Theory of Recursive Functions & Computability from Computer Programmer's View

Paul J. Voda
Comenius University Bratislava

March 28, 2000

Contents

Introduction

We use this text in a third/fourth year computer science course in the theory of computability. Although our development of the subject matter is standard, our approach is based on the systematic use of a programming language and proof system CL (Clausal Language) for the definition of computable functions.

Although we accept the Thesis of Turing which 'defines' effectively computable functions as those computed by *Turing machines* because of its great intuitive appeal, we do not base our programming language on programs for Turing machines. Such programs are basically recipes (not unlike the kitchen variety) describing the sequencing of instructions which manipulate the computer store. As it happens, the major stream in computer programming languages - starting from *Fortran* through *Algol-60* and *Pascal*, to today's two most used languages C^{++} and *Java* consists of languages based on this *imperative* paradigm. The problem with the imperative languages is that they are hard to reason about. Although there are mathematical tools for proving imperative programs correct (for instance, the pre- and post- conditions of Hoare), there is probably not a single computer implemented system for programming and proving of imperative programs which can be used in computer science classes.

We base instead our programming language on a model of computability which actually predates the Turing machines and is today known as *Herbrand-Gödel-Kleene* systems of recursive equations. Such systems serve a dual role. First of all, they define (if correct) *recursive* functions over natural numbers and so they deal with mathematical objects whose properties can be proved in the formal system of elementary (Peano) arithmetic. Secondly - and even more importantly - the equations can be directly used to guide effective computations of functions being defined. The basic idea that the definitions are recursive, i.e. that the functions are defined be means of recurrences, is very natural to computer programmers. Moreover, a growing stream of programming languages - starting from (the pure) *Lisp* through *ML* to today's dominant declarative programming language *Haskell* are based on this *declarative* paradigm. Most of the declarative programming languages are high-order in that they admit functionals into their ontology. CL, and actually also pure Lisp, is based on type one functions over natural numbers and S-expressions respectively.

There are currently about six computer implemented proof systems supporting proofs of properties of declarative programs and all except CL and the system of Boyer and Moore [4] are high order. The advantage of CL over the system of Boyer and Moore is that CL defines and proves computable functions over natural numbers and it is standard in this respect, while the system of Boyer and Moore works with S-expressions.

Declarative programming, in addition to being mathematically sound, is quickly becoming practical. For instance, the CL system, which is a large one (200K lines of source code), is completely written in the declarative programming language *Trilogy* designed and implemented by the author. The design and implementation of CL is a joint project of Ján Komara and the present author.

The development of the theory of computability does not need programs written in a rigorusly defined programming language because the programs can quickly become unreadable. The detailed programming of Turing machines is next to impossible and the original equations of Herbrand-Gödel-Kleene are not much more readable. The informal, but rigorous, style of discourse which was pioneered by Rogers in his well-known 1967 text [16] is today almost universally used. The present text is an exception and uses a formal computer programming language CL. There are two reasons for this. The first one is that the language is implemented on computers and it is used for exercises. Computers, as is well known, cannot work in an informal style of discourse. The second reason is that the modern declarative programming languages which introduced the so called *pattern matching* style of definitions are very readable. The clauses of CL are Horn formulas directly expressing the properties of functions being defined.

There are three aspects of computability of which the first two are of interest to computer scientists and the last two to logicians:

1. the study of concrete computable functions operating on concrete data structures,
2. models of computability and elementary recursion theory,
3. advanced recursion theory, relative computability, degrees, arithmetic and analytic hierarchies.

The aspect (1) is trivial to logicians who, instead of writing down concrete functions, discourse informally and then appeal to the thesis of Church-Turing. On the other hand, this aspect is of central interest to students of computer science because this is what computer programmers do for living. Trivial as this aspect may be to mathematicians, it has a rich texture in computer science which includes the design of efficient data structures and algorithms, complexity theory, and the theory of compiler construction.

The aspect (2) is of joint interest both to computer scientists and mathematicians. The former can use it as the basis for the semantics of programming languages. (this is what we are doing with CL). The latter must develop the topics from (2) because of the key definitions which are then used in the study of topics from the area (3). The last area is beyond the scope of computer science but it is central to recursion theory.

We use the present text for a one-term third/four year introductory course to the theory of computability. Our students are already quite familiar with the programming language CL and with its proof system because they have used it in three preceding courses: *Introduction to Declarative Programming* (given in the second term), *Predicate Logic* (given in the third term), and *Specification and Verification of Computer Programs* (given in the fourth term). Although our students are quite skilled in the reasoning about declarative programs both formally (in CL) and informally (with pencil and paper) this ascpect of CL is not prerequisite to the course in the theory of computability. However, some knowledge of CL is expected in the course because the students work with the CL system in computer labs which complement the theoretical lectures.

In order to make this text self-contained we have included a large chapter 1 discussing the basic facts about CL. This chapter (which we skip in the course on the theory of computability) discusses the domain of natural numbers and the way the data structures are arithmetized (coded) in it. The chapter extensively uses CL-definitions in a self-explanatory way without burdening the reader with the syntax and semantics of CL. The chapter also discusses in detail the issues of efficient computation of CL-definitions. Incidentally, this is the only part of the text where *efficient* computations play a role. The remainder of the text deals with functions which are computable in principle, i.e. with *effective* computations.

The chapter 2 contains the central definitions of (partial) recursive functions in the style of Herbrand-Gödel-Kleene and discusses in detail effective computations of these functions. The chapter also serves as metamathematics, i.e. semantics, of CL. To this end the chapter contains some quite technical theorems with detailed proofs which deal with the terms and formulas used in CL-definitions. We cover the definitions from the chapter in the classes but discuss the metamathematical theorems only informally because, fortunately, they are quite intuitive.

The remaing chapters of the text are standard in its scope. We cover explicit definitions in chapter 3, primitive recursive functions in chapter 4, we characterize the recursive functions by μ-recursion in chapter 5. We cover the Turing computability and the basics of recursion theory in chapter 6. In the final chapter 7 we discuss the recursively eumerable predicates and the arithmetic hierarchy.

We think that the development of μ-recursive functions is improved by our own contribution in Par. 5.2.13 where we give a new and simple Σ_0-definition of the graph of exponentiation $2^x = y$. The graph is then used to show in Thm. 5.2.14 the closure of μ-recursive functions under primitive recursion without the usual device of coding by means of Gödel's β-function.

The major difference with the standard presentation of theory of computability is that we systematically use the language CL to define the computable functions and predicates. To that end we prove in Thm. 4.3.5 the closure of primitive recursive functions under a powerful and liberal scheme of recursion: *regular (course of values) definitions with primitive recursive measures.*

CL offers a great comfort in the development of the subject matter. It also makes the subject of computability accessible and interesting to the students because, by doing exercises in CL in computer labs, they can construct and execute the functions and predicates discussed in the text. We consider this practical aspect of computability afforded by our use of CL as indispensable in giving the students intuition which they may miss in a standardly presented course.

This text would not be possible without the invaluable contribution toward the design and implementation of the CL system by Ján Komara. In addition to this, Ján has greatly contributed to the text in the form of long discussions we have had on the subject. Lastly, but by no means leastly, he was instrumental in the development of carefully prepared problem sets of exercises and assignments for the four courses based on CL.

Chapter 1

Overview of CL

This chapter is included in order to make the text self-contained. Those who are familiar with CL and with the way the usual data structures are coded in its domain of natural numbers can skip it.

In Sect. 1.1 we study a few typical algorithms (programs) operating on natural numbers and discuss the relation between the imperative and declarative styles of programming. The presented programs compute in the monadic (successor) representation of natural numbers which, although standard in the theory of computability, is rather bad from the point of view of computational complexity. The situation is improved in Sect. 1.2 where we present numeric programs computing in efficient logarithmic representation of natural numbers. We have chosen the dyadic over the usual binary representation because the former can better code words over finite alphabets. In Sect. 1.3 we introduce another logarithmic representation of natural numbers by pairing which is in the style of S-expressions of *Lisp*.

Section 1.4 discusses the arithmetization (coding) of data structures usually found in declarative programming languages. Although we informally use definitions in the programming language CL throughout the chapter, it is in this section where the most interesting examples are given. The aim is to make the reader not knowing CL familiar with its style before we formally go into the metamathematical issues of its denotational and operational semantics in Chap. 2.

Section 1.5 discusses the efficiency issues of computing CL-definitions in a mixed representation combining the dyadic and pairing representations. The mixed representation is crucial to a good implementation of CL on computers.

1.1 Natural Numbers

1.1.1 Functions. If we do not state the domain of a function explicitly we mean an n-ary function over the domain of natural numbers $\mathbb{N}$ with $n \geq 1$.

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

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

Instead of division we use the binary *integer division* $x \div y$ function, which together with the binary *remainder* function $x \bmod y$, satisfies:

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

1.1.3 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 $Pr(\tau)$.

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

$$I(x_1, \ldots, x_i, \ldots, x_n) = x_i \ .$$

As a special case we set $I = I^1_1$ and so we have $I(x) = x$.

1.1.5 Constant functions. *Constant functions* are the unary functions C_n yielding n, i.e. such that $C_n(x) = n$ for every natural number x. We customarily denote the function C_0 by Z, i.e. we have $Z(x) = 0$, and call it the *zero function*.

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

$$D(0, y, z) = z$$
$$D(x + 1, y, z) = y \ .$$

Computer programmers may visualize $D(x, y, z)$ as

$$\textbf{if } x > 0 \textbf{ then } y \textbf{ else } z \ .$$

1.1.7 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.

1.1.8 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 $R^c(\vec{x}) \leftrightarrow \neg R(\vec{x})$.

Characteristic function of an n-ary predicate R is the n-ary function R_* such that $R_*(\vec{x}) = 1$ if $R(\vec{x})$ holds and $R_*(\vec{x}) = 0$ otherwise.

The *graph* of an n-ary function f is the $(n+1)$-ary predicate $f(\vec{x}) = y$.

1.1.9 Some conventions. We let all binary propositional connectives in logical formulas group to the right. We assign the highest precedence to the quantifiers $\forall x$, $\exists y$, and to negation $\neg$. Next lower precedence has the conjuction $\wedge$ and then the disjunction $\vee$. The connectives of implication $\rightarrow$ and equivalence $\leftrightarrow$ have the lowest precedence.

We abbreviate $\neg\tau_1 = \tau_2$ to $\tau_1 \neq \tau_2$. We use the same convention with some binary infix relations where, for instance, we write $\tau_1 \not\varepsilon\, \tau_2$ as an abbreviation for $\neg\tau_1 \;\varepsilon\; \tau_2$.

The operators $\cdot$, $\div$, and mod have the same precedence which is higher than that of the operators $+$, $\dot{-}$, $\star$, $\oplus$ and which have the same precedence. The infix operators like $+$ or $\dot{-}$ group to the left. Thus $\tau_1 \dot{-} \tau_2 \dot{-} \tau_3$ abbreviates $(\tau_1 \dot{-} \tau_2) \dot{-} \tau_3$.

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, $f(\vec{\tau})$ wil be an abbreviation for $f(\tau_1, \cdots, \tau_n)$ where $\tau_1, \ldots, \tau_n$ is a sequence of terms. We will also write $f\,g(\vec{\tau})$ instead of $f(g(\vec{\tau}))$ and $R\,f(\vec{\tau})$ instead of $R(f(\vec{\tau}))$.

1.1.10 The algorithm of Euclides. One of the oldest and best known algorithms is the algorithm of Euclides which finds the greatest common divisor of two natural numbers x and y. The algorithm can be expressed by a program in the style of Pascal:

while $y > 0$ *do*
  $z := x \bmod y;\ x := y;\ y := z;$

The computation always terminates with the greatest common divisor of the initial values of the variables x and y stored at the end of computation in x.

1.1.11 Imperative versus declarative programming. The realization of the Euclidian algorithm from the previous paragraph is by an *imperative program.* This term is used in computer science for the style of programming where programs modify memory by obeying sequences of commands. The imperative program for *gcd* illustrates two problems inherent in such a style of programming. The first problem is how to prove termination of the program and the second one is how to capture the relation between the input and output when the contents of variables change during the execution of the program. In the last twenty years many, more or less complex, formal systems were introduced in order to facilitate the answers to these questions.

The idea of proving the correctness of an algorithm by utilizing the simplest formalism of all, i.e. the *elementary arithmetic* (Peano arithmetic), actually predates the imperative style of programming by some twenty years. Theory of recursive functions, which was originally developed for the purposes of mathematical logic, can be used to demonstrate the computability of the Euclidian algorithm realized as a two-argumented function over natural numbers. The formal system of elementary arithmetic then permits to prove properties of the function.

In the retrospective it seems rather surprising that the movement in computer programming away from the imperative style of programming towards this natural style of so-called *declarative programs* has found a wider acceptance only relatively recently. Declarative programs are usually functions over a sufficiently rich domain which contains not only natural numbers but also the so-called *data structures* we might wish to operate upon by computable functions.

The main reason for the relatively minor role of declarative programming is that imperative programming is deeply entrenched in practice and accounts for almost all programs being written today (witness the wide use of the programming language C^{++}). Declarative programming is used mostly at universities for educational purposes. Another reason is that to write an industrial quality compiler for a declarative programming language has been an extremely difficult problem satisfactorily solved only relatively recently (in the last ten years or so).

1.1.12 The Euclidian algorithm declaratively. The Euclidian algorithm can be captured by a binary function $gcd(x,y)$ yielding the greatest common divisor of x and y provided $x + y > 0$. The function satisfies the following recurrences:

$$gcd(x,0) = x \tag{1}$$
$$gcd(x,y+1) = gcd(y+1, x \bmod (y+1)) \ . \tag{2}$$

We note that $x \bmod y+1 < y+1$ and so the definition of gcd is by recursion decreasing the second argument in $<$. This guarantees that the use of recurrences for gcd with any numbers x and y results in an always terminating computation of the greatest common divisor. For instance, we have

$$gcd(12,21) \overset{(2)}{=} gcd(21,12) \overset{(2)}{=} gcd(12,9) \overset{(2)}{=} gcd(9,3) \overset{(2)}{=} gcd(3,0) \overset{(1)}{=} 3 \ .$$

Recursion going down in some measure answers the first problem of imperative programs, i.e. the problem of termination. The second problem, of what is actually computed by the function gcd is answered by the following properties:

$$x + y > 0 \rightarrow gcd(x,y) \mid x \land gcd(x,y) \mid y$$
$$x + y > 0 \land w \mid x \land w \mid y \rightarrow w \leq gcd(x,y)$$

where $y \mid x$ holds if y *divides* x, i.e. $\exists z\, x = y \cdot z$. We know that $z \mid x$ and $z \mid y$ iff $z \mid y$ and $z \mid (x \mod y)$. This is the basis for the proof by complete induction on y of the two properties in elementary arithmetic.

1.1.13 Fibonacci function. The function $\mathit{fib}(n)$ yielding the n-th element of the *sequence of Fibonacci* satisfies the following recurrences:

$$\mathit{fib}(0) = 0 \tag{1}$$
$$\mathit{fib}(1) = 1 \tag{2}$$
$$\mathit{fib}(x+2) = \mathit{fib}(x+1) + \mathit{fib}(x)\ . \tag{3}$$

The definition is by straightforward recursion decreasing the argument in $<$. We can directly use the recurrences for computations. For instance, we have

$$\mathit{fib}(4) \overset{(3)}{=} \mathit{fib}(3) + \mathit{fib}(2) \overset{2\times(3)}{=} \mathit{fib}(2) + \mathit{fib}(1) + \mathit{fib}(1) + \mathit{fib}(0) \overset{(3);2\times(2);(1)}{=}$$
$$\mathit{fib}(1) + \mathit{fib}(0) + 1 + 1 + 0 \overset{(2)(1)}{=} 1 + 0 + 2 = 3\ .$$

The only problem is that the computation sequence is too long. In order to compute the number $\mathit{fib}(x+2)$ one needs to use the recurrence (3) exactly $\mathit{fib}(x+2)$ times as can be proved by a simple induction proof. The Fibonacci function grows as fast as the exponential function and to compute the function in this way is simply too wasteful. The following Pascal-like program computes $\mathit{fib}(x)$ into the variable b:

```
a := 1; b := 0;
while x > 0 do
   x := x - 1; c := a; a := a + b; b := c;
```

The reader will note that the loop is executed only x times. This example is usually given as the 'standard argument' against declarative programming where the recursive version is clearly inferior to the imperative one. The argument is fallacious as one should define an auxiliary ternary function $\mathit{fiba}(x, a, b)$ with two *accumulators* a and b by recursion decreasing the first argument x:

$$\mathit{fiba}(0, a, b) = b \tag{4}$$
$$\mathit{fiba}(x+1, a, b) = \mathit{fiba}(x, a+b, a)\ . \tag{5}$$

By straightforward induction on x we can then prove:

$$\forall y\, \mathit{fiba}(x, \mathit{fib}(y+1), \mathit{fib}(y)) = \mathit{fib}(x+y)$$

and then explicitly define:

$$\mathit{fib}(x) = \mathit{fiba}(x, 1, 0)\ .$$

The number of recursions of fiba is exactly the same as the number of iterations of the loop of the imperative program.

1.1.14 Arguments in favor of declarative programming. We have seen in the previous paragraph that the accumulator function fiba can mimick the operations of the imperative program computing the Fibonacci function. It can be shown in general that to every imperative program consisting of one while-loop there is a function capturing the operation of the program with as many arguments as there are variables in the program. If the imperative program consists of more while-loops we define one function for each loop and then compose the functions. Thus there is no need to base computer programming on imperative programs. Without any loss we can use functions which are mathematical objects and so we can reason about the properties of functions by mathematical means.

1.1.15 Computations are operations on symbols. We have given above two examples of how to compute recursively defined functions. A few comments on the process of computation are needed. In order for a computation to be *effective* it must be performed mechanically by a computer (human or mechanical) following a finite number of instructions. The computer should operate on concrete objects. These are finite sequences of symbols in the case of human computation and a finite number of bits of a mechanical computer's memory.

The recurrences 1.1.12(1)(2) for *gcd* and 1.1.13(4)(5) for *fiba* can be used as instructions for a computer because they consist of finite sequences of symbols. The instructions operate on natural numbers which must be represented by finite sequences of symbols. In our examples of computation we have used the standard decimal representation of $\mathbb{N}$. Arithmetic operations on decimal numerals are performed by algorithms well-known from the elementary school.

The clausal definitions in CL thus have two different aspects. The *extensional* aspect is the definition of a function over $\mathbb{N}$ as a mathematical object and the *intensional* aspect is the determination of a *program* (rule) for computation of the function for concrete numbers as arguments.

1.1.16 Monadic representation of $\mathbb{N}$. The *monadic representation* of natural numbers is the accepted standard in recursion theory. Every natural number n has in it a unique representation as the *monadic numeral*: $S^n(0)$. Monadic numerals are finite sequences composed from symbols 0 and S standing for the number zero and the successor function respectively. Note that adding one to a monadic numeral τ means to form the monadic numeral $S(\tau)$. All other arithmetic operations are specified by primitive recursion (see Par. 1.1.18).

Monadic numerals have another advantage over decimal numerals. The latter are just sequences of decimal digits which must be associated to the numbers they stand for by some rule. Monadic numerals, while also being sequences of symbols, have more structure. Monadic numerals are called *terms* in logic and as such they *denote* (name) natural numbers by an *interpretation*.

1.1.17 Mathematical induction. Closely tied to the monadic representation of natural numbers is the *principle of (mathematical) induction* which is expressed for any unary predicate $R(x)$ as follows:

$$R(0) \land \forall x(\, R(x) \rightarrow R(x+1)\,) \rightarrow \forall x\, R(x)\ ,$$

i.e. in order to prove that R holds for all natural numbers it suffices to prove $R(0)$ and $R(x+1)$ under the *inductive hypothesis* (IH): $R(x)$. Instead of the unary predicate $R(x)$ we can also use $(n+1)$-ary predicates $R(x,\vec{y})$ where the arguments $y_1, \ldots, y_n$ are *parameters*.

1.1.18 Primitive recursion. Mathematical induction is used to prove properties of functions defined by *primitive recursion* . The schema of primitive recursion introduces an $(n+1)$-ary function f from two given functions g (which is n-ary) and h (which is $(n+2)$-ary). The function f satisfies the following recurrences:

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

The schema is justified because its recursion decreases the first-argument.

We also admit *parameterless* primitive recursion defining an unary f from a constant n and a binary h to satisfy:

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

1.1.19 Program for the addition function in monadic representation. The primitive recursive recurrences for the addition function $+$ are well-known:

$$0+y = y \tag{1}$$

$$(x+1)+y = S(x+y) \ . \tag{2}$$

We can, for instance, compute $3+5=8$ in the monadic representation as follows:

$$\begin{aligned} &S^3(0)+S^5(0) \overset{(2)}{=} S(S^2(0)+S^5(0)) \overset{(2)}{=} \\ &\qquad S^2(S^1(0)+S^5(0)) \overset{(2)}{=} S^3(0+S^5(0)) \overset{(1)}{=} S^8(0) \ . \end{aligned}$$

1.1.20 Greatest common divisor. The recurrences for the greatest common divisor function given in Par. 1.1.12 constitute a legal clausal definition:

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

The following clausal definition is slightly more efficient but otherwise expresses the same Euclidian algorithm:

$$\begin{aligned} &gcd(x,y) = x \leftarrow y = 0 \\ &gcd(x,y) = gcd(y, x \bmod y) \leftarrow y > 0 \ . \end{aligned}$$

Both definitions are legal because the second argument goes down in the recursive application, i.e. if $y > 0$ then $x \bmod y < y$.

1.1.21 The whole part of the square root. The function $[\sqrt{x}]$ yields the greatest number whose square does not exceed x:

$$[\sqrt{x}]^2 \leq x < ([\sqrt{x}] + 1)^2 \ . \tag{1}$$

We can derive primitive recursive recurrences for $[\sqrt{x}]$ by assuming (as IH) that (1) holds and then considering the relation between $x+1$ and $([\sqrt{x}]+1)^2$. There are two cases. If

$$[\sqrt{x}]^2 \leq x < x + 1 < ([\sqrt{x}] + 1)^2$$

holds then it clearly suffices to set: $[\sqrt{x+1}] := [\sqrt{x}]$. Otherwise we must have

$$[\sqrt{x}]^2 \leq x < ([\sqrt{x}] + 1)^2 = x + 1 < ([\sqrt{x}] + 2)^2$$

and it suffices to set $[\sqrt{x+1}] := [\sqrt{x}]+1$. This idea is expressed by the following clausal definition:

$$\begin{aligned}
&[\sqrt{0}] = 0\\
&[\sqrt{x+1}] = [\sqrt{x}] \leftarrow x + 1 < ([\sqrt{x}] + 1)^2\\
&[\sqrt{x+1}] = [\sqrt{x}] + 1 \leftarrow x + 1 = ([\sqrt{x}] + 1)^2 \ .
\end{aligned}$$

1.1.22 Assignments. We will investigate more programs (algorithms) for the square root function $[\sqrt{x}]$ because the one given in Par. 1.1.21, although quite pleasing mathematically, is unfeasible in practice for many reasons. One of them is the twofold computation of $[\sqrt{x}]$ for every value x: once in the test and once in the result. This causes an exponential blowup of 2^x recursions in order to compute $[\sqrt{x}]$. The blowup is prevented by the *assignment* in the following clausal definition of the recursive value $[\sqrt{x}]$ to an auxiliary variable z:

$$\begin{aligned}
&[\sqrt{0}] = 0\\
&[\sqrt{x+1}] = z \leftarrow [\sqrt{x}] = z \wedge x + 1 < (z + 1)^2\\
&[\sqrt{x+1}] = z + 1 \leftarrow [\sqrt{x}] = z \wedge x + 1 = (z + 1)^2 \ .
\end{aligned}$$

1.1.23 Bottom up program for $[\sqrt{x}]$. The program for $[\sqrt{x}]$ given in Par. 1.1.22 can be improved in two respects. The first one is the shortening of the computation which goes from x, to $x - 1$, ... , until it reaches 0 and then it does the comparisons and the incrementation by one on the way back. As we already have commented above, the program is mathematically pleasing but every programmer will come up with a 'bottom up' program which searches starting with 0 for the smallest number z such that $x < (z + 1)^2$. This is expressed by *minimalization*:

$$[\sqrt{x}] = \mu_z[x < (z + 1)^2] \ .$$

Only $[\sqrt{x}]$ tests are needed.

The second improvement is the saving of the repeated multiplication in the test $x + 1 < (z + 1)^2$ of the program for $[\sqrt{x}]$ in Par. 1.1.22 by an additional accumulator argument a maintained at the value $a = z^2$ during the computation of the following clausal definition of the ternary function $f(z, a, x)$:

$$f(z,a,x) = z \leftarrow a + 2{\cdot}z + 1 = a_1 \wedge x < a_1$$
$$f(z,a,x) = f(z+1,a_1,x) \leftarrow a + 2{\cdot}z + 1 = a_1 \wedge x \geq a_1 \quad .$$

We claim that

$$[\sqrt{x}] = f(0,0,x) \tag{1}$$

holds. We must show first that the definition of f is legal, i.e. that there is a ternary *measure* $m(z,a,x)$ whose value decreases for the arguments in the recursion. If $x \geq a_1 = a + 2{\cdot}z + 1$ then $x > a$, we have $x \dot{-} a_1 < x \dot{-} a$, and so we can take as measure $m(z,a,x) = x \dot{-} a$ because then $m(z+1,a_1,x) < m(z,a,x)$.

We still have to prove (1). This follows from the property:

$$\forall z \forall a(\, z^2 = a \wedge z + y = x \rightarrow f(z,a,x)^2 \leq x < (f(z,a,x)+1)^2\,).$$

whose proof by induction on y (this is an example of *backward* induction) we leave to the reader. We just note that if $z^2 = a$ holds then we have $(z+1)^2 = z^2 + 2{\cdot}z + 1 = a + 2{\cdot}z + 1$ which should explain the value assigned to a_1. Multiplication by two can be performed on computers as a fast shift.

The definition of the function f is *tail recursive* in the sense that there is nothing to do after the return from recursion. Tail recursive function definitions are efficent to compute from because they directly translate into while loops:

while $x \geq a + 2{\cdot}z + 1$ *do*
 $a := a + 2{\cdot}z + 1;\ z := z + 1;$.

When this loops is started with $z := 0$ and $a := 0$ then we have at its termination $z = f(0,0,x) = [\sqrt{x}]$.

The reader will note that also the definitions of functions *gcd* (see Par. 1.1.12) and *fiba* (see Par. 1.1.13) are tail recursive.

1.1.24 A non-primitive recursive function. A well-known function which is the standard example of a non-primitive but recursive function is the Péter-Ackermann's function (see [15, 19]). It is a binary function with the following clausal definition:

$$A(0,y) = y + 1$$
$$A(x+1,0) = A(x,1)$$
$$A(x+1,y+1) = A(x, A(x+1,y)) \quad .$$

The recursion is legal because we can prove by induction on x that the recurrences determine the value z of $A(x,y)$ for every y. This is trivially true in the base case $x = 0$. In the inductive case we prove by nested induction on y that the value of $A(x+1,y)$ can be detemined under the inductive assumption that for every z the value of $A(x,z)$ can be determined. Indeed, the base case $y = 0$ is trivial. In the inductive case the computation of $A(x+1,y+1)$ needs the value z of $A(x+1,y)$ which can be determined by the inner IH and the value of $A(x,z)$ which can be determined by the outer IH.

The Péter-Ackermann's function cannot be proved total in CL because its formal system, so-called $I\Sigma_1$-arithmetic, does not admit Π_2-induction, i.e. induction on x with the formula $\forall y \exists z\, \phi(x, y, z)$ required to introduce A.

That the function cannot be formally introduced into the CL system is no loss in practice because the function cannot be feasibly computed for the first argument larger than 3. Of course, by taking a stronger $I\Sigma_2$ fragment of arithmetic we would be able to prove the function total but we would not gain much. This is a consequence of the inherent incompleteness of formal arithmetic as proved by Gödel. There are namely recursive functions which are not provably total in any given formal system of arithmetic no matter how strong.

1.2 Dyadic Representation of Natural Numbers

1.2.1 Word domains. Theory of computability fundamentally depends on the coding of sequences of symbols. In order to make the coding simple logicians (see for instance [22]) developed the theory of computable functions over word domains. Word domains are defined with the help of alphabets. An *alphabet* Σ is given by a finite set of *symbols* $\{a_1, a_2, \cdots, a_n\}$. A *word* over an alphabet Σ is a finite sequence of symbols of Σ. The *empty* word is the empty sequence denoted by $\emptyset$. The *length* of a word is the length of its sequence. The *domain of words* over Σ, denoted by Σ^*, is the set of all words over Σ.

We can code a natural number n by a word of length n over a one element alphabet where $a_1 = \mathbf{1}$. We denote by $\mathbf{1}^n$ the sequence $\overbrace{\mathbf{1} \cdots \mathbf{1}}^{n}$. A finite sequence of numbers x_1, x_2, ..., x_n can be then coded as a word over the two element alphabet $a_1 = \mathbf{1}$, $a_2 = \#$ as

$$\#\mathbf{1}^{x_1}\#\mathbf{1}^{x_2} \ldots \#\mathbf{1}^{x_n} \ .$$

1.2.2 The coding of words in $\mathbb{N}$. We can turn the functions over the word domain Σ^* given by the p-element alphabet $\Sigma = \{a_1, a_2, \cdots, a_p\}$ into functions over natural numbers by means of *p-adic representation* of natural numbers. For that purpose we define p functions, called *p-adic successor* functions, S_1, S_2, ..., S_p as follows:

$$S_i(x) = p{\cdot}x + i \ .$$

It is not difficult to see that every natural number has a unique representation as a *p-adic numeral*:

$$S_{i_m}\, S_{i_{m-1}}\, \ldots\, S_{i_2}\, S_{i_1}(0)$$

where $m \geq 0$ and for every $1 \leq j \leq m$ we have $1 \leq i_j \leq p$. This p-adic numeral codes the word $a_{i_1} a_{i_2} \ldots a_{i_m}$. Note that the empty word $\emptyset$ is coded by the p-adic numeral 0. Thus every natural number is a code of exactly one word over Σ. P-adic numerals should be viewed as concrete objects consisting of terms in the form of sequences of function symbols S_i which are terminated by 0.

The reader will note that the monadic representation of natural numbers introduced in Par. 1.1.16 is a special case of p-adic representation with $p = 1$. The monadic successor function $S_1(x) = 1{\cdot}x + 1$ is the successor function S.

1.2.3 Example of arithmetization. Consider the word function $f : \Sigma^* \mapsto \Sigma^*$ where $\Sigma = \{\mathbf{1}, \mathbf{2}\}$ which takes a word over Σ and yields a word with every symbol $\mathbf{1}$ replaced by the symbol $\mathbf{2}$ and vice versa. We, for instance, have

$$g(\mathbf{121}) = \mathbf{212}\ .$$

We can define a function f over natural numbers which does the same on the *dyadic codes* (p-adic codes where $p = 2$), of such words. The process of going from operations over certain domain to the operations over the codes of elements of the domain in $\mathbb{N}$ is called in logic the *arithmetization* of the domain.

Thus the function $f(x)$, which is the arithmetization of g, yields the code of the word coded by x. The function f is defined by recursion decreasing the argument in $<$ to satisfy:

$$f(0) = 0 \tag{1}$$
$$f\,S_1(x) = S_2\,f(x) \tag{2}$$
$$f\,S_2(x) = S_1\,f(x) \tag{3}$$

where we have $S_1(x) = 2{\cdot}x + 1$ and $S_2(x) = 2{\cdot}x + 2$. The word $\mathbf{121}$ is coded by $S_1S_2S_1(0) = 9$ and we obtain a computation sequence

$$f(S_1S_2S_1(0)) \overset{(2)}{=} S_2f(S_2S_1(0) \overset{(3)}{=} S_2S_1f(S_1(0)) \overset{(2)}{=} S_2S_1S_2f(0) \overset{(1)}{=} S_2S_1S_2(0)\ ,$$

yielding the code $S_2S_1S_2(0) = 12$ of the word $\mathbf{212}$.

1.2.4 Dyadic representation of $\mathbb{N}$. The *dyadic representation* of natural numbers, which is a special case of p-adic representation for $p = 2$, was briefly discussed in the previous paragraph. Dyadic representation is one of basic building blocks of CL. The other one is the pair representation discussed in the following section.

Recall that the dyadic representation codes words over the two element alphabet $\{\mathbf{1}, \mathbf{2}\}$ with the help of two *dyadic successor* functions $x\mathbf{1} = 2{\cdot}x + 1$ and $x\mathbf{2} = 2{\cdot}x + 2$. *Dyadic numerals* are the least class of terms containing the constant 0 and with every term τ also the terms $\tau\mathbf{1}$ and $\tau\mathbf{2}$. We use the postfix notation for the dyadic sucessor functions in order to improve the readability.

Consider the first eight words from the sequence of words over the alphabet $\mathbf{1}$, $\mathbf{2}$ which is ordered first on the length and within the same length lexicographically:

$$\emptyset, \mathbf{1}, \mathbf{2}, \mathbf{11}, \mathbf{12}, \mathbf{21}, \mathbf{22}, \mathbf{111}, \ldots$$

The corresponding dyadic numerals are:

$$\begin{aligned}
0 &= 0\\
0\mathbf{1} &= 2{\cdot}0+1 = 1\\
0\mathbf{2} &= 2{\cdot}0+2 = 2\\
0\mathbf{11} &= 2{\cdot}(2{\cdot}0+1)+1 = 3\\
0\mathbf{12} &= 2{\cdot}(2{\cdot}0+1)+2 = 4\\
0\mathbf{21} &= 2{\cdot}(2{\cdot}0+2)+1 = 5\\
0\mathbf{22} &= 2{\cdot}(2{\cdot}0+2)+2 = 6\\
0\mathbf{111} &= 2{\cdot}(2{\cdot}(2{\cdot}0+1)+1)+1 = 7\ .
\end{aligned}$$

From now on we will treat the decimal numerals in clausal definitions as abbreviations for the corresponding dyadic numerals.

1.2.5 Dyadic induction. For an unary predicate $R(x)$ we have the principle of *dyadic induction*:

$$R(0) \land \forall x(\, R(x) \to R(x\mathbf{1}) \land R(x\mathbf{2})\,) \to \forall x\, R(x)\ ,$$

i.e. in order to prove that R holds for all natural numbers it suffices to prove $R(0)$ and both $R(x\mathbf{1})$ and $R(x\mathbf{2})$ under the inductive hypothesis $R(x)$.

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

$$\forall x(\, R(x) \to R(x\mathbf{1}) \land R(x\mathbf{2})\,) \tag{1}$$

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 three cases. If $x = 0$ then we have $R(x)$ from the assumption $R(0)$. If $x = z\mathbf{1}$ for some z then, since $z < x \le n$, we have $R(z)$ from IH: (2) and we get $R(z\mathbf{1})$ from (1). The case when $x = z\mathbf{2}$ is proved similarly. 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 $R(x, y_1, \cdots, x_n)$ with parameters.

1.2.6 Dyadic recursion. Dyadic induction is used to prove properties of functions defined by *dyadic recursion*. The schema of dyadic recursion introduces an $(n+1)$-ary function f from three functions g (which is n-ary) and h_1, h_2 (which are $(n+2)$-ary). The function f satisfies the following recurrences:

$$\begin{aligned}
f(0,\vec{y}) &= g(\vec{y})\\
f(x\mathbf{1},\vec{y}) &= h_1(x,\vec{y},f(x,\vec{y}))\\
f(x\mathbf{2},\vec{y}) &= h_2(x,\vec{y},f(x,\vec{y}))\ .
\end{aligned}$$

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

We also admit *parameterless* dyadic recursion defining an unary f from a constant n and two binary functions h_1, h_2 to satisfy:

$$\begin{aligned} f(0) &= n \\ f(x\mathbf{1}) &= h_1(x, f(x)) \\ f(x\mathbf{2}) &= h_2(x, f(x)) \ . \end{aligned}$$

1.2.7 Dyadic size. The *dyadic size* function $|x|_d$ yields the number of dyadic successors in the dyadic numeral denoting the number x. The function is the arithmetization of the *word-size* function taking a word over $\{\mathbf{1}, \mathbf{2}\}$ and yielding its length. The dyadic size function has a clausal definition by parameterless dyadic recursion:

$$\begin{aligned} &|0|_d = 0 \\ &|x\mathbf{1}|_d = |x|_d + 1 \\ &|x\mathbf{2}|_d = |x|_d + 1 \ . \end{aligned}$$

Clearly, the numbers x such that

$$2^n - 1 = 0\mathbf{1}^n \leq x \leq 0\mathbf{2}^n = 2^{n+1} - 2$$

and no others have the dyadic size n. Hence we have

$$|x|_d = n \leftrightarrow 2^n \leq x + 1 < 2^{n+1} \ . \tag{1}$$

$$n > 0 \rightarrow |2^n|_d = |0\mathbf{1}^{n-1}\mathbf{2}|_d = n \ . \tag{2}$$

1.2.8 Dyadic concatenation. The binary function $x \star y$, called *dyadic concatenation,* yields a number whose dyadic representation is obtained from dyadic representations of x and y by appending the digits of y after the digits of x. The dyadic concatenation function is the arithmetization of the word function concatenating words over the alphabet $\{\mathbf{1}, \mathbf{2}\}$ and it has the following clausal definition:

$$\begin{aligned} &x \star 0 = x \\ &x \star y\mathbf{1} = (x \star y)\mathbf{1} \\ &x \star y\mathbf{2} = (x \star y)\mathbf{2} \quad . \end{aligned}$$

From the recurrences we get:

$$x\mathbf{1} = x \star 1 \tag{1}$$

$$x\mathbf{2} = x \star 2 \tag{2}$$

$$x \star y = x \cdot 2^{|y|_d} + y \tag{3}$$

$$|x \star y|_d = |x|_d + |y|_d \tag{4}$$

$$(x \star y) \star z = x \star (y \star z) \ . \tag{5}$$

The properties are proved as follows. (1): $x\mathbf{1} = (x \star 0)\mathbf{1} = x \star 0\mathbf{1} = x \star 1$. Property (2) is similar. Property (3) is proved by dyadic induction on y. We take any y and consider three cases. If $y = 0$ then we have

$$x \star 0 = x = x{\cdot}2^0 + 0 = x{\cdot}2^{|0|_d} + 0 \ .$$

If $y = z\mathbf{1}$ then we have

$$\begin{aligned} x \star z\mathbf{1} = (x \star z)\mathbf{1} \overset{IH}{=} (x{\cdot}2^{|z|_d} + z)\mathbf{1} = 2{\cdot}(x{\cdot}2^{|z|_d} + z) + 1 = \\ x{\cdot}2^{|z|_d+1} + 2{\cdot}z + 1 = x{\cdot}2^{|z\mathbf{1}|_d} + z\mathbf{1} \ . \end{aligned}$$

The case $y = z\mathbf{2}$ is proved similarly. Property (4) is proved by similar dyadic induction on y. For the property (5) we have

$$\begin{aligned} (x \star y) \star z \overset{(3)}{=} (x \star y){\cdot}2^{|z|_d} + z \overset{(3)}{=} (x{\cdot}2^{|y|_d} + y){\cdot}2^{|z|_d} + z \overset{(3)}{=} \\ x{\cdot}2^{|y|_d+|z|_d} + (y \star z) \overset{(4)}{=} x{\cdot}2^{|y\star z|_d} + (y \star z) \overset{(3)}{=} x \star (y \star z) \ . \end{aligned}$$

1.2.9 Program computing the successor function in dyadic representation. Consider the following clausal definition of the successor function $S(x)$ by parameterless dyadic recursion:

$S(0) = 0\mathbf{1}$
$S(x\mathbf{1}) = x\mathbf{2}$
$S(x\mathbf{2}) = S(x)\mathbf{1}$.

The recurrences hold as properties of the function S. For instance, the last one holds because we have

$$S(x\mathbf{2}) = (2{\cdot}x + 2) + 1 = 2{\cdot}(x + 1) + 1 = S(x)\mathbf{1} \ .$$

We can also visualize the above program as a dyadic example of the algorithm taught on decimal numerals in the elementary school:

$$\begin{array}{r} 0 \\ 0\mathbf{1} \\ \hline 0\mathbf{1} \end{array} \qquad \begin{array}{r} x\mathbf{1} \\ 0\mathbf{1} \\ \hline x\mathbf{2} \end{array} \qquad \begin{array}{r} x\mathbf{2} \\ 0\mathbf{1} \\ \hline S(x)\mathbf{1} \end{array} \ .$$

1.2.10 Program computing the addition function in dyadic representation. The addition function $+$ has the following clausal definition by dyadic recursion on x with *substitution* in the parameter y:

$0 + y = y$
$x\mathbf{1} + 0 = x\mathbf{1}$
$x\mathbf{1} + y\mathbf{1} = (x + y)\mathbf{2}$
$x\mathbf{1} + y\mathbf{2} = S(x + y)\mathbf{1}$
$x\mathbf{2} + 0 = x\mathbf{2}$
$x\mathbf{2} + y\mathbf{1} = S(x + y)\mathbf{1}$
$x\mathbf{2} + y\mathbf{2} = S(x + y)\mathbf{2}$.

We leave it to the reader to convince himself that the recurrences hold as properties of functions concerned. The recurrences in the 'elementary school' form are:

$$
\begin{array}{r}
0\\
y\\ \hline
y
\end{array}
\qquad
\begin{array}{r}
x\mathbf{1}\\
0\\ \hline
x\mathbf{1}
\end{array}
\qquad
\begin{array}{r}
x\mathbf{1}\\
y\mathbf{1}\\ \hline
(x+y)\mathbf{2}
\end{array}
\qquad
\begin{array}{r}
x\mathbf{1}\\
y\mathbf{2}\\ \hline
S(x+y)\mathbf{1}
\end{array}
$$

$$
\begin{array}{r}
x\mathbf{2}\\
0\\ \hline
x\mathbf{2}
\end{array}
\qquad
\begin{array}{r}
x\mathbf{2}\\
y\mathbf{1}\\ \hline
S(x+y)\mathbf{1}
\end{array}
\qquad
\begin{array}{r}
x\mathbf{2}\\
y\mathbf{2}\\ \hline
S(x+y)\mathbf{2}
\end{array}\;.
$$

We can, for instance, compute $3 + 4 = 7$ as follows:

$$0\mathbf{11} + 0\mathbf{12} = S(0\mathbf{1} + 0\mathbf{1})\mathbf{1} = S((0 + 0)\mathbf{2})\mathbf{1} = S(0\mathbf{2})\mathbf{1} = S(0)\mathbf{11} = 0\mathbf{111}\;.$$

1.2.11 Dyadic representation and computational complexity. Computational complexity studies the computation of (word) functions in terms of the number of operations performed (time complexity) and in terms of the maximum length of words used in the computation (space complexity). Both measures are expressed as functions of the length of argument words. This naturaly translates into the domain of natural numbers where the time and space measures are functions of the length of argument numerals.

For two functions f and g we say that $f(x)$ *is* $O(g(x))$, read as $f(x)$ *is at most on the order of* $g(x)$, if there are two numbers a and b such that for all x we have $f(x) \leq a{\cdot}g(x) + b$.

The representation of numbers plays a crucial role in the complexity of a numeric function. The length of the numeral denoting the number x in the dyadic (or in general p-adic when $p \geq 2$) representation is $O(\log(x))$ whereas the length of the monadic numeral for the same number is $O(x)$. This means that the monadic representation is exponentially more wasteful than the dyadic representation. For this reason the complexity theory postulates that both time and space measures of numeric functions are given in terms of the most efficient representation, both binary and p-adic with $p \geq 2$ representations can represent the number x with a numeral of length $O(\log(x))$.

1.2.12 Fast progrem for the addition function. Performing the addition $x + y$ in the monadic representation requires x applications of the identity 1.1.19(2). Thus the length computation is $O(x)$, i.e. $O(2^{\log(x)})$ which is exponential in the dyadic size of x. Performing the same addition in the dyadic representation by recurrences from Par. 1.2.10 needs $O(\log^2 \max(x, y))$ operations. This is because $O(\log \max(x, y))$ recurrences for $+$ are needed and some of them can cause the computation of the successor function S. Each computation of $S(x)$ requires $O(\log(x))$ operations. The square factor can be reduced to a linear one: $O(\log \max(x, y))$ by performing the addition and successor operations in parallel by the familiar method of carry propagation whery we compute a ternary *add-with-carry* function *Addc* satisfying:

$$c \leq 2 \rightarrow Addc(x, y, c) = x + y + c \; .$$

The function has the following clausal definition:

$$\begin{aligned}
&Addc(0, y, c) = Addc(y, c, 0) \leftarrow c > 0 \\
&Addc(0, y, c) = y \leftarrow c = 0 \\
&Addc(2{\cdot}x + i, 0, c) = 2{\cdot}x + d \leftarrow 1 \leq i \wedge i \leq 2 \wedge \\
&\qquad i +_t c = 2{\cdot}c_1 + d \wedge 1 \leq d \wedge d \leq 2 \wedge c_1 = 0 \\
&Addc(2{\cdot}x + i, 0, c) = 2{\cdot}S(x) + d \leftarrow 1 \leq i \wedge i \leq 2 \wedge \\
&\qquad i +_t c = 2{\cdot}c_1 + d \wedge 1 \leq d \wedge d \leq 2 \wedge c_1 > 0 \\
&Addc(2{\cdot}x + i, 2{\cdot}y + j, c) = 2{\cdot}Addc(x, y, c_1) + d \leftarrow 1 \leq i \wedge i \leq 2 \wedge \\
&\qquad 1 \leq j \wedge j \leq 2 \wedge (i +_t j) +_t c = 2{\cdot}c_1 + d \wedge 1 \leq d \wedge d \leq 2 \quad .
\end{aligned}$$

Here the binary function $x +_t y$ is the finite addition function with the arguments restricted as follows:

$$1 \leq x \wedge x \leq 4 \wedge y \leq 2 \rightarrow x +_t y = x + y$$

The finite addition function has the following explicit clausal definition:

$$\begin{aligned}
&1 +_t 0 = 0 \\
&1 +_t 1 = 1 \\
&1 +_t 2 = 3 \\
&\quad\vdots \\
&4 +_t 0 = 4 \\
&4 +_t 1 = 5 \\
&4 +_t 2 = 6 \; .
\end{aligned}$$

The reader will note that, for instance, $5 +_t 1 = 0$ by *default* with the *default clause*

$$5 +_t 1 = 0$$

omitted. Our default convention is that if the given clauses do not specify the value of a function for certain combination of arguments then the value is defined implicitly as equal to zero.

A fast program for addition can be now derived by explicit clausal definition:

$$x + y = Addc(x, y, 0) \quad .$$

1.2.13 A fast program for the square root function. From the point of view of computational complexity the improvement of the accumulator program for the function $[\sqrt{x}]$ in Par. 1.1.23 over the program in Par. 1.1.22 is insignificant because both programs work in the monadic representation. We can obtain a fast $O(|x|_d)$ program for $[\sqrt{x}]$ by recalling a high school algorithm working with the decimal, or rather centennial, notation because it considers two decimal digits at a time. The recursion does not work well in the decimal notation because we have $\sqrt{10{\cdot}x + d} \approx \sqrt{10}{\cdot}\sqrt{x}$ while it works well in the centennial notation because $\sqrt{100{\cdot}x + d} \approx 10{\cdot}\sqrt{x}$.

The same works in the 4-adic representation where for every z we have either $z \leq 2$ or else there are unique numbers x and i such that $z = 4{\cdot}x + i$ and $3 \leq i \leq 6$. The reason why we have chosen this range for i is that we have $|i|_d = 2$ and we can make use of fast pattern matching with the numeric pattern $4{\cdot}x + i$ (just consider the last two dyadic digits of the dyadic representation of z) and fast multiplication $4{\cdot}y + i = y \star i$ (just append the two dyadic digits of i at the end of y).

Thus assume as IH that we have

$$[\sqrt{x}]^2 \leq x < ([\sqrt{x}] + 1)^2 \ .$$

From the last we obtain $x \leq [\sqrt{x}]^2 + 2{\cdot}[\sqrt{x}]$ and so we get for $3 \leq i \leq 6$:

$$(2{\cdot}[\sqrt{x}])^2 < 4{\cdot}x + i \leq 4{\cdot}[\sqrt{x}]^2 + 8{\cdot}[\sqrt{x}] + i < (2{\cdot}[\sqrt{x}] + 3)^2 \ .$$

This means that we have

$$2{\cdot}[\sqrt{x}] \leq [\sqrt{4{\cdot}x + i}] \leq 2{\cdot}[\sqrt{x}] + 2$$

and so the following clausal definition is a fast program for $[\sqrt{x}]$:

$$\begin{array}{l}
[\sqrt{0}] = 0 \\
[\sqrt{1}] = 1 \\
[\sqrt{2}] = 1 \\
[\sqrt{4{\cdot}x + i}] = s \leftarrow 3 \leq i \wedge i \leq 6 \wedge 2{\cdot}[\sqrt{x}] = s \wedge 4{\cdot}x + i < (s+1)^2 \\
[\sqrt{4{\cdot}x + i}] = s + 1 \leftarrow 3 \leq i \wedge i \leq 6 \wedge 2{\cdot}[\sqrt{x}] = s \wedge 4{\cdot}x + i \geq (s+1)^2 \ \wedge \\
\qquad 4{\cdot}x + i < (s+2)^2 \\
[\sqrt{4{\cdot}x + i}] = s + 2 \leftarrow 3 \leq i \wedge i \leq 6 \wedge 2{\cdot}[\sqrt{x}] = s \wedge 4{\cdot}x + i \geq (s+1)^2 \ \wedge \\
\qquad 4{\cdot}x + i \geq (s+2)^2 \quad .
\end{array}$$

A natural question arises. For which arguments z is the last clause taken, i.e. for which z we have to test z against both $(s + 1)^2$ and $(s + 2)^2$? This happens iff $(s + 2)^2 \leq z$ iff $4{\cdot}([\sqrt{x}] + 1)^2 \leq 4{\cdot}x + i$ iff $4{\cdot}((([\sqrt{x}] + 1)^2 - x) \leq i$ iff $z = 4x + i$ for $4 \leq i \leq 6$ and $x + 1 = ([\sqrt{x}] + 1)^2$, i.e. $x + 1$ is a perfect square. These are the numbers of the form $z = 4{\cdot}(y + 1)^2$, $z = 4{\cdot}(y + 1)^2 + 1$, and $z = 4{\cdot}(y + 1)^2 + 2$ when we have $[\sqrt{z}] = 2{\cdot}(y + 1)$.

The reader will note that the assignment $2{\cdot}[\sqrt{x}] = s$ is crucial to the speed of the program because without it we would have two recursive invocations in the test $4{\cdot}x + i < (2{\cdot}x + 1){\cdot}(2{\cdot}x + 1)$. Consequently, as the depth of recursion is $O(|x|_d)$, we would have $O(2^{|x|_d}) = O(x)$ recursive invocations and the definition would not be any faster than the one by primitive recursion.

1.2.14 Binary representation of $\mathbb{N}$. Dyadic representation of natural numbers is similar to the standard *binary representation* of natural numbers which can be expressed with the help of *binary succesor* functions $S_0(x) = 2{\cdot}x$ and $S_1(x) = 2{\cdot}x + 1$. Every number x has a unique representation as a *binary numeral*:

$$S_{i_m}\, S_{i_{m-1}} \cdots S_{i_2}\, S_{i_1}(0)$$

where $m \geq 0$, for every $1 < j \leq m$ we have $0 \leq i_j < 2$, and if $m \geq 1$ then $i_1 = 1$. The reader will note that the word $i_1 i_2 \ldots i_{m-1} i_m$ over the alphabet **0**, **1** represents the number x in the binary notation. If $x \geq 1$ then the *leading digit* i_1 must be 1.

The restriction on the leading digit is imposed in order to quarantee the uniqueness of binary numerals. This eliminates the terms $S_0^n(0)$ as binary numerals whenever $n > 0$. In general, the same kind of restriction must be imposed on the *n-ary representation* of $\mathbb{N}$ for all $n \geq 2$ which uses the functions S_0, S_1, ... , S_{n-1} and differs from the binary representation only in that we have $0 \leq i_j < n$ for every $1 < j \leq m$ and if $m \geq 1$ then $1 \leq i_1 < n$.

We observe that the p-adic representation of $\mathbb{N}$, by using the function S_p instead of S_0, eliminates the restriction on the leading digit of non-zero numerals.

1.2.15 Recursion on notation. One can clearly use n-ary representation in recursive definitions of functions. The following schema of recursion with binary representation is known in the literature as *recursion on notation*:

$$\begin{aligned} f(0,\vec{y}) &= g(\vec{y}) \\ f(S_0(x),\vec{y}) &= h_0(x,\vec{y},f(x,\vec{y})) \leftarrow x > 0 \\ f(S_1(x),\vec{y}) &= h_1(x,\vec{y},f(x,\vec{y}))\ . \end{aligned}$$

The schema was introduced by Cobham in [6] in his recursive theoretic characterization of an important class of computational complexity: polynomial time. We have characterized polynomial time by pair recursion in [26].

