigma; \underline{x_1}, \ldots, \underline{x_n}] \urcorner \,\underline{\bullet} \cdots \underline{\bullet}\, \ulcorner \tau_n[\lambda_n.\sigma; \underline{x_1}, \ldots, \underline{x_n}] \urcorner \stackrel{3.3.4(1)}{=} \\
&= \ulcorner (\lambda_n.\sigma)\big(\tau_1, \ldots, \tau_k, \tau_{k+1}[\lambda_n.\sigma; \underline{x_1}, \ldots, \underline{x_n}], \ldots, \tau_n[\lambda_n.\sigma; \underline{x_1}, \ldots, \underline{x_n}]\big) \urcorner = \\
&= \ulcorner \big((\lambda_n.\sigma)(\tau_1, \ldots, \tau_n)\big)[\lambda_n.\sigma; \underline{x_1}, \ldots, \underline{x_n}] \urcorner.
\end{aligned}$$

The cases when τ is the application either of a defined function or of an oracle are similar. The remaining cases are straightforward and left to the reader.

3.3.6 Auxiliary functions. We will also need two auxiliary functions $Pn(t)$ and $Dn(t_1, t_2, t_3)$ satisfying

$$Pn(\ulcorner \underline{x} \urcorner) = \ulcorner \underline{x \dot{-} 1} \urcorner \qquad (1)$$
$$Dn(\ulcorner \underline{x} \urcorner, t_2, t_3) = D(x, t_2, t_3). \qquad (2)$$

The functions are defined explicitly as a primitive recursive functions:

$Pn(\mathbf{0}) = \mathbf{0}$
$Pn\, \boldsymbol{S}(t) = t$

$Dn(\mathbf{0}, t_2, t_3) = t_3$
$Dn(\boldsymbol{S}(t_1), t_2, t_3) = t_2$.

3.3.7 Arithmetization of one-step reduction. We intend to define a unary function Rd_g satisfying:

$$Rd_g(\ulcorner \underline{x} \urcorner) = \ulcorner \underline{x} \urcorner \tag{1}$$

$$\text{for every } \rho, \text{ if } \tau \rhd_1 \rho \text{ then } Rd_g(\ulcorner \tau \urcorner) = \ulcorner \rho \urcorner. \tag{2}$$

The function Rd_g is defined as primitive recursive in g by parameterless course of values definition:

$$
\begin{aligned}
&Rd_g(\mathbf{0}) = \mathbf{0}\\
&Rd_g\, \boldsymbol{S}(t) = \boldsymbol{S}\, Rd_g(t)\\
&Rd_g\, \boldsymbol{Pr}(t) = Pn(t) \leftarrow Nm(t)\\
&Rd_g\, \boldsymbol{Pr}(t) = \boldsymbol{Pr}\, Rd_g(t) \leftarrow \neg Nm(t)\\
&Rd_g\, \boldsymbol{D}_s(t_1, t_2, t_3) = Dn(t_1, t_2, t_3) \leftarrow Nm(t_1)\\
&Rd_g\, \boldsymbol{D}_s(t_1, t_2, t_3) = \boldsymbol{D}_s(Rd_g(t_1), t_2, t_3) \leftarrow \neg Nm(t_1)\\
&Rd_g(t_1 \bullet t_2) = t_1 \,\underline{\bullet}\, Rd_g(t_2) \leftarrow \exists e \exists ts\, t_1 = e\boldsymbol{[}ts\boldsymbol{]}\\
&Rd_g(t_1 \bullet t_2) = Rd_g(t_1) \,\underline{\bullet}\, t_2 \leftarrow \neg\exists e \exists ts\, t_1 = e\boldsymbol{[}ts\boldsymbol{]}\\
&Rd_g\, (\boldsymbol{\lambda}_n \boldsymbol{.}\, t)\boldsymbol{[}ts\boldsymbol{]} = t\boldsymbol{[}\boldsymbol{\lambda}_n \boldsymbol{.}\, t\boldsymbol{;}\, ts\boldsymbol{]}\\
&Rd_g\, \boldsymbol{g}_i^n\boldsymbol{[}ts\boldsymbol{]} = \ulcorner \underline{g(n, i, Dcs(ts))} \urcorner.
\end{aligned}
$$

Property (1) is proved by induction x. The base case is straightforward. In the inductive case we have

$$Rd_g(\ulcorner \underline{x+1} \urcorner) = Rd_g\, \boldsymbol{S}(\ulcorner \underline{x} \urcorner) \overset{\text{def}}{=} \boldsymbol{S}\, Rd_g(\ulcorner \underline{x} \urcorner) \overset{\text{IH}}{=} \boldsymbol{S}(\ulcorner \underline{x} \urcorner) = \ulcorner \underline{x+1} \urcorner.$$

Property (2) is proved by induction on the structure of the closed R-term τ. Take any ρ such that $\tau \rhd_1 \rho$ and consider two cases. If the term τ is a redex then we continue by the case analysis on the structure of the term τ. If $\tau \equiv \underline{x} \dot{-} 1$ then $\rho \equiv \underline{x \dot{-} 1}$ and we have

$$Rd_g(\ulcorner \underline{x} \dot{-} 1 \urcorner) = Rd_g\, \boldsymbol{Pr}(\ulcorner \underline{x} \urcorner) \overset{\text{def}}{=} Pn(\ulcorner \underline{x} \urcorner) \overset{3.3.6(1)}{=} \ulcorner \underline{x \dot{-} 1} \urcorner.$$

If $\tau \equiv \mathcal{D}_s(\underline{x}, \tau_2, \tau_3)$ then we consider two subcases. If $x \neq 0$ then $\rho \equiv \tau_2$ and we obtain

$$
\begin{aligned}
Rd_g(\ulcorner \mathcal{D}_s(\underline{x}, \tau_2, \tau_3) \urcorner) &= Rd_g\, \boldsymbol{D}_s(\ulcorner \underline{x} \urcorner, \ulcorner \tau_2 \urcorner, \ulcorner \tau_3 \urcorner) \overset{\text{def}}{=} Dn(\ulcorner \underline{x} \urcorner, \ulcorner \tau_2 \urcorner, \ulcorner \tau_3 \urcorner) \overset{3.3.6(2)}{=}\\
&= D(x, \ulcorner \tau_2 \urcorner, \ulcorner \tau_3 \urcorner) = \ulcorner \tau_2 \urcorner.
\end{aligned}
$$

The case when $x = 0$ is similar. If $\tau \equiv (\lambda_n.\sigma[\mathfrak{f}_n; \vec{x}])(\underline{\vec{x}})$ then $\rho \equiv \sigma[\lambda_n.\sigma; \underline{\vec{x}}]$ and we have

$$
\begin{aligned}
&Rd_g(\ulcorner (\lambda_n.\sigma)(\underline{x_1}, \ldots, \underline{x_n}) \urcorner) = Rd_g\, (\boldsymbol{\lambda}_n \boldsymbol{.} \ulcorner \sigma \urcorner)\boldsymbol{[}\ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0] \overset{\text{def}}{=}\\
&\quad = \ulcorner \sigma \urcorner \boldsymbol{[}\boldsymbol{\lambda}_n \boldsymbol{.} \ulcorner \sigma \urcorner \boldsymbol{;} \ulcorner \underline{x_1} \urcorner, \ldots, \ulcorner \underline{x_n} \urcorner, 0] \overset{3.3.5(1)}{=} \ulcorner \sigma[\lambda_n.\sigma; \underline{x_1}, \ldots, \underline{x_n}] \urcorner.
\end{aligned}
$$

If $\tau \equiv g_i^n(\underline{\vec{x}})$ then $\rho \equiv \underline{g_i^n(\vec{x})}$ and we have

$$\begin{aligned}
&Rd_g(\ulcorner g^n_i(\underline{x_1},\dots,\underline{x_n})\urcorner) = Rd_g\ \boldsymbol{g}^n_i[\ulcorner\underline{x_1}\urcorner,\dots,\ulcorner\underline{x_n}\urcorner,0] \stackrel{\text{def}}{=}\\
&\quad = \ulcorner\underline{g(n,i,Dcs(\ulcorner\underline{x_1}\urcorner,\dots,\ulcorner\underline{x_n}\urcorner,0)}\urcorner \stackrel{3.3.3(2)}{=} \ulcorner\underline{g(n,i,x_1,\dots,x_n)}\urcorner \stackrel{3.3.1(1)}{=}\\
&\quad = \ulcorner\underline{g^n_i(x_1,\dots,x_n)}\urcorner.
\end{aligned}$$

Now suppose that the term τ is not a redex. We continue by the case analysis on the structure of the term τ. If $\tau \equiv \tau_1 + 1$ then $\tau_1 \rhd_1 \rho_1$ for some ρ_1. Thus $\rho \equiv \rho_1 + 1$ and we have

$$Rd_g(\ulcorner\tau_1 + 1\urcorner) = Rd_g\ \boldsymbol{S}(\ulcorner\tau_1\urcorner) \stackrel{\text{def}}{=} \boldsymbol{S}\ Rd_g(\ulcorner\tau_1\urcorner) \stackrel{\text{IH}}{=} \boldsymbol{S}(\ulcorner\rho_1\urcorner) = \ulcorner\rho_1 + 1\urcorner.$$

If $\tau \equiv \tau_1 \dot{-} 1$ then $\tau_1 \rhd_1 \rho_1$ for some ρ_1. Thus $\rho \equiv \rho_1 \dot{-} 1$ and we have

$$Rd_g(\ulcorner\tau_1 \dot{-} 1\urcorner) = Rd_g\ \boldsymbol{Pr}(\ulcorner\tau_1\urcorner) \stackrel{\text{def}}{=} \boldsymbol{Pr}\ Rd_g(\ulcorner\tau_1\urcorner) \stackrel{\text{IH}}{=} \boldsymbol{Pr}(\ulcorner\rho_1\urcorner) = \ulcorner\rho_1 \dot{-} 1\urcorner.$$

If $\tau \equiv \mathcal{D}_s(\tau_1,\tau_2,\tau_3)$ then $\tau_1 \rhd_1 \rho_1$ for some ρ_1. Thus $\rho \equiv \mathcal{D}_s(\rho_1,\tau_2,\tau_3)$ and we have

$$\begin{aligned}
&Rd_g(\ulcorner\mathcal{D}_s(\tau_1,\tau_2,\tau_3)\urcorner) = Rd_g\ \boldsymbol{D}_s(\ulcorner\tau_1\urcorner,\ulcorner\tau_2\urcorner,\ulcorner\tau_3\urcorner) \stackrel{\text{def}}{=}\\
&\quad = \boldsymbol{D}_s(Rd_g(\ulcorner\tau_1\urcorner),\ulcorner\tau_2\urcorner,\ulcorner\tau_3\urcorner) \stackrel{\text{IH}}{=} \boldsymbol{D}_s(\ulcorner\rho_1\urcorner,\ulcorner\tau_2\urcorner,\ulcorner\tau_3\urcorner) = \ulcorner\mathcal{D}_s(\rho_1,\tau_2,\tau_3)\urcorner.
\end{aligned}$$

Suppose finally that $\tau \equiv f(\tau_1,\dots,\tau_k,\tau_{k+1},\tau_{k+2},\dots,\tau_n)$, where f is either a defined function or an oracle. The number $k < n$ is such that the terms $\tau_1,\dots,\tau_k$ are numerals and τ_{k+1} not. Then $\tau_{k+1} \rhd_1 \rho_{k+1}$ for some ρ_{k+1} and thus $\rho \equiv f(\tau_1,\dots,\tau_k,\rho_{k+1},\tau_{k+2},\dots,\tau_n)$. We have

$$\begin{aligned}
&Rd_g(\ulcorner f(\tau_1,\dots,\tau_k,\tau_{k+1},\tau_{k+2},\dots,\tau_n)\urcorner) =\\
&\quad = Rd_g\big(\ulcorner f\urcorner[\ulcorner\tau_1\urcorner,\dots,\ulcorner\tau_k\urcorner,0] \bullet \ulcorner\tau_{k+1}\urcorner \bullet \ulcorner\tau_{k+2}\urcorner \bullet \cdots \bullet \ulcorner\tau_n\urcorner\big) \stackrel{\text{def}}{=}\\
&\quad = \ulcorner f\urcorner[\ulcorner\tau_1\urcorner,\dots,\ulcorner\tau_k\urcorner,0] \mathbin{\underline{\bullet}} Rd_g(\ulcorner\tau_{k+1}\urcorner) \mathbin{\underline{\bullet}} \ulcorner\tau_{k+2}\urcorner \mathbin{\underline{\bullet}} \cdots \mathbin{\underline{\bullet}} \ulcorner\tau_n\urcorner \stackrel{\text{IH}}{=}\\
&\quad = \ulcorner f\urcorner[\ulcorner\tau_1\urcorner,\dots,\ulcorner\tau_k\urcorner,0] \mathbin{\underline{\bullet}} \ulcorner\rho_{k+1}\urcorner \mathbin{\underline{\bullet}} \ulcorner\tau_{k+2}\urcorner \mathbin{\underline{\bullet}} \cdots \mathbin{\underline{\bullet}} \ulcorner\tau_n\urcorner \stackrel{3.3.4(1)}{=}\\
&\quad = \ulcorner f(\tau_1,\dots,\tau_k,\rho_{k+1},\tau_{k+2},\dots,\tau_n)\urcorner.
\end{aligned}$$

3.3.8 Arithmetization of reductions. The binary iteration of the reduction function $Rd^k_g(t)$ defined by

$$\begin{aligned}
&Rd^0_g(t) = t\\
&Rd^{k+1}_g(t) = Rd^k_g\ Rd_g(t)
\end{aligned}$$

is primitive recursive function in g by Thm. 3.1.24.

Properties 3.3.7(1)(2) generalizes to

$$Rd^k_g(\ulcorner\underline{x}\urcorner) = \ulcorner\underline{x}\urcorner \tag{1}$$

$$\text{for every } \tau\text{, if } \tau \rhd_k \rho \text{ then } Rd^k_g(\ulcorner\tau\urcorner) = \ulcorner\rho\urcorner. \tag{2}$$

Property (1) is proved by a straightforward induction on k. Property (2) is proved by the same induction as follows. The base case is trivial. In the inductive case take any τ such that $\tau \rhd_{k+1} \rho$. Then $\tau \rhd_1 \sigma \rhd_k \rho$ for some σ and we obtain

$$Rd_g^{k+1}(\ulcorner\tau\urcorner) = Rd_g^k\, Rd_g(\ulcorner\tau\urcorner) \overset{3.3.7(2)}{=} Rd_g^k(\ulcorner\sigma\urcorner) \overset{\text{IH}}{=} \ulcorner\rho\urcorner.$$

3.3.9 Theorem. *The following holds for every closed R-term τ:*

$$\tau \rhd_{\leq k} \underline{x} \text{ iff } Rd_g^k(\ulcorner\tau\urcorner) = \ulcorner\underline{x}\urcorner. \tag{1}$$

Proof. Assume $\tau \rhd_{\leq k} \underline{x}$. Then $\tau \rhd_l \underline{x}$ for some $l \leq k$ and we obtain

$$Rd_g^k(\ulcorner\tau\urcorner) \overset{3.1.23(3)}{=} Rd_g^{k-l}\, Rd_g^l(\ulcorner\tau\urcorner) \overset{3.3.8(2)}{=} Rd_g^{k-l}(\ulcorner\underline{x}\urcorner) \overset{3.3.8(1)}{=} \ulcorner\underline{x}\urcorner.$$

The reverse implication:

$$\text{for every closed R-term } \tau \text{, if } Rd_g^k(\ulcorner\tau\urcorner) = \ulcorner\underline{x}\urcorner \text{ then } \tau \rhd_{\leq k} \underline{x},$$

is proved by induction on k. The base case is obvious. In the inductive case take any closed R-term τ such that $Rd_g^{k+1}(\ulcorner\tau\urcorner) = \ulcorner\underline{x}\urcorner$. By the properties 3.3.7(1)(2) of the reduction function Rd_g we get that there is a closed R-term ρ such that $Rd_g(\ulcorner\tau\urcorner) = \ulcorner\rho\urcorner$ and $\tau \rhd_{\leq 1} \rho$. We have $Rd_g^k(\ulcorner\rho\urcorner) = \ulcorner\underline{x}\urcorner$ and thus $\rho \rhd_{\leq k} \underline{x}$ by IH. We can conclude that $\tau \rhd_{\leq k+1} \underline{x}$. □

Course of Values Recursion with Measure

3.3.10 Course of values recursive definitions with measure. Suppose that

$$f(\vec{x}) = \tau[f; \vec{x}] \tag{1}$$

is a regular recursive definition of an n-ary function f with the measure $\mu[\vec{x}]$ which is into the well-order $<$ of natural numbers. The definition (1) is then called the *course of values recursive definition with measure* and can be viewed as a function operator taking all functions applied in τ and μ and yielding the function f.

3.3.11 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under course of values recursive definitions with measure.*

Proof. Let f be defined by the course of values recursive definition

$$f(x_1, \ldots, x_n) = \tau[f; x_1, \ldots, x_n]$$

from the functions $g_1, \ldots, g_k \in \mathcal{F}$ which is regular in the measure $\mu[\vec{x}]$ with all functions applied from $\mathcal{F}$. We wish to show that also $f \in \mathcal{F}$.

We can suppose without loss of generality that τ is a recursive term of the language $\mathcal{L} = \{g_1^{n_1}, \dots, g_k^{n_k}\}$, where $n_1, \dots, n_k$ are respectively the arities of the functions $g_1, \dots, g_k$. The interpretation $\mathfrak{I}$ of $\mathcal{L}$ interpretes the oracles $g_1^{n_1}, \dots, g_k^{n_k}$ by the corresponding functions $g_1, \dots, g_k$. The ternary function $g(n, i, x)$ representing the interpretation $\mathfrak{I}$ is derived in $\mathcal{F}$ by the following explicit definition:

$$
\begin{aligned}
&g(n_1, 1, x_1, \dots, x_{n_1}) = g_1(x_1, \dots, x_{n_1})\\
&g(n_2, 2, x_1, \dots, x_{n_2}) = g_2(x_1, \dots, x_{n_2})\\
&\vdots\\
&g(n_k, k, x_1, \dots, x_{n_k}) = g_k(x_1, \dots, x_{n_k}).
\end{aligned}
$$

We claim that

$$f(x_1, \dots, x_n) = Dc\, Rd_g^{d_{\lambda_n.\tau}(\mu[x_1,\dots,x_n]+1)}(\ulcorner \lambda_n.\tau \urcorner [\ulcorner \underline{x_1} \urcorner, \dots, \ulcorner \underline{x_n} \urcorner, 0]). \qquad (1)$$

Property (1) follows from

$$
\begin{aligned}
&(\lambda_n.\tau)(x_1, \dots, x_n) = y \overset{\text{Thm. 2.3.24}}{\Rightarrow}\\
&(\lambda_n.\tau)(\underline{x_1}, \dots, \underline{x_n}) \rhd_{\le d_{\lambda_n.\tau}(\mu[x_1,\dots,x_n]+1)} \underline{y} \overset{\text{Thm. 3.3.9}}{\Rightarrow}\\
&Rd_g^{d_{\lambda_n.\tau}(\mu[x_1,\dots,x_n]+1)}\big(\ulcorner (\lambda_n.\tau)(\underline{x_1}, \dots, \underline{x_n}) \urcorner\big) = \ulcorner \underline{y} \urcorner \overset{3.3.3(1)}{\Rightarrow}\\
&Dc\, Rd_g^{d_{\lambda_n.\tau}(\mu[x_1,\dots,x_n]+1)}\big(\ulcorner \lambda_n.\tau \urcorner [\ulcorner \underline{x_1} \urcorner, \dots, \ulcorner \underline{x_n} \urcorner, 0]\big) = y
\end{aligned}
$$

by noting that $f = \lambda_n.\tau$.

Now, we can take the identity (1) as a derivation of the function f in $\mathcal{F}$ by noting that the estimating function $d_{\lambda_n.\tau}$ is primitive recursive (see Par. 2.3.23). ◻

3.3.12 Theorem. *Primitive recursive functions in $\mathcal{F}$ are generated from the same initial functions, the successor function, and the predecessor function by explicit definitions and course of values recursive definitions with measure.*

Proof. Let us denote by $\mathcal{G}$ the class of functions generated from $\mathcal{F}$, $x+1$, and $x \dot{-} 1$ by explicit definitions and course of values recursive definitions with measure.

The class PRIMREC($\mathcal{F}$) contains the predecessor function by Par. 3.1.13 and it is closed both under explicit definitions and course of values recursive definitions with measure by Thm. 3.1.5 and Thm. 3.3.11, respectively. Hence PRIMREC($\mathcal{F}$) $\subseteq \mathcal{G}$.

The converse is proved as follows. The class $\mathcal{G}$ is closed under explicit definitions and therefore, by Par. 2.2.11, it contains the identity functions and the zero function and it is closed under composition. Thus it suffices to show the closure under the operator of primitive recursion. Let the function f is obtained from the functions $g, h \in \mathcal{G}$ by primitive recursion:

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

We can derive f in $\mathcal{G}$ by the following course of values recursive definition:

$$f(x,\vec{y}) = \mathcal{D}_s\big(x, h(x \dot{-} 1, f(x \dot{-} 1,\vec{y}),\vec{y}), g(\vec{y})\big)$$

regular in the first argument. □

3.3.13 Generalized course of values recursive definitions with measure. Suppose that

$$f(\vec{x}) = \alpha[f;\vec{x}] \tag{1}$$

is a generalized regular recursive definition of an n-ary function f with the measure $\mu[\vec{x}]$ which is into the well-order $<$ of natural numbers. The definition (1) is then called the *generalized course of values recursive definition with measure* and can be viewed as a function operator taking all functions applied in α and μ and yielding the function f.

Note that the equation (1) is a generalized course of values recursive definition of f with the measure $\mu[\vec{x}]$ iff its translation $f(\vec{x}) = \alpha^\star[f;\vec{x}]$ is a course of values recursive definition of f with the same measure. As a corollary of Thm. 3.3.11 we obtain the following claim.

3.3.14 Theorem. *Primitively recursively closed classes $\mathcal{F}$ are closed under generalized course of values recursive definitions with measure.*

3.3.15 Theorem. *Primitive recursive functions in $\mathcal{F}$ are generated from the same initial functions, the successor function, and the predecessor function by generalized explicit definitions and generalized course of values recursive definitions with measure.*

Proof. Directly from Thm. 3.3.12 using Thm. 3.1.6 and Thm. 3.3.14. □

3.3.16 n-tuples. In the sequel we will need functions and predicates operating over (non-empty) n-tuples of natural numbers. We will code a n-tuple $x_1, \ldots, x_n$, where $n \geq 1$, by the number $(x_1, \ldots, x_n)$. Codes of n-tuples will be called also n-tuples.

The binary predicate $Tp(n, x)$ holding of (non-empty) n-tuples, i.e.

$$Tp(n, x) \leftrightarrow n \geq 1 \wedge \exists x_1 \ldots \exists x_n \, x = (x_1, \ldots, x_n), \tag{1}$$

is defined by (generalized) course of values recursion regular in n with substitution in parameter as a primitive recursive predicate:

$Tp(1, x)$
$Tp(n+2, x_1, x) \leftarrow Tp(n+1, x)$

The ternary *selection* function $[x]^n_i$ selects the i-th element of an n-tuple x, i.e. we have

$$1 \leq i \leq n \rightarrow [x_1, \ldots, x_n]^n_i = x_i. \tag{2}$$

The function $[x]^n_i$ is defined by course of values recursion regular in n with substitution in parameters as a primitive recursive function:

$[x_1]^1_1 = x_1$
$[x_1, x]^{n+2}_1 = x_1$
$[x_1, x]^{n+2}_{i+2} = [x]^{n+1}_{i+1}.$

The ternary *concatenation* function $x \#_n y$ concatenates an n-tuple x and an m-tuple y, i.e. we have

$$(x_1, \ldots, x_n) \#_n (y_1, \ldots, y_m) = x_1, \ldots, x_n, y_1, \ldots, y_m. \tag{3}$$

The function $x \#_n y$ is defined by course of values recursion regular in n with substitution in parameter x as a primitive recursive function:

$x_1 \#_1 y = x_1, y$
$x_1, x \#_{n+2} y = x_1, x \#_{n+1} y.$

3.3.17 Codes of defined recursive function symbols. We claim that there is a binary primitive recursive predicate $Cdf_n(e)$ satisfying

$$Cdf_n(e) \text{ iff } e = \ulcorner \boldsymbol{\lambda}_n \boldsymbol{.} \tau \urcorner \text{ for some defined R-function symbol } \boldsymbol{\lambda}_n \boldsymbol{.} \tau. \tag{1}$$

For that we need some auxiliary functions and predicates.

The predicate $Nms(ts)$ holds if ts is a list of the codes of numerals. The predicate is defined by course of values recursion as a primitive recursive predicate:

$Nms(0)$
$Nms(t, ts) \leftarrow Nm(t) \wedge Nms(ts).$

The ternary predicate $Tm(t, rs, n)$ satisfies for all $n \geq 1$ and for all R-terms $\rho_1, \ldots, \rho_k$ in the recursor $\mathfrak{f}_n$ and in the variables $x_1, \ldots, x_n$:

the predicate $Tm(t, (\ulcorner\rho_1\urcorner, \ldots, \ulcorner\rho_k\urcorner, 0), n)$ holds iff there is a R-term τ in the recursor $\mathfrak{f}_n$ and in the variables $x_1, \ldots, x_n$ such that

$$\ulcorner\tau\urcorner = t \bullet \ulcorner\rho_1\urcorner \bullet \cdots \bullet \ulcorner\rho_k\urcorner.$$

The predicate is defined by course of values recursion on t with substitution in parameters as a primitive recursive predicate:

$$
\begin{aligned}
&Tm(\boldsymbol{x}_i, 0, n) \leftarrow 1 \leq i \leq n\\
&Tm(\mathbf{0}, 0, n)\\
&Tm(\boldsymbol{S}(t), 0, n) \leftarrow Tm(t, 0, n)\\
&Tm(\boldsymbol{Pr}(t), 0, n) \leftarrow Tm(t, 0, n)\\
&Tm(\boldsymbol{D}_s(t_1, t_2, t_3), 0, n) \leftarrow Tm(t_1, 0, n) \land Tm(t_2, 0, n) \land Tm(t_3, 0, n)\\
&Tm(t_1 \bullet t_2, rs, n) \leftarrow\\
&\quad Tm(t_1, (t_2, rs), n) \land Tm(t_2, 0, n) \land \exists e \exists ts\, t_1 = e[ts] \land \neg Nm(t_2)\\
&Tm(t_1 \bullet t_2, rs, n) \leftarrow\\
&\quad Tm(t_1, (t_2, rs), n) \land Tm(t_2, 0, n) \land \neg\exists e \exists ts\, t_1 = e[ts]\\
&Tm(\boldsymbol{f}_m[ts], rs, n) \leftarrow m \geq 1 \land m = n \land Nms(ts) \land L(ts) + L(rs) = m\\
&Tm((\boldsymbol{\lambda}_m\boldsymbol{.}\, t)[ts], rs, n) \leftarrow\\
&\quad m \geq 1 \land Nms(ts) \land L(ts) + L(rs) = m \land Tm(t, 0, m)\\
&Tm(\boldsymbol{g}_i^m[ts], rs, n) \leftarrow m \geq 1 \land Nms(ts) \land L(ts) + L(rs) = m
\end{aligned}
$$

The predicate $Cdf_n(e)$ holding of the codes of n-ary defined recursive function symbols is defined explicitly as a primitive recursive predicate:

$$Cdf_n(e) \leftrightarrow n \geq 1 \land \exists t \leq e(e = \boldsymbol{\lambda}_n\boldsymbol{.}\, t \land \land Tm(t, 0, n)).$$

3.3.18 Auxiliary functions and predicates. The function $Ar(e)$ takes the code e of a R-function symbol f and yields the arity of f, i.e. we have

$$Ar(\ulcorner\mathfrak{f}_n\urcorner) = Ar(\ulcorner\lambda_n.\tau\urcorner) = Ar(\ulcorner g_i^n\urcorner) = n. \tag{1}$$

The function is defined explicitly as a primitive recursive function:

$$
\begin{aligned}
&Ar(\boldsymbol{f}_n) = n\\
&Ar(\boldsymbol{\lambda}_n\boldsymbol{.}\, t) = n\\
&Ar(\boldsymbol{g}_i^n) = n.
\end{aligned}
$$

The binary function $\ulcorner\underline{x}\urcorner^{(n)}$ applies the coding function to each element of an n-tuple x, i.e. we have

$$\ulcorner\underline{x_1, \ldots, x_n}\urcorner^{(n)} = \ulcorner\underline{x_1}\urcorner, \ldots, \ulcorner\underline{x_n}\urcorner. \tag{2}$$

The function $\ulcorner\underline{x}\urcorner^{(n)}$ is defined by course of values recursion regular in n with substitution in parameter as a primitive recursive function:

$\ulcorner \underline{x_1} \urcorner^{(1)} = \ulcorner \underline{x_1} \urcorner$
$\ulcorner \underline{x_1, x} \urcorner^{(n+2)} = \ulcorner \underline{x_1} \urcorner, \ulcorner \underline{x} \urcorner^{(n+1)}$.

The ternary *iteration contraction* function $t \,\underline{\bullet}_n\, rs$ satisfying

$$t \,\underline{\bullet}_n\, (r_1, \ldots, r_n) = t \,\underline{\bullet}\, r_1 \,\underline{\bullet} \cdots \underline{\bullet}\, r_n \tag{3}$$

is defined by course of values recursion regular in rs with substitution in parameter as a primitive recursive function:

$t \,\underline{\bullet}_1\, r = r$
$t \,\underline{\bullet}_{n+2}\, (r, rs) = t \,\underline{\bullet}\, r \,\underline{\bullet}_{n+1}\, rs$

The binary *application* function $e(ts)$ is such that the following holds

$$\ulcorner f(\tau_1, \ldots, \tau_n) \urcorner = \ulcorner f \urcorner (\ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_n \urcorner) \tag{4}$$

for every R-term $f(\tau_1, \ldots, \tau_n)$. We define the application function explicitly as a primitive recursive function:

$$e(ts) = e[0] \,\underline{\bullet}_{Ar(e)}\, ts.$$

3.4 Inside Primitive Recursive Functions

3.4.1 Increasing functions. A unary function f is *increasing* if

$$x < y \to f(x) < f(y)$$

holds. For increasing functions we have:

$$x \le f(x) \tag{1}$$
$$0 < f(0) \to x < f(x). \tag{2}$$

Property (1) is proved by induction on x. The base case is trivial. For the inductive case we get from IH: $x \le f(x) < f(x+1)$ and so $x+1 \le f(x+1)$. For the property (2) we assume $0 < f(0)$ and prove $x < f(x)$ by induction on x. The base case is trivial and in the inductive case we have from IH: $x < f(x) < f(x+1)$ and so $x+1 < f(x+1)$.

Iterations of increasing functions are increasing:

$$x < y \to f^z(x) < f^z(y) \tag{3}$$
$$0 < f(0) \to x < f^{y+1}(x) \tag{4}$$
$$0 < f(0) \land x < y \to f^x(z) < f^y(z). \tag{5}$$

Property (3) is proved by straightforward induction on z. For the property (4) we assume $0 < f(0)$ and prove $x < f^{y+1}(x)$ by induction on y. For the property (5) we assume $0 < f(0)$ and $x < y$ and obtain

$$f^x(z) \overset{(4)}{<} f^{y-x}f^x(z) = f^y(z).$$

3.4.2 Fast increasing functions. We define a sequence of unary primitive recursive functions A_i:

$$A_0(x) = x + 1$$
$$A_{n+1}(x) = A_n^{x+1}(1).$$

We claim that we have:

$$A_{n+1}(0) = A_n(1) \tag{1}$$
$$A_{n+1}(x+1) = A_n A_{n+1}(x). \tag{2}$$

Indeed, for (1) we have $A_{n+1}(0) = A_n^1(1) = A_n(1)$ and for (2) we have $A_{n+1}(x+1) = A_n^{x+2}(1) = A_n A_n^{x+1}(1) = A_n A_{n+1}(x)$.

By straightforward induction on x we can prove:

$$A_1(x) = x + 2 \tag{3}$$
$$A_2(x) = 2 \cdot x + 3 \tag{4}$$
$$A_3(x) = 8 \cdot 2^x - 3. \tag{5}$$

We prove in the following theorem that the functions A_n are very fast increasing functions. The main property of functions A_n is expressed by Thm. 3.4.5.

3.4.3 Theorem. *The functions A_n are monotone in the following sense. Whenever $n_1 \leq n_2$, $k_1 \leq k_2$, and $x \leq y$ holds then also*

$$A_{n_1}^{k_1}(x) \leq A_{n_2}^{k_2}(y) \tag{1}$$

holds. If at least one of the conditions in the assumptions is strict $<$ then we have $A_{n_1}^{k_1}(x) < A_{n_2}^{k_2}(y)$. We also have

$$A_n(x+1) \leq A_{n+1}(x) \tag{2}$$

$$A_n^k(x) < A_{n+1}(k+x). \tag{3}$$

Proof. We prove first

$$x < A_n(x) \tag{4}$$

by induction on n. In the base case we have $x < x+1 = A_0(x)$. The inductive case $x < A_{n+1}(x)$ is proved by induction on x. In the base case we have

$$0 < 1 \overset{\text{outer IH}}{<} A_n(1) \overset{3.4.2(1)}{=} A_{n+1}(0).$$

In the inductive case we have

$$x+1 \overset{\text{inner IH}}{<} A_{n+1}(x)+1 \overset{\text{outer IH}}{\leq} A_n A_{n+1}(x) \overset{3.4.2(2)}{=} A_{n+1}(x+1).$$

We now prove that A_n are increasing functions:

$$\forall x \forall y (x < y \to A_n(x) < A_n(y)) \tag{5}$$

by induction on n. In the base case we get from $x < y$:

$$A_0(x) = x+1 < y+1 = A_0(y).$$

In the inductive case we know from IH that A_n is increasing for which $0 < A_n(0)$ holds by (4). From $x < y$ we then have:

$$A_{n+1}(x) = A_n^{x+1}(1) \overset{3.4.1(5)}{<} A_n^{y+1}(1) = A_{n+1}(y).$$

Property (2) is proved by induction x. In the base case we have $A_n(1) \overset{3.4.2(1)}{=} A_{n+1}(0)$. In the inductive case we have $x+2 \overset{(4)}{\leq} A_n(x+1) \overset{\text{IH}}{\leq} A_{n+1}(x)$ and so

$$A_n(x+2) \overset{(5)}{\leq} A_n A_{n+1}(x) \overset{3.4.2(2)}{=} A_{n+1}(x+1).$$

We now prove the first of the following properties

$$A_n(x) < A_{n+1}(x) \tag{6}$$

$$n_1 < n_2 \to A_{n_1}(x) < A_{n_2}(x) \tag{7}$$

$$n_1 < n_2 \to A_{n_1}^k(x) < A_{n_2}^k(y). \tag{8}$$

by considering two cases. If $x = 0$ we have $A_n(0) \overset{(5)}{<} A_n(1) \overset{3.4.2(1)}{=} A_{n+1}(0)$. If $x = y+1$ we have $y+1 \overset{(4)}{\leq} A_n(y) \overset{(5)}{<} A_n(y+1) \overset{(2)}{\leq} A_{n+1}(y)$ and so

$$A_n(y+1) \overset{(5)}{<} A_n A_{n+1}(y) \overset{3.4.2(2)}{=} A_{n+1}(y+1).$$

Properties (7) and (8) now follow by simple induction proofs.

For the property (1) assume $n_1 \leq n_2$, $k_1 \leq k_2$, $x \leq y$, and obtain:

$$A^{k_1}_{n_1}(x) \overset{3.4.1(3)}{\leq} A^{k_1}_{n_1}(y) \overset{3.4.1(5)}{\leq} A^{k_2}_{n_1}(y) \overset{(8)}{\leq} A^{k_2}_{n_2}(y).$$

It should be obvious that if any of the assumptions is strict $<$ then the corresponding step in the above chain is also strict.

Property (3) is proved by induction on k. In the base case we have $A^0_n(x) \overset{(1)}{<} A^1_{n+1}(x) = A_{n+1}(0+x)$. In the inductive case we have

$$A^{k+1}_n(x) = A_n A^k_n(x) \overset{\mathrm{IH},(1)}{<} A_n A_{n+1}(k+x) \overset{3.4.2(2)}{=} A_{n+1}(k+1+x). \qquad \square$$

3.4.4 Limited functions. For a given increasing function h we say that an n-ary function f is *limited by h* if there is a number k such that for all $\vec{x}$ we have $f(\vec{x}) \leq h^k \max(\vec{x})$.

3.4.5 Theorem. *Every primitive recursive function $f(\vec{x})$ is limited by A_n for some n.*

Proof. By induction on the construction of primitive recursive functions f we prove $\exists n \exists k \forall \vec{x}\, f(\vec{x}) \leq A^k_n \max(\vec{x})$. For the initial functions we have:

$$\begin{aligned} S(x) &= A_0(x) = A^1_0 \max(x) \\ Z(x) &= 0 \leq x = A^0_0(x) \\ I^m_i(\vec{x}) &= x_i \leq \max(\vec{x}) = A^0_0 \max(\vec{x}). \end{aligned}$$

If f is an m-ary function derived by composition from the primitive recursive functions $g, h_1, \ldots, h_p$ where g is p-ary then we get from IH the numbers $n_0, n_1, \ldots, n_p$ as well as $k_0, k_1, \ldots, k_p$ such that for $n = \max(n_0, n_1, \ldots, n_p)$, $k' = \max(k_1, \ldots, k_p)$, and $k = k_0 + k'$ we have:

$$g(\vec{y}) \overset{\mathrm{IH}}{\leq} A^{k_0}_{n_0} \max(\vec{y}) \overset{3.4.3(1)}{\leq} A^{k_0}_n \max(\vec{y}) \tag{1}$$

$$h_i(\vec{x}) \overset{\mathrm{IH}}{\leq} A^{k_i}_{n_i} \max(\vec{x}) \overset{3.4.3(1)}{\leq} A^{k'}_n \max(\vec{x}). \tag{2}$$

We then have

$$f(\vec{x}) = g(h_1(\vec{x}),\dots,h_p(\vec{x})) \overset{(1)}{\leq} A_n^{k_0}\max(h_1(\vec{x}),\dots,h_p(\vec{x})) \overset{(2)3.4.3(1)}{\leq}$$
$$A_n^{k_0}A_n^{k'}\max(\vec{x}) = A_n^k \max(\vec{x}).$$

If the function f is derived by primitive recursion from primitive recursive functions g and h then by IH there are numbers n_1, k_1, n_2, k_2 such that

$$g(\vec{y}) \overset{\text{IH}}{\leq} A_{n_1}^{k_1}\max(\vec{y}) \tag{3}$$
$$h(x,\vec{y},a) \overset{\text{IH}}{\leq} A_{n_2}^{k_2}\max(x,\vec{y},a). \tag{4}$$

We may assume without loss of generality that $k_2 > 0$. By induction on x we prove:

$$f(x,\vec{y}) \leq A_{\max(n_1,n_2)}^{k_2\cdot x+k_1}\max(\vec{y}). \tag{5}$$

In the base case we have:

$$f(0,\vec{y}) = g(\vec{y}) \overset{(3)}{\leq} A_{n_1}^{k_1}\max(\vec{y}) \overset{3.4.3(1)}{\leq} A_{\max(n_1,n_2)}^{k_2\cdot 0+k_1}\max(\vec{y}).$$

In the inductive case we note that, since

$$x + \max(\vec{y}) = A_0^x \max(\vec{y}) \overset{3.4.3(1)}{\leq} A_{\max(n_1,n_2)}^{k_2\cdot x+k_1}\max(\vec{y}) \tag{6}$$

holds, we obtain

$$f(x+1,\vec{y}) = h(x,\vec{y},f(x,\vec{y})) \overset{(4)}{\leq} A_{n_2}^{k_2}\max(x,\vec{y},f(x,\vec{y})) \overset{3.4.3(1),\text{IH}}{\leq}$$
$$A_{n_2}^{k_2}\max(x,\vec{y},A_{\max(n_1,n_2)}^{k_2\cdot x+k_1}\max(\vec{y})) \overset{(6),3.4.3(1)}{\leq}$$
$$A_{n_2}^{k_2}A_{\max(n_1,n_2)}^{k_2\cdot x+k_1}\max(\vec{y}) \overset{3.4.3(1)}{\leq}$$
$$A_{\max(n_1,n_2)}^{k_2}A_{\max(n_1,n_2)}^{k_2\cdot x+k_1}\max(\vec{y}) = A_{\max(n_1,n_2)}^{k_2\cdot(x+1)+k_1}\max(\vec{y}).$$

It is easy to see that for $k > 0$ we have $A_2^k(x) = 2^k \cdot x + 3\cdot(2^k - 1)$ and so there is a k_3 such that

$$k_2 \cdot x + \max(\vec{y}) \leq (k_2+1)\cdot\max(x,\vec{y}) \leq A_2^{k_3}\max(x,\vec{y}) \tag{7}$$

holds. We set $n = \max(n_1,n_2,1)+1$ and $k = k_1 + 1 + k_3$ and obtain:

$$f(x,\vec{y}) \overset{(5)}{\leq} A_{\max(n_1,n_2)}^{k_2\cdot x+k_1}\max(\vec{y}) = A_{\max(n_1,n_2)}^{k_1}A_{\max(n_1,n_2)}^{k_2\cdot x}\max(\vec{y}) \overset{3.4.3(3)}{\leq}$$
$$A_{\max(n_1,n_2)}^{k_1}A_{\max(n_1,n_2)+1}(k_2\cdot x+\max(\vec{y})) \overset{3.4.3(1)}{\leq}$$
$$A_n^{k_1}A_n(k_2\cdot x+\max(\vec{y})) \overset{(7)}{\leq} A_n^{k_1}A_nA_2^{k_3}\max(x,\vec{y}) \overset{3.4.3(1)}{\leq}$$
$$A_n^{k_1+1+k_3}\max(x,\vec{y}) = A_n^k\max(x,\vec{y}).$$ □

+

3.5 Exercises

Primitive Recursion

3.5.1 Exercise. Show that the binary function $|x-y|$ is primitive recursive.

3.5.2 Exercise. Show that the factorial function $x!$ is primitive recursive.

3.5.3 Exercise. Show that the integer square root function $\lfloor\sqrt{x}\rfloor$ satisfying

$$\lfloor\sqrt{x}\rfloor^2 \leq x < (\lfloor\sqrt{x}\rfloor + 1)^2$$

is primitive recursive.

3.5.4 Exercise. Let $f(y, \vec{x})$ be a primitive recursive function. Show that the finite sum $\sum_{y<z} f(y, \vec{x})$ and product $\prod_{y<z} f(y, \vec{x})$ are primitive recursive functions.

3.5.5 Exercise. Show that the binary predicate $x \,|\, y$ holding if x is a divisor of y is primitive recursive.

3.5.6 Exercise. Show that the function $\mathrm{nd}(x)$ counting the number of nonzero divisors of x is primitive recursive. We set $\mathrm{nd}(0) = 0$.

3.5.7 Exercise. Show that the predicate $\mathit{Prime}(x)$ holding of prime numbers is primitive recursive.

3.5.8 Exercise. Show that the function $\pi(x)$ counting the number of primes $p \leq x$ is primitive recursive.

3.5.9 Exercise. Show that the function p_i yielding the i-th prime number is primitive recursive. For instance $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, and $p_3 = 7$.

3.5.10 Exercise. Show that the binary function $\mathrm{ex}_i(x)$ yielding the exponent of p_i in the prime factorization of the number $x \neq 0$, i.e.

$$\mathrm{ex}_i(p_0^{x_0} p_1^{x_1} \ldots p_n^{x_n}) = \begin{cases} x_i & \text{if } i \leq n \\ 0 & \text{otherwise,} \end{cases}$$

is primitive recursive. We set $\mathrm{ex}_i(0) = 0$.

3.5.11 Exercise. We code a finite sequence $x_0, \ldots, x_{n-1}$ of numbers by the number $\langle x_0, \ldots, x_{n-1}\rangle$ defined by

$$\langle x_0, \ldots, x_{n-1}\rangle = \prod_{i<n} p_i^{x_i+1}.$$

In particular, the empty sequence $\emptyset$ is coded by the number 1.

Show that the predicate $\mathrm{SEQ}(x)$ holding of the codes of finite sequences of numbers is primitive recursive.

Show further that the *length* function $\mathrm{lh}(x)$ satisfying

$$\mathrm{lh}(\langle x_0, \ldots, x_{n-1} \rangle) = n$$

is primitive recursive.

Show also that the *subscription* function $(x)_i$ such that

$$i < n \rightarrow (\langle x_0, \ldots, x_{n-1} \rangle)_i = x_i$$

holds is primitive recursive.

Finally show that the binary *concatenation* function $x * y$ satisfying

$$\langle x_0, \ldots, x_{n-1} \rangle * \langle y_0, \ldots, y_{m-1} \rangle = \langle x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1} \rangle$$

is primitive recursive.

Course of Values Recursion

In the subsequent exercises use the method of course of values functions to show that the functions in question are primitive recursive.

3.5.12 Exercise. Show that the function $\mathrm{fib}(n)$ yielding the n-th element of the *sequence of Fibonacci* is primitive recursive. The function has the following course of values recursive definition:

$$\begin{aligned}
&\mathrm{fib}(0) = 0 \\
&\mathrm{fib}(1) = 1 \\
&\mathrm{fib}(n+2) = \mathrm{fib}(n+1) + \mathrm{fib}(n).
\end{aligned}$$

Course of Values Recursion with Measure

3.5.13 Exercise. Define the predicate $Ctm(t)$ holding of the codes of closed R-terms.

Inside Primitive Recursive Functions

4. Arithmetization of Data Structures

See [Vod00] and [KV01].

4.1 Arithmetization of Word Domains

See the paragraphs 1.2.1 - 1.2.8 in [Vod00] and the chapter 16 in [KV01].

4.2 Arithmetization of Finite Sequences

See the paragraphs 1.4.1 - 1.4.6 in [Vod00] and the chapters 12 and 15 in [KV01].

4.3 Arithmetization of Trees

See the paragraphs 1.4.13 - 1.4.23 in [Vod00] and the chapter 13 in [KV01].

4.4 Arithmetization of Symbolic Expressions

See the paragraphs 1.4.24 - 1.4.25 in [Vod00] and the chapter 14 in [KV01].

4.5 Exercises

See the chapters 11-16 in [KV01].

5. Recursive Functions

In Sect. 5.1 we negatively chacterize the class of primitive recursive functions as being strictly less than the class of effectively computable functions. In Sect. 5.2 we discuss recursive definitions of partial functions in the style of Herbrand-Gödel-Kleene equations. Partial functions defined by recursive equations are called *partial recursive* functions and we show that they are effectively computable. The characterization of Kleene of recursive functions by μ-recursive functions is given in Sect. 5.3.

5.1 Beyond Primitive Recursion

For long time it was assumed that the computable functions coincide with primitive recursive functions. In 1928 W. Ackermann [Ack28] gave the first example of a function which is evidently effectively computable but it is not primitive recursive. The function was simplified by R. Péter [Pét35] in 1935 to the function A given in Par. 5.1.2. We will also show that the universal function for primitive recursive functions is effectively computable but not a primitive recursive function.

5.1.1 Lexicographical order. Consider the binary predicate $x <_{\text{lex}} y$ defined by

$$x <_{\text{lex}} y \leftrightarrow x = 0 \land y > 0 \lor \exists x_1 \exists x_2 \exists y_1 \exists y_2 \big(x = x_1, x_2 \land y = y_1, y_2 \land \\ \land (x_1 < y_1 \lor x_1 = y_1 \land x_2 < y_2)\big).$$

The predicate clearly satisfies

$$0 <_{\text{lex}} y_1, y_2$$
$$x_1, x_2 <_{\text{lex}} y_1, y_2 \leftrightarrow x_1 < y_1 \lor x_1 = y_1 \land x_2 < y_2.$$

The predicate is called the *lexicographical* order of natural numbers. It is a well-order.

Ackermann-Péter Function Grows Too Fast

5.1.2 The Ackermann-Péter function. The Ackermann-Péter function $A(n, x)$ is defined by the following recursive definition:

$$A(0,x) = x+1$$
$$A(n+1,0) = A(n,1)$$
$$A(n+1,x+1) = A\big(n, A(n+1,x)\big)$$

regular in the measure $\mu[n,x] \equiv (n,x)$ which is into the lexicographical order $<_{\mathrm{lex}}$ of natural numbers. This is because either the first argument decreases or it stays the same and the second decreases.

Note also that the above clausal definition of the function A is from primitive recursive functions. Primitive recursive functions are effectively computable and thus so is the function A.

5.1.3 The Ackermann-Péter function is not primitive recursive. We will now show that A is not a primitive recursive function. First of all, the function A is connected to the fast increasing functions A_n as follows:

$$\forall x\, A(n,x) = A_n(x). \tag{1}$$

The property is proved by induction on n. In the base case we clearly have $A(0,x) = x+1 = A_0(x)$. In the inductive case take any x and continue by (inner) induction on x. In the base case we have:

$$A(n+1,0) = A(n,1) \overset{\text{outer IH}}{=} A_n(1) \overset{3.4.2(1)}{=} A_{n+1}(0).$$

In the inductive case we have:

$$A(n+1,x+1) = A(n, A(n+1,x)) \overset{\text{IH's}}{=} A_n A_{n+1}(x) \overset{3.4.2(2)}{=} A_{n+1}(x+1).$$

Suppose now that A is a primitive recursive function. For some m and k we then have

$$A(n,x) < A_{m+3} \max(k,n,x) \tag{2}$$

since

$$\begin{aligned}
A(n,x) &\overset{\text{Thm. 3.4.5}}{\leq} A_m^k \max(n,x) \overset{3.4.3(3)}{<} A_{m+1}(k + \max(n,x)) \overset{3.4.3(1)}{\leq} \\
&\leq A_{m+1}(2 \cdot \max(k,n,x)) \overset{3.4.3(1),3.4.2(4)}{<} A_{m+1} A_2 \max(k,n,x) \overset{3.4.3(1)}{\leq} \\
&\leq A_{m+1} A_{m+2} \max(k,n,x) \overset{3.4.2(2)}{=} A_{m+2}(\max(k,n,x)+1) \overset{3.4.3(2)}{\leq} \\
&\leq A_{m+3} \max(k,n,x).
\end{aligned}$$

We now get a contradiction for $n = m+3$ and $x \geq \max(k,n)$:

$$A_{m+3}(x) = A_n(x) \overset{(1)}{=} A(n,x) \overset{(2)}{<} A_{m+3} \max(k,n,x) = A_{m+3}(x).$$

Universal Function for Primitive Recursive Functions is not Primitive Recursive

We will define in Par. 5.1.6 a universal function for the class primitive recursive functions and show that it is effectively computable but not primitive recursive. For that we need to assign indices to primitive recursive functions.