1.3 Pair Representation of Natural Numbers

1.3.1 Symbolic domain. 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:

P1: $(x, y) = (v, w) \to x = v \land y = w$,

P2: $v < (v, w) \land w < (v, w)$,

P3: $x = 0 \lor \exists v \exists w\, x = (v, w)$.

The property *P1* 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 *P2* 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. $0 \neq (x, y)$. Thus the number 0 is an *atom* and plays the role of the atom *nil* of *Lisp*. The property *P3* 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*:

$$R(0) \land \forall x \forall y\, (\, R(x) \land R(y) \to R((x, y))\,) \to \forall x\, R(x) \ ,$$

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

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

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 *P3*. Since $v < x$ and $w < x$ by *P2*, 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 $R(x, y_1, \cdots, x_n)$ 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 $<$.

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 $|x|$ 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:

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

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 [7]) 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 *P1*, *P2*, and *P3*. The initial segment of J is tabulated in Fig. 1.1. The subscripts of values for $J(x,y)$ give the pair size $|J(x,y)|$.

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 [26]. 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

$J(x,y)$	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	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Figure 1.1: Pairing function $J(x,y)$

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. 1.3.13 that as a consequence of keeping the numbers with the same pair size together we will have $|x| = \Theta(\log(x))$, i.e. $|x| = O(\log(x))$ and $\log(x) = O(|x|)$. This property assures a natural characterization by pairing of computational complexity classes such as polynomial time or space (see [26]).

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 $|x|$. 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 $|(x,y)|$.

1.3.9 Projection functions. The pairing property *P3* asserts that every non-zero number x has a form (v,w) for some numbers v and w which are uniquely determined by *P2*. *Projection* functions are such that $H(x) = v$ and

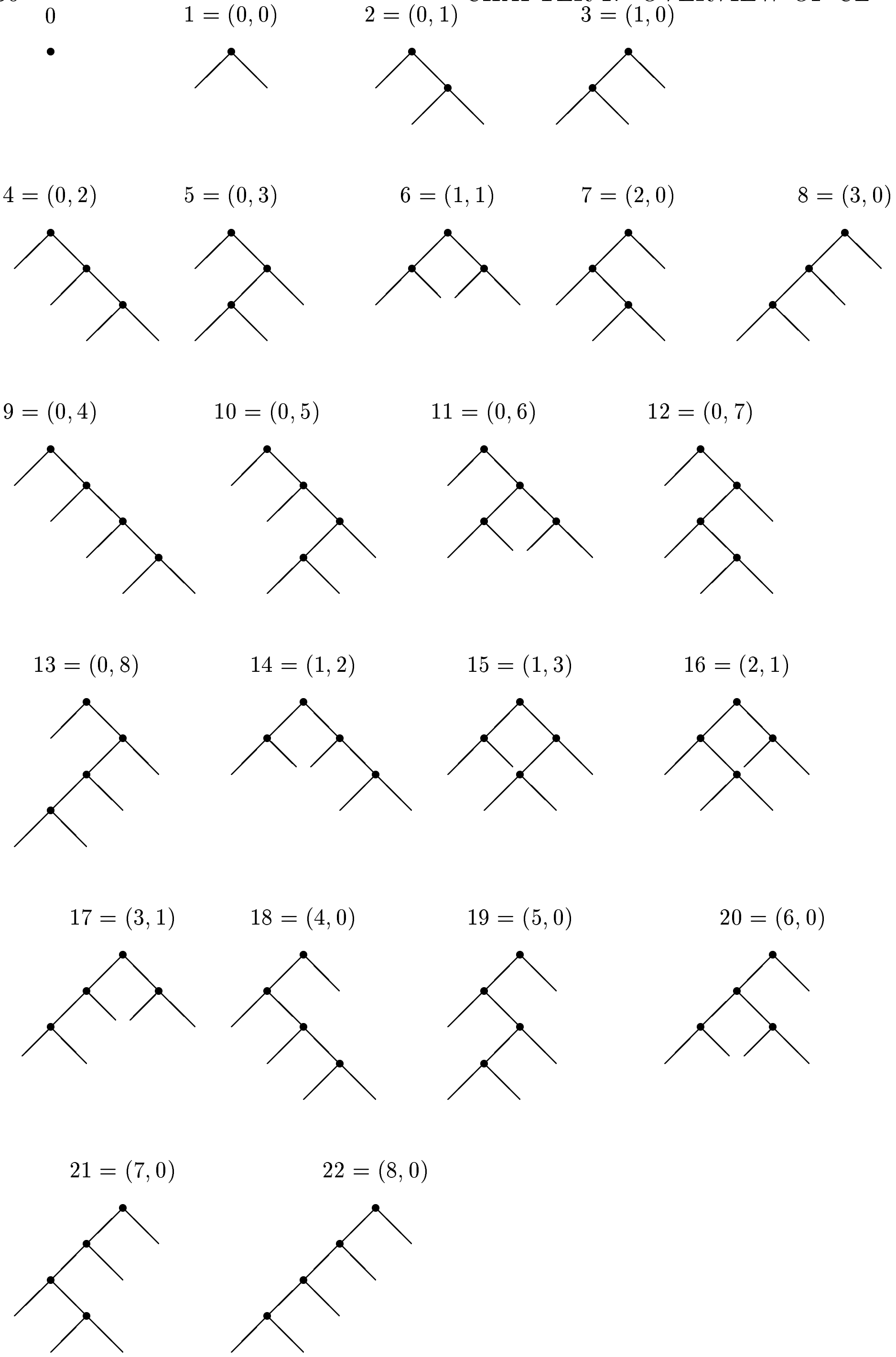


Figure 1.2: Enumeration of binary trees

(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	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Figure 1.3: Pairing function (x, y)

$T(x) = w$.

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

$$H(0) = 0$$
$$H((v, w)) = v \ .$$

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

$$T(0) = 0$$
$$T((v, w)) = w \ .$$

It should be clear that we have

$$x = (H(x), T(x)) \leftrightarrow x > 0 \ .$$

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

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

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, Property (4) holds because we get $|x_1| \leq |x_2|$ from $x_1 < x_2$ by (1). We then consider two cases. If $|x_1| < |x_2|$ then also $|(x_1, y)| < |(x_2, y)|$ and we get the conclusion from (2). If $|x_1| = |x_2|$ then also $|(x_1, y)| = |(x_2, y)|$ 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 [8]) and satisfies the obvious *convolution* recurrences:

$$C(0) = 1 \tag{1}$$

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

because every binary tree with $n+1$ inner nodes has the left son with $i \le n$ 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:

$$\sigma(|x|) \le x < \sigma(|x|+1) \tag{5}$$

$$|\sigma(n)| = n \tag{6}$$

$$\sigma(n) \le C(n) \tag{7}$$

$$2^n \le \sigma(n+1)\ . \tag{8}$$

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 $|x| < n$, can be injectively mapped into the number $(\sigma(n-1-|x|), x)$ 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

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

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

$$2^{|x+1|} \le 2{\cdot}(x+1) \tag{1}$$

$$2^{|x|} \le 2{\cdot}x + 1 \tag{2}$$

$$C(n+1) \le 4^n \tag{3}$$

$$x < 4^{|x|} \tag{4}$$

$$|x| \le x \tag{5}$$

$$(x, y) \le 4{\cdot}(2{\cdot}x+1)^2{\cdot}(2{\cdot}y+1)^2\ . \tag{6}$$

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

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

Property (2) follows from (1) and from $2^{|0|} = 1 = 2{\cdot}0 + 1$.
(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): $x \overset{1.3.11(5)}{<} \sigma(|x|+1) \overset{1.3.11(7)}{\leq} C(|x|+1) \overset{(3)}{\leq} 4^{|x|}$.
Property (5) is proved by induction on x. The base case is trivial. In the inductive case we get $x+1 \leq \sigma(|x|+1)$ from 1.3.11(5) and so we have

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

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

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 $|x| = \Theta(log(x))$. Indeed, for $x > 0$ we have

$$2^{|x|} \overset{1.3.12(1)}{\leq} 2{\cdot}x \leq 2^{\log_2(x)+2}$$

and so $|x| \leq \log_2(x) + 2$, i.e. $|x| = O(\log(x))$. On the other hand, we have

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

and so for $x > 0$ we have $\log_2(x) < 2{\cdot}|x|$, i.e. $\log(x) = O(|x|)$.

1.3.14 Pair versus dyadic size. From the previous paragraph we already know that $|x| = \Theta(|x|_d)$. More precisely, we have:

$$|x| \leq |x|_d + 1 \tag{1}$$
$$|x|_d \leq 2{\cdot}|x| \ . \tag{2}$$

Property (1) holds trivially for $x = 0$ and for $x > 0$ we have

$$|x| \overset{1.2.7(2)}{=} |2^{|x|}|_d \overset{1.3.12(1)}{\leq} |2{\cdot}x|_d \leq |x\mathbf{1}|_d = |x|_d + 1 \ .$$

Property (2) holds trivially for $x = 0$ and for $x > 0$ we have

$$|x|_d \overset{1.3.12(4)}{\leq} |4^{|x|}|_d = |2^{2\cdot|x|}|_d \overset{1.2.7(2)}{=} 2{\cdot}|x| \ .$$

1.3.15 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:

$$\langle f \rangle(x) = \begin{cases} f(x_1, x_2, \ldots, x_n) & \text{if } x = (x_1, x_2, \ldots, x_{n-1}, x_n) \\ 0 & \text{otherwise} \end{cases}$$

where we write $(x_1, x_2, \ldots, x_{n-1}, x_n)$ as an abbreviation for

$$(x_1, (x_2, \ldots, (x_{n-1}, x_n) \ldots)) .$$

From the above we get

$$f(x_1, x_2, \ldots, x_n) = \langle f \rangle ((x_1, x_2, \ldots, x_{n-1}, x_n)) .$$

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

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

and thus we have

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

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

$$\langle R \rangle(x) \leftrightarrow \exists x_1 \exists x_2 \ldots \exists x_n \, (\, x = (x_1, x_2, \ldots, x_{n-1}, x_n) \wedge R(x_1, x_2, \ldots, x_n) \,) .$$

From this we get

$$R(x_1, x_2, \ldots, x_n) \leftrightarrow \langle R \rangle ((x_1, x_2, \ldots, x_{n-1}, x_n)) .$$

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 $\langle f \rangle(x)$.

1.3.16 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 $f(\tau_1, \cdots, \tau_n, \cdots, \tau_{n+m})$ we (obviously) treat the commas as separators of arguments if $m = 0$ and we understand the expression as an abbreviation for $f(\tau_1, \cdots, (\tau_n, \cdots, \tau_{n+m}))$ 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:

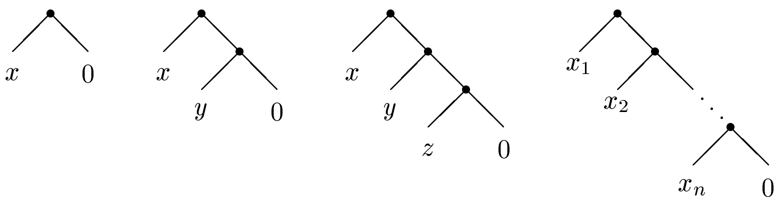
$$\langle f \rangle(x_1, \cdots, x_n) = f(x_1, \cdots, x_n) \ .$$

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 Arithmetization of Data Structures

Arithmetization of Finite Sequences

1.4.1 Coding of finite sequences over $\mathbb{N}$. 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. 1.4). 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.



$x, 0$ $\qquad\qquad$ $x, y, 0$ $\qquad\qquad$ $x, y, z, 0$ $\qquad\qquad$ $x_1, x_2, \ldots, x_n, 0$

Figure 1.4: Lists

1.4.2 Length of lists. The list $x_1, x_2, \ldots, x_n, 0$ codes a finite sequence of natural numbers of length n and we say that the list has the *length* n. The length function $L(x)$ has the following clausal definition:

$$L(0) = 0$$
$$L(x, y) = L(y) + 1 \quad .$$

This kind of recursion is called *list recursion* and it is a special case of pair recursion when we recur only to the tail of the recursive argument.

1.4.3 Concatenation of lists. The binary *list concatenation* function $x \oplus y$ has a definition by list recursion:

$$
\begin{aligned}
&0 \oplus y = y \\
&(v,w) \oplus y = v, w \oplus y \quad .
\end{aligned}
$$

We have

$$
\begin{aligned}
x \oplus 0 &= x && (1)\\
x \oplus (y \oplus z) &= (x \oplus y) \oplus z && (2)\\
|x \oplus y| &= |x| + |y| \ . && (3)\\
L(x \oplus y) &= L(x) + L(y) \ . && (4)
\end{aligned}
$$

The properties are proved by straightforward pair induction on x.

1.4.4 List reversal. The function *Rev* reversing lists has the following clausal definition:

$$
\begin{aligned}
&Rev(0) = 0 \\
&Rev(v,w) = Rev(w) \oplus (v,0) \ .
\end{aligned}
$$

We have

$$
\begin{aligned}
Rev(x \oplus y) &= Rev(y) \oplus Rev(x) && (1)\\
Rev\, Rev(x) &= x && (2)\\
|Rev(x)| &= |x| && (3)\\
L\, Rev(x) &= L(x) \ . && (4)
\end{aligned}
$$

The properties are proved by straightforward pair induction on x.

It takes $O(L(x)^2)$ operations to compute $Rev(x)$. We, namely, need $L(x)$ computations of the list concatenation function $\oplus$ which, in turn, needs $L(x)$ recursions to compute $x \oplus y$. By accumulating the reversed list into an *accumulator* a we can perform the reversal of x in $O(L(x))$ operations with the help of the binary accumulator function $Reva(x,a)$:

$$
\begin{aligned}
&Reva(0,a) = a \\
&Reva((v,w),a) = Reva(w,v,a) \quad .
\end{aligned}
$$

The function is tail recursive and we have

$$
Rev(x) = Reva(x,0)
$$

as a simple consequence of a more general property

$$
\forall a\, Reva(x,a) = Rev(x) \oplus a \qquad (5)
$$

proved by pair induction on x.

1.4.5 List indexing function. The binary *list indexing* function $(x)_i$ yields the i-th element of the list x (counting from 0) or 0 if $i \geq L(x)$. The indexing function has the following clausal definition:

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

Note the default $(0)_x = 0$.

1.4.6 List membership. The binary *list membership* predicate $x \varepsilon y$ has the following basic property:

$$x \varepsilon y \leftrightarrow \exists a \exists b \, y = a \oplus (x,b)$$

and it has a clausal definition by list recursion:

$$x \varepsilon v,w \leftarrow x = v$$
$$x \varepsilon v,w \leftarrow x \neq v \land x \varepsilon w \ .$$

When the clauses of a predicate definition do not specify explictly that the predicate holds for some argument then the predicate is assumed not to hold for the argument by default. Thus we have $x \not\varepsilon 0$ and also $x \not\varepsilon v,w$ whenever $x \neq v$ and $x \not\varepsilon w$.

Arithmetization of finite sets

1.4.7 Arithmetization of finite sets. We can arithmetize finite sets of natural numbers by ordered lists. The predicate $Set(c)$ holds if the list c codes a finite subset of $\mathbb{N}$ and it has the following clausal definition:

$$Set(0)$$
$$Set(x,0)$$
$$Set(x,y,c) \leftarrow x < y \land Set(y,c) \ .$$

Note that we have $\neg Set(7,5,c)$ by default. We identify finite sets with their codes and from now on we will say the *finite set* a instead of the *code of the finite set* a.

1.4.8 Set membership. The binary *set membership* predicate $x \in y$ is true for a finite set a iff $x \varepsilon a$, i.e.:

$$Set(a) \to (x \in a \leftrightarrow x \varepsilon a) \ .$$

The predicate $\in$ has the following clausal definition:

$$x \in y,a \leftarrow x = y$$
$$x \in y,a \leftarrow x > y \land x \in a \ .$$

1.4.9 Binary search. The set membership predicate introduced in the previous paragraph usually executes on ordered lists faster than the list membership predicate. In the worst case both predicates have to go through the entire list and so their complexity is $O(L(x))$. Computer scientists know that the fastest membership algorithm on ordered lists is *binary search* which halves in each step the size of the list to be searched.

The four-place predicate $Bsearch(x,a,i,j)$ searches for x in a at indices $i..j-1$ and it has the following clausal definition:

$$\begin{aligned}&Bsearch(x,a,i,j) \leftarrow i{<}j \land (i{+}j){\div}2 = m \land x < (a)_m \land Bsearch(x,a,i,m)\\&Bsearch(x,a,i,j) \leftarrow i{<}j \land (i{+}j){\div}2 = m \land x = (a)_m\\&Bsearch(x,a,i,j) \leftarrow i{<}j \land (i{+}j){\div}2 = m \land x > (a)_m \land Bsearch(x,a,m{+}1,j)\ .\end{aligned}$$

This is a legal recursion because the measure $j \dot{-} i$ decreases. We clearly have

$$Set(a) \rightarrow (Bsearch(x,a,0,L(a)) \leftrightarrow x \in a)\ .$$

The efficiency of binary search rests with fast indexing of lists which is available if lists are stored in computer memory as arrays. The number of recursive invocations of $Bsearch$ is $O(\log\ L(x))$ but we must count the indexing operation $(a)_i$ as a unit cost operation which is clearly not the case with its definition in Par. 1.4.5.

1.4.10 Unions of finite sets. The binary function $a \cup b$, yields the union of finite sets a and b. The function has the following clausal definition:

$$\begin{aligned}&0 \cup b = b\\&(x,a) \cup 0 = x,a\\&(x,a) \cup (y,b) = x, a \cup (y,b) \leftarrow x < y\\&(x,a) \cup (y,b) = x, a \cup b \leftarrow x = y\\&(x,a) \cup (y,b) = y, (x,a) \cup b \leftarrow x > y\ .\end{aligned}$$

This is a correct definition because the sum of length of the two arguments decreases in recursive applications. Note that we have

$$Set(a) \land Set(b) \rightarrow Set(a \cup b) \land (x \in a \cup b \leftrightarrow x \in a \lor x \in b)\ .$$

We do not care what the value of $x \cup y$ is when x or y is not a finite set.

1.4.11 Union sets. The *set union* function $\bigcup x$ takes a list of finite sets and yields their union. The set union function satisfies the following property:

$$\forall a(a \mathrel{\varepsilon} x \land Set(a)) \rightarrow Set(\bigcup x) \land (z \in \bigcup x \leftrightarrow \exists a(z \in a \land a \mathrel{\varepsilon} x))\ .$$

The function $\bigcup$ has a clausal definition by list recursion:

$$\begin{aligned}&\bigcup 0 = 0\\&\bigcup(v,w) = v \cup \bigcup w\ .\end{aligned}$$

1.4.12 Powerlists. The finite set a is a *subset* of the finite set b, in writing $a \subset b$, if $\forall x(x \in a \to x \in b)$. The function $\mathfrak{P}(a)$ yields a *powerlist* of all subsets of a finite set a:

$$Set(a) \to (b \mathrel{\varepsilon} \mathfrak{P}(a) \leftrightarrow Set(b) \land b \subset a) \ .$$

We obtain a clausal definition of $\mathfrak{P}$ from the observation that the list $\mathfrak{P}(x, a)$ should contain all subsets of a plus all subsets of a unioned with the singleton set $\{x\}$:

$$\begin{aligned}&\mathfrak{P}(0) = 0, 0\\ &\mathfrak{P}(x, a) = p \oplus Singlunions(x, p) \leftarrow \mathfrak{P}(a) = p \quad .\end{aligned}$$

The auxiliary binary function $Singlunions(x, p)$ yields a list which is formed from the members of the list p by prefixing to each member the number x:

$$\begin{aligned}&Singlunions(x, 0) = 0\\ &Singlunions(x, a, p) = (x, a), Singlunions(x, p) \quad .\end{aligned}$$

The reader will note that by the assignment $\mathfrak{P}(a) = p$ in the second clause for $\mathfrak{P}$ we exponentially speed up the computation time.

Arithmetization of Trees

1.4.13 Arithmetization of labeled binary trees. The type Bt of *binary trees labeled by natural numbers* can be defined in most functional programming languages by a *union type*:

$$Bt = E \mid Nd(\mathbb{N}, Bt, Bt) \ ,$$

This asserts that a value of type Bt is either the *leaf* E or a *node* of the form $Nd(x, a, b)$ where a and b are values of type Bt and $x \in \mathbb{N}$ is the *label* of the node. The constant E and the function Nd are called *constructors*.

We arithmetize the values of the type Bt with the help of constructors $E = 0, 0$ and $Nd(x, a, b) = 1, x, a, b$. Note that this guarantees that we have $E \neq Nd(x, a, b)$. The predicate $Bt(a)$ holding of codes of labeled binary trees a has a clausal definition corresponding to the union type definition:

$$\begin{aligned}&Bt(E)\\ &Bt\, Nd(x, a, b) \leftarrow Bt(a) \land Bt(b) \quad .\end{aligned}$$

We identify labeled binary trees with their codes and from now on we will say the *binary tree* a instead of the *code of the binary tree* a.

1.4.14 Structural induction on binary trees. Predicates like Bt whose recursion is positive (i.e. it does not go through negation) are called *inductive* predicates and introduce their own principles of induction. *Bt-induction* principle introduced by the predicate Bt is known in computer science as *structural* induction and for every unary predicate R it asserts:

$$R(E) \land \forall n \forall a \forall b(\, R(a) \land R(b) \to R\, Nd(n,a,b)\,) \land Bt(x) \to R(x)\ .$$

Namely, in order to prove the property $R(x)$ for all binary trees with labels from $\mathbb{N}$, i.e. the property $\forall x(Bt(x) \to R(x))$, it suffices to prove $R(E)$ and also $R\, Nd(n,a,b)$ under IH: $R(a)$ and $R(b)$. The principle of Bt-induction is reduced to mathematical induction similarly as dyadic induction was (see Par. 1.2.5). We assume $R(E)$ as well as

$$\forall n \forall a \forall b(\, R(a) \land R(b) \to R\, Nd(n,a,b)\,) \tag{1}$$

and prove

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

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 $Bt(x)$ and consider two cases corresponding to the clauses for Bt. If $x = E$ then we have $R(x)$ from the assumption $R(E)$. If $x = Nd(n,a,b)$ then we have $Bt(a)$ and $Bt(b)$ from the clausal definition of Bt. Since also $a < x \leq n$ and $b < x \leq n$ hold, we get $R(a)$ and $R(b)$ from IH (2). We now have $R\, Nd(n,a,b)$ from (1). This ends the proof of (2) and we obtain $\forall x(Bt(x) \to R(x))$ by substituting $x+1$ for n.

The reader will note that the principle of Bt-induction is a consequence of the condition of *minimality* imposed on our clausal definitions. The minimality of Bt says that no number codes a binary tree with labels from $\mathbb{N}$ unless it is E or else is of a form $Nd(n,a,b)$ with the numbers a and b coding binary tree with labels from $\mathbb{N}$.

1.4.15 Flattening of binary trees. The function $Flat(a)$ takes a binary tree a and yields a list of all labels in a by reading them off a from left to right. The function has the following clausal definition:

$$\begin{aligned}
&Flat(E) = 0\\
&Flat\; Nd(x,s,t) = Flat(s) \oplus (x, Flat(t))\ .
\end{aligned}$$

We can save the repeated concatenation by keeping the flattened list in an accumulator with a binary function $Flata(t,a)$:

$$\begin{aligned}
&Flata(E,a) = a\\
&Flata(Nd(x,s,t),a) = Flata(s,x,Flata(t,a))\ .
\end{aligned}$$

We have

$$Bt(a) \to Flat(a) = Flata(a,0)$$

which follows from a more general property:

$$Bt(a) \to \forall x\, Flata(a,x) = Flat(a) \oplus x$$

proved by straightforward Bt-induction on a.

1.4.16 Binary search trees. A binary tree a is a *binary search tree* if for every subtree of a labeled by n all labels in its left son are less than n and all labels in the right son are greater than n. This is equivalent to the requirement that the labels of a binary search tress read off from left to right increase. Hence, the predicate $Bst(a)$ holding of binary search trees a has the following clausal definition:

$$Bst(a) \leftarrow Set\,Flat(a) \ .$$

1.4.17 Membership in binary search trees. The binary predicate $x \in_b a$ of *membership in binary search tree*, has the following property:

$$Bst(a) \rightarrow (x \in_b a \leftrightarrow x \in Flat(a)) \ .$$

The predicate has the following clausal definition:

$$\begin{aligned}
&x \in_b Nd(y,a,b) \leftarrow x < y \land x \in_b a \\
&x \in_b Nd(y,a,b) \leftarrow x = y \\
&x \in_b Nd(y,a,b) \leftarrow x > y \land x \in_b b \ .
\end{aligned}$$

1.4.18 Unions of binary search trees. If a and b are binary search trees then the binary function $a \cup_b b$ yields the union of the two trees such that

$$Bst(a) \land Bst(b) \rightarrow Bst(a \cup_b b) \land (x \in_b a \cup_b b \leftrightarrow x \in_b a \lor x \in_b b) \ .$$

The function has the following clausal definition:

$$\begin{aligned}
&E \cup_b b = b \\
&Nd(x,a_1,a_2) \cup_b b = a_1 \cup_b Ins(x, a_2 \cup_b b)
\end{aligned}$$

where the auxiliary binary function $Ins(x,a)$ inserts the element x into the binary search tree a. The function Ins has the following clausal definition:

$$\begin{aligned}
&Ins(x,E) = Nd(x,E,E) \\
&Ins(x,Nd(y,a,b)) = Nd(y,Ins(x,a),b) \leftarrow x < y \\
&Ins(x,Nd(y,a,b)) = Nd(y,a,b) \leftarrow x = y \\
&Ins(x,Nd(y,a,b)) = Nd(y,a,Ins(x,b)) \leftarrow x > y \ .
\end{aligned}$$

1.4.19 Balanced binary trees. The efficiency of the predicate $\in_b$ of membership in binary search trees is comparable to that of binary search in ordered lists when the lists are stored as arrays. There is, however, an additional condition that the binary trees must be *balanced*, i.e the length of their branches must be almost equal. There are many well-known algorithms performing insertion and deletion on binary search trees and which keep the trees balanced. Interested reader can find them in any good text on data structures.

1.4.20 Perfectly balanced binary trees. We call a binary tree *perfectly balanced* if for every subtree the size of its left son is at most one more than the size of its right son. The size $|t|_b$ of a binary tree t has the following clausal definition:

$$
\begin{aligned}
&|E|_b = 0\\
&|Nd(x,a,b)|_b = |a|_b + |b|_b + 1 \quad .
\end{aligned}
$$

The predicate $Pbt(a)$ holding of perfectly balanced binary trees a has the following clausal definition:

$$
\begin{aligned}
&Pbt(E)\\
&Pbt\,Nd(x,a,b) \leftarrow |a|_b = z_1 \land |b|_b = z_2 \land z_1 \geq z_2 \land z_1 \leq z_2 + 1 \land\\
&\quad Pbt(a) \land Pbt(b) \ .
\end{aligned}
$$

1.4.21 Coding of finite sequences by perfectly balanced trees. We normally code finite sequences over $\mathbb{N}$ by lists but we can code them also by perfectly balanced binary trees. For that we need an auxiliary function $Split(a)$ which evenly splits the list a into two lists and yields them as a pair:

$$
\begin{aligned}
&Split(0) = 0, 0\\
&Split(x, 0) = (x, 0), 0\\
&Split(x, y, a) = (x, a_1), y, a_2 \leftarrow Split(a) = a_1, a_2 \quad .
\end{aligned}
$$

The function $L2pbt$ takes a list and yields a perfectly balanced binary tree with members of the lists as labels:

$$
\begin{aligned}
&L2pbt(0) = E\\
&L2pbt(x, a) = Nd(x, L2pbt(a_1), L2pbt(a_2)) \leftarrow Split(a) = a_1, a_2 \ .
\end{aligned}
$$

We now assign the perfectly balanced tree $L2pbt(a)$ as the code of the finite sequence coded by the list a.

1.4.22 Fast indexing of finite sequences. Indexing of finite sequences of length n coded by perfectly balanced binary trees is of time complexity $O(\log(n))$ whereas it is only $O(n)$ if the sequences are coded by lists. The idea is that the label of a perfectly balanced binary tree indexed by i is selected by considering the digits of the dyadic numeral denoting i from right to left. The digit 1 means selection from the left son, the digit 2 means selection from the right son, and $i = 0$ means that the label of the current son is selected. The binary indexing function $a[i]$, has the following clausal definition:

$$
\begin{aligned}
&Nd(x,a,b)[0] = x\\
&Nd(x,a,b)[i\mathbf{1}] = a[i]\\
&Nd(x,a,b)[i\mathbf{2}] = b[i]
\end{aligned}
$$

and satisfies

$$L2pbt(a)[i] = (a)_i \ .$$

1.4.23 Fast modification of finite sequences. Imperative computer programming lanaguages permit modification of finite sequences coded by arrays. This is done by statements $a[i] := x$ which modify the array a at the index i to obtain the new value x. Finite sequences of length n coded by perfectly balanced binary trees can be modified in time $O(\log(n))$ which compares well with time $O(n)$ when the sequences are coded by lists.

The ternary modification function $a[i := x]$ takes a perfectly balanced tree and yields a new tree with the i-th element replaced by x. The function satisfies

$$Pbt(a) \land i \neq j \to a[i := x][j] = a[j] \land \to a[i := x][i] = x$$

and it has the following clausal definition:

$$\begin{aligned}
&E[i := x] = E \\
&Nd(y,a,b)[0 := x] = Nd(x,a,b) \\
&Nd(y,a,b)[i\mathbf{1} := x] = Nd(y,a[i := x],b) \\
&Nd(y,a,b)[i\mathbf{2} := x] = Nd(y,s,t[i := x]) \ .
\end{aligned}$$

Arithmetization of Symbolic Expressions

The symbolic data structures are usually defined in the functional programming languages with the help of union types which can be readily arithmetized. We have seen an example of union type defining binary trees in Par. 1.4.13. We will use in the following paragraph another union type to arithmetize a class of terms.

1.4.24 Numeric terms. Suppose that we wish to operate symbolically on numeric terms, which we call here *expressions*, and which are formed from constants n and variables x_i by the arithmetic operators $+$ and $\cdot$. Functional programming languages use the following union type to specify the domain of expressions:

$$Expr = Cns(\mathbb{N}) \mid Var(\mathbb{N}) \mid Add(Expr, Expr) \mid Mul(Expr, Expr) \ .$$

We arithmetize the expressions with the help of four constructors with explicit clausal definitions:

$$\begin{aligned}
&Cns(x) = 0, x \\
&Var(x) = 1, x \\
&Add(x,y) = 2, x, y \\
&Mul(x,y) = 3, x, y \ .
\end{aligned}$$

We can now code, for instance, the expression $356 \cdot x_3 + x_5$ by the number

$$\begin{aligned}
&Add(Mul(Cns(356),\, Var(3)),\, Var(5)) = \\
&\quad 2,(3,(0,356),1,3),1,5 = 3\,442\,660\,716\,284 \ .
\end{aligned}$$

That the code is larger than three trillion should not be too surprising as its pair size is 25 and the ratio between the length of decimal numbers and pair size is roughly 1 to 2 (the ratio between the length of dyadic and pair size is roughly 1 to 1 cf. Par. 1.3.14).

The predicate $\mathit{Expr}(a)$ true of codes a of expressions corresponds to the above union type and has the following clausal definition:

$$
\begin{aligned}
&\mathit{Expr}\ \mathit{Cns}(c)\\
&\mathit{Expr}\ \mathit{Var}(i)\\
&\mathit{Expr}\ \mathit{Add}(a,b) \leftarrow \mathit{Expr}(a) \land \mathit{Expr}(b)\\
&\mathit{Expr}\ \mathit{Mul}(a,b) \leftarrow \mathit{Expr}(a) \land \mathit{Expr}(b)\ .
\end{aligned}
$$

We get from the definition an induction principle of *Expr-induction* similarly as we have derived the Bt-induction principle in Par. 1.4.14.

1.4.25 Denotation function for expressions. We now define the binary *denotation* (valuation) function $\mathit{Val}(c,v)$ taking the code c of an expression τ and an *assignment* v which is a list assigning the value $(a)_i$ to the variable x_i. The application $\mathit{Val}(c,v)$ yields the value of the expression τ when the variables occurring in the expression take their values from the assignment v. The denotation function has the following clausal definition:

$$
\begin{aligned}
&\mathit{Val}(\mathit{Cns}(c),v) = c\\
&\mathit{Val}(\mathit{Var}(i),v) = (v)_i\\
&\mathit{Val}(\mathit{Add}(a,b),v) = \mathit{Val}(a,v) + \mathit{Val}(b,v)\\
&\mathit{Val}(\mathit{Mul}(a,b),v) = \mathit{Val}(a,v)\cdot \mathit{Val}(b,v)\ .
\end{aligned}
$$

This is a typical example where we wish to use the default clauses in order not to clutter the definition. We do not care what value is yielded by the application $\mathit{Val}(c,n)$ if c does not code an expression. The reader will note that in such case the value is not necessarily 0 because c can be 'almost' an expression. For instance $\mathit{Val}(\mathit{Add}(0,\mathit{Cns}(1)),0) = 1$.

Arithmetization of Numeric Domains

1.4.26 Arithmetization of the domain of integers $\mathbb{Z}$. Integers, which extend the natural numbers with the negative whole numbers, are used in computer programming perhaps more often than the natural numbers. Instead of extending the domain of natural numbers $\mathbb{N}$ to the domain $\mathbb{Z}$ of integers we arithmetize the latter domain in $\mathbb{N}$. For that we need two constructor functions $+(x) = 0,x$ and $-(x) = 1,x$ which respectively code the positive (including 0) and negative (excluding 0) numbers. The predicate $\mathbb{Z}(k)$ which holds if k codes an integer has the following explicit definition:

$$
\begin{aligned}
&\mathbb{Z}(+(n))\\
&\mathbb{Z}(-(n+1))\ .
\end{aligned}
$$

The subtraction function $x-y$ which is closed over the domain $\mathbb{Z}$, i.e. such that

$$\mathbb{Z}(x) \land \mathbb{Z}(y) \to \mathbb{Z}(x-y)$$

holds, has the following explicit clausal definition:

$$
\begin{aligned}
&+(n) - +(m) = +(n \dot{-} m) \leftarrow n \geq m \\
&+(n) - +(m) = -(m \dot{-} n) \leftarrow n < m \\
&+(n) - -(m+1) = +(n+m+1) \\
&-(n+1) - +(m) = -(n+m+1) \\
&-(n+1) - -(m+1) = -(n \dot{-} m) \leftarrow n \geq m \\
&-(n+1) - -(m+1) = +(m \dot{-} n) \leftarrow n < m \ .
\end{aligned}
$$

We can define other arithmetic functions over $\mathbb{Z}$ in a similar style.

1.4.27 Arithmetization of the domain of rational numbers $\mathbb{Q}$. We can artihmetize the domain of rational numbers $\mathbb{Q}$ in $\mathbb{N}$ with the help of two binary constructors: $+(\frac{n}{m}) = 0, n, m$ and $-(\frac{n}{m}) = 1, n, m$ with the obvious intention of defining the predicate $\mathbb{Q}(q)$ holding of codes q of rational numbers as follows:

$$
\begin{aligned}
&\mathbb{Q}(+(\tfrac{n}{m})) \leftarrow m > 0 \land gcd(n,m) = 1 \\
&\mathbb{Q}(-(\tfrac{n}{m})) \leftarrow m > 0 \land n > 0 \land gcd(n,m) = 1 \ .
\end{aligned}
$$

We leave to the reader the clausal definition of the division function x/y which is closed over $\mathbb{Q}$:

$$\mathbb{Q}(x) \land \mathbb{Q}(y) \rightarrow \mathbb{Q}(x/y) \ .$$

Simultaneous recursion

1.4.28 Example: Labeled trees with arbitrarily many sons. The type of trees labeled with natural numbers and with arbitrary number of sons can be in functional programming languages expressed as

$$\mathit{Trees} = E \mid \mathit{Nd}(\mathbb{N}, \mathit{List}(\mathit{Trees})) \ .$$

We can code such trees with the help of the constructor constant $E = 0, 0$ and the binary constructor function $\mathit{Nd}(x,y) = 1, x, y$. The function $\mathit{Sum}(t)$ yielding the sum of all labels in a tree t has the following clausal definition:

$$
\begin{aligned}
&\mathit{Sum}(E) = 0 \\
&\mathit{Sum}\,\mathit{Nd}(n,x) = n + \mathit{Suml}(x) \\
\\
&\mathit{Suml}(0) = 0 \\
&\mathit{Suml}(t,x) = \mathit{Sum}(t) + \mathit{Suml}(x) \quad .
\end{aligned}
$$

The recursive function *Suml* calls on a recursive function $\mathit{Suml}(x)$ summing up the label on all trees in a list of trees x which in turn calls on the function *Suml*. This is an example of *simultaneous* recursion which we will reduce to simple recursion in Par. 1.4.30.

1.4.29 Clausal definitions by simultaneous recursion. Clausal definitions we have presented sofar applied - apart from recursive applications - functions and predicates previously defined.

Clausal definition by *simultaneous recursion* is a definition of $k > 1$ n-ary functions (or predicates) $f_1, \ldots, f_k$ by clausal definitions where each definition can apply previously defined functions plus some or all functions $f_1, \ldots, f_k$. Computer programmers know such definitions as *mutual recursion*.

Simultaneous recursion is reduced to standard recursion by clausally defining an $(n+1)$-ary auxiliary function f whose cluases are obtained by combining the clauses for functions f_i $(1 \leq i \leq k)$ where we replace applications $f_i(\vec{\tau})$ by applications $f(i, \vec{\tau})$. We then explicitly define $f_i(\vec{x}) = f(i, \vec{x})$.

If the functions $f_1, \ldots, f_n$ do not have the same arity then we can use contractions.

1.4.30 Example: reduction of simultaneous recursion. Simultaneous recursion from Par. 1.4.28 is reduced to a definition of the binary function $Sums(x, y)$ by clausal definition decreasing the second argument:

$$\begin{aligned}
&Sums(1, E) = 0\\
&Sums(1, Nd(n, x)) = n + Sums(2, x)\\
&Sums(2, 0) = 0\\
&Sums(2, t, x) = Sums(1, t) + Sums(2, x) \quad .
\end{aligned}$$

The function Sum has now an explicit definition:

$$Sum(t) = Sums(1, t) \ .$$

1.5 Mixed Representation of Natural Numbers

In the development of the theory of computation we are interested only whether a function is *effectively* computable and not whether it is *efficiently* computable. Consequently it does not matter in which representation of natural numbers we perform the computations. In order to be close to the traditional development we will develop the computable functions in Sections 2.3 and 6.1 in monadic representation.

However, for efficient computation one needs the logarithmic size of numerals and it is possible to compute entirely with the dyadic (binary) or pair representations. Conversions between the dyadic and pair representations are possible in linear space and quadratic time (see [26]). This is not much within the theory of complexity but the time spent in conversion between the representations matters in practical computing. The processor for the language CL computes therefore in the *mixed* representation which combines the dyadic and pair representations.

1.5.1 Mixed numerals. Two dyadic numerals are identical as terms iff the natural numbers they denote are equal. Similar property holds for pair numerals. Because the conversions between the dyadic and pair representations are relatively expensive it is of advantage to forego the one-to-one correspondence between numerals and natural numbers and to compute in a mixed representation of natural numbers.

Mixed numerals are the smallest class of terms containing 0 and with every two terms τ_1 and τ_2 also the terms $\tau_1\mathbf{1}$, $\tau_1\mathbf{2}$ and (τ_1, τ_2). Dyadic and pair numerals are thus proper subsets of mixed numerals.

For instance, the number three is denoted by four mixed numerals which are all different as terms:

$$0\mathbf{11} \quad (0,0)\mathbf{1} \quad (0\mathbf{1},0) \quad ((0,0),0)\ .$$

1.5.2 Computation with mixed numerals. If we pay attention to the way we construct our clausal definitions and design them as well-typed we can compute with mixed numerals without doing any conversions between the dyadic and pair numerals.

For instance, if τ_1 and τ_2 are dyadic numerals then the computation of applications $f(\tau_1)$ (see Par. 1.2.3), $\tau_1 \star \tau_2$ (see Par. 1.2.8), $S(\tau_1)$ (see Par. 1.2.9), $\tau_1 + \tau_2$ (see Par. 1.2.10), and $|\tau_1|_d$ (see Par. 1.2.7) proceeds without producing numerals other than dyadic.

If in the application $|\tau|$ of the pair size function the term τ is a pair numeral then the computation of $|\tau|$ proceeds without any conversions when the recurrences for the pair size (Par. 1.3.6), successor (Par. 1.2.9), and addition (Par. 1.2.10) functions are used. The result will be a dyadic numeral.

The function $\mathit{Flat}(\tau)$ (see Par. 1.4.15) converting a binary tree with labels τ into a list of labels works without any conversions if every leaf E of τ is coded as the pair $0, 0$ (and not as a dyadic numeral $0\mathbf{1}$) and every node $Nd(x, a, b)$ of τ as a pair $1, x, a, b$. It does not matter how the tags 1 and the labels x are coded because the tags are tested only for 0 and the values of labels are not needed.

On the other hand, the natural coding of binary search trees with labels from $\mathbb{N}$ is with the labels coded as dyadic numerals. Thus, for instance, if τ_1 is a dyadic numeral and τ_2 a well-typed binary search tree, the application $\tau_1 \in_b \tau_2$ of the predicate of membership in binary search trees (see Par. 1.4.17) can be computed without any conversions. Of course, we assume that the arithmetic comparison relations expect their arguments as dyadic numerals and perform conversions from pairs only when needed.

Another example of mixed representation computation without conversions is the denotation function Val for expressions (see Par. 1.4.25). Assume that the term τ denoting the code of an expression is well-typed, i.e. if, for instance, $\tau = \mathit{Add}(\tau_1, \tau_2)$ then we have $\tau \equiv 2, \tau_1, \tau_2$, or if $\tau = \mathit{Cns}(\tau_1)$ then we have $\tau \equiv 0, \tau_1$ with τ_1 a dyadic numeral. The computation of $\mathit{Val}(\tau, \rho_1, \ldots, \rho_n, 0)$ with the terms $\rho_1, \ldots, \rho_n$ dyadic numerals can then proceed without any conversions.

Computations, by being syntactic processes, proceed on the shape (intension) of terms rather than on their denotation (extension). For instance, although we extensionally have

$$\mathit{Val}((1, 0\mathbf{2}), 0, 0, 0\mathbf{11}, 0) = \mathit{Val}((1, 0, 0, 0), 0, 0, 0\mathbf{11}, 0) = 3\ ,$$

the computation of the first application proceeds without conversions:

$$\mathit{Val}((1,0\mathbf{2}),0,0,0\mathbf{11},0) = (0,0,0\mathbf{11},0)_{0\mathbf{2}} = (0,0\mathbf{11},0)_{0\mathbf{1}} = (0\mathbf{11},0)_0 = 0\mathbf{11}$$

whereas the computation of the second application requires a conversion from a pair to dyadic numeral in the step marked by $*$:

$$\mathit{Val}((1,0,0,0),0,0,0\mathbf{11},0) = (0,0,0\mathbf{11},0)_{0,0,0} \stackrel{*}{=} (0,0,0\mathbf{11},0)_{0\mathbf{2}} = \cdots = 0\mathbf{11}\ .$$

The intensional character of computations cannot be captured extensionally (except by an additional level of coding) but it can be expressed by the intensional typing of clausal definitions.

1.5.3 Fast constructor operations on mixed numerals. The operations (x,y), $x\mathbf{1}$, and $x\mathbf{2}$ are called *constructors* and they can be computed quickly in the mixed representation.

Indeed, the pairing term τ_1,τ_2 where the terms τ_1 and τ_2 are closed can be computed quickly because once we have computed the term τ_1 to a coextensive mixed numeral ρ_1 and the term τ_2 to a coextensive mixed numeral ρ_2 we can form the result as a mixed numeral: (ρ_1,ρ_2).

Similarly, the dyadic successor terms $\tau\mathbf{1}$ and $\tau\mathbf{2}$ where the term τ is closed can be computed quickly because after we have computed the term τ to a coextensive mixed numeral ρ we can form the results as mixed numerals $\rho\mathbf{1}$ and $\rho\mathbf{2}$ respectively.

The fast computation of constructors $x\mathbf{1}$ and $x\mathbf{2}$ can be extended to the fast computation of arithmetic terms of the form $p{\cdot}\tau + k$ where the term τ is closed, the constant p is a power of two, and the constant k is such that $p \le k+1 < 2{\cdot}p$. We, namely first compute the term τ to a coextensive mixed numeral ρ and we continue with the computation of the term $\rho \star \rho_1$ where ρ_1 is the dyadic numeral denoting k. The concatenation is fast because it requires $|k|_d$ recurrences (see Par. 1.2.8) and results in a mixed numeral without any conversions. The justification of this computation is that we have $p = 2^n$ for some n and from $2^n \le k+1 < 2^{n+1}$ we get

$$p{\cdot}x + k = 2^n{\cdot}x + k \stackrel{1.2.7(1)}{=} 2^{|k|_d}{\cdot}x + k \stackrel{1.2.8(3)}{=} x \star k\ .$$

1.5.4 Fast matching operations on mixed numerals. The decision which clause of a clausal definition of f to use in order to compute an application $f(\tau)$ for a mixed numeral τ involves a *matching*, or *destruction* operation.

Suppose that when computing $f(\tau)$ we have to decide between the following clauses for f:

$$\begin{aligned}&f(0) = \cdots\\ &f(\rho_1,\rho_2) = \cdots\ .\end{aligned}$$

If $\tau \equiv 0$ then we select the first clause. If $\tau \equiv \tau_1,\tau_2$ for some mixed numerals τ_1 and τ_2 then this part of matching is fast because we select the second cause and then τ_i must match ρ_i for $i = 1$ or $i = 2$. If, however, we have $\tau \equiv \tau_1\mathbf{1}$ or $\tau \equiv \tau_1\mathbf{2}$ for a mixed numeral τ_1 then we know that the first clause cannot apply and the second clause must be selected but we have to find by a conversion the mixed numerals ρ_1 and ρ_2 such that $\tau = (\rho_1, \rho_2)$ holds.

Now, suppose that when computing $f(\tau)$ we have to decide between the following clauses for f:

$$\begin{aligned}&f(0) = \cdots\\&f(\rho_1\mathbf{1}) = \cdots\\&f(\rho_2\mathbf{2}) = \cdots \quad .\end{aligned}$$

If $\tau \equiv 0$ then we select the first clause. If we have $\tau \equiv \tau_1\mathbf{1}$ for a mixed numeral τ_1 then this part of matching is fast as we select the second clause and τ_1 must then match ρ_1. If we have $\tau \equiv \tau_1\mathbf{2}$ for a mixed numeral τ_1 then we select the third clause and τ_1 must then match ρ_2. If the topmost operator of τ is a pair then we convert τ to a coextensive dyadic numeral of a form $\tau_1\mathbf{1}$ or $\tau_1\mathbf{2}$ and select the second or the third clause accordingly.

1.5.5 Fast matching of p-adic patterns. Terms of the form $p{\cdot}x + i$ are called *p-adic patterns* . When $p > 1$ is a power of two then a fast matching of arguments against such patterns is possible.

Suppose that when computing $f(\tau)$ for a mixed numeral τ we have to decide between the following clauses for f:

$$\begin{aligned}&f(n) = \cdots \leftarrow n < p - 1\\&f(p{\cdot}x + i) = \cdots \leftarrow p - 1 \le i \land i \le p - 2 \ .\end{aligned}$$

where $p = 2^k$ for some $k > 0$. It should be clear that for every number n we have either $n < p - 1$ or there are unique numbers x and i such that $n = p{\cdot}x + i$ and $p - 1 \le i \le 2{\cdot}p - 2$. We, moreover, have $|i|_d = k$ by 1.2.7(1).

The matching of the mixed numeral τ against the above clauses will be fast when τ is a dyadic numeral such that $|\tau|_d < k$ and then we select the first clause or else there are at least k outermost dyadic successor operators in τ. In such a case we select the second clause and we can quickly find a mixed numeral τ_1 which is assigned to the variable x and a dyadic numeral τ_2 which is assigned to the variable i such that $\tau \equiv \tau_1 \star \tau_2$ and $|\tau_2|_d = k$.

1.5.6 Efficient implementation of mixed numerals on computers. Computation is a syntactic process which proceeds by the manipulation of concrete objects. In the above discussion the concrete objects were the dyadic, pair, or mixed numerals. In any practical implementation of such computations on an electronic computer the mixed numerals will have to be mapped into the memory structures of the computer. It is then of advantage to use the fast arithmetic of, say 32-bit, numbers built into the computer. We will now describe the mapping of mixed numerals used in our implementation of CL on PC's.

A mixed numeral is represented by a a 32-bit number whose lowest two bits serve as four tags distinguishing the four kinds of mixed numerals. We assign the tag 0 to represent the numerals which are pairs of the form (τ_1, τ_2). A 32-bit number n with the tag 0 (note that $n \bmod 4 = 0$) is interpreted as a pointer to a *Lisp-cell*, i.e. to two adjacent 32-bit words. The first word represents the first projection τ_1 and the second word the second projection τ_2. A 32-bit number n with the tag 1 represents the mixed numeral $\tau\mathbf{1}$ by interpreting the number $n-1$ as a pointer to a word representing the mixed numeral τ. Similarly, a 32-bit number n with the tag 2 represents the mixed numeral $\tau\mathbf{2}$ by interpreting the number $n-2$ as a pointer to a word representing the mixed numeral τ. Finally, a 32-bit number n with the tag 3 represents the fourth kind of mixed numerals. Normally this would be the single numeral 0. It is, however, advantageous to utilize the remaining 30-bits of such words to store small natural numbers directly. Our CL-implementation interprets accordingly the number $n \div 4$ as a binary numeral.

The memory representation of CL can be described purely in the language of logic by defining the class of *extended mixed numerals* as the smallest class of terms which contains decimal numerals denoting the numbers n such that

$$n \leq 1{,}073{,}741{,}823 = 2^{30} - 1 = 0\mathbf{1}^{30}$$

holds and such that with each two extended mixed numerals τ_1 and τ_2 also the terms $\tau_1\mathbf{1}$, $\tau_2\mathbf{2}$, and (τ_1, τ_2) are extended mixed numerals. Figure 1.5 shows computer memory representation of the mixed numeral $(5, 1073741823\mathbf{1})\mathbf{21}$.

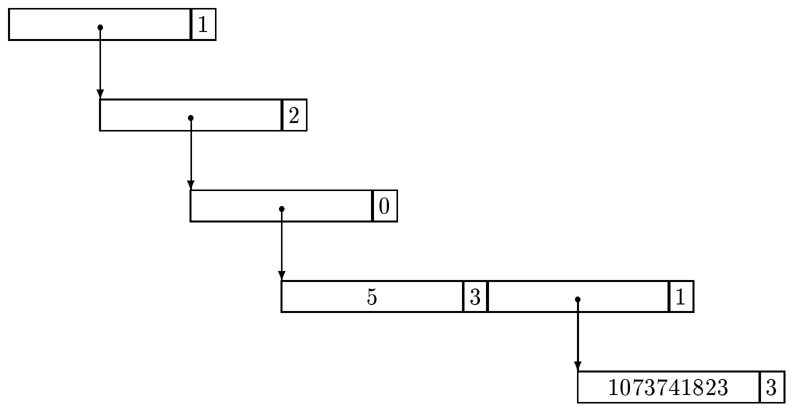


Figure 1.5: Memory representation of the mixed numeral $(5, 1073741823\mathbf{1})\mathbf{21}$

Chapter 2

Recursive Functions

We will discuss in Sect. 2.1 recursive definitions in the style of Herbrand-Gödel-Kleene equations. The equations are presented in a slightly different way as usual (see for instance [12, 10]) because our treatment syntactically guarantees that the equations define partial functions. Partial functions defined by recursive equations are called *partial recursive* functions and we show that they are effectively computable in Sect. 2.3. Section 2.2 deals with the metamathematical issues of the relation between the explicit and recursive definitions by equations and by clauses. Thus the section defines the syntax and semantics of the programming language CL. We prefer CL, because of its readability, to the language of recursive equations.

2.1 Explicit and Recursive Definitions

Alghough our primary interest lies in the effectively computable functions over the domain of natural numbers it is of advantage to study also partial functions whose computation does not stop for all arguments.

2.1.1 Partial functions. A *partial* n-ary function f is a function 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-ry partial function, i.e. the function with the domain $\emptyset \subseteq \mathbb{N}^n$.

The *graph* of f is the $(n+1)$-ary relation denoted by $f(\vec{x}) \asymp y$ which holds if $\langle \vec{x} \rangle \in S$, i.e. if f is defined for $\vec{x}$, and we have $f(\vec{x}) = y$.

We clearly have $\neg\emptyset^n(\vec{x}) \asymp y$ and for (total) functions f we have

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

We say that the (total) function g is the *completion* of a partial function f

if the following holds:

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

We say that a unary partial function $\langle f \rangle$ is the *contraction* of an n-ary partial function if the following holds:

$$\langle f \rangle(x) \asymp \begin{cases} y & x = \vec{x} \land f(\vec{x}) \asymp y \\ 0 & T^{n-2}(x) = 0 \ . \end{cases}$$

2.1.2 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

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

for all $\vec{x}$ and y. Note that we have $\emptyset^n \subseteq f$ 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

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

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

2.1.3 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 constants and variables by applications of function symbols. A variable is *free* in a term if it occurs in it. A *closed* term does not have free variables.