5.1.4 Primitive recursive function symbols. The class PR^n of n-ary *primitive recursive (PR) function symbols* is defined inductively as follows:

- $S \in \mathrm{PR}^1$, $Z \in \mathrm{PR}^1$, and $I_i^n \in \mathrm{PR}^n$ for $1 \le i \le n$,
- if $h \in \mathrm{PR}^m$ and $g_1, \ldots, g_m \in \mathrm{PR}^n$ then $Comp_m^n(h, g_1, \ldots, g_m) \in \mathrm{PR}^n$,
- if $g \in \mathrm{PR}^n$ and $h \in \mathrm{PR}^{n+2}$ then $Rec_{n+1}(g, h) \in \mathrm{PR}^{n+1}$.

We set $\mathrm{PR} = \bigcup_{n \ge 1} \mathrm{PR}^n$.

We interprete n-ary PR-function symbols by n-ary functions. The interpretation $f^{\mathcal{N}}$ of a PR-function symbol f is defined by induction on the structure of PR-function symbols as follows:

- $S^{\mathcal{N}}$ is the successor function $x + 1$,
- $Z^{\mathcal{N}}$ is the zero function $Z(x) = 0$,
- $(I_i^n)^{\mathcal{N}}$ is the identity function $I_i^n(\vec{x}) = x_i$,
- $\big(Comp_m^n(h, g_1, \ldots, g_m)\big)^{\mathcal{N}}$ is the n-ary function defined by composition:

$$\big(Comp_m^n(h, g_1, \ldots, g_m)\big)^{\mathcal{N}}(\vec{x}) = h^{\mathcal{N}}\big(g_1^{\mathcal{N}}(\vec{x}), \ldots, g_m^{\mathcal{N}}(\vec{x})\big), \tag{1}$$

- $\big(Rec_n(g, h)\big)^{\mathcal{N}}$ is the n-ary function defined by primitive recursion:

$$\big(Rec_n(g, h)\big)^{\mathcal{N}}(0, \vec{y}) = g^{\mathcal{N}}(\vec{y}) \tag{2}$$

$$\big(Rec_n(g, h)\big)^{\mathcal{N}}(x + 1, \vec{y}) = h^{\mathcal{N}}\big(x, \big(Rec_n(g, h)\big)^{\mathcal{N}}(x, \vec{y}), \vec{y}\big). \tag{3}$$

It is easy to see that the primitive recursive functions are denoted exactly by PR-symbols, i.e.

$$\mathrm{PRIMREC} = \bigcup_{n \ge 1} \{ f^{\mathcal{N}} \mid f \in \mathrm{PR}^n \}$$

In the sequel we abbreviate $f^{\mathcal{N}}$ to f.

5.1.5 Arithmetization of primitive recursive function symbols. We arithmetize PR-function symbols with the following pair constructors:

$\boldsymbol{S} = 0, 0$	*(successor)*
$\boldsymbol{Z} = 1, 0$	*(zero)*
$\boldsymbol{I}_i^n = 2, n, i$	*(identities)*
$\boldsymbol{Comp}_m^n(h, gs) = 3, n, m, h, gs$	*(composition)*
$g \boldsymbol{,} gs = 4, g, gs$	*(arguments)*
$\boldsymbol{Rec}_n(g, h) = 5, n, g, h.$	*(primitive recursion)*

The arities of constructors are as shown in their definitions. We postulate that the binary constructor $g \boldsymbol{,} gs$ groups to the right and has the same precedence as the pairing function x, y.

We assign to every PR-function symbol f its code $\ulcorner f \urcorner$ inductively on the structure of PR-function symbols:

$$\ulcorner S \urcorner = \boldsymbol{S} \tag{1}$$
$$\ulcorner Z \urcorner = \boldsymbol{Z} \tag{2}$$
$$\ulcorner I_i^n \urcorner = \boldsymbol{I}_i^n \tag{3}$$
$$\ulcorner Comp_m^n(h, g_1, \ldots, g_m) \urcorner = \boldsymbol{Comp}_m^n(\ulcorner h \urcorner, \ulcorner g_1 \urcorner \boldsymbol{,} \ldots \boldsymbol{,} \ulcorner g_m \urcorner) \tag{4}$$
$$\ulcorner Rec_n(g, h) \urcorner = \boldsymbol{Rec}_n(\ulcorner g \urcorner, \ulcorner h \urcorner). \tag{5}$$

5.1.6 Universal function for primitive recursive functions. We define the binary function $\{e\}_p(x)$ by the following recursive definition:

$$\begin{aligned}
&\{\boldsymbol{S}\}_p(x) = x + 1 \\
&\{\boldsymbol{Z}\}_p(x) = 0 \\
&\{\boldsymbol{I}_i^n\}_p(x) = [x]_i^n \\
&\{\boldsymbol{Comp}_m^n(h, gs)\}_p(x) = \{h\}_p\big(\{gs\}_p(x)\big) \\
&\{g \boldsymbol{,} gs\}_p(x) = \{g\}_p(x), \{gs\}_p(x) \\
&\{\boldsymbol{Rec}_n(g, h)\}_p(0, y) = \{g\}_p(y) \\
&\{\boldsymbol{Rec}_n(g, h)\}_p(x + 1, y) = \{h\}_p\big(x, \{\boldsymbol{Rec}_n(g, h)\}_p(x, y), y\big).
\end{aligned}$$

The definition is regular in the measure $\mu[e, x] \equiv (e, x)$ which is into the lexicographical order $<_{\text{lex}}$ of natural numbers. This is because all recursive applications except the one in the last clause the first argument goes down. In the recursive application of the last clause the first argument stays the same and the second argument goes down since $x, y < x + 1, y$ by the property 1.3.2(3) of the pairing function.

Note also that the above clausal definition of the function $\{e\}_p(x)$ is from primitive recursive functions. Primitive recursive functions are effectively computable and thus so is the function $\{e\}_p(x)$.

We claim that $\{e\}_p(x)$ is a universal function for the class of primitive recursive functions, i.e.

$$\text{PRIMREC} = \bigcup_{n \geq 1} \{\lambda x_1, \ldots, x_n.\{e\}_p(x_1, \ldots, x_n) \mid e \in \mathbb{N}\}. \tag{1}$$

Indices of functions w.r.t. to the binary function $\{e\}_p(x)$ are called *primitive recursive (PR) indices*.

For the proof of the inclusion $\subseteq$ in (1) it suffices to show that for every PR-function symbol f its code $\ulcorner f \urcorner$ is the PR-index of the primitive recursive function f, i.e.

$$\forall \vec{x}\, f(\vec{x}) = \{\ulcorner f \urcorner\}_p(\vec{x}).$$

The property is proved by induction on the structure of PR-function symbols. So take any n-ary PR-function symbol f, any n-tuple $\vec{x}$, and continue by the case analysis of f. If $f \equiv Comp^n_m(h, g_1, \dots, g_m)$ then we have

$$
\begin{aligned}
&Comp^n_m(h, g_1, \dots, g_m)(\vec{x}) \overset{5.1.4(1)}{=} h\big(g_1(\vec{x}), \dots, g_m(\vec{x})\big) \overset{\mathrm{IH}}{=} \\
&\quad = \{\ulcorner h \urcorner\}_p\big(\{\ulcorner g_1 \urcorner\}_p(\vec{x}), \dots, \{\ulcorner g_m \urcorner\}_p(\vec{x})\big) \overset{\mathrm{def}}{=} \\
&\quad = \{\ulcorner h \urcorner\}_p\big(\{\ulcorner g_1 \urcorner, \dots, \ulcorner g_m \urcorner\}_p(\vec{x})\big) \overset{\mathrm{def}}{=} \\
&\quad = \{\boldsymbol{Comp}^n_m(\ulcorner h \urcorner, \ulcorner g_1 \urcorner, \dots, \ulcorner g_m \urcorner)\}_p(\vec{x}) \overset{5.1.5(4)}{=} \\
&\quad = \{\ulcorner Comp^n_m(h, g_1, \dots, g_m) \urcorner\}_p(\vec{x}).
\end{aligned}
$$

If $f \equiv Rec_n(g, h)$ then $\vec{x} \equiv z, \vec{y}$ for some z and a non-empty $\vec{y}$. We prove

$$Rec_n(g, h)(z, \vec{y}) = \{\ulcorner Rec_n(g, h) \urcorner\}_p\big(z, (\vec{y})\big)$$

by (inner) induction on z. In the base case we have

$$
\begin{aligned}
&Rec_n(g, h)(0, \vec{y}) \overset{5.1.4(2)}{=} g(\vec{y}) \overset{\text{outer IH}}{=} \{\ulcorner g \urcorner\}_p(\vec{y}) \overset{\mathrm{def}}{=} \\
&\quad = \{\boldsymbol{Rec}_n(\ulcorner g \urcorner, \ulcorner h \urcorner)\}_p\big(0, (\vec{y})\big) \overset{5.1.5(5)}{=} \{\ulcorner Rec_n(g, h) \urcorner\}_p\big(0, (\vec{y})\big).
\end{aligned}
$$

In the inductive case we have

$$
\begin{aligned}
&Rec_n(g, h)(z+1, \vec{y}) \overset{5.1.4(3)}{=} h\big(z, Rec_n(g, h)(z, \vec{y}), \vec{y}\big) \overset{\text{inner IH}}{=} \\
&\quad = h\big(z, \{\ulcorner Rec_n(g, h) \urcorner\}_p\big(z, (\vec{y})\big), \vec{y}\big) \overset{5.1.5(5)}{=} \\
&\quad = h\big(z, \{\boldsymbol{Rec}_n(\ulcorner g \urcorner, \ulcorner h \urcorner)\}_p\big(z, (\vec{y})\big), \vec{y}\big) \overset{\text{outer IH}, 3.3.16(3)}{=} \\
&\quad = \{\ulcorner h \urcorner\}_p\big(z, \{\boldsymbol{Rec}_n(\ulcorner g \urcorner, \ulcorner h \urcorner)\}_p\big(z, (\vec{y})\big), \vec{y}\big) \overset{\mathrm{def}}{=} \\
&\quad = \{\boldsymbol{Rec}_n(\ulcorner g \urcorner, \ulcorner h \urcorner)\}_p\big(z+1, (\vec{y})\big) \overset{5.1.5(5)}{=} \{\ulcorner Rec_n(g, h) \urcorner\}_p\big(z+1, (\vec{y})\big).
\end{aligned}
$$

The remaining cases are straightforward and left to the reader.

The reverse inclusion $\supseteq$ in (1) is proved as follows. We explicitly define the unary function U_e:

$$U_e(x) = \{e\}_p(x)$$

for every e. We first prove by complete induction on e that U_e are primitive recursive functions. If $e = \boldsymbol{S}$ then we have an explicit derivation

$$U_e(x) = x + 1.$$

If $e = \boldsymbol{Z}$ then we have an explicit derivation

$$U_e(x) = 0.$$

If $e = \boldsymbol{I}_i^n$ then we have an explicit derivation

$$U_e(x) = [x]_i^n.$$

If $e = \boldsymbol{Comp}_m^n(e_1, e_2)$ for some e_1 and e_2 then the functions U_{e_1} and U_{e_2} are primitive recursive by IH and we derive U_e as a primitive recursive function by composition:

$$U_e(x) = U_{e_1}\, U_{e_2}(x).$$

If $e = \boldsymbol{Rec}_n(e_1, e_2)$ for some e_1 and e_2 then the functions U_{e_1} and U_{e_2} are primitive recursive by IH and we derive U_e as a primitive recursive unction by course of values recursive definition:

$$U_e(0, y) = U_{e_1}(y)$$
$$U_e(x+1, y) = U_{e_2}(x, U_e(x, y), y).$$

If neither of the above cases applies we have an explicit derivation $U_e(x) = 0$. Now suppose that the number e is an index of the n-ary function f, i.e.

$$\forall x_1 \ldots \forall x_n\, f(x_1, \ldots, x_n) = \{e\}_p(x_1, \ldots, x_n).$$

We can derive f as a primitive recursive function by the following explicit definition:

$$f(x_1, \ldots, x_n) = U_e(x_1, \ldots, x_n).$$

This proves the inclusion $\supseteq$ in (1).

5.1.7 Universal function for primitive recursive functions is not primitive recursive. Suppose that $\{e\}_p(x)$ is a primitive recursive function. Then also the explicitly defined unary function $f(x) = \{x\}_p(x) + 1$ is primitive recursive. Let e be one of its indices. We obtain contradiction by

$$f(e) \stackrel{\text{def}}{=} \{e\}_p(e) + 1 \stackrel{\text{index}}{=} f(e) + 1.$$

We thus have in the function $\{e\}_p(x)$ an effectively computable function which is not primitive recursive.

5.1.8 The graph of $\{e\}_p(x)$ is not primitive recursive. We claim that the graph $\{e\}_p(x) = y$ of the function $\{e\}_p(x)$ is not primitive recursive. Suppose that the predicate is primitive recursive. Then the predicate $R(x)$ explicitly defined by

$$R(x) \leftrightarrow \{x\}_p(x) \neq 1 \tag{1}$$

is primitive recursive. Let e be a primitive recursive index of the characteristic function R_* of the predicate R. By 5.1.6(1) we have

$$R_*(x) = \{e\}_p(x). \tag{2}$$

We obtain a contradiction as follows:

$$\{e\}_p(e) \neq 1 \stackrel{(1)}{\Leftrightarrow} R(e) \Leftrightarrow R_*(e) = 1 \stackrel{(2)}{\Leftrightarrow} \{e\}_p(e) = 1.$$

5.2 Recursive Functions

5.2.1 Partial recursive functions. The class of *partial recursive functions* is generated from the successor function $x+1$ and from the predecessor function $x \dot{-} 1$ by explicit and recursive definitions of partial functions. A *recursive function* is a partial recursive function which is total.

We denote by REC and PREC respectively the class of recursive functions and the class partial recursive functions. We denote by REC($\mathcal{F}$) and PREC($\mathcal{F}$) respectively recursive functions and partial recursive functions in the class $\mathcal{F}$. Clearly we have REC = REC($\emptyset$) and PREC = PREC($\emptyset$).

5.2.2 Theorem. *Recursively closed classes $\mathcal{F}$ are primitively recursively closed.*

Proof. The class $\mathcal{F}$ is closed under explicit definitions and therefore, by Thm. 2.2.3, it contains the identity functions I_i^n and the zero function Z, and it is closed under composition of functions. Thus it suffices to show the closure under the operator of primitive recursion. Let the function f is obtained from the functions $g, h \in \mathcal{F}$ by primitive recursion:

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

We can derive f in $\mathcal{F}$ by the following recursive definition:

$$f(x,\vec{y}) = \mathcal{D}_s\big(x, h(x \dot{-} 1, f(x \dot{-} 1,\vec{y}),\vec{y}), g(\vec{y})\big)$$

regular in the first argument. □

5.2.3 Theorem. *Recursively closed classes $\mathcal{F}$ are closed under explicit definitions of predicates with bounded formulas, under definitions of functions with bounded minimalization, and under the iteration of functions.*

Proof. It follows directly from Thm. 5.2.2 by Thm. 3.1.20, Thm. 3.1.20, and Thm. 3.1.24. □

5.2.4 Theorem. *Recursively closed classes $\mathcal{F}$ are closed under generalized explicit definitions and generalized regular recursive definitions of functions.*

Proof. Directly from Thm. 5.2.2 by Thm. 3.1.6 and Thm. 3.3.14. □

5.2.5 Theorem. *Primitive recursive functions are a proper subclass of recursive functions.*

Proof. We have PRIMREC $\subseteq$ REC by Thm. 5.2.2. The Ackermann-Péter function A introduced in the previous section is recursive by Thm. 5.2.4 but not primitive recursive by Par. 5.1.3. Consequently PRIMREC $\subset$ REC. □

5.2.6 Theorem. *Primitive recursive predicates are a proper subclass of recursive predicates.*

Proof. We have $\mathrm{PRIMREC}_* \subseteq \mathrm{REC}_*$ by Thm. 5.2.5. The universal function $\{e\}_p(x)$ for primitive recursive functions is recursive by Thm. 5.2.4 and so is its graph $\{e\}_p(x) = y$ by Thm. 5.2.3. By Par. 5.1.8. the graph of $\{e\}_p(x)$ is not primitive recursive and hence $\mathrm{PRIMREC}_* \subset \mathrm{REC}_*$. ☐

5.2.7 Operator of minimalization. For every $n \geq 1$ the operator of *(unbounded) minimalization* takes an $(n+1)$-ary function g and yields an n-ary function f satisfying:

$$f(\vec{x}) \asymp y \leftrightarrow g(y, \vec{x}) \asymp 1 \wedge \forall z < y \exists v (g(z, \vec{x}) \asymp v \wedge v \neq 1). \tag{1}$$

The partial function f defined by (1) is such that $f(\vec{x})$ is the smallest number y such that $g(y, \vec{x}) \asymp 1$ and for every $z < y$ we have $g(z, \vec{x})\downarrow$. The application $f(\vec{x})$ is undefined if there is no such number.

In the sequel we abbreviate (1) to

$$f(\vec{x}) \simeq \mu_y[g(y, \vec{x}) \simeq 1].$$

The minimalization (1) is *regular* if g is total and $\forall \vec{x} \exists y\, g(y, \vec{x}) = 1$. Clearly, regular minimalizations yield (total) functions. In the sequel we abbreviate the regular minimalization (1) to

$$f(\vec{x}) = \mu_y[g(y, \vec{x}) = 1].$$

5.2.8 Theorem. *Recursively closed classes $\mathcal{F}$ are closed under the operator of minimalization.*

Proof. Let f be defined by the minimalization $f(\vec{x}) \simeq \mu_y[g(y, \vec{x}) \simeq 1]$ of $g \in \mathcal{F}$. We recursively define in $\mathcal{F}$ an auxiliary partial function h by:

$$h(y, \vec{x}) \simeq \mathcal{D}_s\big(g(y, \vec{x}) =_* 1, y, h(y+1, \vec{x})\big).$$

The partial function h satisfies:

$$\begin{aligned} h(y, \vec{x}) \asymp z \leftrightarrow y \leq z \wedge g(z, \vec{x}) \asymp 1 \wedge \\ \forall z_1 \big(y \leq z_1 < z \rightarrow \exists v (g(z, \vec{x}) \asymp v \wedge v \neq 1)\big). \end{aligned}$$

Consequently $f(\vec{x}) \simeq h(0, \vec{x})$ and thus we can take the last identity as explicit definition of $f \in \mathcal{F}$. ☐

5.2.9 Definitions by minimalization. Definitions of partial functions by *(unbounded) minimalization* are of a form

$$f(\vec{x}) \asymp y \leftrightarrow \phi[\vec{x}, y] \wedge \forall z < y\, \neg\phi[\vec{x}, z], \tag{1}$$

where $\phi[\vec{x}, y]$ is a bounded formula with at most the indicated variables free and without any application of f. The function f defined by (1) is such that $f(\vec{x})$ is the smallest number y such that $\phi[\vec{x}, y]$ holds. The application $f(\vec{x})$ is undefined if there is no such number.

In the sequel we abbreviate (1) to

$$f(\vec{x}) \simeq \mu_y\big[\phi[\vec{x}, y]\big].$$

The minimalization (1) is *regular* if $\forall\vec{x}\exists y\phi[\vec{x}, y]$. Clearly, definitions by regular minimalization define (total) functions. In the sequel we abbreviate the regular minimalization (1) to

$$f(\vec{x}) = \mu_y\big[\phi[\vec{x}, y]\big].$$

5.2.10 Theorem. *Every class $\mathcal{F}$ closed under explicit definitions of predicates and the operator of minimalization is closed under definitions of partial functions with minimalization.*

Proof. Suppose that f is defined by the minimalization $f(\vec{x}) \simeq \mu_y\big[\phi[\vec{x}, y]\big]$ from the functions and predicates from $\mathcal{F}$. We can explicitly define f in $\mathcal{F}$ by

$$\begin{aligned} R(y, \vec{x}) &\leftrightarrow \phi[\vec{x}, y] \\ f(\vec{x}) &\simeq \mu_y[R_*(y, \vec{x}) = 1]. \end{aligned}$$

□

5.2.11 Theorem. *Recursively closed classes $\mathcal{F}$ are closed under definitions of partial functions with minimalization.*

Proof. It follows directly from Thm. 5.2.10 by Thm. 5.2.3 and Thm. 5.2.8.

□

5.2.12 Kleene's T**-predicates.** For every $n \geq 1$ we define the $(n+2)$-ary predicate $\mathrm{T}_n(e, \vec{x}, y)$ explicitly as a primitive recursive predicate:

$$\mathrm{T}_n(e, \vec{x}, y) \leftrightarrow \mathit{Cdf}_n(e) \land \exists k \leq y \exists z \leq y\big(y = k, z \land \mathit{Rd}^k\, e\big(\ulcorner\underline{(\vec{x})}\urcorner^{(n)}\big) = \ulcorner\underline{z}\urcorner\big).$$

From the next lemma we will see that the predicate $\mathrm{T}_n(e, \vec{x}, y)$ holds if e is the code of some n-ary defined recursive function symbol $\lambda_n.\tau$ and $y = k, z$ for some k and z such that $(\lambda_n.\tau)(\underline{\vec{x}}) \rhd_{\leq k} \underline{z}$. The reader will note that we have also $\ulcorner\underline{(\vec{x})}\urcorner^{(n)} = \ulcorner\underline{x_1}\urcorner, \ldots, \ulcorner\underline{x_n}\urcorner$ in the definition of T_n.

The predicates $\mathrm{T}_n(e, \vec{x}, y)$ are called the *Kleene's* T*-predicates*. In the sequel we will often abbreviate $\mathrm{T}_1(e, x, y)$ to $\mathrm{T}(e, x, y)$.

By $U(y)$ below we denote the second projection of the pairing function which is primitive recursive by Par. 3.2.6. That is, the function $U(y)$ satisfies

$$\begin{aligned} U(0) &= 0 \\ U(v, w) &= w. \end{aligned}$$

5.2.13 Lemma. *For every $n \geq 1$, we have*

$$(\lambda_n.\tau)(\underline{\vec{x}}) \triangleright_{\leq k} \underline{y} \leftrightarrow \mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, k, y). \tag{1}$$

Proof. Property (1) follows from

$$\begin{aligned}
&(\lambda_n.\tau)(\underline{\vec{x}}) \triangleright_{\leq k} \underline{y} \overset{\text{Thm. 3.3.9}}{\Leftrightarrow} \\
&Rd^k\big(\ulcorner (\lambda_n.\tau)(\underline{\vec{x}})\urcorner\big) = \ulcorner \underline{y} \urcorner \overset{3.3.18(2)(4)}{\Leftrightarrow} \\
&Rd^k \ulcorner \lambda_n.\tau \urcorner\big(\ulcorner \underline{(\vec{x})} \urcorner^{(n)}\big) = \ulcorner \underline{y} \urcorner \overset{3.3.17(1)}{\Leftrightarrow} \\
&Cdf_n(\ulcorner \lambda_n.\tau \urcorner) \wedge Rd^k \ulcorner \lambda_n.\tau \urcorner\big(\ulcorner \underline{(\vec{x})} \urcorner^{(n)}\big) = \ulcorner \underline{y} \urcorner \overset{\text{def}}{\Leftrightarrow} \\
&\mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, k, y).
\end{aligned}$$

□

5.2.14 Lemma. *For every $n \geq 1$, we have*

$$(\lambda_n.\tau)(\vec{x}) \asymp y \leftrightarrow \exists k\, \mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, k, y). \tag{1}$$

Proof. Directly from Thm. 2.3.14 and Lemma 5.2.13 since we have

$$(\lambda_n.\tau)(\vec{x}) \asymp y \Leftrightarrow \exists k\, (\lambda_n.\tau)(\underline{\vec{x}}) \triangleright_{\leq k} \underline{y} \Leftrightarrow \exists k\, \mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, k, y).$$

□

5.2.15 Lemma. *For every $n \geq 1$, we have*

$$(\lambda_n.\tau)(\vec{x}) \simeq U\, \mu_y[\mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, y)]. \tag{1}$$

Proof. We first prove the implication

$$U\, \mu_y[\mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, y)] \asymp y \rightarrow (\lambda_n.\tau)(\vec{x}) \asymp y. \tag{2}$$

So suppose that $U\, \mu_y[\mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, y)] \asymp y$. The minimalization is regular and thus, from the definition of T_n and U, we obtain $\mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, k, y)$ for some k. Now we apply 5.2.14(1) and get $(\lambda_n.\tau)(\vec{x}) \asymp y$.

Suppose now that $(\lambda_n.\tau)(\vec{x}) \asymp y$. By 5.2.14(1) we have $\mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, k, y)$ for some k and thus there is a number z such that $U\, \mu_y[\mathrm{T}_n(\ulcorner \lambda_n.\tau \urcorner, \vec{x}, y)] \asymp z$ holds. By 5.2.15(2) $(\lambda_n.\tau)(\vec{x}) \asymp z$ and thus $z = y$ from the uniqueness property of the graphs of terms. □

5.2.16 Lemma. *Partial function is recursive iff if it is denoted by some defined recursive function symbol.*

Proof. Let $\mathcal{G}$ be the class of all partial functions denoting by defined recursive function symbols. We wish to prove that PREC $= \mathcal{G}$.

The inclusion $\subseteq$ is proved by showing that the class $\mathcal{G}$ is recursively closed. The initial recursive functions $x+1$ and $x \dot{-} 1$ are in $\mathcal{G}$ since they are denoted by the defined R-function symbols $\lambda_1.(x_1 + 1)$ and $\lambda_1.(x_1 \dot{-} 1)$, respectively.

Suppose now that an n-ary partial function f is obtained by an explicit or recursive definition $f(\vec{x}) \simeq \rho[f;\vec{x}]$ from partial functions from $\mathcal{G}$. Let $h_1,\dots,h_k$ be all partial functions applied in ρ other than f, $x+1$, $x \dot{-} 1$, and D. We obtain a recursive term $\tau[\mathfrak{f}_n;\vec{x}]$ from the term ρ by replacing in it all functions symbols $h_1,\dots,h_k \in \mathcal{G}$ by the corresponding defined recursive function symbols denoting these partial functions and by replacing every constant n by the monadic numeral $\underline{n}$. It should be obvious that the functional equation $f(\vec{x}) \simeq \tau[f;\vec{x}]$ defines the same partial function f. The partial function f is thus denoted by the defined R-function symbol $\lambda_n.\tau$ and hence $f \in \mathcal{G}$.

The converse inclusion that every $\lambda_n.\tau$ denotes a partial recursive function is proved by a straightforward induction on the construction defined recursive function symbols. $\square$

5.2.17 Normal form theorem (Kleene). *For every n-ary partial recursive function f there exists a number e such that*

$$f(\vec{x}) \simeq U\,\mu_y[\mathrm{T}_n(e,\vec{x},y)]. \tag{1}$$

Proof. By Lemma 5.2.16, we have $f = \lambda_n.\tau$ for some defined R-function symbol $\lambda_n.\tau$. Property (1) follows from 5.2.15(1) by taking $\ulcorner\lambda_n.\tau\urcorner$ for e. $\square$

5.2.18 Recursive indices. For every $n \geq 1$ by $\varphi_e^{(n)}$ we denote an n-ary partial function defined by

$$\varphi_e^{(n)}(\vec{x}) \simeq \begin{cases} (\lambda_n.\tau)(\vec{x}) & \text{if } e = \ulcorner\lambda_n.\tau\urcorner \text{ for some } \lambda_n.\tau, \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{1}$$

In other words, if e is the code of an n-ary defined R-function symbol $\lambda_n.\tau$ then $\varphi_e^{(n)} = \lambda_n.\tau$; otherwise $\varphi_e^{(n)} = \emptyset^{(n)}$.

We say that a number e is a *recursive index* (*index* for short) of an n-ary partial function f if $f = \varphi_e^{(n)}$. The index e of the n-ary partial function f is said to be *proper (well-formed)* if $e = \ulcorner\lambda_n.\tau\urcorner$ for some $\lambda_n.\tau$. Note that in such case we have

$$\varphi^{(n)}_{\ulcorner\lambda_n.\tau\urcorner}(\vec{x}) \simeq \tau[\lambda_n.\tau;\vec{x}]. \tag{2}$$

In the sequel we will often abbreviate $\varphi_e^{(1)}(\vec{x})$ to $\varphi_e(\vec{x})$.

5.2.19 Theorem. *For every $n \geq 1$, the $\varphi_e^{(n)}$ is a partial recursive function such that*

$$\varphi_e^{(n)}(\vec{x}) \simeq U\,\mu_y[\mathrm{T}_n(e,\vec{x},y)]. \tag{1}$$

Proof. We consider two cases. If $e = \ulcorner \lambda_n.\tau \urcorner$ for some $\lambda_n.\tau$ then $\varphi_e^{(n)} = \lambda_n.\tau$ is a partial recursive function by Lemma 5.2.16 and the property (1) follows from Lemma 5.2.15. Otherwise, the partial function $\varphi_e^{(n)}$ is the nowhere defined partial function $\emptyset^{(n)}$ which is trivially partial recursive. Note that we have also $\neg \mathrm{T}_n(e, \vec{x}, y)$ for all numbers $\vec{x}$ and y. Consequently, the right hand side of (1) is undefined for all $\vec{x}$. □

5.2.20 Partial enumeration functions. For every $n \geq 1$, the $(n+1)$-ary partial *enumeration* function $\psi^{(n)}$ is defined by

$$\psi^{(n)}(e, \vec{x}) \simeq \varphi_e^{(n)}(\vec{x}). \tag{1}$$

In the sequel we will often abbreviate $\psi^{(1)}(e, \vec{x})$ to $\psi(e, \vec{x})$.

5.2.21 Enumeration theorem (Kleene). *For every $n \geq 1$, the $\psi^{(n)}$ is a partial recursive function enumerating (with repetitions) the class of n-ary partial recursive functions.*

Proof. From 5.2.19(1) and the definition of $\psi^{(n)}$ we get

$$\psi^{(n)}(e, \vec{x}) \simeq U\, \mu_y[\mathrm{T}_n(e, \vec{x}, y)]. \tag{1}$$

We can take (1) as a derivation of $\psi^{(n)}$ as a partial recursive function. It remains to show that $\psi^{(n)}$ enumerates n-ary partial recursive functions, i.e.

$$\mathrm{PREC}^{(n)} = \{\lambda\vec{x}.\psi^{(n)}(e, \vec{x}) \mid e \in \mathbb{N}\}. \tag{2}$$

Suppose that f is an n-ary partial recursive function. Then for some number e we have by the normal form theorem the following:

$$f(\vec{x}) \overset{5.2.17(1)}{\simeq} U\, \mu_y[\mathrm{T}_n(e, \vec{x}, y)] \overset{5.2.19(1)}{\simeq} \varphi_e^{(n)}(\vec{x}) \overset{5.2.20(1)}{\simeq} \psi^{(n)}(e, \vec{x}).$$

This proves the inclusion $\subseteq$ in (2). The reversed inclusion is a straightforward consequence of the first part of the theorem. □

5.2.22 Corollary. *Partial function is recursive iff it has an index.*

Proof. If f is an n-ary partial recursive function then by the enumeration theorem and 5.2.20(1) we have $f = \varphi_e^{(n)}$ for some e. The reverse direction follows from Thm. 5.2.19. □

5.2.23 Total extensions of the partial enumeration functions are not recursive. In this paragraph we are concerned with the existence of non-recursive functions. We already know that there are denumerable many recursive functions and therefore, since there are non-denumerably many functions over $\mathbb{N}$, most functions over $\mathbb{N}$ are not recursive. Can we find a concrete such function?

We give here a positive answer to this question by showing for every $n \geq 1$ that the partial enumeration function $\psi^{(n)}$ cannot be extended to a (total) recursive function. We prove this fact for the case when $n = 1$ and left the proof the general result to the reader.

The proof uses a diagonal argument. So let $f(e,x)$ be a (total) binary function which is an extension the partial enumeration function $\psi(e,x)$, i.e.

$$\psi(e,x) \asymp y \to f(e,x) = y. \tag{1}$$

We claim that the function f is not recursive. Suppose by contradiction that the function is recursive. Then the unary function $g(x)$ explicitly defined by

$$g(x) = f(x,x) + 1 \tag{2}$$

is recursive. Let e be an index of g. Note that we then have

$$g(e) \overset{\text{index}}{=} \varphi_e(e) \overset{5.2.20(1)}{\simeq} \psi(e,e) \tag{3}$$

and thus $\psi(e,e)$ is defined. We further obtain

$$f(e,e) \overset{(1)}{\simeq} \psi(e,e) \overset{(3)}{\simeq} g(e) \overset{(2)}{=} f(e,e) + 1.$$

We thus have $f(e,e) \simeq f(e,e) + 1$ and since f is total $f(e,e) = f(e,e) + 1$. Contradiction.

We have shown that total extensions of the partial enumeration function $\psi(e,x)$ are not recursive functions. In particular, its completion which is a binary function defined by

$$f(e,x) = \begin{cases} y & \text{if } \psi(e,x) \asymp y \\ 0 & \text{otherwise} \end{cases}$$

is not recursive.

5.2.24 Graphs of the partial enumeration functions are not recursive. We claim that, for every $n \geq 1$, the graph $\psi^{(n)}(e,\vec{x}) \asymp y$ of the partial enumeration function $\psi^{(n)}$ is not recursive. For simplicity we prove this for the case when $n = 1$ and left the general case as an exercise.

Suppose by contradiction that the predicate $\psi(e,x) \asymp y$ is recursive. Then the unary predicate $R(x)$ explicitly defined by

$$R(x) \leftrightarrow \psi(x,x) \not\asymp 1 \tag{1}$$

is recursive. Let e be an index of the characteristic function R_* of R. We obtain a contradiction as follows:

$$\psi(e,e) \asymp 1 \overset{5.2.20(1)}{\Leftrightarrow} \varphi_e(e) \asymp 1 \overset{\text{index}}{\Leftrightarrow} R_*(e) = 1 \Leftrightarrow R(e) \overset{(1)}{\Leftrightarrow} \psi(e,e) \not\asymp 1.$$

5.2.25 Indices of identity functions. The binary function $\ulcorner I^n_i \urcorner$ yields an index of the n-ary identity function $I^n_i(\vec{x}) = x_i$, i.e.

$$\varphi^{(n)}_{\ulcorner I^n_i \urcorner}(\vec{x}) = I^n_i(\vec{x}). \tag{1}$$

The function $\ulcorner I^n_i \urcorner$ is defined explicitly as a primitive recursive function:

$$\ulcorner I^n_i \urcorner = \boldsymbol{\lambda}_n \boldsymbol{.}\, \boldsymbol{x}_i.$$

Property (1) follows from

$$\varphi^{(n)}_{\ulcorner I^n_i \urcorner}(\vec{x}) \stackrel{\text{def}}{\simeq} \varphi^{(n)}_{\boldsymbol{\lambda}_n \boldsymbol{.}\boldsymbol{x}_i}(\vec{x}) \simeq \varphi^{(n)}_{\ulcorner \lambda_n . x_i \urcorner}(\vec{x}) \stackrel{5.2.18(2)}{\simeq} x_i = I^n_i(\vec{x}).$$

In the sequel we abbreviate $\ulcorner I^1_1 \urcorner$ to $\ulcorner I \urcorner$.

5.2.26 Indices of constant functions. The binary function $\ulcorner C^n_m \urcorner$ yields an index of the n-ary constant function $C^n_m(\vec{x}) = m$, i.e.

$$\varphi^{(n)}_{\ulcorner C^n_m \urcorner}(\vec{x}) = C^n_m(\vec{x}). \tag{1}$$

The function $\ulcorner C^n_m \urcorner$ is defined explicitly as a primitive recursive function:

$$\ulcorner C^n_m \urcorner = \boldsymbol{\lambda}_n \boldsymbol{.}\, \ulcorner \underline{m} \urcorner.$$

Property (1) follows from

$$\varphi^{(n)}_{\ulcorner C^n_m \urcorner}(\vec{x}) \stackrel{\text{def}}{\simeq} \varphi^{(n)}_{\boldsymbol{\lambda}_n \boldsymbol{.} \ulcorner \underline{m} \urcorner}(\vec{x}) \simeq \varphi^{(n)}_{\ulcorner \lambda_n . \underline{m} \urcorner}(\vec{x}) \stackrel{5.2.18(2)}{\simeq} m = C^n_m(\vec{x}).$$

In the sequel we abbreviate $\ulcorner C^1_m \urcorner$ to $\ulcorner C_m \urcorner$ and $\ulcorner C_0 \urcorner$ to $\ulcorner Z \urcorner$.

5.2.27 Indices of nowhere defined partial functions. We wish to define an unary function $\ulcorner \emptyset^{(n)} \urcorner$ yielding a *proper* index of the n-ary nowhere defined partial functions $\emptyset^{(n)}$, i.e.

$$\varphi^{(n)}_{\ulcorner \emptyset^{(n)} \urcorner}(\vec{x}) \simeq \emptyset^{(n)}(\vec{x}). \tag{1}$$

First note that $\lambda_n . \mathfrak{f}_n(x_1, \ldots, x_n)$ is a defined recursive function symbol denoting $\emptyset^{(n)}$. For that purpose we define an auxiliary binary function $(\boldsymbol{x}_i, \ldots, \boldsymbol{x}_n)$ by the following course of values recursive definition with measure $n \dot{-} i$ as a primitive recursive function:

$$\begin{aligned}
&(\boldsymbol{x}_i, \ldots, \boldsymbol{x}_n) = \boldsymbol{x}_n \leftarrow i = n\\
&(\boldsymbol{x}_i, \ldots, \boldsymbol{x}_n) = \boldsymbol{x}_i, (\boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_n) \leftarrow i < n.
\end{aligned}$$

The function $\ulcorner \emptyset^{(n)} \urcorner$ is then defined explicitly as a primitive recursive function:

$$\ulcorner \emptyset^{(n)} \urcorner = \boldsymbol{\lambda}_n \boldsymbol{.}\, \boldsymbol{f}_n(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n).$$

Property (1) follows from

$$\ulcorner \emptyset^{(n)} \urcorner \stackrel{\text{def}}{=} \boldsymbol{\lambda}_n \boldsymbol{.}\, \boldsymbol{f}_n(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \stackrel{3.3.18(4)}{=} \ulcorner \lambda_n . \mathfrak{f}_n(x_1, \ldots, x_n) \urcorner.$$

In the sequel we abbreviate $\ulcorner \emptyset^{(1)} \urcorner$ to $\ulcorner \emptyset \urcorner$.

5.2.28 Turning recursive indices into proper. The binary function $e^{(n)}$ takes an index e of an n-ary partial recursive function and yields a proper index of the same partial function, i.e. we have

$$Cdf_n(e^{(n)}) \tag{1}$$

$$\varphi^{(n)}_{e^{(n)}}(\vec{x}) \simeq \varphi^{(n)}_e(\vec{x}). \tag{2}$$

The function $e^{(n)}$ is defined explicitly as a primitive recursive function:

$$e^{(n)} = e \leftarrow Cdf_n(e)$$
$$e^{(n)} = \ulcorner \emptyset^{(n)} \urcorner \leftarrow \neg Cdf_n(e).$$

5.2.29 Unary composition. The binary function $e_1 \circ e_2$ takes indices e_1 and e_2 of unary partial recursive functions f_1 and f_2 respectively and yields an index of its unary composition: $f(x) = f_1\, f_2(x)$, i.e. we have

$$\varphi_{e_1 \circ e_2}(x) \simeq \varphi_{e_1}\, \varphi_{e_2}(x). \tag{1}$$

The function $e_1 \circ e_2$ is defined explicitly as a primitive recursive function:

$$e_1 \circ e_2 = \boldsymbol{\lambda}_1 \boldsymbol{.}\, e_1^{(1)}\big(e_2^{(1)}(\boldsymbol{x}_1)\big).$$

Property (1) is proved as follows. By 5.2.28(1) there are unary defined recursive function symbols f_1 and f_2 such that

$$\ulcorner f_1 \urcorner = e_1^{(1)} \wedge \ulcorner f_2 \urcorner = e_2^{(1)} \tag{2}$$

holds. We then have

$$e_1 \circ e_2 \overset{\text{def},(2)}{=} \boldsymbol{\lambda}_1 \boldsymbol{.} \ulcorner f_1 \urcorner\big(\ulcorner f_2 \urcorner(\boldsymbol{x}_1)\big) \overset{3.3.18(4)}{=} \ulcorner \lambda_1 . f_1\, f_2(x_1) \urcorner. \tag{3}$$

Property (1) follows now from

$$\varphi_{e_1 \circ e_2}(x) \overset{(3)}{\simeq} \varphi_{\ulcorner \lambda_1 . f_1\, f_2(x_1) \urcorner}(x) \overset{(2)}{\simeq} f_1\, f_2(x) \overset{(1)}{\simeq} \varphi_{\ulcorner f_1 \urcorner}\, \varphi_{\ulcorner f_2 \urcorner}(x) \overset{(2)}{\simeq}$$
$$\simeq \varphi_{e_1^{(1)}}\, \varphi_{e_2^{(1)}}(x) \overset{5.2.28(2)}{\simeq} \varphi_{e_1}\, \varphi_{e_2}(x).$$

5.2.30 Composition. Unary composition is generalized to arbitrary composition as follows. For every $n, m \geq 1$, there is an $(m+1)$-ary primitive recursive function $\mathcal{C}^n_m(e_0, e_1, \ldots, e_m)$ such that

$$\varphi^{(n)}_{\mathcal{C}^n_m(e_0, e_1, \ldots, e_m)}(\vec{x}) \simeq \varphi^{(m)}_{e_0}\big(\varphi^{(n)}_{e_1}(\vec{x}), \ldots, \varphi^{(n)}_{e_m}(\vec{x})\big). \tag{1}$$

The $(m+1)$-ary function $\mathcal{C}^n_m(e_0, e_1, \ldots, e_m)$ is defined explicitly as a primitive recursive function

$$\mathcal{C}^n_m(e_0, e_1, \ldots, e_m) = \boldsymbol{\lambda}_n \boldsymbol{.}\, e^{(m)}\big(e_1^{(n)}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n), \ldots, e_m^{(n)}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)\big).$$

Property (1) is proved similarly as the property 5.2.29(1).

5.2.31 Parametric function. The binary *parametric* function e/x takes an index e of a binary partial recursive function f and a number x and yields an index of the unary partial recursive function g defined by $g(y) \simeq f(x,y)$, i.e. we have

$$\varphi_{e/x}(y) \simeq \varphi_e^{(2)}(x,y). \tag{1}$$

The parametric function is defined explicitly as a primitive recursive function:

$$e/x = \mathcal{C}_2^1(e, \ulcorner C_x \urcorner, \ulcorner I \urcorner).$$

Property (1) follows from

$$\varphi_{e/x}(y) \overset{\text{def}}{\simeq} \varphi_{\mathcal{C}_2^1(e,\ulcorner C_x \urcorner,\ulcorner I \urcorner)}(y) \overset{5.2.30(1)}{\simeq} \varphi_e^{(2)}\big(\varphi_{\ulcorner C_x \urcorner}(y), \varphi_{\ulcorner I \urcorner}(y)\big) \overset{5.2.26(1),5.2.25(1)}{\simeq}$$
$$\simeq \varphi_e^{(2)}\big(C_x(y), I(y)\big) \simeq \varphi_e^{(2)}(x,y).$$

5.2.32 S-m-n theorem (Kleene). *For every $m, n \geq 1$, there exists an $(m+1)$-ary primitive recursive function $s_n^m(e,\vec{x})$ such that*

$$\varphi^{(n)}_{s_n^m(e,\vec{x})}(\vec{y}) \simeq \varphi_e^{(m+n)}(\vec{x},\vec{y}). \tag{1}$$

Proof. The $(m+1)$-ary function $s_n^m(e,\vec{x})$ is defined explicitly as a primitive recursive function

$$s_n^m(e, x_1, \ldots, x_m) = \mathcal{C}_{m+n}^n(e, \ulcorner C_{x_1}^n \urcorner, \ldots, \ulcorner C_{x_m}^n \urcorner, \ulcorner I_1^n \urcorner, \ldots, \ulcorner I_n^n \urcorner).$$

Property (1) is proved similarly as the property 5.2.31(1) of the parametric function. □

5.2.33 Corollary. *For every $m, n \geq 1$, to every $(m+n)$-ary partial recursive function $f(\vec{x},\vec{y})$ there exists an m-ary primitive recursive function $s(\vec{x})$ such that*

$$\varphi^{(n)}_{s(\vec{x})}(\vec{y}) \simeq f(\vec{x},\vec{y}). \tag{1}$$

Proof. Define s explicitly by $s(\vec{x}) = s_n^m(\ulcorner f \urcorner, \vec{x})$, where $\ulcorner f \urcorner$ is an index of f, and use the *s-m-n* theorem. □

5.2.34 Remark. In the sequel, both the theorem and the corollary will be referred to simply as the *s-m-n* theorem. The reader will note that we have

$$s_1^1(e,x) = e/x.$$

5.2.35 Self-reproducing function. In this paragraph we will solve the following question. Does exist a recursive function which produces its own description? More precisely, we wish to find a unary recursive function $\varphi_e(x)$ with yields its own index e for every input x, i.e.

$$\varphi_e(x) = e. \tag{1}$$

Let g be defined explicitly as a primitive recursive function:

$$g(y) = y \circ \ulcorner C_y \urcorner. \tag{2}$$

Let us denote by $\ulcorner g \urcorner$ the index of g. We set $e = g(\ulcorner g \urcorner)$ and obtain (1) from

$$\varphi_e(x) \overset{\text{def}}{\simeq} \varphi_{g(\ulcorner g \urcorner)}(x) \overset{(2)}{\simeq} \varphi_{\ulcorner g \urcorner \circ \ulcorner C_{\ulcorner g \urcorner} \urcorner}(x) \overset{5.2.29(1)}{\simeq} \varphi_{\ulcorner g \urcorner}\, \varphi_{\ulcorner C_{\ulcorner g \urcorner} \urcorner}(x) \overset{5.2.29(1)}{\simeq}$$
$$\simeq \varphi_{\ulcorner g \urcorner}\, C_{\ulcorner g \urcorner}(x) \simeq \varphi_{\ulcorner g \urcorner}(\ulcorner g \urcorner) \overset{\text{index}}{\simeq} g(\ulcorner g \urcorner) \overset{\text{def}}{=} e.$$

5.2.36 Second recursion theorem (Kleene). *For every $n \geq 1$, there exists a unary primitive recursive function $r_n(e)$ such that*

$$\varphi^{(n)}_{r_n(e)}(\vec{x}) \simeq \varphi^{(n+1)}_e(r_n(e), \vec{x}). \tag{1}$$

Proof. First note that there is a primitive recursive function $k_n(e)$ satisfying

$$\varphi^{(n+1)}_{k_n(e)}(y, \vec{x}) \simeq \varphi^{(n+1)}_e(s^1_n(y, y), \vec{x}). \tag{2}$$

Indeed, the property (2) is equivalent to

$$\varphi^{(n+1)}_{k_n(e)}(y, \vec{x}) \simeq \varphi^{(n+1)}_e\big(s^1_n\big(I^{n+1}_1(y, \vec{x}), I^{n+1}_1(y, \vec{x})\big), I^{n+1}_2(y, \vec{x}), \ldots, I^{n+1}_{n+1}(y, \vec{x})\big)$$

and thus it suffices to define $k_n(e)$ by

$$k_n(e) = \mathcal{C}^{n+1}_{n+1}\big(e, \mathcal{C}^{n+1}_2(\ulcorner s^1_n \urcorner, \ulcorner I^{n+1}_1 \urcorner, \ulcorner I^{n+1}_1 \urcorner), \ulcorner I^{n+1}_2 \urcorner, \ldots, \ulcorner I^{n+1}_{n+1} \urcorner\big),$$

where $\ulcorner s^1_n \urcorner$ is an index of the binary primitive recursive function s^1_n.

We now define $r_n(e)$ explicitly as a primitive recursive function by

$$r_n(e) = s^1_n\big(k_n(e), k_n(e)\big).$$

Property (1) follows from

$$\varphi^{(n)}_{r_n(e)}(\vec{x}) \overset{\text{def}}{\simeq} \varphi^{(n)}_{s^1_n(k_n(e), k_n(e))}(\vec{x}) \overset{5.2.32(1)}{\simeq} \varphi^{(n+1)}_{k_n(e)}(k_n(e), \vec{x}) \overset{(2)}{\simeq}$$
$$\simeq \varphi^{(n+1)}_e\big(s^1_n\big(k_n(e), k_n(e)\big), \vec{x}\big) \overset{\text{def}}{\simeq} \varphi^{(n+1)}_e(r_n(e), \vec{x}). \qquad \square$$

5.2.37 Corollary. *For every $(n+1)$-ary partially recursive function $f(e, \vec{x})$ there is number e such that*

$$\varphi_e^{(n)}(\vec{x}) \simeq f(e, \vec{x}). \tag{1}$$

Proof. Define e by $e = r_n(\ulcorner f \urcorner)$, where $\ulcorner f \urcorner$ is an index of f, and use the previous theorem. □

5.2.38 Remark. In the sequel, both the theorem and the corollary will be referred to simply as the second recursion theorem.

5.2.39 Theorem. *Recursive functions are generated from the successor function and the predecessor function by explicit definitions and regular recursive definitions of functions.*

Proof. Let us denote by $\mathcal{G}$ the class of functions generated from $x+1$ and $x \dot{-} 1$ by explicit definitions and regular recursive definitions of functions. The inclusion $\mathcal{G} \subseteq \mathrm{REC}$ is obvious.

The converse is proved as follows. By Thm. 3.3.12 the class $\mathcal{G}$ is primitively recursively closed and thus it contains all primitive recursive functions and predicates. We now show that $\mathcal{G}$ is closed under regular minimalization of functions. Let f be the function obtained from a function $g \in \mathcal{G}$ by regular minimalization:

$$f(\vec{x}) = \mu_y[g(y, \vec{x}) = 1].$$

Consider the following recursive definition of a function $h \in \mathcal{G}$:

$$\begin{aligned}
&h(y, \vec{x}) = y \leftarrow \forall z < y\, g(z, \vec{x}) \neq 1 \wedge g(y, \vec{x}) = 1 \\
&h(y, \vec{x}) = h(y+1, \vec{x}) \leftarrow \forall z < y\, g(z, \vec{x}) \neq 1 \wedge g(y, \vec{x}) \neq 1 \\
&h(y, \vec{x}) = h(y \dot{-} 1, \vec{x}) \leftarrow \exists z < y\, g(z, \vec{x}) = 1.
\end{aligned}$$

The definition is regular in the measure $|f(\vec{x}) - y|$ which is into the standard well-order $<$ of natural numbers since we have

$$\begin{aligned}
&\forall z \leq y\, g(z, \vec{x}) \neq 1 \Rightarrow f(\vec{x}) > y \Rightarrow |f(\vec{x}) - (y+1)| < |f(\vec{x}) - y| \\
&\exists z < y\, g(z, \vec{x}) = 1 \Rightarrow f(\vec{x}) < y \Rightarrow |f(\vec{x}) - (y \dot{-} 1)| < |f(\vec{x}) - y|.
\end{aligned}$$

By a straightforward induction on $y, \vec{x}$ with measure $|f(\vec{x}) - y|$ we can prove that $h(y, \vec{x}) = f(\vec{x})$ holds. Now we can take the identity $f(\vec{x}) = h(0, \vec{x})$ as an explicit definition of f in $\mathcal{G}$.