We use $\tau, \tau_1, \ldots, \rho, \rho_1, \ldots$ as *syntactic* variables ranging over terms. We write $\tau_1 \equiv \tau_2$ if the two terms are identical sequences of symbols.

2.1.4 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

$$\tau[g;\vec{x}] \equiv \tau[\dot{\lambda}\vec{y}.g(\vec{y});\vec{x}] \ .$$

2.1.5 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 $\tau[\vec{x}]$ with at most the n indicated variables free the notation $\tau[\vec{x}] \asymp y$, called the *graph of* τ, as an $(n+1)$-ary relation defined by induction on the structure of τ. If τ is a constant c then we define

$$c \asymp y \leftrightarrow c = y \ .$$

If τ is a variable x_i where $1 \le i \le n$ then we define

$$x_i \asymp y \leftrightarrow x_i = y \ .$$

If $\tau \equiv g(\tau_1, \ldots, \tau_m)$ for an m-ary partial function g which is not the case discrimination function D then we define:

$$\begin{aligned} &g(\tau_1[\vec{x}], \ldots, \tau_m[\vec{x}]) \asymp y \leftrightarrow \\ &\qquad \exists z_1 \cdots \exists z_m (\, \tau_1[\vec{x}] \asymp z_1 \land \cdots \land \tau_m[\vec{x}] \asymp z_m \land g(z_1, \ldots, z_m) \asymp y \,) \end{aligned}$$

where $z_1, \ldots, z_m$ are *new* variables, i.e. variables which do not occur in τ_i.

If $\tau \equiv D(\tau_1, \tau_2, \tau_3)$ then for a new variable z we define:

$$\begin{aligned} D(\tau_1[\vec{x}], \tau_2[\vec{x}], \tau_3[\vec{x}]) \asymp y \leftrightarrow {} & \exists z(\, \tau_1[\vec{x}] \asymp z \land z > 0 \land \tau_2[\vec{x}] \asymp z \,) \lor \\ & \exists z(\, \tau_1[\vec{x}] \asymp z \land z = 0 \land \tau_3[\vec{x}] \asymp z \,) \ . \end{aligned}$$

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

$$\tau[\vec{x}] \asymp y \land \tau[\vec{x}] \asymp z \to y = z \ .$$

A closed interpreted term τ is assigned as its *partial denotation* the number n if $\tau \asymp n$ 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.

The assignment of partial denotations to terms is usually described by saying that all functions g except D are *strict*, meaning that $g(\vec{\rho})$ denotes iff all terms $\vec{\rho}$ denote and $\langle\vec{\rho}\rangle$ is in the domain of g. The case analysis function D is non-strict because $\tau \equiv D(\tau_1, \tau_2, \tau_3)$ denotes only if τ_1 denotes and then if $\tau_1 > 0$ the term τ denotes the same number as the term τ_2 and if $\tau_1 = 0$ 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.

2.1.6 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.

2.1.7 Non-strict identity relation. In the presence of partial functions we need the *non-strict* relation of identity $\tau_1 \simeq \tau_2$ satisfying:

$$\tau_1[\vec{x}] \simeq \tau_2[\vec{x}] \leftrightarrow \forall y (\, \tau_1[\vec{x}] \asymp y \leftrightarrow \tau_2[\vec{x}] \asymp y \,)$$

where y does not occur among the variables $\vec{x}$. 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 $\tau_1 \simeq \tau_2$ iff $\tau_1 = \tau_2$.

2.1.8 Graphs of terms are monotone. Graphs of terms $\tau[f; \vec{x}] \asymp y$ are *monotone in f* in the following sense:

$$f_1 \subseteq f_2 \land \tau[f_1; \vec{x}] \asymp y \to \tau[f_2; \vec{x}] \asymp y \ . \tag{1}$$

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

$$\forall y \, (\tau[f_1; \vec{x}] \asymp y \to \tau[f_2; \vec{x}] \asymp y)$$

by induction on the construction of τ. So taky any y, assume $\tau[f_1; \vec{x}] \asymp y$, and continue by the case analysis of τ. If τ is a constant or a variable then $\tau[f_2; \vec{x}] \asymp y$

holds trivially. If $\tau \equiv f(\vec{\rho})$ then from the assumption that the term $(f(\vec{\rho}))[f_1;\vec{x}]$ denotes y we have for all $1 \leq i \leq n$ numbers y_i such that $\rho_i[f_1;\vec{x}] \asymp y_i$ and $f_1(y_1,\dots,y_n) \asymp y$ hold. Hence also $f_2(y_1,\dots,y_m) \asymp y$ holds and from n IH's we get $\rho_i[f_2;\vec{x}] \asymp y_i$. But this means that $(f(\vec{\rho}))[f_2;\vec{x}] \asymp y$ holds. If $\tau \equiv g(\vec{\rho})$ where $g \neq D$ then the proof is similar. If $\tau \equiv D(\rho_1,\rho_2,\rho_3)$ then from the assumption $D(\rho_1[f_1;\vec{x}],\rho_2[f_1;\vec{x}],\rho_3[f_1;\vec{x}]) \asymp y$ we get $\rho_1[f_1;\vec{x}] \asymp y_1$ for some y_1 and we have $\rho_1[f_2;\vec{x}] \asymp y_1$ from IH. We now consider two cases. If $y_1 > 0$ then we have $\rho_2[f_1;\vec{x}] \asymp y$ and we obtain $\rho_2[f_2;\vec{x}] \asymp y$ from another IH. But then $D(\rho_1[f_2;\vec{x}],\rho_2[f_2;\vec{x}],\rho_3[f_2;\vec{x}]) \asymp y$ holds. The case $y_1 = 0$ is proved similarly.

2.1.9 Graphs of terms are continuous. Graphs of terms $\tau[f;\vec{x}] \asymp y$ are *continuous in* f in the sense that for any chain of n-ary functions f_0, f_1, ... we have

$$\tau[\bigcup_i f_i;\vec{x}] \asymp y \to \exists i\, \tau[f_i;\vec{x}] \asymp y\ . \tag{1}$$

Indeed, we take any $\vec{x}$ and prove $\forall y(1)$ by induction on the construction of τ. So take any y, assume $\tau[\bigcup_i f_i;\vec{x}] \asymp y$, and continue by case analysis of τ. If τ is a constant or a variable then $\tau[f_0;\vec{x}] \asymp y$ holds trivially. If $\tau \equiv f(\vec{\rho})$ then we have for all $1 \leq j \leq n$ numbers y_j such that $\rho_j[\bigcup_i f_i;\vec{x}] \asymp y_j$ and $(\bigcup_i f_i)(y_1,\dots,y_n) \asymp y$ hold. From the last we have $f_{i_0}(y_1,\dots,y_n) \asymp y$ for some i_0. From n IH's there are numbers i_j such that $\rho_j[f_{i_j};\vec{x}] \asymp y_i$ holds. For $i = \max(i_0,i_1,\dots,i_n)$ we have $\rho_j[f_i;\vec{x}] \asymp y_i$ as well as $f_i(y_1,\dots,y_n) \asymp y$ by 2.1.8(1) and so we also have $\tau[f_i;\vec{x}] \asymp y$. If $\tau \equiv g(\vec{\rho})$ where $g \neq D$ then the proof is similar. If $\tau \equiv D(\rho_1,\rho_2,\rho_3)$ then we have $\rho_1[\bigcup_i f_i;\vec{x}] \asymp y_1$ for some y_1 and we obtain $\rho_1[f_j;\vec{x}] \asymp y_1$ for some j from IH. We now consider two cases. If $y_1 > 0$ then we have $\rho_2[\bigcup_i f_i;\vec{x}] \asymp y$ and we obtain $\rho_2[f_k;\vec{x}] \asymp y$ for some k from another IH. For $i = \max(j,k)$ we have $\rho_1[f_i;\vec{x}] \asymp y_1$ and $\rho_2[f_i;\vec{x}] \asymp y$ by 2.1.8(1) and so $D(\rho_1[f_i;\vec{x}],\rho_2[f_i;\vec{x}],\rho_3[f_i;\vec{x}]) \asymp y$ holds. The case $y_1 = 0$ is proved similarly.

2.1.10 Functional equations. Let $\tau[f;\vec{x}]$ be a term which is constructed from constants and variables among the n indicated variables $\vec{x}$ by applications of function symbols g_1, ..., g_k ($k \geq 0$), and possibly 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, ..., g_k to the corresponding function symbols then we say that a partial function f solves (1) if the property holds for all $\vec{x}$ and y.

We will see in the following paragraph that if the term τ does not apply the function symbol f then the equation is *explicit* and can be uniquely solved. Otherwise the equation is *recursive* and, as the theorem 2.1.12 asserts, there is a solution which is minimal in certain sense.

2.1.11 Explicit definitions. Let the term τ of the equation 2.1.10(1) be such that it does not apply the function symbol f and that the function symbols applied in τ are $g_1, \dots, g_k$.

If the function symbols applied in the term τ are interpreted by the corresponding partial functions $g_1, \dots, g_k$ then clearly there is a unique partial function f defined for every $\vec{x}$ to be equal to the denotation of $\tau[\vec{x}]$ if the term $\tau[\vec{x}]$ converges and undefined otherwise. This f uniquely solves the equation 2.1.10(1) and the equation is said to be *explicit definition* of the partial function f from the partial functions $g_1, \dots, g_k$. If the term τ does not contain constants then the definition is *explicit definition without constants.* Note that if $g_1, \dots, g_k$ are (total) functions then so is f.

2.1.12 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 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 $\emptyset^n \subseteq f_1$ 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 2.1.8(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 $(\rightarrow)$ we assume $(\bigcup_i f_i)(\vec{x}) \asymp y$ and obtain $f_i(\vec{x}) \asymp y$ from 2.1.2(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 2.1.2(2) and 2.1.8(1). In the direction $(\leftarrow)$ we assume $\tau[\bigcup_i f_i;\vec{x}] \asymp y$ and from 2.1.9(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 2.1.2(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 2.1.8(1). Thus $f(\vec{x}) \asymp y$ holds. We then obtain $\bigcup_i f_i \subseteq f$ from 2.1.2(3). ■

2.1.13 Recursive definitions. Let the term τ of the functional equation $f(\vec{x}) \simeq \tau[f;\vec{x}]$ 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 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. 2.1.12.

2.1.14 Governing conditions. We will prove in Thm. 2.1.17 a sufficient condition for recursive definitions to define total functions. For that we need the concept of governing conditions.

To every term τ and an occurrence in τ of a subterm ρ we define by recursion on $||\tau|| - ||\rho||$ the *governing conditions* ϕ *of* ρ *in* τ. Here we denote by $||\tau||$ the *size*, i.e. the number of operations, of the term τ.

If $\rho \equiv \tau$ then $\phi \equiv \top$. If $\rho \equiv g(\rho_1, \ldots, \rho_m)$ and $g \not\equiv D$ then ϕ governing any of $\rho_1, \ldots, \rho_m$ in τ is the same as ϕ governing ρ in τ. If $\rho \equiv D(\rho_1, \rho_2, \rho_3)$ then ϕ governing ρ_1 in τ is the same as ϕ_1 governing ρ in τ. The condition governing ρ_2 in τ is $\phi_1 \land \exists y(\rho_1 \asymp y \land y > 0)$ for a new y. The condition governing ρ_3 in τ is defined as $\phi_1 \land \rho_1 \asymp 0$.

2.1.15 Theorem *If f and g are n-ary partial functions, $\vec{x}$ any numbers, and $\tau[f;\vec{x}]$ a term such that for all applications $f(\vec{\rho})$ governed in τ by conditions $\phi[f;\vec{x}]$ we have:*

$$\phi[f;\vec{x}] \to f(\vec{\rho}[f;\vec{x}]) \simeq g(\vec{\rho}[f;\vec{x}]) \tag{1}$$

then $\tau[f;\vec{x}] \simeq \tau[g;\vec{x}]$ holds.

Proof. Take any numbers $\vec{x}$, assume (1) for all applications of f in τ, and prove an auxiliary property by induction on the structure of ρ:

> for any occurrence of a subterm $\rho[f;\vec{x}]$ of $\tau[f;\vec{x}]$ governed in τ by $\phi[f;\vec{x}]$ we have
>
> $$\phi[f;\vec{x}] \to \rho[f;\vec{x}] \simeq \rho[g;\vec{x}] \; . \tag{2}$$

Thus assume that $\phi[f;\vec{x}]$ governing an occurrence of ρ in τ holds and continue by the case analysis of ρ. If ρ is a constant symbol or a variable then the conclusion of (2) holds trivially. If $\rho \equiv h(\rho_1, \cdots, \rho_m)$ where h is different from f and D then the arguments ρ_i are governed in τ by ϕ and we have

$$\begin{aligned} h(\rho_1, \cdots, \rho_m)[f;\vec{x}] \simeq h(\rho_1[f;\vec{x}], \cdots, \rho_m[f;\vec{x}]) &\overset{\text{IH's}}{\simeq} \\ h(\rho_1[g;\vec{x}], \cdots, \rho_m[g;\vec{x}]) &\simeq h(\rho_1, \cdots, \rho_m)[g;\vec{x}] \; . \end{aligned}$$

If $\rho \equiv f(\rho_1, \cdots, \rho_n)$ then the arguments ρ_i are governed in τ by ϕ and we have

$$\begin{aligned} f(\rho_1, \cdots, \rho_n)[f;\vec{x}] \simeq f(\rho_1[f;\vec{x}], \cdots, \rho_n[f;\vec{x}]) \overset{(1)}{\simeq} g(\rho_1[f;\vec{x}], \cdots, \rho_n[f;\vec{x}]) &\overset{\text{IH's}}{\simeq} \\ g(\rho_1[g;\vec{x}], \cdots, \rho_n[g;\vec{x}]) \simeq f(\rho_1, \cdots, \rho_m)[g;\vec{x}] \; . \end{aligned}$$

If $\rho \equiv D(\rho_1, \rho_2, \rho_3)$ then, since ρ_1 is governed in τ by ϕ we get $\rho_1[f;\vec{x}] \simeq \rho_1[g;\vec{x}]$ from IH and we consider three subcases. If $\rho_1[f;\vec{x}]$ diverges then so does $\rho_1[g;\vec{x}]$ and hence both terms in the conclusion of (2) diverge. If $\phi \land \exists y(\rho_1[f;\vec{x}] \asymp y \land y > 0)$ holds then, since ρ_2 is governed in τ by the last formula, we get $\rho_2[f;\vec{x}] \simeq \rho_2[g;\vec{x}]$ by IH. Thus

$$\begin{aligned} D(\rho_1,\rho_2,\rho_3)[f;\vec{x}] \simeq D(\rho_1[f;\vec{x}], \rho_2[f;\vec{x}], \rho_3[f;\vec{x}]) \simeq \rho_2[f;\vec{x}] \overset{IH}{\simeq} \rho_2[g;\vec{x}] \simeq \\ D(\rho_1[f;\vec{x}], \rho_2[g;\vec{x}], \rho_3[g;\vec{x}]) \overset{IH}{\simeq} D(\rho_1[g;\vec{x}], \rho_2[g;\vec{x}], \rho_3[g;\vec{x}]) \simeq \\ D(\rho_1, \rho_2, \rho_3)[g;\vec{x}] \ . \end{aligned}$$

The subcase when $\phi \land \rho_1[f;\vec{x}] \asymp 0$ is similar.

With the auxiliary property proved we take the term τ itself as the subterm ρ in the property and note that its governing condition $\top$ holds. ■

2.1.16 Regular recursive definitions. We will now describe a sufficient condition on a recursive definition $f(\vec{x}) \simeq \tau[f;\vec{x}]$ from total functions to define a (total) function. Assume that we are given a *measure* function m of the same arity as f and which is into ordinal numbers. We call the recursive definition *regular in m* if for every recursive occurrence $f(\rho_1[f;\vec{x}], \dots, \rho_n[f;\vec{x}])$ in τ and for its governing condition $\phi[f;\vec{x}]$ the following *condition of regularity* holds:

$$\phi[f;\vec{x}] \land \rho_1[f;\vec{x}] \asymp y_1 \land \dots \land \rho_n[f;\vec{x}] \asymp y_n \to m(y_1, \dots, y_n) < m(\vec{x}) \ . \qquad (1)$$

We say that the above recursive definition is *regular in the i-th argument* if $m(\vec{x}) = x_i$.

Regularity asserts the same as the informal phrase that the *recursion in the clausal definition decreases in the measure m.*

Our measure functions m are almost always into finite ordinals, i.e. $m(\vec{x}) < \omega$, and so we can treat them as ordinary functions over $\mathbb{N}$. We will give in Paragraphs 5.1.6 and 5.1.7 two examples of regular definitions with measure functions which are into ordinals $<\omega^2$.

2.1.17 Theorem *Regular recursive definitions define total functions.*

Proof. Assume that a recursive definition $f(\vec{x}) \simeq \tau[f;\vec{x}]$ is regular in a measure function m. Let h be the completion of f. We prove by transfinite induction on the measure m of $\vec{x}$ that $f(\vec{x})\!\downarrow$ holds. So take any $\vec{x}$. By induction on the structure of subterms ρ of τ we prove an auxiliary property:

$$\phi[f;\vec{x}] \to \rho[f;\vec{x}]\!\downarrow$$

where ϕ governs ρ in τ. So assume $\phi[f;\vec{x}]$, take any recursive application $f(\rho_1, \dots, \rho_n)$ occurring in ρ which is governed in ρ by $\phi_1[f;\vec{x}]$, and assume $\phi_1[f;\vec{x}]$. For every $1 \leq i \leq n$ there is by inner IH an y_i such that $\rho_i[f;\vec{x}] \asymp y_i$

holds. Since $\phi \wedge \phi_1$ governs the recursive application in τ, we obtain from the condition of regularity $m(y_1, \ldots, y_n) \leq m(\vec{x})$ and so

$$f(\rho_1[f;\vec{x}], \ldots, \rho_n[f;\vec{x}]) \simeq f(y_1, \ldots, y_n) \overset{\text{outer IH}}{\simeq} h(y_1, \ldots, y_n) \simeq h(\rho_1[f;\vec{x}], \ldots, \rho_n[f;\vec{x}]) \; .$$

Theorem 2.1.15 now applies and we have $\rho[f;\vec{x}] \simeq \rho[h;\vec{x}]$. Since the last term is total, we have $\rho[f;\vec{x}]\!\downarrow$.

With the auxiliary property proved we take $\rho \equiv \tau$ with its governing condition $\phi \equiv \top$ and obtain $\tau[f;\vec{x}]\!\downarrow$ from it. From $f(\vec{x}) \simeq \tau[f;\vec{x}]$ we thus get $f(\vec{x})\!\downarrow$. ■

2.2 Clausal Definitions

We can significantly improve the readability of explicit and recursive definitions by writing them in clausal form. A clausal definition of a function f is a set of clauses which can be folded, i.e. transformed by a set of equivalent transformations, into an explicit or recursive definition of f.

2.2.1 Refinement clauses. A *refinement clause* for an n-ary function symbol f is of a form

$$f(\vec{x}) \asymp v \leftarrow \phi[\vec{x}, \vec{y}] \qquad (1)$$

where v is a variable which will be eventually assigned to by a *value assignment* of a possibly partial term $\tau[\vec{x}, \vec{y}]$ whereby we obtain a *terminal* clause:

$$f(\vec{x}) \asymp v \leftarrow \phi[\vec{x}, \vec{y}] \wedge \tau[\vec{x}, \vec{y}] \asymp v \; . \qquad (2)$$

Until the end of this section we will abbreviate the terminal clauses (2) to the form:

$$f(\vec{x}) \asymp \tau[\vec{x}, \vec{y}] \leftarrow \phi[\vec{x}, \vec{y}] \; .$$

The *body* ϕ of the refinement clause (1) is a conjunction of atomic formulas and their negations. Free variables in ϕ are among the indicated ones. The formula ϕ may be empty and then the refinement clause consists just of its *head*: $f(\vec{x}) \asymp v$.

The refinement clauses

$$\begin{array}{c} f(\vec{x}) \asymp v \leftarrow \phi[\vec{x}, \vec{y}] \wedge \psi_1[\vec{x}, \vec{y}, \vec{z}_1] \\ \vdots \\ f(\vec{x}) \asymp v \leftarrow \phi[\vec{x}, \vec{y}] \wedge \psi_k[\vec{x}, \vec{y}, \vec{z}_k] \end{array} \qquad (3)$$

are called the *refinement* of (1) by formulas $\psi_1[\vec{x}, \vec{y}, \vec{z}_1]$, ... , $\psi_k[\vec{x}, \vec{y}, \vec{z}_k]$. The *local* variables $\vec{z}_i$ must be new. We will describe in Paragraps 2.2.4 through 2.2.13 all refinements presently possible in CL.

2.2.2 Complete clausal definitions. A *complete terminal clausal definition* of f is every set of clauses obtained from the *initial* refinement clause $f(\vec{x}) \asymp v$ by a finite number of refinements successively applied to the clauses in the set until every clause in the set becomes a terminal clause by a value assignment.

We will in the following describe the process of *folding* which reverses the refinements done to the initial clause. We will call the sets of clauses obtained from complete terminal clausal definitions by folding *complete non-terminal clausal definitions*. A *complete clausal definition* is either a terminal or a non-terminal complete clausal definition and it has a (abbreviated) form:

$$\begin{gathered} f(\vec{x}) \asymp \tau_1[\vec{x}, \vec{y}_1] \leftarrow \phi_1[\vec{x}, \vec{y}_1] \\ \vdots \\ f(\vec{x}) \asymp \tau_k[\vec{x}, \vec{y}_k] \leftarrow \phi_k[\vec{x}, \vec{y}_k] \ . \end{gathered}$$

Although the clauses of this complete clausal definition have the form of converse implications we will understand their meaning as if they included also the normal implications:

$$f(\vec{x}) \asymp v \leftrightarrow \bigvee_{i=1}^{k} \exists \vec{y}_i (\, \phi_i[\vec{x}, \vec{y}_i] \wedge \tau_i[\vec{x}, \vec{y}_i] \asymp v \,) \ .$$

This formula is called the *completion* of the above complete clausal definition.

2.2.3 Folding. Folding is applied in reversed order to the same clauses to which refinements were applied. Consider the refinement 2.2.1(3) of 2.2.1(1) by the formulas $\psi_1[\vec{x}, \vec{y}, \vec{z}_1], \ldots, \psi_k[\vec{x}, \vec{y}, \vec{z}_k]$. When the folding process will arrive to the stage when this refinement should be undone, the refinement clauses will be a part of a complete clausal definition and they will look as follows:

$$\begin{gathered} f(\vec{x}) \asymp \tau_1[\vec{x}, \vec{y}, \vec{z}_1] \leftarrow \phi[\vec{x}, \vec{y}] \wedge \psi_1[\vec{x}, \vec{y}, \vec{z}_1] \\ \vdots \\ f(\vec{x}) \asymp \tau_k[\vec{x}, \vec{y}, \vec{z}_k] \leftarrow \phi[\vec{x}, \vec{y}] \wedge \psi_k[\vec{x}, \vec{y}, \vec{z}_k] \ . \end{gathered}$$

The folding, which depends on the particular form of refinement and will be described in the following paragraphs together with the refinements, will transform the complete clausal definition by removing the above k clauses and by replacing them with a single clause of a form:

$$f(\vec{x}) \asymp \tau'[\vec{x}, \vec{y}] \leftarrow \phi[\vec{x}, \vec{y}] \ .$$

Note that the local variables $\vec{z}_i$ are eliminated.

Folding transformations will be such that for the transformation just described we will have:

$$\phi[\vec{x}, \vec{y}] \rightarrow \bigvee_{i=1}^{k} \exists \vec{z}_i (\, \psi_i[\vec{x}, \vec{y}, \vec{z}_i] \wedge \tau_i[\vec{x}, \vec{y}, \vec{z}_i] \asymp v \,) \leftrightarrow \tau'[\vec{x}, \vec{y}] \asymp v$$

for all partial functions f and for all $\vec{x}$, and $\vec{y}$. Thus the completion of the complete clausal definition obtained by one folding is for all partial functions f equivalent to the completion of the complete clausal definition to which the folding was applied.

Folding terminates with a complete clausal definition consisting of a single clause which corresponds to the initial refinement clause and which we show in the unabbreviated form:

$$f(\vec{x}) \asymp v \leftarrow \tau[\vec{x}] \asymp v \ .$$

The completion of this complete clausal definition is $f(\vec{x}) \asymp v \leftrightarrow \tau[\vec{x}] \asymp v$ and so it can be written in a form of an explicit or recursive definition $f(\vec{x}) \simeq \tau[\vec{x}]$. The partial function f defined by this definition, i.e. the minimal partial function satisfying the functional equation, will be clearly also the minimal partial function satisfying the completions of all complete clausal definitions in the chain of folding and this f is said to be *defined* by the complete terminal clausal definition to which the chain of folding was applied.

In the following paragraphs we will describe all refinement/folding pairs presently possible in CL.

2.2.4 Assignments. *Assignment* is a refinement of a clause 2.2.1(1) by a single formula $\rho[\vec{x},\vec{y}]) \asymp z$ where z is a new variable and the term ρ is either total or it has a form $g(\vec{\rho}_1[\vec{x},\vec{y}])$ with the terms $\vec{\rho}_1$ total. We do not exclude $g \equiv f$. When folding reaches the assignment in a form:

$$f(\vec{x}) \asymp \tau[\vec{x},\vec{y},z] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] \asymp z$$

then the clause is folded to:

$$f(\vec{x}) \asymp \tau[\vec{x},\vec{y},\rho[\vec{x},\vec{y}]] \leftarrow \phi[\vec{x},\vec{y}] \quad .$$

if ρ is total or to:

$$f(\vec{x}) \asymp D(g(\vec{\rho}_1[\vec{x},\vec{y}]), \tau[\vec{x},\vec{y},g(\vec{\rho}_1[\vec{x},\vec{y}]], \tau[\vec{x},\vec{y},g(\vec{\rho}_1[\vec{x},\vec{y}]]) \leftarrow \phi[\vec{x},\vec{y}]$$

otherwise. The application of D is necessary because $\tau[\vec{x},\vec{y},g(\vec{\rho}_1[\vec{x},\vec{y}])] \asymp v$ is not in general equivalent to $\exists z(\, g(\vec{\rho}_1[\vec{x},\vec{y}]) \asymp z \wedge \tau[\vec{x},\vec{y},z] \asymp v\,)$. The latter formula is stronger than the former one because we can have $\tau[\vec{x},\vec{y},g(\vec{\rho}_1[\vec{x},\vec{y}])] \asymp v$ even if $g(\vec{\rho}_1[\vec{x},\vec{y}])$ does not denote.

Both kinds of assignments as well as the value assignments are the only refinements which may involve non-total terms.

2.2.5 D-discrimination. *D-discrimination* is a refinement of a clause 2.2.1(1) by two formulas $\rho[\vec{x},\vec{y}] > 0$ and $\rho[\vec{x},\vec{y}] = 0$ where ρ is a total term. When folding reaches the D-discriminated clauses in a form:

$$\begin{aligned} f(\vec{x}) &\asymp \tau_1[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] > 0 \\ f(\vec{x}) &\asymp \tau_2[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = 0 \end{aligned}$$

then the clauses are folded to:

$$f(\vec{x}) \asymp D(\rho[\vec{x},\vec{y}], \tau_1[\vec{x},\vec{y}], \tau_2[\vec{x},\vec{y}]) \leftarrow \phi[\vec{x},\vec{y}] \ .$$

2.2.6 Negation discrimination. *Negation discrimination* is a refinement of a clause 2.2.1(1) by two formulas $R(\vec{\rho}[\vec{x},\vec{y}])$ and $\neg R(\vec{\rho}[\vec{x},\vec{y}])$ where the characteristic function R_* and the terms $\vec{\rho}$ are total. When folding reaches the negation discriminated clauses in a form:

$$\begin{array}{l} f(\vec{x}) \asymp \tau_1[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \land R(\vec{\rho}[\vec{x},\vec{y}]) \\ f(\vec{x}) \asymp \tau_2[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \land \neg R(\vec{\rho}[\vec{x},\vec{y}]) \end{array}$$

then the clauses are folded to:

$$f(\vec{x}) \asymp D(R_*(\vec{\rho}[\vec{x},\vec{y}]), \tau_1[\vec{x},\vec{y}], \tau_2[\vec{x},\vec{y}]) \leftarrow \phi[\vec{x},\vec{y}] \ .$$

2.2.7 Dichotomy discrimination. *Dichotomy discrimination* is a refinement of a clause 2.2.1(1) by two formulas $\rho_1[\vec{x},\vec{y}] \leq \rho_2[\vec{x},\vec{y}]$ and $\rho_1[\vec{x},\vec{y}] > \rho_2[\vec{x},\vec{y}]$ where the terms ρ_1 and ρ_2 are total. When folding reaches the dichotomy discriminated clauses in a form:

$$\begin{array}{l} f(\vec{x}) \asymp \tau_1[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \land \rho_1[\vec{x},\vec{y}] \leq \rho_2[\vec{x},\vec{y}] \\ f(\vec{x}) \asymp \tau_2[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \land \rho_1[\vec{x},\vec{y}] > \rho_2[\vec{x},\vec{y}] \end{array}$$

then the clauses are folded to:

$$f(\vec{x}) \asymp D((\rho_1[\vec{x},\vec{y}] \leq_* \rho_2[\vec{x},\vec{y}]), \tau_1[\vec{x},\vec{y}], \tau_2[\vec{x},\vec{y}]) \leftarrow \phi[\vec{x},\vec{y}] \ .$$

2.2.8 Trichotomy discrimination. *Trichotomy discrimination* is a refinement of a clause 2.2.1(1) by three formulas $\rho_1[\vec{x},\vec{y}] < \rho_2[\vec{x},\vec{y}]$, $\rho_1[\vec{x},\vec{y}] = \rho_2[\vec{x},\vec{y}]$, and $\rho_1[\vec{x},\vec{y}] > \rho_2[\vec{x},\vec{y}]$ where the terms ρ_1 and ρ_2 are total. When folding reaches the trichotomy discriminated clauses in a form:

$$\begin{array}{l} f(\vec{x}) \asymp \tau_1[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \land \rho_1[\vec{x},\vec{y}] < \rho_2[\vec{x},\vec{y}] \\ f(\vec{x}) \asymp \tau_2[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \land \rho_1[\vec{x},\vec{y}] = \rho_2[\vec{x},\vec{y}] \\ f(\vec{x}) \asymp \tau_3[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \land \rho_1[\vec{x},\vec{y}] > \rho_2[\vec{x},\vec{y}] \end{array}$$

then the clauses are folded to:

$$\begin{array}{l} f(\vec{x}) \asymp D((\rho_1[\vec{x},\vec{y}] \leq_* \rho_2[\vec{x},\vec{y}]), \\ \qquad\qquad D((\rho_2[\vec{x},\vec{y}] \leq_* \rho_1[\vec{x},\vec{y}]), \tau_2[\vec{x},\vec{y}], \tau_1[\vec{x},\vec{y}]), \tau_3[\vec{x},\vec{y}]) \leftarrow \phi[\vec{x},\vec{y}] \ . \end{array}$$

2.2.9 Discrimination on constant patterns. *Discrimination on constant patterns* is a refinement of a clause 2.2.1(1) by $k+1$ formulas $\rho[\vec{x},\vec{y}] = c_1$, $\rho[\vec{x},\vec{y}] = c_2$, ... , $\rho[\vec{x},\vec{y}] = c_k$, and

$$\rho[\vec{x},\vec{y}] = z \land z \neq c_1 \land \cdots \land z \neq c_k$$

where the term ρ is total, the variable z new, and the constants c_1, c_2 ... , c_k denote pairwise different numbers. When folding reaches the clauses discriminated on constant patterns in a form:

$$
\begin{aligned}
&f(\vec{x}) \asymp \tau_1[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = c_1\\
&f(\vec{x}) \asymp \tau_2[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = c_2\\
&\quad\vdots\\
&f(\vec{x}) \asymp \tau_k[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = c_k\\
&f(\vec{x}) \asymp \tau_{k+1}[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = z \wedge z \neq c_1 \wedge \cdots \wedge z \neq c_k
\end{aligned}
$$

then the clauses are folded to:

$$
\begin{aligned}
f(\vec{x}) \asymp\ & D(K_*^{c_1}(\rho[\vec{x},\vec{y}]), \tau_1[\vec{x},\vec{y}],\\
& D(K_*^{c_2}(\rho[\vec{x},\vec{y}]), \tau_2[\vec{x},\vec{y}], \dots,\\
& D(K_*^{c_k}(\rho[\vec{x},\vec{y}]), \tau_k[\vec{x},\vec{y}], \tau_{k+1}[\vec{x},\vec{y}]) \dots)) \leftarrow \phi[\vec{x},\vec{y}]
\end{aligned}
$$

where for a constant c we denote by K_*^c the characteristic function of the predicate $x = c$.

2.2.10 Discrimination on pair patterns. *Discrimination on pair patterns* is a refinement of a clause 2.2.1(1) by two formulas $\rho[\vec{x},\vec{y}] = 0$ and and $\rho[\vec{x},\vec{y}] = v, w$ where the variables v and w are new and the term is ρ total. When folding reaches the clauses discriminated on pair patterns in a form:

$$
\begin{aligned}
&f(\vec{x}) \asymp \tau_1[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = 0\\
&f(\vec{x}) \asymp \tau_2[\vec{x},\vec{y},v,w] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = v, w
\end{aligned}
$$

then the clauses are folded to:

$$
f(\vec{x}) \asymp D(\rho[\vec{x},\vec{y}], \tau_2[\vec{x},\vec{y}, H(\rho[\vec{x},\vec{y}]), T(\rho[\vec{x},\vec{y}])], \tau_1[\vec{x},\vec{y}]) \leftarrow \phi[\vec{x},\vec{y}]\ .
$$

2.2.11 Discrimination on p-adic patterns (I). For two constants $p > 0$ and $m > 0$ *discrimination on p-adic patterns* of the first kind is a refinement of a clause 2.2.1(1) by $m+1$ formulas $\rho[\vec{x},\vec{y}] = 0$, $\rho[\vec{x},\vec{y}] = 1$, … , $\rho[\vec{x},\vec{y}] = m-1$, and

$$
\rho[\vec{x},\vec{y}] = p{\cdot}z + i \wedge m \leq i \wedge i < p + m
$$

where the variables z and i are new and the term ρ is total. When folding reaches the clauses discriminated on p-adic patterns in a form:

$$
\begin{aligned}
&f(\vec{x}) \asymp \tau_0[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = 0\\
&f(\vec{x}) \asymp \tau_2[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = 1\\
&\quad\vdots\\
&f(\vec{x}) \asymp \tau_{m-1}[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = m-1\\
&f(\vec{x}) \asymp \tau_m[\vec{x},\vec{y},z,i] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = p{\cdot}z + i \wedge m \leq i \wedge i < p + m
\end{aligned}
$$

then the clauses are folded to:

$$
\begin{aligned}
f(\vec{x}) \asymp\ & D(K_*^{0}(\rho[\vec{x},\vec{y}]), \tau_0[\vec{x},\vec{y}],\\
& D(K_*^{1}(\rho[\vec{x},\vec{y}]), \tau_1[\vec{x},\vec{y}], \dots,\\
& D(K_*^{m-1}(\rho[\vec{x},\vec{y}]), \tau_{m-1}[\vec{x},\vec{y}],\\
& \quad \tau_m[\vec{x},\vec{y}, (\rho[\vec{x},\vec{y}] \dot{-} m) \div p, (\rho[\vec{x},\vec{y}] \dot{-} m) \bmod p + m]) \dots)) \leftarrow \phi[\vec{x},\vec{y}]\ .
\end{aligned}
$$

2.2.12 Discrimination on p-adic patterns (II). For two constants $p > 0$ and $m > 0$ *discrimination on pair patterns* of the second kind is a refinement of a clause 2.2.1(1) by $m+p$ formulas $\rho[\vec{x},\vec{y}] = 0$, $\rho[\vec{x},\vec{y}] = 1$, ... , $\rho[\vec{x},\vec{y}] = m-1$, $\rho[\vec{x},\vec{y}] = p{\cdot}z+m$, $\rho[\vec{x},\vec{y}] = p{\cdot}z+m+1$, ... , and $\rho[\vec{x},\vec{y}] = p{\cdot}z+m+p-1$ where the variable z is new and the term ρ is total. When folding reaches the clauses discriminated on p-adic patterns in a form:

$$
\begin{aligned}
&f(\vec{x}) \asymp \tau_0[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = 0\\
&f(\vec{x}) \asymp \tau_2[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = 1\\
&\quad\vdots\\
&f(\vec{x}) \asymp \tau_{m-1}[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = m-1\\
&f(\vec{x}) \asymp \tau_m[\vec{x},\vec{y},z] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = p{\cdot}z+m\\
&f(\vec{x}) \asymp \tau_{m+1}[\vec{x},\vec{y},z] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = p{\cdot}z+m+1\\
&\quad\vdots\\
&f(\vec{x}) \asymp \tau_{m+p-1}[\vec{x},\vec{y},z] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = p{\cdot}z+m+p-1
\end{aligned}
$$

then the clauses are folded to:

$$
\begin{aligned}
f(\vec{x}) \asymp\ & D(K_*^0(\rho[\vec{x},\vec{y}]),\tau_0[\vec{x},\vec{y}],\\
& D(K_*^1(\rho[\vec{x},\vec{y}]),\tau_1[\vec{x},\vec{y}],\dots,\\
& D(K_*^{m-1}(\rho[\vec{x},\vec{y}]),\tau_{m-1}[\vec{x},\vec{y}],\\
& D(K_*^0((\rho[\vec{x},\vec{y}] \dot{-} m) \bmod p),\tau_m[\vec{x},\vec{y},\rho[\vec{x},\vec{y}] \dot{-} m) \div p],\\
& D(K_*^1((\rho[\vec{x},\vec{y}] \dot{-} m) \bmod p),\tau_{m+1}[\vec{x},\vec{y},\rho[\vec{x},\vec{y}] \dot{-} m) \div p],\dots,\\
& D(K_*^{p-1}((\rho[\vec{x},\vec{y}] \dot{-} m) \bmod p),\tau_{m+p-1}[\vec{x},\vec{y},\rho[\vec{x},\vec{y}] \dot{-} m) \div p],\\
&\quad 0)\dots)))\dots)) \leftarrow \phi[\vec{x},\vec{y}]\ .
\end{aligned}
$$

2.2.13 Discrimination on constructor patterns. A *pair constructor* is either a constant $c = d,0$ or an n-ary function $f(\vec{x}) = d,\vec{x}$ where d is a constant, called the *tag* . A *numeric constructor* is either a constant c or a unary function $f(x) = p{\cdot}x + c$ where $p > 0$ is the *base* and $c > 0$ is the *tag*.

Discrimination on constructor patterns is a refinement of a clause 2.2.1(1) by $k+1$ formulas $\rho[\vec{x},\vec{y}] = \rho_1$, $\rho[\vec{x},\vec{y}] = \rho_2$, ... , $\rho[\vec{x},\vec{y}] = \rho_k$, and

$$\rho[\vec{x},\vec{y}] = z \wedge \forall z \neq \rho_1 \wedge \forall z \neq \rho_2 \wedge \cdots \wedge \forall z \neq \rho_k$$

where the term ρ is total. The terms ρ_i are either constructor constants or applications of constructor functions $f(\vec{w})$ to new variables. All constructors must be either pair or numeric constructors. Pair constructors must have pairwise different tags. Numeric constructors must have the same base $p > 0$ and be such that for some number $m > 0$ the tags t of all constant constructors are pairwise different and such that $t < m$ and the tags t of all function constructors are pairwise different and are such that $m \leq t < m+p$. Here, if $\rho_i \equiv f(\vec{w})$ then we denote by $\forall z \neq \rho_i$ the formula $\forall \vec{w}\, z \neq f(\vec{w})$ and if ρ_i is a constant then $\forall z \neq \rho_i$ is the same as $z \neq \rho_i$. Eventually the folding process reaches the clauses discriminated on constructor patterns in a form:

$$f(\vec{x}) \asymp \tau_1[\vec{x},\vec{y},\vec{w}_1] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = \rho_1[\vec{w}_1]$$

$$
\begin{aligned}
&f(\vec{x}) \asymp \tau_2[\vec{x},\vec{y},\vec{w}_2] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = \rho_2[\vec{w}_2]\\
&\ \vdots\\
&f(\vec{x}) \asymp \tau_k[\vec{x},\vec{y},\vec{w}_k] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = \rho_k[\vec{w}_k]\\
&f(\vec{x}) \asymp \tau_{k+1}[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}] \wedge \rho[\vec{x},\vec{y}] = z \wedge \forall z \neq \rho_1 \wedge \forall z \neq \rho_2 \wedge \cdots \wedge \forall z \neq \rho_k\ .
\end{aligned}
$$

We leave to the reader the folding of the $k+1$ clauses to a single clause:

$$f(\vec{x}) \asymp \tau[\vec{x},\vec{y}] \leftarrow \phi[\vec{x},\vec{y}]$$

by tests on tags and by elimination of the variables $\vec{w}_i$ by means of *destructors* which are compositions of projection functions H and T for the pair constructors and by the function $(x \dot{-} m) \div p$ for the numeric constructors.

2.2.14 Presentation of clausal definitions. We may simplify a complete clausal definition of f for the purposes of presentation (to a human reader) to a *clausal definition* by omitting from it one or more *default* clauses which are clauses with the heads $f(\vec{x}) = 0$ and such that the last atomic formula in their bodies before the value assignment is an discrimination formula from a set of refinement formulas such that at least one of the refinement formulas occurs in some non-default clause.

We may write the clauses of a form:

$$f(\vec{x}) \asymp v \leftarrow \phi_1 \wedge g(\vec{\rho}) \asymp y \wedge \phi_2[y]$$

with g total (we do not exclude $g \equiv f$) in the form

$$f(\vec{x}) \asymp v \leftarrow \phi_1 \wedge g(\vec{\rho}) = y \wedge \phi_2[y]$$

or even as

$$f(\vec{x}) \asymp v \leftarrow \phi_1 \wedge \phi_2[g(\vec{\rho})]\ .$$

If the term τ in the value assignment of the following clause is total:

$$f(\vec{x}) \asymp \tau \leftarrow \phi$$

we may write the clause as:

$$f(\vec{x}) = \tau \leftarrow \phi\ .$$

In all clauses discriminated on (constant, pair, p-adic, and constructor) patterns with the formulas $x_i = \rho_j$ where the variable x_i occurs only once to the left of the discrimination formula we may substitute the term ρ_j for the variable x_i and remove the discrimination formulas in each of the discriminated clauses.

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.

2.2.15 Clausal definitions of (total) functions. Explicit clausal definitions which fold into explicit definitions $f(\vec{x}) \simeq \tau$ from (total) functions obviously define total functions.

We will now formulate *conditions of regularity* on a recursive clausal definition from total functions such that the definition folds to a recursive definition regular in a measure function m. Every clause of the clausal definition can be written as:

$$f(\vec{x}) \asymp \tau \leftarrow \phi_1 \wedge f(\vec{\rho}_1) \asymp y_1 \wedge \phi_2 \wedge f(\vec{\rho}_2) \asymp y_2 \wedge \cdots \wedge \phi_k \wedge f(\vec{\rho}_k) \asymp y_k \wedge \phi_{k+1}$$

where no formula $\phi_1, \phi_2, \ldots, \phi_{k+1}$ applies f and so the formulas contain only total terms. Note that the terms in $\vec{\rho}_i$ may not apply f because they must be total. Initially also the term τ is total but during the folding process the term τ will accumulate applications of f. The above clause is *regular in m* if we have:

$$\begin{aligned} \phi_1 &\to m(\vec{\rho}_1) < m(\vec{x}) \\ \phi_1 \wedge \phi_2 &\to m(\vec{\rho}_2) < m(\vec{x}) \\ &\vdots \\ \phi_1 \wedge \phi_2 \cdots \wedge \phi_k &\to m(\vec{\rho}_k) < m(\vec{x}) \end{aligned}$$

and if τ applies f we must also have

$$\phi_1 \wedge \phi_2 \cdots \wedge \phi_k \wedge \phi_{k+1} \to \phi$$

for every condition of regularity ϕ for the term τ of a form 2.1.16(1).

The clauses are *regular in m* if every clause with recursive applications of f is regular in m. It can be checked that the folding transformations preserve the regularity of clauses and so we obtain at end a recursive definition regular in m. By Thm. 2.1.17 regular clausal definitions thus define total functions.

The reader will note that because of our restrictions on the default clauses there can be no recursive applications of f in default clauses which do not occur in some clause of a definition.

2.2.16 Clausal definitions of predicates. We can define a predicate R by a complete clausal definition which defines its characteristic (total) functions R_* and is such that the heads of the clauses have the form $R_*(\vec{x}) = 1$ or $R_*(\vec{x}) = 0$ and all applications of R_* in the bodies are in clauses D-discriminated with formulas $R_*(\vec{\rho}) > 0$ and $R_*(\vec{\rho}) = 0$.

We can present such a definition in a predicate form where we replace all pairs of D-discrimination formulas $R_*(\vec{\rho}) > 0$ and $R_*(\vec{\rho}) = 0$ by a pair of negation discrimination formulas $R(\vec{\rho})$ and $\neg R(\vec{\rho})$. We must also remove all defaults and replace every $R_*(\vec{x}) = 1$ in the heads by $R(\vec{x})$.

2.3 Partial Recursive Functions

This section defines by metamathematical, i.e. by language, means the class of (partial) recursive functions and shows that the functions in the class are effectively computable. The section contains theorems which are quite intuitive to understand but technically tedious to prove. The theorems relate the definitions of recursive functions (denotational semantics) to their computations (operational semantics).

2.3.1 Terms recursive in a class of functions. For a given denumerable (possibly finite or empty) *class* (set) of (total) functions $\mathcal{F}$ enumerated by

$$g_0 \; g_1 \; g_2 \; \cdots$$

we define the set $R^{\mathcal{F}}$ of terms *recursive in* $\mathcal{F}$, or just $R^{\mathcal{F}}$-terms, as the smallest set of terms satisfying the following;

- the constant 0, and the variables $x_1, x_2, \ldots$ are $R^{\mathcal{F}}$-terms,
- if τ is an $R^{\mathcal{F}}$-term so are $(\tau + 1)$ and $(\tau \dot{-} 1)$,
- if τ_1, τ_2, and τ_3 are $R^{\mathcal{F}}$-terms so is $D(\tau_1, \tau_2, \tau_3)$,
- if $g_i \in \mathcal{F}$ is n-ary, $\vec{\rho}$ is an n-tuple of $R^{\mathcal{F}}$-terms then $g_i(\vec{\rho})$ is an $R^{\mathcal{F}}$-term,
- if $\vec{\rho}$ is an n-tuple of $R^{\mathcal{F}}$-terms then $f(\vec{\rho})$ is an $R^{\mathcal{F}}$-term,
- if $\vec{x}$ is an n-tuple of variables, $\tau \in R^{\mathcal{F}}$, and $\vec{\rho}$ an n-tuple of $R^{\mathcal{F}}$-terms then $(\lambda\vec{x}.\tau)(\vec{\rho})$ is an $R^{\mathcal{F}}$-term subject to restrictions given below.

Expressions of a form $\lambda\vec{x}.\tau[f;\vec{x}]$ are called *$R^{\mathcal{F}}$-function symbols* and are intended to denote partial functions. $R^{\mathcal{F}}$-terms of form $(\lambda\vec{x}.\tau)(\vec{\rho})$ are applications of $R^{\mathcal{F}}$-function symbols.

An occurrence of a variable x in an $R^{\mathcal{F}}$-term τ is *free* if it it does not occur in the term τ_1 of an $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.\tau_1$ which is applied in τ. The occurrence is *bound* otherwise.

Likewise, an occurrence of the *recursor* f in an application $f(\vec{\rho})$ is *free* in τ if the application does not occur in the term τ_1 of an $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.\tau_1$ which is applied in τ. The occurrence of the recursor is *bound* otherwise.

An $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.\tau[f;\vec{x}]$ is legal if all free occurrences of variables in τ are of variables from among the n-tuple of variables $\vec{x}$ and if all free occurrences of recursors f in τ are applied with n-tuples of $R^{\mathcal{F}}$-terms: $f(\vec{\rho})$.

An $R^{\mathcal{F}}$-term is *closed* if it does not have free occurrences of the recursor f or of any variable.

2.3.2 Notation for n-tuples of monadic numerals. We will need a notation for n-tuples of monadic numerals, which are $R^{\mathcal{F}}$-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 $\underline{x} \equiv S^{x}(0)$.

2.3.3 Standard interpretation of $R^{\mathcal{F}}$-terms. For a given a class of functions $\mathcal{F}$ enumerated by $g_0, g_1, \ldots$ the $R^{\mathcal{F}}$-terms obtain the standard partial denotations once we we interprete the function symbols g_i as total functions by making them denote the corresponding functions from the enumeration of $\mathcal{F}$ and we interprete $R^{\mathcal{F}}$-function symbols $\lambda\vec{x}.\tau$ as partial functions. The last is done by induction on the $R^{\mathcal{F}}$-terms τ. A term $\tau[f;\vec{x}]$ such that $\lambda\vec{x}.\tau$ is a legal $R^{\mathcal{F}}$-function symbol has all applied $R^{\mathcal{F}}$-function symbols interpreted by IH and we interprete the function symbol $\lambda\vec{x}.\tau$ 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.

2.3.4 Partial recursive functions. For a class $\mathcal{F}$ of functions we define the class of partial functions *recursive in* $\mathcal{F}$ as the class of all partial functions denoted by $R^{\mathcal{F}}$-function symbols for some enumeration $g_0, g_1, \ldots$ of $\mathcal{F}$. The class of *partial recursive* functions consists of the partial functions recursive in the empty class $\emptyset$.

The class of functions *recursive in* $\mathcal{F}$ is the class of partial functions recursive in $\mathcal{F}$ which happen to be total. The class of *recursive* functions consists of total partial recursive functions.

We will often write *partial* $\mathcal{F}$*-recursive* function instead of *partial function recursive in* $\mathcal{F}$ and $\mathcal{F}$*-recursive* function instead of *function recursive in* $\mathcal{F}$.

The meaning of $R^{\mathcal{F}}$-function symbols depends on the particular enumeration of $\mathcal{F}$ but the class of partial $\mathcal{F}$-recursive functions does not because if such a partial function is denoted by $\lambda\vec{x}.\tau$ in one enumeration of $\mathcal{F}$ it is denoted by $\lambda\vec{x}.\tau'$ in another enumeration where we systematically rename the function symbols g_i to reflect the new enumeration.

2.3.5 Some recursive functions. Every function $g \in \mathcal{F}$ is $\mathcal{F}$-recursive because for any enumeration $g_0, g_1, \ldots$ of $\mathcal{F}$ we have $g = g_i$ and so g is denoted by the $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.g_i(\vec{x})$.

The functions $x+1$, $x \dot{-} 1$, and D are recursive functions because they are denoted by the corresponding R-function symbols $\lambda x_1.x_1 + 1$, $\lambda x_1.x_1 \dot{-} 1$, and $\lambda x_1 x_2 x_3.D(x_1, x_2, x_3)$.

2.3.6 Reductions of closed $R^{\mathcal{F}}$-terms. For any class of functions $\mathcal{F}$ enumerated by $g_0, g_1, \ldots$ we now describe a possibly infinite process by which we can *reduce* (compute, calculate, simplify) a closed $R^{\mathcal{F}}$-term until we obtain a monadic numeral which cannot be further reduced.

If a closed $R^{\mathcal{F}}$-term τ is not a monadic numeral then it must contain at least one occurrence, called the *redex* (for *reducible expression*), of one of the closed

$R^{\mathcal{F}}$-terms listed in the following expressions to the left of the symbols $\blacktriangleright^{\mathcal{F}}$:

$$\begin{aligned}
(\underline{k} \dot{-} 1) &\blacktriangleright^{\mathcal{F}} \underline{k \dot{-} 1} \\
D(0, \tau_1, \tau_2) &\blacktriangleright^{\mathcal{F}} \tau_2 \\
D(\underline{k+1}, \tau_1, \tau_2) &\blacktriangleright^{\mathcal{F}} \tau_1 \\
(\lambda \vec{x}.\tau[f; \vec{x}])(\underline{\vec{k}}) &\blacktriangleright^{\mathcal{F}} \tau[\lambda \vec{x}.\tau; \underline{\vec{k}}] \\
g_i(\underline{\vec{k}}) &\blacktriangleright^{\mathcal{F}} \underline{g_i(\vec{k})} \ .
\end{aligned}$$

Here n, k, $k_1, \ldots$ are arbitrary natural numbers and τ_1, τ_2 are closed $R^{\mathcal{F}}$-terms.

One step reduction consists of locating the leftmost redex in τ and replacing it by its *contraction*, which is the closed term listed to the right of the symbol $\blacktriangleright^{\mathcal{F}}$ of the corresponding redex-contraction pair. By the replacement we obtain again a closed $R^{\mathcal{F}}$-term τ_1. We note that the term τ_1 is uniquely determined by τ.