Now we are in position to prove $\mathrm{REC} \subseteq \mathcal{G}$. By Thm. 5.2.17 every n-ary recursive function f is obtained by one minimalization of the Kleene's T-predicate T_n:

$$f(\vec{x}) = U\, \mu_y[\mathrm{T}_n(e, \vec{x}, y)]. \tag{1}$$

The minimalization is regular and thus we can take (1) as a derivation of f in $\mathcal{G}$ since $\mathcal{G}$ contains the primitive recursive function U and the primitive recursive predicate T_n. □

5.3 μ-Recursive Functions

We have defined in Par. 5.2.1 the class of partial recursive functions by means of explicit and recursive definitions in the style of Herbrand-Gödel-Kleene. This kind of definitions is traditionally considered by mathematicians to be of *meta-mathematical* character in that it *mentions* terms of some language. While definitions based on languages are completely natural to computer scientists who are used deal with programming languages, mathematicians prefer inductive characterizations with minimal reliance on a language. The characterization of Kleene [Kle36, Kle52] by μ-recursive functions is such.

5.3.1 Primitive recursion of partial functions. The operator of *primitive recursion of partial functions* takes an n-ary partial function g and an $(n+2)$-ary partial function h and yields the $(n+1)$-ary partial function f defined by

$$f(0,\vec{y}) \simeq g(\vec{y}) \tag{1}$$

$$f(x+1,\vec{y}) \simeq h(x, f(x,\vec{y}),\vec{y}). \tag{2}$$

It should be clear that there is a unique partial function f satisfying the identities (1) and (2).

5.3.2 Theorem. *Recursively closed classes $\mathcal{F}$ are closed under primitive recursion of partial functions.*

Proof. Let f be a partial function obtained by primitive recursion 5.3.1(1)(2) of $g, h \in \mathcal{F}$. We can derive f in $\mathcal{F}$ by the following recursive definition:

$$f(x,\vec{y}) \simeq \mathcal{D}_s\big(x, h(x \dot{-} 1, f(x \dot{-} 1,\vec{y}),\vec{y}), g(\vec{y})\big). \qquad \square$$

5.3.3 Partial μ-recursive functions. The class of *partial μ-recursive functions* is generated from the successor function $x+1$, the zero function $Z(x) = 0$, and the identity functions $I^n_i(\vec{x}) = x_i$ by the operators of composition, primitive recursion, and minimalization of partial functions. A *μ-recursive function* is a partial μ-recursive function which is total.

We denote by μREC and μPREC respectively μ-recursive functions and partial μ-recursive functions.

5.3.4 Theorem. *Partial recursive functions are exactly partial μ-recursive functions.*

Proof. The class PREC is closed under explicit definitions of partial functions and thus, by Thm. 2.2.3, it contains the zero function and the identity functions, and it is closed under composition of partial functions. Partial recursive functions are closed under primitive recursion of partial functions by

Thm. 5.3.2 and under the operator of minimalization by Thm. 5.2.8. The class PREC is thus μ-recursively closed and hence μPREC $\subseteq$ PREC.

The converse is proved as follows. First note that the class μPREC is primitively recursively closed and thus it contains all primitive recursive functions and predicates. By Thm. 5.2.17 every n-ary partial recursive function f is obtained by one minimalization of the Kleene's T-predicate T_n:

$$f(\vec{x}) \simeq U\,\mu_y[\mathrm{T}_n(e,\vec{x},y)]. \tag{1}$$

The class μPREC contains the primitive recursive function U and the primitive recursive predicate T_n, and thus we can derive f in μPREC by

$$\begin{aligned} g(\vec{x}) &\simeq \mu_y[\mathrm{T}_n(e,\vec{x},y)] \\ f(\vec{x}) &\simeq U\,g(\vec{x}) \end{aligned} \tag{2}$$

since $g \in \mu$PREC by Thm. 3.1.20 and Thm. 5.2.10. □

5.3.5 Theorem. *Recursive functions are generated from the successor function, the zero function, and the identity functions by the operators of composition, primitive recursion, and regular minimalization of functions.*

Proof. Let us denote by $\mathcal{G}$ the class generated from the successor function, the zero function, and the identity functions by the operators of composition, primitive recursion, and regular minimalization of functions. The inclusion $\mathcal{G} \subseteq$ REC follows from Thm. 5.3.4. The reverse inclusion is proved similarly as in the proof of Thm. 5.3.4 by noting that the minimalization in 5.3.4(1) (and therefore in 5.3.4(2) as well) is regular if f is total. □

5.3.6 Discussion. The class of μ-recursive functions could be defined in two ways. An easy way is to assume primitive recursion and define μ-recursive functions as in Par. 5.3.3. This approach makes the initial development of μ-recursive functions easier and it is used by many authors (see, for instance, [BJ74, Dav58, Her65, Kle52]).

Problems occur when one wants to apply the results of the theory of computable functions to mathematical logic and formal (Peano) arithmetic. The language of formal arithmetic contains symbols for addition $+$ and multiplication $\cdot$ but there is no direct provision definitions by primitive recursion. One has to work hard to show that recursive functions can be formally represented by formulas of Peano arithmetic.

In order to pave the way for an easy representation of μ-recursive functions in formal arithmetic many authors (see, for instance, [Sho67]) start with simpler definitions of μ-recursive functions which do not assume the closure under primitive recursion. With a suitable choice of initial functions it is possible to use the power of regular minimalization and prove the closure of μ-recursive functions under primitive recursion. We will take this approach below, but it is possible to skip our alternative development of μ-recursive functions and take Par. 5.3.3 as the definition of μ-recursive functions.

The most difficult part in this development of μ-recursive functions is a temporary coding of course of values sequences. This is needed to show the closure of μ-recursive functions under primitive recursion (see Thm. 5.3.27). After that we can use our pairing and indexing into lists.

One obtains the Cantor's pairing function J as a μ-recursive function easily but the problem is with a μ-recursive definition of the indexing function $(x)_i$. Hence, the most often used approach is to index the codes of finite sequences via Gödel's β-function (see, for instance, [Sho67]) which was introduced in his celebrated proof of incompleteness of formal arithmetic. The β-function relies on the so called Chinese remainder theorem which is a non-trivial theorem concerned with the simultaneous solution of systems of congruences.

In order to make the coding of finite sequences easier many authors have adopted the use of binary (or in general p-ary for primes p) concatenation (see, for instance, [BJ74, Smu61]).

We have chosen to code in the proof of Thm. 5.3.27 finite sequences as numbers in a base $p = 2^{k+1}$. For that we need a μ-recursive derivation of the exponentiation function 2^x. The function can be derived by regular minimalization from its graph $2^x = y$.

The introduction of the graph of exponentiation as a Σ_0-predicate (rudimentary predicate) has a long history. The problem was first formulated by Smullyan in [Smu61] (see also [HP93, Smu92]) and was positively solved by Bennett [Ben62] in his Ph.D. thesis which has greatly influenced the development of Σ_0-predicates. Many Σ_0-derivations of $2^x = y$ were discovered since then but we believe that ours which is given in Thm. 5.3.25 is the simplest one. Σ_0*-predicates* are those which can be obtained by explicit definitions (with bounded quantification) from $x + y$ and $x \cdot y$.

In the development of the class of μ-recursive functions we also use a clever trick of Joan Robinson [Rob49] and define the addition function from the successor and multiplication functions.

We cannot refrain here from pointing out the irony in this controversy of mathematical versus computer science approaches. What seems perfectly natural to a computer scientist, i.e. definitions made by **mentioning** a language of symbols seems to be contrieved to a mathematician who prefers to **use** terms. On the other hand, a computer scientist instictively dislikes the mathematical definition of recursive functions via μ-recursion because he immediately sees the impossibility of feasible computations with such definitions. What is an **effective** process to mathematicians is a horribly **inefficient** one to computer scientists. Computer scientists have developed a good feeling for making their definitions as efficient as they wish by careful formulation of recursive definitions.

5.3.7 Alternative definition of μ-recursive functions. The class of μ_a-*recursive functions* is generated from the identity functions $I_i^n(\vec{x}) = x_i$, the multiplication function $x \cdot y$, and from the characteristic function $x <_* y$ of the less than predicate $x < y$ by composition and regular minimalization of functions.

5.3.8 Remark. By Thm. 2.2.10 and Par. 2.2.11(ii), μ_a-recursive functions are closed under explicit definitions of functions of a form

$$f(\vec{x}) = \tau[\vec{x}], \tag{1}$$

where the term $\tau[\vec{x}]$ is composed only from variables by applications of functions.

μ_a-Recursive functions are closed under definitions of functions with regular minimalization of a form

$$f(\vec{x}) = \mu_y[\tau[\vec{x}, y] = 1], \tag{2}$$

where the term $\tau[\vec{x}, y]$ is composed only from variables by applications of functions, because we can define f as

$$\begin{aligned} g(y, \vec{x}) &= \tau[\vec{x}, y] \\ f(\vec{x}) &= \mu_y[g(y, \vec{x}) = 1]. \end{aligned}$$

Finally, μ_a-recursive functions are closed under definitions of functions with regular minimalization of a form

$$f(\vec{x}) = \mu_y[R(\vec{\tau}[\vec{x}, y])], \tag{3}$$

where the terms $\vec{\tau}[\vec{x}, y]$ are composed only from variables by applications of functions. To see this, it suffices to define f alternatively by

$$f(\vec{x}) = \mu_y[R_*(\vec{\tau}[\vec{x}, y]) = 1].$$

5.3.9 Successor function $x + 1$ is μ_a-recursive. We have $\forall x \exists y\, x < y$ where the number $x + 1$ is the least such. Hence, we define the successor function $x + 1$ as a μ_a-recursive function by regular minimalization:

$$x + 1 = \mu_y[x < y].$$

5.3.10 Unary constant functions are μ_a-recursive. We clearly have $\forall x \exists y\, x < x + 1$ and 0 is the least such number y. Hence, we define the zero function $Z = C_0$ as a μ_a-recursive function by regular minimalization:

$$Z(x) = \mu_y[x < x + 1].$$

We can now define all unary constant functions $C_m(x) = m$ as μ_a-recursive functions by a series of explicit definitions:

$$C_{m+1}(x) = C_m(x) + 1.$$

5.3.11 Remark. By Thm. 2.2.10 and Par. 2.2.11(ii), μ_a-recursive functions are closed under explicit definitions of functions of the form 5.3.8(1), where the term $\tau[\vec{x}]$ is composed only from variables and constants by applications of functions.

We can admit also constants in definitions of μ_a-recursive functions by bounded minimalization of the forms 5.3.8(2)(3). The proof uses similar arguments as those in Par. 5.3.8.

5.3.12 Boolean functions are μ_a-recursive. The boolean functions $\neg_* x$ and $x \wedge_* y$ are μ_a-recursive by explicit definitions:

$$\begin{aligned} \neg_* x &= x <_* 1 \\ x \wedge_* y &= \neg_* \neg_* (x \cdot y). \end{aligned}$$

The remaining boolean functions are defined similarly as μ_a-recursive.

5.3.13 Predicates $x \leq y$ and $x = y$ are μ_a-recursive. The binary predicates $x \leq y$ and $x = y$ are μ_a-recursive by explicit definitions of their characteristic functions:

$$\begin{aligned} x \leq_* y &= \neg_* (y <_* x) \\ x =_* y &= x \leq_* y \wedge_* y \leq_* x. \end{aligned}$$

5.3.14 Discrimination function D is μ_a-recursive. The graph of the discrimination function D satisfies the following obvious property:

$$D(x, y, z) = v \leftrightarrow x = 0 \wedge v = z \vee 0 < x \wedge v = y.$$

We define D as a μ_a-recursive function by regular minimalization:

$$D(x, y, z) = \mu_v[(x =_* 0 \wedge_* v =_* z \vee_* 0 <_* x \wedge_* v =_* y) = 1].$$

5.3.15 Theorem. *μ_a-Recursive functions are closed under explicit definitions of functions and under generalized explicit definitions of functions.*

Proof. μ_a-Recursive functions contain unary constant functions and the function D and so the first part of the theorem follows from Thm. 2.2.10 (see also Par. 2.2.11(i)). The second part follows from Thm. 2.4.19. □

5.3.16 Remark. By Thm. 2.2.7, μ_a-recursive functions are closed under explicit definitions of predicates with quantifier-free formulas.

μ_a-Recursive functions are closed also under definitions of functions with regular minimalization of a form

$$f(\vec{x}) = \mu_y \left[\phi[\vec{x}, y]\right],$$

where the formula $\phi[\vec{x}, y]$ is quantifier-free, because we can define f as

$$R(\vec{x}, y) \leftrightarrow \phi[\vec{x}, y]$$
$$f(\vec{x}) = \mu_y[R(\vec{x}, y)].$$

5.3.17 Lemma. *μ_a-Recursive functions are closed under the operator of bounded minimalization.*

Proof. Let the $(n+1)$-ary function f be defined by bounded minimalization:

$$f(z, \vec{x}) = \mu_{y \leq z}[g(y, \vec{x}) = 1]$$

from a μ_a-recursive function g. We clearly have

$$\forall z \forall \vec{x} \exists y (y \leq z \wedge g(y, \vec{x}) = 1 \vee y > z)$$

and so the auxiliary $(n+1)$-ary function h is defined by regular minimalization as a μ_a-recursive function:

$$h(z, \vec{x}) = \mu_y[y \leq z \wedge g(y, \vec{x}) = 1 \vee y > z].$$

Note that $h(z, \vec{x})$ yields the smallest $y \leq z$ such that $g(y, \vec{x}) = 1$ or $z+1$ if there is none. We now define f by explicit definition

$$f(z, \vec{x}) = h(z, \vec{x}) \leftarrow h(z, \vec{x}) \leq z$$
$$f(z, \vec{x}) = 0 \leftarrow h(z, \vec{x}) > z$$

as a μ_a-recursive function. □

5.3.18 Theorem. *μ_a-Recursive functions are closed under explicit definitions of predicates with bounded formulas and under definitions of functions with bounded minimalization.*

Proof. μ_a-Recursive functions contain the predicate $\leq$ and so the theorem follows from Thm. 2.2.12 and Thm. 2.2.13 by Thm. 5.3.15 and Lemma 5.3.17. □

5.3.19 Addition $x + y$ is μ_a-recursive. We use the following observation by Joan Robinson [Rob49]. If $z > 0$ then we have

$$x + y = z \Leftrightarrow (x+y) \cdot z + x \cdot y \cdot z^2 + 1 = z^2 + x \cdot y \cdot z^2 + 1 \Leftrightarrow$$
$$\Leftrightarrow (x \cdot z + 1) \cdot (y \cdot z + 1) = (x \cdot y + 1) \cdot z^2 + 1.$$

The addition function can be thus derived as a μ_a-recursive function by regular minimalization of its graph:

$$x + y = \mu_z[z = 0 \wedge x = 0 \wedge y = 0 \vee$$
$$z > 0 \wedge (x \cdot z + 1) \cdot (y \cdot z + 1) = (x \cdot y + 1) \cdot z \cdot z + 1].$$

5.3.20 Modified subtraction $x \dot{-} y$ is μ_a-recursive. The binary modified subtraction function $x \dot{-} y$ is μ_a-recursive by bounded minimalization:

$$x \dot{-} y = \mu_{z \le x}[x = y + z].$$

5.3.21 Predicate $x \mid y$ of divisibility is μ_a-recursive. The binary predicate $x \mid y$ of *divisibility* holding if x divides y is μ_a-recursive by explicit definition:

$$x \mid y \leftrightarrow \exists z \le y\, y = x \cdot z.$$

5.3.22 The predicate of beeing a power of two is μ_a-recursive. The predicate $Pow_2(x)$ of x *being a power of two*, i.e. $Pow_2(x) \leftrightarrow \exists y\, x = 2^y$, is μ_a-recursive by explicit definition:

$$Pow_2(x) \leftrightarrow x > 0 \land \forall y \le x(y \mid x \to y = 1 \lor 2 \mid y).$$

5.3.23 Integer division $x \div y$ is μ_a-recursive. We define the integer division $x \div y$ as a μ_a-recursive function by bounded minimalization:

$$x \div y = \mu_{q \le x}[x < (q+1) \cdot y].$$

5.3.24 Remainder function $x \bmod y$ is μ_a-recursive. We define the remainder function $x \bmod y$ as a μ_a-recursive function by explicit definition:

$$x \bmod y = x \dot{-} (x \div y) \cdot y \leftarrow y > 0.$$

5.3.25 Theorem. *The exponentiation function 2^x is μ_a-recursive.*

Proof. See the paragraph 5.2.13 in [Vod00]. □

5.3.26 Bounded indexing function is μ_a-recursive. For the closure of μ_a-recursive functions under primitive recursion we will need to recover digits of numbers represented in the notation with the base 2^k for a given $k \ge 1$. As is well-known, any number x can be uniquely written in such a representation as $x = \sum_i d_i \cdot 2^{k \cdot i}$ with $d_i < 2^k$ for all d_i. We can recover the i-th digit d_i of x by a ternary *bounded indexing function* $(x)_i^{[k]} = d_i$ which can be explicitly defined as a μ_a-recursive function by:

$$(x)_i^{[k]} = x \div 2^{k \cdot i} \bmod 2^k.$$

5.3.27 Theorem. *μ_a-Recursive functions are closed under primitive recursion.*

Proof. Let the $(n+1)$-ary function f be defined by primitive recursion from μ_a-recursive functions g and h:

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

We would like to define as μ_a-recursive the graph $f(x,\vec{y}) = z$ of f. We cannot do this directly but we will able to derive as μ_a-recursive the $(n+3)$-ary predicate $f(x,\vec{y}) \stackrel{k}{=} z$ which can be called the *bounded graph* of f because it is defined to satisfy:

$$f(x,\vec{y}) \stackrel{k}{=} z \leftrightarrow f(x,\vec{y}) = z \land \forall i \leq x\, f(i,\vec{y}) < 2^k.$$

Here the number 2^k *strictly bounds* f *up to* x. The reader will note that the symbol f on the left-hand-side is purely formal. Suppose that we have derived the predicate as μ_a-recursive. For given numbers x and $\vec{y}$ there is clearly a number k such that 2^k strictly bounds f up to x. For every such k there is a unique z such that $f(x,\vec{y}) \stackrel{k}{=} z < 2^k$ holds. We can thus introduce as μ_a-recursive by regular minimalization the $(n+1)$-ary function $\overline{k}(x,\vec{y})$ yielding the smallest such k:

$$\overline{k}(x,\vec{y}) = \mu_k[\exists z < 2^k\, f(x,\vec{y}) \stackrel{k}{=} z].$$

The function f can be then derived as μ_a-recursive by bounded minimalization:

$$f(x,\vec{y}) = \mu_{z<2^{\overline{k}(x,\vec{y})}}[f(x,\vec{y}) \stackrel{\overline{k}(x,\vec{y})}{=} z].$$

It remains to derive the bounded graph of f as μ_a-recursive. If 2^k strictly bounds f up to x we can store the values needed in the computation of $f(x,\vec{y})$ in the course of values sequence s satisfying

$$s \bmod 2^{k\cdot(x+1)} = \sum_{i\leq x} f(i,\vec{y}) \cdot 2^{k\cdot i} < 2^{k\cdot(x+1)}.$$

For that we define an auxiliary $(n+3)$-ary predicate $\mathit{Seq}(k,s,x,\vec{y})$ explicitly as μ_a-recursive by:

$$\mathit{Seq}(k,s,x,\vec{y}) \leftrightarrow (s)^{[k]}_0 = g(\vec{y}) \land \forall i < x\, (s)^{[k]}_{i+1} = h(i, (s)^{[k]}_i, \vec{y}).$$

Clearly, $\mathit{Seq}(k,s,x,\vec{y})$ holds if s the course of values sequence for $f(x,\vec{y})$ provided $f(i,\vec{y}) < 2^k$ holds for all $i \leq x$. We then have

$$f(x,\vec{y}) = z \leftrightarrow \exists k \exists s(\mathit{Seq}(k,s,x,\vec{y}) \land (s)^{[k]}_x = z)$$

and thus the following explicit definition derives the bounded graph of f as a μ_a-recursive predicate:

$$f(x,\vec{y}) \stackrel{k}{=} z \leftrightarrow \exists s < 2^{k\cdot(x+1)}(\mathit{Seq}(k,s,x,\vec{y}) \land (s)^{[k]}_x = z). \qquad \square$$

5.3.28 Theorem. *μ_a-Recursive functions are primitively recursively closed.*

Proof. μ_a-Recursive functions contain the successor function $x + 1$ and the zero function Z by Par. 5.3.9 and Par. 5.3.10, respectively, and are closed under primitive recursion by Thm. 5.3.27. □

5.3.29 Theorem. *Recursive functions are exactly μ_a-recursive functions.*

Proof. It is easy to see that μ_a-recursive functions are a subclass of recursive functions. The converse is proved as follows. By Thm. 5.2.17 every n-ary recursive function f is obtained one regular minimalization of the Kleene's T-predicate T_n:

$$f(\vec{x}) = U\,\mu_y[\mathrm{T}_n(e, \vec{x}, y)].$$

By Thm. 5.3.28 the primitive recursive function U and the primitive recursive predicate T_n are μ_a-recursive. We can derive f as a μ_a-recursive function by

$$\begin{aligned} g(\vec{x}) &= \mu_y[\mathrm{T}_n(e, \vec{x}, y)] \\ f(\vec{x}) &= U\,g(\vec{x}) \end{aligned}$$

since g is μ_a-recursive by Thm. 3.1.20 and Thm. 5.2.10. □

5.3.30 Theorem. *Partial recursive functions is the smallest class containing the identity functions, the multiplication function, and the characteristic function of the less than predicate $x < y$, and closed under composition and minimalization of partial functions.*

Proof. The proof is similar to the proof of Thm. 5.3.4. But now the proof uses Thm. 5.3.28. □

5.4 Inside Recursive Functions

5.5 Exercises

Beyond Primitive Recursion

5.5.1 Exercise. Prove that the graph $A(n, x) = y$ of the Ackermann-Péter function is a primitive recursive predicate.

5.5.2 Exercise. A primitive recursive index e is said to be *proper* if it is the code of some PR-function symbol.

(i) Define the primitive recursive function $Ar(e)$ which yields the arity of the PR-function symbol coded by a proper PR-index e.
(ii) Show that the predicate $Prf(e)$ holding of proper PR-indices is primitive recursive.

5.5.3 Exercise. Show that there is a binary primitive recursive function $\ulcorner C^n_m \urcorner$ such that for every $n \geq 1$ the application $\ulcorner C^n_m \urcorner$ yields the proper PR-index of the n-ary constant function C^n_m.

5.5.4 Exercise. Find a proper PR-index of the case discrimination function $D(x, y, z)$.

5.5.5 Exercise. *Primitive recursive (PR) terms* are formed from variables $x_1, x_2, x_3, \ldots$ and constants n by applications of simple conditionals and PR-function symbols.

We arithmetize PR-terms with following pair constructors:

$$
\begin{aligned}
&\boldsymbol{x}_i = 0, i && \textit{(variables)} \\
&\overline{n} = 1, n && \textit{(constants)} \\
&\boldsymbol{D}_s(t_1, t_2, t_3) = 2, t_1, t_2, t_3 && \textit{(conditional)} \\
&f\boldsymbol{(}ts\boldsymbol{)} = 3, f, ts. && \textit{(PR-functions)}
\end{aligned}
$$

The arities of constructors are as shown in their definitions. We assign to every PR-term τ its code $\ulcorner \tau \urcorner$ inductively on the structure of PR-terms:

$$
\begin{aligned}
\ulcorner x_i \urcorner &= \boldsymbol{x}_i \\
\ulcorner n \urcorner &= \overline{n} \\
\ulcorner \mathcal{D}_s(\tau_1, \tau_2, \tau_3) \urcorner &= \boldsymbol{D}_s(\ulcorner \tau_1 \urcorner, \ulcorner \tau_2 \urcorner, \ulcorner \tau_3 \urcorner) \\
\ulcorner f(\tau_1, \ldots, \tau_n) \urcorner &= \ulcorner f \urcorner \boldsymbol{(} \ulcorner \tau_1 \urcorner \boldsymbol{,} \ldots \boldsymbol{,} \ulcorner \tau_k \urcorner \boldsymbol{)}.
\end{aligned}
$$

Show that the predicate $Prt(t)$ holding of the codes of PR-terms is primitive recursive.

5.5.6 Exercise. Show that there is a binary primitive recursive function $\boldsymbol{\lambda}_n\boldsymbol{.}\, t$ with the following property. If $n \geq 1$ and t is the code of a PR-term $\tau[x_1, \ldots, x_n]$ with all its free variables indicated then $\boldsymbol{\lambda}_n\boldsymbol{.}\, t$ is the proper PR-index of the n-ary function f explicitly defined by

$$f(x_1, \ldots, x_n) = \tau[x_1, \ldots, x_n].$$

In other words, we have

$$
\begin{gathered}
Prf(\boldsymbol{\lambda}_n\boldsymbol{.} \ulcorner \tau \urcorner) \wedge Ar(\boldsymbol{\lambda}_n\boldsymbol{.} \ulcorner \tau \urcorner) = n \\
\{\boldsymbol{\lambda}_n\boldsymbol{.} \ulcorner \tau \urcorner\}_p(x_1, \ldots, x_n) = \tau[x_1, \ldots, x_n].
\end{gathered}
$$

5.5.7 Exercise. Find proper PR-indices of the boolean functions.

5.5.8 Exercise. Find proper PR-indices of addition $x + y$, modified subtraction $x \dot{-} y$, multiplication $x \cdot y$, and exponentiation x^y.

5.5.9 Exercise. Find proper PR-indices of the characteristic functions of the binary predicates $x \leq y$ and $x = y$.

5.5.10 Exercise. Show that there is a binary primitive recursive function $\boldsymbol{\mu}_n[e]$ with the following property. If $n \geq 2$ and e is the proper PR-index of an n-ary function g then $\boldsymbol{\mu}_n[e]$ is the proper PR-index of the n-ary function f obtained from g by bounded minimalization:

$$f(x_1, x_2, \ldots, x_n) = \mu_{y \leq x_1}[g(y, x_2, \ldots, x_n) = 1].$$

In other words, we have

$$Prf(\boldsymbol{\mu}_n[e]) \wedge Ar(\boldsymbol{\mu}_n[e]) = n$$
$$\{\boldsymbol{\mu}_n[e]\}_p(x_1, x_2, \ldots, x_n) = \mu_{y \leq x_1}[\{e\}_p(y, x_2, \ldots, x_n) = 1].$$

5.5.11 Exercise. Find proper PR-indices of integer division $x \div y$ and remainder function $x \bmod y$.

5.5.12 Exercise. Show that there is a primitive recursive function $Iter(e)$ with the following property. If e is the proper index of a unary function $g(x)$then then $Iter(e)$ is the proper PR-index of its iteration $g^n(x)$.

Partial Recursive Functions

μ-Recursive Functions

6. Computable Functions

We present the analysis of A. Turing of what it means for a function over $\mathbb{N}$ to be effectively computable. The analysis leads to the definition of a *Turing computable* partial function as such which is computed by a certain kind of a mechanical machine. We will prove in Thm. 6.2.14 that the Turing computable partial functions coincide with the partial recursive ones and we will discuss in Sect. 6.3 the reasons for the identification of effective computability with Turing computability.

6.1 Turing Machines

6.1.1 Analysis of effective computability by Turing. Turing in [Tur36] has analysed what it means for a (human) computer to compute effectively the value of a function for given arguments. During the computation, which takes only a finite amount of time, the computer uses a finite number of distinct symbols. He can at one time observe only a finite number of occurrences of symbols, although he can use a potentially infinite supply of clean sheets of paper. He can also remember, but only a finite number, of previously observed symbols. During the computation he can use only a finite number of instructions. The computer's mind can be during the computation only in a finite number of states. The instructions specify the conditions under which the computer performs an atomic action.

6.1.2 Turing machines. The analysis of computability by a computer from the preceding paragraph lead Turing to the definition of computing machines, now called a *Turing machines* , which could do the effective computations instead of the human computer. A Turing machine is at any given time in one of finitely many *states* . One of the states is designates as *initial* and one as *terminal*.

The machine is supplied with a linear *tape* which is potentially infinite in both directions and it is divided into *squares*. Each square can *store* (hold) at any given time one of the finitely many *tape symbols*. One of the symbols is *blank* and initially the tape contains only finitely many non-blank symbols.

The Turing machine *scans* at a given time just one square and if its state is not terminal then, depending on the state of the machine and on the currently scanned symbol, it changes the scanned symbol to a possibly different symbol, possibly changes its state, and possibly moves to scan the square immediately to the left or to the right.

A Turing machine can be mathematically defined by two states b and e denoting respectively its initial and terminal states and by a binary *transition* function δ. The transition function is a partial mapping from the cartesian product $Q \times S$ into the cartesian product $Q \times S \times \Delta$, i.e.

$$\delta : Q \times S \mapsto Q \times S \times \Delta$$

and it is finite in the following sense. Each of the sets

- the set of states Q,
- the set of tape symbols S,
- the set of movements $\Delta = \{\blacktriangledown, \blacktriangleleft, \blacktriangleright\}$,

are finite and we have $b \in Q$, $e \in Q$, $\mathbf{0} \in S$, $\mathbf{1} \in S$. We require also that $\langle e, s\rangle \notin \mathrm{dom}(\delta)$ for all $s \in S$.

The tape symbol $\mathbf{0}$ is blank and the contents of a tape will always look as follows:

$$\mathbf{0}^\infty w_1 s_2 \underline{s} s_3 w_2 \mathbf{0}^\infty \tag{1}$$

where the underlined tape symbol s is the currently scanned one, the symbol s_2 is inscribed in the square immediately to the left and the symbol s_3 in the square immediately to the right. Words w_1 and w_2 are finite and are over the set S. Both ends of the tape are inscribed with infinitely many blanks: $\mathbf{0}^\infty$.

The transition function δ controls the operations of the Turing machine A as follows. If A is in the state q scanning the symbol s on (1) then in the next step A will overwrite the symbol s with the symbol s_1, go to the new state q_1, and move, if $d = \blacktriangleleft$, one square left with the new tape situation

$$\mathbf{0}^\infty w_1 \underline{s_2} s_1 s_3 w_2 \mathbf{0}^\infty,$$

move, if $d = \blacktriangleright$, one square rigth with the new tape situation

$$\mathbf{0}^\infty w_1 s_2 s_1 \underline{s_3} w_2 \mathbf{0}^\infty,$$

or stay, if $d = \blacktriangledown$, scanning the same square with the tape situation:

$$\mathbf{0}^\infty w_1 s_2 \underline{s_1} s_3 w_2 \mathbf{0}^\infty.$$

The machine A is *started* in the initial state b and *stops* when it reaches after finitely many steps the terminal state e.

The finiteness of the transition function means that the function can be concretely presented as a finite table of quintuples $\langle q, s, q_1, s_1, d\rangle$ where the states and tape symbols are presented by different concrete symbols, say by decimal numerals.

6.1.3 Partially Turing computable functions. Turing machines can be used to compute word functions over their tape alphabets. We will use them for computation of partial functions over natural numbers in the monadic representation. We say that a Turing machine A *computes* the n-ary partial function f if for all numbers $x_1, x_2, \ldots, x_{n-1}, x_n$ the machine A starting with the tape:

$$\mathbf{0}^\infty \underline{\mathbf{0}}\mathbf{1}^{x_1}\mathbf{0}\mathbf{1}^{x_2}\ldots\mathbf{0}\mathbf{1}^{x_{n-1}}\mathbf{0}\mathbf{1}^{x_n}\mathbf{0}^\infty$$

terminates with the tape:

$$\mathbf{0}^\infty \underline{\mathbf{0}}\mathbf{1}^{y}\mathbf{0}^\infty$$

iff $f(x_1, x_2, \ldots, x_{n-1}, x_n) \asymp y$. Note that an argument x_i or the result y is represented on the tape by a *block* $\mathbf{0}\mathbf{1}^{x_i}$ or $\mathbf{0}\mathbf{1}^{y}$, respectively, with the blank symbol $\mathbf{0}$ called the block *separator*.

A partial function f is *partially Turing computable* if there is a Turing machine computing f; f is *Turing computable* if it is partially Turing computable and total.

6.2 Equivalence of Turing Machines and Recursiveness

Turing Computability Implies Recursiveness

6.2.1 p-ary representation of $\mathbb{N}$. We can turn the functions over word domains given by a p-element alphabet ($p \geq 2$) into functions over natural numbers by means of *p-ary representation* of natural numbers. For that purpose we define p functions $S_0^{(p)}, S_1^{(p)}, \ldots, S_{p-1}^{(p)}$, called *$p$-ary successor* functions, as follows:

$$S_i^{(p)}(x) = p \cdot x + i.$$

It is not difficult to see that every natural number x has a unique representation as a *p-ary numeral*:

$$S_{i_0}^{(p)}\, S_{i_1}^{(p)} \ldots S_{i_{n-2}}^{(p)}\, S_{i_{n-1}}^{(p)}(0), \tag{1}$$

where $n \geq 0$ and $i_{n-1} \neq 0$. We customarily write the p-ary numeral (1) as

$$(i_{n-1}\, i_{n-2} \ldots i_1\, i_0)_p \tag{2}$$

and called the numbers i_j in (2) the *p-ary digits* of the number x in p-ary representation. We clearly have

$$x = \sum_{j<n} i_j \cdot p^j.$$

The reader will note that the binary representation of natural numbers is a special case of p-ary representation with $p = 2$. The binary successor functions are $S_0^{(2)}(x) = 2 \cdot x + 0 = 2 \cdot x$ and $S_1^{(2)}(x) = 2 \cdot x + 1$. As an example we give here the binary representation of the first eight non-zero numbers:

$$\begin{aligned}
&1 = (1)_2 = S_1^{(2)}(0) = 2 \cdot 0 + 1\\
&2 = (1\,0)_2 = S_0^{(2)}\,S_1^{(2)}(0) = 2 \cdot (2 \cdot 0 + 1) + 0\\
&3 = (1\,1)_2 = S_1^{(2)}\,S_1^{(2)}(0) = 2 \cdot (2 \cdot 0 + 1) + 1\\
&4 = (1\,0\,0)_2 = S_0^{(2)}\,S_0^{(2)}\,S_1^{(2)}(0) = 2 \cdot (2 \cdot (2 \cdot 0 + 1) + 0) + 0\\
&5 = (1\,0\,1)_2 = S_1^{(2)}\,S_0^{(2)}\,S_1^{(2)}(0) = 2 \cdot (2 \cdot (2 \cdot 0 + 1) + 0) + 1\\
&6 = (1\,1\,0)_2 = S_0^{(2)}\,S_1^{(2)}\,S_1^{(2)}(1) = 2 \cdot (2 \cdot (2 \cdot 0 + 1) + 1) + 0\\
&7 = (1\,1\,1)_2 = S_1^{(2)}\,S_1^{(2)}\,S_1^{(2)}(1) = 2 \cdot (2 \cdot (2 \cdot 0 + 1) + 1) + 1\\
&8 = (1\,0\,0\,0)_2 = S_0^{(2)}\,S_0^{(2)}\,S_0^{(2)}\,S_1^{(2)}(0) = 2 \cdot (2 \cdot (2 \cdot (2 \cdot 0 + 1) + 0) + 0) + 0.
\end{aligned}$$

6.2.2 p-ary size. For every $p \geq 2$, the *p-ary size* function $|x|_p$ yields the length of x in the p-ary representation, i.e.

$$|(i_{n-1}\ \ldots\ i_0)_p|_p = n.$$

The function is defined by course of values recursion as a primitive recursive function:

$$\begin{aligned}
&|0|_p = 0\\
&|p \cdot x + i|_p = |x|_p + 1 \leftarrow i < p \land x \neq 0.
\end{aligned}$$

6.2.3 p-ary concatenation. For every $p \geq 2$, the binary function $x \#_p y$, called *p-ary concatenation* function, yields a number whose p-ary representation is obtained from p-ary representations of x and y by appending the p-ary digits of y after the p-ary digits of x, i.e.

$$(i_{n-1}\ \ldots\ i_0)_p \#_p (j_{m-1}\ \ldots\ j_0)_p = (i_{n-1}\ \ldots\ i_0\, j_{m-1}\ \ldots\ j_0)_p.$$

The function is defined by course of values recursion on y as a primitive recursive function:

$$\begin{aligned}
&x \#_p 0 = x\\
&x \#_p (p \cdot y + i) = p \cdot (x \#_p y) + i \leftarrow i < p \land y \neq 0.
\end{aligned}$$

6.2.4 Arithmetization of Turing machines. Let A be a Turing machine given by the transition function $\delta : Q \times S \mapsto Q \times S \times \Delta$, initial state b, and terminal state e. Assume further that $Q = \{q_0, q_1, \ldots, q_{n-1}\} \subset \mathbb{N}$, $S = \{s_0, s_1, \ldots, s_{p-1}\}$, and $\Delta = \{\blacktriangledown, \blacktriangleleft, \blacktriangleright\} \subset \mathbb{N}$.

In the subsequent paragraphs we will show how to arithmetize computation of the Turing machine A.

6.2.5 Arithmetization of the transition function. We arithmetize the transition function $\delta : Q \times S \mapsto Q \times S \times \Delta$ of the Turing machine A by the binary function $m(q, s)$, called also *transition function*, satisfying:

$$\langle q, s_i \rangle \in \mathrm{dom}(\delta) \leftrightarrow m(q, i) > 0 \tag{1}$$

$$\delta(q, s_i) = \langle q_1, s_j, d \rangle \rightarrow m(q, i) = q_1, j, d. \tag{2}$$

It should clear that there is a unique primitive recursive function m satisfying (1) and (2).

6.2.6 Arithmetization of tape words. We arithmetize the empty word over S by 0. Finite non-empty words

$$s_{i_{n-1}} \ldots s_{i_0}$$

over S with either $s_{i_{n-1}}$ or s_{i_0} non-blank are arithmetized in two different ways. Words with $s_{i_{n-1}} \not\equiv \mathbf{0}$ are *left* words and they are arithmetized in the p-ary notation as

$$\ulcorner s_{i_{n-1}} \ldots s_{i_0} \urcorner^{(l)} = (i_{n-1} \ldots i_0)_p = \sum_{j<n} i_j \cdot p^j.$$

Words with $s_{i_0} \not\equiv \mathbf{0}$ are *right* words and they are arithmetized in the 'reversed' p-ary notation as

$$\ulcorner s_{i_{n-1}} \ldots s_{i_0} \urcorner^{(r)} = (i_0 \ldots i_{n-1})_p = \sum_{j<n} i_j \cdot p^{n-j}.$$

Note that we have

$$\ulcorner s_{i_{n-1}} \ldots s_{i_0} \urcorner^{(r)} = \ulcorner s_{i_0} \ldots s_{i_{n-1}} \urcorner^{(l)}.$$

The empty word is both left and right at the same time.

6.2.7 Arithmetization of configurations. The current configuration of the Turing machine A is completely determined by the current state q of the machine and by the current content of the tape which can be uniquely written as $\mathbf{0}^\infty w_1 w_2 \mathbf{0}^\infty$, where w_1 is a left word, w_2 is a right word, and the currently scanned square is the first symbol of the (infinite) word $w_2 \mathbf{0}^\infty$. Such a configuration is arithmetized by the number $c = q, l, r$, where l codes the left word w_1 and r the right word w_2.

6.2.8 Arithmetization of one computation step. We arithmetize one computation step of the Turing machine A by a unary function $M(c)$ which takes the current configuration of A coded by c and yields a new configuration coded by $M(c)$ obtained from the previous one by one computation step. The function is defined explicitly as a primitive recursive function:

$M(q,l,s+p\cdot r) = q_1,l,s_1+p\cdot r \leftarrow s<p \wedge m(q,s)=q_1,s_1,\blacktriangledown$
$M(q,l,s+p\cdot r) = q_1,l_1,s_2+p\cdot(s_1+p\cdot r) \leftarrow$
$\quad s<p \wedge m(q,s)=q_1,s_1,\blacktriangleleft \wedge l = p\cdot l_1+s_2 \wedge s_2<p$
$M(q,l,s+p\cdot r) = q_1,p\cdot l+s_1,r \leftarrow s<p \wedge m(q,s)=q_1,s_1,\blacktriangleright.$

Note that the function $M(c)$ yields 0 if the computation with the configuration coded by c is undefined.

6.2.9 Arithmetization of computation. We arithmetize computation of the Turing machine A by a unary partial function $M^*(c)$, called the *unbounded iteration of* M, which takes the current configuration of A coded by c and yields a final configuration coded by $M^*(c)$ obtained from the previous one by computation of A. The function is defined explicitly as a partial recursive function:

$$M^*(c) \simeq \mathcal{D}_s(H(c) \neq_* e, M^*\, M(c), c).$$

6.2.10 Auxiliary function and predicate. The function $\ulcorner \mathbf{1}^x \urcorner^{(r)}$ yields the code of a right word of the length x containing only $\mathbf{1}$'s. The function is defined by primitive recursive definition as primitive recursive:

$\ulcorner \mathbf{1}^0 \urcorner^{(r)} = 0$
$\ulcorner \mathbf{1}^{x+1} \urcorner^{(r)} = p\cdot \ulcorner \mathbf{1}^x \urcorner^{(r)} + 1.$

The predicate $Ones(w)$ holds if w is the code a right word consisting only from $\mathbf{1}$'s, i.e. if $w = \ulcorner \mathbf{1}^x \urcorner^{(r)}$ for some x. The predicate is defined by course of values recursion as a primitive recursive predicate:

$Ones(0)$
$Ones(p\cdot w+1) \leftarrow Ones(w).$

6.2.11 Encoding input. The binary *encoding* function $Enc(n,x)$ takes an n-tuple $x = x_1,\ldots,x_n$ and yields the code of the initial tape configuration of the Turing machine A with the initial state b and the tape of the form $\mathbf{0}^\infty\underline{\mathbf{0}}\mathbf{1}^{x_1}\mathbf{0}\ldots\mathbf{0}\mathbf{1}^{x_n}\mathbf{0}^\infty$, i.e.

$$Enc(n,x_1,\ldots,x_n) = b,0,\ulcorner \mathbf{0}\mathbf{1}^{x_1}\mathbf{0}\ldots\mathbf{0}\mathbf{1}^{x_n}\urcorner^{(r)}.$$

For that purpose we need an auxiliary binary function $Enc_1(n,x)$ which yields the code of the right word $\mathbf{1}^{x_1}\mathbf{0}\ldots\mathbf{0}\mathbf{1}^{x_n}$, where $x = x_1,\ldots,x_n$, i.e.

$$Enc_1(n,x_1,\ldots,x_n) = \ulcorner \mathbf{1}^{x_1}\mathbf{0}\ldots\mathbf{0}\mathbf{1}^{x_n}\urcorner^{(r)}.$$

The function $Enc_1(n,x)$ is defined by primitive recursion on n with substitution in parameter as a primitive recursive function:

$Enc_1(1,x_1) = \ulcorner \mathbf{1}^{x_1}\urcorner^{(r)}$
$Enc_1(n+2,x_1,x) = p\cdot Enc_1(n+1,x) \,\#_p \ulcorner \mathbf{1}^{x_1}\urcorner^{(r)}.$

The encoding function is then defined explicitly as primitive recursive:

$$Enc(n,x) = b, 0, p \cdot Enc_1(n,x).$$

6.2.12 Decoding output. The *decoding* partial function $Dc(c)$ is defined if c is the code of a final tape configuration of the Turing machine A; in such case it yields the result of computation stored on that tape, i.e.

$$Dc(c) \asymp y \leftrightarrow c = e, 0, \ulcorner \mathbf{0}\mathbf{1}^{y} \urcorner(r),$$

or equivalently

$$Dc(c) \asymp y \leftrightarrow \exists r(c = e, 0, p \cdot r \wedge Ones(r) \wedge y = |r|_p).$$

The decoding partial function is defined explicitly from one minimalization as a partial recursive function:

$$Dc(c) \simeq \big|\mu r[c = e, 0, p \cdot r \wedge Ones(r)]\big|_p.$$

6.2.13 Theorem. *Partially Turing computable functions are partial recursive.*

Proof. Let f be an n-ary partial function computed by the Turing machine A given by the transition function $\delta : Q \times S \mapsto Q \times S \times \Delta$, initial state b, and terminal state e. We may assume without loss of generality that the sets Q and Δ are as in Par. 6.2.4 because we can always systematically modify the transition fuction δ whereby we obtain an equivalent Turing machine computing the same partial function f. Using the notation of the previous paragraphs we can derive f as a partial recursive function by the following explicit definition:

$$f(\vec{x}) \simeq Dc\, M^* \, Enc\big(n, (\vec{x})\big).$$ □

Recursiveness Implies Turing Computability

See the paragraphs 6.1.5 - 6.1.11 in [Vod00].

Equivalence of Turing Computability and Recursiveness

6.2.14 Theorem. *Partial recursive functions coincide with partially Turing computable functions.*

Proof. See the corollary 6.1.12 in [Vod00]. □

6.3 Turing-Church Theses and Computable Functions

6.3.1 Turing-Church Theses. See the paragraph 6.1.13 in [Vod00].

6.3.2 Partially computable functions. A partial function f is *partially computable* if it is Turing computable, i.e. if there is a Turing machine computing f. A *computable* function is a partially computable function which is total.

6.3.3 Remark. By Thm. 6.2.14 partially computable functions are exactly partial recursive functions.

6.4 Other Models of Computation

Herbrand-Gödel-Kleene Recursive Functions

Church's λ-Definable Functions

Post's Normal Systems

Markov Systems

Register Machines

6.5 Exercises

Turing Machines

Equivalence of Turing Machines and Recursiveness

Turing-Church Theses and Computable Functions

Other Models of Computation

7. Beyond Computability

7.1 Decidable and Undecidable Predicates

7.1.1 Decidable predicates. We say that an n-ary predicate R is *decidable* if the truth value of $R(\vec{x})$ can be effectively calculated for all numbers $\vec{x}$, i.e. if its characteristic function R_* is effectively computable. By Turing's Thesis decidable predicates are exactly the computable and hence recursive predicates. Are there any undecidable predicates?

We already know that the graphs $\psi^{(n)}(e,\vec{x}) \asymp y$ of the partial enumeration functions are not recursive. We give below some further examples of non-decidable problems. All problems we are interested in are concerned the computation of partial recursive functions.

In the sequel we will often abbreviate $\psi^{(n)}(e,\vec{x})$ to $\varphi_e^{(n)}(\vec{x})$.

7.1.2 The halting problem. Given $n \geq 1$, we consider the problem of deciding, for any $\lambda_n.\tau$ and $\vec{x}$, whether the reduction of a term $(\lambda_n.\tau)(\underline{\vec{x}})$ halts and yields a result after finitely many steps, or equivalently, whether $\varphi^{(n)}_{\ulcorner\lambda_n.\tau\urcorner}(\vec{x})$ is defined. This is called the *halting problem.*

Its arithmetization is the $(n+1)$-ary predicate $W_e^{(n)}(\vec{x})$ in e and $\vec{x}$, called the *halting predicate*, defined by

$$W_e^{(n)}(\vec{x}) \leftrightarrow \varphi_e^{(n)}(\vec{x})\!\downarrow .$$

In the sequel we will abbreviate $W^{(1)}$ to W.

7.1.3 The halting problem is undecidable. We claim that the halting predicates $W^{(n)}$ are not recursive. We give here two differrent proofs of this fact. Each of them uses a diagonal argument. The first one directly; the second indirectly through the partial enumeration functions (see Par. 5.2.23).

First proof. Assume that the predicate $W^{(n)}$ is recursive. Then the $(n+1)$-ary partial function f defined by

$$f(e,\vec{x}) \simeq \begin{cases} \bot & \text{if } \varphi_e^{(n)}(\vec{x})\!\downarrow \\ 0 & \text{if } \varphi_e^{(n)}(\vec{x})\!\uparrow \end{cases}$$

is recursive since it can be derived from $W^{(n)}$ as a partial recursive function by the following explicit definition:

$$f(e,\vec{x}) \simeq \textbf{if } W^{(n)}_e(\vec{x}) \textbf{ then } \emptyset^{(n)}(\vec{x}) \textbf{ else } 0.$$

Consequently, the n-ary partial function g explicitly defined by

$$g(x_1,\dots,x_n) \simeq f(x_1,x_1,\dots,x_n)$$

is recursive as well. Let e be an index of g. We obtain a contradiction as follows ($\vec{e} \equiv e,\dots,e$):

$$W^{(n)}_e(\vec{e}) \Leftrightarrow \varphi^{(n)}_e(\vec{e})\downarrow \overset{\text{index}}{\Leftrightarrow} g(\vec{e})\downarrow \overset{\text{def}}{\Leftrightarrow} f(e,\vec{e})\downarrow \overset{\text{def}}{\Leftrightarrow} \neg W^{(n)}_e(\vec{e}).$$

Second proof. Consider an $(n+1)$-ary function f defined by

$$f(e,\vec{x}) = \begin{cases} \varphi^{(n)}_e(\vec{x}) & \text{if } \varphi^{(n)}_e(\vec{x})\downarrow, \\ 0 & \text{if } \varphi^{(n)}_e(\vec{x})\uparrow. \end{cases}$$

But $f(e,\vec{x})$ is the completion of the partial enumeration function $\psi^{(n)}(e,\vec{x})$ of which we know from Par. 5.2.23 that it is not recursive. Recall that this fact has been proved by a diagonal argument.

Now if the predicate $W^{(n)}$ were recursive then f would be recursive since it could be defined explicitly as a recursive function by

$$f(e,\vec{x}) \simeq \textbf{if } W^{(n)}_e(\vec{x}) \textbf{ then } \varphi^{(n)}_e(\vec{x}) \textbf{ else } 0.$$

7.1.4 The diagonal halting problem is undecidable. Consider a unary predicate K defined by

$$K(x) \leftrightarrow \varphi_x(x)\downarrow .$$

The predicate is called the *diagonal halting* predicate. It solves the problem of deciding, for any $\lambda_1.\tau$, whether the computation of $\lambda_1.\tau$ with the argument $\ulcorner\lambda_1.\tau\urcorner$ halts and yields a result after finitely many steps, or equivalently, whether $\varphi_{\ulcorner\lambda_1.\tau\urcorner}(\ulcorner\lambda_1.\tau\urcorner)$ is defined. This is called the *diagonal halting problem*.

We claim that the diagonal halting problem is undecidable; that is, that the predicate K is not recursive. We give here two differrent proofs of undecidability of the diagonal halting problem. The first one uses the standard diagonal argument; the second the method of *reduction*.