We say that τ_1 *reduces to* τ_2 *in at most* n *steps*, in symbols $\tau_1 \blacktriangleright^{\mathcal{F}}_n \tau_2$, if there is a finite *one step reduction* sequence of closed $R^{\mathcal{F}}$-terms ρ_0, ρ_1, $\ldots$ ρ_k such that $0 \leq k \leq n$, $\rho_0 \equiv \tau_1$, $\rho_k \equiv \tau_2$, and for each $i < k$ we obtain the term ρ_{i+1} by one step reduction from the term ρ_i. We write $\tau_1 \blacktriangleright^{\mathcal{F}} \tau_2$ if $\tau_1 \blacktriangleright^{\mathcal{F}}_n \tau_2$ for some n.

It is not difficult to see that for every closed $R^{\mathcal{F}}$-terms τ, ρ_1, and ρ_2 we have

$$\tau \blacktriangleright^{\mathcal{F}} \rho_1 \wedge \tau \blacktriangleright^{\mathcal{F}} \rho_2 \rightarrow \rho_1 \blacktriangleright^{\mathcal{F}} \rho_2 \vee \rho_2 \blacktriangleright^{\mathcal{F}} \rho_1 \ .$$

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

2.3.7 Effectivity of reductions. We can always effectively determine if a sequence of symbols is a closed $R^{\mathcal{F}}$-term. We can also effectively recognize a monadic numeral. If a closed $R^{\mathcal{F}}$-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 $\mathcal{F} = \emptyset$, the reductions are effective. If $\mathcal{F}$ is not empty then we can perform the reductions only *relatively* in a sense that we obtain the monadic numeral to the right of $\blacktriangleright^{\mathcal{F}}$ of a redex-contraction pair $g_i(\underline{k_1, \ldots, k_n}) \blacktriangleright^{\mathcal{F}} \underline{g(k_1, \ldots, k_n)}$ as an answer to a question to an *oracle* for g_i. 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 $i + k_1 + \cdots + k_n$ 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 are total. However, the process of searching through the sequence is not effective because we cannot in general effectively present the sequence.

2.3.8 Denotational versus operational semantics. $R^{\mathcal{F}}$-function symbols $\lambda\vec{x}.\tau$ can be both used and mentioned. The function symbol $\lambda\vec{x}.\tau$ is used as a name denoting a function f over $\mathbb{N}$. The same symbol is mentioned as a *program* (rule) for the computation of the function f by reductions.

We will prove in Corollary 2.3.15 that the partial recursive functions can be computed by reductions in exactly those points in which they are defined. This will establish the equivalence of so-called *denotational* (definitional) semantics with the *operational* (computational) semantics. That everything computable is definable is proved in Thm. 2.3.9. The converse, that everything definable is computable, will be proved by a detour through approximation functions in two theorems 2.3.12 and 2.3.14.

2.3.9 Theorem *For every class of functions $\mathcal{F}$ and every closed $R^{\mathcal{F}}$-term τ we have*

$$\tau \blacktriangleright_k^{\mathcal{F}} \underline{y} \rightarrow \tau \asymp y \ . \tag{1}$$

Proof. Let $\mathcal{F}$ be enumerated by $g_0, g_1, \ldots$. We prove (1) for all y by induction on the measure $k + s_\tau$ where s_τ is the number of subterms of τ of the form $(\rho + 1)$ such that ρ is not a monadic numeral. So take any closed $R^{\mathcal{F}}$-term τ, any y, assume $\tau \blacktriangleright_k^{\mathcal{F}} \underline{y}$, and continue by the case analysis of τ. If $\tau \equiv 0$ then we have $0 \asymp y$ trivially.

If $\tau \equiv (\rho + 1)$ then there is a y_1 such that:

$$(\rho + 1) \blacktriangleright_k^{\mathcal{F}} (\underline{y_1} + 1) \equiv \underline{y_1 + 1}$$

and so $y_1 + 1 = y$. We have $\rho \asymp y_1$ either directly if $\rho \equiv \underline{y_1}$ or by IH because otherwise we have $s_\rho + 1 = s_\tau$. Hence $\rho + 1 \asymp y$ holds.

If $\tau \equiv (\rho \dot{-} 1)$ then $k > 0$ and there is a y_1 such that $\rho \blacktriangleright_{k-1}^{\mathcal{F}} y_1$ and

$$(\rho \dot{-} 1) \blacktriangleright_{k-1}^{\mathcal{F}} (\underline{y_1} \dot{-} 1) \blacktriangleright_1^{\mathcal{F}} \underline{y_1 \dot{-} 1} \ .$$

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

If $\tau \equiv D(\rho_1, \rho_2, \rho_3)$ then there is a y_1 and and a minimal $k_1 < k$ such that $\rho_1 \blacktriangleright_{k_1}^{\mathcal{F}} y_1$. By IH we have $\rho_1 \asymp y_1$ and we consider two subcases. If $y_1 > 0$ then we have

$$D(\rho_1, \rho_2, \rho_3) \blacktriangleright_{k_1}^{\mathcal{F}} D(\underline{y_1}, \rho_2, \rho_3) \blacktriangleright_1^{\mathcal{F}} \rho_2 \blacktriangleright_{k-k_1-1}^{\mathcal{F}} y \ .$$

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

If $\tau \equiv g_i(\vec{\rho})$ where g_i is n-ary then for all $1 \leq j \leq n$ there are numbers k_j and y_j such that $\rho_j \blacktriangleright_{k_j}^{\mathcal{F}} \underline{y_j}$ and $\sum_j k_j < k$. We then have:

$$g_i(\vec{\rho}) \blacktriangleright_{\sum_j k_j}^{\mathcal{F}} g_i(\underline{\vec{y}}) \blacktriangleright_1^{\mathcal{F}} \underline{g_i(\vec{y})} \ .$$

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

If $\tau \equiv (\lambda\vec{x}.\tau_1[f;\vec{x}])(\vec{\rho})$ where the function symbol $\lambda\vec{x}.\tau_1$ is n-ary then for all $1 \leq i \leq n$ there are numbers k_i and y_i such that $\rho_i \blacktriangleright^{\mathcal{F}}_{k_i} \underline{y_i}$ and $\sum_i k_i < k$. We then have:

$$(\lambda\vec{x}.\tau_1)(\vec{\rho}) \blacktriangleright^{\mathcal{F}}_{\sum_i k_i} (\lambda\vec{x}.\tau_1)(\underline{\vec{y}}) \blacktriangleright^{\mathcal{F}}_{1} \tau_1[\lambda\vec{x}.\tau_1;\underline{\vec{y}}] \blacktriangleright^{\mathcal{F}}_{k-1-\sum_i k_i} \underline{y} \ .$$

We have $\tau_1[\lambda\vec{x}.\tau_1;\underline{\vec{y}}] \asymp y$ by IH and from $(\lambda\vec{x}.\tau_1)(\underline{\vec{y}}) \simeq \tau_1[\lambda\vec{x}.\tau_1;\underline{\vec{y}}]$ we obtain $(\lambda\vec{x}.\tau_1)(\underline{\vec{y}}) \asymp y$. We have $\rho_i \asymp y_i$ by IH's and so $(\lambda\vec{x}.\tau_1)(\vec{\rho}) \asymp y$ holds. ■

2.3.10 Approximation functions. For a given class of functions $\mathcal{F}$ enumerated by g_0, g_1, ... we will now assign an $(n+1)$-ary $\mathcal{F}$-recursive (total) *approximation* function $f^+ = (\lambda\vec{x}.\tau)^+$ to every n-ary partial $\mathcal{F}$-recursive function $f = \lambda\vec{x}.\tau$.

The arguments $\vec{x}$ of $f(\vec{x})$ are passed to f^+ increased by one:

$$f^+(z, x_1{+}1, \ldots, x_n{+}1) = v \ .$$

The additional argument z is the *depth of recursion counter*. If $v > 0$ then we have $f(\vec{x}) \asymp v{-}1$ and this can be computed with the depth z of recursions and/or applications of $R^{\mathcal{F}}$-function symbols. If $v = 0$ then the depth z does not suffice. If $f(\vec{x}) \asymp y$ holds then there is a number z such that $f^+(z, x_1{+}1, \ldots, x_n{+}1) = y + 1$. Once an application yields the 'undefined' value 0 the value is propagated all the way back to the topmost application unless it 'disappears' in a branch of the case discrimination function D which does not apply. Thus the arguments 0 of f^+ code undefined values.

We first explicitly define the $(n+1)$-ary approximation function $(g_i)^+$ for every n-ary function $g_i \in \mathcal{F}$ by the following clausal definition:

$$(g_i)^+(z, x_1, \ldots, x_n) = g_i(y_1, \ldots, y_n) + 1 \leftarrow x_1 = y_1 + 1 \wedge \cdots \wedge x_n = y_n + 1 \quad .$$

The function $(g_i)^+$ is $\mathcal{F}$-recursive because we have

$$(g_i)^+ = \lambda z, x_1, x_2, \ldots, x_n.D(x_1, D(x_2, \ldots, \\ D(x_n, g_i(x_1 \dot{-} 1, x_2 \dot{-} 1, \ldots, x_n \dot{-} 1) + 1, 0) \ldots, 0), 0) \ .$$

We next define by explicit clausal definitions the approximation functions S^+, Pr^+ and D^+ for the corresponding functions $x + 1$, $x \dot{-} 1$, and D:

$$\begin{aligned}
&S^+(z, 0) = 0 \\
&S^+(z, x+1) = (x+1) + 1 \\
&Pr^+(z, 0) = 0 \\
&Pr^+(z, x+1) = (x \dot{-} 1) + 1 \\
&D^+(z, 0, y, v) = 0 \\
&D^+(z, x+1, y, v) = D(x, y, v) \ .
\end{aligned}$$

The functions are recursive because, for instance, we have

$$D^+ = \lambda z, x, y, v.D(x, D(x \dot{-} 1, y, v), 0) \ .$$

We will define the approximation functions f^+ with the help of terms denoted by ρ^+ and defined for all $\rho \in R^{\mathcal{F}}$ to satisfy:

$$\begin{aligned}
(x_i)^+ &\equiv x_i \\
0^+ &\equiv 0 + 1 \\
(\rho + 1)^+ &\equiv S^+(z, \rho^+) \\
(\rho \dot{-} 1)^+ &\equiv Pr^+(z, \rho^+) \\
(D(\rho_1, \rho_2, \rho_3))^+ &\equiv D^+(z, {\rho_1}^+, {\rho_2}^+, {\rho_3}^+) \\
(g_i(\rho_1, \dots, \rho_n))^+ &\equiv (g_i)^+(z, {\rho_1}^+, \dots, {\rho_n}^+) \\
((\lambda \vec{x}.\tau)(\rho_1, \dots, \rho_n))^+ &\equiv (\lambda \vec{x}.\tau)^+(z, {\rho_1}^+, \dots, {\rho_n}^+) \\
(f(\rho_1, \dots, \rho_n))^+ &\equiv f(z, {\rho_1}^+, \dots, {\rho_n}^+) \ .
\end{aligned}$$

We are now ready to define the $\mathcal{F}$-recursive approximation function $(\lambda \vec{x}.\tau)^+$ for every n-ary partial $\mathcal{F}$-recursive functions $\lambda \vec{x}.\tau$ by induction on the construction of $R^{\mathcal{F}}$-terms $\tau[f; \vec{x}]$. By IH the approximation functions of all $R^{\mathcal{F}}$-function symbols applied in τ are $\mathcal{F}$-recursive functions. Thus we can replace in the term $\tau^+[f; z, \vec{x}]$ all approximation function symbols $(\lambda \vec{x}.\tau_1)^+$ by $R^{\mathcal{F}}$-function symbols denoting the same functions whereby we obtain an $R^{\mathcal{F}}$-term $\rho[f; z, \vec{x}]$. Note that we have $\forall z \forall \vec{x}\, \tau^+[f; z, \vec{x}] = \rho[f; z, \vec{x}]$ for all $(n+1)$-ary functions f.

We define $f^+ = (\lambda \vec{x}.\tau)^+$ by the following clausal definition:

$$\begin{aligned}
&f^+(0, \vec{x}) = 0 \\
&f^+(z{+}1, x_1, \dots, x_n) = \tau^+[f^+; z, x_1, \dots, x_n] \leftarrow x_1 = y_1{+}1 \wedge \dots \wedge x_n = y_n{+}1 \ .
\end{aligned}$$

If f is not applied in τ then neither it is applied in τ^+ nor in ρ and the definition is explicit. If f is applied in τ then it is applied both in $\tau^+[f; z, \vec{x}]$ and in $\rho[f; z, \vec{x}]$ and the definition is regular in the first argument. In either case f is a function and it is $\mathcal{F}$-recursive because we have

$$f^+ = \lambda z, x_1, \dots, x_n.D(z, D(x_1, \dots, D(x_n, \rho[f; z \dot{-} 1, \vec{x}], 0) \dots, 0), 0) \ .$$

We now define for arbitrary partial $\mathcal{F}$-recursive function f the function f^+ to be an *approximation function for* f if there is an enumeration g_0, g_1, ... of $\mathcal{F}$ and an $R^{\mathcal{F}}$-function symbol $\lambda \vec{x}.\tau$ such that $f = \lambda \vec{x}.\tau$ and $f^+ = (\lambda \vec{x}.\tau)^+$ hold.

Theorem 2.3.11 asserts the intuitively obvious 'monotone' property of f^+ that once an application $f^+(z, \vec{x})$ yields a positive value it will do the same for every larger depth of recursion counter z. Theorem 2.3.12 asserts that if $f(\vec{x}) \asymp y$ holds then there is a value z of the depth of recursion counter such that $f^+(z, x_1{+}1, \dots, x_n{+}1) = y + 1$ holds.

2.3.11 Theorem *For every class of functions $\mathcal{F}$, every partial $\mathcal{F}$-recursive function f, and every approximation function f^+ for f we have:*

$$f^+(z_1, x_1{+}1, \ldots, x_n{+}1) = y{+}1 \land z_2 \geq z_1 \to f^+(z_2, x_1{+}1, \ldots, x_n{+}1) = y{+}1 \ .$$

Proof. The idea of the proof is that the functions S^+, Pr^+, D^+, and $(g_i)^+$ are monotone in the recursion depth counter z and this property carries over to the functions $(\lambda\vec{x}.\tau)^+$.

Let $\mathcal{F}$ be enumerated by g_0, g_1, ... We temporarily write $\vec{x}{+}\mathbf{1}$ as an abbreviation for $x_1{+}1, \ldots, x_n{+}1$. The theorem is proved by a triple induction. The outer induction is on the size of $R^{\mathcal{F}}$-function symbols $\lambda\vec{x}.\tau$:

$$\forall z_2 \forall z_1 \forall \vec{x} \forall y (\, (\lambda\vec{x}.\tau)^+(z_1, \vec{x}{+}\mathbf{1}) = y{+}1 \land z_2 \geq z_1 \to (\lambda\vec{x}.\tau)^+(z_2, \vec{x}{+}\mathbf{1}) = y{+}1 \,) \ . \tag{1}$$

We take any n-ary $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.\tau$, set $f = \lambda\vec{x}.\tau$, $f^+ = (\lambda\vec{x}.\tau)^+$, and prove by induction on z_2

$$\forall z_1 \forall \vec{x} \forall y (\, f^+(z_1, \vec{x}{+}\mathbf{1}) = y{+}1 \land z_2 > z_1 \to f^+(z_2, \vec{x}{+}\mathbf{1}) = y{+}1 \,) \ . \tag{2}$$

In the base case there is nothing to prove and in the inductive case, we take any $\vec{x}$, y, any $z_1 < z_2 + 1$, and assume $f^+(z_1, \vec{x}{+}\mathbf{1}) = y{+}1$. We can see from the definition of f^+ that $z_1 > 0$, $z_1{-}1 < z_2$, and $\tau^+[f^+; z_1{-}1, \vec{x}{+}\mathbf{1}] = y{+}1$.

We now prove by induction on subterms $\rho[f; \vec{x}]$ of τ:

$$\forall v (\, \rho^+[f^+; z_1{-}1, \vec{x}{+}\mathbf{1}] = v{+}1 \to \rho^+[f^+; z_2, \vec{x}{+}\mathbf{1}] = v{+}1 \,) \ . \tag{3}$$

We leave the details of this proof to the reader because a similar kind of induction is in the proof of Thm. 2.3.14.

With (3) proved we set in it $\rho \equiv \tau$ and obtain $\tau^+[f^+; z_2, \vec{x}{+}\mathbf{1}] = y{+}1$. Thus $f^+(z_2{+}1, \vec{x}{+}\mathbf{1}) = y{+}1$ holds. This ends the proof of (2) and (1). The theorem directly follows from (1). ■

2.3.12 Theorem *For every class of functions $\mathcal{F}$, every partial $\mathcal{F}$-recursive function f, and its approximation function f^+ we have:*

$$f(x_1, \ldots, x_n) \asymp y \to \exists z\, f^+(z, x_1{+}1, \ldots, x_n{+}1) = y{+}1 \ . \tag{1}$$

If $f(\vec{x}) = \tau[f; \vec{x}]$ is a recursive definition from the functions $x + 1$, $x \dot{-} 1$, D, $\mathcal{F}$, if the definition is regular in $m(\vec{x}) < \omega$, if τ contains at most the constant 0, and if $f^+ = (\lambda\vec{x}.\tau)^+$ then we have

$$f^+(m(\vec{x}){+}1, x_1{+}1, \ldots, x_n{+}1) = f(x_1, \ldots, x_n) + 1 \ . \tag{2}$$

Proof. Let $\mathcal{F}$ be enumerated by g_0, g_1, ... We temporarily write $\vec{x}{+}\mathbf{1}$ as an abbreviation for $x_1{+}1, \ldots, x_n{+}1$ and prove the first part of the theorem by a triple induction.

The outer induction is on the structure of $R^{\mathcal{F}}$-terms τ:

> if $\lambda\vec{x}.\tau$ is an $R^{\mathcal{F}}$-function symbol then
> $$\forall\vec{x}\forall y(\,(\lambda\vec{x}.\tau)(\vec{x}) \asymp y \to \exists z\,(\lambda\vec{x}.\tau)^{+}(z, \vec{x}{+}\mathbf{1}) = y{+}1\,)\ . \tag{3}$$

Let $f = \lambda\vec{x}.\tau$ and $f^{+} = (\lambda\vec{x}.\tau)^{+}$ where f is n-ary. Clearly, $f(\vec{x}) = \tau[f;\vec{x}]$ is an explicit or recursive definition from $x+1$, $x \dot{-} 1$, D, from the functions in $\mathcal{F}$, and from the partial functions denoted by $R^{\mathcal{F}}$-function symbols applied in τ. 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:

> for any $\vec{x}$ and y if $f_i(\vec{x}) \asymp y$ then $f^{+}(\max(i,z), \vec{x}{+}\mathbf{1}) = y{+}1$ for some z which is 0 if there are no $R^{\mathcal{F}}$-function symbols applied in τ.

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 $\rho[f;\vec{x}]$ of τ:

> for any y if $\rho[f_i;\vec{x}] \asymp y$ then $\rho^{+}[f^{+};\max(i,z),\vec{x}{+}\mathbf{1}] = y{+}1$ for some z which is 0 if there are no $R^{\mathcal{F}}$-function symbols applied in ρ.

In the following we tacitly use the 'monotone' theorem Thm. 2.3.11. 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 we have $x_i = y$ and we take $z = 0$ because we have $(x_i)^{+}[f^{+};\max(i,0),\vec{x}{+}\mathbf{1}] = x_i{+}1 = y{+}1$.

If $\rho \equiv 0$ then we have $0 = y$ and it suffices to take $z = 0$ because we have $0^{+}[f^{+};\max(i,0),\vec{x}{+}\mathbf{1}] = 1 = y{+}1$.

If $\rho \equiv (\rho_1 + 1)$ then there is a y_1 such that $y_1 + 1 = y$ and $\rho_1[f_i;\vec{x}] \asymp y_1$. We obtain from the inner IH a z as required and such that we have $(\rho_1)^{+}[f^{+};\max(i,z),\vec{x}{+}\mathbf{1}] = y_1{+}1$. We then have

$$\begin{aligned}(\rho_1 + 1)^{+}[f^{+};\max(i,z),\vec{x}{+}\mathbf{1}] = S^{+}(\max(i,z),(\rho_1)^{+}[f^{+};z,\vec{x}{+}\mathbf{1}]) &\overset{\text{inner IH}}{=} \\ S^{+}(\max(i,z),y_1{+}1) = y_1{+}1{+}1 = y{+}1\ .&\end{aligned}$$

If $\rho \equiv (\rho_1 \dot{-} 1)$ then there is a y_1 such that $y_1 \dot{-} 1 = y$ and $\rho_1[f_i;\vec{x}] \asymp y_1$. We obtain from the inner IH a z as required and such that we have $(\rho_1)^{+}[f^{+};\max(i,z),\vec{x}{+}\mathbf{1}] = y_1{+}1$. We then have

$$\begin{aligned}(\rho_1 \dot{-} 1)^{+}[\max(i,z),\vec{x}{+}\mathbf{1}] = Pr^{+}(\max(i,z),(\rho_1)^{+}[f^{+};\max(i,z),\vec{x}{+}\mathbf{1}]) &\overset{\text{inner IH}}{=} \\ Pr^{+}(\max(i,z),y_1{+}1) = (y_1 \dot{-} 1){+}1 = y{+}1\ .&\end{aligned}$$

If $\rho \equiv D(\rho_1,\rho_2,\rho_3)$ then there is a y_1 such that $\rho_1[f_i;\vec{x}] \asymp y_1$ and we obtain from inner IH a z_1 as required and such that $(\rho_1)^{+}[f^{+};\max(i,z_1),\vec{x}{+}\mathbf{1}] = y_1{+}1$. We consider two subcases. If $y_1 > 0$ we then have $\rho_2[f_i;\vec{x}] \asymp y$ and we obtain from inner IH a z_2 as required and such that $(\rho_2)^{+}[f^{+};\max(i,z_1),\vec{x}{+}\mathbf{1}] = y{+}1$.

We set $z = \max(z_1, z_2)$. Note that if ρ does not apply any $R^{\mathcal{F}}$-function symbols then $z = 0$. We have

$$\begin{aligned}
&(D(\rho_1,\rho_2,\rho_3))^+[f^+;\max(i,z),\vec{x}+\mathbf{1}] = \\
&\quad D^+(\max(i,z),(\rho_1)^+[f^+;\max(i,z),\vec{x}+\mathbf{1}],(\rho_2)^+[f^+;\max(i,z),\vec{x}+\mathbf{1}], \\
&\qquad\qquad (\rho_3)^+[f^+;\max(i,z),\vec{x}+\mathbf{1}]) \overset{\text{inner IH}}{=} \\
&\quad D^+(\max(i,z),y_1{+}1,y{+}1,(\rho_3)^+[f^+;\max(i,z),\vec{x}+\mathbf{1}]) = \\
&\quad D(y_1,y{+}1,(\rho_3)^+[f^+;\max(i,z),\vec{x}+\mathbf{1}]) = y{+}1 \ .
\end{aligned}$$

The subcase $y_1 = 0$ is similar.

If $\rho \equiv h(\vec{\rho})$ where h is an m-ary function symbol such that $h \equiv g_j$, or $h \equiv \lambda\vec{x}.\tau_1$, or $h \equiv f$ (in the last case we have $m = n$) then there are numbers $y_1, \ldots, y_m$ such that for all $1 \le k \le m$ we have $\rho_k[f_i;\vec{x}] \asymp y_k$ and we obtain from inner IH a z_k as required and such that ${\rho_k}^+[f^+;\max(i,z_k),\vec{x}+\mathbf{1}] = y_k{+}1$. If $h \equiv g_j$ then we have $g_j(\vec{y}) = y$, we set $z = \max(\vec{z})$, and obtain

$$\begin{aligned}
&(g_j(\vec{\rho}))^+[f^+;\max(i,z),\vec{x}+\mathbf{1}] = \\
&\quad (g_j)^+(z,{\rho_1}^+[f^+;\max(i,z),\vec{x}+\mathbf{1}],\ldots,{\rho_m}^+[f^+;\max(i,z),\vec{x}+\mathbf{1}]) \overset{\text{inner IH's}}{=} \\
&\quad (g_j)^+(\max(i,z),\vec{y}+\mathbf{1}) = g_j(\vec{y}){+}1 = y{+}1 \ .
\end{aligned}$$

If $h \equiv \lambda\vec{x}.\tau_1$ then we have $(\lambda\vec{x}.\tau_1)(\vec{y}) \asymp y$, we obtain a number z_0 as required and such that $(\lambda\vec{x}.\tau_1)^+(z_0,\vec{y}+\mathbf{1}) = y{+}1$ from the outer IH and we set $z = \max(z_0,\vec{z})$. Note that this is the only case when we have $z > 0$ from $z_0 > 0$ regardless whether or not $\vec{\rho}$ applies $R^{\mathcal{F}}$-function symbols. We then have

$$\begin{aligned}
&((\lambda\vec{x}.\tau_1)(\vec{\rho}))^+[f^+;\max(i,z),\vec{x}+\mathbf{1}] = \\
&\qquad (\lambda\vec{x}.\tau_1)^+(\max(i,z),{\rho_1}^+[f^+;\max(i,z),\vec{x}+\mathbf{1}],\ldots, \\
&\qquad\qquad {\rho_m}^+[f^+;\max(i,z),\vec{x}+\mathbf{1}]) \overset{\text{inner IH's}}{=} \\
&\qquad (\lambda\vec{x}.\tau_1)^+(\max(i,z),\vec{y}+\mathbf{1}) \overset{\text{outer IH}}{=} y{+}1 \ .
\end{aligned}$$

If $h \equiv f$ then we have $f_i(\vec{y}) \asymp y$, we obtain from middle IH a z_0 as required and such that $f^+(\max(i,z_0),\vec{y}+\mathbf{1}) = y{+}1$, we set $z = \max(z_0,\vec{z})$, and we get

$$\begin{aligned}
&(f(\vec{\rho}))^+[f^+;\max(i,z),\vec{x}+\mathbf{1}] = \\
&\quad f^+(\max(i,z),{\rho_1}^+[f^+;\max(i,z),\vec{x}+\mathbf{1}],\ldots,{\rho_n}^+[f^+;\max(i,z),\vec{x}+\mathbf{1}]) \overset{\text{inner IH's}}{=} \\
&\quad f^+(\max(i,z),\vec{y}+\mathbf{1}) \overset{\text{middle IH}}{=} y{+}1 \ .
\end{aligned}$$

This ends the proof of the inner induction formula.

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 obtain a z_0 such that $\tau^+[f^+;\max(i,z_0),\vec{x}+\mathbf{1}] = y{+}1$ holds by taking $\rho \equiv \tau$ in the inner induction formula. If there are no applications of $R^{\mathcal{F}}$-function symbols in τ we have $z_0 = 0$ and we set $z = 0$. We set $z = z_0 + 1$ otherwise. In either case we have $\max(i+1, z) = \max(i, z_0) + 1$ and from the definition of f^+ we can see that we have $f^+(\max(i{+}1, z), \vec{x}{+}\mathbf{1}) = y{+}1$.

We can now finish the proof of the outer induction (3). We take any $\vec{x}$ and y such that $f(\vec{x}) \asymp y$ holds. We thus obtain $\tau[f;\vec{x}] \asymp y$. If the definition of f is explicit then, since $f_1(\vec{x}) \simeq \tau[f;\vec{x}]$, we have $f_1(\vec{x}) \asymp y$ and we set $i = 1$. If the definition of f is recursive then by the Recursion theorem 2.1.12 we have $f = \bigcup_i f_i$ and we obtain $\tau[f_{i_0};\vec{x}] \asymp y$ for some i_0 from 2.1.9(1). We set $i = i_0 + 1$ and we get $f_i(\vec{x}) \asymp y$. In any case we obtain a z_0 such that $f^+(\max(i, z_0), \vec{x}{+}\mathbf{1}) = y{+}1$ from the middle induction formula and $z = \max(i, z_0)$ proves the consequent of (3). The first part of the theorem is a direct consequence of (3).

In the second part of the theorem we have a recursive definition $f(\vec{x}) = \tau[f;\vec{x}]$ satisfying the assumptions. There cannot be any applications of $R^{\mathcal{F}}$-function symbols in τ and τ must be an $R^{\mathcal{F}}$-term. We thus have $f = \lambda\vec{x}.\tau$. We prove by induction on i:

$$\forall\vec{x}(\, m(\vec{x}) < i \to f_i(\vec{x}) \asymp f(\vec{x})\,) \;.$$

In the base case there is nothing to prove. In the inductive case we take any $\vec{x}$ and assume $m(\vec{x}) < i + 1$, i.e. $m(\vec{x}) \le i$. We take any recursive application $f(\vec{\rho}[f;\vec{x}])$ in τ and assume its governing condition $\phi[f;\vec{x}]$. By the regularity we have $m(\vec{\rho}[f;\vec{x}]) < m(\vec{x})$ and by IH we obtain $f_i(\vec{\rho}[f;\vec{x}]) \asymp f(\vec{\rho}[f;\vec{x}])$. We thus have

$$f_{i+1}(\vec{x}) \simeq \tau[f_i;\vec{x}] \overset{2.1.15}{\simeq} \tau[f;\vec{x}] \simeq f(\vec{x}) \;.$$

Substituting $i := m(\vec{x}){+}1$ in the just proved property we get: $f_{m(\vec{x})+1}(\vec{x}) \asymp f(\vec{x})$ and from the middle induction formula in the first part of the theorem we get $z = 0$ and so we have $f^+(m(\vec{x}){+}1, \vec{x}{+}\mathbf{1}) = f(\vec{x}) + 1$. Careful reader has noted that although the middle induction formula was proved under the inductive hypothesis of the outer induction formula the assumption was not used for this particular τ since it does not apply any $R^{\mathcal{F}}$-function symbols. ■

2.3.13 Functions estimating the length of reductions. It only remains to connect in Thm. 2.3.14 the non-zero values of approximating functions to terminating reductions. For a given $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.\tau$ and a number z we will bound there by $d_{\lambda\vec{x}.\tau}(z)$ the number of reductions reducing the closed $R^{\mathcal{F}}$-term $(\lambda\vec{x}.\tau)(\underline{\vec{x}})$ into a monadic numeral provided $(\lambda\vec{x}.\tau)^+(z, x_1{+}1, \ldots, x_n{+}1) > 0$ holds. The length of reductions will be independent of the size of numbers $\vec{x}$; only the depth of recursion z will count. The theorem is formulated in a form stronger than needed in this section because we will need the bounds on the length of reduction sequences in Thm. 4.3.5.

Let a class of functions $\mathcal{F}$ be enumerated by g_0, g_1, $\ldots$. For every $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.\tau$ we define the estimating function $d_{\lambda\vec{x}.\tau}(z)$ 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:

$$\begin{aligned}
\delta_{x_i} &\equiv 0 \\
\delta_0 &\equiv 0 \\
\delta_{(\rho+1)} &\equiv \delta_\rho \\
\delta_{(\rho \dot{-} 1)} &\equiv \delta_\rho + 1 \\
\delta_{D(\rho_1,\rho_2,\rho_3)} &\equiv \delta_{\rho_1} + \delta_{\rho_2} + \delta_{\rho_3} + 1 \\
\delta_{g_i(\rho_1,\dots,\rho_n)} &\equiv \delta_{\rho_1} + \cdots + \delta_{\rho_n} + 1 \\
\delta_{(\lambda\vec{x}.\tau)(\rho_1,\dots,\rho_n)} &\equiv \delta_{\rho_1} + \cdots + \delta_{\rho_n} + d_{\lambda\vec{x}.\tau}(z) \\
\delta_{f(\rho_1,\dots,\rho_n)} &\equiv \delta_{\rho_1} + \cdots + \delta_{\rho_n} + f(z) \ .
\end{aligned}$$

We now define the unary estimating functions $d_{\lambda\vec{x}.\tau}$ for all n-ary partial $\mathcal{F}$-recursive functions $\lambda\vec{x}.\tau$ by induction on the construction of $R^{\mathcal{F}}$-terms $\tau[f;\vec{x}]$. The estimating functions for all $R^{\mathcal{F}}$-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 a clausal definition regular in its argument:

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

2.3.14 Theorem (Depth of recursion vs. length of reductions) *For every class of functions $\mathcal{F}$, every partial $\mathcal{F}$-recursive function $f = \lambda\vec{x}.\tau$, and every approximation function $f^+ = (\lambda\vec{x}.\tau)^+$ we have*

$$f^+(z, x_1+1, \dots, x_n+1) = y+1 \to (\lambda\vec{x}.\tau)(\underline{\vec{x}}) \blacktriangleright^{\mathcal{F}}_{d_{\lambda\vec{x}.\tau}(z)} \underline{y} \ . \tag{1}$$

Proof. Let $\mathcal{F}$ be enumerated by g_0, g_1, ... We temporarily write $\vec{x}\mathbf{+1}$ as an abbreviation for $x_1{+}1$, ... , $x_n{+}1$ and prove a stronger property:

> *for all subterms $\rho[f;\vec{x}]$ of $\tau[f;\vec{x}]$ where $\lambda\vec{x}.\tau$ is an n-ary $R^{\mathcal{F}}$-function symbol we have*
>
> $$\forall\vec{x}\forall y(\, \rho^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}\mathbf{+1}] = y+1 \to \rho[\lambda\vec{x}.\tau;\underline{\vec{x}}] \blacktriangleright^{\mathcal{F}}_{\delta_\rho[d_{\lambda\vec{x}.\tau};z]} \underline{y}\,)$$

by induction on $z{\cdot}||\tau|| + ||\rho||$. Thus we take any z, any $R^{\mathcal{F}}$-term $\tau[f;\vec{x}]$ such that $\lambda\vec{x}.\tau$ is an n-ary $R^{\mathcal{F}}$-function symbol, any subterm $\rho[f;\vec{x}]$ of τ, any $\vec{x}$, any y, assume $\rho^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}\mathbf{+1}] = y+1$ and continue by the case analysis of the term ρ.

If $\rho \equiv x_i$ we have

$$y+1 = {x_i}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}\mathbf{+1}] \equiv x_i[(\lambda\vec{x}.\tau)^+; z, \vec{x}\mathbf{+1}] = x_i + 1$$

and so $y = x_i$. We also have $x_i[\lambda\vec{x}.\tau;\underline{\vec{x}}] \equiv \underline{x_i} \blacktriangleright_0^{\mathcal{F}} \underline{x_i} \equiv \underline{y}$ where $0 = \delta_\rho[d_{\lambda\vec{x}.\tau};z]$.

If $\rho \equiv 0$ we have

$$y+1 = 0^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}] \equiv (0+1)[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}] = 1$$

and so $y = 0$. We also have $0[\lambda\vec{x}.\tau;\underline{\vec{x}}] \equiv 0 \blacktriangleright_0^{\mathcal{F}} 0 \equiv \underline{y}$ where $0 = \delta_\rho[d_{\lambda\vec{x}.\tau};z]$.

If $\rho \equiv (\rho_1+1)$ we have

$$y+1 = (\rho_1+1)^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}] \equiv S^+(z,{\rho_1}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}])\ .$$

From the definition of S^+ we can see that there is a number y_1 such that ${\rho_1}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}] = y_1+1$ and from $y+1 = S^+(z,y_1+1) = (y_1+1)+1$ we obtain $y = y_1+1$. The (minor) IH applies to the subterm ρ_1 of τ and we get:

$$(\rho_1+1)[\lambda\vec{x}.\tau;\underline{\vec{x}}] \equiv \rho_1[\lambda\vec{x}.\tau;\underline{\vec{x}}]+1 \overset{\text{min. IH}}{\blacktriangleright^{\mathcal{F}}_{\delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z]}} \underline{y_1}+1 \equiv \underline{y}\ .$$

Thus the total number of reductions does not exceed $\delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z] = \delta_\rho[d_{\lambda\vec{x}.\tau};z]$.

If $\rho \equiv (\rho_1 \dot{-} 1)$ we have

$$y+1 = (\rho_1\dot{-}1)^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}] \equiv Pr^+(z,{\rho_1}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}])\ .$$

From the definition of Pr^+ we can see that there is a number y_1 such that ${\rho_1}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}] = y_1+1$ and from $y+1 = Pr^+(z,y_1+1) = (y_1\dot{-}1)+1$ we obtain $y = y_1\dot{-}1$. The (minor) IH applies to the subterm ρ_1 of τ and we get:

$$(\rho_1\dot{-}1)[\lambda\vec{x}.\tau;\underline{\vec{x}}] \equiv \rho_1[\lambda\vec{x}.\tau;\underline{\vec{x}}]\dot{-}1 \overset{\text{min. IH}}{\blacktriangleright^{\mathcal{F}}_{\delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z]}} \underline{y_1}\dot{-}1 \blacktriangleright_1^{\mathcal{F}} \underline{y_1\dot{-}1} \equiv \underline{y}\ .$$

Thus the total number of reductions does not exceed $\delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z]+1 = \delta_\rho[d_{\lambda\vec{x}.\tau};z]$.

If $\rho \equiv D(\rho_1,\rho_2,\rho_3)$ we have

$$\begin{aligned}y+1 &= (D(\rho_1,\rho_2,\rho_3))^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}] \equiv \\ &\quad D^+(z,{\rho_1}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}],{\rho_2}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}],{\rho_3}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}])\ .\end{aligned}$$

From the definition of D^+ we can see that there is a number y_1 such that ${\rho_1}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}] = y_1+1$ and we have

$$\begin{aligned}y+1 &= D^+(z,y_1+1,{\rho_2}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}],{\rho_3}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}]) = \\ &\quad D(y_1,{\rho_2}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}],{\rho_3}^+[(\lambda\vec{x}.\tau)^+;z,\vec{x}+\mathbf{1}])\ .\end{aligned}$$

The (minor) IH applies to the subterm ρ_1 of τ and we consider two subcases. If $y_1 > 0$ then the (minor) IH applies to the subterm ρ_2 of τ and we get:

$$\begin{aligned}(D(\rho_1,\rho_2,\rho_3))[\lambda\vec{x}.\tau;\underline{\vec{x}}] &\equiv D(\rho_1[\lambda\vec{x}.\tau;\underline{\vec{x}}],\rho_2[\lambda\vec{x}.\tau;\underline{\vec{x}}],\rho_3[\lambda\vec{x}.\tau;\underline{\vec{x}}]) \overset{\text{min. IH}}{\blacktriangleright^{\mathcal{F}}_{\delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z]}} \\ &\quad D(\underline{y_1},\rho_2[\lambda\vec{x}.\tau;\underline{\vec{x}}],\rho_3[\lambda\vec{x}.\tau;\underline{\vec{x}}]) \blacktriangleright_1^{\mathcal{F}} \rho_2[\lambda\vec{x}.\tau;\underline{\vec{x}}] \overset{\text{min. IH}}{\blacktriangleright^{\mathcal{F}}_{\delta_{\rho_2}[d_{\lambda\vec{x}.\tau};z]}} \underline{y}\ .\end{aligned}$$

The total number of reductions thus does not exceed:

$$\delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z] + 1 + \delta_{\rho_2}[d_{\lambda\vec{x}.\tau};z] \leq \delta_\rho[d_{\lambda\vec{x}.\tau};z] \ .$$

The subcase $y_1 = 0$ is similar.

If $\rho \equiv g_i(\rho_1, \ldots, \rho_m)$ we have

$$\begin{aligned} y + 1 = (g_i(\rho_1, \ldots, \rho_m))^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}] \equiv \\ (g_i)^+(z, {\rho_1}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}], \ldots, {\rho_m}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}]) \ . \end{aligned}$$

From the definition of $(g_i)^+$ we can see that there are numbers $y_1, \ldots, y_m$ such that ${\rho_j}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}] = y_j + 1$ holds for all $1 \leq j \leq m$ and from $y+1 = (g_i)^+(\vec{y}{+}\mathbf{1}) = g_i(\vec{y}) + 1$ we obtain $y = g_i(\vec{y})$. The (minor) IH's apply to the subterms $\rho_1, \ldots, \rho_m$ of τ and we obtain:

$$\begin{aligned} (g_i(\rho_1, \ldots, \rho_m))[\lambda\vec{x}.\tau; \underline{\vec{x}}] \equiv g_i(\rho_1[\lambda\vec{x}.\tau; \underline{\vec{x}}], \ldots, \rho_m[\lambda\vec{x}.\tau; \underline{\vec{x}}]) \overset{\text{min. IH's}}{\blacktriangleright^{\mathcal{F}}_{\sum_j \delta_{\rho_j}[d_{\lambda\vec{x}.\tau};z]}} \\ g_i(\underline{\vec{y}}) \blacktriangleright^{\mathcal{F}}_1 \underline{g_i(\vec{y})} \equiv \underline{y} \ . \end{aligned}$$

The total number of reductions thus does not exceed:

$$\delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z] + \cdots + \delta_{\rho_m}[d_{\lambda\vec{x}.\tau};z] + 1 = \delta_\rho[d_{\lambda\vec{x}.\tau};z] \ .$$

If $\rho \equiv (\lambda\vec{x}.\tau_1)(\rho_1, \ldots, \rho_m)$ we have

$$\begin{aligned} y + 1 = ((\lambda\vec{x}.\tau_1)(\rho_1, \ldots, \rho_m))^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}] \equiv \\ (\lambda\vec{x}.\tau_1)^+(z, {\rho_1}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}], \ldots, {\rho_m}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}]) \ . \end{aligned}$$

From the definition of $(\lambda\vec{x}.\tau_1)^+$ we can see that there are numbers $y_1, \ldots, y_m$ such that ${\rho_i}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}] = y_i + 1$ holds for all $1 \leq i \leq m$ and so the (minor) IH's apply to the subterms $\rho_1, \ldots, \rho_m$ of τ. We can also see that $z > 0$ and that we have

$$y + 1 = (\lambda\vec{x}.\tau_1)^+(\vec{y}{+}\mathbf{1}) = {\tau_1}^+[(\lambda\vec{x}.\tau_1)^+; z{-}1, \vec{y}{+}\mathbf{1}] \ .$$

Now, the (major) IH applies to the subterm τ_1 of τ_1 because we have $z{\cdot}||\tau|| + ||\rho|| > (z{-}1){\cdot}||\tau_1|| + ||\tau_1||$ and we get:

$$\begin{aligned} (\lambda\vec{x}.\tau_1)(\rho_1, \ldots, \rho_m)[\lambda\vec{x}.\tau; \underline{\vec{x}}] \equiv (\lambda\vec{x}.\tau_1)(\rho_1[\lambda\vec{x}.\tau; \underline{\vec{x}}], \ldots, \rho_m[\lambda\vec{x}.\tau; \underline{\vec{x}}]) \\ \overset{\text{min. IH's}}{\blacktriangleright^{\mathcal{F}}_{\sum_i \delta_{\rho_i}[d_{\lambda\vec{x}.\tau};z]}} (\lambda\vec{x}.\tau_1)(\underline{\vec{y}}) \blacktriangleright^{\mathcal{F}}_1 \tau_1[\lambda\vec{x}.\tau_1; \underline{\vec{y}}] \overset{\text{maj. IH}}{\blacktriangleright^{\mathcal{F}}_{\delta_{\tau_1}[d_{\lambda\vec{x}.\tau_1};z-1]}} \underline{y} \ . \end{aligned}$$

The total number of reductions thus does not exceed:

$$\begin{aligned} \delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z] + \cdots + \delta_{\rho_m}[d_{\lambda\vec{x}.\tau};z] + 1 + \delta_{\tau_1}[d_{\lambda\vec{x}.\tau_1};z{-}1] = \\ \delta_{\rho_1}[d_{\lambda\vec{x}.\tau};z] + \cdots + \delta_{\rho_m}[d_{\lambda\vec{x}.\tau};z] + d_{\lambda\vec{x}.\tau_1}(z) = \delta_\rho[d_{\lambda\vec{x}.\tau};z] \ . \end{aligned}$$

If $\rho \equiv f(\rho_1, \ldots, \rho_n)$ we have

$$y + 1 = (f(\rho_1, \ldots, \rho_n))^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}] \equiv \\ (\lambda\vec{x}.\tau)^+(z, {\rho_1}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}], \ldots, {\rho_n}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}]) \ .$$

From the definition of $(\lambda\vec{x}.\tau)^+$ we can see that there are numbers $y_1, \ldots, y_n$ such that ${\rho_i}^+[(\lambda\vec{x}.\tau)^+; z, \vec{x}{+}\mathbf{1}] = y_i + 1$ holds for all $1 \leq i \leq n$ and so the (minor) IH's apply to the subterms $\rho_1, \ldots, \rho_n$ of τ. We can also see that $z > 0$ and that we have

$$y + 1 = (\lambda\vec{x}.\tau_1)^+(\vec{y}{+}\mathbf{1}) = {\tau_1}^+[(\lambda\vec{x}.\tau_1)^+; z{-}1, \vec{y}{+}\mathbf{1}] \ .$$

Now, the (major) IH applies to the subterm τ of τ because we have $z{\cdot}||\tau||{+}||\rho|| > (z{-}1){\cdot}||\tau|| + ||\tau||$. Hence, we obtain

$$(f(\rho_1, \ldots, \rho_n))[\lambda\vec{x}.\tau; \underline{\vec{x}}] \equiv (\lambda\vec{x}.\tau)(\rho_1[\lambda\vec{x}.\tau; \underline{\vec{x}}], \ldots, \rho_n[\lambda\vec{x}.\tau; \underline{\vec{x}}]) \\ \overset{\text{min. IH's}}{\blacktriangleright^{\mathcal{F}}_{\sum_i \delta_{\rho_i}[d_{\lambda\vec{x}.\tau}; z]}} (\lambda\vec{x}.\tau)(\underline{\vec{y}}) \blacktriangleright^{\mathcal{F}}_1 \tau[\lambda\vec{x}.\tau; \underline{\vec{y}}] \overset{\text{maj. IH}}{\blacktriangleright^{\mathcal{F}}_{\delta_\tau[d_{\lambda\vec{x}.\tau}; z-1]}} \underline{y} \ .$$

The total number of reductions thus does not exceed:

$$\delta_{\rho_1}[d_{\lambda\vec{x}.\tau}; z] + \cdots + \delta_{\rho_n}[d_{\lambda\vec{x}.\tau}; z] + 1 + \delta_\tau[d_{\lambda\vec{x}.\tau}; z{-}1] = \\ \delta_{\rho_1}[d_{\lambda\vec{x}.\tau}; z] + \cdots + \delta_{\rho_n}[d_{\lambda\vec{x}.\tau}; z] + d_{\lambda\vec{x}.\tau}(z) = \delta_\rho[d_{\lambda\vec{x}.\tau}; z] \ .$$

With the general property proved, we prove (1) by taking any n-ary partial $\mathcal{F}$-recursive function $f = \lambda\vec{x}.\tau$, its approximation function $f^+ = (\lambda\vec{x}.\tau)^+$, any numbers $\vec{x}$, and by assuming $f^+(z, \vec{x} + 1) = y + 1$. From the definition of f^+ we can see that $z > 0$ and so we have

$$y + 1 = f^+(z, \vec{x}{+}\mathbf{1}) = \tau^+[f^+; z{-}1, \vec{x}{+}\mathbf{1}] \ .$$

The general property applies to the last term and we obtain:

$$(\lambda\vec{x}.\tau)(\underline{\vec{x}}) \blacktriangleright^{\mathcal{F}}_1 \tau[\lambda\vec{x}.\tau; \underline{\vec{x}}] \overset{\text{prop.}}{\blacktriangleright^{\mathcal{F}}_{\delta_\tau[d_{\lambda\vec{x}.\tau}; z-1]}} \underline{y}$$

where we have $d_{\lambda\vec{x}.\tau}(z) = \delta_\tau[d_{\lambda\vec{x}.\tau}; z{-}1] + 1$. ■

2.3.15 Corollary (Coincidence of the denotational and operational semantics) *For every class of functions $\mathcal{F}$, every partial $\mathcal{F}$-recursive function $f = \lambda\vec{x}.\tau$, every approximation function $f^+ = (\lambda\vec{x}.\tau)^+$, and all numbers x_1, $\ldots$, x_n, and y, the following are equivalent:*

1. $f(x_1, \ldots, x_n) \asymp y$
2. $\exists z\, f^+(z, x_1{+}1, \ldots, x_n{+}1) = y{+}1$
3. $(\lambda\vec{x}.\tau)(\underline{x_1}, \ldots, \underline{x_n}) \blacktriangleright^{\mathcal{F}} \underline{y}$.

Consequently, if f is total then we have:

$$(\lambda\vec{x}.\tau)(\underline{\vec{x}}) \blacktriangleright^{\mathcal{F}} \underline{f(\vec{x})} \ .$$

Proof. From (1) we get (2) by Thm. 2.3.12, from (2) we get (3) by Thm. 2.3.14, and from (3) we obtain (1) by Thm. 2.3.9. The second part of the theorem follows directly from (1) $\rightarrow$ (3). $\blacksquare$

Chapter 3

Inductively Defined Function Classes

3.1 Inductively Defined Function Classes

3.1.1 Classes of partial functions. We will denote by $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{I}$ (possibly with subscripts) denumerable *classes*, i.e. sets, of partial functions over $\mathbb{N}$.

For a given binary partial function $\{e\}(x)$ and a number e we explicitly define the unary partial function g_e as $g_e(x) \simeq \{e\}(x)$. We say that the number e is an *index* of an n-ary partial function f if $g_e = \langle f \rangle$ holds. Note that we then have

$$\forall \vec{x}\, f(\vec{x}) \simeq \{e\}(\vec{x}) \ .$$

The reader will note that the commas in $f(\vec{x})$ separate the n arguments whereas the commas in $\vec{x}$ of $\{e\}(\vec{x})$ denote the pairing function.

The partial function $\{e\}(x)$ *enumerates* the non-empty class of partial functions $\mathcal{F}$ if every $f \in \mathcal{F}$ has an index and every number is an index of some $f \in \mathcal{F}$. This means that the contractions of partial functions in $\mathcal{F}$ are exactly the partial functions g_e:

$$\{\langle f \rangle \mid f \in \mathcal{F}\} = \{g_e \mid e \in \mathbb{N}\} \ .$$

For the sake of completeness we say that the function $g(i, x) = 0$ *enumerates* the empty class $\mathcal{F} = \emptyset$.

3.1.2 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 $n > 0$ 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.

Most function operators we will be using will take and yield (total) functions. We admit partial functions only in the operators of composition (see Par.

3.1.3), minimalization (see Par. 5.2.1), and in the operators of explicit/recursive definitions (see Par. 3.1.4).

3.1.3 Operators of composition. For every pair of numbers $m > 0$ and $n > 0$ the operator of *composition* takes an m-ary partial function g and m n-ary partial functions $h_1, \ldots, h_m$ and yields an n-ary partial function f satisfying:

$$f(\vec{x}) \simeq g(h_1(\vec{x}), h_2(\vec{x}), \ldots, h_m(\vec{x})) \ . \tag{1}$$

If the argument functions are total so is the function yielded by the composition operators.

We will need two special composition operators both of which take two (total) unary functions and yield a (total) unary function. Composition with $m = n = 1$ is called *unary composition* and yields f from g and h such that

$$f(x) = g\,h(x) \ . \tag{2}$$

Composition with $m = 2$, $n = 1$ and with the first argument g fixed to the pairing function (x, y) is the operator of *pairing* which yields f from g and h such that

$$f(x) = g(x), h(x) \ . \tag{3}$$

3.1.4 Explicit and recursive definitions as function operators. Every explicit or recursive definition $f(\vec{x}) \simeq \tau[f; \vec{x}]$ can be viewed as a function operator taking all partial functions applied in τ other than f as arguments and yielding the partial function f.

3.1.5 Inductively defined classes of functions. For an n-ary function operator $\mathcal{O}$ and a class of partial functions $\mathcal{F}$ we say that $\mathcal{F}$ *is closed under* $\mathcal{O}$ if for any 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 $\mathcal{F}$.

For a class of (total) functions $\mathcal{I}$ and for function operators $\mathcal{O}_1$, $\mathcal{O}_2$, ... we say that the class $\mathcal{F}$ of functions is (inductively) *generated from* $\mathcal{I}$ *by* $\mathcal{O}_1$, $\mathcal{O}_2$, ... if the class $\mathcal{F}$ is the smallest class containing the functions of $\mathcal{I}$ and closed under the function operators. The functions in $\mathcal{I}$ are called the *initial functions* of the class $\mathcal{F}$.

We say that the predicate R is an $\mathcal{F}$*-predicate* if its characteristic function (which is a total function) is in $\mathcal{F}$, i.e. if we have $R_* \in \mathcal{F}$.

3.1.6 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 $\mathcal{O}_i$ will be often used in the following to show inclusions $\mathcal{F} \subseteq \mathcal{F}_1$ where it is clearly sufficient to show $\mathcal{I} \subseteq \mathcal{F}_1$ and the closure of $\mathcal{F}_1$ under the operators $\mathcal{O}_i$.

The minimality of $\mathcal{F}$ also means that we have $f \in \mathcal{F}$ iff there is a finite sequence of functions $f_1, \ldots, f_k$ with $f_k = f$ and such that for every $1 \leq i < k$

we have either $f_{i+1} \in \mathcal{I}$ or else the function f_{i+1} is obtained by a function operator $\mathcal{O}_i$ with arguments from among the functions $f_1, \ldots, f_i$.

The minimality of $\mathcal{F}$ can be also used 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 $f = \mathcal{O}_i(f_1, \ldots, f_n)$ then we have to show $\mathcal{P}(f)$ under the inductive hypotheses that the functions $f_1, \ldots, f_n$ posses the property.

3.1.7 Mathematical versus meta-mathematical definitions of recursive functions. We have defined in Par. 2.3.4 the class of partial $\mathcal{F}$-recursive functions by means of a language of R-function symbols. 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 definitions of classes of functions.

The following theorem characterizes partial and total $\mathcal{F}$-recursive functions by inductive definitions. The characterization has still a meta-mathematical flavor because it uses the language based function operators of explicit and recursive definitions. We will characterize the same classes in Thm. 5.2.16 by completely mathematical means of μ-recursion.

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 **innefficient** 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.

3.1.8 Theorem (Inductive characterization of recursive functions) *For any class of functions $\mathcal{F}$ the class of partial $\mathcal{F}$-recursive functions is generated from the case analysis function D, the successor function $x+1$, the predecessor function $x \dot{-} 1$, and from the functions in $\mathcal{F}$ by explicit and recursive definitions.*

$\mathcal{F}$-recursive functions are generated from the same initial functions by explicit and recursive definitions of (total) functions.

Proof. Let $\mathcal{F}$ be enumerated by $g_0, g_1, \ldots$ and let us denote by $\mathcal{G}$ the partial functions generated from the above initial functions by explicit and recursive definitions. We first prove by induction on construction of partial functions $f \in \mathcal{G}$ that f is a partial $\mathcal{F}$-recursive function. If f is an initial function then it is $\mathcal{F}$-recursive by the results of Par. 2.3.5. If f is defined by an explicit or recursive definition $f(\vec{x}) \simeq \tau[f;\vec{x}]$ where $h_1, \ldots, h_k$ are all partial functions applied in τ other than f, $x+1$, $x \dot{-} 1$, D, and the functions $g_i \in \mathcal{F}$ then the functions $h_1, \ldots, h_k$ are partial $\mathcal{F}$-recursive functions by IH's. We obtain an $R^{\mathcal{F}}$-term τ_1 from τ by replacing in it all functions symbols $h_1, \ldots, h_k$ by the corresponding $R^{\mathcal{F}}$-function symbols denoting these functions and by replacing every constant c by the monadic numeral $S^c(0)$. It should be obvious that the functional equation $f(\vec{x}) \simeq \tau_1[f;\vec{x}]$ defines the same partial function f which is $\mathcal{F}$-recursive because it is denoted by the $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.\tau_1$.

The converse inclusion that the partial $\mathcal{F}$-recursive functions are in $\mathcal{G}$ is proved by induction on the construction of $R^{\mathcal{F}}$-terms τ such that $\lambda\vec{x}.\tau$ is a partial $\mathcal{F}$-recursive function. By induction hypotheses all $R^{\mathcal{F}}$-function symbols applied in τ are interpreted by partial functions in $\mathcal{G}$ and the other function symbols (but not f) applied in τ are interpreted by the initial functions of $\mathcal{G}$. Hence $f(\vec{x}) \simeq \tau[f;\vec{x}]$ is an explicit or recursive definition from the partial functions in $\mathcal{G}$ and so $\lambda\vec{x}.\tau = f \in \mathcal{G}$.

For the proof of the second part of theorem let us denote by $\mathcal{G}_1$ the subset of $\mathcal{G}$ generated from the initial functions by explicit and recursive definitions of (total) functions. From the first part of the theorem we know that every $f \in \mathcal{G}_1 \subseteq \mathcal{G}$ is partial $\mathcal{F}$-recursive and, by being total, it is a $\mathcal{F}$-recursive function.

Vice versa, let f be a $\mathcal{F}$-recursive function. Then we have $f = \lambda\vec{x}.\tau$ for some $R^{\mathcal{F}}$-function symbol $\lambda\vec{x}.\tau$. The term τ may apply $R^{\mathcal{F}}$-function symbols denoting partial functions and we wish to eliminate this detour through partial functions. As all approximation functions for partial $\mathcal{F}$-recursive functions are $\mathcal{F}$-recursive and are defined by explicit and recursive definitions of total functions they are in the class $\mathcal{G}_1$. In particular also $f^+ = (\lambda\vec{x}.\tau)^+ \in \mathcal{G}_1$. To every $\vec{x}$ there is the least number, denoted by $k(\vec{x})$, such that for every $z \geq k(\vec{x})$ we have

$$f^+(z, x_1{+}1, \ldots, x_n{+}1) = f(x_1, \ldots, x_n){+}1$$

from Corollary 2.3.15 and Thm. 2.3.11. Consider the following recursive definition:

$$f_1(z,\vec{x}) = f_1(z+1,\vec{x}) \leftarrow x_1 = y_1{+}1 \land \cdots \land x_n = y_n{+}1 \land f^+(z,\vec{x}) = 0$$
$$f_1(z,\vec{x}) = y \leftarrow x_1 = y_1{+}1 \land \cdots \land x_n = y_n{+}1 \land f^+(z,\vec{x}) = y{+}1$$

which we claim to define $f_1 \in \mathcal{G}_1$ because it is regular in $m(z,\vec{x}) = k(\vec{x}) \dot{-} z$. Indeed, when we go to the recursion in the first clause then we must have $z < k(\vec{x})$ and so $m(z+1,\vec{x}) = k(\vec{x}) \dot{-} (z+1) < k(\vec{x}) \dot{-} z = m(z,\vec{x})$. We can thus derive $f \in \mathcal{G}_1$ by explicit definition:

$$f(x_1, \ldots, x_n) = f_1(0, x_1{+}1, \ldots, x_n{+}1)\ . \blacksquare$$

3.1.9 Relatively defined function classes. Suppose that the class $\mathcal{F}$ is generated from the initial functions in $\mathcal{I}$ by the operators $\mathcal{O}_1, \mathcal{O}_2, \ldots$.

For a class of partial functions $\mathcal{G}$ we say that $\mathcal{G}$ *is* $\mathcal{F}$*-closed* if $\mathcal{I} \subseteq \mathcal{G}$ and $\mathcal{G}$ is closed under all operators $\mathcal{O}_i$. Note that we then have $\mathcal{F} \subseteq \mathcal{G}$. For a class of (total) functions $\mathcal{F}_1$ we say that the class $\mathcal{G}$ *is* $\mathcal{F}$ *in* $\mathcal{F}_1$ if $\mathcal{G}$ is the smallest $\mathcal{F}$-closed class containing $\mathcal{F}_1$.

(Partial) $\mathcal{F}$-recursive functions are (partial) recursive in $\mathcal{F}$ in the above sense because by Thm. 3.1.8 (partial) $\mathcal{F}$-recursive functions are generated from D, $x+1$, $x \dot{-} 1$, and $\mathcal{F}$ by explicit and recursive definitions of (partial) functions and so every (partially) recursively closed class containing $\mathcal{F}$ must contain the the (partial) $\mathcal{F}$-recursive functions.

3.2 Explicit Definitions of Functions

The two theorems proved in this section give sufficient and necessary conditions for a class of partial functions to be closed under explicit definitions of partial functions without and with constants.

3.2.1 Theorem (Characterization of explicit definitions without constants) *A class $\mathcal{F}$ is closed under explicit definitions of partial functions without constants iff $\mathcal{F}$ contains the identity functions I_i^n and it is closed under composition.*

Proof. We prove first the implication ($\rightarrow$). If $\mathcal{F}$ is closed under explicit definitions of partial functions without constants then each identity function I_i^n has the following explicit definition without constants in $\mathcal{F}$:

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

If the n-ary partial function f is defined by composition from the partial functions g, $h_1, \dots, h_m$ in $\mathcal{F}$ then we define $f \in \mathcal{F}$ by the explicit definition without constants 3.1.3(1).

We prove the implication ($\leftarrow$) by assuming that $\mathcal{F}$ contains the identity functions I_i^n and that it is closed under the composition of partial functions. We show that $\mathcal{F}$ is closed under explicit definitions $f(\vec{x}) \simeq \tau[\vec{x}]$ without constants of n-ary functions f by induction on the structure of terms τ.

If $\tau \equiv x_i$ with $1 \le i \le n$ then f explicitly defined as $f(x_1, \dots, x_n) = x_i$ is the identity function $I_i^n \in \mathcal{F}$.

If $\tau \equiv g(\rho_1[\vec{x}], \dots, \rho_m[\vec{x}])$ for an m-ary function symbol g then we explicitly define the n-ary partial functions $h_1(\vec{x}) \simeq \rho_1[\vec{x}]$, $\dots$, $h_m(\vec{x}) \simeq \rho_m[\vec{x}]$ and get $h_1 \in \mathcal{F}$, $\dots$, $h_m \in \mathcal{F}$ by IH. We now obtain $f \in \mathcal{F}$ from g, $h_1, \dots, h_m$ by composition 3.1.3(1). ■

3.2.2 Theorem (Characterization of explicit definitions) *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_n, and it is closed under composition.*

Proof. In the direction ($\rightarrow$) we note that $\mathcal{F}$ is also closed under explicit definitions without constants and so $\mathcal{F}$ contains the identity functions and is closed under composition by Thm. 3.2.1. The class $\mathcal{F}$ also contains each constant function C_n by explicit definition: $C_n(x) = n$.

In the direction ($\leftarrow$) we take an explicit definition s $f(\vec{x}) \simeq \tau[\vec{x}]$ such that all partial functions applied in τ are from $\mathcal{F}$. We replace every constant symbol c denoting the number n and occurring in τ by the application $C_n(x_1)$ whereby we obtain a term τ_1. Clearly we have $\tau[\vec{x}] \simeq \tau_1[\vec{x}]$ and, since all partial functions applied in τ_1 are from $\mathcal{F}$, we define $f \in \mathcal{F}$ by explicit definition without constants: $f(\vec{x}) \simeq \tau_1[\vec{x}]$. ■

3.3 Explicit Definitions of Predicates

3.3.1 Bounded quantification. *Bounded quantifiers* are formulas of forms $\exists y{\le}\tau[\vec{x}]\,\phi[y,\vec{x}]$ and $\forall y{\le}\tau[\vec{x}]\,\phi[y,\vec{x}]$ where the variable y does not occur among $\vec{x}$. Bounded quantifiers are abbreviations for the formulas $\exists x(x \le \tau \land \phi)$ and $\forall x(x \le \tau \rightarrow \phi)$ respectively.

We permit also strict bounds on existential quantifiers: $\exists x{<}\tau\,\phi$ as abbreviation for $\exists x{\le}\tau\,(x \ne \tau \land \phi)$. Strict bounds on universal quantifiers are similar.

3.3.2 Explicit definitions of predicates. We wish to define predicates by *explicit definitions* of a form

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

where the formula ϕ contains at most the indicated n-tuple of variables free, its terms are composed from variables and constant symbols by applications of (total) functions, and ϕ is constructed from *atomic* formulas: $\tau_1 < \tau_2$, $\tau_1 \le \tau_2$, $\tau_1 = \tau_2$, and $Q(\vec{\tau})$ by booelan connectives ($\neg$, $\land$, $\lor$, and $\rightarrow$) and by bounded quantification. The formula ϕ may not apply the predicate symbol R.

If the equivalence (1) is such that the list of $g_1, \ldots, g_k$ ($k \ge 0$) of functions is minimal such that every function f appplied 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.

3.3.3 Theorem (Explicit definitions of predicates without bounded quantification) *Every class $\mathcal{F}$ closed under explicit definitions of functions and such that $D, \le_* \in \mathcal{F}$ is closed under explicit definitions of predicates without bounded quantification.*

Proof. We show that $\mathcal{F}$ is closed under explicit definitions s $R(\vec{x}) \leftrightarrow \phi[\vec{x}]$ wiithout bounded quantification of n-ary predicates by induction on the structure of formulas ϕ. If ϕ is one of $\tau_1 \leq \tau_2$, $\tau_1 < \tau_2$, or $\tau_1 = \tau_2$ we define R in $\mathcal{F}$ by the corresponding explicit clausal definition.

$$R(\vec{x}) \leftarrow \tau_1 \leq \tau_2$$
$$R(\vec{x}) \leftarrow \tau_2 > \tau_1$$
$$R(\vec{x}) \leftarrow \tau_1 \leq \tau_2 \land \tau_2 \leq \tau_1 \ .$$

Note that the clausal definitions fold to explicit definitions of the characteristic functions R_* which are form D and $\leq_*$.

If $\phi \equiv Q(\vec{\rho})$ then, since $Q_* \in \mathcal{F}$, we define R in $\mathcal{F}$ by explicit clausal definition:

$$R(\vec{x}) \leftarrow Q(\vec{\rho}) \ .$$

If $\phi \equiv \neg\phi_1$ we use IH and define an auxiliary n-ary predicate $Q \in \mathcal{F}$ by explicit predicate definition: $Q(\vec{x}) \leftrightarrow \phi_1[\vec{x}]$. We then define R in $\mathcal{F}$ by explicit clausal definition:

$$R(\vec{x}) \leftarrow \neg Q(\vec{x}) \ .$$

If $\phi \equiv \phi_1 \land \phi_2$ 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 clausal definition:

$$R(\vec{x}) \leftarrow Q_1(\vec{x}) \land Q_2(\vec{x}) \ .$$

We leave the cases with the remaining propositional connectives to the reader.
■

3.3.4 Bounded minimalization. The function operator of *bounded minimalization* takes an $(n+1)$-ary (total) function g and yields an $(n+1)$-ary function f such that $f(y,\vec{x}) = z$ where z is the smallest number $z \leq y$ such that $g(z,\vec{x}) = 1$. We have $f(y,\vec{x}) = 0$ if there is no such z. We write

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

Parameterless bounded minimimalization is a function operator taking a binary function g and yielding a unary function f:

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

which yields the least $z \leq x$ for which $g(z,x) = 1$ and $z = 0$ if there is no such z.

Classes closed under explicit definitions of partial functions and under bounded minimalization are also closed under parameterless bounded minimalization because we can explicitly define:

$$\begin{aligned} g_1(z,x,y) &= g(z,x) \\ f_1(x,y) &= \mu_{z \leq x}[g_1(z,x,y) = 1] \\ f(x) &= f_1(x,x) \ . \end{aligned}$$

In classes closed under explicit definitions of partial functions and under bounded minimalization we can admit terms applying (total) functions in bounded minimalization:

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

as abbreviation for the following explicit definitions:

$$\begin{aligned} g(z,y,\vec{x}) &= \rho[z,y,\vec{x}] \\ f_1(b,y,\vec{x}) &= \mu_{z\leq b}[g(z,y,\vec{x}) = 1] \\ f(y,\vec{x}) &= f_1(\tau[y,\vec{x}],y,\vec{x}) \ . \end{aligned}$$

In classes closed under explicit definitions of partial functions and predicates we can admit in bounded minimalization formulas suitable for explicit definition of predicates:

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

as abbreviation for

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

where we explicitly define the predicate R as

$$R(z,y,\vec{x}) \leftrightarrow \phi[z,y,\vec{x}] \ .$$

3.3.5 Theorem (Explicit definitions of predicates) *Every class $\mathcal{F}$ closed under explicit definitions of functions, under bounded minimalization, and such that $D, \leq_* \in \mathcal{F}$ is closed under explicit definitions of predicates.*

Proof. By Thm. 3.3.3 the class is closed under explicit definitions of predicates without bounded quantification and so it suffices to reduce bounded quantification to bounded minimalization. If $R(\vec{x}) \leftrightarrow \exists y{\leq}\tau[\vec{x}]\, \phi[y,\vec{x}]$ where the formula ϕ is without bounded quantification then we define an auxiliary *witnessing function* $f \in \mathcal{F}$ by bounded minimalization:

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

The predicate R has then explicit definition in $\mathcal{F}$ by:

$$R(\vec{x}) \leftrightarrow \phi[f(\vec{x}),\vec{x}] \ .$$

If $R(\vec{x}) \leftrightarrow \forall y{\leq}\tau[\vec{x}]\, \phi[y,\vec{x}]$ we define the we define an auxiliary *counterexample function* $f \in \mathcal{F}$ by bounded minimalization:

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

The predicate R has then explicit definition in $\mathcal{F}$ by:

$$R(\vec{x}) \leftrightarrow \phi[f(\vec{x}),\vec{x}] \ . \qquad \blacksquare$$

Chapter 4

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 measure functions (see Thm. 4.3.5). Contrary to the original expectations, primitive recursive functions are not closed under recursive definitions (see Par. 5.1.6). In Sect. 4.4 we will characterize unary primitive recursive functions by means of pair recursion which is exponentially faster than primitive recursion.

4.1 Primitive Recursion

4.1.1 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 satisfying:

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

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

The contraction of the operator of primitive recursion is the operator of *unary primitive recursion* which takes two unary functions g and h and yields the unary function f satisfying:

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

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

Recursive functions are primitively recursively closed because S is a recursive function and the remaining initial functions are recursive functions by explicit definitions:

$$Z(x) = 0$$
$$I(x_1, \dots, x_i, \dots, x_n) = x_i \ .$$

The closure of recursive functions under composition is trivial because we can use 3.1.3(1) directly as explicit definition and the closure under primitive recursion follows from the following recursive definition:

$$f(x, \vec{y}) = D(x, h(x \dot{-} 1, \vec{y}, f(x \dot{-} 1, \vec{y})), g(\vec{y}))$$

regular in the first argument.

4.1.3 Constant functions are primitive recursive. We show by induction on n that every 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: $f(x) = C_n(x) + 1$.

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

Proof. Directly from Par. 4.1.3 and Thm. 3.2.2. ■

4.1.5 Case analysis function D 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:

$$D(0, y, z) = I_2^2(y, z)$$
$$D(x + 1, y, z) = I_2^4(x, y, z, D(x, y, z)) \ .$$

4.1.6 Predecessor function $x \dot{-} 1$ is primitive recursive. The unary predecessor function $x \dot{-} 1$ satisfies $0 \dot{-} 1 = 0$ and $(x+1) \dot{-} 1 = x$. We derive $x \dot{-} 1$ as a primitive recursive function by explicit definition from an auxiliary binary function f which is defined by primitive recursion from Z and I_1^3:

$$\begin{aligned}&f(0,y) = Z(y)\\&f(x+1,y) = I_1^3(x,y,f(x,y))\\&x \dot{-} 1 = f(x,x)\ .\end{aligned}$$

4.1.7 Primitive recursive definitions. In order to obtain the comfort of recursive definitions in the clausal form we introduce a special form of recursive definitions and show in Thm. 4.1.8 the primitive recursive functions closed under it.

A recursive definition $f(\vec{y},x,\vec{z}) = \tau[f;\vec{y},x,\vec{z}]$ from (total) functions is a *primitive recursive definition* of the function f if $\vec{y}$ is an n-tuple and $\vec{z}$ an m-tuple of variables $(n+m \geq 0)$ and we have

$$\phi[f;\vec{y},x,\vec{z}] \to x > 0 \tag{1}$$

for every occurrence in τ of a recursive application f governed by ϕ and the application has the form $f(\vec{y},x \dot{-} 1,\vec{z})$. The definition is clearly regular in the $(n+1)$-st argument.

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

Proof. If f is defined by primitive recursive definition $f(\vec{y},x,\vec{z}) = \tau[f;\vec{y},x,\vec{z}]$ from functions in $\mathcal{F}$ then we explicitly define in $\mathcal{F}$ two auxiliary functions:

$$\begin{aligned}&g(w,\vec{y},\vec{z}) = \tau[\dot{\lambda}\vec{y},x,\vec{z}.0;\vec{y},0,\vec{z}]\\&h(x,w,\vec{y},\vec{z},a) = \tau[\dot{\lambda}\vec{y},x,\vec{z}.a;\vec{y},x+1,\vec{z}]\ .\end{aligned}$$

We then define an $(n+m+2)$-ary function $f_1 \in \mathcal{G}_2$ by primitive recursion:

$$\begin{aligned}f_1(0,w,\vec{y},\vec{z}) &= g(w,\vec{y},\vec{z})\\f_1(x+1,w,\vec{y},\vec{z}) &= h(x,w,\vec{y},\vec{z},f_1(x,w,\vec{y},\vec{z}))\ .\end{aligned}$$

We now prove by induction on x:

$$f(\vec{y},x,\vec{z}) = f_1(x,0,\vec{y},\vec{z})\ . \tag{1}$$

In the base case let $\phi[f;\vec{y},x,\vec{z}]$ be the condition governing a recursive application $f(\vec{y},x \dot{-} 1,\vec{z})$ in τ. We have

$$\phi[f;\vec{y},0,\vec{z}] \to f(\vec{y},0 \dot{-} 1,\vec{z}) = 0$$

because the antecedent cannot hold by 4.1.7(1). Thus we obtain:

$$\begin{aligned}f(\vec{y},0,\vec{z}) = \tau[f;\vec{y},0,\vec{z}] &\overset{\text{Thm. 2.1.15}}{=} \tau[\dot{\lambda}\vec{y},x,\vec{z}.0;\vec{y},0,\vec{z}] =\\&g(0,\vec{y},\vec{z}) = f_1(0,0,\vec{y},\vec{z})\ .\end{aligned}$$

In the inductive case we have:

$$f(\vec{y}, x+1, \vec{z}) = \tau[f; \vec{y}, x+1, \vec{z}] = \tau[\dot{\lambda}\vec{y}, x, \vec{z}.f(\vec{y}, x, \vec{z}); \vec{y}, x+1, \vec{z}] \stackrel{\text{IH}}{=}$$
$$\tau[\dot{\lambda}\vec{y}, x, \vec{z}.f_1(x, 0, \vec{y}, \vec{x}); \vec{y}, x+1, \vec{z}] = h(x, 0, \vec{y}, \vec{z}, f_1(x, x, \vec{y}, \vec{z})) = f_1(x+1, 0, \vec{y}, \vec{z}) \ .$$

With (1) 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 $n = m = 0$ where we need that the function f_1 has at least one parameter as required by the operator of primitive recursion. ■

4.1.9 Addition $x + y$ is primitive recursive. The addition function $+$ 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 $x+1$.

4.1.10 Modified subtraction $x \dot{-} y$ 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 $x \dot{-} 1$.

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

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

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

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

4.1.13 Predicate $x \leq y$ is primitive recursive. The binary predicate $x \leq y$ is primitive recursive by explicit definition:

$$x \leq y \leftarrow x \dot{-} y = 0 \ .$$

4.1.14 Theorem *Primitively recursively closed classes $\mathcal{F}$ are closed under bounded minimalization.*

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

$$\begin{aligned}
&f(0,\vec{x}) = 0\\
&f(z+1,\vec{x}) = f(z,\vec{x}) \leftarrow g(f(z,\vec{x}),\vec{x}) = 1\\
&f(z+1,\vec{x}) = z+1 \leftarrow g(f(z,\vec{x}),\vec{x}) \ne 1 \land g(z+1,\vec{x}) = 1\\
&f(z+1,\vec{x}) = 0 \leftarrow g(f(z,\vec{x}),\vec{x}) \ne 1 \land g(z+1,\vec{x}) \ne 1 \ .
\end{aligned}$$ ■

4.1.15 Theorem *Primitively recursively closed classes $\mathcal{F}$ are closed under explicit definitions of predicates with bounded quantification.*

Proof. The class $\mathcal{F}$ contains the function D and the predicate $\le$ and so the theorem follows from Theorems 3.3.5 and 4.1.14. ■

4.1.16 Modified division $\div$ and remainder functions mod **are primitive recursive.** We define the modified division function as a primitive recursive function by bounded minimalization:

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

We then define the remainder function mod as a primitive recursive function by explicit definition:

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

4.1.17 The operator of iteration. We have encountered in Par. 1.1.7 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:

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

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:

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

The contraction of the operator of iteration is the operator of *unary iteration* which takes a unary function g and yields the unary function f satisfying:

$$f(0) = 0$$
$$f(0, y) = y$$
$$f(x+1, y) = g\, f(x, y) \ .$$

We will often write $g^{\tau_1}(\tau_2)$ as an abbreviation for the application $f(\tau_1, \tau_2)$.

For a unary function g we say that the unary function f is defined by the *unary length iteration of* g if f satisfies:

$$f(0) = 0$$
$$f(x, y) = g^{L(x)}(y) \ .$$

4.1.18 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:

$$f(0, y) = y$$
$$f(x+1, y) = g\, f(x, y) \ . \qquad \blacksquare$$

4.2 Course of Values Recursion

Primitive recursive definitions are a special case of stronger *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. In this section we prove that primitive recursive functions are closed under course of values recursive definitions. For that we need to show that the pairing function (x, y) and the projection functions H and T are primitive recursive functions.

4.2.1 Catalan function C 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:

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

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 $|\tau| = n$. 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

$$\begin{aligned} C(n+1) \stackrel{4.2.1(1)}{=} {} & \frac{1}{n+2} \cdot \binom{2 \cdot n + 2}{n+1} = \\ & \frac{1}{n+2} \cdot \frac{(2 \cdot n + 2) \cdot (2 \cdot n + 1)}{(n+1)^2} \cdot \binom{2 \cdot n}{n} = \frac{4 \cdot n + 2}{n+2} \cdot C(n) \; . \end{aligned}$$

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

$$\begin{aligned} & C(0) = 1 \\ & C(n+1) = (4 \cdot n + 2) \cdot C(n) \div (n+2) \quad . \end{aligned}$$

4.2.2 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:

$$\begin{aligned} & \sigma(0) = 0 \\ & \sigma(n+1) = \sigma(n) + C(n) \; . \end{aligned}$$

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

$$\begin{aligned} & |0| = 0 \\ & |v, w| = |v| + |w| + 1 \end{aligned}$$

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

We use instead Property 1.3.11(5) which says that $\sigma(|x|) \le x < \sigma(|x|+1)$ and and Property 1.3.12(5) which says that $|x| \le x$ and define $|x|$ as a primitive recursive function by bounded minimalization:

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

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

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

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

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

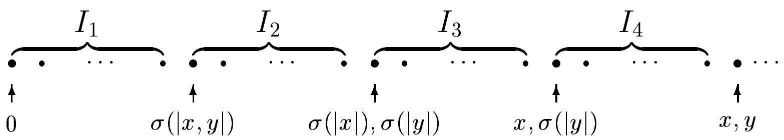


Figure 4.1: Arithmetic derivation of the pairing function (x, y)

4.2.5 Pairing function (x, y) **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. 4.1 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 $\sigma(|x,y|) = \sigma(|x|+|y|+1)$ numbers z such that

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

The interval I_2 consists of numbers v, w such that

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

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

$$\begin{aligned}&\textstyle\sum_{i<0} C(i){\cdot}C(z \dot{-} i) = 0\\ &\textstyle\sum_{i<x+1} C(i){\cdot}C(z \dot{-} i) = C(x){\cdot}C(z \dot{-} x) + \sum_{i<x} C(i){\cdot}C(z \dot{-} i) \quad .\end{aligned}$$

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

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

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

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

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:

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

4.2.6 Projection functions H and T 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:

$$\begin{aligned} H(x) &= \mu_{v \leq x}[\exists w {\leq} x\,(x = v, w)] \\ T(x) &= \mu_{w \leq x}[\exists w {\leq} x\,(x = v, w)] \ . \end{aligned}$$

4.2.7 List indexing function $(x)_i$ is primitive recursive. The list indexing function $(x)_i$ (see Par. 1.4.5) 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 eliminable, 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:

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

4.2.8 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:

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

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

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:

$$(\overline{f}(x+1,\vec{y}))_{x+1-(z+1)} = (f(x,\vec{y}),\overline{f}(x,\vec{y}))_0 = f(z,\vec{y}) \ .$$

If $z < x$ then we have:

$$(\overline{f}(x+1,\vec{y}))_{x+1-(z+1)} = (f(x+1,\vec{y}),\overline{f}(x,\vec{y}))_{x+1-(z+1)} = \\ (\overline{f}(x,\vec{y}))_{x-(z+1)} \stackrel{IH}{=} f(z,\vec{y}) \ .$$

4.2.9 Course of values recursive definitions. Recursive definitions where the recurisive argument goes arbitraily down in the recursive applications while the parameters stay unchanged are the obvious generalization of primitive recursive definitions.

A recursive definition $f(\vec{y},x,\vec{z}) = \tau[f;\vec{y},x,\vec{z}]$ from (total) functions is a *course of values recursive definition* of the function f if $\vec{y}$ is an n-tuple and $\vec{z}$ an m-tuple of variables $(n + m \geq 0)$ and we have

$$\phi[f;\vec{y},x,\vec{z}] \to \rho[f;\vec{y},x,\vec{z}] < x$$

for every occurrence in τ of a recursive application f governed by ϕ and the application has the form $f(\vec{y},\rho,\vec{z})$ where ρ is without recursive applications of f. The definition is clearly regular in the $(n+1)$-st argument.

Pair recursion (see Par. 1.3.4) is an example of course of values recursive definition:

$$f(0,\vec{y}) = g(\vec{y})$$
$$f((v,w),\vec{y}) = h(v,w,\vec{y},f(v,\vec{y}),f(w,\vec{y}))$$

because the parameters $\vec{y}$ are not changed in recursion and the first arguments of both recursive applications are strictly less than (v,w).

The following theorem asserts that primitive recursive functions are closed under course of values recursive definitions.

4.2.10 Theorem *Primitively recursively closed classes $\mathcal{F}$ are closed under course of values recursive definitions.*

Proof. Let the $(n + m + 1)$-ary function f be defined by course of values recursive definition: $f(\vec{y},x,\vec{z}) = \tau[f;\vec{y},x,\vec{z}]$ from the functions of $\mathcal{F}$. We define an auxiliary $(n + m + 1)$-ary function $f_1 \in \mathcal{F}$ by primitive recursive definition:

$$f_1(0,\vec{y},\vec{z}) = 0$$
$$f_1(x+1,\vec{y},\vec{z}) = \tau[\dot{\lambda}\vec{y},v,\vec{z}.(a)_{x \dot{-} (v+1)};\vec{y},x,\vec{z}], a \leftarrow f_1(x,\vec{y},\vec{z}) = a \ .$$

The function f_1 is actually the course of values function for the explicitly defined function $f_2(x,\vec{y},\vec{z}) = f(\vec{y},x,\vec{z})$ but for our purposes it is sufficient to show only the equivalent of Property 4.2.8(2):

$$\forall v\,(v < x \to (f_1(x,\vec{y},\vec{z}))_{x-(v+1)} = f(\vec{y},v,\vec{z}))$$

In the base case there is nothing to prove. In the inductive case we set $a = f_1(\vec{y}, x, \vec{z})$. For every condition ϕ governing an application $f(\vec{y}, \rho, \vec{z})$ in τ we have $\phi[f; \vec{y}, x, \vec{z}] \to \rho[f; \vec{y}, x, \vec{z}] < x$ and so we obtain from IH:

$$\phi[f; \vec{y}, x, \vec{z}] \to (a)_{x-(\rho[f;\vec{y},x,\vec{z}]+1)} = f(\vec{y}, \rho[f; \vec{y}, x, \vec{z}], \vec{z}) \ .$$

Thus we have:

$$f_1(x+1, \vec{y}, \vec{z}) = \tau[\dot{\lambda}\vec{y}, v, \vec{v}.(a)_{x \dot{-} (v+1)}; \vec{y}, x, \vec{z}], a \overset{\text{Thm. 2.1.15}}{=} \tau[f; \vec{y}, x, \vec{z}], a = f(\vec{y}, x, \vec{z}), a \ .$$

We now assume $v < x + 1$ and consider two cases. If $v = x$ we have

$$(f_1(x+1, \vec{y}, \vec{z}))_{x+1-(v+1)} = (f_1(x+1, \vec{y}, \vec{z}))_0 = (f(\vec{y}, x, \vec{z}), a)_0 = f(\vec{y}, v, \vec{z}) \ .$$

If $v < x$ then we have

$$(f_1(x+1, \vec{y}))_{x+1-(v+1)} = (f(\vec{y}, x, \vec{z}), a)_{x+1-(v+1)} = (a)_{x-(v+1)} \overset{IH}{=} f(\vec{y}, v, \vec{z}) \ .$$

We now use the just proved property with $x+1$ for x and x for v and explicitly define $f \in \mathcal{F}$ by:

$$f(\vec{y}, x, \vec{z}) = (f_1(x+1, \vec{y}, \vec{z}))_0 \quad . \quad \blacksquare$$

4.3 Recursion Regular in a Primitive Recursive Measure

R. Péter [15] 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 [24] (see also [19]) k-recursive functions can be defined by $< \omega^{\omega^k}$-ordinal recursion. For instance, the well known Péter-Ackermann's function A (see Par. 1.1.24) is 2-recursive. Péter has proved in [14] 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. 4.3.5 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.

4.3.1 Arithmetization of $R^{\mathcal{F}}$-terms. Let $\mathcal{F}$ be a class of functions enumerated by g_0, g_1, $\ldots$. We arithmetize the $R^{\mathcal{F}}$-terms with the following constructors:

$$\begin{aligned}
&\boldsymbol{C}(n) = 0, n\\
&\boldsymbol{X}(i,n) = 1, i, n\\
&\boldsymbol{S}(a) = 2, a\\
&\boldsymbol{P}(a) = 3, a\\
&\boldsymbol{D}(a,b,c) = 4, a, b, c\\
&\boldsymbol{G}(a) = 5, a\\
&(\boldsymbol{\lambda} a)(b) = 6, a, b\\
&\boldsymbol{F}(a) = 7, p\\
&\boldsymbol{A}(a,b) = 8, a, b
\end{aligned}$$

The arities of constructors are as shown in their definitions.

We assign to every number m and every $R^{\mathcal{F}}$-term τ such that all free variables x_i of τ have $i \leq m$ its code $\ulcorner\tau\urcorner^m$. Codes satisfy the following:

$$\begin{aligned}
\ulcorner x_{i+1}\urcorner^m &= \boldsymbol{X}(i,m)\\
\ulcorner 0\urcorner^m &= \boldsymbol{C}(0)\\
\ulcorner S(\tau)\urcorner^m &= \begin{cases} \boldsymbol{C}(k+1) & \tau \text{ is a numeral } S^k(0)\\ \boldsymbol{S}(\ulcorner\tau\urcorner^m) & \text{otherwise}\end{cases}\\
\ulcorner(\tau \dot{-} 1)\urcorner^m &= \boldsymbol{P}(\ulcorner\tau\urcorner^m)\\
\ulcorner D(\tau_1,\tau_2,\tau_3)\urcorner^m &= \boldsymbol{D}(\ulcorner\tau_1\urcorner^m, \ulcorner\tau_2\urcorner^m, \ulcorner\tau_3\urcorner^m)\\
\ulcorner g_i(\rho_1,\ldots,\rho_n)\urcorner^m &= \boldsymbol{G}(i, \ulcorner(\rho_1,\ldots,\rho_n)\urcorner^m)\\
\ulcorner(\lambda x_1,\ldots,x_n.\tau)(\rho_1,\ldots,\rho_n)\urcorner^m &= (\boldsymbol{\lambda}\ulcorner\tau\urcorner^n)(\ulcorner(\rho_1,\ldots,\rho_n)\urcorner^m)\\
\ulcorner f(\rho_1,\ldots,\rho_n)\urcorner^m &= \boldsymbol{F}(\ulcorner(\rho_1,\ldots,\rho_n)\urcorner^m)\\
\ulcorner(\rho_1,\rho_2,\ldots,\rho_n)\urcorner^m &= \begin{cases} \ulcorner\rho_1\urcorner^m & \text{if } n = 1\\ \boldsymbol{A}(\ulcorner\rho_1\urcorner^m, \ulcorner(\rho_2,\ldots,\rho_n)\urcorner^m) & \text{if } n > 1\ .\end{cases}
\end{aligned}$$

The sole purpose of the argument m in $\ulcorner\tau\urcorner^m$ is to ship it to the code $\ulcorner x_{i+1}\urcorner^m = \boldsymbol{X}(i,m)$ of the $(i{+}1)$-th out of m variables. Note how the term τ of an n-ary $R^{\mathcal{F}}$-function symbol $\lambda x_1,\ldots,x_n.\tau$ which may have free the variables x_1, $\ldots$, x_n is coded by $\ulcorner\tau\urcorner^n$. Also note the use in the last reccurrence of the binary constructor $\boldsymbol{A}$ to code the n-tuples of arguments of applications of g_i, f, and $\lambda x_1,\ldots,x_n.\tau$.

We could now define by course of values recursive definition a primitive recursive predicate $Rterm(m,c)$ holding of codes c of $R^{\mathcal{F}}$-terms and also the primitive recursive predicate $Crterm(m,c)$ holding of codes of closed $R^{\mathcal{F}}$-terms. Since we will not need these predicates, we leave their definitions to the reader who should get the idea of how to set up the course of values recursive definitions from the definitions in the following paragraphs.

4.3.2 Auxiliary coding predicates and functions. Applications of $\lambda\vec{x}.\tau$ or of functions $g_i \in \mathcal{F}$ are reduced when their arguments are reduced to monadic numerals. In order to recognize when the codes of arguments are reduced we will need a unary predicate $\boldsymbol{Num}$ holding only when $\boldsymbol{Num}(\ulcorner(\underline{\vec{x}})\urcorner^m)$ where $\vec{x}$ are arbitrary n-tuples of numbers. The predicate is primitive recursive by parameterless course of values recursive definition:

$$
\begin{aligned}
&\boldsymbol{Num}\,\boldsymbol{C}(n)\\
&\boldsymbol{Num}\,\boldsymbol{A}(\boldsymbol{C}(n),a) \leftarrow \boldsymbol{Num}(a)\ .
\end{aligned}
$$

We will also need a unary *decoding* function $\boldsymbol{Dc}$ taking codes of arguments reduced to monadic numerals and yielding the numbers paired, i.e. such that we have:

$$\boldsymbol{Dc}(\ulcorner(\underline{\vec{x}})\urcorner^m) = \vec{x}\ .$$

The function is primitive recursive by parameterless course of values recursive definition:

$$
\begin{aligned}
&\boldsymbol{Dc}\,\boldsymbol{C}(n) = n\\
&\boldsymbol{Dc}\,\boldsymbol{A}(\boldsymbol{C}(n),a) = n, \boldsymbol{Dc}(a)\ .
\end{aligned}
$$

4.3.3 Arithmetization of substitution is primitive recursive. The substitution function $\tau[f;\vec{\rho}]$ is over $R^{\mathcal{F}}$-terms. Its arithmetization $a[b;c]$ takes three codes of $R^{\mathcal{F}}$-terms and yields another code satisfying the following:

$$\ulcorner\tau_1\urcorner^n[\ulcorner\tau_2\urcorner^n;\ulcorner(\rho_1,\ldots,\rho_n)\urcorner^n] = \ulcorner\tau_1[\lambda\vec{x}.\tau_2;\rho_1,\ldots,\rho_n]\urcorner^n\ .$$

For the arithmetization of substitution we will need an auxiliary ternary selection function $Sel(i,n,a)$ selecting the code of the $(i{+}1)$-th out of n arguments coded by a. The basic property of the selection function is thus:

$$i < n \rightarrow Sel(i,n,\ulcorner(\rho_1,\ldots,\rho_n)\urcorner^m) = \ulcorner\rho_{i+1}\urcorner^m\ .$$

For reasons which will become clear in the proof of 4.3.4(3) we wish to select also the i-th components from numbers of a form $\boldsymbol{C}(x_1,\ldots,x_n)$. The selection function will thus satisfy also:

$$i < n \rightarrow Sel(i,n,\boldsymbol{C}(x_1,\ldots,x_n)) = \boldsymbol{C}(x_{i+1})\ .$$

The selection function is primitive recursive by the following clausal definitions:

$$
\begin{aligned}
&\boldsymbol{H}\,\boldsymbol{A}(a,b) = a\\
&\boldsymbol{H}\,\boldsymbol{C}(n) = \boldsymbol{C}\,H(n)\\
&\boldsymbol{T}\,\boldsymbol{A}(a,b) = b\\
&\boldsymbol{T}\,\boldsymbol{C}(n) = \boldsymbol{C}\,T(n)\\
&Sel(i,n,a) = \boldsymbol{H}\,\boldsymbol{T}^i(a) \leftarrow i+1 < n\\
&Sel(i,n,a) = \boldsymbol{T}^i(a) \leftarrow i+1 = n\ .
\end{aligned}
$$

The arithmetized substitution function is primitive recursive by course of values definition recursive in the first argument:

$$
\begin{aligned}
&\boldsymbol{C}(n)[b;c] = \boldsymbol{C}(n)\\
&\boldsymbol{X}(i,n)[b;c] = Sel(i,n,c)\\
&\boldsymbol{S}(a)[b;c] = \boldsymbol{C}(n+1) \leftarrow a[b;c] = a_1 \wedge \boldsymbol{Num}(a_1) \wedge a_1 = \boldsymbol{C}(n)\\
&\boldsymbol{S}(a)[b;c] = \boldsymbol{S}(a_1) \leftarrow a[b;c] = a_1 \wedge \neg \boldsymbol{Num}(a_1)\\
&\boldsymbol{P}(a)[b;c] = \boldsymbol{P}(a_1) \leftarrow a[b;c] = a_1\\
&\boldsymbol{D}(a_1,a_2,a_3)[b;c] = \boldsymbol{D}(a_1[b;c], a_2[b;c], a_3[b;c])\\
&\boldsymbol{G}(i,a)[b;c] = \boldsymbol{G}(i,a[b;c])\\
&(\boldsymbol{\lambda} a_1)(a_2)[b;c] = (\boldsymbol{\lambda} a_1)(a_2[b;c])\\
&\boldsymbol{F}(a)[b;c] = (\boldsymbol{\lambda} b)(a[b;c])\\
&\boldsymbol{A}(a_1,a_2)[b;c] = \boldsymbol{A}(a_1[b;c], a_2[b;c]) \ .
\end{aligned}
$$

Substitution in codes $\boldsymbol{S}(a)$ needs explanation. Because we directly code monadic numerals with the constructor $\boldsymbol{C}$ we should for $a = \ulcorner S^k(x_{i+1})\urcorner^n$ and $Sel(i,n,c) = \ulcorner S^m(0)\urcorner^n = \boldsymbol{C}(m)$ have $a[b;c] = \ulcorner S^{k+m}(0)\urcorner^n = \boldsymbol{C}(k+m)$. This is achieved by the additional test whether the substitution into a yields the code of a monadic numeral.

It should be clear that because of the special nature of the selection function we have for all n-ary $R^{\mathcal{F}}$-function symbols $\lambda\vec{x}.\tau$:

$$\ulcorner\tau\urcorner^n[\ulcorner\tau\urcorner^n; \ulcorner(\underline{\vec{x}})\urcorner^n] = \ulcorner\tau\urcorner^n[\ulcorner\tau\urcorner^n; \boldsymbol{C}(\vec{x})] \ . \tag{1}$$

4.3.4 Arithmetization of reductions. Let $\mathcal{F}$ be a class of functions enumerated by g_0, g_1, ... and let the binary function g enumerate the class. Thus for every n-ary function $g_i \in \mathcal{F}$ and every n-tuple of numbers $\vec{x}$ we have

$$g_i(\vec{x}) = g(i,\vec{x}) \ .$$

We intend to define a unary function $\boldsymbol{Rd}_g$ such that for two closed $R^{\mathcal{F}}$-terms τ_1 and τ_2 we will have

$$\tau_1 \text{ reduces to } \tau_2 \text{ in one reduction step iff } \boldsymbol{Rd}_g(\ulcorner\tau_1\urcorner^n) = \ulcorner\tau_2\urcorner^n. \tag{1}$$

Note that the parameter n is irrelevant for closed terms because they have no free variables.

The function $\boldsymbol{Rd}_g$ is defined as primitive recursive in g by parameterless course of values recursive definition:

$$
\begin{aligned}
&\boldsymbol{Rd}_g\, \boldsymbol{C}(n) = \boldsymbol{C}(n)\\
&\boldsymbol{Rd}_g\, \boldsymbol{S}(a) = \boldsymbol{C}(n+1) \leftarrow \boldsymbol{Rd}_g(a) = a_1 \wedge \boldsymbol{Num}(a_1) \wedge a_1 = \boldsymbol{C}(n)\\
&\boldsymbol{Rd}_g\, \boldsymbol{S}(a) = \boldsymbol{S}(a_1) \leftarrow \boldsymbol{Rd}_g(a) = a_1 \wedge \neg\boldsymbol{Num}(a_1) \wedge a_1 > 0\\
&\boldsymbol{Rd}_g\, \boldsymbol{P}(a) = \boldsymbol{C}(n \dot{-} 1) \leftarrow \boldsymbol{Num}(a) \wedge a = \boldsymbol{C}(n)\\
&\boldsymbol{Rd}_g\, \boldsymbol{P}(a) = \boldsymbol{P}(a_1) \leftarrow \neg\boldsymbol{Num}(a) \wedge \boldsymbol{Rd}_g(a) = a_1 \wedge a_1 > 0\\
&\boldsymbol{Rd}_g\, \boldsymbol{D}(a,b,c) = D(n,b,c) \leftarrow \boldsymbol{Num}(a) \wedge a = \boldsymbol{C}(n)\\
&\boldsymbol{Rd}_g\, \boldsymbol{D}(a,b,c) = \boldsymbol{D}(a_1,b,c) \leftarrow \neg\boldsymbol{Num}(a) \wedge \boldsymbol{Rd}_g(a) = a_1 \wedge a_1 > 0\\
&\boldsymbol{Rd}_g\, \boldsymbol{G}(i,a) = \boldsymbol{C}\, g(i, \boldsymbol{Dc}(a)) \leftarrow \boldsymbol{Num}(a)
\end{aligned}
$$

$$\begin{aligned}
&\boldsymbol{Rd}_g\, \boldsymbol{G}(i,a) = \boldsymbol{G}(i,a_1) \leftarrow \neg \boldsymbol{Num}(a) \wedge \boldsymbol{Rd}_g(a) = a_1 \wedge a_1 > 0\\
&\boldsymbol{Rd}_g((\boldsymbol{\lambda} a)(b)) = a[a\boldsymbol{;}\, b] \leftarrow \boldsymbol{Num}(b)\\
&\boldsymbol{Rd}_g((\boldsymbol{\lambda} a)(b)) = (\boldsymbol{\lambda} a)(b_1) \leftarrow \neg \boldsymbol{Num}(b) \wedge \boldsymbol{Rd}_g(b) = b_1 \wedge b_1 > 0\\
&\boldsymbol{Rd}_g\, \boldsymbol{A}(a,b) = \boldsymbol{A}(a,b_1) \leftarrow \boldsymbol{Num}(a) \wedge \boldsymbol{Rd}_g(b) = b_1 \wedge b_1 > 0\\
&\boldsymbol{Rd}_g\, \boldsymbol{A}(a,b) = \boldsymbol{A}(a_1,b) \leftarrow \neg \boldsymbol{Num}(a) \wedge \boldsymbol{Rd}_g(a) = a_1 \wedge a_1 > 0\ .
\end{aligned}$$

The reduction of codes $\boldsymbol{S}(a)$ needs an adjustment for codes of monadic numerals similar to that in the definition of the arithmetized substitution function.

Property (1) generalizes to:

$$\tau \blacktriangleright_k^{\mathcal{F}} \underline{y} \leftrightarrow \boldsymbol{Rd}_g^k(\ulcorner \tau \urcorner^n) = \boldsymbol{C}(y) \tag{2}$$

for any closed $R^{\mathcal{F}}$-term τ. We will need the following property where $\lambda\vec{x}.\tau$ is an n-ary $R^{\mathcal{F}}$-function symbol and $\vec{x}$ an n-tuple of natural numbers:

$$(\lambda\vec{x}.\tau)(\underline{\vec{x}}) \blacktriangleright_k^{\mathcal{F}} \underline{y} \leftrightarrow \boldsymbol{Rd}_g^k((\boldsymbol{\lambda}\ulcorner \tau \urcorner^n)(\boldsymbol{C}(\vec{x}))) = \boldsymbol{C}(y)\ . \tag{3}$$

(3) holds because from (2) we have $(\lambda\vec{x}.\tau)(\underline{\vec{x}}) \blacktriangleright_k^{\mathcal{F}} \underline{y}$ iff $\boldsymbol{Rd}_g^k(\ulcorner \lambda\vec{x}.\tau)(\underline{\vec{x}})\urcorner^n) = \boldsymbol{C}(y)$ and, since it must be the case that $k > 0$, we obtain (3) from:

$$\begin{aligned}
\boldsymbol{Rd}_g(\ulcorner \lambda\vec{x}.\tau)(\underline{\vec{x}})\urcorner^n) = \boldsymbol{Rd}_g((\boldsymbol{\lambda}\ulcorner \tau \urcorner^n)(\ulcorner(\underline{\vec{x}})\urcorner^n)) = \ulcorner \tau \urcorner^n[\ulcorner \tau \urcorner^n\boldsymbol{;}\, \ulcorner(\underline{\vec{x}})\urcorner^n] \overset{4.3.3(1)}{=}\\
\ulcorner \tau \urcorner^n[\ulcorner \tau \urcorner^n\boldsymbol{;}\, \boldsymbol{C}(\vec{x})] = \boldsymbol{Rd}_g((\boldsymbol{\lambda}\ulcorner \tau \urcorner^n)(\boldsymbol{C}(\vec{x})))\ .
\end{aligned}$$

4.3.5 Theorem *Primitively recursively closed classes $\mathcal{F}$ are closed under recursive definitions regular in measure functions from $\mathcal{F}$.*

Proof. Let f be defined by the recursive defintion $f(\vec{x}) = \tau_0[f;\vec{x}]$ form the functions of $\mathcal{F}$ which is regular in a measure $m \in \mathcal{F}$. Let $\mathcal{G} = \{g_0, g_1, \ldots, g_k\}$ be the finite class of functions applied in τ_0 other than D, $x+1$, $x \dot{-} 1$, and f.

We replace every constant c in the term τ_0 by the monadic numeral $S^c(0)$ whereby we obtain a $R^{\mathcal{G}}$-term τ. We have $f = \lambda\vec{x}.\tau$ and for $f^+ = (\lambda\vec{x}.\tau)^+$ we get

$$f^+(m(x_1,\ldots,x_n)+1, x_1+1, \ldots, x_n+1) = f(x_1,\ldots,x_n)+1$$

from Thm. 2.3.12. From Thm. 2.3.14 we then obtain

$$(\lambda\vec{x}.\tau)(\underline{\vec{x}}) \blacktriangleright_{d(m(\vec{x})+1)}^{\mathcal{G}} \underline{f(\vec{x})} \tag{1}$$

where $d = d_{\lambda\vec{x}.\tau}$ has the following primitive recursive definition:

$$\begin{aligned}
&d(0) = 0\\
&d(z+1) = k_1 + k_2{\cdot}d(z)
\end{aligned}$$

with k_1 the number of applications of D, $x \dot{-} 1$, g_0, g_1, $\ldots$, g_k and k_2 the number of recursive applications of f in τ.

If $\mathcal{G}$ is empty then we define $g(i,x) = 0$. Otherwise, let the arities of the functions g_0, g_1, $\ldots$, g_k applied in τ be in that order a_0, a_1, $\ldots$, a_k and we explicitly define the binary function g:

$$
\begin{aligned}
&g(0, x_1, \ldots, x_{a_0}) = g_0(x_1, \ldots, x_{a_0}) \\
&g(1, x_1, \ldots, x_{a_1}) = g_1(x_1, \ldots, x_{a_1}) \\
&\quad \vdots \\
&g(k, x_1, \ldots, x_{a_k}) = g_k(x_1, \ldots, x_{a_k}) \\
&g(n + k + 1, x_1, \ldots, x_{a_0}) = g_0(x_1, \ldots, x_{a_0}) \ .
\end{aligned}
$$

In both cases g enumerates the class $\mathcal{G}$ and we have $g \in \mathcal{F}$.

We can see from (1) and 4.3.4(3) that the following clausal definition derives the function $f \in \mathcal{F}$:

$$f(\vec{x}) = y \leftarrow \boldsymbol{Rd}_g^{d(m(\vec{x})+1)}(((\boldsymbol{\lambda}\ulcorner\tau\urcorner^n))(\boldsymbol{C}(\vec{x}))) = \boldsymbol{C}(y)$$

because the function $\boldsymbol{Rd}_g$, by being primitive recursive in g, is in $\mathcal{F}$. ■

4.3.6 Theorem *The approximation functions for partial $\mathcal{F}$-recursive functions are primitive recursive in $\mathcal{F}$ for every class of functions $\mathcal{F}$.*

Proof. Let $\mathcal{F}$ be enumerated by g_0, g_1, $\ldots$. The functions S^+, Pr^+, D^+, and g^+ for $g \in \mathcal{F}$ are primitive recursive in $\mathcal{F}$. We prove by induction on the structure of $R^{\mathcal{F}}$-terms τ such that $\lambda\vec{x}.\tau$ is an $R^{\mathcal{F}}$-function symbol that the approximation functions $(\lambda\vec{x}.\tau)^+$ are primitive recursive in $\mathcal{F}$. The $R^{\mathcal{F}}$-function symbols applied in τ are primitive recursive in $\mathcal{F}$ by IH. If τ does not apply f then $(\lambda\vec{x}.\tau)^+$ is explicitly defined and it is primitive recursive in $\mathcal{F}$. If τ applies f then $(\lambda\vec{x}.\tau)^+$ is defined by a recursive definition regular in the first argument, i.e. in the primitive recursive measure $m(z, \vec{x}) = z$, and so $(\lambda\vec{x}.\tau)^+$ is primitive recursive in $\mathcal{F}$ by Thm. 4.3.5. ■

4.4 Pair Recursion

It is of some interest to have an inductive definition of the class of unary primitive recursive functions. The class has a well-known characterization by R. M. Robinson [18, 19] as the class generated from the successor function $S(x) = x+1$ and the *quadratic residuum* function $q(x) = x - [\sqrt{x}]^2$ by the operators of unary composition, *addition*: $f(x) = g(x) + h(x)$, and *pure iteration*: $f(x) = g^x(0)$.