First proof. Assume that the predicate K is recursive. Then f defined by

$$f(x) \simeq \begin{cases} \bot & \text{if } \varphi_x(x)\downarrow \\ 0 & \text{if } \varphi_x(x)\uparrow \end{cases}$$

is a partial recursive function since it can be derived from K as a partial recursive function by the following explicit definition:

$$f(x) \simeq \textbf{if } K(x) \textbf{ then } \emptyset(x) \textbf{ else } 0.$$

Let e be an index of f. We obtain a contradiction as follows:

$$K(e) \overset{\text{def}}{\Leftrightarrow} \varphi_e(e)\!\downarrow \overset{\text{index}}{\Leftrightarrow} f(e)\!\downarrow \overset{\text{def}}{\Leftrightarrow} \neg K(e).$$

Second proof. We show that the halting problem can be *reduced* to the diagonal halting problem by showing that if we could effectively solve the diagonal halting problem we could use this to get an effective method for solving the halting problem. More specifically, we claim that there is a recursive (in fact primitive recursive) function f satisfying

$$W_e(x) \leftrightarrow K\, f(e,x). \tag{1}$$

For if the diagonal halting problem were decidable then we could take (1) as an explicit definition of W as a recursive predicate which, by Par. 7.1.3, it is not.

It remains to show that such recursive function f does exist. We define f explicitly as a primitive recursive function

$$f(e,x) = e \circ \ulcorner C_x \urcorner$$

and derive (1) as follows:

$$K\, f(e,x) \overset{\text{def}}{\Leftrightarrow} K(e \circ \ulcorner C_x \urcorner) \overset{\text{def}}{\Leftrightarrow} \varphi_{e \circ \ulcorner C_x \urcorner}(e \circ \ulcorner C_x \urcorner)\!\downarrow \overset{5.2.29(1)}{\Leftrightarrow}$$
$$\varphi_e\, \varphi_{\ulcorner C_x \urcorner}(e \circ \ulcorner C_x \urcorner)\!\downarrow \overset{5.2.26(1)}{\Leftrightarrow} \varphi_e\, C_x(e \circ \ulcorner C_x \urcorner)\!\downarrow \Leftrightarrow \varphi_e(x)\!\downarrow \Leftrightarrow W_e(x).$$

7.1.5 Remark. In the previous paragraph we have used two different methods for demonstrating undecidability of the diagonal halting problem. The first one uses the *direct* method which is based on a diagonal argument and leads to contradiction. The second one is *indirect* and it takes a problem already known to be undecidable (in this case the halting problem) and reduces it to the problem in question (in this case the diagonal halting problem). The latter method is called the method of *reduction* and will be our basic machinery in demonstrating undecidability of further problems.

7.1.6 Reducibility of predicates. An n-ary predicate R is *reducible* to an m-ary predicate Q if for all numbers $\vec{x}$ we have

$$R(\vec{x}) \leftrightarrow Q(f_1(\vec{x}), \ldots, f_m(\vec{x})) \tag{1}$$

for some recursive functions $f_1, \ldots, f_m$.

It is easy to see that if we have

$$R(\vec{x}) \leftrightarrow Q(\tau_1[\vec{x}], \dots, \tau_m[\vec{x}]), \tag{2}$$

where $\tau_1, \dots, \tau_m$ are terms applying only recursive functions, then the predicate R is reducible to the predicate Q. For it suffices to define m-recursive functions $f_1, \dots, f_m$ explicitly by $f_1(\vec{x}) = \tau_1[\vec{x}], \dots, f_m(\vec{x}) = \tau_m[\vec{x}]$, whereby we obtain that (1) holds. For this reason we say that R reducible to Q if (2) holds.

7.1.7 Reducibility lemma. *If the predicate R is reducible to Q and Q is a recursive predicate then so is R.*

Proof. Directly from Thm. 5.2.3. □

7.1.8 Corollary. *If the halting predicate $W^{(n)}$ is reducible to Q then Q is not recursive.*

Proof. Directly from the reducibility lemma and Par. 7.1.3. □

7.1.9 Existence of programs with undecidable halting problems. In this paragraph we will consider the following problem. Given a program $\lambda_n.\tau$ decide for any numbers $\vec{x}$ whether the computation of $\lambda_n.\tau$ with the input $\vec{x}$ halts and terminate after finitely many steps. This is called the *halting problem for the program $\lambda_n.\tau$*. Natural questions arise. Is there any program whose halting problem decidable? Can we find a program whose halting problem undecidable?

The arithmetization of the problem is, for a given number e_0, the n-ary predicate $W^{(n)}_{e_0}(\vec{x})$ in $\vec{x}$. The first question can be restated as follows. Can we find a number e_0 such the predicate $W^{(n)}_{e_0}$ is recursive? The answer is simple. If e_0 is arbitrary index of an n-ary recursive function then the predicate $W^{(n)}_{e_0}(\vec{x})$ holds for every $\vec{x}$ and hence it is trivially a recursive predicate. The second question, i.e. whether there is a number e_0 such the predicate $W^{(n)}_{e_0}$ is not recursive, is much harder.

We claim that the partial enumeration functions have undecidable halting problems. Indeed, let e_0 be an index of the $(n+1)$-ary partial enumeration function $\psi^{(n)}$. First note that we have

$$W^{(n+1)}_{e_0}(e,\vec{x}) \Leftrightarrow \varphi^{(n+1)}_{e_0}(e,\vec{x})\!\downarrow \overset{\text{index}}{\Leftrightarrow} \psi^{(n)}(e,\vec{x})\!\downarrow \Leftrightarrow \varphi^{(n)}_{e}(\vec{x})\!\downarrow \Leftrightarrow W^{(n)}_{e}(\vec{x})$$

and thus

$$W^{(n)}_{e}(\vec{x}) \leftrightarrow W^{(n+1)}_{e_0}(e,\vec{x}).$$

We have shown that the predicate $W^{(n)}$ is reducible to the predicate $W^{(n+1)}_{e_0}$. By Thm. 7.1.8, the predicate $W^{(n+1)}_{e_0}$ is not recursive.

We already knew that there are non-recursive predicates $W_{e_0}^{(n)}$ for $n > 1$. Can we find a number e_0 such the unary predicate W_{e_0} is not recursive? To see this, consider a unary partial recursive function f defined by

$$f(x) \simeq \varphi_x(x).$$

Let e_0 be an index of f. We have

$$W_{e_0}(x) \Leftrightarrow \varphi_{e_0}(x)\!\downarrow\; \Leftrightarrow f(x)\!\downarrow\; \Leftrightarrow \varphi_x(x)\!\downarrow\; \Leftrightarrow K(x).$$

Thus the predicate K can be reduced to the predicate W_{e_0}. By Par. 7.1.4, the predicate K is not recursive and thus, by Thm. 7.1.7, so is the predicate W_{e_0}. In other words the halting problem of the partial function f is undecidable.

7.1.10 The input problem is undecidable. In this paragraph we will consider the following problem. Given $n \geq 1$ and n fixed numbers $\vec{x}_0$ decide, for any $\lambda_n.\tau$, whether the computation of $\lambda_n.\tau$ with the input $\vec{x}_0$ halts and terminate after finitely many steps. This is called the *input problem for* $\vec{x}_0$. Its arithmetization is the unary predicate $W_e^{(n)}(\vec{x}_0)$ in e.

We claim that the input problem is undecidable; that is, the predicates $W_e^{(n)}(\vec{x}_0)$ are not recursive. We give here two differrent proofs of this fact. Both reduce the halting problem to the input problem by providing a recursive (in fact primitive recursive) function k satisfying

$$W_e(x) \leftrightarrow W_{k(e,x)}^{(n)}(\vec{x}_0). \tag{1}$$

First proof. Consider an $(n+2)$-ary partial function f defined by

$$f(e,x,\vec{y}) \simeq \begin{cases} 0 & \text{if } \varphi_e(x)\!\downarrow, \\ \bot & \text{if } \varphi_e(x)\!\uparrow. \end{cases} \tag{2}$$

The f is a partial recursive function since it could be defined explicitly by

$$f(e,x,\vec{y}) \simeq I_1^2\big(0, \varphi_e(x)\big).$$

By the *s-m-n* theorem there is a primitive recursive function $k(e,x)$ such that

$$\varphi_{k(e,x)}^{(n)}(\vec{y}) \simeq f(e,x,\vec{y}). \tag{3}$$

If $W_e(x)$ then $\varphi_{k(e,x)}^{(n)}(\vec{y}) = 0$ for all $\vec{y}$, i.e. $\varphi_{k(e,x)}^{(n)} = C_0^n$; otherwise $\varphi_{k(e,x)}^{(n)}(\vec{y})$ is undefined for all $\vec{y}$, i.e. $\varphi_{k(e,x)}^{(n)} = \emptyset^{(n)}$. Thus we have for all $\vec{y}$

$$W_e(x) \leftrightarrow \varphi_{k(e,x)}^{(n)}(\vec{y})\!\downarrow.$$

From this we get (1).

Second proof. Now we consider, for given numbers e and x, n-ary partial functions $f_{e,x}$ defined by

$$f_{e,x}(\vec{y}) \simeq \begin{cases} 0 & \text{if } \varphi_e(x)\downarrow, \\ \bot & \text{if } \varphi_e(x)\uparrow. \end{cases}$$

From the definition we obtain that $f_{e,x}$ is total iff $W_e(x)$ holds. The partial function $f_{e,x}$ is recursive since it could be defined explicitly by

$$f_{e,x}(\vec{y}) \simeq I_1^2\big(0, \varphi_e(x)\big),$$

or equivalently, by

$$f_{e,x}(\vec{y}) \simeq I_1^2\big(C_0^n(\vec{y}), \varphi_e\, C_x^n(\vec{y})\big).$$

Consider now a binary function $k(e,x)$ defined explicitly as a primitive recursive function:

$$k(e,x) = \mathcal{C}_2^n\big(\ulcorner I_1^2 \urcorner, \ulcorner C_0^n \urcorner, \mathcal{C}_1^n(e, \ulcorner C_x^n \urcorner)\big).$$

It is easy to see that $k(e,x)$ is the index of $f_{e,x}$, i.e. $f_{e,x} = \varphi_{k(e,x)}$. Clerly, we have for all $\vec{y}$

$$W_e(x) \leftrightarrow \varphi_{k(e,x)}^{(n)}(\vec{y})\downarrow .$$

From this we get (1).

7.1.11 The uniform halting problem is undecidable. Given $n \geq 1$, we consider the problem of deciding, for any $\lambda_n.\tau$, whether the reduction of a term $(\lambda_n.\tau)(\underline{\vec{x}})$ halts and yields a result after finitely many steps for all numbers $\vec{x}$. In other words, we consider the problem of deciding, for any $\lambda_n.\tau$, whether the 'program' $\lambda_n.\tau$ computes an n-ary (total) function, or equivalently, whether the partial function $\varphi_{\ulcorner \lambda_n.\tau \urcorner}$ is total. This is called the *uniform halting problem*.

Its arithmetization is the unary predicate $Tot_n(e)$, called the *uniform halting predicate*, defined by

$$Tot_n(e) \leftrightarrow \forall \vec{x}\, \varphi_e^{(n)}(\vec{x})\downarrow .$$

In the sequel we usually ommit the subscript n in $Tot_n(e)$ for the case $n = 1$.

We claim that the uniform halting problem is undecidable; that is, the predicates Tot_n are not recursive. We use again the method of reduction. Consider the $(n+2)$-ary partial function f defined by 7.1.10(2) and the primitive recursive function k such that 7.1.10(3) holds. It easy to see that $\varphi_{k(e,x)}^{(n)}$ is total iff $W_e(x)$ holds. From this we get

$$W_e(x) \leftrightarrow Tot_n\, k(e,x).$$

7.1.12 Further examples of undecidable problems. We now give further examples of undecidable problems. The demonstration of their undecidability is left to the reader as an exercise. We consider the following problems:

(i) Given $n \geq 1$, the problem of deciding, for any e_1 and e_2, whether $\varphi_{e_1}^{(n)} = \varphi_{e_2}^{(n)}$. This is called the *equivalence problem.*

(ii) Given $n \geq 1$ and an n-ary partial recursive function f, the problem of deciding, for any e, whether $\varphi_e^{(n)} = f$.

(iii) Given $n \geq 1$, the problem of deciding, for any y and e, whether $y \in \mathrm{rng}(\varphi_e^{(n)})$.

(iv) Given $n \geq 1$ and a number y_0, the problem of deciding, for any e, whether $y_0 \in \mathrm{rng}(\varphi_e^{(n)})$. This is called the *output problem.*

(v) Given $n \geq 1$, the problem of deciding, for any e, whether $\mathrm{rng}(\varphi_e^{(n)})$ is infinite.

(vi) Given $n \geq 1$, the problem of deciding, for any e, whether $\varphi_e^{(n)}$ is a constant function.

(vii) Given $n \geq 1$, the problem of deciding, for any e, whether $\varphi_e^{(n)}$ is the characteristic function of an n-ary recursive predicate.

Further examples of undecidable problems are given in Sect. 7.5.

7.1.13 Theorem (Rice). *For every $n \geq 1$ and $\mathcal{F}$, if $\emptyset \subset \mathcal{F}^{(n)} \subset \mathrm{PREC}^{(n)}$ then the problem $\varphi_e^{(n)} \in \mathcal{F}^{(n)}$ is undecidable.*

Proof. By the method of reduction. First note that from assumptions we obtain that there is an n-ary partial recursive function f such that

$$f \in \mathcal{F}^{(n)} \leftrightarrow \emptyset^{(n)} \notin \mathcal{F}^{(n)}. \tag{1}$$

Consider an $(n+2)$-ary partial function g defined by

$$g(e,x,\vec{y}) \simeq \begin{cases} f(\vec{y}) & \text{if } \varphi_e(x)\downarrow, \\ \perp & \text{if } \varphi_e(x)\uparrow. \end{cases}$$

The g is a partial recursive function since it could be defined explicitly by

$$g(e,x,\vec{y}) \simeq I_1^2\big(f(\vec{y}), \varphi_e(x)\big).$$

By the s-m-n theorem there is a primitive recursive function $k(e,x)$ such that

$$\varphi_{k(e,x)}^{(n)}(\vec{y}) \simeq g(e,x,\vec{y}).$$

It is easy to see that we have

$$W_e(x) \to \varphi_{k(e,x)}^{(n)} = f \tag{2}$$

$$\neg W_e(x) \to \varphi_{k(e,x)}^{(n)} = \emptyset^{(n)}. \tag{3}$$

By combining the properties (1), (2), and (3) we can conclude that

$$W_e(x) \leftrightarrow \varphi^{(n)}_{k(e,x)} \in \mathcal{F}^{(n)} \leftrightarrow f \in \mathcal{F}^{(n)}.$$

Now, if $f \in \mathcal{F}^{(n)}$ then we have

$$W_e(x) \leftrightarrow \varphi^{(n)}_{k(e,x)} \in \mathcal{F}^{(n)}.$$

Otherwise, if $f \notin \mathcal{F}^{(n)}$ then we have

$$\neg W_e(x) \leftrightarrow \varphi^{(n)}_{k(e,x)} \in \mathcal{F}^{(n)}.$$

In either case we have found an undecidable problem which can be reduced to the problem $\varphi^{(n)}_e \in \mathcal{F}^{(n)}$ which is undecidable by Lemma 7.1.7. □

7.1.14 Extensional and non-trivial predicates. We say that an unary predicate R is *extensional* if the following holds for every e_1 and e_2:

$$\varphi_{e_1} = \varphi_{e_2} \to R(e_1) \leftrightarrow R(e_2).$$

The predicate R is *non-trivial* if $\emptyset \subset R \subset \mathbb{N}$.

7.1.15 Corollary of the Rice's theorem. *Every extensional non-trivial predicate is not recursive.*

Proof. Consider a class $\mathcal{F}$ of unary partial functions defined as

$$f \in \mathcal{F} \leftrightarrow \exists e\big(f = \varphi_e \land R(e)\big).$$

Clearly $\mathcal{F} \subseteq \mathrm{PREC}^{(1)}$. By extensionality of R we get

$$R(e) \leftrightarrow \varphi_e \in \mathcal{F} \tag{1}$$

and thus, since R is non-trivial, we obtain $\emptyset \subset \mathcal{F} \subset \mathrm{PREC}^{(1)}$. By the Rice's theorem, the problem $\varphi_e \in \mathcal{F}$ is undecidable and thus, by (1), the predicate R is not recursive. □

7.1.16 Remark. In the sequel, both the theorem and the corollary will be referred to simply as the Rice's theorem.

7.1.17 Application of the Rice's theorem. As an example we give here an alternative proof of undecidability of the uniform halting problem for unary partial recursive functions. It is easy to see that the predicate $Tot(e)$ is extensional and non-trivial. Now it suffices to apply the corollary of the Rice's theorem.

7.2 Semidecidable Predicates

7.2.1 Semidecidable predicates. In this section we study an important class of predicates which are in general undecidable but if they hold then we can effectively confirm this.

We say that an n-ary predicate R is *(positively) semidecidable* if there is an algorithm which, when applied to the input $\vec{x}$, gives a result iff $R(\vec{x})$ holds. Note that for any $\vec{x}$ we are able to confirm effectively that $R(\vec{x})$ holds by computing the algorithm for the input $\vec{x}$. However, we are not able to refute $R(\vec{x})$ by this computation because it goes forever. By Turing's Thesis semidecidable predicates are exactly predicates which are domains of partially computable functions. This leads to the following definition.

7.2.2 Semicomputable predicates. An n-ary predicate R is a *(positively) semicomputable predicate* if it is the domain of an n-ary partially computable function f, i.e. the following holds for all $\vec{x}$: $R(\vec{x}) \leftrightarrow f(\vec{x})\!\downarrow$.

7.2.3 Theorem. *Computable predicates are semicomputable predicates.*

Proof. If R is an n-ary computable predicate then it is clearly the domain of the n-ary partially computable function f defined by $f(\vec{x}) \simeq \mu_y[R(\vec{x})]$. $\square$

7.2.4 Normal form theorem (Kleene). *For every n-ary semicomputable predicate R there exists a number e such that*

$$R(\vec{x}) \leftrightarrow \exists y\, \mathrm{T}_n(e, \vec{x}, y). \tag{1}$$

Proof. If R is an n-ary semicomputable predicate then $R(\vec{x}) \leftrightarrow f(\vec{x})\!\downarrow$ for a partially computable function f. By the normal form theorem there is a number e such that $f(\vec{x})\!\downarrow\ \leftrightarrow \exists y\, \mathrm{T}_n(e, \vec{x}, y)$ and hence (1) holds. $\square$

7.2.5 Indices of semicomputable predicates. If e is an index of the n-ary partially computable function f then the number e is called the *index* of the semicomputable predicate defined by $R(\vec{x}) \leftrightarrow f(\vec{x})\!\downarrow$. Note that we then have $R(\vec{x}) \leftrightarrow W_e^{(n)}(\vec{x})$ since $f(\vec{x})\!\downarrow\ \overset{\text{index}}{\Leftrightarrow} \varphi_e^{(n)}(\vec{x})\!\downarrow\ \Leftrightarrow W_e^{(n)}(\vec{x})$.

7.2.6 Theorem. *For every $n \geq 1$ and e, the $W_e^{(n)}$ is a semicomputable predicate such that*

$$W_e^{(n)}(\vec{x}) \leftrightarrow \exists y\, \mathrm{T}_n(e, \vec{x}, y). \tag{1}$$

Proof. By definition $W_e^{(n)}(\vec{x}) \leftrightarrow \varphi_e^{(n)}(\vec{x})\!\downarrow$ and since, by Thm. 5.2.19, the partial function $\varphi_e^{(n)}$ is computable, we have that the predicate $W_e^{(n)}$ semicomputable. By Thm. 5.2.19 again we have $\varphi_e^{(n)}(\vec{x})\!\downarrow\ \leftrightarrow \exists y\, \mathrm{T}_n(e, \vec{x}, y)$ and hence (1) holds. $\square$

7.2.7 Enumeration of classes of predicates. Given a number $n \geq 1$ and a class $\mathcal{F}$, we say that an $(n+1)$-ary predicate $Q(e, \vec{x})$ *enumerates* the class of n-ary predicates of $\mathcal{F}$ if the following holds for every n-ary predicate R:

$$R \in \mathcal{F}_*^{(n)} \leftrightarrow \exists e \forall \vec{x} \big(R(\vec{x}) \leftrightarrow Q(e, \vec{x})\big).$$

7.2.8 Enumeration theorem (Kleene). *The $(n+1)$-ary predicate $W^{(n)}$ is semicomputable enumerating the class of n-ary semicomputable predicates.*

Proof. The predicate $W^{(n)}$ is semicomputable since it is the domain of the partial enumeration function $\psi^{(n)}$:

$$W_e^{(n)}(\vec{x}) \Leftrightarrow \varphi_e^{(n)}(\vec{x})\!\downarrow \; \Leftrightarrow \psi^{(n)}(e, \vec{x})\!\downarrow .$$

We have to show that $W^{(n)}$ enumerates the class of n-ary semicomputable predicates. If R is an n-ary semicomputable predicate then by the normal form theorem there is a number e such that $R(\vec{x}) \leftrightarrow \exists y\, \mathrm{T}_n(e, \vec{x}, y)$. Now it suffices to apply 7.2.6(1) whereby we obtain that $R = W_e^{(n)}$. The reverse direction that every $W_e^{(n)}$ is a semicomputable predicate follows from the first part of Thm. 7.2.6. □

7.2.9 Corollary. *A predicate is semicomputable iff it has an index.*

Proof. Directly from the enumeration theorem. □

7.2.10 Theorem (Post). *A predicate R is computable iff both R and its complement R^c are semicomputable predicates.*

Proof. If R is a computable predicate then its complement R^c is computable and thus, by Thm. 7.2.3, both predicates are semicomputable.

Vice versa, if R and R^c are n-ary semicomputable predicates then, by the normal form theorem, there are numbers e_1 and e_2 such that

$$R(\vec{x}) \leftrightarrow \exists y\, \mathrm{T}_n(e_1, \vec{x}, y)$$
$$R^c(\vec{x}) \leftrightarrow \exists y\, \mathrm{T}_n(e_2, \vec{x}, y).$$

Consider an n-ary computable function f defined by *regular* minimalization:

$$f(\vec{x}) = \mu_y[\mathrm{T}_n(e_1, \vec{x}, y) \vee \mathrm{T}_n(e_2, \vec{x}, y)].$$

We derive R as a computable predicate by the following explicit definition:

$$R(\vec{x}) \leftrightarrow \mathrm{T}_n(e_1, \vec{x}, f(\vec{x})).$$ □

7.2.11 Complements of halting problems are not semidecidable. If the complement of the halting predicate $W^{(n)}$ were semicomputable then, by the Post's theorem, the halting predicate $W^{(n)}$ would be computable which, by Par. 7.1.3, it is not. The same argument can be applied to the diagonal halting predicate K; that is, its complement K^c is not a semicomputable predicate.

7.2.12 Theorem. *Semicomputable predicates are not closed under complements.*

Proof. Directly from the Post's theorem and Par. 7.2.11. □

7.2.13 Reducibility lemma. *If the predicate R is reducible to Q and Q is a semicomputable predicate then so is R.*

Proof. Suppose that R is an n-ary predicate reducible to an m-ary semicomputable predicate Q, i.e. we have $R(\vec{x}) \leftrightarrow Q\bigl(f_1(\vec{x}), \dots, f_m(\vec{x})\bigr)$ for some recursive functions $f_1, \dots, f_m$. By definition, the predicate Q is the domain of an m-ary partially computable function f. We then have

$$Q\bigl(f_1(\vec{x}), \dots, f_m(\vec{x})\bigr) \leftrightarrow f\bigl(f_1(\vec{x}), \dots, f_m(\vec{x})\bigr)\downarrow$$

and thus the predicate R is the domain of the n-ary partially computable function g explicitly defined by $g(\vec{x}) \simeq f\bigl(f_1(\vec{x}), \dots, f_m(\vec{x})\bigr)$. □

7.2.14 Corollary. *If the complement $\neg W^{(n)}$ of the halting predicate is reducible to Q then Q is not semicomputable.*

Proof. Directly from the reducibility lemma and Par. 7.2.11. □

7.2.15 Auxiliary predicate. For every $n \geq 1$, we define the $(n+2)$-ary predicate $\mathrm{S}_n(e, \vec{x}, k)$ explicitly as a primitive recursive predicate by

$$\mathrm{S}_n(e, \vec{x}, k) \leftrightarrow \mathit{Cdf}_n(e) \wedge \mathit{Nm}\, \mathit{Rd}^k\, e\bigl(\ulcorner \underline{(\vec{x})} \urcorner^{(n)}\bigr).$$

It is easy to see that the predicate satisfies

$$\mathrm{S}_n(e, \vec{x}, k_1) \wedge k_1 \leq k_2 \rightarrow \mathrm{S}_n(e, \vec{x}, k_2)$$
$$\exists k\, \mathrm{S}_n(e, \vec{x}, k) \leftrightarrow \exists y\, \mathrm{T}_n(e, \vec{x}, y).$$

The predicate $\mathrm{S}_n(e, \vec{x}, k)$ can be read as *the computation of $\varphi_e^{(n)}(\vec{x})$ converges in $\leq k$ steps.* In the sequel we will often abbreviate $\mathrm{S}_1(e, x, k)$ to $\mathrm{S}(e, x, k)$.

7.2.16 The uniform halting problem is not semidecidable. We claim that the uniform halting problem is not semidecidable; that is, that the predicates Tot_n are not semicomputable. We use the method of reduction by providing a computable function k satisfying

$$\neg W_e(x) \leftrightarrow \mathit{Tot}_n\, k(e, x) \tag{1}$$

Consider the $(n+3)$-ary partial function f defined by

$$f(e, x, t, \vec{y}) \simeq \begin{cases} \bot & \text{the computation of } \varphi_e(x) \text{ converges in} \leq t \text{ steps,} \\ 0 & \text{otherwise.} \end{cases}$$

The f is computable since it can be defined from S as a partially computable function by the following explicit definition:

$$f(e,x,t,\vec{y}) \simeq \textbf{if } \mathrm{S}(e,x,t) \textbf{ then } \emptyset^{(n)}(\vec{y}) \textbf{ else } 0.$$

By the s-m-n theorem there is a computable (in fact primitive recursive) function $k(e,x)$ such that ($\vec{y} \equiv y_1, \ldots, y_n$):

$$\varphi^{(n)}_{k(e,x)}(\vec{y}) \simeq f(e,x,y_1,\vec{y}).$$

It is easy to see that

$$\begin{aligned} W_e(x) &\to \varphi^{(n)}_{k(e,x)} \text{ is finite} \\ \neg W_e(x) &\to \varphi^{(n)}_{k(e,x)} = Z. \end{aligned}$$

From this the property (1) follows. By the corollary of Thm. 7.2.13, the predicate Tot_n is not semicomputable.