We have given in [27] a pair-based characterization of unary primitive recursive functions which is in the style of R. M. Robinson as the class generated from the successor function S and from the contraction $\langle D \rangle$ of the case analysis function by unary composition, pairing, and pure iteration.

We give in this section a different inductive pair-based characterization of unary primitive recursive functions which is interesting because the functions are generated from a single function by recursion on pair notation which is exponentially faster than primitive recursion.

4.4.1 Simple definitions. An explicit definition $f(x) = \tau[x]$ is a *simple definition of f from functions $g_1, \ldots, g_k$* if the functions $g_1, \ldots, g_k$ are unary and f is defined from functions among D, (x, y), H, T, and $g_1, \ldots, g_k$. Note that this means that we do not count the first four functions in the list towards the arguments of a simple definition.

A rich subclass of explicit clausal definitions of unary functions can be folded into simple definitions. The subclass permits the discrimination on pairing patterns, on constants, and on negation of defined predicates. Absent are the arithmetic discriminations (p-adic patterns, dichotomy, trichotomy, and on the negation of the identity relation) The following theorem inductively characterizes simple explicit definitions.

4.4.2 Theorem *A class of unary functions $\mathcal{F}$ is closed under simple definitions iff $\mathcal{F}$ contains the identity function I, the zero function Z, the contraction $\langle D\rangle$ of the case analysis function, and it is closed under pairing and (unary) composition.*

Proof. We prove first the implication ($\rightarrow$). If $\mathcal{F}$ is closed under simple definitions then the identity and zero function have simple definitions in $\mathcal{F}$ as $I(x) = x$ and $Z(x) = 0$ respectively. The function $\mathcal{D}$ is defined in $\mathcal{F}$ by the following simple definition as follows:

$$\langle D\rangle(0, y, z) = z$$
$$\langle D\rangle((x_1, x_2), y, z) = y \ .$$

The class $\mathcal{F}$ is trivially closed under unary composition $f(x) = g\,h(x)$ and pairing $f(x) = g(x), h(x)$.

We prove the implication ($\leftarrow$) by assuming that $\mathcal{F}$ contains I, Z, $\langle D\rangle$ and it is closed under (unary) composition and pairing. We prove by induction on the structure of terms τ that $\mathcal{F}$ is closed under simple definitions $f(x) = \tau[x]$. We may assume without loss of generality that τ does not apply any constants other than 0 because we can always replace them by pair numerals denoting the same numbers.

If $\tau \equiv 0$ then $f = Z \in \mathcal{F}$. If $\tau \equiv x$ then $f = I \in \mathcal{F}$.

If $\tau \equiv T(\rho)$ then the function $g(x) = \rho$ is in $\mathcal{F}$ by IH. We use the fact that $T(x) = \langle D\rangle(0, x)$ and define the function $f \in \mathcal{F}$ to satisfy:

$$f(x) = \langle D\rangle(Z(x), I(x))$$

by pairing and composition.

If $\tau \equiv H(\rho)$ then the function $g(x) = \rho$ is in $\mathcal{F}$ by IH. We use the fact that $H(x) = \langle D\rangle(1, x)$ and define the function $f \in \mathcal{F}$ to satisfy:

$$f(x) = \langle D\rangle((Z(x), Z(x)), I(x))$$

by two pairings and one composition.

If $\tau \equiv D(\rho_1, \rho_2, \rho_3)$ then the functions $g_1(x) = \rho_1$, $g_2(x) = \rho_2$, and $g_3(x) = \rho_3$ are in $\mathcal{F}$ by IH. We define the function $f \in \mathcal{F}$ to satisfy:

$$f(x) = \langle D \rangle(g_1(x), g_2(x), g_3(x))$$

by two pairings and one composition.

If $\tau \equiv g(\rho)$ then the function $h(x) = \rho$ is in $\mathcal{F}$ by IH. We define the function $f \in \mathcal{F}$ to satisfy:

$$f(x) = g\, h(x)$$

by composition. ■

4.4.3 Primitive pair recursive functions. The class of *primitive pair recursive* functions is generated from the second projection function T by (unary) composition, pairing, and by *unary pair recursion* which takes unary functions g and h and yields the unary function f satisfying:

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

Note that the operator of unary pair recursion can be viewed as the contraction of the operator of pair recursion (see Par. 1.3.4).

A computer scientist could say that the class of primitive pair recursive functions is defined by an extremely simple programming language with only one 'built-in' function (T) and three kinds of simple program constructors (composition, pairing, and unary pair recursion). The utmost simplicity of this programming language is also its greatest disadvantage. It is a nightmare to program, i.e. to define, functions in it. This can be seen from the laborious derivations in the proof of the following theorem.

4.4.4 Theorem *Primitive pair recursive functions are closed under simple definitions.*

Proof. By Thm. 4.4.2 it suffices to derive the functions I, Z, and $\langle D \rangle$ as primitive pair recursive.

The zero function can be obtained by a sufficiently long unary iteration of T because we have $T^z(x) = 0$ for any $z \geq L(x)$. Consider the function f satisfying:

$$\begin{aligned} f(0) &= 0 \\ f(0, y) &= T(y) \\ f((v, w), y) &= T\, f(w, y) \;. \end{aligned}$$

Clearly, we have $f(x, y) = T^{L(x)+1}(y)$ and so $f(x, x) = 0$. We are not able to derive $Z(x) = f(x, x)$ yet but we can define $Z(x) = f\, g(x)$ by composition where $g(x) = T(x), T(x)$ is primitive pair recursive by pairing. We have

$$Z(x) = f(T(x), T(x)) = T^{L\,T(x)+1} T(x) = 0 \ .$$

We derive the function f as primitive pair recursive by unary pair recursion from T and h where the auxiliary function h, such that $h(v, w, y, f(v, y), f(w, y)) = T\,f(w, y)$ holds, has a primitive pair recursive derivation by four compositions to satisfy: $h(x) = T\,T\,T\,T\,T(x)$.

For the functions $\langle D\rangle$ and I we need a primitive pair recursive derivation of the first projection function H for which we can easily see that the following holds:

$$\begin{aligned} H(0) &= 0 \\ H(0, y) &= Z(y) \\ H((v, w), y) &= H(v, y), T(w, y) \ . \end{aligned}$$

This constitutes a derivation of H by unary pair recursion from Z and an auxiliary function h satisfying:

$$h(v, w, y, H(v, y), H(w, y)) = H(v, y), H(w, y) \ .$$

The function h is primitive pair recursive by two compositions: $h(x) = T\,T\,T(x)$.

We have seen in the proof of Thm. 4.4.2 that the function $\langle D\rangle$ satisfies the following:

$$\begin{aligned} \langle D\rangle(0) &= 0 \\ \langle D\rangle(0, y) &= T(y) \\ \langle D\rangle((v, w), y) &= H(y) \ . \end{aligned}$$

This constitutes a primitive pair recursive derivation of $\langle D\rangle$ by pair recursion from T and an auxiliary function h satisfying:

$$h(v, w, y, \langle D\rangle(v, y), \langle D\rangle(w, y)) = H(y) \ .$$

The function h is a primitive pair recursive function by three compositions: $h(x) = H\,H\,T\,T(x)$.

The identity function $I(x) = x$ has the obvious property

$$I(x) = \langle D\rangle(x, (H(x), T(x)), Z(x))$$

but we cannot turn it directly into a primitive pair recursive derivation by composition and pairing because we do not know what to the with the first occurrence of the variable x in the argument of $\langle D\rangle$. If, however, we succeeded to derive the *signum* function, which is the characteristic function of the unary predicate $x > 0$, as a primitive pair recursive function then we could derive I as a primitive pair recursive function from the property

$$I(x) = \langle D\rangle((x >_* 0), (H(x), T(x)), Z(x))$$

by compositions and pairings. The signum function is derived from an auxiliary function primitive pair recursive function g such that $g(x) = 1$ holds. The function g is defined by pairing: $g(x) = Z(x), Z(x)$. The signum function is now defined as a primitive pair recursive function by unary pair recursion from g and g:

$$\begin{aligned}(0{>_*}0) &= 0\\ ((0,y){>_*}\,0) &= g(y)\\ (((v,w),y){>_*}0) &= g(v,w,y,((v,y){>_*}0),((w,y){>_*}0))\ .\end{aligned}$$ ■

4.4.5 Unary pair recursive definitions. In order to obtain the comfort of clausal definitions by unary pair recursion we consider the following kind of recursive definitions.

A recursive definition $f(x) = \tau[f;x]$ from (total) functions of a unary function is a *unary pair recursive definition* if for some numbers i and n such that $1 \le i \le n$ we have

$$\phi[f;x] \to \exists x_1 \exists x_2 \cdots \exists x_{n-1} \exists x_n\,(x = x_1, x_2, \ldots, x_i, \ldots, x_{n-1}, x_n \wedge x_i > 0)$$

for every occurrence in τ of a recursive application f governed by ϕ and the application can be written as

$$f(H\,T^0(x), H\,T^1(x) \ldots, p\,H\,T^{i-1}(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x))$$

when $i < n$ and the form $f(H\,T^0(x), H\,T^1(x) \ldots, H\,T^{n-2}(x), p\,T^{n-1}(x))$ when $i = k$ and where p stands for H or T. The definition is clearly regular in the measure $m(x) = |H\,T^{i-1}(x)|$ if $i < n$ and in $m(x) = |T^{n-1}(x)|$ if $i = n$. In either case we say that the definition is regular in *the i-th pair component*.

We do not count the functions D, (x,y), H, and T if applied in τ towards the arguments of unary pair recursive definitions.

4.4.6 Theorem *Primitive pair recursive functions are closed under unary pair recursive definitions.*

Proof. Assume that if f is defined by unary pair recursive definition $f(x) = \tau[f;x]$. We define two auxiliary primitive pair recursive functions by simple definitions:

$g(y) = \tau[\dot{\lambda} x.0;0]$
$h(v,w,y,a,b) = \tau_1[a,b,(v,w)]$

where the term $\tau_1[a,b,x]$ is obtained from τ by replacing every recursive application of f with the variable a or b depending on whether the function variable p discussed in Par. 4.4.5 stands for H or T. We then define a pair recursive function f_1 by pair recursion from g and h. We can prove by pair induction on x similarly as in Thm. 4.1.8 that we have $f(x) = f_1(x,0)$ and so the last explicitly derives f as a primitive pair recursive function. ■

4.4.7 Contraction of list concatenation $\langle\oplus\rangle$ is a primitive pair recursive function. The contraction of the list concatenation function $\langle\oplus\rangle$, where we abbreviate $\langle\oplus\rangle(\tau_1,\tau_2)$ to $\tau_1 \langle\oplus\rangle \tau_2$, is a primitive pair recursive function by unary pair recursive definition regular in the first pair component:

$$0 \langle\oplus\rangle y = y$$
$$(v,w) \langle\oplus\rangle y = v, w \langle\oplus\rangle y \ .$$

This kind of recursive definition is called a *unary list recursive definition* because we do not recur to the head of the recursive component.

4.4.8 Right and left trees. The numbers x in the range of the function σ, i.e. such that we have $\exists n\, \sigma(n) = x$ are called *right trees* because they 'lean' to the right as we have $(x)_i = 0$, or equivalently $H\,T^i(x) = 0$, for all $i < n$.

The unary function $Rt(x)$ taking a number x into the right tree of the same pair size satisfies:

$$Rt(x) = \sigma\,|x| \ .$$

Right trees $Rt(x)$ are the least numbers of pair size $|x|$ because from 1.3.11(5) we have for all numbers x:

$$Rt(x) \le x < 0, Rt(x) \ . \tag{1}$$

Since we have

$$L\,Rt(x) = |x| \ , \tag{2}$$

we can derive Rt as a primitive pair recursive function by the following unary pair recursive definition regular in its argument:

$$Rt(0) = 0$$
$$Rt(v,w) \leftarrow 0, Rt(v) \oplus Rt(w) \ .$$

We can see from (1) that the numbers $(0, Rt(x)) - 1$ are the greatest numbers of the pair size $|x|$ and they are called *left trees* because they 'lean' to the left as we have $T\,H^i(x) = 0$ for all $i < |x|$. The unary predicate Le of being a left tree is primitive pair recursive by the following unary pair recursive definition regular in its argument:

$$Le(0)$$
$$Le(x,0) \leftarrow Le(x) \ .$$

4.4.9 The successor function is primitive pair recursive. The successor function $x+1$ is derived as a primitive pair recursive function by the following unary pair recursive definition regular in its argument:

$$0 + 1 = 1$$
$$(v,w) + 1 = v, w + 1 \leftarrow \neg Le(w)$$
$$(v,w) + 1 = v + 1, Rt(w) \leftarrow Le(w) \land \neg Le(v)$$
$$(v,w) + 1 = (0, Rt(v)), Rt(w_1) \leftarrow Le(w) \land Le(v) \land w = w_1, 0$$
$$(v,w) + 1 = 0, Rt(v,w) \leftarrow Le(w) \land Le(v) \land w = 0$$

which performs the case analysis suggested by the ordering properties of the pairing function (see Par. 1.3.10).

4.4.10 Theorem *Primitive pair recursive functions are closed under unary length iteration.*

Proof. If f is defined by unary length iteration of a primitive pair recursive g we derive f in the same class by unary list recursion:

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

4.4.11 Contraction of identity $\langle = \rangle$ is a primitive pair recursive predicate. Consider the following simple definition of the primitive pair recursive function *Eq*:

$$\begin{aligned}&Eq(0,0) = 0,0\\&Eq(0,(0,0),s) = 0,s\\&Eq(0,((v,w),(v_1,w_1),s) = 0,(v,v_1),(w,w_1),s\ .\end{aligned}$$

The first pair component is a dummy one and the second one is a stack on which we store pairs of numbers. The idea is that a sufficiently long iteration of *Eq* with two numbers x and y as the only pair on the stack: $Eq^i(0,(x,y),0)$ yields eventually $0,0$ iff $x = y$ and 0 otherwise. As each iteration either removes two zeros from the stack, splits two pairs on its top, or else yields 0 by default, the maximum number of iterations exhausting the stack cannot be larger than the total number of zeros and commas in the argument x, i.e. more than

$$2{\cdot}|x| + 1 = |x,x| \overset{4.4.8(2)}{=} L\,Rt(x,x)\ .$$

We can thus derive $\langle = \rangle$ as a primitive pair recursive predicate by a simple definition from the unary length iteration of *Eq*:

$$x \langle = \rangle y \leftarrow Eq^{L\,Rt(x,x)}(0,(x,y),0) = v,w\ .$$

The reader will note that discrimination on identity tests $x = y$ is now primitively pair recursively foldable.

4.4.12 Exponentiation of length. We will reduce in the following lemma the operator of unary primitive recursion (see Par. 4.1.1) to unary length iteration. For that it is sufficient to obtain a primitive pair recursive bound b such that $x \le L\,b(x)$ holds.

Consider the primitive pair recursive function *Exp* defined by unary list recursive definition regular in the argument:

$$\begin{aligned}&Exp(0) = 0,0\\&Exp(v,w) = Exp(w) \oplus Exp(w)\ .\end{aligned}$$

By straightforward pair induction on x we can show that

$$L\,Exp(x) = 2^{L(x)} \tag{1}$$

holds and so we have

$$x \overset{1.3.12(4)}{<} 4^{|x|} = 2^{2\cdot|x|} \overset{4.4.8(2)}{=} 2^{2\cdot L\,Rt(x)} \overset{1.4.3(4)}{=}$$
$$2^{L(Rt(x)\oplus Rt(x))} \overset{(1)}{=} L\,Exp(Rt(x)\oplus Rt(x))\ . \tag{2}$$

4.4.13 Lemma *Primitive pair recursive functions are closed under unary primitive recursion.*

Proof. Suppose that f is defined by unary primitive recursion from unary primitive pair recursive functions g and h and consider the simple definition of a primitive pair recursive function f_1:

$$f_1(x,y,i,a) = x,y,i+1,h(i,y,a) \leftarrow x \neq i$$
$$f_1(x,y,i,a) = x,y,i,a \leftarrow x = i\ .$$

By induction on i we now prove

$$i \leq x \rightarrow f_1^i(x,y,0,g(y)) = x,y,i,f(i,y)\ .$$

In the base case we have

$$f_1^0(x,y,0,g(y)) = x,y,0,g(y) = x,y,0,f(0,y)\ .$$

In the inductive case we assume $i+1 \leq x$ and we have:

$$f_1^{i+1}(x,y,0,g(y)) = f_1\,f_1^i(x,y,0,g(y)) \overset{IH}{=} f_1(x,y,i,f(i,y)) \overset{x\neq i}{=}$$
$$x,y,i+1,h(i,y,f(i,y)) = x,y,i+1,f(i+1,y)\ .$$

because of the second clause of f_1 we clearly have

$$f_1^{x+i}(x,y,0,g(y)) = x,y,x,f(x,y)\ .$$

Using 4.4.12(2) we can thus derive the function f as a primitive pair recursive function by a simple definition from the unary length iteration of f_1:

$$f(x,y) = a \leftarrow f_1^{Exp(Rt(x)\oplus Rt(x))}(x,y,0,g(y)) = x_1,y_1,i,a\ . \qquad \blacksquare$$

4.4.14 Lemma *Contraction $\langle f\rangle$ of every primitive recursive function f is a primitive pair recursive function.*

Proof. By induction on the construction of primitive recursive functions f. The functions Z and S are their own contractions and they are primitive pair recursive functions by the simple definition $Z(x) = 0$ and by Par. 4.4.9. If $f = I_i^n$ we define $\langle I_i^n\rangle$ as a primitive pair recursive function by simple definition:

$$\langle I_i^n\rangle(x_1,\dots,x_i,\dots,x_n) = x_i \quad .$$

If the n-ary function f is obtained by composition from m-ary g, and n-ary h_1, ..., h_m then we use IH and define $\langle f\rangle$ a primitive pair recursive function by simple definition:

$$\langle f\rangle(\vec{x}) = \langle g\rangle(\langle h_1\rangle(\vec{x}),\cdots,\langle h_m\rangle(\vec{x})) \ .$$

If the $(n+1)$-ary function $f(x,\vec{y})$ is obtained by primitive recursion from g and h we ise IH and define an auxiliary primitive pair recursive function h by a simple definition:

$$h_1(x,(\vec{y}),v) = \langle h\rangle(x,\vec{y},v)$$

We then use Lemma 4.4.13 and define another auxiliary primitive pair recursive function f_1 by unary primitive recursion:

$$\begin{aligned} f_1(0) &= 0 \\ f_1(0,y) &= \langle g\rangle(y) \\ f_1(x+1,y) &= h_1(x,y,f_1(x,y)) \ . \end{aligned}$$

The function $\langle f\rangle$ is primitive pair recursive by simple definition:

$$\langle f\rangle(x,\vec{y}) = f_1(x,\vec{y}) \ . \qquad \blacksquare$$

4.4.15 Theorem (Characterization of unary primitive recursive functions) *Unary primitive recursive functions are exactly the primitive pair recursive functions.*

Proof. If f is an unary primitive recursive function f then, since $f = \langle f\rangle$, the function f is primitive pair recursive by Lemma 4.4.14.

Vice versa, primitive pair recursive functions are primitive recursive because the latter functions are primitevely pair recursively closed. Note that the closure under unary pair recursion follows from Thm. 4.3.5 because the operator defines functions by recursive definitions regular in H. $\blacksquare$

Chapter 5

Characterization of Recursive Functions

In Sect. 5.1 we negatively chacterize the class of recursive functions as being strictly larger than the class of primitive recursive functions and in Sect. 5.2 we positively characterize the same class by μ-recursion.

5.1 Beyond Primitive Recursion

For long time it was assumed that the computable functions coincide with primitive recursive functions. In 1928 W. Ackermann [1] 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 [13] in 1935 to the function A given in Par. 1.1.24. In the following we give essentially the Ackermann-Péter argument which proceeds by showing that the unary functions A_n such that $A_n(x) = A(n, x)$ increase so fast that they eventually majorize every primitive recursive function.

We will also show that the universal function for primitive recursive functions is a recursive but not a primitive recursive function.

Ackermann-Péter function A grows too fast

5.1.1 Increasing functions. A unary function f is *increasing* if

$$x < y \to f(x) < f(y)$$

holds. For increasing functions we have:

$$x \leq f(x) \tag{1}$$

$$0 < f(0) \to x < f(x) \; . \tag{2}$$

Property (1) is proved by induction on x. The base case is trivial. For the inductive case we get from IH: $x \leq f(x) < f(x+1)$ and so $x+1 \leq f(x+1)$. For the property (1) 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:

$$x < y \rightarrow f^z(x) < f^z(y) \tag{3}$$
$$0 < f(0) \rightarrow x < f^{y+1}(x) \tag{4}$$
$$0 < f(0) \wedge x < y \rightarrow f^x(z) < f^y(z) \ . \tag{5}$$

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

$$f^x(z) \overset{(4)}{<} f^{y-x} f^x(z) = f^y(z) \ .$$

5.1.2 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. 5.1.5.

5.1.3 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

$$0 < 1 \overset{\text{outer IH}}{<} A_n(1) \overset{5.1.2(1)}{=} A_{n+1}(0) \ .$$

In the inductive case we have

$$x+1 \overset{\text{inner IH}}{<} A_{n+1}(x)+1 \overset{\text{outer IH}}{\leq} A_n A_{n+1}(x) \overset{5.1.2(2)}{=} A_{n+1}(x+1) \ .$$

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:

$$A_{n+1}(x) = A_n^{x+1}(1) \overset{5.1.1(5)}{<} A_n^{y+1}(1) = A_{n+1}(y) \ .$$

Property (2) is proved by induction x. In the base case we have $A_n(1) \overset{5.1.2(1)}{=} A_{n+1}(0)$. In the inductive case we have $x+2 \overset{(4)}{\leq} A_n(x+1) \overset{IH}{\leq} A_{n+1}(x)$ and so

$$A_n(x+2) \overset{(5)}{\leq} A_n A_{n+1}(x) \overset{5.1.2(2)}{=} A_{n+1}(x+1) \ .$$

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 $A_n(0) \overset{(5)}{<} A_n(1) \overset{5.1.2(1)}{=} A_{n+1}(0)$. 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

$$A_n(y+1) \overset{(5)}{<} A_n A_{n+1}(y) \overset{5.1.2(2)}{=} A_{n+1}(y+1) \ .$$

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:

$$A_{n_1}^{k_1}(x) \overset{5.1.1(3)}{\leq} A_{n_1}^{k_1}(y) \overset{5.1.1(5)}{\leq} A_{n_1}^{k_2}(y) \overset{(8)}{\leq} A_{n_2}^{k_2}(y) \ .$$

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 $A_n^0(x) \overset{(1)}{<} A_{n+1}^1(x) = A_{n+1}(0 + x)$. In the inductive case we have

$$A_n^{k+1}(x) = A_n A_n^k(x) \overset{\text{IH},(1)}{<} A_n A_{n+1}(k + x) \overset{5.1.2(2)}{=} A_{n+1}(k + 1 + x) \ . \qquad \blacksquare$$

5.1.4 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 $f(\vec{x}) \leq h^k \max(\vec{x})$.

5.1.5 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 $\exists n \exists k \forall \vec{x}\, f(\vec{x}) \leq A_n^k \max(\vec{x})$. For the initial functions we have:

$$S(x) = A_0(x) = A_0^1 \max(x)$$
$$Z(x) = 0 \leq x = A_0^0(x)$$
$$I_i^m(\vec{x}) = x_i \leq \max(\vec{x}) = A_0^0 \max(\vec{x}) \ .$$

If f is an m-ary function derived by composition from the primitive recursive functions g, h_1, ... , h_p where g is p-ary then we get from IH the numbers n_0, n_1, ... , n_p as well as k_0, k_1, ... , 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:

$$g(\vec{y}) \overset{\text{IH}}{\leq} A_{n_0}^{k_0} \max(\vec{y}) \overset{5.1.3(1)}{\leq} A_n^{k_0} \max(\vec{y}) \tag{1}$$
$$h_i(\vec{x}) \overset{\text{IH}}{\leq} A_{n_i}^{k_i} \max(\vec{x}) \overset{5.1.3(1)}{\leq} A_n^{k'} \max(\vec{x}) \ . \tag{2}$$

We then have

$$f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_p(\vec{x})) \overset{(1)}{\leq} A_n^{k_0} \max(h_1(\vec{x}), \ldots, h_p(\vec{x})) \overset{(2)5.1.3(1)}{\leq}$$
$$A_n^{k_0} A_n^{k'} \max(\vec{x}) = A_n^k \max(\vec{x}) \ .$$

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:

$$f(0, \vec{y}) = g(\vec{y}) \overset{(3)}{\leq} A_{n_1}^{k_1} \max(\vec{y}) \overset{5.1.3(1)}{\leq} A_{\max(n_1,n_2)}^{k_2 \cdot 0 + k_1} \max(\vec{y}) \ .$$

In the inductive case we note that, since

$$x + \max(\vec{y}) = A_0^x \max(\vec{y}) \overset{5.1.3(1)}{\leq} A_{\max(n_1,n_2)}^{k_2 \cdot x + k_1} \max(\vec{y}) \tag{6}$$

holds, we obtain

$$f(x+1, \vec{y}) = h(x, \vec{y}, f(x, \vec{y})) \overset{(4)}{\leq} A_{n_2}^{k_2} \max(x, \vec{y}, f(x, \vec{y})) \overset{5.1.3(1),\text{IH}}{\leq}$$
$$A_{n_2}^{k_2} \max(x, \vec{y}, A_{\max(n_1,n_2)}^{k_2 \cdot x + k_1} \max(\vec{y})) \overset{(6),5.1.3(1)}{\leq}$$
$$A_{n_2}^{k_2} A_{\max(n_1,n_2)}^{k_2 \cdot x + k_1} \max(\vec{y}) \overset{5.1.3(1)}{\leq}$$
$$A_{\max(n_1,n_2)}^{k_2} A_{\max(n_1,n_2)}^{k_2 \cdot x + k_1} \max(\vec{y}) = A_{\max(n_1,n_2)}^{k_2 \cdot (x+1) + k_1} \max(\vec{y}) \ .$$

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:

$$f(x, \vec{y}) \overset{(5)}{\leq} A_{\max(n_1,n_2)}^{k_2 \cdot x + k_1} \max(\vec{y}) = A_{\max(n_1,n_2)}^{k_1} A_{\max(n_1,n_2)}^{k_2 \cdot x} \max(\vec{y}) \overset{5.1.3(3)}{\leq}$$
$$A_{\max(n_1,n_2)}^{k_1} A_{\max(n_1,n_2)+1}(k_2 \cdot x + \max(\vec{y})) \overset{5.1.3(1)}{\leq}$$
$$A_n^{k_1} A_n(k_2 \cdot x + \max(\vec{y})) \overset{(7)}{\leq} A_n^{k_1} A_n A_2^{k_3} \max(x, \vec{y}) \overset{5.1.3(1)}{\leq}$$
$$A_n^{k_1 + 1 + k_3} \max(x, \vec{y}) = A_n^k \max(x, \vec{y}) \ .$$

■

5.1.6 The Ackermann-Péter function A is not primitive recursive. The Ackermann-Péter function A was defined in Par. 1.1.24 by the following recursive definition:

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

which is regular in the ordinal measure $m(x,y) = \omega{\cdot}x+y$ because either the first argument decreases or it stays the same and the second one decreases. Thus A is a recursive function and so it is effectively computable. 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:

$$\forall x\, A(n,x) = A_n(x)\ . \tag{1}$$

This is proved by induction on n. In the base case we have $A(0,x) = x+1 = A_0(x)$. In the inductive case we proceed by induction on x. In the base case we have:

$$A(n+1,0) = A(n,1) \overset{\text{outer IH}}{=} A_n(1) \overset{5.1.2(1)}{=} A_{n+1}(0)\ .$$

In the inductive case we have:

$$A(n+1,x+1) = A(n,A(n+1,x)) \overset{\text{IH's}}{=} A_nA_{n+1}(x) \overset{5.1.2(2)}{=} A_{n+1}(x+1)\ .$$

Suppose now that A is a primitive recursive function. For some m and k we then have

$$
\begin{aligned}
A(n,x) \overset{5.1.5}{\le} A_m^k \max(n,x) &\overset{5.1.3(3)}{<} A_{m+1}(k+\max(n,x)) \overset{5.1.3(1)}{\le}\\
&A_{m+1}(2{\cdot}\max(k,n,x)) \overset{5.1.3(1),5.1.2(4)}{<} A_{m+1}A_2\max(k,n,x) \overset{5.1.3(1)}{\le}\\
&A_{m+1}A_{m+2}\max(k,n,x) \overset{5.1.2(2)}{=} A_{m+2}(\max(k,n,x)+1) \overset{5.1.3(2)}{\le}\\
&A_{m+3}\max(k,n,x)\ .
\end{aligned} \tag{2}
$$

We now get a contradiction for $n = m+3$ and $x \ge \max(k,n)$:

$$A_{m+3}(x) = A_n(x) \overset{(1)}{=} A(n,x) \overset{(2)}{<} A_{m+3}\max(k,n,x) = A_{m+3}(x)\ .$$

Enumeration function of primitive recursive functions is not primitive recursive

We will define in Par. 5.1.8 a recursive function enumerating the primitive recursive functions and show that is is not primitive recursive.

5.1.7 Universal function for unary primitive recursive functions. We use the characterization by Thm. 4.4.15 of unary primitive recursive functions as primitive pair recursive functions and assign a code to every such function. For that we define the following constructors: $Tx = 0,0$; $Cx(a,b) = 1,a,b$; $Px(a,b) = 2,a,b$; and $Rx(a,b) = 2,a,b$. The codes are assigned to satisfy:

$$\begin{array}{ll}
\ulcorner T \urcorner = Tx & \\
\ulcorner f \urcorner = Cx(\ulcorner g \urcorner, \ulcorner h \urcorner) & \text{if } f(x) = g\, h(x) \\
\ulcorner f \urcorner = Px(\ulcorner g \urcorner, \ulcorner h \urcorner) & \text{if } f(x) = g(x), h(x) \\
\ulcorner f \urcorner = Rx(\ulcorner g \urcorner, \ulcorner h \urcorner) & \text{if } f \text{ is def. by unary pair. rec. from } g \text{ and } h\ .
\end{array}$$

We define the binary function U by the following recursive definition:

$$\begin{array}{l}
U(Tx, x) = T(x) \\
U(Cx(g,h), x) = U(g, U(h,x)) \\
U(Px(g,h), x) = U(g,x), U(h,x) \\
U(Rx(g,h), 0) = 0 \\
U(Rx(g,h), 0, y) = U(g,y) \\
U(Rx(g,h), (v,w), y) = U(h, v, w, y, U(Rx(g,h), v, y), U(Rx(g,h), w, y))\ \ .
\end{array}$$

The definition is regular in the ordinal measure $m(c,x) = \omega \cdot c + |H(x)| < \omega^2$ because in all recursive applications except the two nested ones in the last clause the first argument goes down. In the nested applications of the last clause the first argument stays the same and the head of the second argument v,w goes down in the pair size to v or w. The function U is thus a recursive function and it is called a *universal* function for unary primitive recursive functions because its main property is:

$$\forall x\, U(\ulcorner f \urcorner, x) = f(x) \tag{1}$$

for all unary primitive recursive functions f. This is proved by induction on the construction of the primitive pair recursive function f. If $f = T$ the property follows from the first clause. If $f(x) = g\, h(x)$ we have

$$\begin{aligned}
U(\ulcorner f \urcorner, x) = U(Cx(\ulcorner g \urcorner, \ulcorner h \urcorner), x) = U(\ulcorner g \urcorner, U(\ulcorner h \urcorner, x)) \stackrel{IH}{=} \\
U(\ulcorner g \urcorner, h(x)) \stackrel{IH}{=} g\, h(x)\ .
\end{aligned}$$

The reasoning for $f(x) = g(x), h(x)$ is similar.

If f is defined by unary pair recursion from g and h then $\ulcorner f \urcorner = Rx(\ulcorner g \urcorner, \ulcorner h \urcorner)$ and we prove

$$U(Rx(\ulcorner g \urcorner, \ulcorner h \urcorner), x) = f(x)$$

by induction on $|H(x)|$. We consider three cases. If $x = 0$ we have

$$U(Rx(\ulcorner g \urcorner, \ulcorner h \urcorner), 0) = 0 = f(0)\ .$$

If $x = 0, y$ we have

$$U(Rx(\ulcorner g\urcorner, \ulcorner h\urcorner), 0, y) = U(\ulcorner g\urcorner, y) \overset{\text{outer IH}}{=} g(y) = f(0, y) \ .$$

If $x = (v, w), y$ we have

$$\begin{aligned}
&U(Rx(\ulcorner g\urcorner, \ulcorner h\urcorner), (v, w), y) = \\
&\quad U(\ulcorner h\urcorner, v, w, y, U(Rx(\ulcorner g\urcorner, \ulcorner h\urcorner), v, y), U(Rx(\ulcorner g\urcorner, \ulcorner h\urcorner), w, y)) \overset{\text{inner IH's}}{=} \\
&\quad U(\ulcorner h\urcorner, v, w, y, f(v, y), f(w, y)) \overset{\text{outer } IH}{=} \\
&\quad h(v, w, y, f(v, y), f(w, y)) = f((v, w), y) \ .
\end{aligned}$$

5.1.8 Recursive enumeration of primitive recursive functions. We claim that the recursive function U enumerates the primitive recursive functions. For that we explicitly define the unary function $U_e(x) = U(e, x)$ for every e. To an n-ary primitive recursive function f we assign an index $e = \ulcorner\langle f\rangle\urcorner$ because we have $U_e = f$:

$$U_e(x) = U(e, \vec{x}) = U(\ulcorner\langle f\rangle\urcorner, \vec{x})) \overset{5.1.7(1)}{=} \langle f\rangle(\vec{x}) = f(\vec{x}) \ .$$

On the other hand we prove that U_e are primitive recursive functions by complete induction on e. If $e = Tx$ we have an explicit derivation $U_e(x) = T(x)$. If $e = Cx(g, h)$ for some g and h the functions U_g and U_h are primitive recursive by IH and we explicitly derive $U_e(x) = U_g U_h(x)$. The case when $e = Px(g, h)$ is similar. If $e = Rx(g, h)$ for some g and h the functions U_g and U_h are primitive recursive by IH and we derive U_e by pair recursion:

$$\begin{aligned}
&U_e(0, y) = U_g(y) \\
&U_e((v, w), y) = U_h(v, w, y, U_e(v, y), U_e(w, y)) \ .
\end{aligned}$$

If neither of the above cases applies we have an explicit derivation $U_e(x) = 0$.

Now, suppose that U is a primitive recursive function. Then also the explicitly defined unary function $f(x) = U(x, x) + 1$ is primitive recursive and we obtain contradiction by

$$f(\ulcorner f\urcorner) \overset{\text{def.}}{=} U(\ulcorner f\urcorner, \ulcorner f\urcorner) + 1 \overset{5.1.7(1)}{=} f(\ulcorner f\urcorner) + 1 \ .$$

We thus have in the enumerating function U a recursive function which is not primitive recursive.

5.2 μ-Recursive Functions

Our introduction of recursive functions by explicit and recursive definitions in the style of Herbrand-Gödel is a meta-mathematical (see Par. 3.1.7). Mathematicians prefer inductive characterizations with minimal reliance on a language. The characterization of Kleene [11, 12] by μ-recursive functions is such.

We can define the class μ-recursive of functions in two ways. An easy way is to assume primitive recursion and define μ-recursive functions as generated from I_i^n, S, and Z by composition, primitive recursion, and regular minimalization. This approach makes the initial development of μ-recursive functions easier and it is used by many authors (see, for instance, [3, 7, 10, 12]).

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, [21]) 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 here, but it is possible to skip our development of μ-recursive functions and take Thm. 5.2.15 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. 5.2.14). 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, [21]) 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, [3, 23]).

We have chosen to code in the proof of Thm. 5.2.14 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 [22] (see also [9, 23]) and was positively solved by Bennett [2] 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 Par. 5.2.13 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 [17] and define the addition function from the successor and multiplication functions.

5.2.1 The operator of minimalization. For an $(n+1)$-ary (total) function g we write $\mu_y[g(y,\vec{x})=1]$ for the n-ary partial function in $\vec{x}$ obtained by the *minimalization of* g and such that it yields the least number y such that $g(y,\vec{x})=1$ holds and it is undefined otherwise. In other words, the graph of $\mu_y[g(y,\vec{x})=1]$ satisfies:

$$\mu_y[g(y,\vec{x})=1] \asymp y \leftrightarrow g(y,\vec{x})=1 \land \forall z<y\, g(z,\vec{x}) \neq 1 \; .$$

The minimalization is *regular* if the partial function is total, i.e. if $\forall\vec{x}\exists y\, g(y,\vec{x})=1$ holds.

In function classes closed under explicit definitions of functions we will adapt the notation for bounded minimalization (see Par. 3.3.4) and admit terms in the minimalization notation:

$$f(\vec{x}) \simeq \mu_y[\tau[y,\vec{x}]=1]$$

as abbreviation for $f(\vec{x}) \simeq \mu_y[g(y,\vec{x})=1]$ where g has an explicit definition: $g(y,\vec{x})=\tau[y,\vec{x}]$.

If a class of functions is closed under explicit definitions and contains a predicate R we can admit the following minimalization notation:

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

as abbreviation for $f(\vec{x}) \simeq \mu_y[R_*(\vec{\tau})=1]$.

In function classes closed under both explicit definitions of functions and predicates we can admit formulas in the minimalization notation:

$$f(\vec{x}) \simeq \mu_y[\phi[y,\vec{x}]]$$

as abbreviation for $f(\vec{x}) \simeq \mu_y[R_*(y,\vec{x})=1]$ where R has an explicit definition $R(y,\vec{x}) \leftrightarrow \phi[y,\vec{x}]$.

The following theorem characterizes recursive functions as those obtainable from primitive recursive functions by one regular minimalization.

5.2.2 Theorem (Non-uniform normal form of (partial) recursive functions) *For every class of functions $\mathcal{F}$, every partial $\mathcal{F}$-recursive function f, and any approximation function f^+ for f we have*

$$f(x_1,\ldots,x_n) \simeq T\,\mu_v[f^+(H(v),x_1{+}1,\ldots,x_n{+}1)=T(v){+}1] \; .$$

If f is total then the minimalization is regular.

Proof. By Corollary 2.3.15 we have $f(x_1,\ldots,x_n) \asymp y$ iff $f^+(z,x_1{+}1,\ldots,x_n{+}1) = y{+}1$ for some z iff, there is a v such that $f^+(H(v),x_1{+}1,\ldots,x_n{+}1)=T(v){+}1$ and $y=T(v)$. If f is total then there is such v for every $\vec{x}$. ∎

5.2.3 μ-Recursive functions. The class of *partial μ-recursive functions* is generated from the identity functions $I_i^n(\vec{x}) = x_i$, multiplication function $x{\cdot}y$, and from the characteristic function $x <_* y$ of the *less than* predicate by composition and minimalization. A *μ-recursive function* is a partial μ-recursive function which is total.

By Thm. 3.2.1 the μ-recursively closed classes are thus closed under explicit definitions of partial functions without constants.

5.2.4 Successor function $x+1$ is μ-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-recursive function by regular minimalization:

$$x + 1 = \mu_y[x < y] \ .$$

5.2.5 Constant functions are μ-recursive. We 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-recursive function by regular minimalization:

$$Z(x) = \mu_y[x < x + 1] \ .$$

We can now define all constant functions $C_n(x) = n$ as μ-recursive functions by a series of explicit definitions:

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

By Thm. 3.2.2 the μ-Recursively closed classes are thus closed under explicit definitions of partial functions.

5.2.6 Boolean functions are μ-recursive. The *boolean negation function* $\neg_*(x)$ is such that $\neg_*(0) = 1$ and $\neg_*(1) = 0$. The boolean *conjunction* $x \wedge_* y$ and *disjunction* $x \vee_* y$ functions have the obvious truth table properties. The three boolean functions are μ-recursive by explicit definitions:

$$\begin{aligned}
&\neg_*(x) = x <_* 1 \\
&x \wedge_* y = x{\cdot}y \\
&x \vee_* y = \neg_*(\neg_*(x) \wedge_* \neg_*(y)) \ .
\end{aligned}$$

5.2.7 Predicates $x \leq y$ and $x = y$ are μ-recursive. The binary predicates $x \leq y$ and $x = y$ are μ-recursive by explicit definitions of their characteristic functions:

$$\begin{aligned}
&x \leq_* y = \neg_*(y <_* x) \\
&x =_* y = (x \leq_* y) \wedge_* (y \leq_* x) \ .
\end{aligned}$$

5.2.8 Discrimination function D is μ-recursive. The graph of the discrimination function D has the following obvious explicit definition:

$$D(x, y, z) = v \leftrightarrow x = 0 \wedge v = z \vee 0 < x \wedge v = y \ .$$

We clearly have $\forall x \forall y \forall z \exists v\, D(x,y,z) = v$ and so we define the function D as a μ-recursive function by regular minimalization:

$$D(x,y,z) = \mu_v[((x =_* 0) \wedge_* (v =_* z) \vee_* (0 <_* x) \wedge_* (v =_* y)) = 1] \; .$$

By Thm. 3.3.3 the μ-Recursively closed classes are now closed under explicit definitions of predicates without bounded quantification. Our next task is to show the closure under bounded minimalization and bounded quantification.

5.2.9 Theorem *Any μ-recursively closed class of partial functions $\mathcal{F}$ is closed under bounded minimalization and under explicit definitions of predicates with bounded quantification.*

Proof. Let the $(n+1)$-ary function f be defined by bounded minimalization:

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

from a function $g \in \mathcal{F}$. We clearly have

$$\forall x \forall \vec{y} \exists z\, (z \le x \wedge g(z,\vec{y}) = 1 \vee z > x)$$

and so the auxiliary $(n+1)$-ary function $f_1 \in \mathcal{F}$ is defined by regular minimalization:

$$f_1(x,\vec{y}) = \mu_z[z \le x \wedge g(z,\vec{y}) = 1 \vee z > x] \; .$$

Note that $f_1(x,\vec{y})$ yields the smallest $z \le x$ such that $g(z,\vec{y}) = 1$ or $x+1$ if there is none. We now define $f \in \mathcal{F}$ by explicit definition:

$$\begin{aligned} &f(x,\vec{y}) = z \leftarrow f_1(x,\vec{y}) = z \wedge z \le x \\ &f(x,\vec{y}) = 0 \leftarrow f_1(x,\vec{y}) = z \wedge z > x \; . \end{aligned}$$

The closure of $\mathcal{F}$ under explicit definitions of predicates with bounded quantification now follows from Thm. 3.3.5. ■

5.2.10 Addition $x+y$ is μ-recursive. We use the following observation by Joan Robinson [17]. If $z > 0$ then we have

$$\begin{aligned} x + y = z \Leftrightarrow (x+y)\cdot z + x\cdot y\cdot z^2 + 1 = z^2 + x\cdot y\cdot z^2 + 1 \Leftrightarrow \\ (x\cdot z + 1)\cdot(y\cdot z + 1) = (x\cdot y + 1)\cdot z^2 + 1 \; . \end{aligned}$$

The addition function can be thus derived as a μ-recursive function by regular minimalization of its graph:

$$\begin{aligned} x + y = \mu_z[&z = 0 \wedge x = 0 \wedge y = 0 \vee \\ &z > 0 \wedge (x\cdot z + 1)\cdot(y\cdot z + 1) = (x\cdot y + 1)\cdot z\cdot z + 1] \; . \end{aligned}$$

5.2.11 Predicate $x \mid y$ of divisibility is μ-recursive. The binary predicate $x \mid y$ of *divisibility* holding if x divides y is μ-recursive by explicit definition:

$$x \mid y \leftrightarrow \exists z{\le}y\,(y = x{\cdot}z)\ .$$

5.2.12 The predicate $\exists n\, x = 2^n$ is μ-recursive. The unary predicate $\exists n\, p = 2^n$ holding of powers of two x is μ-recursive by explicit definition:

$$\exists n\, x = 2^n \leftrightarrow x > 0 \land \forall y{\le}x\,(y \mid x \to y = 1 \lor 2 \mid y)\ .$$

5.2.13 Exponentiation function 2^x is μ-recursive. Before we can introduce a μ-recursive exponentiation function 2^x we will need to derive its graph $2^x = y$ as a μ-recursive predicate. We will actually derive the graph as a Σ_0-predicate and for that we consider the derivation of the exponentiation function 2^x by recursion on notation:

$$\begin{aligned}
2^0 &= 1 \\
2^{2\cdot x+0} &= (2^x)^2 &\qquad& \text{if } x > 0 \\
2^{2\cdot x+1} &= 2{\cdot}(2^x)^2\ .
\end{aligned}$$

For a given x let us define the numbers $x_i = x \div 2^i$ for all $i = 0, 1, \ldots, l$ where 2^l is the smallest number such that $2^l > x$. Note that we have $x_0 = x$, $x_l = 0$, and for every $i < l$ we have $x_{i+1} = x_i \div 2$. The computation of 2^x goes through the recursive arguments x_0, x_1, ..., until for $x_l = 0$ we can compute $2^{x_l} = 1$ directly.

We first derive an auxiliary binary predicate $Prn(p, y)$ holding of the *predecessor p of y in recursion on notation*, i.e. such that for $y = 2^x$ we have $Prn(p, y)$ iff $p = 2^{x_j}$ for some $j \le l$. The predicate can be introduced with the help of course of values sequences $y + s$ satisfying

$$\exists j\,(p = 2^{x_j} \land y + s = \sum_{i \le j} 2^{x_i})$$

where $s < y$, the smallest power in the sum is $p = 2^{x_j}$, and for every two 'neighboring' powers $p_1 = 2^{x_i}$ and $p_2 = 2^{x_{i+1}}$ in the sum we have $p_1 = p_2^2$ or $p_1 = 2{\cdot}p_2^2$. This fact is used in the following μ-recursive derivation of the predicate Prn:

$$\begin{aligned}
&Prn(p, y) \leftrightarrow p \ge 1 \land \exists n\, y = 2^n \land \exists s{<}y \forall p_1{\le}y \forall a{<}y{+}s \forall b{<}p_1\,(\\
&\quad \exists n\, p_1 = 2^n \land y + s = a{\cdot}2{\cdot}p_1 + p_1 + b \to p_1 = p \land b = 0 \lor \\
&\qquad p_1 > p \land \exists p_2{\le}b \exists c{<}p_2\,(b = p_2 + c \land (p_1 = p_2{\cdot}p_2 \lor p_1 = 2{\cdot}p_2{\cdot}p_2))\,)\ .
\end{aligned}$$

With the help of Prn we can 'store' and recover the recursive arguments x_i needed in the computation of $y = 2^x$ in a course of values sequence: $x{\cdot}y + s = \sum_{i \le l} x_i{\cdot}2^{x_i}$ with $s < y$ (see Fig. 5.1). Every argument in the sequence $x{\cdot}y + s$

can be located at some power p for which $Prn(p,y)$ holds. The arguments x_i cannot run into each other because $x_i < 2^{x_i}$ and the next higher bit set in $x{\cdot}y+s$ cannot come sooner than at the power $(2^{x_i})^2$ and so we have space for at least x_i bits. Of course, we do not know which argument x_i is at a power p but we know the relation between the arguments at two neighboring powers. This leads to the following μ-recursive derivation of the graph of exponentiation :

$$\begin{aligned}
&2^x = y \leftrightarrow x < y \land \exists n\, y = 2^n \land \exists s{<}y \forall p{\leq}y \forall a{\leq}x{\cdot}y{+}s \forall b{<}p \forall c{<}p\,(\\
&\quad Prn(p,y) \land x{\cdot}y + s = a{\cdot}p{\cdot}p + b{\cdot}p + c \rightarrow p = 1 \land b = 0 \lor p > 1 \land \\
&\quad\quad \exists q{<}p \exists d{<}q \exists e{<}q\, (c = d{\cdot}q + e \land (\, p = q{\cdot}q \land b = 2{\cdot}d \lor \\
&\qquad\qquad\qquad\qquad\qquad\qquad p = 2{\cdot}q{\cdot}q \land b = 2{\cdot}d + 1))\,)\ .
\end{aligned}$$

p	2^{x_0}	$\cdots$	2^{x_i}	$2^{x_{i+1}}$	$\cdots$	2^1	2^0
$s =$	$x_0{=}x$	$\cdots$	x_i	x_{i+1}	$\cdots$	$x_{l-1}{=}1$	$x_l{=}0$

$\geq x_i$ bits $x_{i+1}+$ x_i mod 2 bits x_{i+1} bits

x_i bits

Figure 5.1: Course of values sequence s for $2^x = y$

The exponentiation function 2^x has now a μ-recursive definition by the regular minimalization of its graph:

$$2^x = \mu_y[2^x = y]\ .$$

5.2.14 Theorem *Every μ-recursively closed class of partial functions $\mathcal{F}$ is closed under primitive recursion.*

Proof. Let the $(n+1)$-ary function f be defined by primitive recursion from functions g and h in $\mathcal{F}$. We would like to introduce into $\mathcal{F}$ the graph $f(x,\vec{y}) = z$ of f. We cannot do this directly but we will able to derive in $\mathcal{F}$ 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:

$$f(x,\vec{y}) \stackrel{k}{=} z \leftrightarrow f(x,\vec{y}) = z \land k > 0 \land \forall i{\leq}x\, f(i,\vec{y}) < 2^k\ .$$

Here the positive power of two 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 in $\mathcal{F}$. For given numbers x and $\vec{y}$ there is clearly a number $k > 0$ 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 into $\mathcal{F}$

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 in $\mathcal{F}$ 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 in $\mathcal{F}$. If $2^k > 1$ 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 $z{\cdot}2^{k\cdot x} + s = \sum_{i\le x} f(i,\vec{y}){\cdot}2^{k\cdot i}$ where $s{<}2^{k\cdot x}$ and whose digits in the 2^k-ary representation are:

digit position:	$2^{k\cdot x}$	$2^{k\cdot(x-1)}$	$\cdots$	$2^{k\cdot 1}$	$2^{k\cdot 0}$
$z{\cdot}2^{k\cdot x} + s =$	$f(x,\vec{y})$	$f(x-1,\vec{y})$	$\cdots$	$f(1,\vec{y})$	$f(0,\vec{y})$

We can establish the relation between two neighboring digits in the course of values sequence as follows:

$$i \le x \wedge z{\cdot}2^{k\cdot x} + s = a{\cdot}2^{k\cdot(i+2)} + v{\cdot}2^{k\cdot(i+1)} + w{\cdot}2^{k\cdot i} + b \rightarrow v = h(i,\vec{y},w)\ .$$

Thus the following explicit definition derives the bounded graph of $f \in \mathcal{F}$ as a predicate μ-recursive in g and h:

$$\begin{aligned}
&f(x,\vec{y}) \stackrel{k}{=} z \leftrightarrow k > 0 \wedge z < 2^k \wedge \exists s{<}2^{k\cdot x}\forall i{\le}x\forall a{\le}z{\cdot}2^{k\cdot x}{+}s\forall v{<}2^k\forall b{<}2^{k\cdot i}\,(\\
&\quad z{\cdot}2^{k\cdot x} + s = a{\cdot}2^{k\cdot(i+1)} + v{\cdot}2^{k\cdot i} + b \rightarrow\\
&\qquad i = 0 \wedge v = g(\vec{y}) \vee\\
&\qquad i > 0 \wedge \exists j{<}i\exists w{<}2^k\exists c{<}2^{k\cdot j}\,(\\
&\qquad\quad i = j+1 \wedge b = w{\cdot}2^{k\cdot j} + c \wedge v = h(j,\vec{y},w))\,)\ .
\end{aligned}$$ ■

5.2.15 Theorem (Alternative definition of (partial) μ-recursive functions) *The class of partial μ-recursive functions is the smallest primitively recursively closed class which is closed under minimalization.*

Proof. Partial μ-recursive functions contain the identity, zero, and successor functions, and are closed under composition and primitive recursion (the last by Thm. 5.2.14). Hence, the class of partial μ-recursive functions is primitively recursively closed.

Let $\mathcal{F}$ be any primitively recursively closed class of partial functions which is also closed under minimalization. The class $\mathcal{F}$ then contains the partial μ-recursive functions. ■

5.2.16 Theorem (Characterization of recursive functions) *Partial recursive functions are exactly the partial μ-recursive funtions.*

Recursive functions are generated from I^n_m, $x{\cdot}y$, and $x <_* y$ by composition and regular minimalization.

Proof. By Thm. 5.2.2 every partial recursive function f is obtained through a primitive recursive function T by one minimalization of a primitive recursive approximation function. Thus f is a partial μ-recursive function by Thm. 5.2.14. Vice versa, the initial μ-recursive functions are recursive and partial recursive functions are closed under composition. Thus it suffices to show the closure of partial recursive functions under minimalization. Let the partial function f be defined by minimalization from an $(n+1)$-ary recursive function g: $f(\vec{x}) = \mu_y[g(y,\vec{x}) = 1]$. Consider the following recursive definition of a partial recursive function f_1:

$$
\begin{aligned}
&f_1(y,\vec{x}) \asymp y \leftarrow g(y,\vec{x}) = 1\\
&f_1(y,\vec{x}) \asymp z \leftarrow g(y,\vec{x}) \neq 1 \land f_1(y+1,\vec{x}) \asymp z \ .
\end{aligned}
$$

We clearly have $f_1(y,\vec{x}) \asymp z$ iff z is the least number such that $g(z,\vec{x}) = 1$ and $z \geq y$. Thus we derive f as a partial recursive function by explicit definition: $f(y,\vec{x}) \simeq f_1(0,\vec{x})$.

For the proof of the second part of the theorem let us denote by $\mathcal{F}$ the given inductively defined class of (total) functions. $\mathcal{F}$ is clearly a proper subclass of partial μ-recursive functions. From the first part of the theorem we know that recursive functions are exactly the μ-recursive functions and so every $f \in \mathcal{F}$ is a recursive function. The converse assertion that all recursive, i.e. μ-recursive, functions are in $\mathcal{F}$ asserts that detours through partial functions can be eliminated from the definitions of μ-recursive functions. By going through all definitions made in this section (including the ones in the proofs of previous theorems) we can see that all minimalizations were regular and so we never made a detour through a partial function. Thus all primitive recursive functions are in $\mathcal{F}$ by Thm. 5.2.14 and by Thm. 5.2.2 every recursive function f can be obtained by one regular minimalization from primitive recursive functions. This means that f is in the class $\mathcal{F}$. ∎

Chapter 6

Computability

In Sect. 6.1 we give the analysis of the notion of effective computability by A. Turing. In Sect. 6.2 we develop the basics of the theory of computability.

6.1 Turing Machines

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 *computable* partial function as such which is computed by a certain kind of a mechanical machine. We will prove in Corollary 6.1.12 that the computable partial functions coincide with the partial recursive ones and we will discuss in Par. 6.1.13 the reasons for the identification of effective computability with the computability (in the sense of Turing).

6.1.1 Analysis of effective computability by Turing. Turing in [25] 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 natural numbers b and e denoting respectively its initial and terminal states and by a binary *transition* function m. The transition function is over $\mathbb{N}$ and it is finite in the following sense.

Both sets of states $Q = \{q \mid \exists s\, m(q,s) > 0\}$ and of *tape symbols* $S = \{s \mid \exists q\, m(q,s) > 0\}$ are finite, we have $b \in Q$, $e \in Q$, $0 \in S$, $1 \in S$, and for every $q \in Q$ and $s \in S$ there are numbers q_1, s_1, and d such that $m(q,s) = q_1, s_1, d$ where $q_1 \in Q$, $s_1 \in S$, and $0 \leq d \leq 2$.

The tape symbol 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 m 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 $d = 1$, 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 $d = 2$, 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 $d = 0$, 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 where we require that $m(e,s) = e, s, 0$ holds for all $s \in S$.

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 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 started with the tape:

$$\mathbf{0}^\infty\, \underline{\mathbf{0}}\mathbf{1}^{x_1}\, \mathbf{0}\mathbf{1}^{x_2}\, \cdots\, \mathbf{0}\mathbf{1}^{x_{n-1}}\, \mathbf{0}\mathbf{1}^{x_n}\, \mathbf{0}^\infty \tag{1}$$

terminates iff $f(x_1, x_2, \dots, x_{n-1}, x_n)\downarrow$ with the tape:

$$\mathbf{0}^\infty\, \underline{\mathbf{0}}\mathbf{1}^{f(x_1,x_2,\dots,x_{n-1},x_n)}\, \mathbf{0}^\infty \ .$$

Note that an argument or result x is represented on the tape by a *block* $\mathbf{01}^x$ with the blank symbol $\mathbf{0}$ called the block *separator*.

A partial function f is *partially computable* if there is a Turing machine computing f; f is *computable* if it is partially computable and total.

6.1.4 Lemma *Partially computable functions are partial recursive.*

Proof. Let the n-ary partial function f be computed by a Turing machine A given by the initial state b, terminal state e, and a transition function m. We may assume without loss of generality that the set S of its tape symbols is $\{0, 1, \dots, p{-}1\}$ for a $p \geq 2$ because we can always systematically modify the transition fuction m. The function m is primitive recursive because it can be defined by a finite number of non-recursive clauses.

We arithmetize the empty word over S by 0. Finite non-empty words $s_0 \cdots s_m$ over S with either s_0 or s_m non-zero are arithmetized in two different ways. Words with $s_0 > 0$ are *left* words and they are arithmetized in the p-ary notation as $\sum_{i \leq m} s_i{\cdot}p^{m-i}$. Words with $s_m > 0$ are *right* words and they are arithmetized in the 'reversed' p-ary notation as $\sum_{i \leq m} s_i{\cdot}p^i$. The empty word is both left and right at the same time. Tapes for A can be uniquely written as $\mathbf{0}^\infty\, w_1\, w_2\, \mathbf{0}^\infty$ with w_1 a left and w_2 a right word. We assume that the currently scanned square is the first symbol of the (infinite) word $w_2\,\mathbf{0}^\infty$. Such a tape is arithmetized by the pair of numbers l, r where l codes the left word w_1 and r the right word w_2.

The Turing machine A is arithmetized by a ternary partial function $M(q, l, r)$ with q the current state of A and the pair (l, r) the code of the current tape of A. $M(b, l, r)$ is defined iff A stops when started with the initial tape coded by (l, r). If $M(b, l, r)$ is defined then $M(q, l, r) \asymp l_1, r_1$ where (l_1, r_1) codes the final state of the tape. The partial function M has the following clausal definition:

$$\begin{aligned}
&M(e, l, r) \asymp l, r\\
&M(q, l, p{\cdot}r{+}s) \asymp t \leftarrow q \neq e \wedge s < p \wedge m(q, s) = q_1, s_1, 0 \wedge M(q_1, l, p{\cdot}r{+}s_1) \asymp t\\
&M(q, l, p{\cdot}r{+}s) \asymp t \leftarrow q \neq e \wedge s < p \wedge m(q, s) = q_1, s_1, 1 \wedge\\
&\qquad\quad l = p{\cdot}l_1{+}s_2 \wedge s_2 < p \wedge M(q_1, l_1, p{\cdot}(p{\cdot}r{+}s_1){+}s_2) \asymp t\\
&M(q, l, p{\cdot}r{+}s) \asymp t \leftarrow q \neq e \wedge s < p \wedge m(q, s) = q_1, s_1, 2 \wedge M(q_1, p{\cdot}l{+}s_1, r) \asymp t \ .
\end{aligned}$$

For the computation of $f(x_1, x_2, \dots, x_{n-1}, x_n)$ we will start the arithmetized machine as $M(b, 0, \mathit{Encode}(x_1, x_2, \dots, x_{n-1}, x_n, 0))$ where the pair formed from the second and third arguments codes the tape 6.1.3(1). The function *Encode* takes a list of numbers and yields the code of a right word consisting of the concatenated blocks representing the numbers in the list. The function has the following clausal definition:

$$\mathit{Pconc}(0, a) = p{\cdot}a$$

$$Pconc(x+1,a) = Pconc(x, p{\cdot}a{+}1)$$
$$Encode(0) = 0$$
$$Encode(x,t) = Pconc(x, Encode(t))$$

where $Pconc(x, \ulcorner w \urcorner)$ yields the right word code of $\mathbf{01}^x\, w$. The partial function f is recursive by:

$$f(\vec{x}) \asymp |r|_p \leftarrow M(b,0,Encode(\vec{x},0)) \asymp t \wedge t = 0, p{\cdot}r{+}0$$

where the *p-ary size* function $|x|_p$ yields the length of x in the p-ary representation and has the following clausal definition regular in its argument.:

$$|0|_p = 0$$
$$|p{\cdot}r{+}i|_p = |r|_p + 1 \leftarrow i < p \wedge r + i > 0 \ .$$

Note that we have $|\ulcorner \mathbf{1}^x \urcorner|_p = x$. ■

6.1.5 Register machines. We will prove the converse of Lemma 6.1.4 by a detour through partial functions computed by *register machines.* An (n, m) register machine M takes an n-tuple $\vec{x}$ of natural numbers $(n \geq 1)$ as *input* and if it stops then it produces an m-tuple of natural numbers $\vec{y}$ $(m \geq 1)$ as *output.* We capture this by the *input-output relation for* M: $\vec{x}\,[M]\,\vec{y}$. If M stops then the values $\vec{y}$ are uniquely determined. The i-th value x_i in $\vec{x}$ can be viewed as the input value of the i-th register R_i of M and the j-th value y_j in $\vec{y}$ as the output value of the j-th register R_j.

Our register machines are limited in that the basic operations modify only the registers with the highest numbers. This is because we intend to represent the input registers $x_1, x_2, \ldots, x_{n-1}, x_n$ of M by a tape

$$\mathbf{0}^\infty\, w\, \underline{\mathbf{0}}\mathbf{1}^{x_1}\, \mathbf{01}^{x_2} \cdots \mathbf{01}^{x_{n-1}}\, \mathbf{01}^{x_n}\, \mathbf{0}^\infty \tag{1}$$

where w is a finite number of blocks which will not be affected by Turing machines interpreting the machine M. When M stops with $y_1, y_2, \ldots, y_{m-1}, y_m$ so does A with the output tape:

$$\mathbf{0}^\infty\, w\, \underline{\mathbf{0}}\mathbf{1}^{y_1}\, \mathbf{01}^{y_2} \cdots \mathbf{01}^{y_{m-1}}\, \mathbf{01}^{y_m}\, \mathbf{0}^\infty \ . \tag{2}$$

There are four *basic* register machines. The *increment* machine denoted by $R_1 := R_1{+}1$ is $(1,1)$ and has the input-output relation:

$$x\,[R_1 := R_1{+}1]\, x+1 \ .$$

The *decrement* $R_1 := R_1 \dot{-} 1$ machine is $(1,1)$ with the input-output relation:

$$x\,[R_1 := R_1 \dot{-} 1]\, x \dot{-} 1 \ .$$

The *extension* *Ext* machine is $(1,2)$ with the input-output relation:

$$x\,[Ext]\, x, 0 \ .$$

The *assignment* machine $R_1 := R_2$ is $(2,1)$ with the input-output relation:

$$x, y\,[R_1 := R_2]\,y\ .$$

All register machines are obtained from the basic machines by four operations on register machines. For an (n,k) machine M_1 and an (k,m) machine M_2 we denote by $M_1; M_2$ the *composition* register machine which is (n,m) and has the input-output relation:

$$\vec{x}\,[M_1; M_2]\,\vec{y} \leftrightarrow \exists \vec{z}(\,\vec{x}\,[M_1]\,\vec{z} \wedge \vec{z}\,[M_2]\,\vec{y}\,)\ .$$

For an (n,m)-register machine M the *right* register machine $R(M)$ is $(n{+}1, m{+}1)$ and its input-output relation satisfies:

$$v, \vec{x}\,[R(M)]\,v, \vec{y} \leftrightarrow \vec{x}\,[M]\,\vec{y}\ .$$

For an $(n, n{+}1)$-register machine M the *while* register machine $W(M)$ is $(n{+}1, n)$ and its input-output relation satisfies:

$$\vec{x}, v\,[W(M)]\,\vec{y} \leftrightarrow v = 0 \wedge \vec{x} = \vec{y} \vee v > 0 \wedge \vec{x}\,[M; W(M)]\,\vec{y}\ .$$

For an (n,n)-register machine M the *for* register machine $F(M)$ is $(n{+}1, n{+}1)$ and its input-output relation satisfies:

$$v, \vec{x}\,[F(M)]\,v, \vec{y} \leftrightarrow v = 0 \wedge \vec{x} = \vec{y} \vee v > 0 \wedge v \dot{-} 1, \vec{x}\,[R(M); F(M)]\,v \dot{-} 1, \vec{y}\ .$$

We say that an $(n,1)$-register machine M *computes* the n-ary partial function f if we have

$$f(\vec{x}) \asymp y \leftrightarrow \vec{x}\,[M]\,y\ .$$

A partial function f is *partially register computable* if there is a register machine computing f; f is *register computable* if it is partially register computable and total.

6.1.6 Bootstrapping the register machines. We let the operation of composition group to the left and abbreviate $(M_1; M_2); M_3$ to $M_1; M_2; M_3$. We also introduce generalized right $(n+k, m+k)$-register machines $R^k(M)$ where M is an (n,m)-register machine and $k \geq 1$. The machine $R^k(M)$ satisfies:

$$\vec{z}, \vec{x}\,[R^k(M)]\,\vec{z}, \vec{y} \leftrightarrow \vec{x}\,[M]\,\vec{y}$$

where $\vec{z}$ is a k-tuple of variables. The machine is constructed by the k-fold iteration of R: $\overbrace{R\,R\,\cdots\,R}^{k}(M)$.

We introduce some additional register machines generalizing the basic operations. For $n > 1$ the $(n, n{+}1)$-register machine $Ext(n)$ generalizes the extension machine and it satisfies:

$$\vec{x}\,[Ext(n)]\,\vec{x}, 0\ .$$

The machine is constructed as $R^{n-1}(Ext)$. We let $Ext(1)$ to stand for Ext.

The $(n{+}1,n)$ *contraction* register machine $R_{n+1} := 0$ is constructed as $W(Ext(n))$ and satisfies:

$$\vec{x}, z\,[R_{n+1} := 0]\,\vec{x}\ .$$

For $n > 1$ the (n,n)-register machine $R_n := R_n{+}1$ generalizes the increment machine to satisfy

$$x_1, \ldots, x_{n-1}, x_n\,[R_n := R_n{+}1]\,x_1, \ldots, x_{n-1}, x_n + 1\ .$$

The machine is constructed as $R^{n-1}(R_1 := R_1{+}1)$.

The $(n{+}1,1)$ generalized assignment machines $R_1 := R_{n+1}$ satisfy the input-output relation:

$$\vec{x}, y\,[R_1 := R_{n+1}]\,y$$

and are constructed recursively. For $n = 1$ the machine is the basic machine $R_1 := R_2$ and for $n > 1$ we construct the $(n,1)$-register machine $R_1 := R_n$ by IH and then construct the assignment machine as $R(R_1 := R_n); R_1 := R_2$.

The $(n,n{+}1)$ *extend and copy the first register* machine $R_{n+1} := R_1$ satisfies

$$x_1, \ldots, x_n\,[R_{n+1} := R_1]\,x_1, \ldots, x_n, x_1$$

and it is constructed as $Ext(n); F(R_n := R_n{+}1)$.

For $1 \le m \le n$ the $(n,n{+}m)$ *extend and copy the first m registers* register machine $R_{n+1..n+m} := R_{1..m}$ is constructed by recursion on m to satisfy:

$$x_1, \ldots, x_m, \ldots, x_n\,[R_{n+1..n+m} := R_{1..m}]\,x_1, \ldots, x_m, \ldots, x_n, x_1, \ldots, x_m\ .$$

For $m = 1$ the machine $R_{n+1..n+1} := R_{1..1}$ is the copy the first register machine $R_{n+1} := R_1$. For $m > 1$ we obtain by IH the machine $R_{n+1..n+m-1} := R_{1..m-1}$ copying the first $m{-}1$ registers and construct the register machine as:

$$R_{n+1} := R_1; R(R_{n+1..n+m-1} := R_{1..m-1})\ .$$

6.1.7 Lemma *Partially register computable functions are closed under explicit definitions.*

Proof. We show first that every identity function I^n_i is $(n,1)$-register machine computable by considering three cases. The machine computing $I^1_1 = I$ does nothing: $R_1 := R_1{+}1; R_1 := R_1 \dot{-} 1$. When $n > 1$ and $i = 1$ the register machine computing I^n_1 is constructed as:

$$R_n := 0; R_{n-1} := 0; \ldots ; R_2 := 0\ .$$

When $n > i > 1$ the register machine computing I^n_i is constructed as:

$$R_n := 0; R_{n-1} := 0; \ldots ; R_{i+1} := 0; R_1 := R_i\ .$$

We show next the closure of register machine computable functions under composition. Let the partial function $f(\vec{x}) \simeq g(h_1(\vec{x}), \dots, h_m(\vec{x}))$ be such that the $(m,1)$-register machine M_g computes g and the $(n,1)$-register machines $M_{h_1}, \dots, M_{h_m}$ compute in that order $h_1, \dots, h_m$. The following $(n,1)$ register machine then computes f:

$$
\begin{aligned}
&R_{n+1..n+n} := R_{1..n}; R^n(M_{h_1});\\
&R_{n+2..n+n+1} := R_{1..n}; R^{n+1}(M_{h_2});\\
&\qquad\vdots\\
&R_{n+m..n+m-1} := R_{1..n}; R^{n+m-1}(M_{h_m}); R^n(M_g); R_1 := R_{n+1}\ .
\end{aligned}
$$

By the Characterization theorem 3.2.1 the partially register machine computable functions are thus closed under explicit definitions without constants.

In order to show the closure also under explicit definitions we use Thm. 3.2.2 and show that the constant functions C_n are register machine computable. This is proved by induction on n. The function $C_0 = Z$ is computed by the $(1,1)$-register machine $Ext; R_1 := R_2$ and the function $C_{n+1}(x) = S\,C_n(x)$ is defined by explicit definition without constants from the successor function which is computed by the machine $R_1 := R_1{+}1$ and from the function C_n which is register computable by IH. ■

6.1.8 Some register computable functions. The addition function is computed by the $(2,1)$-register machine: $F(R_1 := R_1{+}1); R_1 := R_2$.

The multiplication function is computed by the $(2,1)$-register machine:

$$Ext(2); FF(R_1 := R_1{+}1); R_1 := R_3\ .$$

The modified subtraction function $x \dot{-} y$ is computed by the $(2,1)$-register machine:

$$R_3 := R_1; R\,F(R_1 := R_1 \dot{-} 1); R_1 := R_3\ .$$

The *signum* function $x >_* 0$ is register computable because it has the following explicit definition: $x >_* 0 = 1 \dot{-} (1 \dot{-} x)$.

The characteristic function of $<$ is register computable because it has the following explicit definition: $x <_* y = (y \dot{-} x) >_* 0$.

We will need to know that the following explicitly defined function is register computable:

$$x \neq_* 1 = ((x \dot{-} 1) >_* 0) + (1 \dot{-} x)\ .$$

6.1.9 Lemma *Partial μ-recursive functions are register computable.*

Proof. By Lemma 6.1.7 and results of Par. 6.1.8 it is sufficient to show the closure of partially register computable functions under minimalization. Thus let $f(\vec{x}) \simeq \mu_y[g(y,\vec{x}) = 1]$ be defined from an $(n+1)$-ary register computable function g. Then also the explicitly defined function $g'(y,\vec{x}) = g(y,\vec{x}) \neq_* 1$ is register computed by an $(n+1,1)$-register machine $M_{g'}$. The auxiliary $(n+1,n+2)$-register machine $Comp_{g'}$ constructed as

$$R^n(R_2 := R_1); R_{n+3..n+n+2} := R_{1..n}; R^{n+1}(M_{g'})$$

satisfies the input-output relation $\vec{x},y\,[Comp_{g'}]\,\vec{x},y,(g(y,\vec{x})\neq_* 1)$. The partial function f is thus computed by the $(n,1)$-register machine:

$$Ext(n); Comp_{g'}; W(R_{n+1} := R_{n+1}+1; Comp_{g'}); R_1 := R_{n+1}\ . \qquad \blacksquare$$

6.1.10 Operations on Turing machines. We will show in Lemma 6.1.11 that register computable partial functions are partially computable. This will be done by the 'microprogramming' of register machines by certain Turing machines working with the tape alphabet $\{0,1\}$ and constructed from five basic Turing machines by three operators over Turing machines.

The basic Turing machines are denoted by $\boldsymbol{\epsilon}$, $\boldsymbol{r}$, $\boldsymbol{l}$, $\mathbf{0}$, and $\mathbf{1}$ and they will in that order stop without doing anything, move to the right by one square, move to the left by one square, write $\mathbf{0}$ on the currently scanned square, and write $\mathbf{1}$ on the currently scanned square. The machine $\boldsymbol{\epsilon}$ has both the initial and terminating states 0, the remaining four machines have the initial states 1 and termination states 0 and their transition functions have the following clausal definitions:

$$
\begin{aligned}
&m_{\boldsymbol{\epsilon}}(0,s) = 0,s,0\\
&m_{\boldsymbol{l}}(1,s) = 0,s,1\\
&m_{\boldsymbol{l}}(0,s) = 0,s,0\\
&m_{\boldsymbol{r}}(1,s) = 0,s,2\\
&m_{\boldsymbol{r}}(0,s) = 0,s,0\\
&m_{\mathbf{0}}(1,s) = 0,0,0\\
&m_{\mathbf{0}}(0,s) = 0,s,0\\
&m_{\mathbf{1}}(1,s) = 0,1,0\\
&m_{\mathbf{1}}(0,s) = 0,s,0\ .
\end{aligned}
$$

For two given Turing machines A and B given by b_A, e_A, m_A and b_B, e_B, m_B respectively we denote by AB the Turing machine obtained by the *composition* of the two machines. When the machine AB is started it operates like the machine A until this stops and then the machine AB starts operating like the machine B from the same position of the tape. The initial state of the composition-machine AB is $b_{AB} = (0,b_A)$ and the terminating state is $e_{AB} = (1,e_B)$. Its transition function m_{AB} has the following clausal definition:

$$
\begin{aligned}
&m_{AB}((0,q),s) = (1,b_B),s_1,d \leftarrow m_A(q,s) = e_A,s_1,d\\
&m_{AB}((0,q),s) = (0,q_1),s_1,d \leftarrow m_A(q,s) = q_1,s_1,d \wedge q_1 \neq e_A\\
&m_{AB}((1,q),s) = (1,q_1),s_1,d \leftarrow m_B(q,s) = q_1,s_1,d\ .
\end{aligned}
$$

We abbreviate $(AB)C$ to ABC. We will use the iteration notation A^0 where we let A^0 to denote the do-nothing $\boldsymbol{\epsilon}$ machine and A^{n+1} to denote the machine AA^n.

For two given Turing machines A and B we denote by $\boldsymbol{D}(A,B)$ the *conditional* Turing machine which at start tests the currently scanned square. If the sqaure contains $\mathbf{1}$ then the conditional-machine acts as the machine A otherwise it acts as the machine B. The initial state of the $\boldsymbol{D}(A,B)$ machine is $b_{\boldsymbol{D}(A,B)} = (0,0)$ and the terminating one is $e_{\boldsymbol{D}(A,B)} = 0$. Its transition function $m_{\boldsymbol{D}(A,B)}$ has the following clausal definition:

$$
\begin{aligned}
&m_{\boldsymbol{D}(A,B)}((0,0),1) = (1,b_A),1,0\\
&m_{\boldsymbol{D}(A,B)}((0,0),0) = (2,b_B),0,0\\
&m_{\boldsymbol{D}(A,B)}(0,s) = 0,s,0\\
&m_{\boldsymbol{D}(A,B)}((1,e_A),s) = 0,s,0\\
&m_{\boldsymbol{D}(A,B)}((1,q),s) = (1,q_1),s_1,d \leftarrow q \neq e_A \wedge m_A(q,s) = q_1,s_1,d\\
&m_{\boldsymbol{D}(A,B)}((2,e_B),s) = 0,s,0\\
&m_{\boldsymbol{D}(A,B)}((2,q),s) = (2,q_1),s_1,d \leftarrow q \neq e_B \wedge m_B(q,s) = q_1,s_1,d\ .
\end{aligned}
$$

For a given Turing machine A we denote by $(A)^*$ the *while* Turing machine which on start tests the currently scanned square. If the square contains $\mathbf{0}$ the while machine stops. Otherwise the while machine acts as the machine A until this terminates and then the while-machine returns to the test of the currently scanned square. Thus the machines $(A)^*$ and $\boldsymbol{D}(A(A)^*,\boldsymbol{\epsilon})$ have identical behaviors. The initial state of the *while*-machine is $b_{(A)^*} = (0,0)$ and the terminating one is $e_{(A)^*} = 0$. Its transition function $m_{(A)^*}$ has the following clausal definition:

$$
\begin{aligned}
&m_{(A)^*}((0,0),1) = (1,b_A),1,0\\
&m_{(A)^*}((0,0),0) = 0,0,0\\
&m_{(A)^*}(0,s) = 0,s,0\\
&m_{(A)^*}((1,e_A),s) = (0,0),s,0\\
&m_{(A)^*}((1,q),s) = (1,q_1),s_1,d \leftarrow q \neq e_A \wedge m_A(q,s) = q_1,s_1,d\ .
\end{aligned}
$$

6.1.11 Lemma *Partially register computable functions are computable.*

Proof. Until the end of the proof we denote by $\boldsymbol{L}$ the *move-left-one-block* Turing machine $\boldsymbol{l}(\boldsymbol{l})^*$ and by $\boldsymbol{R}$ the *move-right-one-block* Turing machine $\boldsymbol{r}(\boldsymbol{r})^*$.

By induction on the construction of register machines we assign to every (n,m)-register machine M a Turing machine A interpreting M in the sense that if A is started with the tape 6.1.5(1) where w is any word over $\{0,1\}$ then A stops with the tape 6.1.5(2) iff $\vec{x}\,[M]\,\vec{y}$ holds. Note that the word w to the left of the n input blocks of 6.1.5(1) is not affected by the operations of A.

If M is the $(1,1)$ increment machine $R_1 := R_1{+}1$ we assign to it the Turing machine $\boldsymbol{R1L}$ which increases by one the number represented in the first block.

If M is the $(1,1)$ decrement machine $R_1 := R_1 \dot{-} 1$ we assign the following Turing machine to M:

$$\boldsymbol{RlD}(\boldsymbol{0L},\boldsymbol{\epsilon})\ .$$

If M is the $(1,2)$ extension machine *Ext* then we assign to it the Turing machine $\boldsymbol{\epsilon}$ which does nothing as the tape to the right of its first block is already empty.

If M is the $(2,1)$ assignment machine $R_1 := R_2$ then suppose that the input to M is x, y. We need an auxiliary Turing machine $\boldsymbol{S}$ which shifts to the left by one square the scanned second block storing y. The situation on the tape before the start of $\boldsymbol{S}$ will be $\mathbf{0}^\infty\, w\, \mathbf{0}\mathbf{1}^x\underline{\mathbf{0}}\mathbf{1}^y\mathbf{0}^\infty$ and the situation after the termination will be $\mathbf{0}^\infty\ \mathbf{0}\mathbf{1}^{x-1}\underline{\mathbf{1}}\mathbf{1}^y\mathbf{0}^\infty$ if $x > 0$ and $\mathbf{0}^\infty\ \underline{\mathbf{0}}\mathbf{1}^y\mathbf{0}^\infty$ otherwise. The Turing machine $\boldsymbol{S}$ is

$$\boldsymbol{R}\boldsymbol{l}\boldsymbol{D}(\mathbf{0}\boldsymbol{L}\mathbf{1}, \boldsymbol{\epsilon})\boldsymbol{l}\ .$$

We now assign to M the Turing machine $\boldsymbol{R}\boldsymbol{S}(\mathbf{0}\boldsymbol{S})^*$. The machine repeats the shifting of the block y to the left while the block x is non-zero.

If M is the (n,m) composition register machine $M_1; M_2$ where M_1 is (n,k) and M_2 is (k,m) we obtain from IH the Turing machines A_1 and A_2 assigned respectively to M_1 and M_2 and we assign the Turing machine $A_1 A_2$ to M.

If M is the $(n{+}1, m{+}1)$ right machine $R(M_1)$ we obtain from IH the Turing machine A_1 assigned to the (n,m)-register machine M_1 and assign the Turing machine $\boldsymbol{R}A_1\boldsymbol{L}$ to M.

If M is the $(n{+}1, n)$ while machine $W(M_1)$ we obtain from IH the Turing machine A_1 assigned to the $(n, n{+}1)$-register machine M_1 and assign the following Turing machine to M:

$$\boldsymbol{R}^{n+1}\boldsymbol{l}((\mathbf{0}\boldsymbol{l})^*\boldsymbol{L}^n A_1 \boldsymbol{R}^{n+1}\boldsymbol{l})^*\boldsymbol{L}^n\ .$$

Finally, if M is the $(n{+}1, n{+}1)$ for machine $F(M_1)$ we obtain from IH the Turing machine A_1 assigned to the (n,n)-register machine M_1 and assign the following Turing machine to M:

$$\boldsymbol{r}(\mathbf{0}\boldsymbol{R}A_1\boldsymbol{L}\mathbf{1}\boldsymbol{r})^*\boldsymbol{L}\ .$$

We note that if M has as input $v, \vec{x}$ with $v > 0$ then when the i-th iteration such that $i < v$ starts the situation on the tape is:

$$\mathbf{0}^\infty\, w\, \mathbf{0}\mathbf{1}^i\underline{\mathbf{1}}\mathbf{1}^{v-(i+1)}\, w_{M_1}\, \mathbf{0}^\infty\ .$$

This ends the proof by induction on the construction of register machines.

The just proved property assigns to every $(n,1)$-register machine M computing a partial function f a Turing machine A computing the same f. ■

6.1.12 Corollary *Partial recursive functions coincide with partially computable functions.*

Proof. Partial recursive functions are partial μ-recursive by the Characterization theorem 5.2.16 and that the latter class is computable follows from Lemmas 6.1.9 and 6.1.11. The inclusion in the oposite direction follows from Lemma 6.1.4. ■

6.1.13 Theses of Turing and Church. The analysis by Turing of effective computability (see Par. 6.1.1) has led to his definition of Turing machines and of partially computable functions. That every partially computable function is effectively computable by a computer with limited resources as described in Par. 6.1.1 is obvious. Turing in [25] proved also the converse that limited computers cannot compute any partial functions which are not partially computable.

Turing then proposed to identify the imprecise notion of of effective computability with the computability by Turing machines. This constitutes a definition of effective computability and it is today known as the *Turing's Thesis*. The thesis was enthusiastically accepted by K. Gödel who himself had earlier proposed the model of effective computability in a form of what is now called Herbrand-Gödel-Kleene systems of equations and which we have adopted in a slightly adapted form as the basis of the definition of partial recursive functions. Gödel thought that the analysis of Turing is much more intuitive than computability by recursive functions. Gödel has previously rejected as unconvincing the *Thesis of Church* [5] who proposed to identify the effective computability with the functions definable by Herbrand-Gödel-Kleene systems of equations. Corollary 6.1.12 shows that both definitions of effective computability lead to the same class of partial functions.

Despite the great intuitive appeal of Turing's thesis we have decided to start investigating the effectively computable functions through partial recursive functions because they lead directly to the modern declarative programming languages. On the other hand, the notions of state and store appearing in the definition of (Turing) computability lead to the imperative programming languages which we consider (their tremendous current popularity notwithstanding) a blind alley in the development of modern computing.

The interested reader is referred to excellent study by R. Soare [20] who compares in depth the history of development of theory of computability by recursive and (Turing) computable functions.

6.2 Fundamentals of Computability Theory

6.2.1 Partial reduction function $\boldsymbol{Rd}^*(a)$. Let $\mathcal{F}$ be a class of functions enumerated by g. The function $\boldsymbol{Rd}_g$ introduced in Par. 4.3.4 is primitive recursive in g and it arithmetizes one reduction in computations of partial $\mathcal{F}$-recursive functions. The following clausal definition of the partial g-recursive function $\boldsymbol{Rd}^*_g$ iterates $\boldsymbol{Rd}_g$ in the hope of obtaining the code of a monadic numeral:

$$
\begin{aligned}
&\boldsymbol{Rd}^*_g(a) \asymp y \leftarrow \boldsymbol{Num}(a) \wedge a = \boldsymbol{C}(y)\\
&\boldsymbol{Rd}^*_g(a) \asymp y \leftarrow \neg\boldsymbol{Num}(a) \wedge \boldsymbol{Rd}^*_g\,\boldsymbol{Rd}_g(a) \asymp y\ .
\end{aligned}
$$

The partial function has the following properties where $\lambda\vec{x}.\tau$ is an n-ary $R^{\mathcal{F}}$-function symbol:

$$\boldsymbol{Rd}^*_g(a) \asymp y \leftrightarrow \exists k\, \boldsymbol{Rd}^k_g(a) = \boldsymbol{C}(y) \tag{1}$$

$$\boldsymbol{Rd}^*_g((\boldsymbol{\lambda}\ulcorner\tau\urcorner^n)(\boldsymbol{C}(\vec{x})) \asymp y \leftrightarrow (\lambda\vec{x}.\tau)(\underline{\vec{x}}) \blacktriangleright^{\mathcal{F}} \underline{y}\ . \tag{2}$$

For the proof of (1) we note that the chain of partial functions $\boldsymbol{Rd}^*_k$ from the Recursion theorem 2.1.12 whose limit is the partial function $\boldsymbol{Rd}^*_g$ can be defined as $\boldsymbol{Rd}^*_0 = \emptyset^1$ and

$$\begin{aligned}&\boldsymbol{Rd}^*_{k+1}(a) \asymp y \leftarrow \boldsymbol{Num}(a) \land a = \boldsymbol{C}(y)\\ &\boldsymbol{Rd}^*_{k+1}(a) \asymp y \leftarrow \neg\boldsymbol{Num}(a) \land \boldsymbol{Rd}^*_k\, \boldsymbol{Rd}_g(a) \asymp y\ .\end{aligned}$$

Hence we have $\boldsymbol{Rd}^*_{k+1}(a) \asymp y \leftrightarrow \boldsymbol{Rd}^k_g(a) = \boldsymbol{C}(y)$ and (1) directly follows. Property (2) follows from (1) by 4.3.4(3).

6.2.2 Partial enumeration function $\{e\}_g(x)$. Let g be a binary function. We define the binary partial *enumeration* function $\{e\}_g(x)$ as recursive in g by

$$\{e\}_g(x) \asymp y \leftarrow \boldsymbol{Rd}^*_g((\boldsymbol{\lambda} e)(\boldsymbol{C}(x))) \asymp y\ .$$

For $g(i,x) = 0$ we explicitly define the following binary partial recursive function:

$$\{e\}(x) \asymp y \leftarrow \{e\}_g(x) \asymp y\quad .$$

6.2.3 Computation of $\{e\}_g(x)$ calls on a finite number of oracles. Let $\mathcal{F}$ be a class of functions enumerated by g. We wish to prove in Thm. 6.2.5 that $\{e\}_g(x)$ enumerates all partial $\mathcal{F}$-recursive functions. The problem is that we do not know whether every e is an index of such a function, i.e. whether the unary partial function $f(x) = \{e\}_g(x)$ is $\mathcal{F}$-recursive although we know that f is partial recursive in g. For the answer we need to investigate the role the 'oracle calls' g_i, i.e. $\boldsymbol{G}(i,a)$, play in the computation of $\{e\}_g(x)$.

The one step reduction $\boldsymbol{Rd}_g(a) = b$ may reduce in a an 'oracle call' $\boldsymbol{G}(i,c)$ to the function g_i. We wish to define a binary primitive recursive function $\boldsymbol{Rd}$ 'externalizing' the oracle calls in the sense that we have

$$\boldsymbol{Rd}_g(a) = \boldsymbol{Rd}(a, (i, \boldsymbol{Dc}(c), g\, \boldsymbol{Dc}(c)))$$

if this is the case. If there is no oracle call in the reduction $\boldsymbol{Rd}_g(a) = b$ then we wish $\boldsymbol{Rd}^a_g = \boldsymbol{Rd}(a,0)$ to hold. The function $\boldsymbol{Rd}$ is defined by a clausal definition regular in its first argument:

$$\begin{aligned}&\boldsymbol{Rd}(\boldsymbol{C}(n),0) = \boldsymbol{C}(n)\\ &\boldsymbol{Rd}(\boldsymbol{S}(a),z) = \boldsymbol{C}(n+1) \leftarrow \boldsymbol{Rd}(a,z) = a_1 \land \boldsymbol{Num}(a_1) \land a_1 = \boldsymbol{C}(n)\\ &\boldsymbol{Rd}(\boldsymbol{S}(a),z) = \boldsymbol{S}(a_1) \leftarrow \boldsymbol{Rd}(a,z) = a_1 \land \neg\boldsymbol{Num}(a_1) \land a_1 > 0\\ &\boldsymbol{Rd}(\boldsymbol{P}(a),z) = \boldsymbol{C}(n \mathbin{\dot-} 1) \leftarrow \boldsymbol{Num}(a) \land a = \boldsymbol{C}(n) \land z = 0\\ &\boldsymbol{Rd}(\boldsymbol{P}(a),z) = \boldsymbol{P}(a_1) \leftarrow \neg\boldsymbol{Num}(a) \land \boldsymbol{Rd}(a,z) = a_1 \land a_1 > 0\end{aligned}$$

$$
\begin{aligned}
&\boldsymbol{Rd}(\boldsymbol{D}(a,b,c),z) = D(n,b,c) \leftarrow \boldsymbol{Num}(a) \wedge a = \boldsymbol{C}(n) \wedge z = 0\\
&\boldsymbol{Rd}(\boldsymbol{D}(a,b,c),z) = \boldsymbol{D}(a_1,b,c) \leftarrow \neg\boldsymbol{Num}(a) \wedge \boldsymbol{Rd}(a,z) = a_1 \wedge a_1 > 0\\
&\boldsymbol{Rd}(\boldsymbol{G}(i,a),z) = \boldsymbol{C}(v) \leftarrow \boldsymbol{Num}(a) \wedge z = i,x,v \wedge \boldsymbol{Dc}(a) = x\\
&\boldsymbol{Rd}(\boldsymbol{G}(i,a),z) = \boldsymbol{G}(i,a_1) \leftarrow \neg\boldsymbol{Num}(a) \wedge \boldsymbol{Rd}(a,z) = a_1 \wedge a_1 > 0\\
&\boldsymbol{Rd}((\boldsymbol{\lambda} a)(b),z) = a[a\boldsymbol{;}b] \leftarrow \boldsymbol{Num}(b) \wedge z = 0\\
&\boldsymbol{Rd}((\boldsymbol{\lambda} a)(b),z) = (\boldsymbol{\lambda} a)(b_1) \leftarrow \neg\boldsymbol{Num}(b) \wedge \boldsymbol{Rd}(b,z) = b_1 \wedge b_1 > 0\\
&\boldsymbol{Rd}(\boldsymbol{A}(a,b),z) = \boldsymbol{A}(a,b_1) \leftarrow \boldsymbol{Num}(a) \wedge \boldsymbol{Rd}(b,z) = b_1 \wedge b_1 > 0\\
&\boldsymbol{Rd}(\boldsymbol{A}(a,b),z) = \boldsymbol{A}(a_1,b) \leftarrow \neg\boldsymbol{Num}(a) \wedge \boldsymbol{Rd}(a,z) = a_1 \wedge a_1 > 0\ .
\end{aligned}
$$

We will need an auxiliary unary predicate $G_g(s)$ which holds if all triples (i,x,v) in the list s are in the graph of g, i.e. if $g(i,x) = v$ holds. The predicate is explicitly defined as primitive recursive in g:

$$G_g(s) \leftrightarrow \forall i \forall x \forall v(\,(i,x,v) \in s \rightarrow g(i,x) = v\,)\ . \tag{1}$$

It should not be difficult to see that we have:

$$G_g(z_1,0) \wedge G_g(z_2,0) \wedge \boldsymbol{Rd}(a,z_1) = \boldsymbol{Rd}(a,z_2) > 0 \rightarrow z_1 = z_2 \tag{2}$$

$$b > 0 \rightarrow \boldsymbol{Rd}_g(a) = b \leftrightarrow \exists z(\,\boldsymbol{Rd}(a,z) = b \wedge G_g(z,0)\,)\ . \tag{3}$$

The reader can see from (3) that the effect of externalization of oracle calls is achieved by removing the oracle calls from within $\boldsymbol{Rd}_g$ to the tests in the auxiliary predicate $G_g(z,0)$ and thus turning the reduction function to the primitive recursive function $\boldsymbol{Rd}$. This is generalized to iterated reductions with a binary primitive recursive function $\boldsymbol{Rds}(a,s)$ which iterates $L(s)$-times the reduction of a where the oracle calls are supplied in the list s. The function has the following clausal definition regular in the second argument:

$$
\begin{aligned}
&\boldsymbol{Rds}(a,0) = a\\
&\boldsymbol{Rds}(a,z,s) = \boldsymbol{Rds}(\boldsymbol{Rd}(a,z),s)\ .
\end{aligned}
$$

It should not be difficult to see that we have:

$$G_g(s_1) \wedge G_g(s_2) \wedge \boldsymbol{Rds}(a,s_1) = \boldsymbol{Rds}(a,s_2) > 0 \rightarrow s_1 = s_2 \tag{4}$$

$$b > 0 \rightarrow \boldsymbol{Rd}_g^k(a) = b \leftrightarrow \exists s(\,\boldsymbol{Rd}(a,s) = b \wedge L(s) = k \wedge G_g(s)\,)\ . \tag{5}$$

The relation between $\boldsymbol{Rds}$ and $\{e\}_g(x)$ is given by the following property:

$$\{e\}_g(x) \asymp y \leftrightarrow \exists s(\,\boldsymbol{Rds}((\boldsymbol{\lambda} e)(\boldsymbol{C}(x)),s) = \boldsymbol{C}(y) \wedge G_g(s)\,) \tag{6}$$

which holds because we have:

$$
\begin{aligned}
\{e\}_g(x) \asymp y \Leftrightarrow \boldsymbol{Rd}_g^*((\boldsymbol{\lambda} e)(\boldsymbol{C}(x))) \asymp y \overset{6.2.1(1)}{\Leftrightarrow} \exists k\, \boldsymbol{Rd}_g^k((\boldsymbol{\lambda} e)(\boldsymbol{C}(x))) = \boldsymbol{C}(y) \overset{(5)}{\Leftrightarrow}\\
\exists s(\,\boldsymbol{Rds}((\boldsymbol{\lambda} e)(\boldsymbol{C}(x)),s) = \boldsymbol{C}(y) \wedge G_g(s)\,)\ .
\end{aligned}
$$

6.2.4 Estimates of i in oracle calls g_i. Assume that $\{e\}_g(x) \asymp y$ holds. Then the list s from 6.2.3(6) is unique by 6.2.3(4). The elements $(i, z, v)\ \varepsilon\ s$ obviously depend on both e and x but we claim that the oracle numbers i depend only on the index e. In order to prove this we define a binary primitive recursive predicate $Apg(i, a)$ holding iff for some b the code $\boldsymbol{G}(i, b)$ occurs in a:

$$
\begin{aligned}
&Apg(i, \boldsymbol{S}(a)) \leftarrow Apg(i, a)\\
&Apg(i, \boldsymbol{P}(a)) \leftarrow Apg(i, a)\\
&Apg(i, \boldsymbol{D}(a, b, c)) \leftarrow Apg(i, a)\\
&Apg(i, \boldsymbol{D}(a, b, c)) \leftarrow \neg Apg(i, a) \land Apg(i, b)\\
&Apg(i, \boldsymbol{D}(a, b, c)) \leftarrow \neg Apg(i, a) \land \neg Apg(i, b) \land Apg(i, c)\\
&Apg(i, \boldsymbol{G}(j, a)) \leftarrow i = j\\
&Apg(i, \boldsymbol{G}(j, a)) \leftarrow i \neq j \land Apg(i, a)\\
&Apg(i, \boldsymbol{(\lambda}a\boldsymbol{)}(b)) \leftarrow Apg(i, a)\\
&Apg(i, \boldsymbol{(\lambda}a\boldsymbol{)}(b)) \leftarrow \neg Apg(i, a) \land Apg(i, b)\\
&Apg(i, \boldsymbol{F}(a)) \leftarrow Apg(i, a)\\
&Apg(i, \boldsymbol{A}(a, b)) \leftarrow Apg(i, a)\\
&Apg(i, \boldsymbol{A}(a, b)) \leftarrow \neg Apg(i, a) \land Apg(i, b)\ .
\end{aligned}
$$

It is not difficult to see that we have:

$$
\begin{aligned}
Apg(i, e) &\rightarrow i < e\\
Apg(i, a[b\boldsymbol{;}\, c]) &\rightarrow i < \max(a, b, c)\\
Apg(i, \boldsymbol{Rd}(a, z)) &\rightarrow i < a
\end{aligned}
$$

and thus also the following holds:

$$
\boldsymbol{Rds}(a, s) > 0 \land (i, x, v)\ \varepsilon\ s \rightarrow i < a\ . \tag{1}
$$

6.2.5 Enumeration theorem (Kleene). *For every class of functions $\mathcal{F}$ enumerated by g the function $\{e\}_g(x)$ enumerates partial $\mathcal{F}$-recursive functions. As a special case $\{e\}(x)$ enumerates the partial recursive functions.*

Proof. We first prove the theorem for $\{e\}_g(x)$. If an n-ary partial function f is $\mathcal{F}$-recursive then $f = \lambda\vec{x}.\tau$ for an $R^{\mathcal{F}}$-term τ and we have:

$$
\begin{aligned}
f(\vec{x}) \asymp y \overset{2.3.15}{\Leftrightarrow} (\lambda\vec{x}.\tau)(\underline{\vec{x}}) \blacktriangleright^{\mathcal{F}} \underline{y} \overset{6.2.1(2)}{\Leftrightarrow} \boldsymbol{Rd}^*_g(\boldsymbol{(\lambda}\ulcorner\tau\urcorner^n\boldsymbol{)}(\boldsymbol{C}(\vec{x}))) \asymp y \Leftrightarrow\\
\{\ulcorner\tau\urcorner^n\}_g(\vec{x}) \asymp y\ .
\end{aligned}
$$

Thus $\ulcorner\tau\urcorner^n$ is an index of f.

Vice versa, for any number e we wish to prove that the explicitly defined unary partial function $f(x) \simeq \{e\}_g(x)$ is $\mathcal{F}$-recursive. For all $i < e$ the unary functions $g_i(x) = g(i, x)$ are contractions of functions from $\mathcal{F}$ and so they are primitive recursive in $\mathcal{F}$. We explicitly define the binary function h as primitive recursive in $\mathcal{F}$:

$$
\begin{aligned}
&h(0,x) = g_0(x)\\
&h(1,x) = g_1(x)\\
&\quad\vdots\\
&h(e-1,x) = g_{e-1}(x)\ .
\end{aligned}
$$

If $\boldsymbol{Rds}((\boldsymbol{\lambda}e)(\boldsymbol{C}(x)),s) \asymp \boldsymbol{C}(y)$ holds then we have $i < e$ for every $(i,z,v)\ \varepsilon\ s$ from 6.2.4(1) and so clearly $G_g(s) \leftrightarrow G_h(s)$ holds. We then have

$$
\begin{aligned}
\{e\}_g(x) \asymp y \overset{6.2.3(6)}{\Leftrightarrow} \exists s(\,\boldsymbol{Rds}((\boldsymbol{\lambda}e)(\boldsymbol{C}(x)),s) = \boldsymbol{C}(y) \wedge G_g(s)\,) \Leftrightarrow\\
\exists s(\,\boldsymbol{Rds}((\boldsymbol{\lambda}e)(\boldsymbol{C}(x)),s) = \boldsymbol{C}(y) \wedge G_h(s)\,) \overset{6.2.3(6)}{\Leftrightarrow} \{e\}_h(x) \asymp y\ ,
\end{aligned}
$$

i.e. $\{e\}_g(x) \simeq \{e\}_h(x)$ and the last is a partial $\mathcal{F}$-recursive function.

The partial function $\{e\}(x)$ enumerates from its definition and from the just proved first part of the theorem the partial fuctions recursive in the class $\{Z\}$ which are exactly the partial recursive functions. ■

6.2.6 A non-computable function. The classes of partial recursive and partially computable functions are identical by Corollary 6.1.12. Since every partially computable function has an index by the Enumeration theorem there are at most denumerably many computable functions. From the set-theoretical theorem of Cantor we know that there are non-denumerably many functions over $\mathbb{N}$. This means that non-denumerably many functions are not computable. Can we find a concrete such function?

Suppose that the partial enumeration function $\{e\}(x)$ is total. This means that it is a recursive function and so is the following explicitly defined function:

$$f(x) = \{x\}(x) + 1\ .$$

The function f has an index e by the Enumeration theorem and we obtain a contradiction:

$$f(e) \overset{\text{def.}}{=} \{e\}(e) + 1 \overset{\text{enum.}}{=} f(e) + 1\ .$$

The enumeration function $\{e\}(x)$ is explicitly defined from the partial recursive function $\boldsymbol{Rd}^*_g$ which thus cannot be a total function.

Denote by U the completion of the partial function $\{e\}(x)$. If U were a recursive function then so would be $f(x) = U(x,x) + 1$ and we would obtain a contradiction in the same way as above. The function U is an example of a function which is not computable. It is a universal function for recursive functions because for any index e of a recursive function f we have $f(\vec{x}) = U(e,\vec{x})$.

6.2.7 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 and by Corollary 6.1.12 decidable predicates are exactly the recursive predicates. Are there any undecidable predicates?

Consider the binary predicate $W_e(x)$ satisfying:

$$W_e(x) \leftrightarrow \{e\}(x){\downarrow} \ . \tag{1}$$

For any computable n-ary partial function f take any of its indices e. We have $W_e(\vec{x})$ iff $\{e\}(\vec{x}){\downarrow}$ iff $f(\vec{x}){\downarrow}$ iff the computation of f for $\vec{x}$ stops. $W_e(x)$ is called the *halting* predicate because of this property.

Suppose that the halting predicate were computable. It would then be recursive and the following clausal definition would define a binary recursive function:

$$\begin{aligned} &U(e,x) = y \leftarrow W_e(x) \land \{e\}(x) \asymp y \\ &U(e,x) = 0 \leftarrow W_e(x) \quad . \end{aligned}$$

But U is the completion of the enumeration function $\{e\}(x)$ of which we know from Par. 6.2.6 that it is not recursive. This means that the halting predicate is undecidable. We say that we have *reduced* the problem of computability of U to that of $W_e(x)$.

We give yet another proof of undecidability of the halting predicate. Suppose that the predicate is recursive and consider the partial recursive function f defined as follows:

$$\begin{aligned} &f(x) \asymp 0 \leftarrow \neg W_x(x) \\ &f(x) \asymp 1 \leftarrow W_x(x) \land f(x) \asymp y \ . \end{aligned}$$

The partial function f has an index e and we obtain a contradiction as follows: $f(e){\downarrow}$ iff, by the definition of f, $f(e) \asymp 0$ iff not $W_e(e)$ iff, by the definition of the halting predicate, $\{e\}(e){\uparrow}$ iff, because e is an index of f, $f(e){\uparrow}$.

6.2.8 Uniform halting. Consider the following explicitly defined unary predicate *Tot*:

$$Tot(e) \leftrightarrow \forall x\, W_e(x) \ .$$

We claim that for every n-ary partial recursive function f and for any of its indices $e = \ulcorner\tau\urcorner$ where $f = \lambda\vec{x}.\tau$ we have $Tot(e)$ iff f is total. In the direction $\rightarrow$ assume $Tot(e)$ and let $\vec{x}$ be any numbers. We have $W_e(\vec{x})$ and so $\{e\}(\vec{x}){\downarrow}$, i.e. $f(\vec{x}){\downarrow}$.

For the proof in the direction $\leftarrow$ take $g(x,i) = 0$, assume that f is total, and take any number x. From $f(H\,T^0(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x)){\downarrow}$ we have

$$\begin{aligned} &\{e\}(H\,T^0(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x)){\downarrow} \Rightarrow \\ &\qquad \boldsymbol{Rd}^*_g((\boldsymbol{\lambda}e)(\boldsymbol{C}(H\,T^0(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x)))){\downarrow} \Rightarrow \\ &\qquad \boldsymbol{Rd}^*_g\, \boldsymbol{Rd}_g((\boldsymbol{\lambda}e)(\boldsymbol{C}(H\,T^0(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x)))){\downarrow} \Rightarrow \\ &\qquad \boldsymbol{Rd}^*_g(e[e;\boldsymbol{C}(H\,T^0(x), \ldots, H\,T^{n-2}(x), T^{n-1}(x))]){\downarrow} \overset{*}{\Rightarrow} \\ &\qquad \boldsymbol{Rd}^*_g(e[e;\boldsymbol{C}(x)]){\downarrow} \Rightarrow \boldsymbol{Rd}^*_g\, \boldsymbol{Rd}_g((\boldsymbol{\lambda}e)(\boldsymbol{C}(x))){\downarrow} \Rightarrow \\ &\qquad \boldsymbol{Rd}^*_g((\boldsymbol{\lambda}e)(\boldsymbol{C}(x))){\downarrow} \Rightarrow \{e\}(x){\downarrow} \Rightarrow W_e(x) \Rightarrow Tot(e) \end{aligned}$$

where the step marked by $*$ follows from the definition of the arithmetized substitution function in Par. 4.3.3.