7.2.17 Theorem (Rice-Shapiro). *For every $n \geq 1$ and every class $\mathcal{F}$ of n-ary partially computable functions, if the problem $\varphi^{(n)}_e \in \mathcal{F}$ is semidecidable then the following holds for every n-ary partially computable function f:*

$$f \in \mathcal{F} \textit{ iff there is a finite } \theta \subseteq f \textit{ such that } \theta \in \mathcal{F}. \tag{1}$$

Proof. Take any n-ary partially computable function $f \in \mathcal{F}$. We wish to find a finite $\theta \subseteq f$ such that $\theta \in \mathcal{F}$ holds. Consider the $(n+3)$-ary partially computable function g defined by

$$g(e,x,t,\vec{y}) \simeq \begin{cases} \bot & \text{the computation of } \varphi_e(x) \text{ converges in} \leq t \text{ steps,} \\ f(\vec{y}) & \text{otherwise.} \end{cases}$$

By the s-m-n theorem there is a primitive recursive function $k(e,x)$ such that $\varphi^{(n)}_{k(e,x)}(\vec{y}) \simeq g(e,x,y_1,\vec{y})$. It is easy to see that $\varphi^{(n)}_{k(e,x)} \subseteq f$ and

$$\begin{aligned} W_e(x) &\to \varphi^{(n)}_{k(e,x)} \text{ is finite} \\ \neg W_e(x) &\to \varphi^{(n)}_{k(e,x)} = f. \end{aligned}$$

Now if $\varphi^{(n)}_{k(e,x)}$ were not in $\mathcal{F}$ then we would have

$$\neg W_e(x) \leftrightarrow \varphi^{(n)}_{k(e,x)} \in \mathcal{F}^{(n)}$$

and thus the predicate $\neg W_e(x)$ would be semicomputable which it is not. So it must be the case $\varphi^{(n)}_{k(e,x)} \in \mathcal{F}$ and it suffices to take $\varphi^{(n)}_{k(e,x)}$ for θ.

Vice versa, if $\theta \in \mathcal{F}$ is finite such that $\theta \subseteq f$ for an n-ary partially computable function f we wish to show that $f \in \mathcal{F}$. Consider the $(n+2)$-ary partially computable function g defined by

$$g(e, x, \vec{y}) \simeq \begin{cases} f(\vec{y}) & \text{if } \theta(\vec{y})\downarrow \text{ or } \varphi_e(x)\downarrow, \\ \perp & \text{otherwise.} \end{cases}$$

By the s-m-n theorem there is a primitive recursive function $k(e, x)$ such that $\varphi^{(n)}_{k(e,x)}(\vec{y}) \simeq g(e, x, \vec{y})$. It is easy to see that we have

$$W_e(x) \rightarrow \varphi^{(n)}_{k(e,x)} = f$$
$$\neg W_e(x) \rightarrow \varphi^{(n)}_{k(e,x)} = f|\operatorname{dom}(\theta) = \theta.$$

Now if f were not in $\mathcal{F}$ then we would have

$$\neg W_e(x) \leftrightarrow \varphi^{(n)}_{k(e,x)} \in \mathcal{F}^{(n)}$$

and thus the predicate $\neg W_e(x)$ would be semicomputable which it is not. So it must be the case $f \in \mathcal{F}$. $\square$

7.2.18 Application of the Rice-Shapiro's theorem. We apply the previous theorem to show that the equivalence problem $\varphi^{(n)}_e = g$ for n-ary partially computable functions is not semidecidable. Let $\mathcal{F}$ be the set of all n-ary partially computable functions $\varphi^{(n)}_e$ such that $\varphi^{(n)}_e \in \mathcal{F} \leftrightarrow \varphi^{(n)}_e = g$. Clearly $\mathcal{F} = \{g\}$. It is easy to see that $\mathcal{F}$ does not satisfy the property 7.2.17(1). Indeed, either g is finite and then the $(\leftarrow)$-direction of the property does not hold, or the domain of g is infinite and then the $\mathcal{F}$ violates $(\rightarrow)$-direction of the property. Thus, by the Rice-Shapiro's theorem, the problem $\varphi^{(n)}_e \in \mathcal{F}$ is not semidecidable and so is the equivalence problem $\varphi^{(n)}_e = g$.

7.2.19 Alternative characterization of semidecidable predicates. In the next theorem we give two characterization of semicomputable predicates. The first one is by existential projections of decidable predicates; the second by predicates which can be effectivelly enumerate by recursive functions.

7.2.20 Σ_1-predicates. An n-ary predicate R is a *Σ_1-predicate* if R is the *existential projection* of an $(n+1)$-ary recursive predicate Q, i.e. the following holds for all $\vec{x}$:

$$R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x}).$$

7.2.21 Recursively enumerable predicates. An n-ary predicate R is a *recursively enumerable predicate* if it is empty or if it can be *enumerated* by a unary recursive function f, i.e. the following holds for all $\vec{x}$:

$$R(\vec{x}) \leftrightarrow \exists y\, f(y) = (\vec{x}).$$

7.2.22 Theorem. *The following classes are equivalent:*

(i) semicomputable predicates,
(ii) Σ_1-predicates,
(iii) recursively enumerable predicates.

Proof. If R is an n-ary semicomputable predicate then by the normal form theorem there is a number e such that $R(\vec{x}) \leftrightarrow \exists y\, \mathrm{T}_n(e, \vec{x}, y)$. Let Q be an $(n+1)$-ary predicate defined explicitly as a primitive recursive predicate by $Q(y, \vec{x}) \leftrightarrow \mathrm{T}_n(e, \vec{x}, y)$. We then have $R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x})$ and thus R is a Σ_1-predicate.

Let R be an n-ary Σ_1-predicate, i.e. we have $R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x})$ for some recursive predicate Q. If R is empty then there is nothing to prove. Otherwise, there are numbers $\vec{x}_0$ such that $R(\vec{x}_0)$ holds. We define a unary function f explicitly as a recursive function by:

$$f(z) = (\vec{x}) \leftarrow z = y, (\vec{x}) \wedge Q(y, \vec{x})$$
$$f(z) = (\vec{x}_0) \leftarrow \neg\exists y \exists \vec{x}\big(z = y, (\vec{x}) \wedge Q(y, \vec{x})\big).$$

We claim that f enumerates R, i.e. that we have $R(\vec{x}) \leftrightarrow \exists z\, f(z) = (\vec{x})$. If $R(\vec{x})$ holds then we have $Q(y, \vec{x})$ for some y and it suffices to take $z = y, (\vec{x})$. Vice versa, if $f(z) = (\vec{x})$ then we have to show that $R(\vec{x})$ holds. Consider two cases. Either there is a number y such that $z = y, (\vec{x})$ and $Q(y, \vec{x})$ holds and then also $R(\vec{x})$ holds. If there is no such number then $f(z) = (\vec{x}_0) = (\vec{x})$ and we have $R(\vec{x})$ by the selection of $\vec{x}_0$.

Let R be an n-ary recursively enumerable predicate. If R is empty then it is trivially semicomputable since it is the domain of the nowhere defined partially computable function $\emptyset^{(n)}$. Otherwise, there is a unary recursive function f enumerating R, i.e. $R(\vec{x}) \leftrightarrow \exists y\, f(y) = (\vec{x})$. Consider the n-ary partially computable function g defined by $g(\vec{x}) \simeq \mu_y[f(y) = (\vec{x})]$. We clearly have $R(\vec{x}) \leftrightarrow g(\vec{x})\!\downarrow$ and thus R is a semicomputable predicate. □

7.2.23 The uniform halting problem is not semidecidable (2nd proof). We give here an alternative proof of the fact that the uniform halting predicates Tot_n are not semicomputable predicates. The proof is based on the characterization of semicomputable predicates by recursive enumerability.

So suppose, by contradiction, that the predicate Tot_n is semicomputable. Then by Thm. 7.2.22 there is a recursive function k enumerating Tot_n, i.e. we have $Tot_n(e) \leftrightarrow \exists y\, k(y) = e$. Consider $(n+1)$-ary partially computable function f explicitly defined by $f(y, \vec{x}) \simeq \varphi^{(n)}_{k(y)}(\vec{x})$. The f is total since $k(y)$ are indices of (total) computable functions for all y. Consequently, the n-ary function g explicitly defined by $g(x_1, \ldots, x_n) = f(x_1, x_1, \ldots, x_n)$ is computable. Then there is a number y_0 such that $k(y_0)$ is an index of g. We obtain a contradiction as follows ($\vec{y}_0 \equiv y_0, \ldots, y_0$):

$$g(\vec{y}_0) \stackrel{\text{def}}{=} f(y_0, \vec{y}_0) \stackrel{\text{def}}{\simeq} \varphi^{(n)}_{k(y_0)}(\vec{y}_0) + 1 \stackrel{\text{index}}{\simeq} g(\vec{y}_0) + 1.$$

7.2.24 Σ_1-formulas. *Σ_1-formulas* are formulas constructed from *atomic* formulas: $\tau_1 < \tau_2$, $\tau_1 \leq \tau_2$, $\tau_1 > \tau_2$, $\tau_1 \geq \tau_2$, $\tau_1 = \tau_2$, $\tau_1 \neq \tau_2$, and $Q(\vec{\tau})$ by propositional connectives $\wedge$ and $\vee$, by bounded quantification, and by existential quantification. The terms of bounded formulas are composed from variables and constant symbols by applications of (total) functions.

7.2.25 Explicit definitions of predicates with Σ_1-formulas. *Explicit definitions* of predicates *with Σ_1-formulas* are of a form

$$R(\vec{x}) \leftrightarrow \phi[\vec{x}], \tag{1}$$

where ϕ is a Σ_1-formula with at most the indicated n-tuple of variables free and without any application of the predicate symbol R.

7.2.26 Lemma. *If Q, Q_1, and Q_2 are semicomputable predicates and every term below contains only applications of computable functions then each of the following definition defines a semicomputable predicate:*

$$R(\vec{x}) \leftrightarrow Q(\vec{\tau}[\vec{x}]) \tag{1}$$
$$R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x}) \tag{2}$$
$$R(\vec{x}) \leftrightarrow Q_1(\vec{x}) \wedge Q_2(\vec{x}) \tag{3}$$
$$R(\vec{x}) \leftrightarrow Q_1(\vec{x}) \vee Q_2(\vec{x}) \tag{4}$$
$$R(\vec{x}) \leftrightarrow \exists y \leq \tau[\vec{x}]\, Q(y, \vec{x}) \tag{5}$$
$$R(\vec{x}) \leftrightarrow \forall y \leq \tau[\vec{x}]\, Q(y, \vec{x}). \tag{6}$$

Proof. Suppose that the n-ary predicate R is defined by (1) from an m-ary semicomputable predicate Q. By definition the predicate Q is the domain of an m-ary partially computable function f. We then have $Q(\vec{\tau}) \leftrightarrow f(\vec{\tau})\!\downarrow$ and thus the predicate R is the domain of the n-ary partially computable function g explicitly defined by $g(\vec{x}) \simeq f(\vec{\tau}[\vec{x}])$.

Suppose that the n-ary predicate R is defined by the existential projection (2) of an $(n+1)$-ary semicomputable predicate Q. By Thm. 7.2.22 we have

$$Q(y, \vec{x}) \leftrightarrow \exists z\, Q_1(z, y, \vec{x}) \tag{7}$$

for some recursive predicate Q_1. We further have

$$R(\vec{x}) \overset{(2)}{\Leftrightarrow} \exists y\, Q(y, \vec{x}) \overset{(7)}{\Leftrightarrow} \exists y \exists z\, Q_1(z, y, \vec{x}) \overset{(*)}{\Leftrightarrow} \exists w\, Q_1(T(w), H(w), \vec{x}). \tag{8}$$

The step marked by $(*)$ is called *contraction of quantifiers*. Consider now an $(n+1)$-ary recursive predicate R_1 explicitly defined by

$$R_1(w, \vec{x}) \leftrightarrow Q_1(T(w), H(w), \vec{x}).$$

From (8) we can see that $R(\vec{x}) \leftrightarrow \exists w\, R_1(w, \vec{x})$ holds. The predicate R thus is a Σ_1-predicate and hence, by Thm. 7.2.22, a semicomputable predicate.

Suppose that the n-ary predicate R is defined by disjunction (4) of n-ary semicomputable predicates Q_1 and Q_2. By Thm. 7.2.22 we have

$$Q_1(\vec{x}) \leftrightarrow \exists y\, P_1(y, \vec{x}) \tag{9}$$
$$Q_2(\vec{x}) \leftrightarrow \exists y\, P_2(y, \vec{x}) \tag{10}$$

for some recursive predicates P_1 and P_2. We further have

$$\begin{aligned} R(\vec{x}) &\overset{(4)}{\Leftrightarrow} Q_1(\vec{x}) \lor Q_2(\vec{x}) \overset{(9),(10)}{\Leftrightarrow} \exists y\, P_1(y, \vec{x}) \lor \exists y\, P_1(y, \vec{x}) \Leftrightarrow \\ &\Leftrightarrow \exists y_1 \exists y_2 \big(P_1(y_1, \vec{x}) \lor P_2(y_2, \vec{x})\big). \end{aligned} \tag{11}$$

Consider now an $(n+2)$-ary recursive predicate P explicitly defined by

$$P(y_2, y_1, w, \vec{x}) \leftrightarrow P_1(y_1, \vec{x}) \lor P_2(y_2, \vec{x}).$$

From (11) we can see that $R(\vec{x}) \leftrightarrow \exists y_1 \exists y_2\, P(y_2, y_1, \vec{x})$. Now it suffices to apply (2) and Thm. 7.2.22 and we obtain that R is a semicomputable predicate.

Suppose that the n-ary predicate R is defined by the universal bounded quantification (6) of an $(n+1)$-ary semicomputable predicate Q. By Thm. 7.2.22 we have

$$Q(y, \vec{x}) \leftrightarrow \exists z\, Q_1(z, y, \vec{x}) \tag{12}$$

for some recursive predicate Q_1. We further have

$$\begin{aligned} R(\vec{x}) &\overset{(6)}{\Leftrightarrow} \exists y \le \tau\, Q(y, \vec{x}) \overset{(12)}{\Leftrightarrow} \exists y \le \tau \exists z\, Q_1(z, y, \vec{x}) \Leftrightarrow \\ &\Leftrightarrow \exists z \exists y \le \tau\, Q_1((z)_y, y, \vec{x}). \end{aligned} \tag{13}$$

Consider now an $(n+1)$-ary recursive predicate R_1 explicitly defined by

$$R_1(z, \vec{x}) \leftrightarrow \exists y \le \tau\, Q_1((z)_y, y, \vec{x}).$$

From (13) we can see that $R(\vec{x}) \leftrightarrow \exists z\, R_1(z, \vec{x})$ holds. The predicate R thus is a Σ_1-predicate and hence, by Thm. 7.2.22, a semicomputable predicate.

The remaining cases are similar and left to the reader. □

7.2.27 Theorem. *Semicomputable predicates are closed under explicit definitions of predicates with Σ_1-formulas.*

Proof. We show that semicomputable predicates are closed under explicit definitions of n-ary predicates with Σ_1-formulas of a form $R(\vec{x}) \leftrightarrow \phi[\vec{x}]$ by induction on the structure of ϕ.

If ϕ is one of $\tau_1 \le \tau_2$, $\tau_1 < \tau_2$, $\tau_1 \ge \tau_2$, $\tau_1 > \tau_2$, $\tau_1 = \tau_2$, or $\tau_1 \ne \tau_2$ then R is computable and hence, Thm. 7.2.3, a semicomputable predicate. If $\phi \equiv Q(\vec{\tau})$ then the claim follows from Lemma 7.2.26.

If $\phi \equiv \phi_1 \wedge \phi_2$ we obtain two auxiliary n-ary semicomputable predicates $Q_1(\vec{x}) \leftrightarrow \phi_1[\vec{x}]$ and $Q_2(\vec{x}) \leftrightarrow \phi_2[\vec{x}]$ by IH. We have $R(\vec{x}) \leftrightarrow Q_1(\vec{x}) \wedge Q_2(\vec{x})$ and it suffices to use Lemma 7.2.26.

If $\phi \equiv \exists y \leq \tau\, \psi$ we use IH and define an $(n+1)$-ary semicomputable predicate Q by: $Q(y, \vec{x}) \leftrightarrow \psi[y, \vec{x}]$. We have $R(\vec{x}) \leftrightarrow \exists y \leq \tau\, Q(y, \vec{x})$ and it suffices to use Lemma 7.2.26.

If $\phi \equiv \exists y\, \psi$ we use IH and define an $(n+1)$-ary semicomputable predicate Q by explicit definition: $Q(y, \vec{x}) \leftrightarrow \psi[y, \vec{x}]$. We have $R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x})$ and it suffices to use Lemma 7.2.26.

The remaining cases are similar and left to the reader. □

7.2.28 Arithmetic Σ_1-formulas and Σ_1-definable predicates. Arithmetic Σ_1-formulas are Σ_1-formulas containing only applications of the successor function $x + 1$, addition $x + y$, and multiplication $x \cdot y$. Predicates defined by arithmetic Σ_1-formulas are called *Σ_1-definable predicates*.

7.2.29 Lemma. *If the graph of a function is a Σ_1-definable predicate then so is its complement.*

Proof. Suppose that an arithmetic Σ_1-formula $\phi[\vec{x}, y]$ defines the graph of an n-ary function f. We have

$$f(\vec{x}) \not\asymp y \overset{(*)}{\Leftrightarrow} \exists z(z \neq y \wedge f(\vec{x}) \asymp z) \Leftrightarrow \exists z(z \neq y \wedge \phi[\vec{x}, z]).$$

Thus the last formula is an arithmetic Σ_1-formula defining the complement of the graph of f. Note that in the step marked by $(*)$ the assumption about totality of f is crucial; the lemma does not generalise to partial functions. □

7.2.30 Lemma. *Graphs of recursive functions are Σ_1-definable predicates.*

Proof. Recall that by Thm. 5.3.29 recursive functions are exactly μ_a-recursive functions. The latter is the class of functions generated from the identity functions $I_i^n(\vec{x}) = x_i$, the multiplication function $x \cdot y$, and from the characteristic function $x <_* y$ of the less than predicate $x < y$ by composition and regular minimalization of functions. So it suffices to show that the graphs of μ_a-recursive functions are Σ_1-definable predicates.

We prove the claim by induction on construction of μ_a-recursive functions. That the graphs of the initial μ_a-recursive functions are Σ_1-definable can be seen from the following equivalences:

$$\begin{aligned} I_i^n(\vec{x}) \asymp y &\leftrightarrow y = x_i \\ x_1 \cdot x_2 \asymp y &\leftrightarrow y = x_1 \cdot x_2 \\ (x_1 <_* x_2) \asymp y &\leftrightarrow x_1 < x_2 \wedge y = 1 \vee x_1 \geq x_2 \wedge y = 0. \end{aligned}$$

Suppose now that a μ_a-recursive function f is obtained by the composition $f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_m(\vec{x}))$ of μ_a-recursive functions. By IH there are arithmetic Σ_1-formulas $\phi_h[\vec{z}, y], \phi_{g_1}[\vec{x}, z_1], \ldots, \phi_{g_m}[\vec{x}, z_m]$ defining the graphs of the corresponding functions. We then have ($\vec{z} \equiv z_1, \ldots, z_m$):

$$f(\vec{x}) \asymp y \Leftrightarrow \exists \vec{z}(\bigwedge_{i=1}^{m} g_i(\vec{x}) \asymp z_i \wedge h(\vec{z}) \asymp y) \Leftrightarrow \exists \vec{z}(\bigwedge_{i=1}^{m} \phi_{g_i}[\vec{x}, z_i] \wedge \phi_h[\vec{z}, y])$$

and therefore the last formula is an arithmetic Σ_1-formula defining the graph of the function f.

Suppose finally that a μ_a-recursive function f is obtained by the regular minimalization $f(\vec{x}) = \mu_y[g(y, \vec{x}) = 1]$ of a μ_a-recursive function g. By IH there is an arithmetic Σ_1-formula $\phi[y, \vec{x}, z]$ defining the graph of g. By Lemma 7.2.29, there is an arithmetic Σ_1-formula $\phi^c[y, \vec{x}, z]$ defining the complement of the graph of g. We have

$$f(\vec{x}) \asymp y \Leftrightarrow g(y, \vec{x}) \asymp 1 \wedge \forall z < y\, g(z, \vec{x}) \not\asymp 1 \Leftrightarrow \phi[y, \vec{x}, 1] \wedge \forall z < y\, \phi^c[z, \vec{x}, 1]$$

and therefore the last formula is an arithmetic Σ_1-formula defining the graph of the function f. □

7.2.31 Theorem. *Semicomputable predicates are exactly Σ_1-definable predicates.*

Proof. By Thm. 7.2.27 every Σ_1-definable predicate is semicomputable. Vice versa, if $R(\vec{x})$ is an n-ary semicomputable predicate then, by Thm. 7.2.22, it is the existential projection of some $(n + 1)$-ary recursive predicate $Q(y, \vec{x})$. By Lemma 7.2.30, the graph of the characteristic function Q_* of Q is Σ_1-definable, i.e. we have $Q_*(y, \vec{x}) = z \leftrightarrow \phi[y, \vec{x}, z]$ for some arithmetic Σ_1-formula $\phi[y, \vec{x}, z]$. Note that we then have

$$R(\vec{x}) \Leftrightarrow \exists y\, Q(y, \vec{x}) \Leftrightarrow \exists y\, Q_*(y, \vec{x}) = 1 \Leftrightarrow \exists y\, \phi[y, \vec{x}, 1]$$

and thus the arithmetic Σ_1-formula $\exists y\, \phi[y, \vec{x}, 1]$ defines the predicate R. □

7.3 Arithmetical Hierarchy

See the section 7.2 in [Vod00].

7.4 Degrees of Unsolvability

7.5 Exercises

Decidable and Undecidable Predicates

7.5.1 Exercise. Show without the use of the Rice's theorem that the following problems are undecidable:

(i) Given $n \geq 1$, the problem of deciding, for any e, whether $\varphi_e^{(n)}$ has a non-empty domain.

(ii) Given $n \geq 1$, the problem of deciding, for any e, whether the domain of $\varphi_e^{(n)}$ is empty.

(iii) Given $n \geq 1$, the problem of deciding, for any e, whether $\varphi_e^{(n)}$ has an infinite domain.

(iv) Given $n \geq 1$, the problem of deciding, for any e_1 and e_2, whether $\mathrm{dom}(\varphi_{e_1}^{(n)}) = \mathrm{dom}(\varphi_{e_2}^{(n)})$.

Semidecidable Predicates

Arithmetical Hierarchy

Degrees of Unsolvability

GKP89

[Ack28] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[Ben62] J. H. Bennett. *On Spectra*. PhD thesis, Princeton University, 1962.

[BJ74] G. S. Boolos and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, 1974.

[Dav58] M. Davis. *Computability and Unsolvability*. McGraw-Hill, 1958.

[GKP89] R. L. Graham, D. F. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison Wesley, 1989.

[Her65] H. Hermes. *Enumerability, Decidability, Computability*. Springer Verlag, 1965.

[HP93] P. Hájek and P. Pudlák. *Metamathematics of First-Order Arithmetic*. Springer Verlag, 1993.

[Kle36] S. C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. Wolters-Noordhoff and North-Holland, 1952.

[KV01] J. Komara and P. J. Voda. Metamathematics of Computer Programming, 2001. Unpublished manuscript. Available through WWW from http://dent.ii.fmph.uniba.sk/~komara/meta.ps.gz.

[Pét35] R. Péter. Konstruktion nichtrekursiver Funktionen. *Mathematische Annalen*, 111:42–60, 1935.

[Pét36] R. Péter. Über die mehrfache Rekursion. *Mathematische Annalen*, 113:489–527, 1936.

[Pét67] R. Péter. *Recursive Functions*. Academic Press, 1967.

[Rob49] J. Robinson. Definability and decision problems in arithmetic. *Journal of Symbolic Logic*, 14:98–114, 1949.

[Ros82] H. E. Rose. *Subrecursion: Functions and Hierarchies*. Number 9 in Oxford Logic Guides. Clarendon Press, Oxford, 1982.

[Sho67] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[Smu61] R. M. Smullyan. *Theory of Formal Systems*. Princeton University Press, 1961.

[Smu92] R. M. Smullyan. *Gödel's Incompleteness Theorems*. Oxford Logic Guides. Oxford University Press, 1992.

[Tai61] W. W. Tait. On nested recursion. *Mathematische Annalen*, 143:236–250, 1961.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proc. London Math. Soc.*, volume 42 of *2*, pages 230–265, 1936.

[Vod95] P. J. Voda. Subrecursion as a basis for a feasible programming language. In L. Pacholski and J. Tiuryn, editors, *Proceedings of CSL'94*, number 933 in LNCS, pages 324–338. Springer Verlag, 1995.

[Vod00] P. J. Voda. Theory of Recursive Functions & Computability (from Computer Programmer's View), 2000. Available through WWW from http://dent.ii.fmph.uniba.sk/∼voda/text.ps.gz.

Index of Notation

Index