That the predicate *Tot* is undecidable can be proved by reducing the halting problem to the problem of recognizing a total function:

$$W_e(x) \leftrightarrow \mathit{Tot}((\boldsymbol{\lambda} e)(\boldsymbol{C}(x))) .$$

The equivalence follows from the fact that $(\boldsymbol{\lambda} e)(\boldsymbol{C}(x))$ is an index of a total function iff $\{e\}(x)\!\downarrow$.

6.2.9 Kleene's predicate T_g. Kleene has arithmetized systems of Herbrand-Gödel-Kleene equations and defined a ternary primitive recursive predicate $T(e,x,y)$ holding if the system of equations for a function f coded by e leads to a value for $f(x)$ by a proof (computation) coded by y.

We present an equivalent ternary predicate T_g in a form relativized to a class of functions $\mathcal{F}$ enumerated by g. The predicate T_g is primitive recursive in g by explicit clausal definition:

$$T_g(e,x,(s,z)) \leftarrow \boldsymbol{Rds}((\boldsymbol{\lambda} e)(\boldsymbol{C}(x)),s) = \boldsymbol{C}(z) \wedge G_g(s)$$

From 6.2.3(6) we obtain:

$$\{e\}_g(x) \asymp z \leftrightarrow \exists s\, T_g(e,x,(s,z)) \tag{1}$$

Note that if g enumerates the empty class $\emptyset$, i.e. if $g(i,x) = 0$, then T_g is a primitive recursive predicate and we have:

$$W_e(x) \leftrightarrow \exists y\, T_g(e,x,y) . \tag{2}$$

The following theorem shows that the predicate T_g turns the theorem 5.2.2 asserting that every partial $\mathcal{F}$-recursive function can be obtained by one minimalization of a predicate primitive recursive in $\mathcal{F}$ into a uniform form where the predicate is always T_g.

6.2.10 Normal form theorem (Kleene). *For every class of functions $\mathcal{F}$ enumerated by g, any n-ary partial $\mathcal{F}$-recursive function f, and any of its indices e we have*

$$f(\vec{x}) \simeq T\,\mu_y[T_g(e,(\vec{x}),y)] .$$

Proof. We have $f(\vec{x}) \asymp z$ iff $\{e\}_g(\vec{x}) \asymp z$ iff, by 6.2.9(1), $T_g(e,(\vec{x}),(s,z))$ holds for some s. If also $T_g(e,(\vec{x}),(s_1,z_1))$ holds then we have $z = z_1$ and so $f(\vec{x}) \simeq T\,\mu_y[T_g(e,(\vec{x}),y)]$. ∎

6.2.11 Parameter function p/x. Consider the following explicit definition of a binary primitive recursive function e/x:

$$e/x = (\boldsymbol{\lambda} e)(\boldsymbol{A}(\boldsymbol{C}(x), \boldsymbol{X}(0,1)) \ .$$

The function e/x, called *parameter* function, operates on indices and it takes an index e of a binary partial recursive function g together with a number x and yields an index of the unary partial recursive function f such that $f(y) \simeq g(x,y)$ holds. In other words, we have:

$$\{e/x\}(y) \simeq \{e\}(x,y) \ . \tag{1}$$

Indeed, we take any numbers e, x, y, and z and for $g(i,x) = 0$ we have

$$\begin{aligned}
\{e/x\}(y) \simeq{} & \boldsymbol{Rd}^*_g((\boldsymbol{\lambda}(e/x))(\boldsymbol{C}(y))) \simeq \boldsymbol{Rd}^*_g\,\boldsymbol{Rd}_g((\boldsymbol{\lambda}(e/x))(\boldsymbol{C}(y))) \simeq \\
& \boldsymbol{Rd}^*_g((e/x)[(e/x); \boldsymbol{C}(y)]) \simeq \\
& \boldsymbol{Rd}^*_g((\boldsymbol{\lambda} e)(\boldsymbol{A}(\boldsymbol{C}(x), \boldsymbol{X}(0,1))[(e/x); \boldsymbol{C}(y)]) \simeq \\
& \boldsymbol{Rd}^*_g((\boldsymbol{\lambda} e)(\boldsymbol{A}(\boldsymbol{C}(x), \boldsymbol{C}(y)))) \simeq \boldsymbol{Rd}^*_g\,\boldsymbol{Rd}_g((\boldsymbol{\lambda} e)(\boldsymbol{A}(\boldsymbol{C}(x), \boldsymbol{C}(y)))) \simeq \\
& \boldsymbol{Rd}^*_g(e[e; \boldsymbol{A}(\boldsymbol{C}(x), \boldsymbol{C}(y))]) \overset{*}{\simeq} \boldsymbol{Rd}^*_g(e[e; \boldsymbol{C}(x,y)]) \simeq \\
& \boldsymbol{Rd}^*_g\,\boldsymbol{Rd}_g((\boldsymbol{\lambda} e)(\boldsymbol{C}(x,y))) \simeq \boldsymbol{Rd}^*_g((\boldsymbol{\lambda} e)(\boldsymbol{C}(x,y))) \simeq \{e\}(x,y)
\end{aligned}$$

where in the step marked by $(*)$ we use the easily proved property

$$\mathit{Sel}(i,n,\boldsymbol{A}(\boldsymbol{C}(x), \boldsymbol{C}(y))) = \mathit{Sel}(i,n,\boldsymbol{C}(x,y))$$

of the selection function (see Par. 4.3.3) and so we have $e[e; \boldsymbol{A}(\boldsymbol{C}(x), \boldsymbol{C}(y))] = e[e; \boldsymbol{C}(x,y)]$.

The parameter function is used in the proofs of the following theorems.

6.2.12 *S-m-n* theorem (Kleene). *For every computable $m+n$-ary partial function f there is an m-ary primitive recursive function s such that for every $\vec{x}$ and $\vec{y}$ we have:*

$$f(\vec{x},\vec{y}) \simeq \{s(\vec{x})\}(\vec{y}) \ . \tag{1}$$

Proof. Consider the following explicit definition of an auxiliary $(n+1)$-ary partial recursive function g:

$$g((\vec{x}),\vec{y}) \asymp z \leftarrow f(\vec{x},\vec{y}) \asymp z \ .$$

The partial function g has an index e by the Enumeration theorem and we explicitly define the primitive recursive function s as follows:

$$s(\vec{x}) = e/(\vec{x}) \ .$$

We then have

$$\{s(\vec{x})\}(\vec{y}) \simeq \{e/(\vec{x})\}(\vec{y}) \overset{6.2.11(1)}{\simeq} \{e\}((\vec{x}),\vec{y}) \simeq g((\vec{x}),\vec{y}) \simeq f(\vec{x},\vec{y}) \ . \qquad \blacksquare$$

6.2.13 Second recursion theorem (Kleene). *For every* $(n+1)$*-ary partial recursive function* g *there is an* n*-ary partial recursive function* f *with an index* e *and such that the following holds for all* $\vec{x}$*:*

$$f(\vec{x}) \simeq g(e, \vec{x}) \ . \tag{1}$$

Proof. Consider the auxiliary $(n+1)$-ary partial recursive function h explicitly defined as:

$$h(p, \vec{x}) \asymp y \leftarrow g(p/p, \vec{x}) \asymp y \ .$$

The partial function h has an index p by the Enumeration theorem. We set $e = p/p$ and explicitly define

$$f(\vec{x}) \asymp y \leftarrow \{e\}(\vec{x}) \asymp y$$

as an n-ary partial recursive function with an index e. We have

$$f(\vec{x}) \simeq \{e\}(\vec{x}) \simeq \{p/p\}(\vec{x}) \overset{6.2.11(1)}{\simeq} \\ \{p\}(p, \vec{x}) \simeq h(p, \vec{x}) \simeq g(p/p, \vec{x}) \simeq g(e, \vec{x}) \ . \qquad \blacksquare$$

Chapter 7

Beyond Computability

We are now going to study the effect of unbounded quantifiers in explicit definitions of predicates. We have seen in 6.2.9(2) that one unbounded quantification changes a primitive recursive predicate T_g to the non-computable halting predicate. We will see that additional unbounded quantifiers construct larger and and larger classes of predicates.

7.1 Recursively Enumerable 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.

7.1.1 Semicomputable predicates. For a class of functions $\mathcal{F}$ enumerated by g an n-ary predicate R is *positively calculable in* $\mathcal{F}$, or *semicomputable in* $\mathcal{F}$, if it is the domain of an n-ary partial $\mathcal{F}$-recursive function f i.e. we have

$$R(\vec{x}) \leftrightarrow f(\vec{x})\downarrow .$$

A *semicomputable* predicate is semicomputable in the empty class $\emptyset$.

Note that for any $\vec{x}$ we are able to confirm effectively that $R(\vec{x})$ holds by computing $f(\vec{x})$. However, we are not able to refute $R(\vec{x})$ by this computation because it goes forever.

If R is the domain of an n-ary partial $\mathcal{F}$-recursive f with an index e we explicitly define for $j < e$ the unary functions $g_j(x) = g(j, x)$ as primitive recursive in $\mathcal{F}$. Note that g_j are contractions of functions from $\mathcal{F}$. We then explicitly define a unary predicate $G_{e,g}$:

$$G_{e,g}(s) \leftrightarrow \forall i \forall x \forall v(\,(i, x, v)\ \varepsilon\ s \rightarrow \bigvee_{j=0}^{e-1} (i = j \wedge g_j(x) = v)\,)\ .$$

The universal quantifiers are bounded because, for instance, we have $i < s$ and so $G_{e,g}$ is primitive recursive in $\mathcal{F}$. Moreover, we have

$$R(\vec{x}) \leftrightarrow \exists s \exists z(\, \boldsymbol{Rds}((\boldsymbol{\lambda} e)(\boldsymbol{C}(\vec{x})), s) = \boldsymbol{C}(z) \wedge G_{e,g}(s)\,) \tag{1}$$

because

$$\begin{aligned} R(\vec{x}) \Leftrightarrow f(\vec{x})\downarrow \Leftrightarrow \exists z\, \{e\}_g(\vec{x}) \asymp z \overset{6.2.9(1)}{\Leftrightarrow} \exists s \exists z\, T_g(e, (\vec{x}), (s, z)) \Leftrightarrow \\ \exists s \exists z(\, \boldsymbol{Rds}((\boldsymbol{\lambda} e)(\boldsymbol{C}(\vec{x})), s) = \boldsymbol{C}(z) \wedge G_g(s)\,) \overset{6.2.4(1)}{\Leftrightarrow} \\ \exists s \exists z(\, \boldsymbol{Rds}((\boldsymbol{\lambda} e)(\boldsymbol{C}(\vec{x})), s) = \boldsymbol{C}(z) \wedge G_{e,g}(s)\,)\ . \end{aligned}$$

7.1.2 Recursively enumerable predicates. For a class of functions $\mathcal{F}$ an n-ary predicate R is *recursively enumerable in* $\mathcal{F}$, or shortly *r.e. in* $\mathcal{F}$, if it is *empty*, i.e. if $\forall \vec{x}\, \neg R(\vec{x})$ holds, or if its contraction $\langle R \rangle$ can be *enumerated* by an unary $\mathcal{F}$-recursive function f, i.e. if $\langle R \rangle$ is the range of f:

$$\langle R \rangle(x) \leftrightarrow \exists y\, f(y) = x\ .$$

From the last we get:

$$R(\vec{x}) \leftrightarrow \exists y\, f(y) = \vec{x}\ .$$

Recursively enumerable predicates are recursively enumerable in the empty class $\emptyset$.

If the enumerating function is primitive recursive we call recursively enumerable predicates *primitively recursively enumerable.*

7.1.3 Unbounded quantification. The n-ary predicate predicate R is the *existential projection* of an $(n+1)$-ary predicate Q if

$$R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x})\ .$$

R is the *universal projection* of Q if

$$R(\vec{x}) \leftrightarrow \forall y\, Q(y, \vec{x})\ .$$

The following important theorem connects the semicomputable, recursively enumerable, and existential projections of recursive predicates.

7.1.4 Theorem (Characterization of r.e. predicates) *For any class of functions $\mathcal{F}$ the following are identical classes:*

1. *predicates semicomputable in $\mathcal{F}$,*

2. *existential projections of predicates primitive recursive in $\mathcal{F}$,*

3. *existential projections of predicates recursive in $\mathcal{F}$,*

4. *predicates primitively recursively enumerable in $\mathcal{F}$,*

5. *predicates recursively enumerable in $\mathcal{F}$.*

Proof. Let $\mathcal{F}$ be enumerated by g.

(1) $\subseteq$ (2): If R is an n-ary predicate semicomputable in $\mathcal{F}$ then we have 7.1.1(1) for some e. The explicit clausal definition

$$Q((s,z),\vec{x}) \leftarrow \boldsymbol{Rds}(\boldsymbol{(\lambda}e)(\boldsymbol{C}(\vec{x})), s) = \boldsymbol{C}(z) \wedge G_{e,g}(s)$$

defines the $(n{+}1)$-ary Q as a predicate primitive recursive in $\mathcal{F}$ for which we have $R(\vec{x}) \leftrightarrow \exists y\, Q(y,\vec{x})$.

(2) $\subseteq$ (3) and (4) $\subseteq$ (5) hold trivially.

(2) $\subseteq$ (4) and (3) $\subseteq$ (5): Let R be an n-ary existential projection $R(\vec{x}) \leftrightarrow \exists y\, Q(y,\vec{x})$ of a predicate Q primitive recursive (recursive) in $\mathcal{F}$. If R is empty there is nothing to prove. Otherwise, there is a number s such that $\langle R\rangle(s)$ holds. We note that we have $\langle R\rangle(v) \leftrightarrow \exists y \langle Q\rangle(y,v)$ and we define a unary function f primitive recursive (recursive) in $\mathcal{F}$ by the following explicit clausal definition:

$$\begin{aligned}
&f(0) = s\\
&f(y,v) = v \leftarrow \langle Q\rangle(y,v)\\
&f(y,v) = s \leftarrow \neg\langle Q\rangle(y,v)\ .
\end{aligned}$$

We claim that the function f enumerates $\langle R\rangle$, i.e. that we have

$$\langle R\rangle(v) \leftrightarrow \exists z\, f(z) = v\ .$$

If $\langle R\rangle(v)$ holds then we have $\langle Q\rangle(y,v)$ for some y and it suffices to take $z = y,v$.

Vice versa, it suffices to show $\langle R\rangle\, f(z)$ for all z. So take any z and consider two cases. If $z = 0$ then, since $f(0) = s$, we have $\langle R\rangle\, f(z)$. If $z = y,v$ for some y and v we consider two subcases. If $\langle Q\rangle(y,v)$ holds then also $\langle R\rangle(v)$ holds and we have $f(z) = v$. If $\langle Q\rangle(y,v)$ does not hold then $f(z) = s$ and we have $\langle R\rangle\, f(z)$ again.

(5) $\subseteq$ (1): Let R be an n-ary predicate r.e. in $\mathcal{F}$. If R is empty then it is the domain of the nowhere defined n-ary partial function $\emptyset^n$ which is $\mathcal{F}$-recursive by the the explicit definition $\emptyset^n(\vec{x}) \simeq \emptyset^n(\vec{x})$.

If $\langle R\rangle$ is enumerated by an unary $\mathcal{F}$-recursive function h then we define an auxiliary binary partial $\mathcal{F}$-recursive function f_1;

$$\begin{aligned}
&f_1(z,v) \asymp z \leftarrow h(z) = v\\
&f_1(z,v) \asymp y \leftarrow h(z) \neq v \wedge f_1(z+1,v) \asymp y\ .
\end{aligned}$$

It now suffices explicitly define $f(\vec{x}) \simeq f_1(0,\vec{x})$ as a partial $\mathcal{F}$-recursive function because we clearly have $f(\vec{x}) \simeq \mu_z[h(z) = \vec{x}]$ and so $f(\vec{x})\!\downarrow$ iff there is a z such that $h(z) = \vec{x}$ iff $R(\vec{x})$ holds. ■

7.1.5 The halting predicate is recursively enumerable. The Kleene's predicate T_g is primitive recursive for $g(i,x)=0$ and so is the ternary predicate $R(y,e,x) \leftrightarrow T_g(e,x,y)$. We have

$$W_e(x) \overset{6.2.9(2)}{\Leftrightarrow} \exists y\, T_g(e,x,y) \Leftrightarrow \exists y\, R(y,e,x)\ . \tag{1}$$

This makes the halting predicate a recursively enumerable predicate by the Characterization theorem 7.1.4. The predicate is not recursive by Par. 6.2.7.

7.1.6 Theorem (Post) *For every class of functions $\mathcal{F}$ a predicate R is $\mathcal{F}$-recursive iff both R and its complement R^c are r.e. in $\mathcal{F}$.*

Proof. Take any n-ary predicate R such that both R and R^c are r.e. in $\mathcal{F}$. By the Characterization theorem 7.1.4 we have

$$R(\vec{x}) \leftrightarrow \exists y\, Q_1(y,\vec{x})$$
$$R^c(\vec{x}) \leftrightarrow \exists y\, Q_2(y,\vec{x})$$

for some $\mathcal{F}$-recursive predicates Q_1 and Q_2. Since the sentence

$$\forall\vec{x}\exists y(Q_1(y,\vec{x}) \lor Q_2(y,\vec{x}))$$

is true, we can derive R as a $\mathcal{F}$-recursive predicate by regular minimalization:

$$R(\vec{x}) \leftrightarrow Q_1(\mu_y[Q_1(y,\vec{x}) \lor Q_2(y,\vec{x})],\vec{x})\ .$$

Vice versa, take any n-ary $\mathcal{F}$-recursive predicate R. For explictly defined $\mathcal{F}$-recursive predicate Q:

$$Q(y,\vec{x}) \leftrightarrow R(\vec{x})$$

we clearly have $R(\vec{x}) \leftrightarrow \exists y Q(y,\vec{x})$ and $R^c(\vec{x}) \leftrightarrow \exists y Q^c(y,\vec{x})$ and so by the Characterization theorem R is r.e. in $\mathcal{F}$. ■

7.2 Arithmetical Hierarchy

We will show that a string of alternating universal and existential projections starting from recursive predicates builds a hierarchy of progressively larger classes of predicates called the *Arithmetical* hierarchy.

7.2.1 Arithmetical predicates. Let $\mathcal{F}$ be a class of functions. An m-ary ($m \geq 1$) predicate $R(\vec{x})$ is *arithmetical in* $\mathcal{F}$ if there is an $(n+m)$-ary $\mathcal{F}$-recursive predicate $Q(\vec{y},\vec{x})$ and we get R from Q by $n \geq 0$ projections, i.e. we have

$$R(\vec{x}) \leftrightarrow \mathcal{Q}_n y_n \mathcal{Q}_{n-1} y_{n-1} \cdots \mathcal{Q}_2 y_2 \mathcal{Q}_1 y_1\, Q(y_1,y_2,\ldots,y_{n-1},y_n,\vec{x}) \tag{1}$$

with each $\mathcal{Q}_i$ either $\forall$ or $\exists$. A predicate is *arithmetical* if it is arithmetical in $\emptyset$.

The string of quantifiers $\mathcal{Q}_n y_n \mathcal{Q}_{n-1} y_{n-1} \cdots \mathcal{Q}_2 y_2 \mathcal{Q}_1 y_1$ is called the *prefix* and $Q(y_1, y_2, \ldots, y_{n-1}, y_n, \vec{x})$ the *matrix* of (1). We call the prefix *alternating* if for no $i < n$ two successive quantifiers $\mathcal{Q}_i$ and $\mathcal{Q}_{i+1}$ are of the same kind, i.e. both existential or both universal. We will see that the length of the alternating prefix is crucial because it measures how far is the definition removed from being recursive.

7.2.2 Classes of predicates Σ_n and Π_n. For every class of functions $\mathcal{F}$ and for every $n \geq 1$ we define the class of predicates Σ_n *(Π_n) in* $\mathcal{F}$ to consist of those predicates arithmetical in $\mathcal{F}$ whose definitions have alternating prefixes with n-quantifiers and starting with existential (universal) quantifiers. Thus if n is odd the Σ_n in $\mathcal{F}$ predicates have a form:

$$R(\vec{x}) \leftrightarrow \exists y_n \forall y_{n-1} \cdots \forall y_2 \exists y_1\, Q(y_1, y_2, \cdots, y_{n-1}, y_n, \vec{x})$$

and if n is even the Σ_n in $\mathcal{F}$ predicates have a form:

$$R(\vec{x}) \leftrightarrow \exists y_n \forall y_{n-1} \cdots \exists y_2 \forall y_1\, Q(y_1, y_2, \cdots, y_{n-1}, y_n, \vec{x}) \; .$$

For Π_n in $\mathcal{F}$ predicates the existential quantifiers are replaced by universal ones and vice versa.

The class of predicates Δ_n *in* $\mathcal{F}$ consists of predicates which are in the classes Σ_n in $\mathcal{F}$ and Π_n in $\mathcal{F}$ at the same time.

Note that the predicates Σ_1 (Π_1) in $\mathcal{F}$ are existential (universal) projections of $\mathcal{F}$-recursive predicates and for $n \geq 2$ every predicate Σ_n (Π_n) in $\mathcal{F}$ is an existential (universal) projection of a predicate Π_{n-1} (Σ_{n-1}) in $\mathcal{F}$.

Σ_n, Π_n, and Δ_n predicates are the corresponding predicates Σ_n, Π_n, and Δ_n in the empty class $\emptyset$.

Note that by the Characterization theorem 7.1.4 the predicates Σ_1 in $\mathcal{F}$ are exactly those which are r.e. in $\mathcal{F}$.

7.2.3 Reducibility of predicates. An n-ary predicate R is *$\mathcal{F}$-reducible* to an m-ary predicate Q if for all $\vec{x}$ we have

$$R(\vec{x}) \leftrightarrow Q(f_1(\vec{x}), \ldots, f_m(\vec{x})) \tag{1}$$

for some $\mathcal{F}$-recursive functions $f_1, \ldots, f_m$. R is *reducible* to Q if it is $\emptyset$-reducible to Q.

7.2.4 Theorem *If the predicate R is $\mathcal{F}$-reducible to Q and Q is $\mathcal{F}$-recursive, Σ_n in $\mathcal{F}$, or Π_n in $\mathcal{F}$ then also R is in the corresponding class.*

Proof. If Q is $\mathcal{F}$-recursive then R is $\mathcal{F}$-recursive by explicit definition 7.2.3(1). For the other two cases we prove by induction on n that for all R which is $\mathcal{F}$-reducible to a predicate Σ_n (Π_n) in $\mathcal{F}$ also R is Σ_n (Π_n) in $\mathcal{F}$.

If Q is Σ_n in $\mathcal{F}$ then $Q(\vec{y}) \leftrightarrow \exists z Q_1(z,\vec{y})$ for some Q_1 and we have:

$$R(\vec{x}) \Leftrightarrow Q(f_1(\vec{x}),\ldots,f_m(\vec{x})) \Leftrightarrow \exists z Q_1(z,f_1(\vec{x}),\ldots,f_m(\vec{x})) \overset{(*)}{\Leftrightarrow} \exists z Q_2(z,\vec{x})$$

where in the step marked by $(*)$ the predicate Q_2 which $\mathcal{F}$-reduces to Q_1 is $\mathcal{F}$-recursive when $n=1$ and Π_{n-1} in $\mathcal{F}$ by IH when $n>1$. The proof when Q is Π_n is similar. ■

7.2.5 Contraction of quantifiers. Two adjacent quantifiers of the same kind can be *contracted* to a single quantifier because we have:

$$\exists x \exists y \phi[x,y] \leftrightarrow \exists z \phi[H(z),T(z)] \tag{1}$$
$$\forall x \forall y \phi[x,y] \leftrightarrow \forall z \phi[H(z),T(z)]\ . \tag{2}$$

The property (2) follows from (1). In the direction $\rightarrow$ of (1) we assume $\phi[x,y]$ for some x and y and for $z = x,y$ we have $\phi[H(z),T(z)]$. In the direction $\leftarrow$ we assume $\phi[H(z),T(z)]$ for some z. For $x = H(z)$ and $y = T(z)$ we have $\phi[x,y]$.

7.2.6 Theorem *Every predicate arithmetical in $\mathcal{F}$ is $\mathcal{F}$-recursive, Σ_n in $\mathcal{F}$, or Π_n in $\mathcal{F}$.*

Proof. If R is arithmetical in $\mathcal{F}$ and not $\mathcal{F}$-recursive then R can be expressed in a form 7.2.1(1) for some $n>0$ and we successively contract all adjacent quantifiers of the same kind. ■

R_1,R_2	$\exists y R_1$	$\forall y R_1$	$R_1 \vee R_2$	$R_1 \wedge R_2$	$\neg R_1$	$\forall z{\le}y\, R_1$	$\exists z{\le}y\, R_1$
Σ_n	Σ_n	Π_{n+1}	Σ_n	Σ_n	Π_n	Σ_n	Σ_n
Π_n	Σ_{n+1}	Π_n	Π_n	Π_n	Σ_n	Π_n	Π_n
Δ_n	Σ_n	Π_n	Δ_n	Δ_n	Δ_n	Δ_n	Δ_n

Figure 7.1: Closure properties of arithmetical predicates under logical operations

7.2.7 Closure properties of arithmetical predicates. The table in Fig. 7.1 summarizes the closure properties of logical operations on arithmetical predicates. If the predicates R_1 and R_2 are as shown in the rows two to four of the first column then the classification of the predicate obtained by the logical operations identified in the first row is shown in the corresponding row and column.

We prove the closure properties under quantifier projections $\exists y R_1$, $\forall y R_1$ in Par. 7.2.8, under boolean operations $R_1 \vee R_2$, $R_1 \wedge R_2$, $\neg R_1$ in Par. 7.2.9, and under bounded quantification $\forall z{\le}y\, R_1$, $\exists z{\le}y\, R_1$ in Par. 7.2.10. The closure properties *relativize*, i.e. hold also for the predicates relative to any class of functions $\mathcal{F}$.

7.2.8 Closures of arithmetical predicates under quantifier projections. Let $\mathcal{F}$ be any class of functions. If $R_1(y,\vec{x})$ is Σ_n in $\mathcal{F}$ then its universal projection $R(\vec{x}) \leftrightarrow \forall y R_1(y,\vec{x})$ is Π_{n+1} in $\mathcal{F}$ directly from the definition.

If $R(\vec{x}) \leftrightarrow \exists y R_1(y,\vec{x})$ is the existential projection of R_1 then, since we have $R_1(y,\vec{x}) \leftrightarrow \exists z\, R_2(z,y,\vec{x})$ for a predicate R_2 which is Π_{n-1} in $\mathcal{F}$ if $n > 1$ and $\mathcal{F}$-recursive otherwise, we obtain:

$$R(\vec{x}) \Leftrightarrow \exists y R_1(y,\vec{x}) \Leftrightarrow \exists y \exists z R_2(z,y,\vec{x}) \overset{7.2.5(1)}{\Leftrightarrow} \exists v R_2(T(v),H(v),\vec{y}) \overset{*}{\Leftrightarrow} \exists v R_3(v,\vec{y})$$

where $(*)$ marks an explicit definition of a predicate R_3. By Thm. 7.2.4 R_3 is Π_{n-1} in $\mathcal{F}$ if $n > 1$ and $\mathcal{F}$-recursive otherwise. This means that R is Σ_n.

The closure properties of Π_n-predicates under quantifier projections are proved similarly.

If $R_1(y,\vec{x})$ is Δ_n in $\mathcal{F}$ then its existential projection $R(\vec{x}) \leftrightarrow \exists y R_1(y,\vec{x})$ is Σ_n in $\mathcal{F}$ because R_1 is Σ_n in $\mathcal{F}$. Its universal projection is Π_n in $\mathcal{F}$ by similar reasoning.

7.2.9 Closures of arithmetical predicates under boolean operations. The closure properties of arithmetical predicates under boolean operations rely on the following well-known prenex rules for bringing quantifiers to the front of a formula:

$$\begin{aligned}
\exists x \phi_1[x] \lor \exists x \phi_2[x] &\leftrightarrow \exists x(\phi_1[x] \lor \phi_2[x]) \\
\mathcal{Q}x \phi_1[x] \lor \phi_2 &\leftrightarrow \mathcal{Q}x(\phi_1[x] \lor \phi_2) \\
\phi_1 \lor \mathcal{Q}x \phi_2[x] &\leftrightarrow \mathcal{Q}x(\phi_1 \lor \phi_2[x]) \\
\forall x \phi_1[x] \land \forall x \phi_2[x] &\leftrightarrow \forall x(\phi_1[x] \land \phi_2[x]) \\
\mathcal{Q}x \phi_1[x] \land \phi_2 &\leftrightarrow \mathcal{Q}x(\phi_1[x] \land \phi_2) \\
\phi_1 \land \mathcal{Q}x \phi_2[x] &\leftrightarrow \mathcal{Q}x(\phi_1 \land \phi_2[x](x)) \\
\neg \exists x \phi[x] &\leftrightarrow \forall x \neg \phi[x] \\
\neg \forall x \phi[x] &\leftrightarrow \exists x \neg \phi[x] \ .
\end{aligned}$$

The symbol $\mathcal{Q}x$ in the equivalences stands for $\forall x$ or $\exists x$ and the variable x is free in the formulas ϕ_i only if it is explicitly indicated.

We leave to the reader the straightforward relativization of the closure properties to any class of functions $\mathcal{F}$.

If $R_1(\vec{x})$ and $R_2(\vec{x})$ are both Σ_n (Π_n) then we show that $R(\vec{x}) \leftrightarrow R_1(\vec{x}) \lor R_2(\vec{x})$ is Σ_n (Π_n) by induction on n. Both in the base and inductive case we have for $R_i \in \Sigma_n$:

$$\begin{gathered}
R(\vec{x}) \Leftrightarrow R_1(\vec{x}) \lor R_2(\vec{x}) \Leftrightarrow \exists y Q_1(y,\vec{x}) \lor \exists y\, Q_2(y,\vec{x}) \Leftrightarrow \\
\exists y(Q_1(y,\vec{x}) \lor Q_2(y,\vec{x})) \overset{*1}{\Leftrightarrow} \exists y Q_3(y,\vec{x})
\end{gathered}$$

and we have for $R_i \in \Pi_n$:

$$R(\vec{x}) \Leftrightarrow R_1(\vec{x}) \vee R_2(\vec{x}) \Leftrightarrow \forall y Q_1(y,\vec{x}) \vee \forall y\, Q_2(y,\vec{x}) \Leftrightarrow$$
$$\forall y_1 \forall y_2 (Q_1(y_1,\vec{x}) \vee Q_2(y_2,\vec{x})) \Leftrightarrow$$
$$\forall y (Q_1(H(y),\vec{x}) \vee Q_2(T(y),\vec{x})) \overset{*_2}{\Leftrightarrow} \forall y Q_4(y,\vec{x}) \ .$$

In the step marked by $*_j$ we have explicitly defined the predicate Q_{j+2} which is recursive if $n = 1$ and if $n > 1$ we have $Q_3 \in \Pi_{n-1}$ and $Q_4 \in \Sigma_{n-1}$ by IH.

If $R_1(\vec{x})$ and $R_2(\vec{x})$ are both Δ_n then the fact that their disjunction is a Δ_n-predicate follows directly from the closures under disjunction of Σ_n and Π_n-predicates.

The closure properties of arithmetic predicates under conjunction as shown in the column five of the table 7.1 are proved similarly as closures under disjunction.

We show by induction on n that the complements of Σ_n (Π_n) predicates are Π_n (Σ_n) predicates. If $R_1 \in \Sigma_n$ then both in the base and inductive case we have:

$$R_1^c(\vec{x}) \Leftrightarrow \neg R_1(\vec{x}) \Leftrightarrow \neg\exists y Q(y,\vec{x}) \Leftrightarrow \forall y \neg Q(y,\vec{x}) \Leftrightarrow \forall y Q^c(y,\vec{x}) \ .$$

The predicate Q^c is recursive in the base case and Σ_{n-1} in the inductive case. thus in both cases R^c is Π_n. If $R_1 \in \Pi_n$ then the proof is similar.

If $R_1 \in \Delta_n$ then R_1 is both Σ_n and Π_n and so its complement R_1^c is both Π_n and Σ_n by the just proved property. Thus $R_1^c \in \Delta_n$.

7.2.10 Closures of arithmetical predicates under bounded quantification. The $(n{+}1)$-ary predicate R is defined by *bounded existential quantification* from an $(n{+}2)$-ary predicate Q if the following holds:

$$R(y,\vec{x}) \leftrightarrow \exists z {\le} y\, Q(z,y,\vec{x}) \ .$$

Bounded universal quantification is similar.

In the proof of the closure properties of arithmetic predicates under bounded quantification we rely on the following equivalences:

$$\forall y{\le}x \forall z \phi[y,x,z] \leftrightarrow \forall z \forall y{\le}x\, \phi[y,x,z] \tag{1}$$
$$\exists y{\le}x \exists z \phi[y,x,z] \leftrightarrow \exists z \exists y{\le}x\, \phi[y,x,z] \tag{2}$$
$$\forall y{\le}x \exists z \phi[y,x,z] \leftrightarrow \exists z \forall y{\le}x\, \phi[y,x,(z)_y] \tag{3}$$
$$\exists y{\le}x \forall z \phi[y,x,z] \leftrightarrow \forall z \exists y{\le}x\, \phi[y,x,(z)_y] \ . \tag{4}$$

The equivalences (1), (2) are obvious, and (4) follows from (3) by De Morgan laws. The equivalence (3) is proved in the direction $\rightarrow$ by assuming its left-hand-side. Thus there are numbers z_0, z_1, $\ldots$, z_x such that for all $y \le x$ we have $\phi[y,x,z_y]$. Take $z = z_0, z_1, \ldots, z_x, 0$ and for all $y \le x$ we have $\phi[y,x,(z)_y]$. Hence the right-hand-side holds. In the direction $\leftarrow$ assume that $\forall y{\le}x\, \phi[y,x,(t)_y]$ holds for some t. We take any $y \le x$ and for $z = (t)_y$ we have $\phi[y,x,z]$.

We leave to the reader the straightforward relativization of the closure properties to any class of functions $\mathcal{F}$.

We show by induction on n that the bounded universal quantification of Σ_n (Π_n) predicates are Σ_n (Π_n) predicates as shown in the column seven of the table 7.1. If $R_1 \in \Sigma_n$ then we have in both the base and inductive case:

$$\begin{aligned} R(y,\vec{x}) \Leftrightarrow \forall z{\le}y R_1(z,y,\vec{x}) \Leftrightarrow \forall z{\le}y \exists v Q(v,z,y,\vec{x}) \overset{(3)}{\Leftrightarrow} \\ \exists v \forall z{\le}y\, Q((v)_z,z,y,\vec{x}) \overset{*}{\Leftrightarrow} \exists v R_2(v,y,\vec{x}) \ . \end{aligned}$$

In the step marked by $*$ we have defined the predicate R_2 by bounded quantification of Q. R_2 is recursive in the base case because Q is and R_2 is Π_{n-1} by IH in the inductive case because so is Q. Thus in any case we have $R \in \Sigma_n$. If $R_1 \in \Pi_n$ then $R \in \Pi_n$ by the closure properties proved in the preceding paragraphs because we can treat the bounded universal bounded quantification as unbounded one.

The closure properties for bounded existential quantification follow from the just proved properties by De Morgan laws.

If $R_1 \in \Delta_n$ predicate then it is both Σ_n and Π_n and so its bounded universal (existential) quantification R is both Σ_n and Π_n by the just proved properties. Thus $R \in \Delta_n$.

7.2.11 Enumeration of classes of predicates. For a binary predicate Q and a number e we explicitly define the unary predicate Q_e as $Q_e(x) \leftrightarrow Q(e,x)$. We say that the number e is an *index* of an n-ary predicate R if $Q_e = \langle R \rangle$. Note that we then have $\forall \vec{x}(R(\vec{x}) \leftrightarrow Q(e,\vec{x}))$.

The predicate Q *enumerates* a non-empty class of predicates $\mathcal{R}$ if the contractions of predicates in $\mathcal{R}$ are exactly the predicates Q_e:

$$\{\langle R \rangle \mid R \in \mathcal{R}\} = \{Q_e \mid e \in \mathbb{N}\} \ .$$

i.e. if every $R \in \mathcal{R}$ has an index and every number is an index of some $R \in \mathcal{R}$.

7.2.12 Arithmetical enumeration theorem (Kleene). *The explicitly defined binary predicates:*

$$\begin{aligned} Q_1(e,x) &\leftrightarrow W_e(x) \\ Q_{n+1}(e,x) &\leftrightarrow \exists y\, Q_n^c(e,(y,x)) \end{aligned}$$

are such that for every n the predicate Q_n is Σ_n and enumerates the class Σ_n while its complement Q_n^c is Π_n and enumerates the class Π_n.

Proof. We first note that if a binary Σ_n (Π_n) predicate Q is such that every Σ_n (Π_n) predicate R has an index e such that $R(\vec{x}) \leftrightarrow Q(e,\vec{x})$ holds then Q enumerates the class of Σ_n (Π_n) predicates because for every number e the unary predicate $Q_e(x) \leftrightarrow Q(e,x)$ is Σ_n (Π_n) by Thm. 7.2.4.

We prove next that if a Σ_n-predicate Q enumerates Σ_n-predicates then its complement Q^c, which is by Fig. 7.1 Π_n, enumerates Π_n-predicates. By the above it is sufficient to show that any m-ary Π_n-predicate R has an index e such that $R(\vec{x}) \leftrightarrow Q^c(e,\vec{x})$ holds. The complement R^c of R is Σ_n and has an index e in Q which is also an index of R in Q^c because we have:

$$R(\vec{x}) \Leftrightarrow \neg R^c(\vec{x}) \Leftrightarrow \neg Q(e,\vec{x}) \Leftrightarrow Q^c(e,\vec{x}) \; .$$

In order to prove the theorem it thus suffices to show by induction on n that the predicate Q_n is Σ_n and that all Σ_n-predicates have indices in Q_n.

In the base case $n = 1$ we take an m-ary Σ_1-predicate R which is semi-computable by the Characterization theorem 7.1.4. Thus R is the domain of a partial recursive function f with an index e and we have:

$$R(\vec{x}) \Leftrightarrow f(\vec{x})\!\downarrow \Leftrightarrow \{e\}(x)\!\downarrow \overset{6.2.7(1)}{\Leftrightarrow} W_e(x) \Leftrightarrow Q_1(e,x) \; .$$

Q_1 thus enumerates Σ_1-predicates and it is itself Σ_1 by Par. 7.1.5.

In the inductive case the predicate Q_n is Σ_n and enumerates the Σ_n-predicates by IH. Its complement Q^c_n is Π_n and enumerates Π_n-predicates. We take any m-ary Σ_{n+1}-predicate $R(\vec{x}) \leftrightarrow \exists y\, R_1(y,\vec{x})$ and an index e of the Π_n-predicate R_1. We have

$$R(\vec{x}) \Leftrightarrow \exists y\, R_1(y,\vec{x}) \Leftrightarrow \exists y\, Q^c_n(e,(y,\vec{x})) \Leftrightarrow Q_{n+1}(e,\vec{x})$$

and so e is an index of R. The predicate Q_{n+1} is Σ_{n+1} by Thm. 7.2.4. ■

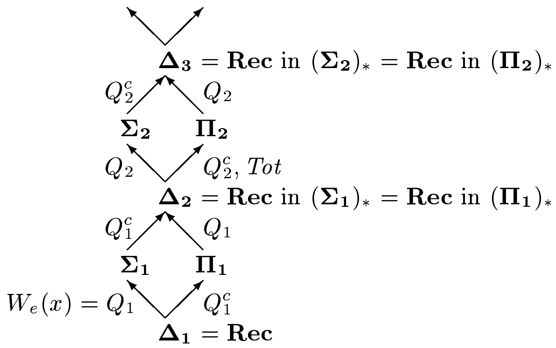


Figure 7.2: Inclusions of classes of arithmetical predicates

7.2.13 Arithmetical hierarchy. We will prove the Arithmetical Hierarchy Theorem 7.2.14 which asserts that the classes pointed to by upward arrows in Fig. 7.2 are proper supersets of the classes from which the arrows emanate.

The predicates placed next to the arrows belong to the higher but not to the lower classes. The indicated predicates are actually *complete* for the classes they belong to. A predicate R *is $\mathcal{R}$-complete* if $R \in \mathcal{R}$ and every $Q \in \mathcal{R}$ is reducible to R. $\mathcal{R}$-complete predicates are mutually reducible and they are the most complex predicates in the class $\mathcal{R}$. Every predicate enumerating a class $\mathcal{R}$ and belonging to it is $\mathcal{R}$-complete. We will prove in Par. 7.2.17 that the predicate of uniform halting *Tot* is Π_2-complete without enumerating the class.

We will also a theorem of Post from which the properties of Δ_n-predicates shown in Fig. 7.2 follow (see Corollary 7.2.16).

7.2.14 Arithmetical Hierarchy theorem (Kleene).

1. $\Delta_n \subseteq \Sigma_n$, $Q_n \in \Sigma_n \setminus \Delta_n$
2. $\Delta_n \subseteq \Pi_n$, $Q_n^c \in \Pi_n \setminus \Delta_n$
3. $\Sigma_n \subseteq \Delta_{n+1}$, $Q_n^c \in \Delta_{n+1} \setminus \Sigma_n$,
4. $\Pi_n \subseteq \Delta_{n+1}$, $Q_n \in \Delta_{n+1} \setminus \Pi_n$.

Proof. We prove first the set inclusions. For (1), (2) we have $\Delta_n \subseteq \Sigma_n$ and $\Delta_n \subseteq \Pi_n$ from the definition of Δ_n-predicates.

For the proof $\Sigma_n \subseteq \Pi_{n+1}$ of (3) we take an m-ary predicate $R \in \Sigma_n$ and explicitly define an $(m+1)$-ary predicate $R_1(y, \vec{x}) \leftrightarrow R(\vec{x})$ which is Σ_n by Thm. 7.2.4. Since we then have $\forall y\, R_1(y, \vec{x}) \leftrightarrow R(\vec{x})$, we can see that $R \in \Pi_{n+1}$. The proof of $\Pi_n \subseteq \Sigma_{n+1}$ of (4) is similar.

It remains to prove $\Sigma_n \subseteq \Sigma_{n+1} \wedge \Pi_n \subseteq \Pi_{n+1}$ which we do by induction on n. In the base case $n = 1$ an m-ary predicate $R \in \Sigma_1$ has a form $R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x})$ with Q recursive. We explicitly define an $(m+2)$-ary recursive predicate $Q_1(z, y, \vec{x}) \leftrightarrow Q(y, \vec{x})$ and from

$$R(\vec{x}) \Leftrightarrow \exists y\, Q(y, \vec{x}) \Leftrightarrow \exists y \forall z\, Q_1(z, y, \vec{x})$$

we obtain $R \in \Sigma_2$. The proof when $R \in \Pi_1$ is similar.

In the inductive case an m-ary predicate $R \in \Sigma_{n+1}$ has a form $R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x})$ with $Q \in \Pi_n$. We have $Q \in \Pi_{n+1}$ by IH and so $R \in \Sigma_{n+2}$. The proof when $R \in \Pi_{n+1}$ is similar.

We now show $Q_n \in \Sigma_n \setminus \Pi_n$. We know from the Arithmetical enumeration theorem that $Q_n \in \Sigma_n$ enumerates Σ_n-predicates. The unary predicate $R(x) \leftrightarrow Q_n(x, x)$ is Σ_n by Thm. 7.2.4. If it were the case that $R \in \Pi_n$ then its complement R^c would be Σ_n and it would have an index e. We would then obtain a contradiction as follows:

$$Q_n(e, e) \overset{\text{enum.}}{\Leftrightarrow} R^c(e) \Leftrightarrow \neg R(e) \overset{\text{def.}}{\Leftrightarrow} \neg Q_n(e, e)\ .$$

It cannot be the case that $Q_n \in \Pi_n$ because then also $R \in \Pi_n$. This proves $Q_n \in \Sigma_n \setminus \Delta_n$ and $Q_n \in \Delta_{n+1} \setminus \Pi_n$

From $Q_n \in \Sigma_n \setminus \Pi_n$ we get $Q_n^c \in \Pi_n \setminus \Sigma_n$. This proves $Q_n^c \in \Pi_n \setminus \Delta_n$ and $Q_n^c \in \Delta_{n+1} \setminus \Sigma_n$. ■

7.2.15 Theorem (Post) *R is Σ_{n+1} iff it is Σ_1 in $(\Pi_n)_*$.*

Proof. If $R \in \Sigma_{n+1}$ then we have $R(\vec{x}) \leftrightarrow \exists y\, Q(y, \vec{x})$ with $Q \in \Pi_n$. Thus $Q_* \in (\Pi_n)_*$ which makes Q a $(\Pi_n)_*$-recursive predicate and R is Σ_1 in $(\Pi_n)_*$.

Vice versa, if R is Σ_1 in $(\Pi_n)_*$ then it is semicomputable in $(\Pi_n)_*$ by Thm. 7.1.4. Let g enumerate $(\Pi_n)_*$. For some e we then have 7.1.1(1), i.e.

$$R(\vec{x}) \leftrightarrow \exists s \exists z(\, \boldsymbol{Rds}((\boldsymbol{\lambda} e)(\boldsymbol{C}(\vec{x})), s) = \boldsymbol{C}(z) \wedge G_{e,g}(s)\,)$$

where $G_{e,s}$ satisfies:

$$G_{e,g}(s) \leftrightarrow \forall i \forall x \forall v(\, (i, x, v) \,\varepsilon\, s \to \bigvee_{j=0}^{e-1} (i = j \wedge g_j(x) = v)\,)\ .$$

For every $j < e$ the function $g_j(x) = g(j, x)$ is the characteristic function of the contraction $\langle Q_j \rangle$ of an m_j-ary Π_n-predicate Q_j. We have

$$\langle Q_j \rangle(x) \leftrightarrow \exists \vec{x}(\, x = \vec{x} \wedge Q_j(\vec{x})\,)$$

where the quantified variables are bounded by $x_i \leq x$ and from Fig. 7.1 we can see that $\langle Q_j \rangle$ is Π_n. The graph of g_j is Σ_{n+1} because we have

$$g_j(x) = v \leftrightarrow \langle Q_j \rangle(x) \wedge v = 1 \vee \neg \langle Q_j \rangle(x) \wedge v = 0$$

and so the predicate $G_{e,g}$ is also Σ_{n+1} because the quantified variables i, x, and v are bounded by s. This makes R a Σ_{n+1}-predicate. ∎

7.2.16 Corollary *R is Δ_n iff it is $\mathcal{F}$-recursive where $\mathcal{F} = \emptyset$ if $n = 1$ and $\mathcal{F} = (\Sigma_{n-1})_*$ or $\mathcal{F} = (\Pi_{n-1})_*$ if $n > 1$.*

Proof. Take any n and let $\mathcal{F}$ be $(\Pi_{n-1})_*$ if $n > 1$ and $\emptyset$ otherwise. We have $R \in \Delta_n$ iff R and R^c are Σ_n iff, by Thm. 7.2.15 for $n > 1$ and trivially otherwise, R and R^c are Σ_1 in $\mathcal{F}$ iff, by Thm. 7.1.4, R and R^c are r.e. in $\mathcal{F}$ iff, by Thm. 7.1.6, R is $\mathcal{F}$-recursive.

If $f \in (\Pi_n)_*$ then $f = Q_*$ for some $Q \in \Pi_n$. Since $Q^c \in \Sigma_n$ and $f(x) = 1 \dot{-} Q^c_*(x)$, f is $(\Sigma_n)_*$-recursive. Thus $(\Pi_n)_*$-recursive functions are $(\Sigma_n)_*$-recursive. The converse inclusion that $(\Sigma_n)_*$-recursive functions are $(\Pi_n)_*$-recursive is proved similarly. ∎

7.2.17 The predicate *Tot* of uniform halting is Π_2-complete. The predicate of uniform halting $\mathit{Tot}(e)$ which holds of codes e of recursive functions was introduced in Par. 6.2.8 where we have seen that it is not recursive. It is a Π_2-predicate because we have:

$$\mathit{Tot}(e) \Leftrightarrow \forall x\, W_e(x) \overset{7.1.5(1)}{\Leftrightarrow} \forall x \exists y\, R(y, e, x).$$

with R primitive recursive.

We now prove that *Tot* is Π_2-complete. So let R be any Π_2-predicate. We have $R(\vec{x}) \leftrightarrow \forall y\, R_1(y, \vec{x})$ for some $(n{+}1)$-ary predicate $R_1 \in \Sigma_1$. The explicitly defined binary predicate

$$Q(z, y) \leftrightarrow \exists \vec{x}(\, z = \vec{x} \wedge R_1(y, \vec{x})\,)$$

is thus Σ_1 by Fig. 7.1. By the Characterization theorem Q is the domain of a partial recursive function f with an index e. We then have

$$\begin{aligned} &R(\vec{x}) \Leftrightarrow \forall y\, R_1(y, \vec{x}) \Leftrightarrow \forall y\, Q((\vec{x}), y) \Leftrightarrow \forall y\, f((\vec{x}), y)\!\downarrow\, \Leftrightarrow \\ &\qquad \forall y\, \{e\}((\vec{x}), y)\!\downarrow \overset{6.2.11(1)}{\Leftrightarrow} \forall y\, \{e/(\vec{x})\}(y)\!\downarrow \overset{6.2.7(1)}{\Leftrightarrow} \forall y\, W_{e/(\vec{x})}(y) \Leftrightarrow \mathit{Tot}(e/\vec{x})\ . \end{aligned}$$

Thus R reduces to *Tot* because the parameter function is primitive recursive.

If it were the case that $\mathit{Tot} \in \Delta_2$ then, since Q_2^c reduces to *Tot*, we would get a contradiction $Q_2^c \in \Delta_2$, i.e. $Q_2 \in \Delta_2$.

Bibliography

[1] W. Ackermann. *Zun Hilbertschen Aufbau der reellen Zahlen*. Math. Ann. vol. 99 (1928), pages 118-33.

[2] J. H. Bennett. *On Spectra*. Ph.D. dissertation. Princeton, 1962.

[3] G. Boolos and R. Jeffrey, *Computability and Logic*, Cambridge University Press, Cambridge, 1974.

[4] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[5] A. Church, *An Unsolvable problem of Elementary Number Theory*, American Journal of Mathematics, vol. 58 (1936), pages 345-63.

[6] A. Cobham, *The Intrinsic Computational Difficulty of Functions*, in Proc. Int. Conf. Logic, Methodology and Philosophy Sci. (ed. Y. Bar Hillel), North Holland 1965, pages 24-30.

[7] M. Davis, *Computability and Unsolvability*, McGraw Hill, New York, 1958.

[8] R. L. Graham, D. F. Knuth, O. Patashnik, *Concrete Mathematics*, Addison Wesley 1989.

[9] P. Hájek and P. Pudlák. *Metamathematics of First-Order Arithmetic*. Springer Verlag, 1993.

[10] H. Hermes, *Enumerability, Decidability, Computability*, Springer Verlag, 1965.

[11] S. C. Kleene. *General recursive functions of natural numbers*. Math Ann. vol. 112 (1936), pages 727-742.

[12] S. C. Kleene. *Introduction to Meta-mathematics*. North-Holland, 1952.

[13] R. Péter. *Konstruktion nichtrekursiver Funktionen*. Math. Ann. vol 111 (1935), pages 42-60.

[14] R. Péter. *Über die mehrfache Rekursion*. Math. Ann. vol 113 (1936), pages 489-527.

[15] R. Péter. *Recursive Functions.* Academic Press, 1967.

[16] Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.

[17] J. Robinson. *Definability and decision problems in arithmetic.* Journal of Symbolic Logic, vol. 14 (1949), pages 98-114.

[18] R. M. Robinson. *Primitive Recursive Functions.* Bull. Amer. Math. Soc. vol. 53, (1947), pages 925-945.

[19] H. E. Rose. *Subrecursion: Functions and Hierarchies.* Number 9 in Oxford Logic Guides. Clarendon Press, Oxford, 1982.

[20] R. I. Soare, *Computability and recursion*, in Handbook of Computation Theory, Elsevier, 1999.

[21] J. R. Shoenfield. *Mathematical Logic*, Addison-Wesley, 1967.

[22] R. M. Smullyan *Theory of Formal Systems*, Annals of Math. Stud. No. 47, Princeton Univ. Press, 1961.

[23] R. M. Smullyan. *Gödel's Incompleteness Theorems.* Oxford Logic Guides, Oxford University Press 1992.

[24] W. W. Tait. On nested recursion. *Mathematische Annalen*, vol. 143 (1961), pages 236-250.

[25] A. M. Turing. *On computable numbers with an application to the Entscheidungsproblem.* Proc. London Math. Society, 2 vol. 42 (1936).

[26] 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 Springer Verlag, 1995, pages 324-338.

[27] P. J. Voda. *Programming by Logic & Logic by Programming*, (Volume 1/ Computability), Bratislava 1996. Available through www.fmph.uniba.sk/∼voda.

Index of Notation

Index